Roman Punskyy

Pattern - Oriented Software Architecture.

Somewhere in the deeply remote past it seriously traumatized a small random group of atoms drifting through the empty sterility of space and made them cling together in the most extraordinary unlikely patterns. These patterns quickly learnt to copy themselves (this was part of most extraordinary about the patterns) and went on to cause massive trouble on every planet they drifted on to.
That was how life began in the Universe…

Douglas Adams, The Hitchhikers's guide to the Galaxy.

When experts need to solve a problem, they seldom invent a totally new solution. More often they will recall a similar problem they have solved previously and reuse the essential aspects of the old solution to solve the new problem. They tend to think in problem-solution pairs. Identifying the essential aspects of specific problem - solution pairs leads to descriptions of problem - solving ***patterns*** that can be reused. The concept of a pattern as used in software architecture is borrowed from the field of (building) architecture, in particular from the writings of architect Christopher Alexander.

The goal of patterns within the software community is to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions.

One of the widely used definitions is: "A ***pattern for software architecture*** describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." [Buschmann].

Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves. Context section describes the situation in which the design problem arises. Problem section describes the problem that arises repeatedly in the context. And finally, solution section describes a proven solution to the problem.

So aside from recurrence, a pattern must describe *how* the solution balances or resolves its forces, and why this is a "good" resolution of forces. We need both of these to convince us that the pattern is neither sheer speculation (pure theory) nor is the pattern blindly following others (rote practice). We want to show that the practice is more than just "theory", and that the theory really has been practiced. We might even say that: A pattern is where theory and practice meet to reinforce and complement one another, by showing that the structure it describes is useful, useable, and used!

A pattern must be *useful* because this shows how having the pattern in our minds may be transformed into an instance of the pattern in the real world, as something thing that adds value to our lives as developers and practitioners. A pattern must also be *useable* because this shows how a pattern described in literary form may be transformed into a pattern that we have in our minds. And a pattern must be *used* because this is how patterns that exist in the real world first became documented as patterns in literary form.

This yields a continuously repeating cycle from pattern writers, to pattern readers, to pattern users: writers documenting patterns in literary form make them usable to pattern readers, who can then remember them in their minds, which makes them useful to practitioners and developers, who can use them in the real world, and enhance the user's quality of life.
Patterns categorized into three levels: **architectural patterns**, **design patterns**, and **idioms** (idioms are sometimes called **coding patterns**). The authors of Patterns of Software Architecture define three types of patterns as follows:

Architectural Patterns

An *architectural pattern* expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Design Patterns

A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.

Idioms

An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

The differences between these three kinds of patterns are in their corresponding levels of abstraction and detail. Architectural patterns are high-level strategies that concern large-scale components and the global properties and mechanisms of a system. They have wide-sweeping implications which affect the overall skeletal structure and organization of a software system. Design patterns are medium-scale tactics that flesh out some of the structure and behaviour of entities and their relationships. They do *not* influence overall system structure, but instead define *micro-architectures* of subsystems and components. Idioms are paradigm-specific and language-specific programming techniques that fill in low-level internal or external details of a component's structure or behaviour.

Alexander says that "every pattern we define must be formulated in the form of a rule which establishes a relationship between a context, a system of forces which arises in that context, and a configuration, which allows these forces to resolve themselves in that context." A pattern consists of the following elements:

Name

It must have a meaningful name. This allows us to use a single word or short phrase to refer to the pattern, and the knowledge and structure it describes. Sometimes a pattern may have more than one commonly used or recognizable name in the literature. In this case it is common practice to document these nicknames or synonyms under the heading of **Aliases** or **Also Known As**. Some pattern forms also provide a **classification** of the pattern in addition to its name.

Problem

A statement of the problem which describes its **intent**: the goals and objectives it wants to reach within the given context and forces. Often the forces oppose these objectives as well as each other.

Context

The *preconditions* under which the problem and its solution seem to recur and for which the solution is desirable. This tells us the pattern's **applicability**. It can be thought of as the initial configuration of the system before the pattern is applied to it.

Forces

A description of the relevant *forces* and constraints and how they interact/conflict with one another and with goals we wish to achieve (perhaps with some indication of their priorities). A concrete scenario which serves as the **motivation** for the pattern is frequently employed. Forces reveal the intricacies of a problem and define the kinds of *trade-offs* that must be considered in the presence of the tension or dissonance they create. A good pattern description should fully encapsulate all the forces which have an impact upon it

Solution

Static relationships and dynamic rules describing how to realise outcome. The description of the pattern's solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting a concrete **implementation** of the solution. Sometimes possible **variants** or specializations of the solution are also described.

Examples

One or more sample applications of the pattern which illustrate: a specific initial context; how the pattern is applied to, and transforms, that context; and the resulting context left in its wake. Examples help the reader understand the pattern's use and applicability. Visual examples and analogies can often be especially illuminating. An example may be supplemented by a *sample implementation* to show one way the solution might be applied.

One of the most famous patterns is MVC (Model – View – Controller) pattern. This pattern is used when we developing an interactive system. In any program that requires user actions there are two things to consider:

1. Changes to user interface shall be easy and possible at runtime (the last is optional).
2. Changes in the user interface should not have impact in the main core (engine) of the application.

To solve these problems as Buschmann’s book advices we should divide an interactive application into three categories: *processing*, *output* and *input*:

a) The *Model* the core of the application. This maintains the data and state that the application represents. When significant changes occur in the model, it updates all of its views.
b) *View*: the user interface which displays information about the model to the user. Any objects that needs information about the model needs to be a registered view with the model.
c) *Controller*: the user interface presented to the user to manipulate the application. Controllers receive input as events of mouse clicks or keyboard. Events are translated to service which are sent to Model or View. User may interact with the system only by controllers.

To conclude patterns are not invented, they discovered from practical experience and are *not* a "silver bullet"! They are extremely valuable tools to improve software quality and productivity by addressing fundamental issues in the development of software.

Perhaps the final remarks from **Pattern-Oriented Software Architecture** best describe the significance of patterns for software:
Patterns expose knowledge about software construction that has been gained by many experts over many years. All work on patterns should therefore focus on making this precious resource widely available. Every software developer should be able to use patterns effectively when building software systems. When this is achieved, we will be able to celebrate the human intelligence that patterns reflect, both in each individual pattern and in all patterns in their entirety.

Bibliography and References:

1. Pattern-Oriented Software Architecture - A System of Patterns
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Wiley and Sons Ltd., 1996

3. http://www.sts.tu-harburg.de/teaching/ss-02/SoftArch/entry.html

4. http://www.cs.olemiss.edu/~hcc/softArch/notes/patterns.html

5. http:// www.enteract.com