What is a pattern?

Aleksandra Tešanović

Linköping University
Department of Computer and Information Science
Linköping, Sweden
alete@ida.liu.se

1 Introduction

The goal of this paper is to give an introduction in a broad emerging area of software patterns, by presenting basic concepts of patterns in software, such as pattern template, pattern language, pattern system, and pattern catalog.

Patterns are effective means of communication between software developers. Patterns, in a sense, help bring order into the chaos. Patterns represent best practice, proven solutions, and lessons learned, that aid in evolving software engineering into a mature engineering discipline. The aim is that every software developer should be able to use patterns when building software systems. Thus, to reach this aim, large pattern sets have to be presented in a form of languages, catalogs, systems, or even handbooks.

The rest of the paper is organized as follows. In section 2 a definition and basic characteristics of a software pattern are given. Section 3 presents the classification of patterns in software, with the emphasis on design patterns. This is followed by description of the most common used templates for recording patterns in section 4, and different pattern collections in section 5. Paper finishes with the short history of software patterns, section 6.

2 What is a pattern?

This section consists of three parts. First part (section 2.1) presents the definition of a pattern. Second part (section 2.2) presents the most common misconceptions about patterns. Third part (section 2.3) presents an overview of the characteristics of a good pattern. Combined, these three parts, give a basic notion of what a software pattern really is.

2.1 Definition

Patterns are all around us. Patterns can be found in nature, e.g., bubbles and waves, buildings, e.g., windows, etc. Patterns can also be found in software. The term pattern is adopted in software from the work of the architect Christopher Alexander, who was

exploring patterns in architecture. Thus, in attempt to define a pattern, a good starting point is a definition of a pattern given by Alexander [2]:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Coplein [6] gives a very similar generic definition of a software pattern, as

> ...the thing and the instructions for making the thing.

A pattern can be viewed as a prose format for recording solutions, i.e., design information, such that solutions, i.e., designs, which have worked well in the past can be applied again in similar situations in the future [4]. The need for the introduction of patterns in software originates form the need of an engineering discipline to mature, and like a mature engineering discipline, to provide handbooks for describing successful solutions to known problems.

2.2 Common misconceptions

Common misconceptions about patterns can be summarized as follows:

- patterns are only object-oriented,
- patterns provide only one solution,
- patterns are implementations, and
- every solution is a pattern.

Patterns are not only object-oriented Patterns in the software are often identified only with object-oriented software. Although most of the patterns are object-oriented, patterns can also be found in variety of software systems, independently of the methods used in developing those systems [4]. Patterns are widely applicable to every software system, since they describe software abstractions.

Patterns provide more than one solution Patterns describe solutions to the recurring problems, but do not provide an exact solution, rather capture more than one solution. This implies that a pattern is not an implementation, although it may provide hints about potential implementation issues. The pattern only describes when, why, and how one could create an implementation.

Every solution is not necessary a pattern Not every solution, algorithm, or heuristic can be viewed as a pattern. In order to be considered as a pattern, the solution must be verified as recurring solution to a recurring problem. The verification of the recurring phenomenon is usually done by identifying the solution and the problem (the solution solves) in at least three different existing systems. This method of verification is often referred to as the *rule of three*.

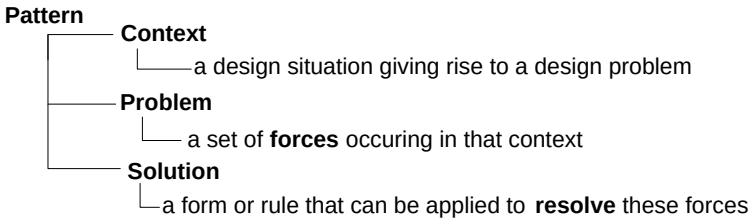

Figure 1: Basic ingredients of a pattern.

2.3 Pattern characteristics

Every pattern has three essential elements, which are: a context, a problem, and a solution (see figure 1). The context describes a recurring set of situations in which the pattern can be applied. The problem refers to a set of forces, i.e, goals and constraints, which occur in the context. Generally, the problem describes when to apply the pattern. The solution refers to a design form or a design rule that can be applied to resolve the forces. Solution describes the elements that constitute a pattern, relationships among these elements, as well as responsibilities and collaboration.

Following example is to illustrate basic ingredients of a pattern and their relationship.

Window place Consider one simple problem that can appear in the architecture. Let us assume that a person wants be comfortable in a room, implying that the person needs to to sit down to really feel comfortable. Additionally, the sunlight is an issue, since the person is most likely to prefer to sit near the light. Thus, *forces* in this example are: (i) the desire to sit down, and (ii) the desire to be near light. The *solution* to this *problem* could be that in every room the architect should make one window into a *window place.*

Not every pattern can be considered to be a good pattern. There is a set of properties that a pattern must fulfill in order to be a good one. A pattern encapsulating:

- a solution (but not obvious),
- a proven concept,
- relationships, and
- human component,

is considered to be a good pattern [7]. Thus, a good pattern should solve a problem, i.e., patterns should capture solutions, not just abstract principles or strategies. A good pattern should be a proven concept, i.e., patterns should capture solutions with a track record, not theories or speculation. A good pattern should not provide an obvious solutions, i.e., many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns generate a solution to a problem indirectly, which is a necessary approach for the most difficult

problems of design. A good pattern also describes a relationship, i.e, it does not just describe modules, but describe deeper system structures and mechanisms. Additionally, a good pattern should contain a significant human component (minimize human intervention). All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetic and utility.

3 Classification of software patterns

There exist many different classifications of software patterns. In general, all software patterns can be classified as generative and non-generative, as presented in section 3.1. Both generative and non-generative patterns could be classified to be design patterns, organization patterns, analysis patterns, etc., depending on the aspect of software they refer to, as explained in section 3.2. Design patterns are the most known and used patterns today, as they are patterns in software engineering and reflect both low-level strategies for design of components in the system, and high-level strategies that impact the design of the overall system. Thus, design patterns and its different categories are presented in section 3.3.

3.1 Generative and non-generative patterns

All software patterns can be classified as generative or non-generative [6]. Patterns that can be observed in a system, which has already been built are considered to be non-generative patterns. Non-generative patterns are also referred to as Gamma patterns [6] (Gamma patterns are discussed in more detail in section 5.3). Such patterns are descriptive and passive, since they describe recurring phenomenons without necessary describing the way of reproducing these phenomenons. Generative patterns, on the other hand, are designed to shape system architecture, and can generate systems or parts of the systems. Thus, generative patterns not only show characteristics of a good system, but also teach how to build such systems. In other words, generative patterns help generate other patterns.

3.2 Types of software patterns

Initially, software community has focused primarily on design patterns. The patterns described by Gamma et al. [9] [1] are object-oriented design patterns. However, there are many other types of software patterns besides design patterns. Patterns can be found in all aspects of software. This includes: development organization, software process, project planning, requirements engineering, and software configuration. Although design patterns appear to be the most popular, other types of patterns are also gaining momentum, e.g., organization patterns.

There exist number of different classification of software patterns. One possible classification of the patterns is as follows [3]:

- design patterns, which are patterns in software engineering,

[1] Gang of Four (GoF), i.e., Gamma, Helm, Johnson, Vlissides, are the pioneers that worked on introducing patterns in software.

- analysis patterns, which are patterns that describe recurring and reusable analysis models,
- organization patterns, which are patterns that describe software process design, and
- other domain-specific patterns.

Reihle and Zullighoven [11] give a somewhat different classification of patterns, as they identify following three types of patterns:

- conceptual patterns,
- design patterns, and
- programming patterns.

Conceptual patterns are patterns which are described by means of the terms and concepts from an application domain, and which are geared towards a restricted application domain. Design patterns are patters which are described by means of software design constructs, e.g., objects, classes, inheritance, aggregation, and reuse relationship. Design patterns complement the conceptual space opened by the conceptual patterns. Programming patterns are patterns which are described by means of programming language constructs.

3.3 Types of design patterns

Design patterns can further be divided into three categories of patterns [5]:

- architectural patterns,
- design patterns, and
- idioms.

The difference between these three categories of design patterns is in their corresponding levels of abstraction and detail. Architectural patterns are high-level strategies that are concerned with large-scale components and global properties and mechanisms of a system. Architectural patterns have implications affecting the overall structure and organization of a software system. Design patterns are medium-level strategies that are concerned with the structure and behavior of entities, and their relationships. Design patterns do not influence the overall system structure, but instead define micro-architectures of subsystems and components. Design patterns in this classification correspond to design patterns in classification by Reihle and Zullighoven [11], and to Gamma patterns, i.e., patterns described by Gamma et al. [9]. Idioms are paradigm-specific and language-specific programming techniques that fill in low-level internal or external details of a component's structure or behavior.

Name	The pattern must have a meaningful name.
Problem	The statement of the problem the pattern is trying to solve.
Context	A situation giving rise to a problem.
Forces	A concrete scenario, i.e., description of forces.
Solution	Proven solution to the problem.
Examples	A sample of the application of the pattern.
Resulting context	The state of the system after the pattern has been applied.
Rationale	Explanation of steps or rules in the pattern.
Related patterns	Static and dynamic relationship between other patterns.
Known use	Occurrences of the pattern within the existing systems.

Table 1: Alexandrian form for describing patterns.

4 Pattern templates

Number of different formats, i.e., templates, exists for describing software patterns. Two most commonly used pattern templates are: Alexandrian form and GoF format. *Alexandrian form* is the pattern description format adopted from work of Alexander Christopher. Alexandrian form is also referred to as canonical form. *GoF* [2] *format* is the format of describing patterns introduced by Gamma et al. [9]. There is also a number of other different formats (templates) that patterns can be described in, but the two mentioned are the most widely used and known ones. Therefore, in the rest of this section more detailed description of the Alexandrian template (section 4.1) and GoF template (section 4.2) is given, followed by a brief introduction to other existing templates (section 4.3).

4.1 Alexandrian form

Table 1 presents all elements (sections) that constitute an Alexandrian form for describing a pattern, i.e., Alexandrian pattern template. Bellow follows the description of each section in the pattern template.

Name Patterns with meaningful names allow referring to it by a single word or short phrase. Good pattern names form a vocabulary for discussing conceptual abstractions.

Problem The problem contains the statement of the problem a pattern aims to solve. General idea behind a description of the problem in the pattern is that the problem description should give a designer the knowledge of the problem a pattern solves, which enables the designers to know when to apply given pattern.

Context Context are the preconditions under which the problem and its solution seem to recur, and for which the solution is desirable. Context describes patterns applicability. A pattern solves a problem in a given context and, usually, it might not

[2] GoF is abbreviation of Gang of Four: Gamma, Helm, Johnson, and Vlissides.

make sense in other contexts. Context can also be viewed as the initial configuration of the system before the pattern is applied to it.

Forces Forces represent a concrete scenario which provides the motivation for the use of pattern in certain context to solve a certain problem. Forces make clear intricacies of a problem, since not all problems are clear cut. A good pattern description should encapsulate all the forces upon which it might have an impact.

Solution The solution should describe not only static structure but also dynamic behavior of the pattern. The static structure represents the form and organization of the pattern, but often the behavioral dynamics is what makes the pattern alive. The description of the solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting to create a concrete implementation of the solution.

Examples Existence of examples could be a reflection of a good pattern. Examples should be one or more sample applications, which illustrate the specific initial context, how the pattern is applied to the context, and how a pattern transforms that context. Examples help understand applicability of the pattern. Visual examples and analogies are often desirable, since they could be especially illuminating. An example may be supplemented by a sample implementation to show one way the solution might be realized.

Resulting context Resulting context is the state or configuration of the system after the pattern has been applied. This includes the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context. Resulting context describes the postconditions and side-effects of applying the pattern. Few patterns are perfect or stand on their own. A good pattern indicates what forces it leaves unresolved, or what other pattern must be applied, and how the context is changed by the pattern.

Rationale Rationale provides insight into the deep structures and key mechanisms that are going on beneath the surface of the system. In other words, rationale describes where the pattern comes from, why it works, and why expert use it. To be able to produce a good design, the designer should be able to apply patterns insightfully, i.e., designers should be able to understand how the pattern works.

Related patterns Related patterns describe the static and dynamic relationships between this pattern and other patterns.

Known uses Known uses describe known occurrences of the pattern and its application within existing systems. This helps in validating a pattern, i.e., verifying that pattern is indeed a proven solution to a recurring problem. Known uses of the pattern could also serve as instructional examples.

Name and classification	Classification according to Gamma et al. [9].
Intent	A problem that the pattern addresses.
Also known as	Other well-known names of the pattern.
Motivation	A scenario illustrating a design problem.
Applicability	Situations where pattern can be applied.
Structure	A graphical representation of classes in the pattern.
Participants	Classes and objects and their relationships.
Collaboration	Participants collaborating to carry out responsibilities.
Consequences	Trade-offs and results of using a pattern.
Implementation	Things to be aware of when implementing a pattern.
Sample code	Code fragment illustrating one implementation.
Known uses	Examples of patterns found in real systems.
Related patterns	Other closely related patterns.

Table 2: GoF format for describing patterns.

4.2 GoF format

As in the Alexandrian template, in the GoF template each pattern is divided into a number of sections, as shown in table 2. Pattern in GoF format must have a **Name**. Additionally, **Classification** of the pattern should be stated. Here classification refers to the classification according to Gamma et al. [9]. Sometimes a pattern may have more than one commonly used name in the literature. In this case, it is common to document these synonyms under the section **Also Known As**. **Intent** describes the design issue a pattern is addressing, thus reflecting what a pattern does. **Motivation** section of a template should illustrate a design problem and how the class and object structures in the pattern solve this problem. **Applicability** describes situations in which the design pattern can be applied, as well as examples of poor designs the pattern is addressing, with hint how to recognize such situations. **Structure** refers to graphical representation of classes in the pattern. Classes, objects, and their responsibilities are described under the **Participant** section of the template. Further, the issue how participants carry out their responsibilities is described in **Collaborations**. Trade-offs and results of using a patterns should be documented as well, under the section **Consequences**. **Implementation** describes the pitfalls, hints or techniques that the designer should be aware of when implementing given pattern. Code fragments illustrating one possible implementation of a pattern should be available in **Sample code** section. Finally, examples of patterns found in the existing applications, and relationship to other patterns should be documented under **Known uses** and **Related patterns** sections, respectively.

Now, comparing the two templates, Alexandrian and GoF, it can be noted that templates differ form one another in the headings of the sections, but the information that a pattern should give to the designer is almost the same in both templates.

[PATTERN-NAME]
Author
[YOUR-NAME] ([YOU@YOUR.ADDR]).
Last updated on [TODAY'S-DATE]
Context
[PARAG-1]
[PARAG-2]
Problem
[ONE-ASPECT]
[ANOTHER-ASPECT]
Examples
Forces
1.[FORCE-1]
2.[FORCE-2]
Design
[PARAG-1]
[PARAG-2]
An Implementation
[SOME-CODE]
Examples
Variants
[VARIANT]
[ANOTHER-VARIANT]
See Also
[ANOTHER-REF]

Figure 2: An example of pattern template from [10].

4.3 Other templates

There are several other formats in which patterns could be described. One format is given in figure 2, and can be found in [10]. Another example of a template is shown in figure 3. These examples merely serve for illustrating the diversity of formats in which patterns could be described. There is a lot of pattern templates which differ from one another in some ingredient, e.g., one or more sections of the pattern template. However, each template should give the same essential information to the designer (possibly, in a slightly different format), i.e., outlook of the templates can be different but the content is usually the same.

```
IF   you find yourself in CONTEXT
              for example EXAMPLES,
              with PROBLEM,
              entailing FORCES
         THEN for some REASONS,
              apply DESIGN FORM AND/OR RULE
              to construct SOLUTION
              leading to NEW CONTEXT and OTHER PATTERNS
```

Figure 3: Pattern template from [8].

5 Pattern language, pattern system and pattern catalog

Three different types of pattern collections have been identified [5]: pattern language, pattern system, and pattern catalog. These three pattern collections possess varying degrees of structure and interaction. In this section the definition and basic characteristics of each of these pattern collections is given, i.e., pattern language in section 5.1, pattern system in section 5.2, and pattern catalog in section 5.3.

5.1 Pattern language

One commonly used definition of a pattern language can be found in [7].

> A pattern language defines a collection of patterns and the rules to combine them into an architectural style. Pattern languages describe software frameworks or families of related systems.

A slightly different definition of the pattern language can be found in [6].

> A pattern language is a structured collection of patterns that build on each other to transform needs and constraints into an architecture.

A pattern language is not a programming language, rather it is a prose document, with a purpose to guide and inform the designer. A pattern language includes rules and guidelines which explain how and when to apply its patterns to solve a problem, which an individual pattern could not solve. These rules and guidelines suggest the order and granularity for applying each pattern in the language. A pattern language could also be viewed both as a lexicon of patterns and a grammar. The grammar defines how to weave patterns from lexicon together into valid sentences, i.e., software artifacts. Ideally, good pattern languages should be generative, and capable of generating all the possible sentences from a rich and expressive pattern vocabulary.

5.2 Pattern system

Pattern system is defined as follows [5].

> A pattern system is a cohesive set of related patterns which work together to support the construction and evolution of whole architectures. It describes many interrelationships between the patterns and their groupings and how they may be combined and composed to solve more complex problems.

The primary difference between the pattern system and a pattern language (defined in section 5.1) is that, ideally, pattern languages are computationally complete, showing all possible combinations of patterns and their variations to produce complete architectures. In practice however, the difference between pattern systems and pattern languages could be extremely difficult to ascertain.

SCOPE	PURPOSE		
	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Momento Observer Observer State Strategy Visitor

Table 3: Design pattern catalog.

5.3 Pattern catalog

Pattern catalog is a collection of related patterns, where patterns are subdivided into a small number of broad categories, which usually include some amount of cross referencing between patterns [5]. Pattern catalog introduced by Gamma et al.[9] is discussed in more detail.

Design pattern catalog, according to Gamma et al. [9]

Gamma et al.[9] classified patterns based on two criteria, as shown in table 3. The first criterion, called *purpose*, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns are concerned with the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility. The second criterion is called *scope*, and it specifies whether the pattern applies primarily on classes or on objects. Class patterns deal with relationships between classes and their sub-classes. These relationships are established through inheritance, so they are static (fixed at compile time). Object patterns deal with object relationships, which are more dynamic and can be changed at run-time. Patterns labeled as *class* patterns are those that focus on class relationships.

Creational class patterns defer some part of the object creation to the sub-classes. In contrast, creational object patterns defer some part of the object creation to other objects. Structural class patterns use inheritance to compose classes, while structural object patterns describe ways to assemble objects. Behavior class patterns use inheritance to describe algorithms and flow of control, while behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can perform on its own.

6 History of software patterns

Patterns history of course involves the work of Christopher Alexander on patterns in the architecture. Between 1964 and 1979 Alexander has published several books related to patterns in urban planing and building architecture.

In software, however, Ward Cunningham and Kent Beck where the first to use some of the Alexander's ideas to develop a small pattern language for guiding novice Smalltalk programmers. The results of their work, they presented in the paper *Using Pattern Languages for Object-Oriented Programs* at OOPSLA in 1987.

Soon after, Jim Coplien has begun to catalog language-specific C++ patterns he called idioms. Addison-Wesley published Coplien's book *Advanced C++ Programming Styles and Idioms* in 1991.

In 1990 members of the GoF have started their work on compiling the catalog of patterns. Things really got going at the OOPSLA workshop given by Bruce Andersen in 1991. Many of the pattern notables participated in the workshop, e.g., Jim Coplien, Doug Lea, Desmond D'Souza, Norm Kenth, Wolfgang Pree, and members of the GoF.

In August of 1993, Kent Beck and Grady Booch sponsored a mountain retreat in Colorado, the first meeting of what is now called the Hillside Group. Another pattern workshop was held at OOPSLA in 1993, and in 1994. Shortly after, the book by GoF, *Design Patterns: Elements of Reusable Objects*, was published, and the rest is history…

For more details on history of patterns see [1].

References

[1] History of patterns. Available at http://c2.com/cgi-bin/wiki?HistoryOfPatterns. Ward Cunningham's Wiki Wiki Web.

[2] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[3] B. Appleton. Patterns and software: Essential concepts and terminology. http://www.enteract.com/˜braddapp, 2000.

[4] K. Beck, J. O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch, and J. Vlisside. Industrial experience with design patterns. In *Proceedings of the 18th International Conference on Software Engineering*, pages 103–114. IEEE Computer Society Press, 1996.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, New York, 1996.

[6] J. O. Coplein. *The Patterns Handbook: Techniques, Strategies, and Applications*, chapter Software Design Patterns: Common Questions and Answers, pages 311–320. Cambridge University Press, New York, January 1998.

[7] J. O. Coplien. A pattern definition. Patterns Home Page, 2001. http://hillside.net/patterns/.

[8] Patterns-discussion FAQ. Available at http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Proffesional Computing Series. Addison-Wesley, 1995.

[10] Dag Lea. Pattern template. Patterns Home Page. Available at http://hillside.net/patterns/writing/writingpatterns.htm.

[11] D. Riehle and H. Zullighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.