International Journal of Current Advanced Research
ISSN: O: 2319-6475, ISSN: P: 2319-6505, Impact Factor: SJIF: 5.995
Available Online at www.journalijcar.org
Volume 7; Issue 2(E); February 2018; Page No. 9924-9927
DOI: http://dx.doi.org/10.24327/ijcar.2018.9927.1656

Research Article

AN INTRODUCTION ON SOFTWARE DESIGN PATTERNS

Deepa Raj*

Department of Computer Science, BBAU (Central University) Lucknow

A R T I C L E I N F O

Article History:
Received 18th November, 2017
Received in revised form 4th December, 2017
Accepted 23rd January, 2018
Published online 28th February, 2018

Key words:
Design patterns, information visualization, software engineering, object-oriented programming

A B S T R A C T

Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development. Software design patterns are recognized as a valuable part of good engineering practices. Capture successful solutions in design patterns, abstract descriptions of interacting software components that can be customized to solve design problems within a particular context. This paper discussed about how design pattern is useful for design a software using the concepts of GOF design patterns classification such as structural, creational and behavioral design pattern

INTRODUCTION

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. Find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Design should be specific to the problem at hand but also general enough to address future problems and requirements. Avoid redesign, or at least minimize it.

Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what good object-oriented design is all about.

One thing expert designers know *not* to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

Corresponding author:* **Deepa Raj
Department of Computer Science, BBAU (Central University) Lucknow

What is a Design Pattern

Christopher Alexander says, "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".

Definitions

Some definitions are

“Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development.” [Pree (94)]

“Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design… and implementation.” [Coplien *et al.* (95)].

“A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it” [Buschmann, *et. al.* (96)]

“Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.” [Gamma, *et al.* (93), also know as GoF (Gang of four)]

In general, a pattern has four essential elements

The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.

The problem describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

The consequences are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample C++ and (sometimes) Smalltalk code to illustrate an implementation. Design patterns are based on practical solutions that have been implemented in mainstream object-oriented programming languages like Smalltalk and C++ rather than procedural languages (Pascal, C, Ada) or more dynamic object-oriented languages (CLOS, Dylan, Self).We chose Smalltalk and C++ for pragmatic reasons: Our day-to-day experience has been in these languages, and they are increasingly popular.

Describing Design Patterns

How do we describe design patterns? Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action. We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

***Pattern Name and Classification*:** The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

***Intent*:** A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

***Also Known*: As** Other well-known names for the pattern, if any.

***Motivation*:** A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

***Applicability*:** What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure: A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT). We also use interaction diagrams to illustrate sequences of requests and collaborations between objects.

***Participants*:** The classes and/or objects participating in the design pattern and their responsibilities.

***Collaborations*:** How the participants collaborate to carry out their responsibilities.

Consequences How does the pattern support its objectives? What are the trades-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

***Implementation*:** What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

***Sample Code*:** Code fragments that illustrate how you might implement the pattern inC++ or Smalltalk.

***Known Uses* :** Examples of the pattern found in real systems. We include at least two examples from different domains.

***Related Patterns*:** What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Gof Design Patterns Classification

The common 23 design patterns which are divided under three categories and it was given by Gamma, Helm, Johnson and Vlissides known as Gang of Four (GoF):

Creational Design Pattern

This design patterns is all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

***Abstract Factory*:** **Provide** an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder **:** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method**:** Define an interface for creating an object, but let subclasses decide which class to instantiate. **Factory** Method lets a class defer instantiation to subclasses.

Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton**:** Ensure a class only has one instance, and provide a global point of access to it.

Structural Design Pattern

This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

Adapter**:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge**: Decouple** an abstraction from its implementation so that the two can vary independently.

Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treating individual objects and compositions of objects uniformly.

Decorator**:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Façade: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight**:** Use sharing to support large numbers of fine-grained objects efficiently.

Proxy: Provide a surrogate or placeholder for another object to control access to it.

Behavioral Design Pattern

This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

Chain of Responsibility**:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command**:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Interpreter**:** Given a language, define a representation for its grammar along with an interpreter that uses there presentation to interpret sentences in the language.

Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction in dependently.

Memento: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

State**:** Allow an object to alter its behavior when it's internal state changes. The object will appear to change its class.

Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor**:** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Organizations of Classification

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them. This section classifies design patterns so that we can refer to families of related patterns. The classification helps to learn the patterns in the catalog faster, and it can direct efforts to find new atternpt as well.

Another classification of design patterns is by two criteria (Table 1). The first criterion, called **purpose**, reflects what a pattern does. Patterns can have creational, **structural**, or **behavioral** purpose.

- Creational patterns concern 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, called **scope**, specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static-fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the Object scope.

Table 1 Design pattern space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	**Factory Method**	**Adaptor**	**Interpreter Template Method**
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Benefits of Design Patterns

Design patterns have two major benefits. First, they provide a way to solve issues related to software development using a proven solution. The solution facilitates the development of highly cohesive modules with minimal coupling. They isolate the variability that may exist in the system requirements, making the overall system easier to understand and maintain. Second, design patterns make communication between designers more efficient. Software professionals can immediately picture the high-level design in their heads when they refer the name of the pattern used to solve a particular issue when discussing system design

CONCLUSION

Design Patterns as such have always been considered as a useful and promising area that describes Software Reuse of design experiences. Design patterns have been an important part of research and study in the field of software engineering. As in other engineering disciplines there is also an important role of reusing products of this discipline i.e. reuse of software. Not only products, reuse of experiences of expert designers is also very important and useful. Design patterns are one of the examples of reuse of experiences. They help designers reuse successful designs by basing new designs on prior experience. It captures experts' design decisions, etc. Thus, at the time of design, designers should look for existing design patterns to design better software. Apart from this sometimes we have legacy software that does not have any documentation. In that situation also if we identify existing, well defined design patterns, it will be easy to understand the design.

References

1. Jeffrey Heer and Maneesh Agrawala. Software Design Patterns for Information Visualization, IEEE Transactions On Visualization And Computer Graphics, VOL. 12, NO. 5, September/October 2006
2. Shuai Jiang, Huaxin Mu. Design Patterns in Object Oriented Analysis and Design, Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on Beijing, China, 2011
3. Bederson, B. B., J. Grosjean, J. Meyer. Toolkit Design for Interactive Structured Graphics, IEEE Transactions on Software Engineering, 30(8): 535-546. 2004.
4. P´eter Heged˝us, D´enes B´an, Rudolf Ferenc, and Tibor Gyim´othy. Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability, ASEA/DRBC 2012, CCIS 340, pp. 138-145, 2012, Springer Verlag Berlin Heidelberg (2012).
5. Pankhuri Jain, Sourav Shaw, and Manjari Gupta. Improving Design of Library Management System using Design Patterns, *International Journal of Advanced Research in Computer Science*, Volume 8, No. 3, 2017
6. Card, S. K., J. D. Mackinlay, B. Schneiderman (eds.). Readings in Information Visualization: Using Vision to Think. Morgan- Kaufman, 1999.
7. Chen, H. towards Design Patterns for Dynamic Analytical Data Visualization, Proceedings of SPIE Visualization and Data Analysis, 2004.
8. Chi, E. H., J. T. Riedl. An Operator Interaction Framework for Visualization Systems, IEEE Symposium on Information Visualization (InfoVis), 1998.
9. Chi, E. H. Expressiveness of the Data Flow and Data State Models in Visualization Systems, Advanced Visual Interfaces (AVI), 2002.
10. R.Subburaj Professor, Gladman Jekese, Chiedza Hwata. Impact of Object Oriented Design Patterns on Software Development, *International Journal of Scientific & Engineering Research,* Volume 6, Issue 2, February-2015, ISSN 2229-5518
11. Eick, S. G. Visual Discovery and Analysis, IEEE Transactions on Visualization and Computer Graphics, 6(10). January 2000.
12. Fekete, J.-D. The InfoVis Toolkit, IEEE Symposium on Information Visualization (InfoVis), 2004.

How to cite this article:

Deepa Raj (2018) 'An Introduction on Software Design Patterns', *International Journal of Current Advanced Research,* 07(2), pp. 9924-9927. DOI: http://dx.doi.org/10.24327/ijcar.2018.9927.1656

****