Description of Patterns

Software patterns are reusable solutions to recurring problems that occur during software development.

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.

Organizing the Catalog

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 you learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

We classify design patterns by two criteria.

The first criterion, called purpose, reflects what a pattern does. Patterns can have either 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 .

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

How Design Patterns Solve Design Problems

Design patterns solve many of the day-to-day problems object-oriented designersface, and in many different ways. Here are several of these problems and howdesignpatterns solve them.

Finding Appropriate Objects

Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations. An object performs an operation when it receives a request(or message) from a client.

Requests are the `only` way to get an object to execute an operation. Operations are the `only` way to change an object's internal data. Because of these restrictions ,the object's internal state is said to be encapsulated; it cannot be accessed directly, and its representation is invisible from outside the object.

Determining Object Granularity

Objects can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. How do we decide what should be an object?

Design patterns address this issue as well. The Facade pattern describes how to represent complete subsystems as objects, and the Flyweight pattern describes how to support huge numbers of objects at the finest granularities .Other design patterns describe specific ways of decomposing an object into smaller objects. Abstract Factory and Builder yield objects whose only responsibilities are creating other objects. Visitor and Command yield objects whose only responsibilities are to implement a request on another objector group of objects

Specifying Object Interfaces

Every operation declared by an object specifies the operation's name, the objects it takes asparameters, and the operation's return value. This is known as the operation's signature. The set of all signatures defined by an object's operations is called the interface to the object. An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object.

Specifying Object Implementations

So far we've said little about how we actually define an object. An object's implementation is defined by its class. The class specifies the object's internal data and representation and defines the operations the object can perform.

.

How to Select a Design Pattern

With more than 20 design patterns in the catalog to choose from, it might be hardto find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you. Here are several different approaches to finding the design pattern that's right for your problem:

1. Consider how design patterns solve design problems. Design patterns help you find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems. Referring to these discussions can help guide your search for the right pattern.

2. Scan Intent sections .the Intent sections fromall the patterns in the catalog. Read through each pattern's intent to find one or more that sound relevant to your problem. You can use the classification scheme to narrow your search.

3. Study how patterns interrelate .Relationships betweendesign patterns graphically. Studying these relationships can help direct you to the right pattern or group of patterns

4. Study patterns of like purpose .The catalog has three chapters, one for creational patterns, another for structural patterns, and a third for behavioral patterns. Each chapter starts off with introductory comments on the patterns and concludes with a section that compares and contrasts them. These sections give you insight into the similarities and differences between patterns of like purpose.

5. Examine a cause of redesign .Look at the causes of redesign starting to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign.

6. Consider what should be variable in your design.

This approach is the opposite of focusing on the causes of redesign. Instead of considering what might force a change to a design, consider what you want to be Able to change without redesign. The focus here is on encapsulating the concept that varies ,a theme of many design patterns. Table 1.2 lists the design aspect(s) that design patterns let you vary independently, thereby letting you change them without redesign.

How to Use a Design Pattern

Once you've picked a design pattern, how do you use it? Here's a step-by-step approach to applying a design pattern effectively:

1.Read the pattern once through for an overview .Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.

2.Go back and study the Structure, Participants, and Collaborations sections. Make sure you understand the classes and objects in the pattern and how they relate to one another.

3.Look at the Sample Code section to see a concrete example of the pattern in code. Studying the code helps you learn how to implement the pattern.

4.Choose names for pattern participants that are meaningful in the application context .The names for participants in design patterns are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application. That helps make the pattern more explicit in the implementation .For example, if you use the Strategy pattern for a text compositing algorithm ,then you might have classes Simple Layout Strategy or TeXt Layout Strategy.

5.Define the classes .Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.

6. Define application-specific names for operations in the pattern. Here again ,the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions. For example, you might use the "Create-" prefix consistently to denote a factory method

7. Implement the operations to carry out the responsibilities and collaborations in the pattern .The Implementation section offers hints to guide you in the implementation. The examples in the Sample Code section can help as well.

Software design pattern

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. So patterns are formalized best practices that you must implement yourself in your application.[1] Object-oriented design patterns typically show relationships and interactions betweenclasses or objects, without specifying the final application classes or objects that are involved. Many patterns imply object-orientation or more generally mutable state, and so may not be as applicable in functional programming languages, in which data is immutable or treated as such.

Design patterns reside in the domain of modules and interconnections. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.[2]

There are many types of design patterns, like

- **Algorithm strategy patterns** addressing concerns related to high-level strategies describing how to exploit application characteristics on a computing platform.
- **Computational design patterns** addressing concerns related to key computation identification.
- **Execution patterns** that address concerns related to supporting application execution, including strategies in executing streams of tasks and building blocks to support task synchronization.
- **Implementation strategy patterns** addressing concerns related to implementing source code to support
    1. program organization, and
    2. the common data structures specific to parallel programming.
- **Structural design patterns** addressing concerns related to high-level structures of applications being developed.

Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioral patterns, and described using the concepts of delegation,aggregation, and consultation. For further background on object-oriented design, see coupling and cohesion, inheritance, interface, and polymorphism. Another classification has also introduced the notion of architectural design pattern that may be applied at the architecture level of the software such as the Model–View–Controller pattern.

Creational pattern

In software engineering, **creational design patterns** are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Creational design patterns are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system use. Another is hiding how instances of these concrete classes are created and combined.[1]

Creational design patterns are further categorized into Object-creational patterns and Class-creational patterns, where Object-creational patterns deal with Object creation and Class-creational patterns deal with Class-instantiation. In greater details, Object-creational patterns defer part of its object creation to another object, while Class-creational patterns defer its objection creation to subclasses.[2]

Five well-known design patterns that are parts creational patterns are:

- Abstract factory pattern, which provides an interface for creating related or dependent objects without specifying the objects' concrete classes.
- Factory method pattern, which allows a class to defer instantiation to subclasses.
- Builder pattern, which separates the construction of a complex object from its representation so that the same construction process can create different representation.
- Prototype pattern, which specifies the kind of object to create using a prototypical instance, and creates new objects by cloning this prototype.
- Singleton pattern. which ensures a class only has one instance, and provides a global point of access to it.

The **abstract factory pattern** is a software design pattern that provides a way to encapsulate a group of individual factories that have a common theme. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the theme. Theclient does not know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage.

The **factory method pattern** is an object-oriented design pattern to implement the concept of factories. Like othercreational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The essence of the Factory method Pattern is to "Define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.

The **builder pattern** is an object creation software design pattern. The intention is to abstract steps of construction of objects so that different implementations of these steps can construct different representations of objects. Often, the builder pattern is used to build products in accordance with the composite pattern.

The **prototype pattern** is a creational design pattern used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used to:

- avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
- avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.

In software engineering, the **singleton pattern** is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The terms comes from the mathematical concept of a singleton.

Creational patterns

Name	**Description**	**In** *Design Patterns*	**In** *Code Complete*[17]	**Other**
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.	Yes	Yes	N/A
Builder	Separate the construction of a complex object from its representation allowing the same construction process to create various	Yes	No	N/A

	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 (dependency injection[18]).	Yes	Yes	N/A
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.	No	No	PoEAA[19]
Multiton	Ensure a class has only named instances, and provide global point of access to them.	No	No	N/A
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No	No	N/A
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.	Yes	No	N/A
Resource acquisition is initialization	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	No	No	N/A
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	N/A