As the Gang of Four note in their book, there are

three types of design patterns:

* [Creational patterns](#4kbflgri8vn0) deal with the creation of objects and instances.
* [Structural patterns](#5w0su4bblwjc) deal with the structure of classes and code.
* Behavioral patterns deal with the behavior of objects.

## Inheritance

* Inheritance is a core principle of OO programming
  + Inheritance uses ‘IS-A’
* When you overuse inheritance, you can end up with designs that are quite inflexible and not amendable to change.

## Composition

* Composition uses ‘HAS-A’

## Design Principles (not patterns)

1. Identify the aspects of your code that vary and separate them from what stays the same.
   1. Encapsulate what varies  
      *“Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t”*
2. Code to interfaces, not implementations  
   “*Program to an interface*” really means “*Program to a supertype*”  
   “The declared type of variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be of any concrete implementation of the supertype, which means the class declaring them doesn’t have to know about the actual object types”
3. Favor composition over inheritance
4. Share common behavior via inheritance

## ------------------------------------------------------------------------ *Design Patterns, Elements of reusable Object-Oriented Software* Page 15

|  |
| --- |
| **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 pattern has four essential elements:

* Pattern Name
* Problem
* Solution
* Consequence

## The Catalog of Design Patterns Page 20

|  |
| --- |
| **Abstract Factory** (87)  Provide an interface for creating families of related or dependent objects without specifying their concrete classes.   * p.58 *“Factories and products are the key participants in the Abstract Factory (87) pattern. This pattern captures how to create families of related product objects without instantiating classes directly. It's most appropriate when the number and general kinds of product objects stay constant, and there are differences in specific product families. We choose between families by instantiating a particular concrete factory and using it consistently to create products thereafter. We can also swap entire families of products by replacing the concrete factory with an instance of a different one. The Abstract Factory pattern's emphasis on families of products distinguishes it from other creational patterns, which involve only one kind of product object.”* * *P.89 “Use the Abstract Factory pattern when*   + *a system should be independent of how its products are created, composed, and represented.*   + *a system should be configured with one of multiple families of products.*   + *a family of related product objects is designed to be used together, and you need to enforce this constraint.*   + *you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.* * *P.90 “The Abstract Factory pattern has the following benefits and liabilities:*   + *It isolates concrete classes. The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.*   + *It makes exchanging product families easy. The class of a concrete factory appears only once in an application—that is, where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use different product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once. In our user interface example, we can switch from Motif widgets to Presentation Manager widgets simply by switching the corresponding factory objects and recreating the interface.*   + *It promotes consistency among products. When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.* * *4. Supporting new kinds of products is difficult. Extending abstract factories to produce new kinds of Products isn't easy. That's because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses. We discuss one solution to this problem in the Implementation section.”*   **Adapter** (139)  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** (151)  Decouple an abstraction from its implementation so that the two can vary independently.   * p.64 *“The intent behind Bridge is to allow separate class hierarchies to work together even as they evolve independently.  … The Bridge pattern lets us maintain and enhance our logical windowing abstractions without touching window system- dependent code, and vice versa.”*   **Builder** (97)  Separate the construction of a complex object from its representation so that the same construction process can create different representations.  **Chain of Responsibility** (223)  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** (233)  Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.   * p.69 *“The Command pattern prescribes a uniform interface for issuing requests that lets you configure clients to handle different requests. The interface shields clients from the request's implementation. A command may delegate all, part, or none of the request's implementation to other objects.”*   **Composite** (163)  Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.   * p.47 *“The Composite pattern captures the essence of recursive composition in object-oriented terms”*   **Decorator** (175)  Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.   * p.54 *“The Decorator pattern captures class and object relationships that support embellishment by transparent enclosure.” ... “In the Decorator pattern, embellishment refers to anything that adds responsibilities to an object”*   **Facade** (185)  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.  **Factory Method** (107)  Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.   * p.56 *“We say that factories create product objects. Moreover, the products that a factory produces are related to one another”*   **Flyweight** (195)  Use sharing to support large numbers of fine-grained objects efficiently.  **Interpreter** (243)  Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.  **Iterator** (257)  Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.   * P.74 *“It's applicable not only to composite structures but to collections as well. It abstracts the traversal algorithm and shields clients from the internal structure of the objects they traverse. The Iterator pattern illustrates once more how encapsulating the concept that varies helps us gain flexibility and reusability”*   **Mediator** (273)  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 independently.  **Memento** (283)  Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.  **Observer** (293)  Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.  **Prototype** (117)  Specify the kinds of objects to create using a prototypical instance, and create  new objects by copying this prototype.  **Proxy** (207)  Provide a surrogate or placeholder for another object to control access to it.  **Singleton** (127)  Ensure a class only has one instance, and provide a global point of access to it.   * p.57 *“There's even a design pattern, Singleton, for managing well-known, one-of-a-kind objects like this.”*   **State** (305)  Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.  **Strategy** (315)  Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.   * p.56 *“The key to applying the Strategy pattern is designing interfaces for the strategy and its context that are general enough to support a range of algorithms”*   **Template Method** (325)  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** (331)  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.   * P. 80 *“The Visitor pattern captures the technique we've used to allow an open-ended number of analyses of glyph structures without having to change the glyph classes themselves. Another nice feature of visitors is that they can be applied not just to composites like our glyph structures but to any object structure. That includes sets, lists, even directed- acyclic graphs. Furthermore, the classes that a visitor can visit needn't be related to each other through a common parent class. That means visitors can work across class hierarchies.  An important question to ask yourself before applying the Visitor pattern is, Which class hierarchies change most often? The pattern is most suitable when you want to be able to do a variety of different things to objects that have a stable class structure. Adding a new kind of visitor requires no change to that class structure, which is especially important when the class structure is large. But whenever you add a subclass to the structure, you'll also have to update all your visitor interfaces to include a Visit... operation for that subclass.”* |

|  |
| --- |
| Here are several of these problems and how design patterns solve them.   * Finding Appropriate Objects * Determining Object Granularity * Specifying Object Interfaces * Specifying Object Implementations |

|  |
| --- |
| **Creational Patterns** |

|  |
| --- |
| **Structural Patterns**  Structural patterns are concerned with how classes and objects are composed to form larger structures.  Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.  **Adapter** (Wrapper)  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** (Handle/Body)  [**Composite**](#f5i1lxadikyb)  **Decorator** (Wrapper)  **Facade**  **Flyweight**  **Proxy** (Surrogate) |

## 23 Original Design Patterns Invented by the “gang of four”:

* Erich Gamma
* Richard Helm
* Ralph Johnson
* John Vlissides

## Types of Design Patterns

### Dynamically change the behavior of classes

* **Strategy**\*
* **Decorator**
* **State**

### Manage communication between objects

* **Observer**

The **Observer** pattern is a proven method to structure object so they aren’t too tightly dependant on one another

### Manage object creation

* **Singleton**

### Encapsulate aspects of code that are likely to change

* **Iterator**
* **Factory**\*

\* Used with Laravel