Module 1

Design patterns

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". Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns.

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.
- 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.
- 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.
- 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.

The **design patterns** are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. 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 choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily.

Describing design pattern (How do we describe design pattern?)

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.

The **templates** used are:-

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.

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).

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 trade-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 in C++ 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?

The catalog of design patterns

The catalog beginning contains 23 design patterns. Their names and intents are listed next to give you an overview.

Abstract Factory

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

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.

Builder

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

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 undo able operations.

Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat 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.

Facade

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

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 subclass.

Flyweight

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

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation 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 independently.

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.

Prototype

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

Proxy

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

Singleton

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

State

Allow an object to alter its behavior when it’s internal state changes. The object will appear to change its class.

Strategy

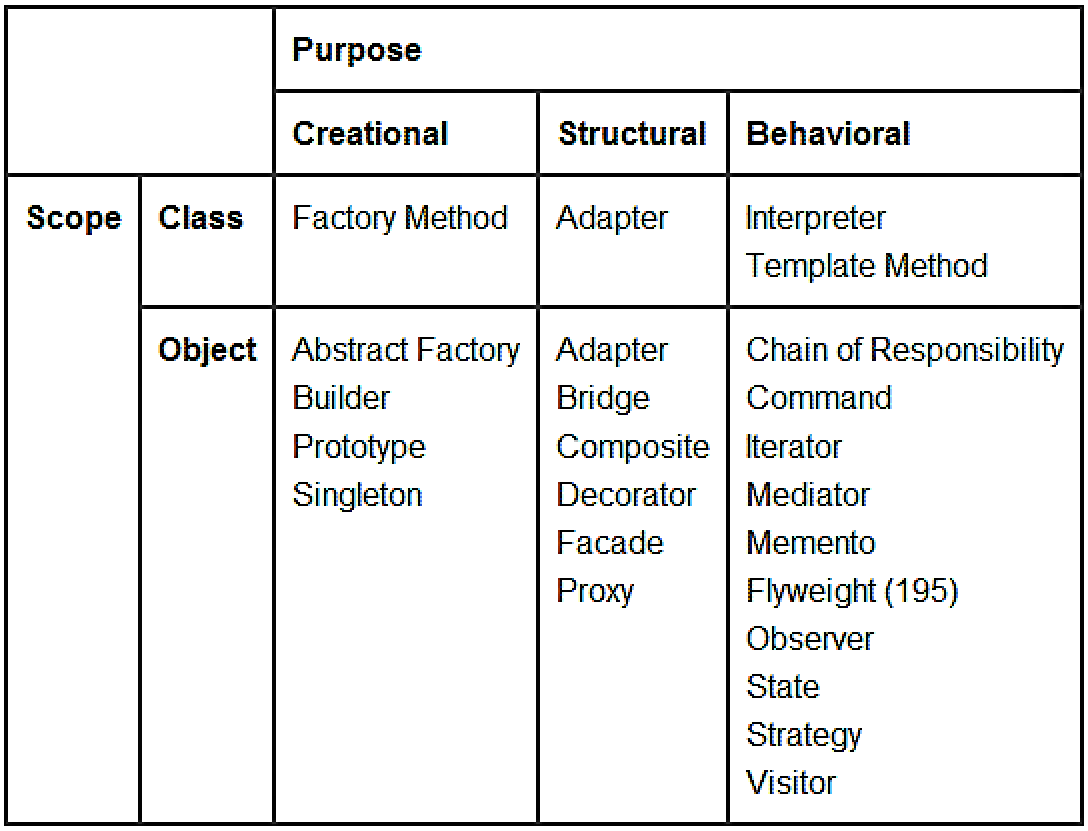
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

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.



		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight (195) Observer State Strategy Visitor

Table 1.1 Design pattern space

Organizing the catalog

We classify design patterns by two criteria (Table 1.1). 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.

Some patterns are often used together .For example; Composite is often used with Iterator or Visitor. Some patterns are alternatives: Prototype is often an alternative to Abstract Factory. Some patterns result in similar designs even though the patterns have different intents. For example, the structure diagrams of Composite and Decorator are similar.

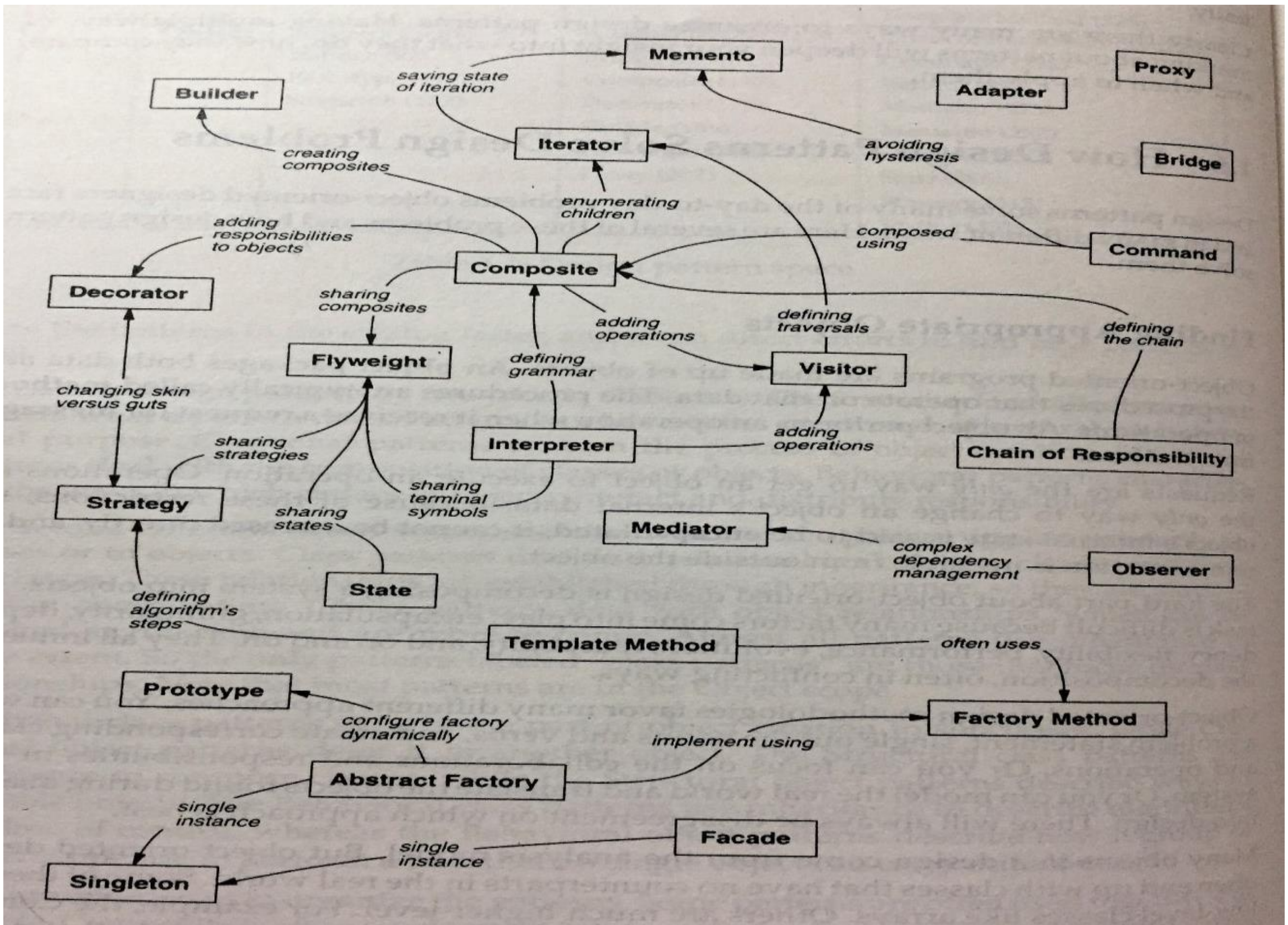


Figure 1.1: Design pattern relationships

Figure 1.1 depicts the relationship among design patterns correctly.

How design patterns solve design problem

Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways. Here are several of these problems and how design patterns 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.

Determining Object Granularity

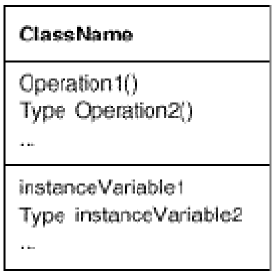
Objects can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. How do we decide what should be an object? Design patterns address this issue as well. The Facade pattern describes show to represent complete subsystems as objects, and the Flyweight pattern describes how to support huge numbers of objects at the finest granularities.

Specifying Object Interfaces

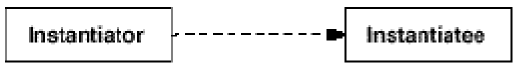
Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's **signature**. The set of all signatures defined by an object's operations is called the **interface** to the object. A **type** is a name used to denote a particular interface. We say that a type is a **subtype** of another if its interface contains the interface of **super type.**

Specifying Object Implementations

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.

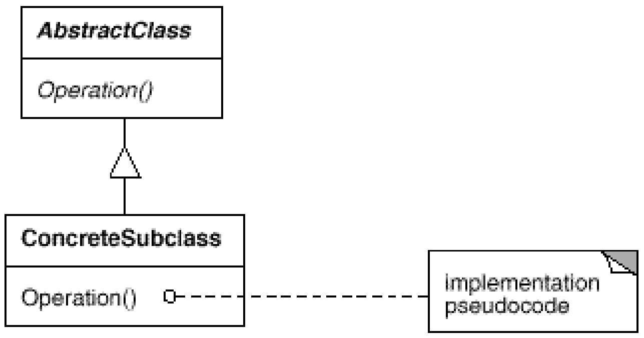


Objects are created by **instantiating** a class. The object is said to be an instance of the class. The process of instantiating a class allocates storage for the object's internal data (made up of instance variables) and associates the operations with these data. Many similar instances of an object can be created by instantiating a class.



An **abstract class** is the one whose main purpose is to define a common interface for its subclass. The operations that an abstract class declares but doesn't implement are called **abstract operations**. Classes that aren't abstract are called **concrete classes**.

A **mix in** class is a class that's intended to provide an optional interface or functionality to other classes. It's similar to an abstract class in that it’s not intended to be instantiated. Mix in classes require multiple inheritance.



Class versus Interface Inheritance

It's important to understand the difference between an object's class and its type. An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations. In contrast, an object's type only refers to its interface—the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.

Programming to an Interface, not an Implementation

Class inheritance is basically just a mechanism for extending an application's functionality by reusing functionality in parent classes.

There are **two benefits to manipulating objects** solely in terms of the interface defined by abstract classes:

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.

2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class (es) defining the interface.

Putting Reuse Mechanisms to Work

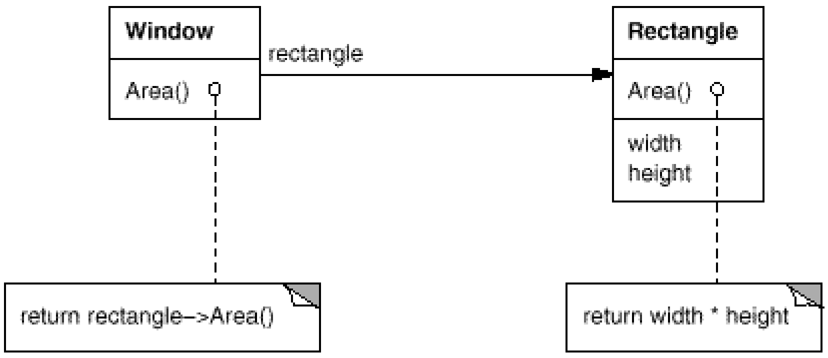
Most people can understand concepts like objects, interfaces, classes, and inheritance. The challenge lies in applying them to build flexible, re usable software, and design patterns can show you how.

Inheritance versus Composition

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition. Class inheritance lets us define the implementation of one class in terms of another's. Reuse by sub classing is often referred to as white-box reuse. The term "white-box" refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.

Delegation

Delegation is a way of making composition as powerful for reuse as inheritance.



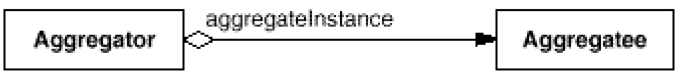
The diagram depicts the Window class delegating its Area operation to a Rectangle instance.

Inheritance versus Parameterized Types

Another (not strictly object-oriented) technique for reusing functionality is through parameterized types, also known as generics (Ada, Eiffel) and templates (C++). This technique lets us define a type without specifying all the other types it uses. The unspecified types are supplied as parameters at the point of use.

Relating Run-Time and Compile-Time Structures

An object-oriented program's run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's run-time structure consists of rapidly changing networks of communicating objects.



In diagram, a plain arrow head line denotes acquaintance. An arrow head line with a diamond at its base denotes aggregation.

Designing for Change

The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.

Here are some common causes of redesign along with the design pattern(s) that address them:

1. **Creating an object by specifying a class explicitly**. Specifying a class name when you create an object commits us to a particular implementation instead of a particular interface. This commitment can complicate future changes. To avoid it, create objects indirectly.

2. **Dependence on specific operations**. When we specify a particular operation, we commit to one way of satisfying a request. By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and at run-time.

3. **Dependence on hardware and software platform**. External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms. Software that depends on a particular platform will be harder to port to other platforms.

4. **Dependence on object representations or implementations**. Clients that knowhow an object is represented, stored, located, or implemented might need to be changed when the

Object changes. Hiding this information from clients keeps changes from cascading.

5. **Algorithmic dependencies**. Algorithms are often extended, optimized, and replaced during development and reuse. Objects that depend on an algorithm will have to change when the algorithm changes.

6. **Tight coupling**. Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes.

7. **Extending functionality by sub classing**. Customizing an object by sub classing often isn't easy. Every new class has a fixed implementation overhead (initialization, finalization, etc.). Defining a subclass also requires an in-depth understanding of the parent class.

8. **Inability to alter classes conveniently**. Sometimes you have to modify a class that can't be modified conveniently. Perhaps you need the source code and don't have it (as may be the case with a commercial class library).

Application Programs

If we are building an application program such as a document editor or spread sheet, then internal reuse, maintainability, and extension are high priorities.

Toolkits

Often an application will incorporate classes from one or more libraries of predefined classes called toolkits. A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality.

Frameworks

A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software.

Because patterns and frameworks have some similarities, people often wonder how or even if they differ. They are different in three major ways:

1. **Design patterns are more abstract than frameworks**. Frameworks can be embodied in code, but only examples of patterns can be embodied in code. Strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly of a design.

2. **Design patterns are smaller architectural elements than frameworks**. Atypical framework contains several design patterns, but the reverse is never true.

3. **Design patterns are less specialized than frameworks**. Frameworks always have a particular application domain. A graphical editor framework might be used in a factory

simulation, but it won't be mistaken for a simulation framework. In contrast, the design patterns in this catalog can be used in nearly any kind of application.

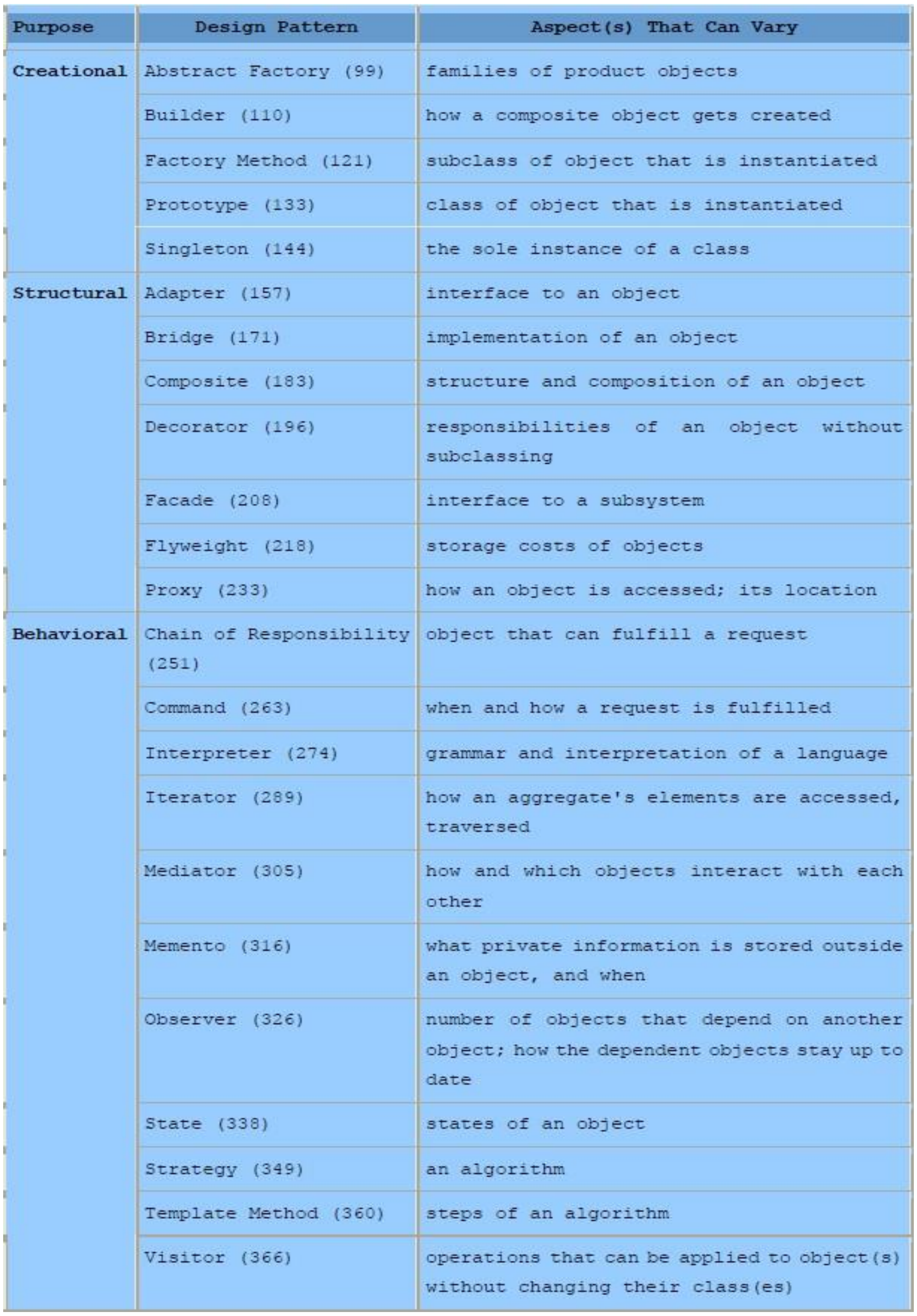
How to Select a Design Pattern

With more than 20 design patterns in the catalog to choose from, it might be hard to 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.
2. Scan Intent sections.
3. Study how patterns interrelate.
4. Study patterns of like purpose.
5. Examine a cause of redesign.
6. Consider what should be variable in your design

How to use a design pattern

1. Read the pattern once through for an overview
2. Go back and study the Structure, Participants, and Collaborations sections.
3. Look at the Sample Code section to see a concrete example of the patternin code.
4. Choose names for pattern participants that are meaningful in the application context..
5. Define the classes.
6. Define application-specific names for operations in the pattern.



Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (99)	families of product objects
	Builder (110)	how a composite object gets created
	Factory Method (121)	subclass of object that is instantiated
	Prototype (133)	class of object that is instantiated
	Singleton (144)	the sole instance of a class
Structural	Adapter (157)	interface to an object
	Bridge (171)	implementation of an object
	Composite (183)	structure and composition of an object
	Decorator (196)	responsibilities of an object without subclassing
	Facade (208)	interface to a subsystem
	Flyweight (218)	storage costs of objects
	Proxy (233)	how an object is accessed; its location
Behavioral	Chain of Responsibility (251)	object that can fulfill a request
	Command (263)	when and how a request is fulfilled
	Interpreter (274)	grammar and interpretation of a language
	Iterator (289)	how an aggregate's elements are accessed, traversed
	Mediator (305)	how and which objects interact with each other
	Memento (316)	what private information is stored outside an object, and when
	Observer (326)	number of objects that depend on another object; how the dependent objects stay up to date
	State (338)	states of an object
	Strategy (349)	an algorithm
	Template Method (360)	steps of an algorithm
	Visitor (366)	operations that can be applied to object(s) without changing their class(es)

Above table depicts the design aspects that design pattern let us vary.

What is object-oriented development?

- ✓ The object-oriented paradigm is currently the most popular way of analyzing, designing, and developing application systems, especially large ones.
- ✓ The traditional view of a computer program is that of a process that has been encoded in a form that can be executed on a computer. This view originated from the fact that the first computers were developed mainly to automate a well-defined process (i.e., an algorithm) for numerical computation, and dates back to the first stored-program computers.
- ✓ Accordingly, the software creation process was seen as a translation from a description in some 'natural' language to a sequence of operations that could be executed on a computer.
- ✓ As many would argue, this paradigm is still the best way to introduce the notion of programming to a beginner, but as systems became more complex, its effectiveness in developing solutions became suspect.
- ✓ This change of perspective on part of the software developers happened over a period of time and was fuelled by several factors including the high cost of development and the constant efforts to find uses for software in new domains. One could safely argue that the software applications developed in later years had two differentiating characteristics:

• Behaviour that was hard to characterise as a process

• Requirements of reliability, performance, and cost that the original developers did not face

- ✓ The 'process-centered' approach to software development used what is called topdown functional decomposition. The first step in such a design was to recognize what the process had to deliver (in terms of input and output of the program), which was followed by decomposition of the process into functional modules. Structures to store data were defined and the computation was carried out by invoking the modules, which performed some computation on the stored data elements.
- ✓ The life of a process-centered design was short because changes to the process specification (something relatively uncommon with numerical algorithms when compared with business applications) required a change in the entire program. This in turn resulted in an inability to reuse existing code without considerable overhead.

- ✓ As a result, software designers began to scrutinize their own approaches and also study design processes and principles that were being employed by engineers in other disciplines. Cross-pollination of ideas from other engineering disciplines started soon after, and the disciplines of 'software design' and 'software engineering' came into existence.
- ✓ **Define a software system as a collection of objects of various types that interact with each other through well-defined interfaces. Unlike a hardware component, a software object can be designed to handle multiple functions and can therefore participate in several processes.**
- ✓ **A software component is also capable of storing data, which adds another dimension of complexity to the process. The manner in which all of this has departed from the traditional process-oriented view is that instead of implementing an entire process end-to-end and defining the needed data structures along the way, we first analyze the entire set of processes and from this identify the necessary software components. Each component represents a data abstraction and is designed to store information along with procedures to manipulate the same. The execution of the original processes is then broken down into several steps, each of which can be logically assigned to one of the software components. The components can also communicate with each other as needed to complete the process.**

Key Concepts of Object-Oriented Design

During the development of this paradigm, as one would expect, several ideas and approaches were tried and discarded. Over the years the field has stabilized so that we can safely present the key ideas whose soundness has stood the test of time.

- **The Central Role of Objects**

Object-orientation, as the name implies, makes objects the centre piece of software design. The design of earlier systems was centered around processes, which were susceptible to change, and when this change came about, very little of the old system was 're-usable'. The notion of an object is centered on a piece of data and the operations (or methods) that could be used to modify it. This makes possible the creation of an abstraction that is very stable since it is not dependent on the changing requirements of the application. The execution of each process relies heavily on the objects to store the data and provide the necessary operations; with some additional work, the entire system is 'assembled' from the objects

- **The Notion of a Class Classes**

Allow a software designer to look at objects as different types of entities. Viewing objects this way allows us to use the mechanisms of classification to categories these types, define hierarchies and engage with the ideas of specialization and generalization of objects.

- **Abstract Specification of Functionality**

In the course of the design process, the software engineer specifies the properties of objects (and by implication the classes) that are needed by a system. This specification is abstract in that it does not place any restrictions on how the functionality is achieved. This specification, called an interface or an abstract class, is like a contract for the implementer which also facilitates formal verification of the entire system.

- **A Language to Define the System**

The Unified Modeling Language (UML) has been chosen by consensus as the standard tool for describing the end products of the design activities. The documents generated in this language can be universally understood and are thus analogous to the 'blueprints' used in other engineering disciplines.

- **Standard Solutions**

The existence of an object structure facilitates the documenting of standard solutions, called design patterns. Standard solutions are found at all stages of software development, but design patterns are perhaps the most common form of reuse of solutions.

- **An Analysis Process to Model a System**

Object-orientation provides us with a systematic way to translate a functional specification to a conceptual design. This design describes the system in terms of conceptual classes from which the subsequent steps of the development process generate the implementation classes that constitute the finished software.

- **The Notions of Expendability and Adaptability**

Software has a flexibility that is not typically found in hardware, and this allows us to modify existing entities in small ways to create new ones. Inheritance, which creates a new descendant class that modifies the features of an existing (ancestor) class, and composition, which uses objects belonging to existing classes as elements to constitute a new class, are mechanisms that enable such modifications with classes and objects.

Other Related Concepts

As the object-oriented methodology developed, the science of software design progressed too, and several desirable software properties were identified. Not central enough to be called object-oriented concepts, these ideas are nonetheless closely linked to them and are perhaps better understood because of these developments.

- **Modular Design and Encapsulation**

Modularity refers to the idea of putting together a large system by developing a number of distinct components independently and then integrating these to provide the required functionality. This approach, when used properly, usually makes the individual modules relatively simple and thus the system easier to understand than one that is designed as a monolithic structure. In other words, such a design must be modular. The system's functionality must be provided by a number of well-designed, cooperating modules. Each module must obviously provide certain functionality that is clearly specified by an interface. The interface also defines how other components may interact or communicate with the module. We would like that a module clearly specify what it does, but not expose its implementation. This separation of concerns gives rise to the notion of encapsulation, which means that the module hides details of its implementation from external agents. The abstract data type (ADT), the generalization of primitive data types such as integers and characters, is an example of applying encapsulation. The programmer specifies the collection of operations on the data type and the data structures that are needed for data storage. Users of the ADT perform the operations without concerning themselves with the implementation.

- **Cohesion and Coupling**

Each module provides certain functionality; **cohesion** of a module tells us how well the entities within a module work together to provide this functionality. Cohesion is a measure of how focused the responsibilities of a module are. If the responsibilities of a module are unrelated or varied and use different sets of data, cohesion is reduced. Highly cohesive modules tend to be more reliable, reusable, and understandable than less cohesive ones. To increase cohesion, we would like that all the constituents contribute to some well-defined responsibility of the module. This may be quite a challenging task. In contrast, the worst approach would be to arbitrarily assign entities to modules, resulting in a module whose constituents have no obvious relationship.

Coupling refers to how dependent modules are on each other. The very fact that we split a program into multiple modules introduces some coupling into the system. Coupling could result because of several factors: a module may refer to variables defined in another module or a module may call methods of another module and use the return values. The amount of coupling between modules can vary. In general, if modules do not depend on each others implementation, i.e., modules depend only on the published interfaces of other modules and not on their internals; we say that the coupling is low. In such cases, changes in one module will not necessitate changes in other modules as long as the interfaces themselves do not change. Low coupling allows us to modify a module without worrying about the ramifications of the changes on the rest of the system. By contrast, high coupling means that changes in one module would necessitate changes in other modules, which may have a domino effect and also make it harder to understand the code.

- **Modifiability and Testability**

A software component, unlike its hardware counterpart, can be easily modified in small ways. This modification can be done to change both functionality and design. The ability to change the functionality of a component allows for systems to be more **adaptable;** the advances in object-orientation have set higher standards for adaptability. Improving the design through incremental change is accomplished by refactoring, again a concept that owes its origin to the development of the object oriented approach. There is some risk associated with activities of both kinds; and in both cases, the organization of the system in terms of objects and classes has helped develop systematic procedures that mitigate the risk.

Testability of a concept, in general, refers to both falsifiability, i.e., the ease with which we can find counterexamples, and the practical feasibility of reproducing such counterexamples. In the context of software systems, it can simply be stated as the ease with which we can find bugs in software and the extent to which the structure of the system facilitates the detection of bugs. Several concepts in software testing (e.g., the idea of unit testing) owe their prominence to concepts that came out of the development of the object-oriented paradigm.

Benefits and Drawbacks of the Paradigm

From a practical standpoint, it is useful to examine how object-oriented methodology has modified the landscape of software development. As with any development, we do have pros and cons. The advantages listed below are largely consequences of the ideas presented in the previous sections.

1. Objects often reflect entities in application systems. This makes it easier for a designer to come up with classes in the design. In a process-oriented design, it is much harder to find such a connection that can simplify the initial design.
2. Object-orientation helps increase productivity through reuse of existing software. Inheritance makes it relatively easy to extend and modify functionality provided by a class. Language designers often supply extensive libraries that users can extend.
3. It is easier to accommodate changes. One of the difficulties with application development is changing requirements. With some care taken during design, it is possible to isolate the varying parts of a system into classes.
4. The ability to isolate changes, encapsulate data, and employ modularity reduces the risks involved in system development.

The above advantages do not come without a price tag. Perhaps the number one casualty of the paradigm is efficiency. The object-oriented development process introduces many layers of software, and this certainly increases overheads. In addition, object creation and destruction is expensive. Modern applications tend to feature a large number of objects that interact with each other in complex ways and at the same time support a visual user interface.