GOF Patterns – Complete Summary

**The Catalog of Design Patterns**

**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 undoable 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 subclassing for extending functionality.

**Facade**

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**

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

**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 its 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.

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.

The Strategy pattern describes how to implement interchangeable families of algorithms. The State pattern represents each state of an entity as an object.

The Facade pattern describes how to represent complete subsystems as objects, and the Flyweight (195) 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 object or group of objects.

Dynamic binding means that issuing a request doesn't commit you to a particular

Implementation until run-time. Moreover, dynamic binding lets you substitute objects that have identical interfaces for each other at run-time.

The Memento pattern describes how to encapsulate and save the internal state of an object so that the object can be restored to that state later.

An abstract class is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated.

Subclasses can refine and redefine behaviors of their parent classes.

**Programming to an Interface, not an Implementation**

**Inheritance versus Composition**

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition. As we've explained, class inheritance lets you define the implementation of one class in terms of another's. Reuse by subclassing 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.

Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or *composing* objects to get more complex functionality. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as "black boxes."

Inheritance and composition each have their advantages and disadvantages. Class

inheritance is defined statically at compile-time and is straightforward to use, since it's supported directly by the programming language. Class inheritance also makes it easier to modify the implementation being reused. When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.

But class inheritance has some disadvantages, too. First, you can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time. Second, and generally worse, parent classes often define at least part of their subclasses' physical representation. Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation" [Sny86]. The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.

Object composition has another effect on system design. Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task. Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters. On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.

***Favor object composition over class inheritance****.*

Delegation is a way of making composition as powerful for reuse as inheritance. In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**. The main advantage of delegation is that it makes it easy to compose behaviors at run-time and to change the way they're composed.

Several design patterns use delegation. The State , Strategy , and Visitor patterns depend on it. In the State pattern, an object delegates requests to a State object that represents its current state. In the Strategy pattern, an object delegates a specific request to an object that represents a strategy for carrying out the request. An object will only have one state, but it can have many strategies for different requests. The purpose of both patterns is to change the behavior of an object by changing the objects to which it delegates requests. In Visitor, the operation that gets performed on each element of an object structure is always delegated to the Visitor object. Delegation is an extreme example of object composition. It shows that you can always replace inheritance with object composition as a mechanism for code reuse.

Chain of Responsibility handles requests by forwarding them from one object to another along a chain of objects.

Bridge decouples an abstraction from its implementation.

Composite and Decorator are especially useful for building complex run-time structures.

**Designing for Change**

*Creating an object by specifying a class explicitly*

Design patterns: Abstract Factory , Factory Method , Prototype

*Dependence on specific operations*

Design patterns: Chain of Responsibility , Command

*Dependence on hardware and software platform.*

Design patterns: Abstract Factory , Bridge

*Dependence on object representations or implementations*

Design patterns: Abstract Factory , Bridge , Memento , Proxy

*Algorithmic dependencies*

Design patterns: Builder , Iterator , Strategy , Template Method , Visitor

*Tight coupling and Loose coupling*

Loose coupling increases the probability that a class can be reused by itself

and that a system can be learned, ported, modified, and extended more easily. Design patterns use techniques such as abstract coupling and layering to promote loosely coupled systems.

Design patterns: Abstract Factory , Bridge , Chain of Responsibility ,

Command , Facade , Mediator , Observer

*Extending functionality by subclassing.*

Design patterns: Bridge , Chain of Responsibility , Composite , Decorator ,

Observer , Strategy .

*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). Or maybe any change would require modifying lots of existing subclasses. Design patterns offer ways to modify classes in such circumstances.

Design patterns: Adapter , Decorator , Visitor .

Encapsulating an algorithm in an object is the intent of the Strategy pattern. 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.

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.

Factories and products are the key participants in the Abstract Factory pattern. This pattern captures how to create families of related product objects without instantiating classes directly.

First we define a **Command** abstract class to provide an interface for issuing a request. The basic interface consists of a single abstract operation called "Execute." Subclasses of Command implement Execute in different ways to fulfill different requests. Some subclasses may delegate part or all of the work to other objects. Other subclasses may be in a position to fulfill the request entirely on their own (see Figure 2.11). To the requester, however, a Command object is a Command object—they are treated uniformly.

**Creational Patterns**

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.

Creational patterns become important as systems evolve to depend more on object composition than class inheritance.

**Abstract Factory**

**Intent**

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

**Applicability**

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.

**Consequences**

The Abstract Factory pattern has the following benefits and liabilities:

1. *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.

2. *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.

3. *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.

AbstractFactory usually defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operation signatures. Adding a new kind of product requires changing the AbstractFactory interface and all the classes that depend on it.

**Related Patterns**

AbstractFactory classes are often implemented with factory methods (Factory Method ), but they can also be implemented using Prototype.

A concrete factory is often a singleton (Singleton ).

**Builder**

**Intent**

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

**Applicability**

Use the Builder pattern when

* the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
* the construction process must allow different representations for the object that's constructed.

**Consequences**

Here are key consequences of the Builder pattern:

1. *It lets you vary a product's internal representation.*
2. *It isolates code for construction and representation.* The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure; such classes don't appear in Builder's interface.
3. *It gives you finer control over the construction process.*

**Related Patterns**

Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.

A Composite is what the builder often builds.

**Factory Method**

**Intent**

Define an interface for creating an object, but let subclasses decide which class to

instantiate. Factory Method lets a class defer instantiation to subclasses.

**Applicability**

Use the Factory Method pattern when

* a class can't anticipate the class of objects it must create.
* a class wants its subclasses to specify the objects it creates.
* classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

**Prototype**

**Intent**

Specify the kinds of objects to create using a prototypical instance, and create new objects

by copying this prototype.

**Applicability**

Use the Prototype pattern when a system should be independent of how its products are

created, composed, and represented; *and*

* when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
* to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
* when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

**Consequences**

1. *Adding and removing products at run-time*
2. *Specifying new objects by varying values*
3. *Specifying new objects by varying structure. Reduced subclassing.* Factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all.

The main liability of the Prototype pattern is that each subclass of Prototype must

implement the Clone operation, which may be difficult.

The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references.

**Related Patterns**

Prototype and Abstract Factory are competing patterns in some ways, They can also be used together, however. An Abstract Factory might store a set of prototypes from which to clone and return product objects.

**Singleton**

**Intent**

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

**Applicability**

Use the Singleton pattern when

* there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
* when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

**Consequences**

The Singleton pattern has several benefits:

1. *Controlled access to sole instance.* Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

2. *Reduced name space.* The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.

3. *Permits refinement of operations and representation.* The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class.

**Discussion of Creational Patterns**

There are two common ways to parameterize a system by the classes of objects it creates. One way is to subclass the class that creates the objects; this corresponds to using the Factory Method pattern. The main drawback of this approach is that it can require creating a new subclass just to change the class of the product.

The other way to parameterize a system relies more on object composition: Define an object that's responsible for knowing the class of the product objects, and make it a parameter of the system. This is a key aspect of the Abstract Factory , Builder , and Prototype patterns. All three involve creating a new "factory object" whose responsibility is to create product objects. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object building a product by copying a prototype object. In this case, the factory object and the prototype are the same object, because the prototype is responsible for returning the product.

**Structural Patterns**

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural *class* patterns use inheritance to compose interfaces or implementations.

Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize new functionality.

Composite is an example of a structural object pattern. It describes how to build a class hierarchy made up of classes for two kinds of objects: primitive and composite. The composite objects let you compose primitive and other composite objects into arbitrarily complex structures. In the Proxy pattern, a proxy acts as a convenient surrogate or placeholder for another object. A proxy can be used in many ways. It can act as a local representative for an object in a remote address space.

The Flyweight pattern defines a structure for sharing objects. Objects are shared for at least two reasons: efficiency and consistency. Flyweight focuses on sharing for space efficiency.

A facade is a representative for a set of objects. The Bridge pattern separates an object's abstraction from its implementation so that you can vary them independently.

Decorator describes how to add responsibilities to objects dynamically. Decorator is a structural pattern that composes objects recursively to allow an open-ended number of additional responsibilities.

**Adapter**

**Intent**

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

**Applicability**

Use the Adapter pattern when

* you want to use an existing class, and its interface does not match the one you need.
* you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
* *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

**Related Patterns**

Bridge has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an *existing* object.

Decorator enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters.

Proxy defines a representative or surrogate for another object and does not change its interface.

**Bridge**

**Intent**

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

**Applicability**

Use the Bridge pattern when

* you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
* both the abstractions and their implementations should be extensible by subclassing.

**Consequences**

The Bridge pattern has the following consequences:

1. *Decoupling interface and implementation.* An implementation is not bound

permanently to an interface. The implementation of an abstraction can be configured at run-time.

2. Decoupling Abstraction and Implementor also eliminates compile-time

dependencies on the implementation.

**Related Patterns**

An Abstract Factory can create and configure a particular Bridge. The Adapter pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed. Bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

**Composite**

**Intent**

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

The key to the Composite pattern is an abstract class that represents *both* primitives and their containers.

**Applicability**

Use the Composite pattern when

* you want to represent part-whole hierarchies of objects.
* you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

**Consequences**

The Composite pattern

* defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
* makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.
* makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.

Equipment such as computers and stereo components are often organized into part-whole or containment hierarchies. For example, a chassis can contain drives and planar boards, a bus can contain cards, and a cabinet can contain chassis, buses, and so forth. Such structures can be modeled naturally with the Composite pattern.

Another example of this pattern occurs in the financial domain, where a portfolio aggregates individual assets. You can support complex aggregations of assets by implementing a portfolio as a Composite that conforms to the interface of an individual asset [BE93].

The Command pattern describes how Command objects can be composed and sequenced with a MacroCommand Composite class.

**Related Patterns**

Often the component-parent link is used for a Chain of Responsibility.

Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Flyweight lets you share components, but they can no longer refer to their parents.

Iterator can be used to traverse composites.

Visitor localizes operations and behavior that would otherwise be distributed across

Composite and Leaf classes.

**Decorator**

**Intent**

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Sometimes we want to add responsibilities to individual objects, not to an entire class. One way to add responsibilities is with inheritance. A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's

clients.

**Applicability**

Use Decorator

* to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
* for responsibilities that can be withdrawn.
* when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

**Consequences**

The Decorator pattern has at least two key benefits and two liabilities:

1. *More flexibility than static inheritance.* The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. Decorators also make it easy to add a property twice.

2. *Avoids feature-laden classes high up in the hierarchy.* Decorator offers a pay-as-you-go approach to adding responsibilities.

3. *A decorator and its component aren't identical.* A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.

4. *Lots of little objects.* A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables.

We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy pattern is a good example of a pattern for changing the guts.

Strategies are a better choice in situations where the Component class is intrinsically heavyweight, thereby making the Decorator pattern too costly to apply. In the Strategy pattern, the component forwards some of its behavior to a separate strategy object. The Strategy pattern lets us alter or extend the component's functionality by replacing the strategy object.

Streams are a fundamental abstraction in most I/O facilities. A stream can provide an interface for converting objects into a sequence of bytes or characters. That lets us transcribe an object to a file or to a string in memory for retrieval later. A straightforward way to do this is to define an abstract Stream class with subclasses MemoryStream and FileStream.

**Related Patterns**

Adapter : A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

Composite : A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.

Strategy : A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

**Facade**

**Intent**

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higherlevel interface that makes the subsystem easier to use.

**Motivation**

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a **facade** object that provides a single, simplified interface to the more general facilities of a subsystem.

**Applicability**

Use the Facade pattern when

* you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

**Collaborations**

* Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
* Clients that use the facade don't have to access its subsystem objects directly.

**Implementation**

Consider the following issues when implementing a facade:

1. *Reducing client-subsystem coupling.* The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing

which implementation of a subsystem is used.

An alternative to subclassing is to configure a Facade object with different subsystem objects. To customize the facade, simply replace one or more of its subsystem objects.

**Related Patterns**

Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.

Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them.

Facade objects are often Singletons

**Flyweight**

**Intent**

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

A **flyweight** is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between **intrinsic** and **extrinsic** state. Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared. Client objects are

responsible for passing extrinsic state to the flyweight when it needs it.

Flyweights model concepts or entities that are normally too plentiful to represent with objects. For example, a document editor can create a flyweight for each letter of the alphabet. Each flyweight stores a character code, but its coordinate position in the document and its typographic style can be determined from the text layout algorithms and formatting commands in effect wherever the character appears. The character code is intrinsic state, while the other information is extrinsic.

**Applicability**

The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when *all* of the following are true:

* An application uses a large number of objects.
* Storage costs are high because of the sheer quantity of objects.
* Most object state can be made extrinsic.
* Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
* The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

**Related Patterns**

The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes. It's often best to implement State and Strategy objects as flyweights.

**Proxy**

**Intent**

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

**Motivation**

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation.

**Applicability**

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different

address space.

2. A **virtual proxy** creates expensive objects on demand.

3. A **protection proxy** controls access to the original object. Protection proxies

are useful when objects should have different access rights.

**Proxy** (ImageProxy)

* maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
* provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
* controls access to the real subject and may be responsible for creating and deleting it.
* *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
* *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
* *protection proxies* check that the caller has the access permissions required to perform a request.

*Proxy doesn't always have to know the type of real subject.*

**Related Patterns**

Adapter : An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject.

A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

**Adapter versus Bridge**

The Adapter and Bridge patterns have some common attributes. Both promote flexibility by providing a level of indirection to another object. Both involve forwarding requests to this object from an interface other than its own.

The key difference between these patterns lies in their intents. Adapter focuses on

resolving incompatibilities between two existing interfaces. It doesn't focus on how those interfaces are implemented, nor does it consider how they might evolve independently. It's a way of making two independently designed classes work together without reimplementing one or the other. Bridge, on the other hand, bridges an abstraction and its (potentially numerous) implementations. It provides a stable interface to clients even as it lets you vary the classes that implement it. It also accommodates new implementations as the system evolves.

An adapter often becomes necessary when you discover that two incompatible classes should work together, generally to avoid replicating code. The coupling

is unforeseen. In contrast, the user of a bridge understands up-front that an abstraction must have several implementations, and both may evolve independently. The Adapter pattern makes things work *after* they're designed; Bridge makes them work *before* they are.

**Composite versus Decorator versus Proxy**

Decorator is designed to let you add responsibilities to objects without subclassing.

Composite has a different intent. It focuses on structuring classes so that many related objects can be treated uniformly, and multiple objects can be treated as one. Its focus is not on embellishment but on representation.

Another pattern with a structure similar to Decorator's is Proxy . Both patterns describe how to provide a level of indirection to an object, and the implementations of both the proxy and decorator object keep a reference to another object to which they forward requests. Once again, however, they are intended for different purposes.

Like Decorator, the Proxy pattern composes an object and provides an identical interface to clients. Unlike Decorator, the Proxy pattern is not concerned with attaching or detaching properties dynamically, and it's not designed for recursive composition. Its intent is to provide a stand-in for a subject when it's inconvenient or undesirable to access the subject directly.

In the Proxy pattern, the subject defines the key functionality, and the proxy provides (or refuses) access to it. In Decorator, the component provides only part of the functionality, and one or more decorators furnish the rest. Decorator addresses the situation where an object's total functionality can't be determined at compile time.

**Behavioral Patterns**

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time.

A template method is an abstract definition of an algorithm. It defines the algorithm

step by step. Each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations. The other behavioral class pattern is Interpreter , which represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.

The Mediator pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.

Chain of Responsibility provides even looser coupling. It lets you send requests to an object implicitly through a chain of candidate objects. Any candidate may fulfill the request depending on run-time conditions. The number of candidates is open-ended, and you can select which candidates participate in the chain at run-time.

The Observer pattern defines and maintains a dependency between objects.

Other behavioral object patterns are concerned with encapsulating behavior in an object and delegating requests to it. The Strategy pattern encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses. The Command pattern encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways. The State pattern encapsulates the states of an object so that the object can change its behavior when its state object changes. Visitor encapsulates behavior that would otherwise be distributed across classes, and Iterator abstracts the way you access and traverse objects in an aggregate.

**Chain of Responsibility**

**Intent**

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.

The idea of this pattern is to decouple senders and receivers by giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until one of them handles it.

**Applicability**

Use Chain of Responsibility when

* more than one object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically.
* you want to issue a request to one of several objects without specifying the receiver explicitly.
* the set of objects that can handle a request should be specified dynamically.

**Collaborations**

When a client issues a request, the request propagates along the chain until a

ConcreteHandler object takes responsibility for handling it.

**Consequences**

Chain of Responsibility has the following benefits and liabilities:

***Reduced coupling****.* The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled "appropriately."

***Receipt isn't guaranteed.***Since a request has no explicit receiver, there's no *guarantee* it'll be handled—the request can fall off the end of the chain without ever being handled. A request can also go unhandled when the chain is not configured properly.

**Command**

**Intent**

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

**Motivation**

Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. The key to this pattern is an abstract Command class, which declares an interface for executing operations. In the simplest form this interface includes an abstract Execute operation. Concrete Command subclasses specify a receiver-action pair by storing the receiver as an instance variable and by implementing Execute to invoke the request. The receiver has the knowledge required to carry out the request.

**Applicability**

Use the Command pattern when you want to

* parameterize objects by an action to perform, as MenuItem objects did above. You can express such parameterization in a procedural language with a **callback** function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.
* specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
* support undo. The Command's Execute operation can store state for reversing its effects in the command itself.
* support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations,you can keep a persistent log of changes.
* **Collaborations**
* The client creates a ConcreteCommand object and specifies its receiver.
* An Invoker object stores the ConcreteCommand object.
* The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
* The ConcreteCommand object invokes operations on its receiver to carry out the request.

**Consequences**

The Command pattern has the following consequences:

1. Command decouples the object that invokes the operation from the one that knows how to perform it.

2. Commands are first-class objects. They can be manipulated and extended like any other object.

3. You can assemble commands into a composite command. An example is the MacroCommand class described earlier. In general, composite commands are an instance of the Composite pattern.

4. It's easy to add new Commands, because you don't have to change existing classes.

The Memento pattern can be applied to give the command access.

**Related Patterns**

A Composite can be used to implement MacroCommands.

A Memento can keep state the command requires to undo its effect.

A command that must be copied before being placed on the history list acts as a Prototype

**Interpreter**

**Intent**

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

**Applicability**

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when

* the grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time.
* efficiency is not a critical concern. The most efficient interpreters are usually *not* implemented by interpreting parse trees directly but by first translating them into another form. For example, regular expressions are often transformed into state machines.

**Consequences**

The Interpreter pattern has the following benefits and liabilities:

* *It's easy to change and extend the grammar.*
* *Implementing the grammar is easy.*

**Related Patterns**

Composite : The abstract syntax tree is an instance of the Composite pattern.

Flyweight shows how to share terminal symbols within the abstract syntax tree.

Iterator : The interpreter can use an Iterator to traverse the structure.

Visitor can be used to maintain the behavior in each node in the abstract syntax tree in one class.

**Iterator**

**Intent**

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

**Motivation**

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.

**Applicability**

Use the Iterator pattern

* to access an aggregate object's contents without exposing its internal representation.
* to support multiple traversals of aggregate objects.
* to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

**Collaborations**

A ConcreteIterator keeps track of the current object in the aggregate and can

compute the succeeding object in the traversal.

**Related Patterns**

Composite : Iterators are often applied to recursive structures such as Composites.

Factory Method : Polymorphic iterators rely on factory methods to instantiate the

appropriate Iterator subclass.

Memento is often used in conjunction with the Iterator pattern. An iterator can use a

memento to capture the state of an iteration. The iterator stores the memento internally.

**Mediator**

**Intent**

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.

A mediator is responsible for controlling and coordinating the interactions of a group of objects. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections.

**Applicability**

Use the Mediator pattern when

* a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
* reusing an object is difficult because it refers to and communicates with many other objects.
* a behavior that's distributed between several classes should be customizable without a lot of subclassing.

**Consequences**

The Mediator pattern has the following benefits and drawbacks:

1. *It limits subclassing.* A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only; Colleague classes can be reused as is.

2. *It decouples colleagues.* A mediator promotes loose coupling between colleagues. You can vary and reuse Colleague and Mediator classes independently.

3. *It simplifies object protocols.* A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. Oneto- many relationships are easier to understand, maintain, and extend.

4. *It abstracts how objects cooperate.* Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior. That can help clarify how objects interact in a system.

5. *It centralizes control.* The Mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain.

**Related Patterns**

Facade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but not vice versa. In contrast, Mediator enables cooperative behavior that colleague objects don't or can't provide, and the protocol is multidirectional.

**Memento**

**Intent**

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

**Motivation**

Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.

**Applicability**

Use the Memento pattern when

* a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
* a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

**Related Patterns**

* Command : Commands can use mementos to maintain state for undoable operations.
* Iterator : Mementos can be used for iteration as described earlier.

**Observer**

**Intent**

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

The key objects in this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

This kind of interaction is also known as **publish-subscribe**.

**Applicability**

Use the Observer pattern in any of the following situations:

* When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
* When a change to one object requires changing others, and you don't know how many objects need to be changed.
* When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

**Consequences**

The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following:

1. *Abstract coupling between Subject and Observer.* All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.

2. *Support for broadcast communication.* Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it.

**Related Patterns**

Mediator : By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.

Singleton : The ChangeManager may use the Singleton pattern to make it unique and globally accessible.

**State**

**Intent**

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

Consider a class TCPConnection that represents a network connection. A TCPConnection object can be in one of several different states: Established, Listening, Closed. When a TCPConnection object receives requests from other objects, it responds differently depending on its current state. The State pattern describes how TCPConnection can exhibit different behavior in each state.

The key idea in this pattern is to introduce an abstract class called TCPState to represent the states of the network connection. The TCPState class declares an interface common to all classes that represent different operational states. TCPEstablished and TCPClosed implement behavior particular to the Established and Closed states of TCPConnection.

**Consequences**

The State pattern has the following consequences:

1. *It localizes state-specific behavior and partitions behavior for different states.*

The State pattern puts all behavior associated with a particular state into one

object. Because all state-specific code lives in a State subclass, new states

and transitions can be added easily by defining new subclasses.

1. *State objects can be shared.* If State objects have no instance variables—that is, the state they represent is encoded entirely in their type—then contexts can share a State object. When states are shared in this way, they are essentially flyweights (see Flyweight (195)) with no intrinsic state, only behavior.

**Related Patterns**

The Flyweight pattern explains when and how State objects can be shared.

State objects are often Singletons.

**Strategy**

**Intent**

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

**Applicability**

Use the Strategy pattern when

* many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
* an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
* a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

**Consequences**

* *An alternative to subclassing.* Inheritance offers another way to support a variety of algorithms or behaviors. You can subclass a Context class directly to give it different behaviors. But this hard-wires the behavior into Context. It mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically. You wind up with many related classes whose only difference is the algorithm or behavior they employ. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
* *Strategies eliminate conditional statements.* The Strategy pattern offers an alternative to conditional statements for selecting desired behavior. When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right behavior. Encapsulating the behavior in separate Strategy classes eliminates these conditional statements.

**Related Patterns**

Flyweight : Strategy objects often make good flyweights.

**Template Method**

**Intent**

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.

**Applicability**

The Template Method pattern should be used

* to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
* when common behavior among subclasses should be factored and localized in a common class to avoid code duplication. This is a good example of "refactoring to generalize" as described by Opdyke and Johnson [OJ93]. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
* to control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

It's important for template methods to specify which operations are hooks (*may* be overridden) and which are abstract operations (*must* be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

**Related Patterns**

Factory Methods (107) are often called by template methods. In the Motivation example, the factory method DoCreateDocument is called by the template method OpenDocument. Strategy : Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.

**Visitor**

**Intent**

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.

**Applicability**

Use the Visitor pattern when

* an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
* many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
* the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

**Consequences**

*Visitor makes adding new operations easy.* Visitors make it easy to add operations that depend on the components of complex objects. You can define a new operation over an object structure simply by adding a new visitor.

*Double dispatch.* Effectively, the Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a technique called **double-dispatch**.

"Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of *two* receivers. Accept is a double dispatch operation. Its meaning depends on two types: the Visitor's and the Element's. Double-dispatching lets visitors request different operations on

each class of element.

This is the key to the Visitor pattern: The operation that gets executed depends on both the type of Visitor and the type of Element it visits. Instead of binding operations statically into the Element interface, you can consolidate the operations in a Visitor and use Accept to do the binding at run-time.

Extending the Element interface amounts to defining one new Visitor subclass rather than many new Element subclasses.

A composite will commonly traverse itself by having each Accept operation traverse the element's children and call Accept on each of them recursively.

The main reason to put the traversal strategy in the visitor is to implement a particularly complex traversal, one that depends on the results of the operations on the object structure.

**Related Patterns**

Composite : Visitors can be used to apply an operation over an object structure defined by the Composite pattern.

Interpreter : Visitor may be applied to do the interpretation.

Actually, double-dispatch is just a special case of **multiple dispatch**, in which the operation is chosen based on any number of types. (CLOS actually supports multiple dispatch.) Languages that support double- or multiple dispatch lessen the need for the Visitor pattern.

**Discussion of Behavioral Patterns**

**Encapsulating Variation**

Encapsulating variation is a theme of many behavioral patterns. When an aspect of a program changes frequently, these patterns define an object that encapsulates that aspect. The patterns usually define an abstract class that describes the encapsulating object, and the pattern derives its name from that object.

* a Strategy object encapsulates an algorithm (Strategy ),
* a State object encapsulates a state-dependent behavior (State ),
* a Mediator object encapsulates the protocol between objects (Mediator ), and
* an Iterator object encapsulates the way you access and traverse the components of an aggregate object (Iterator ).

Most patterns have two kinds of objects: the new object(s) that encapsulate the aspect, and the existing object(s) that use the new ones.

Chain of Responsibility deals with an arbitrary number of objects (i.e., a chain), all of which may already exist in the system.

**Objects as Arguments**

Several patterns introduce an object that's *always* used as an argument. One of these is Visitor . A Visitor object is the argument to a polymorphic Accept operation on the objects it visits. The visitor is never considered a part of those objects, even though the conventional alternative to the pattern is to distribute Visitor code across the object structure classes.

Other patterns define objects that act as magic tokens to be passed around and invoked at a later time. Both Command and Memento fall into this category. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. In both cases, the token can have a complex internal representation, but the client is never aware of it. But even here there are differences. Polymorphism is important in the Command pattern, because executing the Command object is a polymorphic operation. In contrast, the Memento interface is so narrow that a memento can only be passed as a value. So it's likely to present no polymorphic operations at all to its clients.

**Should Communication be Encapsulated or Distributed?**

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing Observer and Subject objects, whereas a Mediator object encapsulates the communication between other objects.

In the Observer pattern, there is no single object that encapsulates a constraint. Instead, the Observer and the Subject must cooperate to maintain the constraint.

The Mediator pattern centralizes rather than distributes.

The Observer pattern promotes partitioning and loose coupling between Observer and Subject, and that leads to finer-grained classes that are more apt to be reused.

**Decoupling Senders and Receivers**

Command, Observer, Mediator, and Chain of Responsibility address how you can decouple senders and receivers, but with different trade-offs. The Command pattern supports decoupling by using a Command object to define the binding between a sender and receiver:

The Command object provides a simple interface for issuing the request (that is, the Execute operation). Defining the sender-receiver connection in a separate object lets the sender work with different receivers.

The Subject and Observer interfaces in the Observer pattern are designed for communicating changes. Therefore the Observer pattern is best for decoupling objects when there are data dependencies between them.

The Mediator pattern decouples objects by having them refer to each other indirectly through a Mediator object.

A Mediator object routes requests between Colleague objects and centralizes their communication.

Finally, the Chain of Responsibility pattern decouples the sender from the receiver by passing the request along a chain of potential receivers:

Chain of Responsibility may also require a custom dispatching scheme.

The lifecycle of object-oriented software has several phases. Brian Foote identifies these phases as the **prototyping**, **expansionary**, and **consolidating** phases.