Design Pattern

Dominik Panzarella

September 13, 2023

Contents

Chapter 1

Creational Pattern

Introduction

Creational design patterns abstract the **instantiation** proess. *They help make a system indipendent of how its objects are created, composed and represented.*

There are two recurring themes in these patterns. First, the all encapsulate knowledge about which concrete classes the system uses. Second, the hide how instances of these classes are created and put together. They let you configure a system with "product" objects that vary widely in structure and functionality. Configuration can be **static** (that is, specified at compile-time) or dynamic (at run-time).

Creational patterns are divided into:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Because the creational patterns are closely related, we'll study all five of them to highlight their similarities and differences, using a common example - building a maze for a computer game - to illustrate their implementations We'll ignore many details of what can be in a maze and whether a maze game has a single or multiple players. Instead, we'll just focus in how mazes get created

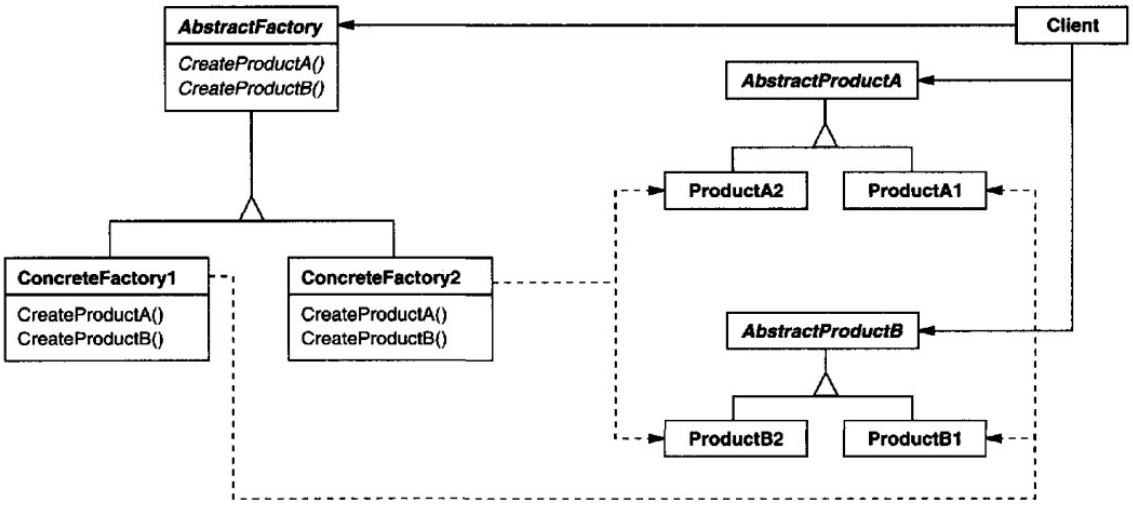


Figure 1.1: Structure of the Abstract Factory Pattern

Abstract Factory

Intent

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

Also Know As

Kit

Applicability

Use the abstract factory when:

- a system should be indipendent of how its products are creates, composed, and represented.
- a system should be configured with one of multiple families of products.
- a famility of related product objects is designed to be used together, and you need to enforce this constraint,
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Structure

See **Figure 1.1**

Partecipants

- AbstractFactory
  - declares an interface for operations that create abstract product.
- ConcreteFactory
  - implements the operations to create concrete product objects.

- AbstractProduct
  - declares an interface for a type of product object.
- ConcreteProduct
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- Client
  - uses only interface declared by AbstractFactory and AbstractProduct classes.

Consequences

Benefits and liabilites of the Abstract Factory pattern:

- It isolates concrete classes.
- It makes exchanging product families easy.
- It promotes consistency among products.
- Supporting new kinds of products is difficult.

Implementation

Useful techniques for implementing the AbstractFactory pattern:

- Factories as singletons.
- Creating the products: define a factory method for each product. If many product families are possibile, the concrete factory can be implemented using the Prototype pattern
- Defining extensible factories: add a parameter to operations that create objects.

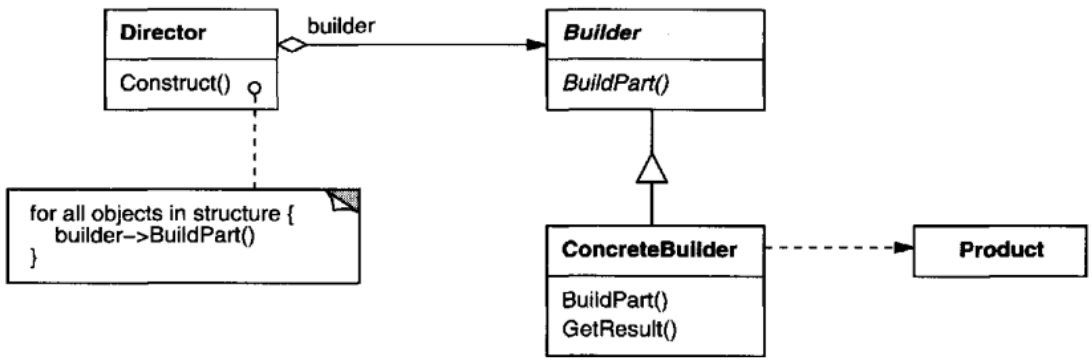


Figure 1.2: Structure of the Builder Pattern

Builder

Intent

Separate the construction of 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 indipendent of the parts that make up the object and how they are assembled.
- the construction process must allows different representation for the object that is constructed.

Structure

See **Figure 1.2**

Partecipants

- Builder
  - specifies an abstract interface for creating parts of a Product object.
- ConcreteBuilder
  - constructs and assembles parts of the product by implementing the Builder interface.
  - defines and keep track of the representation it creates.
  - provides an interface for retrieving the product
- Director
  - constructs an object using the Builder interface.
- Product
  - represents the complex object under construction. ConcreteBuilder build the product's internal representation and defines the process by which it is assembled.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result-

Consequences

Key consequences:

1. It lets you vary a product's internal representation.
2. It isolates code for construction and representation.
3. It gives you finer control over the construction process.

Implementation

Typically, there is an AbstractFactory class that defines an operation for each component that a director may ask it to create - they do nothing by default. A ConcreteBuilder class overrides operations for components it is interested in creating.
Other implementation issue to consider:

1. Assembly and construction interface
2. Why no abstract class for products? In the common case, the products produced by the concrete builders differ so greatly that there is a little or null gain from using a common parent class.
3. Empty methods as default in Builder.

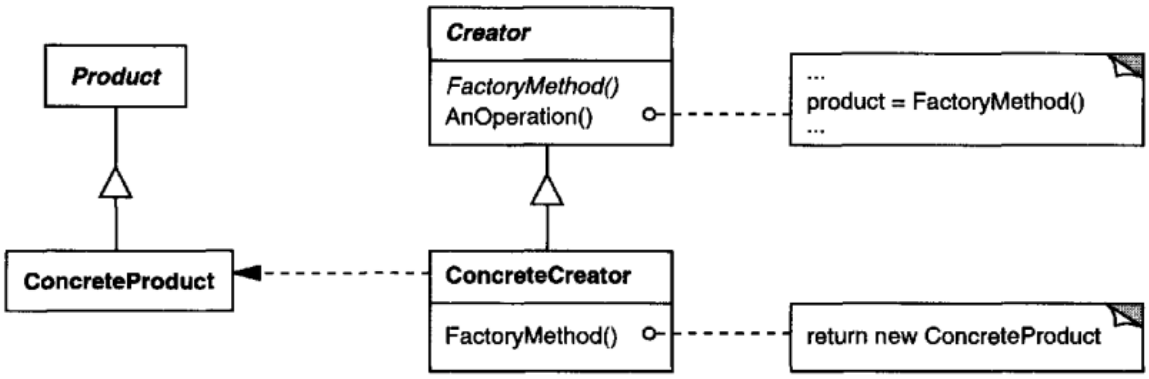


Figure 1.3: Strucre of the Factory Method Pattern.

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.

Also Know As

Virtual Constructor

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 responsability to one of several helper subclasses, and you want to localize the knoledge of which helper subclass is the delegate.

Structure

See **Figure 1.3**.

Partecipants

- Product
  - defines the interface of objects the factory method creates.
- ConcreteFactory
  - implements the Product interface.
- Creator
  - declares the factory method, which returns an object of type Prodcut. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- ConcreteCreator
  - overrides the factory method to return an instance of a ConcreteProduct.

Consequences

A potential disadvantage: client may have to subclass the Creator class just to create a particular ConcreteProduct object.
Two additional consequences of the Factory Method pattern:

1. Provides hooks for subclasses.
2. Connects parallel class hierarchies.

Implementations

Consider the following issue:

1. Two major varieties
   (a) the Creator class is an abstract class and do not provide an implementation for the factory method it declares (it is also possible to have an abstract class that define a default implementation).
   (b) the Creator is a concrete class and provides a default implementation for the factory method .
2. Parameterized factory methods.
3. Language-specific variants and issue.
4. Using templates to avoid subclassing.
5. Naming conventions.

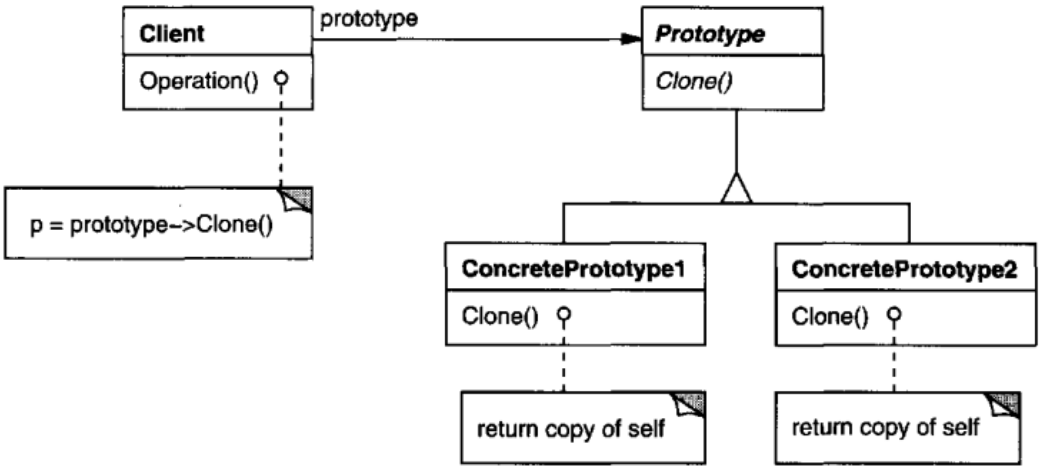


Figure 1.4: Structure of the Prototype Pattern.

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 indipendent of how its products are create, composed, and represented; **and**:

- when the classes to instantiate are specified at run-time; *or*
- to avoid building a class hierarchy of factories that parallels the class hierarchy of product; *or*
- when instances of a class can 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

Structure

See **Figure 1.4**.

Partecipants

- Prototype
  - declares an interface for cloning itself.
- ConcretePrototype
  - implements an operation for cloning itself.
- Client
  - creates a new object by asking a prototype to clone itself.

Consequences

Prototype has as many of the same consequences that Abstract Factory and Builder have: it hides the concrete product classes from the client. Here additional benefits:

1. Adding and removing products at run-time.
2. Specifying new objects by varying values.
3. Specifying new objects by varying structure.
4. Reduces subclassing.
5. Configuring an application with classes dynamically

Implementations

Consider the following issues when implementing prototypes:

1. Using a prototype manager.
2. Implementing the Clone operation.
3. Initializing clones

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 client 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.

Structure

See **Figure 1.5**.

Participants

- Singleton
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - may be responsible for creating its own unique instance.

Consequences

The Singleton pattern has several benefits:

- Controlled access to sole instance.
- Reduced name space.
- Permits refinement of operations and representation.
- Permits a variable number of instance.
- More flexible than class operations.

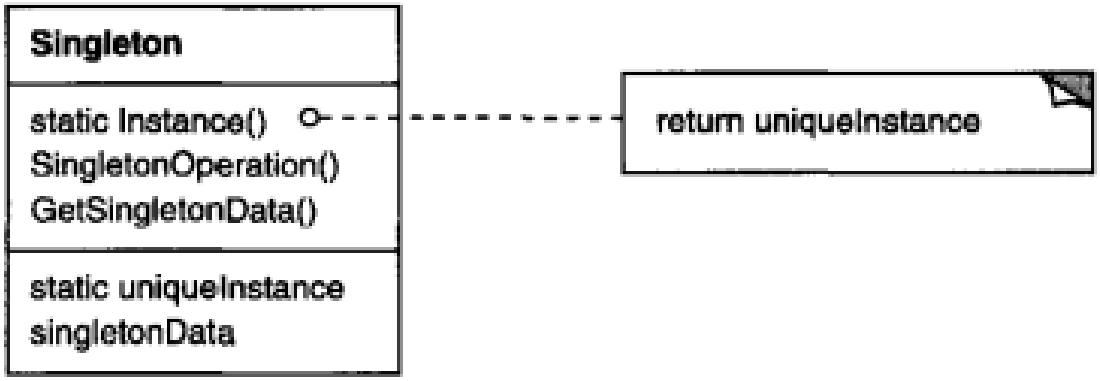


Figure 1.5: Structure of the Singleton Pattern.

Implementation

Here are implementation issues to consider when using the Singleton pattern:

- Ensuring a unique instance.
- Subclassing the Singleton class. The main issue is not defining the subclass but installing its unique instance so that clients will be able to use it. In essence, **the variable that refers to the singleton instance must be initialized with an instance of the subclass**. There are three different implementations:

  1. Determine which singleton you want to use in the Singleton's Instance operation.
  2. Take the implementation of Instance out of the parent class and put it in the subclass.
  3. Uses a **registry of singleton**. the Singleton classes can register their singleton instance by name in a well-known registry;the registry maps between string names and singleton. This approach frees Instance operation from knowing all possible Singleton class or instances, all it requires is a common interface for all Singleton classes.

Chapter 2

Structural Pattern

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. In the other hand, structural *object* patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition come from the ability to change the composition at **run-time**, which is impossible with static class composition.

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.

Also Known As

Wrapper

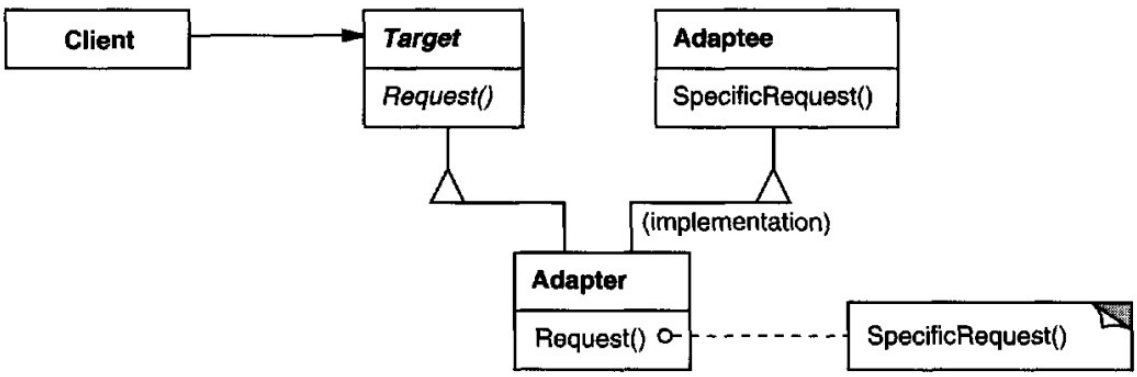
Applicability

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.

Structure

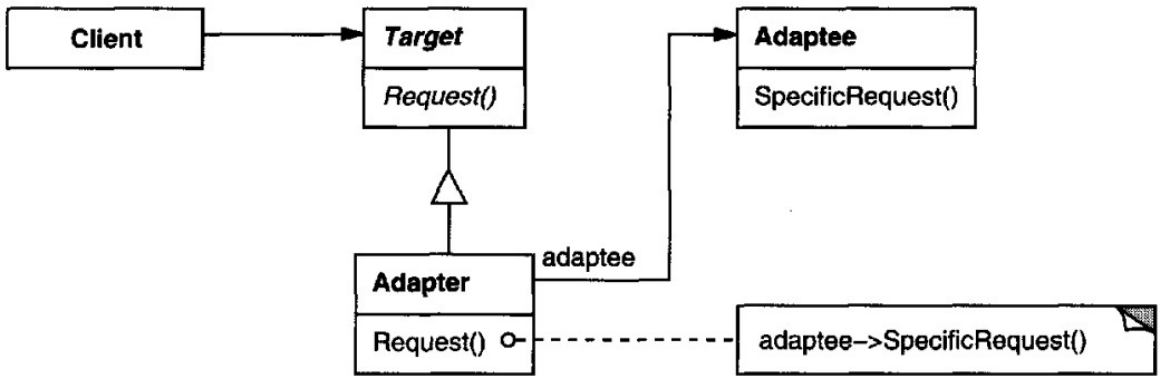
A **class adapter** uses multiple inheritance to adapt one interface to another:



An **object adapter** relies on object composition:

Partecipants

- Target
  - defines the domain-specific interface that Client uses.
- Client
  - collaborates with objects conforming to the Target interface.
- Adaptee
  - defines an existing interface that needs adapting.
- Adapter
  - adapts the interface of Adaptee to the Target interface.



Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences

A **class adapter**:

- adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
- lets Adapter override some of Adaptee's behavior.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An **object adapter**:

- lets a single Adapter work with many Adaptees. The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior.

Issue

1. *How mush adapting does Adapter do?* The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.
2. *Pluggable adapters.* A class is more reusable when you minimize the number of assumptions other classes must make to use it. By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface. The term **pluggable adapter** describes classes with built-in interface adaptation.
3. *Using two-way adapters to provide transparency.*

Implementation

Some issues to keep in mind:

1. *Implementing class adapters in C++.* Adapter would inherit publicly from Target and privately from Adaptee. Thus, Adapter would be a subtype of Target but not of Adaptee.
2. *Pluggable adapters.* There are three different ways to implement pluggable adapters. The first step, common to all three implementations, is to find a "narrow" interface for Adaptee, that is, the smalles subset of operations that lets us do the adaptation.

(a) *Using abstract operations.* Define corresponding abstract operations for the narrow Adaptee interface in the Target class. Subclasses must implement the abstract operations and adapt the hierarchically structured object.

(b) *Using delegate objects.*

(c) *Parameterized adapters.*

Bridge

Intent

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

Also Know AS

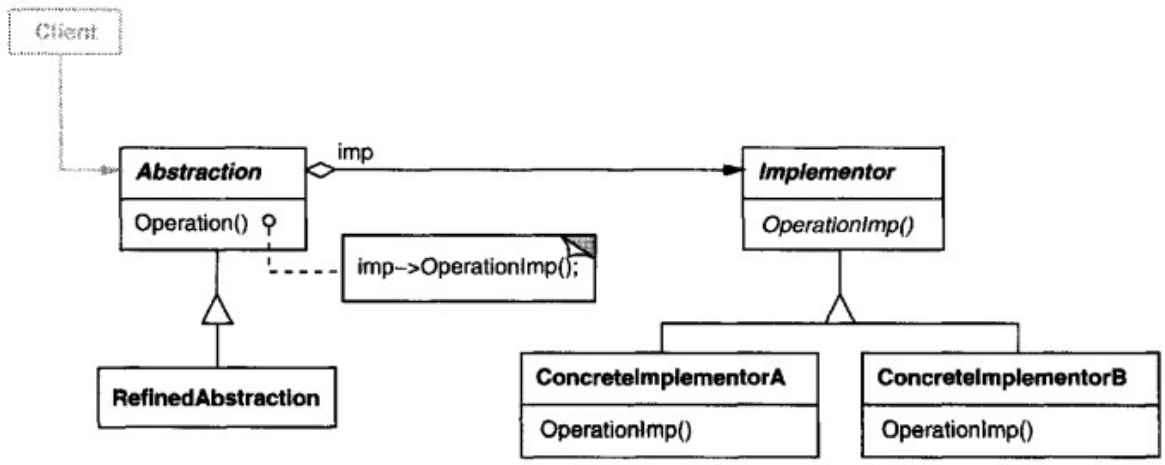
Wrapper

Applicability

Use the Bridge pattern when:

- you want to avoid a permanent binding between an abstraction and its implementation.
- both the abstractions and their implementations should be extensible by subclassing.
- changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
- (C++) you want to hide the implementation of an abstraction completely from clients.
- you have proliferation of classes. Such class hierarchy indicates the need for splitting an object into two parts.
- you want to share an implementation among multiple objects, and this fact should be hidden from the client.

Structure



Partecipants

- Abstraction
  - defines the abstraction's interface.
  - mantains a reference to an object of type Implementor.
- RefinedAbstraction
  - extends the interface defined by Abstraction.

- Implementor
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface. . Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- ConcreteImplementor
  - implements the Implementor interface and defines its concrete implementation.

Collaborations

- Abstraction forwards client requests to its Implementor object.

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. It is even possible for an object to change its implementation at run-time. Decoupling Abstraction and Implementor also eliminates compile-time dependencies on the implementation.
2. *Improved extensibility.*
3. *Hiding implementation details from clients.*

Implementation

Consider the following implementation issues when applying the Bridge pattern:

1. *Only one implementor.* In situations where there's only one implementation, creating an abstract Implementor class isn't necessary; ; there's a one-to-one relationship between Abstraction and Implementor
2. *Creating the right Implementor object.* **How, when, and where do you decide which Implementor classto instantiate when there's more than one?**
   (a) If Abstraction knows about all Concretelmplementor classes, then it can instantiate one ofthem in its constructor; it can decide between them based on parameters passed to its constructor.
   (b) Another approach is to choose a default implementation initially and change it later according to usage.
   (c) It's also possible to delegate the decision to another object altogether.
3. *Sharing implementors.*
4. *Using multiple inheritance.* You can use multiple inheritance in C++ to combine an interface with its implementation. But because this approachrelies on staticinheritance, it binds an implementation permanently to its interface. Therefore you can't implement a true Bridge with multiple inheritance—at least not in C++

Composite

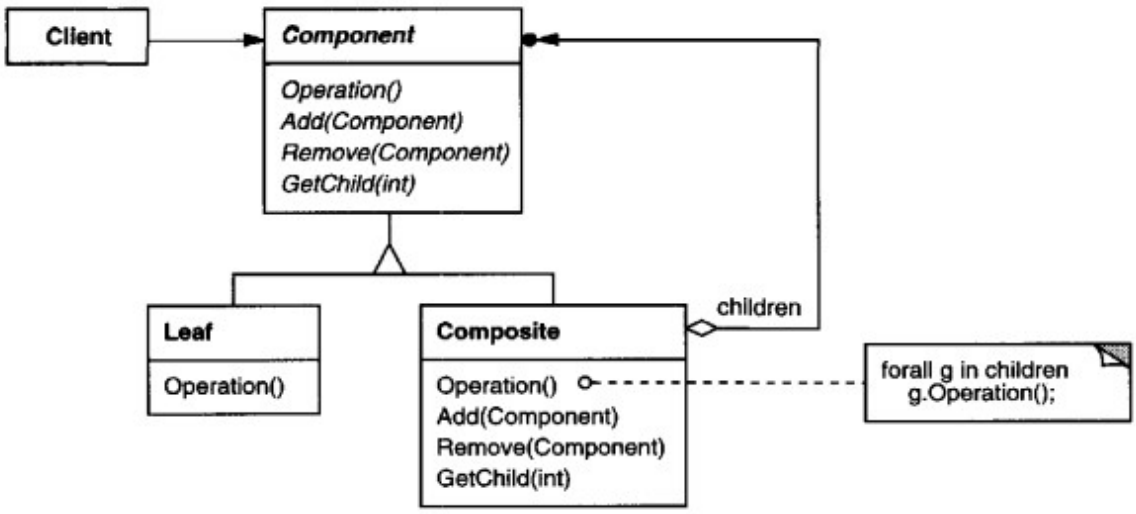
Intent

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

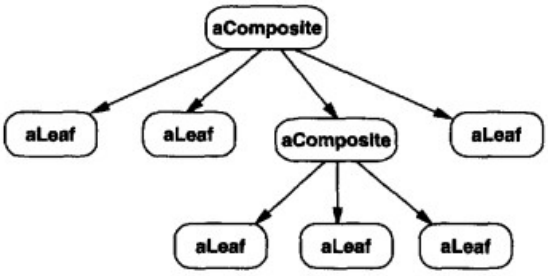
Applicability

- 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.

Structure



A typical Composite object structure might look like this:



Partecipants

- Component
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes.
  - declares an interface for accessing and managing its child components.

  - (optional) define an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

- Leaf
  - represents leaf objects in the composition. A leaf has no children.
  - define behavior for primitive objects in the composition.

- Composite
  - define behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.

- Client
  - manipulates objects in the composition through the Component interface.

Collaborations

Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

The composite pattern

- defines class hierarchies consisting of primitive objects and composite objects. Wherever client code expects a primitive object, it can also take a composite object.
- makes the client simple. e. 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.
- make it easier to add new kinds of components. Clients don't have to be changed for new Component classes,
- can make your desing overly general.

Implementation

Consider the following implementation issues when applying the Composite pattern:

1. *Explicit parent references.* Maintaining references from child components to their parent can simplify the traversal and management of a composite structure. The usual place to define the parent reference is in the Component class. With parent references,it's essential tomaintain the invariant that all children of a composite have as their parent the composite that in turn has them as children. The easiest way to ensure this is to change a component's parent only when it's being added or removed from a composite.
2. *Sharing components.* A possible solution is for children to store multiple parents.
3. *Maximizing the Component interface.* One of the goals of the Composite pattern is to make clients unaware of the specific Leaf or Composite classes they're using. To attain this goal, the Component class should define as many common operations for Composite and Leaf classes as possible. *The Component class usually provides default implementations for these operations, and Leaf and Composite subclasses will override them.*

4. *Declaring the child management operations.* The decision involves a trade-off between safety and transparency:

   - Defining the child management interface atthe root of the class hierarchy gives you transparency, because you can treat all components uniformly. It cost you safety, , because clients may try to do meaningless things like add and remove objects from leaves.
   - Defining child management in the Composite class gives you safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like C++. But you lose transparency, because leaves and composites have different interfaces.

   The only way to provide transparency is to define default Add and Remove operations in Component, but there's no way to implement Component's add operation without introducing the possibility of it failing.

   (a) Usually it's better to make Add and Remove fail by default (perhaps by raising an exception) if the component isn't allowed to have children or if the argument of Remove isn't a child of the component, respectively.

   (b) Another alternative is to change the meaning of "remove" slightly. If the component maintains a parent reference, then we could redefine Composite's Remove operation to remove itself from its parent.

5. *Should Component implement a list of Components?* But putting the child pointer in the base class incurs a space penalty for every leaf, even though a leaf never has children.This is worthwhile only if there are relatively few children in the structure.

6. *Child ordering.*

7. *Caching to improve performance.* If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children. The Composite can cache actual results or just information that lets it short-circuit the traversal or search. Changes to a component will require invalidating the caches of its parents. This works best when components know their parents. So if you're using caching, you need to define an interface for telling composites that their caches are invalid.

8. *Who should delete component?* In languages without garbage collection, it's usually best to make a Composite responsible for deleting its children when it's destroyed. An exception to this rule is when Leaf objects are immutable and thus can be shared.

9. *What is the best data structure for storing component?* . The choice of data structure depends (as always) on efficiency. In fact, it isn't even necessary to use a general-purpose data structure at all.

Decorator

Intent

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

Also Known As

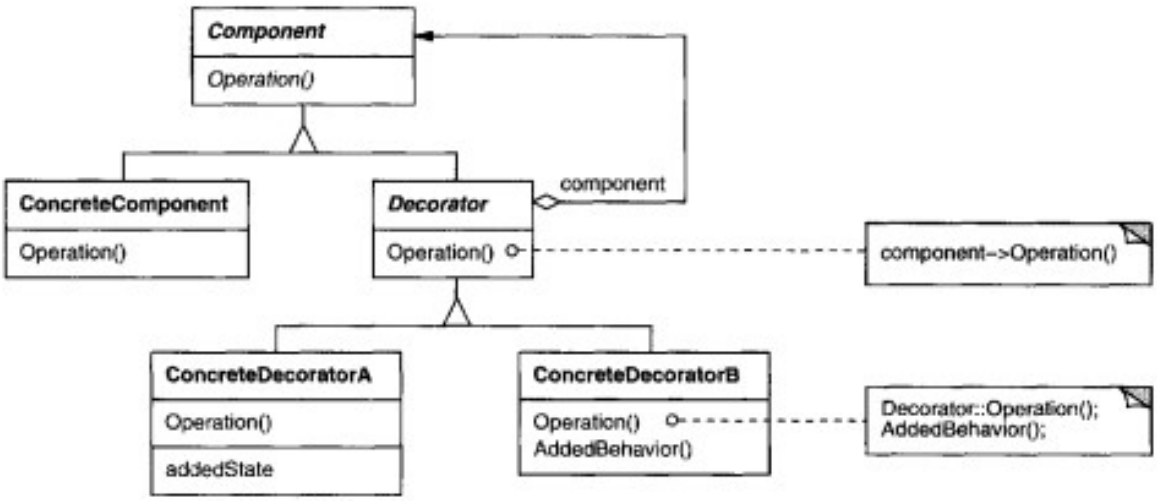
Wrapper

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 or a class definition may be hidden or otherwise unavailable for subclassing.

Structure



Partecipants

- **Component**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent**
  - defines an object to which additional responsibilites can be attached.
- **Decorator**
  - mantains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**
  - adds responsibilities to the component.

Collaborations

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences

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

1. *More flexibility that static inheritance.* With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility. Decorator also make it easy to add a property twice.

2. *Avoid feature-laden classes high up in the hierarchy.*

3. *A decorator and its component aren't identical.* l. 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.

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.

Implementation

Consider the following implementation issues when applying the Decorator pattern:

1. *Interface conformance.* A decorator object's interfacemust conformto the interface ofthe component it decorates.ConcreteDecoratorclasses must therefore inherit from a common class (at least inC++).

2. *Omitting the abstract Decorator class.* There's no need to define an abstract Decorator class when you only need to add one responsibility. . In that case, you can merge Decorator's responsibility for forwardingrequests to the component into theConcreteDecorator.

3. *Keeping Component classes lightweight.* Components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data. The definition of the data representation should be deferred to subclasses.

4. *Changing the skin of an object versus changing its guts.* 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.

Facade

Intent

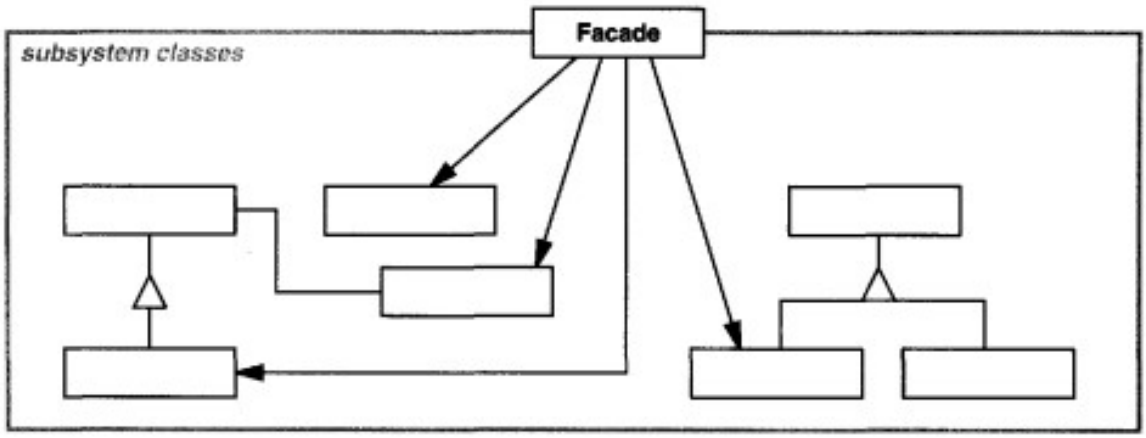
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Applicability

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem. A Facade can provide a simple default view of the subsystem that is good enough for molt clients. Only clients needing more customizability will need to loo beyond the Facade.
- there aremany dependencies between clients and theimplementation classes of an abstraction.Introduce a facade to decouple the subsystem from clients and other subsystems.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Structure



Participants

- **Facade**
  - knows which subsystem classes are responsible for a request
  - delegates client requests to appropriate subsystem objects.
- **subsystem classes**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

Collaborations

- Clients communicate with the subsystem by sending requests to Facade,which forwards them to the appropriate subsystem object(s).
- Clients that use the facade don't have to access its subsystem objects directly.

Consequences

The Facade pattern offers the following benefits:

1. It shields clients from subsystem components.
2. It promotes weak coupling between the subsystem and its clients.
3. it doesn't prevent applications from using subsystem classes if they need to.

Implementation

Consider the following issues when implementing a Facade:

1. *Reducing client-subsystem coupling.*
   (a) A solution is by making Facade an abstract class with concrete subclasses for different implementations of a subsystem.
   (b) An alternative to subclassing is to configure a Facade object with different subsystem objects.
2. *Public versus private subsystem classes.* The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. The Facade class is part of the public interface, of course, but it's not the only part. Other subsystem classes are usually public as well. Making subsystem classes private would be useful, but few object-oriented languages support it.

Flyweight

Intent

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

Note

A flyweightis 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.

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:

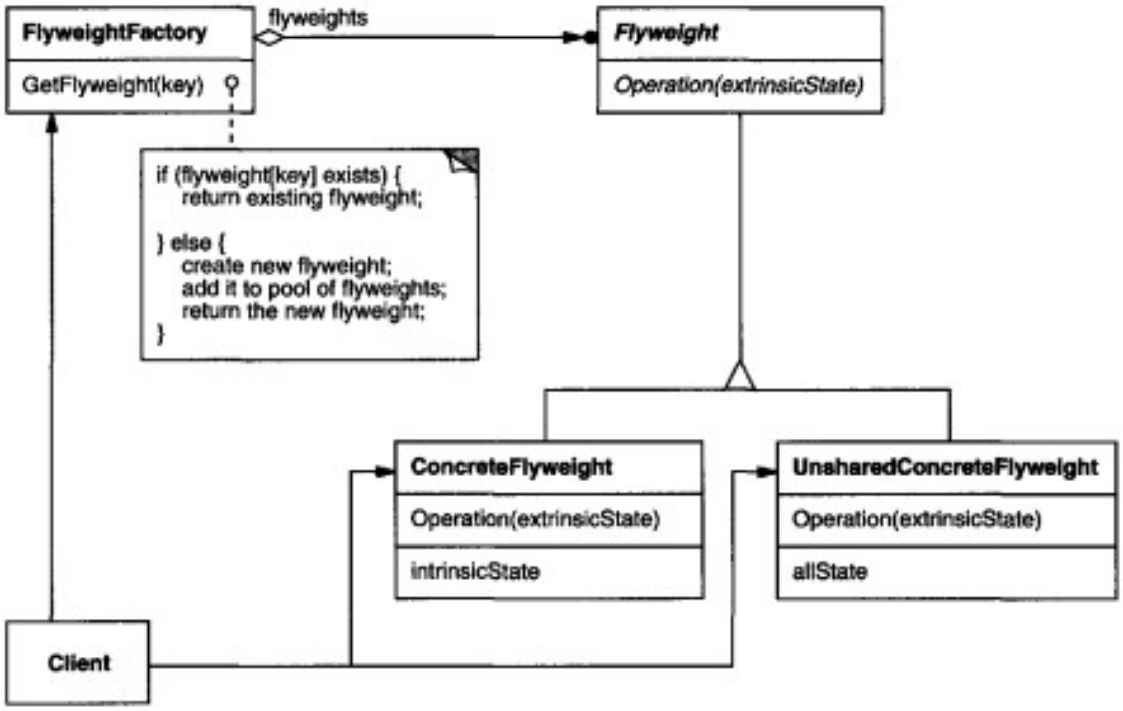
- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many group of objects may be replaced by relatively few shared objects one extrinsic state is removed.
- The application doesn't depend on object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

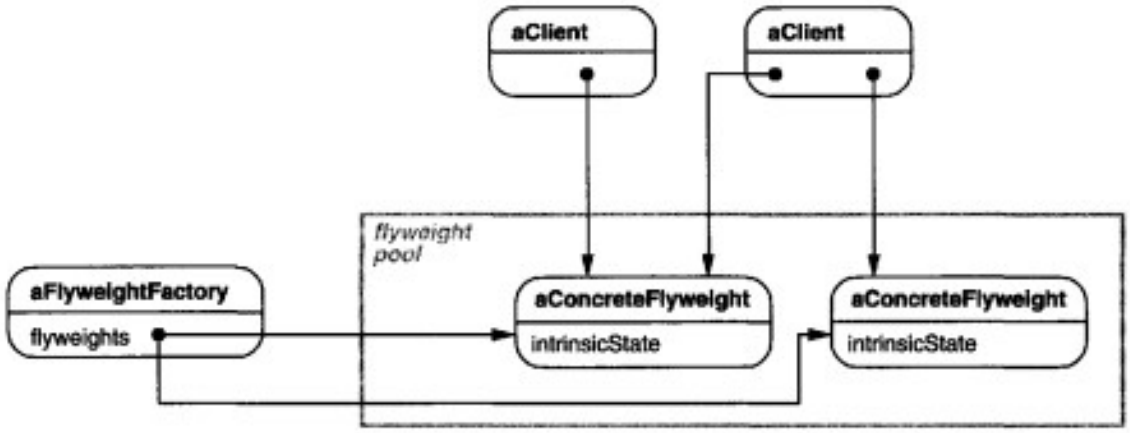
Structure

The following object diagram shows how flyweights are shared:

Participants

- **Flyweight**
  - declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight**
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A Concrete-Flyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent ofthe ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight**
  - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it.
- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly.
- **Client**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).





Collaborations

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic. Intrinsic state is stored in the ConcreteFlyweight object; extrinsic state is stored or computed by Client objects. Clients pass this state to the flyweight when they invoke its operations

- Clients should notinstantiate ConcreteFlyweights directly.Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Consequences

Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by

space savings, which increase as more flyweights are shared. Storage savings are a function of several factors:

- the reduction in the total number of instances that comes from sharing.
- the amount of intrinsic state per object.
- whether extrinsic state is computed or stored.

The more flyweights are shared, the greater the storage savings. The savings increase with the amount of shared state. The greatest savings occur when the objects use substantial quantities of both intrinsic and extrinsic state, and the extrinsic state can be computed rather than stored. Then you save on storage in two ways: Sharing reduces the cost of intrinsic state, and you trade extrinsic state for computation time.

Implementation

Consider the following issues when implementing the Flyweight pattern:

1. *Removing extrinsic state.* The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Ideally, extrinsic state can be computed from a separate object structure, one with far smaller storage requirements.

2. *Managing shared objects.* Because objects are shared, clients shouldn't instantiate them directly. FlyweightFactory lets clients locate a particular flyweight. FlyweightFactory objects often use an associative store to let clients look up flyweights of interest.
   Sharability also implies some form of reference counting or garbage collection to reclaim a flyweight's storage when it's no longer needed.However, neither is necessary if the number of flyweights is fixed and small . In that case, the flyweights are worth keeping around permanently.

Proxy

Intent

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

Also Know As

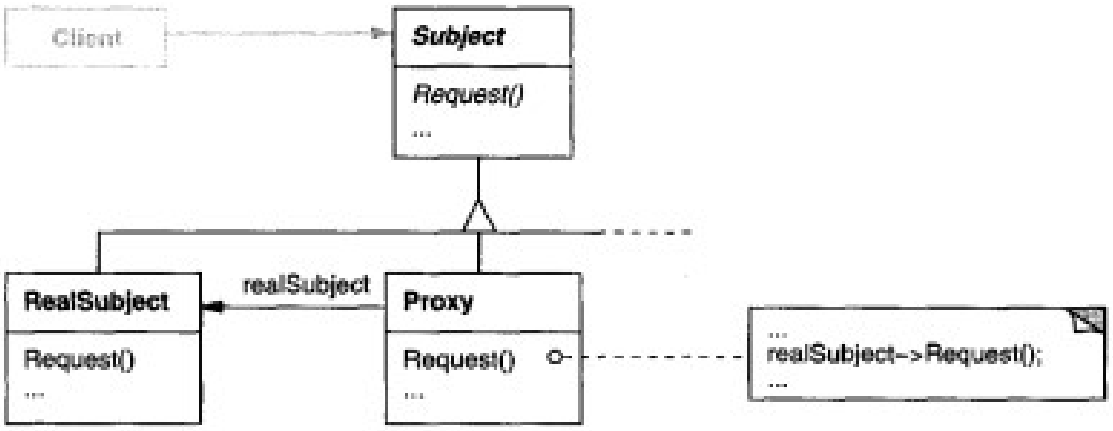
Surrogate

Applicability

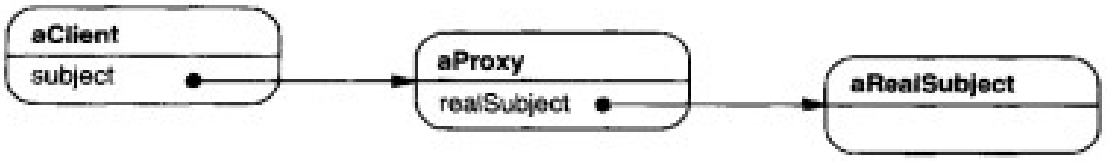
Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations:

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** control access to the original object. Protection proxies are useful when objects should have different access rights.
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include
   - counting the number of references to the real object so that it can be freed automatically when there are no more references
   - loading a peristent object into memory when it's first referenced.
   - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure



Here's a possible object diagram of a proxy structure at run-time:



Participants

- **Proxy**
  - maintains a reference that letsthe proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subjectinterfaces 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.
  - other responsibilites depend on the kind of proxy:
    - *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* s may cache additional information about the real subject so that they can postpone accessing it.
    - *protection proxies* check that the caller has the access permissions required to perform a request.
- **Subject**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- RealSubject
  - define the real object that the proxy represents.

Collaborations

- Proxy forwards request to RealSubject when appropriate, depending on the kind of proxy.

Consequences

The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

1. A remote proxy can hide the fact that an object resides in a different address space.
2. A virtual proxy can perform optimizations such as creating an object on demand.
3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

There's another optimization that the Proxy pattern can hide from the client. It's called **copy-on-write**,and it's related to creation on demand. Copying a large and complicated object can be an expensive operation. If the copy is never modified, then there's no need to incur this cost.

To make copy-on-write work, the subject must be reference counted. Copying the proxy will do nothing more than increment this reference count. Only when the client requests an operation that modifies the subject does the proxy actually copy it. In that case the proxy must also decrement the subject's reference count. When the reference count goes to zero, the subject gets deleted

Implementation

The Proxy pattern can exploit the following language features:

1. *Overloading the member access operator in C++.* C++ supports overloading operator->, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced.

   Overloading the member access operator isn't a good solution for every kind of proxy.Some proxies need to know precisely which operation is called, and overloading the member access operator doesn't work in those cases.

2. *Proxy doesn't always have to know the type of real subject.* If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubjectclasses uniformly. But if Proxies are going to instantiate RealSubjects (such as in a virtual proxy),then they have to know the concrete class.

3. *How to refer to the subject beore it's instantiated.*