Design Patterns

Andy Verkeyn
April 2001 - March 2003

Third version

Andy.Verkeyn@rug.ac.be
http://allserv.rug.ac.be/~averkeyn

Contents

Foreword

This text is mainly taken from the excellent book *"Design patterns: Elements of reusable Object-Oriented Software"*, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1995 (also known as the GoF, Gang of Four). It is provided for the students Software Engineering at the Ghent University and adapted to use the UML (Unified Modeling Language) notation. Furthermore, the given examples are all taken from C++ and Java only (in most cases from their standard libraries) and the text is a bit summarized. For a more detailed discussion about the described patterns and a number of other useful patterns, I refer to the original book.

Introduction

Probably the most important difference between a novice object oriented designer and an expert is their ability to recognize standard design problems and the reuse of a solution that turned out to be good in the past. When expert designers find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object oriented designs. These patterns solve specific design problems and make object oriented design more flexible, elegant and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them. In the context of object oriented software design, each such pattern is called a design pattern.

Each design pattern is described using a consistent format, summarized in the following table.

Name	A vital part of a design pattern to ease communication with other designers. These names should become part of the basic vocabulary.
Intent	Summarizes the rationale and the particular design issue that the pattern addresses.
Synonyms	Other frequently used names for the same design pattern (if any)
Applicability	What are the situations in which the design pattern can be applied?
Structure	A UML representation of the classes and/or objects in the pattern, and their interactions (if necessary)
Participants	The responsibilities of the classes and/or objects in the pattern
Consequents	What are the trade-offs and results of using the pattern?
Implementation issues	What pitfalls, hints or techniques could be used when implementing the pattern? Language-specific issues?
Uses	Examples of the pattern found in real systems.

Iterator

Intent

Provide a way to access the elements of an aggregate (container, collection) object sequentially without exposing its underlying representation.

Synonyms

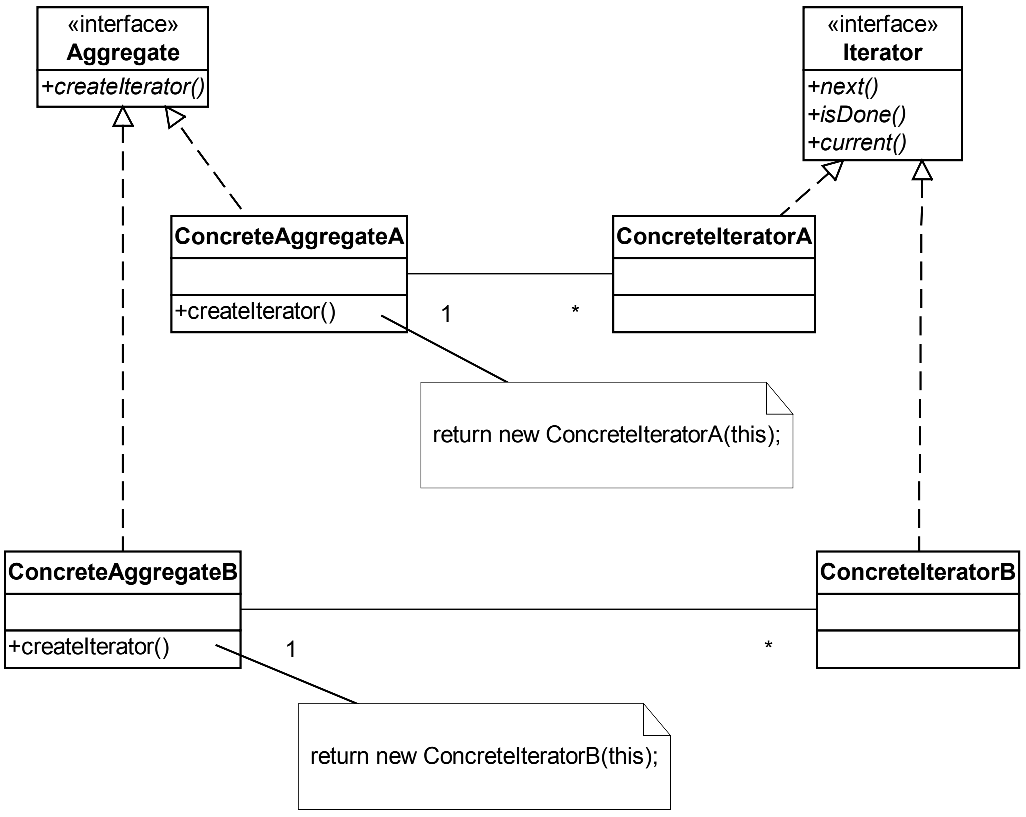
Cursor

Applicability

Use the Iterator pattern when

- 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 (to support polymorphic iteration).

Structure



Participants

Iterator	defines an interface for accessing and traversing elements
ConcreteIterator	• implements the `Iterator` interface • keeps track of the current position in the traversal of the aggregate
Aggregate	defines an interface for creating an `Iterator` object. Usually this interface also defines method to add, remove, count,... aggregated elements.
ConcreteAggregate	implements the `Iterator` creation interface to return an instance of the proper `ConcreteIterator`

Consequences

- It supports variations in the traversal of an aggregate. Complex aggregates may have several iterators that traverse its objects in different ways (e.g. a tree can be traversed in pre-order, in-order or post-order), even a simple list structure can be traversed from head to tail and reverse.
- Iterators simplify the `Aggregate` interface by removing the traversal functionality.
- More than one traversal can be pending on an aggregate.
- Because of the polymorphic iteration, the iterator implementation can easily be changed without any implications for the client code. Even creating another aggregate object (e.g. a list instead of a vector container) requires no additional changes: the `createIterator()` method will return a corresponding iterator object that has the same interface as before. This method is in fact an example of the Factory Method design pattern.

Implementation issues

- Robustness. Adding or deleting elements from the aggregate while traversing it with an iterator can corrupt the iterator. A robust iterator ensures that insertions and removals won't interfere with traversal (without copying the whole aggregate into the iterator before traversing it). Most rely on registering the iterator with the aggregate. On insertion or removal, the aggregate can take proper actions to guard the integrity of its pending iterators.

Uses

- The C++ STL collections

Iterator	STL has a complete hierarchy of iterator types, where each type provides a separate number of operations: `InputIterator`, `OutputIterator`, `ForwardIterator`, `BidirectionalIterator` and `RandomAccessIterator`
Aggregate	Although every STL container adheres somewhat to the same interface, there are no explicit abstract classes (interfaces) defined is most implementation.
ConcreteAggregate	A whole number of -template based- collection classes is available: `vector, deque, list, set, multiset, map, multimap,...`
ConcreteIterator	some nested class in the concrete aggregate class

The C++ iterators are not robust.

- The Java Collection framework:

Iterator	`Iterator` which is further specialized in `ListIterator`
Aggregate	`Collection` which is further specialized as `List` and `Set`. `Collection` contains a method `iterator` to create concrete iterator instances.
ConcreteAggregate	• `List`: `ArrayList`, `LinkedList` • `Set`: `HashSet`, `TreeSet`
ConcreteIterator	some private class in the concrete aggregate class

The java iterators are robust in the sense that they are fail-fast. When a modification of the aggregate is detected during iteration, an exception is thrown immediately. However, the iterator itself allows the removal of the last returned element in a safe way. In java, the `next()` and `current()` operations are combined in one `next()` method that returns the current element while moving to the next.

Adapter (class)

Intent

Convert the interface of a class into another interface. This intent can be accomplished in two ways: a class-based approach and an object-based approach. This chapter describes the class-based variant.

Synonyms

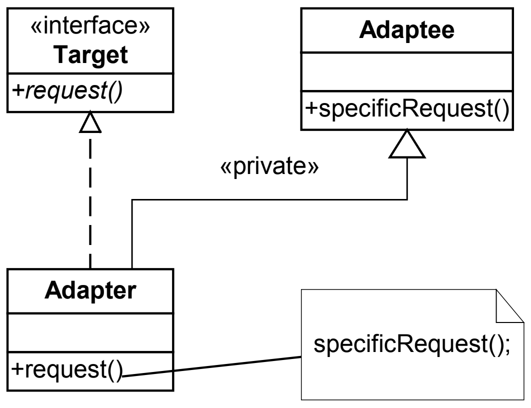
Wrapper

Applicability

Use the Adapter (class) 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.

Structure



Note: It's also possible that the `Target` is a full-blown class instead of just an interface, and that multiple inheritance is needed to implement the `Adapter` class.

Participants

Target	defines the domain-specific interface the client uses
Adaptee	defines an existing interface that needs adaption
Adapter	adapts the interface of `Adaptee` to the `Target` interface

Consequences

- A class adapter lets `Adapter` override some of `Adaptee's` behavior, since `Adapter` is a subclass of `Adaptee`.
- A class adapter introduces only one object, and no additional level of indirection is needed to get to the adaptee.
- The interface of `Adaptee` remains available in the `Adapter` class

Uses

- The Java collection classes: the class `Stack` (adapter) is derived from class `Vector` (adaptee) and provides the typical stack functionality, although the `Vector` interface is not hidden from the user. In this case, their is no separate stack interface defined.
- CORBA (Common Object Request Broker Architecture) uses the adapter pattern to adapt the implementation of the servant to the client interface by inheriting from the abstract CORBA servant (skeleton) class.

Adapter (object)

Intent

Convert the interface of a class into another interface. This intent can be accomplished in two ways: a class-based approach and an object-based approach. This chapter describes the object-based variant.

Synonyms

Wrapper

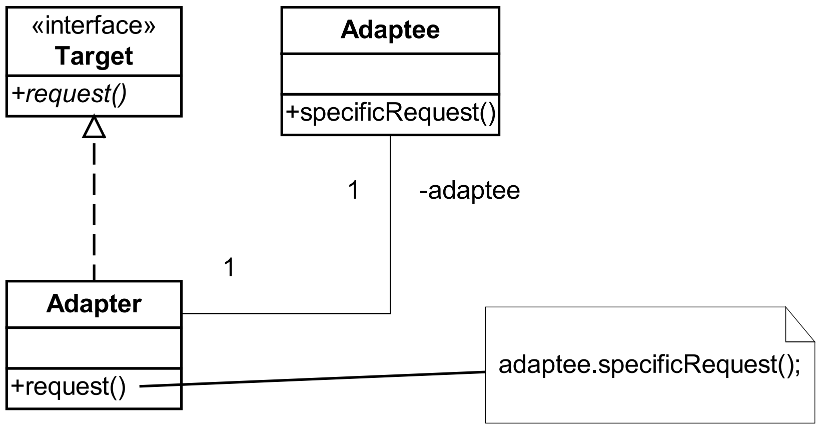
Applicability

Use the Adapter (object) 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.
- you need to use several existing subclasses, but it is impractical to adapt their interface by subclassing everyone. This is possible by using an object of the parent class as adaptee.

Structure



Participants

Target	defines the domain-specific interface the client uses
Adaptee	defines an existing interface that needs adaption
Adapter	adapts the interface of `Adaptee` to the `Target` interface

Consequences

- An object adapter does not allow to override `Adaptee` behavior.
- There are additional adaptee objects introduced, which add an extra level of indirection.
- The interface of `Adaptee` is hidden from the user of the `Adapter` class.

Uses

- The C++ STL collections provide various -template based- (object) adapter classes on three levels:
  - container adapters:
    - `stack<T, Container<T> >`
    - `queue<T, Container<T> >`
    - `priority_queue<T, Container<T>, Compare<T> >`

    These are all provided by containment and message delegation so that the interface of the used (underlying) container is completely hidden from the user.
  - iterator adapters: reverse iterators
  - function adapters that change a function object:
    - `not1` (negate unary predicate)
    - `not2` (negate binary predicate)
    - `bind1st` (bind a value to the first argument)
    - `bind2nd` (bind a value to the second argument).
- CORBA (Common Object Request Broker Architecture) uses the adapter pattern to adapt the abstract CORBA servant (skeleton) class to the client interface by generating a "tie"-class which inherits from the skeleton and delegates the client interface to the implementation of the servant.

Decorator

Intent

Dynamically add responsabilities to an object.

Synonyms

Wrapper

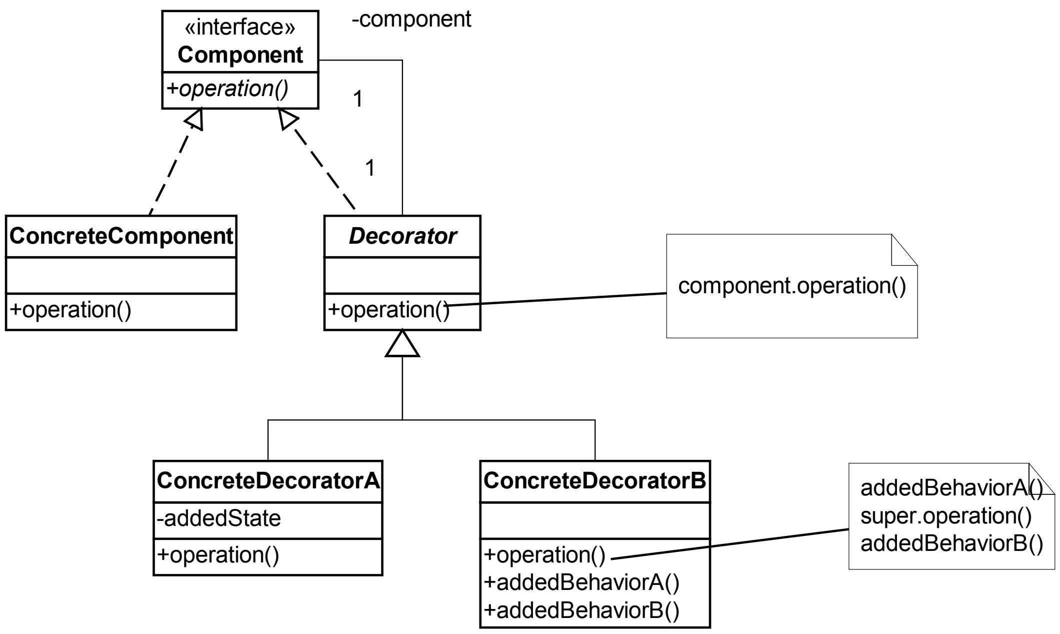
Applicability

Use the Decorator

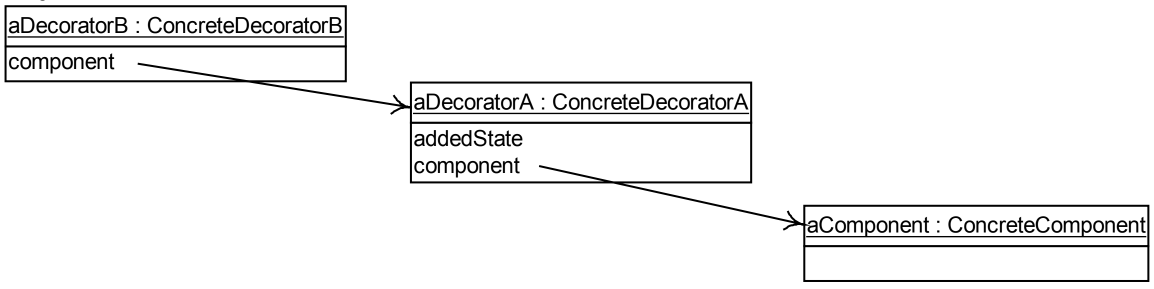
- to add responsabilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsabilities that can be withdrawn.
- when extension by subclassing is impractical (because it might result in an explosing of subclasses to support every combination).

Structure

Class diagram

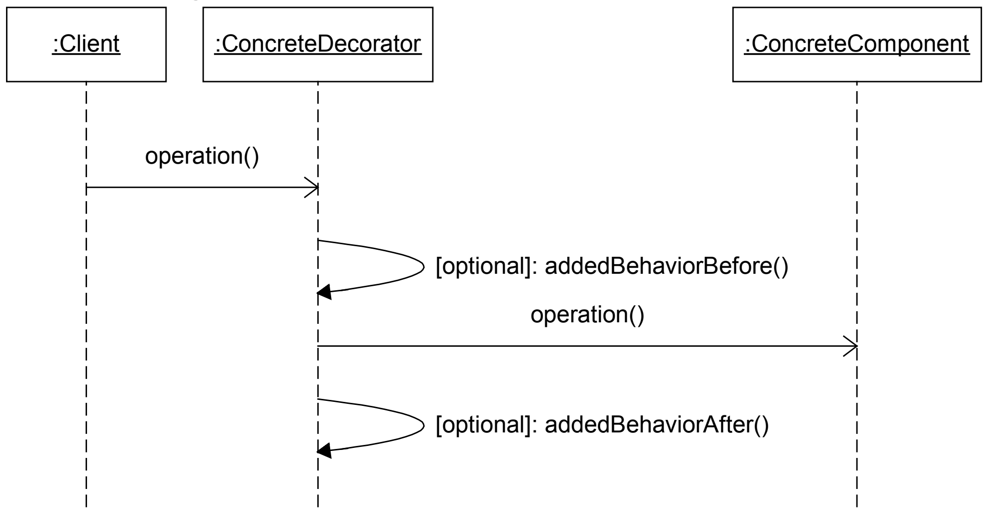


Object structure



aDecoratorB : ConcreteDecoratorB
component
aDecoratorA : ConcreteDecoratorA
addedState
component
aComponent : ConcreteComponent

Interaction diagram


:Client
:ConcreteDecorator
:ConcreteComponent
operation()
[optional]: addedBehaviorBefore()
operation()
[optional]: addedBehaviorAfter()

Participants

Component	defines the interface for objects that can have responsabilities added to them dynamically
ConcreteComponent	defines an object to which additional responsabilities can be attached
Decorator	maintains a reference to a `Component` object and defines an interface that comforms to `Component`'s interface
ConcreteDecorator	adds responsabilities to the components.

Consequences

The Decorator has at least two key benefits and two liabilities

- more flexibility than static inheritance: responsabilities can be added dynamically at run-time. The number of provided features (decorators) can be easily extended in the future.
- avoids feature-laden classes high up in the hierarchy: functionality can be incrementally added with small `Decorator` objects.
- a decorator and the component itself are not the same object, hence you should not rely on object identity. This can be a problem when components are decorated after the original components were stored in container classes.
- lots of little objects: easy to customize by those who understand them, but possibly hard to learn and debug.

Implementation issues

- Sometimes, there is no need to define an abstract `Decorator` class, e.g. when you only need to add one responsability.

Uses

- Java I/O stream classes

Component	`InputStream`
ConcreteComponent	`FileInputStream, ObjectInputStream, PipedInputStream`: these components fix the data source from where the input is taken.
Decorator	`FilterInputStream`
ConcreteDecorator	`BufferedInputStream, PushbackInputStream`: add functionality

Note that in the Java IO implementation, there are some irregularities in this decorator structure. E.g., not all concrete decorator classes (BufferedReader) are derived from the abstract decorator class (FilterReader). This could have been done for reasons of efficiency.

- Borders in the Java Swing GUI framework

Component	`Border` (an interface that is implemented by an `AbstractBorder` class, from which all other border classes are derived)
ConcreteComponent	`BevelBorder, EmptyBorder, EtchedBorder, LineBorder`
Decorator	none
ConcreteDecorator	`CompoundBorder, TitledBorder`

- In the Java Swing GUI framework, `JScrollPane` is a component decorator that simply add scrollbars.
- The Java Collection framework provides a number of unsynchronized, modifiable container classes. However, there are times when a collection must be thread-safe (when working in multithreaded environments) or when the objects in the collection must be immutable once they are added to the container. Instead of creating additional classes for each possible combination, the `Collections` utility class provides two methods to decorate the standard collection classes with synchronization and immutability:
  - `X Collections.synchronizedX (X x)` which adds synchronization to a collection class
  - `X Collections.unmodifiableX (X x)` which makes the elements in the collection immutable.

  The decorator classes itself are not explicit to the user.

Singleton

Intent

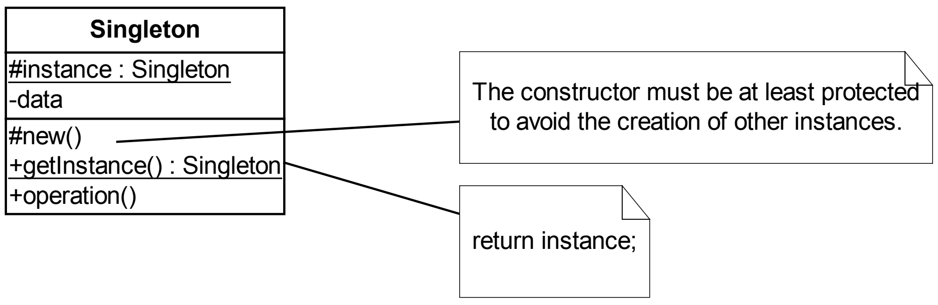
Ensures a class has only one instance, and provides a global access point to it.

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.

Structure


Singleton
#instance : Singleton
-data
#new()
+getInstance() : Singleton
+operation()
The constructor must be at least protected to avoid the creation of other instances.
return instance;

Participants

Singleton	• defines a `getInstance()` operation that lets clients access its unique instance. • may be responsible for creating its own instance.

Consequences

Singleton has the following benefits

- controlled access to the sole instance.
- permits refinement of operations and representation by subclassing
- permits a variable number of instances. The pattern makes it easy to change your mind and allow more than one instance of the Singleton class.
- more flexible than a utility class (a class with attributes and operations declared as static)
  - methods that are static can not be used to implement an interface.
  - all references to the class must be hard coded with the name of the class. There is no way to overwrite the operations by subclassing.

Implementation issues

- The singleton object can also be constructed only when it is needed (lazy initialization). In Java:

  ```
  public class Singleton {
    static protected Singleton instance = null;
    protected Singleton() { }
    static public Singleton getInstance() {
      if(instance == null)
        instance = new Singleton();
      return instance;
    }
    // ...
  }
  ```

  This method also allows to pass arguments for the Singleton constructor.
- In a multi-threaded environment such as Java, one must make sure that it is completely impossible to create two singleton instances. Therefore, the `getInstance()` operation must be synchronized.
- Within the given implementation, you can subclass the `Singleton` class and change the creation of the `Singleton` instance to the new derived class, or you can provide the type of the `Singleton` object to create as a parameter to the `getInstance()` operation. More flexibility can be obtained when you separate the creation process from the `Singleton` class itself.

  ```
  public interface SingletonFactory {
    public Singleton createInstance();
  }

  public final class SingletonWrapper {
    static private SingletonFactory factory = null;
    static private Singleton instance = null;
    // default factory method when no user defined factory is set
    static private Singleton createInstance() {
      return new Singleton();
    }
    static synchronized public Singleton getInstance() {
      if(instance == null)
        instance = (factory == null) ?
  ```

```
                createInstance() : factory.createInstance();
    return instance;
  }
  static synchronized public void setFactory(SingletonFactory f) {
    factory = f;
  }
  // set this to null to force the creation of a new instance
  static synchronized public void setSingleton(Singleton i) {
    instance = i;
  }
}
```

This implementation (based on the Factory design pattern) allows to dynamically change the singleton instance that is published. It is even possible to use pre-existing classes, not designed with the singleton functionality in mind, as singletons. However, it prohibits to make the `Singleton` constructor protected.

- When implementing the Singleton pattern in C++ an additional "headbreaker" is correct memory management, as C++ has no garbage collector contrary to Java. If you use a static pointer to refer to the single Singleton instance, C++ will only free the pointer variable automatically, but not the object pointed to. And not using a pointer is not a good solution, because we then loose the advantage of (possibly) using a polymorph singleton instance. A better solution is using a separate (template) class `SingletonDestroyer`. As the `cDestroyer` variable is static in the following code, its destructor will automatically be called by the compiler before exiting the application.

```
SingletonDestroyer<Singleton> Singleton::cDestroyer;

class Singleton {
  public:
    /** The destroyer should have access to the protected destructor,
     * hence it must be a friend class. */
    friend class SingletonDestroyer<Singleton>;
    static Singleton& getInstance() {
      if(!cSingleton) {
        cSingleton = new Singleton();
        cDestroyer.setSingleton(cSingleton);
      }
      return *cSingleton;
    }
  protected:
    static Singleton* cSingleton;

    /** The automatic destruction of this static object (hence it's
     * automatic destruction) takes care of destructing the singleton
     * object. */
    static SingletonDestroyer<Singleton> cDestroyer;

    /** Empty destructor to make it virtual. The destructor of a
     * singleton class should not be public. Because the singleton
     * OWNS the instance, it is also responsible for deleting it, not
     * a client. It shouldn't also be private, because we want to be
     * able to subclass the singleton. Hence, the destructor
     * must be protected. */
    virtual ~Singleton() { }
    Singleton() { }
};

template <class T> class SingletonDestroyer {
  public:
    /** Constructor, optionally providing the singleton instance.
     * @param pSingleton pointer to the singleton instance. */
    SingletonDestroyer(T* pSingleton = 0) : iSingleton(pSingleton) {}

    /** Destructor that cleans up the singleton instance.
     * @warning When the singleton instance was not set this may
     * result in a NULL-pointer run-time exception. */
    ~SingletonDestroyer() { delete iSingleton; }

    /** Set the singleon instance.
     * @param pSingleton pointer to the singleton instance. */
    void setSingleton(T* pSingleton) { iSingleton = pSingleton; }

  private:
    /// Pointer to the singleton instance.
    T* iSingleton;

    /** Prevent users from making copies of a SingletonDestroyer to
     * avoid double deletion. */
    SingletonDestroyer(const SingletonDestroyer<T>&);
    void operator=(const SingletonDestroyer<T>&);
};
```

Uses

- The relationship between metaclasses (whose instances are classes) and classes is a singleton in Java (and also in Smalltalk). This becomes obvious as the `Class` class in Java has no constructor, only a `forName(String)` method to return the metaclass of the class name given as argument. Hence, `Class` is implemented as a singleton.
- Singleton patterns are often used for managing connections to resource, e.g., a pool of database connections or a `PrinterSpooler` class that manages all installed printers.
- Configuration parameters of an application (e.g. kept in a Java `Property` object) are also ideal candidates to apply the Singleton pattern.

Chain of responsibility

Intent

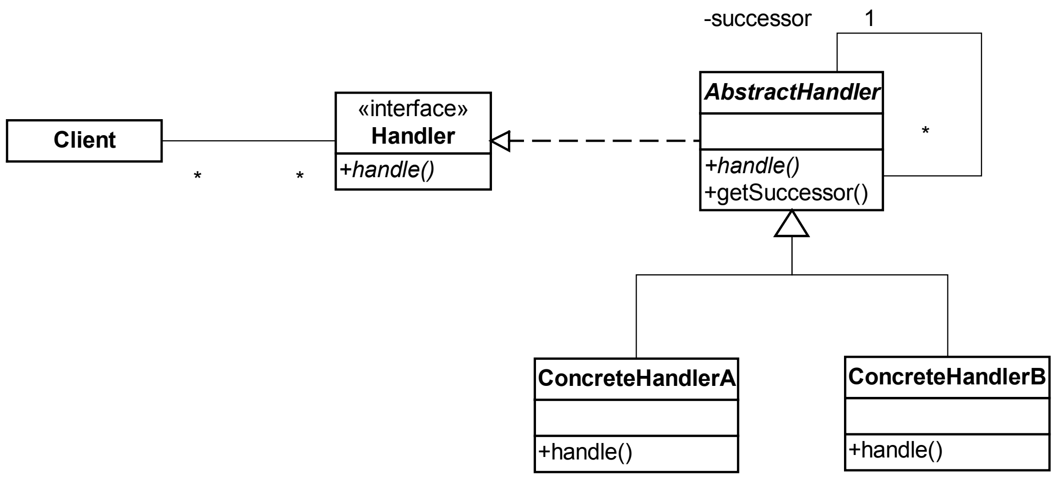
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request (cfr. the daisy chain algorithm for busarbitrage that is used to give permission on the bus to one of the I/O devices).

Applicability

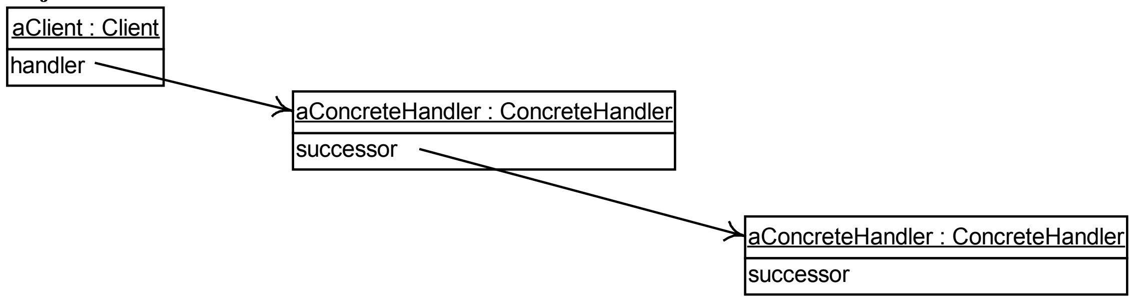
Use the chain of responsibility pattern when
- more than one object may handle a request and the handler is not known a priori.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

Structure

Class diagram



Object structure



aClient : Client
handler
aConcreteHandler : ConcreteHandler
successor
aConcreteHandler : ConcreteHandler
successor

Participants

Handler	defines an interface for handling the requests
AbstractHandler	implements the successor link (optionally)
ConcreteHandler	• handles requests it is responsible for • if it can't handle a request, it forwards the request to its successor.
Client	initiates the request to a ConcreteHandler object on the chain

Consequences

Chain of responsibilities has the following benefits and liabilities

- reduced coupling: the pattern frees an object from knowing which other object handles a request.
- added flexibility in assigning responsibilities to objects, You can add or change responsibilities for handling a request by modifying the chain at run-time.
- receipt isn't garuanteed

Implementation issues

- Instead of defining new links and providing the successor functionality in an abstract class, sometimes, existing object references can be used to form the chain (e.g., parent references in a part-whole hierarchy can define a part's successor).
- To represent requests, different options are available
  - a hard-coded operation invocation (only a fixed number of requests that the handler defines can be forwarded)
  - the `Handler` interface can take a request code as parameter (an integer of a string), which is less safe than invoking an operation directly.

- A `Request` class can represent requests explicitly, and new requests can be added by subclassing (that can define different parameters). In this case, the receiver can rely on an identifier from the `Request` class or on run-time type information to identify the request.

Uses

- Plug-ins in browsers can be implemented using this pattern. Each object on the chain could handle a specific type of file. When the browser loads a file that is not recognized internally (e.g., TIFF, AVI, PDF, CLASS,...), the browser passes it to the first installed plug-in. When it cannot be handled there, the request is forwarded to the next plug-in. Ultimately, some plug-in handles the request or the browser displays an error message.
- A Java application that should be dynamically extensible, could use the reflection capabilities of Java (to dynamically load selected classes and instantiatie objects) in combination with this pattern.

Observer

Intent

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

Synonyms

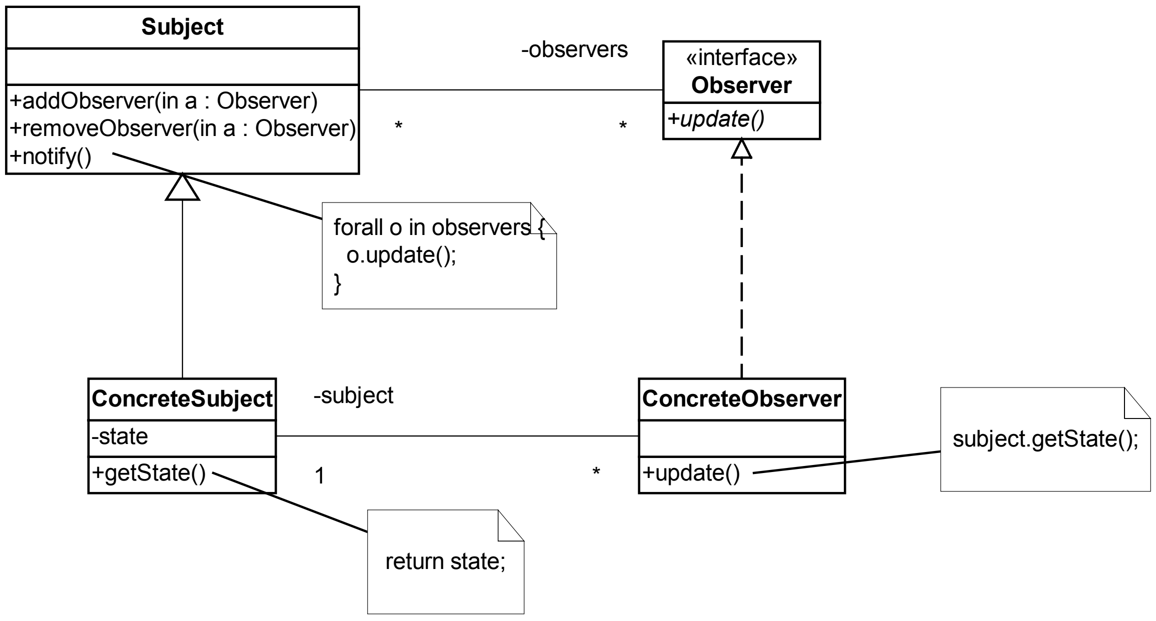
Dependants, Publish-subscribe

Applicability

Use the Observer pattern when

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

Structure



Participants

Subject	knows its observers. Any number of `Observer` objects may observe an `Subject` object (also called observable)
Observer	defines an updating interface for objects that should be notified of changes in a subject.
ConcreteSubject	• stores state of interest to `ConcreteObserver` objects. • sends an notification to its observers when its state changes.
ConcreteObserver	• maintains a reference to a `ConcreteSubject` object. • stores state that should stay consistent with the object's. • implements the `Observer` updating interface to keep its state consistent with the subject's.

Consequences

The Observer pattern lets you vary subjects (observables) 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.

- Abstract coupling between `Subject` and `Observer`.
- Support for broadcast communication.
- Unexpected updates. When the dependency criteria aren't well-defined or maintained, a slight change to the subject's state might lead to a cascade of updates to observers and there dependants (which can be hard to track).

Implementation issues

- It might make sense in some situations for an observer to depend on more than one subject. In this case, it is necessary to extend the `update()` interface to let the observer know which subject is sending the notification. The subject can simply pass itself as a parameter in the update operation.
- Who should trigger the update through a call to the notify operation?
  - When state-setting operations on `Subject` call notify after they changed the state, clients don't have to remember to call notify explicitly. However, several consecutive operations will cause several consecutive updates, which may be inefficient.
  - When clients are responsible for calling notify at the right time, they can wait to trigger the update after a series of state changes has been made. However, clients have an added responsability which is error-prone.
- Implementations of the Observer pattern often have the subject broadcast additional information about the change as a parameter to update.
  - push model: the subject sends observers detailed information about the change and the state of the subject.
  - pull model: the subject sends nothing but the most minimal notification and the observers ask for details explicitly.
- The subject must guard the integrity of its observers list. Especially in a multi-threaded environment, adding, removing and notification of observers must be made thread-safe operations. The common strategy (in a language like Java) is to make the `addObserver` and `removeObserver` operations synchronized. The notification operation first creates a (shallow) copy of the observers list in a synchronized block and then notifies each observer in that copied list. When another thread adds or removes observers while the notification operation is being executed, the changes will only take effect when the next update is triggered.

Uses

- The MVC (model/view/controller) paradigm relies fully on the observer pattern to communicate between the model and the views.
- Java provides a class `Observable` (subject) implementing the registration and notification of observers and an interface `Observer` with the `update(Observable)` operation.
- Java event delegation model (e.g. the `ActionListener`)

Observer	`ActionListener` All listener interfaces extend the base `EventListener` interface which contains no operations.
Subject	A GUI component, e.g. a button object, that can be activated by the user. When the event is triggered, the subject passes information to the observers as an `ActionEvent` object (derived from the general `EventObject` base class).
ConcreteObserver	Implementation of this listener interface as provided by the application programmer, e.g. the "view" of the application.
ConcreteSubject	The "model" of the application that is responsible to update its views. In Java, every `EventObject` object has an operation `getSource()` that the concrete observer can use to know the concrete subject.

The event delegation model also uses the Command design pattern.

Command

Intent

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

Synonyms

Action, Transaction

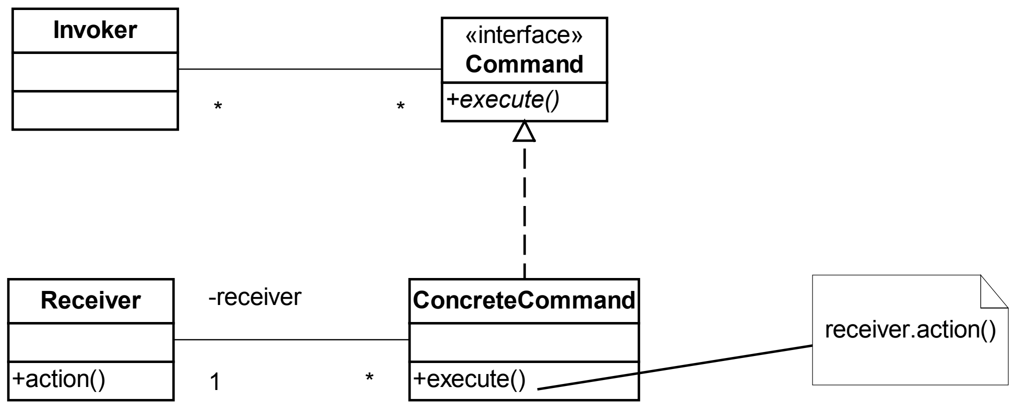
Applicability

Use the Command pattern when you want to

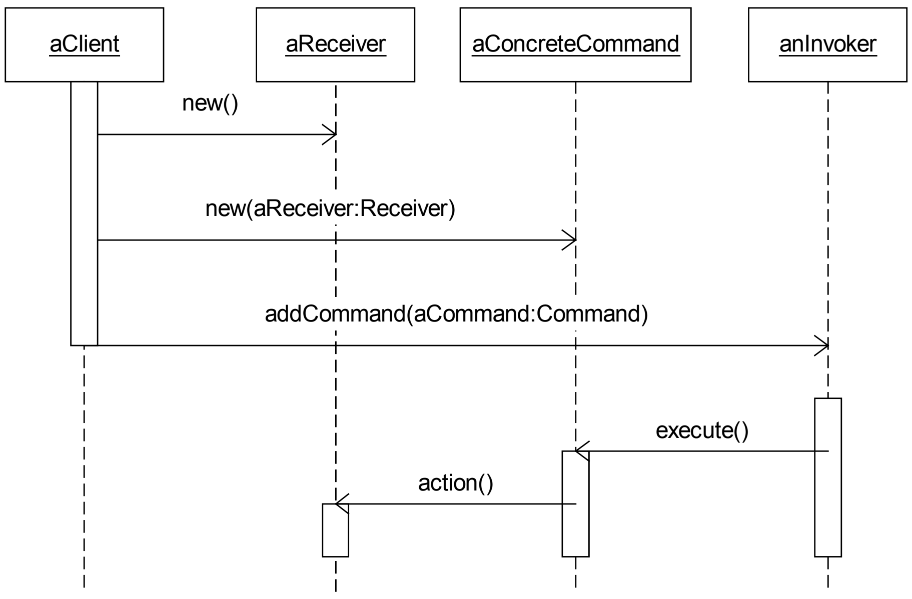
- parametrize objects by an action to perform. Commands are an object-oriented replacement for callbacks in a procedural language (most often implemented with pointers to functions).
- specify, queue and execute requests at different times.
- support undo.
- support logging changes so that they can be reapplied in case of a system crash.
- structure a system around high-level operations built on primitive operations. Such a structure is called a transaction and encapsulates a set of changes to data.

Structure

Class diagram



Sequence diagram



Participants

Command	declares an interface for executing an operation
ConcreteCommand	• defines a binding between a `Receiver` object and an action • implements `execute()` by invoking the corresponding operation(s) on the `Receiver`
Invoker	asks the command to carry out the request
Receiver	knows how to perform the operation associated with carrying out a request. Any class may serve as Receiver.

Note: The registration of the `Command` object to the invoker is an application of the Observer design pattern.

Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.

- It is easy to add new commands, because you don't have to change existing classes.

Uses

- Java event delegation model (e.g. the `ActionListener`)

Command	`ActionListener`
ConcreteCommand	Implementation of this listener interface as provided by the application programmer. Most often, this class is implemented as an inner class, e.g. in the "controller" of the application.
Invoker	A GUI component, e.g. a button object, that was activated by the user
Receiver	The "model" of the application that knows how to react to the event (e.g. a button click)

  Java also provides a more direct `Action` command class
- Undo and redo support is a typical application of the Command pattern. Each transaction (set of changes to data) is kept in a history list with additional information how to invert the transaction. E.g., when text is inserted in a text editor, a new command object is stored in the history list with additional information how to remove the inserted text. This way, any number of transactions can be undone and redone at any time. The Java library contains support classes to provide these features.
- The C++ STL makes frequent use of function objects (functors), which are a simplified version of commands. The simplification is due the fact that the concrete commands don't have any receiver. However, maintaining the relationship with a receiver is just one of the biggest strenghts of the command pattern.
- The Java Collection framework also extensively uses such function objects, for instance for the `Comparator` interface.

Bridge

Intent

Decouple an abstraction from its implementation.

Synonyms

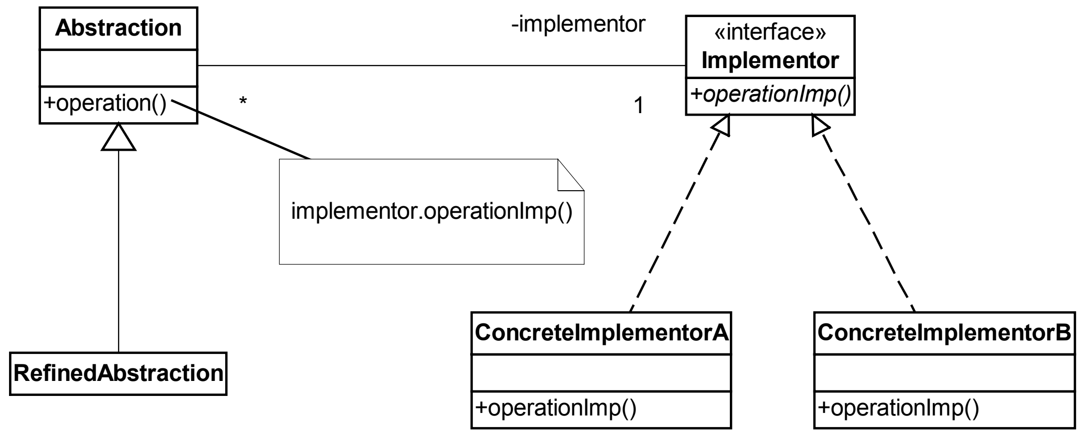
Handle/Body

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.
- you want to share an implementation among multiple objects, and this fact should be hidden from the client.

Structure



Participants

Abstraction	• defines the abstraction interface. • maintains a reference to an object of type `Implementor`.
RefinedAbstraction	extends the interface defined by `Abstraction`.
Implementor	defines the interface for implementation classes. Typically, this interface only defines primitive operations, while `Abstraction` defines higher-level operations based on these primitives.
ConcreteImplementor	implements the `Implementor` interface.

Consequences

- decoupling interface and implementation. An abstraction can be configured with an implementation at run-time. It's even possible for an abstraction to change its implementation at run-time.
- improved extensibility. You can extend the `Abstraction` and the `Implementor` hierarchies independently.
- hiding implementation details from clients. You can shield clients from implementation details, like the sharing of implementor objects and the accompagnying reference counting (if any).

Uses

- All components in the Java AWT (Abstract Windowing Toolkit) GUI library are "heavyweight" components, which means that they are registered as native "peer" components in the native windowing system. The AWT components and peers are bridges.

Abstraction	`Component`
RefinedAbstraction	`Button, List,...`
Implementor	`ComponentPeer`
ConcreteImplementor	In the AWT, each RefinedAbstraction has its own corresponding interface, so there is an interface `ButtonPeer` extending `ComponentPeer`. This interface is implemented for each specific operating system, `WinButtonPeer`, `MacButtonPeer`,...

- The Java Swing GUI library provides a “pluggable look-and-feel (PLAF)” which is based on the Bridge pattern. Each look-and-feel class realizes the same interface that is used within the GUI -lightweight- component hierarchy to draw on the screen.

Abstraction	`JComponent`
RefinedAbstraction	`JButton, JList,...`
Implementor	`ComponentUI`
ConcreteImplementor	Also in Swing, each RefinedAbstraction has its own corresponding interface, so there is an interface `ButtonUI` extending `ComponentUI`. This interface is implemented for each specific operating system, `WinButtonUI`, `MetalButtonUI`,...

- The Java JDBC (Java DataBase Connection) library also uses the bridge pattern to achieve independence from a specific database engine.

Factory method

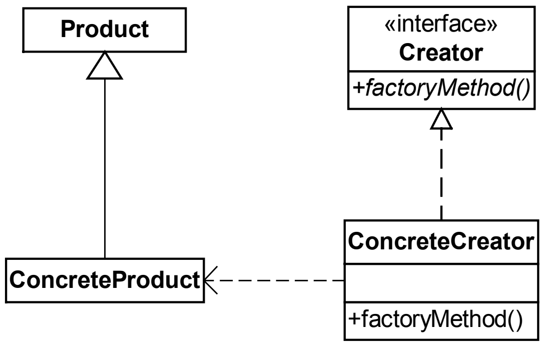
Intent

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.

Structure



Participants

Product	Defines the interface objects the factory method creates.
ConcreteProduct	Implements the Product interface.
Creator	Declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
ConcreteCreator	Overrides the factory method to return an instance of a ConcreteProduct.

Consequences

- It provides hooks for subclassing.
- It connects parallel class hierarchies.

Uses

This pattern is typically used in combination with other patterns where the uses are discussed. See the Abstract Factory pattern.
Also the iterator design pattern applies the factory method pattern to create its concrete iterator.

Abstract factory

Intent

Provide an interface for creating families of related objects.

Synonyms

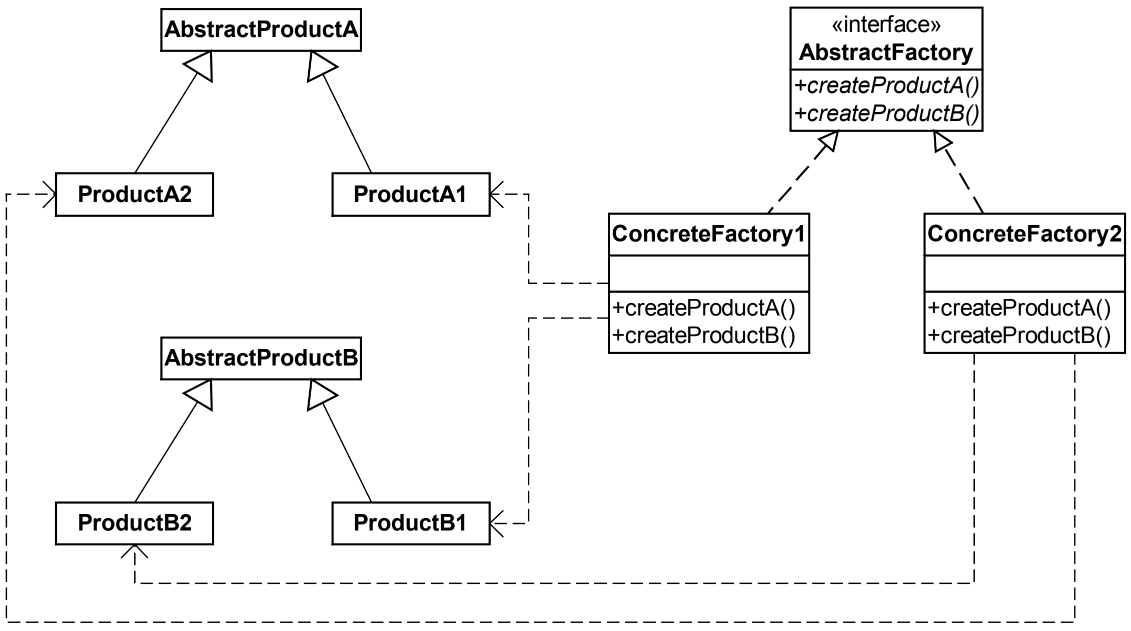
Kit

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.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure



Participants

AbstractFactory	Declares an interface for operations that create abstract product objects.
ConcreteFactory	Implements the operations to create concrete product objects.
AbstractProduct	Declares an interface for a type of product object.
Product	• Defines a product object to be created by the corresponding concrete factory. • Implements the AbstractProduct interface.

Consequences

The Abstract Factory pattern has the following benefits and liabilities:

- It isolates concrete classes. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes.
- It makes exchanging product families easy.
- It promotes consistency among products.
- Supporting new kinds of products is difficult because it requires extending the factory interface, which involves changing the AbstractFactory class and all its subclasses.

Implementation issues

- The concrete factory is often implemented as a singleton.
- The factory only declares an interface for creating products. Usually, it's up to concrete subclasses to actually create them by using the factory method design pattern for each product.

Uses

- The Java AWT uses the abstract factory pattern to create its heavyweight peer components.

AbstractFactory	The abstract class `Toolkit` provides the methods `createButton`, `createList`,...
ConcreteFactory	`WinToolkit, MacToolkit,...`
AbstractProduct	`Button, List,...`
Product	• `WinButtonPeer, WinListPeer,...` • `MacButtonPeer, MacListPeer,...`

- The Java Swing library also uses this pattern to support its PLAF (Pluggable Look And Feel), and to ensure that all components are instantiated from the same PLAF family.

Proxy

Intent

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

Synonyms

Surrogate (Ambassador)

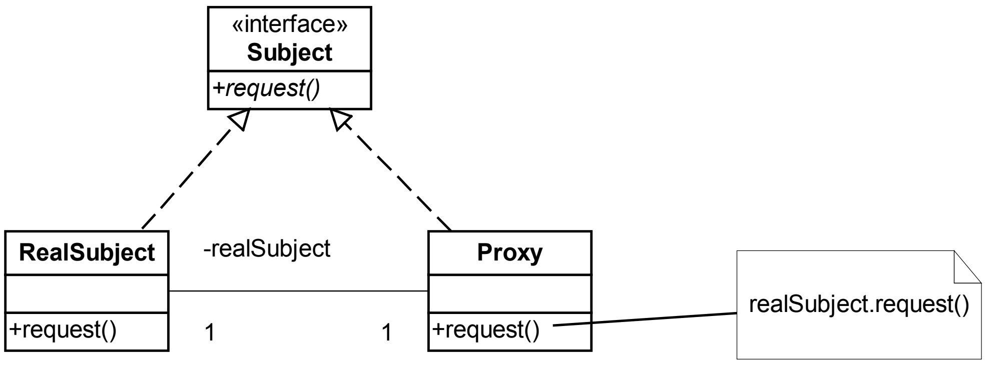
Applicability

Use the Proxy pattern when

- a **remote proxy** provides a local representative for an object in a different address space. Such a kind of proxy is also called an "ambassador".
- a **virtual proxy** creates expensive objects on demand.
- a **protection proxy** controls access to the original object.

- a **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed (e.g. reference counting).

Structure



Participants

Proxy	• maintains a reference that lets the proxy access the real subject • provides an interface identical to the `Subject`'s so that a proxy can be substituted for the real object • controls access to the real subject and may be responsible for creating and deleting it. • other responsibilities depend on the kind of proxy: o 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. o virtual proxies may cache additional information about the real subject so that they can postpone accessing it. o protection proxies check that the caller has 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	defines the real subject that the proxy represents

Consequences

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

- A remote proxy can hide the fact that an object resides in a different address space.
- A virtual proxy can perform optimizations such as creating an object on demand.
- A protection proxy allows additional housekeeping tasks when an object is accessed.

Implementation issues

- It is often convenient to overload the member access operator (`->` and `*`) when implementing the proxy pattern in C++. Overloading this operator in a proxy class (e.g., `ImagePtr`) lets you perform additional work whenever an object is dereferenced.
- Java provides support to facilitate easy implementation of proxies (since 1.3) through the classes Proxy, Method and InvocationHandler in the java.reflect package.

Uses

CORBA as well as Java RMI (Remote Method Invocation) provide a stub object for each remote object and is thus an instance of a remote proxy object. This stub object takes care of the "marshalling" (passing) of parameters and triggers the remote method implementation.

Other patterns used in Java

Strategy

Java's Abstract Window Toolkit (AWT) provides implementations of common user interface components such as buttons, menus, scrollbars, and lists. Those components are laid out, sized and positioned, inside containers such as panels, dialog boxes, and windows. But AWT containers do not perform the actual layout. Instead, those containers delegate layout functionality to another object known as a layout manager. That delegation is an example of the Strategy design pattern. For the AWT, the clients are containers and the family of algorithms are layout algorithms encapsulated in layout managers. If a particular layout algorithm other than the default algorithm is required for a specific container, an appropriate layout manager is instantiated and plugged into that container. In this way, layout algorithms can vary independently from the containers that use them.

Composite

To facilitate complex user interface screens, user interface toolkits must allow nested containers, effectively composing components and containers into a tree structure. Additionally, it's crucial for components and containers in that tree structure to be treated uniformly, without having to distinguish between them. For example, when the AWT determines the preferred size of a complex layout containing nested components and containers, it walks the tree structure and asks each component and container for its preferred size. If that traversal of the tree structure required distinction between components and containers, it would unnecessarily complicate that code, making it harder to understand, modify, extend, and maintain.
The AWT accomplishes nesting containers and uniform treatment of components and containers by implementing the Composite pattern. The Composite pattern dictates that containers are components, typically with an abstract class that represents both. In the AWT, that abstract class is `java.awt.Component`, the superclass of `java.awt.Container`. Therefore, an AWT container can be passed to the `add(Component)` method from `java.awt.Container` because containers are components.

Mediator

The intent of the Mediator design pattern is to define an object that encapsulates how a set of objects interact. The Mediator design pattern promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. The Java class `java.awt.MediaTracker` which manages the loading of graphical, audio, or other data, adheres to the Mediator design pattern controlling various sub-elements and making sure they function together.