Design Patterns

Design patterns were developed by the architect Christopher Alexander in the 1970's to describe elements of architectural design.[1]

His ideas were picked up by the object-oriented programming community in the mid 1990's and applied to software design problems.[2]

You can think of design patterns as a language about solving problems in system design without committing to any particular programming language:

> *First study the possible system architectures, then decide on a suitable implementation language.*

This is important; you want the architecture and problem structure to drive your implementation NOT the programming language.

Patterns fall into categories including:

- Creational
- Behavioral
- Structural
- *Etc.*

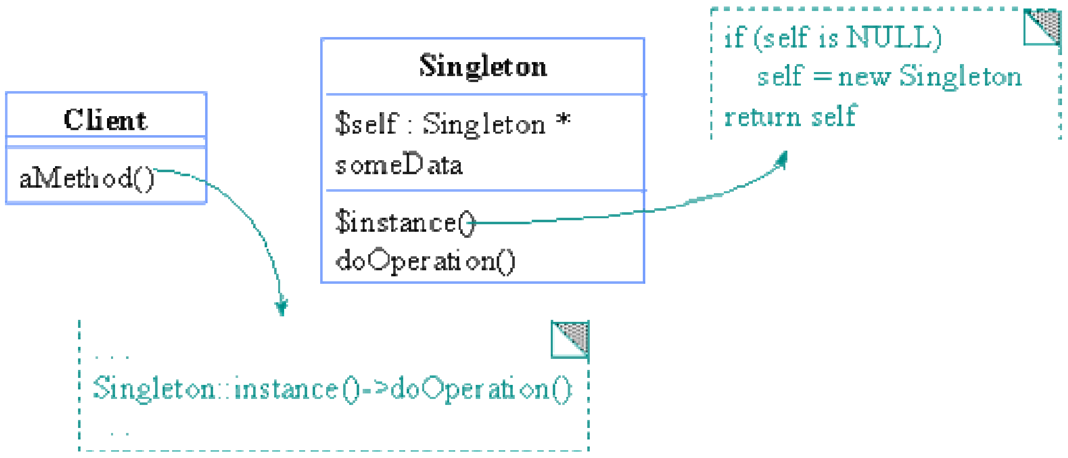
Some examples of patterns follow. [3]

[1] *A Pattern Language*, C. Alexander, Oxford University Press, 1977.
[2] *Design Patterns*, Erich Gamma *et al*, Addison-Wesley, 1995.
[3] Based on patterns published at http://www.patterndigest.com

Name: Singleton **Category**: Creational

Intent

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



Solution:

1. Declare the constructor(s) as Protected;
2. Declare a static "self" pointer to hold a reference to the single instance when it is created;
3. Declare a static "instance" function which creates and returns the single instance if the self pointer is NULL, otherwise it just returns the previously created instance.
4. Declare a protected destructor that sets the "self" pointer to NULL
5. Declare other methods and attributes as needed normally.
6. Clients access the singleton by calling the static instance function to get a reference to the single instance and then using it to call other methods.
7. Clients must not store this instance reference -- Each code segment that needs to access the singleton should call the instance function, perform the work needed and then discard the reference as quickly as possible.

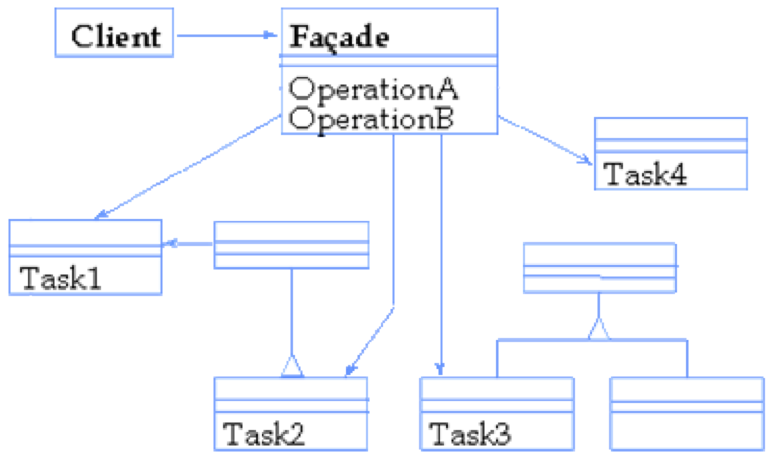
Consequences

- Insures that only one instance of the class can ever be created.
- Access to the sole instance is strictly controlled by the object itself.
- Singleton classes may be extended by subclassing. Code written to use the base singleton class will automatically use an instance of the subclass provided that the instance function of the subclass is called by the client code prior to any call to the base singleton class' instance function.
- Reduces the number of global names defined in a system. Singletons avoid polluting the name space with global variables.
- More flexible than using class (static) methods for functionality, since static functions (in C++) cannot be polymorphically overridden by subclasses.

Name: Facade **Category:** Structural

Intent

Encapsulate a subsystem using a high-level interface, simplifying subsystem usage and hiding structural details.



Solution

1. Clients communicate with subsystem objects by calling methods in Façade
2. Clients never (or as seldom as possible) directly accessing objects in subsystem -- any such access weakens the encapsulation.
3. Subsystem objects usually retain no knowledge of client
4. Subsystem objects do not normally maintain any reference to Façade

Consequences

- Eliminates hard to control tangled networks of object associations
- Reduces number of objects clients need to interface with.
- Promotes weak coupling, which enhances overall flexibility.

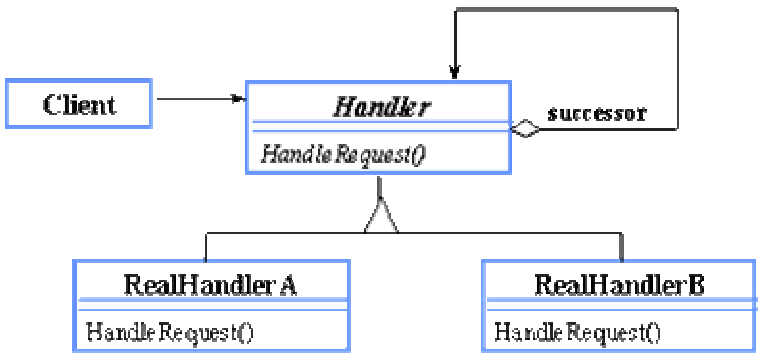
Related Patterns

- Façades are often implemented as Singletons
- Mediator is somewhat similar to Façade

Name: Chain of Responsibility **Category*:*** Behavioral

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.



Solution

Chain the receiver objects in a linked list and pass the request along the chain until an object handles it.

Applicability

- More than one object may handle a request and the specific handler needs to be ascertained automatically
- A request needs to be issued to a set objects that changes at run-time
- You want to avoid any specific knowledge by the client of the available request handlers in the system

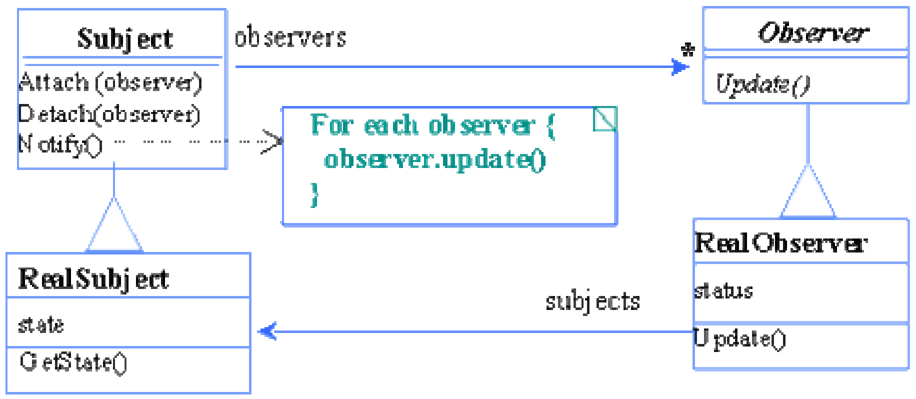
Consequences

- Reduces coupling between a client and its responsibilities. New or alternate responsibilities can be easily added at runtime
- Clients have no explicit knowledge of specific Receivers since it knows only an abstraction (Dependancy Inversion Principle)
- The abstract Receiver class is usually small, stable and not highly susceptible to change, thus it is "closed" (Open/Closed Principle)
- There is no guarantee that any particular request may be actually handled, since it's possible no handler in the chain can process it.
- Clients are responsible to pass the request along the chain, calling each object in turn. If many clients exist, this will lead to code duplication and violates the Law of Demeter.

Name: Observer **Category***:* Behavioral , Architectural

Intent

Define relationship between objects such that whenever one object is updated all others are notified automatically.



Applicability

- Changes to one object in a group requires changes to the others in the group
- The number of objects in the group may vary
- Loose coupling is desired between the objects in the group

Consequences

- Subjects have limited knowledge of the objects in the observed group
- Objects may easily be added or removed without changing existing objects
- Update notifications are automatically broadcast to all interested objects
- The cost of updating any object in the group may greater than expected
- Update notifications may occur more frequently than desired or expected

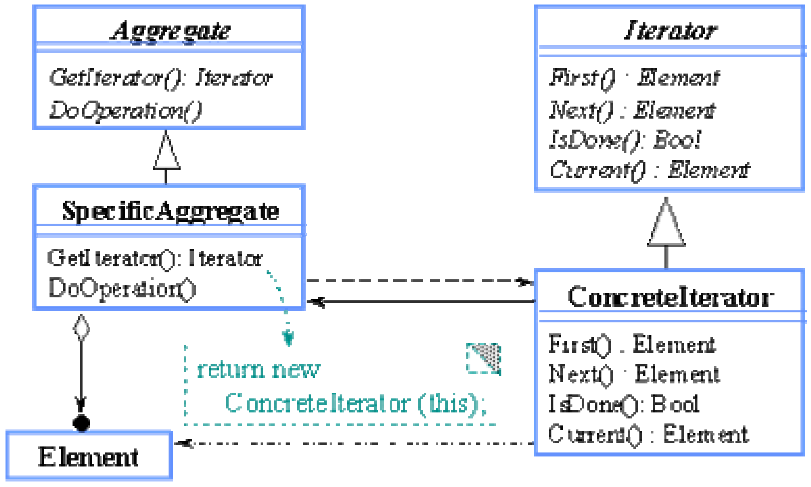
Related Patterns

The ChangeManager in Mediator can be used to control or restrict update notifications

Name: Iterator **Category***:* Behavioral , Downcasting

Intent:

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



Applicability:

- You have some object classes which encapsulate aggregations of other objects and you wish to hide details of the structures to minimize the effects of future changes.
- You need to perform various operations which must traverse the elements of the aggregate structure in various ways.
- You want to avoid "polluting" the interface of the aggregate objects with those operations to simplify future changes or additions.
- You wish to provide a common polymorphic interface for iteratively traversing different aggregate structures to perform those operations.

Solution:

1. Create an Iterator class interface which defines a basic, uniform set of functions for traversing and accessing elements. Create an Aggregate class interface for maintainance of an aggregate structure and creation of Iterator(s) for traversal.
2. Define the SpecificAggregate classes to implement an Aggregate structure of specific Elements.
3. For each method of traversal needed, create a ConcreteIterator which implements the Iterator interface to traverse and access elements encapsulated by a SpecificAggregate class.
4. Create the Element class for objects which will be encapsulated by Aggregate and returned by the traversal functions implemented by the ConcreteIterator(s).
5. **Note**: The Iterator and Aggregate class interfaces are typically defined in C++ using class templates, for example, see the C++ STL library.

Consequences:

- Downcasting may not be required to access elements of the aggregate structure (unless the elements are formed from an inheritance hierarchy).

- A variety of structure traversal methods can be easily supported.
- Usage of iterators simplifies the interface of Aggregate objects.
- Multiple traversals can be performed on a single structure at the same time since the iterators must store the current position.
- Application may require changes to existing code, which may be unacceptable if the base classes resides in a reuse libarary.
- Adding a new iterator requires adding a new GetIterator function to every SpecificAggregate (many solutions separate construction of ConcreteIterators from the Aggregate class to avoid this problem.)
- If the SpecificAggregate encapsulates elements related through an inheritance hierarchy, the downcasting may be needed to access specific members of the element classes.