Object-Oriented Programming

Iuliana Bocicor

iuliana@cs.ubbcluj.ro

Babes-Bolyai University

2020

Overview

Design Patterns I

- When designing something new (a building, a novel, a computer program), designers make certain decisions.
- Experienced designers (architects, writers, software architects) know **not** to solve a problem from first principles, but to reuse good solutions that have worked in the past.
- Patterns are like templates that can be applied in many different situations.
- Software design patterns are recurring descriptions of classes and communicating objects that are customized to solve a general design problem in a particular context.

Design Patterns II

- They are general, flexible, reusable solutions to commonly occurring problems within a given context in software design.
- Object-oriented design patterns show relationships and interactions between classes or objects.
- Christopher Alexander: *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"*.

Design Patterns III

- One of the most influential books for software engineering and object-oriented design theory and practice:

  Design Patterns: Elements of Reusable Object-Oriented Software, by *Erich Gamma*, *Richard Helm*, *Ralph Johnson* and *John Vlissides*.

- They are often referred to as the **Gang of Four (GoF)**.
- This book introduces the principles of design patterns and then offers a catalogue of such patterns (23 classic design patterns).

Essential elements of a pattern I

- **Pattern name**
  - a word or two which describe the desing problem, its solution and consequences;
  - it is a part of the software developer vocabulary;
  - "Finding good names has been one of the hardest parts of developing our catalog." (GOF)
- **Problem**
  - describes when to apply the pattern;
  - explains the problem and its context;
  - it might include a list of conditions that must be met before it makes sense to apply the pattern.

Essential elements of a pattern II

- **Solution**
  - describes the elements that make up the design, their relationships, responsibilities and collaborations;
  - provides an abstract description of a design problem and how general arrangement of elements (classes and objects) solves it.
- **Consequences**
  - describe the results and trade-offs (space and time trade-offs) of applying the pattern;
  - may address language and implementation issues as well;
  - they include the pattern's impact on a system's flexibility, extensibility, or portability.

Patterns' purposes I

- The purpose of a pattern reflects what a pattern does.
- **Creational patterns**
  - concern the process of object creation;
  - E.g.: Abstract Factory, Builder, Factory Method, Prototype, Singleton.
- **Structural patterns**
  - are concerned with how classes are composed to form larger structures;
  - E.g.: Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy.

Patterns' purposes II

- **Behavioural patterns**
  - are concerned with algorithms and the assignment of responsibilities between objects;
  - E.g.: Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

- Some patterns are often used together (e.g. Composite is often used with Iterator).

- Some patterns are alternatives (e.g. Prototype is often an alternative to Abstract Factory).

Observer I

- Defines and maintains a dependency between objects.
- Classic example: in the MVC approach - all views of a model are notified when the model changes.

Intent

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

Observer II

Also Known As

- Dependents, Publish-Subscribe.

Motivation

- A common consequence of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- The goal is to maintain consistency, but at the same time to avoid tightly coupled objects (coupling reduces reusability).

Example

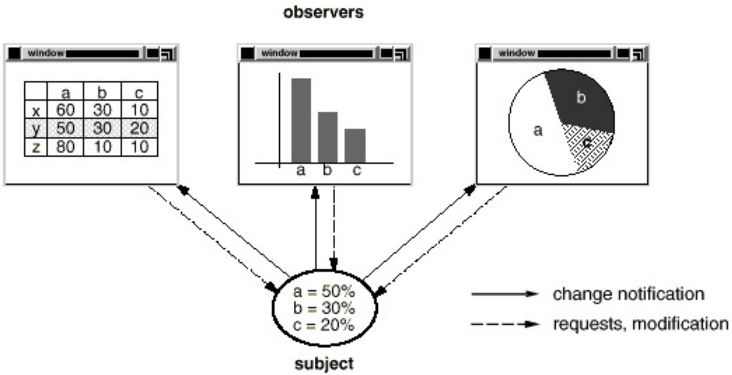


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

How it works

- Key objects: **subject** and **observer**.
- A subject may have many observers.
- All observers are notified when the subject is changed.
- Each observer will query the subject to synchronise its state with the subject's state.

Pattern class structure

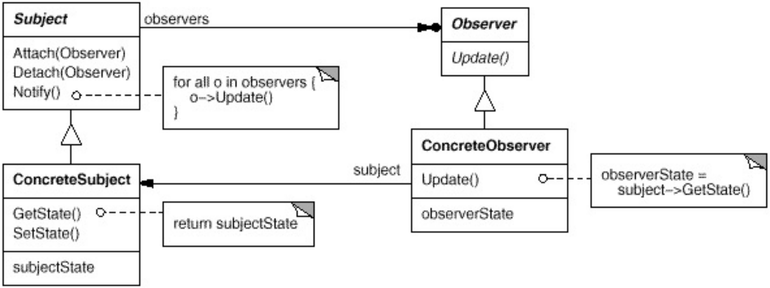


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

Applicability and consequences

Applicability

- When a change to an object requires changes in other objects, and you don't know how many need to be changed.
- When an object should be able to notify other objects without knowing who these objects are.

Consequences

- The subject and the observer are *loosely coupled* (the subject knows that it has a list of observers, but does not know their concrete classes).
- Support for broadcast communication; observers can be added or removed at any time.

Example - Auction I

- We have a castle auction.
- The auctioneer starts the bidding for a specific castle. He is the subject, all the bidders "observe" him, to see the last bid.
- The bidders can accept the bid and increase the last bidding price.
- When a bidder "raises a paddle" to increase the price, the auctioneer updates the price and the new price is broadcast to all the other bidders.

Example - Auction II

- The **Auctioneer** class extends **Observable**, it knows about all the castles and calls **notify()** when a change is made.
- The classes **BidderWithDescription** and **BidderWithPhoto** extend **Observer** and register for the notification.

DEMO

Observer example - castle auction (*Lecture12_demo_observer*).

Example - Auction III

Advantages

- The Observable class (Auctioneer) does not depend on any of the GUI classes (views).
- The GUI classes are independent of each other.
- New GUI classes can easily be added.

Adapter I

Intent

- Convert the interface of a class to another interface that a client expects.
- Is used to allow classes that could not communicate because of incompatible interfaces to work together.

Also Known As

- Wrapper.

Motivation

- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

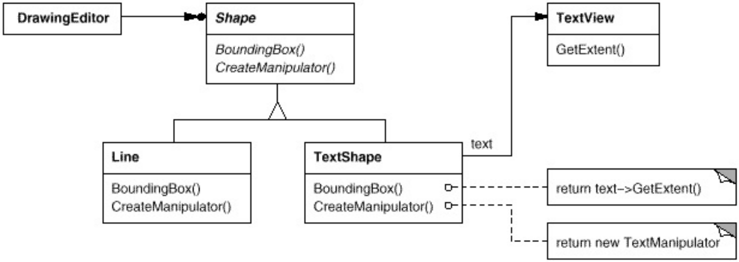


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

Adapter III

Applicability

- When an existing class could be used, but its interface does not match the one we need.
- When we want to create a reusable class that cooperates with unrelated classes (classes that don't necessarily have compatible interfaces).

Pattern class structure I

- A class adapter uses multiple inheritance to adapt one interface to another.

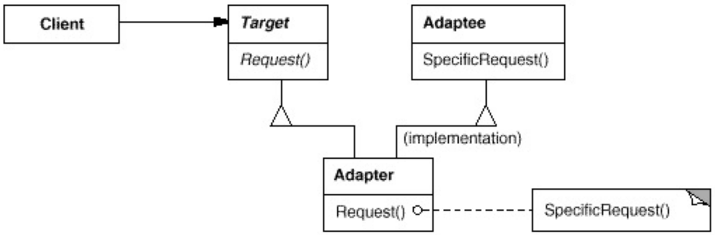


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

Pattern class structure II

- An object adapter relies on object composition.

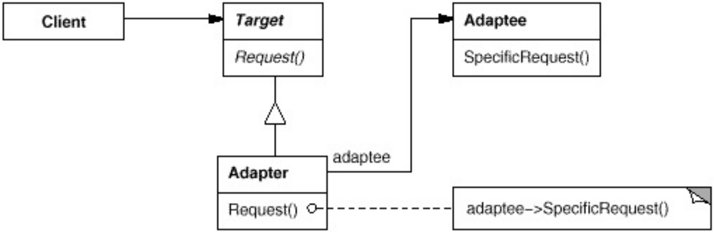
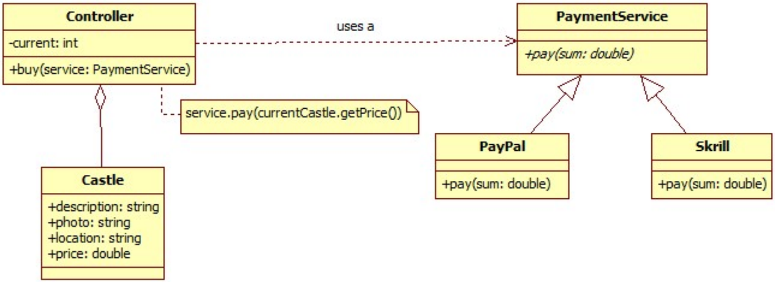


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

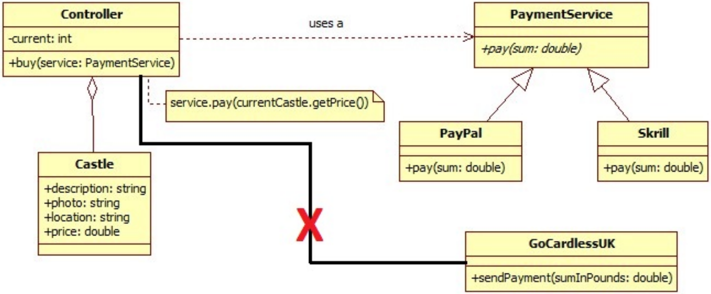
How it works

- The **Target** defines the domain-specific interface that the Client uses.
- The **Client** collaborates with objects conforming to the Target interface.
- The **Adaptee** defines an existing interface that needs adapting.
- The **Adapter** adapts the interface of Adaptee to the Target interface. The Adapter can be responsible for functionality the adapted class doesn't provide.
- Clients call operations on an Adapter instance.
- The Adapter calls Adaptee operations that carry out the request.

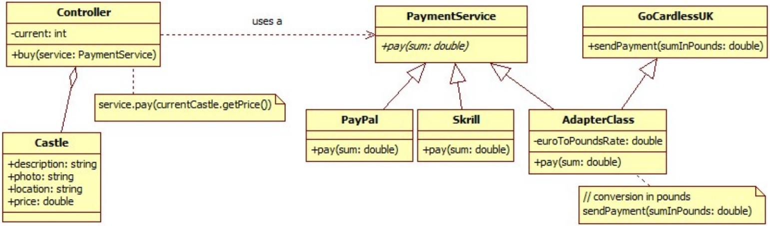
Example - Payment service providers I



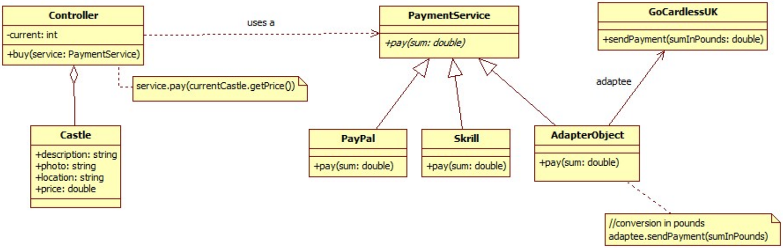
Example - Payment service providers II



Example - Payment service providers III



Example - Payment service providers IV



DEMO

Adapter example - payment service providers (*Lecture12_demo_Adapter*).

Consequences

- A class adapter:
  - because it extends the Adaptee class, it will not work when we want to adapt a class *and all its subclasses*.
  - the Adapter can override some of the Adaptee's behaviour.
- An object adapter:
  - lets a single Adapter work with many Adaptees.
  - makes it harder to override Adaptee behavior (will require subclassing Adaptee and making Adapter refer to the subclass).

Adapter pattern in STL I

Container adapters

- These are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements.
- std::stack, std::queue, std::priority_queue
- Each class has a template parameter of type *Sequence Container* and it only exports the operations that are allowed on that specific abstract data type.

Adapter pattern in STL II

- **stack**
  - LIFO (last in first out) strategy for inserting/extracting elements;
  - Elements are pushed/popped from the "back" of the specific container, which is known as the top of the stack.
  - Operations: empty(), push(), pop(), top();

```
template <class T, class Container =
    deque<T> > class stack;
```

- **queue**
  - FIFO (first in first out) strategy for insert/extract elements;
  - Elements are pushed into the "back" of the specific container and popped from its "front".

Adapter pattern in STL III

- Operations: empty(), front(), back(), push(), pop(), size();

```
template < class T, class Container =
    deque<T> > class queue;
```

- **priority_queue**
  - accesses and extracts elements based on their priorities;
  - Operations: empty(), top(), push(), pop(), size();

```
template <class T, class Container =
    vector<T>,  class Compare = less<
    typename Container::value_type> >
    class priority_queue;
```

Iterator I

Intent

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

Also Known As

- Cursor.

Iterator II

Motivation

- An aggregate (a container) should provide a way to access its elements without exposing its internal structure.
- We might want to traverse the container in different ways, but we shouldn't add operations for different traversals to the container's interface.
- The Iterator:
  - defines an interface to access the container's elements;
  - contains a reference of the container that it iterates;
  - is responsible for keeping track of the current element.

Iterator III

- Separating the traversal mechanism from the container allows the definitions of iterators for different traversal policies, without enumerating them in the container's interface.
- E.g. FilteringListIterator might provide access only to those elements that satisfy specific filtering constraints.
- Remember your lab? - iterate through elements (dogs, movie trailers, trench coats, tutorials) that satisfy certain conditions, one by one.

Iterator IV

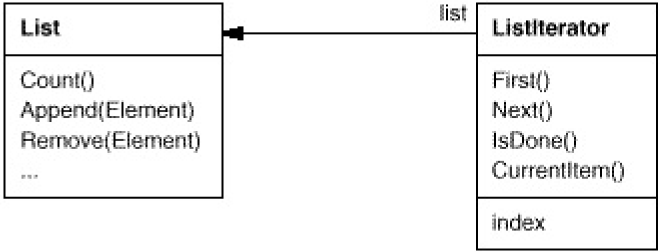


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

Pattern class structure

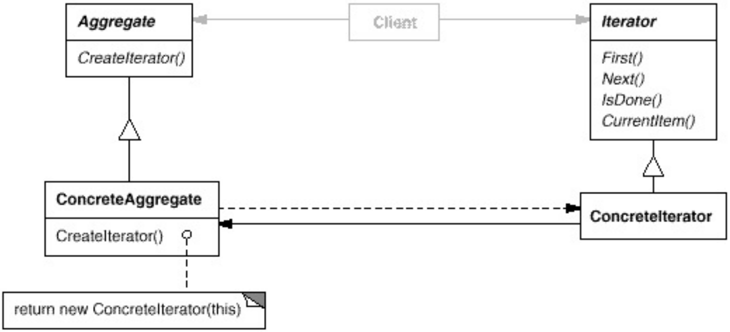


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

How it works

- **Iterator** - defines an interface for accessing and traversing elements.
- **ConcreteIterator** - implements the Iterator interface, for the concrete container. It also keeps track of the current position and can compute the succeeding object in the transversal.
- **Aggregate** - defines an interface for creating an Iterator object.
- **ConcreteAggregate** - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Applicability and consequences I

Applicability

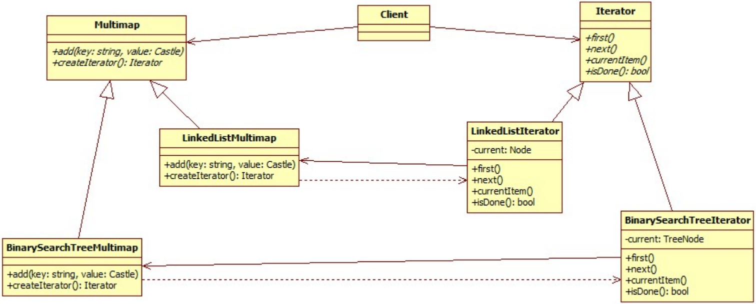
- to access a container's objects without exposing its internal representation;
- to support multiple traversals of containers;
- to provide an uniform interface for traversing different aggregate structures.

Applicability and consequences II

Consequences

- It supports variations in the traversal of a container. Complex aggregates may be traversed in many ways (e.g. binary search trees - in-order, pre-order $\Rightarrow$ just replace the iterator with a different one).
- Iterators simplify the container's interface.
- More than one traversal at once (each iterator keeps track of its own transversal state).

Example - Multimap I



Example - Multimap II

DEMO

Iterator example - countries, castles multimap (*Lecture12_demo_Iterator*).

Iterators in STL I

- Iterators in STL are objects that keep track of a location within an associated STL container object.
- They provide support for traversal (increment/decrement), dereferencing and container bounds detection.
- Iterators *are fundamental in many of the STL algorithms*. They are the mechanism that makes it possible to decouple algorithms from containers.

Iterators in STL II

- Each STL container type includes member functions begin() and end(), which effectively specify iterator values for the first and the "first - past - last" element.
- There are several kinds of iterators:
  - input/output iterators (istream_iterator, ostream_iterator);
  - forward iterators, bidirectional iterators, random access iterators;
  - reverse iterators.

Iterator types

Iterator type	Behavioural description	Operations
random access (most powerful)	Store and retrieve values. Move forwards and backwards. Access values randomly.	$*, =, ++, ==, !=$ $--, +, -, [], \rightarrow, <, >$ $<=, >=, +=, -=$
bidirectional	Store and retrieve values. Move forwards and backwards.	$*, =, ++, ==, !=$ $--, \rightarrow$
forward	Store and retrieve values. Move forwards only.	$*, =, ++, ==, !=, \rightarrow$
input	Retrieve, but not store values. Move forwards only.	$*, =, ++, ==, !=, \rightarrow$
output (least powerful)	Store, but not retrieve values. Move forwards only.	$*, =, ++$

Iterator types provided by the STL containers

Container class	Iterator type	Container category
vector	random access	sequential
deque	random access	
list	bidirectional	
set	bidirectional	associative
multiset	bidirectional	
map	bidirectional	
multimap	bidirectional	

- The container adapters (stack, queue, priority_queue) do not provide iterators.

Iterator adapter for insertion

- Insert iterators (inserters) allow algorithms to operate in insert mode, rather than overwrite mode (default).
- They solve the problem that arises when an algorithm tries to write elements to a destination container not already big enough to hold them, by making the destination grow as needed.

Types of inserters I

- **back_inserter(), back_insert_iterator** - can be used if the recipient container supports the push_back() member function.
- **front_inserter(), front_insert_iterator** - can be used if the recipient container supports the push_front() member function.
- **inserter(), insert_iterator** - can be used if the recipient container supports the insert() member function.

Types of inserters II


```
int main()
{
    std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
    std::vector<int> odds;
    std::deque<int> evens;

    copy_if(v.begin(), v.end(), back_inserter(odds), [](
        int x) { return x \% 2 == 1; });
    for_each(odds.begin(), odds.end(), [](int x) { std::
        cout << x << " "; });    // 1 3 5

    std::front_insert_iterator<std::deque<int>>
        evensIterator(evens);
    copy_if(v.begin(), v.end(), evensIterator, [](int x)
        { return x \% 2 == 0; });
    for_each(evens.begin(), evens.end(), [](int x) { std
        ::cout << x << " "; }); // 6 4 2
}
```

Composite I

Intent

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

Motivation

- Graphic application for building complex diagrams out of simple components.

Composite II

- The user can group components to form larger components, which in turn can be grouped to form still larger components.
- A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.
- Code must treat primitive and container objects differently and this makes the application more complex.

Composite III

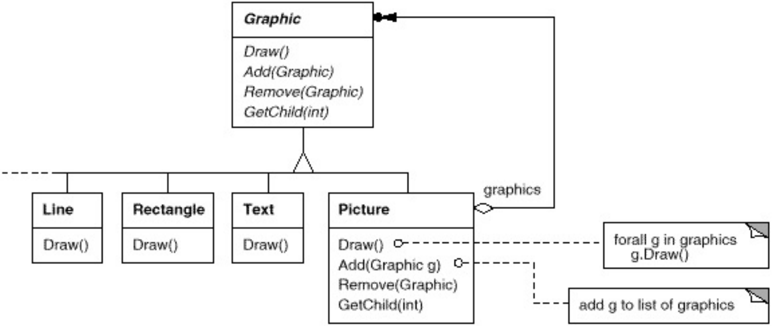


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

Composite IV

- The key to the Composite pattern is an abstract class that represents *both* primitives and their containers - class **Graphic**.
- This class will define operations that are specific to all graphical objects (e.g. *Draw()* and implement child-related operations).
- The "primitives" (Line, Rectangle, Text) will know how to draw themselves, but they do not need to manage any children.
- More complex objects (e.g. a Picture), which contain more Graphic objects will call *Draw()* on its children.

Composite V

Applicability

- When we want to represent part-whole hierarchies of objects.
- When we 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.

Pattern class structure I

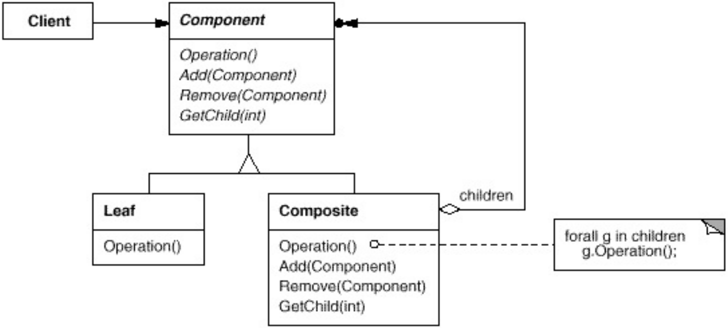


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

Pattern class structure II

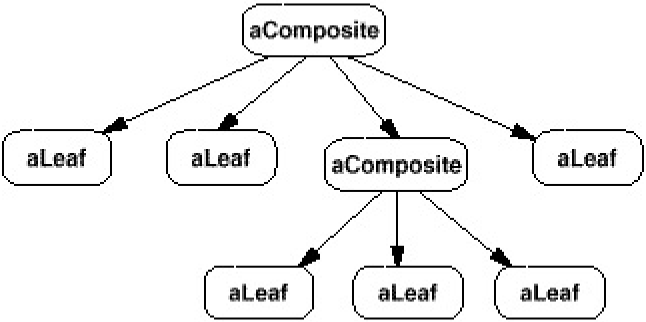


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

How it works I

- **Component** (Graphic)
  - declares the interface for objects in the composition.
  - declares an interface for accessing and managing its child components.
- **Leaf** (Rectangle, Line, Text, etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behaviour for primitive objects in the composition.

How it works II

- **Composite** (Picture)
  - defines behaviour 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.

Consequences

- Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
- Makes the client simple. Clients can treat composite structures and individual objects uniformly.
- Makes it easier to add new kinds of components. Clients don't have to be changed for new Component classes.

Demo

DEMO

Composite example - graphics (*Lecture12_demo_Composite*).

Stategy I

Intent

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

Also Known As

- Policy.

Motivation

- There can be many different algorithms for doing a certain operation (e.g. for data compression: LZ77, Huffman, RLE).
- Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:
  - Different algorithms will be appropriate at different times.
  - It's difficult to add new algorithms and vary existing ones when the operation is an integral part of a client.

Stategy III

Applicability

- When many related classes differ only in their behaviour. Strategies provide a way to configure a class with one of many behaviours.
- When we need different variants of an algorithm. For example, we might define algorithms reflecting different space/time trade-offs.
- When an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- When a class defines many behaviours, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Pattern class structure

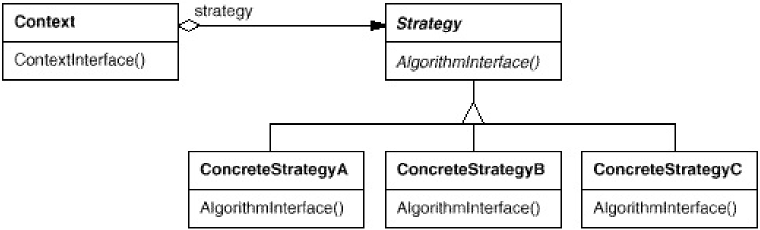


Figure source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*

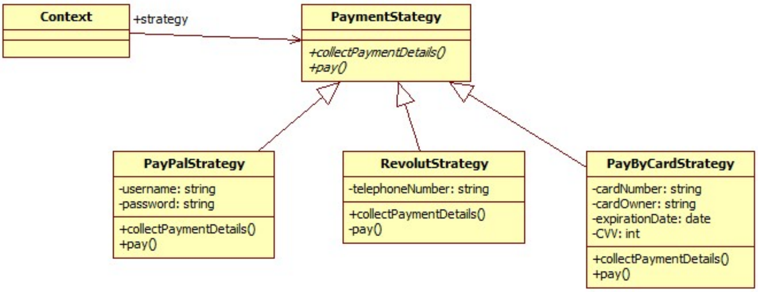
How it works

- **Strategy**
  - declares an interface common to all supported algorithms.
  - context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy**
  - implements the algorithm using the Strategy interface.
- **Context**
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

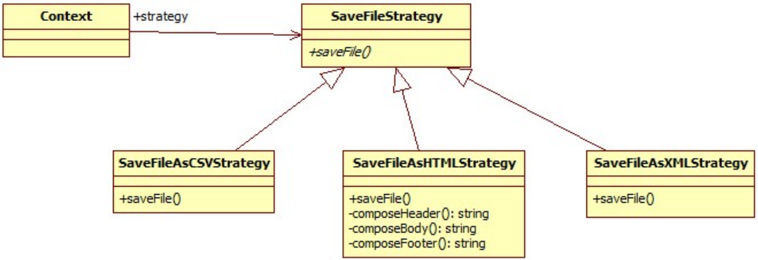
Consequences

- Hierarchies of Strategy classes define a family of algorithms or behaviours for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.
- Strategies eliminate conditional statements. The Strategy pattern offers an alternative to conditional statements for selecting desired behaviour.
- The pattern has a potential drawback: a client must understand how Strategies differ before it can select the appropriate one.

Example


Context
+strategy
PaymentStategy
+collectPaymentDetails()
+pay()
PayPalStrategy
-username: string
-password: string
+collectPaymentDetails()
+pay()
RevolutStrategy
-telephoneNumber: string
+collectPaymentDetails()
-pay()
PayByCardStrategy
-cardNumber: string
-cardOwner: string
-expirationDate: date
-CVV: int
+collectPaymentDetails()
+pay()

Demo



DEMO

Strategy example - save to CSV, HTML, XML (*Lecture12_demo_Strategy*).

Summary

- Software design patterns: recurring descriptions of classes and communicating objects that are customized to solve a general design problem in a particular context.
- Three categories: creational, structural (adapter, composite), behavioural (iterator, observer, strategy).
- Knowing what pattern to apply in which situation (applicability) and the consequences it has allows one to design better, reusable architectures.
- *"Design patterns help a designer get a design "right" faster"*. (GoF book).