GoF Design Patterns

by Paul Ianas, Edocti
paul.ianas@edocti.com

Short introduction

Design patterns are recipes which solve common design problems. They are not concrete / finished solutions! They are simply "templates" which need to be filled with your specific classes and methods.

Can we live without them?

Yes, we can... BUT we might reinvent the wheel.

Why are they useful?

1) They constitute a common design language spoken by developers. Developers spend more than 50% of their time reading code, instead of writing code. Design patterns make it easier for developers to read new code.

2) We already have good solutions to common OOP problems. Designing a solution in another way will probably yield to poor results.

There are three types of design patterns:

- creational patterns
- structural patterns
- behavioral patterns

Creational patterns

They encapsulate the logic of creating objects (== class instances). The main result: the new operator disappears.

Structural patterns

Deals with how we can combine classes and objects to obtain a larger structure which best solves certain problems.

Behavioral patterns

The first thing to note about this class of patterns is that classes will encapsulate (mainly) behavior. These patterns will show various ways of assigning behavior between multiple classes, allowing them to collaborate in a flexible / loose-coupled way.

Creational patterns

We will study the following patterns:

- Factory method
- Abstract factory
- Prototype
- Builder
- Singleton

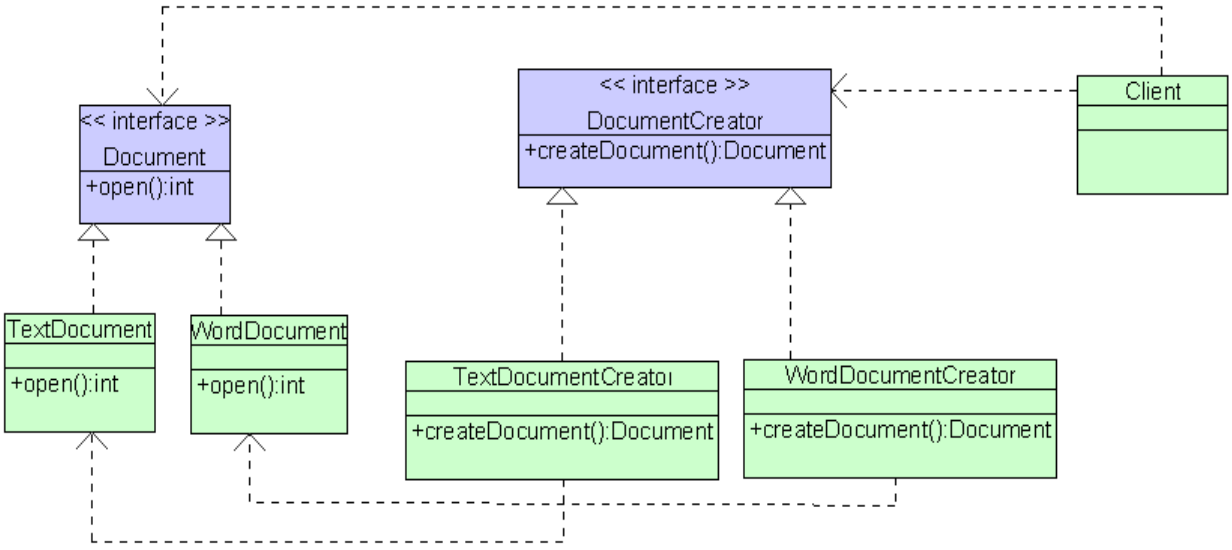
Factory method

Intent and motivation

It is primarily used to create an instance of a class (from a hierarchy of classes) in a more flexible way.

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Hide the usage of the new operator from the "client"

UML structure



As you can see, the `Client` is dependent on the abstractions only (`Document` and `DocumentCreator`). There is a decision (business logic) somewhere in the Client code (this pattern is not concerned with that) where one of `DocumentCreator's` realization is chosen.

Implementation

```
interface Document {
}

class TextDocument implements Document {
      TextDocument(String name) {
            this.name = name + ".txt";
            out.println("new TextDocument created: " + this.name);
      }
      private String name;
}

class WordDocument implements Document {
      WordDocument(String name) {
            this.name = name + ".docx";
            out.println("new WordDocument created: " + this.name);
      }
      private String name;
}

interface DocumentCreator {
      Document createDocument(String name);
}


class TextDocumentCreator implements DocumentCreator {
      public Document createDocument(String name) {
            return new TextDocument(name);
      }
}

class WordDocumentCreator implements DocumentCreator {
      public Document createDocument(String name) {
            return new WordDocument(name);
      }
}
```

Usage scenarios

Remember JDBC: there is a so-called JDBC driver which is used by clients in a flexible / loose-coupled way: obtaining a connection is not done through

```
new MySqlJdbcConnection("url", ...)
```

Being so would mean the client would have to change each time we want to work with a different DBMS provider!

```
DriverManager.setLoginTimeout(60);
try {
    String url = new StringBuilder()
        .append("jdbc:")
        .append(type)  // "mysql" / "db2" / "mssql" / "oracle" / ...
        .append("://").append(host) .append(":") .append(port).append("/")
        .append(dbName).append("?user=").append(user)
        .append("&password=").append(pw).toString();
    return DriverManager.getConnection(url);
} catch (SQLException e) {
    // error handling
}
```

Other examples

This pattern is ubiquitous throughout the Java API. Some examples:

`java.util.Pattern.compile("*.java")` creates an instance of the same class whose pattern is specified in the argument. Typical usage:

```
Pattern p = Pattern.compile("A*l");  // factory method
Matcher m = p.matcher("Animal");  // another factory method
boolean b = m.matches();
```

```
Collections.synchronizedCollection:
Collection c = Collections.synchronizedCollection(someCollection);
```

```
Logger appLog = LogManager.getLogManager().getLogger("myAppLog");
// OR simply:
Logger appLog = Logger.getLogger("myAppLog");
```

Note: there is a single instance of `LogManager` => `LogManager` is a singleton (will talk soon about this).

Pros and cons

Pros

The module which uses various documents will not be directly dependent on `TextDocument` or `WordDocument` => loose coupling <=> the same code can be used with various types of documents.

The factory class will be able to control the way in which the class is instantiated. Some examples:

- grant or decline creation rights (based on license type)
- control number of instances (maybe)

You use this in places where you have some logic for creating an object one way or another. You usually don't want to know the details for creating one type of object versus the other => let this decision be encapsulated inside a factory.

Cons

The client code may not know which exact subclass of `Document` it is using. This is usually not that important, since the client wants to work with interfaces / abstractions rather than concrete implementations (remember SOLID)

Abstract Factory

Intent and motivation

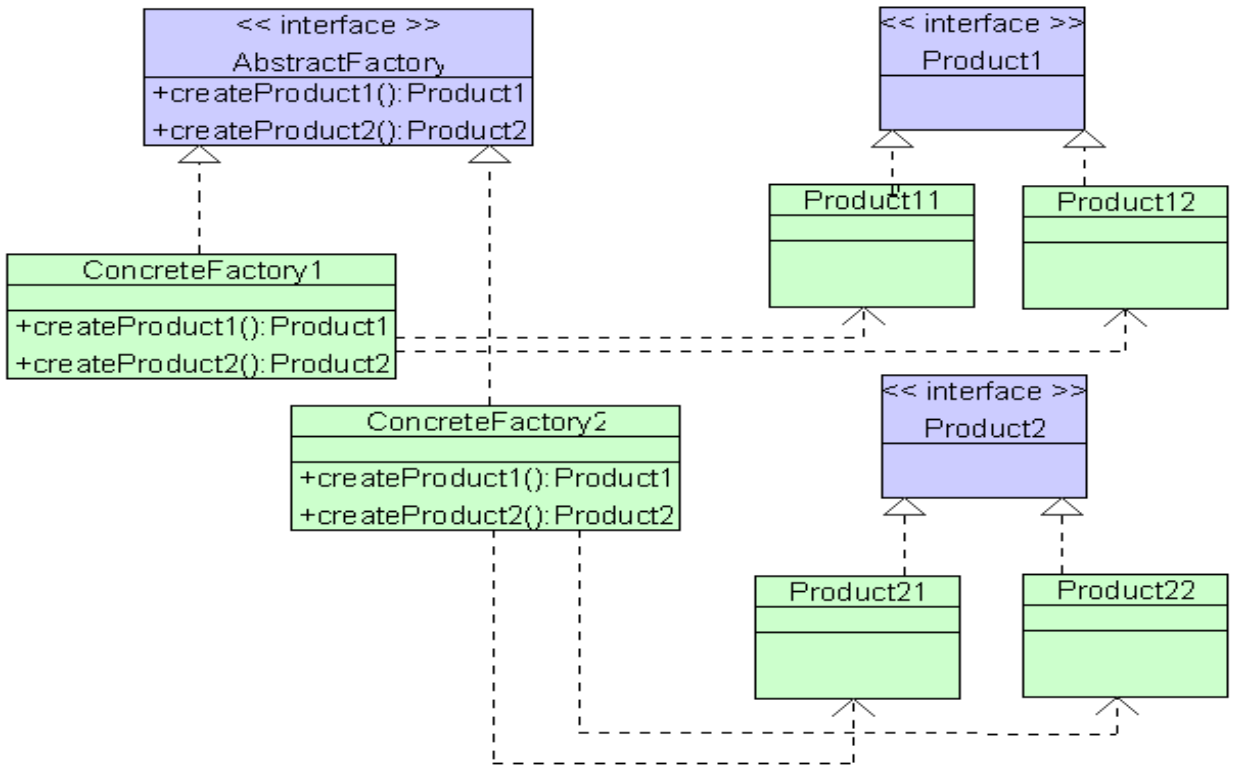
Provide an interface for creating ***families*** of related objects without specifying the actual / concrete class. This is also known as ***Kit***.

What is the difference between this and Factory Method? Well, Abstract Factory lets you to basically create / manage factories that you can then use to create your actual objects <=> it is a *factory of factories* :)

Here is why it allows you to create objects without specifying the actual type: you just work with interfaces.

Q: How do you tell it which factory to create?
A: This is usually solved via some configuration or via a parameter that you send when creating the factory => your first job is to first create / obtain the concrete factory. The pattern itself doesn't detail / impose how you do that; it's up to you. You can use for instance a call to a factory method, or the factory itself is created at application bootstrap.

UML structure



Implementation

```
import static java.lang.System.out;

interface Product1 { }
interface Product2 { }

interface ProductsFactory {
      Product1 createProduct1();
      Product2 createProduct2();
}

class Product11 implements Product1 {
      Product11() { out.println("Product11"); }
}

class Product12 implements Product1 {
      Product12() { out.println("Product12"); }
}

class Product21 implements Product2 {
      Product21() { out.println("Product21"); }
}

class Product22 implements Product2 {
      Product22() { out.println("Product22"); }
}

class ProductsFactory1 implements ProductsFactory {
      @Override public Product1 createProduct1() {
            return new Product11();
      }
      @Override public Product2 createProduct2() {
            return new Product21();
      }
}

class ProductsFactory2 implements ProductsFactory {
      @Override public Product1 createProduct1() {
```

```
            return new Product12();
        }
        @Override public Product2 createProduct2() {
            return new Product22();
        }
}
```

Usage scenarios

The previous example (the one with JDBC) is more than met the eyes in the first chapter. True, the following line implies a factory method pattern:

```
Connection conn = DriverManager.getConnection(url);
```

But the connection itself is nothing more than a factory: it creates various instances of JDBC classes:

```
Statement st =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet rs = st.executeQuery("select * from authors");
```

... etc.

So Connection is an interface which works with other JDBC abstractions (also interfaces) and provides multiple methods for creating instances of these abstractions. Again, your client is unaware of the actual class which it receives. Clients only work with interfaces => loose coupling.

Some examples:

`Connection.createStatement()` - overloaded
`Connection.prepareStatement()` - overloaded
`Connection.prepareCall()` - overloaded
`Connection.set/getAutocommit()`

Note: JDBC uses multiple design patterns, not only these two. It also uses Bridge and Proxy, which we'll study later.

Pros and cons

Pros

- Leads to a loosely coupled design
- Allows for the usage of the "right" family of classes: e.g. if working with MySQL, DriverManager.getConnection() will create the right factory for us => cannot use DB2-related classes, because the factory takes care of instantiating what is right for us

- It is easy to add or remove a configuration

Cons

The number of classes that can be instantiated is limited <=> it is fixed in the interface of the abstract factory. Adding new classes means extending that interface!

Q: Why not simply add new methods in the `AbstractFactory` interface?
A: If you'd do that, then all already-implemented clients will break. Instead, if really necessary, you can extend the `AbstractFactory` interface and use that new interface in new code.

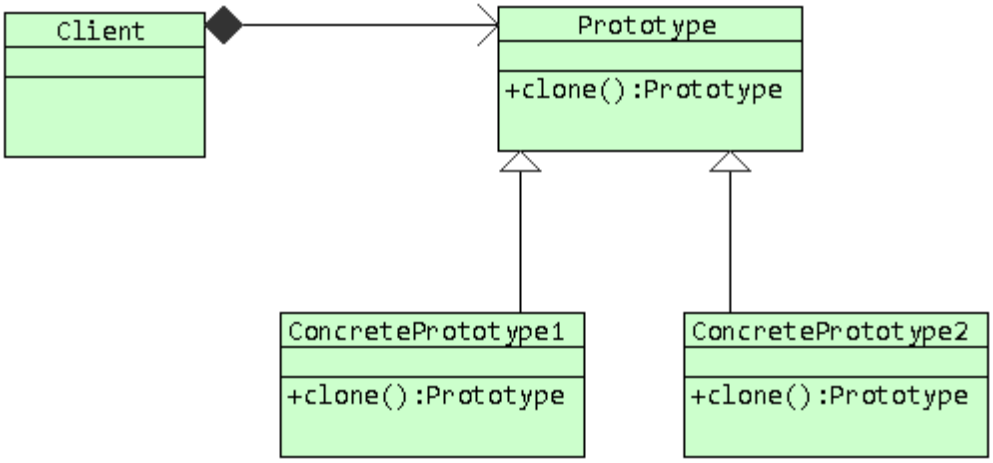
Prototype

Intent and motivation

Specify the kinds of objects to create using a prototypical instance which is then cloned. So new instances are obtained via cloning of this prototypical instance.

This can be used in cases when the creation of objects includes some expensive operations => the result of the previous instantiation is basically cached by using this prototypical instance. Usually most members of the cloned instance will remain unchanged and only some members will change.

UML structure



Usage scenarios

You can use this pattern when

- you deal with many instances which have a short life-cycle and whose creation is expensive
- you need to dynamically load classes in your application. Note: in Java there is already support for `clone()` and serialization!
    - Object.clone() implements shallow copy. If this is not enough, you'll need to override the clone() method
- avoid parallel hierarchy of factories for your (`Product`) classes

Pros and cons

Pros

- Concrete classes can be hidden from the clients => loose coupling
- New classes can be added at runtime! This may be useful in languages which don't support reflection (e.g. C++). However, dynamic library loading will be necessary also in these cases.
- Saves time by not performing expensive operations

Cons

- Each class has to inherit the clone() method from a common base class
- Copying might not be straight forward for some classes (e.g. copying and sharing socket connections, etc.)

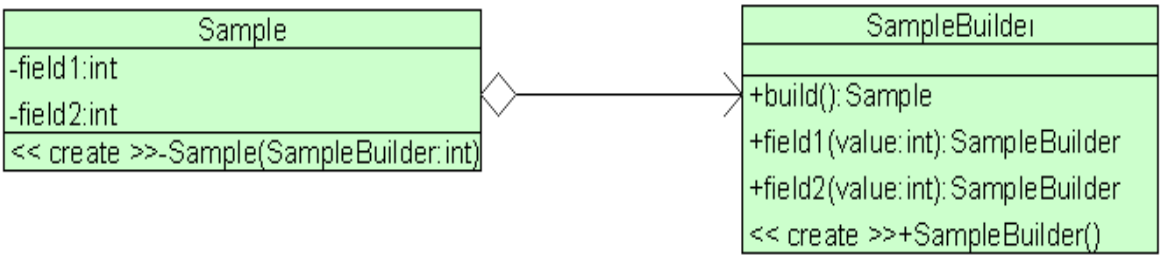
Builder

Intent and motivation

In some cases, classes might have multiple / a lot of constructors for solving multiple instantiation needs. This is what is called *telescoping constructor anti-pattern*: you have the need to create various constructors and, due to the big number of attributes, the number of constructors rapidly increases.

Builder lets you restrict the number of constructors and instead provide a single build() method which is able to construct a complex object in a unified way => the same construction model can create a variety of different construction representations.

UML structure - simplified (Java-specific) version



Builder is different than other creational patterns, in the sense that it creates the instances in multiple successive steps. It offers an elegant solution: the alternative to build a complex object would be to:

- create a constructor which takes as many parameters as possible, and call that constructor for any instantiation need
- call a variable number of setter methods until the desired object is built

Instead, we have a single constructor for our class (`Sample(SampleBuilder builder)`) and every field of the `Sample` class will be set using the same field from the `SampleBuilder` class:

```
Sample s = new Sample.SampleBuilder().field1(10).field2(20).build();
```

Implementation

Live activity...

Usage scenarios

You can use this pattern any time you have a class with many attributes and with different instantiation needs. In that case your class will use a nested static class as the Builder. In case

you don't have access to the source code of the class, you can develop the builder outside the builded class.

Pros and cons

Pros

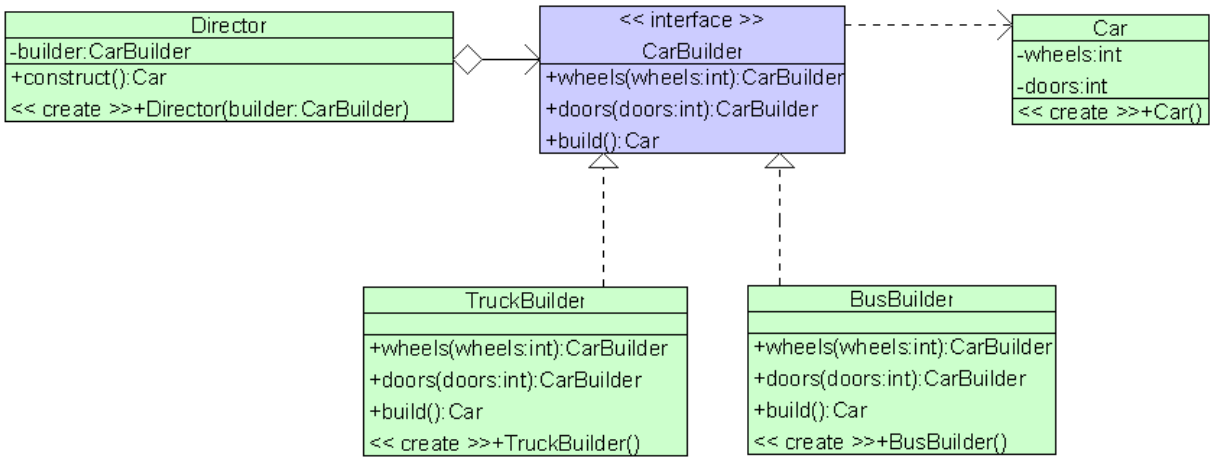
- Lets you vary the internal representation of your class
- Encapsulates the creation and representation process

Cons

Requires you to create a new Builder class for each class you want to construct in this way.

The text-book UML structure

In our implementation, the builder is a static class inside the class we want to instantiate. In the original text book, the builder is outside the class and it consists of an interface + one or more implementations:



Singleton

Intent and motivation

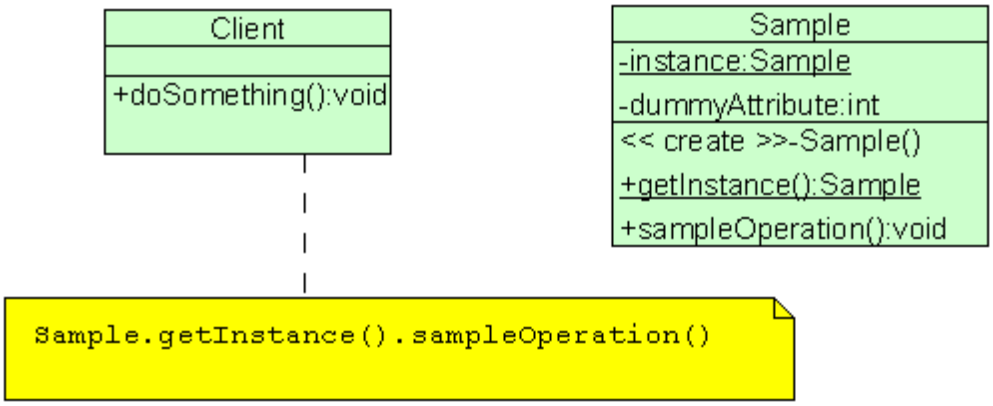
Ensures that a certain class has one and only one instance and provides a single, global method (factory method) for accessing it. Certain kind of classes require only one instance:

Examples

The LogManager is a class responsible for returning a Logger based on a logger name. There is only one instance of the LogManager.

A class that manages threads in a pool of threads should have only one instance. That instance will solve the problem of threads management (creation, destruction <=> the hysterezis behavior)

UML structure



So the Sample() constructor is made private, so that no one can access it outside the class
There is a public static getInstance() method which is responsible with returning the only instance of the class (in case the instance is null, it will instantiate it).

Implementation

```
public class Singleton {
	private int dummy;
	private static Singleton instance;

	private Singleton(int dummy) {
		this.dummy = dummy;
		out.println(getClass());
	}
```

```
	public static Singleton getInstance() {
		if (instance == null)
			instance = new Singleton(10);

		return instance;
	}

	public void doSomething() {
		out.printf("dummy = %d\n", dummy);
	}
}
```

Problem: in case multiple threads want to call getInstance(), then we run into trouble because getInstance() is not properly synchronized, and multiple instances may escape (see live activity).

Solutions:

- `private static final Singleton instance = new Singleton(10);`
  - eagerly create an instance => getInstance() will simply return it, without checking whether instance is `null`
- `public static synchronized Singleton getInstance()`
  - this will ensure that only one thread will ever enter the `getInstance()` method
  - remember: `synchronized` acquires in this case the monitor associated to the `Singleton.class` object (an object of type `Class`)
  - this solution is quite expensive: only one thread can get inside the `getInstance()` method
- double-checked locking: is thread safe AND less expensive; it is the best solution to implement in case you care about performance and correctness.

```
class DoubleCheckedSingleton {
	// volatile is important: volatile ensures that the last write
	// made by any thread will be seen by any subsequent read (made
	// from any other thread)
	private static volatile DoubleCheckedSingleton instance;

	private DoubleCheckedSingleton() { out.println(getClass()); }

	public void doSomething() { out.println("doSomething()"); }
```

```
    public static DoubleCheckedSingleton getInstance() {
        DoubleCheckedSingleton i = instance;  // this is atomic
        if (i == null) {  // i is now thread-local
            // only the first ~8 calls will reach this point
            synchronized(DoubleCheckedSingleton.class) {
                i = instance;
                if (i == null) {
                    // a single thread will be here:
                    guaranteed
                    instance = new DoubleCheckedSingleton();
                    i = instance;
                }
            }
        }
        return i;
    }
}
```

Usage scenarios

You need to have only one instance of the class. That instance will be made available via a public static method.

Note: Factories are ususally singletons.

Pros and cons

Pros

Complete control over the creation process: because you have private constructor. You can even control the creation process further:

- you can implement a "multiton": you can limit the number of instances to a number greater than 1
- you may deny object creation based on some rules

Singleton is found all over the place, including in some other design patterns (e.g. Facade and State **may be** implemented using this pattern).

Cons

You need to take care how you build your instances in a multi-threaded application (we saw this in our implementation)

Structural patterns

Deals with how we can combine classes and objects to obtain a larger structure which best solves certain problems.

We will study the following patterns:

- Adapter
- Facade
- Proxy
- Decorator
- Composite
- Flyweight
- Bridge

Adapter

Intent and motivation

Converts an interface (or a class' set of public methods) into another interface the client expects. Adapter lets classes with incompatible interfaces work together.
This pattern is also known as ***Wrapper***.

Imagine you implemented a class which is able to stream various kinds of video formats: mp4, avi, mp3, etc. Each format is handled by a separate class, but each of these classes inherits from a base class called Format and which defines the API each format must adhere to.

But now a new format emerges and you need to provide support for that new format. The new format uses an already-implemented algorithm for encode and decode, and the creators want to reuse the code that's already there. The only problem is that the new API is simply incompatible to what the base class requests.

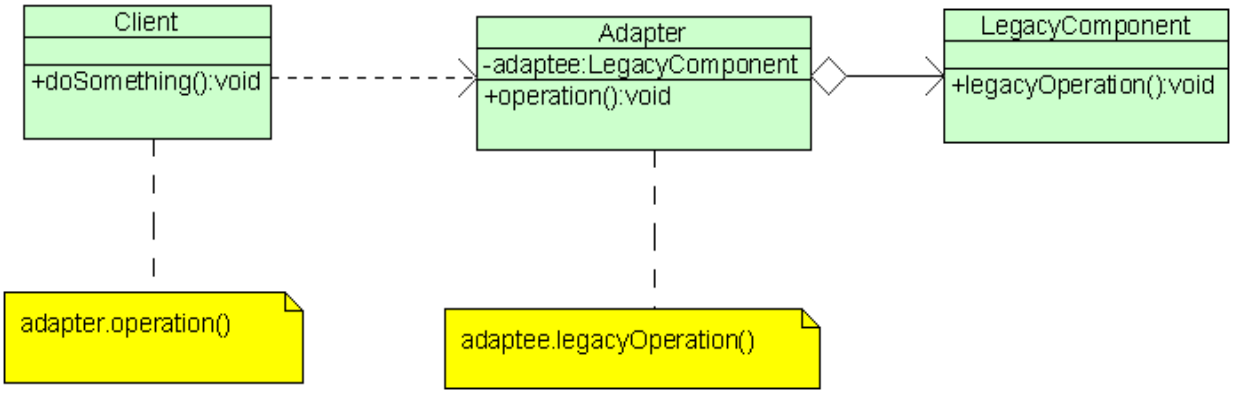
The solution is the Adapter pattern: the developers will create a new class which has an interface that the Format base class expects, and which internally calls the methods from the new (and incompatible) class.

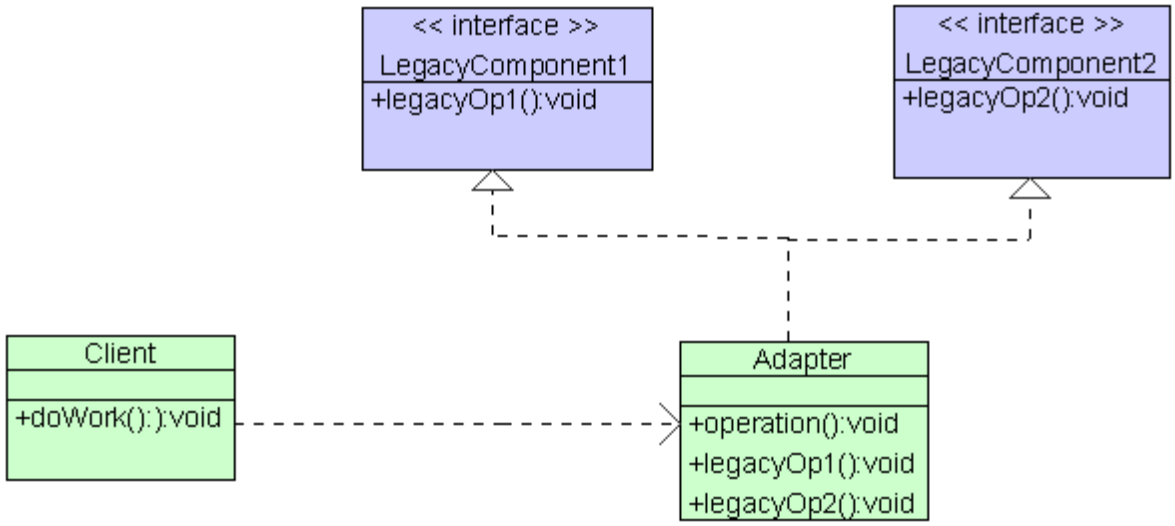
Another problem: the new class will be compatible with clients of the Format base class, but it will be incompatible with clients which expect the original API.
Solution: create a two-way adapter: an adapter which incorporates both sets of public methods <=> provide an interface which is the "sum" (reunion) of the two interfaces.

UML Structure

There are two types of adapters:

- object adapter: the adapter contains one or more `LegacyComponents` (composition + pointer indirection)
- class adapter: uses multiple inheritance in order to adapt multiple classes





Implementation

Live activity...

Usage scenarios

- Any time you need to use a type which is incompatible with your types.
- Any time you want to use several classes of the adaptee, you can use an object adapter.
- Any time you need to override some of the adaptee's behavior you can use class adaper.

Pros and cons

Pros

A class adapter

- uses the adaptee directly by calling the inherited methods
- can override adaptee's behavior
- use no additional "pointer" indirection

An object adapter lets a single adapter work with multiple adaptees

Cons

- A class adapter cannot be used in case subtypes of the adaptee need to be adapted as well
- An object adapter cannot override adaptee's behavior

Facade

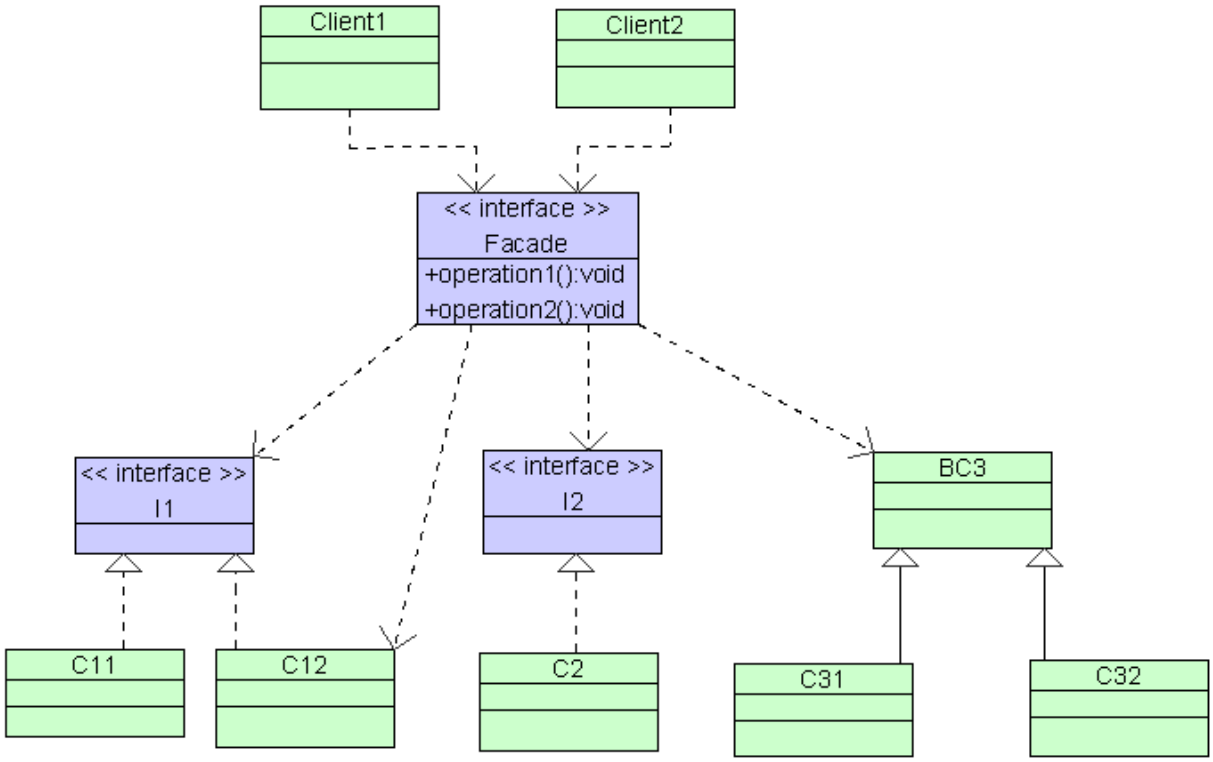
Intent and motivation

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

Consider a library of DSP classes that is used by media players to change the audio stream for different effects. It contains several classes that collaborate together to change the audio stream.

It's quite clear that there is a tight coupling between the clients and these classes => any change in the classes determine changes in the clients. This is bad!

UML Structure



Implementation

A facade can be a class or a set of classes. To increase loose coupling, the facade can be an abstract class. Different facades can be created that implement different functionalities for the target subsystem.

Usage scenarios

- Provide a simple interface to a complex subsystem
- Make a system easier to use for newcomers, but keep the old way of using it

accessible to "old timers"
- Reduce dependencies between clients and the classes they use

Pros and cons

Pros

- It isolates the clients from the complex subsystem they use => the clients will interract with a ***reduced*** number of classes.
- Loose coupling between classes and the target subsystem => changes in the subsystem won't affect its clients.
- Facade does not prevent client classes to use directly the classes from the subsystem. Facade is simply an alternative.
- For low-level data structures and functions facade provides an object-oriented interface => programming errors are reduced (via encapsulation). For instance, Java's multithreading support is a facade, as it wraps the low-level OS multithreading API and provides the same API regardless of the OS it is installed on.

Cons

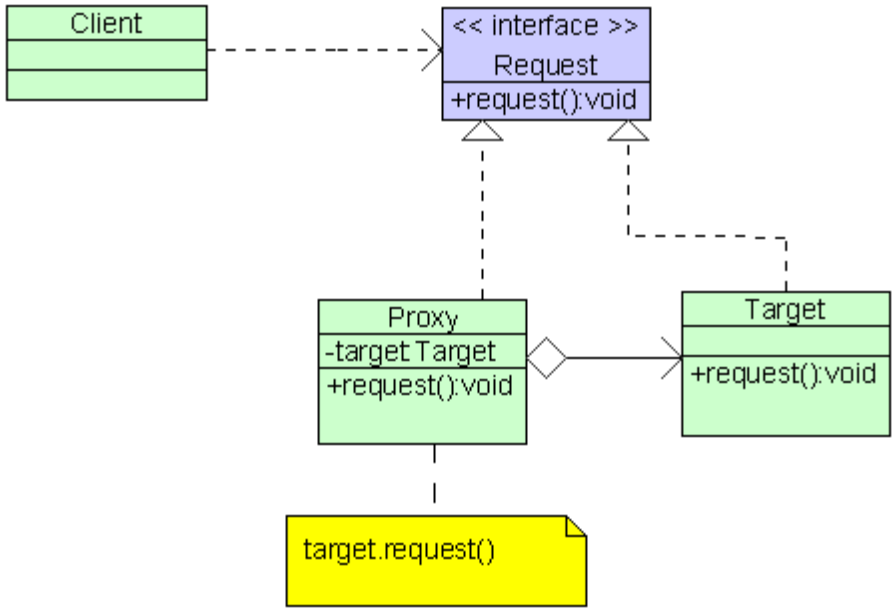
It adds another layer and this may affect performance => don't overuse this pattern.

Proxy

Intent and motivation

- Provides a surrogate or placeholder for another object in order to control access to it. You make a call to the surrogate object, and this turns around, calls the target object, then returns you the result.
- Use an extra layer of indirection to support distributed, controlled access
- Add a wrapper and delegation to protect the real component from undue complexity
- Makes it possible to support resource-hungry objects in such a way that they will be used only when actually requested by the client.

UML Structure



Implementation

- The proxy should have the same interface as the target. Why? Because the client doesn't need to know it is talking to a proxy.
- Some proxies will internally create an instance to the target object. This is the case with so-called protective proxies. It's possible to use only one proxy with several target objects if the targets are part of the same hierarchy.

Live activity...

Usage scenarios

- A ***virtual proxy*** is a placeholder for "expensive to create" / heavy objects. The real

objects are created only when requested. The proxy will remember the identity of the object => subsequent requests will be forwarded to this object

- A ***remote proxy*** provides a local "representative" to an object which resides on a different address space (and possibly a different machine). This is the so-called STUB object in RMI (Remote Method Invocation) terminology (or RPC - Remote Procedure Call).
- A ***protective proxy*** (surrogate) controls access to a sensitive object (the target object). The surrogate checks the access rights of the caller before making the actual call on the target object
- A ***smart proxy*** adds additional actions when the target object is accessed. Examples:
    - count the number of references to the real object (C++ smart pointers), in order to know when the target object can be safely deleted
    - load a persistent object into memory when it's first referenced
    - check that the real object is locked before it's accessed to ensure that no other object can change it

Pros and cons

The indirection layer may be a good thing and also a bad thing. The classic response: it depends on what you're trying to achieve.

Pros

- If you're trying to achieve some sort of control over the target object => indirection is a good thing.
- Remote proxies hide the location of the target object from the clients. This means the clients don't need to worry with communication details + there is a better security.
- Virtual proxies can create objects on demand
- Protective proxies improve security
- Smart proxies are smart and "smart" is usually a good thing :) For instance automatic memory management for C++ is currently the standard.

Cons

Indirection means less performance.

There is a tight coupling between the proxy and the target. So any change in the target translates in a change in the proxy => change in the client. BUT: the purpose of the proxy is not to allow some sort of loose coupling. That is achieved with other means.

Decorator

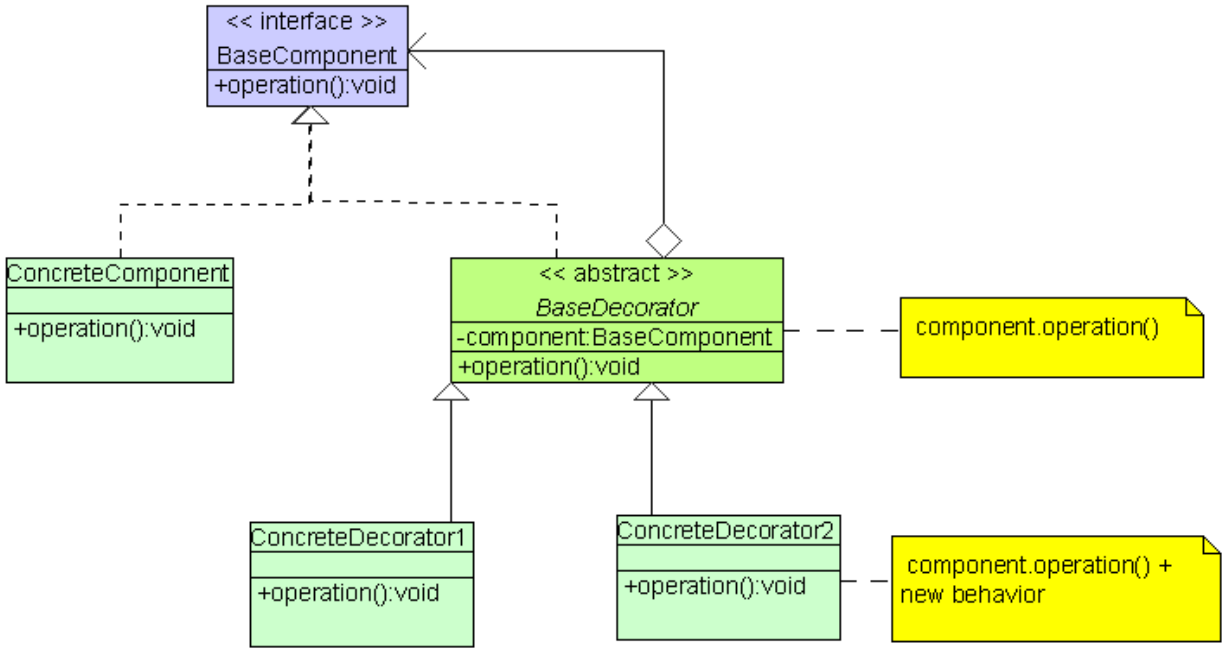
Intent and motivation

Attach additional responsibility to an object dynamically. Decorators provide a flexible alternative to subclassing.

Remember that we used Builder to solve the problem of "too many constructors". Well, decorator solves the problem of too many classes. While in the Builder pattern you created an object in multiple successive steps, here you basically want to create a new type in multiple successive steps.

Why? Because the alternative would be to have lots of classes, one for each combination of operations.

UML Structure



Implementation

Live activity...

Usage scenarios

The usage scenario is simple: use this whenever the number of classes in a hierarchy would otherwise be too big.

Pros and cons

Pros

- Using decorators is a ***flexible*** way of overriding behavior without using inheritance, but by using composition
- Features are added incrementally => there is no need to put every combination of features in a separate class
- It is easy to combine new classes together to achieve new functionality

Cons

- Lots of small objects are created
- Clients might need to know which objects interactions are possible. Possible solutions: encapsulate various combinations or otherwise have a bigger number of classes implement `BaseComponent` and thus have a decreased number of Decorator classes

Composite

Intent and motivation

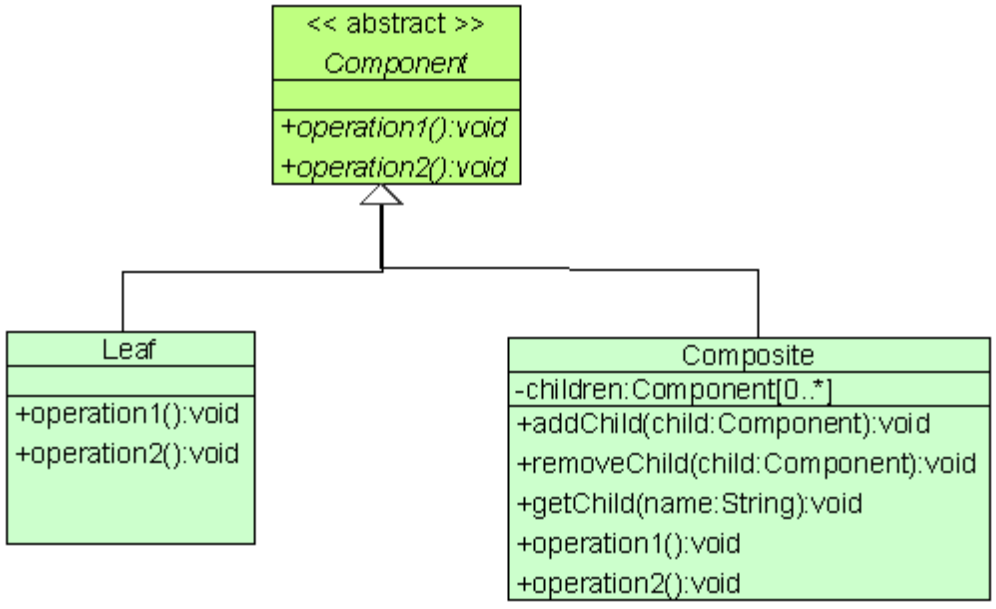
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects **uniformly** (<=> same operation).

There are situations where representing data as a **tree-like structure** makes sense. For instance in GUI applications we usually have a hierarchy of 2D widgets, where every node can be a leaf node or can act also as a collection of nodes. In this hierarchy we might need to perform the same operation on each node from the hierarchy => we use Traversers. Traversers usually have a single method for doing the operation, plus methods like `getNext()`, `getParent()` which determine the traversal order.

The traverser must not be obliged to know the exact type of the node in order to perform the operation => it makes sense to define a hierarchy of classes where at the top we have the base (maybe abstract) class.

Each node can be recursively composed of other nodes. If the node is a leaf it is said to be a **primitive**, while all the other nodes are **composites**.

UML Structure



Implementation

Some common implementation ideas:

- every node might keep a reference to its parent
- the base class should define as many operations as possible (all that are common to all node types)
- the composite has methods for children management (add, remove, ...)

Live activity...

Usage scenarios

This pattern is pretty natural when it comes to managing tree-like structures of objects. It can be used in conjunction with Visitor or Iterator (will be discussed soon) in order to add functionality. You can also use Decorator in case you need to add behavior to some of the nodes.

Pros and cons

Pros

- It simplifies the representation of part-whole hierarchies. Clients treat both Components and Compositions in the same way
- It supports the addition of new types of Components, since the Composites work with abstract Components => no changes.

Cons

In case you need to restrict the type of Components you want to accept in your composite, then you need to struggle a bit and find the right super type.

Flyweight

Intent and motivation

Use memory sharing in order to support a large number of fine-grained objects efficiently. This pattern basically is a solution to the problem of **too many objects** and more important, **too much memory**.

There is a category of applications where we deal with many objects with a very similar memory structure and naturally you'd want to **share as much memory as possible between all these objects**. Flyweight allows that and urges you to find two states for your objects:

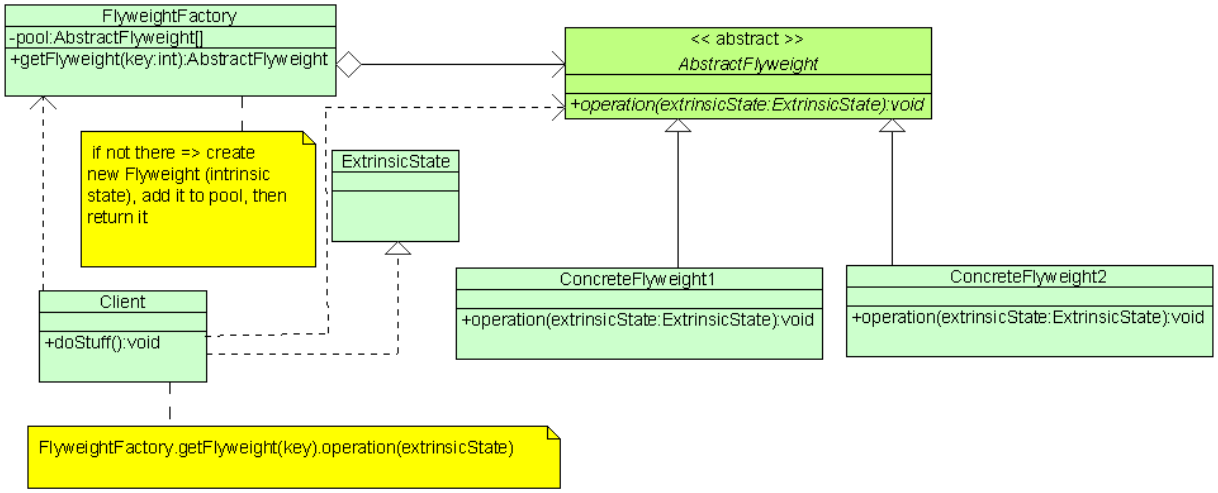
- common state (**intrinsic** state)
- individual state (**extrinsic** state)

Flyweight reduces the number of objects and the quantity of memory.

Example: in a word processing application, you have many glyph instances inside a document. However, most of them have the same attributes: font, size, color, style (bold / italic / underline) and maybe "value" (the character itself). We could think of position as being the only "individual state" member. But: the position can also be computed when the caret moves to that place.

UML Structure

- There is a factory of flyweights which internally manages the flyweight instances => the client talks to this factory in order to obtain a flyweight.
- The flyweight cannot exist on its own. It has a shareable state which is provided by the factory. The intrinsic state will be provided by the client!
- The extrinsic state (instance-specific) has been de-encapsulated and has to be provided by the client



Implementation

Flyweight objects share state and must also be thread safe => by default flyweight objects must be immutable. They are seen as data objects.

Usage scenarios

You can use this pattern when all of the following are true:

- Your application needs a big number of objects
- Due to this big number of objects, memory consumption is big
- You can find some common state between all objects
- The application doesn't depend on the identity (i.e. address) of the objects
- Many objects can be represented by relatively few shared objects once the extrinsic state is removed

Its usage in "normal" application is a bit limited. You can use it for implementing pools of resources and things like that (and these are pretty low-level modules).

Flyweight is usually combined with Composite to implement shared Primitive nodes.

Pros and cons

Pros

You can save lots of memory in case you can "extract" a big extrinsic state

Cons

- There will be some runtime costs due to computing the extrinsic state and transferring it to the objects
- Since the object state is shared, the objects become indistinguishable from each other

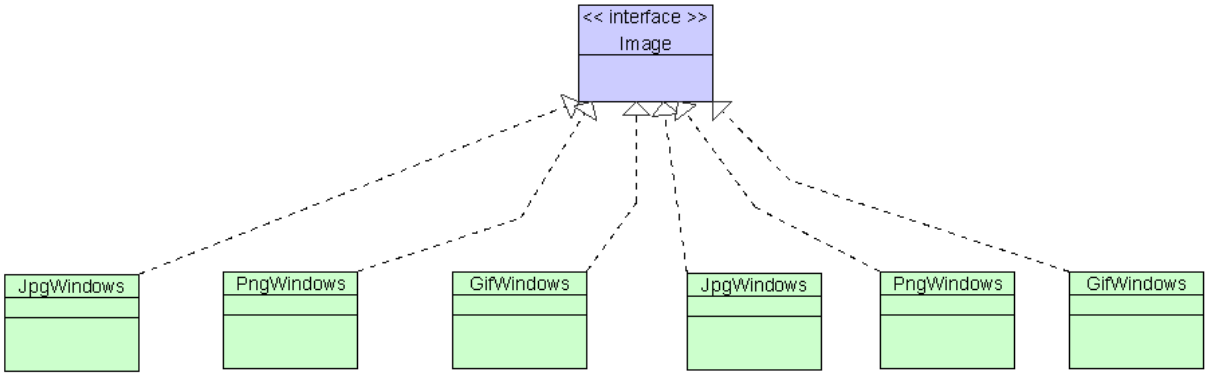
Bridge

Intent and motivation

- Decouple an abstraction from its implementation so that the two can vary independently. Also known as **handle / body**.
- Publish interface in an inheritance hierarchy and bury implementation in a separate hierarchy

Consider you need to support a hierarchy on a multitude of platforms. Without bridge, you would need to multiply the hierarchies, one hierarchy for each platform.

Example: you need to implement an image viewer, capable of displaying jpeg, png, gif, etc. You want this viewer to work on Linux, Windows, and Mac. And the first idea you could have is to simply create something like this:



This obviously seems like 'too many classes' - we basically duplicate the hierarchy, because each implementation is based on let's say DirectX on Windows and OpenGL on Linux and Mac.
More: if in the future we would like to support a new type of image, we would need to create as many new classes as the number of platforms.

UML Structure

Implementation

The decision of which Platform to instantiate can be delegated to a factory method. It's not mandatory that Platform is an interface or an abstract class, but it's usually cleaner.

Live activity...

Usage scenarios

- When you want to avoid a static / direct coupling between abstraction and implementation.
- When you want to extend the two hierarchies independently and without affecting each other
- When you want to be able to change the implementation without affecting the clients (because the clients depend on an abstraction)
- When you want to limit the number of classes in your code: WindowsPng, etc.
- When you want to share an implementation among multiple objects, transparently from to the clients.

Pros and cons

Pros

- The abstraction and the implementation are decoupled through composition => the implementation can be decided at runtime.
- Avoids having too many classes. Note: having too many classes means some "hard-wiring" logic
- It's easy to add new platforms and new kinds of images

Cons

No serious disadvantages. It's true that it creates an indirection, but that doesn't really affect performance.

Behavioral patterns

We will study the following behavioral patterns:

- Command
- Visitor
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Chain of responsibility

Command

Intent and motivation

Encapsulates a request inside an object => parameterize clients with different requests. Also known as **Action / Transaction**.

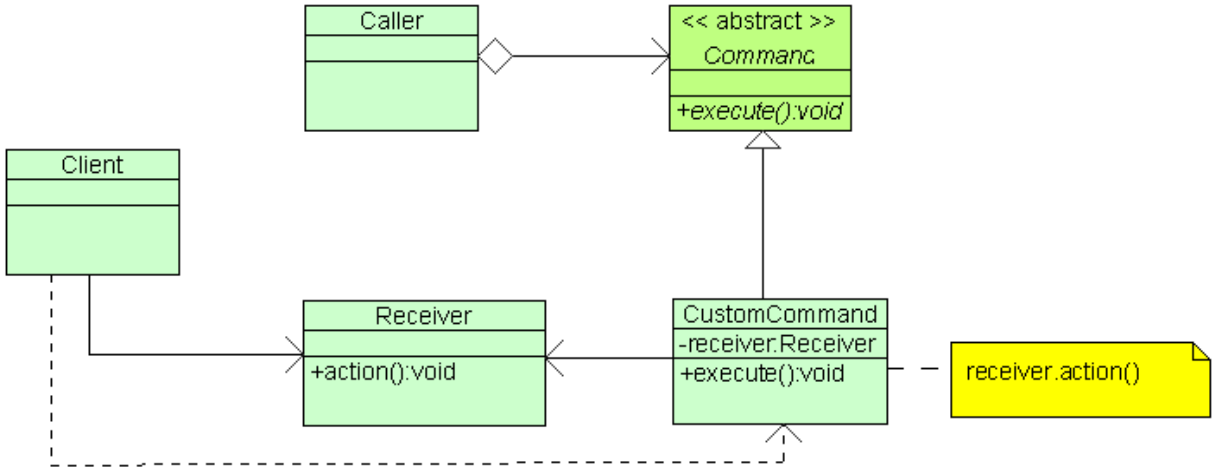
Many times applications need to implement actions based on some event; e.g. you press a button and you want a custom action to happen. Of course, GUI framework developers don't want to include your action in their code => they provide you means to specify your custom action when firing an event on a button instance (e.g. pressing the button). Then, your action will be called only when the event / action will happen. This mechanism is called **callback** ("call me back when an action happens").

So *your* ***action is encapsulated inside an object***. This object is called a **command** [object]. Its class implements a **well-defined interface**. That interface is defined inside the framework and contains a single method, i.e. `doAction()`. The Button class allows you to register your callback for a given action.

Example:

`JButton#addActionListener()`: lets you register / "hook" an `ActionListener` instance to your `JButton` object. `ActionListener` is an interface defined by the `Swing` creators. It has a single method: `public void actionPerformed(ActionEvent e)` - `ActionEvent` is the event generated when you press the button. You will have to define a class (it's usually an anonymous class) which implements `ActionListener` => it implements `actionPerformed()` - here you put your business logic <=> what happens when you press the button:

UML Structure



Implementation

```
import static java.lang.System.*;
interface ActionEvent {}

class ButtonPressed implements ActionEvent {}

interface ActionListener {
      public void actionPerformed(ActionEvent e);
}

class Button {
      private boolean pressed;
      private ActionListener listener;

      public Button() {}

      public void addActionListener(ActionListener a) { listener = a;
}

      public void eventOccured(ActionEvent e) {
            pressed = !pressed;
```

```
            if (listener != null) {
                  listener.actionPerformed(e);
            }
      }
}

public class Command {
      private static long tid() {
            return Thread.currentThread().getId();
      }

      private static void sleep(long millis) {
            try {
                  Thread.sleep(millis);
            } catch(Exception e) {}
      }

      public static void main(String... args) {
            Button b = new Button();

            out.printf("Thread %d: register listener...", tid());
            b.addActionListener(new ActionListener() {
                  @Override public void actionPerformed(ActionEvent e)
            {
                        out.printf("Thread %d: Button pressed\n",
tid());                       }
            });
            out.println("OK. Resume work...");

            // work continues; actionPerformed() called asynchronously
            sleep(100);

            // at some point in time an action occurs:
            new Thread(new Runnable() {
                  @Override public void run() {
                        b.eventOccured(new ButtonPressed());
                  }}).start();
            sleep(100);
            out.printf("Thread %d: still working...\n", tid());
```

```
        }
}
```

Usage scenarios

Use this pattern whenever you want to enable your clients to specify what happens for a given action <=> give your clients the possibility to register callbacks in an OOP manner.
Massively used in UI programming (including Web and mobile).

Pros and cons

Pros

- It lets the invoker and the handler to be loosely coupled through the command object.
- It can be manipulated and extended like any other object => you can create variations of your commands
- To add new commands, you don't need to modify your framework classes

Cons

No idea :)

Visitor

Intent and motivation

Encapsulates an operation to be performed on the elements of an object structure. Visitor lets you define actions which are not part of the class structure that you operate on. **You can thus add new behavior without changing the "target" classes.**

Consider you have a fixed hierarchy of classes and you want to add new behavior polymorphically => you would normally add these new methods to the base class, but this might break the single responsibility principle. Visitor lets you add new behavior dynamically without modifying the hierarchy itself.

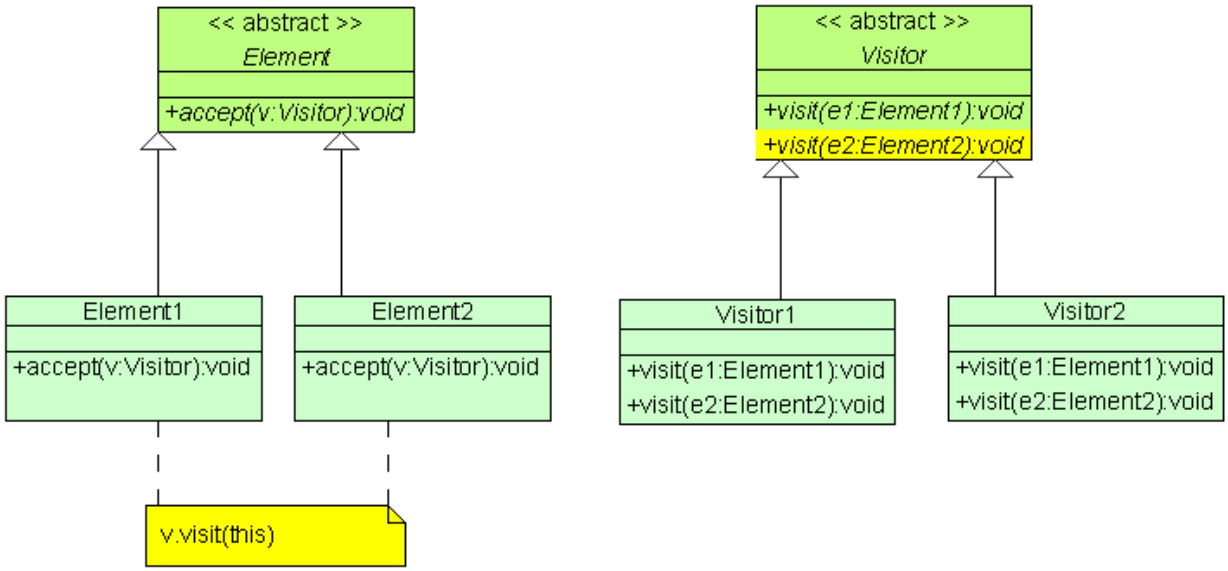
Example:

A class which abstracts an Employee. And remember that a good class design satisfies the Single Responsibility principle => we won't add all possible actions that could happen with an Employee inside the Employee class. Such operations might be:

- `calculateSalary()`
- `calculateTax()`
- `calculateBonus()`
- `addAccessRights(Resource r)`

More, you want these actions to be "specialized" for certain types of Employees (managers, programmers, accountants, etc.) => you would need to override all those methods inside those "special" classes.

UML Structure



Implementation

You add a new hierarchy of classes which implement Visitor. This interface allows you to virtualize an operation from the fixed class hierarchy <=> you add a method for visiting each type of class in your fixed hierarchy.

Live activity...

Usage scenarios

- Use this whenever you have a set of **possibly unrelated classes (usually inside some object structure)** and you want to add behavior to those classes without affecting the classes themselves.
- The classes defining the object structure won't change, but the operations will

Pros and cons

Pros

- Adding new operations is easy: just implement a new Visitor.
- Allows you to respect the single responsibility principle and to also add new behavior to your classes.
- Being objects, visitors can accumulate state => you can implement various accumulate algorithms with the objects from a structure.

Cons

The Visitor interface is fixed (one operation for each type of class) => adding new classes is difficult.

Iterator

Intent and motivation

Proposes a **standardized way of iterating** through the items of a collection (or object aggregate) in a unified way. Each collection type might have its own methods for going to the next element, to the first or previous element, and so on.

More, each collection type usually has some unique internal representation. Without Iterator you would need to know the internal representation in order to correctly iterate through it. Let's see how.

Example

Imagine you have a **circular buffer**: a buffer with a fixed number of elements; if the buffer if full, and a new element is added => that element will be placed on position 0 => we have a FIFO queue where it is possible to "overflow", because overflow simply translates into reset of the current index and overwrite of the first element.

With the help of an iterator you would iterate through this collection in the same way as you'd do with other collections. Iterator is an interface which can be implemented by any actual iterator. That iterator would have an internal attribute whose type is your collection. It's life cycle will be shorter than the life cycle of the collection object => aggregation.

UML Structure

Implementation

Live activity... Let's implement a circular buffer, which uses an Iterator and which allows use to use the for-each syntax.

Usage scenarios

- If you design a new collection you will also define an Iterator to it. Always :)
- All major collection frameworks (Java, C++ STL, etc.) use this pattern
- Composite is often used together with iterator to support traversal in a unified way.

Pros and cons

Pros

- Decouple the iteration type from the collection itself; e.g. you may have multiple iterators <=> collection traversal can be done in multiple ways, according to your needs
- If you decide to add an iterator to an aggregate => the aggregate interface will be kept simple <=> the logic will be outside the aggregate class

Cons

No idea :)

Mediator

Intent and motivation

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.

In other words, mediator will encapsulate the interaction between multiple classes, such that these classes will not be dependent of each other.

UML Structure

Implementation

Usually there is no need to create an abstract mediator. That would be needed only if the clients would want to work with more mediators.

Usage scenarios

In real applications you may want to use Model-View-Controller. MVC is a form of Mediator where the mediator itself is split into model, view and controller. Here

Pros and cons

Pros

- "Hides" the interaction into a single object => all involved parties have one, central dependency.
- => loose coupling between colleagues
- <=> replaces many-to-many relationships / interactions into one-to-many

Cons

- If the interaction is "heavy" => the mediator class can become too big and hard to maintain.
- All involved parties use an extra indirection in order to talk to each other. This may affect performance.

Memento

Intent and motivation

Without violating encapsulation, capture and externalize on object's internal state, in order for it to be restored at a later time.

This is encountered very frequently in applications, but we usually referred to this as serializing / deserializing. For instance, your IDE needs to keep some settings and some state, such that next time when you open it, to have the same set of source files opened, font size to be the same, and so on.

Another use is the UNDO behavior. You want to be able to undo the actions you are performing in an application => the action itself is encapsulated, stored in an action history FIFO queue, then you are able to undo. **Of course**: your application should be architected in such a way that its state ca be easily stored and restored using a chain of events! Otherwise the undo itself is not possible. BUT here we are not concerned with how the application uses these objects to update its state. We are only interested in storing and restoring the object's internal representation, e.g. serialization / de-serialization.

This pattern proposes three actors: the originator, the memento, and the caretaker:

- the originator is the object whose state we want to control (set, restore, undo)
- the memento is the object which encapsulates the state
- the caretaker is the object which works wit both these actors: it is the "controller" which actually implements the undo behavior

UML Structure

Implementation

Live activity...

Usage scenarios

Use it anytime you want to be able to restore the state of an object (originator). In most programming languages there is built-in support for serialization and de-serialization. In Java, an object ca be serialized if it implements the Serializable interface. The `transient` specifier can be used for restricting certain class attributes from being serialized.

Pros and cons

Pros

- This pattern doesn't break encapsulation; the state of the originator does not leak to the caretaker!
- The pattern itself is easy. More complicated could be the actual state restore (you have to design your originator with this goal from the start)

Cons

These three actors are not enough for restricting the access to Memento's state.

Observer

Intent and motivation

This pattern allows you to elegantly inform interested actors (observers) when the state of an object (the subject) has changed. This pattern is also known as **publish - subscribe**.

This pattern is used in a lot of projects. Most big projects use it to implement this notification scheme. The pattern uses the same callback mechanism that we saw previously: "call me back when the state of the subject changes".
Example: say you have an application which displays information about a product: price and number of available items and for the same product your server serves 10 mobile applications and 5 web applications. So you have 15 customers buying your product. As soon as somebody buys it, you want to inform all the other that the number of available items has decreased. This kind of scenario is elegantly solved using the observer pattern.

UML Structure

The publisher is the object whose state is the subject of observation. The objects interested in publisher's state change are called observers. There are two methods for registration and deregistration of observers.

Implementation

The publishers must keep references to all of their observers in order to inform them of state change.
Observers can observe the state of more than one publisher => the subjects send observers the `this` reference.

Java

In Java you already have the `Observer` interface and the `Observable` class => don't reinvent the wheel: use them.
Live activity...

Usage scenarios

- Use this pattern whenever you want to trigger actions as a response to a change in an object's state.
- Use this pattern also when you want to decouple your subject from its observers.
- This pattern (together with mediator) *is at the core of MVC*.

Pros and cons

Pros

- Reduced coupling between subjects and observers:
    - any class can use a subject without being dependent on its observers
    - any number of observers can be added without modifying the subject
    - observers can be added and removed at runtime (yes, we remember: composition is a magic tool which allows this kind of things)
- Broadcast and multicast communication support:
    - by default observer uses the broadcast communication type: notify everyone that a state change occurred
    - BUT: it is also possible to notify a group of observers for a certain "kind" of state change (a sub-state) => some observers are notified for some sub-state, while others are notified for other sub-states

Cons

Not serious: te simple update interface makes it difficult for the observers to deduce the changed item.

State

Intent and motivation

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. This pattern is applicable when most if not all of the methods from your class contains the same conditional statements.

Sometimes the behavior of a class needs to change based on its state. Often this is represented as **conditional statements in (most if not all of) the methods of the class**. When the values of the attributes change (for instance a boolean attribute), the behavior change is determined through conditional statements and these may occur in many functions of the class.

If your class has the same conditional statements in its methods, it's better to extract the conditional statements' branches into multiple classes and use composition to switch between these states / classes. So what you basically do is simply **abstract the states of your class in the same number of states and use them to delegate behavior**. This is the main idea behind the State pattern.

Each state class will encapsulate behavior and the **original class will delegate behavior to one of its state classes**. One of this classes will be used depending on the state of the containing class.

The logic of transitioning between states will be usually placed inside the state classes themselves. This is a disadvantage.

UML Structure

Implementation

State objects can be implemented as singletons (usually). Of course, it means that the state objects themselves don't ave internal state. That should be the case normally, since all we need to encapsulate is behavior.

Let's say you have a class called Car and it controls various components of the car depending on the driving mode:

- sport
- comfort

Imagine that the code for controlling the injection could look like this:

```
void controlnjectors() {
      switch (drivingMode) {
      case SPORT:
            // big fuel consumption
      case COMFORT:
            // small fuel consumption
}
```

The code controlling the dampers also depends on the driving mode:

```
void controlDampers() {
      switch (drivingMode) {
      case SPORT:
            // rigid dampers
      case COMFORT:
            // relaxed dampers
}
```

... and so on. We observe that multiple operations of the car can depend on the driving mode => the car itself has 2 states. Having two states which we can abstract => the class definition for the state is immediate:

```
enum DrivingMode {
      SPORT(SportState.getInstance()),
      COMFORT(ComfortState.getInstance());
      public DrivingMode(DrivingState state) {
            this.state = state;
      }
      public DrivingState getDrivingState() { return state; }
      private DrivingState state;
```

```
}

interface DrivingState {
	void controlInjectors();
	void controlDampers();
}

public class Car {
	private DrivingMode drivingMode;
	private DrivingState delegate;
	public void controlDamper() {
		delegate.controlDamper();
	}
	public void controlInjectors() {
		delegate.controlInjectors();
	}
	public void changeState(DrivingMode mode) {
		this.drivingMode = mode;
		this.delegate = mode.getDrivingState();
	}
}

class SportState implements DrivingState {
	public void controlInjectors() {
		// big fuel consumption
	}
	public void controlDampers() {
		// rigid dampers
	}
	private static final SportState instance = new SportState();
	public static SportState getInstance() { return instance; }
}

class ComfortState implements DrivingState {
	public void controlInjectors() {
		// reduced fuel consumption
	}
	public void controlDampers() {
		// relaxed dampers
```

```
    }
    private static final ComfortState instance = new ComfortState();
    public static ComfortState getInstance() { return instance; }
}
```

Usage scenarios

Main idea: if you find yourself in front of a class and realize that you have to implement many conditional statements because the class has a few states that you want to support, then this is a good sign that you can use the state pattern.
Observation: of course you have conditionals inside classes, but **a true indicator is that you have *the same* conditions in multiple methods**.

Pros and cons

Pros

- State-specific behavior is exported / encapsulated in a separate class. There is some loose coupling between the "original" class and its states => it is possible to add new states without changing the class.
- State transitions are represented by objects, not by conditional statements + variables.
- We basically extract the multi-state behavior of a class into a smaller class + one state class for each state => errors will occur less frequently

Cons

Having transitions performed inside the state classes means that states know about each other. This is a disadvantage, because they are tightly coupled.
When state objects are created on the fly, performance can decrease.

Question

What is the difference between ***proxy*** and ***state***? They are both delegating behavior, still there are separate design patterns for them.

Strategy

Intent and motivation

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy allows the algorithms to vary independently from clients that use it. Also known as **Policy**.

So what is the main difference between State and Strategy? State allows your class to be in a single state at a time: **several methods of the class behave differently according to the state**. Strategy allows to have multiple combinations of states: one state determines the behavior of method 1, while another state determines the behavior of method 2 => multiple combinations of these states determine multiple combinations of method behavior for method 1 and method 2.

Strategy is thus a combination of states.

Example

In the Car example above, we can think of also other states for the car:

- driving speed less than 60 km/h => LKAS deactivated and ACC deactivated
- driving speed bigger than 60 km/h => LKAS activated and ACC activated
- raining => braking behaves in a certain way
- not raining => braking behaves in another way

... and so on. We can observe that the Car has more than one State class. It has multiple states => multiple ways of driving, depending on the combination of states.

UML Structure

Implementation

Strategies might require data to operate on from the context. The Context can define an interface for the Strategy to access its data.

Usage scenarios

- You want to configure the behavior of your class
- Same as for state: if you detect that some methods have the same conditional statements
- Refactoring: you see complex class hierarchy: many classes, and in each class only a few methods are overridden. This is the inheritance way of solving this problem. Instead, use states for all versions of every method which is overridden, then delete the children classes.

Pros and cons

Pros

- It can be used as an alternative to inheritance. Strategy can be used instead of adding new behavior to the Context class through inheritance.
- It can be used as an alternative to conditional statements inside class methods
- It allows the client to easily modify behavior

Cons

- The number of classes increases.
- May lead to overhead in the sense that you use strategies no matter how complicated or simple the algorithm is. In some situations it's not worth to over-engineer. Try to avoid if the code is trivial and won't change.

Template method

Intent and motivation

Define the skeleton of an algorithm in an operation, deferring some steps to the subclasses => the subclasses can override those steps, and thus define / implement those steps, such that the algorithm is now complete.

Application frameworks function this way. They define the main structure of the application and hide the "unwanted" code from the developers. They allow developers to concentrate only on what matters most.

For instance, whenever you create a servlet you are using an application framework. Most of the code is hidden in the base class and solves things like representing requests, servicing requests, sending requests further, working with filters, etc. All this code is hidden; you just override the `doGet()` method and you have a working servlet. This is good: you are not interested in all the other operations which are needed for handling HTTP requests.

UML Structure

Implementation

Usage scenarios

The template method is used in application frameworks. The applications which are built on top of an application framework has two parts: a part which doesn't change (invariant) and a part which can change or which can be customized. The application framework will implement the invariant part and will let application developers implement all the other parts.

Pros and cons

Pros

It eases the development for certain kinds of applications. E.g. Spring framework allows you to concentrate on high-level details / business logic, models, views, etc. All the code needed for supporting various authentication mechanisms is already there. You only need to configure your application to do what you want.

Cons

It forces the developer to adopt the general architecture imposed by the framework. It depends whether this is a good thing or a bad thing; it may be bad for performance because the framework is general enough to allow many types of applications => your stack becomes huge even for simple operations => real-time applications usually aren't developed using these kind of frameworks.

On the other hand software development starts to be like a game of Lego, because the developer's role will be to fit the pieces together such that the application is working.

Chain of Responsibility

Intent and motivation

The CoR is thought for situations where you want a combination of handlers to process your requests => the idea is simple: chain a set of handlers into a "chain" and let each handler decide what to do with the request:

- don't process the request and pass it further
- process and pass further
- process and exit (break the processing chain)

So each handler from the handling chain can decide what to do with the request. This means that each handler has the responsibility over the request, hence the name Chain of Responsibility.

Maybe the most common and natural example is the "mail filter" feature in an email client application. The email clients needs to perform certain actions based on email properties: sender, receiver, subject, etc.
There are multiple email filters, and some are configured to stop processing once the filter is activated and executed => email enters the system and the filters start their processing: if the filter doesn't match => the request is sent to the next filter. If no match => probably there is an internal default "filter" which simply adds the incoming email to Inbox.

This is the exact way this pattern works. Another example is HTTP filters from the Servlet API.
A filter is an object that may get activated on either the HTTP request, or on the HTTP response, or on both.

Of course, the application shouldn't be concerned with the exact type of the filters => we'll have an interface which will be implemented by the filters.

UML Structure

Implementation

The pattern gives some freedom of implementation. Basically what we have is a chain of strategies (single-method) which gradually try to solve the request. The processing will end in two situations:
the request has been solved (whatever that means in your application)
the end of the chain has been reached. What you do in this situation is up to your application; you can either process the request using a default handler, or you consider your request as being unsolved.

Let's implement a simple email filtering example: live activity...

Usage scenarios

If we think about this, we actually implement a ***chain of strategies***. This chain will gradually / iteratively solve our request => flexibility: *while problem not solved, try to solve it using the next strategy*.

- Thus, we use this whenever the caller doesn't know which object will solve its request.
- Used for handling GUI events
- Exception handling (internally)
- The filter mechanism from the Servlet API.

Pros and cons

Pros

- The sender doesn't need to concern who will actually solve its request. The whole application behaves like an expert system; a system which is able to determine the

right handler for the client's request.
- Handlers can be added dynamically

Cons

Since the request has no explicit handler, it's not mandatory that a handler will process it.

Interpreter

Intent and motivation

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

It is sometimes very attractive to solve a problem using more than one programming language; a problem that is very difficult to solve in one language can often be solved quickly and easily in another. ***If the user needs greater run time flexibility, for example to create scripts describing what he/she wants from your system***, then this pattern will help you achieve that in a flexible way:

- Map a problem domain to a language
- Define the grammar for that problem domain; use a **recursive grammar**:
    - each rule in the grammar is either a 'composite' (rule referencing other rules)
    - or a terminal (leaf) rule
- Represent / map the grammar to an object-oriented design: each class is mapped to a grammar rule

The algorithms which operate in this area first tokenize the input and construct some sort of tree-like structure, called an **abstract syntax tree (AST)**, which is managed internally. The Interpreter relies on the recursive traversal of the Composite pattern to interpret the sentences.

UML Structure

Interpreter suggests having a hierarchy of expressions where some expressions are "leaf" or terminal expressions and others are non-terminal expressions (they aggregate multiple terminal and non-terminal expressions).
The **parsing context** simply includes input and output and is used to keep some state while traversing.

Implementation

Live activity...

Usage scenarios

- Use it for giving your users the possibility to control what happens in a very flexible way: task automation, scripting support, expression languages, etc.
- If the rules are too few or the language does not evolve => you probably don't want to use this solution.
- The pattern does not address parsing. This is done separately.

Pros and cons

Pros

- grammar rules are represented by classes => easy to define new rules (new classes) and to specialize the rules (inheritance or composition)
- classes which define nodes in the AST (abstract syntax tree) are easy to implement (same interface, similar implementations)
- it is easy to evaluate expressions in a new and different way.

Cons

Every rule corresponds to a class => complex grammars (many rules) will be hard to maintain; OOP is not well suited for complex grammars.