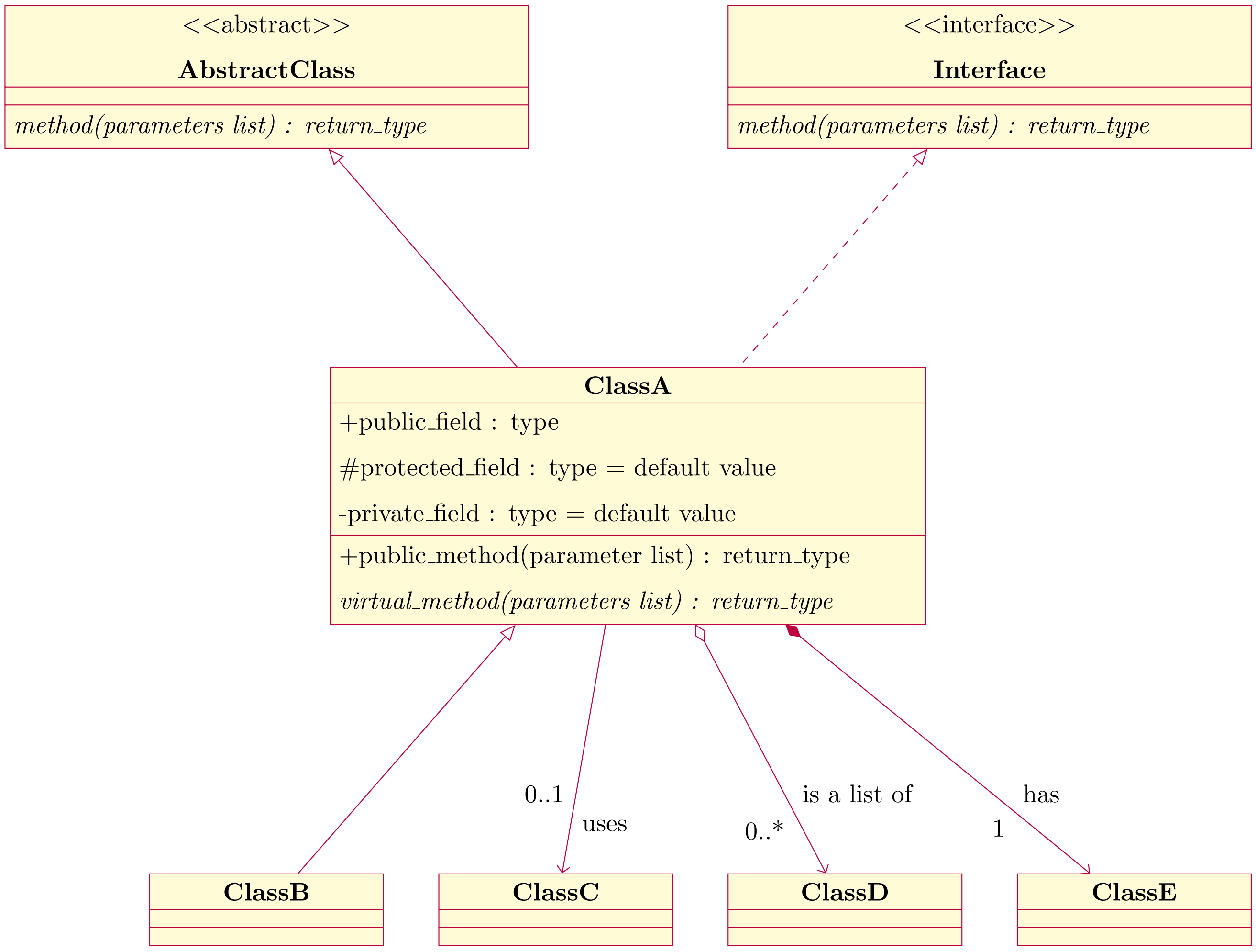
Design Patterns

Wai Tsang

Jun 1 2021
updated March 20, 2025

Contents

1 UML


<<abstract>>
AbstractClass
method(parameters list) : return_type
<<interface>>
Interface
method(parameters list) : return_type
ClassA
+public_field : type
#protected_field : type = default value
-private_field : type = default value
+public_method(parameter list) : return_type
virtual_method(parameters list) : return_type
0..1
uses
is a list of
0..*
has
1
ClassB
ClassC
ClassD
ClassE

- **inherits** ClassB inherits from ClassA and ClassA inherits from AbstractClass
- **implements** ClassA implements the interface Interface
- **association** ClassA uses ClassC. (Somewhere in the code it refers to ClassC)
- **aggregation** ClassA has a field that refers to ClassD, e.g. it might have a list of ClassD. The referenced object does not belong to ClassA, it has independent lifetime and may be shared.
- **composition** ClassA has a field that refer to ClassE. It is a stronger relationship than aggregation and suggests ownership. This referenced object does not have independent lifetime.

2 SOLID Principles

- **S**ingle Responsibility - Classes should have small responsibility. This helps to reduce the frequency it changes over future versions

- **O**pen for extension / Closed for modification. Changes should be made by adding classes rather than modifying them. Typically this would involve extracting interfaces and implementing those.
- **L**iskow Substitution Principle - In an inheritance chain, it should be possible provide a subclass whenever a base class is expected and the program carries on working. This basically means that every subclass should be able to do everything that the base class does; i.e. support for the base class's behaviour should not be reduced.
- **I**nterface Segregation - Interfaces should be small and specific. Classes should not be required to implement functions on interfaces it cannot implement. So large interfaces with lots of loosely related methods should be avoided.
- **D**ependency Inversion Principle. Higher level components should not depend on low level components. In practice this often means introducing abstractions and interfaces at the highest level of the program which components can depend on.

2.1 Inversion of Control

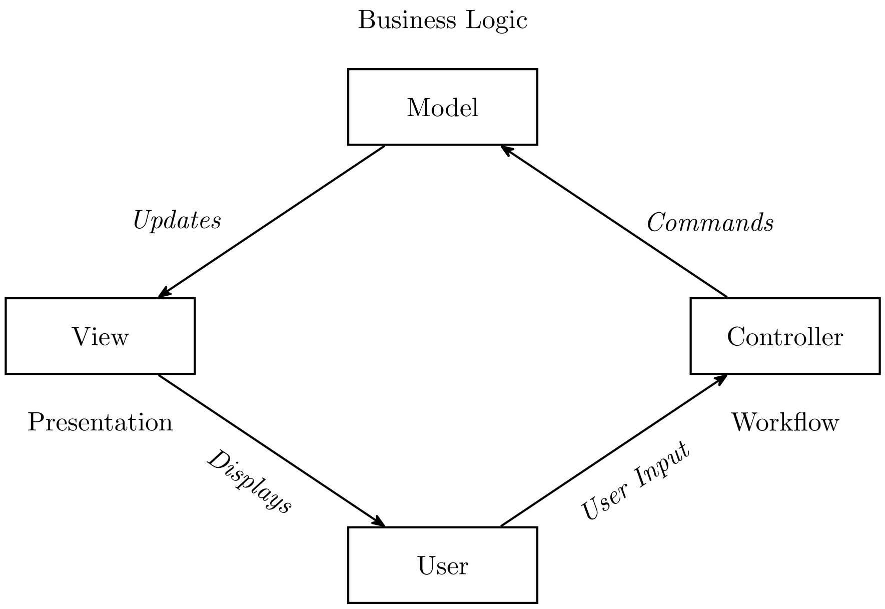
Components should not need to go looking for it's dependencies (i.e. working out how to locate them or instantiating them.) This allows the code in Components to focus on the job they are responsible for.

Instead, the acquistion of resources should be performed by a dedicated part of the program or by an application container, if one is being used (such as ASP.NET). ASP.NET and the Spring framework both use Dependency Injection.

Inversion of Control is also provided by the Service Locator pattern 3.2.6.

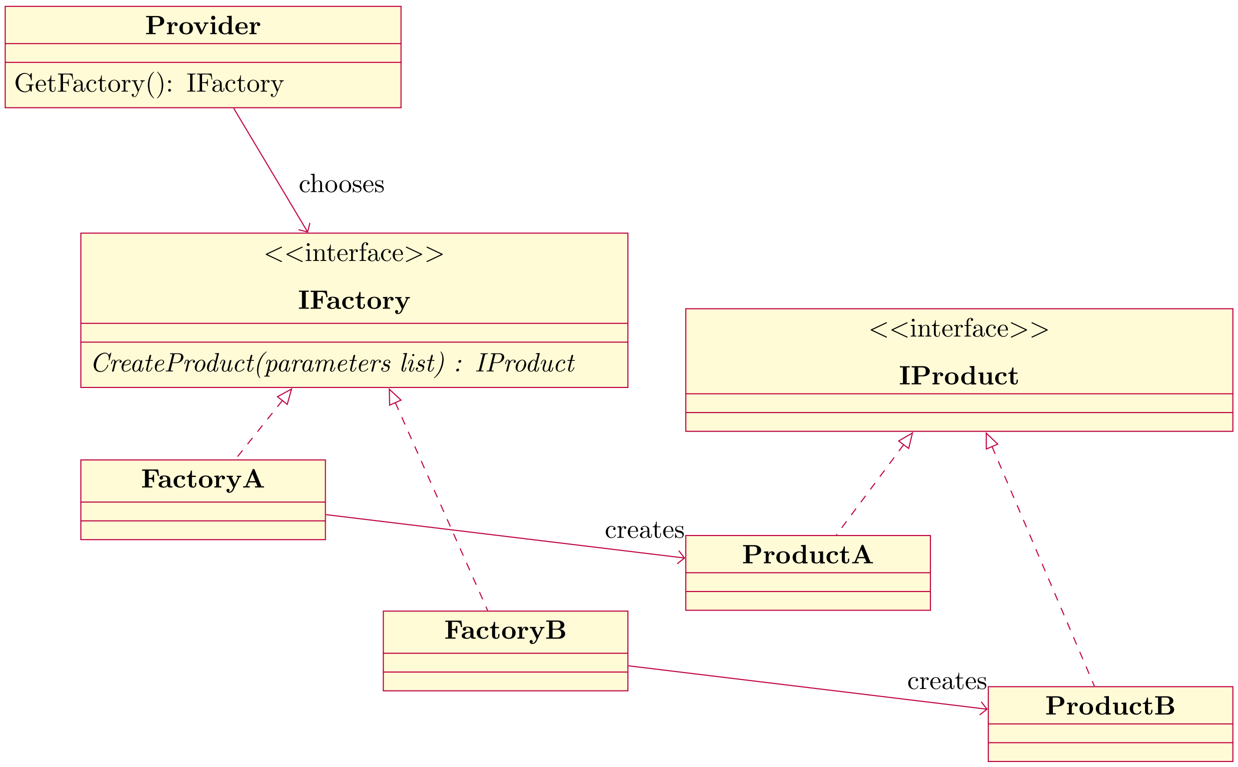
3 Design Patterns

3.1 Model View Controller



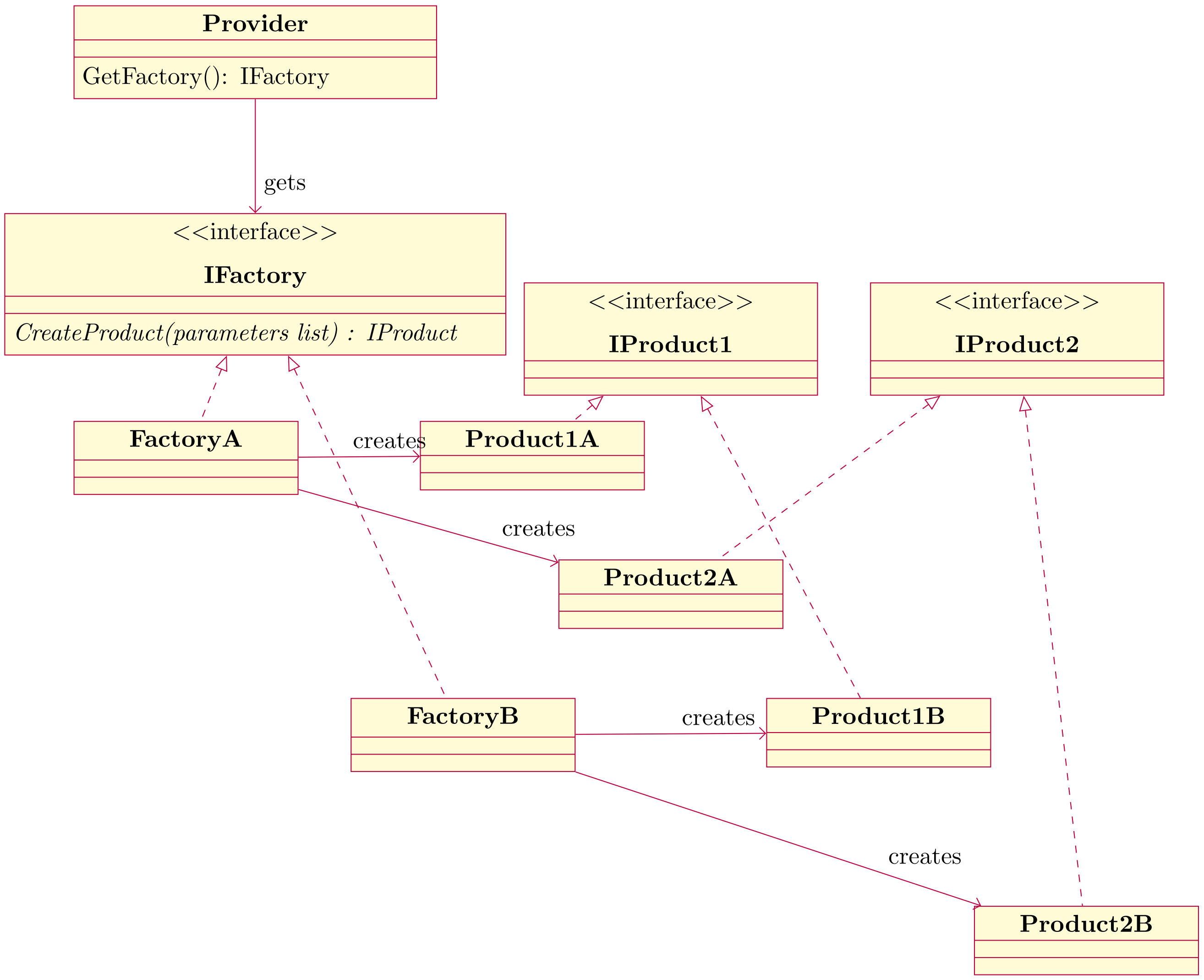
3.2 Creational

3.2.1 Factory



The Provider class chooses which factory is used, and the IFactory implementation contains the code to create the IProduct. This achieves Separation of Concerns as well as Dependency Inversion. The client is only concerned with something that implements the IProduct interface. Examples would include database connections to different database platforms.

3.2.2 Abstract Factory



The Abstract Factory is an extension of the Factory method, where an IFactory instance provides a collection of products which belong together. Essentially the IFactory interface has more than one method so that it can create multiple products in a family. Examples are look-and-feel themes, where UI components might have different shapes, fonts, and colours.

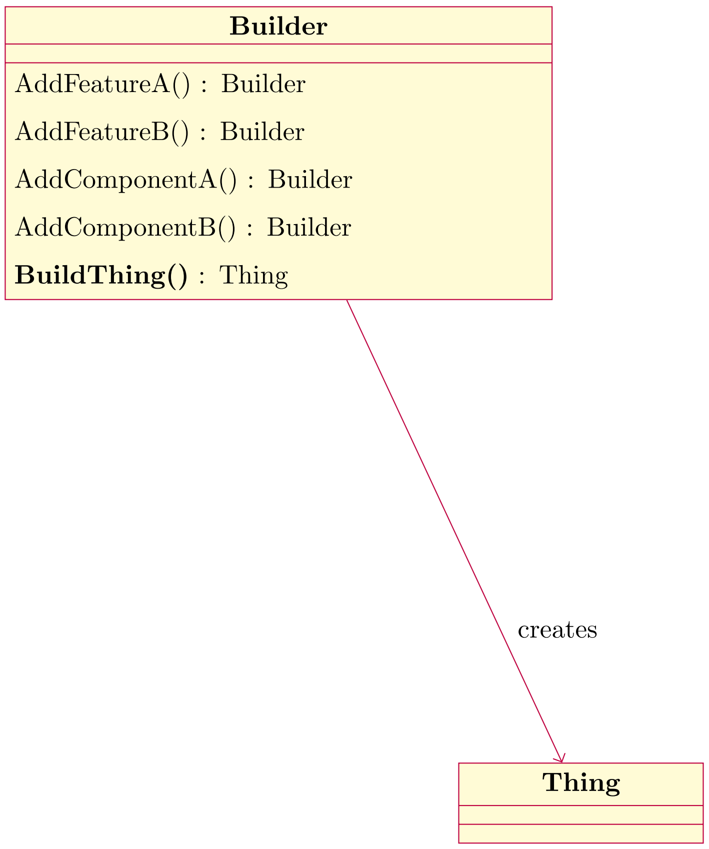
3.2.3 Singleton

The Singleton pattern ensures that there is only one instance of a type within the running program. This is needed to manage resources such as output log files, and factories where it doesn't make sense to have more than one instance.

Declaring a static variable, method or class is an easy way to implement this pattern without using a framework:

```
public static class Singleton {
 private static MyResource resource = null;
 public static MyResource GetResource() {
  lock(...) { // lock the thread
   if (MyResource == null) {
    resource = new MyResource();
   }
  return resource;
  }
 }
}
```

3.2.4 Builder



The Builder pattern allows variations of a type to be built dynamically. The Builder class contains the logic for creating the resource, and provides methods that can be called to add features and components to the final resource. This provides variability in the output resource, and often is more efficient by deferring the final creation of the resource to be end. For example, the StringBuilder class does not append to the final string with each additional string segment, but maintains a list which is then concatenated in one operation at the end.

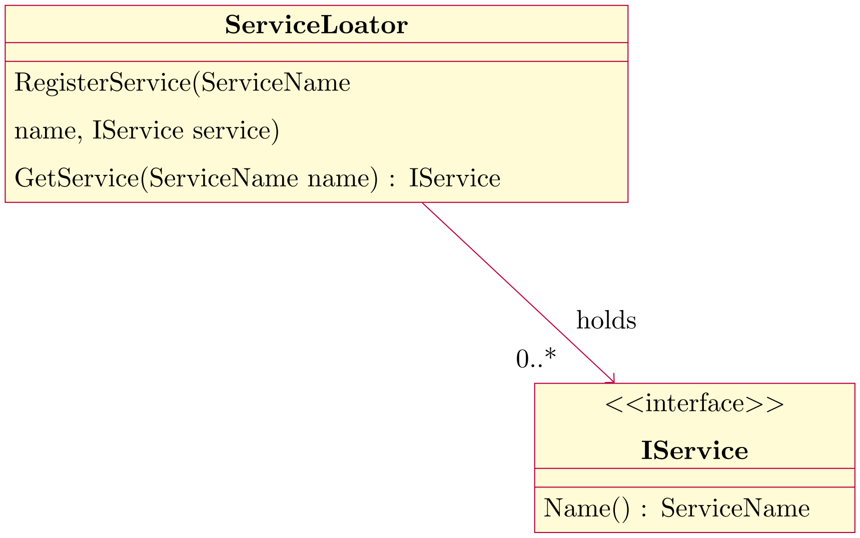
```
StringBuilder builder = new StringBuilder();
builder.append("abcde");
builder.append("fghi");
builder.append("klmno");
String mystring = builder.build();
```

3.2.5 Prototype

The Prototype pattern is used when the creation of an object takes a long time, but that object can then be used in multiple operations. For example, if a long and complex database operation is required to build a result set, and the user might want to then view the result set in different sort orders, or calculate different summaries of it, then a prototype result set can be held in memory and cloned for each re-ordering operation or calculation.

3.2.6 Service Locator

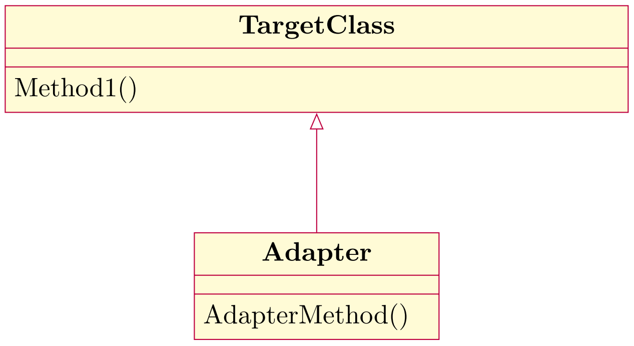
(not an original GoF)


ServiceLoator
RegisterService(ServiceName name, IService service)
GetService(ServiceName name) : IService
holds
0..*
<<interface>>
IService
Name() : ServiceName

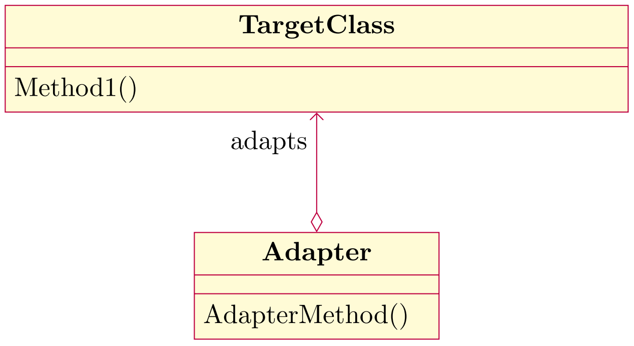
A Service Locator is usually a Singleton class that allows the provider of services to register them by key (e.g. a simple string) and allows the consumer of services to get a reference to them using the same key. A Service Locator might hold services that are for different functions, like a network connection to an email or API server, or it may be for alternatives to the same function like a connection to a regional service.

3.3 Structural

3.3.1 Adapter



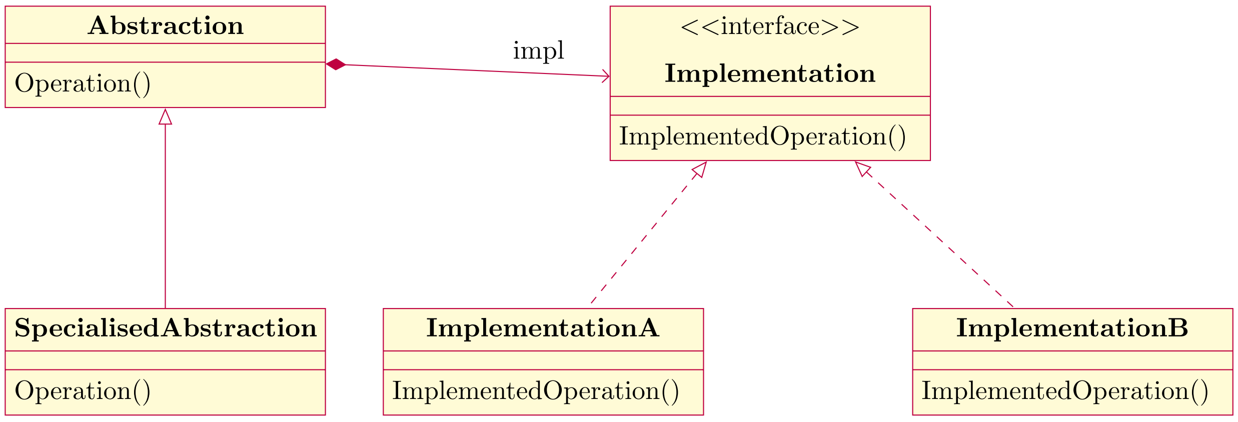
By Inheritance



By Composition

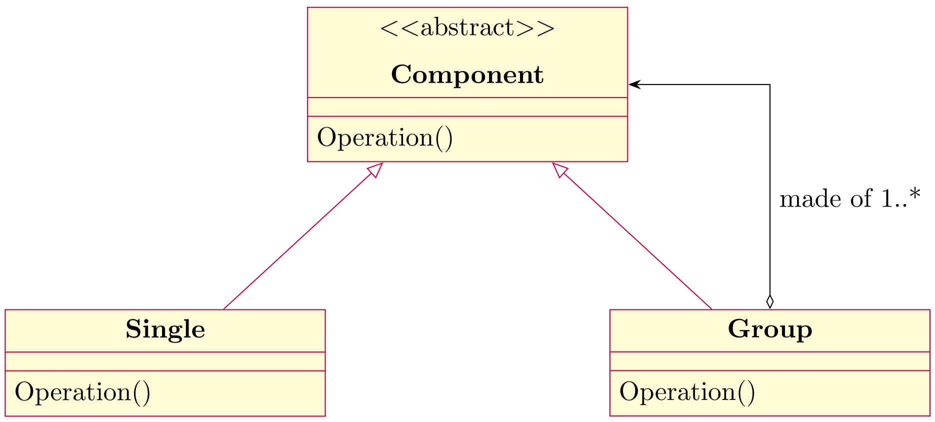
An Adapter is a class that implements methods that a client can call and fowards control to the TargetClass that implements different methods. This pattern is used when the client wants to be able to transparently call different classes but the classes are not compatible with each other.

3.3.2 Bridge



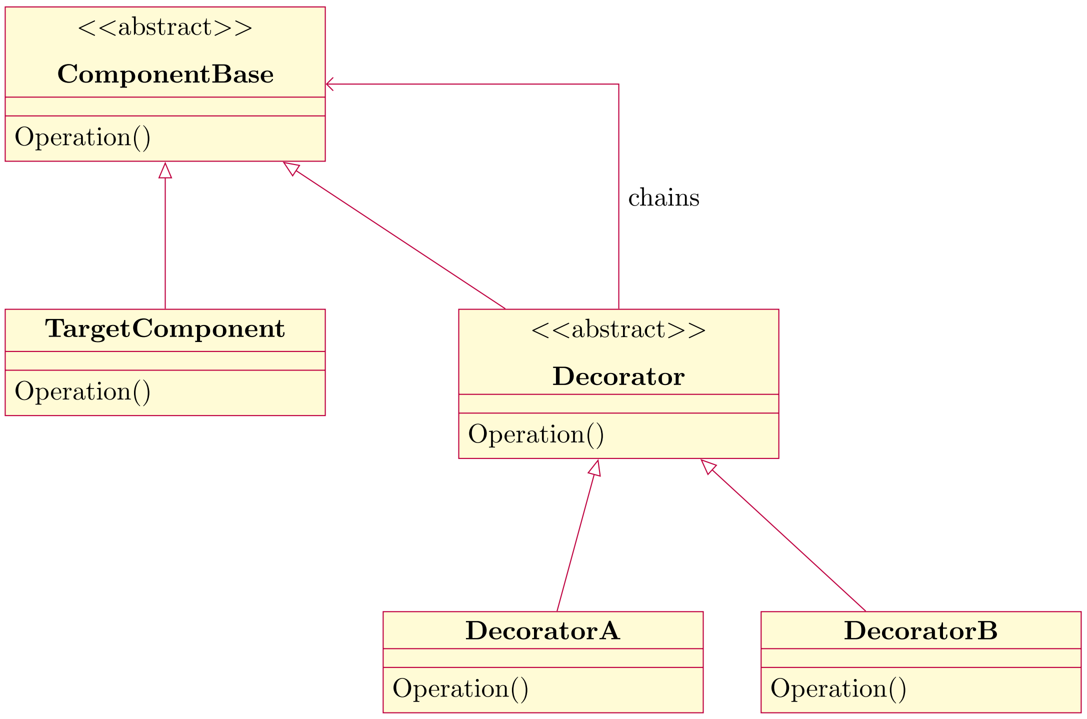
The Bridge pattern allows the separation of the abstraction of some logic from its implementation. The abstraction means the higher level description, i.e. what operations it needs to do (like sorting, displaying, saving), while the implementation is about how it is to do the operation (like quicksort, combobox, zipped file). This structure therefore allows the abstraction to vary independently of the implementation.

3.3.3 Composite



The composite pattern allows single items and groups of items to present the same interface. One use is when building a tree of nodes and some of those nodes might be leaf nodes and some might be group nodes.

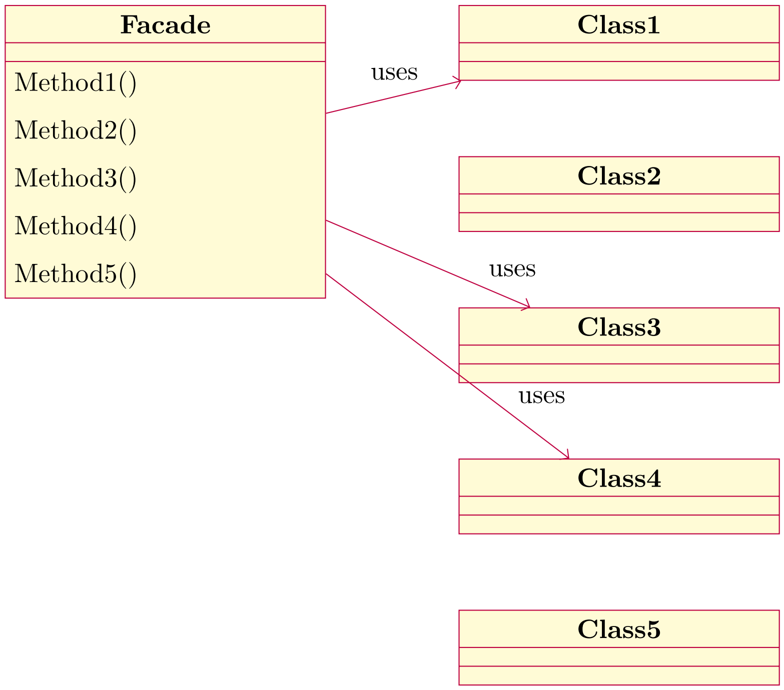
3.3.4 Decorator



The Decorator pattern allows the behaviour of a class to be extended dynamically without needing to subclass it.

Multiple Decorators can be combined into a Chain of Responsibility so that each decorator can modify the behaviour it is responsible for before passing it to the next in the chain. Because they can be combined arbitrarily, N decorator classes effectively do the job of at least 2^N derived classes or ($N!$ if the order of the chain is significant). Because the Decorator class inherits from ComponentBase, the decorated component can be used wherever ComponentBase is used.

3.3.5 Facade

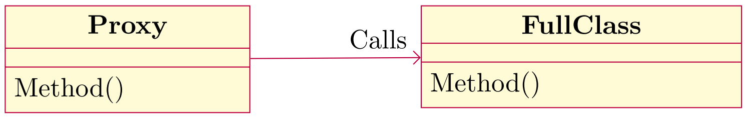


The Facade pattern uses a class to provide a simplified flat API in front of a complex system of classes. Use cases for class models are often not obvious, and the documentation is complex, so the facade allows the idiomatic operations to be supported more simply.

3.3.6 Flyweight

The Flyweight pattern is used when there are distinct objects that share a large amount of common values and are differentiated by a small number of values. For example, different icons may be used to adorn different categories of objects, so instead of duplicating the icon (which might be made of many pixels) for the objects in a category, a FlyweightFactory class ensures that Singletons of icons (the Flyweights) are created and shared as needed. Each object will then contain a shared reference to the Flyweights, in addition it it's own unique values.

3.3.7 Proxy



A Proxy class is similar to an Adapter class but it will present the same interface to the caller. It preserves the semantics of the interface but might modify some runtime characteristics such as deferred loading or execution on demand of expensive operations. Other uses include providing an authentication or authorisation layer. This allows the proxy to efficiently carry out some operations that are needed early such as layout constraints for large images, while deferring the loading of the image until later.

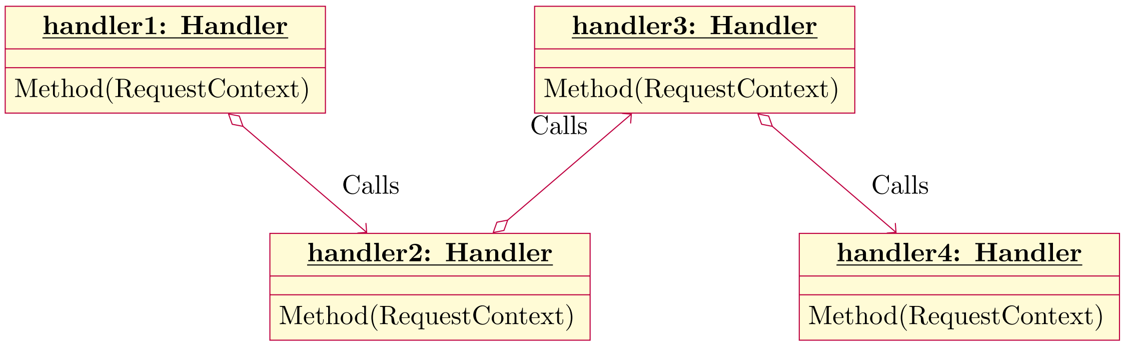
3.3.8 Value Object

(not an original GoF)

This involves the use of immutable data only objects to transmit related values in a structured way. Equality is based on value rather than by reference.

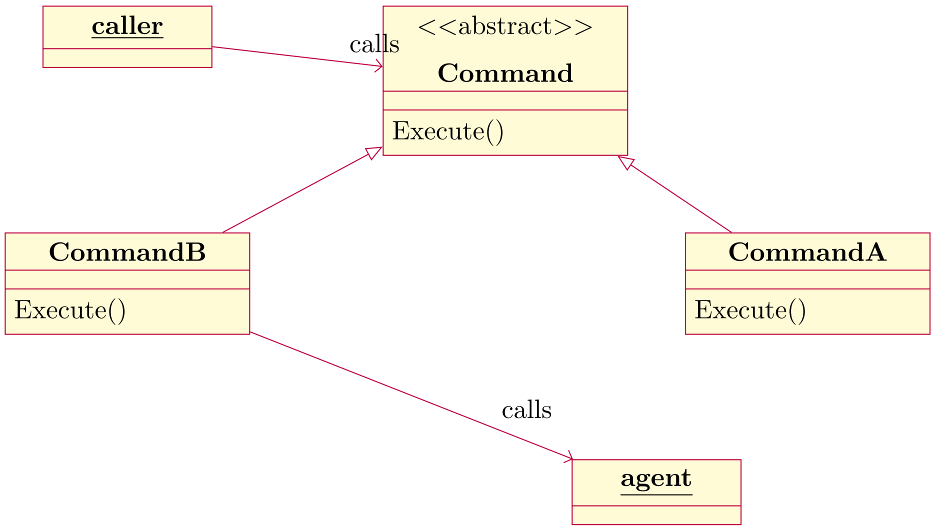
3.4 Behavioural

3.4.1 Chain of Responsibility



The Handlers in a Chain of Responsibility implement the same interfaces or base class methods so that they form a pipeline of chained calls. The RequestContext argument usually contains composite information that each handler can inspect in order to decide whether to take action on it, terminate the chain, optionally add more information to the argument and pass it to the next handler in line. This pattern promotes Separation of Concerns and reduces coupling between the handlers.

3.4.2 Command



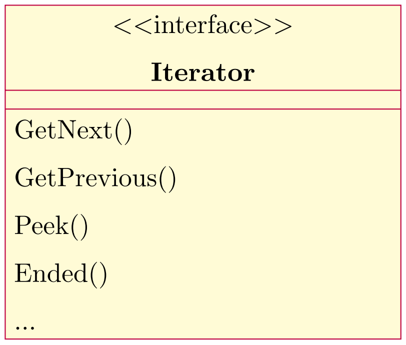
A Command class abstracts the command call from it's execution, and decouples the caller making the request from the system that actually performs it. The command is implemented as a subclass or contains a class that has knowledge of how to perform a specific request. This pattern allows generic user interface objects like links and buttons to trigger executions which perform business operations.

Having the Command object as a standalone object allows the command sequence to be recorded to implement playback and undo sequences (by adding an Undo() method)

3.4.3 Interpreter

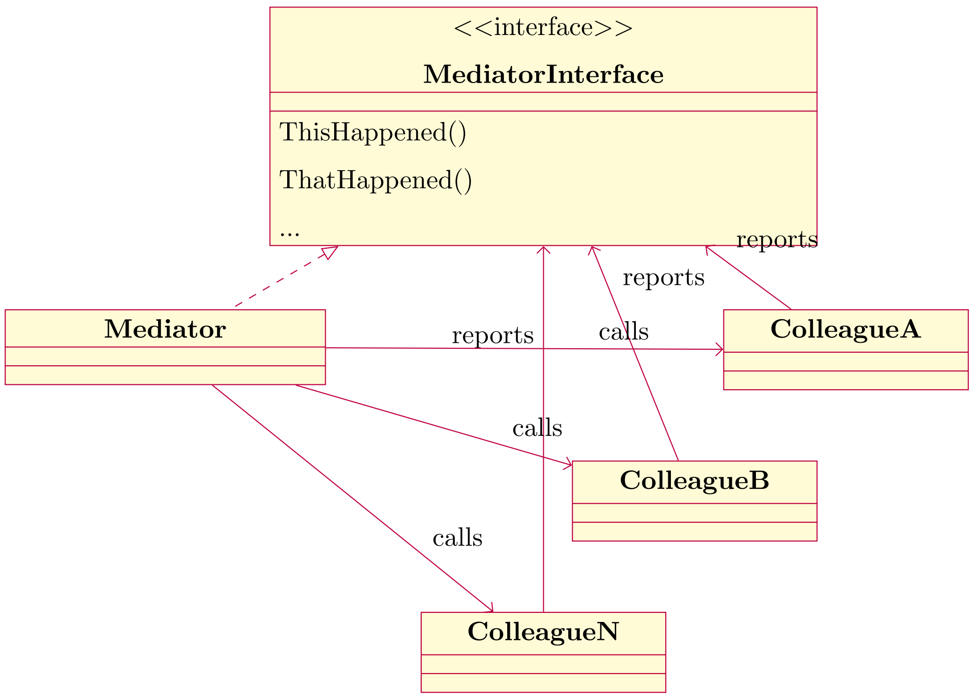
More of a genre of patterns than a single pattern. The idea is to create a grammar in which instructions can be constructed and interpreted by the program. It crops up in many areas, such as UI's which offer a recording function. It is also required where runtime configurations are too variable to be supported by statically defined types.

3.4.4 Iterator



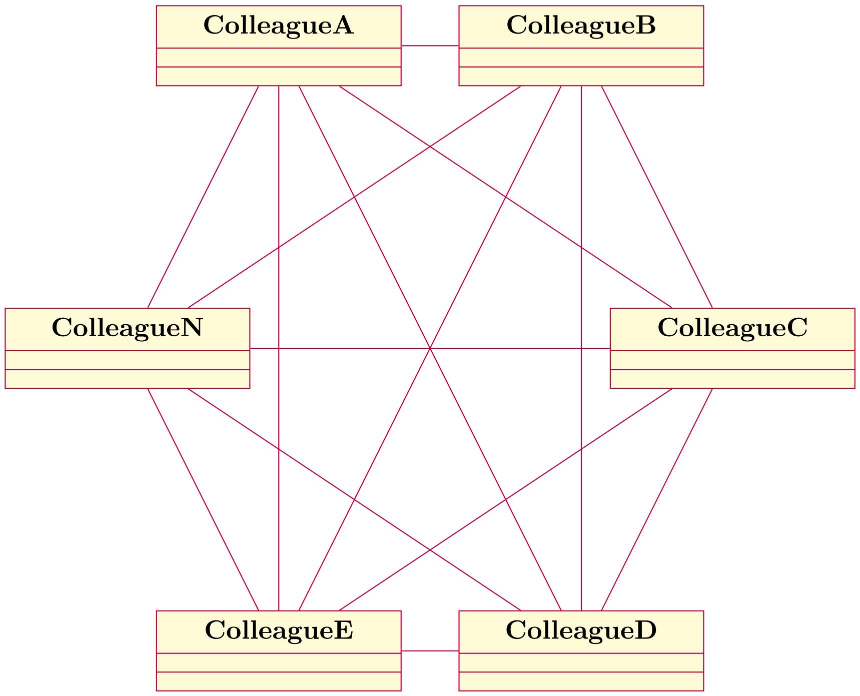
This is a very common pattern. It separates the job of maintaining a collection of objects from the job of how to visit them one at a time. The methods offered in an iterator vary. Some iterators provide for modification of the element at the current position. Some iterators allow backwards movement. It's all part of the iterator pattern. Iterators are also offered as templates (generics) since it is a pattern that can be applied on general types.

3.4.5 Mediator



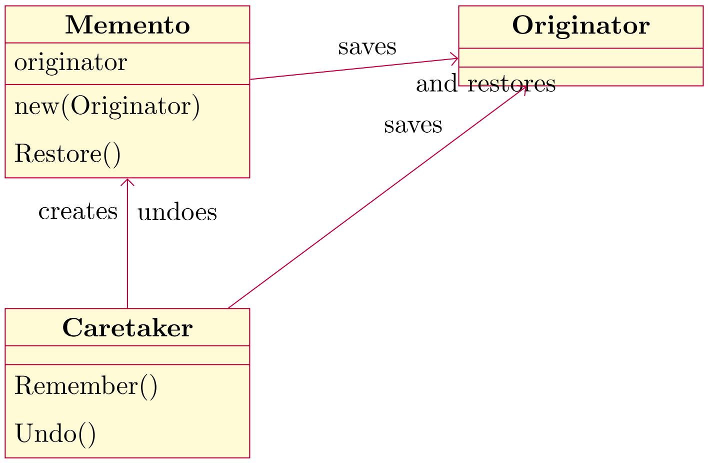
When a system of cooperating classes (Colleagues) has complex mutual interactions, the Mediator pattern allows these interactions to be encapsulated within the Mediator class. Often this will avoid the need to duplicate logic in the colleagues. Importantly, the Mediator becomes the only source that defines the interaction between the Colleagues. The Colleagues are only responsible for informing the Mediator that something has happened that requires attention.

The anti-pattern (see figure below) potentially involves every Colleague needing to determine some of the behaviour of every other Colleague. The complexity of these interactions scales as $N(N-1)$ whereas the complexity of the Mediator pattern will scale as $2N$.



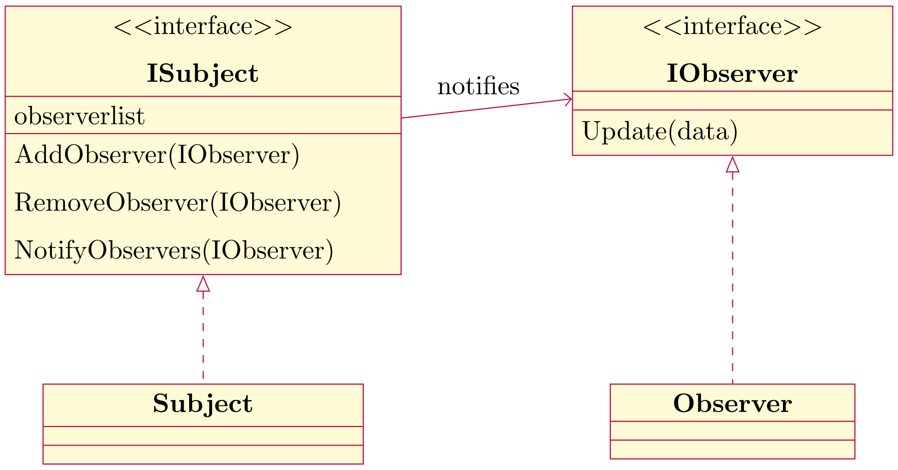
The anti-pattern

3.4.6 Memento



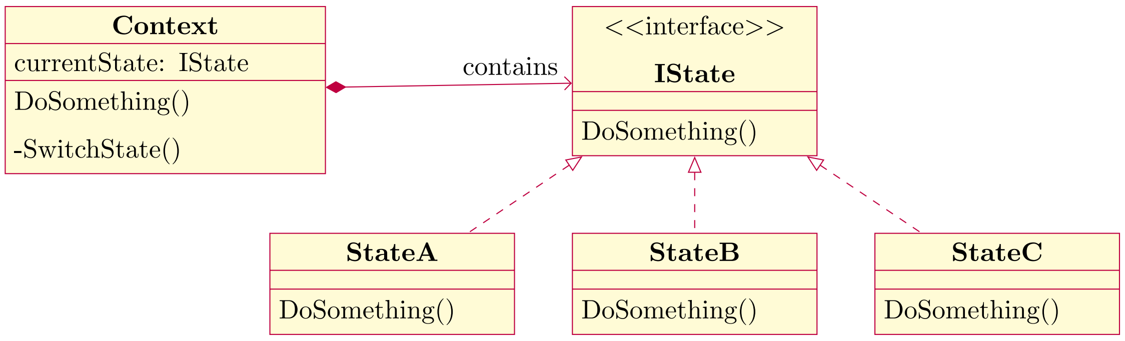
The Memento pattern supports the Undo operation by saving the state of an object (the Originator) before it is changed in some way. The Memento class holds this state, along with a reference to the Originator. This can range from holding a clone to directly saving the relevant internal fields of the Originator. In languages like C# this might be a use case for the visibility modifiers that allow classes in the same package to access fields without being a derived class. The Caretaker acts as a Mediator between these classes and is responsible for recording the original states and triggering their restoration. In practice there will be multiple Originator types, and therefore Memento types, which the Caretaker will need to manage.

3.4.7 Observer



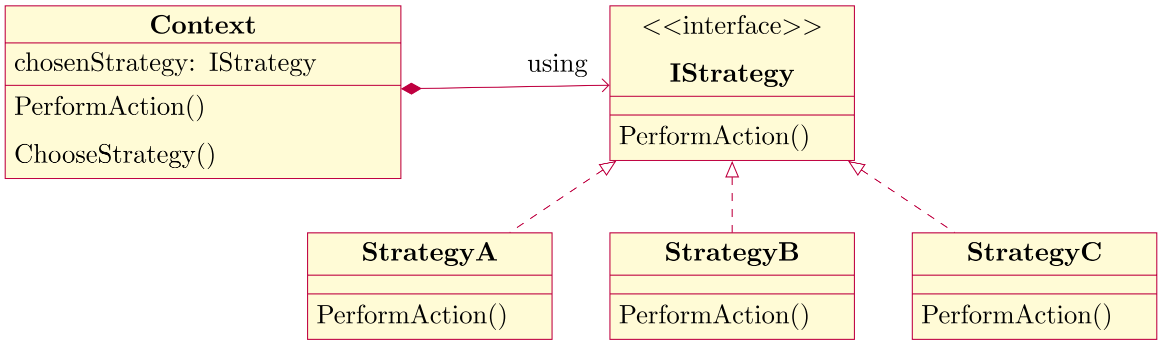
The Observer pattern decouples a class (Subject) that contains data that needs to be monitored from the classes (Observer) that are interested in monitoring it. This is an essential pattern in the common cases where it is convenient to report that some data has changed, but it is not possible to predict what Observers might be interested in the Subject. Often the Subject is in a library package which has no way to predict what code will be using the library. The Observer pattern allows multiple Observers to be attached to a subject so that when an update occurs they are all notified.

3.4.8 State



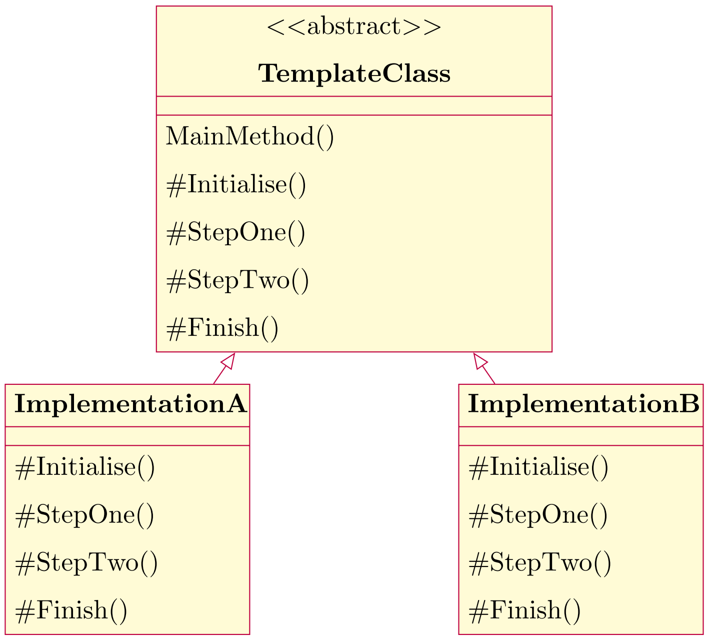
The State pattern allows a class (the Context) to change its internal behaviour while presenting the same interface to external clients. The states can represent current settings in the program, or reflect the stage of an operation.

3.4.9 Strategy



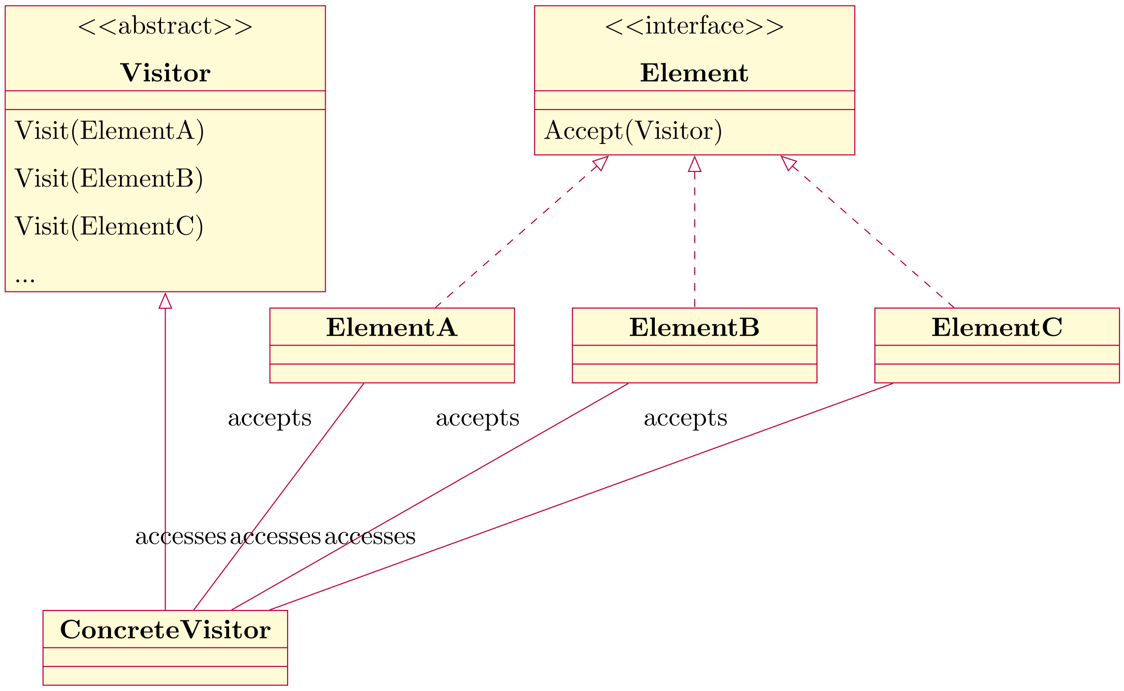
The Strategy pattern has the same basic structure as the State pattern, but the Strategy is intended to implement a variety of algorithms that achieve the same outcome (whereas in the State pattern, each state implements different behaviours that achieve different outcomes). For example, in an equation solving application, a Strategy might be one of the many known techniques that solve an equation using different mathematical methods. The other difference is that the Strategy is commonly chosen by the client, whereas in the State pattern, there is a "state machine" that dictates the rules which determine the states to visit.

3.4.10 Template



The Template pattern is used when there is a "recipe" for an algorithm that is described in terms of a sequence of abstract steps, but where the detailed implementation of each step can be done in different ways. This is basically what happens when a Template/Generic is implemented in those languages which support them. The code inside the Template/Generic is written at an abstract level and calls the methods provided by the implementing class to do the concrete work. The TemplateClass can also provide full default implementations of methods where it is useful.

3.4.11 Visitor



The Visitor pattern is used when an operation needs to be carried out differently on different types of Element, and it is not appropriate to implement the operation inside the Element class, because the implementation of the operation may vary depending on the ConcreteVisitor. So the ConcreteVisitor contains the various implementations within its Visit(ElementX) method corresponding to each type. The technique is a double dispatch of methods, because the client initiates processing by calling the Element.Accept(Visitor) method passing in the Visitor, and the Element then calls the appropriate overload of the Visitor.Visit(ElementX) method on Visitor. In a quant finance application this pattern might be used to allow a variety of different financial products (the Elements) to be priced by a variety of pricing models (the Visitors)