Design Patterns

A *design pattern* is a solution to a common design problem. Design patterns in object-oriented languages take the form of fixed class structures or relationships. If you are familiar with common design patterns in your programming space or problem domain, you can apply these patterns to appropriate situations to improve your design, address specific implementation objectives, and generally reduce the effort involved in completing your project.

The original Design Patterns text was published in 1994, but much of its content still remains relevant today. Much like refactorings, there is no single authorative list of design patterns but there are several which are common and general purpose which you should know about. For reference, you might refer to Head First Design Patterns (one of the recommended readings for the course, termed HFDP for the remainder of this page) or to the Wikipedia page on the topic.

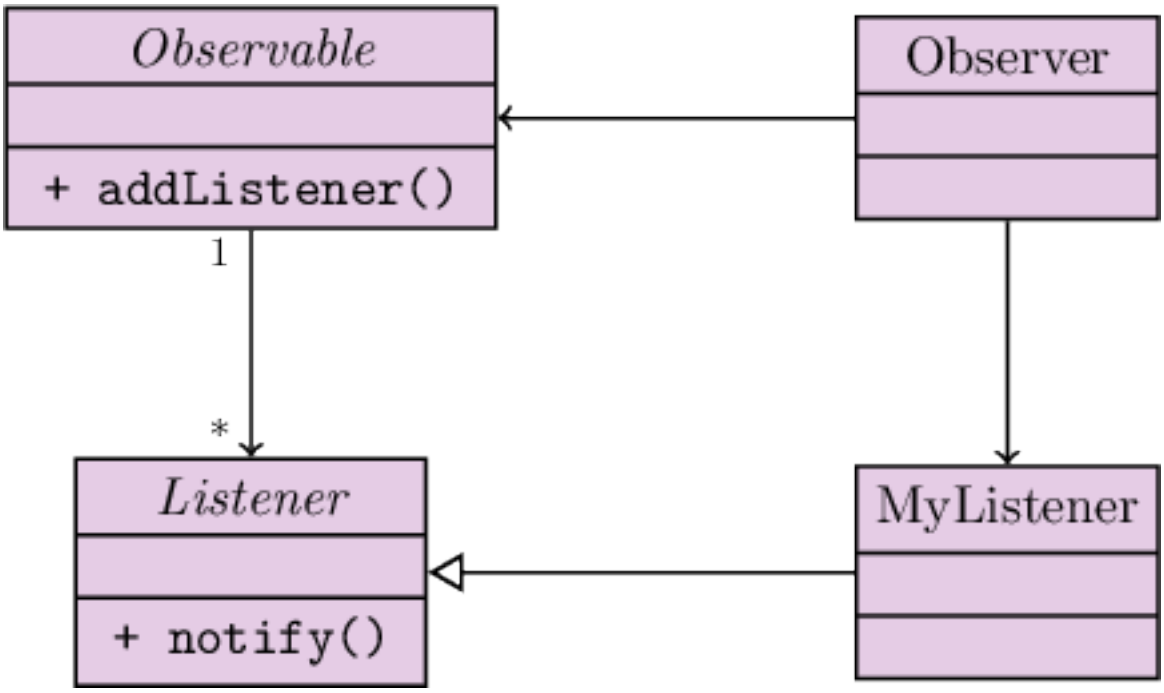
A common theme among design patterns is *separation of concerns*: an elegant structure is used to prevent code from relying on or interacting with other code as much as possible. This helps to maintain the design principles we have discussed and keep code manageable and receptive to change in the future.

This is a short reference guide for design patterns discussed in class.

- Familiar Patterns
    - Observer
    - Iterator
- Functional Patterns
    - Strategy
    - Command
    - Abstract Factory
- Wrapper Patterns
    - Decorator
    - Adapter
    - Proxy
- Other Behavioral Patterns
    - State
    - Template Method
    - Visitor
- Other Structural Patterns
    - Composite
    - Façade

Familiar Patterns

Observer



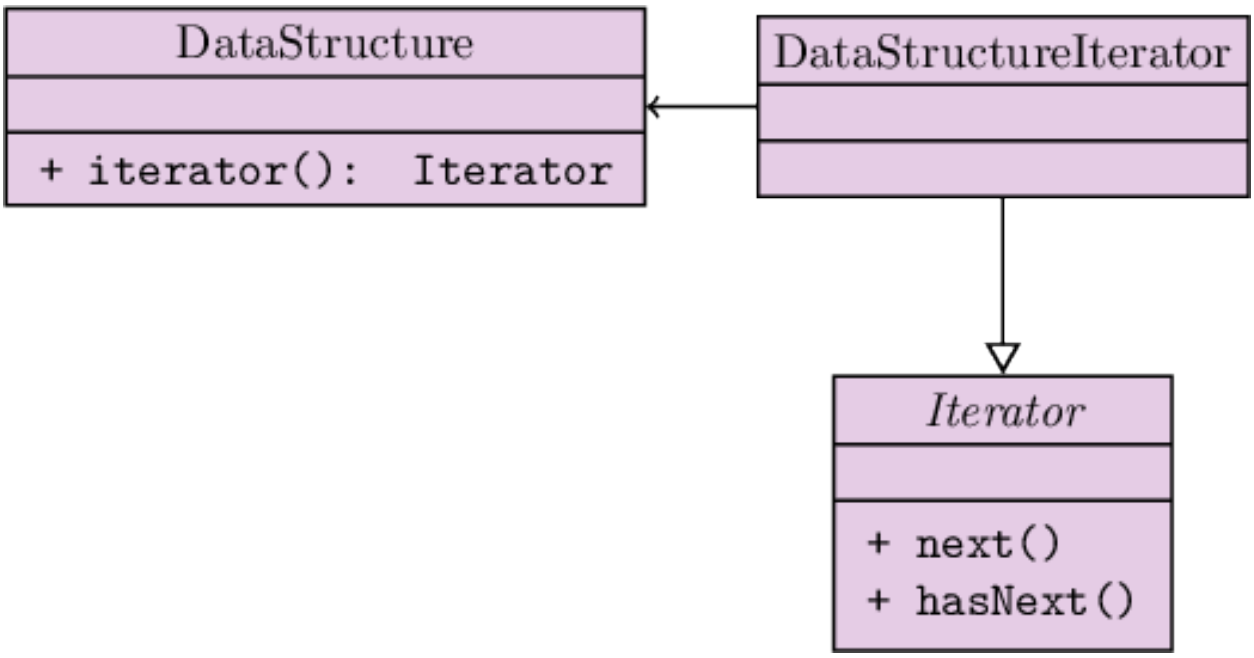
(HFDP Chapter 2)

The Observer Pattern should already be quite familiar: we've seen an example of Observer in the event handlers used in JavaFX to respond to mouse clicks as well as the listeners we add to a model to support MVC. In this pattern, an *observable* object allows listeners (sometimes called handlers or callbacks) to be added to it, ensuring that they will be notified in the future of important events.

The advantage of the Observer Pattern is that it decouples the observable object from the objects which need to receive notifications. In an MVC design, for instance, this allows the model to be implemented without any information about the view or the controller. This form of decoupling is often why the Observer Pattern is found in GUI libraries: to allow application logic to be separated from presentation.

Instances of the Observer Pattern may also be called Publish/Subscribe ("pub-sub") systems.

Iterator



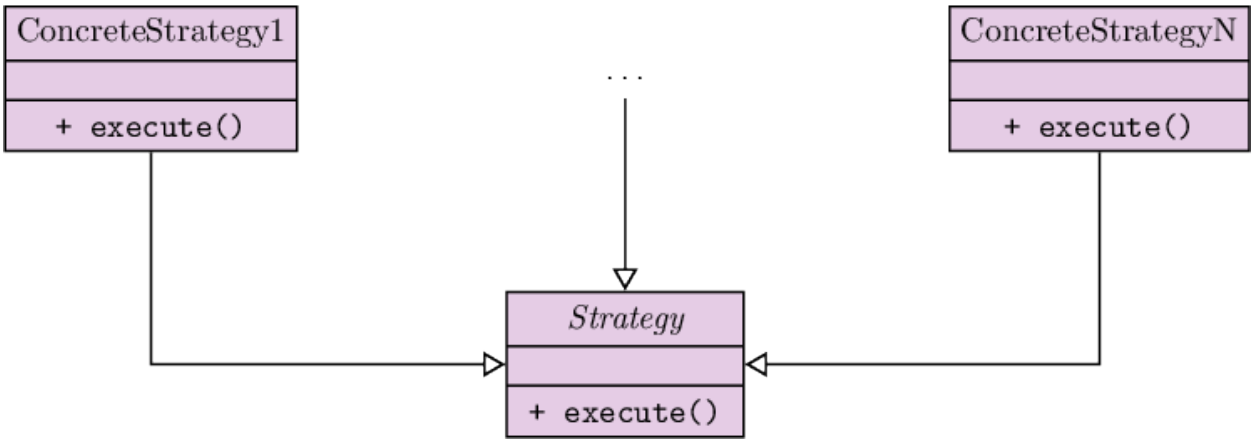
(HFDP Chapter 9)

The Iterator Pattern allows sequential access to a data structure through an intermediate object without needing to know which data structure is being used. This is perhaps one of the most basic uses of a generic data structure – to perform a "for-each" on its contents – and an algorithm using an `Iterator` can be used with any data structure, regardless of how it iterates over its elements. In Java, any data structure which can produce an `Iterator` should implement the `Iterable` interface. Any `Iterable` object is usable in a Java for-each loop (i.e. `for (int i : myList) { ... }` calls `iterator()` on `myList`).

This is another example of separation of concerns: the algorithm need not be concerned with the data structure itself but only its contents.

Functional Patterns

Strategy

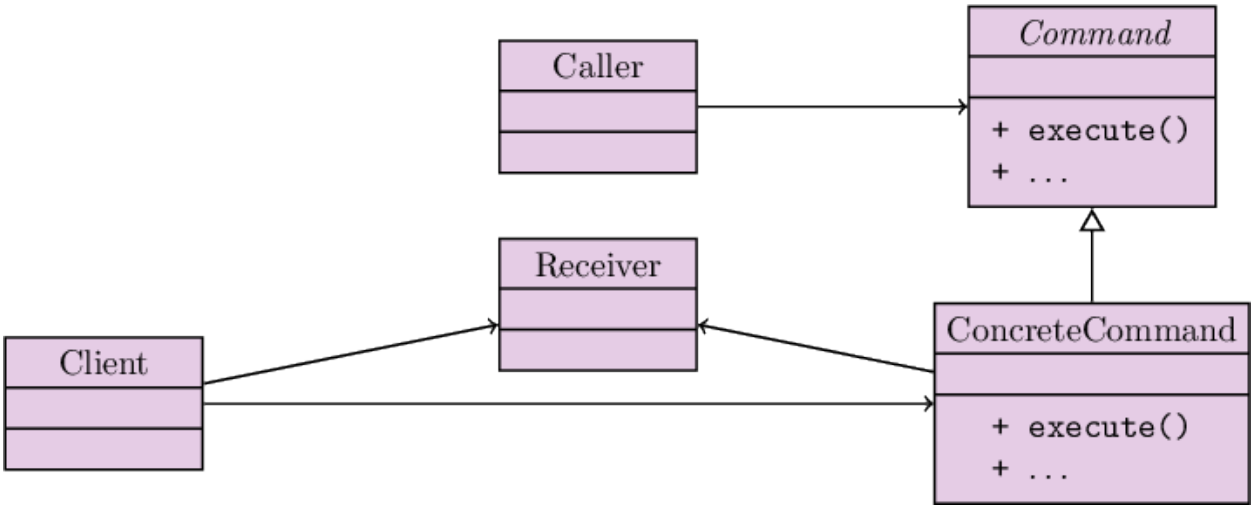


(HFDP Chapter 1)

The Strategy Pattern is used to define a family of algorithms and make them interchangable. A single interface defines the algorithmic family and each implementation defines one of the algorithms. Using this approach, a different class may hold a reference to a single `Strategy` object and make use of it without knowing which algorithm it implements. This essentially makes the algorithm "pluggable".

Programmers familiar with functional languages (such as OCaml or Haskell) or languages with functional features (such as Python `lambda`s or Java's first-class functions) might recognize the `Strategy` pattern as a class-based encoding of higher-order functions. In Java, it isn't actually possible to pass functions around as values, but we *can* pass objects around and objects may have a single method and no fields, making them equivalent to functions. Before higher-order functions were added to Java 1.8, the Strategy Pattern made it possible to implement algorithms in a functional style without first-class functions.

Command



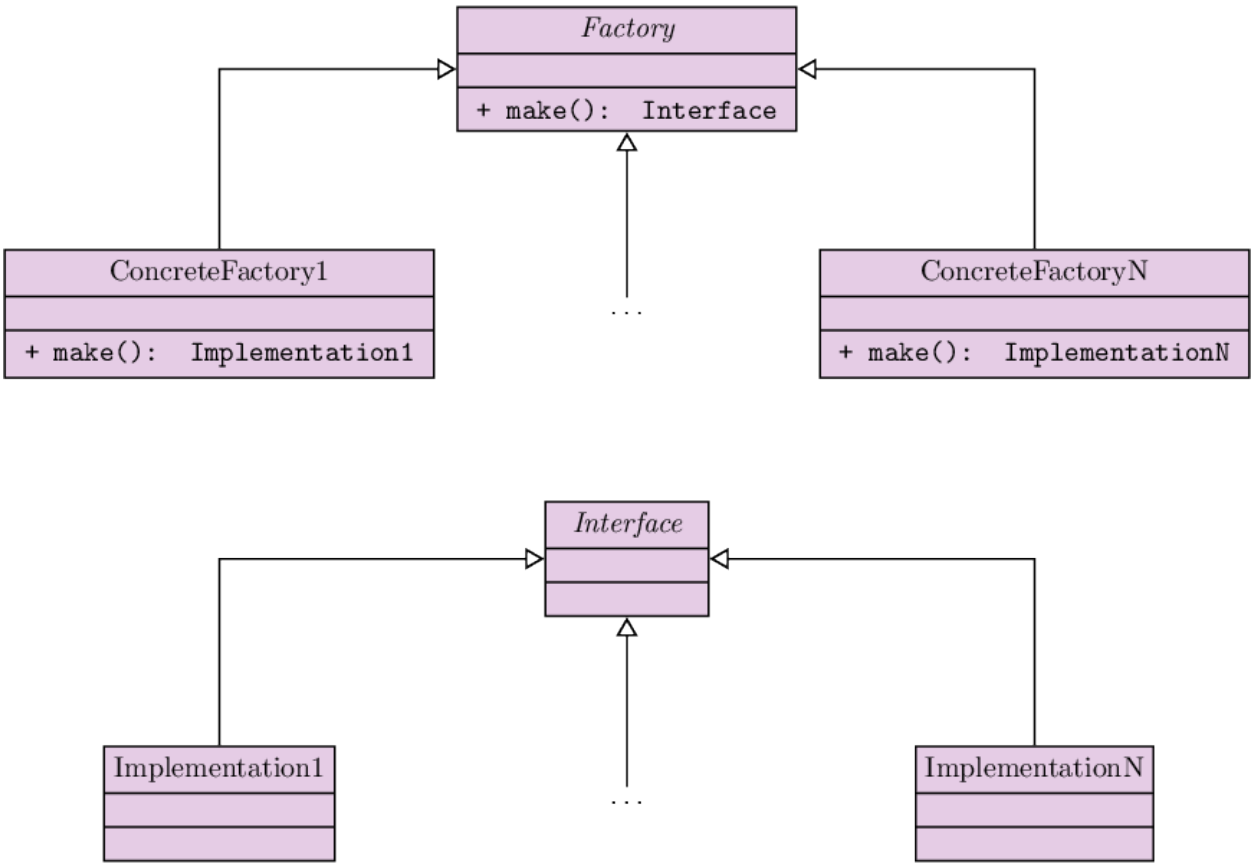
(HFDP Chapter 6)

The Command Pattern captures work in the form of a *command object*, which is then passed to another entity in the system to be executed generically. This pattern is useful when the operations should be logged, prioritized, undoable, etc. These additional operations are possible because the work has been represented as an object and the object's class can include extra methods (e.g. `undo()`).

The Command Pattern is commonly used to describe undo operations in desktop software as well as multi-device systems such as computational clusters. This is also an example of separation of concerns: the `Caller` is unaware of which `Command` is running and is simply responsible for running the commands.

Command Pattern is similar to Strategy Pattern in that both patterns represent work as a single object. Command Pattern differs in that additional metadata or operations are associated with the work. While a `Strategy` simply represents the operation a `Command` is a specialized case that may have a priority, undo functionality, or other similar behavior.

Abstract Factory

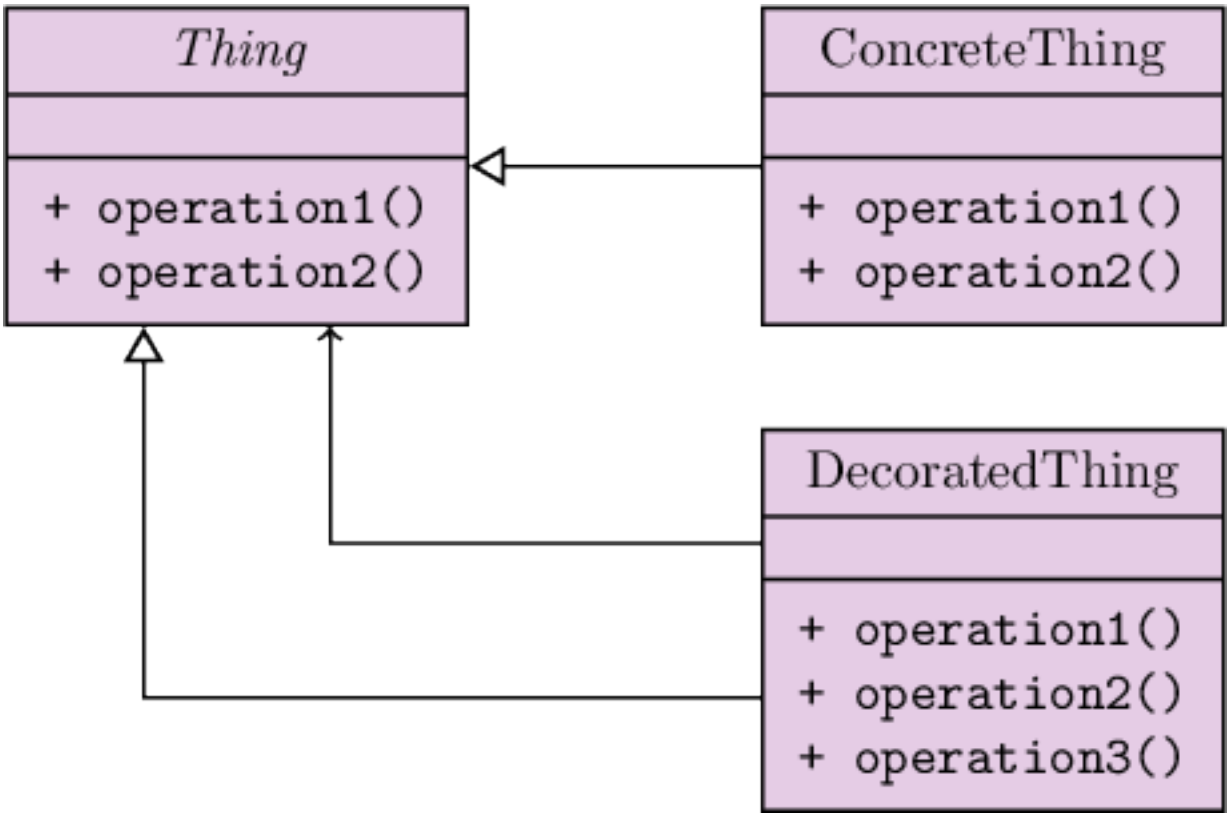


In the Abstract Factory Pattern, a single `Factory` interface defines a routine which will create an object of a given `Interface` type. Implementations of the `Factory` create different implementations of that `Interface`. Algorithms may use a `Factory` object without being aware of which specific implementation is being created.

The Abstract Factory Pattern is quite similar to the Strategy Pattern. Where the Strategy Pattern encodes the passing of *functions* using objects, the Abstract Factory Pattern encodes the passing of *constructors* using objects.

Wrapper Patterns

Decorator



(HFDP Chapter 3)

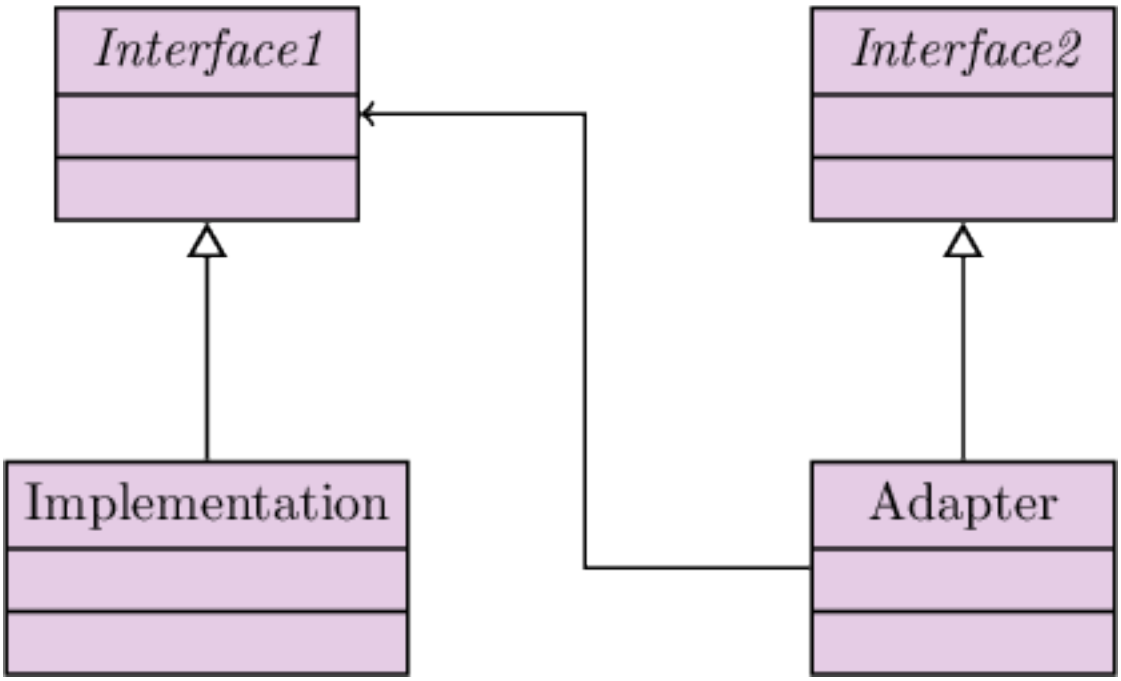
The Decorator Pattern is used to add extra behavior to an existing implementation of an interface. Here, the `Thing` interface is capable of performing some set of operations and `ConcreteThing` is an implementation of them. A user of this code may wish to add functionality to the object without changing its class definition. To do so, that user creates a `DecoratedThing` class. The `DecoratedThing` implements the `Thing` interface by using some other `Thing` object (calling that object's methods whenever its own methods are called). It *also* provides an additional behavior (here, `operation3`), effectively extending the concrete implementation without modifying its definition.

An example of this can be seen in the Java IO libraries. The `InputStream` abstract class takes on the role of `Thing` here: there are many possible `InputStream` implementations, such as `FileInputStream`. Examples of decorators include `DataInputStream` (which allows values other than bytes, such as `int`s, to be read from the stream) and `LineNumberInputStream` (which

keeps track of the current line number). In both cases, the decorators do not change the underlying behavior of the `InputStream` that they use, but they add extra behavior that `InputStream` doesn't ordinarily have.

The biggest advantage of the Decorator Pattern is that it allows this functionality to be added generally to any `Thing` (here, `InputStream`) implementation without creating a dedicated subclass. The `LineNumberInputStream`, for instance, can be wrapped around both files and on network sockets to count line numbers on either one. This is an illustration of the Open-Closed Principle: the behavior of `InputStream` is being extended without changing `InputStream`'s definition.

Adapter



(HFDP Chapter 7)

The Adapter Pattern is used to allow a class to appear to implement an interface when it actually does not. Here, the `Implementation` class implements `Interface1` but not `Interface2`. The `Adapter` class takes an object of the `Interface1` type and implements `Interface2` with it. This allows an object of type `Interface1` to seem to implement `Interface2` when it is wrapped inside of the `Adapter`.

The Adapter Pattern is most useful when we need a class to implement a particular interface but we're unable to change the class definition. The most straightforward example comes from the Java API itself, which includes an `Enumeration` interface (which is distinct from the `enum` keyword!). As of Java 1.2, the `Enumeration` interface was replaced by the `Iterator` interface, but plenty of old code existed which used the `Enumeration` interface and so it couldn't be removed from the standard libraries.

The following code is an adapter between those two interfaces:

```
public class EnumerationToIteratorAdapter<T> implements Iterator<T> {
    private Enumeration<T> enumeration;
    public EnumerationToIteratorAdapter(Enumeration<T> enumeration) {
        this.enumeration = enumeration;
    }
    public boolean hasNext() {
        return this.enumeration.hasMoreElements();
    }
    public boolean next() {
        return this.enumeration.nextElement();
    }
}
```

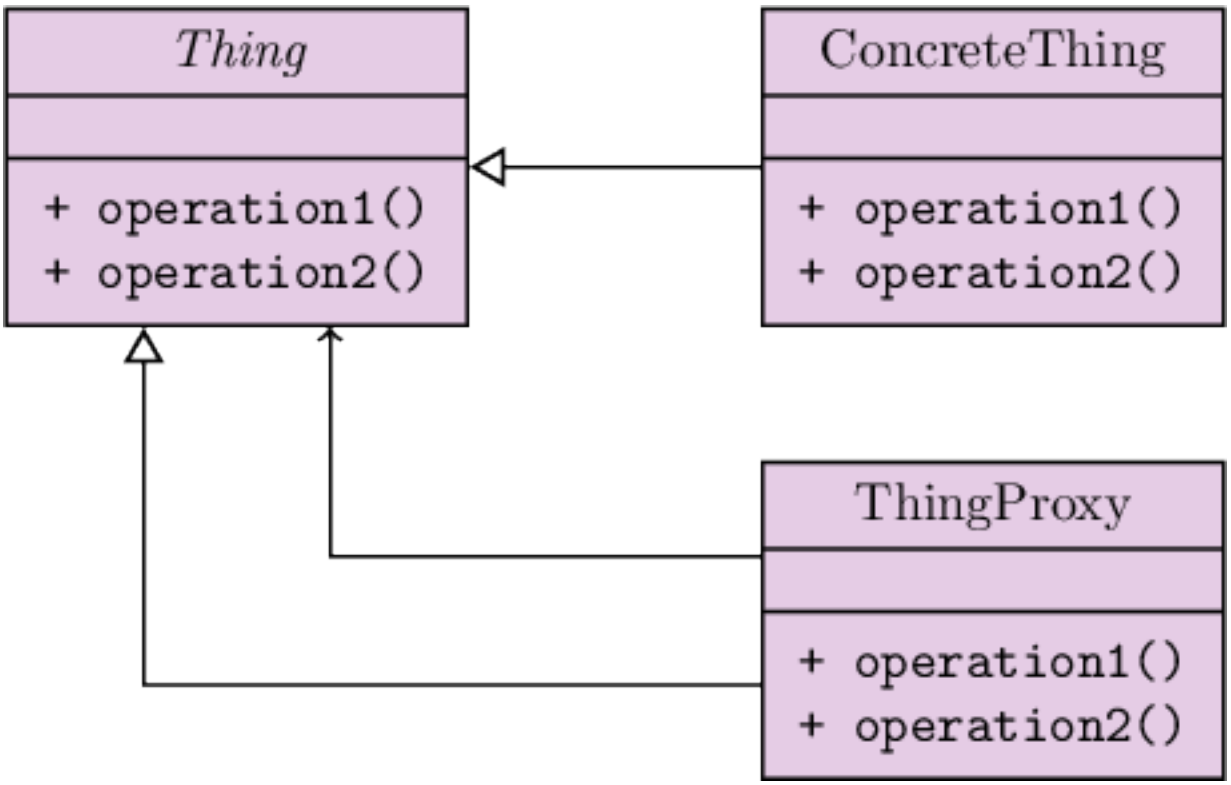
If we were writing code using a pre-1.2 library that returned an `Enumeration` and we needed to pass the result to a library that expected an `Iterator`, we could do the following:

```
Enumeration<Integer> enumeration = oldLibraryCall();
Iterator<Integer> iterator = new EnumerationToIteratorAdapter<>(enumeration);
newLibraryCall(iterator);
```

The Adapter Pattern is intuitive because there are a number of real-life analogies. Power adapters, for instance, allow power from one interface (e.g. 110V AC) to be sent to a device expecting another interface (e.g. 18V DC). Other cable adapters are similar: Apple computers, for instance, are infamous for their requirement that interface adapters be provided to connect to a variety of hardware.

The Adapter Pattern is similar to the Decorator Pattern in that it wraps around an object. Instead of providing *new* functionality for the wrapped object, however, the Adapter pattern allows *existing* functionality to be used through a new interface.

Proxy



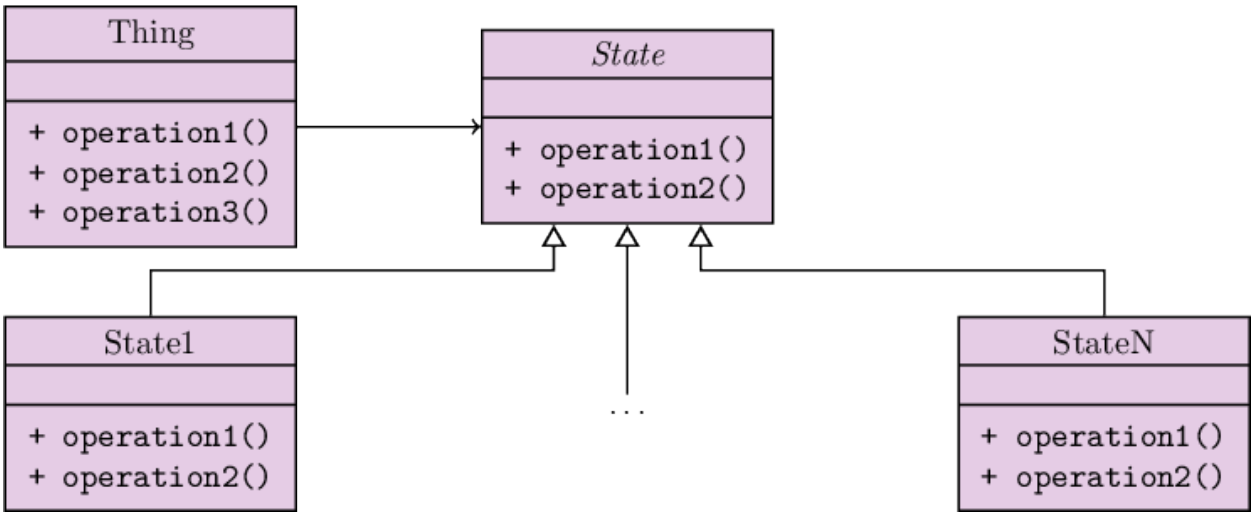
The Proxy Pattern is used to control or monitor access to an object. Much like the Decorator Pattern and the Adapter Pattern, the Proxy Pattern defines a class to wrap around an existing object. Instead of *adding* behavior or changing the *interface* for behavior, the Proxy Pattern *intercepts* the behavior and imposes additional rules.

The Proxy Pattern might be helpful, for instance, in an MVC desktop application. The GUI requires an implementation of the `Model` interface and one could imagine a `ModelImpl` class which provides that. We can also consider a `DefensiveModelProxy` class which wraps around a `Model` object and, in every method, checks to make sure the values returned by the `Model` are valid (e.g. only `null` when appropriate, never returning negative size, etc.). If the `Model` misbehaves, the `DefensiveModelProxy` can take some uniform behavior (such as sending a crash report).

As with the Decorator and Adapter patterns, the Proxy Pattern can be used without modifying the implementation class and without creating a subclass. It can also be reused on different implementations of the interface without duplicating code.

Other Behavioral Patterns

State



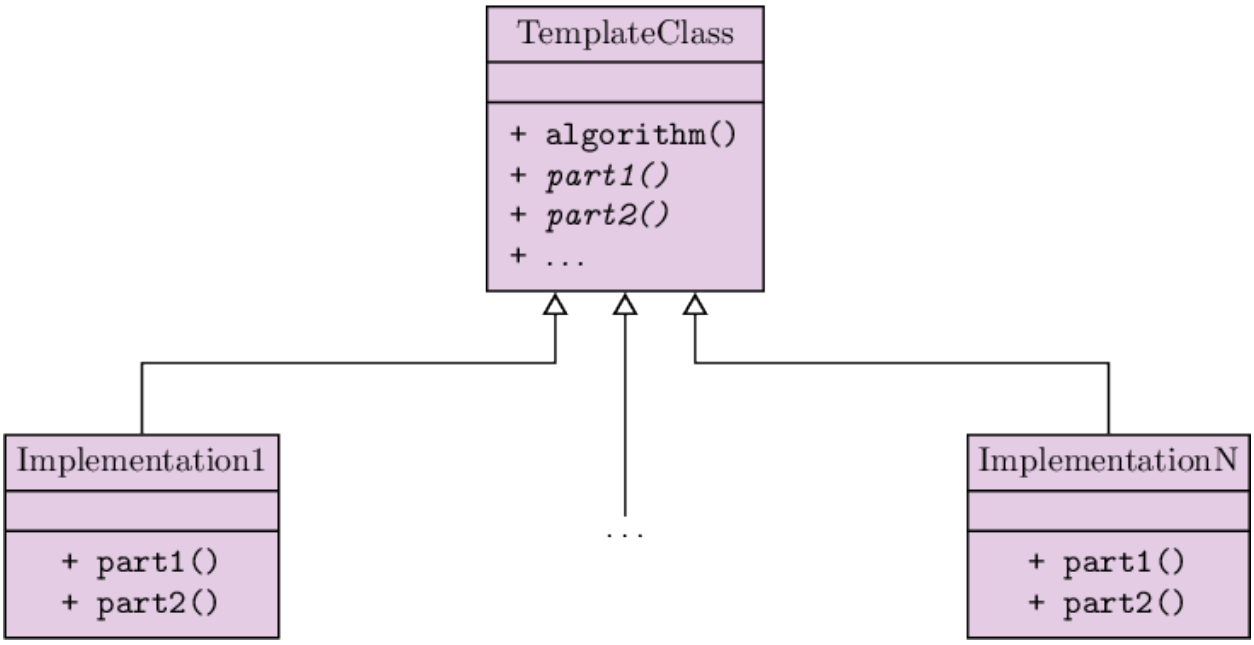
(HFDP Chapter 10)

In Java and most other object-oriented languages, an object cannot change its type after it has been created. This is usually the right behavior, but there are occasional situations in which we need the object's type to *appear* to change; that is, we need the implementation of the object's methods to change. The State Pattern is useful for this.

In the State Pattern, the primary class (here called `Thing`) defers some of its methods' behavior (here, `operation1` and `operation2`) to another object (here, a `State` object). Since the `Thing` has a `State`, we can change the behavior of `operation1` and `operation2` just by changing the `State` to which the `Thing` points. We then have a number of implementations of `State` to represent the different states our `Thing` can be in.

An image displayed in a web browser might motivate the use of the State Pattern. The `Image` class could contain a `State` which dictates the behavior of `getWidth` and `getHeight`. For the `LoadingState`, those methods could reflect a default size. For the `LoadedState`, those methods could reflect the contents of an actual image. For the `ErrorState` (if e.g. the link is broken), those methods could reflect the size of an error icon.

Template Method

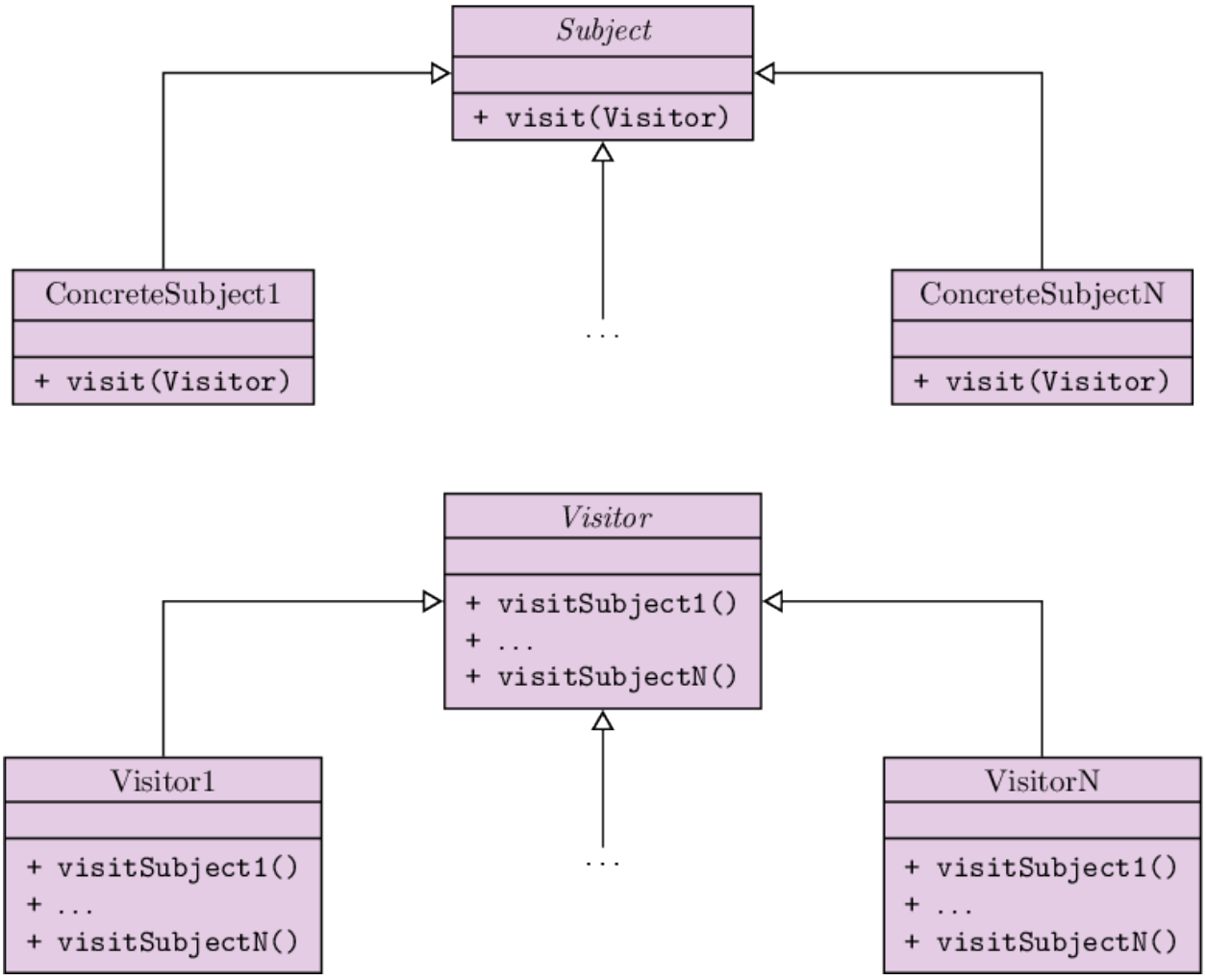


(HFDP Chapter 8)

The Template Method pattern allows a parent class to defer parts of its algorithm to its subclasses. In Java, this is accomplished by adding `abstract` methods to the superclass; this makes the class itself `abstract` and, like an interface, it cannot be instantiated. Non-abstract methods can be added to the class, though, and those methods can make use of the abstract methods.

This pattern tends to occur when the template class captures a particular algorithm. For instance, the template class may represent an operation which performs a web request. The template class could include a `performRequest` method which sets up the connection, sends a message, receives a response, closes the connection, and processes that message into a result. Methods such as `processResponseIntoResult` may be left abstract to be implemented by subclasses, each of which would handle responses in different ways.

Visitor



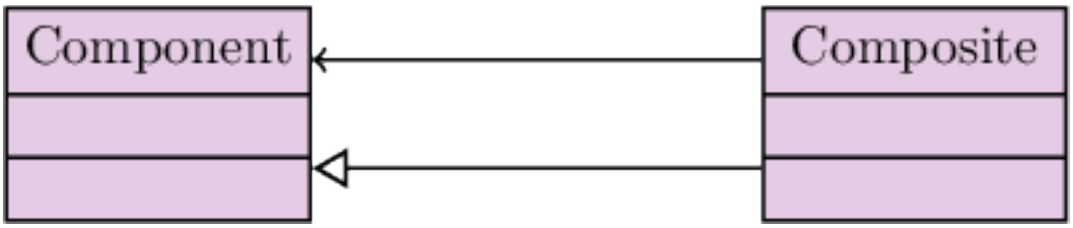
(HFDP Appendix A)

Object-oriented languages have the advantage of openness: new subclasses can be created for existing classes without changing the original classes. This leads to the valuable ability to separate concerns and is one of the fundamental points of the Open-Closed Principle. In some cases, however, we know that a type hierarchy is fixed: there are only so many cases and new ones will not be added.

In this case, the Visitor Pattern can be used to perform case analysis on this fixed set to give us additional separation and guarantee that we consider each of these cases correctly. In a sense, the Visitor Pattern requires that we enumerate all of the subclasses of a given class and, in return, allows us to guarantee that we've considered each of the cases and add new operations to the type hierarchy without modifying the underlying classes.

Other Structural Patterns

Composite

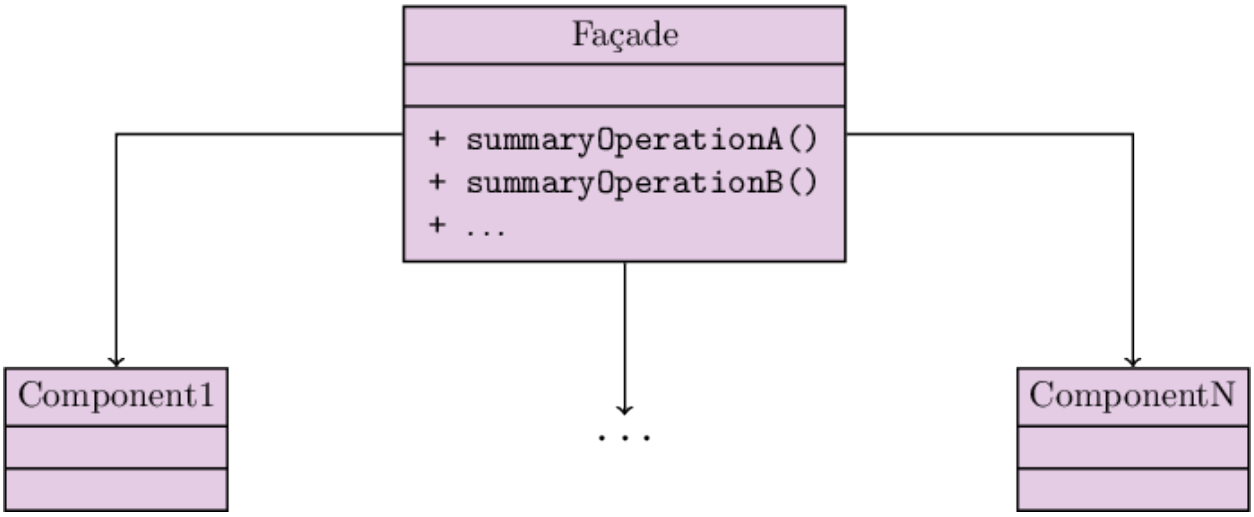


(HFDP Chapter 9)

In the Composite Pattern, a container type is a subtype of the same sort of elements it contains. Here, for instance, the `Composite` class contains `Component` objects but is also a kind of `Component` object. This allows the `Composite` to be recursively stored within another `Composite`.

An example of this pattern appears in many GUI libraries. In JavaFX, for example, all GUI components are subclasses of the `Node` class. Containers such as `VBox` are themselves subclasses of `Node`; this way, `VBox`es can contain other `VBox`es. The Composite Pattern allows containers and their components to be arranged in a tree structure with different rules (dictated by the containers) at each non-leaf node.

Façade



The Façade Pattern is used to hide the implementation-specific details of a system from its user. The façade object effectively acts as a summary of the system, allowing the caller to make high-level, abstract requests without understanding how the system satisfies them.

For instance, consider a desktop application designed to play movies. Such an application may have a `VideoFileLoader`, an `AudioPlaybackManager`, a `VideoPlaybackManager`, a `ScreensaverSuspender`, and so on. Consider the act of pausing the movie: this would require us to pause audio playback, pause video playback, and re-enable the screensaver. Furthermore, we

need to do so in a way that allows playback to resume correctly in the future (e.g. audio and video stay in sync).

To address this, we may create a `VideoPlayerFaçade` class with methods such as `pause()`, `resume()`, `play(File)`, and so on. These high-level actions hide the details about how all of the parts of the system are coordinated in order to accomplish the task. The façade methods don't really do much of their work work – they mostly just call methods from other objects – but they are useful because they separate the implementation of the high-level operation from the caller.