Name: Michael Towns

Pattern: Adapter

Category: Structural

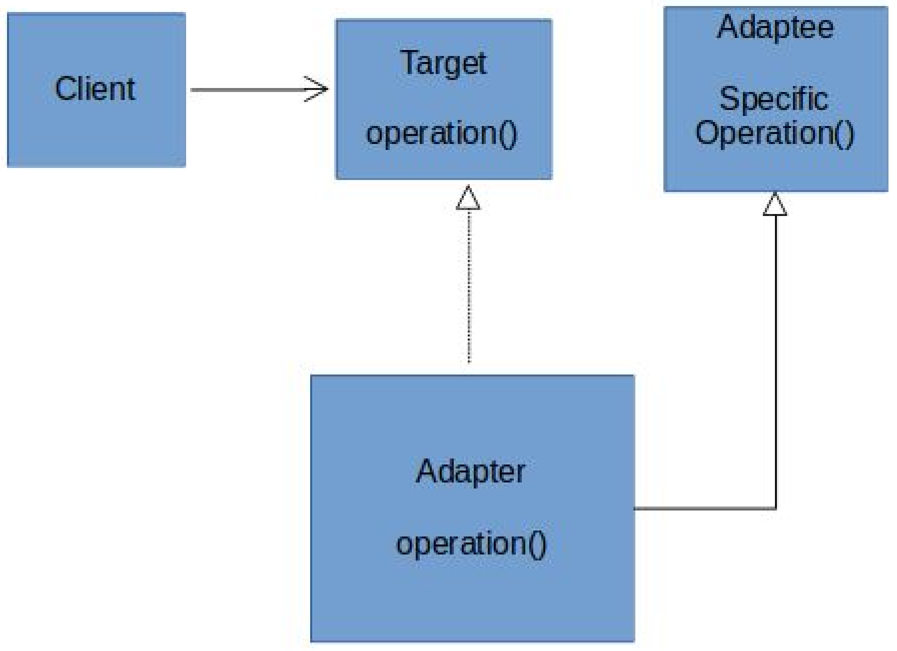
Description: Allows two incompatible interfaces to work together. Allows incompatible classes to work together

When to use: When you have two classes that have separate interfaces that must work together

Advantages:
- Do not need to completely re-write a class to follow a new interface
- Doesn't break compatibility
- Allows new APIs

Disadvantages:
- Longer, duplicate code
- May need chain of adapters as you add new functionality

UML Diagram:



Name: Michael Towns

Pattern: Specification

Category: Behavioral

Description: Chaining true/false statements together with boolean logic

When to use: When you want to separate the logic behind a rule from the rest of the program

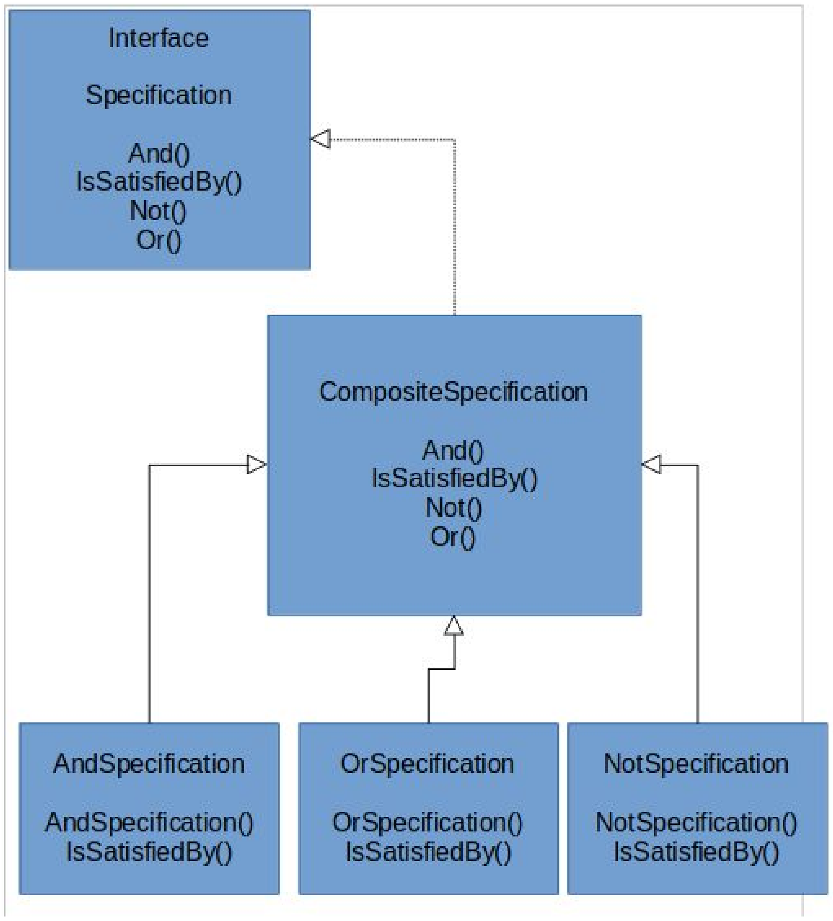
Advantages:

- Allows easier testing of the logic
- Allows modification of the logic in one place rather than many

Disadvantages:

- Longer code to implement new class
- More bouncing between classes to find the logic for a decision

UML diagram:



Name: Christopher Hebert

Pattern: Proxy

Category: Structural

Description: Provide a surrogate or placeholder for another object to control access to it

When to use the pattern: This pattern is used when there is need to create a wrapper to cover the main object's complexity from the client/user. There are multiple common situations where this pattern is applicable:

- *Virtual proxies*- delaying the creation and initialization of expensive objects until needed, where the objects are created on demand (For example creating the 'RealSubject' object only when the 'doSomething' method is invoked).

-

- *Remote proxies*- providing a local representation for an object that is in a different address space. A common example is Java RMI stub objects. The stub object acts as a proxy where invoking methods on the stub would cause the stub to communicate and invoke methods on a remote object (called skeleton) found on a different machine.

- Protection proxies- where a proxy controls access to 'RealSubject' methods, by giving access to some objects while denying access to others.

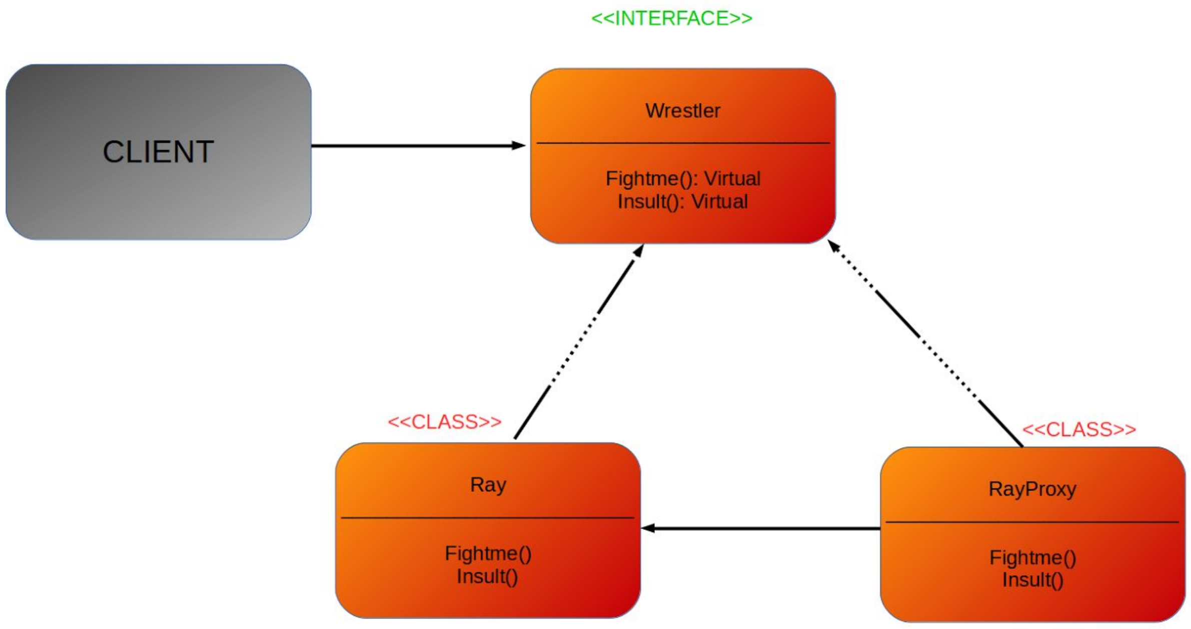
- *Smart references*- providing a sophisticated access to certain objects such as tracking the number of references to an object and denying access if a certain number is reached, as well as loading an object from database into memory on demand.

Advantages:

- Is more secure;
- Avoids duplication of objects, which saves space. This, in turn, helps performance;
- The remote proxy also ensures about security by installing the local code proxy in the client machine and then accessing the server with help of the remote code.

Disadvantages: This pattern *introduces another layer of abstraction* which may be an issue if the 'RealSubject' code is accessed by some of the clients directly and some of them might access the Proxy classes. This might cause disparate behavior.
So, added complexity.

UML diagram:


<<INTERFACE>>
CLIENT
Wrestler
Fightme(): Virtual
Insult(): Virtual
<<CLASS>>
Ray
Fightme()
Insult()
<<CLASS>>
RayProxy
Fightme()
Insult()

Name: Christopher Hebert

Pattern: Iterator

Category: Behavioral

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

When to use the pattern:

- When you want to access a collection of objects without exposing its internal representation

- When there are multiple traversals of the objects need to be supported in the collection

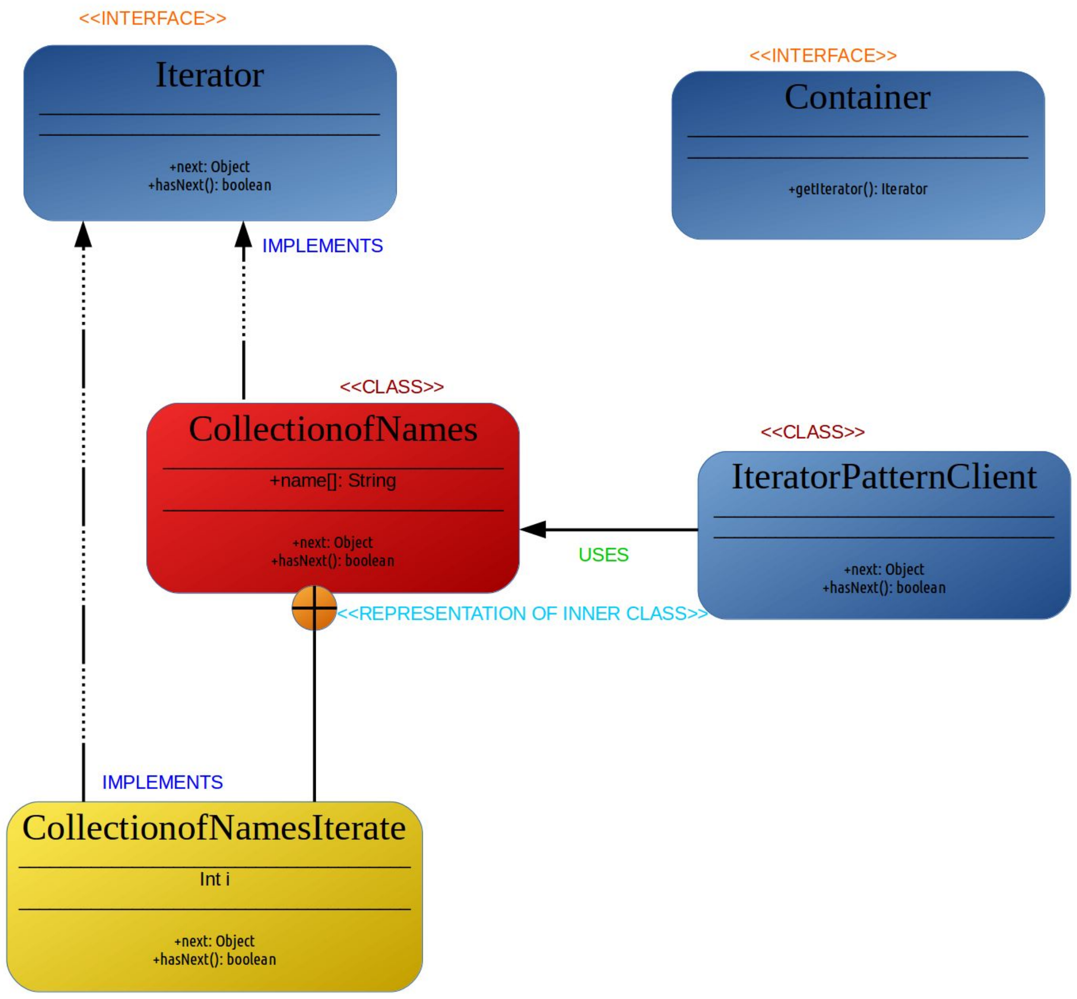
Advantages:

- Supports variations in the traversal of a collection

- Simplifies the interface into the collection

Disadvantages: In the 'single integral iterator,' the main disadvantage is that it supports only one traversal at a time.

UML diagram:


<<INTERFACE>>
Iterator
+next: Object
+hasNext(): boolean
<<INTERFACE>>
Container
+getIterator(): Iterator
IMPLEMENTS
<<CLASS>>
CollectionofNames
+name[]: String
+next: Object
+hasNext(): boolean
<<CLASS>>
IteratorPatternClient
+next: Object
+hasNext(): boolean
USES
<<REPRESENTATION OF INNER CLASS>>
IMPLEMENTS
CollectionofNamesIterate
Int i
+next: Object
+hasNext(): boolean

Name: Connor Costello

Pattern Name: Model-View-Controller (MVC)

Pattern Category: Software Architectural Pattern

Description: Divides an application into three interconnected parts. The Model represents the data, the view displays the model data and sends user actions to the controller, the controller provides model data to the view and interprets user actions.

Problem Description: This pattern should be used in situations where a GUI is being developed. It is particularly good for web interfaces or mobile applications. However, it is good for any application that has something to display and takes I/O.

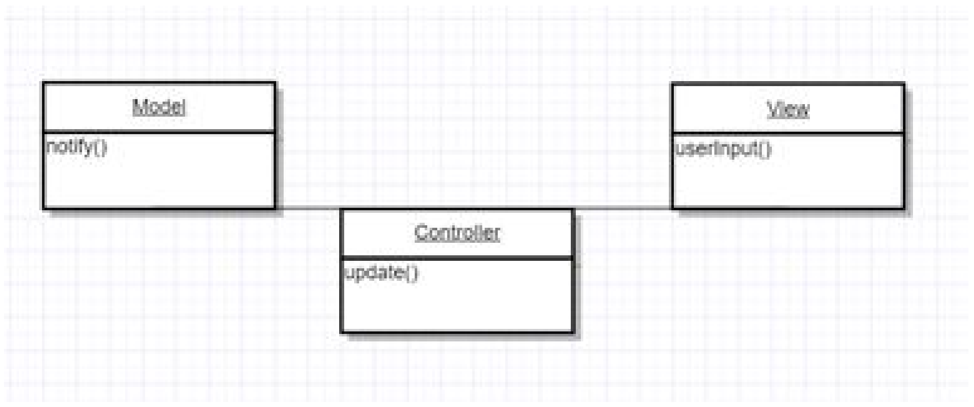
Advantages:

- Simultaneous development between developers
- High code reusability
- Easy to modify

Disadvantages:

- Controller does most of the work
- Code is hard to navigate
- High learning curve
- Different Implementations

Solution Description: The code will be split into three classes the model, the view, and the controller. The model class encapsulates all the data specific to the software and will define how the data will be changed and processed. The model is updated by the controller and the model notifies the controller of any changes in data. The controller class is the intermediary between the view and the model. Taking input from both and deciding how to pass it along to the other. This includes taking input from a user and passing it to the model and taking data from the model and giving it to the view class when needed. The view class is everything a user can see and interact with, how things are displayed and direct input and output. The view class passes user actions to the controller and the controller updates the view class. This can also be implemented several different ways in how the three communicate but each still keeps the same job.



Name: Connor Costello

Pattern Name: Module

Pattern Category: Structural Pattern and Creational Pattern

Description: Implement the concept for software modules for modular programming into programming languages with incomplete direct support for it.

Problem Description: The pattern can be used in any programming language that doesn't have direct support for modules. And should be used when wanting to implement a form of modular programming into the language.

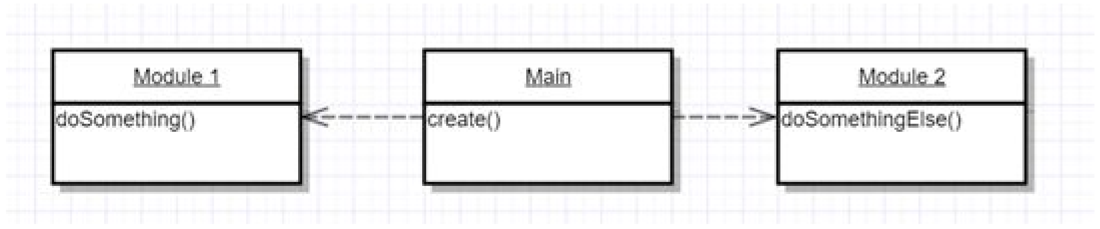
Advantages:

- Adds and initializer making up for a constructor
- Adds a finalizer making up for a destructor
- Allows for true public and private code encapsulation
- Allows developers to use modular programming concepts

Disadvantages:

- Code can become bulky easily
- Makes code harder to follow when having to switch between files

Solution Description: The implementation will be different for every language. Some implementations may be built into other design patterns such as java and singleton. But the core idea is that one file serves on purpose this could be a header file in C, a file that contains one class, of something like MVC split into 3 separate files.


Module 1
doSomething()
Main
create()
Module 2
doSomethingElse()