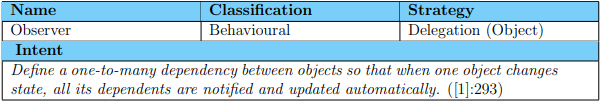
Observer Pattern



A diagram of a computer

Description automatically generatedSubject

* Provides an interface for observers to attach and detach to the concrete subject.

ConcreteSubject

* Implementation of the subject being observed.
* Implements the functionality to store objects that are observing it and sends update notifications to these objects.

Observer

* Defines the interface of objects that may observe the subject.
* Provides the means by which the observers are notified regarding change to the subject.

ConcreteObserver

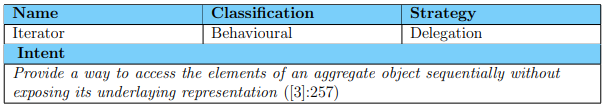
* Maintains a reference to the subject it observes.
* Updates and stores relevant state information of the subject in order to keep consistent with the state of the subject.

Observer Explained

The observer design pattern offers a mechanism by which observers of a subject register with the subject and will be notified when a change occurs in the subject. Figure 2 shows how the communication between the observers and the subject takes place. Figure 2: Overview of the interaction between the subject and its observers An observer will register with the subject. When the subject changes it will notify all the observers registered with it. When an object no longer is an observer of the subject it will deregister from the subject.

Iterator Design Pattern

A diagram of a software developer

Description automatically generated

Iterator

• Defines an interface for accessing and traversing elements.

Concrete Iterator

• Implements the Iterator interface

• Keeps track of the current position in the traversal of the aggregate

Aggregate

• Defines an interface for creating an Iterator object

Concrete Aggregate

• Implements the Iterator creation Interface to return an instance of the proper concrete iterator.

Iterator Pattern Explained

The Iterator design pattern moves the responsibility of traversing objects away from the aggregate to another class called an iterator. The aggregate class, therefore, can have a simpler interface and implementation because it needs only to cater for maintenance of the aggregate and no longer for its traversal

The iterator design pattern takes this good design a step further. Instead of just implementing every aggregate in two classes (one for maintenance, and one for traversal), this pattern is a design that provides a generic way to traverse the objects in aggregates that is independent of the structure of the various aggregates. This is achieved by defining two abstract interfaces – one for iteration and one for the rest of the functionality of aggregates. This way the system is more flexible when either aggregates or their iterations needs maintenance.

Improvements achieved

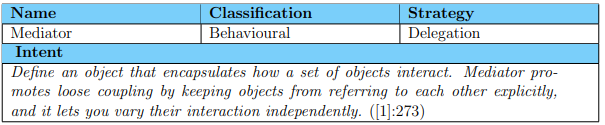
* All the functionality related to access and traversal is removed from the aggregate interface and placed in the iterator interface (More cohesive)
* Code becomes more flexible. if you change the underlying container, it’s easy to change the associated iterator.
* Iterators contribute to the reusability of your code – algorithms that were written to operate on a containers that use an iterator can easily be reused on other containers provided that they use compatible iterators
* It is possible to execute simultaneous yet independent iterations through the same structure.

Disadvantage

If the aggregate changes while the iterator is iterating, it may cause errors. Or if the data is first copied then traversed. And the data is then changed it may not traverse the correct data.

Mediator Design Pattern

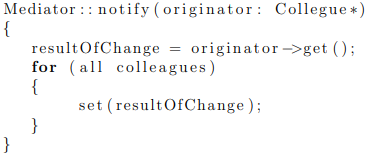
A diagram of a software code

Description automatically generated with medium confidence

Improvements achieved

Simplification of code updates: Objects can be added and removed at any time.

Increased reusability of code: Decoupling colleagues from each other is easy and increases coherence.

Simplification of object protocol: When refactoring into the mediator pattern a many-to-many relationship that exists between the elements in a group of objects is changed to a one-to-many relationship which is easier to understand and maintain.

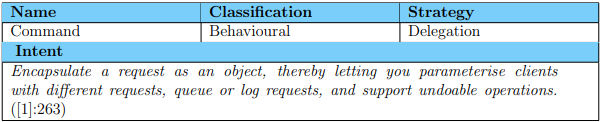
A diagram of a computer

Description automatically generated

A table of text on a white background

Description automatically generated

Command Design Pattern



A diagram of a computer

Description automatically generated

A diagram of a computer program

Description automatically generatedCommand

• declares an interface for executing an operation.

ConcreteCommand

• defines a binding between a Receiver object and an action.

• implements execute() by invoking the corresponding operation(s) on Receiver.

Client (Application)

• creates a ConcreteCommand object and sets its receiver.

Invoker

• asks the command to carry out the request.

Receiver

• knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Adapter Design Pattern

A close-up of a computer screen

Description automatically generated

A diagram of a computer component

Description automatically generatedA diagram of a computer component

Description automatically generated

Adaptee • The existing interface that needs to be adapted

Target • Domain specific interface used by the client

Adapter • Adapts the interface of Adaptee to the Target interface

Client • Manipulates objects conforming to the interface specified by the abstract class Target

Object Adapter

Object Adapter makes use of object composition to delegate to Adaptee.

Class Adapter

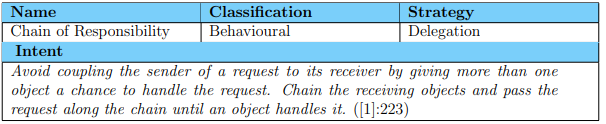
Class Adapter makes use of mixin idiom [4]. A mixin is an object-orientated concept by which a class provides functionality, either to be inherited or just used, but is not explicitly instantiated. Adapter inherits and implements Target (public inheritance). Adapter inherits only the implementation, or functionality, and therefore the use of private inheritance of Adaptee resulting in a linearisation of the hierarchy

Try to always use delegation. Use composition (inheritance) only when necessary

Real World Uses

When converting legacy code into modern code, it is easier to create an interface to use the legacy code rather than going and changing potentially highly coupled legacy code.

Chain of Responsibility Design Pattern



Handler

A diagram of a function

Description automatically generated• Defines an interface for handling requests.

• Implements the successor link.

Concrete Handler

• Handles requests it is responsible for.

• Can Access its successor.

• If the Concrete Handler can handle the request, it does so; otherwise it delegates the request to its successor via Handler.

Client • Initiates the request to a ConcreteHandler object in the chain.

Improvements Made

Reduced coupling: instead of each object knowing where the request needs to be handled. It instead only knows what It can handle and if it cant p[asse3s it on to its successor.

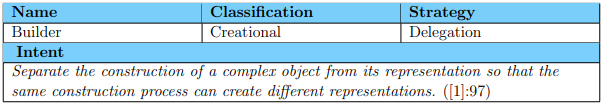
Added flexibility in assigning responsibilities to objects: flexibility in distributing responsibilities among objects and being able to change them easily.

Disadvantage

Receipt isn’t guaranteed: it is possible that no handlers can process the request and the last handler cannot pass it on to I ts successor as none exist, in this case an error is returned. If the chain of responsibility is not configured correctly other issues may arise.

Builder Design Pattern

A diagram of a software developer

Description automatically generated

Builder • specifies an abstract interface for creating parts of a Product object.

Concrete Builder • constructs and assembles parts of the product by implementing the Builder interface. • defines and keeps track of the representation it creates. • provides an interface for retrieving the product

Director • constructs an object using the Builder interface.

Product • represents the complex object under construction. ConcreteBuilder builds the product’s internal representation and defines the process by which it’s assembled. • includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Builder compared with Factory Method

1. Builder pattern contains a director class which the factory method does not.
2. Unlike the factory method which has a common interface for each of the subclasses to implement. The builder methods subclasses will most likely be building many different object which are not related and therefore cannot have a common interface.

Improvements achieved

**Variation product’s internal representation**: Builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled.

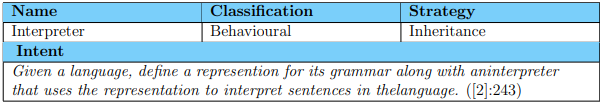
**Separation of code for construction and representation**: The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients need not know anything about the classes that define the product’s internal structure.

**Finer control over the construction process**: Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the director’s control. Only when the product is finished does the client retrieve it from the builder.

Extending a product

Each concrete builder creates a unique product. A concrete builder is allowed to define and add parts to a product that is not controlled by the director. Concrete builders usually define and maintain instance variables that can eliminate the need for the director to pass many values by means of parameters to the methods that assemble the product.

Interpreter Design Pattern

A diagram of a function

Description automatically generated

AbstractExpression • declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

TerminalExpression • implements an Interpret operation associated with terminal symbols in the grammar. • an instance is required for every terminal symbol in a sentence. NonterminalExpression • one such class is required for every rule R ::= R1 R2 ... Rn in the grammar. • maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn. • implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.

Context • contains information that’s global to the interpreter.

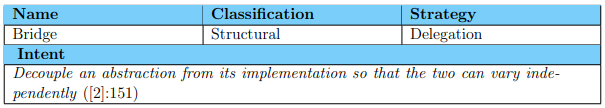
Client • builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes. • invokes the Interpret operation.

Improvements achieved

**Improved adaptability**: It’s easy to change and extend the grammar. Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar. Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.

**Implementation can be automated**: Once a grammar is defined, the class design and its implementation is completely determined by the rules of the grammar. The generation of code for classes defining nodes in the abstract syntax tree can often be automated with a compiler or parser generator using the grammar as input.

Bridge Design Pattern

****

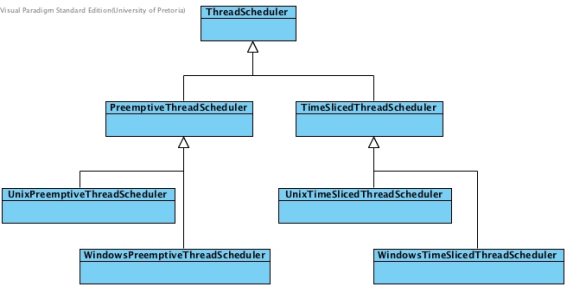
Abstraction • defines the abstraction’s interface. • maintains a reference to an object of type Implementor.

Refined Abstraction • Extends the interface defined by Abstraction.

Implementor • defines the interface for implementation classes. This interface doesn’t have to correspond exactly to Abstraction’s interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.

Concrete Implementor • implements the Implementor interface and defines its concrete implementation.

A diagram of a computer program

Description automatically generated**A diagram of a process

Description automatically generated** Improvements achieved

**Greater flexibility**: The coupling between an interface and its implementation is no longer fixed in terms of an inheritance relation.

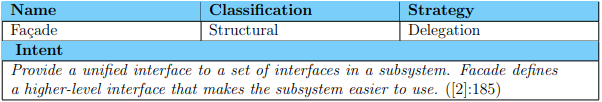
**Saving on compile-time:** Decoupling Abstraction and Implementor eliminates compile-time dependencies on the implementation.

**Improved Structure**: The high-level part of a system only has to know about Abstraction and Implementor.

**Hiding implementation details from clients:**

You can shield clients from implementation details. The definition of the Implementor class need only be visible to the Abstraction. It can be specified as private to the Abstraction class, hiding it completely from the clients using the Abstraction

Facade Design Pattern

****

**A diagram of a structure

Description automatically generated**

Facade • Knows which subsystem classes are responsible for a request. • Delegates client requests to appropriate subsystem objects.

Subsystem classes • Implements subsystem functionality • Handle work assigned by the Facade object. • Have no knowledge of the facade and perform operations independent of the facade.

Improvements achieved.

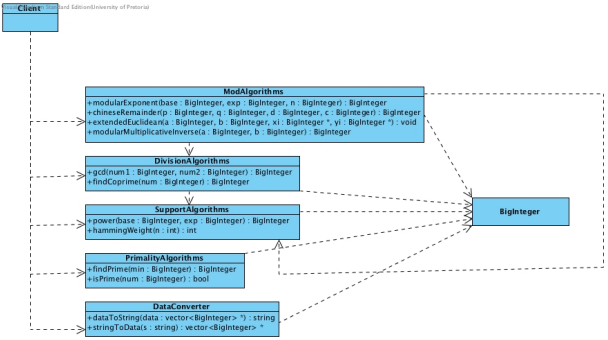
**Reduce coupling between clients and the system**: It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.

**Promotes weak coupling between subsystems**: Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layering a system and the dependencies between objects thereby reducing compilation dependencies and promoting portability of the code.

Common Misconceptions

• The Facade is not any automated process. To be an implementation of the Facade, the steps in the automated process should still be available as individual functions that can be performed without the aid of the Facade. It is important to notice that the facade should be implemented in such a way that It doesn’t prevent applications from using subsystem classes if they need to. Thus the clients must have the freedom to choose between ease of use (using the facade) and generality (bypassing the facade).

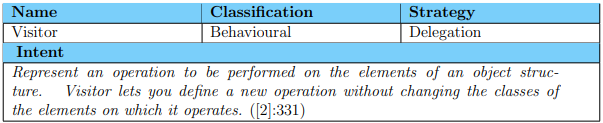
A diagram of a computer

Description automatically generated with medium confidence 

Before After

Visitor Design Pattern

A diagram of a construction site

Description automatically generated

Visitor • Each class of ConcreteElement has a visit() operation declared for it. • The operation’s signature identifies the class that sends the visit() request to the visitor. • The particular class is then accessed through the interface defined for it.

ConcreteVisitor • Implements the operations defined by visitor. • May store information about objects that are visited.

Element • Defines an accept() operation that takes an object of Visitor as a parameter.

ConcreteElement • Implements the accept() operation that takes an object of Visitor as a parameter.

ObjectStructure • Has a highlevel interface that allows the Visitor access and traverse its elements. • This structure may be a Composite or a collection such as an array, list or a set.

Improvements achieved

* The operations of a conceptual operation is kept together rather than being scattered in different classes in an aggregate. Thus cohesion is increased because related operations are logically grouped in different visitors.
* The different players are independent. This independence reduces coupling [1]
* Aside from potentially improving separation of concerns, the visitor pattern has an additional advantage over simply calling a polymorphic method: a visitor object can have state. This is extremely useful in many cases where the action performed on the object depends on previous such actions [5].

Implementation Issues

The Object structure is dependant on the Visitor to compile as it has visitors as parameters to its accept() methods. The Visitor in turn is also dependant on the Concrete Elements in the Object structure to compile for the same reason. This is called a cyclic dependancy. Fortunately they depend only on the names of the classes. Therefore, the problem can be avoided by including a forward declaration of the Visitor class in the Object structure and first compiling the Object structure or vice versa.

Proxy Design Pattern

A blue and white box with black text

Description automatically generated

**A diagram of a program

Description automatically generated**

Subject • Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject • Defines the real object that is represented by the proxy.

Proxy • serves as substitute for the real subject • maintains a reference to the real subject • controls access to the real subject • may be responsible for creating and deleting the real subject • more responsibilities specific to its kind

Proxy Pattern Explained

**Remote proxy**: The remote proxy provides a local representation of an object in a different address space. It therefor is used to hide the fact the the object may be on a different computer. The remote proxy is responsible for: • encoding a request and its arguments; and 4 • sending the encoded request to the real subject in a different address space.

**Virtual proxy**: The virtual proxy provides a local placeholder for an object that is expensive to create and maintain. It is used to postpone access to the expensive object until it is really needed. The virtual proxy is responsible for: • creating expensive objects on demand; and • caching information about the real subject so that access to it can be avoided if possible.

**Protection proxy**: The protection proxy controls access to the real object. It is used to control access rights to the real subject. Different users/objects may have different access rights. The protection proxy checks the access rights of a user/object for the particular request being issued. It may perform additional housekeeping tasks when the object is accessed.

**Smart reference**: When a proxy is implemented as a smart reference, it replaces a bare pointer and performs additional actions when the object is accessed. The typical uses of a smart pointer are:

Memory management - count the number of references to the real object

Load on demand - load a persistent object into memory on first reference

Safe updating - lock the real object before it is accessed

Smart Pointers

auto\_ptr: This strategy effectively transfers ownership of the pointer but leaves the original owner with nothing. It however does not provide a solution when copy semantics are required. cout << value.get() << endl; //access to the pointer

Unique\_ptr: shared\_ptr<int> value(new int(10));

unique\_ptr <int> value(new int(10));

unique\_ptr<int> newValue = move(value)//transfer ownership

weak\_ptr: shared\_ptr<int> value(new int(10));

weak\_ptr<int> newValue = value // value owns the mem, newValue holds a ptr

shared\_ptr:

shared\_ptr<int> value(new int(10));

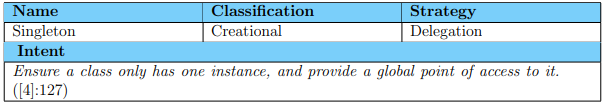
Shared\_ptr<int> newValue = value // newValue and value both own the momory

Value.reset(); // memory still exists due to dual ownership

newValue.reset(); // deletes memory

Singleton Design Pattern

A blue box with black text

Description automatically generated

Singleton

• defines an instance() operation that lest clients access its unique instance. This method is a static member function in C++.

• may be responsible for creating and destroying its own unique instance.

Improvements achieved

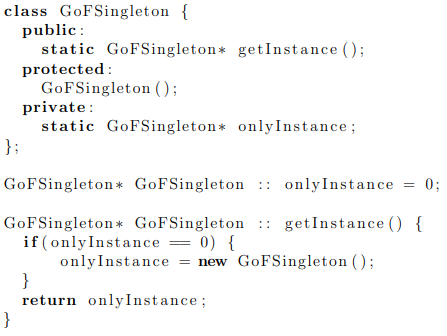
* If the Singleton class provide access to resources that are shared. The usage of these resources is more robust than would be without the pattern. The Singleton encapsulates its sole instance, hence it can have strict control over how and when clients access it.
* If the Singleton class is resource intensive, the application of the pattern may contribute to more optimal resource usage.
* The Singleton Pattern is an improvement over global variables. Although global variables can provide global access (part of the intent of this pattern) it cannot guarantee that only one instance is instantiated (the more crucial aspect of the intent of this pattern).

Disadvantages

* The application of the Singleton pattern violates the One class, one responsibilty principle to an extent. This principle is aimed at increasing cohesion in classes
* In multithreaded applications the creation step in the singleton design pattern should be made a critical section requiring thread concurrency control.

GoF Implementation Muldner implementation

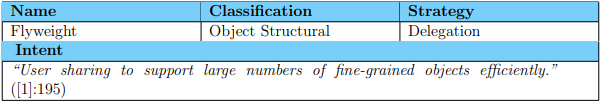
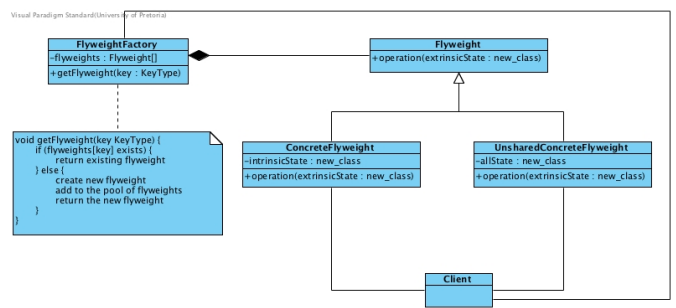
A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generatedLarman Implementation

Flyweight Design Pattern



**FlyweightFactory**: Creates an instance of a flyweight if it does not exist or supplies an existing one.

**Flyweight**: Defines the interface through which flyweights are instantiated

ConcreteFlyweight: Implements the interface and adds intrinsic (shareable) state storage. 2

**UnsharedConcreteFlyweight**: Not all flyweights need to be shared. Therefore not all need to store intrinsic state. UnsharedConcreteFlyweights may have ConcreteFlyweights as children

Flyweights have both intrinsic and extrinsic state.

**Intrinsic state**: refers to the internal state of the flyweight and can be shared as it is independent of the context in which the flyweight is. For example: a flyweight may represent a letter.

**Extrinsic state**: refers to the context in which flyweight is and therefore cannot be shared. For example: flyweights are ordered in terms of the context to form words.

