Design Patterns

Gregory Fay

Santa Clara University

December 2017

1. Introduction

Design patterns are general repeatable solutions to commonly occurring problems in software design. A design pattern is not code in and of itself, but instead it is a concept used to address common design obstacles. This paper will focus on design patterns for Object Oriented Design (OOD). Design patterns help with software design in regards to class interfaces, inheritance hierarchies, and establishing key relationship among them. Effective design patterns can speed up the development process by providing proven development paradigms. These designs can solve future problems before they become visible to the developer.

Design patterns allow developers to communicate using well-known, well-understood software principles. This can ease the communications between team members and help align a software development team behind the same mission. The design patterns discussed are not intended for use in solidarity. A majority of the time, a software product includes multiple complementary design patterns.

In this paper, I will discuss and describe the design patterns as established in “Design Patterns” by Erich Gamma, Richard Helm, Ralph Johhson, and John Vlissides. The patterns are split up into three major sections: creational, structural, or behavioral. Creational patterns are concepts for object creation. Structural patterns deal with object or class composition. Behavioral patterns detail concepts for the interaction and delegation of responsibility between classes. Design patterns are meant to be used together to achieve a productive application. They help a designer recognize less obvious abstrctions that can have large impacts on their code’s resilience and reusability.

Some principles help guide the overall reasoning behind design patterns. One of these concepts is to “program to an interface, not an implementation”. This creational strategy abstracts the creational process. When changes occur, less code will need to be altered when programmed to an interface. Another principle is to “favor object composition over class inheritance.” Ideally a designer should be able to assemble existing components in a new manner to achieve new functionality instead of creating a new component.

Design patterns help a software team design a full functioning application, tooklit, or framework. They promote code readability by establishing naming conventions for the concepts. They aid code maintainability because these concepts are more quickly understood. They promote communication and intent amongst programmers by quickly aligning readers to its purpose. They promote code re-use by identifying common, tested solutions to problems, and they require less code by deriving common functionality from the base class. Overall, design patterns establish tested, and proven solutions in software design. Designers are encouraged to create their own design patterns as well within their framework. The design patterns discussed below are an introduction into the infinite possibility of design patterns that can be created to facilitate software design.

1. Creational Patterns

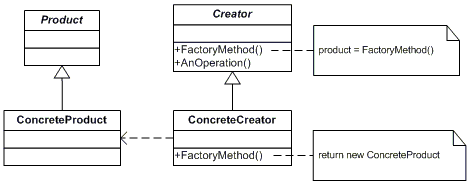
Creational patterns define the design for class instantiation. These patterns help abstract the instantiation process so systems are independent of how objects are created, composed, and represented. There are two main types of creational patterns, class creational patterns and object creational patterns. Class creational patterns use inheritance to alter the instantiated class, while object creational patterns delegate instantiation to another object. Creational patterns help make code more flexible and promote code reuse. They accomplish this by abstracting the object creational process. This facilitates the addition of new classes to improve the code and handle the ever-changing needs of the system.

**Factory Method**

The Factory method utilizes the Virtual constructor method in C++ to let classes defer instantiation to its subclasses. This has become a standard class instantiation principle in C++.

The problem arises in that frameworks seek to standardize the architectural model for a variety of applications, and these individual applications also need unique domain objects and unique instantiations.

Factory Method Structure



In Factory Method, a superclass defines all standardizations and generic methods by using “virtual” placeholders. It then delegates creation details to subclasses. A client requests an object through the Creator interface, and it matched with a corresponding ConcreteProduct. This abstraction allows the code to only deal with product interface, and delegate instantiation to the Factory Method

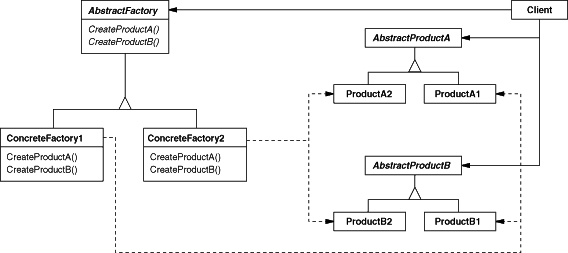
The Factory method is a simple approach, and many designs start out using this method and then evolve towards Abstract Factory, Builder, or Prototype as the designer determines where more flexibility is needed.

**Abstract Factory**

The abstract factory concept discusses creating an abstract interface for creating families of related or dependent objects without specifying their concrete classes. This design creates a hierarchy that encapsulates many possible “platforms” and the construction of a suite of “products” within a platform.

If a platform is to be portable, it often has to accommodate platform dependencies. With the abstract factory design, a client does not specify its concrete classes; it instead goes through the abstract factory to obtain its concrete classes. The Abstract Factory facilitates exchanging product families because the class of the factory object only appears where it is instantiated in the application. The application can replace the entire family of products by instantiating a different concrete instance of the abstract factory. The application can also enforce that a family of products are used in conjuncture. An example of this is a user interface supporting multiple look and feel standards such as Motif and Presentation manager. I would also hypothesize that React Native utilizes Abstract Factory to allow the client-code interface to be implemented on both the iOS platform and the Android platform.

Abstract Factory Structure



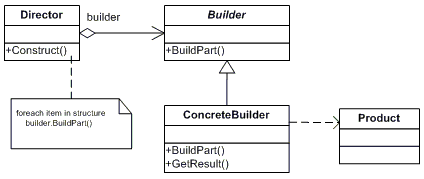
The abstract Factory defines a factory method per abstract product. Object initialization is contained within the AbstractFactory, and each concreteFactory produces a concrete product for each abstract Product. The client accesses each product through this AbstractProduct interface. The client will access the corresponding concrete product from the AbstractProduct interface depending on which ConcreteFactory platform is in use.

**Builder**

The Builder design splits up the construction of an object from its representation. This allows the same construction process to create different representations.

This problem is encountered when an application needs to create the elements of a complex aggregate. Many representations need to be built into primary storage.

Builder Structure

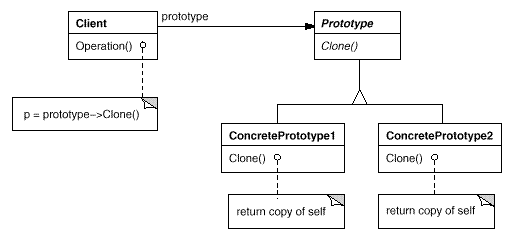


In this structure, the Director reads the data that needs to be aggregated and invokes the Builder to construct the product format. The Builder creates parts of the complex object each time it is called from Director and stores it in the Product Object. When the building is completed, the client retrieves the result from Builder. The Reader encapsulates parsing the common input, and the Builder hierarchy allows the polymorphic creation of different concrete Builders. This abstraction allows the Director to use the same construction process with a different concrete Builder to create a different representation.

**Prototype**

The Prototype pattern creates new objects by cloning itself and creating a new prototypical instance. Instead of calling ‘new’ the designer would call prototype->clone() to get a new instance of an object. Prototype can avoid expensive creation from scratch calls and support cheap cloning of a pre-initialized instance. Prototype is useful when the classes to instantiate are specified at run time. To use the Prototype method, declare an abstract Prototype class with a pure virtual Clone() method. Then create a dictionary of concrete cloneable derived classes. The client now will write a clone operation on the abstract base class with a string or enumerated data type specifying the particular concrete class that is desired. The client will no longer invoke the new method; this is abstracted in the concrete Prototype classes.

Prototype Structure



This method abstracts the object instantiation and makes it easy to create a copy of an object that is already in use. The client creates a new object by requesting the prototype to clone itself. The prototype declares the abstract interface for cloning oneself, while a concretePrototype implements the operations for cloning oneself, invoking the new operation.

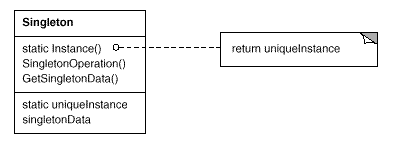
An example of this method in action would be in a music score editor. Imagine a palette of tools that has various different musical notes. These objects would all be concrete prototypes of the prototype class. In order to add a new musical note to the score, a user would click on a tool, and then add it to the editor. This would allow the program to run smoothly at run time by simply cloning whichever concrete class was selected, and adding this newly instantiated object to the editor.

Prototype can often be a strong complement to the Composite and Decorator design. Prototype is unique in that it only requires an object to create a new instance, not a class.

**Singleton**

The Singleton method ensures that a class has only one instance, and provides a global point of access to it. This is necessary when a system needs one, and only one instance of an object. The Singleton method is desirable when ownership of the single instance can not be reasonable assigned, lazy initiation is desirable, and global access is not otherwise provided for. The single instance is responsible for creation, initialization, access, and enforcement. Declare this instance as a private static data member. It is important to note that the static member function accessor does not support subclassing of the Singleton class.

Singleton Structure



It is important to note the static member function declaration. The advantage of Singleton over Global Variables is that you are absolutely positive there is only one instance when you use Singleton.

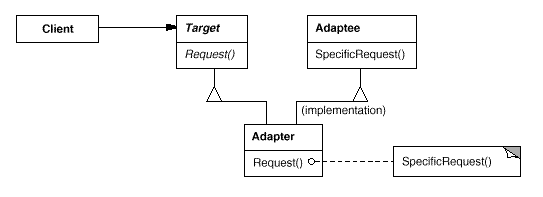
1. Structural Patterns

Structural Patterns deal with how classes and objects are composed to form larger structures. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to achieve new functionality. By using inheritance and interferences, structural patterns allow objects to be designed in a manner providing new functionality.

**Adapter**

The Adapter pattern does what it sounds like; it lets classes work together that couldn’t otherwise by adapting the representation and creating an interface. This is also known as Wrapper design. This should be used when you want to use an existing class, and its interface does not match what you need. It is also useful when you create a reusable class that cooperates with unrelated or unforeseen classes, as this class can handle situations where interfaces are not necessarily compatible. It works well to wrap an existing class with a new interface.

Adapter Structure



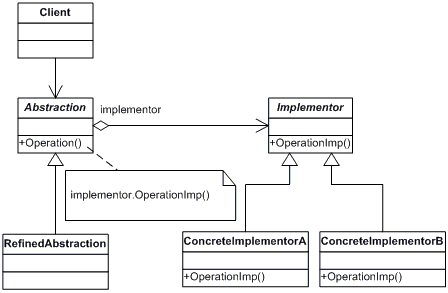
To utilize this design pattern, first identify the client and the component that needs to be adapted (the Adaptee). Identify the Target interface the client requires. Design an Adapter class that can impedance match the Adaptee to the Client. The Adapter class has an instance of the Adaptee class, and it maps the client interface to the Adaptee interface. In this example, the Adapter would perform SpecificRequest() for client and return it in the Target format.

**Bridge**

The idea of the bridge is to decouple an abstraction from its implementation so that both can vary independently. To accomplish this, decompose the components interface and implementation into orthogonal class hierarchies. The interface contains a pointer to the abstract implementation superclass. Many frameworks utilize the bridge pattern to handle UI/UX components. For example, the code used to define the layout of a component would be abstracted from the actual rendering of the component. This way, the abstract implementation can focus on what layout is desired, and a separate implementation hierarchy can handle what concrete rendering is necessary for the platform in use.

To use this pattern, first decide if two orthogonal dimensions exist in the domain. This could be abstraction / platform, front-end / back-end, domain / infrastructure, or interface / implementation. Then design the separation of abstraction to implementation.

Bridge Structure



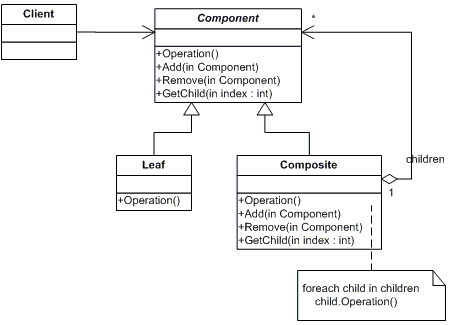
The client does not need to deal with platform-dependent details. The Bridge abstraction encapsulates the implementation complexity behind an abstraction wrapper. Design an abstract Implementor that minimally handles the client’s requests. Its goal is to decouple abstraction from the platform. Then define a concrete Implementation of each abstract operation. The abstraction class has an implementor instance that specifies which implementation should be used, and does not need to worry about its implementation besides knowing its type.

Bridge is designed upfront to allow abstraction and implementation to vary independently. Meanwhile, the adapter class is retrofitted to make unrelated classes work together.

**Composite**

This pattern composes objects into tree structures to represent part-whole hierarchies. Composite allows clients to treat individual objects and compositions of objects uniformly. This utilizes recursive composition. An example of this is directory that contains files, each of which could be another directory. Often when dealing with tree-structured data, programmers need to differentiate when dealing with a leaf node or a branch. The Composite interface allows the programmer to treat complex and primitive objects uniformly. The key concept is that you can manipulate a single instance of the object just as easily as manipulating a group of instances.

Composite Structure

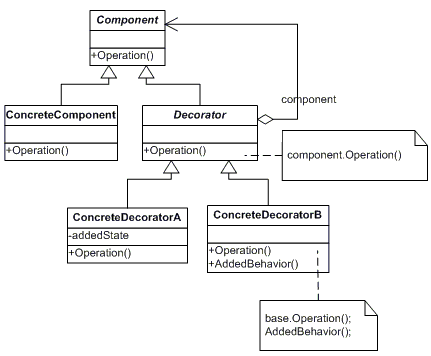


Component declares the interface for the client to interact with a composite and accessing and managing its child components. The Leaf defines behavior for primitive objects in the composition. The Composite defines behavior for components having children. It stores child components and implements child-related operations in the Component interface. The Client manipulates objects in the composition via the Component interface.

**Decorator**

The decorator design attaches additional responsibilities to an object dynamically. Decorators extend functionality without sub-classing. It provides client-specified embellishment of a core object by recursively wrapping it. It is used when you want to add behavior or state to individual objects at run time, but inheritance is not possible because you don’t want these changes to apply to the entire class. The decorator attaches additional responsibilities to an object dynamically. Suppose you are creating a user interface toolkit and you wish to support adding borders and scroll bars to the window. If you encapsulate the original object inside an abstract wrapper interface, you can allow the core object to dynamically increase its functionality. Both the Decorator object and the core object inherit from this abstract wrapper interface. The interface uses recursive composition to aggregate any number of Decorator layers to the core object.

Decorator Structure

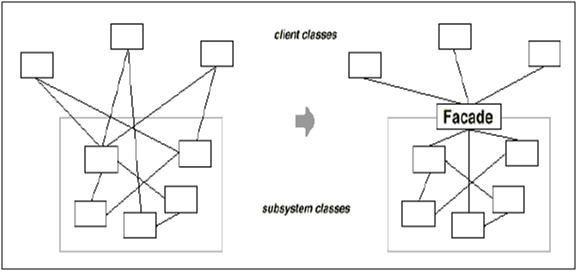


The Component defines the interface for objects with optional responsibilities added to them dynamically. This also incorporates the lowest common denominator that makes both classes interchangeable. The ConcreteComponent defines the core object that can add additional responsibilities. The Decorator maintains a reference to a Component object and defines an interface that conforms to Components interface. It forwards request to its Component objects with the option of performing additional operations before and after forwarding. The ConcreteDecorator is the specific embellishment that adds responsibility to the core component.

**Facade**

Facade provides a unified interface to a set of interfaces in a subsystem. It wraps a complicated subsystem with a simpler interface. This occurs when a segment of the client community needs a simplified interface to the overall functionality of a complex system. The facade design lowers learning curve necessary to leverage a subsystem. This object should be thought of as a facilitator or advocator. Introducing a facade decouples the subsystem from clients, and promotes portability and independence in the subsystem.

Facade Structure



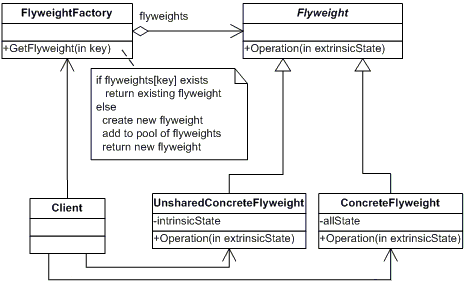
In this structure, the Facade knows where to forward subsystem requests. It maps client requests to the appropriate subsystem objects. The subsystem classes handle work assigned by the Facade object, but have no references to it. The subsystem now will receive requests only from the Facade object instead of from each client individually. An example of a Facade object is a Compiler. The Scanner, parser, ProgramNode, and etc. would be part of the subsystem classes. This allows for more managed subsystem access through a single object, but can limit the features and flexibility power users may need.

**Flyweight**

Flyweight uses sharing to support large number of fine-grained objects efficiently. Designing objects to the lowest level of granularity can invoke unacceptable performance and memory usage. Before using this pattern, ensure that object overhead is an issue requiring attention, and that the client is willing to accept the responsibility realignment. It should be applied when an application uses a large number of objects. In order to utilize Flyweight, many groups of objects must be able to be replaced by relatively few shared objects once extrinsic state is removed.

Each flyweight object is divided into two pieces, the state dependent (extrinsic) part and the state-independent (intrinsic) part. The intrinsic state is stored in the shared Flyweight object pool. The extrinsic part is stored or computed by the client object, and passed to the flyweight when its methods are used. The Flyweight contains part of the object, but the client needs to provide the remaining pieces to make it a complete object.

Flyweight Structure



Flyweights are stored in the FlyweightFactory’s repository. The client does not instantiate flyweights directly, and instead requests flyweights from the Factory. Each concrete flyweight cannot stand on its own. It needs the extrinsic value from the client. The Flyweight declares the interface which flyweights receive the extrinsic states from the client and acts on it. The concreteFlyweight implements the flyweight’s interface and stores its intrinsic state. A concreteFlyweight must be sharable and store intrinsic states. However, not all Flyweight subclasses need to be shareable. The unsharedConcreteFlyweight interface enables sharing, however it does not enforce it. The flyweightFactory manages and creates Flyweight objects. It ensures they are shared properly. When a client requests a flyweight, the flyweight factory will either provide one from the pool of current flyweight objects, or it will create a new one if it doesn’t exist. The client maintains a reference to flyweight(s). The client also computes or stores the extrinsic state of flyweights. This assigns additional responsibilities to the client.

Flyweight may increase run time as it transfers, finds, and computes extrinsic state. However, it will provide storage savings in relation to the reduction of total number instances that comes from sharing and the amount of intrinsic state per object. The more flyweights are shared, the greater reduction in storage.

**Proxy**

The proxy class defines a wrapper for an object to control access to it and act as an interface. The proxy also may attach additional behavior onto the wrapped object, without needing to alter the base object class behavior. This extra level of indirection supports distributed, controlled, or intelligent access. The added wrapper protects the real component from complexity. The problem often arises when you need to support resource hungry objects, but you do not want to instantiate them until they are actually requested by the client. Proxy can act like a smart pointer, and is applicable when there is need for a more versatile or sophisticated reference to an object. A smart pointer has options to perform additional actions when referencing an object, such as counting the number of references to a real object so that it can be freed when no more references exist. It can check locked conditions. It can load a persistent object into memory when it is first referenced.

Proxy Structure



The Subject interface makes the presence of the proxy object standing in the place of the real object transparent to the client. The realSubject is the object the proxy represents. The proxy references the realSubject when necessary. It provides an interface identical to the subject’s. This allows the proxy to be substituted for the real subject. The proxy controls access to the real subject by forcing the client to make requests through it. The proxy may be responsible for object creation and destruction. Other responsibilities can differ based on the type of proxy. Remote proxies encode a request and it’s arguments and sends the encoded request to the real subject in a different address space. Virtual proxies cache additional information about the real subject so they can postpone accessing it. Protection proxies check if the client has authorized access to perform a request. The Subject defines the common interface for the real subject and proxy. This allows the proxy to be used by the client anywhere the real subject was expected.

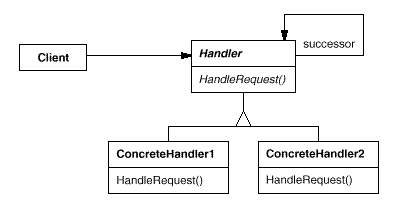
1. Behavioral Patterns

Behavioral patterns deal with class’s object communication. They deal with the assignment, responsibilities, and interconnectedness between objects. These patterns use inheritance to distribute behavior between classes.

**Chain of Responsibility**

This pattern avoids coupling the sender of a request to the receiver by giving multiple objects a chance to handle the request. The request is passed through a chain of objects until an object handles it. This chain is a single processing pipeline. This is essentially an object oriented linked list with recursive traversal. Use chain of responsibility when more than one object may handle a request, and the handler can’t be determined logically. The set of objects able to handle a request should be specified dynamically and you don’t want to explicitly specify the receiver.

Chain of Responsibility Structure

****

In this pattern, the Handler superclass defines the interface for handling requests and also implements the successor link. The concreteHandler handles the request if able to, otherwise it forwards the request to its successor, which it has access to. The Client initiates the request into the abstracted handler and the correct concreteHandler object handles the request.

The Chain of Responsibility reduces coupling, and adds flexibility in assigning responsibilities to objects, however, there is no guarantee that the request will be handled. However, this error will only happen if the chain is not configured properly.

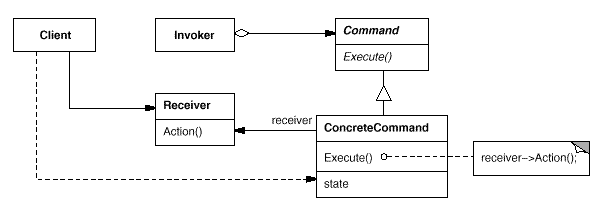
**Command**

This pattern encapsulates a request as an object, letting you parameterize clients with request objects, log or queue requests, and support undoable operations. This promotes an invocation of a method on an object to full object status, and creates object oriented callback functionality. This pattern decouples the object that invokes an operation from the one performing it. An abstract base class (Command object) maps a receiver object with an action (a pointer to a member function). The concrete class contains an execute method that calls the action on the receiver.

The command pattern is useful when objects specify, queue, and execute requests at different times. Command pattern can also support an undo command, by storing state for reversing the effects in the command itself. Sequences of command objects can also be assembled into composite commands.

Clients that create a command are not the same clients that execute it. This separation provides flexibility in timing and sequencing of commands. Since commands are objects, they can be passed, shared, staged, loaded in a table, or otherwise manipulated like any other object. It may be helpful to think of command objects as tokens, with one client knowing what needs to be done (token creation), and another client having access to the resources necessary to perform that task.

Command Structure

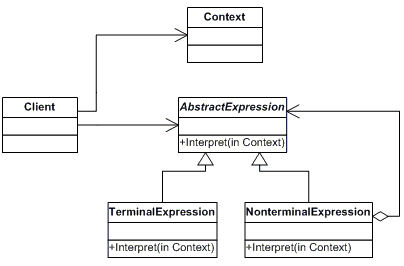
****

The client creates a ConcreteCommand object and sets its receiver. The Invoker asks the command to carry out the request. The Command declares an interface for performing a command (operation). The ConcreteCommand defines a binding between an action and its Receiver object. The Receiver object has the resources and knows how to perform the associated command.

**Interpreter**

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. Map a domain to a language, the language to the grammar, and the grammar to a hierarchical object-oriented design. For this pattern to be applied, a class of problems appears in a well-defined and well-understood domain. If this domain could be characterized as a “language,” then the interpreter engine would convert content into an abstract syntax tree. An example of this from COEN 175, where we used a parser to convert text input into an abstract grammar tree representation to represent a compiler. Since the text we were working with followed well-defined syntax rules, the interpreter is a great fit.

Interpreter Structure



This pattern will interpret an input into an abstract syntax tree, making it possible to perform more object oriented techniques on the interpreted context. The Context contains information that is global to the interpreter. This information part of a well defined structure, but it is not yet mapped to our object oriented analysis. The AbstractExpression declares an abstract virtual expression that is common to all nodes in the abstract syntax tree. This will either be a terminal expression or a compound expression containing one or more terminalExpressions. The TerminalExpression is a rudimentary building block of the language to be interpreted that maps directly to an abstract object in the syntax tree. The CompoundExpression process the symbol they represent in addition to child nodes that compose the full expression. These operations may perform specified operations from the defined grammar, and will ultimately trace their expression down to a TerminalExpression. The Client invokes the Interpreter expression, and houses the abstract syntax tree. This abstract syntax tree is constructed from the terminal and nonterminal expressions, and by following the defined grammar.

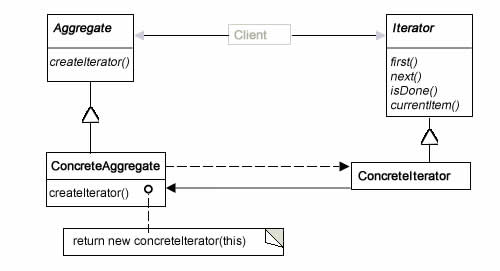
This pattern works well when there is a well-defined grammar structure that needs representation. It is easy to change and extend the grammar, however complex grammars can become hard to maintain. Implementing the grammar is often easy once it has been defined. It also benefits by facilitating the addition of new ways to interpret expressions.

To utilize this pattern, first define a grammar for the language. Then map each production in the grammar to a class. Organize the suite of classes into the Composite pattern and define an interpret(context) function in the composite hierarchy. The context object encapsulates the current input, and as the parser increments through the context, the abstract syntax tree is accumulated. The context is shortened as each grammar class operates its corresponding grammar rules interprets the input into the output.

**Iterator**

The Iterator provides a way to sequentially access elements of an aggregate object without exposing its internal structure. This implementation abstracts the traversal of completely different data structures so that algorithms can be defined that are capable of interfacing with each data structure. The iterator provides ways for an iterator object to traverse a data structure list. Often these operations include first(), last(), next(), is\_done(), current\_item(), and etc to allow the iterator access to the collection class.

Iterator Structure



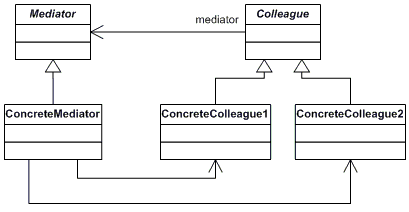
Iterator designs the abstract interface for traversing a list. Then each list needing iteration would implement their concrete implementation for performing these actions. The client uses the concreteIterator to traverse the list, and this concreteIterator stores the current position of its iteration. The Aggregate defines the common interface for creating an iterator object. Each concreteAggregate will initiate the corresponding concreteIterator. This means that the client must request an iterator from the Aggregate, and then use its functionality through the abstracted Iterator interface.

Iterators are very common in object-oriented systems, as they provide a means for traversing data structures created by the developer. Iterators are often used with the Composite pattern to traverse their recursive structures. The factory method is invoked to create concrete iterators for the appropriate Iterator subclass. Iterators provide an abstracted method to traverse data structures, explicitly separating the data structure from the algorithm acting on it.

**Mediator**

The mediator object encapsulates how a set of objects interacts. Mediator creates loose coupling by restraining objects from communicating to each other directly, and allows you to vary their interaction independently. The mediator pattern is applicable when a set of objects communicates in a well defined but complex manner. Their intercommunication results in an unstructured, difficult to understand fashion. The mediator approach encapsulates all interconnections, is responsible for the control and coordination of client interactions, and promotes loose coupling by keeping objects from explicitly referencing each other. It acts as the hub for all communication.

Mediator Structure

****

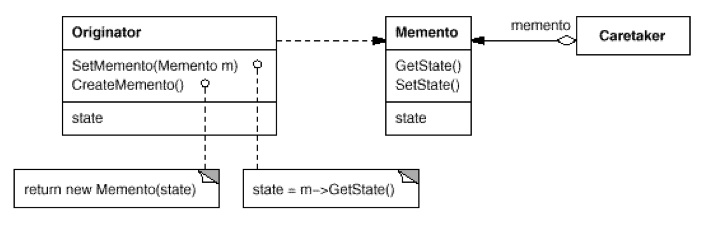
The Mediator defines the interface for communications between colleague objects. The ConcreteMediator coordinates colleague object requests enforces that they are carried out in a defined manner. The concrete mediator has references to and maintains its colleagues. The colleague class knows its mediator, and it send requests through the mediator instead of sending requests directly to other colleagues.

This pattern was initially described as a one-to-many relationship, but it has since adapted to also include a many-to-many relationship. In this scenario, the mediator is the hub for communication between two sets of objects. For example, the mediator could establish the interface for communications between a set of Consumer objects, and a set of Producer objects. They would not be in direct contact with each other, but would contact each other through the mediator that would in turn orchestrate the communications. A call center could map consumers to Producers, based on availability, priority, and any other methods the mediator desired to implement.

**Memento**

The Memento pattern captures and externalizes an objects internal state. This is done without encapsulation and allows the object to be restored to this previous state later. This allows a client to undo to rollback an object to its previous state. The client requests a memento from the source object when it wants to save a state. The client acts as a caretaker for the memento (or “token”). The client does not save the objects full implementation, instead it saves a token that when passed to the originator allows it to instantiate its previous state. This is helpful because the originator is not burdened with saving additional data, and the client is decoupled from the originator. An unlimited amount of undo and redo capability can be performed with a stack of command objects and memento objects.

Memento Structure



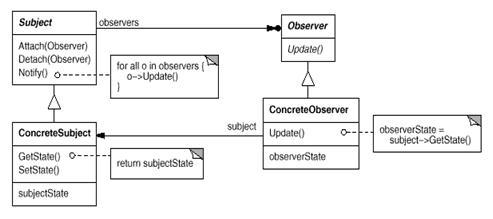
The Originator is the object that knows how to save itself, and is able to restore itself by using the memento. The Caretaker does not operate on or examine the memento, but is responsible for its safekeeping. The Memento is the lock box written and read by the Originator, and referenced by the Caretaker. The Caretaker knows when to checkpoint the originator, stores this reference, and passes it back to the Originator when desired.

Command and Memento are complimentary design patterns that maintain state for an undo operation. Memento is also often used with Iterator to capture the state of an iteration.

**Observer**

The observer pattern defines a one-to-many dependency, where the many observers are notified any time the subject changes state. This allows the dependents to be notified and updated automatically. The idea is to encapsulate the core components in the subject abstraction, and the variable components in the Observer hierarchy. This pattern allows the subject to broadcast a message when it changes states, and allow variable observers to subsequently update their state with the updated information. Use this pattern when an abstraction has two components, and one is dependent on the other. The Observer pattern allows observers to update based off a change in the Subject, without tightly coupling the two.

Memento Structure



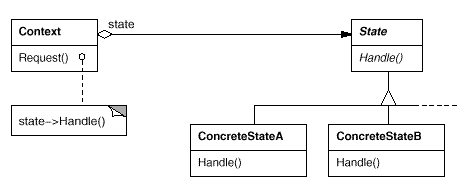
The Subject knows its observers and provides an interface for linking and unlinking Observer objects. The Observer defines the abstract interface that is notified of any change in the subject’s state. The ConcreteObserver then in turn requests the updated value from the ConcreteSubject and updates its local copy of the state. The ConcreteSubject stores its data of interest, and notifies it’s observers whenever that data of interest changes state.

This protocol describes a ‘pull’ interaction model where observers are responsible for pulling the information out of the subject. Subsequent designs have led to a subject ‘push’ model where the subject broadcasts its alteration to the set of observers. The Observer design pattern is a major part of Model-View-Controller architecture which has been a huge part of implementing user interfaces on computers and in iOS development.

**State**

The State pattern allows an object to alter its behavior when its internal state changes. This allows an object to act as if it were a different class based on changes to its internal state. Therefore, this pattern should be used when an object’s behavior depends on its state and it needs to alter behavior at run time based on its changed state.

State Structure

****

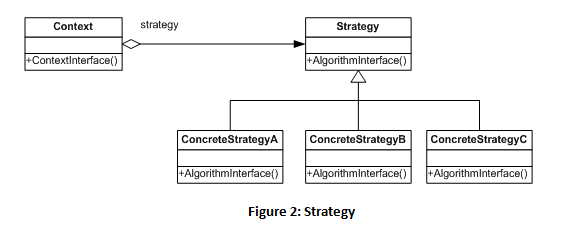
To utilize the State design, begin by defining a Context class that acts as the single interface to the outside code. Define an abstract State base class and represent the different states of the state machine as subclasses of the State class. Implement the state specific behavior of each ConcreteState. The context class maintains a pointer the to the current ‘state.’ To change the current state, change the context’s pointer to the desired ConcreteState.

The design does not specify where state transitions should be made (context vs. subclasses). If the criteriais fixed, it can be implemented entirely in the Context. However, this is not likely. Generally, the state subclasses themselves specify when the state should be changed and what it should change to. This provides more flexibility but also requires adding an interface to Context that lets State objects set the Context’s current state explicitly.

**Strategy**

This design pattern creates a family of interchangeable algorithms, and abstracting their interface so clients can use them interchangeably. This pattern captures the abstraction in the interface and implementation details in the derived classes. This is useful when you have a family of related classes that differs in their algorithmic behavior. Certain algorithms provide better space/time trade-offs based off which algorithm is invoked. The strategy pattern allows for a class hierarchy of algorithms, allowing the Context to invoke an optimal algorithm for its context.

Strategy Structure

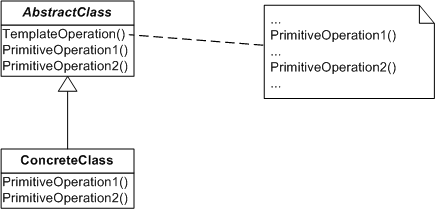
****

The strategy declares an abstract interface that each concrete strategy implements. Context is configured with a concreteStrategy object and uses the Strategy interface to call the corresponding algorithm. It maintains a reference to a Strategy object to access the correct algorithm. There are many contexts that relate to the Strategy hierarchy.

**Template Method**

Template Method defines the skeleton of an algorithm in an operation, while delegating some steps to subclasses. This pattern allows subclasses to redefine steps of an algorithm without changing the algorithms structure. The base class declares ‘virtual’ placeholders and the subclasses implement these methods. This method is particularly useful when two different components are strikingly similar, but differ in a small number of ways. This method implements the invariant parts of an algorithm once and leaves it up to subclass to implement the behavior that varies.

Template Method Structure

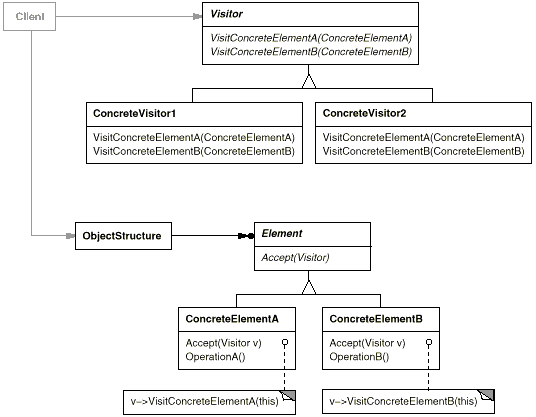


The AbstractClass provides the virtual primitive operations that subclasses implement. It defines the skeleton of an algorithm. The concreteClass implements the primitive operations to carry out its specific step of the algorithm. It is important to note that the Abstract class hierarchy can branch in many different directions. Template methods also must specify which operations may be overridden versus which are abstract operations that must be overridden. This is done in C++ by using virtual and pure virtual functions. Virtual functions may be overridden while pure virtual functions must be overridden.

**Visitor**

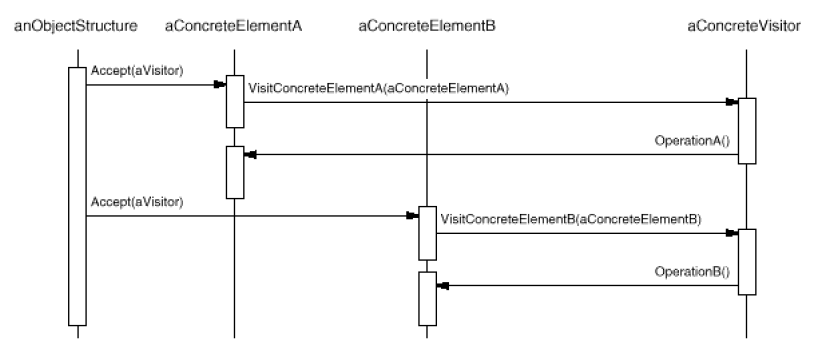
The Visitor method allows you to define new operations without changing the classes of the elements it works on. The name of this pattern is a bit of a misnomer, as its purpose is not about visitation, it is about providing a way to design hierarchies so that new virtual-acting functions can be added without changing class hierarchies. This is called double dispatching and it allows a class to be implemented in its standard way, and to also implement additional functions based off the visitor hierarchy definitions. The visitor pattern is a way to double dispatch, in languages that do not natively support it such as C++. This allows a class to be referenced by its class and the secondary Visitor condition, which will invoke different virtual-like functions based off which visitor is used.

Visitor Structure



The design is split into two hierarchies. The element hierarchy is the standard class hierarchy while the visitor hierarchy enhances the functionality of the element class. The Visitor class can implement new operations without changing the Element’s structure. The only change necessary to the Element class is to add the Accept(Visitor) method. This function takes in the visitor, and then sends itself to the Concrete visitor class. The visitor then has a reference to the ConcreteElement, and is able to perform new functionality. Moreover, the concreteElement can be visited by any number of ConcreteVisitors, which allows double dispatch functionality in the C++ language.

Visitor Interaction Diagram

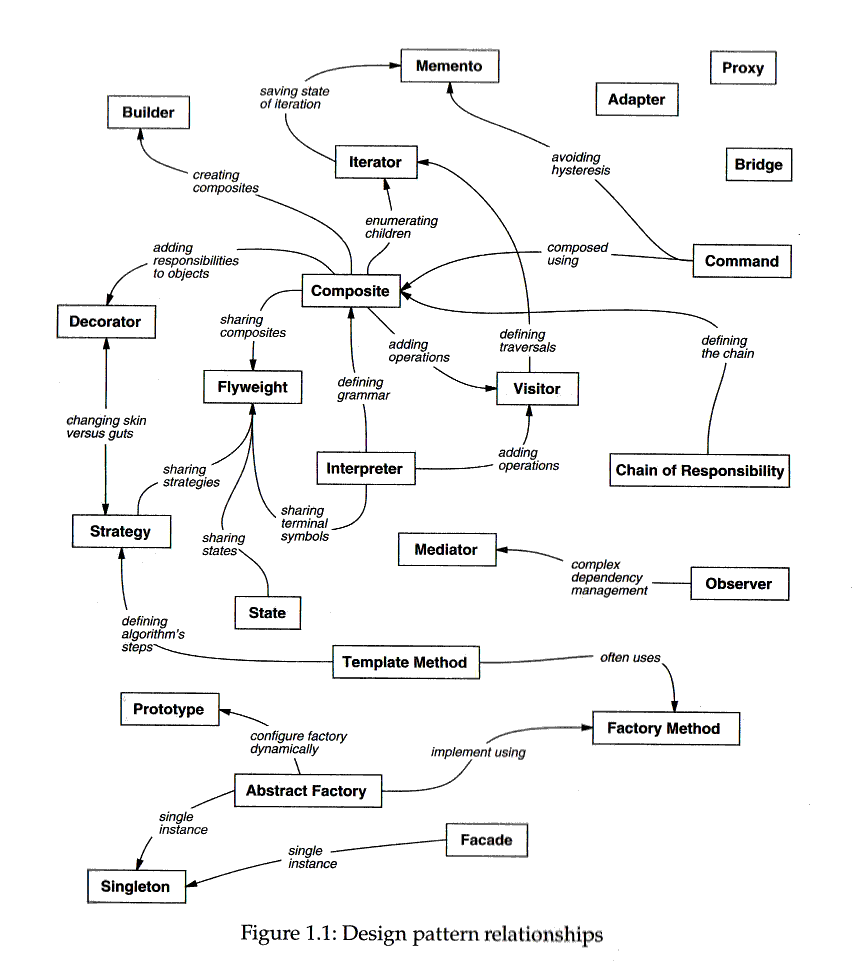


The interaction diagram illustrates how a object can implement new operations without changing its classes by accepting the visitor, and sending a reference of itself to the visitor class, which then performs the new operations.

1. Conclusion

The study and cataloguing of design patterns provide a common vocabulary for software designers to communicate, document, and explore design alternatives. Understanding the concepts behind design patterns makes it easier to understand the way existing systems operate. This catalogue illustrates the concepts behind good software design with a focus on objects, inheritance, and polymorphism. Even if the best design patterns were used to create a system, there will eventually be new constraints, requirements, and business objectives for the code. Great design cannot prevent this, but it can help the designer determine what changes need to be made to support new requirements. Refactoring or reorganization of the code structure must be performed at some point; knowing the limitations and benefits of your design provide targets for this refactoring.

**Design Pattern Relationships**



Works Cited

Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 2015.