Design Patterns

Ben

October 9, 2023

Contents

Chapter 1

Intro To Design Patterns

1.1 What?

Design Patterns unlike libraries or frameworks are not off the shelf tools you can copy and paste into your code to carry out a specific function, instead they are similar to building blueprints, where they show you a way to structure your solution to your problem.

1.2 Why?

Design Patterns are a group of tested solutions to common problems in software design, so when a particular problem arises you know how to structure your solution. Design Patterns also remove the programming language barriers when discussing a solution to a problem, instead of having to know how to implement a solution in Python or C++ you can just say “ This problem needs to use the singleton pattern ” and it doesn’t matter how its implemented the solution should form that pattern.

Chapter 2

Creational

Creational design patterns are used to manage how a developer creates objects by making the creation of an object suite a solution to a certain problem and reduce overall complexity.

2.1 Factory Method

A pattern for when function calls and properties need to be different and come from different objects but give the same result. For example a Button.Click() function should carry out the same functionality regardless if its a Windows button or Mac button but the creation of the button will differ and will be created through either a windowsButton.create() function or a macButton.Create function.

2.2 Abstract Factory

This Factory Takes the Factory method a step further, where the Factory method would have a Mac or Windows button factory the abstract factory would have a Mac or Windows UI factory which would inherit from an abstract class that would define all the elements each factory would have to create be it a button, scroll bar, cursor.

2.3 Builder

The Builder pattern removes the setup of a class from a constructor and uses class methods to build up a class in module components, eg a house class, when instantiated, would come with no windows or doors, but through a method like House.AddWindows() windows would be added to the house, these can also be added as extension methods with the first element as the object and as they also return the object they can be chained together.

2.3.1 Pros and Cons

Pros

1. Makes clear how an object is being constructed.
2. Easy to create two objects the same way.
3. Complex construction code can be removed from the business logic of an object making it easier to adhere to the Single Responsibility Principle.

Cons

1. Due to multiple new classes needed to be created to construct and object the overall code complexity can increase significantly.

2.4 Prototype

Prototypes allow a developer to quickly clone an existing class by using a clone function eg Object.Clone(). This works by the class constructor taking an argument of its own class of which it copies across all the properties of the existing class to the new class, the clone function then creates a new object using this constructor and returns the new object. This is useful when a developer would want to create lots of copies of an object without having to pass in every property to a constructor or reassign each property manually.

2.5 Singleton

The Singleton pattern is used when a developer only wants one instance of a class to exist in the program at any one time. This pattern is mainly used to access a shared resource like a service or a database. This Pattern is one of the easiest to implement, all it requires is a method to check if an instance of that object already exists, if it does return that instance, if not create a new instance and return it.

2.5.1 Pros and Cons

Pros

1. The Singleton pattern ensures that a class has only a single instance, offering precise control over instantiating objects.
2. It provides a global access point to the sole instance of the class, simplifying data sharing and access.
3. Objects in the Singleton pattern are initialised only when first requested, optimising resource usage.

Cons

1. The Singleton pattern often violates the Single Responsibility Principle, as it combines multiple responsibilities within a single class, potentially leading to architectural complexity.

2. Special handling is required in multi threaded environments to prevent the simultaneous creation of multiple Singleton instances, introducing additional complexity.

3. Unit testing client code utilising the Singleton pattern can be challenging due to the private constructor and the difficulty of overriding static methods in most programming languages, potentially discouraging comprehensive testing.

Chapter 3

Structural

3.1 Adapter

The Adapater design pattern allows two objects with incompatible interfaces to work with each other. For example if an input to a piece of software is a JSON file however it needs to be exported as an XML document that code would need to adapt one object to another through a JSON to XML adapter.

3.2 Bridge

The Bridge design pattern is a way of managing abstractions and implementation of a monolithic class that has multiple variants of a specific functionality. For example a class that manages database communication might have multiple ways of getting information from a database depending on if it is communicating with a Mongo database or a T-SQL database. This pattern can be used by having a class which taken an interface in its constructor so that each method on the class can call the interfaces implementation of any varied logic, and the surrounding generic logic can be kept in the original class.

3.3 Composite

The Composite pattern is a way of using a tree structure of objects as if it was a single object. Similar to a military hierarchy the Composite pattern works by giving instructions to the object highest in the hierarchy and have the command propagate down to the lowest node to gather the information needed. The Composite design pattern works by utilising two main object types, composites and leaves, a composite object can contain either more composite objects or leaves where a leaf object is a single object and the last node in the tree structure, for example a folder on a computer can contain more folders and files, this would be the composite, whereas a file cannot contain more object within it, just information about itself, this would be the leaf. Both objects would implement a function of the same name from an interface, eg getFileNames, if getFileNames is called on a composite it will loop through all of its nested objects and call getFileNames, if getFileNames is called on a leaf it will return the file name. This way if getFileName is called on the parent object all children object information will be gathered.

3.4 Decorator

The Decorator pattern allows a developer to add extra functionality to a class without inheriting the class or changing the class itself. This is achieved via a decorator class inheriting the same interface as the class that is being extended, the decorator class calls the class it is extended in any functions it is inheriting from the interface however it will also do the extended functionality as well. The most common example of when to use the decorator class is the pizza problem, with normal inheritance an IPizza class would be created and every type of pizza would inherit from it, however you then need to create a class for each type of pizza and every combination of toppings, the decorator pattern fixes this by dynamically being able to add toppings via a IToppingDecorator interface which the toppings would inherit from. this decorator would store a temporary version of the pizza class which it will add to each time a new topping is added.

3.5 Facade

Facade is a very simple pattern, as it is a way of hiding complex subsystems behind simple function calls, most libraries will use a facade as their user facing entry point, for example if a developer adds a library to deserialise JSON files to an object, the developer would typically just call JsonLibrary.convery(JsonFile) and not worry about how any of the subsystems in the library work.

3.5.1 Pros and Cons

Pros

1. Code complexity is kept separate to where it is used, allowing easier integration.

Cons

1. A facade can become a god object within an application due to it being a simple entry point to all functionality requiring it reference all classes.

3.6 Flyweight

The Flyweight pattern is used to reduce memory usage when a large amount of objects need to be stored which share common properties, it is similar to how foreign keys work in relational databases, there are two types of information required for the flyweight pattern, intrinsic, data which is common across multiple instances of a class, and extrinsic, data which is unique per instance. The flyweight pattern is used via creating a factory which checks when a new instance of a class is created if the intrinsic values in that class already exist and if they do, use the other object to get that data and dont store the data again, if the intrinsic data doesn't exist, create a whole new object. An example would be creating an in memory music database, for each song the common data would be the artist, album and artwork, with the unique data being the song name, then a reference to the album data is stored with the unique song data so that only one instance of the album data exists and is shared among all instances of the song data in that album. without the flyweight pattern each song would create and store a new album object with identical data in.

3.7 Proxy

The Proxy design pattern is most commonly used when communicating with a third party service or database, it is used via creating an interface which maps directly to the functionality of the service that is being proxied, for example if a database GetAll function would have a corresponding GetAll function on the interface, a concrete class of the proxy interface would then implement the GetAll function which would call the GetAll function of the database service however it allows the developer to add extra functionality around the service calls, like lazy loading or logging. Due to the service and the proxy class having the same functions and interface the proxy can therefore be dropped in anywhere the original service would be.

Chapter 4

Behavioral

4.1 Chain of Responsibility

The Chain of Responsibility pattern is used when a program is expected to process different kinds of requests in various ways, but the sequence and type of request is unknown. This pattern works via each class in the chain accepting the next class in the chain as a parameter as to where to pass the information being passing into it next, for example if you have three handlers HandlerOne.Next(HandlerTwo), HandlerTwo.Next(HandlerThree) etc. then when HandlerOne is called the next class in the chain gets called straight after. this can be used in many places such as bubbling events in UI where if a button gets clicked it can cause multiple different things to happen.

4.2 Command

The Command pattern allows requests to be their own object so that they can be delayed, queued or undone while also reducing repetitive logic if the same request can be carried out in multiple ways. This pattern requires a command object to store the requests and how to execute them, a receiver which can take multiple commands as input and transform their behaviour if needed and finally an invoker which takes a command and executes it.

4.3 Iterator

The iterator pattern allows a collection of elements to be iterated over without exposing the underlying representation eg list array etc.. This pattern can be used in c via the Ienumberable interface. The iterator class will encapsulate all of the ways of traversing a collection so that no matter what the underlying representation is the interface handles the traversal meaning any collection can be used in a function that accepts a class using the iterator interface

4.4 Mediator

The Mediator pattern is used to decouple dependencies between lots of objects and instead they all communicate via a mediator class instead. Objects communicate to the mediator class via notifications which the mediator can then use to notify other classes about information/state etc being updated. For example, on a website if on a login page the register button is clicked, instead of that buttons logic communicating with multiple different services in order to register the user, change the page to the profile page and send a validation email, the button logic is just to send the notification to the AuthenticationMediator class that the button has been pressed then the mediator class notifies all of the other services needed to carry out the authentication logic.

4.5 Momento

The Momento design pattern lets you manage the previous state of a object without revealing details of its implementation. It works via utilising two classes, a Memento class, where the previous state will be stored and a caretaker class, which will manage switching the current state out for one of the memento states and handling the list of all mementos. The memento class will take in the required information to a form a 'state' in its constructor, it will then have a getState function to return the information needed to restore that state. The memento class can also contain metadata which will allow the caretaker class to identify certain mementos, such as a name and date. The caretaker class will then have methods which apply the mementos state to the original object and will ideally store all the saved memtos in a list so that it can get the latest memento and on a successful state restoration remove that memento from the history list.

4.6 Observer

The Observer pattern is similar to the Mediator pattern where one class can notify many classes about a change in state, however the observer pattern only goes one way, unlike the mediator pattern which its dependants can communicate back to it, the observer pattern allows multiple objects to observe the state of one object, a typical use case would be the settings object on an application, many other classes would observe the state of the settings object so that when it is update they can also update to reflect the new changes in the settings. This pattern is implemented via the addition of an observer and a subject class, the subject class being the object to which all the classes observing it depend on. The subject class should allow observers to attach and detach themselves, and when the subjects state changes it should call a method, eg update, on all the classes observing it. The observer class will be inherited via and class wishing to observe the subject, typically this observer class would be an interface with an update method requiring to be implemented, this update class will be called by the subject class so that each class observing can react to the subjects states updating.

4.7 State

The State pattern allows a classes behaviour to change when its internal state changes. This pattern can be implemented by isolating areas in which an objects behaviour changes depending on its current 'state' and abstracting that behaviour to new classes, this way when that behaviour is called next it will just invoke the active states behaviour without doing any other checks. For example, if a class is communicating with a database, to eliminate race conditions in an application if one user is editing a file another user should be locked out of editing that file, this can be achived by have two different states for the editor, a unlocked and locked state, both of these classes inherit from the IEditorState interface so they have the same methods, such as save, edit, load, new file, on the locked interface on new and load file works, however when unlocked all functions work, the Editor object state will be either class and when the user tries to interact with a file if the file is locked state.savefile will not do anything however if the state is unlocked when state.savefile is called the file will be saved. this allows the editor object to not have to worry about the current state as the state classes will handle any difference in behaviour while the Editor class deals with the common behaviour of all states.

4.8 Strategy

The Strategy pattern is similar to the State pattern as it abstracts the behaviour of an object so that it can dynamically change depending on other factors, however in the strategy pattern instead of the behaviour being handled internally the behaviour is passed into the object within the constructor. For example if an object is used to calculate Taxes on a object that is passed into it, the taxes.calculate function adds common tax for example 20 percent, then it calls strategy. execute, which if the sugar tax strategy has been passed to the object will add another 10 percent however if its alcohol tax its another 20 percent.

4.9 Template Method

The Template Pattern takes advantage of abstract methods within classes to allow a developer to only extend certain steps of an algorithm but not the algorithm in its entirety or its structure. Its similar to the strategy pattern however as the Template Method is based on inheritance this is static where Strategies can be decided at runtime. For example, taking the example of the strategy class, where before we would pass in the strategy to the constructor the SugarTax and AlcoholTax would be two sepreate classes which inherit the Taxes abstract class, which when Taxes.Calculate would be called the Calculate method would: first add the 20 percent base tax, second call addExtraTaxes, this addExtraTaxes method would be an abstract method which the sugar and alcohol classes would implement. Therefore instead of calling Taxes.Calculate the program would call either SugarTax.Calculate or AlcoholTax.calculate which would call the base class calculate method with their respective addExtraTax implementation.

4.10 Visitor Pattern

The visitor pattern is a way to separate functionality from the objects which they operate on. For Example if you have three items, all which would have different tax, like sugary food, alcohol and bread a developer could create a Tax visitor class which each of the item classes would accept via a visitor accept method, the tax visitor class would then contain the logic of what to do when each of the classes are passed into it.