Contents

[Design Patterns 3](#_Toc87203179)

[Memento Design Pattern 3](#_Toc87203180)

[Structure: 3](#_Toc87203181)

[Problem: 3](#_Toc87203182)

[Participants: 3](#_Toc87203183)

[When to use: 4](#_Toc87203184)

[Template Method 4](#_Toc87203185)

[Structure: 4](#_Toc87203186)

[Participants: 4](#_Toc87203187)

[Problem: 4](#_Toc87203188)

[Coding Achievements: 4](#_Toc87203189)

[Implementation Issues: 5](#_Toc87203190)

[Prototype Design Pattern 5](#_Toc87203191)

[Structure: 5](#_Toc87203192)

[Participants 5](#_Toc87203193)

[Problem: 6](#_Toc87203194)

[Alternate structure: 6](#_Toc87203195)

[Abstract Factory 7](#_Toc87203196)

[Structure: 7](#_Toc87203197)

[Participants: 7](#_Toc87203198)

[Clarification: 7](#_Toc87203199)

[Strategy Design Pattern 8](#_Toc87203200)

[Structure: 8](#_Toc87203201)

[Participants: 8](#_Toc87203202)

[Improvements Achieved: 8](#_Toc87203203)

[Disadvantages: 8](#_Toc87203204)

[State Design Pattern 9](#_Toc87203205)

[Structure: 9](#_Toc87203206)

[Participants: 9](#_Toc87203207)

[Explained: 9](#_Toc87203208)

[Improvements achieved: 9](#_Toc87203209)

[Disadvantages: 9](#_Toc87203210)

[Implementation Issues: 10](#_Toc87203211)

[Composite Design Pattern 10](#_Toc87203212)

[Structure: 10](#_Toc87203213)

[Participants: 10](#_Toc87203214)

[Problem: 10](#_Toc87203215)

[Implementation Issues: 11](#_Toc87203216)

[Decorator Design Pattern 11](#_Toc87203217)

[Structure: 11](#_Toc87203218)

[Participants: 11](#_Toc87203219)

[Difference between Composite and Decorator: 12](#_Toc87203220)

[Code Improvements Achieved: 12](#_Toc87203221)

[Observer Design Pattern 12](#_Toc87203222)

[Structure: 12](#_Toc87203223)

[Participants: 12](#_Toc87203224)

[Implementation Issues 13](#_Toc87203225)

[Iterator Design Pattern 13](#_Toc87203226)

[Problem: 13](#_Toc87203227)

[Structure: 14](#_Toc87203228)

[Participants: 14](#_Toc87203229)

[Improvements achieved: 14](#_Toc87203230)

[Disadvantages: 14](#_Toc87203231)

[Implementation Issues: 15](#_Toc87203232)

[Additional Functionality: 15](#_Toc87203233)

[Mediator Design Pattern 15](#_Toc87203234)

[Structure: 16](#_Toc87203235)

[Participants: 16](#_Toc87203236)

[Problem: 16](#_Toc87203237)

[Purpose: 16](#_Toc87203238)

[Improvements Achieved 16](#_Toc87203239)

[Implementation Issues: 16](#_Toc87203240)

[Command Design Pattern 17](#_Toc87203241)

[Problem: 17](#_Toc87203242)

[Structure: 17](#_Toc87203243)

[Participants: 17](#_Toc87203244)

# Design Patterns

## Memento Design Pattern

* Behavioural
* Delegation
* Without violating encapsulation, capture and externalise an object’s internal state so that the object being stored can be restored to this state later

### Structure:

Diagram

Description automatically generated

* Client works with originator but must instantiate the caretaker
* The caretaker has a memento
* Originator instantiates the memento

### Problem:

* Allows for an undo implementation to occur in the system

### Participants:

* Originator:
  + Object which creates a snapshot to be stored
  + Can be restored
* Memento:
  + Takes snapshots of the Originator
  + Only Originator can access the state
* Caretaker:
  + Keeps memento safe
  + Unaware of the structure of the memento

### When to use:

Need to keep information intact for use for a later state, and does not impact the performance of the system

## Template Method

* Uses public inheritance to provide polymorphic operations
* Introduces refactoring (reduced duplication of code)
* Derived classes are a is-a base class
* Behavioural
* Inheritance
* Define a skeleton of an algorithm in an operation, deferring some steps to the sub-classes
* Let subclasses redefine certain steps of an algorithm without changing the algorithm’s structure

### Structure:

Diagram

Description automatically generated

### Participants:

* Abstract Class:
  + Implements template method defining skeleton of algorithm, calls many primitive operations
  + Defines abstract primitive operations that will appear in the template method
* Concrete Class:
  + Implements primitive operations to carry out the class specific steps of the algorithm

### Problem:

Different components have significant similarities but demonstrate no reuse of common interface or implementations.

### Coding Achievements:

* Reduced code duplication
* Algorithm is easier to maintain
* System is easier to extend
* Coupling is reduced because classes that call the template method is separated from the concrete classes

### Implementation Issues:

* Virtual methods should be declared as private so only the template method can only call these methods

Diagram

Description automatically generated

## Prototype Design Pattern

* Basic concept of caching
* Improves performance of a system
* Copy existing objects (deep copy and not shallow copy)
* Deep Copy Assign the same data but to a different pointer so the pointers can work independently
* Copy Constructor – const parameter so it cannot change anywhere
* Creational
* Delegation
* Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype

### Structure:

Diagram

Description automatically generated

Client works with the Prototype Interface, The subclasses are different types of prototypes

### Participants

* Prototype:
  + Declares interface for cloning itself
* Concrete Prototype:
  + Implements an operation for cloning itself
* Client:
  + Creates a new object by asking a prototype to clone itself

### Problem:

Constructure contains computationally expensive time-consuming procedures. This can be avoided by creating a copy of the object rather than going through the entire creation process each time from the beginning.

### Alternate structure:

* Add a prototype manager:
  + Maintains a list of existing objects that can be used as prototypes that are used when spawning new object
  + Allows for creating and removing prototypes

Diagram

Description automatically generated

* Prototype Manager:
  + Has an aggregate of Prototypes
  + Has a add, remove, get Prototype methods
* Prototype:
  + Has a clone method that returns a prototype
* Concrete Prototypes:
  + Implement the cloning methods

## Abstract Factory

* Produces a product with a common theme
* Creational
* Delegation
* Provide an interface for creating families of related or dependent objects without specifying the concrete classes

### Structure:

Diagram

Description automatically generated

### Participants:

* Abstract factory:
  + Creates interface to produce abstract product objects
  + Has abstract methods on which product
* Concrete Factor:
  + Implements the interface for product objects
* Abstract Product:
  + Provides interface for creating products
* Concrete Product:
  + Implements the abstract operations that produce objects that are created by the corresponding Concrete Factory
* Client:
  + Makes use of both Abstract Factory and Abstract Product
  + Needs to know about both interfaces

### Clarification:

* Abstract Factory comprises of concrete factories that produce concrete products

## Strategy Design Pattern

* Controlled Coupling
* Behavioural
* Delegation
* Define a family of algorithms, encapsulate each one, and make them interchangeable
* Let the algorithm vary independently from clients that use it

### Structure:

Diagram

Description automatically generated

Context has-a Strategy. Strategy has an algorithm that is overridden in subclasses. Concrete strategies override the algorithm defined in Strategy

### Participants:

* Strategy:
  + Declares interface common to all supported algorithms
  + Context uses this interface to call the algorithm defined by the concrete strategy
* Concrete Strategy:
  + Implements the algorithms defined in the Strategy interface
* Context:
  + Configured with a Concrete Strategy object (Not an aggregate)
  + Maintains a reference to a Strategy object
  + May define an interface that lets Strategy access its data

### Improvements Achieved:

* Different classes provide different implementations for same routine, there is not impact on system when one of the concrete classes change

### Disadvantages:

* Interface needs to cater for everything that is needed by the different Concrete Factories

## State Design Pattern

* Behavioural
* Delegation
* Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class

### Structure:

Diagram

Description automatically generated

Context has a state which calls a concrete’s handle request in request(). Concretes inherit from State

### Participants:

* State:
  + Defines an interface for encapsulating the behaviour associated with a particular state of the Context
* Concrete State:
  + Implements behaviour associated with a state of the Context
* Context:
  + Maintains an instance of a Concrete State that defines the current state
  + Defines the interface of interest to clients

### Explained:

System needs to change its behaviour based on its state. Applies polymorphism to define different behaviours

### Improvements achieved:

* Increased maintainability
  + All behaviour of a state in one object
* Eliminate large conditional statements
* Makes state transitions explicit

### Disadvantages:

* Higher Coupling:
  + Increase the number of classes and is less compact than a single class and will require more communication between classes that would be the case of all was implemented in a single class

### Implementation Issues:

* Who decides which class will be responsible for implementation of the change of state?
  + Context applying fixed Criteria
    - Context will be responsible for determining the next state to change to
  + Context applying variable criteria
    - Context holds a state
    - State holds a context
  + Concrete states applying variable criteria
    - Concrete States are responsible for changing state

## Composite Design Pattern

* Builds a structure of related objects in the form of a tree
* Sequence containers: vectors, dequeues, lists
* Associative containers: sets, multisets, maps, multimaps
* Structural
* Delegation
* Compose objects into a tree structure to present part-whole hierarchies. Composite lets clients treat individual objects and compositions of object uniformly

### Structure:

Diagram

Description automatically generated

The composite has a Component.

### Participants:

* Component:
  + Provides the interface with which the client interacts
* Leaf:
  + Do not have children, define primitive objects of the composition
* Composite:
  + Contains children that are either composite or leaves
* Client:
  + Manipulates the objects that comprise the composite

### Problem:

* Used in hierarchies where some objects are composite of other. Makes use of a store for the children defined by Composite

### Implementation Issues:

* Destruction of the composite
  + Make it the clients responsibility
    - Objects must be defined in the client
  + Leaves the destruction on the Composite
    - Allows for anonymous objects to be passed in

## Decorator Design Pattern

* Used to extend functionality of an object without changing the physical structure of the object
* Extension is either to elaborate the state or behaviour
* Done at compile time
* Structural
* Delegation
* Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

### Structure:

Diagram

Description automatically generated

Decorator has 0 or 1 component. Holds a component reference. Virtual operation function

### Participants:

* Component:
  + Interface for objects that can have responsibilities dynamically added to them
* Concrete Component:
  + The object to which the additional responsibilities can be attached
* Decorator:
  + Defines a reference to a Component
* Concrete Decorator:
  + Extends the Decorator interface

### Difference between Composite and Decorator:

* Composite:
  + Comprises of multiple components
  + Composite class is concrete
  + Builds a tree structure
* Decorator:
  + May or may not have component
  + Decorator is abstract
  + Only a list data structure

### Code Improvements Achieved:

* Additional features added onto an object, is not embedded into the object

## Observer Design Pattern

* One to many relationships between subjects and observers
* Single subject may have many observers
* When the subject changes state, all the observers are notified and updated automatically
* Works well in multithreaded environment
* Behavioural
* Delegation
* Define a one-to-many relationship between objects so that when one object changes state, all its dependents are notified and updated automatically

### Structure:

Diagram

Description automatically generated

Subject has a list of observers. Concrete Observer affects Concrete Subjects

### Participants:

* Subject:
  + Provides interface for observers to attach and detach to the concrete subject
* Concrete Subject:
  + Implement the subject being observed
  + Store objects that are observing it and sends update notifications to these objects
* Observer:
  + Defines interface of objects observing subject
  + Provides means in which observers are notified regarding change to the subject
* Concrete Observer:
  + Maintain reference to the subject it observers
  + Updates and stores relevant state information of the subject to keep consistent with the state of the subject

### Implementation Issues

* Detaching an observer:
  + Destructor in Concrete Observer must call the relevant detach function as defined in the Subject hierarchy
  + Destructor must be declared virtual in Concrete Observer, so object calls the parent destructor, and the detachment of the observer takes place
* Transfer of state to the observer:
  + Push:
    - Subject takes responsibility by packaging its public state and providing the observers with this when issuing an update function.
  + Pull:
    - Requires subject to provide a public interface by which the observer can request specific state information from the subject

## Iterator Design Pattern

* Aggregate is used for a collection of objects
* Applies separation of concern
* Applies decoupling of concern
* The iteration of objects should be applied to a separate hierarchy so the aggregate can focus on maintenance of objects
* Functionality to traverse if decoupled from maintenance of aggregate
* Behavioural
* Delegation
* Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

### Problem:

Must have the iterator and aggregate interfacing transparently

### Structure:

Diagram

Description automatically generated

Client needs to know about both the Aggregate and Iterator. Aggregate has an Iterator reference that is implemented in the Concrete Aggregate. Iterator provides an interface for traversing the Aggregate. Concrete Aggregate instantiates the iterator.

### Participants:

* Iterator:
  + Defines interface for accessing and traversing elements
* Concrete Iterator:
  + Implements the Iterator interface
  + Keeps track of current position in the traversal of the aggregate
* Aggregate:
  + Defines interface for creating an Iterator object
* Concrete Aggregate:
  + Implements the Iterator creation interface to return an instance of the proper concrete Iterator

### Improvements achieved:

* Iterator simplifies the Aggregate interface – separation of concern
* Iterators contribute to flexibility of code
* Iterators contribute to the reusability of code
* Easy to provide different ways in iterating through objects
* Possible to execute simultaneous yet independent iterations through the same structure

### Disadvantages:

* Difficult to have the iterator sync with the aggregate
* Make the iterator a friend class

### Implementation Issues:

* Concrete iterators must be able to access the elements in the aggregate
* Can be done in 4 different ways:
  + Make a copy of the aggregate inside the iterator
    - Most robust solution
    - Not memory efficient
    - Cannot see immediate updates on aggregates
  + Create an object storing the state of the aggregate inside the iterator
    - Storing a memento
    - Not easy to implement
    - Cannot see immediate changes
  + Keep pointer to aggregate inside the iterator and use a call back mechanism to access the elements of the aggregate:
    - Memory efficient
    - Method calls need to be public in aggregate, or iterator can be declared friend class in the aggregate
    - This will reflect real time changes
  + Use the pimp principle:
    - Most efficient
    - Memory efficient

### Additional Functionality:

* First(), next(), isDone(), currentItem()
* Remove(): remove current item from aggregate
* Previous(): should step backwards instead of forwards to enable iterations that can go in separate directions
  + Need additional pointers: prev
* skipTo(): position iterator to an object matching the specific criteria

## Mediator Design Pattern

* Extends the observer pattern
* Difference between observer and mediator:
  + Observer:
    - Registers observers to a subject that get updated whenever there is a change
  + Mediator:
    - Registers colleagues that get updated whenever one of the other colleagues notifies the mediator of an update
* Behavioural
* Delegation
* Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently

### Structure:

Diagram

Description automatically generated

Mediator owns a Colleague as a parameter. Colleague has abstract methods, get and set. Concrete Mediator keeps track of aggregate of Colleagues, overrides the Mediator function.

### Participants:

* Mediator:
  + Defines interface for communicating with Colleague objects
* Concrete Mediator:
  + Implements cooperative behaviour by coordinating Colleague Objects
  + Knows and maintains its colleagues
* Colleague:
  + Each colleague knows about its Mediator
  + Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

### Problem:

* Design reusable components
* Reuse only one or a few of the classes in a group of classes
* Difficult to isolate them

### Purpose:

* Responsible for controlling and coordinating the interactions of a group of objects
* Avoid proliferating interconnections by encapsulating the interconnections in a separate mediator object

### Improvements Achieved

* Simplification of code updates:
  + Allow you to change only a specific class/object instead of the all of them
* Increased reusability of code:
  + Increases individual code coherence
* Simplification of object protocol:
  + Converts many-to-many to one-to-many relationship

### Implementation Issues:

* Change():
  + Implemented in Colleague interface, allows each Concrete Colleague to call it (not a pure virtual method)
  + Notifies the mediator of changes
  + Each Concrete Colleague calls this method whenever it makes a change
    - Mediator->notify(this);
* Notify():
  + Called every time there is a change in the Concrete Colleague
  + Pointer of the Concrete Colleague is sent as a reference so the mediator knows which colleague called it
  + Looks a little like this:
    - Mediator::notify(Colleague\* originator){
      * resultOfChange = originator->get();
      * for(all colleague){
        + set(resultOfChange);
      * }
    - }

## Command Design Pattern

* Behavioural
* Delegation
* Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queues or logs, and support undoable operations

### Problem:

* Used to modify interfaces to make it work after it has been designed

### Structure:

Diagram

Description automatically generated

Invoker has a Command. Command has an execute method. Concrete Command has a state which is used to make changes based off the value it holds. Implements the execute method

### Participants:

* Command:
  + Declares an interface for executing an operation
* Concrete Command:
  + Defines binding relationship between Receiver object and an action
  + Implement execute()
* Client(Application):
  + Creates a Concrete Command object and sets its Receiver
* Invoker:
  + Asks the command to carry out the request
* Receiver:
  + Knows how perform the operations associated with carrying out a request
  + Any class can be the receiver