Software Engineering

Iterator

- *Behavioural Pattern*
- Problem : An aggregate collection of objects
- Aim : To access items in collection, without exposing their internal structure. Access independent of collection, multiple access allowed independently.
- Solution : Iterator object controls access.
- Types of iterators:
    - Embedded
    - Separate
    - Internal - *Iterator controls transversal*
    - External - *Collection controls transversal*
- Problems with embedded iterators :
    - Mixed structure and maintenance
    - One transversal at a time
    - Adds to collection interface and implementation
    - Clumsy when adding different collection algorithms
- Separate iterator attached to client at time of creation or via a method
- Pros of separate iterator:
    - Supports various transversal algorithms
    - Simplified collection interface
    - Multiple transversals
- Internal Iterator:
    - **Iterator controls transversal**
    - Client hands operation to perform on each element in aggregate
    - Iterators transverses aggregate
- External Iterator:
    - **Collection controls transversal**

Builder

- *Creational Pattern*
- Problem : Need to construct complex objects incrementally.
- Aim : To abstract the construction process.
- Solution : Parameterized construction of many variants through concrete builders which the client can invoke for each piece.
- Use When?
    - Need algorithm/assembly independence.
    - Do not need specificity.
- Variants:
    - Construction fixed
    - Construction incremental, etc.
- Builder structure diagram
- Flow :
    - Client creates Director, configures it with a Builder.
    - Director notifies Builder when a part is to be added to Product.
    - Builder adds parts to Product.
    - Builder returns Product to Client.
- Pros/features:
    - New types of products, common process
    - Specifics of assembly process are hidden
    - **Process** of construction encapsulated in Director
    - **Product** of construction encapsulated in Builder
    - Can vary both, for an independent Builder
    - Iterative, step by step, product retrieved when Director has finished construction
- Changes to Director:
    - Must recognize new types of parts
    - Must know how to invoke construction of part in builder
- Changes to Builder:
    - Add new part construction method to interface and update subclases

Command

- *Behavioural Pattern*
- Problem : Multiple commands issued through various mechanisms
- Aim : Decouple invoking of action from knowledge of how to perform it, knowledge of the receiver of request
- Solution : Create Command objects that know how to execute operation
- Pattern Structure : *Command, Concrete Command, Client, Invoker, Receiver*
- Consequences:
    - Invocation decoupled from execution
    - Command extensible
    - Macro commands accomodated
    - Easy to add new commands to interface
- How to undo/redo
    - Memento Pattern
    - Prototype Pattern

Template Method

- *Behavioural Pattern*
- Problem : Repetitive code for different classes with similar structure is not desirable
- Aim : To reuse code
- Solution : Create a template class
- Defines skeleton of an algorithm in a method
- Some specifications and steps deferred to subclasses
- Tea Coffee example
- Allows subclass to redefine certain methods without changing algorithm's structure
- Class diagram
- Hollywood principle

State

- *Behavioral Pattern*
- Problem : Object responds differently depending on its state, all responses may not be valid at all times, states are not extensible
- Aim : To optimise object response with state dependence
- Naive Solution: Variable based conditional response -- extensibility issues, logically complex
- Solution : Basic interface defined, different subclass for each state, context accesses concrete subclass based on state
- Class Diagram -- *Context, State Concrete State*
- Use When?
    - Object's behavior depends on state which changes at runtime
    - Operations with multi-parts, conditional on object state
- Consequences:
    - Localized and partitioned state specific behaviour
    - Easy addition/deletion of states

Strategy

- *Behavioral Pattern*
- Problem : Bunch of closely related algorithms
- Aim : To abstract implementation from client, encapsulate for interchangeable usage
- Solution : Common interface for each group of algorithms
- Class Diagram -- *similar to State Pattern, Context, Strategy, Concrete Strategies*
- Applicability:
    - Hide algorithm specific data structures
    - Eliminate multiple conditional testing
- Benefits:
    - Encapsulated family of algorithms

    - Eliminate conditional switching
    - Dynamic choice for context
- Drawbacks:
    - Client must be aware of strategies
    - Same interface for all strategies, simple ones may have overhead of information

Chain of Responsibility

- *Behavioral Pattern*
- Problem : Distribution of responsibility and handing it
- Aim : To best distribute responsibility among objects and optimise responsibility handing
- Solution : Chain of objects that examine a request, handle it or pass it to other objects
- Structure Diagram -- *Client, Handler, Concrete Handlers*
- When to use
    - To decouple a request's sender and receiver
    - Hide specific handlers
    - Issue request without specifying receiver explicitly
- Benefits:
    - Decoupling of sender and receiver
    - Simplifies objects
    - Addition, removal of responsibilities dynamically
- Drawbacks:
    - Execution of request isn't guaranteed
    - Hard to observe and debug runtime

Mediator

- *Behavioral Pattern*
- Problem : Interaction and interdependence of different objects can get messy

- Aim : to facilitate inter-object interaction and make it easy, every object need not be aware of the existence of every other object
- Solution : A new object that encapsulates object interactions
- Structure Diagram -- *Mediator, Colleagues*
- **Mediator and Observer are competing patterns**
    - Observer distributes communication by introducing 'Observer' and 'Subject' objects, Mediator encapsulates communication between other objects
    - Mediator can leverage Observer by dynamically registering colleagues
- **Mediator is similar to Facade**
    - Abstracts functionality of existing classes
    - ***Difference :*** Mediator defines a multi-directional protocol, while Facade defines a unidirectional protocol

Memento

- *Behavioural Pattern*
- Problem : Restoration of object state to previous state by client
- Aim : Preserve encapsulation of state privacy while allowing for state restoration
- Solution : Have object create a Memento of its current state for restoration
- Pattern Structure -- *Memento, Originator, Caretaker*
- Consequences:
    - Encapsulation maintained
    - Mementos can be expensive/large
    - Caretaker may need overhead on storage
- Similar to Command pattern, only here token implies internal state
- Used in conjunction with Iterator, iterator uses memento to capture the state of the iteration

Decorator Pattern

- *Structural Pattern*
- Adds additional responsibility to an object dynamically by wrapping itself around the object, has the same interface as the object.
- Object's responsibilities are delegated before or after decorator responsibilities.
- Structure Diagram -- *Component, Decorator, Concrete Component, Concrete Decorator*
- Adapter provides different interface, proxy provides same interface, decorator provides enhanced interface
- **Composite and Decorator both rely on recursive composition**, have similar structure diagrams
- Decorator changes skin, Strategy changes guts of an object