benefits of behavioral patterns

* increasing system flexibility
* increasing development efficiency
* streamlining documentation
* promoting communication

Architectural Patterns

* depict fundamental structural organization for software systems
* responsible for how these apps will use shared resources, interact with each other, and cater to the client’s request
* Layered pattern- presentation, business, persistence, database layers (all 4 layers decoupled and independent) persistence layer picks data that is needed by business layer
* Master-slave pattern – we designate responsibility of managing requests through the master and the master designates the task execution to the appropriate slave
* MVC pattern- used for apps where the user interface is important- decouple into 3 components: view is used by users to prepare requests, controller calls the appropriate business logic and modifies the real data, after change the model will automatically persist in the db

GoF Design Patterns

* creational
* structural
* behavioral

Creational Patterns

* factory method- ensures that we are decoupling construction policy from the object, if there is a change in the construction policy, we can just change the implementation in the factory rather than changing the client code
* abstract factory
* builder patterns
* prototype
* singleton

Behavioral Patterns

* designed in order to help us identify how common communication can exist among different objects to complete a particular business use case. we can use to increase the flexibility of a system in order to ensure seamless communication between different objects
* command- encapsulates a command request as an object and can then change it
* mediator- defines or simplifies communication between various classes
* strategy patterns – takes responsibility of providing multiple algorithms for the same task
* template-method pattern- simplifies an app by providing the exact steps of an algorithm that we want to include as a subclass and then we can reuse
* interpreter
* iterator
* visitor

Structural Patterns

* simplifies design of our application
* helps identify how we can realize relationships among different objects in a simple way
* adapter pattern- provides capability of using incompatible interfaces, good for adding on to an existing system with different(updated) technology
* façade pattern – provides a unified interface which can communicate with a set of interfaces in some subsystem
* decorator pattern- for dynamically adding additional responsibilities to an object
* proxy – prevents direct access to an object and uses a proxy instead
* bridge – decouples the abstraction from the implementation
* composite
* flyweight

Specification Patterns

* specify business rules
* aggregation with other business rules
* used for domain-driven design
* problems are criteria-based selection of objects, only suitable objects used

Model-View-Presenter (MVP) Pattern

* to build inactive and robust user interface
* model- business objects of the domain problem
* view- content of the model, interacts with users’ events regarding the UI
* presenter- interprets the users’ events, communicates with the model objects, implements the main workflow of the application

Model View and View Model (MVVM) Pattern

* architectural pattern separating GUI from the development of business logic
* for a shared environment where all developers work simultaneously
* want reusable components that can be used across projects
* user interface customization with no change in code
* needs thorough testing

Integration Patterns

* collection of around 65 patterns
* solves common integration problems

Lazy Load Patterns

* concept of delaying object loading
* improves performance of system by deferring initialization of objects
* objects are loaded as and when needed
* capable of memory optimization
* provides quick outcomes
  + virtual proxy – memory saving technique, recommends postponing an object creation until it is needed
  + lazy initialization – checks the value of a class field when its being used, field gets loaded with null value until request when null value is replaced by actual value
  + value holder – generic object holding lazy loading behavior, appears in place of the objects data fields, call “GetValue” method to check the value of value holder, at that time (and only then), the value gets loaded from a db or service

Event Aggregator

* aggregates events from multiple objects into a single object
* used for multiple objects of potential event sources
* simplifies event process architecture
* simplifies integration
* works as a hub

Service Locator Pattern

* capable of obtaining a service through a central registry known as service locator
* decouples components from dependencies
* replacing a service will not impact entire architecture
* enables independent applications development
* enables independent service registration

Creational Patterns

Builder Patterns

* common construction process to create different objects
* compile complex representation
* create one of several targets
* separates complex object construction from representation
* good for simplifying a complex build by extracting the construction codes and making it its own class

Factory Patterns

* factory function to deal with object construction
* define interface for object construction
* allow subclasses to decide the class
* virtual constructor definition
* eliminates problem of “new” which externalizes construction
* implements creation of an object without exposing the instantiation logic

Behavioral Patterns

Command Pattern

* capability of requesting objects (without know about the operation being requested, or the receiver of the request)
* parameterize clients with diversified request processing needs
* need of undo/redo operations

Interpreter Pattern

* define domain-specific language for a different language
* representation of grammar with interpreter
* domain mapped with language, language with grammar

Mediator Pattern

* promotes mediation between various objects where they can refer to each other explicitly and provide abstraction in terms of their interaction
* mediator object encapsulating communication between objects
* promotes loose coupling
* defines intermediary to decouple
* many-to-many relationship between interacting parties

Null Object Pattern

* encapsulate absence of object by providing a mock that provides suitable default indicating “do nothing” behavior
* for when an object needs a collaborator
* or for abstract handling null

Observer Pattern

* defines and identifies dependency between objects
* dependents are notified when the object state changes
* object state change is notified to all the dependents
* encapsulate the main components in a subject abstraction

State Pattern

* used when an object changes behavior based on its internal state
* allows object to alter behavior when internal state changes
* defines different “states” of the state machine
* always maintains current state

Strategy Pattern

* provides the capability of selecting from multiple implementation of strategies
* selection with changing implementation
* defines a family group of interchangeable algorithm
* captures abstraction in interface
* defines implementation in derived classes

Visitors Pattern

* separates logic from the object structure on which it operates
* contains operation to be performed on elements of object structure
* information recovery
* double dispatch

AntiPatterns – solutions that generate negative outcomes and consequences

1. Development antipatterns
2. Architecture antipatterns
3. Project Management antipatterns

* functional decomposition
* poltergeists
* golden hammer
* cut/paste
* vendor lock-in