Katie Dillan

Python Design Patterns Notes:

Design patterns are reusable solutions to common problems encountered during software development. They provide structured approaches for solving specific design challenges, promoting code maintainability, flexibility, and scalability. In Python, various design patterns can be applied to address different aspects of software design and development.

Creational Design Patterns

Factory Method Pattern:

Provides an interface for creating objects, but allows subclasses to alter the type of objects created.

Use cases: Object creation without specifying the exact class, dynamic object creation based on runtime conditions, eg smal big medium chair.

Inheritance = is a , has its own functional behavior

Abstract Factory Pattern:

Creates families of related or dependent objects without specifying their concrete classes.

Use cases: Creating families of objects, supporting multiple platforms or implementations, enforcing consistent interfaces for object creation.

Singleton Pattern:

Ensures a class has only one instance and provides a global point of access to that instance.

Use cases: Managing global state, controlling access to shared resources, implementing logging, configuration settings.

Builder Pattern:

Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Use cases: Creating complex objects with many optional parameters, avoiding telescoping constructors.

Prototype Pattern:

Creates new objects by copying an existing object, reducing the need for subclassing.

Use cases: Creating objects with similar initial states, avoiding repeated instantiation and initialization logic.

Structural Design Patterns

Adapter Pattern:

Allows incompatible interfaces to work together by providing a bridge between them.

Use cases: Integrating with third-party libraries, legacy system integration, interface standardization.

Bridge Pattern:

Decouples an abstraction from its implementation, allowing both to vary independently.

Use cases: Separating abstraction and implementation, managing platform-specific implementations.

Proxy Pattern:

Acts as a surrogate for another object, controlling access to it. It's used for remote, expensive, or sensitive operations, caching, access control, logging, and lazy initialization

Use cases: implementing virtual proxies for resource-intensive objects, protection proxies for access control, remote proxies for communication, logging proxies for debugging, and caching proxies for performance optimization.

Composite Pattern:

Composes objects into tree structures to represent part-whole hierarchies.

Use cases: Representing hierarchies of objects, treating individual and composite objects uniformly.

Decorator Pattern:

Attaches additional responsibilities to an object dynamically, providing a flexible alternative to subclassing.

Use cases: Adding functionality to objects without modifying their structure, promoting modularity and extensibility.

Facade Pattern:

Provides a simplified interface to a complex subsystem, hiding its complexity from clients.

Use cases: Simplifying complex APIs, providing a unified interface to subsystems.

Behavioral Design Patterns

Observer Pattern:

Defines a one-to-many dependency between objects, allowing multiple observers to be notified of changes in a subject.

Use cases: Implementing event handling, decoupling observers from subjects, pub-sub architectures.

Strategy Pattern:

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Use cases: Selecting algorithms at runtime, separating algorithms from clients, enabling algorithmic flexibility.

Command Pattern:

Encapsulates a request as an object, allowing parameterization of clients with queues, requests, and operations.

Use cases: Implementing undo/redo functionality, queuing requests, supporting composite operations.

Iterator Pattern:

Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.

Use cases: Traversing collections, decoupling clients from collection implementations, providing a uniform interface for iteration.

Mediator Pattern:

Defines an object that encapsulates how a set of objects interact, promoting loose coupling.

Use cases: Managing communication between components, centralizing interaction logic, reducing dependencies between components.

These design patterns provide structured approaches to solve common software design challenges. By understanding their principles and applications, developers can create more maintainable, flexible, and scalable Python applications.

1. Retry Pattern:

Description: The Retry Pattern enables the automatic retrying of failed operations, typically in scenarios where transient errors are expected.

Use Cases:

- Handling network or database connectivity issues.
- Integrating with unreliable external services or APIs.
- Managing asynchronous tasks that may fail due to temporary issues.

2. Flyweight Pattern:

Description: The Flyweight Pattern focuses on efficient memory usage by sharing common, immutable objects among multiple contexts.

Use Cases:

- Text processing tasks where a large number of identical strings or tokens are used.
- Graphical user interface components, such as icons or fonts, where memory efficiency is crucial.
- Optimizing performance in systems dealing with a vast number of lightweight objects.

Interpreter Pattern

Overview: The Interpreter Pattern is used to define a grammar for interpreting sentences in a language, and it provides a way to evaluate language grammar or expressions. In Python, it can be implemented using the ast module for parsing and interpreting abstract syntax trees.

Uses:

- Implementing DSLs (Domain Specific Languages).
- Processing and evaluating expressions or rules dynamically.

Decimal Pattern

Overview: The Decimal Pattern is a design approach for handling precise decimal arithmetic, especially when dealing with financial or monetary calculations. In Python, the decimal module provides support for decimal floating-point arithmetic.

Uses:

- Financial calculations requiring precise results.
- Applications where floating-point arithmetic may introduce rounding errors.

3. Visitor Pattern:

Description: The Visitor Pattern facilitates the traversal of a complex object structure while allowing different operations to be performed on its elements without modifying their classes.

Use Cases:

- Parsing or analyzing abstract syntax trees (ASTs) in compilers or interpreters.
- Document processing systems where different operations need to be applied to various elements of a document structure.
- Representing complex data structures where the behavior of elements needs to vary independently.

4. Template Pattern:

Description: The Template Pattern defines the skeleton of an algorithm in a superclass, with specific steps implemented by subclasses.

Use Cases:

- Defining common workflows or processes with varying implementation details.
- Implementing code generation frameworks where the overall structure remains constant, but specific parts vary.
- Providing a reusable blueprint for creating similar objects or processes with minor differences.

5. Model View Controller (MVC) Pattern:

Description: The MVC Pattern separates an application into three interconnected components: Model, View, and Controller, to promote modularity and maintainability.

Use Cases:

- Web development frameworks, such as Django or Flask, where MVC architecture organizes application logic and presentation separately.
- Graphical user interface frameworks, where the separation of concerns simplifies development and enhances code reusability.
- Large-scale applications requiring scalability and maintainability, where clear separation of concerns is essential.

6. Circuit Breaker Pattern:

Description: The Circuit Breaker Pattern helps in building resilient systems by preventing cascading failures and providing fault tolerance.

Use Cases:

- Integrating with external services or APIs, where repeated failures can overload the system.
- Implementing microservices architectures, where failures in one service can affect others.
- Managing distributed systems or cloud-based applications to handle temporary outages or degraded performance.

7. Cache-Aside Pattern:

Description: The Cache-Aside Pattern involves caching frequently accessed data in a separate cache layer to improve performance and reduce database load.

Use Cases:

- Accelerating read-heavy applications by caching database query results or computed values.
- Scaling systems with high read-to-write ratios, where caching can significantly reduce latency.
- Reducing the load on backend services or databases, particularly in systems with predictable data access patterns.

8. Throttling Pattern:

Description: The Throttling Pattern limits the rate at which requests are processed or resources are accessed to prevent overload and ensure fair resource allocation.

Use Cases:

- Preventing denial-of-service (DoS) attacks by limiting the number of requests a client can make within a specified time frame.

- Managing system resources, such as CPU or memory, to avoid degradation of service during peak loads.
- Implementing rate limiting in APIs or web services to enforce usage quotas and prevent abuse or excessive consumption.