Design Patterns Cheat Sheet

A quick reference guide to common software design patterns, categorized by their intent: creational, structural, and behavioral. Includes pattern descriptions, use cases, and implementation notes to help you apply them effectively in your projects.

Creational Patterns

Singleton

Intent:	Ensure a class only has one instance and provide a global point of access to it.
Use Case:	Managing resources like database connections or configuration settings.
Implementation Notes:	Private constructor, static method to access the instance. Thread safety is a key consideration.
Example (Python):	see code below

```python
class Singleton:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance
```

Factory Method

Intent:	Define an interface for creating an object, but let subclasses decide which class to instantiate. Promotes loose coupling.
Use Case:	Creating objects of different types based on runtime configuration or user input.
Implementation Notes:	Abstract creator class with a factory method, concrete creators that override the method to return specific product types.
Example (Java):	see code below

```java
interface Product {}
class ConcreteProductA implements Product {}

interface Creator {
    Product factoryMethod();
}
class ConcreteCreatorA implements Creator {
    public Product factoryMethod() {
        return new ConcreteProductA();
    }
}
```

Abstract Factory

Intent:	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Use Case:	Supporting multiple look-and-feels in a GUI or working with different database systems.
Implementation Notes:	Abstract factory interface, concrete factories for each family, abstract products, and concrete products.
Example Scenario:	Imagine creating a GUI factory that can produce Windows or MacOS specific UI elements (buttons, text fields, etc.).

Structural Patterns

Adapter

Intent:	Allow incompatible interfaces to work together. Acts as a wrapper converting the interface of a class into another interface clients expect.
Use Case:	Integrating legacy systems with new systems or using third-party libraries with different interfaces.
Implementation Notes:	Adapter class implements the target interface and holds an instance of the adaptee. Methods in the adapter call corresponding methods in the adaptee.
Example:	Adapting a Fahrenheit temperature sensor to a system that expects Celsius.

Decorator

Intent:	Dynamically add responsibilities to an object without modifying its structure. Provides a flexible alternative to subclassing for extending functionality.
Use Case:	Adding logging, caching, or security features to an object at runtime.
Implementation Notes:	Decorator class implements the same interface as the component it decorates and holds an instance of the component. It adds extra behavior before or after calling the component's methods.
Example:	Adding borders or scrollbars to a GUI component.

Facade

Intent:	Provide a simplified interface to a complex subsystem. Hides the complexities of the subsystem from the client.
Use Case:	Simplifying the use of a complex library or framework.
Implementation Notes:	Facade class provides simple methods that delegate to the underlying subsystem components.
Example:	A `Compiler` facade that simplifies the process of compiling code by hiding the individual steps of lexical analysis, parsing, and code generation.

Behavioral Patterns

Observer

Intent:	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Use Case:	Implementing event handling systems or model-view-controller (MVC) architectures.
Implementation Notes:	Subject (observable) maintains a list of observers. When the subject's state changes, it notifies all registered observers.
Example:	A stock ticker application where multiple displays (observers) update when the stock price (subject) changes.

Strategy

Intent:	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Use Case:	Implementing different sorting algorithms or payment processing methods.
Implementation Notes:	Strategy interface defines the algorithm. Concrete strategy classes implement specific algorithms. Context holds a reference to a strategy object.
Example:	Allowing a user to choose between different compression algorithms (e.g., ZIP, GZIP) when saving a file.

Template Method

Intent:	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Use Case:	Implementing a build process where some steps are common and others are specific to different types of projects.
Implementation Notes:	Abstract class defines the template method, which calls abstract and concrete methods. Concrete classes implement the abstract methods to provide specific behavior.
Example:	Implementing a report generation process where the steps of loading data, formatting data, and outputting data are defined, but the specific formatting and output methods are different for different report types.

Advanced Concepts

Anti-Patterns

These are patterns that are commonly used but are ineffective and often lead to negative consequences.

Examples:

- **God Object:** A class that knows too much or does too much.
- **Spaghetti Code:** Code that is difficult to read and trace.
- **Copy-Paste Programming:** Duplicating code instead of using proper abstraction.

GRASP Principles

Information Expert:	Assign responsibility to the class that has the information needed to fulfill it.
Creator:	Assign responsibility of object creation to the class that contains or closely uses the created objects, or that has the initializing data.
Low Coupling:	Design classes with minimal dependencies on other classes.
High Cohesion:	Keep related responsibilities grouped together in the same class.
Polymorphism:	Use polymorphism to handle variation based on type.
Protected Variations:	Protect elements from the variations by wrapping them with an interface.
Pure Fabrication:	Assign a high cohesion set of responsibilities to an artificial class that does not represent a problem domain concept.
Controller:	Assign the responsibility of receiving or handling a system event to a class that is not a UI class.

SOLID Principles

Single Responsibility Principle (SRP):	A class should have only one reason to change.
Open/Closed Principle (OCP):	Software entities should be open for extension, but closed for modification.
Liskov Substitution Principle (LSP):	Subtypes must be substitutable for their base types.
Interface Segregation Principle (ISP):	Clients should not be forced to depend upon interfaces that they do not use.
Dependency Inversion Principle (DIP):	Depend upon abstractions, not concretions. High-level modules should not depend on low-level modules.