

Software Design Patterns

Software Design Patterns – Quick Reference Notes

1. What Is a Design Pattern?

- **Definition:** A reusable, proven solution to a common software design problem.
 - **Purpose:** Improves code readability, promotes best practices, and speeds up development.
 - **Characteristics**
 - Language-agnostic (conceptual, not code-specific)
 - Described by name, intent, problem, solution, and consequences
 - Organized into families (Creational, Structural, Behavioral)
-

2. Pattern Classification

Category	Goal	Typical Use Cases
Creational	Abstract object creation, hide instantiation details	Managing complex object creation, controlling lifecycle
Structural	Organize classes/objects to form larger structures	Simplifying relationships, adding responsibilities
Behavioral	Define communication between objects	Managing algorithms, workflows, and object interaction

3. Creational Patterns

Pattern	Intent	Core Idea	Typical Implementation	Pros	Cons
Singleton	Ensure a class has only one instance	Private constructor + static accessor	<code>static getInstance()</code>	Global access, lazy init	Hard to test, hidden dependencies

Software Design Patterns

Pattern	Intent	Core Idea	Typical Implementation	Pros	Cons
Factory Method	Define an interface for creating an object, let subclasses decide which class to instantiate	Subclass overrides a <code>create()</code> method	Abstract creator ? concrete creators	Open/Closed, decouples client	More complex
Abstract Factory	Provide an interface for creating families of related objects without specifying concrete classes	Factory of factories	GUIFactory ? WindowsFactory / MacFactory	Consistent families, easy swapping	Complex hierarchy
Builder	Separate construction of a complex object from its representation	Step-by-step builder with fluent API	<code>new Builder().setA().setB().build()</code>	Readable, immutable objects	More complex, simple objects
Prototype	Create new objects by copying an existing prototype	Clone method (shallow/deep)	<code>prototype.clone()</code>	Fast creation, dynamic types	Cloning pitfalls (mutable state)

4. Structural Patterns

Pattern	Intent	Core Idea	Typical Implementation	Pros	Cons
---------	--------	-----------	------------------------	------	------

Software Design Patterns

Adapter	Convert one interface to another	Wrapper class translates calls	<code>class SocketAdapter implements Target { ... }</code>	Reuse existing code	Extra indirection
Bridge	Decouple abstraction from its implementation	Separate hierarchy for abstraction & implementation	Abstraction ? Implementor	Independent evolution	More classes
Composite	Treat individual objects and compositions uniformly	Tree structure with common component interface	Component ? Leaf / Composite	Simplifies client code	Can be over-engineered
Decorator	Add responsibilities dynamically	Wrap object with additional behavior	<code>new ConcreteDecorator(new ConcreteComponent())</code>	Flexible, avoids subclass explosion	Can lead to many small classes
Facade	Provide a simplified interface to a subsystem	Single class aggregates complex calls	<code>class SubsystemFacade { ... }</code>	Reduces coupling	May become a "god object"
Flyweight	Share fine-grained objects to support large numbers	Intrinsic (shared) vs. extrinsic (context) state	<code>FlyweightFactory.get(key)</code>	Memory efficiency	Complexity in managing state
Proxy	Control access to another object	Placeholder that forwards requests	<code>RemoteProxy, VirtualProxy,</code>		