# **Patterns Summary** *(99% from the Gang of 4 book)*

## General
A pattern is a design that has been systematically named, explained, and evaluated.
Normally talked about in an object-oriented context.

In a structural context "Encapsulation" would be a pattern.

A pattern consits of a name, a problem that it's intended for, a solution, and it's consequences.
Some patterns may be substitutes or they may be complementary in different situations.

## Classification

Patterns are classifed as Creational, Structural, and Behavioral.
Patterns are further classiffied at the class level and object level.

### Creational Patterns

*Abstract Factory*

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

*Builder*

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

*Factory Method*

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. Also Known As Virtual Constructor.

*Prototype*

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

1. when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
2. to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
3. *when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.*

*Singleton*

Ensure a class only has one instance, and provide a global point of access to it.

**Structural Patterns**
Structural patterns are concerned with how classes and objects are composed to form larger structures.

*Adapter*

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Also Known As Wrapper.

*Bridge*

It has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an existing object. Also Known As Handle/Body. For example, a collection may be implemented as a linked-list for smaller sizes, and then as the size grows it gets switched to a hash table. However the client sees one interface only without knowing which implementation is being used.

**Important**: Here there is no “interface mismatch” as in the case of an Adapter. There is one and only one interface exposed by the object which is used by the client. The bridge merely gives us the capability to switch out implementations at run-time.

*Composite*

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. For example, a drawing program might want to treat a primitive such as a “line” object the same way as it treats a more complex “picture” object (which may itself be composed of many primitives and/or complex objects) uniformly by just telling them to “draw” themselves.

*Decorator*

It enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters. One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.  A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a decorator. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities. The Decorator is like a wrapper that wraps up the object. Here too there is no interface mismatch. The decorator exposes the same interface as the object it wraps (unlike the wrapper in the case of Adapter where its sole purpose is to address interface mismatch and not to provide any additional functionality on top of what the object has). How about those Java Stream classes, cool huh.

*Facade*

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

*Flyweight*

Use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between intrinsic and extrinsic state. Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it. (Stateless Objects in MTS!!!)

*Proxy*

It defines a representative or surrogate for another object and does not change its interface. Also called, what else, “surrogate”.

1. A remote proxy provides a local representative for an object in a different address space. Coplien [Cop92] calls this kind of proxy an "Ambassador."
2. A virtual proxy creates expensive objects on demand.
3. A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.
4. A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include counting the number of references to the real object so that it can be freed automatically when there are no more references (also called smart pointers [Ede92]), loading a persistent object into memory when it's first referenced, checking that the real object is locked before it's accessed to ensure that no other object can change it.

**Behavioral Patterns**
Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's dif ficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected. Behavioral class patterns use inheritance to distribute behavior between classes. Behavioral object patterns use object composition rather than inheritance.

*Chain of Responsibility*

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. How about the Java 1.1 Event handling model where you can chain listeners, or those VisiBrokker ORB Interceptors, or a chain of virtual functions each calling the parent object's implementation.

*Command*

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. Also Known As Action or Transaction. For example, a button may be parameterized with a "Command" object upon which the button may call "Execute" when it is pushed. It may take a series of command objects.

*Interpreter*

Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

*Iterator*

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Also Known As Cursor. An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list. The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already. How about those STL iterators.

*Mediator*

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. Though partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again. Lots of interconnections make it less likely that an object can work without the support of others—the system acts as though it were monolithic. For example, a dialog box that manages consitancy relationship between different controls in the dialog box would be an example of a mediator. All controls communicate a change in their state to the dialog box and the dialog box in turn communicates this to other controls that need to know. Looks like our Relationship manager.

*Memento*

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later. Also Known As Token. Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors. You must save state information somewhere so that you can restore objects to their previous states. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally. Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility. Mementos have two interfaces: a wide one for originators (the object that created the memento and stored its state in it) and a narrow one for other objects. Ideally the implementation language will support two levels of static protection. C++ lets you do this by making the Originator a friend of Memento and making Memento's wide interface private. Only the narrow interface should be declared public.

*Observer*

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Also Known As Dependents or Publish-Subscribe. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state. This kind of interaction is also known as publish-subscribe. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications. Eg. Connection points, CORBA Event Service. The relationship may be push-pull or pull-push.

*State*

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. It resembles a convention state machine scenario with the difference that each state will have a state object associated with it. Most often impleneted as singletons and flyweights.

*Strategy*

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. Also Known As Policy. Most often impleneted as singletons and flyweights.

*Template Method*

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. For example each step of the algorithm in the template may be a “virtual” function which are redefined, or the template may be parameterized taking objects implementing the steps as it customizing parameters. The template makes sure that the steps are called in the right order.

*Visitor*

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. Use the visitor pattern when an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes OR when  many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.