**Design Patterns**

Table of Contents

[1. Facts about Design Patterns 2](#_Toc154489554)

[2. Gang of Four Design Patterns 3](#_Toc154489555)

[2.1 Creational Design Patterns 3](#_Toc154489556)

[2.1.1 Factory Method 3](#_Toc154489557)

[2.1.2 Abstract Factory 4](#_Toc154489558)

[2.1.3 Singleton 5](#_Toc154489559)

[2.1.4 Builder 5](#_Toc154489560)

[2.1.5 Prototype 6](#_Toc154489561)

[2.2 Structural Design Patterns 7](#_Toc154489562)

[2.2.1 Adaptor 7](#_Toc154489563)

[2.2.2 Bridge 8](#_Toc154489564)

[2.2.3 Composite 9](#_Toc154489565)

[2.2.4 Decorator 9](#_Toc154489566)

[2.2.5 Façade 10](#_Toc154489567)

[2.2.6 Flyweight 11](#_Toc154489568)

[2.2.7 Proxy 12](#_Toc154489569)

[2.3 Behavioural Design Patterns 13](#_Toc154489570)

[2.3.1 Interpreter 13](#_Toc154489571)

[2.3.2 Template method 14](#_Toc154489572)

[2.3.3 Chain of responsibility 15](#_Toc154489573)

[2.3.4 Command 15](#_Toc154489574)

[2.3.5 Iterator 16](#_Toc154489575)

[2.3.6 Mediator 17](#_Toc154489576)

[2.3.7 Memento 18](#_Toc154489577)

[2.3.8 Observer 19](#_Toc154489578)

[2.3.9 State 20](#_Toc154489579)

[2.3.10 Strategy 20](#_Toc154489580)

[2.3.11 Visitor 21](#_Toc154489581)

[3. Architectural Principles 22](#_Toc154489582)

[3.1 Increases Cohesion 22](#_Toc154489583)

[3.2 Reduces Coupling 22](#_Toc154489584)

[3.3 Divide and Conquer 23](#_Toc154489585)

[3.4 Increases Abstraction 23](#_Toc154489586)

[3.5 Increases Reusability 23](#_Toc154489587)

[3.6 Increases Reuse using API’s (Interfaces) 23](#_Toc154489588)

[3.7 Increases Flexibility 23](#_Toc154489589)

[3.8 Anticipate Obsolescence 23](#_Toc154489590)

[3.9 Design for Portability 23](#_Toc154489591)

[3.10 Design for Testability 23](#_Toc154489592)

[3.11 Design Defensively 23](#_Toc154489593)

[4. Architectural Patterns 23](#_Toc154489594)

[4.1 Client-server 23](#_Toc154489595)

[4.2 Multi-layer 24](#_Toc154489596)

[4.3 Broker 24](#_Toc154489597)

[4.4 Transaction Processing 24](#_Toc154489598)

[4.5 Pipe and Filter 24](#_Toc154489599)

[4.6 Model-View-Controller 24](#_Toc154489600)

[5. Architectural Strategies 25](#_Toc154489601)

[6. Anti-patterns 25](#_Toc154489602)

# Facts about Design Patterns

Design pattern is a general repeatable solution to a commonly occurring problem in software design. It is a template about how to solve a problem; same template can be used in many different situations.

Consistency, clarity, time, and reuse are the pros of using design patters, whereas, anti-patterns, careful while choosing the pattern, negative consequences, subjective dependence on concrete scenarios, different interpretations are the cons in using design patterns.

**Now come to the Facts**

1. Design patterns are different ways to enforce the fundamental design principles.
2. Design patterns (DP) can speed up the development process by providing tested, proven development paradigms.
3. Reusing DP helps to prevent issues that can cause major problems (extensibility).
4. Improves code readability for coders and architects familiar with the patterns.
5. DP provides general solutions demonstrated in a format that does not require specific tie to a particular problem.
6. Patterns allow developers to communicate using well-known and well-understood names of software interactions.
7. Common DP can be improved over time, making them robust than ad-hoc designs.
8. DP use OO Concepts like decomposition, inheritance, and polymorphism to improve software development process.
9. DP provides a way to get benefit from the experience and knowledge of your predecessors those have worked on the same type of project.
10. Appropriate DP used in development of applications make development fast and easy documented.
11. DP are more sophisticated and advance approaches than basic data structures.
12. None of the DP describes anything about new or unproven design changes. They only include designs applied more than once in different systems.
13. DPs are part of the OO-community or DPs are the elements of some successful OO-systems.
14. DPs should be easy to learn for inexperienced developers.
15. DPs are not dealing with designing user interfaces.

**Posted in Medium:**

<https://medium.com/@ramisetty.kavya06/quick-read-on-design-patterns-793c1bdc4c5d>

# Gang of Four Design Patterns

**Erich Gamma, Richard Helm, Ralph Johnson,**and**John Vlissides written a book titled “Design Patterns: Elements of Reusable Object-Oriented Software” in 1994 which consists of 23 design patterns grouped under three categories.**

* **Creational DP: deals with object creation in a defined systematic manner.**
* **Structural DP: provides a** mechanism to organize the classes and objects for larger structures.
* **Behavioural DP: deals with communication between objects.**

**Patterns under each category is again sub-divided into either class scope patterns or object scope patterns. Class scope patterns are to be applied during design time, whereas object scope patterns to be applied during runtime.**

## Creational Design Patterns

**CLASS SCOPE: Factory method**

**OBJECT SCOPE: Abstract factory, singleton, builder, prototype**

### Factory Method

**Real-time scenario:** A car company like Hyundai has different cars like venue, Creta, i10, etc. It manufactures a certain type of car only based on the request from client (customer). Factory pattern is used when a class consists of many sub-classes but object creation for a particular class is done my a “Factory Method” based on the input from client.

|  |  |
| --- | --- |
| **Intent** | Factory method is like an interface for creating an object.  Subclasses decide which class to be instantiated for object creation based on the input from client. |
| **Also Known as** | Virtual constructor |
| **Motivation** | It encapsulates the client from having the knowledge about subclasses.  Abstract classes are used to define and maintain relationships between objects |
| **Applicability** | When a class wants its subclasses to specify the objects it creates.  When a class can’t anticipate the class of objects it creates. |
| **UML Structure** | enter image description here |
| **Participants** | Product: defines the interface of objects  Conc\_product: implements the product interface  Creator: call the factory method to create product object  Conc\_creator: overrides the factory method to return an instance of Conc\_product |
| **Collaborations** | Creator depends on its subclasses to define factory method so that it returns the instance of an appropriate Conc\_product |
| **Consequences** | Adds complexity to the code.  Tight coupling between the factory and concrete classes |
| **Implementation** | Naming convections  Case 1: when creator is abstract class and does not provide any implementation for the factory method.  Case 2: when creator is concrete class and provides a default implementation for the factory method |
| **Sample Code** |  |
| **Known Users** | Factory method pervades toolkits and frameworks. |
| **Related Patterns** | Abstract factory, prototype. |

### Abstract Factory

For example, the same company which manufactures cars, will also manufacture bikes, trucks, etc vehicles. Then the object creation for remaining vehicles is kind of similar to factory method of cars. This means that Abstract factory is one level above the factory method.

|  |  |
| --- | --- |
| **Intent** | Provide an interface for creating a family of related objects without specifying their concrete classes |
| **Also Known as** | kit |
| **Motivation** | The Abstract Factory pattern is useful when a client object wants to create an instance of one of a suite of related, dependent classes without having to know which specific concrete class is to be instantiated.  Abstract Factories are usually implemented using the Factory Method pattern. |
| **Applicability** | System should be independent of how its products are created, composed, represented (manufactured).  System should be configured with one of multiple families of products.  You want to reveal just their interfaces not their implementation.  Provide class library of products.  Family of related product objects is designed to be used together and you need to enforce this constraint. |
| **UML Structure** | UML Class diagram describing the AbstractFactory pattern |
| **Participants** | Abs\_factory: declares interfaces to create abstract product object  Conc\_factory: implements operations to create concrete product object  Abs\_product: declares interfaces for the type of product object  Conc\_product: defines product object to be created by the corresponding concrete factory  Client: uses only interfaces declared by Abs\_factory and Abs\_product |
| **Collaborations** | To create different product objects, client should use different conc\_factory  Conc\_factory creates single instance at run-time |
| **Consequences** | Supporting new kinds of products is difficult. |
| **Implementation** | Factories acts as singleton.  Creating the products implemented using prototype pattern.  Defines different operations for each kind of product |
| **Sample Code** |  |
| **Known Users** | To achieve portability across different window systems  Interviews uses “kit” to denote abstract factory classes |
| **Related Patterns** | Singleton, prototype, factory method |

### Singleton

Like president of India, only one object at a time.

|  |  |
| --- | --- |
| **Intent** | Class has only one instance.  Provides a global point of access |
| **Also Known as** | dispensers |
| **Motivation** | It is important for same classes to have exactly one instance.  Global variable makes an object accessible.  Make the class itself responsible for keeping track of its sole instance |
| **Applicability** | Exactly one instance and globally accessible  Client should be able to use an extended instance without modifying their code |
| **UML Structure** |  |
| **Participants** | Singleton: responsible for creating its own unique instance |
| **Collaborations** | Clients can access singleton instance solely through singleton instance operation |
| **Consequences** | Controlled access to instance  Reduced namespace.  Permits the refinements of operations.  More flexible than class operations |
| **Implementation** | Ensure a unique instance.  Subclass the singleton class |
| **Sample Code** |  |
| **Known Users** |  |
| **Related Patterns** | Abstract factory, builder, prototype |

### Builder

Burger from KFC

|  |  |
| --- | --- |
| **Intent** | Construction of a complex object from its representation so that the same construction process can create different representations |
| **Also Known as** | construction |
| **Motivation** | Same process for constructing in different representations.  Complexity is hidden |
| **Applicability** | Allow different representations for the object that is created.  The algorithm for creating a complex object should be independent of the parts that make up the object |
| **UML Structure** | Builder Structure |
| **Participants** | Builder: specifies abstract interface for creating parts of product objects  Conc\_builder: implements the builder interface to keep track of representation. Provides an interface for retrieving the product.  Director: constructs an object using the builder interface  Product: represents the complex object under construction |
| **Collaborations** |  |
| **Consequences** | Vary a product internal representation.  Isolates code for construction and representation.  Gives you finer control over the construction process |
| **Implementation** | Builder constructs their products in step-by-step fashion.  Empty methods as default in builders |
| **Sample Code** |  |
| **Known Users** | Common pattern in small talk  Classes use to create sub classes for themselves |
| **Related Patterns** | Abstract factory, composite |

### Prototype

Clones in JIRA

|  |  |
| --- | --- |
| **Intent** | Create new object by copying this prototype |
| **Also Known as** | Replicas, skeleton |
| **Motivation** | The prototype class is used to create new instance by copying or cloning into a subclass.  Need to use same object to create replicas.  Add new objects with slight variations |
| **Applicability** | System should be independent of how its products manufactured.  Avoid inheritance and parallelism.  Hides cloning process from users |
| **UML Structure** | 4: Prototype design pattern UML-like class diagram | Download Scientific  Diagram |
| **Participants** | Client: creates a new object  Prototype: declares interface for cloning  Conc\_prototype: implements an operation for cloning |
| **Collaborations** | Client asks prototype to clone itself |
| **Consequences** | Adding and removing product at runtime  Specify new object by varying values.  Specify new object by varying structure.  Configure your application with classes dynamically.  If clone already exist, it is difficult |
| **Implementation** | Use prototype manager.  Implement the clone operation.  Initialise the clone |
| **Sample Code** |  |
| **Known Users** | Graphical editor |
| **Related Patterns** | Abstract factory |

## Structural Design Patterns

**CLASS SCOPE: Adaptor-class**

**OBJECT SCOPE: Adaptor-object, bridge, composite, decorator, façade, flyweight, proxy**

### Adaptor

Laptop mobile charger

|  |  |
| --- | --- |
| **Intent** | Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces. |
| **Also Known as** |  |
| **Motivation** |  |
| **Applicability** | You want to use an existing class and its interface does not match the one you need.  You want to create a reusable class that cooperates with unrelated classes.  You need to use several existing subclasses but its impractical to adapt their interface by sub-classing each. |
| **UML Structure** |  |
| **Participants** | Target: defines the domain specific interface that client uses.  Client: collaborates with objects conforming to the target interface.  Adaptee: defines an existing interface that needs adapting.  Adapter: adapts the interface of adaptee to the target interface. |
| **Collaborations** | Clients call operations on an adapter instance.  In turn, adapter calls adaptee operation that carry out the request. |
| **Consequences** | Class Adapters: adapter override some of adaptee’s behaviour. Adapter is a subclass of adaptee.  Object Adapters:harder to override adaptee behaviour. Adapters refer to the subclass rather than to the adaptee itself.   * Adapters vary in the amount of work they do to adapt adaptee to the target interface. * Class is more reusable when you minimise the assumptions other classes must make to use it. * Using two-way adapter to provide transparency. |
| **Implementation** | Implementing class adapters in c++  Pluggable adapters  Using delegate objects  Parameterised adapters |
| **Sample Code** |  |
| **Known Users** | Pluggable adapters are common in small talk. |
| **Related Patterns** | Bridge: like object adapter  Decorator: enhances another object without changing its interface  Proxy: defines a representative or surrogate for another object and does not change its interface |

### Bridge

Connecting two different classes, like cities in real time

|  |  |
| --- | --- |
| **Intent** | Decouple an abstraction from its implementation so that the two can vary independently. |
| **Also Known as** | Handle/body |
| **Motivation** | When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.  Concrete subclasses implement it in different ways.  It is not flexible enough always.  It is inconvenient to extend the window abstraction to cover different kinds of windows or new platforms. |
| **Applicability** | You want to avoid a permanent binding between an abstraction and its implementation.  Both the abstractions and their implementations should be extensible by sub-classing.  Changes in the implementation of an abstraction should have no impact on clients.  When you want to hide the implementation of an abstraction completely from clients.  You want to share implementation among multiple objects. |
| **UML Structure** | Bridge pattern - Wikipedia |
| **Participants** | Abstraction: defines the abstraction’s interface and maintains a reference to an object of type implementer.  Refined abstraction: extends the interface defined by abstraction.  Implementer: defines the interface for implementation classes. This interface does not have to correspond exactly to abstraction’s interface. Two interfaces can be quite different.  Conc\_implementor: implements the implementer interface and defines its concrete implementation. |
| **Collaborations** | Abstraction forwards client requests to its implementer object |
| **Consequences** | Decoupling interfaces and implementation  Improved extensibility  Hiding implementation details from clients |
| **Implementation** | Only one implementer  Creating the right implementer object  Sharing implementers  Using multiple inheritance |
| **Sample Code** |  |
| **Known Users** |  |
| **Related Patterns** | Adapter: used to make two unrelated classes work together. Usually applied after system is designed; Bridge does similar work, but it is done up front during design  Abstract factory: used to create the implementer and remove all knowledge of which one from the abstraction. |

### Composite

|  |  |
| --- | --- |
| **Intent** |  |
| **Also Known as** |  |
| **Motivation** |  |
| **Applicability** |  |
| **UML Structure** |  |
| **Participants** |  |
| **Collaborations** |  |
| **Consequences** |  |
| **Implementation** |  |
| **Sample Code** |  |
| **Known Users** |  |
| **Related Patterns** |  |

### Decorator

|  |  |
| --- | --- |
| **Intent** | Dynamically attaching additional responsibility to an object  Provide a flexible alternative to subclass for extending functionality |
| **Also Known as** | wrapper |
| **Motivation** | Sometimes we want to add responsibilities to individual objects not to an entire class.  One way to add responsibilities is within inheritance.  More flexible approach to enclose the component. This enclosing object is called decorator. |
| **Applicability** | Adding responsibilities to individual objects without affecting other objects.  Responsibilities can be withdrawn.  Extension by sub-classing is impractical.  Sometimes large no of independent extensions is possible. |
| **UML Structure** | Decorator pattern - Wikipedia |
| **Participants** | Component: defines interface for objects that can have responsibilities added.  Conc\_component: defines an object to which additional responsibilities can be attached.  Decorator: defines interface that conforms to components interface. Maintains a reference to a component object.  Conc\_decorator: adds responsibilities to the component. |
| **Collaborations** | Decorator requests component object performs additional operations before and after forwarding the request. |
| **Consequences** | More flexibility than static inheritance.  Avoids feature classes high up in the hierarchy.  Decorator and its components are not identical.  Lots of little objects. |
| **Implementation** | Interface performance.  Omitting the abstract decorator class.  Keeping component classes lightweight.  Changing the skin of an object vs changing its guts. |
| **Sample Code** |  |
| **Known Users** | Many object-oriented interface toolkits use decorators. |
| **Related Patterns** | Adapter: decorator only changes object responsibilities not its interfaces.  Composite: decorator can be viewed as a degenerate composite with only one component.  Strategy: decorator lets you change the skin of an object whereas strategy lets you change the guts. |

### Façade

ATM Machine

|  |  |
| --- | --- |
| **Intent** | Provide a unified interface to a set of interfaces in a subsystem. |
| **Also Known as** |  |
| **Motivation** | Reduce complexity.  Minimise the communication and dependencies between subsystems.  Façade object provides a single simplified interface to a more general facilities of subsystem. |
| **Applicability** | Provide simple interface to a complex subsystem.  There are many dependencies between clients and the implementation classes of an abstraction.  Façade to decouple the subsystem from clients and other subsystems.  Promote subsystem independence and portability. |
| **UML Structure** |  |
| **Participants** | Façade: which subsystem classes are responsible for a request. Delegates client request to appropriate subsystem objects.  Subsystem classes: implement subsystem functionality. Handles work assigned by façade object. No knowledge about façade. No reference. |
| **Collaborations** | Clients communicate with subsystem by sending requests to façade.  Façade forwards to appropriate subsystem.  Clients that use the façade don’t have to access its subsystem objects directly. |
| **Consequences** | Makes subsystem easier to use.  Shields clients from subsystem components.  Promotes weak coupling between the subsystem and its clients.  Façade helps layer a system and the dependencies between objects.  Eliminate complex dependencies.  It does not prevent apps from using subsystem classes if they need.  Choose between ease of use and generality. |
| **Implementation** | Reducing client subsystem coupling.  Public vs private subsystem classes. |
| **Sample Code** |  |
| **Known Users** | Application can have built in browsing tools for inspecting its objects at runtime.  The choices use façade to compose many frameworks into one. |
| **Related Patterns** | Abstract factory: provide interface for creating subsystem objects in a subsystem independent way.  Mediator: abstracts functionality of existing classes.  Façade objects are always singleton. |

### Flyweight

|  |  |
| --- | --- |
| **Intent** | Use sharing to support large no of objects efficiently. |
| **Also Known as** |  |
| **Motivation** | Flyweight is a shared object that can be used in multiple contexts simultaneously.  It cannot make assumptions about the context in which they operate.  Intrinsic: this state is stored in the flyweight. Consists of information that is independent of flyweight’s context. Makes it sharable.  Extrinsic: varies with the flyweight context. Can’t be shared. |
| **Applicability** | Application uses a large no of objects.  Storage costs are high.  Most of the object states are extrinsic.  Many groups of objects may be replaced by few shared objects.  Does not depend on object identify. |
| **UML Structure** | C# Flyweight Design Pattern |
| **Participants** | Flyweight: declares an interface through which flyweights can receive and act on extrinsic state.  Conc\_flyweight: implements flyweight interface object must be sharable. Any state it stores must be intrinsic. Must be independent of the conc\_flyweight object’s context.  Unshared concrete flyweight: not all flyweight subclasses need to be shared. Interface enables sharing. It is common for unshared conc\_flyweight. Objects to have conc\_flyweight objects as children at some level in object structure.  Flyweight factory: creates and manages flyweight objects ensure that flyweights are shared properly. When client requests a flyweight, flyweight factory object supplies an existing instance or creates new.  Client: maintains a reference to flyweight. Stores the extrinsic state of flyweight. |
| **Collaborations** | Flyweight state – intrinsic – conc\_flyweight.  Flyweight state – extrinsic – client objects.  \*Clients should not initiate conc\_flyweight directly. |
| **Consequences** | Reduction in the total no of instances that comes from sharing.  Amount of intrinsic state per object.  More flyweights are shared, greater the storage savings. |
| **Implementation** | The following issues can be considered:  Removing extrinsic state.  Managing shared objects. |
| **Sample Code** |  |
| **Known Users** | FT++ uses flyweights to support look and feel independence. |
| **Related Patterns** | Composite: to implement a logically hierarchical structure in terms of a graph. |

### Proxy

|  |  |
| --- | --- |
| **Intent** | External service request.  Need use of external object for creating the exact replica of server.  Provides identical interface to clients.  Implementation of proxy object keeps reference of another object to which they forward requests.  Provide a placeholder for another object to control access to it. |
| **Also Known as** | surrogate |
| **Motivation** | Proxy may hide information about the real object to the client.  Proxy may perform optimisation.  Creating objects even though they are not required may be expensive, so proxy allows us to postpone this creation until we need the actual object.  If object creation is complex or expensive then we can postpone that object creation. |
| **Applicability** | Proxy is applicable whenever there is a need for more sophisticated reference to an object (behaviour of an object depends on its state).  Remote proxy: provides a local representative in different address space.  Virtual proxy: creates expensive objects on demand.  Protection proxy: controls access to the original object.  Smart reference: replacement for a bare pointer that performs additional actions when an object is accessed.  Count the no. of references to the real object so that it can freed easily and automatically when there are no more references.  Loading a persistent object into memory.  Checking that real object is locked before it is accessed.  Ensure that no other object can change it. |
| **UML Structure** | A UML class diagram for the Proxy pattern - Learning Python Design Patterns  - Second Edition [Book] |
| **Participants** | Subject: defines a common interface for real subject and proxy so that proxy can be used anywhere a real subject is expected.  Real subject: defines the real object that the proxy represents.  Proxy: maintains the reference. Let the proxy access the real subject. Provides interface identical to the real subject so that proxy can be substituted for the real subject. Controls access to the real subject and may be responsible for creating and deleting it. |
| **Collaborations** | Proxy forwards requests to real subject when appropriate.  Depending on the kind of proxy. |
| **Consequences** | Introduces a level of indirection when accessing an object.  Uses of indirection:   * Remote proxy hides the fact that object reside in different address space. * Virtual proxy can perform optimisations such as creating and deleting. * Allows additional housekeeping tasks when an object is accessed. * Another optimisation that is hidden from client is “COPY ON WRITE” – copy a large and complicated object is expensive. If the copy is never modified, then waste. So, when the client requests an operation that modifies the subject then the proxy actually copy it. |
| **Implementation** | Overloading the member access operator in C++  This lets us to perform additional work whenever an object is de-referenced.  Helps in implementing some kinds of proxy just like pointer.  Using doesNotUnderstand in small talk. Proxy class (with no super class) can redefine doesNotUnderstand so that the message forwarded to the subject.  Proxy does not always have to know the type of real subject. |
| **Sample Code** | 1. **Remote proxy:** client-server example 2. **Virtual proxy:** image load example 3. **Protection proxy:** database login example |
| **Known Users** | In Microsoft OLEDB provides hid whether the server is local or remote from client.  www proxy runs on firewall machine and allows people inside the firewall provides concurrent access to the outside world. |
| **Related Patterns** | Adapter: provides a different interface to its subject but proxy provides the same interface.  Decorator: provides enhanced interface. |

## Behavioural Design Patterns

**CLASS SCOPE: Interpreter, template method**

**OBJECT SCOPE: Chain of responsibility, command, iterator, mediator, memento, observer, state, strategy, visitor**

### Interpreter

|  |  |
| --- | --- |
| **Intent** | Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the languages. |
| **Also Known as** |  |
| **Motivation** | The interpreter is one of the published patterns which is not really used. Usually, the interpreter pattern is described in terms of formal grammars like it was described in the original form but the area where this design pattern is used can be extended. |
| **Applicability** | The grammar is simple.  Efficiency is not a critical concern. |
| **UML Structure** | Interpreter pattern - Wikipedia |
| **Participants** | Abstractexpression: declares an abstract interpret operation that is common to all nodes in the abstract syntax tree.  Terminalexpression: implements an interpret operation associated with terminal symbols in the grammar.  Nonterminal expression: for nonterminal symbols.  Context: contains info that’s global to the interpreter.  Client: builds an abstract syntax tree representing a particular sentence. |
| **Collaborations** | The client builds the sentence as an abstract syntax tree.  The interpret operations at each node use the context to store and access the state of the interpreter. |
| **Consequences** | It is easy to change and extend the grammar.  Implementing the grammar is easy.  Complex grammars are hard to maintain.  Adding new ways to interpret expressions. |
| **Implementation** | Creating the abstract syntax tree.  Defining the interpret operation.  Sharing terminal symbols with the flyweight pattern. |
| **Sample Code** |  |
| **Known Users** | Language translator by Google. |
| **Related Patterns** | Composite: abs syntax tree is an instance of the composite pattern.  Flyweight: shows how to share terminal symbols.  Iterator: the interpreter can use iterator to traverse the structure.  Visitor: can be used to maintain the behaviour in each node in the abstract syntax tree in one class. |

### Template method

|  |  |
| --- | --- |
| **Intent** |  |
| **Also Known as** |  |
| **Motivation** |  |
| **Applicability** |  |
| **UML Structure** |  |
| **Participants** |  |
| **Collaborations** |  |
| **Consequences** |  |
| **Implementation** |  |
| **Sample Code** |  |
| **Known Users** |  |
| **Related Patterns** |  |

### Chain of responsibility

|  |  |
| --- | --- |
| **Intent** | Avoid coupling sender of request to its receiver by giving more than one object chance to handle request.  Chain receiving objects and pass request along until an object handles it. |
| **Also Known as** |  |
| **Motivation** | Help facility.  User can obtain help info on any part of the interface just by clicking on it.  Help depends on the parts of the selected interface.  Help info is organised according to its generality from the most specific to the most general.  The problem is that the object that ultimately provides the help is no known explicitly to the object that initiates the help request. |
| **Applicability** | When more than one object may handle a particular request and the handles is not known ahead of time.  When you want to issue a request to one of several objects without specifying the receiver explicitly.  When the set of objects to handle a request should be specified dynamically. |
| **UML Structure** | Chain of Responsibility Design Pattern - GeeksforGeeks |
| **Participants** | Handler: defines an interface for handling the request.  Conc\_handler: handles the request if it is responsible for accessing its successor, if it can handle the request it can so, otherwise it forwards the request to its successor.  Client: initiates the request to a conc\_handler object on the chain. |
| **Collaborations** | When a client issues a request, the request propagates along the chain until a conc\_handler object takes responsibility for handling it. |
| **Consequences** | Reduced coupling.  Added flexibility in assigning responsibilities of objects.  Receipt is not guaranteed. |
| **Implementation** | Implementing the successor chain.  Connecting successors.  Representing requests. |
| **Sample Code** |  |
| **Known Users** |  |
| **Related Patterns** | Often applied in conjunction with composite. |

### Command

|  |  |
| --- | --- |
| **Intent** |  |
| **Also Known as** |  |
| **Motivation** |  |
| **Applicability** |  |
| **UML Structure** |  |
| **Participants** |  |
| **Collaborations** |  |
| **Consequences** |  |
| **Implementation** |  |
| **Sample Code** |  |
| **Known Users** |  |
| **Related Patterns** |  |

### Iterator

|  |  |
| --- | --- |
| **Intent** | Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. |
| **Also Known as** | cursor |
| **Motivation** | The idea of iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object.  Iterator object maintains the state of iterator.  Keeping track of the current item and having a way of identifying what elements are next to be iterated. |
| **Applicability** | To access an aggregate objects content without exposing its internal representation.  To support multiple traversals of aggregate objects.  To provide a uniform interface for traversing different aggregate structures. |
| **UML Structure** | Carlos Caballero |
| **Participants** | Iterator: defines an interface for accessing and traversing elements.  Conc\_iterator: implements the iterator interface. Keeps track of the current position in the traversal of the aggregate.  Aggregate: defines an interface for creating and iterator object.  Conc\_aggregate: implements the iterator creation interface to return an instance of the proper conc\_iterator. |
| **Collaborations** | A conc\_iterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal. |
| **Consequences** | It supports variations in the traversal of an aggregate.  Iterators simplify the aggregate interface.  More than one traversal can be pending on an aggregate. |
| **Implementation** | When client controls the iterator, it is external iterator.  When an iterator controls it, then it is internal iterator.  Robust iterator ensures that insertions and removal wont interface with traversal and it does it without copying the aggregate.  Additional iterator options are first, next, isdone, currentItem  Use polymorphic iterators in C++  Iterators may have privileged access.  Iterators are composites.  Null iterators help in handling boundaries. |
| **Sample Code** |  |
| **Known Users** | MP3 player |
| **Related Patterns** | Composite: iterators are often applied to recursive structure such as composites.  Factory method: polymorphic iterators rely on factory methods to instantiate the appropriate iterator subclass.  Memento: is often used in conjunction with the iterator pattern to capture the state of an iteration. |

### Mediator

|  |  |
| --- | --- |
| **Intent** | 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. |
| **Also Known as** |  |
| **Motivation** | To have a good object-oriented design we must create lots of classes interacting one with each other.  To avoid tight coupling framework.  Interaction between objects in a manner in that objects are not aware of the existence of other objects. |
| **Applicability** | Set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.  Reusing an object is difficult because it refers to and communicates with many other objects.  A behaviour that is distributed between several classes should be customisable without a lot of sub-classing. |
| **UML Structure** | The Mediator Design Pattern in Java | Baeldung |
| **Participants** | Mediator: defines an interface for communicating with colleague objects.  Conc\_mediator: implements co-operative behaviour by co-ordinating colleague objects.  Colleague\_classes: each colleague class know its mediator object. Each colleague communicates with its mediator whenever it would have otherwise it communicates with another colleague. |
| **Collaborations** | Colleagues send and receive requests from a mediator object. The mediator implements the cooperative behaviour by routing request between the appropriate colleagues. |
| **Consequences** | It limits sub-classing.  It decouples colleagues.  It simplifies object protocols.  It abstracts how objects co-operates.  It centralises control. |
| **Implementation** | Omitting the abstract mediator class.  Colleague-mediator communication. |
| **Sample Code** |  |
| **Known Users** |  |
| **Related Patterns** | Façade – it abstracts a subsystem of objects to provide a more convenient interface.  Colleagues can communicate with the mediator using the observer pattern. |

### Memento

|  |  |
| --- | --- |
| **Intent** | Without violating encapsulation, capture and externalise an object is internal state so that the object can be restored to this state later. |
| **Also Known as** | Token |
| **Motivation** | Often, we need to backtrack when a particular path proves unproductive.  Memento stores a snapshot of other objects internal state.  Compromise the applications reliability and extensibility.  During undo operation, the editor gives the memento back to the originator.  Origination creates and returns a memento.  Based on the info in the memento, the originator restores itself to its previous state. |
| **Applicability** | A snapshot of an object state must be saved so that it can be restored to that state later.  A direct interface to obtaining the state would expose implementation details and break the objects encapsulation. |
| **UML Structure** | Design Patterns - Memento Pattern |
| **Participants** | Memento: stores internal state of the originator object. Protect against access by objects other than originator.  Originator: creates a memento containing a snapshot of its current internal state. Uses the memento to restore its internal state.  Caretaker: responsible for the memento’s safekeeping. Never operates on or examines the contents of a memento. |
| **Collaborations** |  |
| **Consequences** | Preserving encapsulation boundaries.  It simplifies originator.  Using memento might be expensive.  Defining narrow and wide interfaces.  Hidden costs in caring for memento. |
| **Implementation** | Language support.  Storing incremental changes. |
| **Sample Code** |  |
| **Known Users** |  |
| **Related Patterns** | Command – can use memento to maintain state for undoable operation.  Iterator – memento can be used for iteration as described easily. |

### Observer

|  |  |
| --- | --- |
| **Intent** | Is to define one to many dependencies.  It can have a relation between subject and many observers.  All dependents (observers) notified an update automatically. |
| **Also Known as** | Dependents, publish-subscribe. |
| **Motivation** | Partitioning a system into a collection of co-operating classes requires maintaining consistency between related objects.  Achieving such consistency by tightly coupled classes reduces their reusability.  Needs to be decoupled to make the subject and observer independently reusable. |
| **Applicability** | Broadcasting messages.  The abstraction has two aspects with one dependent on another.  Subject object does not know exactly how many observer objects it has.  Subject object should be able to notify its observer object without knowing who these observer objects are. (Objects need to be de-coupled).  Subject is not depending on objects.  Objects are independent of other objects. |
| **UML Structure** | Observer Design Pattern |
| **Participants** | Subject: has a list of many observers; interface for attach/detaching an observer.  Observer: an updating interface for object that gets notified of changes in a subject.  Conc\_subject: stores “state of interest” to observer; sends notification when state changes.  Conc\_observer: implements updating interface. |
| **Collaborations** | Concrete subject notifies its observers whenever a change occurs that could make its observers state inconsistent with its own.  A concrete observer may query the subject for info and then change its internal state accordingly. |
| **Consequences** | Abstract coupling between subject and observer.  Dynamic relationship between subject and observer can be established at runtime giving programming flexibility.  Support for broadcast communication.  Unexpected updates. |
| **Implementation** | Mapping subjects to their observers.  Observing more than one subject.  Trigger notification.  Dangling references to deleted subject.  Specific consistency/inconsistency notification. |
| **Sample Code** |  |
| **Known Users** | Broadcasting messages.  Dynamic updates. |
| **Related Patterns** | Mediator, singleton |

### State

|  |  |
| --- | --- |
| **Intent** | Allows an object to alter its behaviour when its internal state changes. The object will appear to change its class. |
| **Also Known as** | Objects for states |
| **Motivation** | The state pattern is useful when you want to have an object represent the state of an application.  Change the state by changing that object.  Allow object to alter its behaviour in response to internal state changes.  Benefit is that state-specific logic is localised in classes that represent that state. |
| **Applicability** | Objects behaviour depends on its state.  Must change its behaviour of runtime.  Operations have large multipart conditional statements that depend on the object state.  Objects vary independently from other objects. |
| **UML Structure** | State pattern - Wikipedia |
| **Participants** | Content: defines the interface of interest to clients. Maintains an instance of a conc\_state. Subclass that defines the current state.  State: defines an interface for encapsulate the behaviour associated with a particular state of the context.  Conc\_state subclass: each subclass implements a behaviour associated with a state of the context. |
| **Collaborations** | Context delegates state-specific requests to the current conc\_state object.  A context may pass itself as an argument to its state.  Context is the primary interface for clients. |
| **Consequences** | It localises state-specific behaviour and partitions behaviour for different states.  An alternative is to use data values to define internal states.  It makes state transitions explicit.  State objects can be shared. |
| **Implementation** | Defines the state transitions.  A table-based alternative.  Creating and destroying state objects.  Using dynamic inheritance. |
| **Sample Code** |  |
| **Known Users** | Johnson and zweig characterise the state pattern and its application to TCP. |
| **Related Patterns** | Flyweight – explains when and how state objects can be shared.  State objects are often singletons. |

### Strategy

|  |  |
| --- | --- |
| **Intent** | Define a family of algorithms, encapsulate each one and make them interchangeable.  Strategy let the algorithm vary independently from clients that use it. |
| **Also Known as** | policy |
| **Motivation** | Different algorithms will be appropriate at different times.  It is difficult to add new algorithms and vary existing ones.  Clients that need line-breaking get more complex if they include the line-breaking code. |
| **Applicability** | Many related classes differ only in their behaviour.  You need different variants of an algorithm.  An algorithm uses data that client should not know about.  A class defines many behaviours and there appear as multiple conditional statements in its operations. |
| **UML Structure** | 26: UML class diagram for Strategy pattern | Download Scientific Diagram |
| **Participants** | Strategy: declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by conc\_strategy.  Conc\_strategy: implements the algorithm using the strategy interface.  Context: is configured with a concrete state object. May define an interface that lets strategy access its data. |
| **Collaborations** | Strategy and context interact to implement the chosen algorithm.  A context forwards requests from its clients to its strategy. |
| **Consequences** | Families of related algorithms.  Alternative to sub-classing.  Strategies eliminate conditional statements.  A choice of implementations.  Clients must be aware of different strategies.  Communication overhead between strategy and context.  Increased no of objects. |
| **Implementation** | Defining the strategy and context interfaces.  Strategies as template parameters.  Making strategy objects optional. |
| **Sample Code** |  |
| **Known Users** | Interviews use strategies to encapsulate different line-breaking algorithms as we have described. |
| **Related Patterns** | Flyweight – strategy objects often make good flyweight. |

### Visitor

|  |  |
| --- | --- |
| **Intent** | Let’s you define a new operation without changing the classes of the elements on which it operates. |
| **Also Known as** | Open-close principle  Double dispatcher |
| **Motivation** | Book example (AST)  Collections are data types widely used in object-oriented programming. Often collections contain objects of different types and in those cases some operations have to be performed on all the collection elements without knowing the type. |
| **Applicability** | Rarely changing object structures.  Using unrelated operations.  Many classes with different interfaces. |
| **UML Structure** | Visitor pattern - Wikipedia |
| **Participants** | Element: defines an accept operation that takes a visitor as an argument.  Conc\_element: implements an accept operation.  objectStructure: can enumerate its elements; may provide a high-level interface; either be a composite or a collection such as a list or a set.  visitor: declares a visit operation for each class of conc\_element in the object structure. Identifies the class that sends visit request to the visitor; then visitor can access the elements directly through its interface.  Conc\_visitor: implements each operation declared by visitor. |
| **Collaborations** |  |
| **Consequences** | Visitor makes adding new operations easy.  Visitor gathers related operations and separates unrelated ones.  Adding new conc\_element classes is hard.  Visiting across class hierarchies.  Accumulating state.  Breaking encapsulation. |
| **Implementation** | Each object structure will have an associated visitor class.  Implementation issues will be double dispatcher, who is responsible for traversing the object structure. |
| **Sample Code** |  |
| **Known Users** | Smalltalk compiler has a visitor class. |
| **Relateds Patterns** | Composite: visitors can be used to apply an operation over an object structure defined by the composite pattern.  Interpreter: visitor may be applied to do the interpretation. |

# Architectural Principles

## Increases Cohesion

Cohesion indicates that software entities (class, function, module) should have closely related responsibilities.

The tasks performed should closely related to each other.

Having high cohesion increases maintainability.

## Reduces Coupling

Coupling refers to the level of dependency between two entities.

Two components A and B are said to be coupled, when you can’t change A without changing B.

Good software design should have low coupling between components.

Having low coupling increases maintainability and reuse of software.

Separation of concern is achieved using modular programming (separation of code into modules) and use of encapsulation (information hiding).

Modules will have their own interfaces to communicate with other modules and hide internal implementation details.

## Divide and Conquer

Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things.

Separate people can work on each part.

Each individual component is smaller and easier to understand.

Parts can be replaced or changed without having to replace or extensively change other parts.

## Increases Abstraction

Ensures that your design allows you to hide details to reduce complexity.

Allows you to understand the essence of a subsystem without having to know unnecessary details.

## Increases Reusability

Generalize and simplify your design as much as possible.

So that they can be used again in other similar contexts.

## Increases Reuse using API’s (Interfaces)

Design with reuse is complimentary to design for reusability.

Cloning should not be seen as a form of reuse.

## Increases Flexibility

Do not hard copy anything.

Do not restrict the options of people who must modify the system later.

Use reusable code and make code reusable.

## Anticipate Obsolescence

Plan for changes in technology and environment so that software can run long by changing easily.

Avoid using early releases – software libraries for environment, software from companies that are less likely to give long term support.

Use standard languages and technologies that are supported by multiple vendors.

## Design for Portability

Have the software that run on as many platforms as possible.

## Design for Testability

Take steps to make testing easier.

Design a program to automatically test the software.

## Design Defensively

Never trust how others will use component you designed.

Test all the input types on your component.

# Architectural Patterns

## Client-server

Lower layer will not know what is happening in higher layer.

Lower layer will not communicate with higher layers.

Layers are replaceable.

Higher layers see lower layers as a set of services.

* User interaction layer: Normal for presentation
* Application layer: provides apps and functions.
* Domain layer: general domain level services.
* Services/support layer: provide essential services.

## Multi-layer

It is a 3-tier model architecture consists of client, server, and middleware (CORBA and ORB).

Peer to peer direct connection.

Set of services provided by servers and set of clients that use these services.

Servers need not know about clients, but clients must have the known server id

Variations in peer to peer with location sharing.

## Broker

Mediator

Proxy server can also be a broker.

Client has no knowledge about server.

Object reusability.

Object service request.

## Transaction Processing

A process reads a series of input one by one.

Each input describes a transaction.

Transaction is a command that change to the data stored by the system.

There is a transaction handler that decides what to do with each transaction.

## Pipe and Filter

Break down the process required for each stream into a set of separate filters.

Each performs a single task.

Mainly used for streaming data.

These streams can be combined into a pipeline.

Helps to avoid duplicating code.

Easy to remove, replace, or integrate additional components.

## Model-View-Controller

Used for implementing UI on computers.

Divides a given application into three interconnected parts (Separation of Concern).

More implemented in J2EE and SmallTalk.

**Model:** central component of the pattern. Encapsulates the core data and functionality.

**View:** output representation of info. Multiple views of the same info.

**Controller:** accepts any input in any form. Converts input into commands for the model or view.

**Goals of MVC:**

* Simulated development.
* Code reuse.

**Advantages of MVC:**

* Simultaneous development
* High cohesion
* Low coupling
* Ease of modification
* Multiple views

**Disadvantages of MVC:**

* Code navigability
* Multi artifact
* Consistency

**Flow of MVC:**

Client (browser) controller model Data source

Controller view

View Model.

View client (browser)

**Interactions among MVC:**

**Model:** stores data and retrieves according to commands from controller and displayed in view.

**View:** generates new output to the user based on the changes in model.

**Controller:** send commands to the model to update model state. Also send commands to the view to change the views presentation.

# Architectural Strategies

The four main architectural strategies are listed as follows:

1. Have independent classes for aggregation rather than inheritance.
2. Encapsulate the functionalities.
3. Establish common terminology and follow organisational standards.
4. Prioritise reusability, readability, maintainability, extensibility, scalability, and testability.

# Anti-patterns

Anti-pattern is the knowledge to prevent and recover from common mistakes. It is to deal with the gap between architectural concepts and real-world implementations.

Provide negative solutions.

Solution that represents more problems than they address.

Extension to design patterns.

**Viewpoints:** to avoid bad occurrences and for smooth running.

**Three types of viewpoints:** manager viewpoint, architect viewpoint, developer viewpoint

**Applications:**

**Prototype:** file sharing, OS installation

**Singleton:** shopping cart, ID card, OTP, Barcodes

**Builder:** car parts, building

**Guidelines:**

* Study the applications of design.
* Map your application to the design pattern.
* Study the patterns which has similar solutions.
* Reusability if you are going to redesign for another solution.
* Having a wide knowledge about design patterns provide a structure for given pattern.

**Problem-solving by Design patterns:**

* Finding appropriate object
* Determining object granularity (object can vary in size and number)
* Specifying object interfaces