**Bridge Design Pattern:** Connecting components together with through abstractions

Bridge prevents a ‘Cartesian Product’ complexity Explosion.

Example:

a) Base class thread scheduler.

b) Can be preemptive or cooperative

c) Can run on Windows or Unix

d) End up with with a 2\*2 scenario: WindowsPTS, UnixPTS, WindowsCTS, UnixCTS.

Note: Bridge Pattern avoids the entity explosion.

**Note:** A mechanism that decouples an interface(hierarchy) from an implementation(hierarchy).

Its main goal to hide the implementation details.

Its main idea is on **Pimpl Idiom/implementation is**  to hide the implementation details a class, which is bridged into a separate class. As we provide only .h file to the users, all the implementation can be placed inside the .cpp file and we can hide the inner implementation details.

**Pimpl implementation** advantage is the user need not to compile the code also again. as we change the implementation file(.cpp), but not the .h file, it will improve the compilation speed.

**Observer Design Pattern:**

a) We need to be notified when certain things happen

i) Object’s field changes

ii) Object does something

iii) Some external event occurs

b) We want to listen to the events and notified when they occur.

c) Terminology:

i) events and subscribers

ii) Signal and Slot (boost, QT)

**Observer:** An observer is an object that wishes to be informed about events happening in the system. And entities generating the events is an Observable.

**Chain of Responsibility:**

A chain of components who all get a chance to process a command or a query, and optionally having default processing implementation and an ability to terminate the processing chain.

**Command Design Pattern:**

Ordinary C++ statements are perishable:

a) Cannot undo a field/property assignment

b) Cannot directly serialize a sequence of actions (calls)

Want an object that represents an operation:

a) X should change its field Y to the value Z.

b) X should do W()

Uses:

GUI Commands, multi-level undo/redo , macro recording and more.

**Command DP**: Its an Object which represents an instruction or set of instructions to perform a particular action. Contains all the information necessary for the action to be taken.

**Command Query Separation:**

Command = asking for an action or change

Qurey = asking for information (e.g please give me attack value)

CQS = having separate means of sending commands and queries.

In GoF(Gang of Four) context, both commands and queries are represented with Command Design pattern.

**Mediator Design Pattern:**

Gamma Categorization: Design patterns are split into the 3 types:

a) Creational design patterns:

* deal with creation (constructor of objects)
* Explicit (constructor) vs implicit (DI(dependency injection, reflection, etc..))
* Wholesale (single statement) vs piecewise(step by step)

b) Structural design patterns

* Concerned with the structure (e.g. class members)
* Many patterns are wrappers that mimic the underlying class’ interface.
* Stress the importance of the good API design.

c) Behavioral design patterns:

* They are all different, no central theme.

**Builder design patterns:**

**Motivation:**

1. Some objects are simple and can be created in a single constructor call.
2. Other objects require a lot of ceremony to create
3. Having an object with 10 constructor args is not productive
4. Instead, opt for piecewise construction
5. Builder provides an API for the constructing an object step by step.

When piecewise object construction is complicated, provide an API for doing it succinctly.

**Factory Design pattern:**

a) Factory Design pattern:

A Component solely responsible for the wholesale (not piecewise) creation of objects.

b) Abstract Design pattern:

**Summary:**

a) A factory method is a static method that creates objects

b) A factory can take care of object creation.

c) A factory can be external or reside inside the object as an inner class

d) Hierarchies of factories can be used to create related objects.

**Motivation:**

a) Object creation logic becomes too convoluted.

b) constructor is not descriptive

* Name mandated by name of containing type
* Cannot overload with same set of args with different names.
* Can turn into “optional parameter hell”

c) Object creation (non-piecewise, unlike Builder) can be outsourced to

* A separate function (Factory Method)
* That may exist in a separate class (Factory)
* Can create hierarchy of factories with Abstract Factory.

**Prototype Design Pattern:**

**Prototype:** A partially or fully initialized object that you copy (clone) and make use of.

**Motivation:**

a) Complicated objects (e.g cars) aren’t designed from scratch (They reiterate existing designs).

b) An existing (partially or fully constructed) design is a Prototype.

c) We make a copy (clone) the prototype and customize it

-- Require “deep copy” support

**Composite Design pattern:**

A mechanism Treating individual and aggregate objects in a uniform manner.

**why do we need composite DP:**

a) Objects use other object’s fields/properties/members through inheritance and composition.

b) Composition lets us make compound objects:

Eg: a) A mathematical expression composed of simple expressions; or

b) A grouping of shapes that consist of several shapes.

c) Composite DP is used to treat both single and composite objects uniformly.

i.e Foo and Collection<Foo> have common APIs

**Façade Design Pattern: (**Its pronounced as Fasade)

**Motivation:**

a) Provides a simple, easy to understand/user interface over a large and sophisticated body of code.