**Aggregation (HOLDS/ Part of life):**

* **I holds a car, I holds a apartment**
* **Provide a instance by factory, dependency inversion container (DI container)**

**Composition (contains/ uses/ has-a):**

* TempratureParameterValidator -> uses logger. (or) has a logger dependency
* Or -> TempratureParameterValidator **contains** the object of logger.
* Here Logger will die when the obj of the TempratureParameterValidator is destroyed
* Example: Human contains emotions -> emotions will dies when human dies.
* Violation of dependency inversion principal

Graphical user interface, text, application, email

Description automatically generated

Above code can be refactored as : (Strategy pattern)

Text

Description automatically generated

Timeline

Description automatically generated with low confidence

**Strategy pattern:**

Diagram

Description automatically generated

Here context -> TempratureValidator

**composite pattern**

Now lets say I want to add the constraint to add the multiple loggers like consoleLogger, FileLogger etc. i.e broadcasting to multiple -> i.e **composite pattern**

I.e One logger is now changing to many

When ever there is tree structure/ hierarchy we use composite pattern. Example XML parser, UI components

Chart

Description automatically generated

Diagram

Description automatically generated with medium confidence

**Example 2: (composite Pattern)**

Diagram

Description automatically generated

Diagram

Description automatically generated

**Observer pattern:**

Diagram

Description automatically generated

Observer pattern notes:

* One to many communication
* Say we need to update the order status via Email, SMS

**Mediator pattern**: (Many to many, Single spa applications, angular event emitter, RXJS, Rabbit Mq)

Here M1, M2, are the individual UI module.

Diagram

Description automatically generated

In Enterprise applications:

* Event Aggregator is also called as the Event bus, Service Bus
* Pub sub model

**Bridge pattern**

* <https://www.codeproject.com/Articles/890/Bridge-Pattern-Bridging-the-gap-between-Interface>

**State pattern:**

Example: Bank account state handle

Diagram

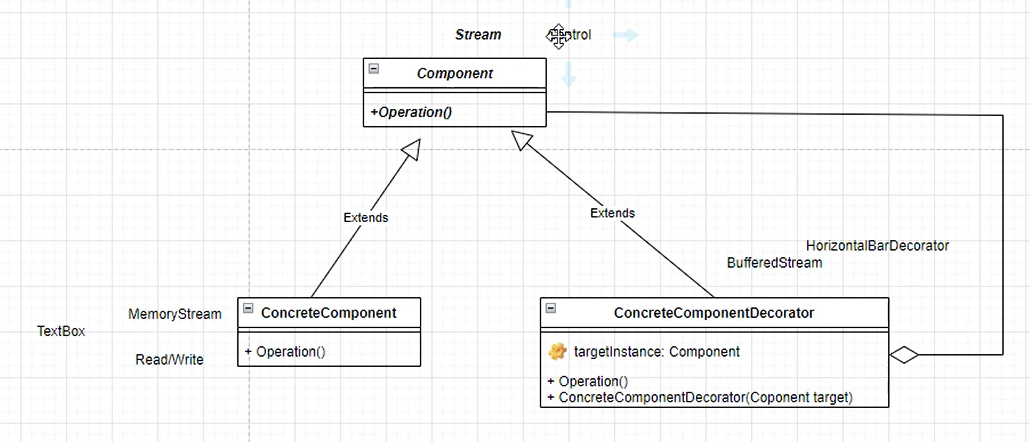
Description automatically generated

Example 2: Bug life cycle

<https://www.bugzilla.org/docs/2.18/html/lifecycle.html>

DDiagram

Description automatically generated

**Decorator pattern**

Graphical user interface, text, application, chat or text message, email

Description automatically generated

Chain of responsibilities

* Example: Interceptors

‘'Diagram

Description automatically generated

Adapter pattern:

Diagram

Description automatically generated

Separator pattern (Same as object adapter) -> Command pattern

Diagram

Description automatically generated

* Delegates in C# written using command pattern -> create delegate obj, and can be invoked any number of times. -> which method and which class need to be called.
* In Java we have Action listeners equivalent to delegates in c#
* Adapter will have intelligence -> it matchs the compatibility, command will just executes the command method just routes.

**Proxy pattern:**

Diagram

Description automatically generated

Factory will create the real object and proxy object and then real object copies to proxy and then proxy object gives to client.

* Important: both proxy and real obj should be substitutable to each other.
* Mother acts as a proxy between me and father.
  + My father is the one who fullfill the needs by the request going via my mother instead of I directly ask him.
* SPRING AOP -> They don’t allow the obj directly -> handling the method when throws some exception, Aspect oriented programming(AOP) works on proxy

**Memento pattern.**