**Observer DP:**

The Observer DP is a behavioral design pattern that defines a one-to-many dependency between objects, so that when one object (the Subject) changes state, all its dependents (Observers) are notified and updated automatically.

Subject(observable), Abstract observer(interface), Concrete observer and Client are the components.

Loose coupling - Subjects don't need to know about concrete Observer classes. Dynamic relationship - Observers can subscribe/unsubscribe(attach/detach) at runtime. Push/Pull models - Subjects can send data and let Observers fetch it. Broadcasts state changes to subscribed users. Suitable for Event handling systems.

Order order; is a Stack object which will be automatically deleted when program goes out of scope. Order\* order = new Order(); is a Heap object and need to be deleted manually. If stack object then ‘.’ is used to invoke function. If heap object then ‘->’ is used to invoke function.

Examples: Event handling systems like Newsfeed notifications, Weather forecast notifications, Realtime traffic updates to GPS etc.

**Command DP:**

The Command is a behavioral design pattern that turns a request into a stand-alone object containing all information about the request.

It lets you parameterize methods with different requests. Enables queuing and logging requests by serializing the commands. Decouples Invoker from Receiver. Easy to add new commands(OCP) by creating new classes.

Components are: Command Interface – Declares the execute() method. Concrete Command – Implements the Command interface. Receiver – The object that performs the action. Invoker – Triggers the Concrete command. Client – Creates the command and sets the receiver.  
Each command is an object that implements a common interface. These objects can be stored in a list (queue/stack/log), reordered and can be re-executed.

vector<Command\*> commandQueue;

commandQueue.push\_back(new RideRequestCommand (&rideservice,…));

commandQueue.push\_back(new CancelRideCommand ((&rideservice,…));

for (Command\* cmd : commandQueue) {

cmd->execute();

}

Example: Task scheduler, Transaction management(by creating class for each operation).

**Chain of Responsibility DP:**

The Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Each handler decides either to process the request or to pass it to the next handler in the chain.

Allows dynamic addition/removal/reordering of handlers in hierarchy at runtime. Divides responsibilities across multiple handlers(SRP).

Promotes loose coupling between sender(client) and receiver(handlers). As client need not know call hierarchy and it can be managed among classes and client just need to initiate start of chain. We can have different processes instead of only processOrder() alone and can manage order among classes. Alternative is see last comment in main().

Each handler can either process the request or pass it to the next handler in the chain.   
class PaymentProcessingHanlder : public OrderHandler {

public:

PaymentProcessingHanlder(OrderHandler\* nextHandler) : OrderHandler(nextHandler) {}

void processOrder(const string& order) override {

cout << "Processing payment for order: " << order << endl;

if (order == "BadOrder") {

cout << "Payment failed! Order processing stopped.\n";

return; // Stop chain here

}

if (nextHandler)

nextHandler->processOrder(order);

}

};

Chain of responsibility can be handled by Command DP by storing objects in required order into a vector and execute them but dynamic reordering cannot be followed here. Also handlers cannot decide the flow here.

Managing chain hierarchy is complex as any disturbance can halt processing the request. When multiple objects can handle a request, but the exact handler isn’t known in advance.

Example: Pipeline systems like authentication-validation-execution etc.

**Iterator DP:**  
The Iterator Pattern is a behavioral design pattern that provides a way to access the elements of a collection (like arrays, lists, trees, etc.) sequentially without exposing the internal representation of that collection.

When you want to decouple traversal logic from the collection class itself. Works across different container types. Providing multiple traversal algorithms (DFS, BFS, in-order) for the same collection.   
  
Follows SRP, OCP as new iterator types(forward/backward/sorted) can be added without changing collections. Parallel traversal i.e., multiple iterators can traverse the same collection. Maintaining iterators is complex.

Element is a class. Iterator is a common interface. Concrete iterator implements Iterator and implement methods according to type of traversal. Aggregate/Collection is an interface/concrete class as required which provides iterator and collects together the elements.

Examples: File system Iterator, Managing song playlists etc.