Unit-5 OOAD flash notes

Behavioral pattern - Chain of responsibility.

-> Behavioral patterns identify common communication patterns. These influence how state flows through the system.

-> COR: pass state through a chain, where each "handle" decides whether to process or not.

-> Problem: Due to ever increasing complexity & # of comps in a system. The comprehension needed for easy reuse becomes inherity harder.

-> Solution: Each handler has a field for storing a refrence to the next handler in the chain.

The common approach is to have only one single handler process the (mostly gui) request. But a more usefull is to allow many handlers. (this is considered dfault)

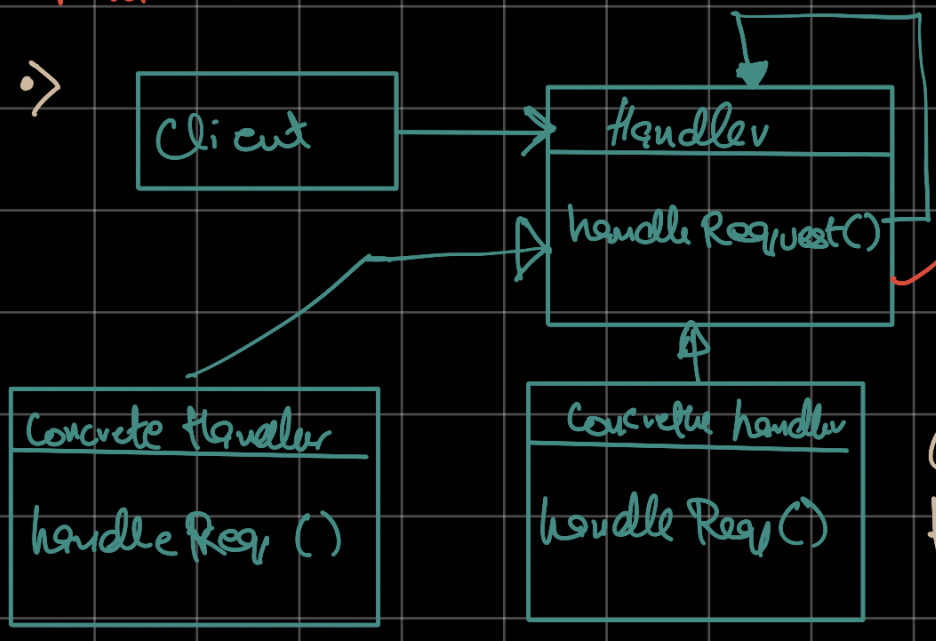
-> - promotes loose coupling (ADV)
- adds flexibity.
- simplifies object structure not having to know next step from initator/caller

- Receipt is not gaoreuted (DADV)
- Can be hard to observe runtime behavior or debug.

-> - use CoR when program is expected to process a large # of types of req. Each in an unique way & type of req. not known before hand.

- Use CoR when one sees a linearly Processing order for a bunch of Services very commonly

- use CoR when Order of handlers is to change at runtime.



Look at slide for code example, Specially the transaction manager ex!

Behavioral pattern: Command pattern

-> pattern that allows conversion of req to a standalone object with all data in Context.

-> Problem: If the caller function is extended / subclassed to handle the various possible request leaves a very large # of classes.

-> Solution: Break into layers where the handler creates a Context object for the request & pass it onto a request handler (which can be CoR or whatever).

•> - Decouples Sender & recv — - Complex Code.

A/D - deffered executions

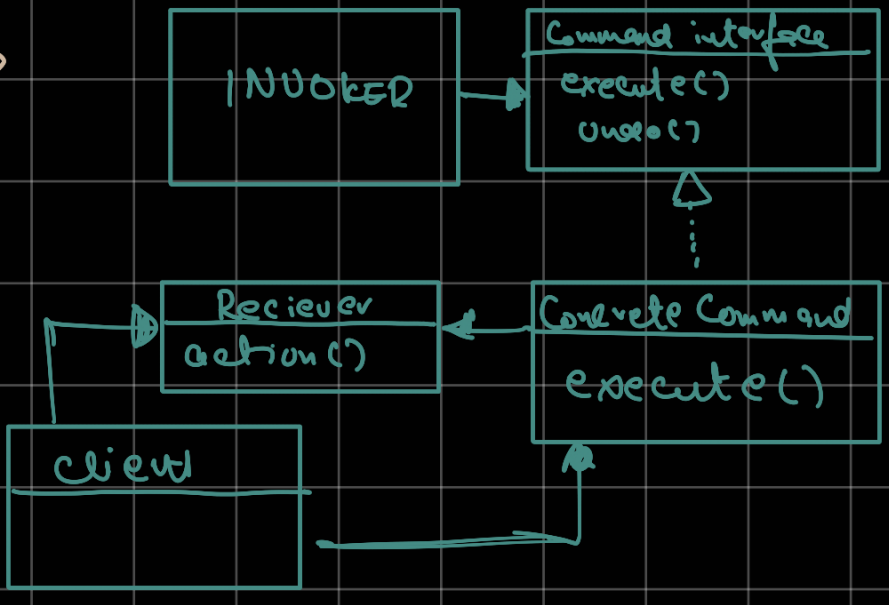
- undo can be impl

•> - when one wants to parameterize handler methods

- When one wants to deffer execution like on a queue.

- " " " reversible ops

•>



example in slides is tvcsn, Checkout refactoring guru.

→ Behavioural patterns: Interpreter

•> Defines a way to interpret grammers. It allows you to define a grammar & way to eval sentences using the grammar.

•> The basic idea is to have a class for each Symbol

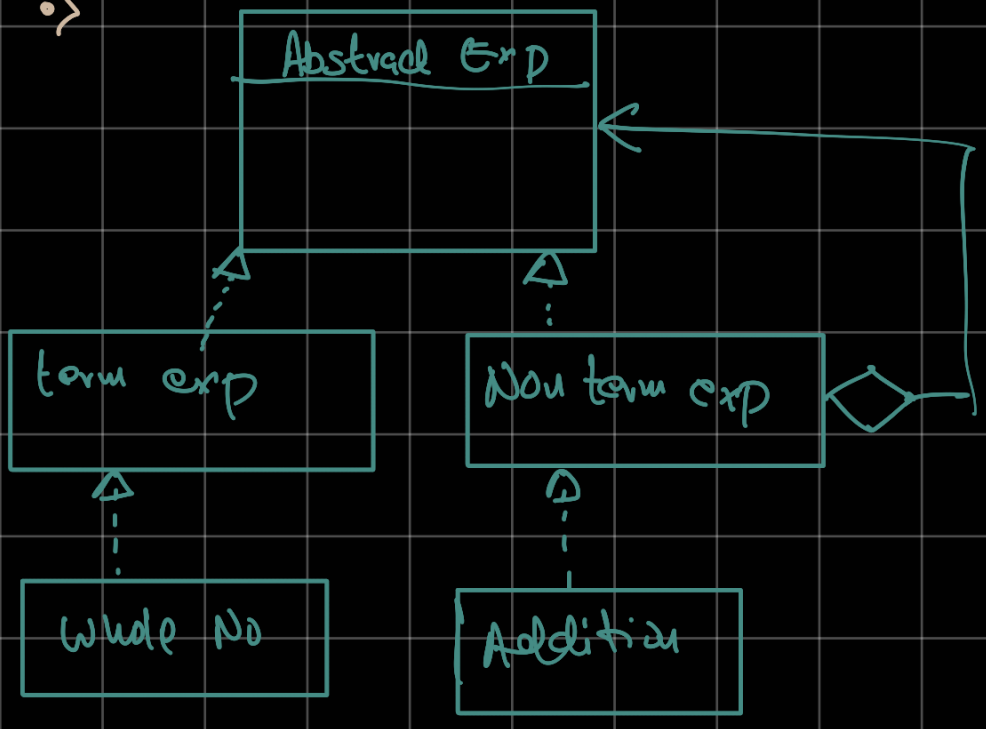
•> problem: To split def & interpretation of a language. More over letting the language tobe easily extendable.

•> Solu: class (subclassed) for every token which terce in a Common superclass.

•> - flexibility
AND - Extensibility
- Seperation

- Complexity (only easy lang)
- Performance (interpreting is slow)
- Maintenance

•>



No examples any where :(

→ Behavioral pattern: Iterator pattern.

•> To allow others to traverse a collection without exposing it underlying representation.

•> Problem: Often times we need many diff ways of accessing the same collection of data. giving generic class to cover all these is weird as the underlying logic is specific & varies.

•> Solution: Get the traversal behaviour into a seperate object called Iterator.

•> When underlying DS is complex & you want to provide
- read access but still abstract the complexity.
- Parallel Iterations of same data.

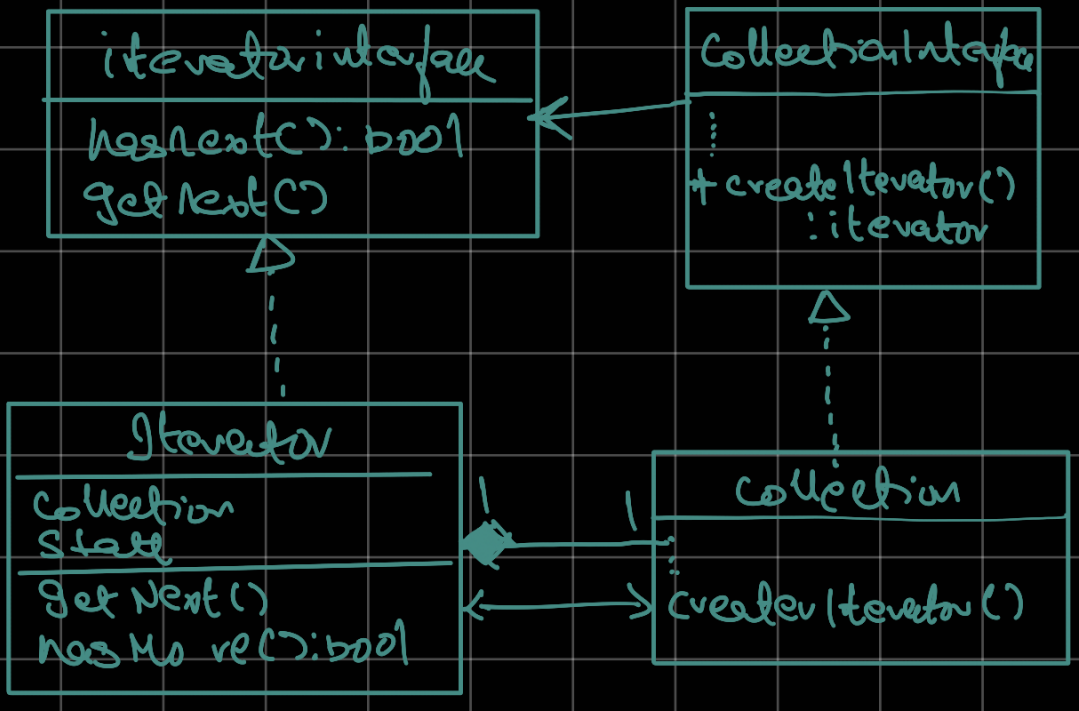
•> - parallel iteration on same data as each iter has its own state
- lazy/deffered iteration & at will

- large overhead & inefficiat
- overkill for simple DS.

A/D

•) each DS has a factory impl for the iterator.

•>



→ Antipatterns

•> Commonly occuring solution to a problem that has negative consequences.

•> Antipattern can been seen by 3 entities:

- Software developer
- Software architect
- Software manager

↳ This has to do with organisation management for progression of project.

most serious antipatterns happen here.

•) Sw design is often handled in many levels:
(& issues)

objects & classes → micro service arch → frameworks → App. level → System level → Enterprise level ← Global/indus. level

① PM anti-patterns (Analysis Paralysis)

•) Symptoms:
- multiple restarts
- design & impl issues
- pattern choices made in analysis phase
- analysis is speculative without involving the user.

•) Causes:
- Management assume waterfall rather than iterative.
- Goals not well def in analysis phase
- No firm decision to restrict domain.
- Management have high confidence in their own analysis ability.

•) Soln:
- Incremental development
- 2 types of Incremental: internal, external (with client)

② Architectural Anti pattern: Vendor-lock-in

•) Symptoms & Consequences:
- external Product upgrades drive SDLC.

- promised features are delayed.
- product varies significantly from advertised.

-> Causes:
- product varies from open standard
- product chosen purely by marketing rather than technical.
- no isolation of app sw from product
- needs deep API knowledge of product.

-> Solution: Isolation layer.
A layer of abstraction of the product in means of middleware / OS etc to give easier API.

(3) Sw development Antipattern: The blob

-> Symptoms:
- Single class with large # of attr, ops or both. (> 60) Indicates this.
- Blob class too Complex for reuse or testing.
- Expensive to load to new

-> Causes:
- Lack of OO-design
- Lack of any arch
- Lack of enforcement
- limite intervention.

-> Solution:

1: Identify & Categorize related attr & methods.

2: look for natural homes & migrate attr & methods there

3: remove for-coupled items.

4: Change associations to derived from common base class

5: No transient rels.