Object oriented development process

•> OO dev process gives emphasis on how to build a software keeping it language agnostic. This often encourages:
- re-use of Sw components - employs international UML std.

•> Steps of OOM are as follows:

i) System Conception : Business eval & req spec.

ii) Analysis : what sys must do & not how. Must understand big picture & also predict workflow

iii) Sys design : A HLD is formed

iv) class design : Add details to analysis model & HLD about Algorithm impl by each class

v) implementation : Convert classes & relations to Code.

•> A HLD gives the organisation of system into Sub-System & also sets the context for more detailed discussion

(*) System design / System Arch - OOM.

•> Estimate perf → Reuse plan → break to sub-sys
(approx. back of envelope) These are the imp ones

Identify Concurrency ← allocate to hw ← ... ← trade-offs
↳ Select arch style.

(*) Reuse plan - System desig - OOM

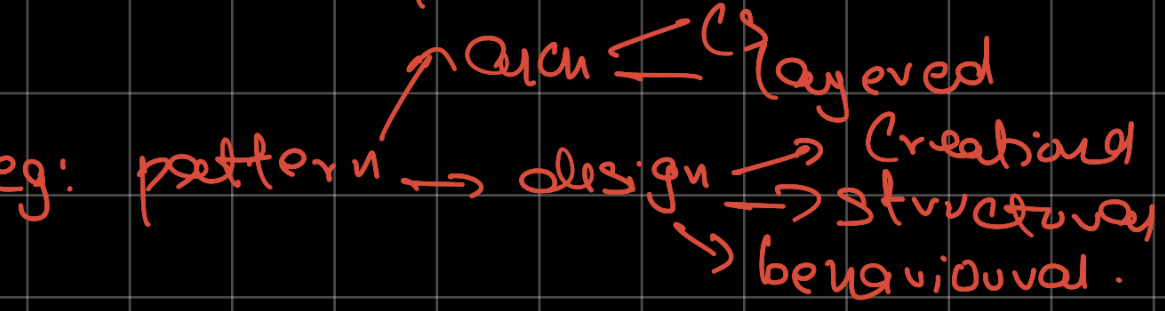
i) Libraries : Collection of classes.
The classes need to be expose carefully & correctly!

A good library will have the following:

- Coherence (organised for 1 theme)
- Completeness (all functions)
- Consistency (names & signatures)
- Efficiency (all fns efficient)
- Extensibility (can define sub classes)
- Genericity (parameterised class where possible)

example: net protocols, string libs, regex.

ii) Patterns: best practice soln to a recurring problem



(*) frame work: reuse - System design - OOM

•> frameworks provide a skeleta structure. This skeletal structure needs to be extended to build app

Contains more than just code, like:

- flow control
- shared variants
- triggers

Often times a programer extends frameworks using Subclassing:

•> In library, you call the method. In framework, the framework controls the flow & calls your code.

•> Some Java frameworks: Collections, Swing, AWT

frameworks emphasize on design reuse over code reuse

Architectural Patterns

•> Arch pattern is a general reusable solu to a common recurring problem.

often tackle problems like: Hw perf, HA, HT, Scale.

such patterns provide 1 of 2 → Highly Scalable
→ Highly agile.

① Model View Controller

•> Seperates application logic from presentation. Made up of 3 components. Splits Data model, processing & UI.

•> View → Controller → Model

UI

-> A pattern well known for web development.org:

Model: represents data & rules that govern access & updates of the data.

View: Renders the content. Data from model can either be gotten by view using push or pull model.

Controller: In webapps Controller is the GET/POST HTTP API's

-> Advantages:
- faster dev time
- Loose coupling
- no massive codebase
- Independent modifications

-> MVC is JAVA:
Controller are the "Servelets"

(2) GRASP: Objects with Responsibility

-> Contains guidelines to assign responsibility to classes.

GRASP: General Responsibility Assignment SW Patterns.

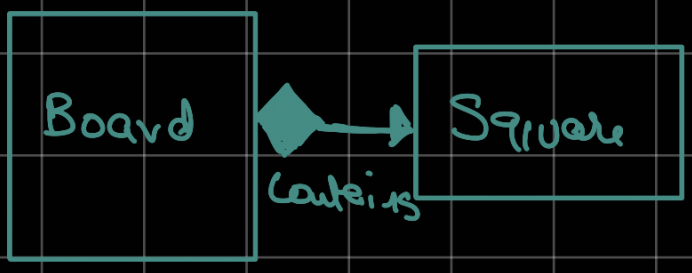
In all there are 9 tools to achieve this.

-> A responsibility is an obligation that a class must accomplish. A resp can be covered by many objects too

Responsibility → Doing
Responsibility → Knowing

i) Creator: who creates an object. Containers create contained objects.

In monopoly:

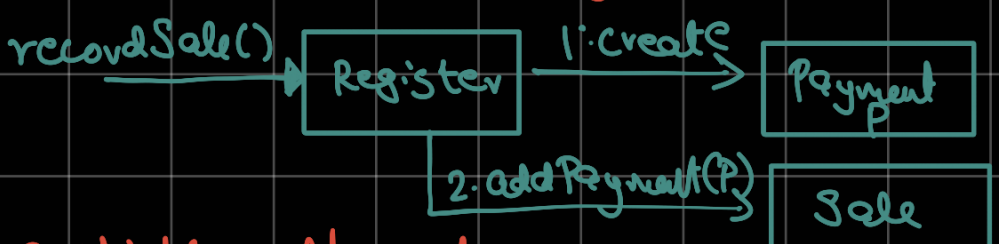


ii) Information expert: Objects do things related to the information they have.

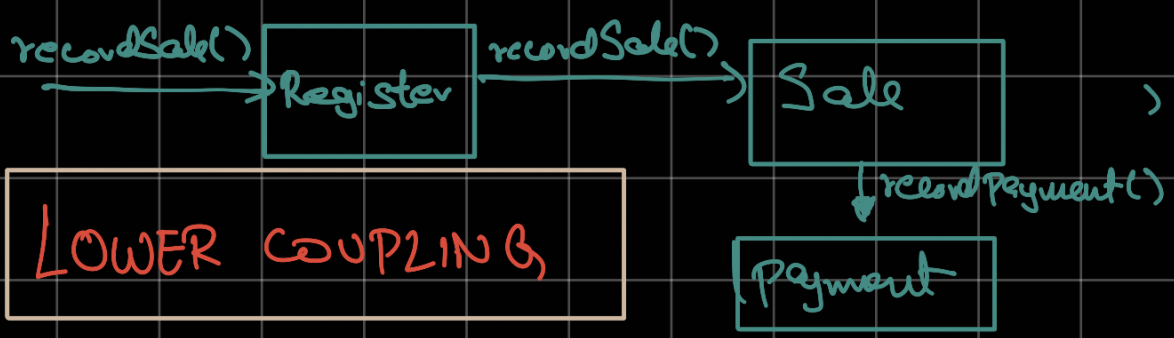
In monopoly, the board has all the details to get status of any Square. Hence is info expert

iii) Low coupling: Assign responsibilities such that coupling is low. 2 elements are said to be coupled if one element has aggregation / composition of the other.

Register creates payment & sends it to payment.



A viable alternative:



LOWER COUPLING

→ Subclassing is considered as high coupling.

(So is interface & refrences to other class in method calls [above example])

Think in terms of DEPENDENCIES

iv) Controller: How to delegate Jobs from UI to domain layer

Soln: Assig Job to class that reps a subsystem OR class that reps use case scenario (use-case controller)
↳ (facade controller)

Bloated controller: class that handles all System events.
↳ Avoid doing this.

v) High Cohesion: Keep cohesion high.

vi) Polymorphism: How to handle related but varying elements
Use polymorphism for those places where behavior varies by type but results are related & similar.

vii) Indirection: How to avoid direct coupling to keep reusing viable. Soln: Use intermediate class to link class.

viii) Pure fabrication: When you dont want to violate HCLC but info expert doesnt work wdyd? You assign a highly cohesive set of responsibilities to helper class that is not part of the domain.

ix) Controlled/protected variation: How to design objects so the variation of one doesnt cause instability in another.
Soln: predict variable points & create resp. to maintain stability.

Hence you have to look into: Variation point & evolution point.

(3) SOLID:

-> Bad design: Rigidity, fragility, immobility, viscosity

-> Solid is an acronym for:

Single responsibility (SRP)
Open close (OCP)
Liskov substitution (LSP)
Interface Segregation (ISP)
Dependency inversion (DIP)

i) SRP: Single responsibility.

every class should have resp. only over a single part of the function.

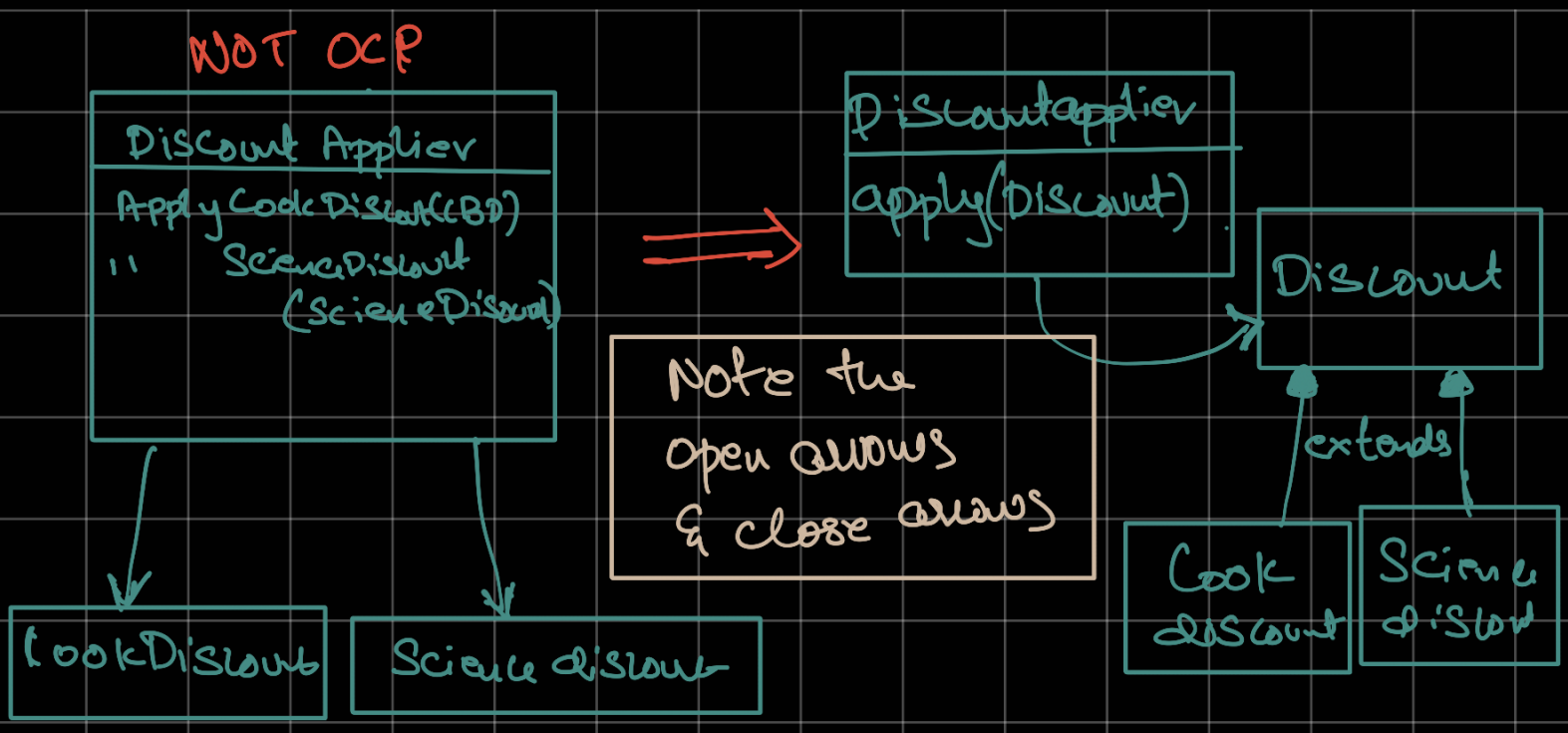
Sdu:

- find things that are changing
- group things that change for same reason
- decouple the others.

ii) OCP: All components must be open for extention but closed for modification

NEVER MODIFY EXISTING CODE AS IT CAN LEAD TO UNDEFINED BEHAVIOUR.

See slides for book discount example.



for every new discount we need new apply method.

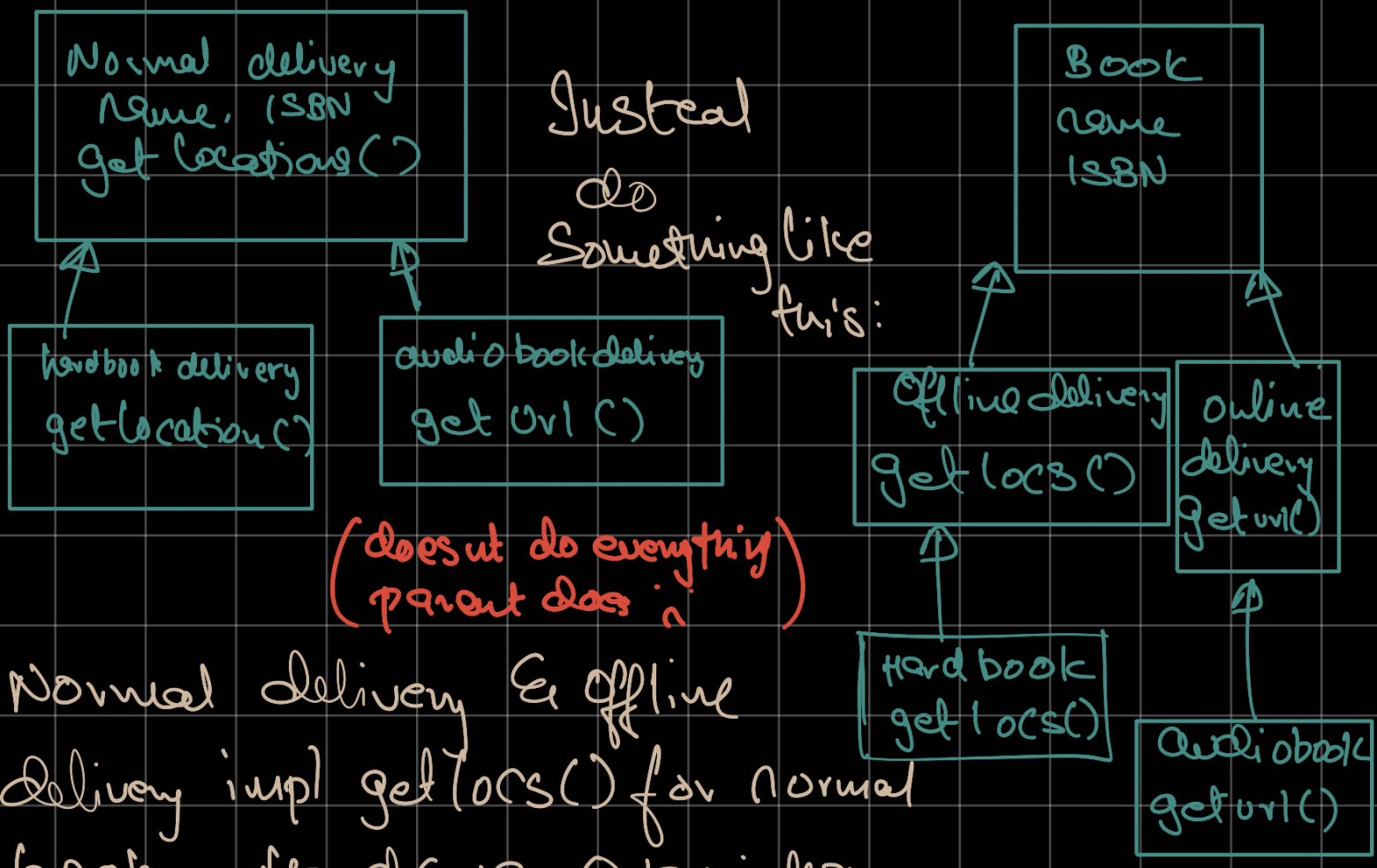
iii) LSP: Liskous Substitution principle

derived types must be entirely substitutable for their base class!

Derived class must always have additionally to their super classes.

Say in the same example, we want to add delivery functionality. As of now it only shows places/way of getting the book. We also have separate delivery locations for normal & hard covers. Well impl something like this.

Note: we also have audio books that need to show urls instead of offline locations!



Instead do something like this:

(does not do everything parent does)

Normal delivery & offline delivery impl getLocs() for normal book. HardCover overrides

iv) ISP: Interface Seperation principle.

Should not force to imple interface that they do not use.

Say we need some actions on these books, we can build interface for the following:

The solution is to grp all common attr in one interface & extend interface if type specific actions exist.

v) DIP: dependency inversion principle.

High level modules should not depend on low level modules. both should depend only on abstraction