

SOLID:

- 1) Design Principle: guidelines for designing software based on OOP
 - Aims: understandability, extensibility, maintenance
 - Avoids: rigidity, fragility, immobility and viscosity
- 2) Single Responsibility (S): Each class only has 1 job
- 3) Open / closed (O): open for extension, closed for modification
 - Meaning we can add new functionality without altering old code
- 4) Liskov Subs (L): objects of a superclass can be replaced by its subclasses' objects without damaging the program
- 5) Interface Seg (I): don't force the clients to dep on things they don't use
- 6) Dependency Inv (D): high-level class doesn't depend on low-level one

GRASP: 9 principles

focus on assigning responsibilities to classes and objects in OOP

- 1) Creator (pattern): object created by the class that has the most info to create it
 - When? aggregation or containment; tracking; close use; initialization data
- 2) Info Expert: arg resp to the class that has the essential info to create it
 - Note: consider the type of resp and what info are needed
- 3) Low Coupling: reduce dep between classes as much as possible
 - How? use interfaces/ abstract classes, dep injection, compo > inheritance, behavior encapsulation; avoid bidirectional relationship

- High Cohesion : Similar to S in SOLID
- Controller : Keep the UI separate from the Model
 - How? When handling a user event, ask it to a non-UI class that represents the whole user / a use case scenario the user responds.
- Polymorphism : Flexible and dynamic behavior in an application
 - Essential for low coupling
- Facade Fabrication : ask resp to classes that don't represent a concept
 - When? when we cannot find a sol within the problem domain
 - middle party
- Indirection : Intro an intermediate class or object to mediate between classes
 - When? when coupling is high
 - Similar to "Don't talk to strangers"
- Protected Variation: Protecting elements from the change of others
 - How? One element's change doesn't make others change

Refactoring : small, inde techniques to keep programs readable, understandable, maintainable, improving design without changing functionality

- Key feature: ensure program works at every detailed step
- Why? Req changes, design improvements, programmer's development
- Outcome: make prog more flexible
- EI: renaming, collapse hierarchy, consolidate / decompose condns,
 - (ex: merge) (ex: combine condn that → same result)
 - extract class / method (ex: replace code fragment by a method)
 - replace complex condns by method
- (ex: a class doing ≥ a work → make a new class)

- When? When adding a new feature / review code / fixing bug
- When not? broken code / close deadline / no need to

- ⇒ Code Smell: A surface indication that usually orgd to a deeper prob in sys
- Usually found when refactoring / examining code
 - How come? rushed design / technical debt (lazy 1st → work work later) disregard
 - Right way: use best practices + develop a design that can scale
 - Fast way: "hacked together" design, faster
 - Bloaters: something ↑ too large and out of control (> 10 lines)
 - Sign: contain long / duplicated code; long class method; data clump; long para list
 - How come? dev's may be tempted to put new feature into an existing class
 - Treatment: extract class / subclass / method; try deleting a num to see if code still makes sense, extract class; breakup algo, use methods; pass the whole object as data (Preserve whole object)
 - OO Abuser: sol doesn't fully exploit oo design
 - Sign: complex "switch" / sequence of "if"; refused request (subclsses refused to use all parent's methods)
 - Why? For refused request, dev's are tempted to use inheritance to ↑ reusability
 - Treatment: Extract method; polymorphism; push down method; use interface; eliminate inheritance

- change brevetter : st that hinders changing / upgrading
 - Sign : make modifi reqs make small changes to many other ; many changes made to only one class (Divergent change)
 - Why? For divergent change : poor structure / copy paste too much
 - Treatment : combine to a single class ; Extract class / method
- Disposable & : unnecessary things
 - Sign : duplicated code ; lazy class ; data class
 - Why? For lazy class, a class may become very small after several refactoring, or it's designed for future dev that is never done
 - Treatment : Extract method ; Inline class ; Move / Extract method , encapsulation
- Coupler : closely connected routines / modules
 - Sign : a method seems more interested in a class than the one it's in ; classes know too much about each other ; class only do one action and delegate work to other classes
 - ↑
Inappropriate intimacy
 - ↑
middle man
 - Why? For middle man, moving to much from a class resulting in it can only delegate work
 - Treatment : move method ; move method / field , change " \Rightarrow " association to " \Rightarrow or \leq " ; inline class

Design pattern : typical solutions to common probts in software design
 (like a blueprint if algo is like a recipe)

Category of Pat

- Creational Pat: try to create suitable objects → ↑ flexibility and code reuse
- Structural Pat: explain how to assemble & clarrer ~~the~~ larger structure
(still flexible and efficient)
- Behavioral Pat: algorithm & argn of resp of objects

Singleton Pat: t Crea, ensure only one instance of a class ever gets created and ∃! one global way to access it

- Components: private constructor, public static method, private static instance variable
 - (to access)
 - ↑ only create an instance if null
 - (store one instance)

Strategy Pat: t Beh, define a fam of algo by putting them in separate class and making their objects interchangeable

- ↑ maintain & grow
- Components: A strategy interface includes # concrete strategies, a context class that implement the strat interface to choose the strategy to use

Factory Method Pat: t Crea, using a factory method to create instances of a class instead of directly create them

- Why? changing the type of object used can be challenging . We can decouple instance creation using factory method
- Components: Interface Product includes # concrete products, a creator class that is inherited by concrete creator class espd to each concrete products

Observe Pat: t Beh, define a subscription mechanism to have multiple events issued by the "publisher" and multiple "observer" notified

- Why? Sometimes a change to sys needed to be notified to all other wrks

- Component: publisher class that holds a list of subscribers, and concrete subscriber classes (implements the interface Subscriber)

Composite Pat: E struct, all about creating tree-like structures of objects.

Treat individual objects and compositions uniformly

- Components: interface Component, extended by ^{↑ Components: delegate} all work to leaf ^{↓ Leaf: do all the work}

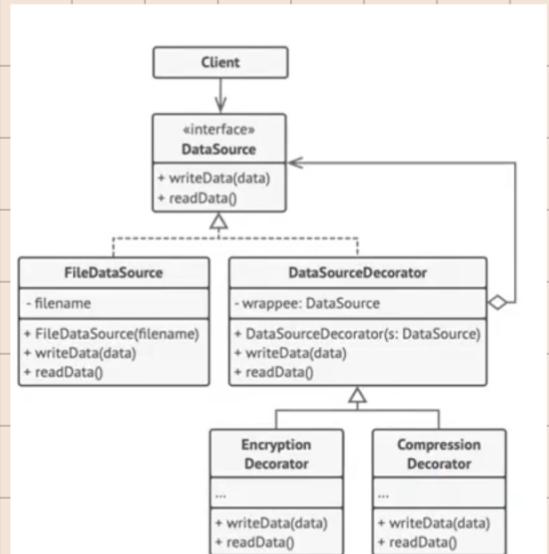
Visitor Pat: E Beh, helps add operations to objects without altering those objects themselves (use an external class for this)

- Component: Interface Visitor, # concrete visitor classes impl Visitor, # concrete element upd to each visitor class impl interface Element

Decorator Pat: E Struct, helps add features or behaviors to objects without changing the object's original class

- Components: a Notifier class
Base Decorator (where we will add features)
Concrete Notifiers inherit from Base Decorator

- Why?: Dynamic Extension, avoid class explosion, Single Resp Prin, Open / Closed Prin



Software Assessment & Testing :

-) Dynamic Verification : test sys until it fails (executing code in a runtime env), include various levels
-) Static Verification : examining code w/out executing it. Use tool to analyze code . Can be performed very early on. Doesn't req runtime env or compiled code
-) Inspection / Review / Walkthrough : Human activity , group-based + manual
 - Inspection : formal checklist + moderator
 - Review : less formal + peers
 - Walkthrough : informal + author
-) Formal verification : Math-exhaustive , can be time-consuming + expensive
-) Testing Stages:
$$\underbrace{\text{Dev Test}}_{\substack{\text{within orga} \\ (\text{black box})}} \subseteq \alpha \text{ Test} \subseteq \beta \text{ Test} \subseteq \gamma \text{ Product Release}$$
$$\underbrace{\text{outside orga}}_{\substack{(\text{open box})}}$$
$$\underbrace{\text{outside orga}}_{\substack{(\text{clear box test})}}$$
-) Testing technique : opaque box Test
-) Test Driven Development (TDD) : Test written before code , and ~ all production code have test unit
 - Rules:
 - make it fail
 - make it work (as simply as possible)
 - make it better (re-factor)
-) Test Cycle:

```
graph TD; A[Run Test] --> B[New req]; B --> C[Write new test]; C --> D[Run Test]; D --> E[Refactor]; E --> F[Code]; F --> A;
```
-) Why TDD ?
 - ① ↑ dev discipline
 - ② provide incremental specification
 - ③ Avoid regression errors
 - ④ ↑ confidence while changing