SOLID :

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

GRASP : 9 priciples

focus on assigning responsibilities to classes and objects in OOD

- Creator (Pattern) : object created by the class that has the most info to create it
  - When ? aggregation or containment ; tracking ; close use ; initialization data
- 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
- Low Coupling : reduce dep between classes as much as possible
  - How ? use interfaces / abstract classes, dep injection, compo > inheritance, behavior encapsulation, avoid bidirect relationship

- High Cohesion : Similar to S in SOLID

- Controller : Keep the UI separate from the Model
  - How? When handling a sys event, asg it to a non-UI class that repre the whole sys / a use case scenario the sys responds

- Polymorphism : Flexible and dynamic behavior in an application
  - Essential for low coupling

- Pure Fabrication : asg resp to classes that don't repre a concept
  - When ? when we cannot find a sol within the problem domain

- Indirection : Intro an intermediate class or object to mediate between classes (middle party)
  - When ? when coupling is high
  - Similar to "Don't talk to stranger"

- 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, understand, maintain, improving design without changing functionality
- Key feature : ensure program works at every detailed step
- Why? Req changes, design improvements, programmer's sloppiness
- Outcome : make prog more flexible
- Ex: renaming, collapse hierarchy, consolidate / decompose conds,
  (ex: merge) (ex: · combine conds that → same result · replace complex conds by method)
  extract class / method
  (ex: replace code fragment by a method)

(ex: a class doing ≥ 2 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 crpd to a deeper prob in sys
- Usually found when refactoring / examining code
- How come? rushed design / technical debt (lazy 1st → mark work disregard later)
- Right way: use best practices + develop a design that can scale
- Fast way: "hacked together" design, faster
- Bloaters: something ↑ so large and out of control
  - Sign: contain long / duplicated code; long class) method; data clump; long para list (identical groups of var) (>10 liner)
  - How come? devs 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 Abusers: sol doesn't fully exploit oo designs
  - Sign: complex 'switch' / sequence of 'if'; refused bequest (subclass refused to use all parent's methods)
  - Why? For refused bequest, devs are tempted to use inheritance to ↑ reusability
  - Treatment: Extract method, polymorphism; push down method, use interface, eliminate inheritance

- Change preventers : st that hinders changing / upgrading (Shortgun surgery)
  - 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
- Dispensables : unnecessary things
  - Sign : duplicated codes ; 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
- Couplers : closely connected routines / modules (Feature envy)
  - 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 ↑ Inapporiate 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 "⇔" association to "⇒ or ⇐" ; inline class

Design pattern : typical solutions to common probs 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 & classes ⋙ larger structure (still flexible and efficient)

:) Behavioral Pat: algorithms & assgn of resps of objects

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

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

Strategy Pat: ∈ 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 inteface to choose the strategy to use

Factory Method Pat: ∈ 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 rspd to each concrete products

Observe Pat: ∈ 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 users

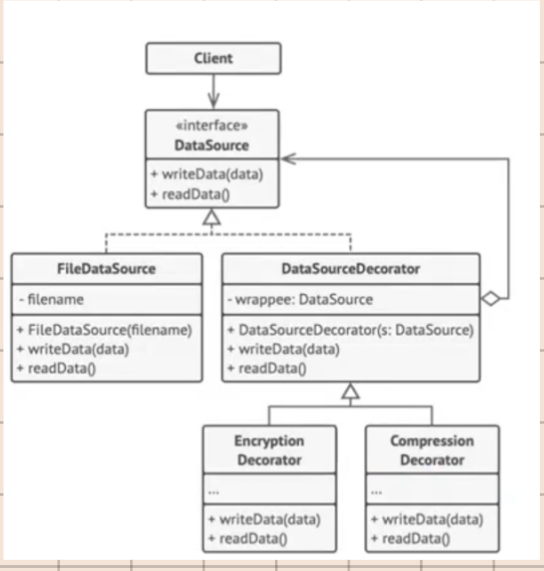
- Component: publisher class that holds a list of subscribers, and concrete subscriber classes (implements the interface Subscriber)

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 object themselves (use an external class for this)
- Component: Interface Visitor, # concrete visitor classes imple Visitor,
  # Concrete element impl to each visitor class imple 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 inherited 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 runtim 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

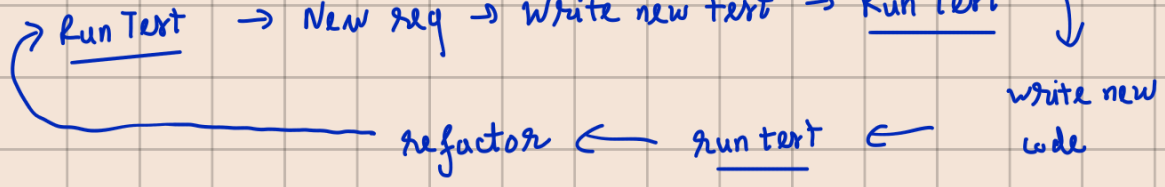
- Formal verification : Math - exhaustive , can be time - consuming + expensive

- Testing Stages: Dev Test (within orga) ⊆ α Test (within orga) ⊆ β Test (outside orga) ⊆ Product Release (outside orga)

- Testing technique : opaque Box Test (black box) — Clear Box Test (open box)

- 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 (refactor)

- Test Cycle:



- Why TDD ?
  1. ↑ dev discipliner
  2. provide incremental specification
  3. Avoid regression error
  4. ↑ confidence while changing