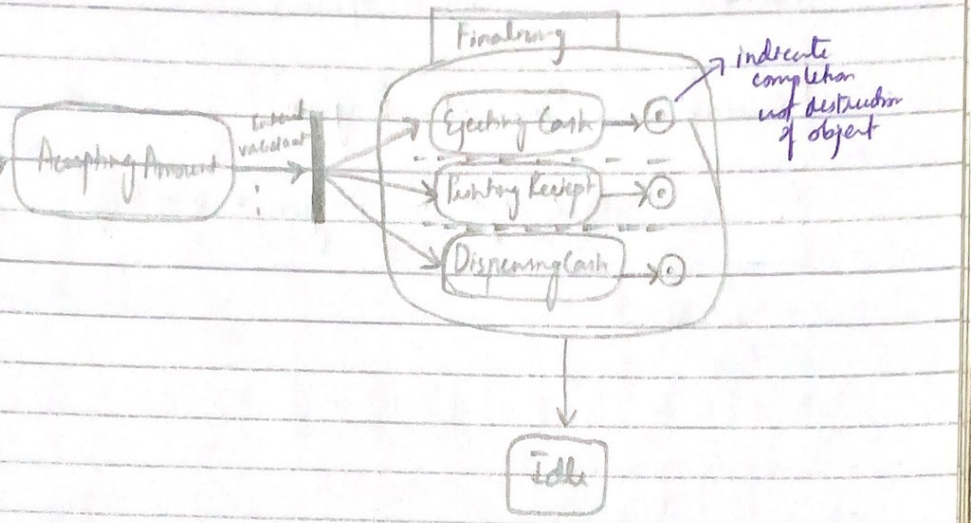
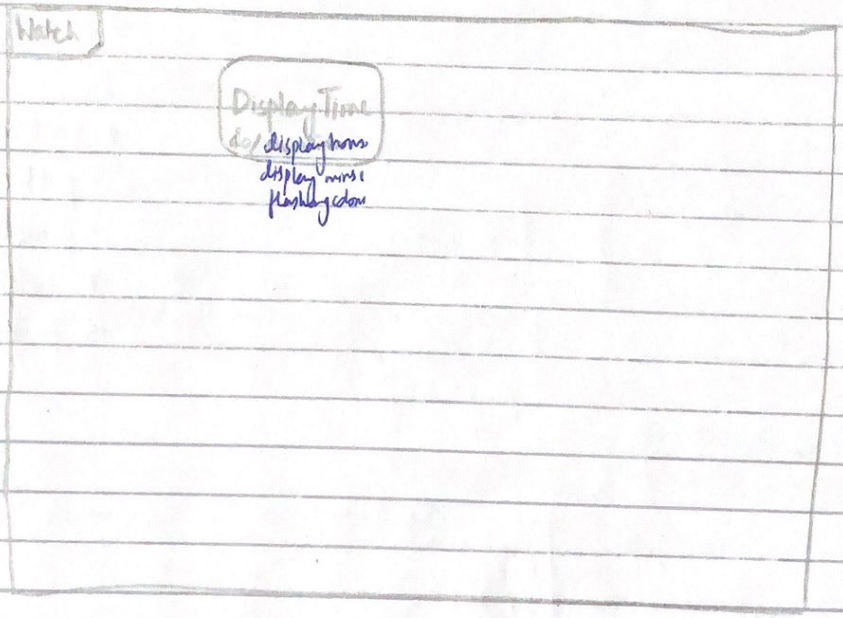
Software Design Analyses

Lecture # 13



Case Study



Software Design & Analysis

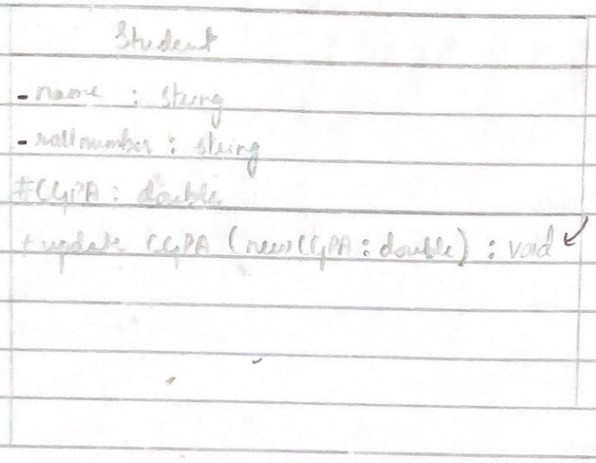
~~Importance~~

Solution Space:

Design class Diagram

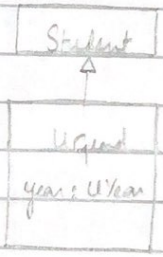
=

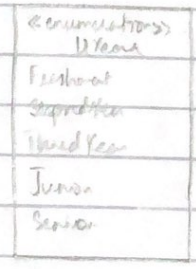
Analysis class Diagram + more things

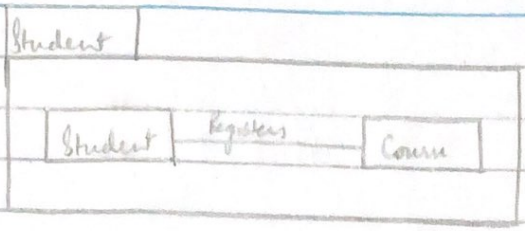


* visibility
+ public
- private
protected
~ package

only used by descendant

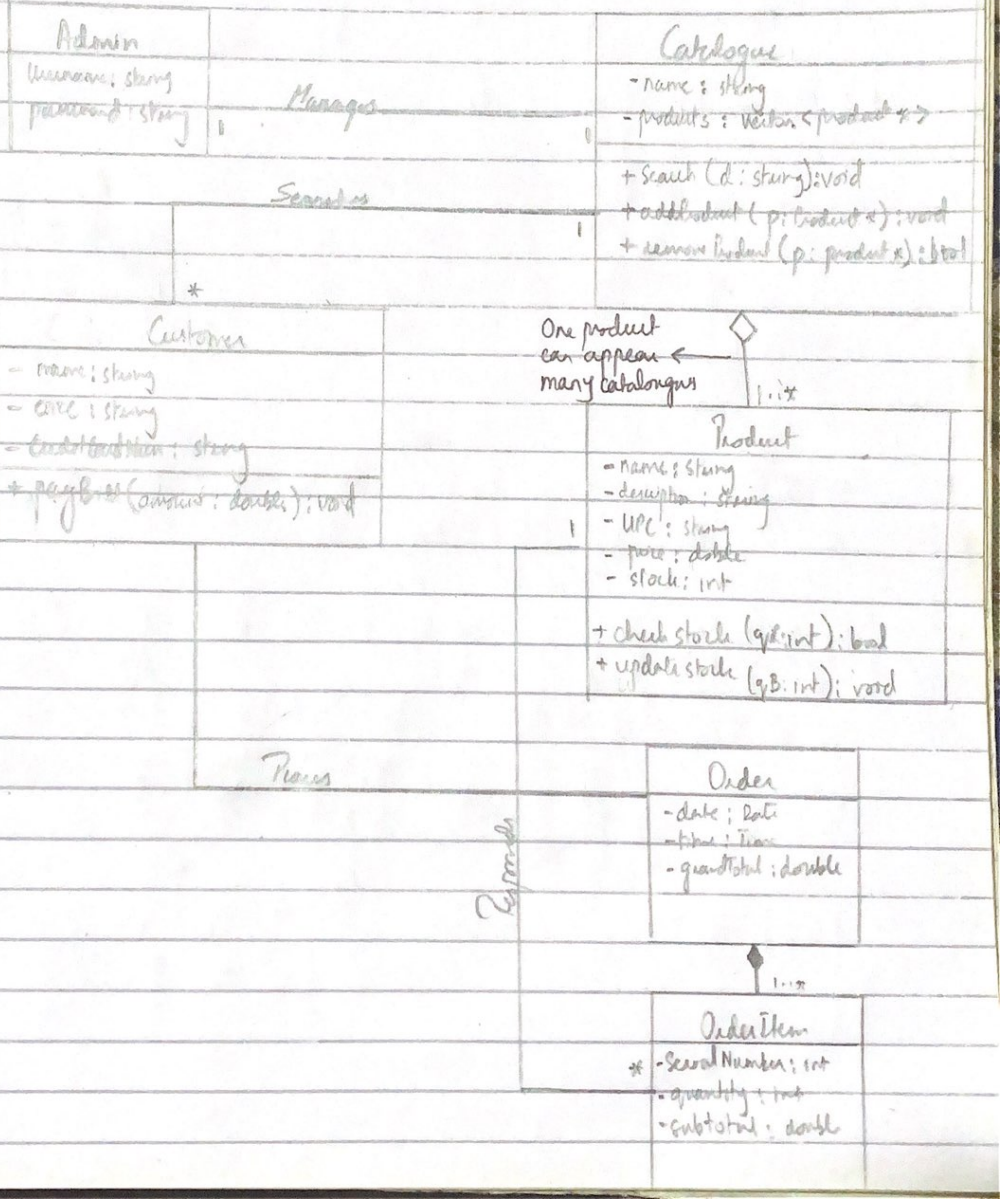




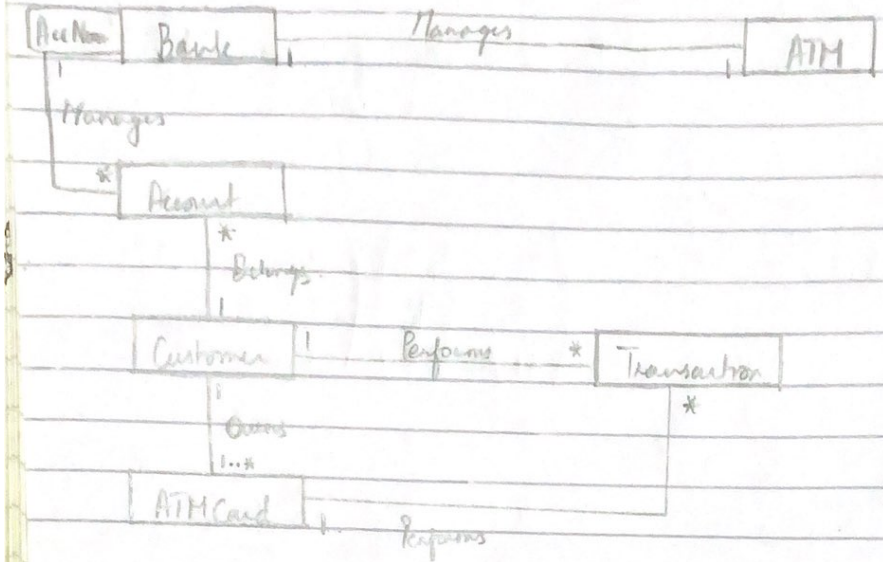


← Package diagram which means any attribute with package visibility in student class is accessible by other classes in the package which is in this case is course class..

★ Ecommerce Case Study



* Case Study ATM



ATM
- cashInHand : int
+ updateCash() :

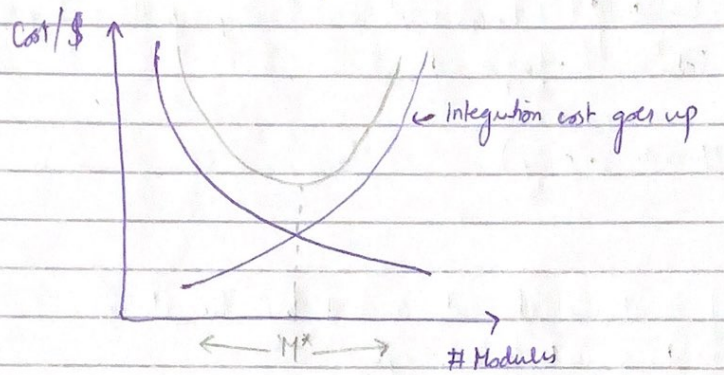
Software Design & Analysis

Design Concepts & Principles

Modularity:

~~Not~~ Monolithic vs Modular

Avoid over modularization / under modularization



We need those types of modules which are Functionally independent

Functional Independence

- → cohesion ↑ maximum "single-mindedness" Modules should do fewer things as possible
- → coupling ↓ minimize "connectedness"

Different types of cohesion

1. Functional Cohesion / Perfect Cohesion ★
   Functions just perform one task and return control eg getters / setters
2. Layered Cohesion ↓ Top to bottom
   Upper layers access lower layers not vice versa
3. Communicational Cohesion
   All functions that access same data should be a part of one class

} Higher types of cohesion

Lower levels of cohesion (Ideally we should go for higher level)

④ Procedural Cohesion

Functions called in specific order grouped in same class.

- Sequential
  - ↳ If you pass data then sequential.
- Temporal
  - ↳ Deals with Time
    Everything that happens at same time group them together.
- Utility Cohesion (Ikkatha cohesion)

* Types of Coupling (Minimize coupling)

① Content coupling

If the content of one class is changed / amened by other class.
Friend class also generate content coupling

② Common Couplings

Shared data such as global variables.

③ Routine Call coupling

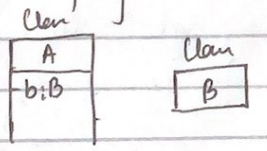
One function calls another function.

↳ Data coupling

subtype: function call each other and pass data.
Sometimes data has control information

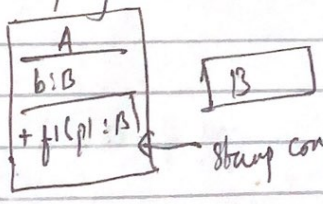
Control coupling ←

④ Type use coupling



A is dependent on B

⑤ Stamp coupling



Software Design & Analysis

Solid Principals

① Single Responsibility Principle — Single reason to change class.

② Open / Closed " → to modification.
→ to extension

③ Liskov Substitution " Sub classes should be substitutable

④ Interface Separation " Separate interfaces for separate clients.

⑤ Dependency Inversion Invert your dependencies / depend on abstraction so that code is easy to change

SDA

Object Oriented Metrics

↓

Numerical value

LK Metrics

↳ Directly related to size of software.

↳ Larger the size, more resources will be needed.

Metrics :-

(1) Number of Scenario Scripts (NSS) = # Use cases

(2) Number of Key Classes (NKC) = All classes in ACD (Analysis Class Diagram)

(3) Number of Support Classes (NSC) = Classes that are added to support key classes in DCDs.

~~(4) Average~~

(4) Average number of support classes per key class = NSC / NKC

(5) Number of Sub-Subsystem (NSuS)

→ Non of them is applicable at the level of individual class.

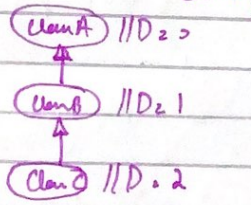
(6) Class Size (CS) = NA + NOp → no. of all → no of operations

- In case of inheritance you also include NAs and NOp of parent classes as well

(7) Number of Operation Added (by a subclass) ⇒ NOpA → Proxy for new functionality added

(8) Number of Operation Overridden (by a subclass) = NOpO → Proxy for modification of functionality

(9) Specialization Index (SI) = $\frac{NOpO \times Depth}{NOp}$



Software Design And Analyses

Object Oriented Metrics

CK Metrics

Chedemben Kemerer

(1) WMC → Weighted Methods per class

Let say a class has

$M_1, M_2 \ldots M_n$ methods

$c_1 \quad c_2 \quad \ldots \quad c_n$ corresponding complexity

$$WMC = \sum_{i=1}^{n=n} c_i$$ Let c_i be = 1

WMC considers immediate operations of that class not the inherited ones unlike NOP.

But it does consider the inherited operations that are overridden.

(2) NOC → Number of Children

Keep in count of number of immediate children of a class.

↓

no grandchildren

(3) DIT → Depth of Inheritance Tree

↳ Same as LK dept metric. start from 0.

(4) CBO → Coupling between objects

Number of classes associated to given class.

↳ any Type of association (aggregation, composition)

Simply count the number of lines incoming to a class

In a tree type symbol count every branch separately

(5) RFC → Response For a Class

Methods + Methods called by these method.

⑥ ~~LOC~~ LCOM → lack of Cohesion of Methods.

Suppose we have a class with a set of instance variables, I and set of methods M

$$I = \{i_1, i_2, \ldots, i_m\}, \quad |I| = m$$

$$M = \{M_1, M_2, \ldots, M_n\}, \quad |M| = n$$

function — used variables

$M_1 \rightarrow I_1$
$M_2 \rightarrow I_2$ } subsets of I or I
$\vdots \quad \vdots$
$M_n \rightarrow I_n$

Not taking of local variable

(I_r, I_s), $i \neq s$

uncommon

$P = \{(I_r, I_s) \mid I_r \cap I_s = \phi\}$ high value of P is very bad if ↑ Value ↓ or high, ↑ Cohesion ↓ Low

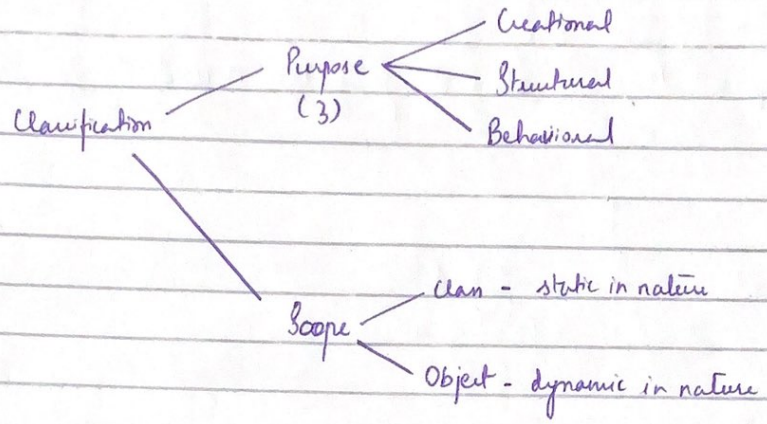
$Q = \{(I_r, I_s) \mid I_r \cap I_s \neq \phi\}$ → high value of Q is Good

$LCOM = |P| - |Q|$ if $|P| > |Q|$

Else $LCOM = 0$ otherwise

SDA

Design Patterns



Scope / Purpose

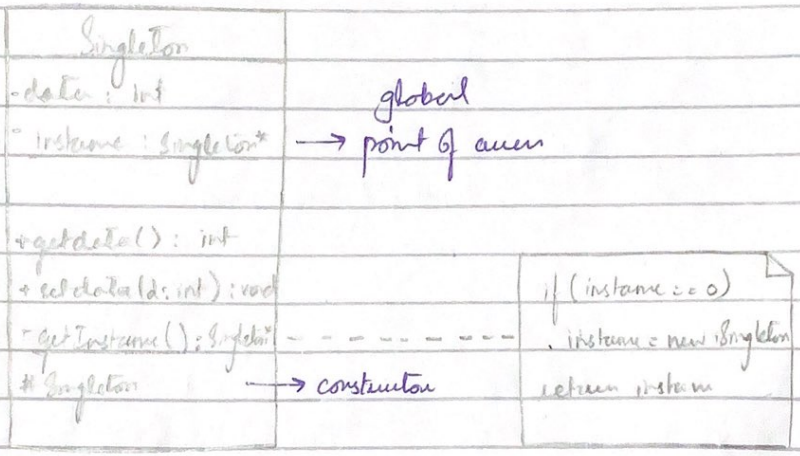
	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Template Method
Object	Singleton	Adapter Composite	Observer

Template
- Pattern Name & Classification
- Intent
- A.K.A
- Motivation
- Applicability
- Structure
- Participants
- Collaboration
- Implementation
- Sample Code
- Consequences · (v. Imp)

Singleton Design Pattern: (Object Creational)

↓

Single instance required then we this design pattern is used

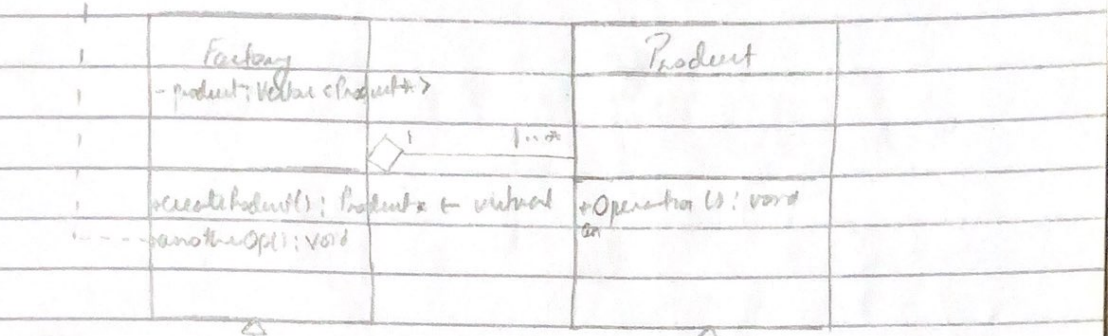


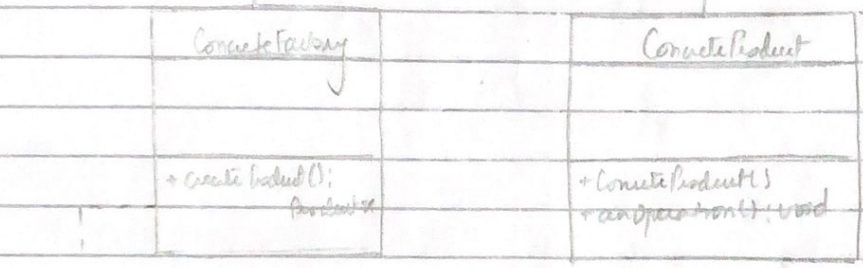
SDA

Factory Method

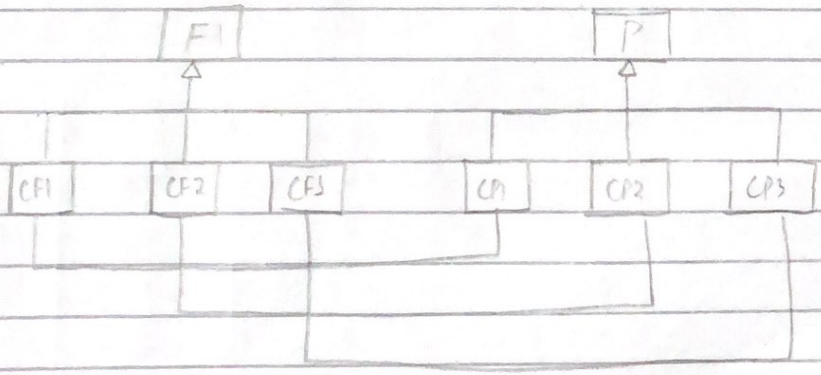
Class / Creational
Scope / Purpose

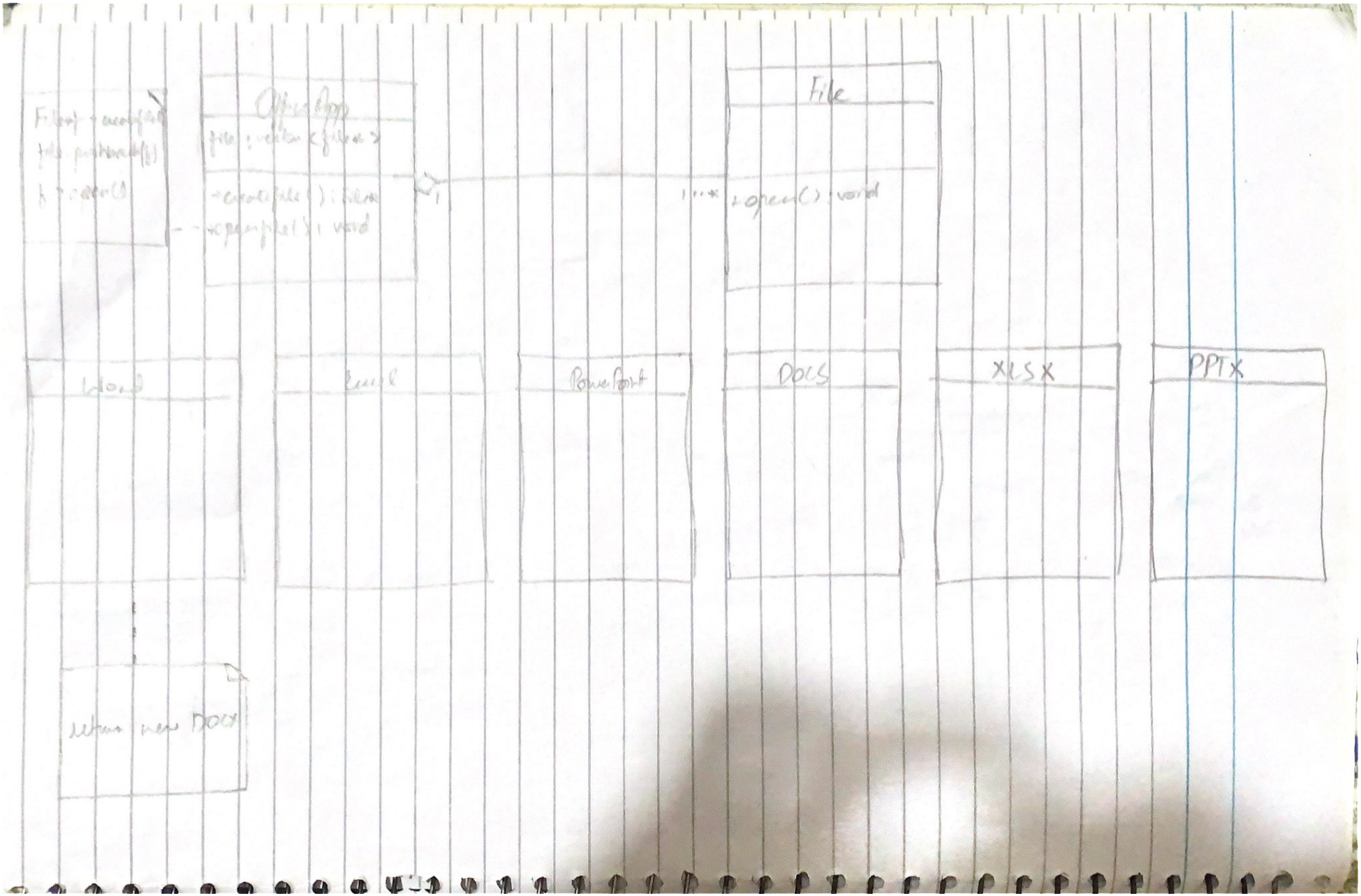
```
Product p* = createProduct();
products.push_back(p);
p->anOperation();
```






```
return new
    ConcreteProduct();
```

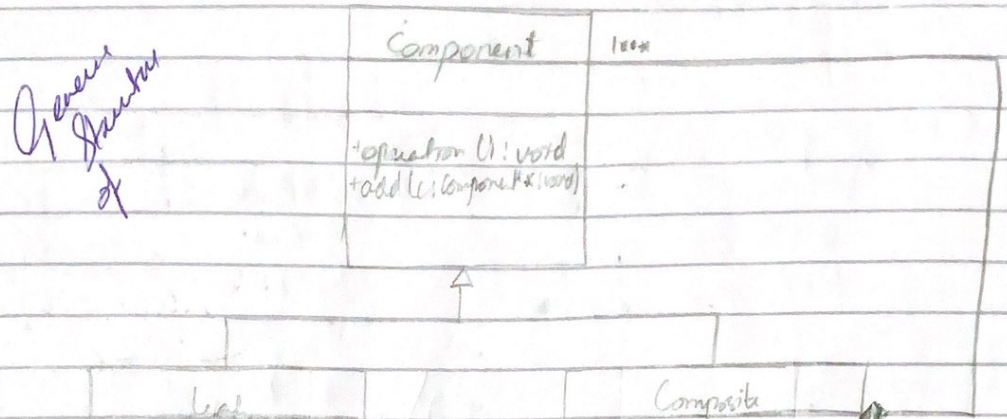



File
+open() : void
DOCS
XLSX
PPTX
PowerPoint
When new Docx

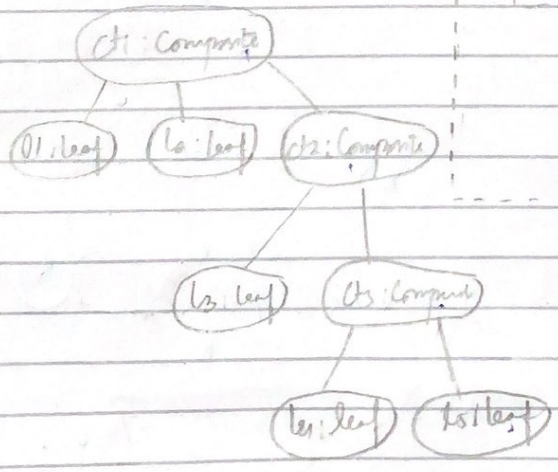
Design Patterns

Composite — Object scope — Structural Purpose

Generic Structure of




```
for(int i=0 , component.size() ; i++)
{
  components[i] -> operation()
}...
```

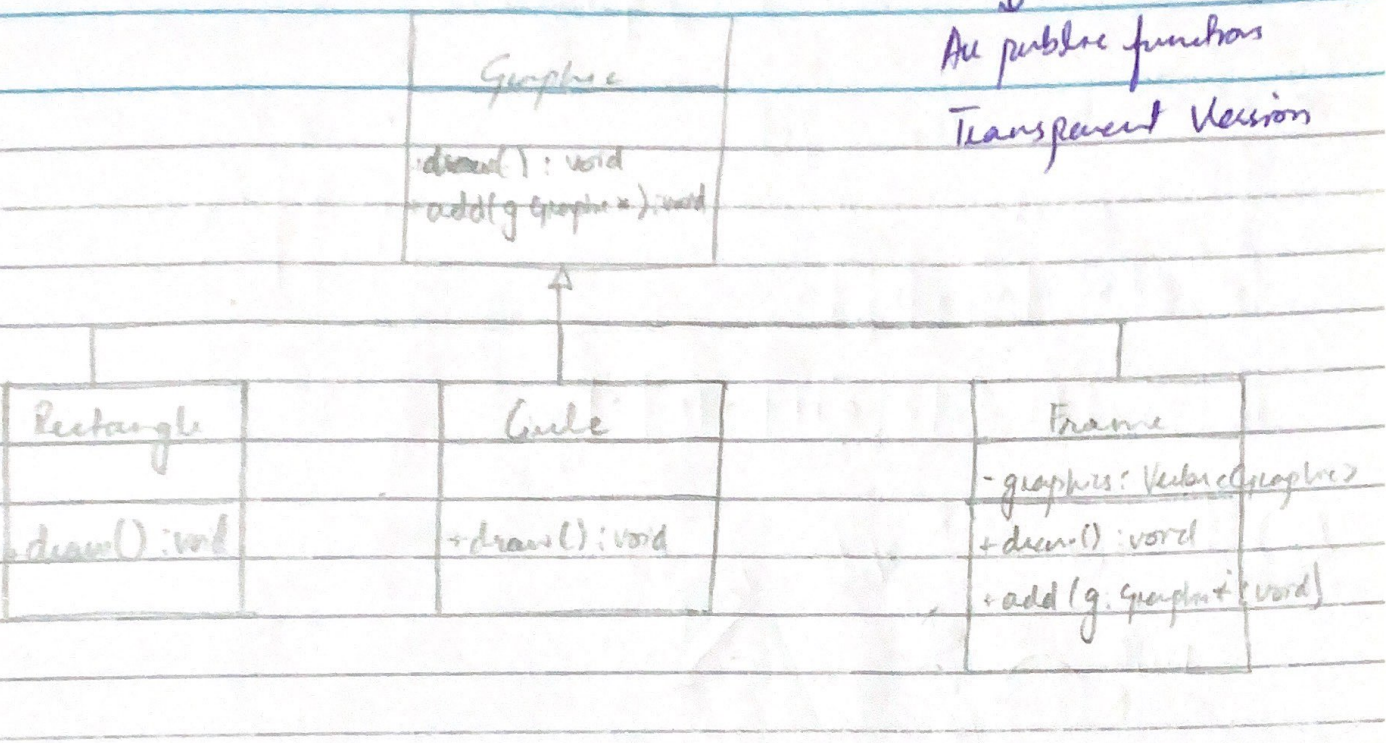



```
components.push_back(c);
```

Interface

All public functions

Transparent Version

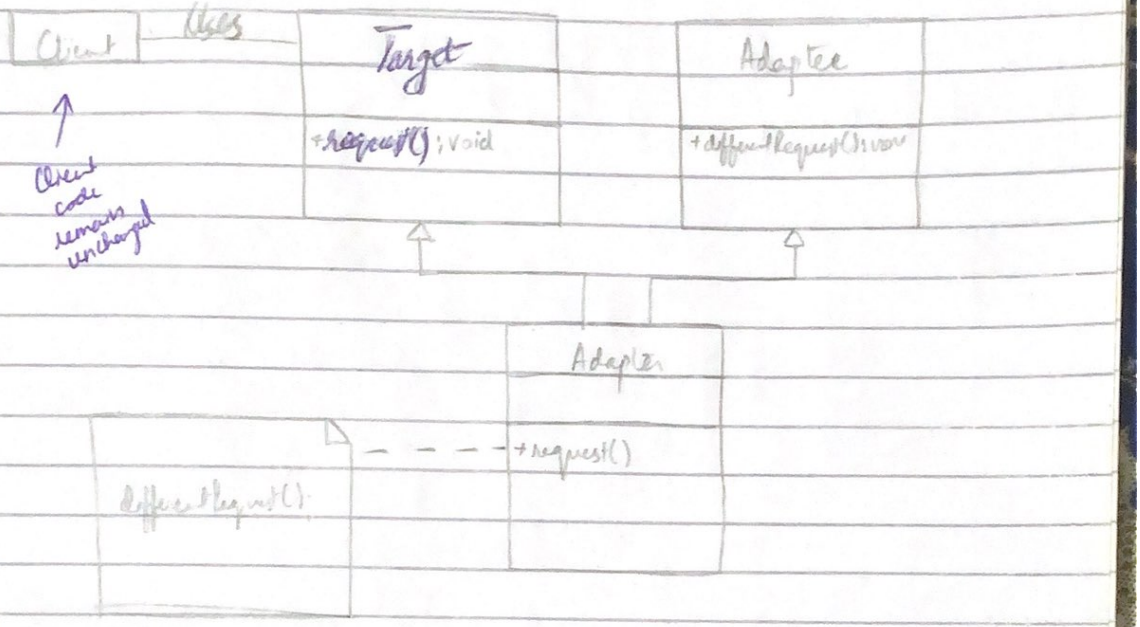


Software Design Pattern

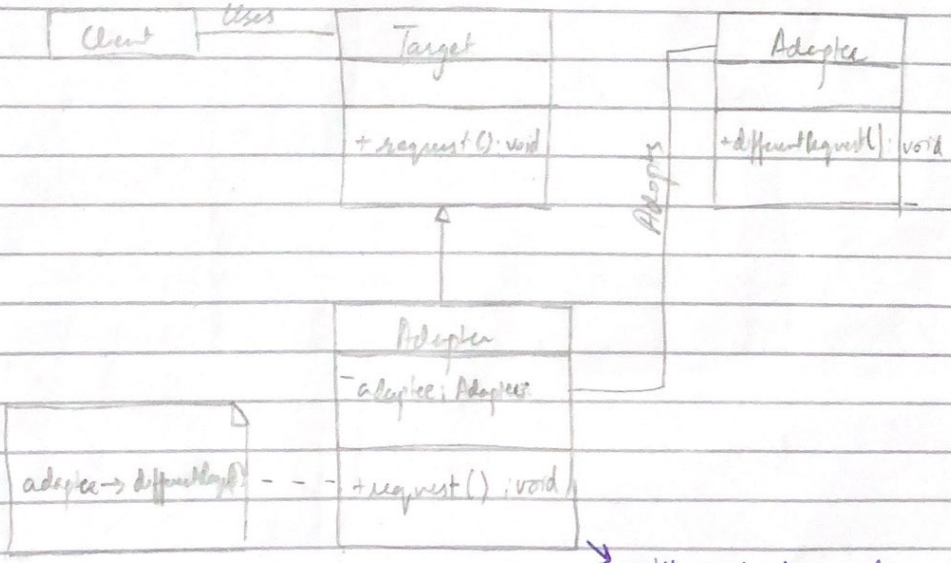
Creational → deals with instantiation

Adapter — Class/Object (Scope) — Structural (Purpose) → deals with composition

* Class Scope



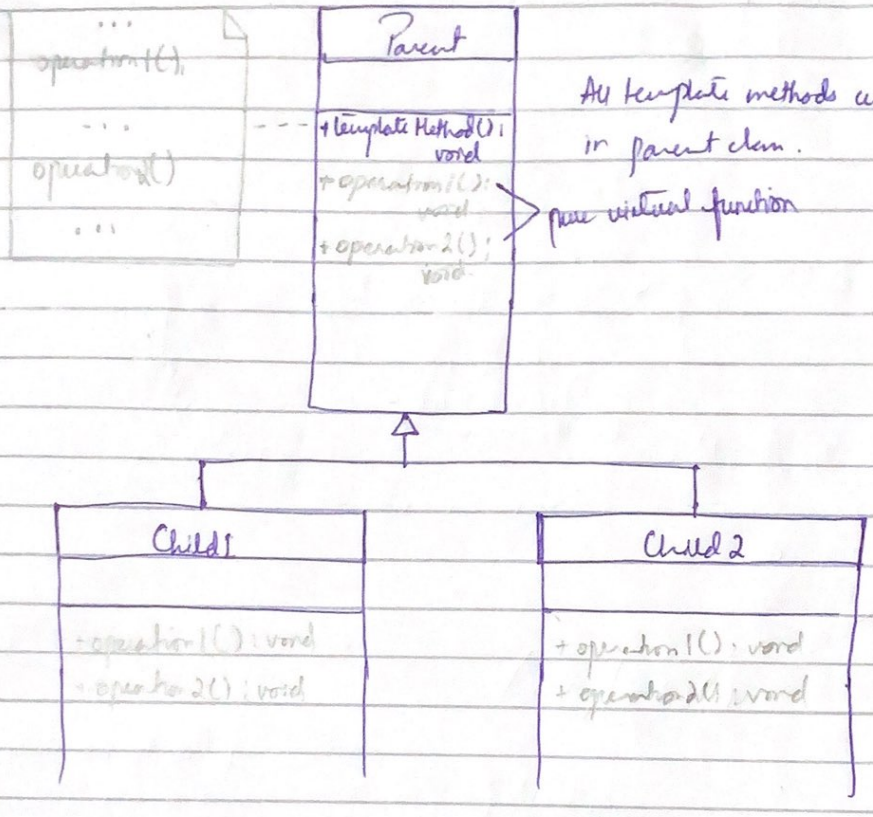
* Object Scope (More flexible)



→ will work fine with any subclass of adaptee

SDA

Skeleton Method / Template Method — Class — Behavioral
Scope — Purpose



All template methods are implemented in parent class.

Factory Method instantiates