24-01-2024

SDA L#01

Architecture: high-level design

Software: code + documentations + data

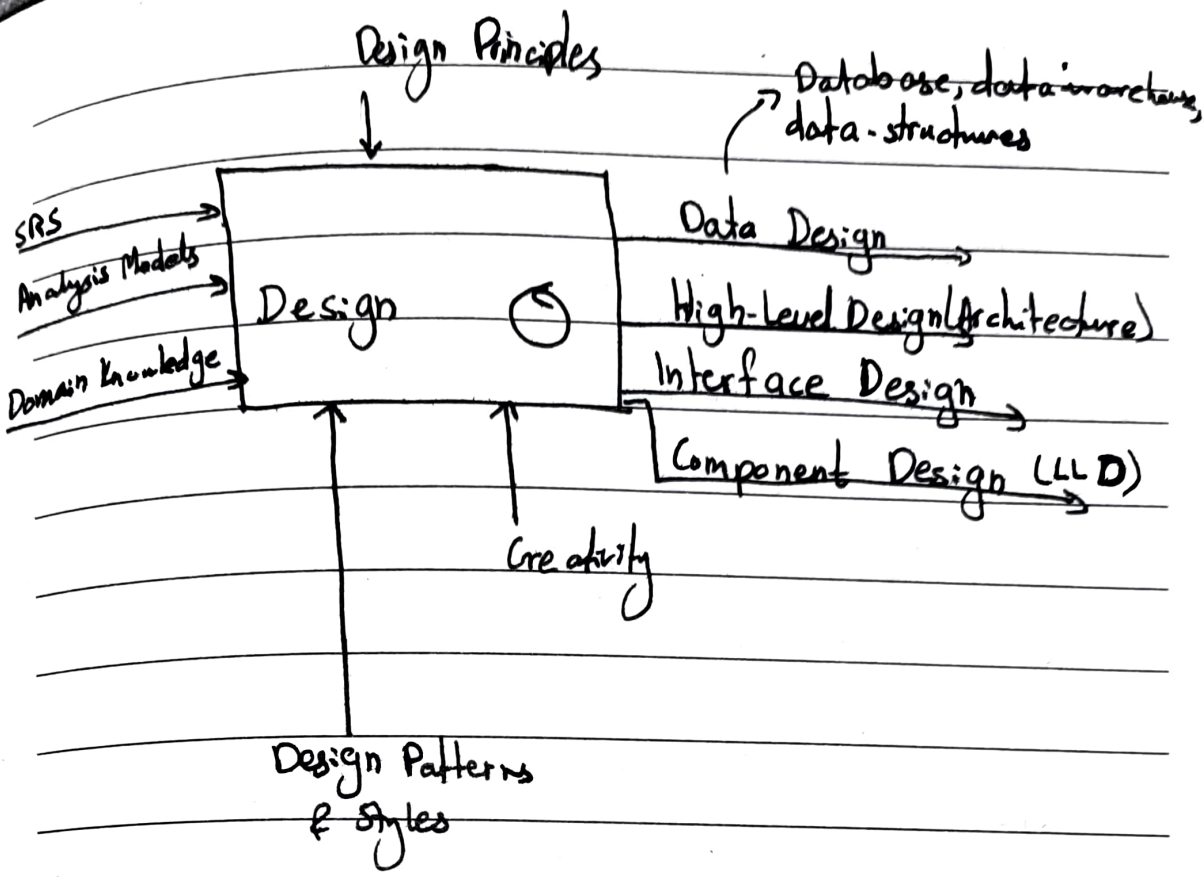
1945: ENIAC made in University of Pennsylvania, for the military. Software development began in the late 40s. Late 50s: Fortran released: high-level language. People with math or engineering degrees wrote programs. SE isn't very mature, but has made some progress. 1992, the first browser was made. Because software is important, it should be built carefully. Engineering is applied science.

Problem State { Solution State.

Inception → RE → Analysis → Design → Implementation → Testing → Deployment/Delivery

Operations Maintenance ← Obsolation/Retirement.

- Functional and non-functional requirements affect the architecture.
- Data-design is also design. HCI is design. Algorithms are design.



External interfaces: talk to other software
Internal interface: modules talk
User interface: human users.

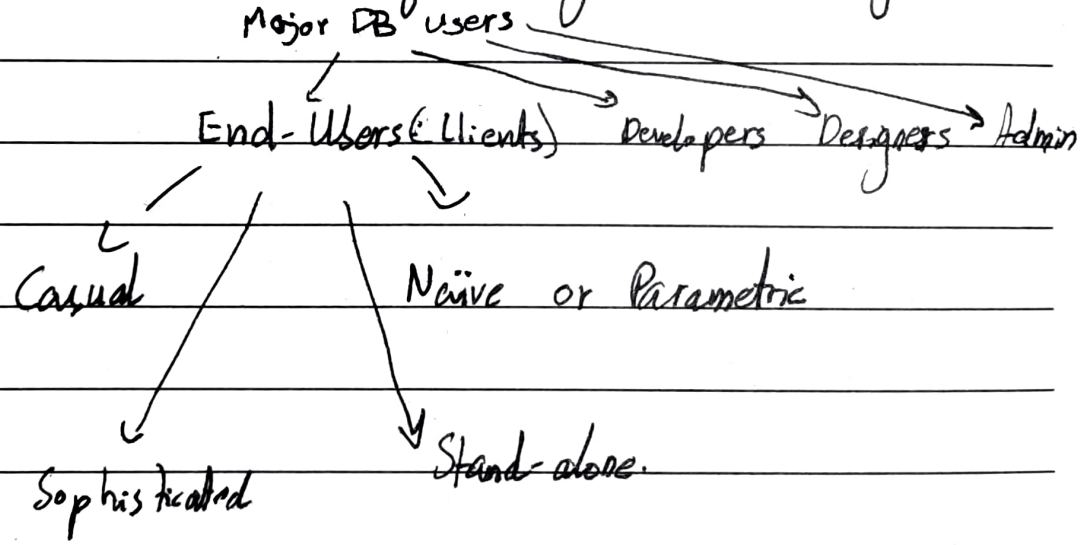
- Almost all design and engineering fields involve modelling activities.

25-1-24

DBS L#02

- It is required only to know the column names and data-types, not the actual data.
- Data can be retrieved independently of the way it is stored in the table.
- Data Abstraction: storage details hidden, users are given interface.
- Support multiple views of the data: different role-based views, show only desired data.
-

CMMI : Capability Maturity Model Integration



SPA L#02

Professional / Industrial Software

Measures of software size: Complexity:

- Lines of code
- size on disk
- no. of semicolons

- Large software is difficult to understand because of human limitation.

- Team: provides efficiency, productivity.

"If you review your own work, you tend to be mild, you don't tend to be ~~wise~~ ruthless."

- Marketing team is a proxy for the real customers, providing requirements from market research.

Personality Models

- SS - SC

- Risks ----- > countered using mitigation techniques.
  - Technical
  - Non-technical
    - personnel turnover.

The left whiteboard:

The whiteboard is, well, a whiteboard. It has a white surface which should ideally be bright cool white. In its present states, it has some leftover marks from dry-erase markers and some substance which is unknown in composition but seems brown and sticky like dried glue. The whiteboard is mounted about a meter from the platform below it, when measured from the bottom. It is rectangular: wider than it is tall. A metallic border covers its edges.

"What is important and what isn't depends on the problem domain."

Design Concepts:

- Abstractions: ~~descriptions~~ anything like descriptions, models, ~~stim~~ simulations, etc. of real things. It is selective examination: focus on necessary details but hide details. It deals with complexity.

- Decomposition/ Refinement (step-wise): divide & conquer.

- Modularity: non-monolithic. Too much modularity creates too many connections. Balance between over- and under-modularization.

Module size ↓ Module cost ↓ Integration cost ↑

- Functional Independence: minimize coupling, maximize cohesion. Coupling is the degree of connectedness. Cohesion is the degree of single-mindedness. This allows reuse and these modules are mature.

- Refactoring: changes to internal structure for improvements without changes to external structure or working.

- Encapsulation: grouping things together

- Data Hiding: hiding, giving limited access.

SDA L# 03

31-1-24

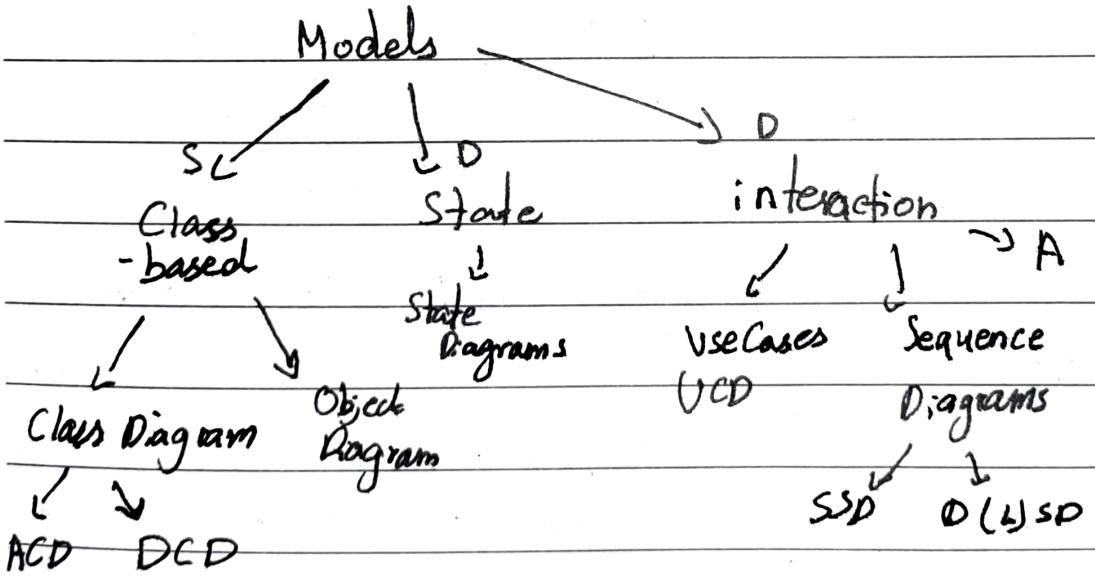
SDLC:

Inception → Requirements Engineering → Analysis → Design ----
----→ I/P/C → T → D/D → O-M → O/R.

UML: Unified Modelling Language

1997 → UML is first version.
2000 → UML 2.0 Latest: 2.5.1

Booch Notation ; Object Modelling



Class-based models are static while others are dynamic.

- Use cases start with verbs → lower camel case
- Non-human actors in box, name inside.
- Human actors stick figure, singular nouns. First letter capital
- No arrows

Class Diagrams:

Analysis CD: no implementation details:
- datatypes
- return types
- visibility
- getters/setters

Design CD:

feature: attribute or operation.

- private
+ public
# protected
~ package

- class name: upper camel case, center line, singular noun
- features: left aligned, lower camel case.

operation op → parameters unknown

op() → no parameters

op(...) → known parameter(s)

- Models to be drawn with a pencil even in exams.

~~Student~~

- Association lines should follow a rectilinear grid: —, |, or ⌝
- Association names: italics

Symmetric association: same in all directions

Multiplicity → 2

→ 3..6

→ * = 0..*

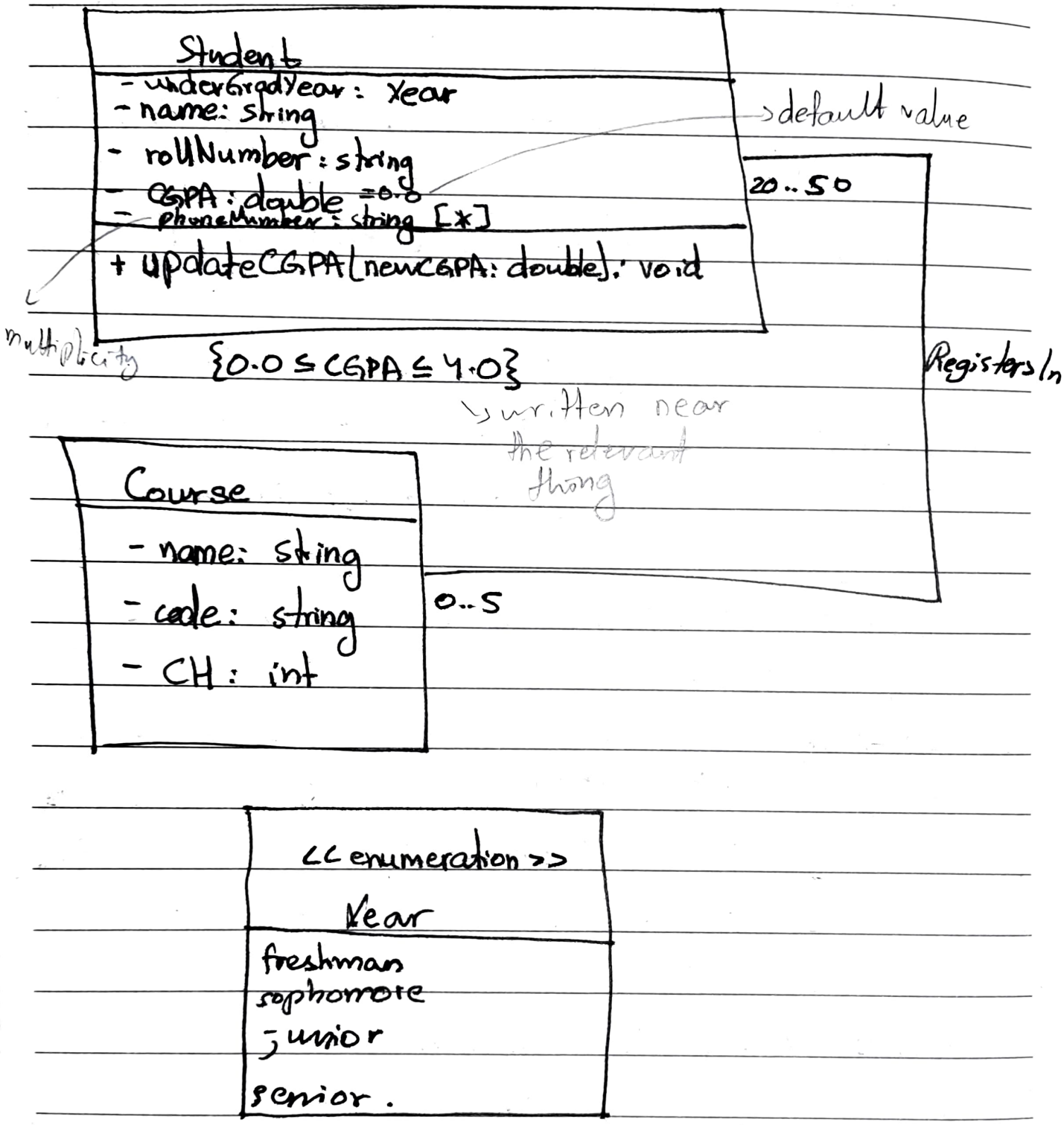
→ 1..*

CD: A graph with classes as nodes and associations as edges.

Cardinality: similar to multiplicity.

↓ Actual count

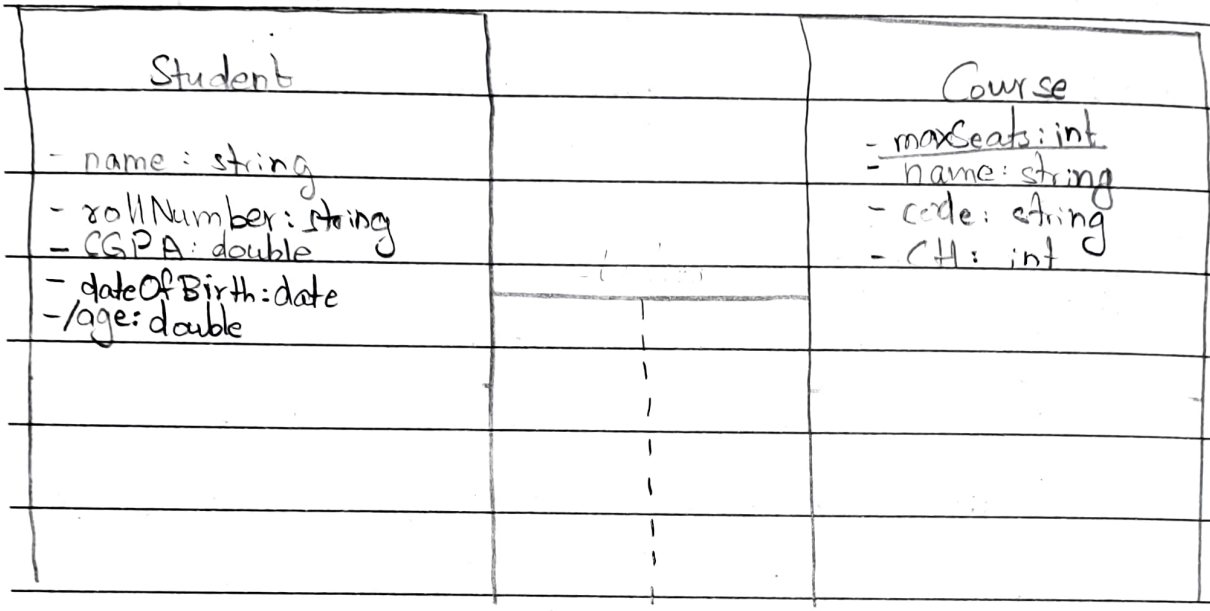
↓ Constraints on counts.


Student
- underGradYear : Year
- name: String
- rollNumber : string
- CGPA : double =0.0
- phoneNumber : string [*]
+ updateCGPA(newCGPA: double): void
default value
20..50
multiplicity
{0.0 ≤ CGPA ≤ 4.0}
written near the relevant thing
Registers In
Course
- name: string
- code: string
- CH: int
0..5
<<enumeration>>
Year
freshman
sophomore
junior
senior.

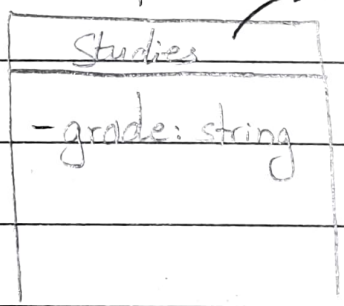
enumeration: assume a single value from a finite data set.

SPA L#04 2-2-24

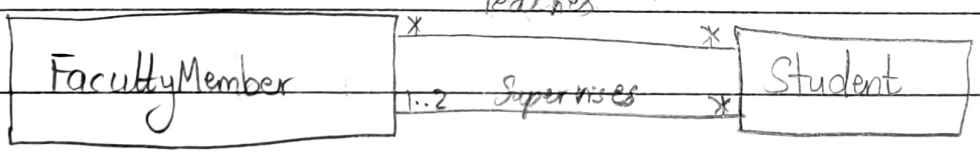
Class Diagrams:

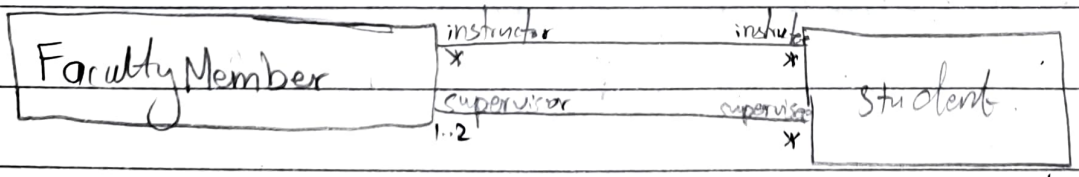


{age = currentDate − dateOfBirth}



→ non-italicized.



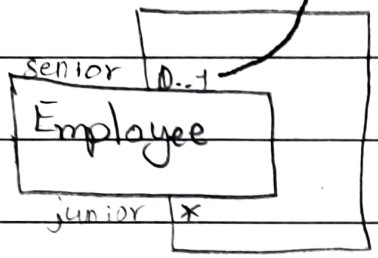


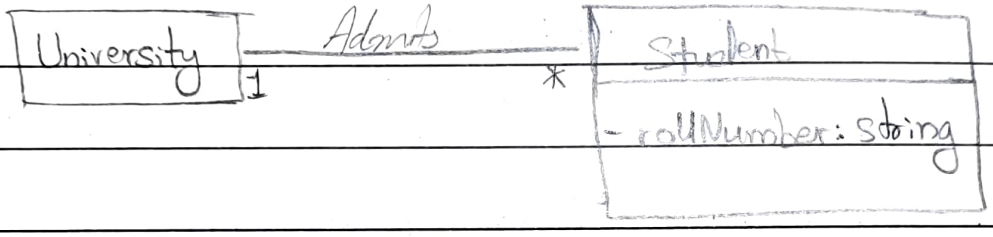
- Association class: an association which is also a class. Can be a verb while normal classes are singular nouns.
- Derived attributes: like age from DoB.
- Derived association: Student studies course taught by FacultyMember
  studies from

- object scope for normal variables; if the value is same for each object, make it a class scope variable (like static variables).
  - customary to list all class scope features at the top in UML.
-  : may or may not have functions
   : no functions if third compartment is empty.

- Multiple associations may exist but all of them should be labeled.
- Lower camel case for association end names.
  - May use both end names and association names, use at least 1.
- Association name in italics.

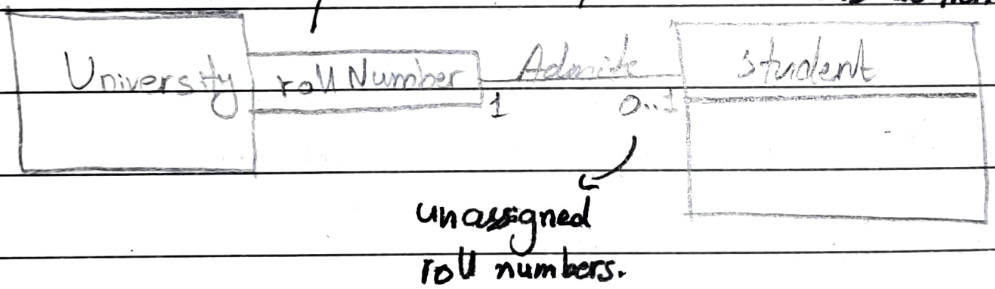
Most Senior employee not managed by anyone.



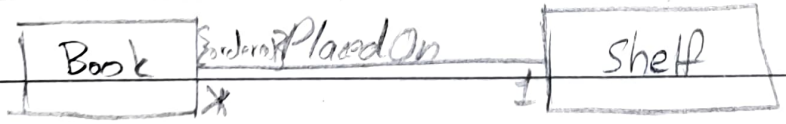


↳ same rollNumber values possible.

↗ Qualifier ↗ A qualified association.



unassigned roll numbers.



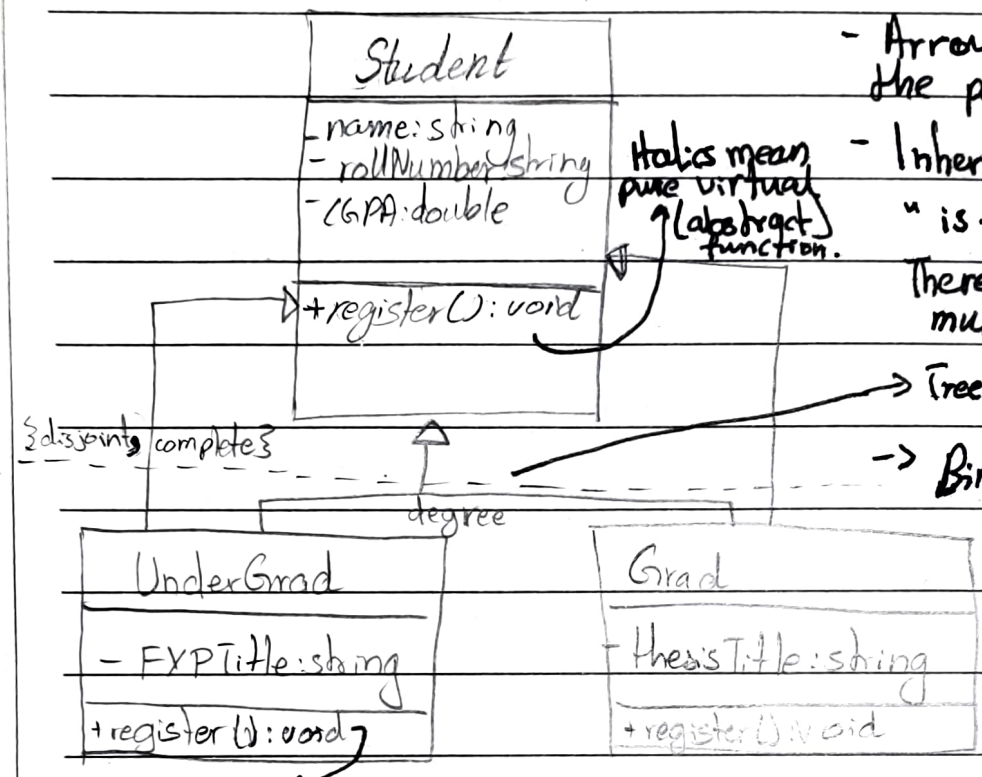
- Qualifier Dissociation.
- UML keyword "ordered" means that a collection or set isn't random but it follows same order. These are ordered sets.
- UML keyword "bag": order unimportant but duplicates are allowed. These are bags or multi-sets.

Duplicates Allowed?	Order Important? Yes	Order Important? No
Yes	sequence	ordered
No	bag	Set

circled words are UML keywords which can be used as constraints

- enumeration is also a keyword.

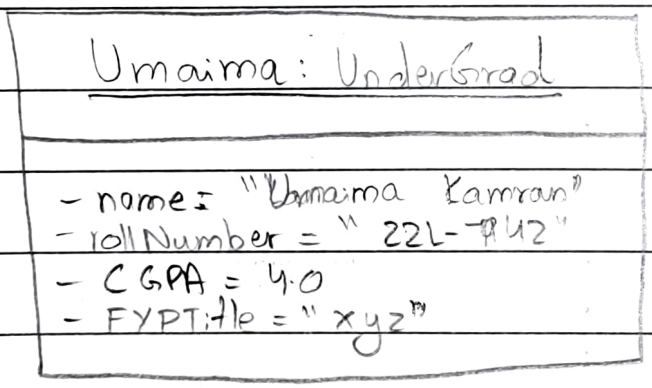
Italicized name means abstract class.



- Arrows always point to the parent/superclass
- Inheritence is an "is-a" relationship. There is no need for multiplicity.

→ Tree type shortcut (a custom)
→ Binary relation

normal font (non-italicized) means concrete function.



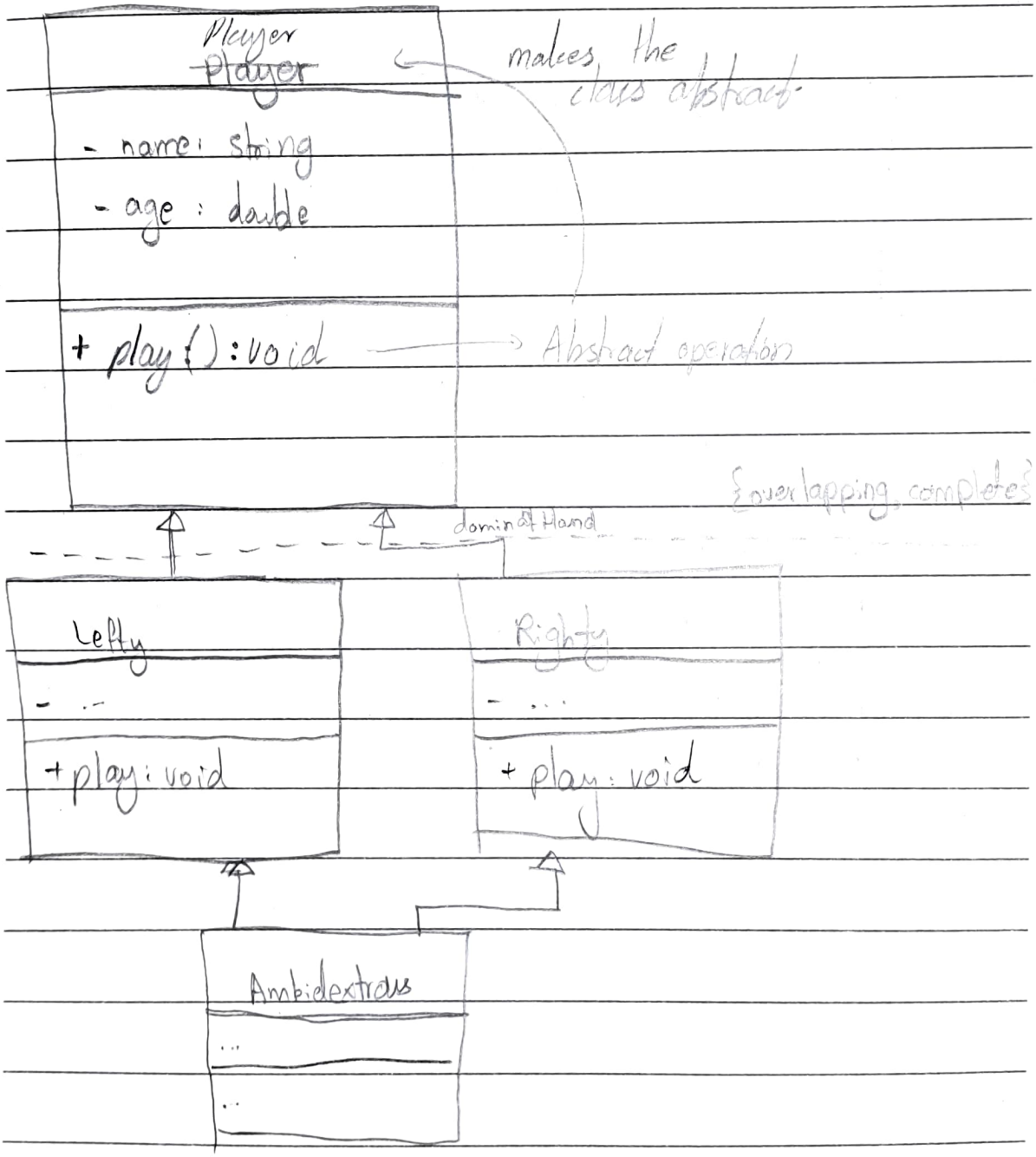
- Object diagram for above class diagram
- Functions not shown in ~~class dia~~ object diagrams.

- "disjoint" and "overlapping"
- "complete" → no other child class.
- "incomplete".
- Generalization set named: aspects of generalization.

Dog Ears

This is a comment.

Multiple Inheritance



Inheritance: is-a

Aggregation, composition: has-a (whole-part)

◇ weaker

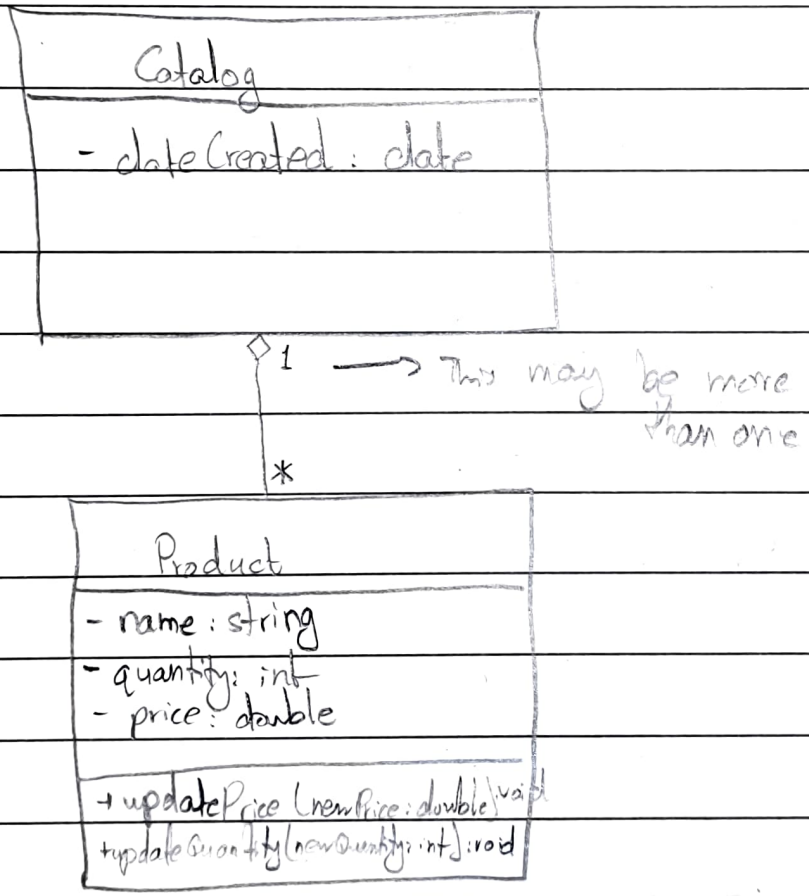
◆ stronger

Diamond attached to the "whole" part.

: C

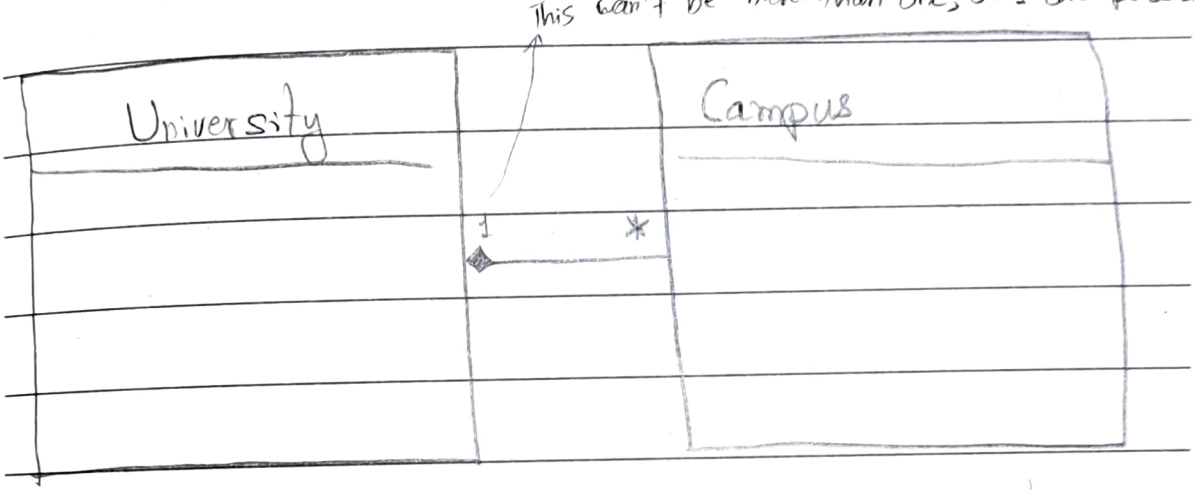
Hollow Diamond

→ independent existence possible, lifetimes not coincident.

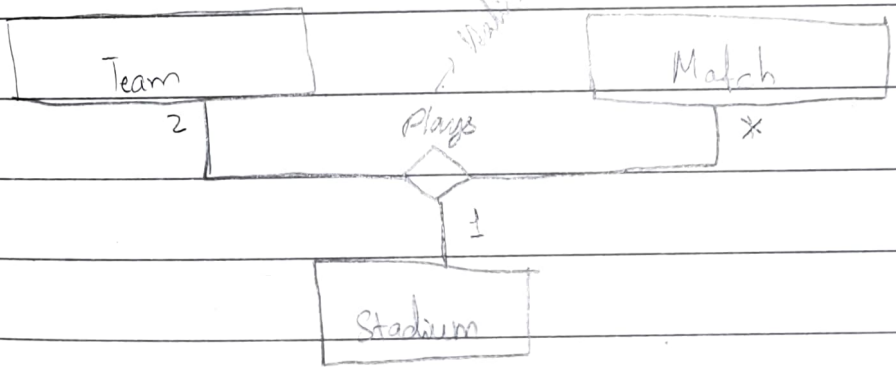


Composition: has-a (whole-part)

↳ filled diamond at the composition class, not the constituent class.



Ternary Associations connect three classes. Use with care.



Team, match are many ends, stadium is a single end. This is a fake ternary association. True ternary association has many multiplicity at all three ends. Fix two ends at 1 to determine third end's multiplicity.

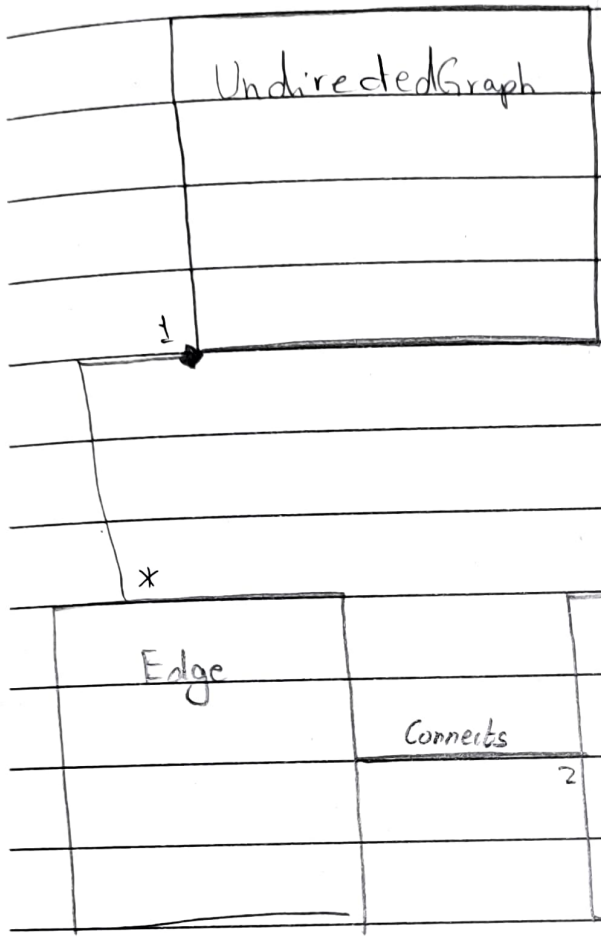
Genuine Ternary Associations.

Cannot be split into two binary associations without losing information. Fake ones are broken down into

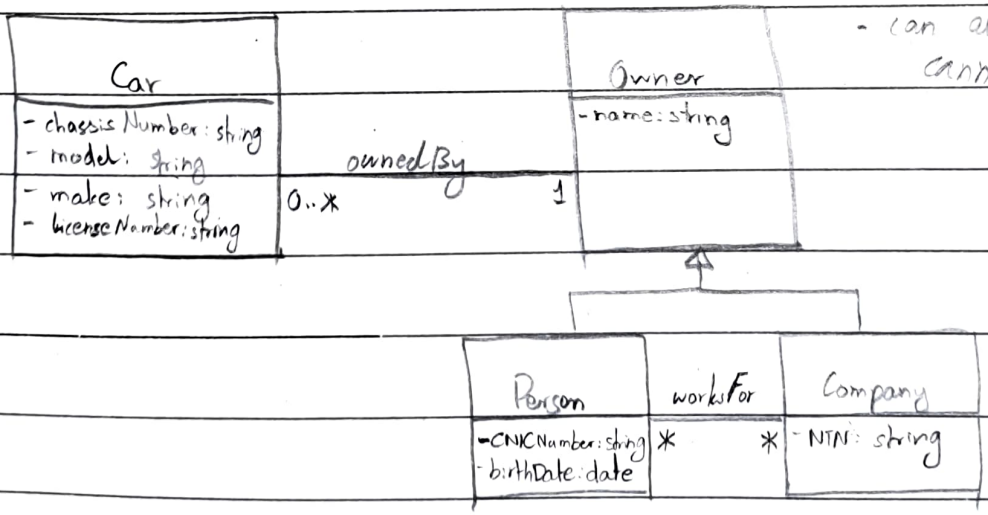
1- Keep project scope in mind.
2- Solve an unsolved problem.
3- Practical problems.
4- Something to proudly shout about.

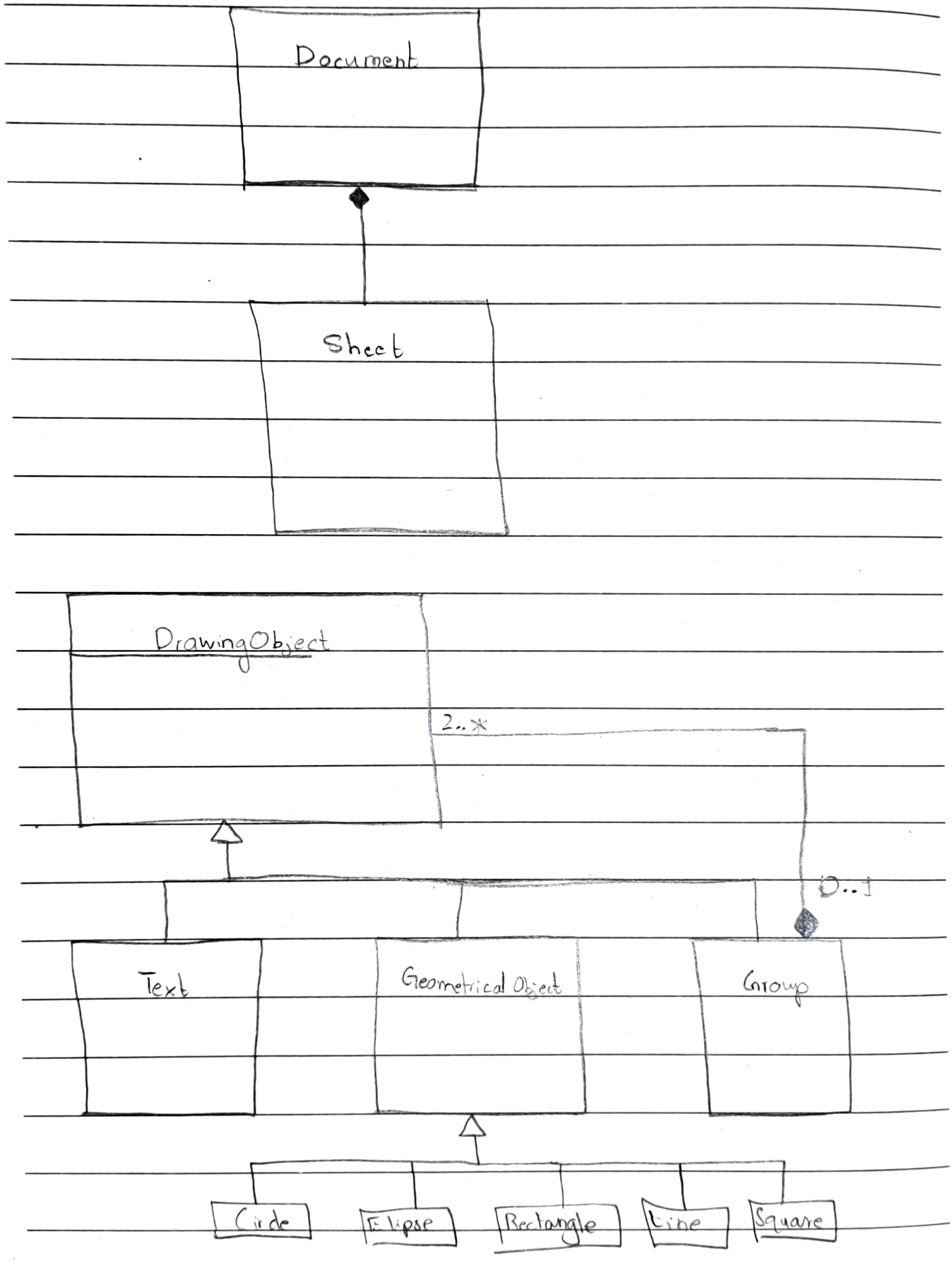
Proposal

set = 0..* = *



- Not all relations "go" in the class diagram. some are general information
- can also implies cannot.




Document
Sheet
DrawingObject
2..*
0..1
Text
Geometrical Object
Group
Circle
Ellipse
Rectangle
Line
Square

SDA L#07

Sequence Diagrams: interaction-based model, dynamic in nature.

- System-level
- Design.

objectName : ClassName → object created later on in time.

: ClassName1 → anonymous object

→ has an instance but doesn't have control. Gets control here

Transient object

: ClassName2

→ passive:

continuous lifeline: always active

deletion

datatypes and return types are in the class-diagram.

→ System-level (asynchronous messages)

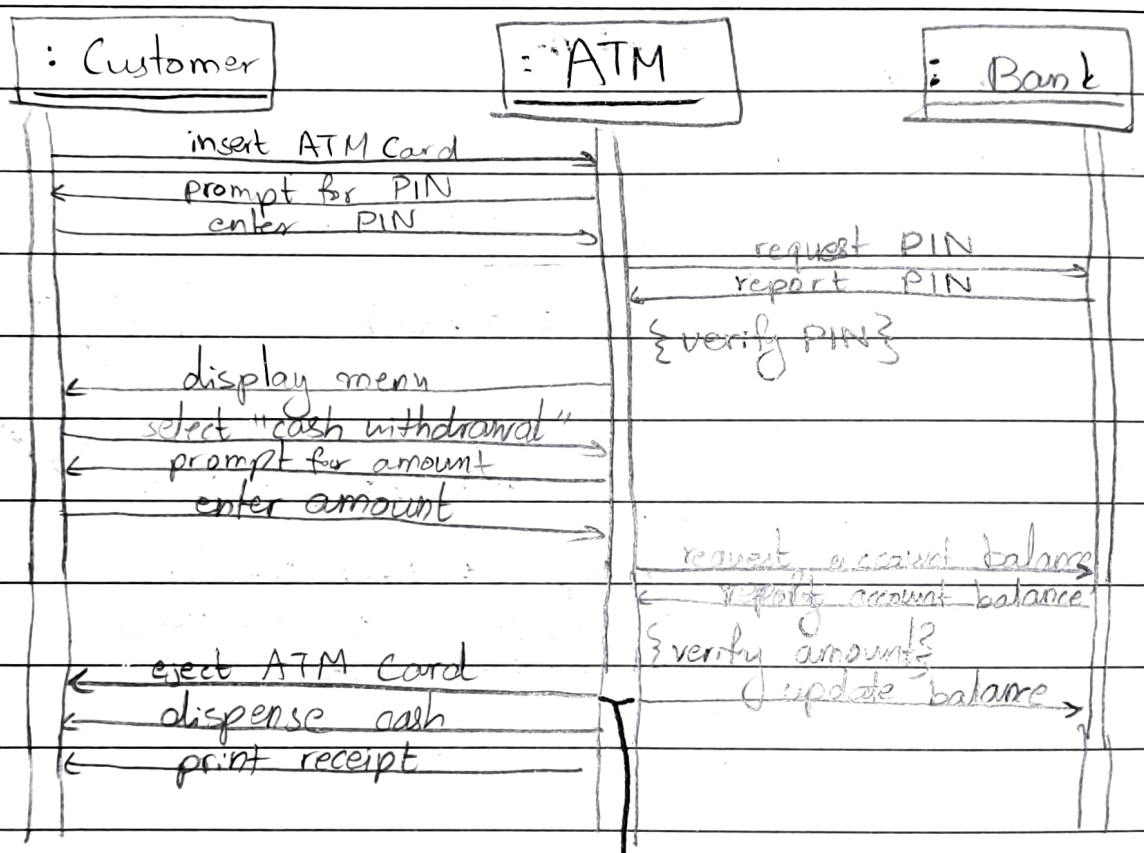
f1(p1, p2) → Design (synchronous procedure calls)

<-- return value (function returns)

↗ 1 of these for each scenario

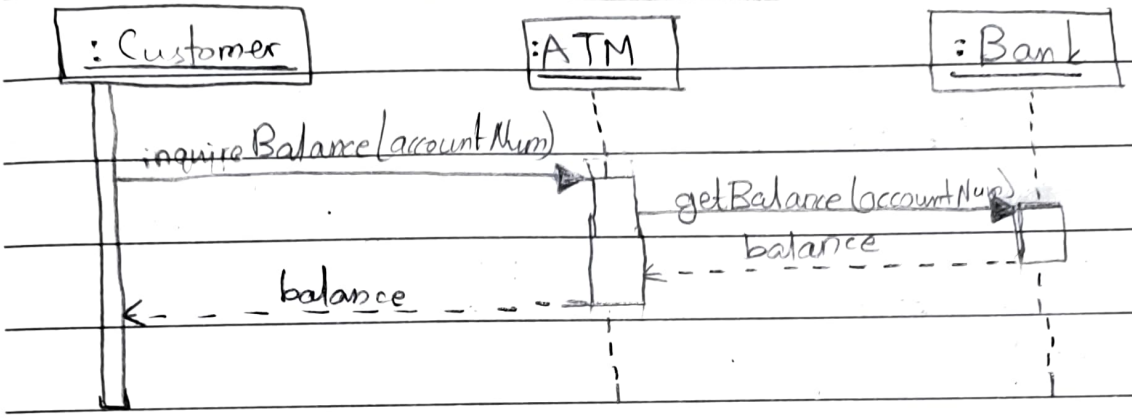
System_Level Sequence Diagram

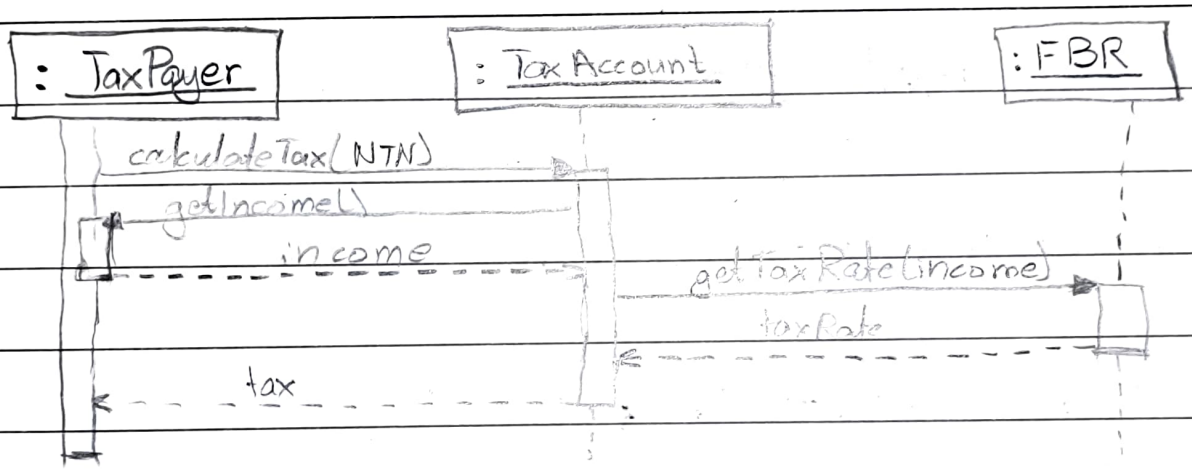
↳ Problem Phase.



↳ Concurrent messages originate at the same point in time.

Inquire Balance





SDA L#08 16-02-2024 SEQUENCE DIAGRAMS

Interaction Frames

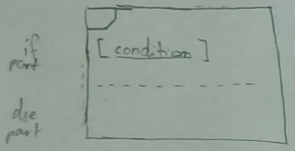
↳ refalt, loop

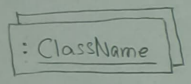
pentagonal tag

overlayed on the sequence diagram

ref: refer to another sequence diagram.

alt: &conditional or 'if' statements.



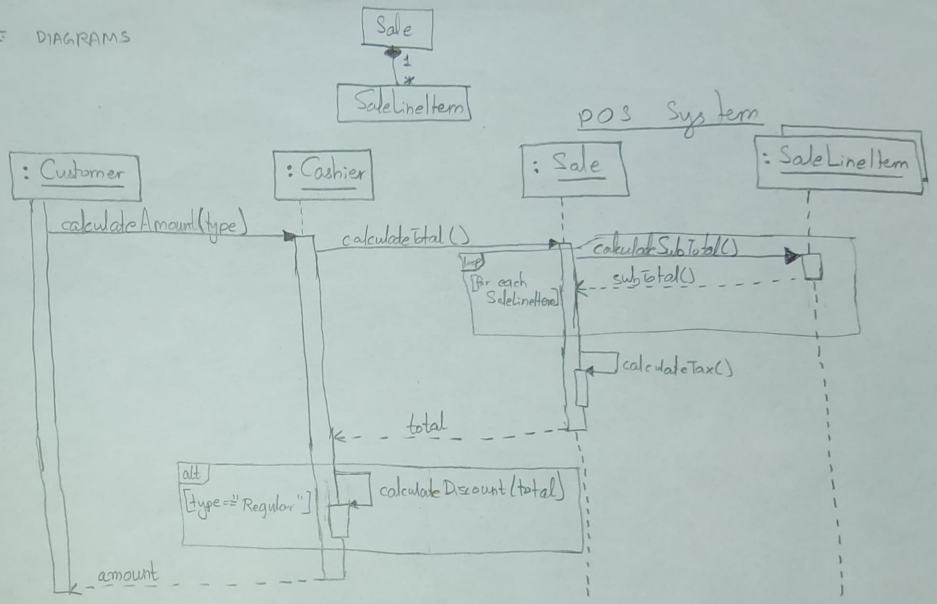


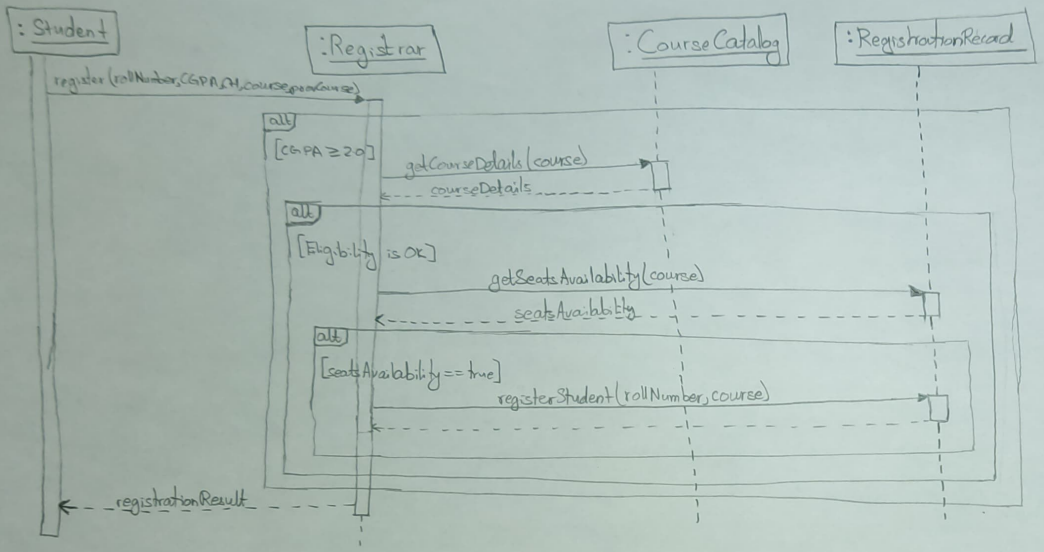
→ set, collection, array, list.

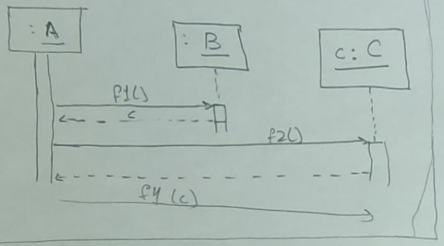
unlabelled arrow used to show void functions. Having no arrow is acceptable, but it is good to have an unlabelled line for readability.

- → UC-1, UC-X for use-case IDs.
- → replace all <content>
- → teamwork on ACD, UCD; divide use-cases among the team.

01 → UCD
02 → ACD
then UC-1, UC-2, ...




:Student
:Registrar
:CourseCatalog
:RegistrationRecord
register(rollNumber,CGPA,[],courseperCourse)
alt
[CGPA ≥ 20]
getCourseDetails(course)
courseDetails
alt
[Eligibility is OK]
getSeatsAvailability(course)
seatsAvailability
alt
[seatsAvailability == true]
registerStudent(rollNumber,course)
registrationResult

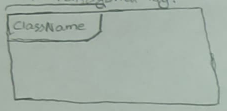

: A
: B
c: C
f1()
c
f2()
f4 (c)

STATE DIAGRAMS (state-machine or state-transition)

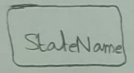
- Diagram for class with important temporal behavior
- Dynamic in nature
- Provide temporal information
- State is a snapshot of an object.
- Objects have states, not classes.

NOTATION

→ Pentagonal Tag.



Bounding Box

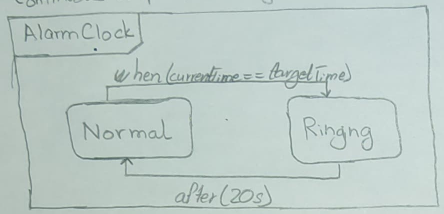


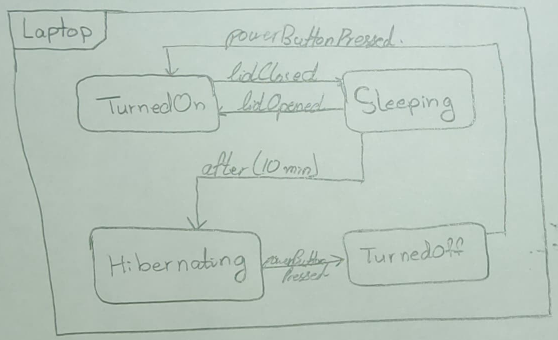
State: rounded rectangle

- State Names often end with -ing (printing, sleeping) or -ed (jammed).

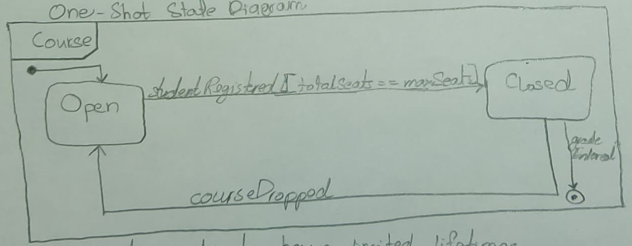
eventName → Rectilinear Grid, open arrowhead for transition.

Continuous Loop State Diagrams:





One-Shot State Diagram.



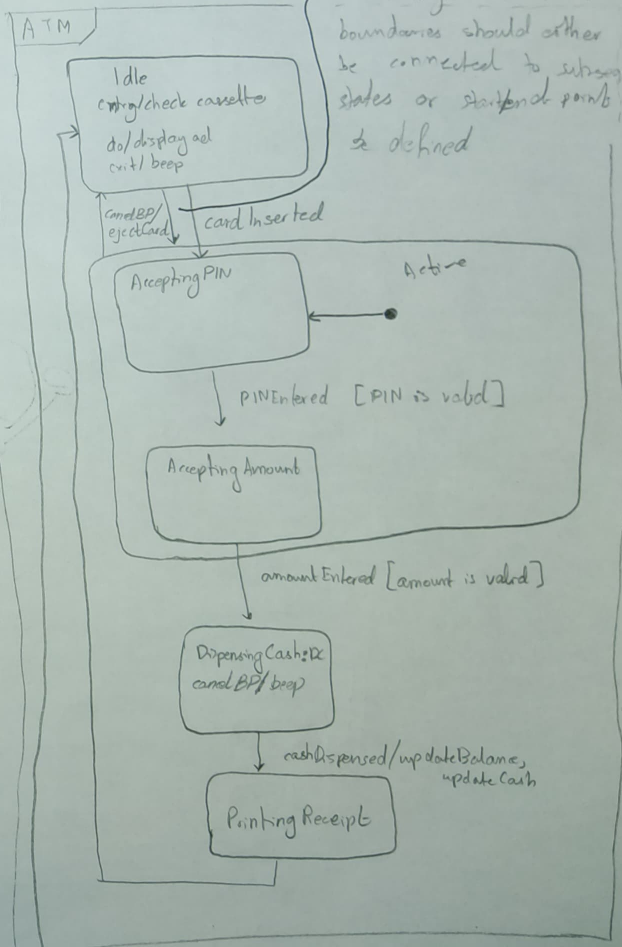
- Used when objects have limited lifetimes
- 'when' continuously checks for conditions, however the guard conditions [] checks only when the event occurs.

Types of Events:

- Signal Event: sending or receiving of information
- Change Event: satisfaction of a boolean condition.
  - ↳ when (...)
- Time Event: → Relative time event: after (...)
  - ↳ Absolute time events: when (...)

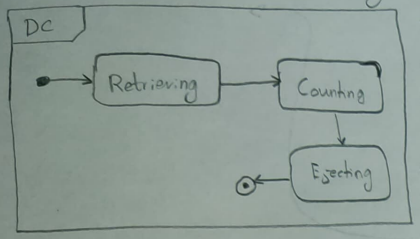
transition labels in italics.

eventName/action.



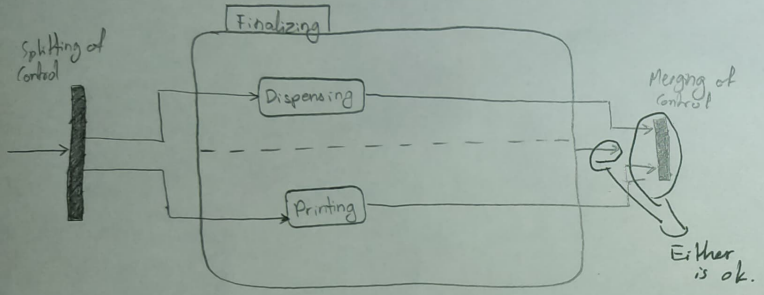
do something without exiting the state: activities within states.

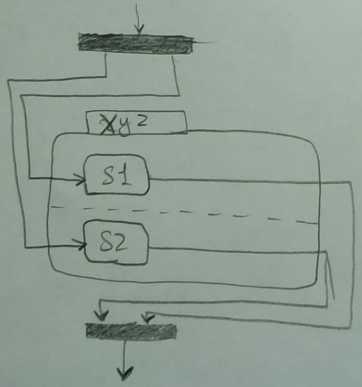
↝ Submachine for DispensingCash in ATM



This replaces the "DispensingCash : DC" ~~cash~~ state in the ATM diagram. These aren't drawn there because they are too low level.

unlabelled transitions
↳ completion transitions





→ If you get regular status updates you won't get surprises.

→ Air Force time: 5 minutes early.

Models

Structural
- ED
- OD
- CD

Behavioral
- UCD
- Activity
- State-transition
- Sequence

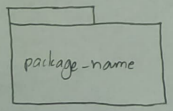
Models: Class, State, Interaction

-> Group similar model elements.

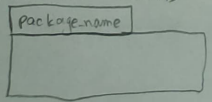
-> Things which are similar should be in the same package.

Model elements:
- classes
- use-cases.

Package:



↳ ~~may or may~~ does not have contents

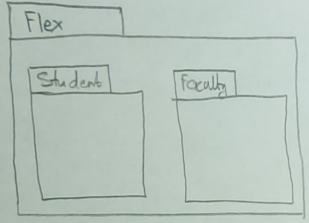


contents exist but are unknown

PACKAGE DIAGRAM

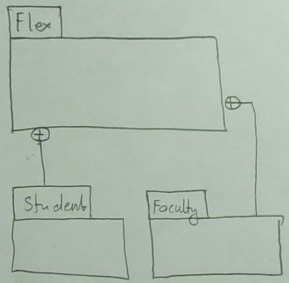
Grady Booch.

-> name of outer-most package should be the name of the system.

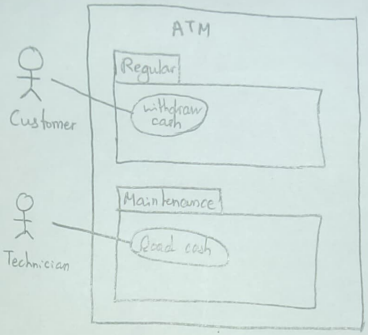


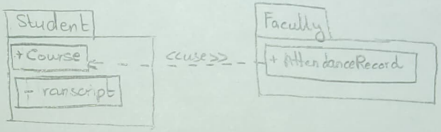
↳ nested or 'contained' packages

OR



- Rectilinear grid
- Singular nouns, upper camel case without spaces.





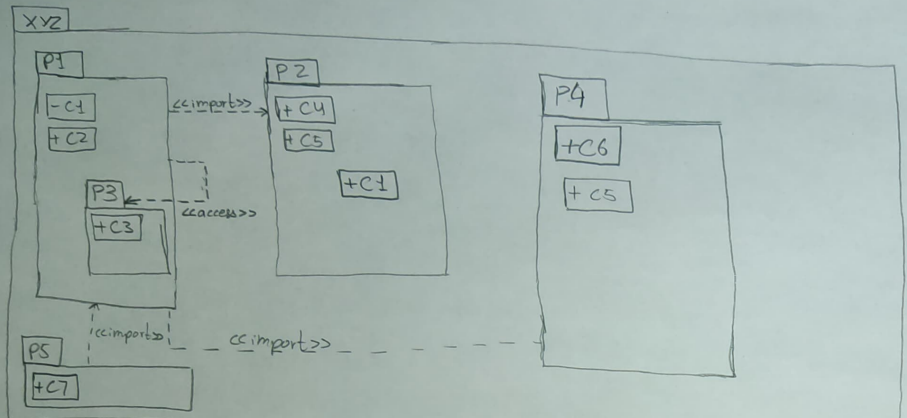
Class visibility in package diagrams

+ public: outside world may access.
- private: elements of containing package and the nested packages may access.

Dependency: this ----> depends on this

tail: client
head: server.

import: in next example, C4 and C5 will be imported into C1 and C2. No need to use fully qualified names. Private content does not get imported, other non-conflicting does.



C1, C2 access:

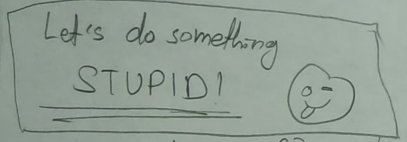
P3::C3

C4, C5 access:

P1::P3::C3

C7 access:

C2, C1, C4, C6 directly.
P3::C3.



The C1 in P2 wont be imported into P1 but will only be accessible using its fully qualified name.

<<access>>:

-> import is a public package import.

-> access is a private package import: P1 may access P3 but P5 ~~may~~ can not access P3.

Types of Cohesion & Coupling

Cohesion: Degree of Single-mindedness.

HIGHER-LEVELS OF COHESION:

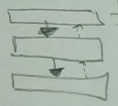
→ FUNCTIONAL COHESION:

Functions which do 'one' task and return control. Cohesive functions do one conceptual thing.

- a.k.a Perfect Cohesion.

→ LAYER COHESION:

Architectural style may be "layered" in which high-level layers access controls of low-level layers but not the other way.



→ COMMUNICATIONAL COHESION:

- All functions which access the same data are bunched into the same class.
- a.k.a informational cohesion.

LOWER LEVELS OF COHESION

→ PROCEDURAL COHESION

- Order-based cohesion
- Functions which are called in one sequence are ~~called~~ put in the same class.

→ SEQUENTIAL COHESION:

- functions pass data in a certain sequence. These are grouped into the same class.

→ TEMPORAL COHESION:

- Functions which are called at the same 'time' are put together into the same class.
- e.g startups, error-condition, etc. are 'times' which trigger certain set of functions.

→ COINCIDENTAL COHESION:

- not as bad as it sounds, but probably the weakest one.
- e.g. String class, Stat class, These contain functions which access different data to do different things, but they do the same 'sort' of thing and belong to the same 'heading'.

TYPES OF COUPLING

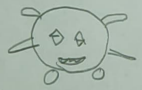
↳ "Degree of connectedness"

Coupling is a necessary evil: we can not eliminate it completely, but we may minimize it.

-> CONTENT COUPLING:

- The darkest side.
- e.g. make everything public so anyone can change anything.
- e.g. Friend functions and classes, also lead to content coupling.
- e.g. Inheritance: sub-classes may change protected attributes directly, violating the principle of data hiding.

-> COMMON COUPLING:



- Accessing common data.
- e.g. global variables.
- global constants may not be changed, so not as risky.

-> Instance variables in class
-> Local variables in function bodies.

-> ROUTINE CALL COUPLING

- functions calling other functions.

-> DATA COUPLING:

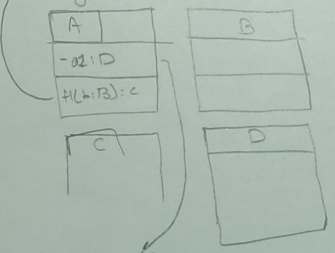
- functions passing data to other functions.
- all coupled functions should have the same interpretation of data.

-> CONTROL COUPLING

- functions calling other functions, passing data, and controlling the flow based on this data.

-> STAMP COUPLING:

- signature of function identifies it.



TYPE-USE COUPLING:

- use data type of one class in another.

-> IMPORT COUPLING:

- happens when packages are imported.

-> EXTERNAL COUPLING

- access OS, DB, networking components

SOLID Principles

S -> Single-responsibility Principle (SRP)
O -> Open-Closed Principle (OCP)
L -> Liskov Substitution Principle (LSP)
I -> Interface Segregation Principle (ISP)
D -> Dependency Inversion Principle (DIP)

-> Uncle Bob (Robert C. Martin) formulated these principles.
-> Michael Feathers created the acronym SOLID.

S: Each class/component should have a single responsibility.
-> There should be a single main reason to change a class.

O: Classes should be open for extension but closed to modification.
-> Existing code shouldn't be 'changed' too much. Instead, it should be extended.

L: Dr. Liskov received the Turing Award, which is the Nobel Prize equivalent of computing.
-> Sub-classes should be substitutable for the super-class.
-> Sub-classes shouldn't change the concept of the superclass. The specialization is another thing.
-> Sub-classes shouldn't change the semantic meaning of super-classes.
-> derived from the domain.

I: It is better to have multiple interfaces for separate clients instead of one big interface for all clients.
-> Changes for one client shouldn't affect other clients.

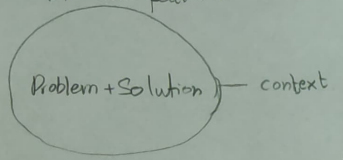
D: Classes should depend on abstractions, not on concretions.
-> Invert the dependency from concretions to abstractions.

DESIGN PATTERNS

- Book G:
  - The "Gang of Four" Book
  - Classic
  - ↳ Patterns Hatched
  - ↳ Erich Gamma's Ph.D. Dissertation
- Chunks of 'Design' can be reused across applications.

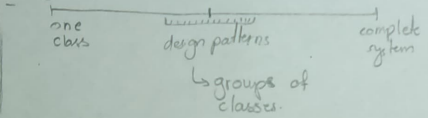
Pattern → repetition.
↳ kind of template, blueprint

- Implementations of patterns may vary; infinitely many instances may be generated for one pattern.



Scope \ Purpose	Creational	Structural	Behavioral
Class	Factory Method		Template Method (4)
Object	Singleton	Adapter, Composite	Observer (20)

(+24)

- All design patterns have 1 scope 'adaptor' being the only exception.
- 

↳ groups of classes.

> A design pattern is an invariant solution to a recurring problem in a certain context.

Template

- Pattern Name & Classification
  - Scope (2) — Class, Object
  - Purpose (3) — Creational (Instantiation), Structural (Composition), Behavioral (Interactions)
- Intent (summary)
- Also known As (a.k.a) (optional).
- Motivation (what motivates us to use this, from the real world)
- Applicability (when to apply this pattern)
- Structure (DCD)
- Participants (Participating Classes)
- Collaboration (DSD: how objects collaborate).
- Consequences (Pros & Cons)
  - ↳ Extra classes being added cause complexity in the design and affects performance.
- Implementation
- Sample Code
- Known Uses
  - ↳ these designs are not hypothetical.
- Related Patterns.

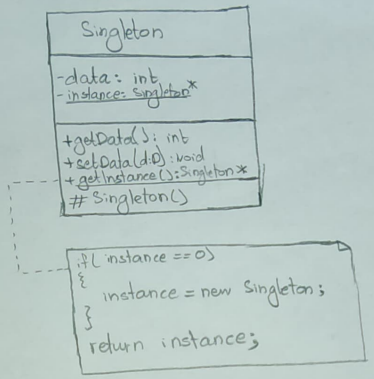
DESIGN PRINCIPLES: SINGLETON

↳ Object Creational

→ Restricting the no. of instances.
→ Make sure that a class has only one instance, and provide a global access point to it.

- Protect the constructor
- Design a function getInstance()

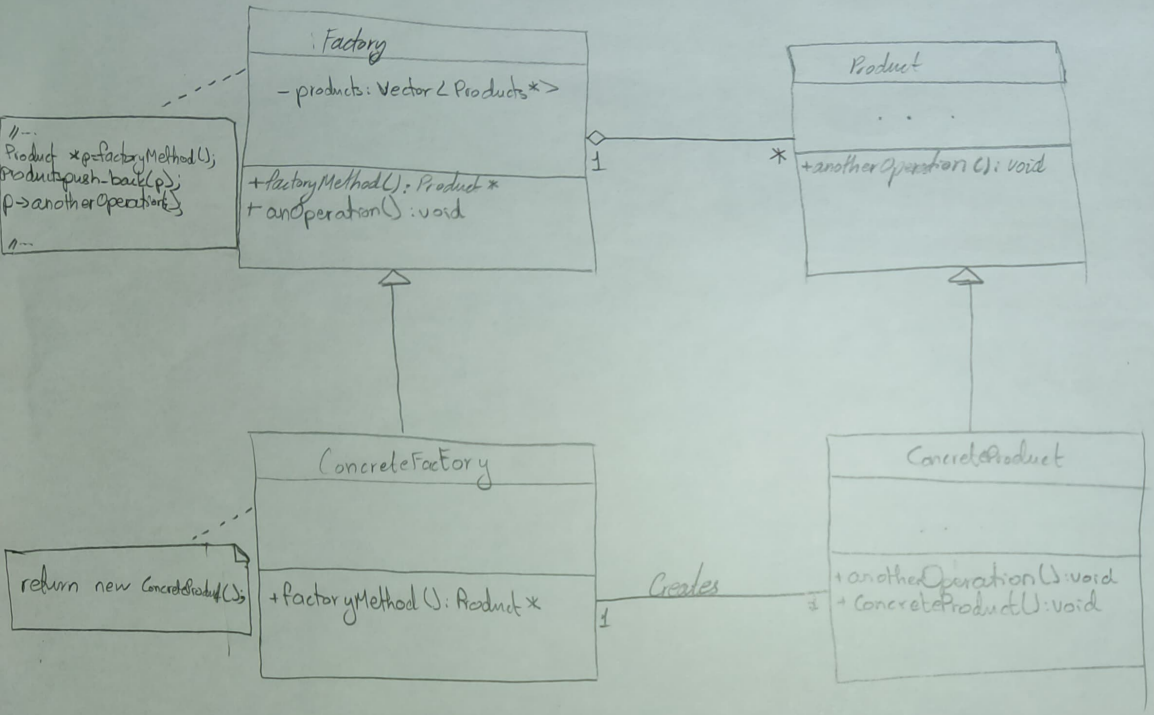
- Protected constructors also allow a cheap and dirty way to make abstract classes: classes which cannot be instantiated.




```
class Singleton
{
private:
    int data;
    static Singleton* instance;
public:
    int getData(){
    |   return this->data;
    }
    void setData(int data){
    |   this->data = data;
    }
static Singleton* getInstance(){
    |   if (this-> instance == 0)
    |   {   instance = new Singleton();
    |   }
    |   return instance;
    }
protected:
    Singleton() {}
};
```

FACTORY METHOD

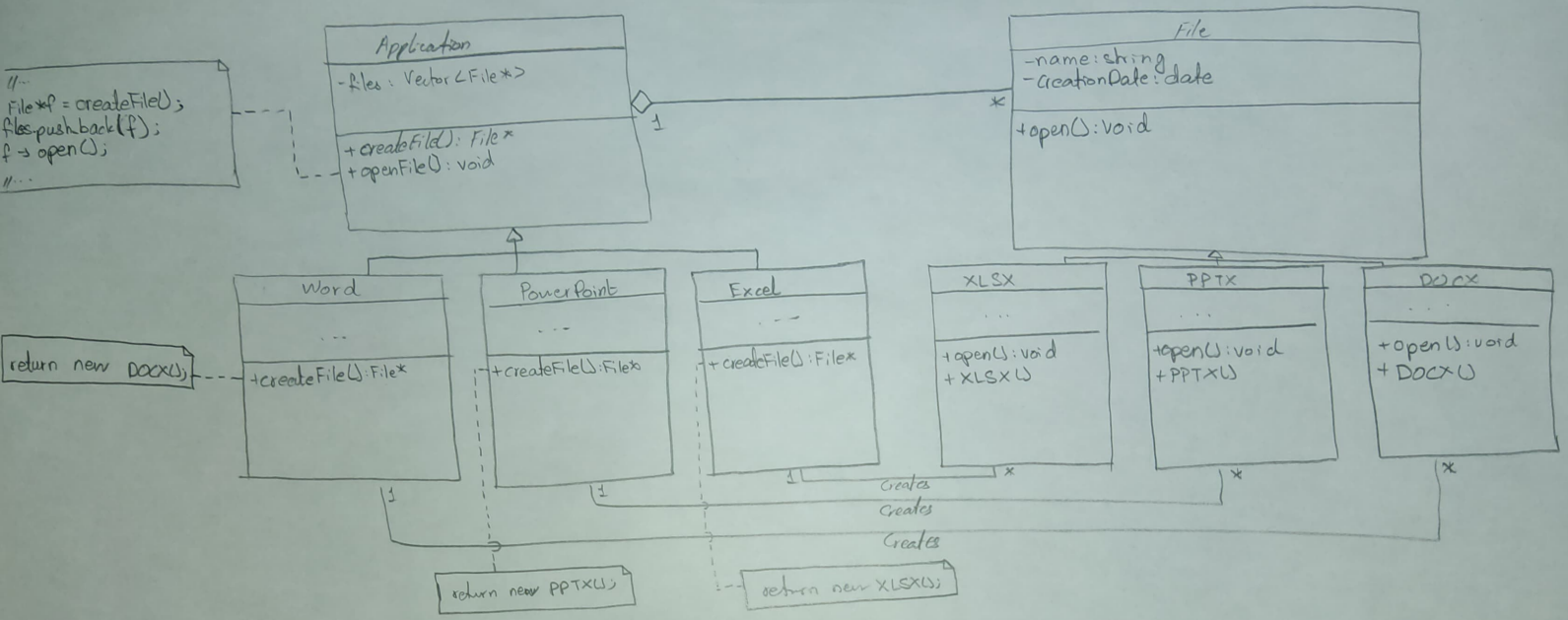
Class / Scope — Creational / Purpose



→ Define an interface for creating an object, but let subclasses decide which class to instantiate.
→ Defer instantiation to subclasses.
→ a.k.a Virtual Constructor.

Convention:
- Prefix factory method with words like create, make, doCreate, doMake

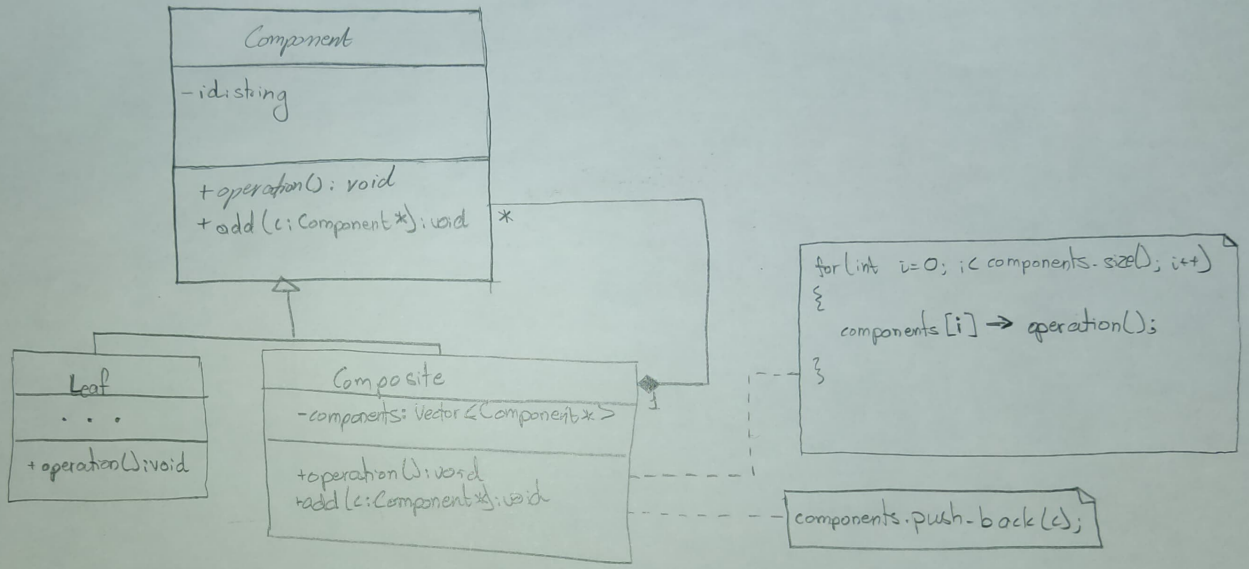
FACTORY METHOD EXAMPLE

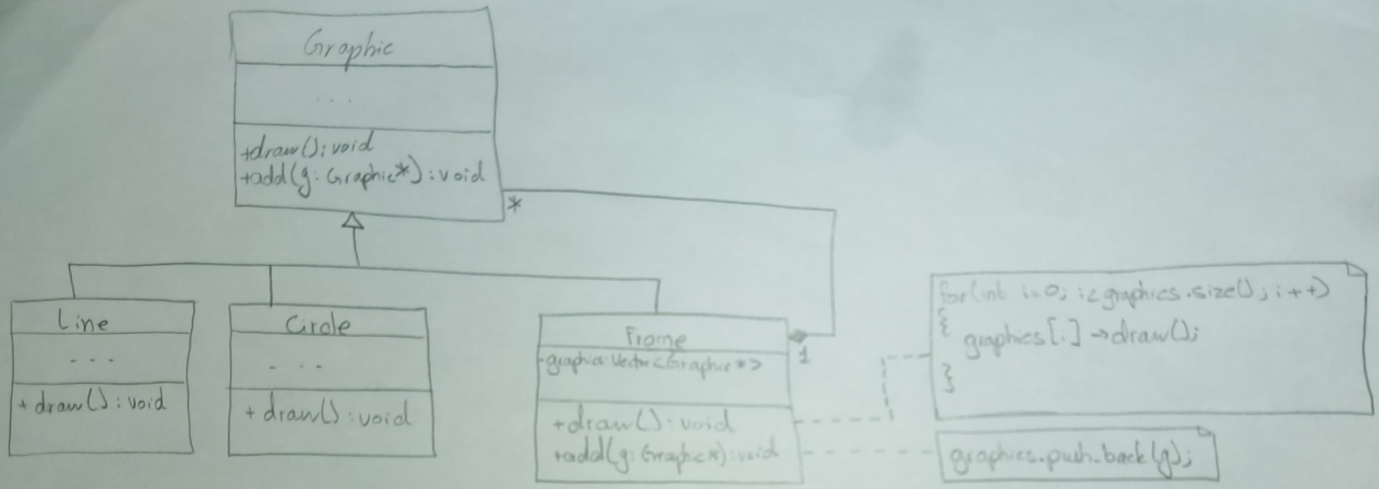


DESIGN PATTERNS: COMPOSITE

↳ Object Scope — Structural Purpose.

- → Part-whole hierarchies (containment)
- → Whole is composed of parts.
- → Arranged like a tree
- → Client should treat all nodes (leaf & non-leaf) in the same way.

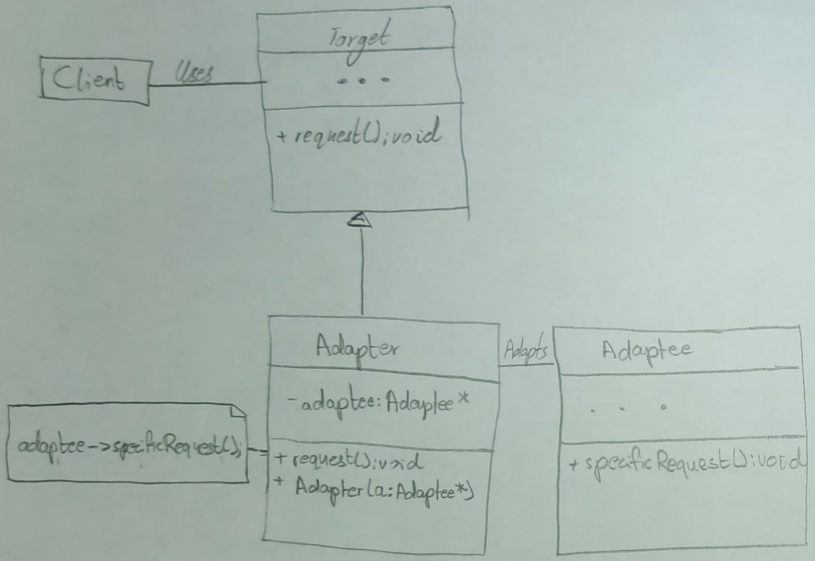



Graphic
...
+draw():void
+add(g:Graphic*):void
*
Line
...
+draw():void
Circle
...
+draw():void
Frame
-graphics:Vector<Graphic*>
+draw():void
+add(g:Graphic*):void
1
for(int i=0; i<graphics.size(); i++)
{
graphics[i]->draw();
}
graphics.push_back(g);

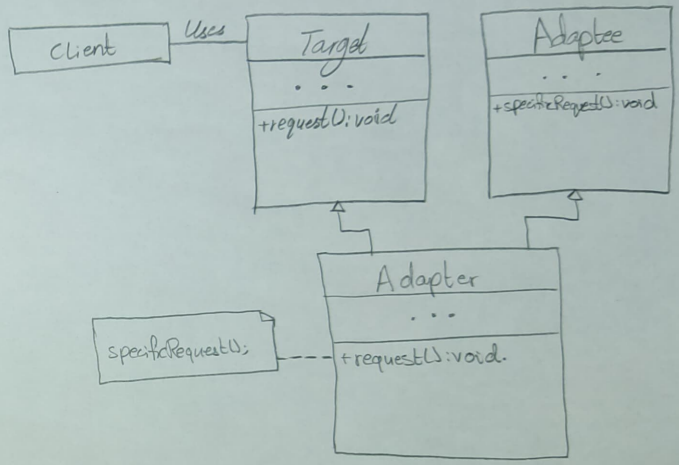
DESIGN PATTERNS: ADAPTER

Object/Class Structural

Object Variant



Class Variant



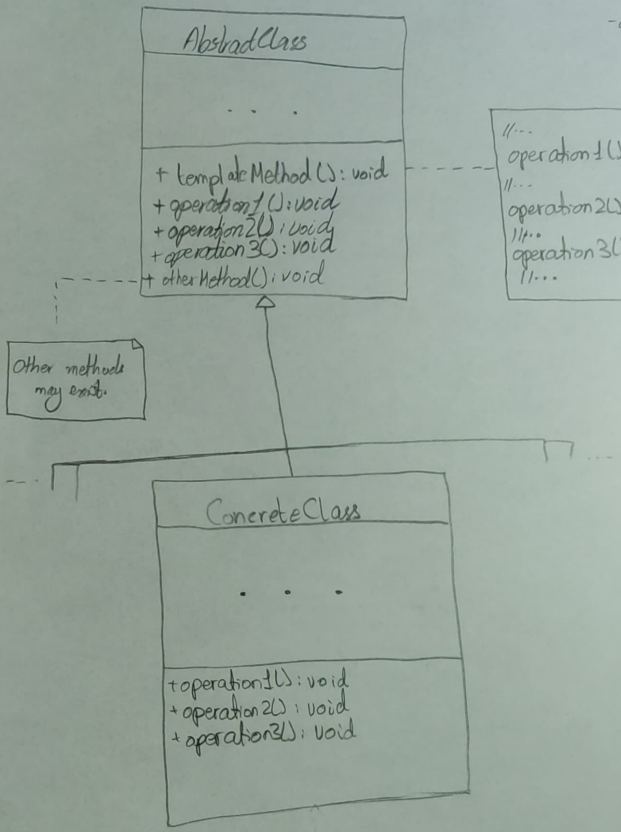
NOTES:

- Adapters enable communication between incompatible interfaces.
- a.k.a wrapper.

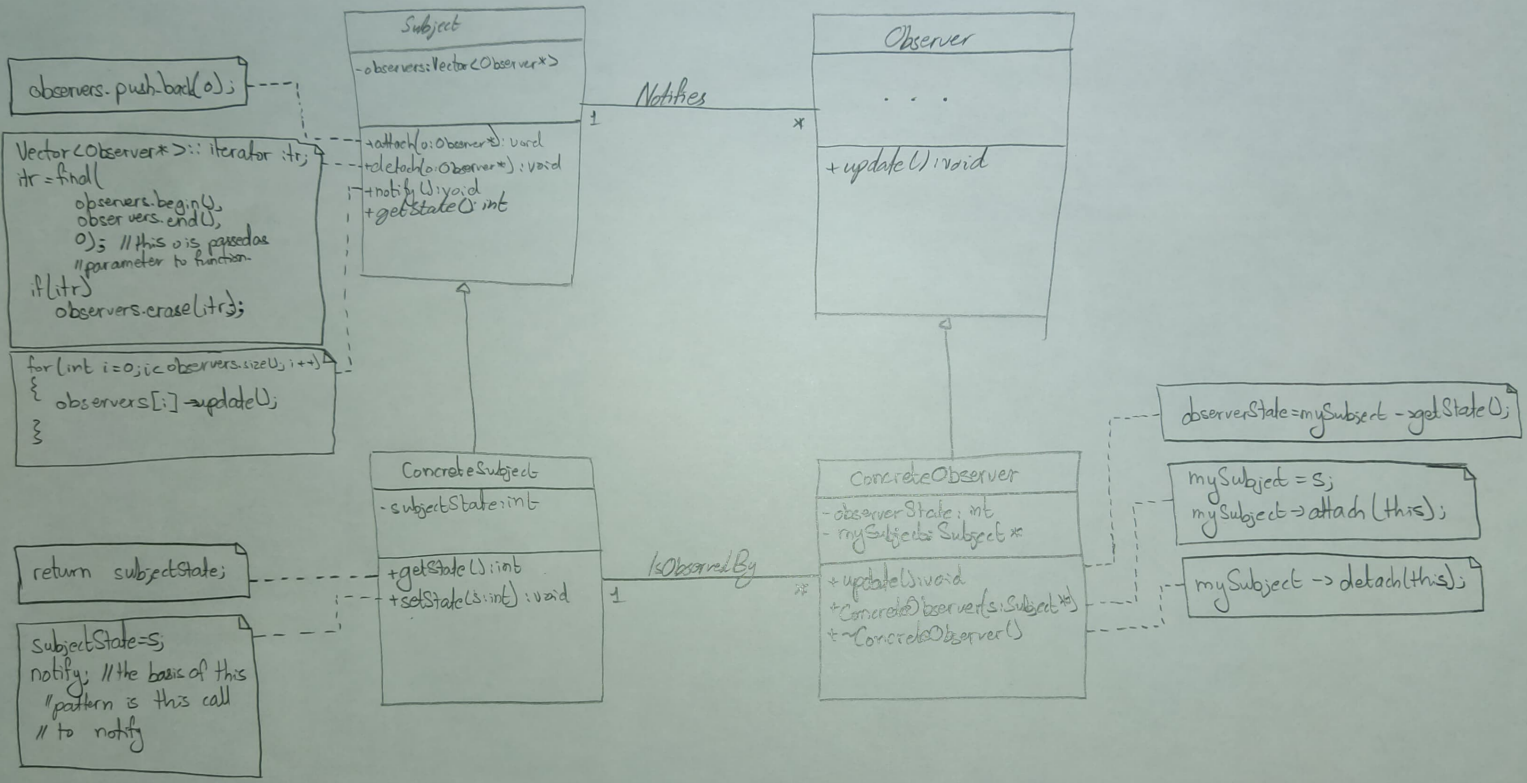
DESIGN PATTERNS: Template Method

Class Scope / Behavioral Purpose

- a.k.a skeleton method
- a.k.a hollywood principle.
  - directors tell auditioning actors: "don't call us, we'll call you."
  - usually subclasses call superclass methods. This is the other way around.



DESIGN PATTERNS: Observer



Date

Software Quality Metrics for Object-Oriented software system.

Defect Density

Cyclomatic complexity.

Suites of OO Metrics:

[LK] and [CK] (both introduced c. 1994).

Lorenz & Kidd Set (LK)

1- Number of Scenario Scripts (NSS)
- essentially use-cases - a measure of size

2- Number of Key Classes (NKC)
- the classes coming from the problem domain. See these in the ACD.
- measure of size.

3- Number of Supporting Classes (NSC)
- the classes added in DCD

4- Average Number of Supporting Class Per Key Class (ASC)
- $= \frac{NSC}{NKC}$

5- Number of Sub-Systems (NSuS)
- packages sort of work as sub-systems.
- high-level measure of size.

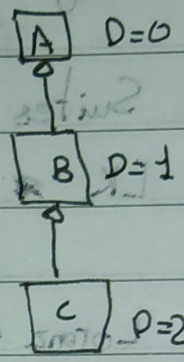
6- Class Size (CSize) = NO + NA
- NO: Number of Operations
- NA: Number of Attributes
- Count all operations, even if they are inherited.

↳ includes abstract methods implemented.

7- Number of Operations Added by a Subclass (NOA).

8- Number of operations Overridden by a Subclass (NOO)

9- Specialization Index (SI) = $\frac{NOO \times D}{NO}$



- D = depth of a class in an inheritance hierarchy.
- the more operations a subclass overrides, the more specialized it is. Hence the name SI.
- if A::a() is the only operation and B::b() is the only operation in A and B, NO will be:
  NO(A) = 1 ; NO(B) = 2.

Chidamber & Kemerer (CK) Suite of Metrics for OOSS.

1- Weighted Methods per Class (WMC)
- methods can be weighted based on complexity.
- $WMC = \sum_{i=1}^{n} c_i$ where n = no. of methods.
- assumption: $c_i = 1$. $\Rightarrow$ WMC = n. (Only for SDA)
- * similar to NO from CK suite.
- * does not count methods inherited.

2- Number of Children (NOC)
- no. of immediate children

Date

3- Depth of Inheritance Tree (DIT)

- same as D from CK suite.

4- Coupling Between Objects (CBO)

- no. of connections a class has.
- count no. of lines connecting a class to other classes, including associations, aggregations, compositions, and inheritance.
  - tree-type don't count as 1.
  - n-ary counted separately, not once
  - count direct lines, not inherited associations.

5- Response for a Class (RFC)

- no. of operations of a class + no. of remote operations.
- transitive closure also possible.

6- Lack of Cohesion of Methods (LCOM).

Never do double counting. For all metrics. General Advice!!!

$I = \{i_1, i_2, \cdots, i_j\}$ (Set of instance variables)

$M = \{m_1, m_2, \cdots, m_n\}$ (Set of methods)

- include inherited functions in M only if they are overridden.

$m_1 \xrightarrow{\text{accesses}} I_1$

$m_2 \rightarrow I_2$

$\vdots$

$m_n \rightarrow I_n$

I_s are sets of instance variables accessed by ms.

Pairwise Comparison: (I_1, I_2)

Total Pairs = $\frac{n(n-1)}{2}$

$P = \{(I_r, I_s) \mid I_r \cap I_s = \emptyset\}$ (bad)

$Q = \{(I_r, I_s) \mid I_r \cap I_s \neq \emptyset\}$ (good)

$LCOM = \begin{cases} |P| - |Q| & ; |P| > |Q| \\ 0 & \text{otherwise} \end{cases}$

$= |P| + |Q|$

Date

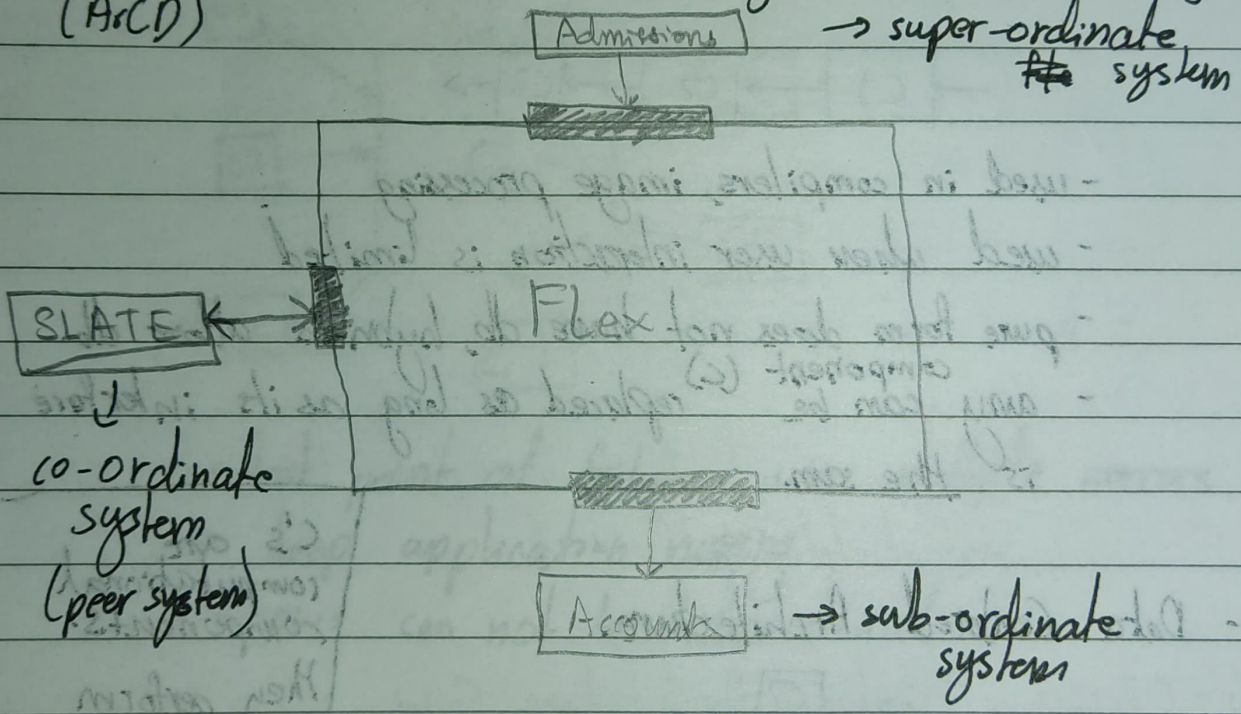
SDA: Software Architecture.

High-Level Design = Architecture.
↳ Bird's eye view

Software Systems Solutions Data } Architecture

Non-Functional Requirements = Architecturally Significant Requirements
NFRs = ASRs

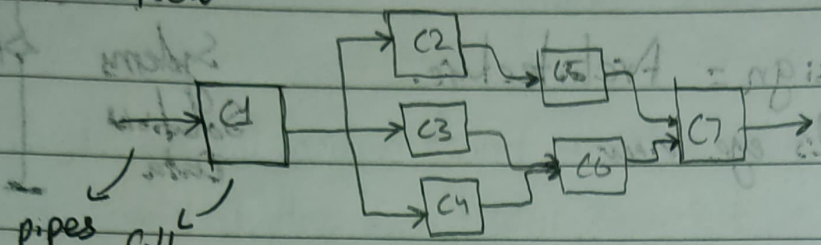
Architecture Context Diagram. (not a UML diagram)
(ArCD)



- After the ArCD is made, architecture is developed based on well-known architectural style.

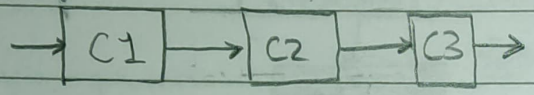
Common Architectural Styles:

1- Data Flow



Pipes and Filters

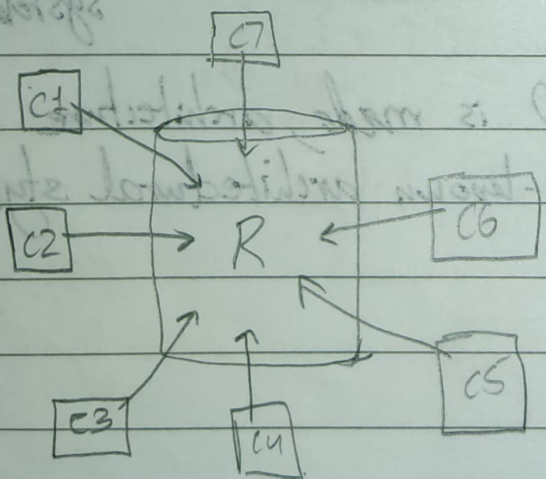
- Data flows in one direction only.
- A special case is Batch Sequential



- used in compilers, image processing
- used when user interaction is limited
- pure form does not have db, hybrids allow db.
- any component (C) can be replaced as long as its interface is the same.

C's are computational components: They perform computations. R is not a computational component

2- Data Centered Architecture.

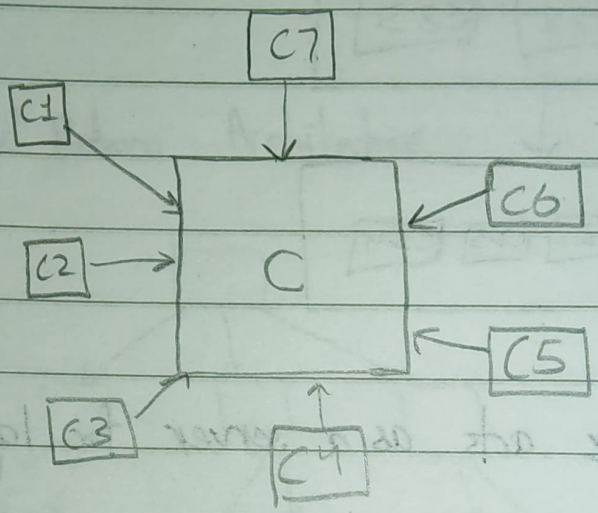


- arrows may be bidirectional:

Date

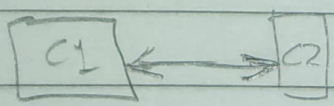
- All the C's are independent: addition, removal, or change of any C does not effect others.
- Central point of failure.

3- Client/Server Architecture:

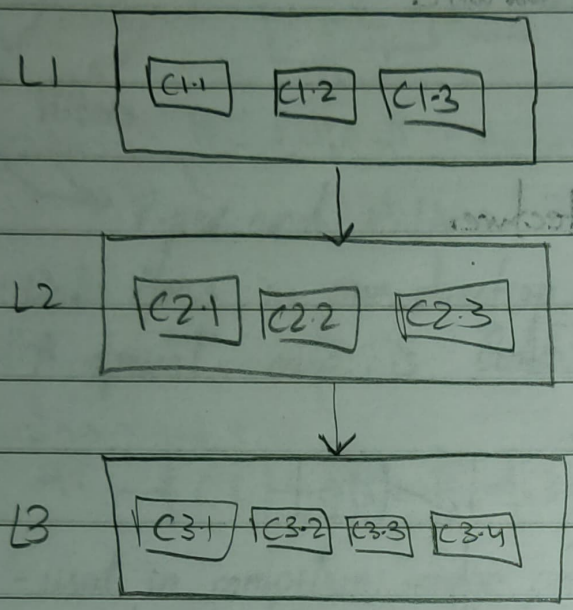


- central point of failure: use multiple servers to keep application running.
- arrows can not be bidirectional.

4- Peer-to-Peer Architecture

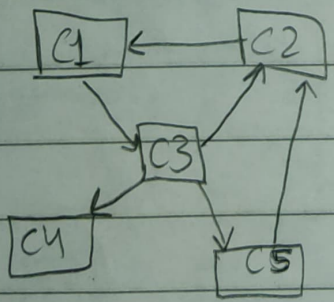


5- Layered / Tiered (n-Tiered)



- Layer x acts as a server for layer (x-1)
- A layer can only access the layer immediately below it.

6- Object-Oriented Architecture Style:



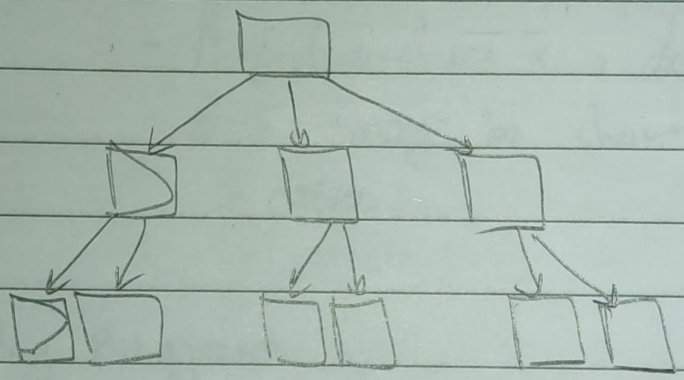
Date

SAD (Software Architecture Document)

1- ArCD
2- Box and Arrow diagram of choosen architecture
3- Description (Text).

Global Software Development (GSD)

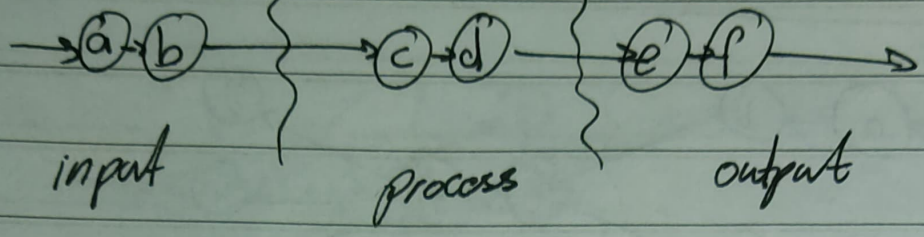
7- Call & Return Architecture

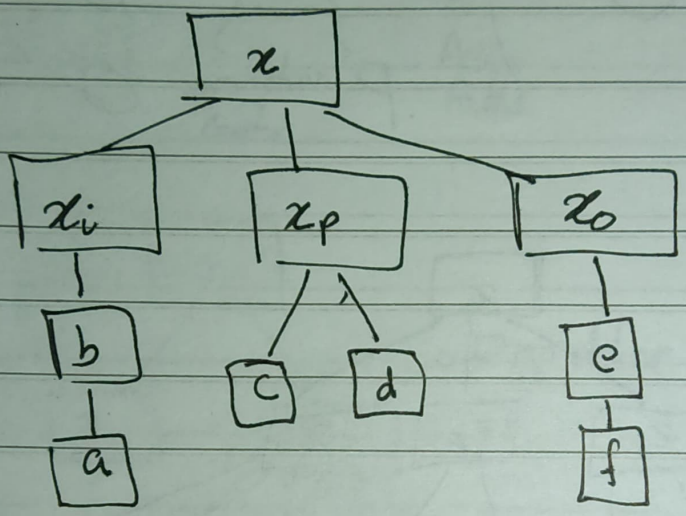


-not all C's at the same level of abstraction.

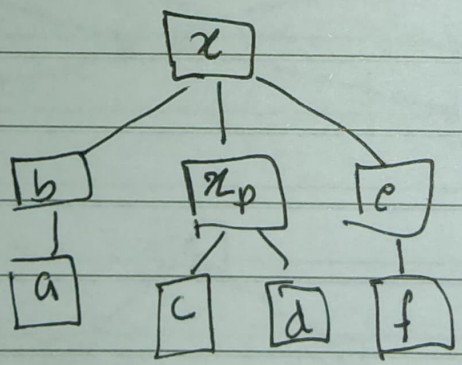
Date

Transform Flow

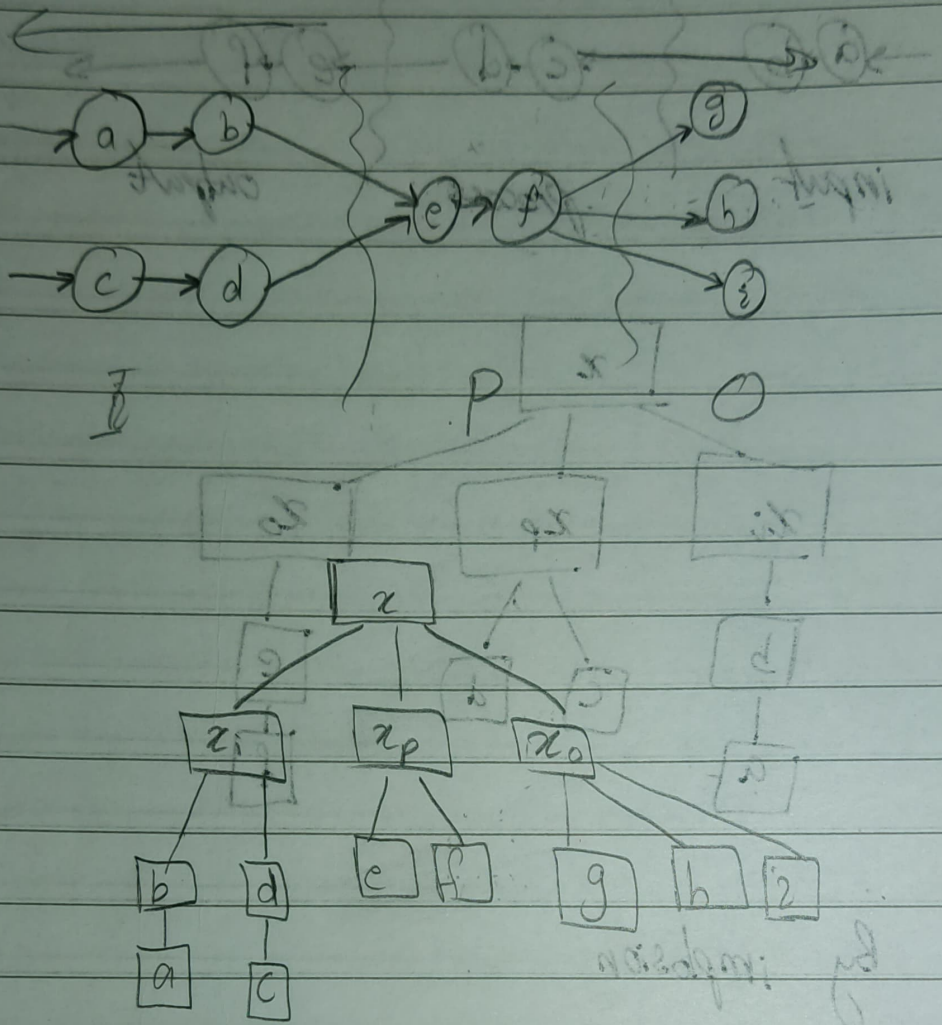




By implosion

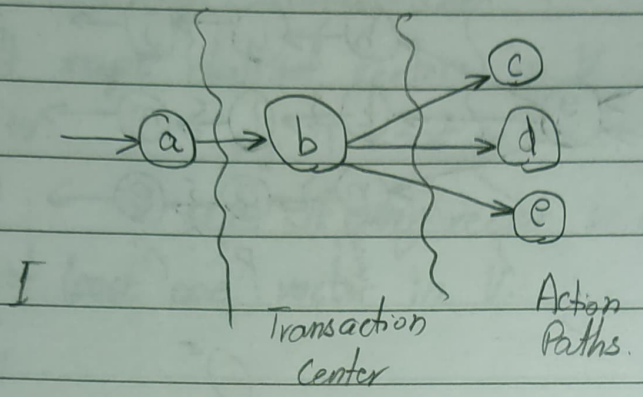


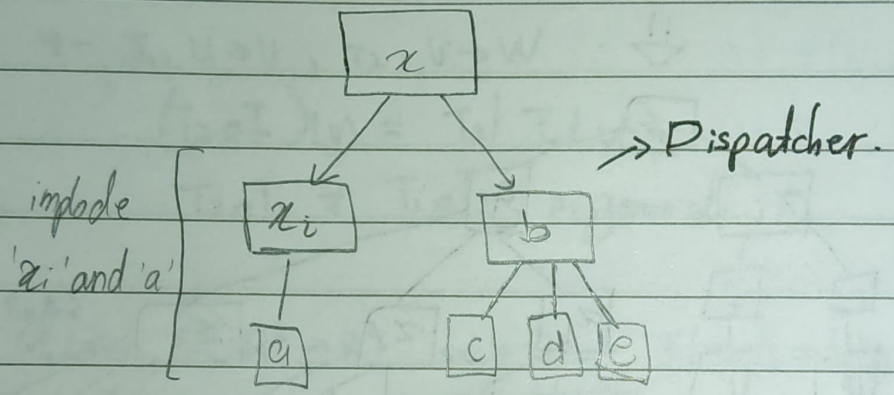
rule is to proceed outwards.

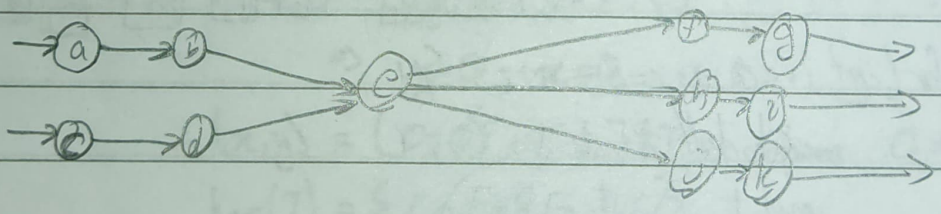


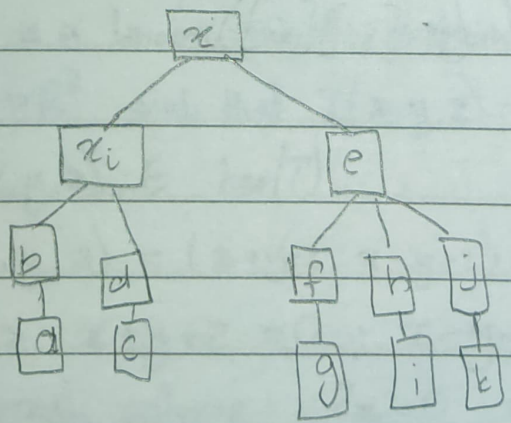
Date

Transaction flow

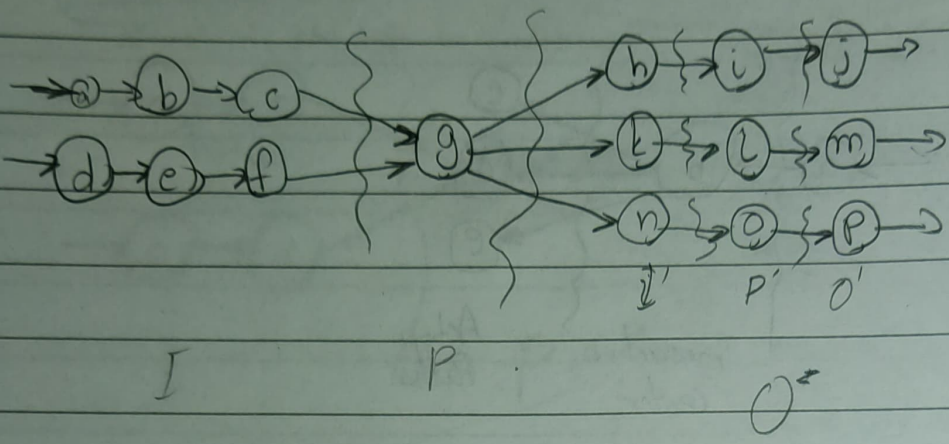


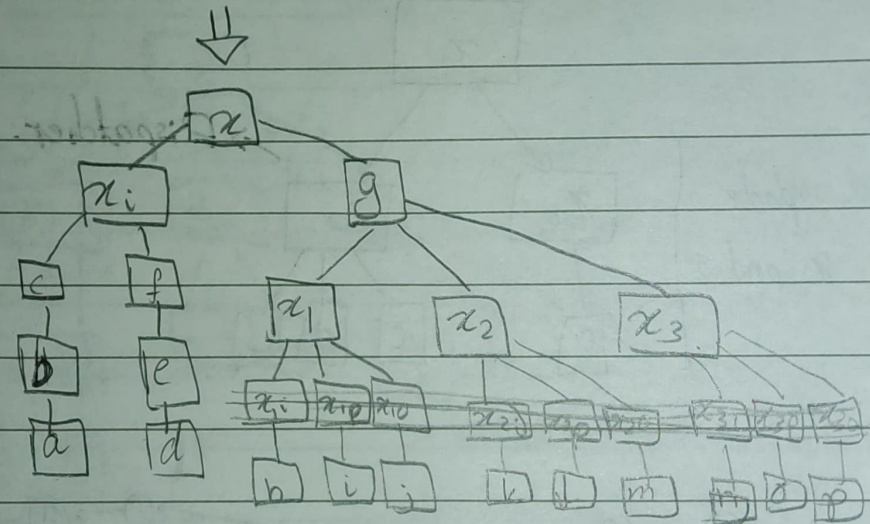






Supermassive Transform + Transaction Flow.






```
for (int ii=0; ii<=3; ii++) {
    implode(x[ii][2]);
    implode(x[ii][P]);
    implode(x[ii][O]);
}
```