SEG2105 A | Chapter 4

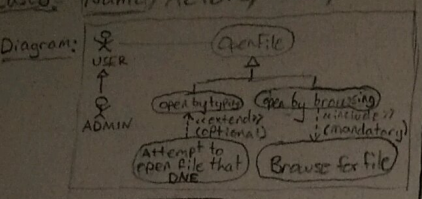
Requirement: Define system, Verb (shall/~~may~~/Must), End result, Measurable quality criteria [Must explain what it does not how it does it]

- **Functional:** what system does (I/O, computations, timing, stored data, etc.)
- **Non-Functional:** Constraints on → Quality (RUMER), Platform, Process
  - ↳ show system does it)

Use Cases: Name, Actors, Goals, Preconditions, Summary, Related use cases, Steps (2-column format: 1. User ... 2. System ...), Postconditions

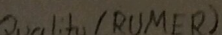
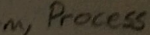
1. User → System
1. User clicks search
2. System shows search form
3. User → System

↳ Diagram:

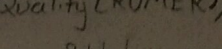


Gathering Requirements: Observation & videotaping, Interviewing, Brainstorming, Prototyping, Use case analysis (see chapter 7 for more)

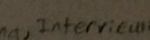
Chapter 5 | Attributes = Instance Variables, Operations = methods

Association:  (avoid 1—1 multiplicity) w/ directionality: 

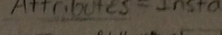
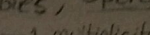
↳ use AC

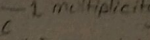
Reflexive Association: 

↳ is Mutually Inclusive With ← Lower bounds must be 0

Association Class: when attribute needs to be in both classes: 

Generalization: ↑ (IsA), points to superclass, **Propagation:** Occurs when aggregate performs operation on its parts (must be manually implemented) ↳ eg. delete 1 → delete other

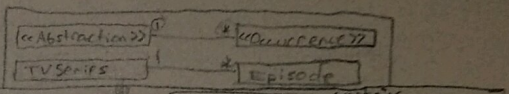
Aggregation: Represents part-whole rela :  , **Composition:** Aggregation where whole destroyed → part destroyed: 

System Model: Models everything from domain of system to UI (can generate code). **System Domain Model:** Models domain only, used to model code. 

Exploratory Domain Model: Models parts of domain to learn about domain, no code generated from this. Interfaces & Abstract Class: <<Interface>> or <<Abstract>>

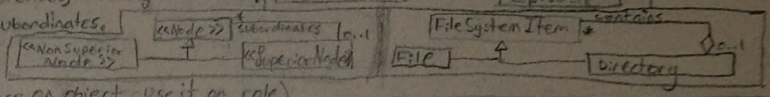
Chapter 6 | Patterns

Abstraction-Occurrence: given a set of related objects (occurrences) sharing common info.

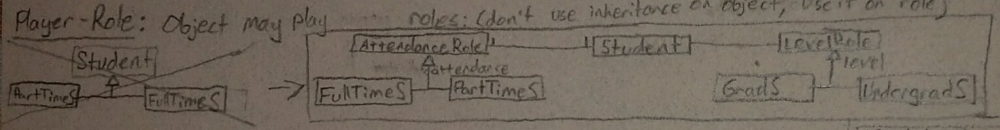


General Hierarchy: Hierarchy where some objects cannot have subordinates.

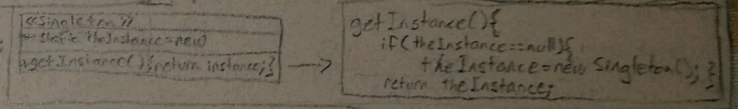
↳ Only use if you know ∃ a fixed # of subclasses.



Player-Role: Object may play ... roles: (don't use inheritance on object, use it on role)



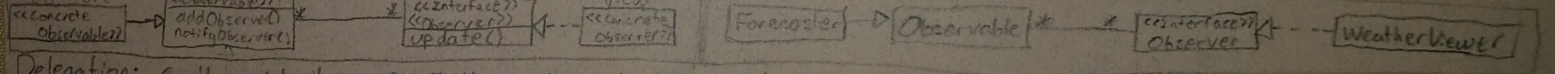
Singleton: Want only one instance of class (eg. database). Have private constructor:



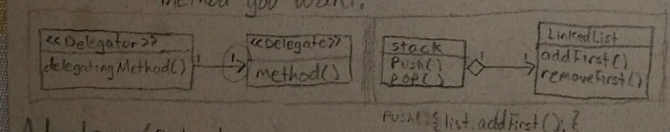
get Singleton: Singleton.getInstance();

Thread-Safe (better)

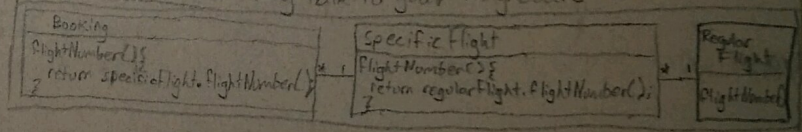
Observer: (AKA Listener) Want to reduce associations s.t. you can reuse one class w/o reusing other. (eg. buttons & actionListener)



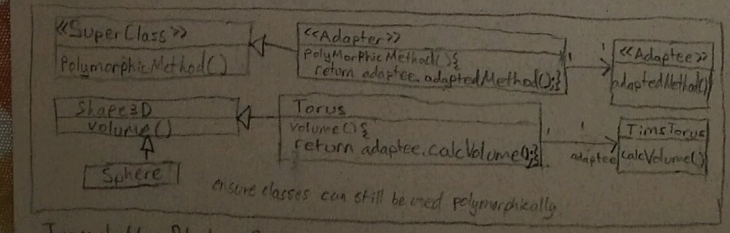
Delegation: Can't use inheritance (no IsA), but other class has method you want.



↳ Law of Demeter: Only talk to your neighbours

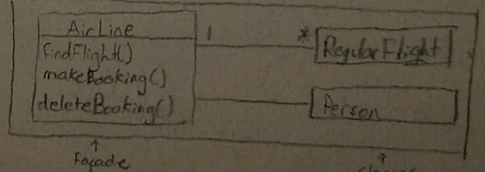


Adapter: (Delegation + Inheritance)



Façade: Simplify a programmer's view of complex system (packages)

↳ Façade is often a singleton

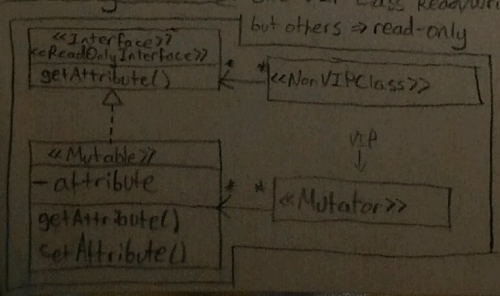


façade is kind of a "controller"

Immutable: State of object does not change after creation → Constructor: only place where instance variables are set/modified.

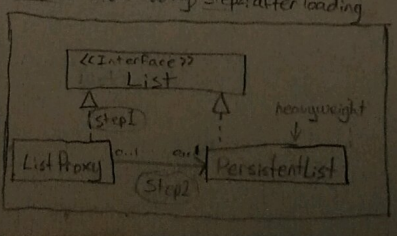
→ If method would modify inst.var, it returns new instance of class instead.

Read-Only Interface: Give VIP class Read/Write, but others ⇒ read-only



Proxy: Have to load heavyweight class

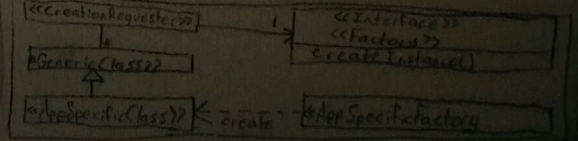
step1: before loading, step2: after loading



Factory: Framework wants to create objects, but class of object depends on application

→ Factory is generic interface defined in framework

→ Interface declares method whose purpose is to create subclass of some generic class



Chapter 7

IF don't understand, suggest answers, highlight keywords

Usability: •**Learnability:** Speed at which new user becomes proficient; **Efficiency of use:** speed @ which expert user does work;
•**Error Handling:** Extent to which system prevents/detects/corrects errors; •**Acceptability:** Extent to which users like the system

User-centered Design: Have users give feedback during design and creation steps.

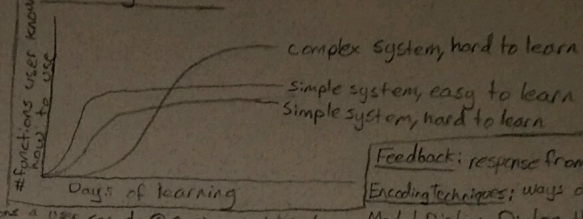
Utility: Everything the system *can* do. **Usability:** How the system allows users to learn and use raw capabilities easily.

Heuristic Evaluation: (users not involved) Pick use cases to evaluate, ∀ window, page, or dialog appearing in execution of use case, look for usability defects. When defect is found, write down short description & ideas to fix.

Observation/Video-Tape: find users (actors), ask users to complete a task (important use case), (may talk w/ users), note difficulties, recommend changes.

Dialog: popup Yes/No **Control/Widget:** UI component **Affordance:** All operations a user can do @ a given point in time

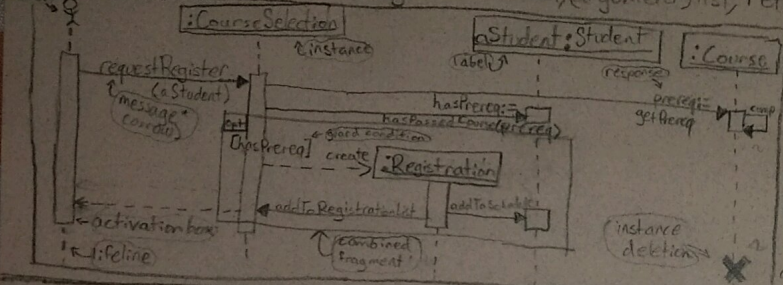
Learning Curves:



Feedback: response from system when user does something.
Encoding Techniques: ways of displaying info to user.
Modal Dialog: Dialog that blocks rest of system.

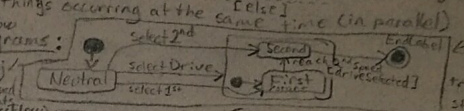
Chapter 8 | Interaction Diagrams: Sequence & Communication ~ diagrams, Sequence > Communication (< has more details)

Sequence Diagrams: *message has a label, (argument) list, return value (response) → Syntax: response := message(arg, ...)

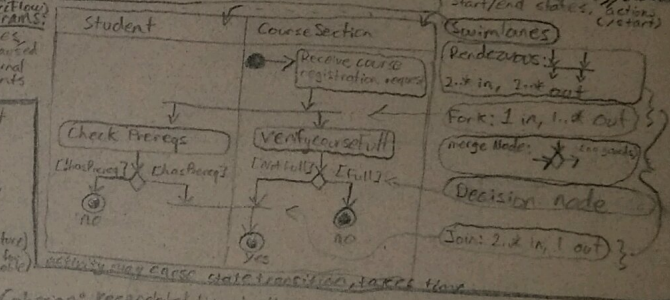


Possible combined Fragments: opt[ ] = if(condition){} (may not be executed) | loop min..max = for(i:=min; i≤max; i++){} | alt [condition] = if(cond.){}else{} (one of many will be executed) [else] | par = 2 things occurring at the same time (in parallel)

State Diagrams: (flow of control) State of 1 obj; transitions caused by external events (signals) (composite) has: nested substates, guard conditions, transitions (events, actions), start/end states. (restart)



Activity Diagrams: (flow) many classes, transitions can be triggered usually by internal events



Chapter 9 | Design Space: space of all possible designs that could be achieved by choosing different sets of alternatives.

System: (soft+hard ware) logical entity with a set of definable responsibilities/objectives, can have subsystems.
Component: (soft+hard) has a clear role, can be isolated & replaced w/ equivalent, reusable, or specific function.
Module: Component defined @ programming language level. eg: for Java: Class, method, package
Top-down design: Start w/ high-level structure, work down to detailed decisions (give system good structure)
Bottom-up design: Start w/ decisions about reusable low-level utilities, then decide how to put together (good for reuse)

Design Principles (DP): **DP1: Divide & Conquer:** Separate work in small parts.

DP2: Increase Cohesion: •Encapsulate things together, other things away. 7 types (most to least important)
•Functional: Code computing particular result kept together → reuse, easier to understand/replace
•Layer: all facilities to provide access set of related services kept together, high layers access lower, lower layers don't access higher, to replace layer, just deal w/ API.
•Communicational (OO): all data-storing/modification in same class.
•Sequential: procedures providing inputs to the next are kept together.
•Procedural: procedures occuring one after the other are kept together.
•Temporal: operations performed in same phase of execution are kept together. eg: initialization
•Utility: related utilities which cannot be placed in other cohesive units are together

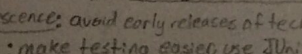
DP3: Reduce Coupling: reduce interdependencies between modules, 10 types (least to most desirable)
•Content: one component modifies public inst.var. in other class → use setters/getters. (private)
•Common: global variables (eg: singleton) → all components using global variable become coupled.
•Control: using "flag" that explicitly controls second procedure → use polymorphic operation or LDI.
•Stamp: when a class is the type of a method argument → use interfaces or simple variables
•Data: when method has too many arguments → don't have unnecessary args
•Routine-Call: when 1 method calls another → reduce by creating a method that encapsulates the sequence
•Type Use: Inst.Variables & local var. w/ other class as type → use interfaces & general types
•Inclusion/Import: Importing packages → don't do "java.swing.*", get specific classes.
•External: Rely on Framework/library/OS/hardware → use façades

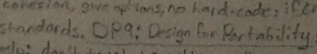
DP4: Increase Abstraction: Use classes/interface for info hiding ⇒ simpler code
DP5: Increase Flexibility: generalize, simplify, have hooks

DP6: Reuse (but don't clone) **DP7: Design for Flexibility:** Prepare for design changes → coupling/cohesion, give options, no hard-code: if(cmax≥25) ← 25 should come from property file

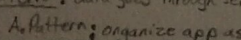
DP8: Anticipate Obsolescence: avoid early releases of tech, avoid environment-specific library, use standards. **DP9: Design for Portability:** Have soft. run on many platforms

DP10: Design Testability: make testing easier, use JUnit, test-driven dev. **DP11: Design Defensively:** don't trust how others will use your components. → validate front + back end → Design by Contract

Component Diagrams: 

<physical **Deployment Diagrams:** 

<more general than component diagrams

Multi-Layer architectural Pattern: only communicate with layer directly below eg TCP/IP. **Broker A. Pattern:** Distribute aspects of soft. system to different nodes.
Pipe & Filter A.Pattern: data goes through series of processes eg: 

MVC: Separate UI from rest ← this is an A. Pattern.
Service-Oriented A.Pattern: organize app as a collection of services communicating through well-defined interfaces eg. Web services
Message-Oriented AP: sub-systems communicate only by exchanging messages

Chapter 10 | Failure: unacceptable behaviour (requirement violation) > Defect: flaw in any aspect of system (requirements, design, code) (fault) > Error: slip-up by soft eng. (fault)

Structural/Glass/White-box test: have access to code → Flow Graph: (activity diagram) Cover all paths (infeasible), all edges (efficient), all nodes (simpler) "decision" nodes

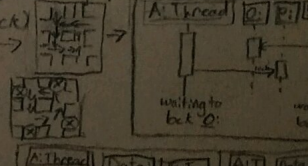
Black-box testing: No access to code, provide inputs & observe outputs. → separate inputs into equivalence classes: data that should behave the same (vary variables) → 1-3 tests / eq.class, test edges of eq.class, test susceptible combos of eq.class

Logical Defect: logical conditions that govern looping, if/else are wrongly formulated. Use eq.class boundary (coding diagram) → Test to get correct Ans & correct # iterations

Loop Defect: Loop/recursion does not always terminate → create tests to check stopping. **Off-by-One Errors:** Program adds/subtracts one, loops 1-2 too many times (test large #s) (test boundaries (0, 1) usual)

Defects in numerical Algorithms: 1- Not enough bits/digits or places after decimal point. 2- Assuming double is exactly equal to some other value. ← Use >< instead of ==

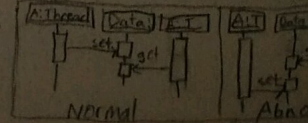
Defects in timing/co-ordination: 1- **Deadlock:** 2 threads waiting for each other to do something (complete block)



→ Use inspection to detect, test by: Varying time consumption, run large # Threads, denying resources to 1+ threads

2- **Livelock:** System can do some computations but can't get out of some states (partial block) →

Critical Race: One thread has failure because other thread interferes w/ normal sequence of events



↳ Test by slowing down one thread / using inspection.
↳ Resolve w/ Semaphores / Synchronized to sync threads (but slows process down)