SE 6205 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]

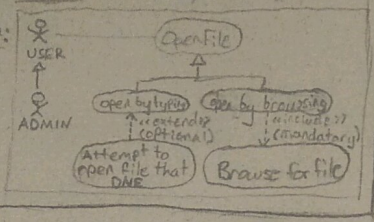
↳ **Types:** • **Functional:** what system does (I/O, computations, timing, stored data, etc.)

• **Non-Functional:** Constraints on → Quality (RUMER), Platform, Process

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

1. User clicks search
2. System shows search form
3. User ... 5. System

↳ **Diagram:**



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

Chapter 5 | Attributes = Instance Variables, Operations = methods

Association: (avoid 1—1 multiplicity) w/ directionality: → (use AC)

Reflexive Association: course, is Mutually Exclusive With, ← Lower bounds must be 0

Association Class: when attribute needs to be in both classes: (must be manually implemented)

Generalization: Δ (IsA), points to superclass, **Propagation:** Occurs when aggregate performs operation on its parts (eg. delete 1 → delete other)

Aggregation: Represents part-whole relat: Whole (Vehicle) ◇ Part (Wheel), **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 part 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)

```
getInstance(){
  if(theInstance==null){
    theInstance=new Singleton();}
  return theInstance;
}
```

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.

push(){list.addFirst();}

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

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

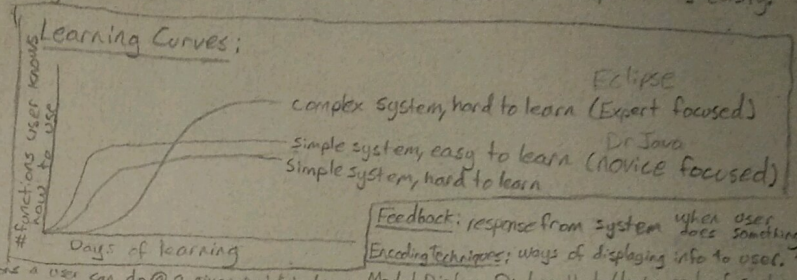
User-centered Design: Have users give feedback during design and creation steps.

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.

Learning Curves:

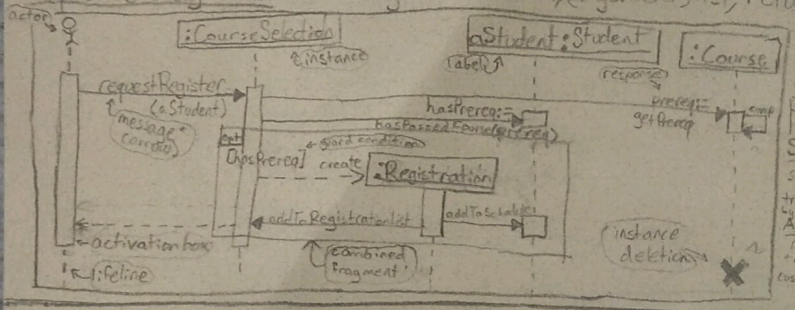


Feedback: response from system when user does something. **Encoding Techniques:** ways of displaying info to user.

Dialog: popup yes/no **Control/Widget:** UI component **Affordance:** All operations a user can do @ a given point in time **Modal Dialog:** Dialog that blocks rest of system.

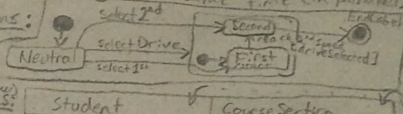
Chapter 8 | Interaction Diagrams: Sequence & Communication ~ diagrams, Sequence > Communication (< has more detail)

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

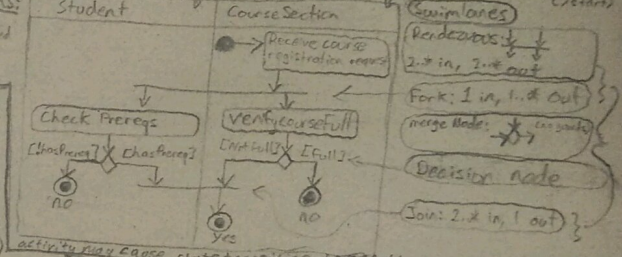


Possible combined fragments: opt = if(condition){} (may not be executed); loop = for(i=min; i<max; i++){} / [condition]; alt = if(cond.){}else{} (one of many will be executed); par = 2 things occurring at the same time (in parallel)

State Diagrams: (composite) has: nested sub states, guard conditions, transitions (event [guard] / actions), start/end states, (actions: entry/exit)



Activity Diagrams: (flow) many classes, transitions caused (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 reusable)

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

DP3: Reduce Coupling: reduce interdependencies between modules, (OO types (least to most desirable): content, common, control, stamp, data, routine-call, type use, inclusion/import, external) → use setters/getters.

DP2: Increase Cohesion: keep related things together, other things away. (7 types (most to least important))

- Functional: Code computing particular result kept together → reuse, easier to understand/replace.
- *Layers: 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: init
- Utility: related utilities which cannot be placed in other cohesive units are together (java.lang.math)

DP4: Increase Abstraction: use classes/interface for info hiding → simpler code

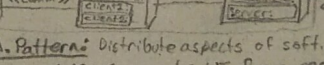
DP5: Increase Reusability: generalize, simplify, have hooks

DP6: Reuse (but don't clone) **DP7: Design for Flexibility:** Prepare for design changes → coupling/cohesion, give options, no hard-code: if(maxe<26) ← 26 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: Client —design→ Server ← 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: → [cancel echo] → [cancel noise] → etc. **MVC:** Separate UI from rest ← this is an A. Pattern

Service-Oriented A. Pattern: organize apps as a collection of services communicating through well-defined interfaces eg. Web services **Message-Oriented API:** sub-systems communicate only by exchanging messages

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

remove line of code: several FIXME shows instead of 2016 Failure, ADD instead of list: defect

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

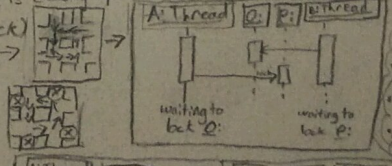
Black-box testing: No access to code, provide inputs & observe outputs. → separate inputs into equivalence classes: data that should behave the same (vary variables, 1-3 tests / eq.class, test edges of eq.class, test susceptible combos of eq.class) → test to get correct Ans (correct # iterations)

Logical (cording) Defect: logical conditions that govern looping, if/else are wrongly formulated. (use eq.class + boundary cording) **Off-by-One Errors:** Program adds/subtracts one, loops 1-2 too many times (test boundaries (as usual)) ← use > / < instead of ==

Loop (cording) Defect: Loop/recursion does not always terminate → create tests to check stopping.

Defects in numerical Algorithms: 1- Not enough bits/digits (test large #s) or places after decimal point. 2- Assuming double is exactly equal to some other value.

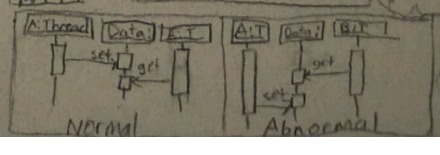
Defects in timing/co-ordination: 1- **Deadlock:** 2 threads waiting for each other to do something (complete block) → Use inspection to detect, test by: varying time consumption, run large # Threads, denying resources to 1+ threads



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/ Semaphore / Synchronized to sync threads (but slow process down)



Levels of Test: •Unit (1 component) •Integration (2 components) •System (requirements) •(Acceptance (User))

By doing Bottom-up Top-down sandwich