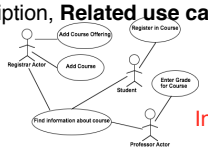
**Chapter 4 :** Use Case Diagrams - describing how the user will use the system // A *use case* is a typical sequence of actions that a user performs in order to complete a given task . **Name, Actors** - users, **Goals** - what the user is trying to achieve, **Preconditions** - state of system before, **Summary** - description, **Related use cases, Step** - two column format, **Post Condition** - state of system after.**Name and step is most important.**




**Use Case Diagrams — >** Extensions - Used to make *optional* interactions explicit or to handle *exceptional* cases.
Generalizations - A generalized use case represents *several similar* use cases.
Inclusions - Allow one to express *commonality* between several different use cases.

**Functional Requirements -** What *inputs* the system should accept // What *outputs* the system should produce // What data the system should *store* that other systems might use // What *computations* the system should perform // The timing and synchronization of the above
**Quality Requirement -** C*onstraints* on the design to meet specified levels of quality
**Platform Requirements -** C*onstraints* on the environment and technology of the system
**Process Requirements -** C*onstraints* on the project plan and development methods

---

**Chapter 5: Attributes -** Instance variables, **Associations -** how two classes are related, avoid 1 - 1 associations, **Reflexive Association -** class that connects to itself, lower bound 0. **Aggregation -** Aggregations are special associations that represent 'part-whole' relationships, **Composition** - if the aggregate is destroyed, then the parts are destroyed as well.
**System Domain Mode -** Can contain less then half of the classes of the system. Doesn't include UI classes or architectural classes.
**System Model -** Contains the system domain model , user interface classes , architectural classes and utility classes.

---

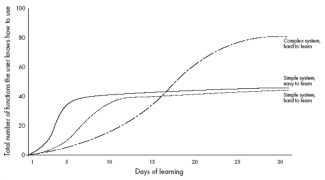
**Chapter 6:** Patterns - **Abstraction Occurrence -** one abstract, multiple occurrences, Example - Title 1 — * Library Item **General Hierarchy** - Hierarchy classes that contain superiors and subordinates. **Player role -** Player is connected to an abstract role with multiple subclasses, Example - Animal 1 — * Habitat Role <— (sub) Aquatic Role, Land Role
**Singleton** - private constructor, getInstance() method. **Observer -** reduces the amount of connections between subsystems. **Delegation Pattern -** reuse methods other classes, Example: implementing stack's Push() using list.addFirst

| *Type of model* | *Contains elements that represent things in the domain* | *Models only things that will actually be implemented* | *Contains elements that do not represent things in the domain, but are needed to build a complete system* |
|---|---|---|---|
| **Exploratory domain model:** developed in domain analysis to learn about the domain | Yes | No | No |
| **System domain model:** models those aspects of the domain represented by the system | Yes | Yes | No |
| **System model:** includes classes used to build the user interface and system architecture | Yes | Yes | Yes |

**Adapter Pattern :** polymorphism w/ , **Façade Pattern:** class that connects to multiple packages, Example - <Facade> 1 — <Package 1> <Package 2>. **Immutable Pattern -** constructor of immutable class is the only place the instance variables are set or modified. All methods that modify return a new instance. **Read-only Interface -** interface with only getters, subclass with getters and setters. Example - <Interface> * — * <Read Class> // <Subclass> * — * <Mutator Class>
**Proxy Pattern:** Reduce the need to instantiate a heavy weight class. Example - <Client> * —> * <Proxy> 0..1 —> 0..1 <Heavy>
**Factory Pattern:** The framework delegates the creation of application-specific classes to a specialized class, the Factory.

---

**Chapter 7: Usability - Learnability - t**he speed with which a new user can become proficient with the system. **Efficiency - h**ow speed with which an expert user can do their work. **Feedback - t**he extent to which it prevents the user from making errors, detects errors, and helps to correct errors. Adaptability - the extent to which users *like* the system. **Learning Curve -**



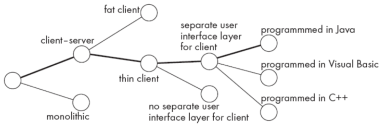

**Dialog -** A specific window with which a user can interact, but which is not the main UI window **Widget (Control) -** specific components of a user interface, **Modal Dialog -** a dialog in which the system is in a very restrictive mode. **Feedback -** The *response from the system* whenever the use feedback. **Encoding -** ways of encoding information so as to communicate it to the user. Types of encoding: text & font, icons, photographs, diagrams & graphics, colours, grouping & bordering spoken words, music, sounds, animation & video, flashing. **Heuristic Evaluation - 1.** pick some use cases to evaluate. **2.** for each window, page or dialog that appears, study each detail to look for possible usability defects. **3.** when you discover a usability defect write down the following information - a short description of the defects - your ideas for how the defect might be fixed. **Video tape evaluation (observation) - 1.** select users based on actors, **2.** select important use cases, **3.** write sufficient instructions, **4.** arrange sessions with users, **5.** explain purpose, **6.** video each session, **7.** converse with users while they perform tasks, **8.** de brief users when they finish tasks, **9. -** note any difficulties user experience, **10.** formulate recommended changes. **Affordance -** what the user can do.

---

**Chapter 8: Activity -** something that takes place while the system is *in* a state, It takes a period of time, the system may take a transition out of the state in response to completion of the activity, other outgoing transition may result in: The interruption of the activity or an early exit from the state. **Events :** transitions are caused by events, Example - the completion of a computation. **Transitions -** represents a change of state in response to an event, events is considered to occur instantaneously, the label on each transition is the event that causes the change of state. **Guard Condition-** condition for how many times loop runs**. Concurrency -** is shown using forks, joins and rendezvous, **Fork -** has one incoming transition and multiple outgoing transitions, **Rendezvous -** has multiple incoming and multiple outgoing transitions. **OPT -** for optional behaviour, **ALT -** for conditions, **PAR -** for concurrent, **Loop -** for loops, **Composite -** combined

---

**Chapter 9: Design Space -** The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space,*



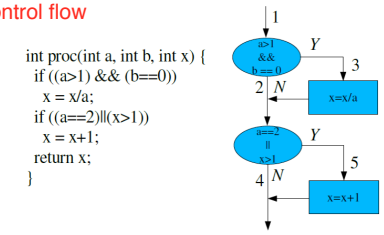
**System -** A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both, A system can have a specification which is then implemented by a collection of components, A system continues to exist, even if its components are changed or replaced, the goal of requirements analysis is to determine the responsibilities of a system. **Components -** Any piece of software or hardware that has a clear role, A component can be isolated, allowing you to replace it with a different component that has equivalent functionality, Many components are designed to be reusable, Conversely, others perform special-purpose functions. **Module -** A component that is defined at the programming language level, Example - methods, classes and packages are modules in Java. **Top-Down -** start with high level structures , gradually work down to detailed decisions , finally arrive at detailed decisions. such as data items and particular algorithms. **Bottom-Up -** make decisions about reusable-low-level utilities, then decide how these will be put together to create high level constructs. Usually using a mixture of both. **Divide & Conquer -** basically separate roles into engineers strengths. **Cohesion -** Increase Cohesion, **functional**: all the code that computes a particular result is kept together, **layer**: all facilities for providing a set of related functions are kept together, **communicational :** modules that access certain data are kept together, **sequential :** procedures in which one provides input to the next, **Procedural :** procedures that are used one after another are kept together. **Temporal :** operations that are performed during the same phase are kept together. **Utility :** related utilities which cannot be logically placed in other cohesive units are kept together, **Coupling -** reduce coupling, interdependencies between two modules **Content -** one component modifies internal data of another component <private>, **Common -** occurs when global variables are used, **Control -** one procedure calls another using a command Example - if command().equals("Draw Circle") drawCircle() else DrawSquare(); **Stamp -** occurs when a method uses a class as it's type, **Data -** too many primitive method arguments. **Routine Call -** occurs when one method calls another (unavoidable) **Type Use -** variables that have class type, **Inclusion or Import -** import the whole package instead of what you need **External -** module has dependency on hardware or operating system. **Abstraction -** classes, UML , Interface (hide information) **Reusability -** generalize when possible, design hooks. **Flexibility -** reduce coupling / increase cohesion, do not hard code anything, lease options open, code reusable. **Portability -** run on as many platforms as possible. **Testability -** design a program to test automatically. **Design by contract -** each method has a contract with preconditions, postconditions and invariants. **Layer architecture -** each layer only communicates with the layer below it Example - UI —> Application Logic —> Database **Broker -** object can call methods of another object without knowing that this object is remotely located. Example - Client —> Broker —> Remote Object. **Pipe Filter -** stream of data in simple format is passed through a series of processes, characterized by functionally cohesive components Example - microphone recording **MVC -** model : underlaying classes, view - handles UI, Controller - handles user inputs. **Service Oriented -** organizes an application as a collection of services that communicate with each other Example - http requests. **Message Oriented -** sub-systems accomplish tasks by exchanging messages // (1. D&C 2. increase cohesion 3. reduce coupling 4. incr abstract 5. inc reusability 6. incr reuse 7. incr flexibility 8. design for obsolescence 9. Design for portability 10. Design for portability 11. Design defensively

---

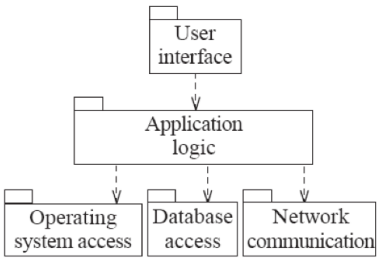
**Chapter 10: Failure -** an unacceptable behaviour exhibited by the system **Defect (bug) -** a flaw in the system that may contribute to a failure. **Error -** a slip up decision by the developer that leads to a defect. **Black box -** Testers provide the system with inputs and observe the outputs, they cannot see the internals of the system. **White-Box -** have access to code and can run debuggers, run a trace and inject new code. **Equivalent Classes -** impossible to test all cases, instead divide inputs into groups that will be treated similar by the system. **Logical Defect -** loops / if else statements are wrongly formulated. **Loop defect -** not terminating a loop or recursion. **Off-by-one -** adds or subtracts one, loop one less or too many times. **Numerical Algorithm -** x*y+z should be x*(y+z) **Deadlock -** Two threads are stop waiting for each other to do something. **Livelock -** system can do computations but can't get out of states. **Critical Race -** one thread experiences failure because another thread interferes with normal sequence of events.

---

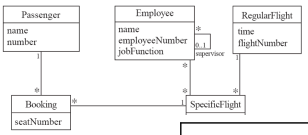
control flow




layer

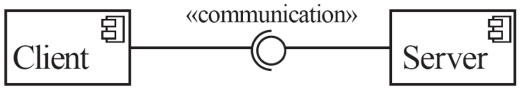



UML Class

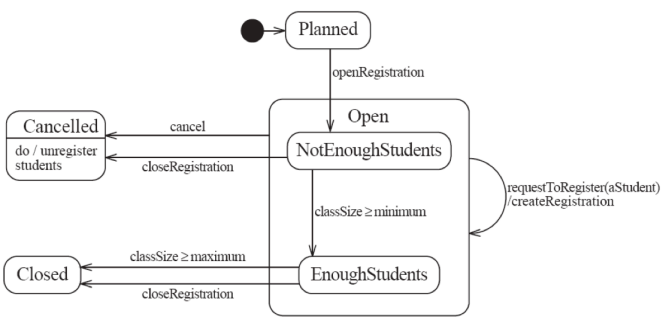



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Multi-layers | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| Client-server | ■ | ■ | ■ | ■ | | ■ | ■ | | ■ | ■ | |
| Broker | ■ | | | | ■ | ■ | ■ | | ■ | | ■ |
| Transaction processing | ■ | ■ | ■ | | | | ■ | | | | ■ |
| Pipe-and-filter | ■ | ■ | ■ | ■ | ■ | ■ | ■ | | | ■ | ■ |
| MVC | ■ | ■ | ■ | ■ | | | ■ | | | ■ | |
| Service-oriented | ■ | ■ | ■ | ■ | ■ | ■ | | ■ | ■ | ■ | ■ |
| Message-oriented | ■ | ■ | ■ | ■ | ■ | ■ | ■ | | | ■ | ■ |

Component




UML State




UML Sequence