**[INTRODUCTION]**

**Types of Systems**

* Soft – difficult to define precisely
  + Depends on viewpoint of person describing it
  + Difficult to agree on boundaries and behaviour
  + Eg, all human activity systems
* Hard – well defined
  + Easy to agree on boundaries
  + Eg, mechanical operation of car

**Problem** **statement** – clear and concise description of the issue(s) that need(s) to be addressed by a problem solving team. Issues, Goals and Objectives, Method, and Scope.

**Requirements** **engineering** – set of activities concerned with identifying and communicating the purpose of a software-intensive system and the context in which it will be used

Requirement analyst must identify the problem/opportunity

* Which problems need to be solved? (problem boundaries)
* Where is the problem? (context/problem domain)
* Whose problem is it? (stakeholders)
* Why does it need solving? (goals)
* How might a software system help? (scenarios)
* When does it need solving? (development constraints)
* What might prevent us solving it? (feasibility/risk)

Typical problems in requirement specifications:

* Noise
* Silence
* Over specification
* Contradiction
* Ambiguity
* Forward reference

Things you want:

* Valid/correct
* Unambiguous
* Complete
* Understandable
* Consistent
* Ranked
* Verifiable
* Modifiable
* Traceable

**Requirements** **Specification** – Purpose

* Communication
* Contractual
* Baseline for evaluating the software
* Baseline for change control

Re**q**uirements **Specification** – Audience

* Customers and users
* Systems analysts
* Developers/programmers
* Testers
* Project managers

Specification should include

* Functionality
* External influences
* Required performance
* Quality attributes
* Design constraints

**Critical** **System** – a computer, electronic or electromechanical system that the failure of which may have serious consequences (human death/injury, failure of goal directed activity, financial loss/failure of business)

* Safety-critical systems
* Mission-critical systems
* Business-critical systems

**Validation** – ensure that the specification of the software system satisfy the requirements of the stakeholders

**Verification** – ensure that the finished product satisfies the specification

**[REQUIREMENTS ANALYSIS]**

Requirements 🡪 Design 🡪 Implementation 🡪 Verification 🡪 Maintenance

**Requirements** – description of something a product must do/quality it must have

* Functional – things the product must do or provide to be useful for users
* Non-functional – qualities the product must have
* Design – how the system will be designed and implemented

**Constraints** – global issues that shape the requirements

**Assumptions** – describe properties of the environment of a software product that the software can’t directly control, but necessary for overall functioning.

**Verifiable** – A requirement should be verifiable.

* Should be a clear way of checking that an implementation can satisfy requirements
* Eg, by defining test cases for that requirement

**PROCESS**

**1. Scope** – provides a high level view of the planned product: Need, Goals, Case, High-level operational concepts, Stakeholders. Purpose:

* Helps keep bigger picture in view
* Ensure all stakeholders have a common view
* Ideally should remain fixed throughout development process

**2. Use Case** – abstractions from scenarios for usage of the system typically describing series of interactions between users and the system to be developed.

* Help people quickly gain an understanding of what is intended
* Convenient way of generating requirements
* Helps to avoid omitting requirements

**3. Interfaces –** points where the system interacts with environment (users, other software components, physical devices)

* Helps to define boundaries of responsibility for the development teams
* Important for defining ways of testing implemented systems

**[Software Modelling and UML]**

**UML** – visual modelling language for specifying, analysing and designing/documenting essential aspects of software systems before construction. 14 types of diagram.

* Structural: show static structure of the system and its parts on different abstraction and implementation levels and how they relate to each other

Eg, class diagram, component diagram

* Behaviour: shows dynamic behaviour of the objects in a system, which can be described as a series of changes to the system over time

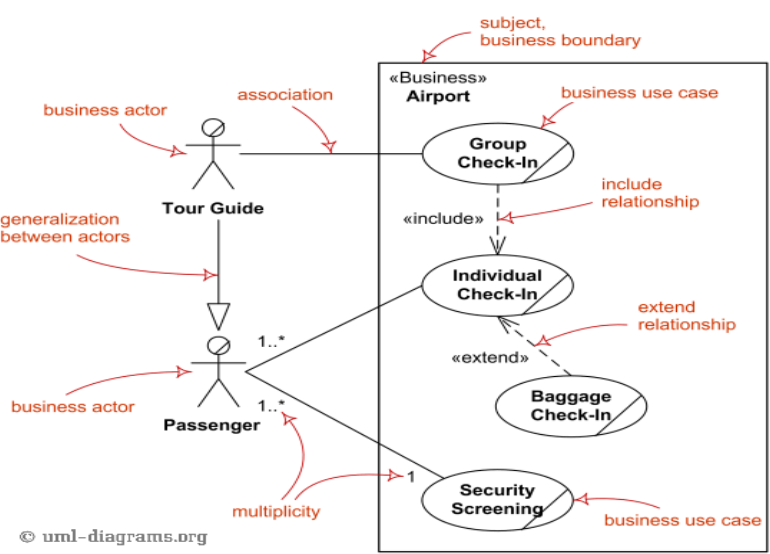
Eg, use case, sequence diagram, state diagram, activity diagram

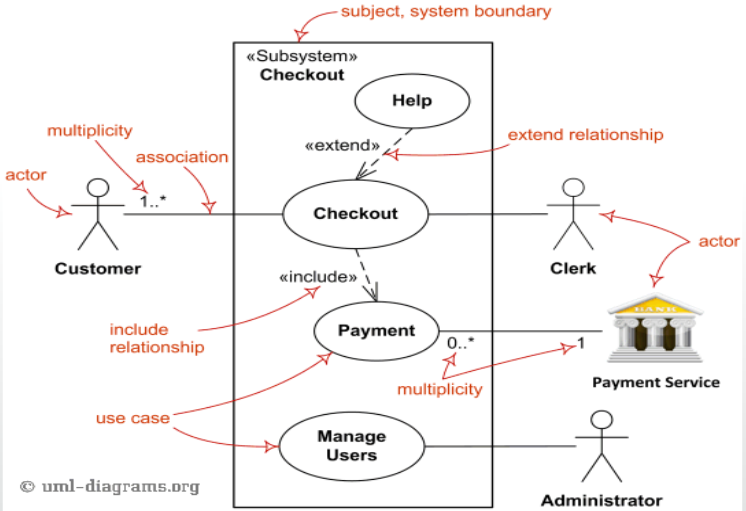
**Use Case Diagrams -** generalised description of how a system will be used**.** Provides an overview of the intended functionality of the system

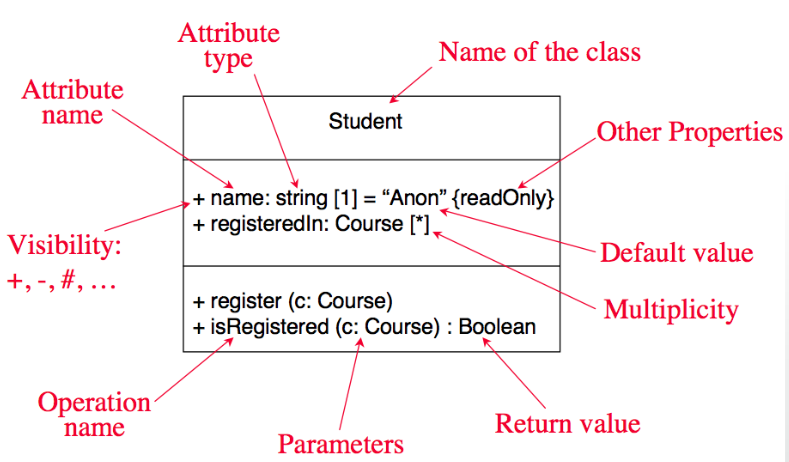
* System boundaries (rectangle)
* Actors (people, eg users)
* Use case: set of scenarios describing interactions between a user and system (ovals)

Relationships:

* Association: communication between an actor and use case
* Generalisations: relationship between one general use case and one specific use case. Arrow towards parent use case
* Include: a particular use case must include another use case to perform function. A dotted arrow from base use class to include use class
* Extend: The extending use class may add behaviour to the base use case. The base class declares ‘extension points’





Class Diagram - classes describe groups of objects with similar attributes, common operations, relationships and meanings

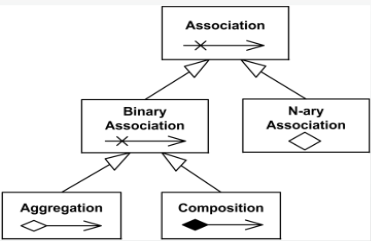
Modifiers used to indicate visibility:

* + Public
* # Protected
* – Private

Relationships:

Directed

* Generalization: an inheritance link indicating one class is a superclass of the other.

Association:

* a relationship between instances of the two classes. Eg, if an instance of one class must know about the other to perform its work.
  + Bi-directional
  + Uni-directional
* Aggregation: association where one class belongs to a collection
* Composition: ‘strong’ form of aggregation. Whole/part relationship.

Class Diagrams

+ easy to discover related data and attributes

+ gets a quick picture of important entities

+ see whether you have too few/many classes

+ see whether the relationships between objects are too complex

+ spot indecencies between one class/object and another

- doesn’t discover algorithmic (not data-driven) behaviour

- doesn’t find the flow of steps for objects to solve a given problem

- doesn’t understand the apps overall control flow (event driven/sequential?)

**State Machine Diagram** – graph where nodes correspond to states and directed arcs correspond to transitions labelled with event names. Visualise how objects respond to stimuli

* State: duration of time during which an object is doing an action
* Event: occurs at a point in time and transmits information from one object to another
* Action: occurs in response to an event and cannot be interrupted
* Activity: is an operation with certain duration that can be interrupted by another activity
* Guard: logical condition placed before a transition (true/false)

**Sequence Diagram**

Object interactions

* Self-call – a message that an object sends to itself
* Condition – indicates when a message is sent (only sent is condition is true)

Object life span

* Lifelines (dotted lines)
* Messages (arrows, direction indication direction of call)
* Activation bar (rectangles)
* Depletion (placing an X on lifeline. Objects life ends at that point)

**[MORE BLOODY UML]**