**[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

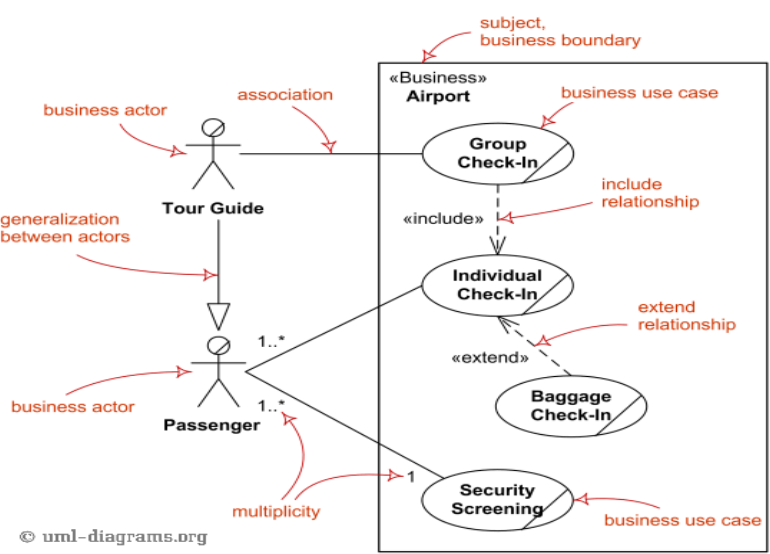
Eg, use case, sequence diagram, state diagram, activity diagram

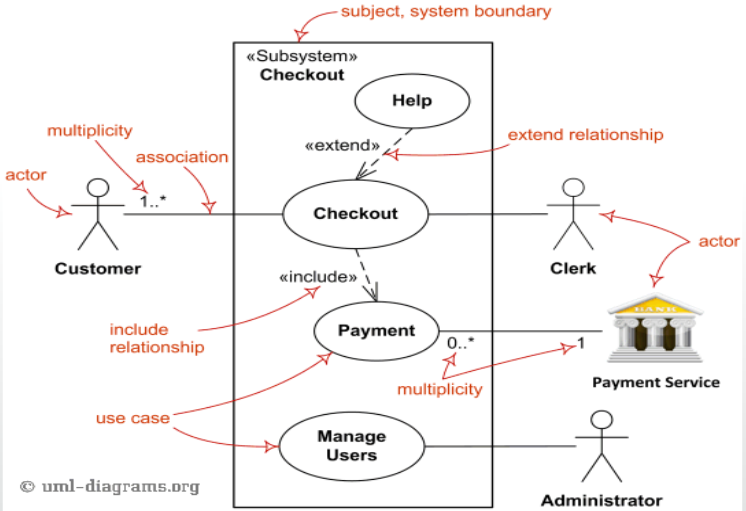
**Use Case Diagrams -** generalised description of how a system will be used**.** Provides an overview of the intended functionality of the system

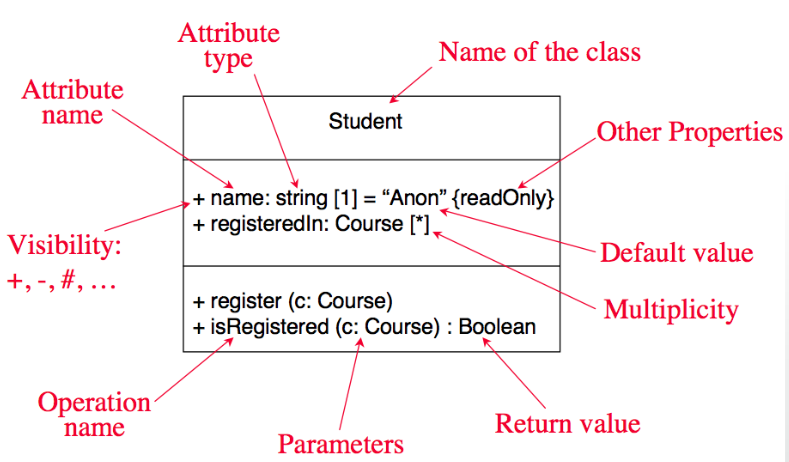
* System boundaries (rectangle)
* Actors (people, eg users)
* Use case: set of scenarios describing interactions between a user and system (ovals)

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’





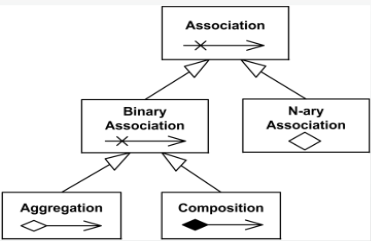
**Class Diagram** - classes describe groups of objects

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.

+ 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

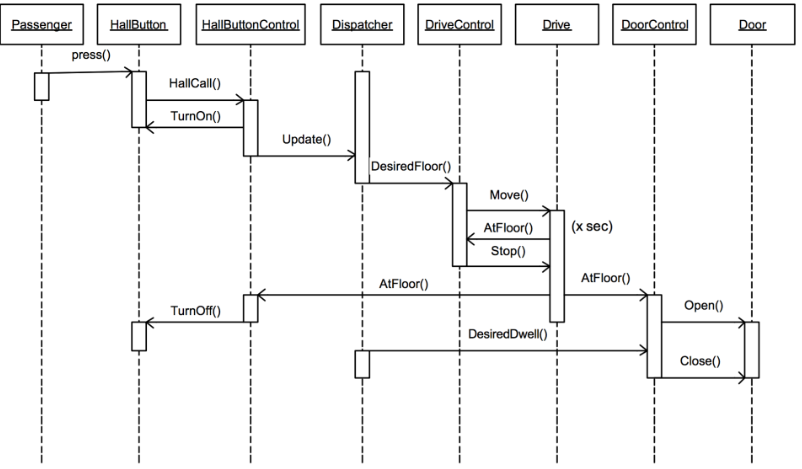
- doesn’t understand the apps overall control flow (event driven/sequential?)

**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

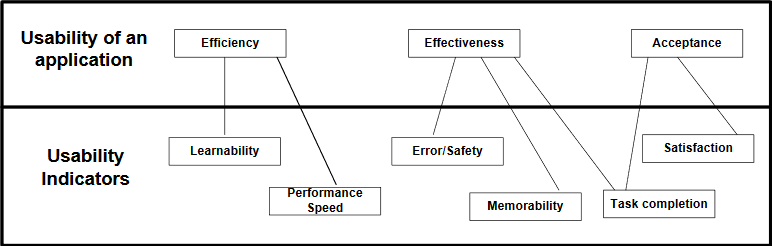
* Self-call – a message that an object sends to itself
* Condition – indicates when a message is sent (only sent is condition is true)

Object life span

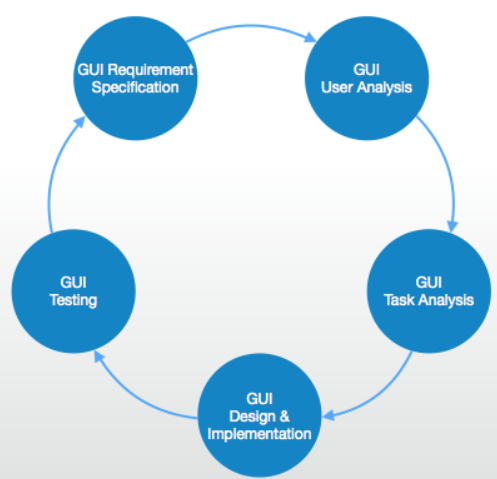
* Lifelines (dotted lines)
* Messages (arrows, direction indication direction of call)
* Activation bar (rectangles)
* Depletion (placing an X on lifeline. Objects life ends at that point)

**[UI DESIGN]**

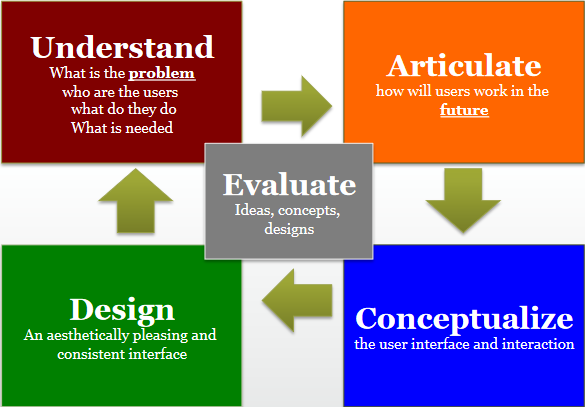
User interface – allows users to accomplish their goals using software without unnecessary effort. Provides appropriate means for using the system in an efficient manner (layout, controls

Effective interaction is determined by and measured using Usability Indicators

UI Design – iterative process including:

* Create prototype of system
* End user testing prototype
* Gathering data from these tests
* Re-designing the interface to address discovered problems

Steps:



Objectives – build an understanding of the design problems and requirements

* Business problems – to identify what we are trying to solve
* User characteristics – what they are like, what do they want, try to avoid, goals, needs
* Use environment – where users use the product/application

Articulate – how will people use the product in the future?

Conceptualize – objectives. Build a clear and shared vision of the product or app

* How UI will appear
* Key user tasks are accounted for
* Select and wireframe the most suitable design

Wire frame diagram – schematic produced displaying where all the agreed features and functions within the website spec will appear

* Cost effective and save time further down the line
* Basic layout composition is planned and agreed to before any complex actual implementation work is undertaken

Design objective – why?

* Create complaint and aesthetically pleasing render of the app wireframe
* Ensure clarity and simplicity at the user interface
* Help new uses quickly master product usage by adding guidance and instruction where needed

Design – methods

* Review and research and conceptual design to understand goals and proposed design
* Skin each wireframe component to comply with corporate guidelines
* Identify and design special components required by the design (icons, special labels, highlight colour, etc)
* Review visuals with ux and documentation

Evaluate – objectives

* Provide constructive and actionable input to specific design questions/issues
* Continuously verify proposed design solutions
* Ensures usability both at the micro and macro level

Evaluate – methods

* User reviews
* Local and remote usability tests
* A/B testing compering design alternatives
* On-line surveys

General UI Design Principles

1. Visibility of system status
2. Match between system and real world
3. User control and freedom (eg, undo and redo)
4. Consistency and standards (in line with users expectations)
5. Error prevention (help avoid mistakes)
6. Recognition rather than recall (rapid and easy learning of system)
7. Flexibility and efficiency of use (eg shortcuts/accelerators for experienced users)
8. Aesthetics and minimalist design (only necessary info)
9. Help users recognise, diagnose and recover from errors (sensible error messages)
10. Help and document

Lab tests and stats

* Number of subjects who can complete the tasks within a certain time
* Length of time required to complete different tasks
* Number of times help functions was needed
* Number of times redo was used and where
* Number of times shortcuts were used

Utility – does the system provide the raw capabilities to allow the user to achieve their goal?

Usability – does the system allow the user to learn and use the raw capabilities easily?

* Learnability
* Efficiency of use
* Error handling
* Acceptability

Usability principles:

1. Do not rely on usability guidelines – always test with users
2. Base UI design on users’ tasks (use case analysis)
3. Ensure sequences of actions to achieve a task are as simple as possible (˅ reading)
4. Ensure user always knows what he or she can and should do next
5. Provide good feedback including effective error messages (inform about progression and location)
6. Ensure the user can always get out, go back, or undo an action
7. Ensure that response time is adequate
8. Use understandable encoding techniques
9. Ensure that the UIs appearance is uncluttered (effective organisation and don’t overload with info)
10. Consider the needs of different groups (locales and people with disabilities, and ensure that the system is usable by both beginners and experts)
11. Provide all necessary help
12. Be consistent

Difficulties and risks

* Users differ widely
* UI implementation technology changes rapidly
* UI design and implementation can often take the majority of work in an app
* Developers often underestimate the weakness of a GUI

GUI frameworks

* Java Swing (java)
* Windows Forms (.NET languages, Windows)
* Qt (C++)
* Gtk2 (C/C++)

**[SOFTWARE TESTING]**