5SENG007W Software Engineering Principles and Practice

Lecture Week 3

Requirements Engineering 2: revision of Lecture week 2, types of requirements, priority, formalisation (UML), use case analysis

Lecture Outline

- Review of Week 2 Lecture
- Requirement prioritisation
- Requirements documentation
- Requirements formalisation - UML

Levels of Requirements

- Requirement can be high-level abstract statements, detailed descriptions or a formal specification
- We often categorise requirements as
  - *User requirements*
  - *System requirements*
- This categorisation is useful because they communicate information about the system to different types of readers

User Requirements

- Written for users and customers
- High-level abstract statements
- Describe ***user goals*** or ***tasks*** that the users must be able to perform with the system
- Should describe requirements in such a way that they are understandable by non-technical people
- Should be written in simple ***natural language (NL)***, with simple tables and forms and intuitive diagrams

User Requirements: Examples

- Different formats exist
  - <type of user> shall/will/should some <task/goal>

    Example:

    Library members shall be able to place reservations on items
  - As a <type of user>, I want <some task/goal> so that <some reason>.

    Example:

    As a student, I want to be able to access my timetable online so that I check where/when my classes are when I need to

System Requirements

- Detailed descriptions of the system’s functions, services and operational constraints
- They add detail to user requirements
  - Usually, one user requirement leads to several system requirements
- Define in detail exactly what is to be implemented
- Should be precise, complete and consistent specification of the whole system
- They serve as a basis for the design
- May be part of a contract between client and contractor

System Requirements

- Can be written
  - In natural language (NL) in text
  - In a more specialised notations e.g. structured language or graphical notations
    - Use case description template
    - Class diagrams
    - Sequence diagrams
    - Activity diagrams
    - .... and similar

Example – User/System Requirements in NL

1. Library members shall be able to reserve books

User requirement

- 1.1 The system shall allow a member to reserve an item only if no copies of the book is available at that time.
- 1.2 The system shall allow a member to have up to 10 reservations at any time
- 1.3 The system shall not allow a member to make a reservation if their account is suspended
- 1.4 The system shall place reservation on items rather than item copies
- 1.5 The system shall send an email confirmation containing the reservation number and the book details once the reservation is made.
- 1.6 ....

Systems requirements

Example - User/System Requirements

User story

– As a library member, I want to be able to reserve a book online so that I can borrow it as soon as it is back in the library. (FUNCTIONAL)

Confirmations

– Reservation won’t be made if student account is suspended
– The system shall allow a member to have up to 10 reservations at any time
– Reservations will be placed on books titles, not book copies
– Reservation will only be allowed if all copies of a book are out on loan
– The system shall send an email confirmation containing the reservation number and the book details once the reservation is made.
– …..

Requirements Types

- Requirement conventionally separated as
  - *Functional Requirements*
  - *Non-functional requirements*
- Functional Requirements → Statement of system services (i.e. system capabilities)
- Non-functional requirements → Statement of constraints (e.g. conditions and qualities)

Functional Requirements

- Statements of services the system should provide
- Describe:
  - *what the system should do*
  - *how the system should behave in a particular situation*
- Include:
  - *processes*
  - *interfaces with users and other systems*
  - *data that the system must hold*

Functional Requirements

- Can be fairly abstract or detailed depending on whether it expresses a user or system requirement
- Often modelled / documented with diagrams such as
  - Use Case Diagrams
  - Interaction Diagrams
  - Class Diagrams

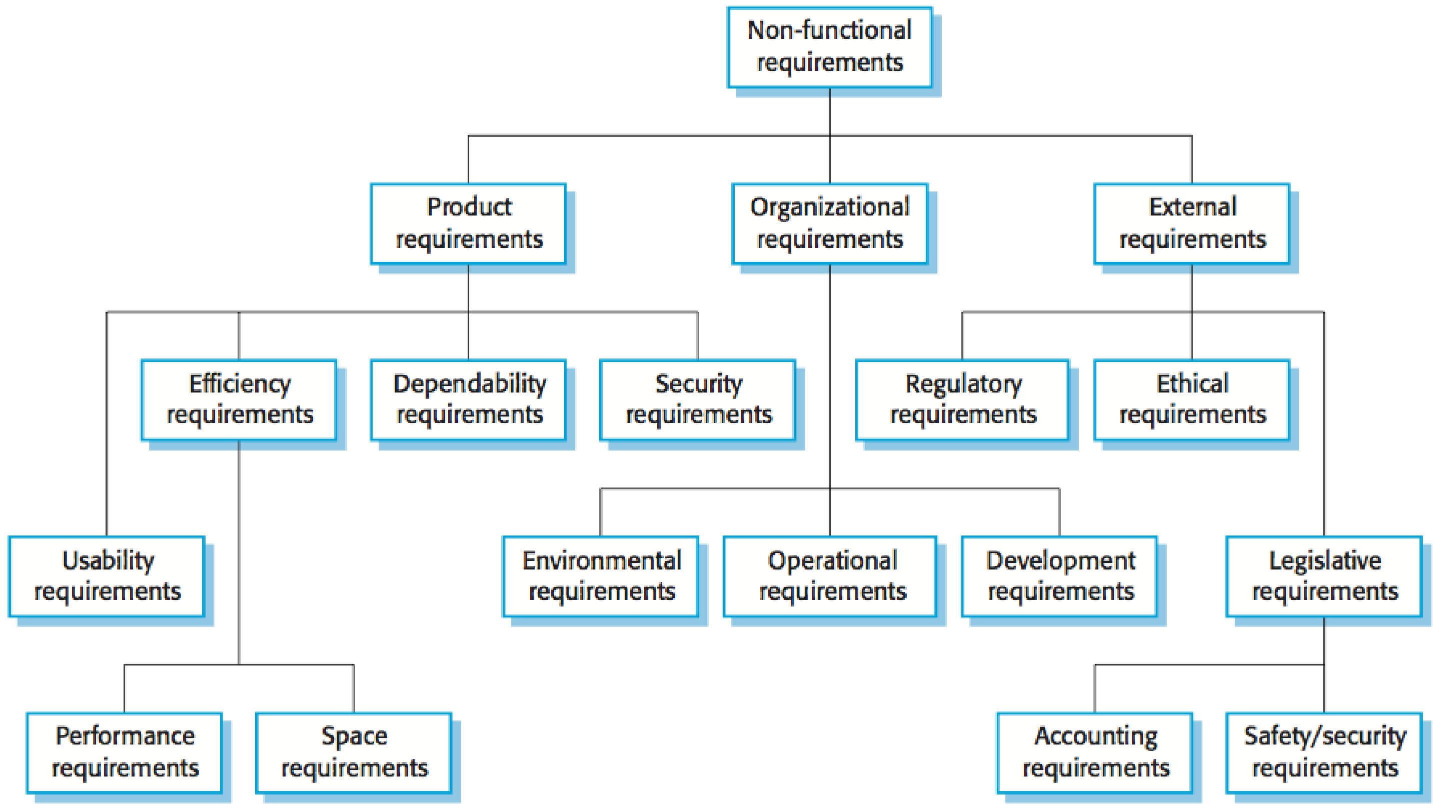
Functional Requirements: Examples

- Library members shall be able to renew their loans one by one or all at the same time (library system)
- Customers shall have the option to save their credit card details for their future purchases (online retail site)
- Users shall be able to comment on their friends' photos (a social networking site)
- The system will allow to record a programme while the customer watching another (TV set)
- For a given postcode, the system shall be able to work out the quickest route for a journey (Satnav)
- etc.

Non-functional Requirements

- Concerned with
  - How the system should behave
  - How well the system should perform
- Include conditions or constrains around
  - Performance – for example response times
  - Capacity – for example, volumes of data
  - Reliability
  - Availability
  - Portability and interoperability
  - Maintainability
  - Usability
  - Security and privacy,
  - … and so on.

Non-functional Requirements



Source: (Sommerville, 2016)

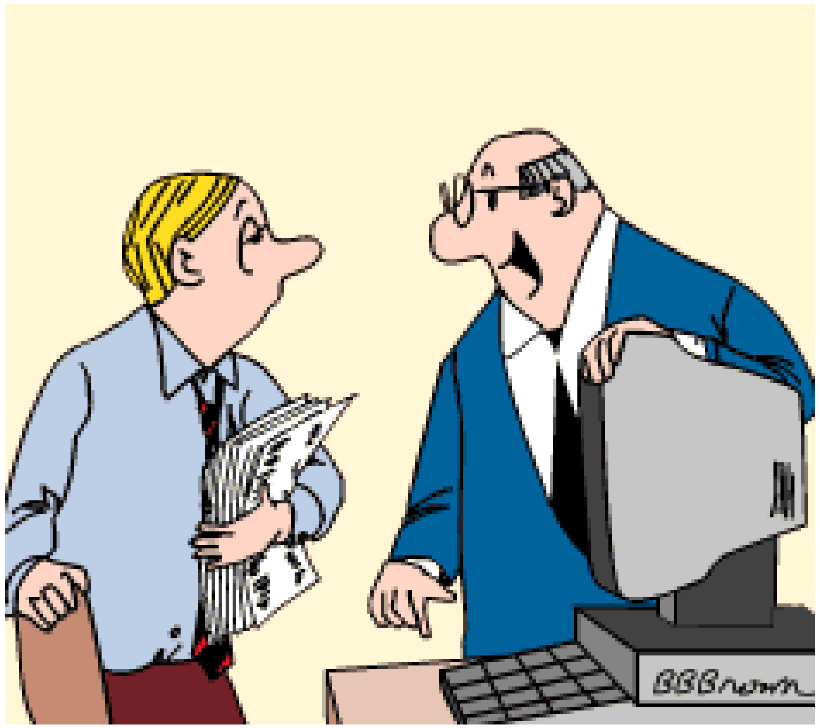
Non-functional Requirements

- They can be more critical than functional requirements
  - For example, if a system doesn't meet its reliability requirements it may not be certified as safe for operation
- Some non-functional requirements (NFR) may relate to the development process
- Often documented as a list but may be linked to specific use cases descriptions
- Whenever possible quantify the non-functional requirements so that they are measurable

Non-functional Requirements: Examples

- The system should be able to handle up to 500 users at the same time without causing any delays to user requests.
- All simulations will complete within 2 hours no matter how complex the problem is (simulation software)
- Sales reports will not be accessible by users, other than store managers (POS system)
- The software shall be compatible with macOS and Windows OS.
- Experienced controllers shall be able to use all the system functions after two hours training. After this training, the average number of errors shall not exceed two per day

Requirements Prioritisation



"THIS COMPUTER IS EQUIPPED WITH AN AIRBAG IN CASE YOU FALL ASLEEP!"

Prioritising Requirements

- All requirements are important
- However, all project have constraints such as time, budget, skills, …
- Stakeholders may have conflicting views e.g. not agree on what is essential and what is luxury when too many requirements are identified
- Requirements should be prioritised so that the requirements that delivers the greatest business benefits early

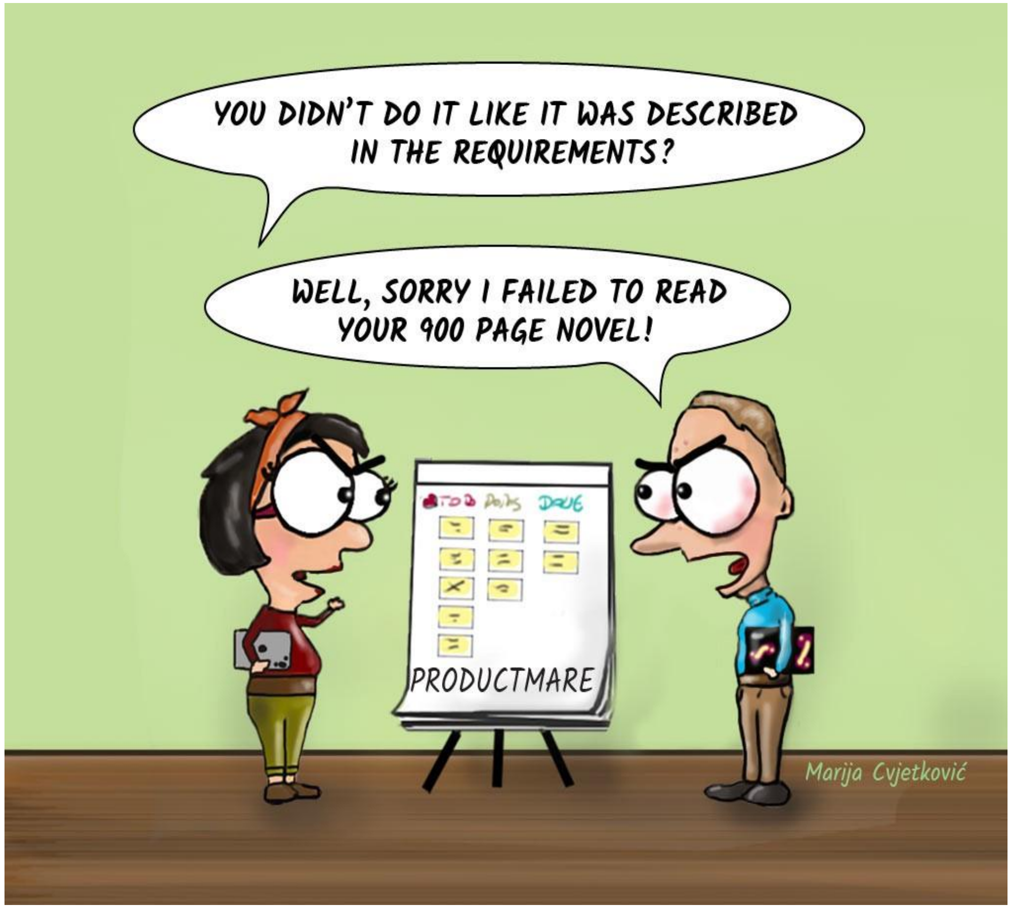
Prioritising Requirements

- Some software development or project management methods provides guidance & techniques on this
- So, make sure you understands the approach of your method to prioritisation
- For example ‘MoSCoW’ rule is a techniques from DSDM, an Agile Method, to prioritise requirements
  - **M**ust have
  - **S**hould have
  - **C**ould have
  - **W**on’t have (may be in future projects)

Prioritising Requirements

- Alternatively, requirements can be categorised as:
  - Essential
  - Desirable
  - Out of scope
- Or requirements may be given a weighting e.g. some use this when outsourcing software
  - e.g. 1 to 5 based on their value or importance
  - 1 indicating lowest and 5 indicating the highest value requirements

Requirements Documentation


YOU DIDN'T DO IT LIKE IT WAS DESCRIBED IN THE REQUIREMENTS?
WELL, SORRY I FAILED TO READ YOUR 900 PAGE NOVEL!
PRODUCTMARE
Marija Cvjetković

Requirements Document

- The requirements document is the official statement of what is required of the software/system developers.
- Sometimes called Software Requirements Specification (SRS)
- It should include both the user and a detailed specification of the system requirements.
- It is NOT a design document. As far as possible, it should set WHAT the system should do rather than HOW it should do it

Requirements Specification

- The chosen development methodology affects the way the requirements are identified and documented.
- Those methodologies based on structured model capture & specify all requirements upfront after the Requirements Analysis Phase.
- Those methodologies based on iterative & incremental model
  - first capture the high level requirements, and
  - develop the details of requirements in each iteration/increment as the project progresses

Requirements Specification

- Natural Language (NL)
- Structured Language
  - Standard forms or templates which guides and limit the freedom of the requirements writer hence all requirements are written in a standard way e.g. use case templates
- Graphical Models
  - Such as Use Case Diagrams, ER Models, Class Diagrams, Activity Diagram, Flowcharts, Sequence Diagrams …
- Other formal methods e.g. Design Description Language or Mathematical Specification

Problems with NL

- Lack of clarity, ambiguity
  - *Some thing are* only meaningful if you know the exact context.
- Over-flexibility
  - *The same thing may be said in a number of different ways*
- *Lack of modularisation*
  - *NL structures are inadequate to structure system requirements*
- Requirements amalgamation
  - *Several different requirements tend to be expressed together*
- Requirements confusion
  - *Different types of requirements (i.e. Functional and non-functional requirements) tend to be mixed-up*

Writing Requirements in NL

To minimise misunderstandings

- A standard ***template*** may help. For example,
  - description
  - rationale
  - source
  - reference to the system requirement
- Use language consistently e.g. use '***shall***' for essential and '***should***' for desirable requirements
- Avoid computer jargon as far as possible
- Use text highlighting to identify key parts of requirements

Modelling Languages

- Use of a standard modelling language addresses the problems with the NL
- Enables unambiguous communication between members of team.
- Help with dealing with complexity
- This can then become a central feature of a quality process.

Requirements Document - Indicative Content

- Stakeholder Analysis
- Requirements gathering
- User requirements definition
- System architecture
- System requirements specification
- System models
- Appendices
  - Interview transcripts

Sources

- I Sommerville, Software Engineering, 10/E, Pearson, 2016 [electronic resource]
- I F. Alexander and L Beus-Dukic, Discovering Requirements: How to Specify Products and Services, Wiley, 2009 [electronic resource]
- K U Wieger and J Beatty, Software Requirements, 3/E, Microsoft Press, 2013 [electronic resource]
- S Bennett, S McRobb and R Farmer, Object Oriented Systems Analysis and Design: Using UML, 3/E, McGraw Hill, 2006.
- Kotonya and Sommerville: Requirements Engineering, Wiley, 1998
- Ian K. Bray: An introduction to Requirements Engineering, Addison Wesley, 2002
- S Lauesen, *Software Requirements: Styles and Techniques,* Addison Wesley, 2002

Software Development Project Essentials

- Understand the problem (gather and analyse the requirements)
- Specify the solution (document and validate the requirements)
- Validate and implement the Solution

Philosophy

- Requirements are frequently described as stating WHAT the system will do, rather than HOW it will do it.
- Requirements:
- Gathering => Analysis => Specification

Requirements Analysis

- Priority, Hierarchy
- Importance
- Interaction

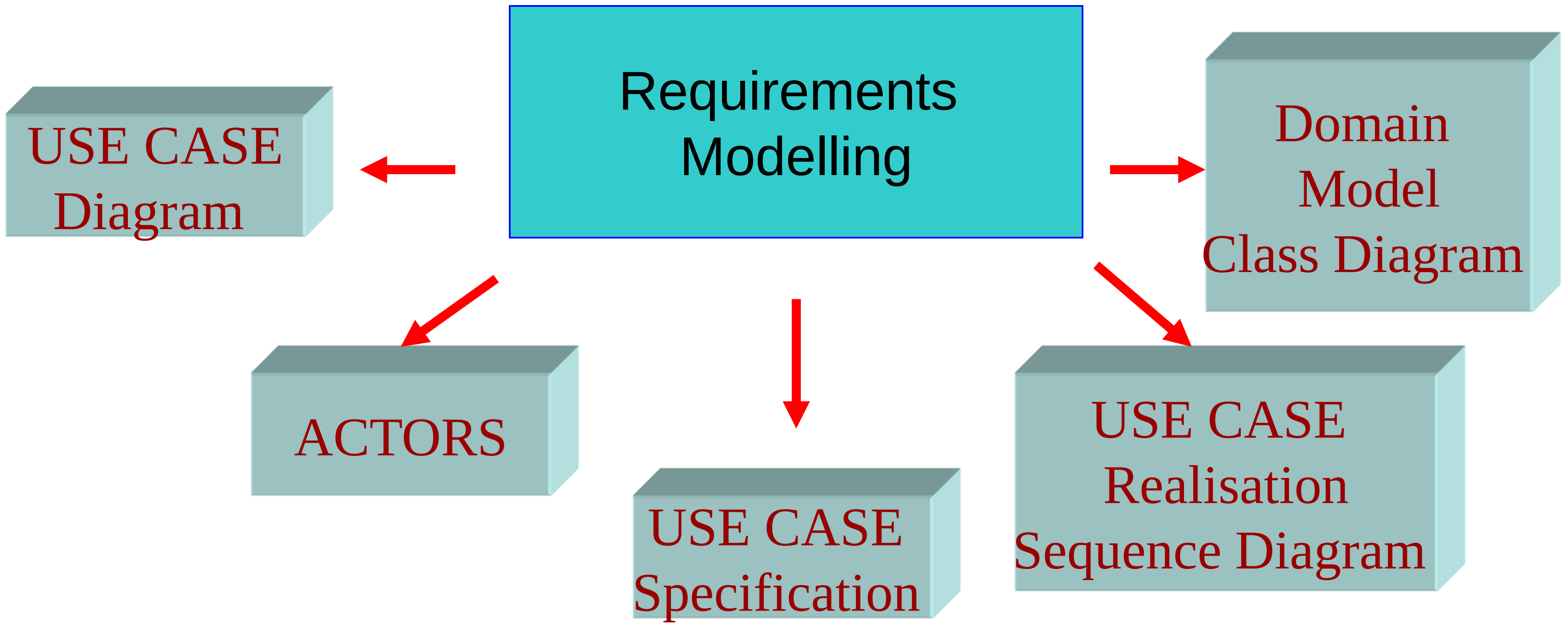
$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\}$ Requirements Structure

- Feasibility (can we achieve given the resources and the scope)
- Coverage (anything missed? Anything redundant?)

MODELLING

- Is a powerful method
- Like in many other areas when we make complex information systems, write computer programs we first build a model
- Mathematical models are rigorous and widely used in computer science and programming

Requirements Analysis: MODELLING



UML MODELLING

- UML is a modelling language or graphical/diagrammatic notation for object-oriented programming – a way to express the "blueprints" of your system.
- UML Diagrams
  - USE CASE DIAGRAM
  - SEQUENCE MODEL DIAGRAM
  - CLASS DIAGRAM
  - Activity DIAGRAM
  - STATE MACHINE DIAGRAM

What is a Use Case?

- a *use case* for a system represents a collection of scenarios describing ways in which the system is used
- a *scenario* is a particular sequence of actions carried out by the system

Use Case Modelling- Purpose of Technique

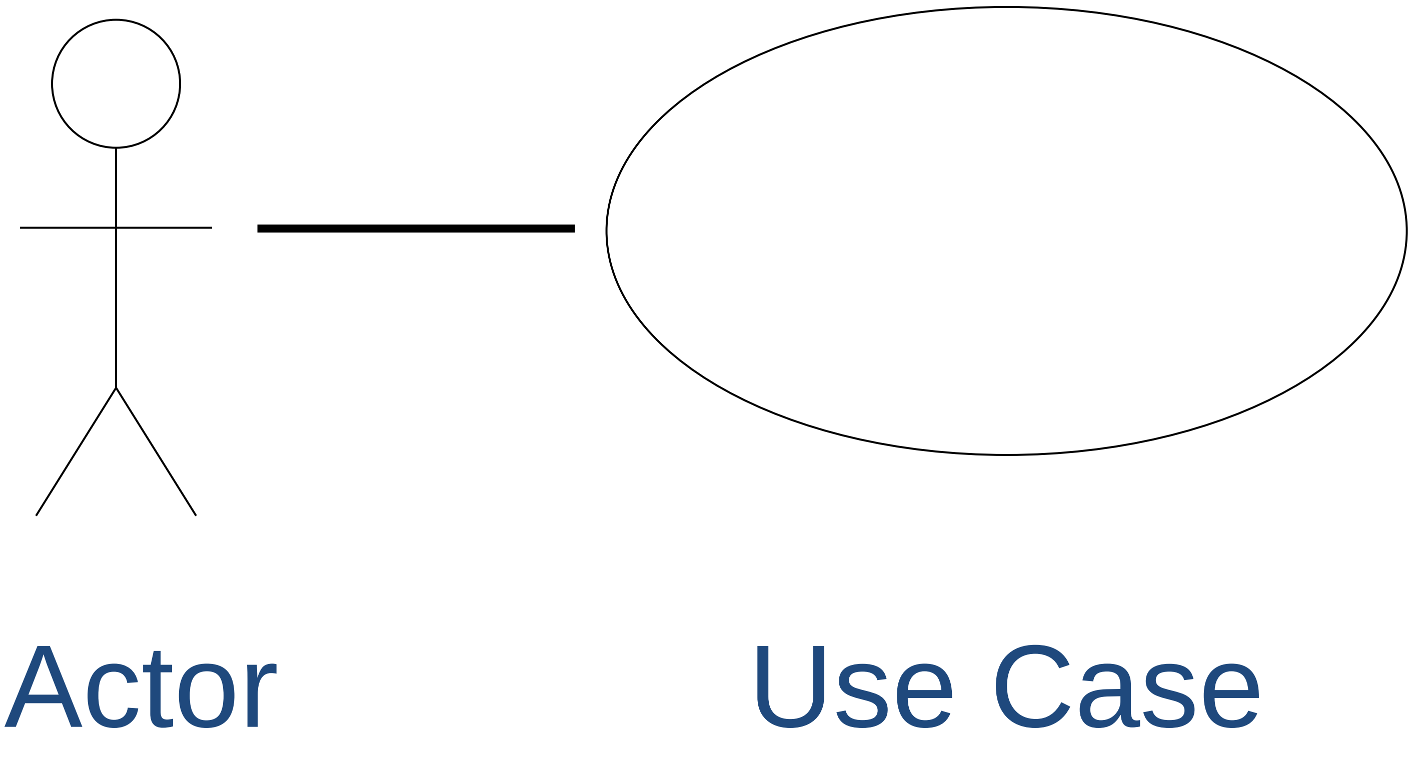
- Main purposes of the technique are:
  - To provide high level view of what the system does and who uses it
    - Model of what is happening in the system
    - Model of actions that are carried by the system that is observed from outside of the system by actor
    - Model of alternative scenarios for specific use cases that can result in different sequences of actions
  - To provide the basis for determining the interfaces of the system
  - Used in design and testing
  - Easier to write user manual

Use Case Modelling- USERS perspective

– Use cases specify the functionality of the system from users’ perspective
– Use cases are used to document the scope of the system
– Use cases show developer’s understanding of users requirements

USE CASE – FIRST VIEW

Use case is related to a user (actor)



Use Case Notation

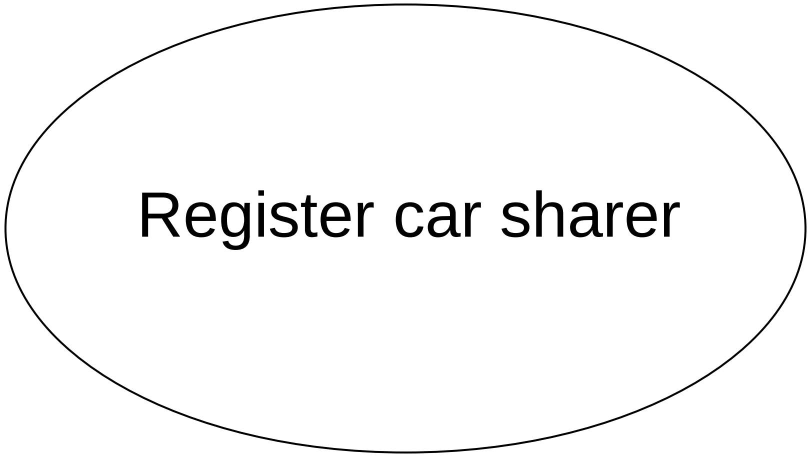
- *use case* drawn as ellipse
- *use case* name drawn in or beneath ellipse
- *use case* name is text string of letters, numbers, punctuation (except colon)
- *use case* name is:

  <u>active verb</u>

  (upper case initial letter)

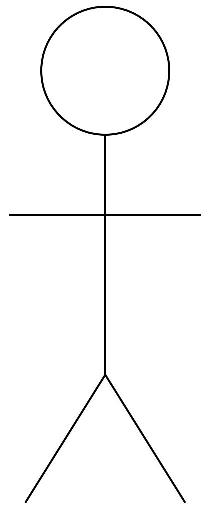
  followed by

  <u>noun phrase</u>



Actor

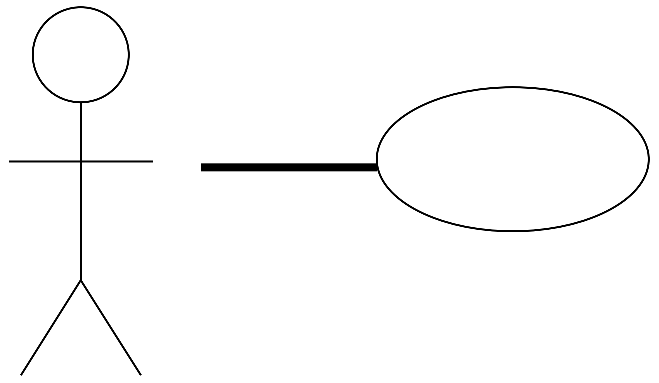
- an *actor* is a person or a system that interacts with a use case
- *actors* are generally the users of the system being modelled
- the *actors* generally define the *boundary* of the system
- an actor is represented as a stick person with the name of the role underneath; it is the ***role*** that is being represented



System Administrator

Relationship

- a line is drawn between the actor and the use case to represent a relationship
- a single actor may be associated with more than one use case
- a single use case may be associated with more than one actor



ACTORS – USE CASES

- Let U stands for a set of the identified use cases and A for a set of dedicated users of a system S.
- Then the relation “actor – use case” is defined as a product of A and S

$$AxU = \{<x,y>|\ x\in A, y\in U\}$$

Behaviour Specification

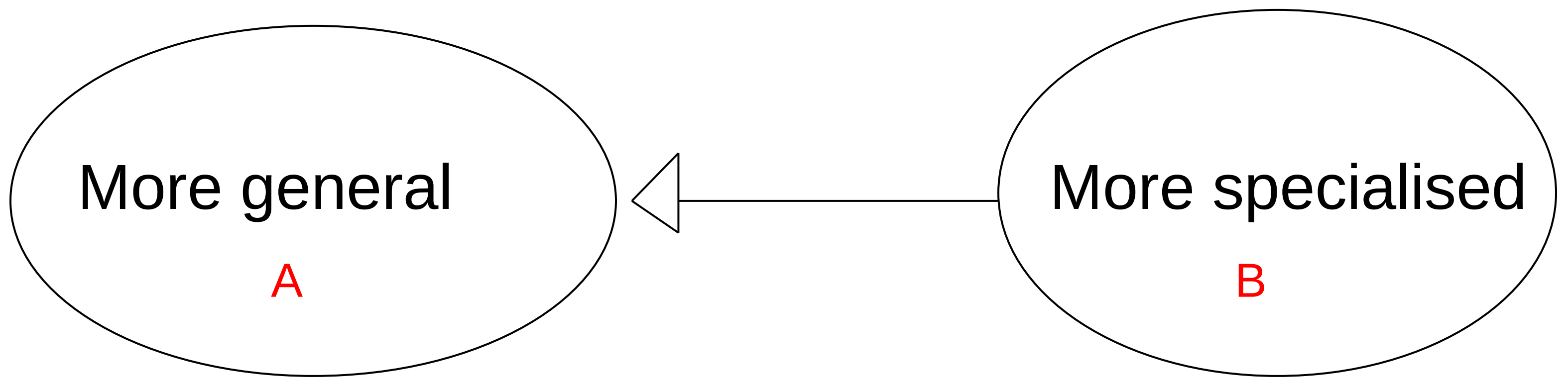
- a use case represents a sequence of activities
- these activities have some outcome
- the outcome is observable to the related actor(s)
- the sequence of activities is written up as a behaviour specification (many approaches to this)

Relationships

- generalisation between use cases
- generalisation between actors
- *include* relationship between use cases
- *extend* relationship between use cases

Use Case Generalisation

UML Notation



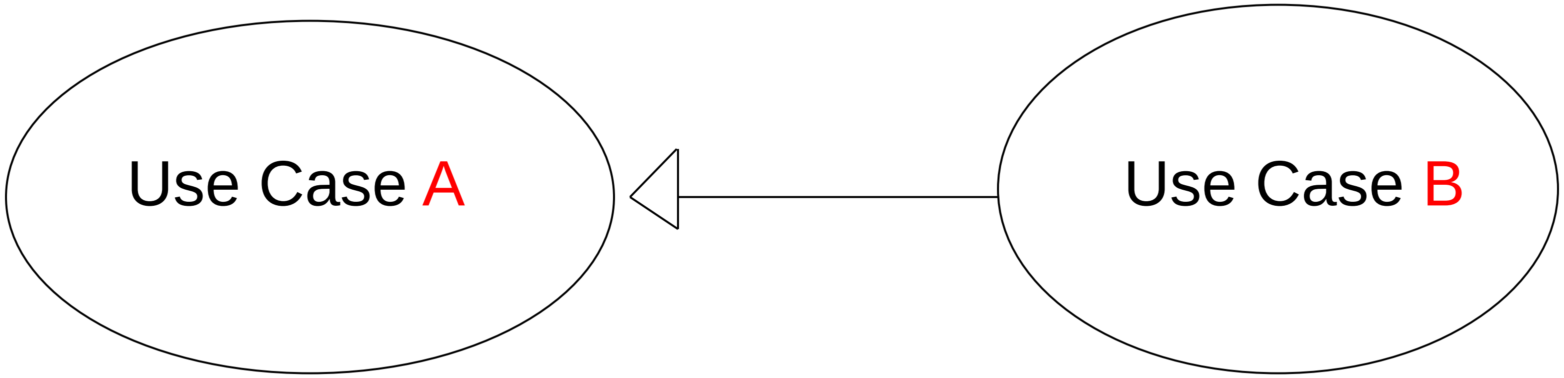
Set Theoretic Background

Use case B is more specialised than Use case A (or A is more general than B): all actions in A are in B but B has something extra

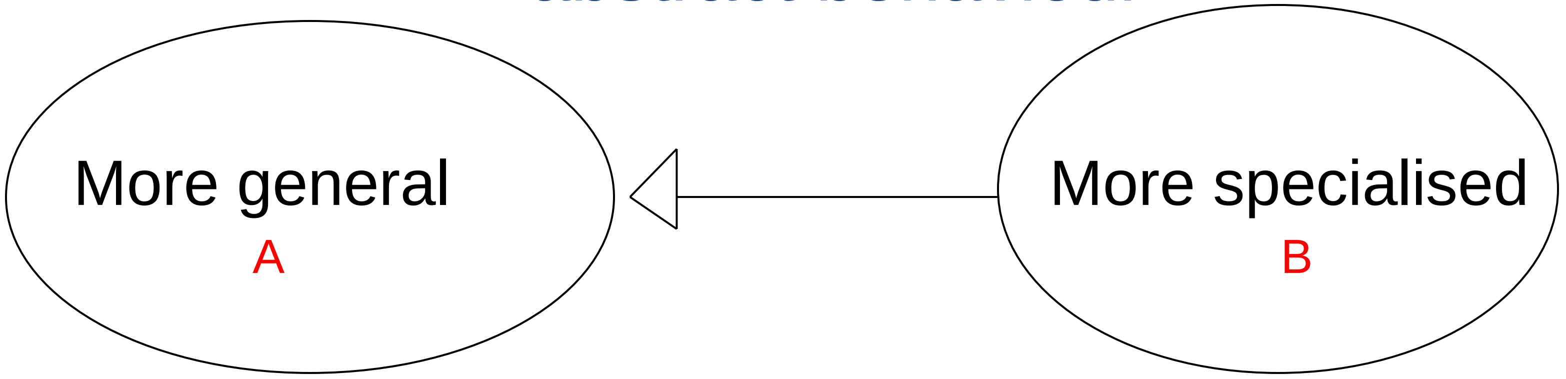
$A \subset B$

Example

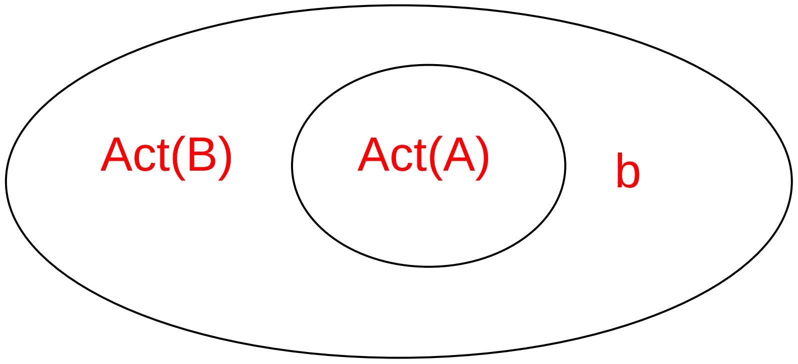
- Use Case *B* has all the actions of Use Case *A*, plus some other things.
- Let Act(A)={a|a is action of use case A} be actions of use case A and Act(B)={b|b is action of use case B} be actions of use case B.
- For any action $a \in \mathrm{Act}(A)$ we also have $a \in \mathrm{Act}(B)$ and there is an action $b \in \mathrm{Act}(B)$ such that $b \notin \mathrm{Act}(A)$
- *Use Case B is more specialised than Use Case A*
- *Use Case A is more general than Use Case B*



A more general use case describes more abstract behaviour



Set Theoretic Background. Act(A)$\subset$Act(B)



b is a new action

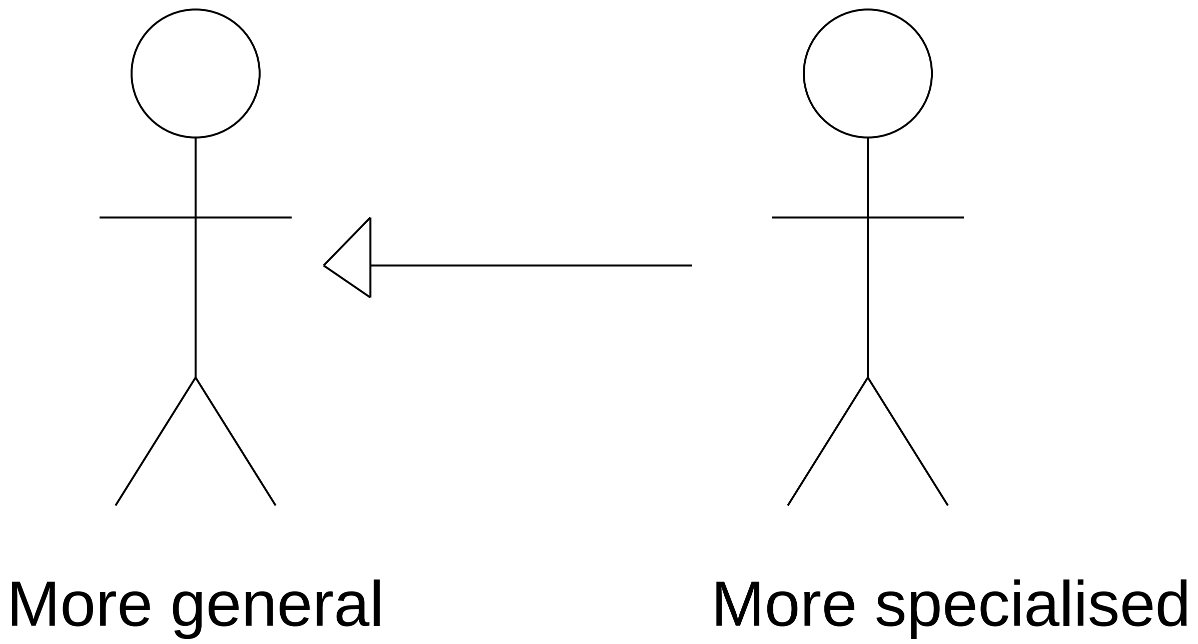
$b \in Act(B)$

$b \notin Act(A)$

Use case generalisation: Example- and importances

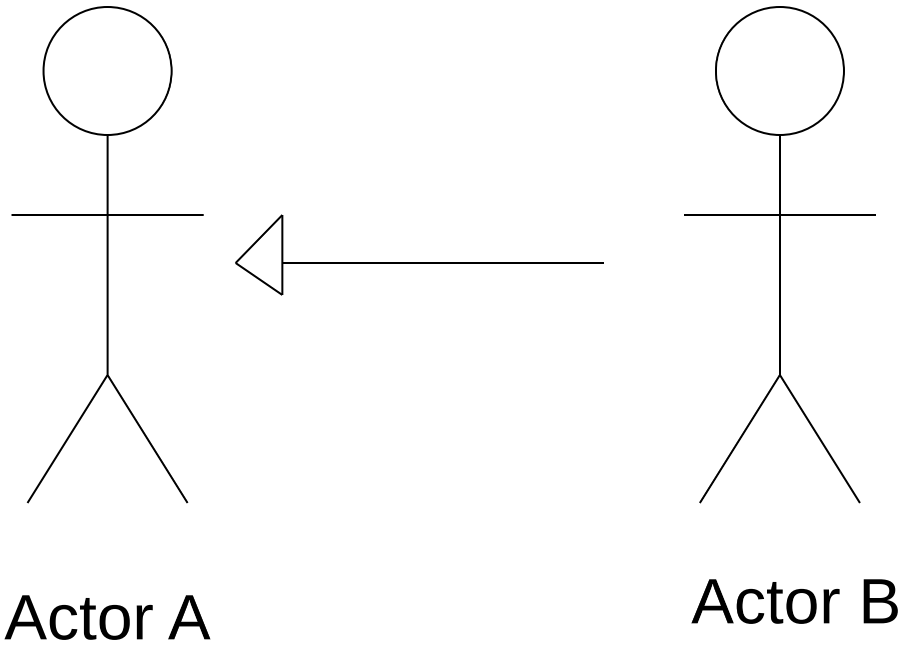
- Consider an electricity account: the “Pay Bills” use case can be generalized to “Pay by Credit Card”, “Pay by Bank Balance” etc.
- We often carry out modelling by going from “general” to “more specific” – getting more details, more knowledge on the problem/solution, more requirements…#
- Ultimately, we want to present the most details Use Cases, but more general are useful to communicate more general ideas on the design

Actor Generalisation



Actor Generalisation

- Actor B can do everything that Actor A does, plus some other things.
- *Actor* B *is more specialised than Actor* A
- *Actor* A *is more general than Actor* B



Legal Relationships

- Association between actors and use cases
- generalisation between use cases
- generalisation between actors
- *include* relationship between use cases
- *extend* relationship between use cases

Relation between Use Cases

- There are two other kinds of relationships between use cases: include and extend.
- «include» is used when a chunk of behavior is similar across more than one use case, and you don't want to keep copying the description of that behaviour.
- So we break out re-used functionality in a program into its own methods that other methods invoke for the functionality.
- For example, suppose many actions of a system require the user to login to the system before the functionality can be performed. These use cases would *include* the login use case.

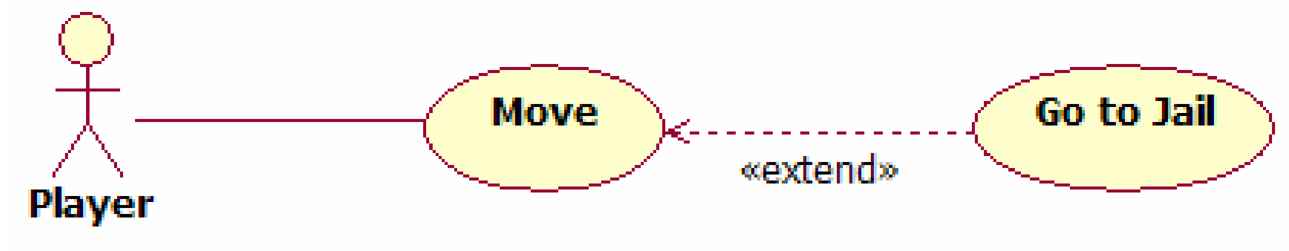
- You use the «extend» relationship when you are describing a variation on normal behavior or behaviour that is only executed under certain, stated conditions.
- The extend relationship is used when the alternative flow is fairly complex and/or multi-stepped, possibly with its own sub-flows and alternative flows.

Extends Use Case

For example, consider the players moving on a Monopoly board.

A player moves on the board because he or she has to go to jail or to go to Free Parking.

This scenario involves a player moving. We can extend the Move use case with the “Go to Jail” and the “Go to Free Parking” use case



“Include” and “Extend” again

- It is common to be confused as to whether to use the include relationship or the extend relationship.
- Consider the following distinctions between the two:

• Use Case X includes Use Case Y: X has a multi-step subtask Y. In the course of doing X or a subtask of X, Y will **always** be completed.

• Use Case X extends Use Case Y: Y performs a sub-task and X is a similar but more specialized way of accomplishing that subtask (e.g. closing the door is a sub-task of Y; X provides a means for closing a blocked door with a few extra steps). X **only happens in an exception situation**. Y can complete without X ever happening.

“INCLUDE”

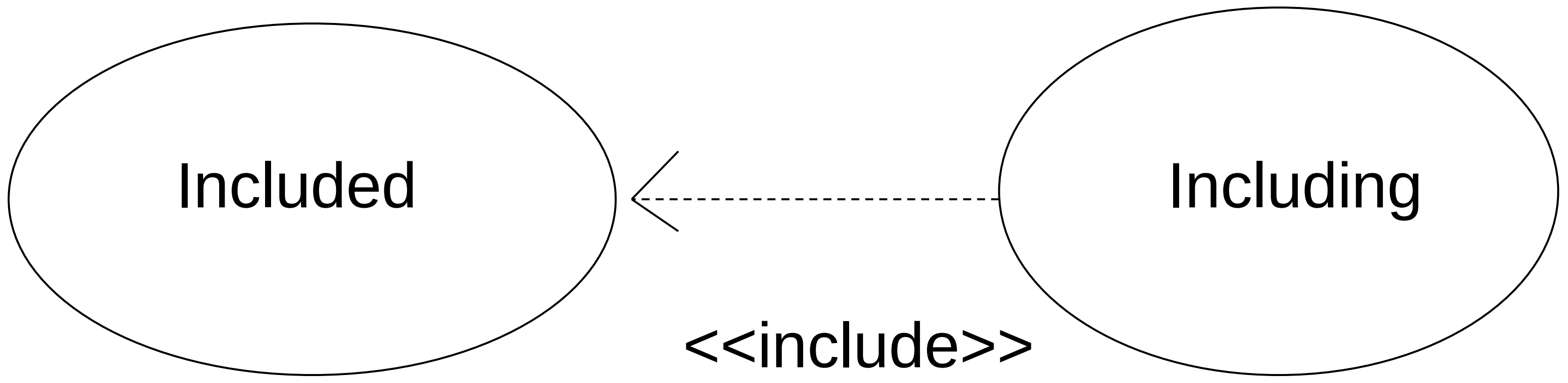
- The include relationship signifies that one use class is included in another’s functionality.

- You use the include relationship when a chunk of behavior is similar across more than one use case and you don’t want to keep copying the description of that behavior. This is similar to breaking out re-used functionality in a program into its own methods that other methods invoke for that functionality.

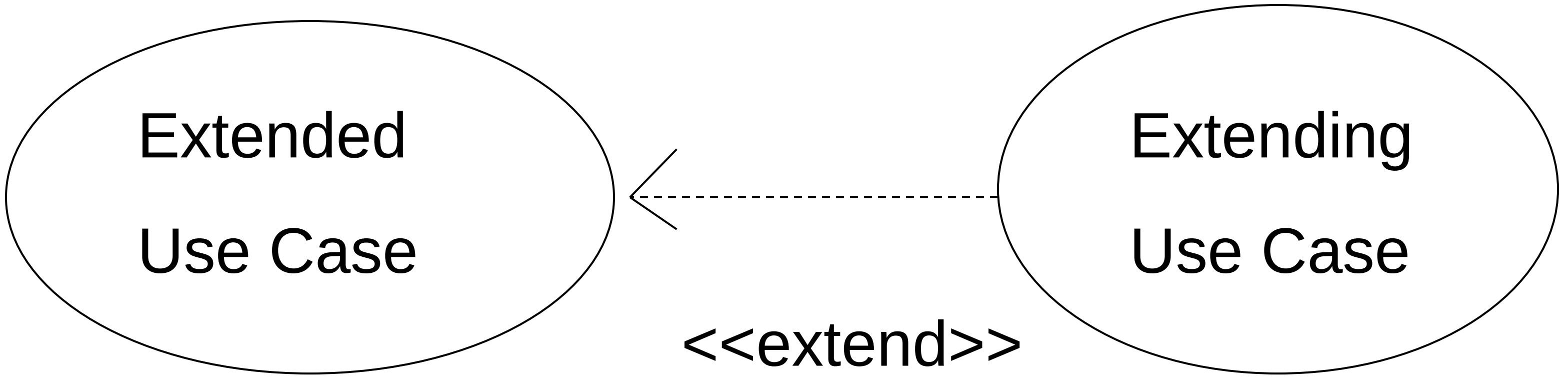
- For example, suppose many actions of a system require the user to log into the system before the functionality can be performed. These use cases would *include* the Login use case.

- HINT. *You should not break out a use case to be included by other use cases unless more than one other use case will include it (i.e. in a case diagram there should be more than one arrow coming into the included use case).*

Include Relationship



Extend Relationship



How to produce Use Cases

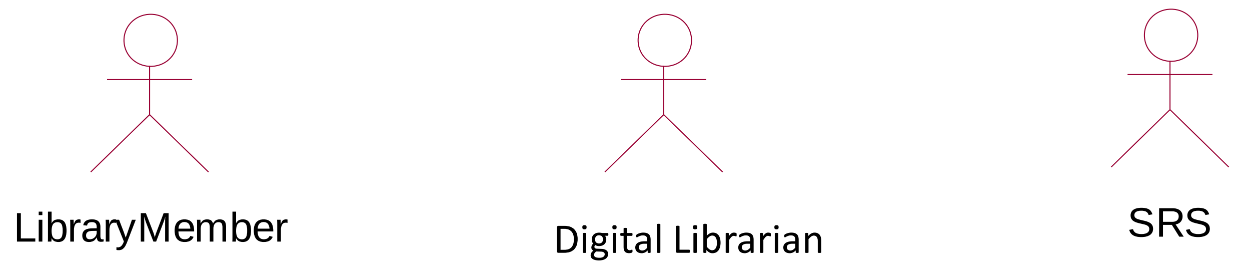
- Identify Actors
- Identify Use Cases
- Prioritise Use Cases
- Develop Use Cases in order of priority
- Structure the Use Case Model

Exercise: Library System Example

- Consider the Library as a system
  - What are the main actors?
  - What are the main use cases?
  - Which actors should interact with which use case?

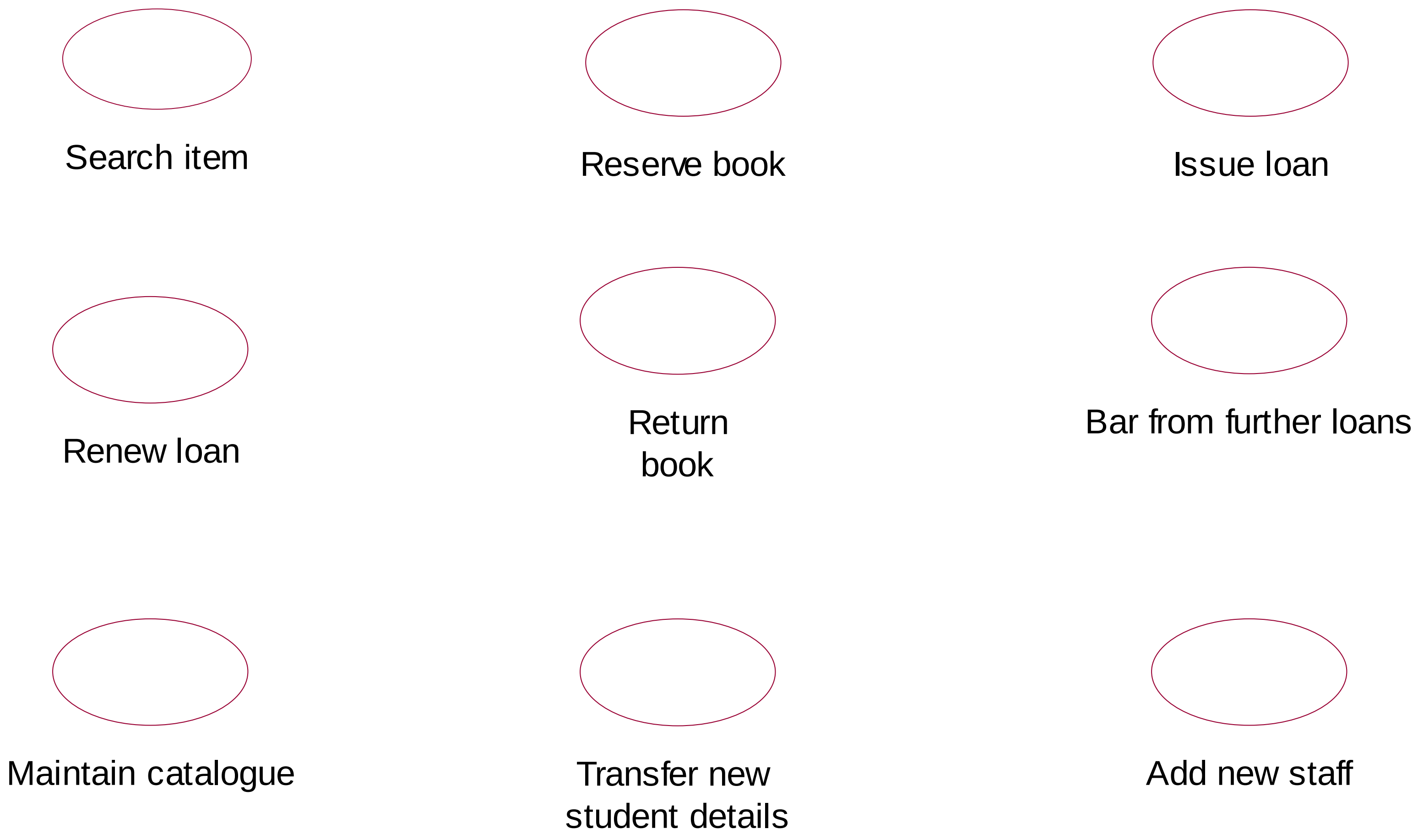
Main Actors

- Note that staff can also borrow books, videos etc. from library as well as students
- Assume that all students and STAFF become members automatically i.e. their details are transferred from the relevant record systems.

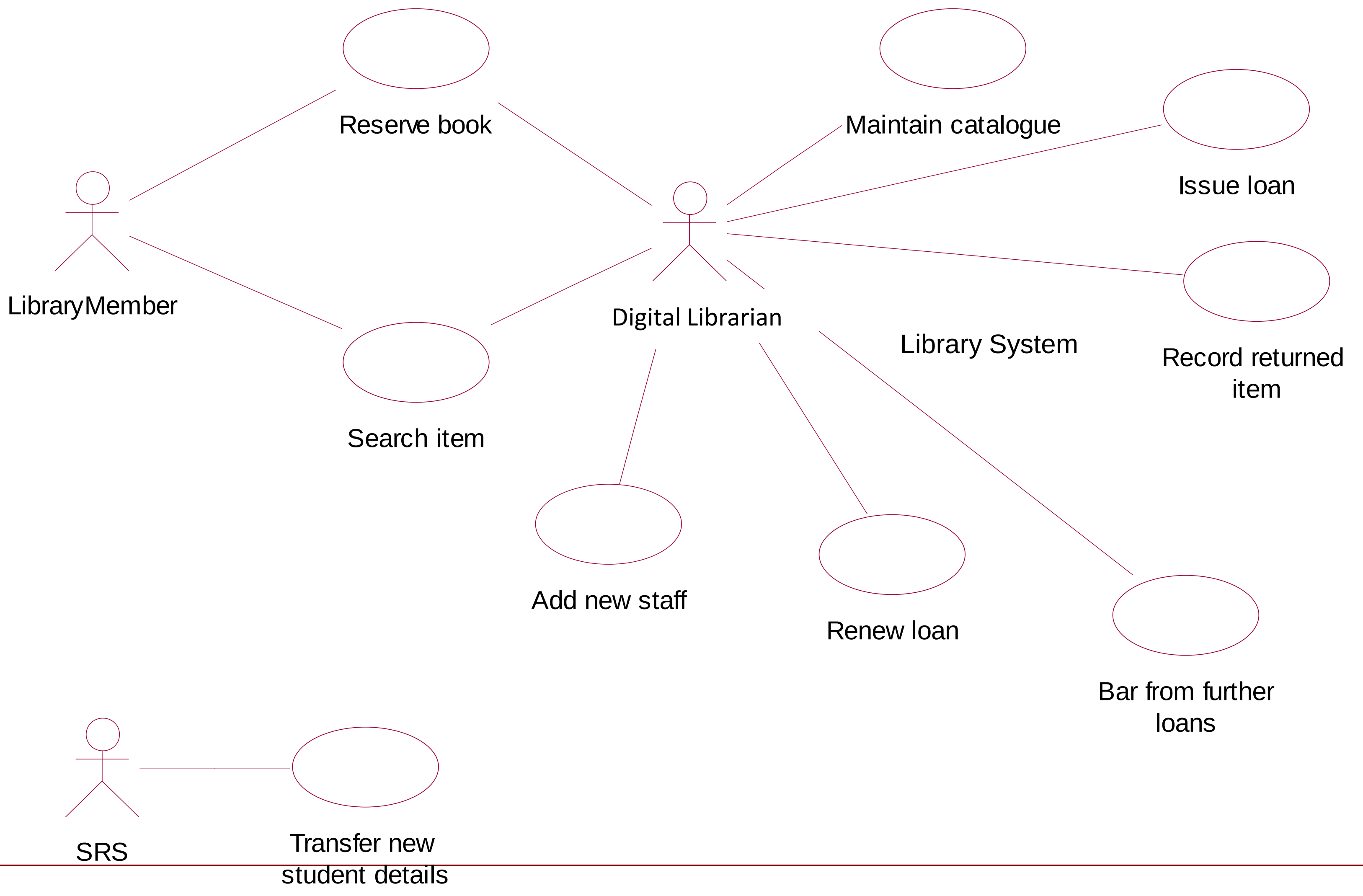


- Notice that the Actor is not given a specific name like ‘Student’. However a student could play the role of ‘LibraryMember’. So could a university staff.

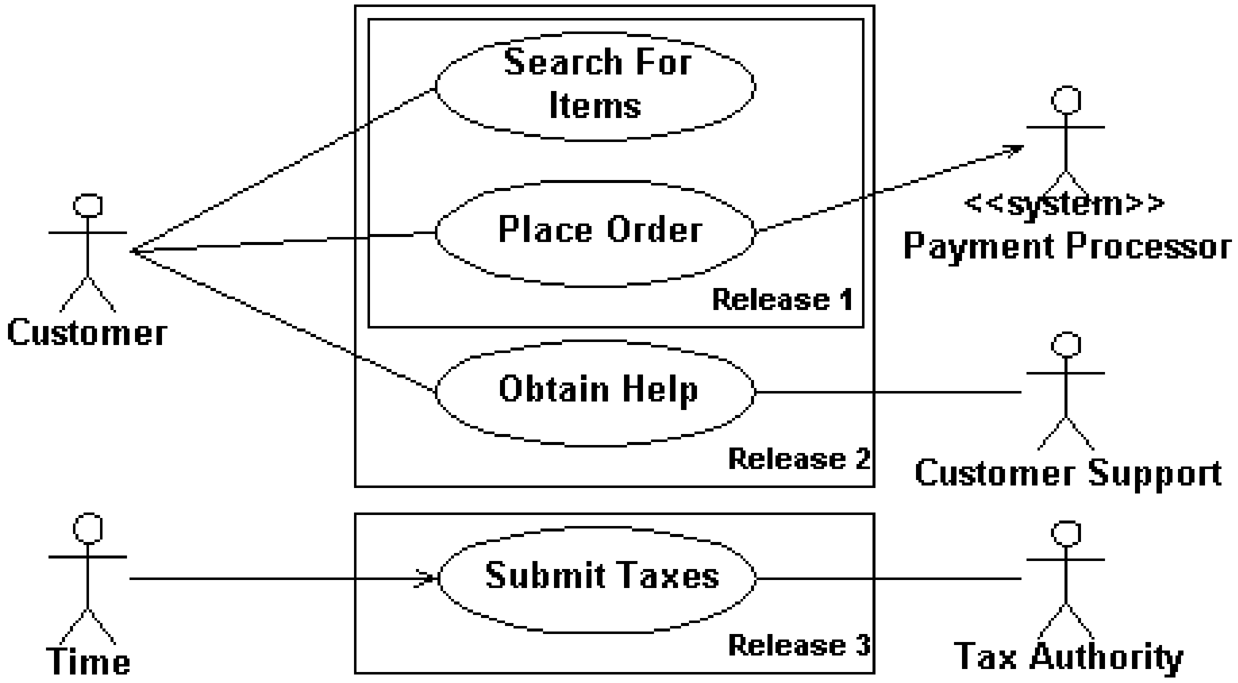
Main Use Cases



Actor-Use Case Interaction



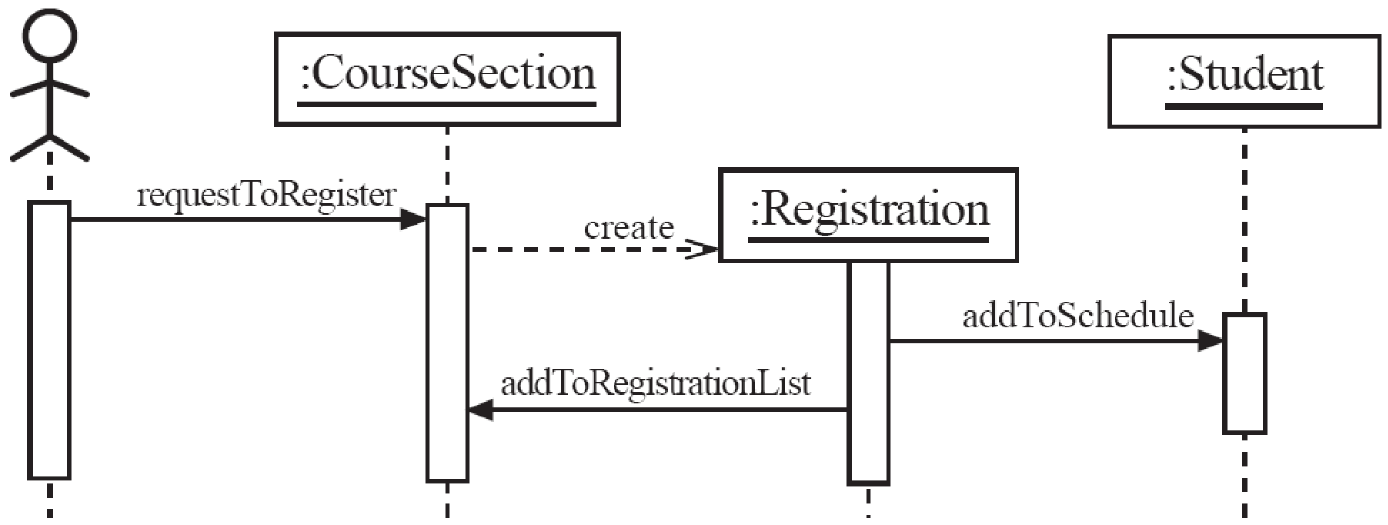
USE CASE 2 – SHOPPING EXAMPLE

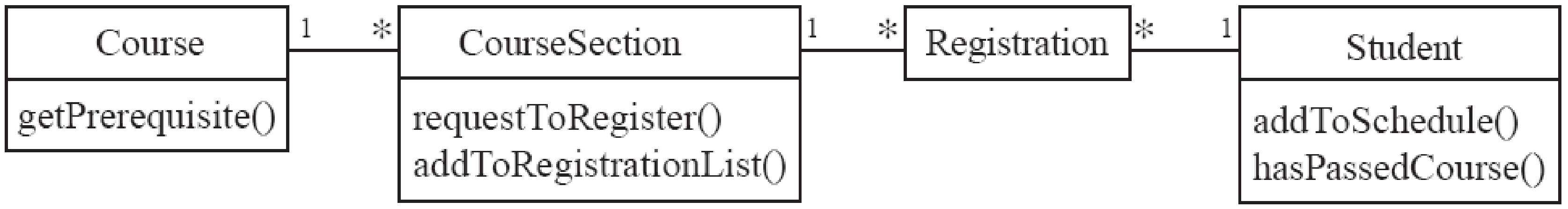


Sequence diagrams

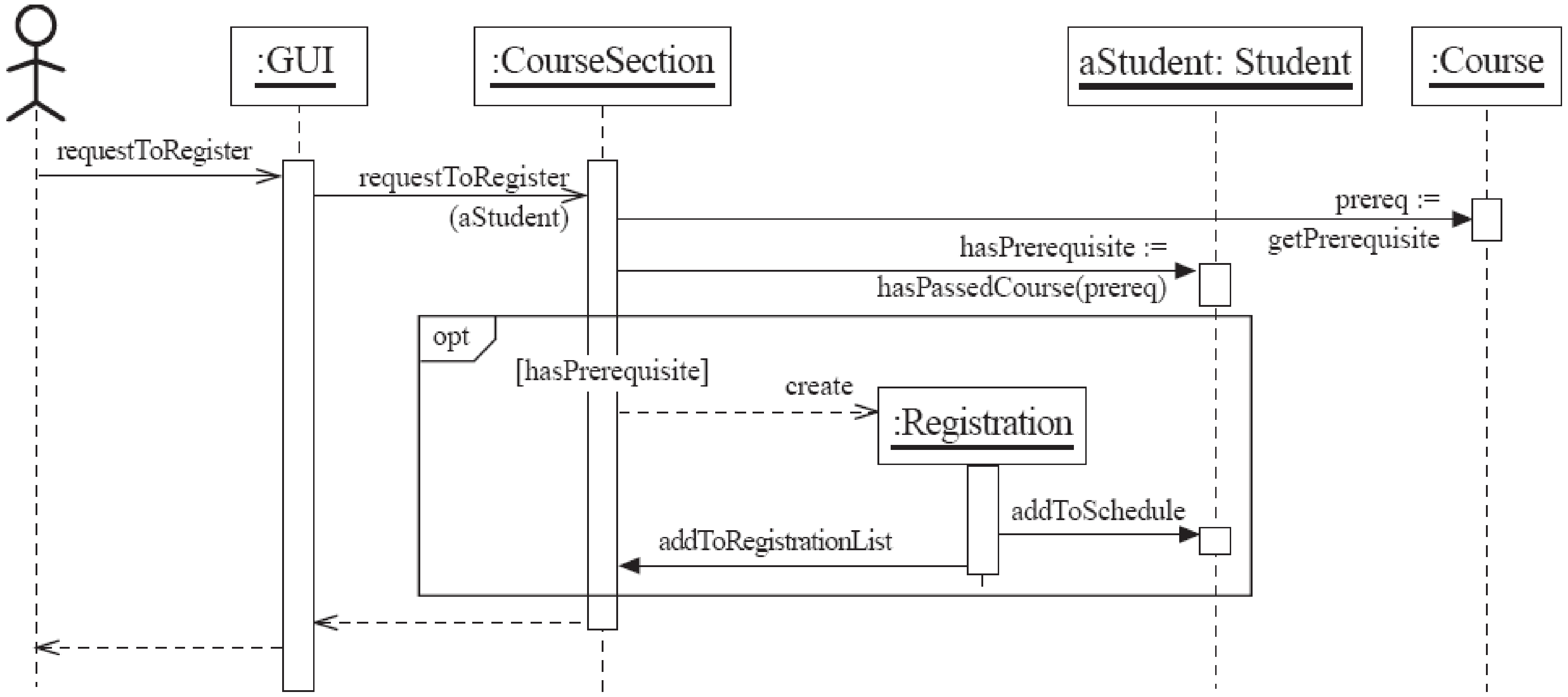
- A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task
  - The objects are arranged horizontally across the diagram.
  - An actor that initiates the interaction is often shown on the left.
  - The vertical dimension represents time.
  - A vertical line, called a *lifeline*, is attached to each object or actor.
  - The lifeline becomes a broad box, called an *activation box* during the *live activation* period.
  - A message is represented as an arrow between activation boxes of the sender and receiver.
    - A message is labelled and can have an argument list and a return value.

Sequence diagrams – an example



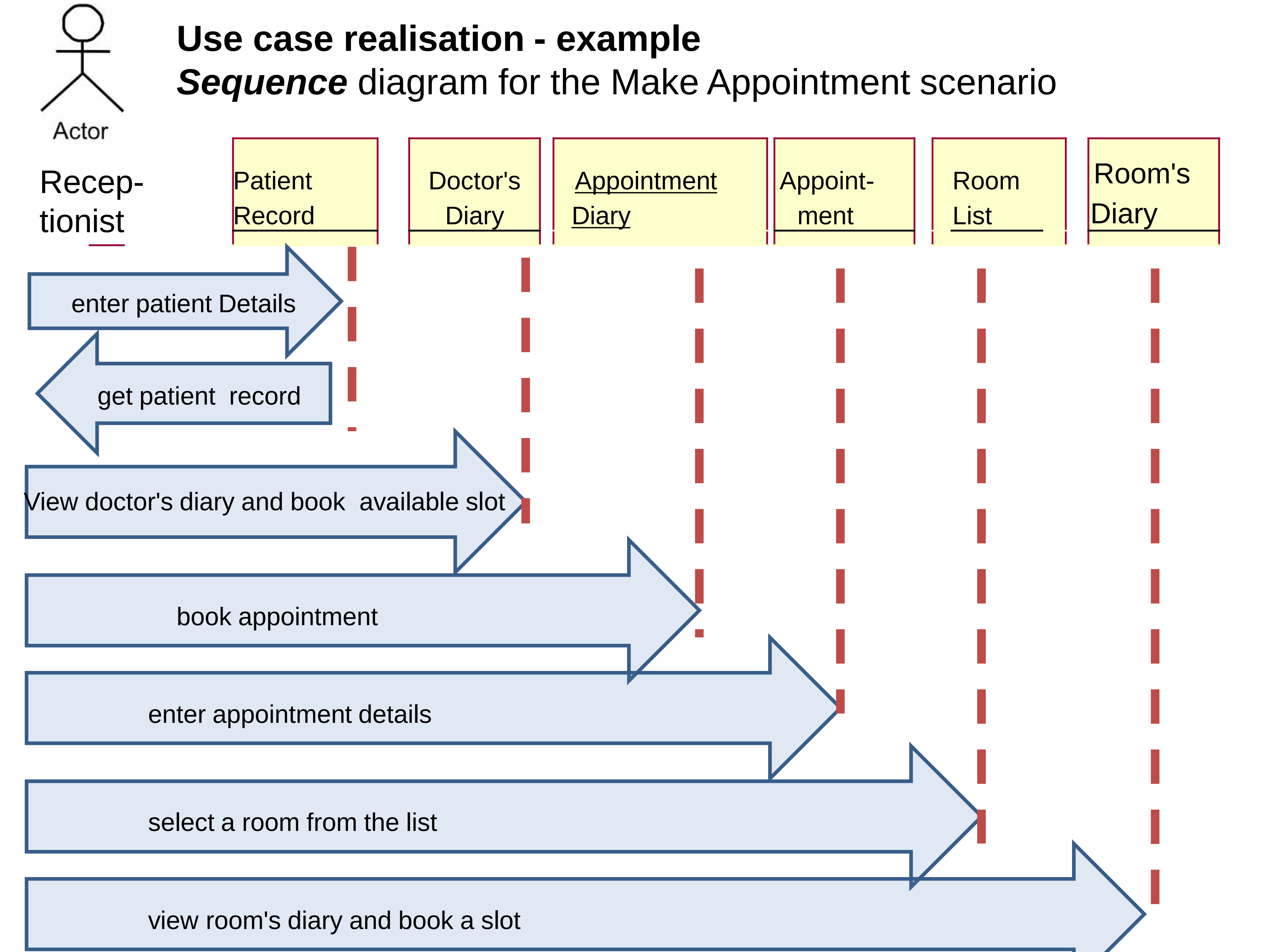


Sequence diagrams – same example, more details



Use case realisation - example

Sequence diagram for the Make Appointment scenario



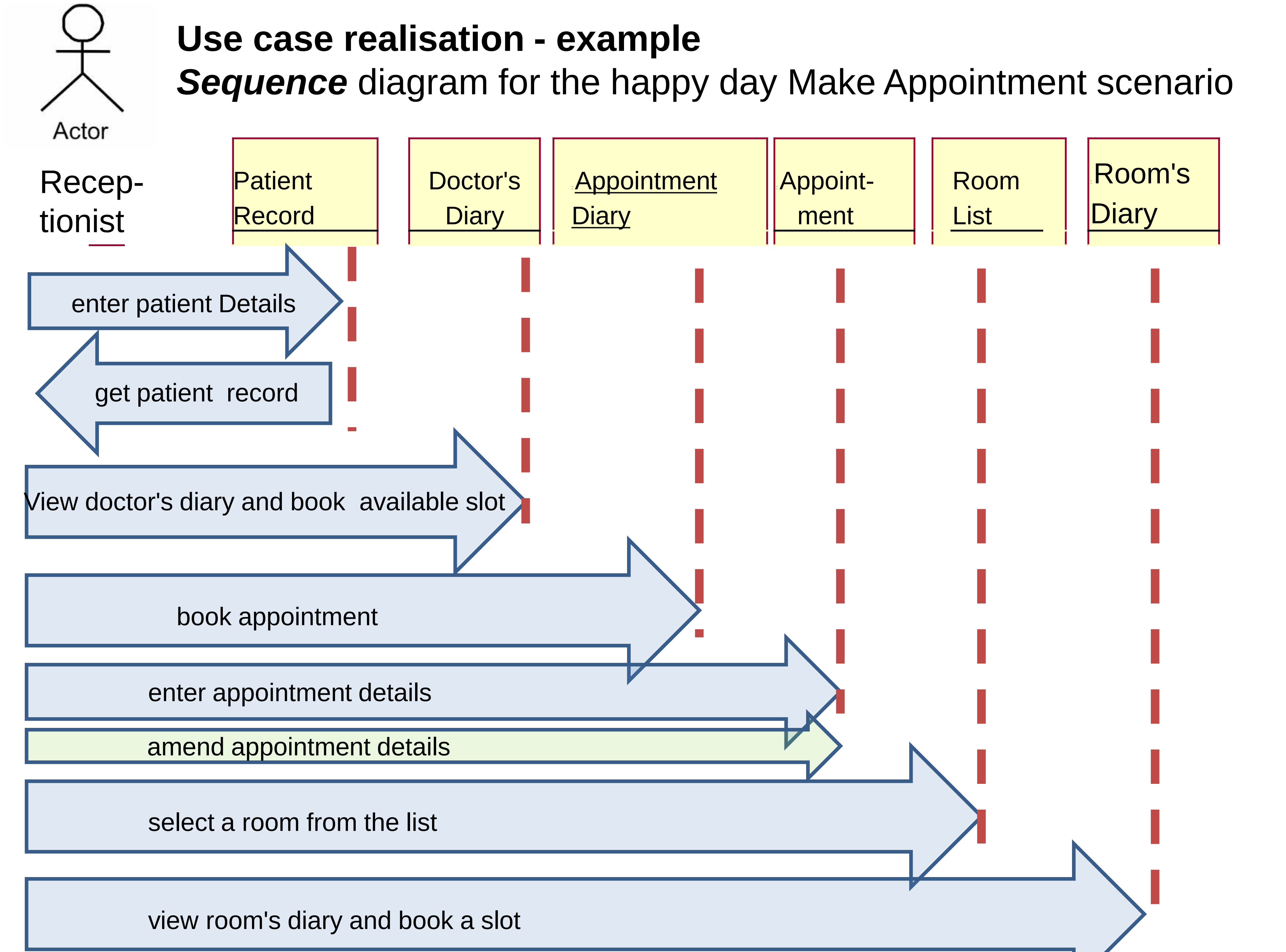
Use case realisation - example

Sequence diagram for the Make Appointment scenario

- Consider Sequence Diagrams to check your Use Case
- What about changing the appointment details because of the room features
- What about cancelling?
- This is tackled in “amending” scenario

Use case realisation - example

Sequence diagram for the happy day Make Appointment scenario



Actor
Recep-tionist
Patient Record
Doctor's Diary
Appointment Diary
Appoint-ment
Room List
Room's Diary
enter patient Details
get patient record
View doctor's diary and book available slot
book appointment
enter appointment details
amend appointment details
select a room from the list
view room's diary and book a slot

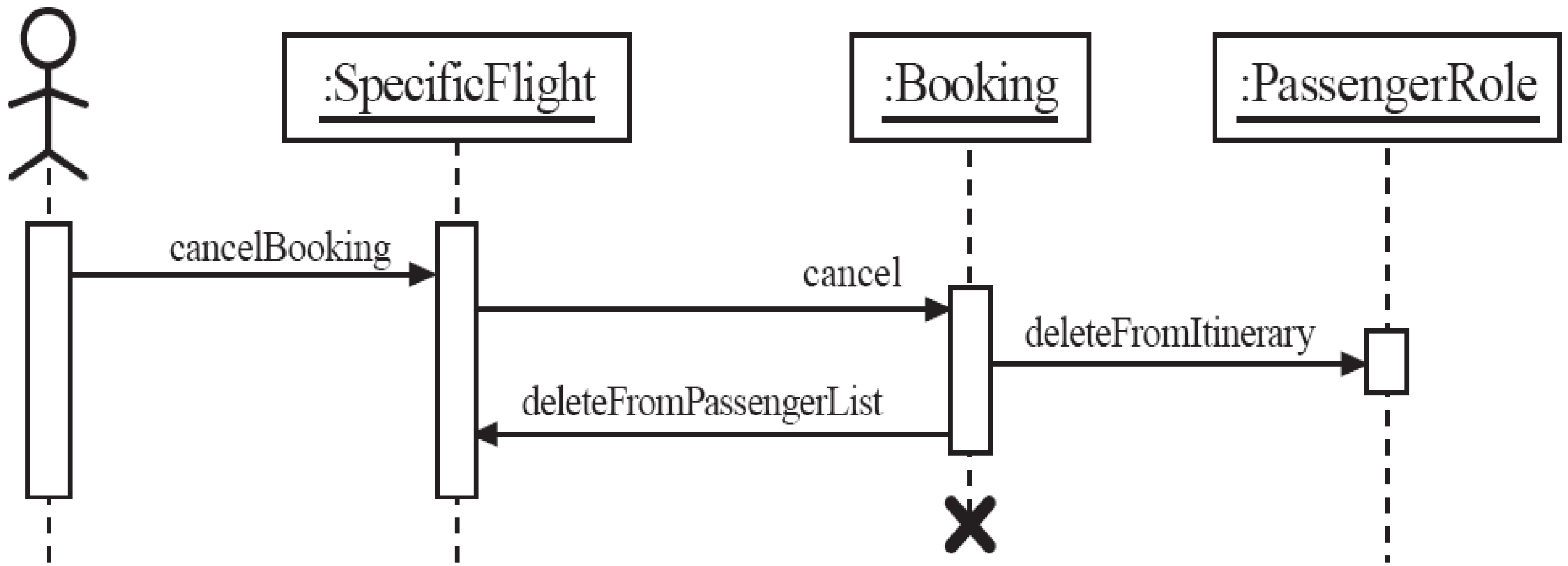
Use case realisation - example

Sequence diagram for the happy day Make Appointment scenario

- Consider again your Sequence Diagrams to check your Use Case
- Is anybody else involved into the scheme?
- What about nurses?
- What about cancelling the appointment?

Sequence diagrams – an example with object deletion

- If an object's life ends, this is shown with an X at the end of the lifeline



A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies.

The main constituents of a class diagram are classes and
their relationships: generalization, aggregation, association, and various kinds of dependencies.

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written
using mixed case convention and begins with an uppercase. The class names are usually chosen to be singular nouns

KEYWORDS

- DATA, DATA MANAGEMENT
- COMPUTATION – DATA TRANSFORMATION
- ANALYSIS, DESIGN, MODELLING

EXAMPLES
DATA

- In Programming we start with the ***data declaration***:
  - int a, b, c; // Declares three integers a, b, and c.
  - int a = 10, b = 10; // Example of initialisation
  - double pi = 3.14159; // declares and assigns a value of PI.
  - char a = 'a'; // the character variable a is initialized with value 'a'

EXAMPLES DATA TRANSFORMATIONS

-
  ```
  public static int add(int a, int b)
   {
    return a + b;
   }
  ```
- In JAVA we define ***methods (functions)*** that transform data
- add(int a, int b) is a method which transforms two input integers into an integer, defined as the SUM of a and b as it returns a + b

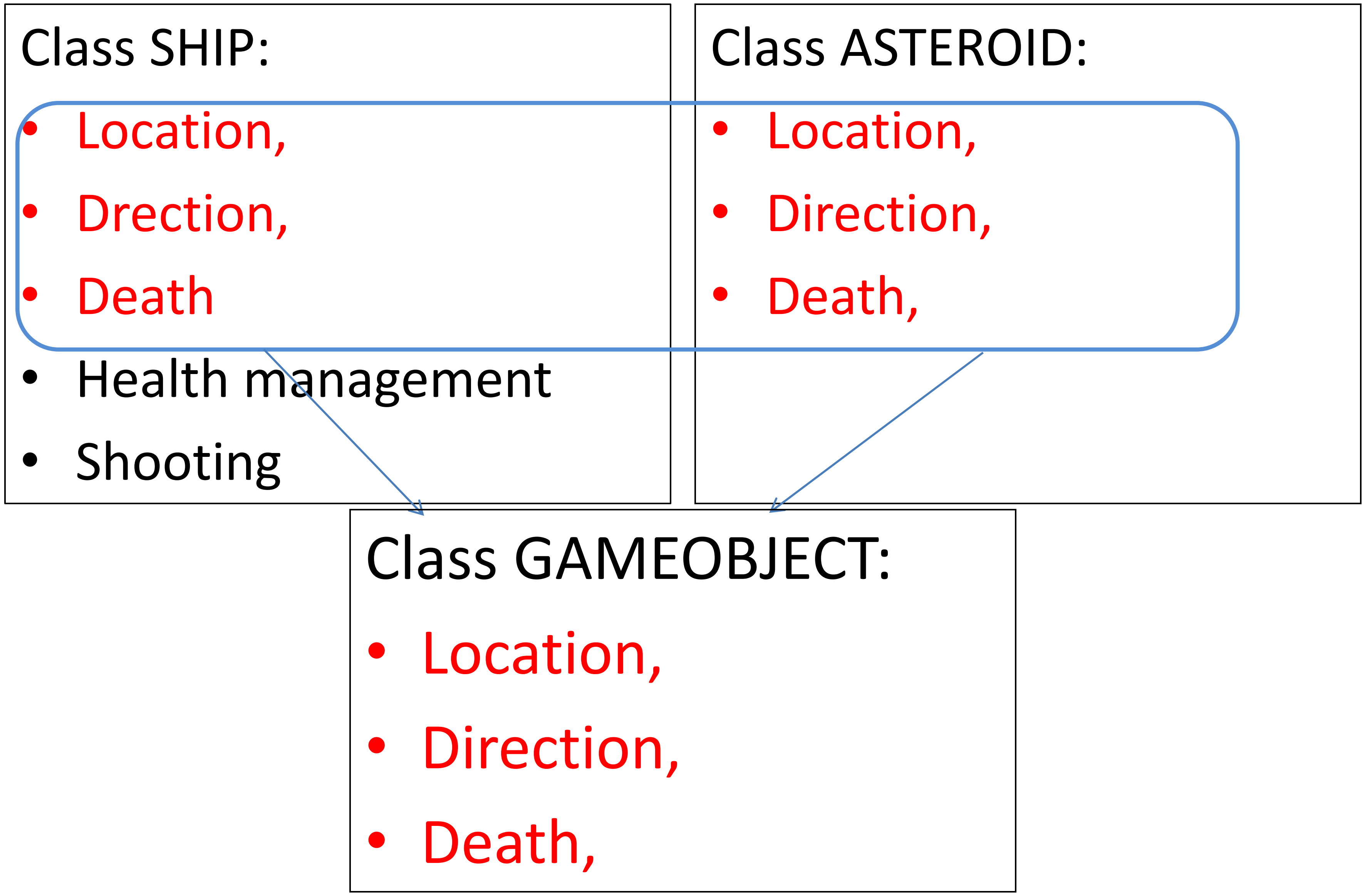
OBJECTS and CLASSES

- Object-Oriented Framework
- Often it is useful to observe that several objects behave in the same fashion
- Imagine that I am programming a computer game, asteroids, for example
- So in my game I will have different objects: ships, asteroids, etc. Now let us think what do we need to know in order to describe that something is a ship or a asteroid?

OBJECTS and CLASSES

- For the SHIP we would need to know location, direction, when it would explode, how it looses and gains health, how it shoots, etc
- For the ASTEROID we would need to know location, direction, when it would explode, etc
- What we do – we create classes SHIP and ASTEROID and define in them those characteristics that are common for any ship and any asteroid

OBJECTS and CLASSES



Class SHIP:

- Location,
- Drection,
- Death
- Health management
- Shooting

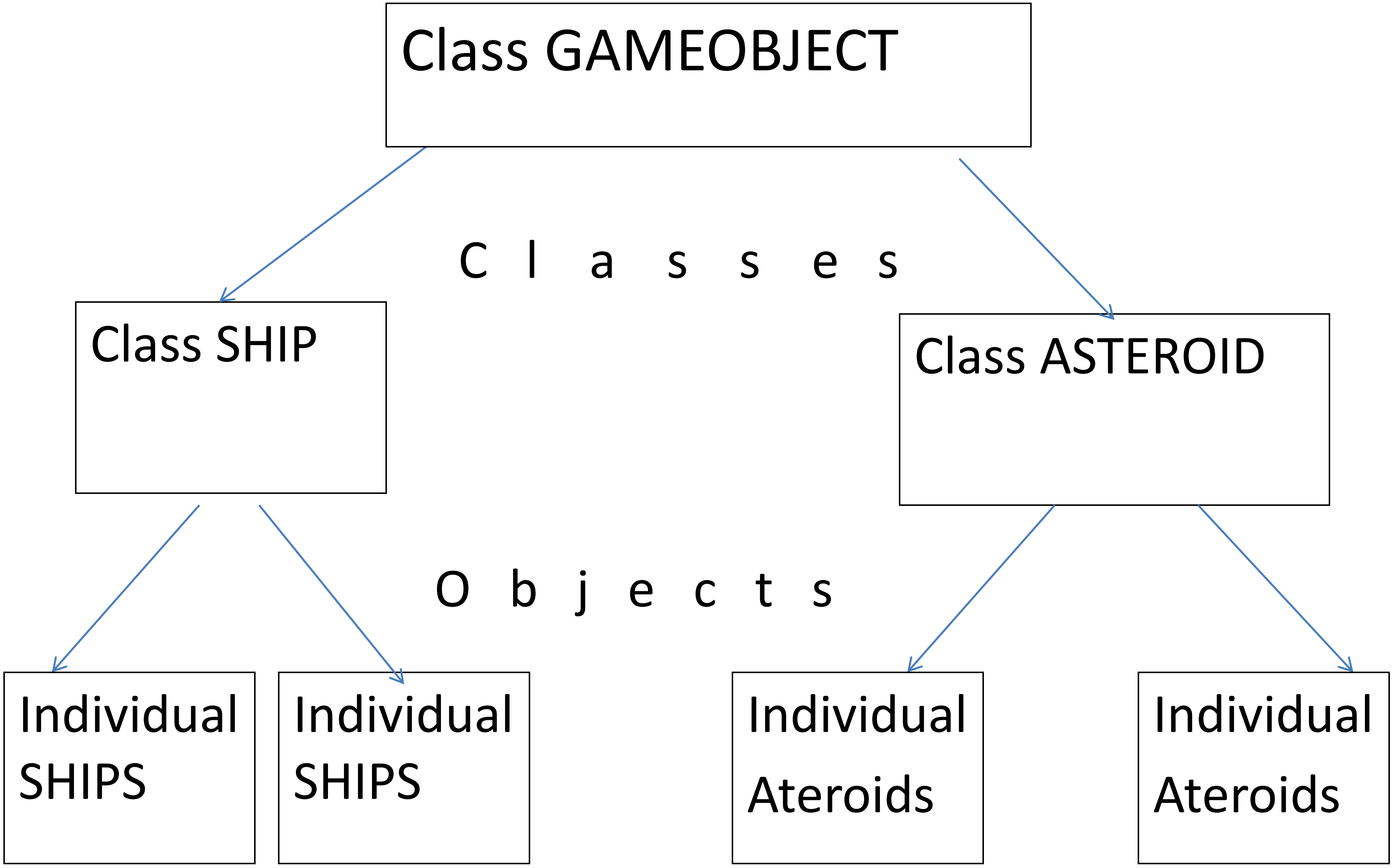
Class ASTEROID:

- Location,
- Direction,
- Death,

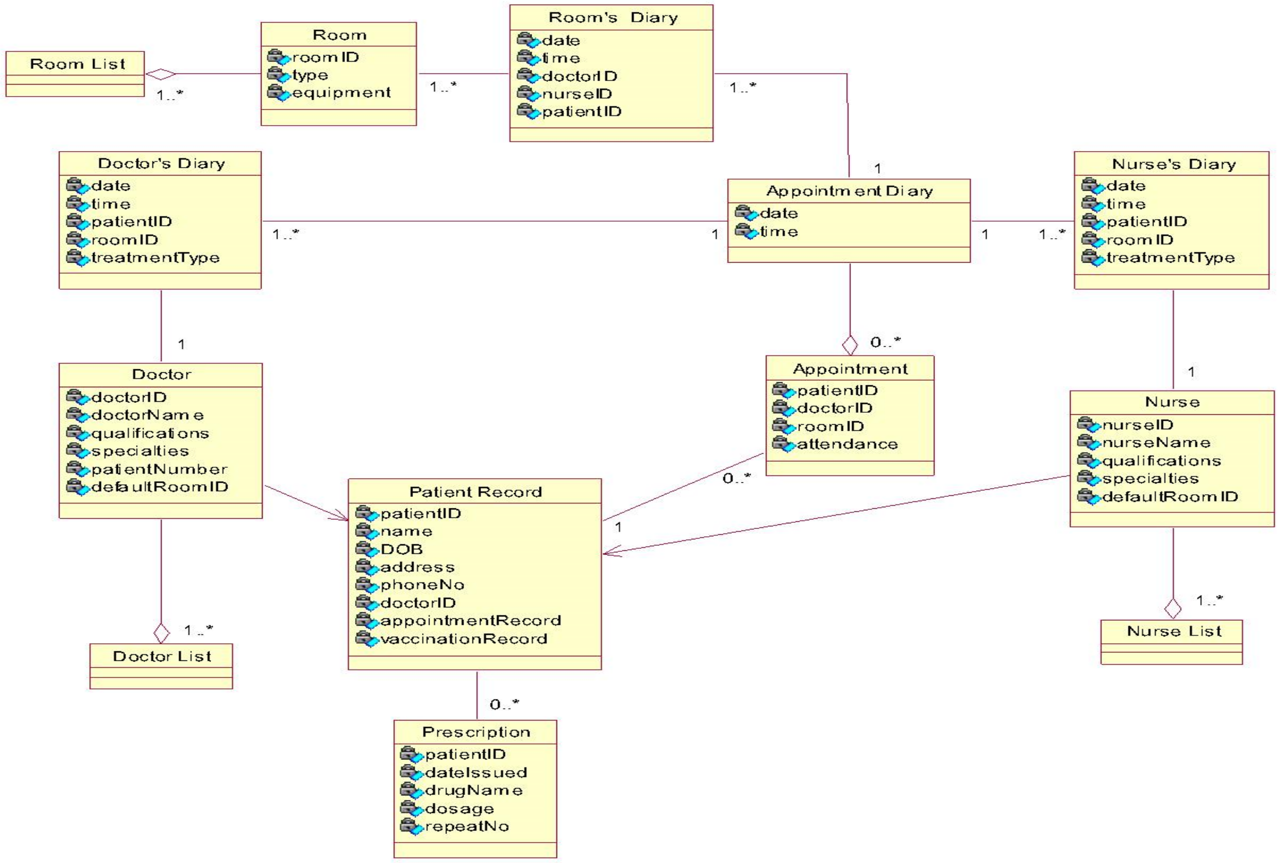
Class GAMEOBJECT:

- Location,
- Direction,
- Death,

OBJECTS and CLASSES



Class diagram for Community Health Care system at the analysis stage


Room List
1..*
Room
roomID
type
equipment
1..*
Room's Diary
date
time
doctorID
nurseID
patientID
1..*
1
Doctor's Diary
date
time
patientID
roomID
treatmentType
1..*
1
Appointment Diary
date
time
1
1..*
Nurse's Diary
date
time
patientID
roomID
treatmentType
1
0..*
1
Doctor
doctorID
doctorName
qualifications
specialties
patientNumber
defaultRoomID
Appointment
patientID
doctorID
roomID
attendance
0..*
1
Nurse
nurseID
nurseName
qualifications
specialties
defaultRoomID
Patient Record
patientID
name
DOB
address
phoneNo
doctorID
appointmentRecord
vaccinationRecord
1..*
Doctor List
1..*
Nurse List
0..*
Prescription
patientID
dateIssued
drugName
dosage
repeatNo