MUNI
FI

Use Case Diagram + System Requirements

PB007 Software Engineering I

Jakub Levčík
536336@mail.muni.cz

Project recap

- Customer: IT company Mice in Black (MIB)
- **Desktop** application (mainly for Windows)
- Target users: company workers, managers, accountants (company evidence system)
- Expectations
  - plan for future extensions: company will decide based on our work

Today‘s goals

Find out what the system requirements for the project are

Based on the requirements – create an initial use case diagram

Requirements – why first?

We need to know what is expected from us – represented by **requirements**

Functional requirements

What the system usage

- Describe and influence the system’s **functionality**
- A **functional requirement** tells you **WHAT** the system should (or should not) do
- Common format: **<id><system><function>**
  <id><who><does what>
- Examples (EasyFood recipe app)
  - 01. The EasyFood app sends a notification to the user when a competition ends
  - 02. The EasyFood app allows the user to create and manage ingredients
  - 03. The EasyFood app allows user to import and export their stored recipes

Non-functional requirements

How the system should meet functional requirements

- **Non-functional requirement** is a constraint imposed on the system
- Often related to qualitative attributes like performance, security, availability… or environment and regulations
- Can be used to further specify functional requirements
- **Testability** is a must
- Examples (EasyFood recipe app)
  - 01. The EasyFood app will be programmed in Java
  - 02. The EasyFood app will use H2 database as persistent storage.
  - 03. The EasyFood app will import/export data in asynchronous mode
- Influence system **architecture**

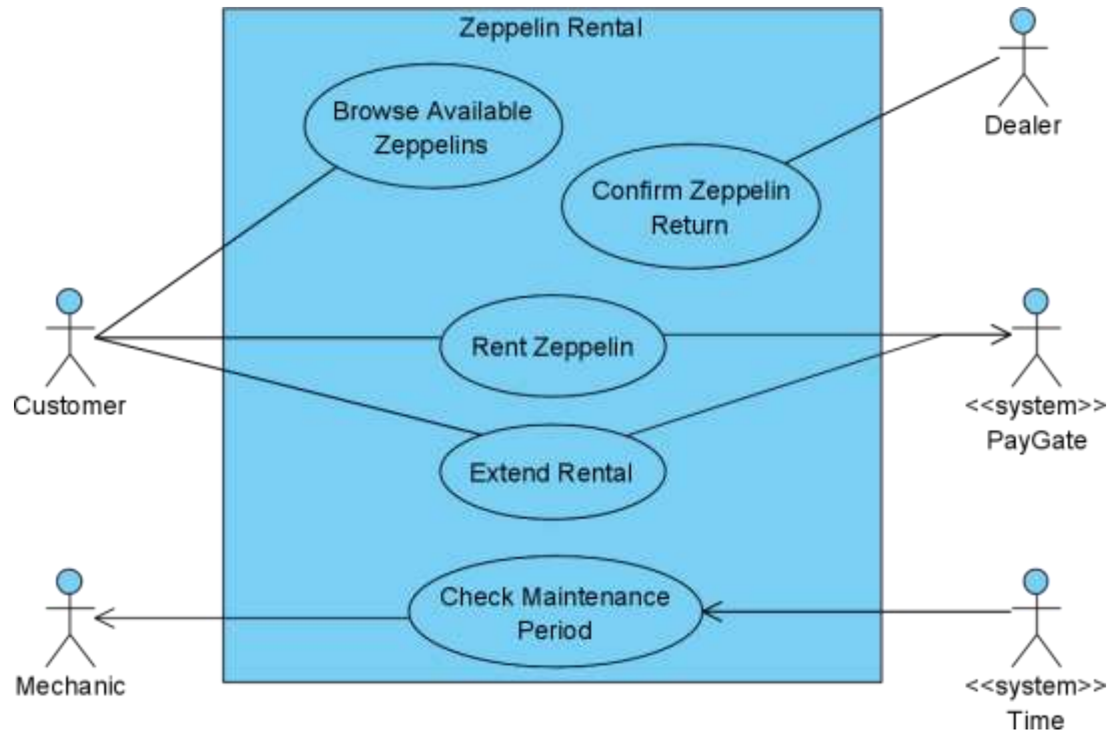
Activity: (Non)Functional requirements

Quiz time

MUNI FI

Use case diagram

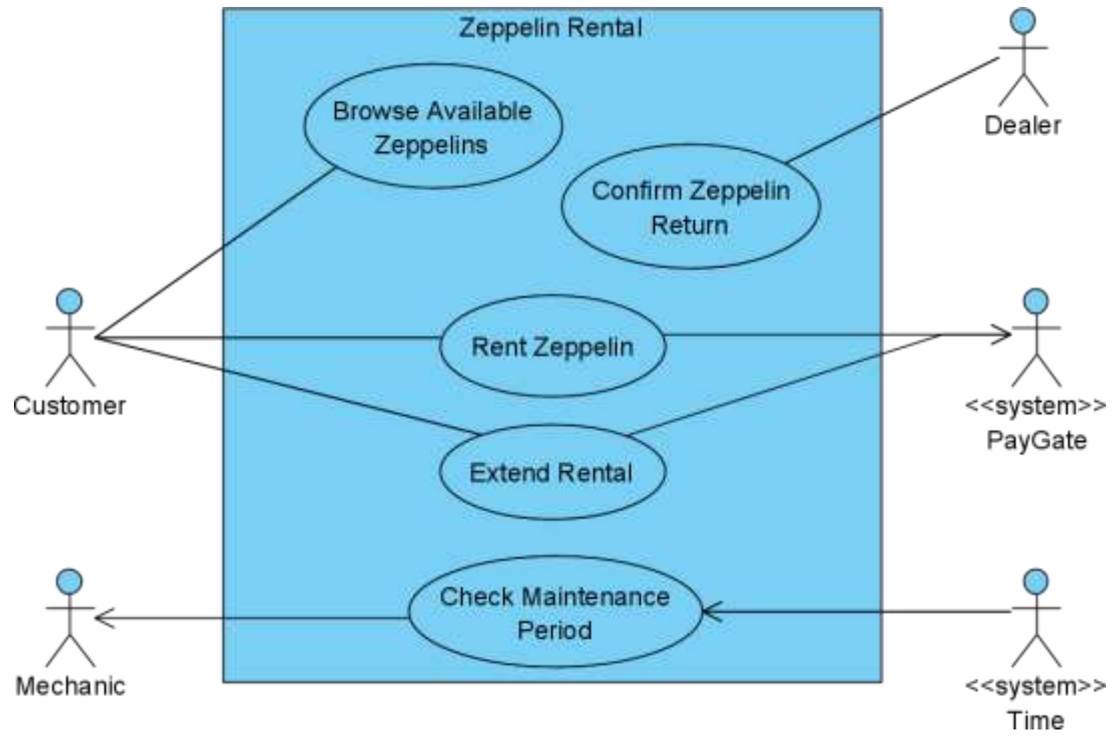
- Graphical representation of **functional** requirements
- Simple and understandable
- Consists of:
  - **System boundary + name** (Zeppelin rental)
  - **Actors** (human icons)
  - **Use cases** (ovals with verbs)
  - **Relationships** (lines/arrows)



Use case diagram

Actor

- A **role** representing an external entity
  - External with respect to system
  - Communicates directly with the system
  - Not a single person
  - A specific person can act as multiple actors, which could change over time
  - Can be also another **system**, **time**…
  - **Primary actor** (triggers an action, "active") vs. **secondary actor** (becomes involved without triggering an action, "passive")
  - Must have a **clear** name, should have a description

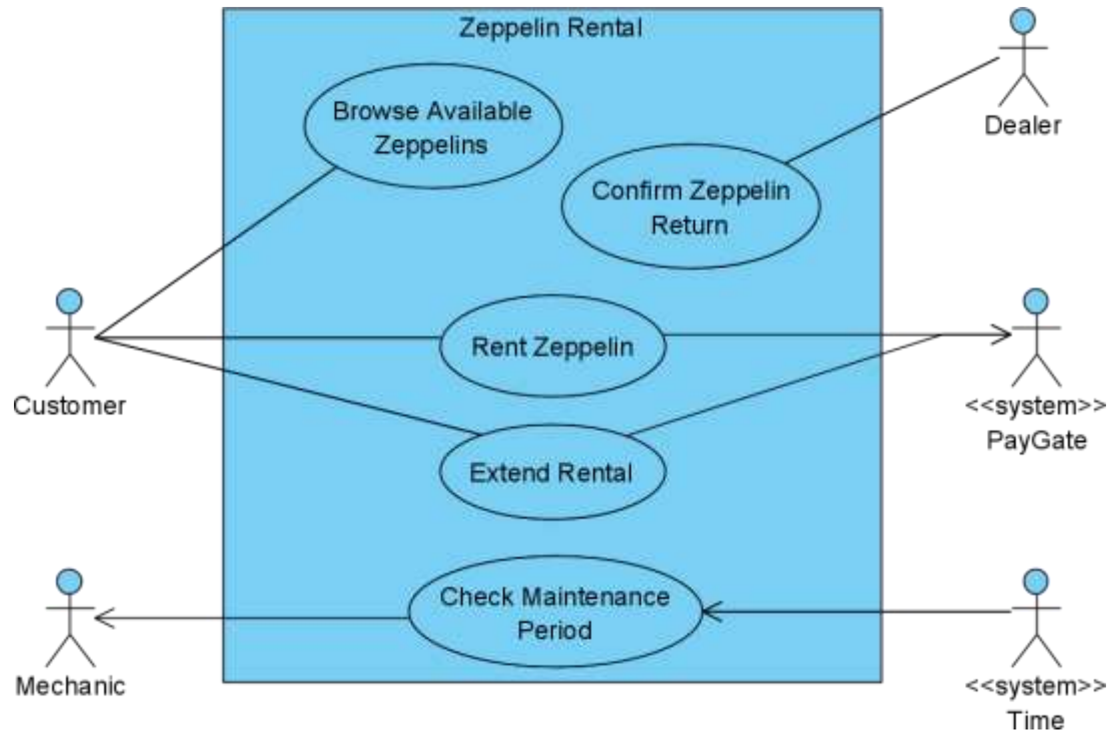


MUNI FI

Use case diagram

Use case

- An action describing interaction of the external actor with the system
  - Always begins with an action triggered by a primary actor
  - Other (secondary) actors may join during the interaction
  - Described from actors' point of view (not as requirements written from system point of view)
  - Name should represent the activity
- Use cases can have **preconditions**

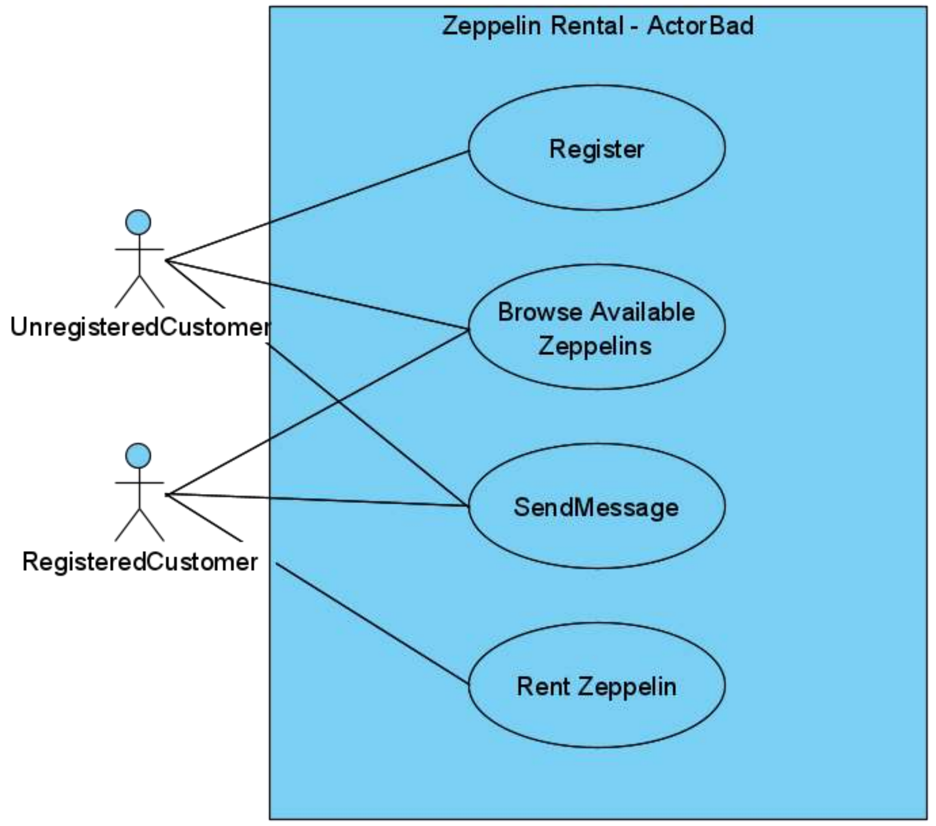


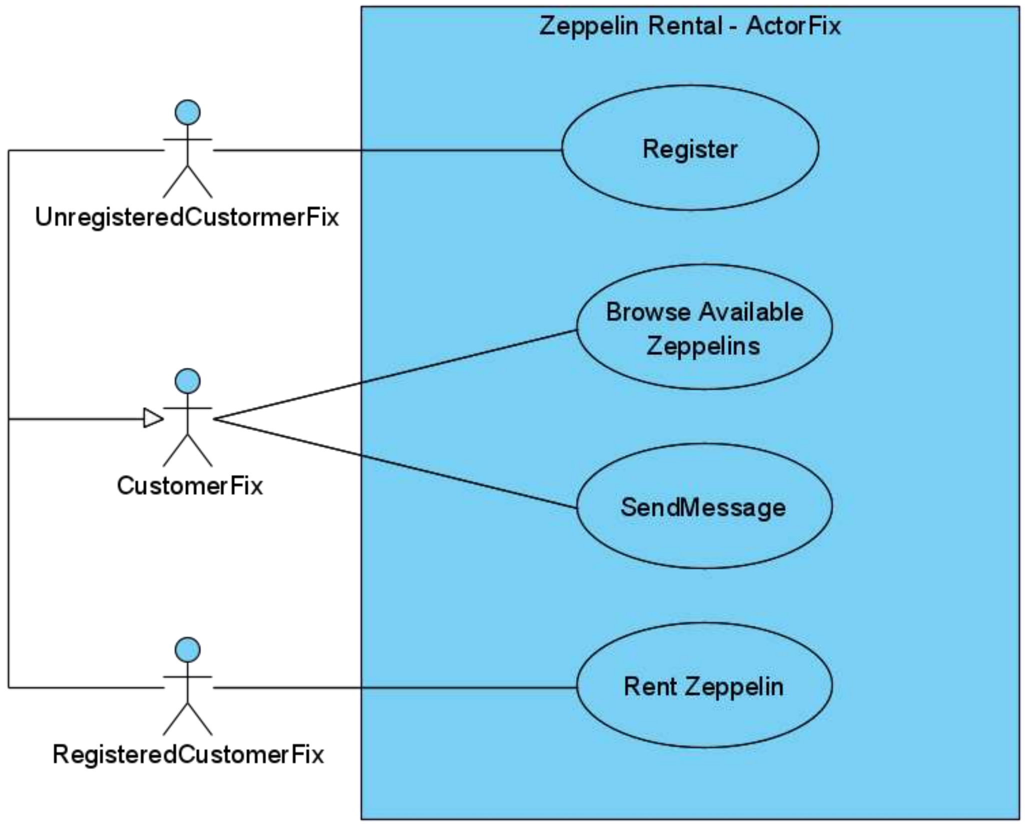
MUNI FI

Generalization - inheritance

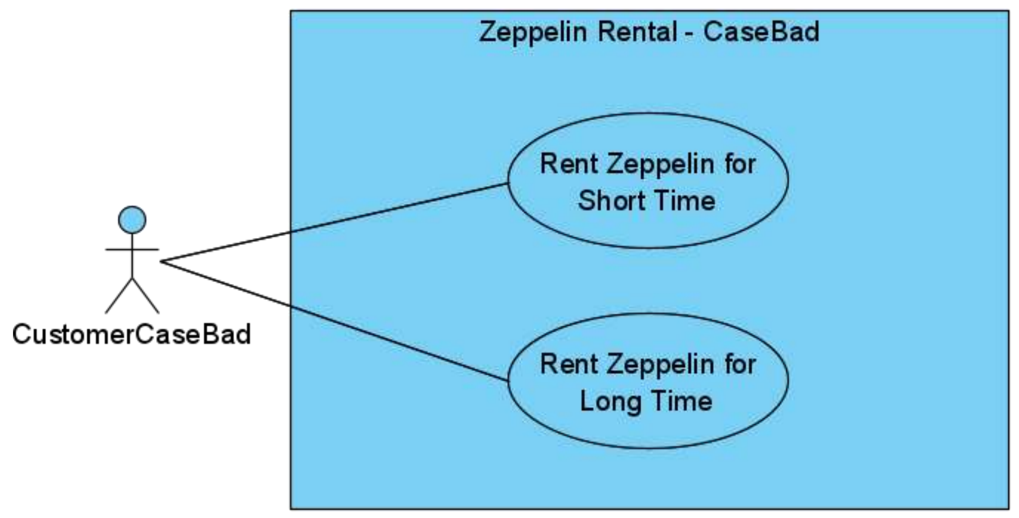
- To simplify the diagram
- **Actor generalization**
  - Should be used when multiple actors share use cases
  - Children inherit all roles from their parent and can trigger all use cases of their parent
- **Use case generalization**
  - Used when use cases share the same logic – they vary only in details
  - specialized use cases inherit all properties from their parent, but add new features, can override the inherited properties (they cannot override the parents' extension points)
- Often, parents are abstract

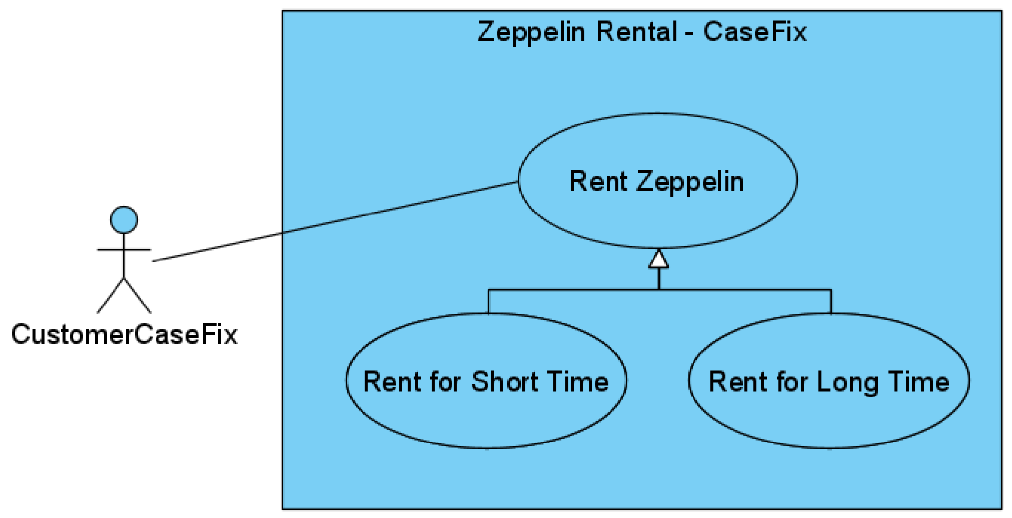
Actor generalization





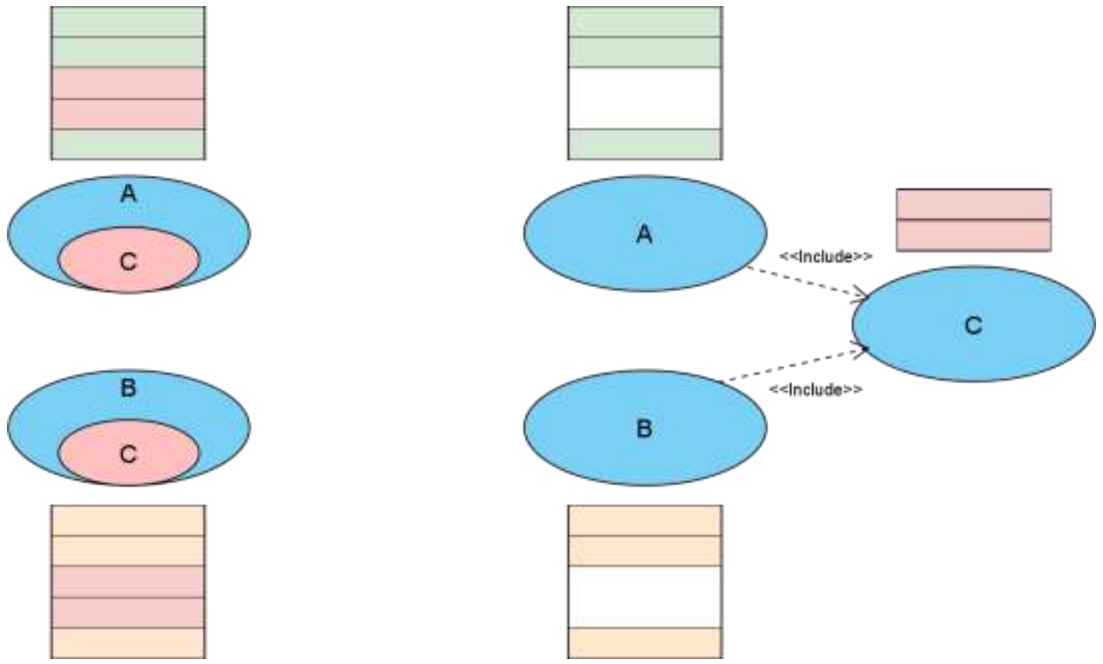
Use case generalization


Zeppelin Rental - CaseBad
Rent Zeppelin for Short Time
Rent Zeppelin for Long Time
CustomerCaseBad



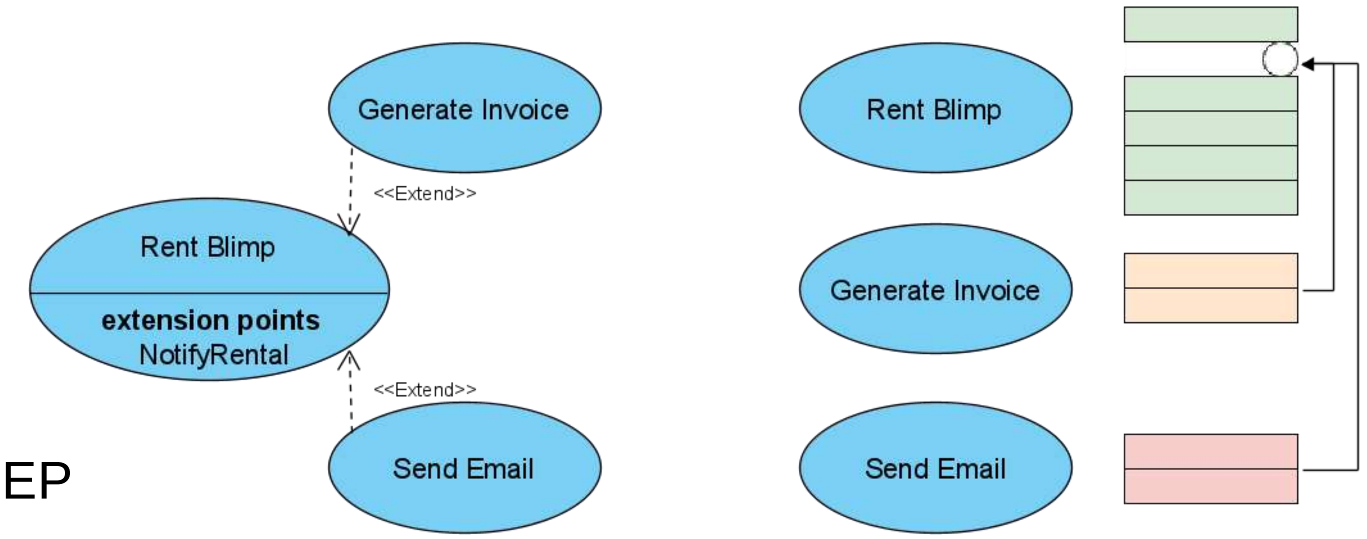
Include

- Extracts repetitive steps of multiple use cases into a separate use case
- A use case refers to another use case that will be executed afterwards
- Syntax:
  A -> C = A includes C
  = C is included in A

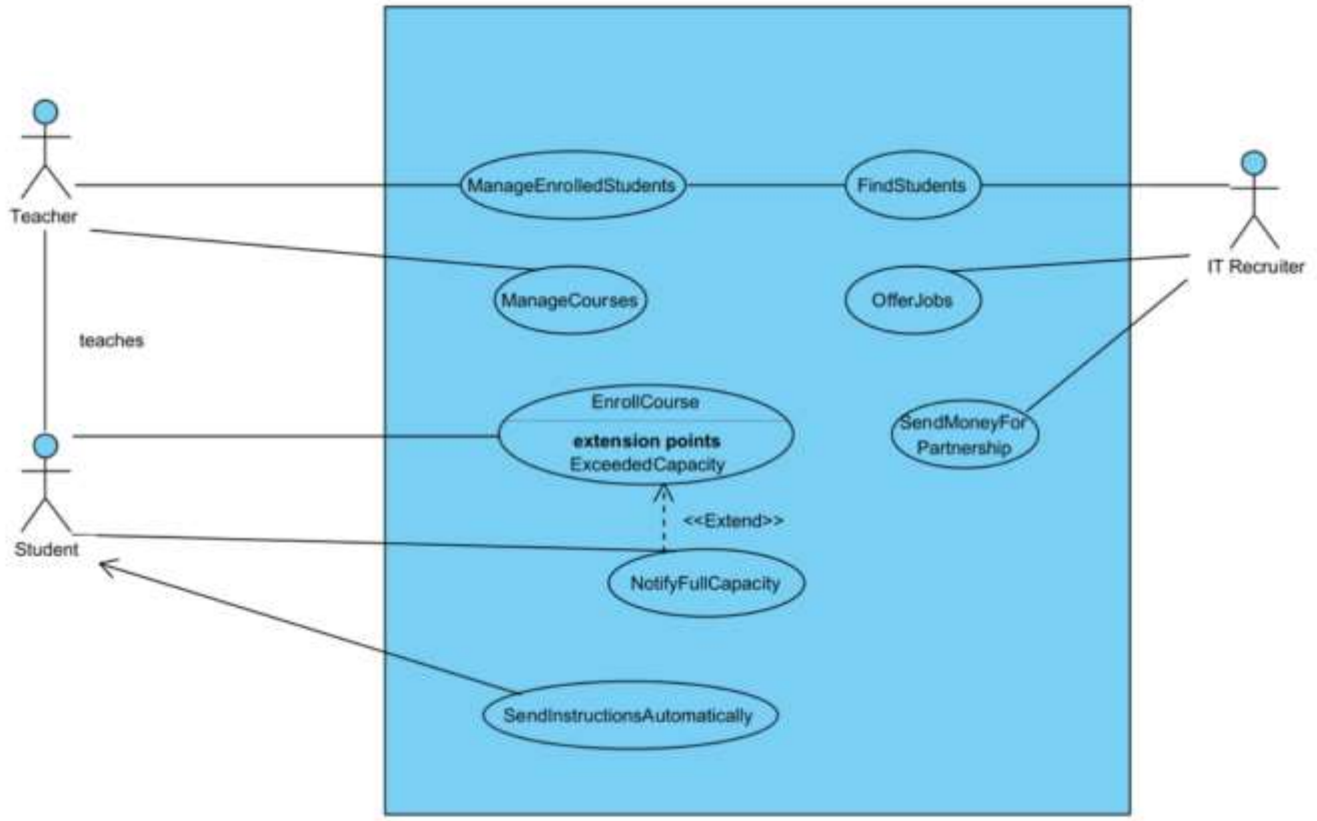


Extend

- Allows insertion of additional behaviour into base use case
  - **extension points = EP** – specifically defined place where the behavior is inserted
  - If a condition defined in the EP is met, the extended UC is executed
  - Syntax:
    A -> B = **A extends B**
    = B is extended by A
  - The base does not know about its extensions
  - There can be multiple EP for one UC or multiple UC for one EP



Activity – What's wrong here?



Task for this week

You gotta do what you gotta do

- Create a list of requirements – 10+ functional, 5+ non-functional
- Create a use case diagram according to your requirements (use at least one include and generalization instance)
- **Submit reports until Wednesday (9th Oct) 8:00 am**