CS130: Software Engineering

Lab 1

Agenda

1. Introductions
2. Course Overview
3. Capstone Project Overview
4. Brief Introduction of UML
5. Team Formation

Course Overview

Course Goal

To learn ***systematic software development methods and tools*** in the context of developing a software system in collaboration with other people.

Systematic design, construction, verification, and evolution methods include

- Object-oriented design patterns
- Refactoring
- Unit testing, test coverage, and test generation
- Hoare logic, preconditions, post-conditions, and loop invariants

Grade Breakdown

Project: 35%

Midterm: 20%

Final: 35%

Homework: 8% (2 assignments, 4% each)

Participation: 2%

Capstone Project Overview

Team Size: 5 (or 6)

Project Guidelines:

1. Application should be a new and unique creation
2. When proposing a project, you should investigate whether it is feasible to implement and test it.
3. You are expected to perform a live demonstration of your project

Grade Breakdown:

1. a new feature proposal and presentation (6%)
2. a midpoint implementation and testing progress report and presentation (11%)
3. a final demonstration, presentation, and a youtube video about your project demo (18%)

Grading:

1. Students in the same team **will not** always receive the same grades, and project grades will account for individual effort and contributions
2. We will consider the peer feedback and collaboration history when assigning individual grades
3. TAs are incharge of grading projects

More info on:
http://web.cs.ucla.edu/~miryung/teaching/CS130-Fall2019/main.html

Project Video Examples:

Nomad: https://www.youtube.com/watch?v=Rhl0HH_OGVg&

GradePortal: https://www.youtube.com/watch?v=WBTGdxXyoII

Rockmates: https://www.youtube.com/watch?v=X_L3-dA1iKQ

Let’s Revise Some Concepts

Subclassing

Extends

B inherits from **A**.

Eg: Consider animals:
- Mammal (B) inherits from Animal(A)
- Cat (C) inherits from Mammal (M)

Subclassing refers to reuse of implementations. Used for factoring out repeated code.

Definition: A type **B** is a subclass of another type **A** if some functions for **B** are written in terms of functions of **A**

Subtyping

Implements

Every **B** is an **A**.

Eg: In a library database:
- every book (B) is a library holding (A)
- every CD (C) is a library holding (A)

Subtyping refers to compatibility of interfaces.

Definition: A type **B** is a subtype of **A** if every function that can be invoked on an object of type **A** can also be invoked on an object of type **B**

Dynamic Dispatch

```
public class SurveyBase
{
    public virtual void DoSurvey()
    {
       Console.WriteLine("Base Class");
    }
}

public class Survey : SurveyBase
{
    public override void DoSurvey()
    {
        Console.WriteLine("Derived Class");
    }
}

SurveyBase base = new Survey();
base.DoSurvey();                                // Prints "Derived Class"
```

In Object Oriented Programming, Dynamic Dispatch refers to the process of determining the actual method of derived class at run time by the actual instance type.

https://www.codeproject.com/Tips/875393/Static-Dynamic-Dispatch-ReExplained

UML: Unified Modeling Language

A BRIEF INTRODUCTION

UML - Some Motivation

Why do we need a modeling language for Software?

- Provides the development community with a stable and common design language that could be used to develop and build computer applications.

- Sort of like a blueprint for a building. Potential errors become visible before actual development.

- Programming language independent, easy to understand for everyone

Requirements Engineering

- First step of the waterfall model
- Why is it important? Think-Pair-Share
    - To decide the scope of the project – Feasibility studies
    - Discover problems and inconsistencies early
    - Lay the foundation

UML - Use Case Diagram

- Illustrates a unit of functionality provided by the system.
- Main purpose is to help development teams visualize the functional requirements of a system
- Capture requirements from a user’s perspective
- Actors – Human beings who will interact with the system

Components of a Use Case Diagram

Actors – Stick figures

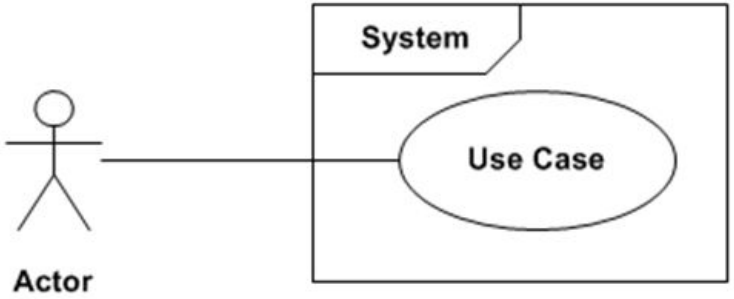
A role that a user takes when invoking a use case

A single user may be represented by multiple actors

Use cases – Ovals

Edges from actor to use case showing that the actor is involved in that use case

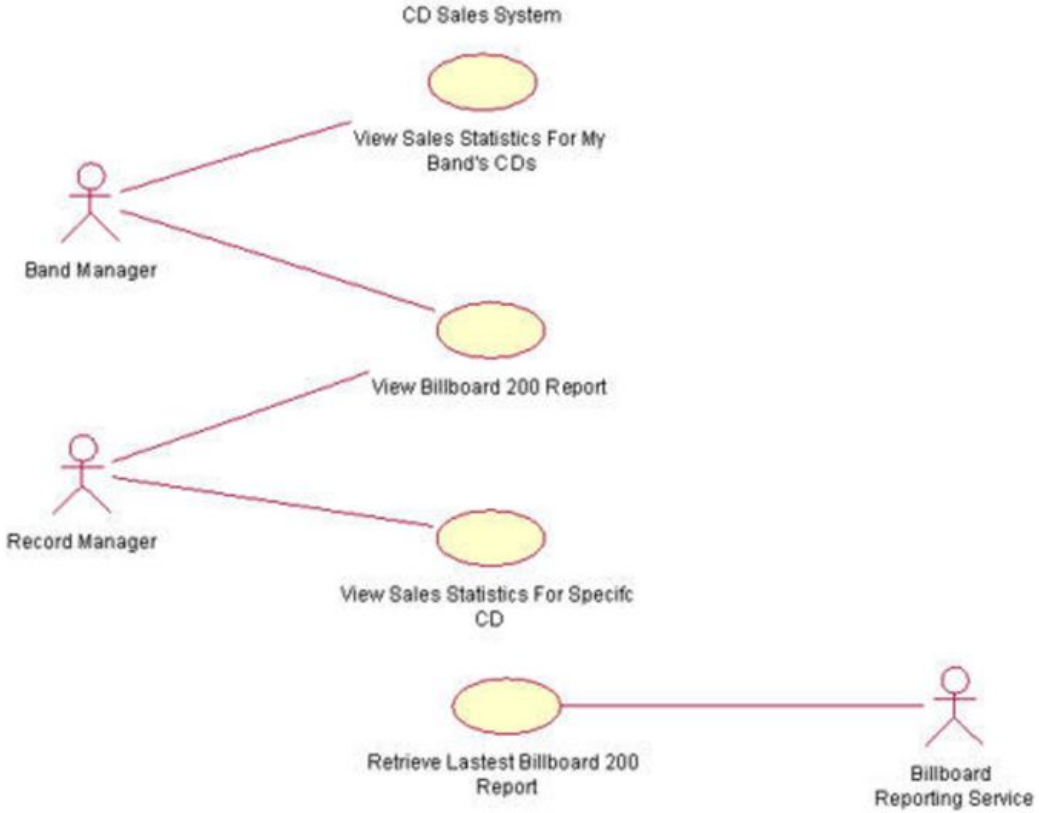
Denote association



Example: CD Sales System

Requirement :

- Band Manager should be able to view a sales statistics report and the Billboard 200 report for the band’s CDs.
- Record Manager should view the sales statistics report and the Billboard 200 report for a particular CD.
- Billboard reports are delivered from an external system called Billboard Reporting Service


CD Sales System
View Sales Statistics For My Band's CDs
Band Manager
View Billboard 200 Report
Record Manager
View Sales Statistics For Specifc CD
Retrieve Lastest Billboard 200 Report
Billboard Reporting Service

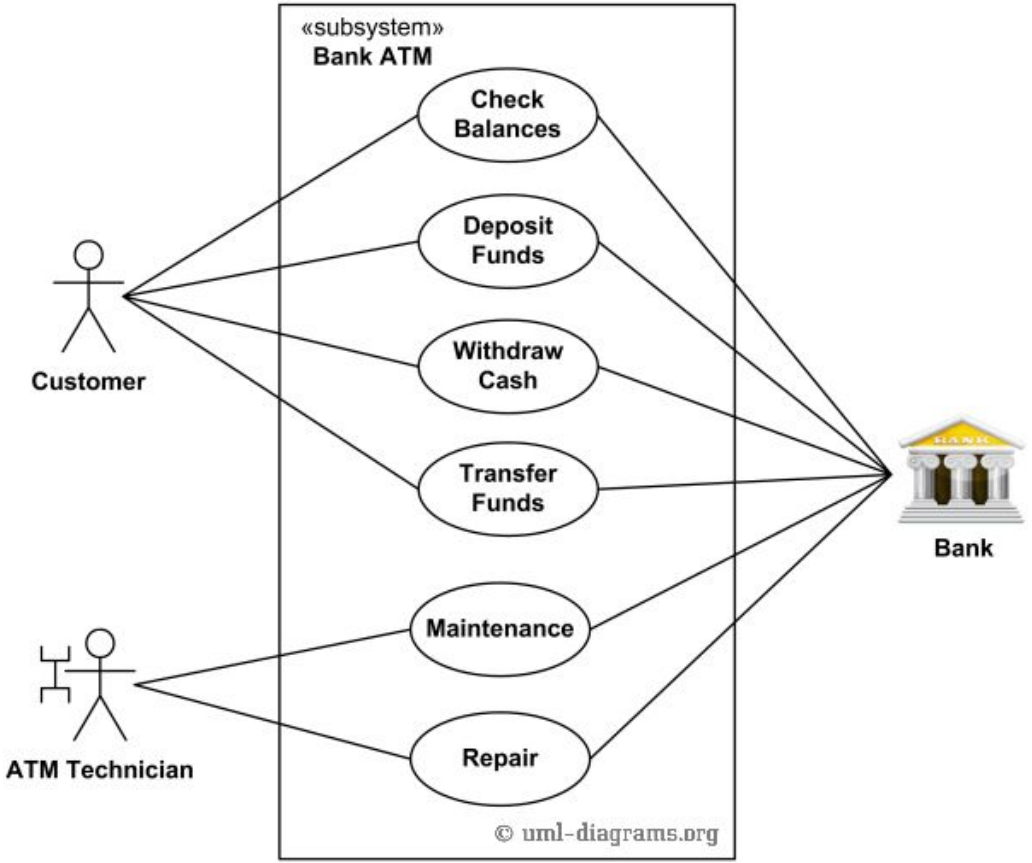
Let’s do a quick exercise!

Bank ATM

- Customer (**actor**) uses bank ATM to Check Balances of his/her bank accounts, Deposit Funds, Withdraw Cash and/or Transfer Funds (**use cases**).

- ATM Technician provides Maintenance and Repairs.

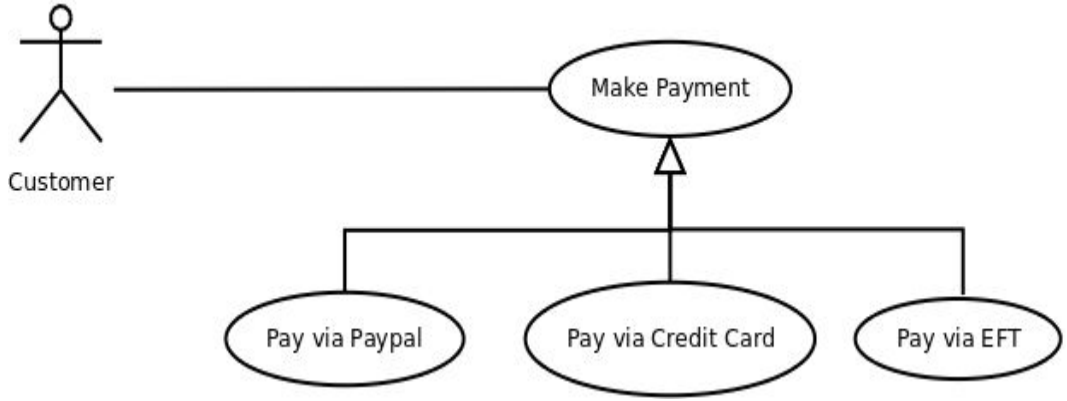
- All these use cases also involve Bank actor whether it is related to customer transactions or to the ATM servicing.


«subsystem»
Bank ATM
Check Balances
Deposit Funds
Withdraw Cash
Transfer Funds
Maintenance
Repair
Customer
ATM Technician
Bank
© uml-diagrams.org

More components of Use-Case Diagrams

Generalization

Used when you find two or more use cases that have commonalities in behavior, structure, and purpose.



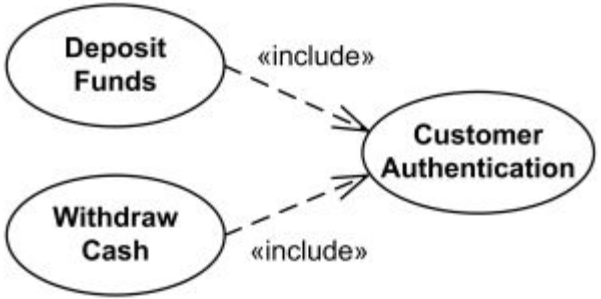
More components of Use-Case Diagrams

Inclusion

Used to extract use case fragments that are *duplicated* in multiple use cases.

Caveats:

1. The included use case cannot stand alone
2. The original use case is not complete without the included one.

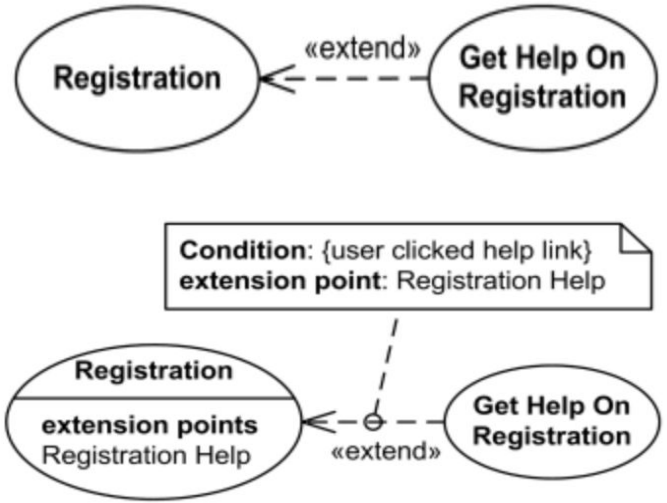


More components of Use-Case Diagrams

Extension

Used when a use case adds steps to another first-class use case. The extended use-case is not *required*.

You could choose to specify an *extension point* and *condition* for dealing with special cases



Let’s use these components to model how DoorDash works!

Statechart Diagrams

Another way of specifying behavioral requirements

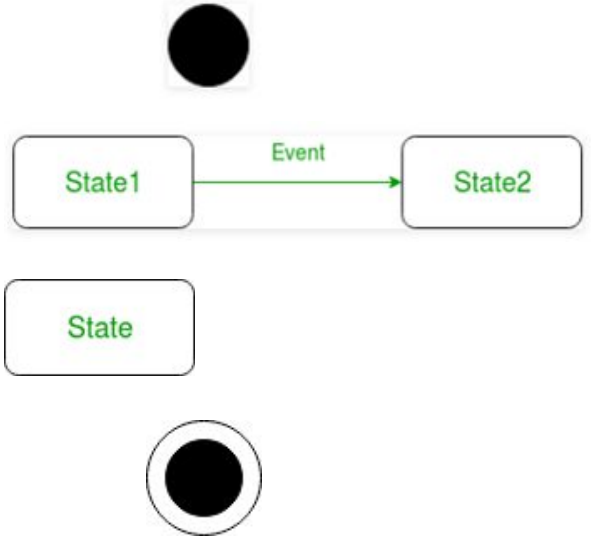
Built on state machines

Purpose :

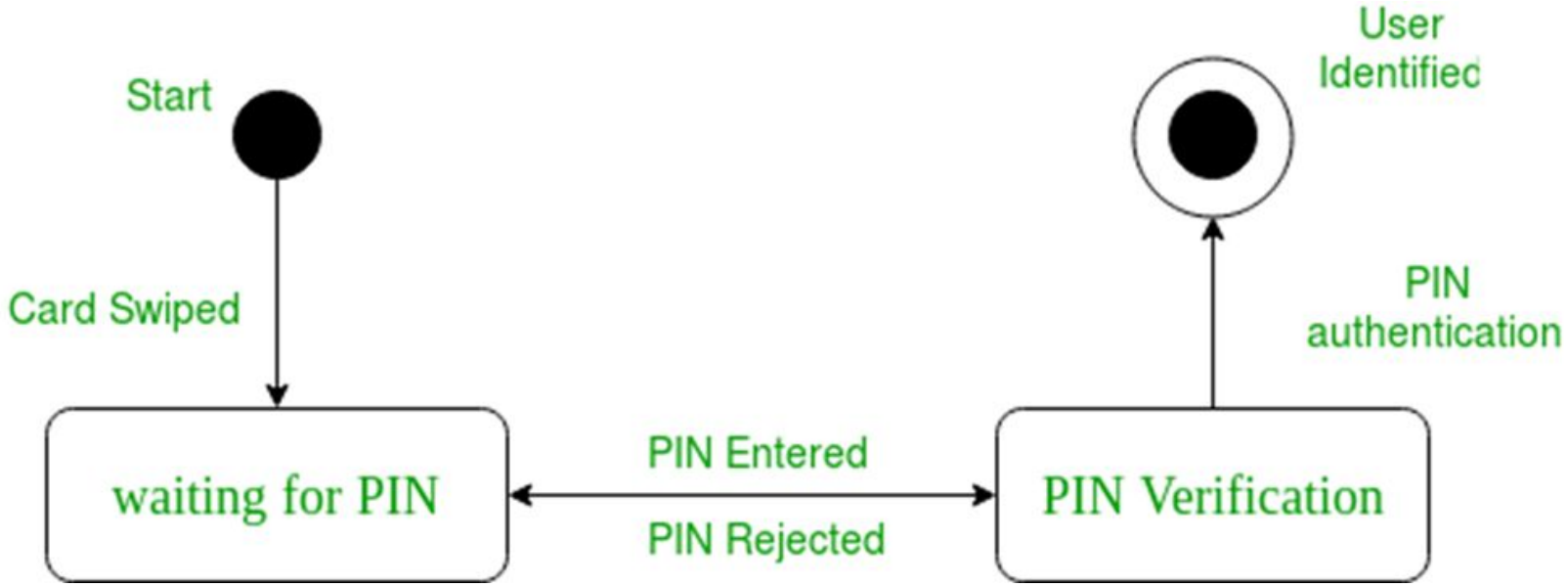
- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.
- Every class can have a state, but every class need not have a statechart diagram.

Components of State Chart Diagrams

- Initial starting point – Drawn using a solid circle
- Transition between states – Drawn using a line with an open arrowhead
- State – Drawn using a rectangle with rounded corners
- One or more termination points – Drawn using a circle with a solid circle inside



Example


Start
Card Swiped
waiting for PIN
PIN Entered
PIN Rejected
PIN Verification
PIN
authentication
User
Identified