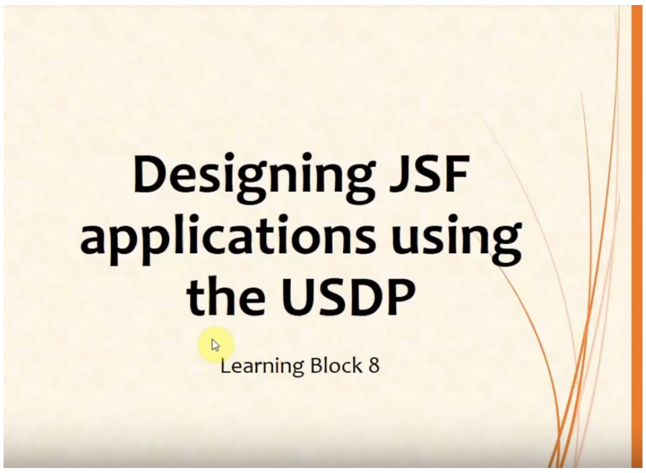


Designing JSF applications using the USDP

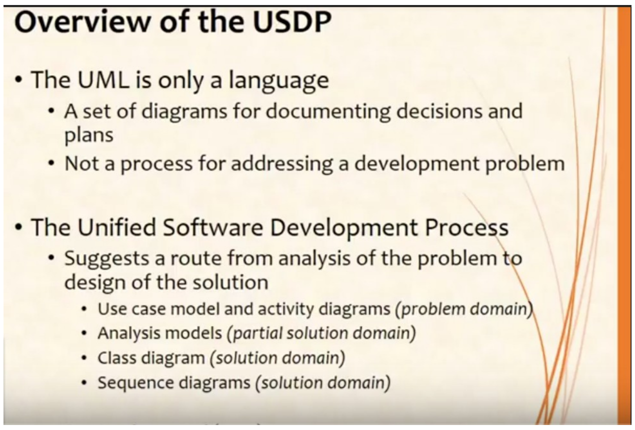
Learning Block 8

The USDP process is made of five stages.

- Usecase model (Problem domain).
- Activity diagram (Problem domain).
- Analysis models (Partial solution domain).
- Class diagram (Solution domain).
- Sequence diagrams (Solution domain).

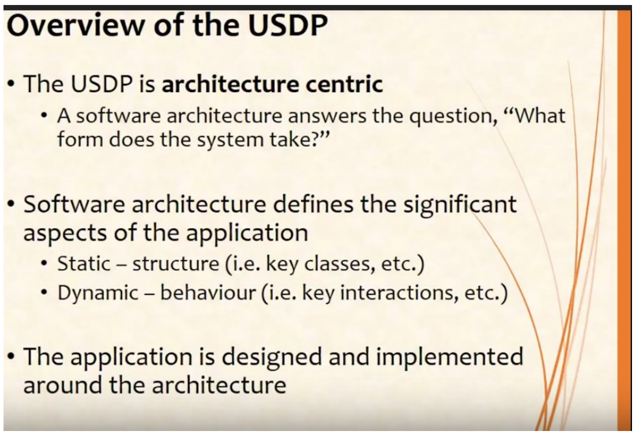
Think about the problem domain. The problem domain is defined by the real world situation in which that problem finds itself.

So if it's a problem about library books then we go to the library and we look at what we find in a library and we talk to the librarian we talk to other people who work in or use the library and we start to learn about their viewpoint the users viewpoint of the problem domain we also look for the terminology



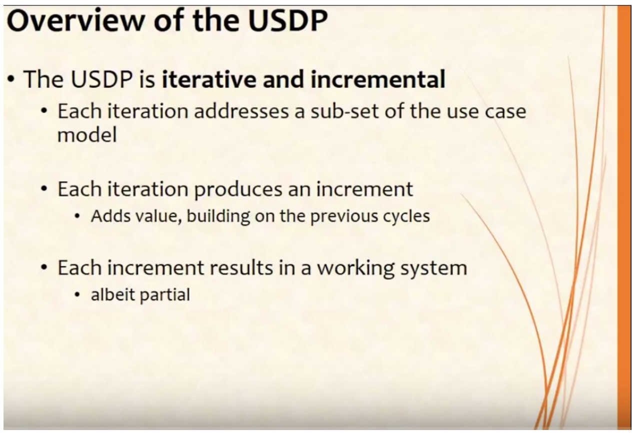
Overview of the USDP

- The UML is only a language
  - A set of diagrams for documenting decisions and plans
  - Not a process for addressing a development problem
- The Unified Software Development Process
  - Suggests a route from analysis of the problem to design of the solution
    - Use case model and activity diagrams *(problem domain)*
    - Analysis models *(partial solution domain)*
    - Class diagram *(solution domain)*
    - Sequence diagrams *(solution domain)*



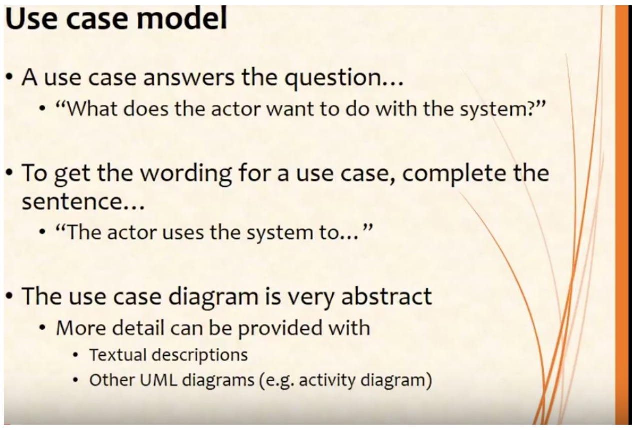
Overview of the USDP

- The USDP is **architecture centric**
  - A software architecture answers the question, "What form does the system take?"
- Software architecture defines the significant aspects of the application
  - Static – structure (i.e. key classes, etc.)
  - Dynamic – behaviour (i.e. key interactions, etc.)
- The application is designed and implemented around the architecture



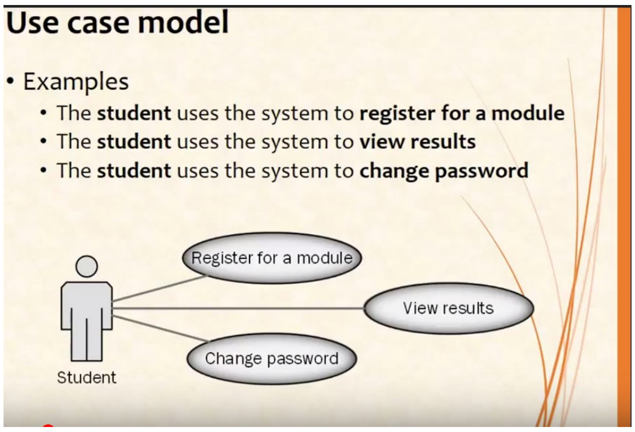
Overview of the USDP

- The USDP is **iterative and incremental**
  - Each iteration addresses a sub-set of the use case model
  - Each iteration produces an increment
    - Adds value, building on the previous cycles
  - Each increment results in a working system
    - albeit partial



Use case model

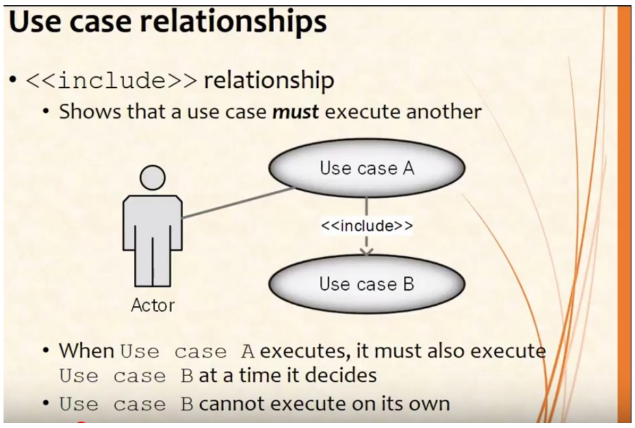
- A use case answers the question…
  - "What does the actor want to do with the system?"
- To get the wording for a use case, complete the sentence…
  - "The actor uses the system to…"
- The use case diagram is very abstract
  - More detail can be provided with
    - Textual descriptions
    - Other UML diagrams (e.g. activity diagram)



Use case model

- Examples
  - The **student** uses the system to **register for a module**
  - The **student** uses the system to **view results**
  - The **student** uses the system to **change password**

Student
Register for a module
View results
Change password

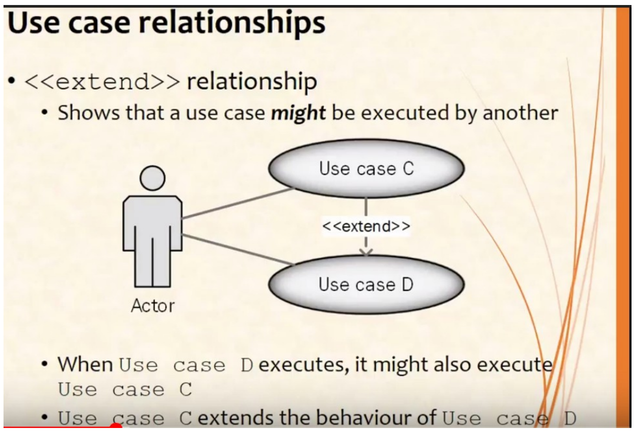


Use case relationships

- `<<include>>` relationship
  - Shows that a use case ***must*** execute another

Actor
Use case A
<<include>>
Use case B

- When `Use case A` executes, it must also execute `Use case B` at a time it decides
- `Use case B` cannot execute on its own

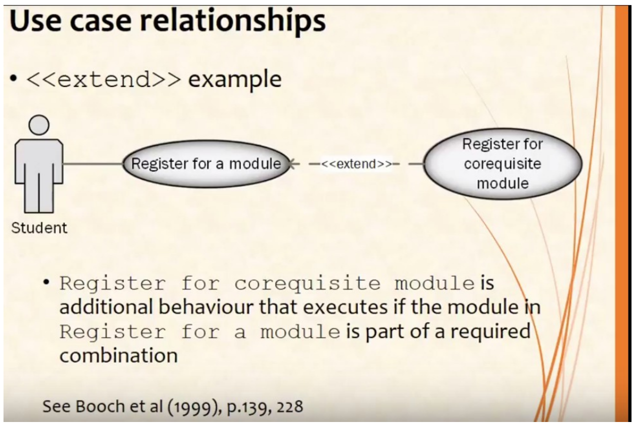


Use case relationships

- `<<extend>>` relationship
  - Shows that a use case ***might*** be executed by another

Actor
Use case C
<<extend>>
Use case D

- When `Use case D` executes, it might also execute `Use case C`
- `Use case C` extends the behaviour of `Use case D`



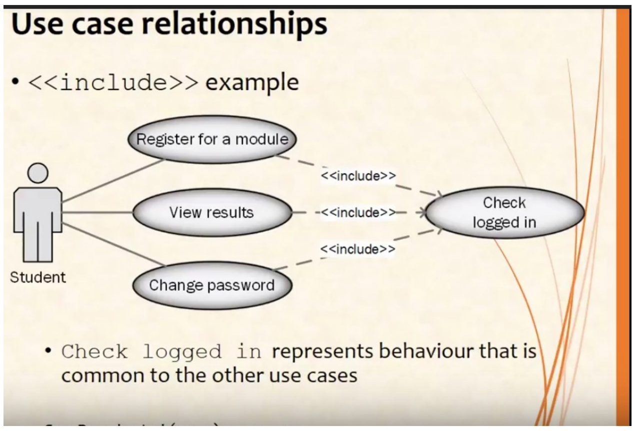
Use case relationships

- `<<extend>>` example

Student
Register for a module
<<extend>>
Register for corequisite module

- `Register for corequisite module` is additional behaviour that executes if the module in `Register for a module` is part of a required combination

See Booch et al (1999), p.139, 228

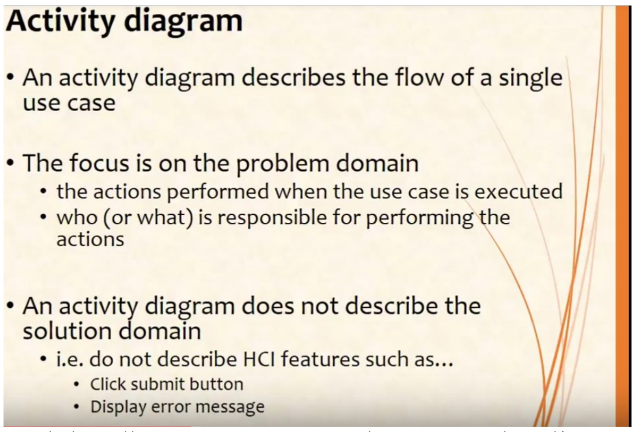


Use case relationships

- `<<include>>` example

Student
Register for a module
View results
Change password
<<include>>
<<include>>
<<include>>
Check logged in

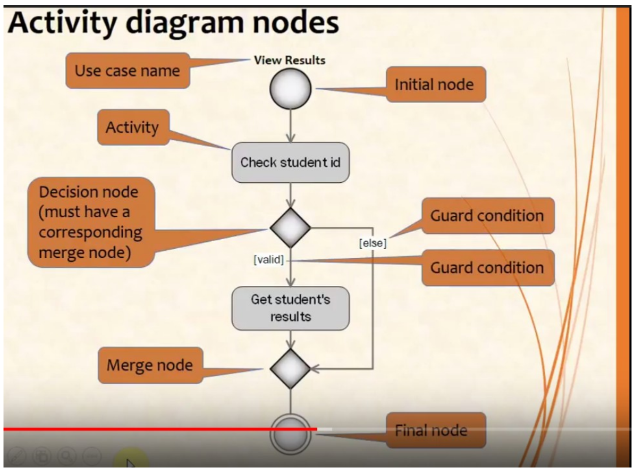
- `Check logged in` represents behaviour that is common to the other use cases



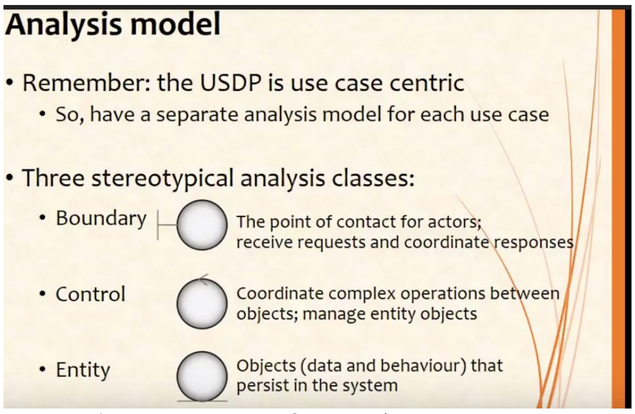
Activity diagram

- An activity diagram describes the flow of a single use case
- The focus is on the problem domain
  - the actions performed when the use case is executed
  - who (or what) is responsible for performing the actions
- An activity diagram does not describe the solution domain
  - i.e. do not describe HCI features such as…
    - Click submit button
    - Display error message

Activity diagram starts to give more detail about a use case it describes a given use case and it will represent a flow of activity for that use case that it describes again the focus is on the problem domain so we are using the vocabulary and our observations of what is happening in the real world to define the activities that should take place within this use case.



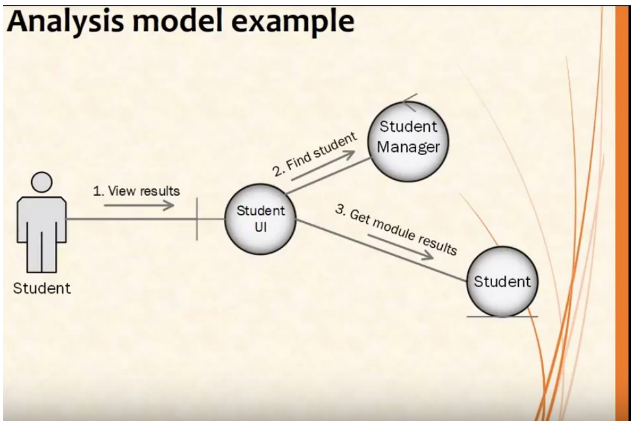
Activity diagram nodes



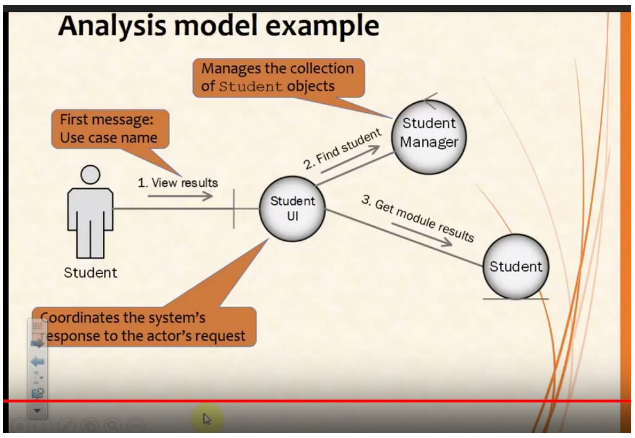
Analysis model

- Remember: the USDP is use case centric
  - So, have a separate analysis model for each use case
- Three stereotypical analysis classes:
  - Boundary — The point of contact for actors; receive requests and coordinate responses
  - Control — Coordinate complex operations between objects; manage entity objects
  - Entity — Objects (data and behaviour) that persist in the system

Boundary or Interface class.
Entity class= Business class.
Control = Handler class.

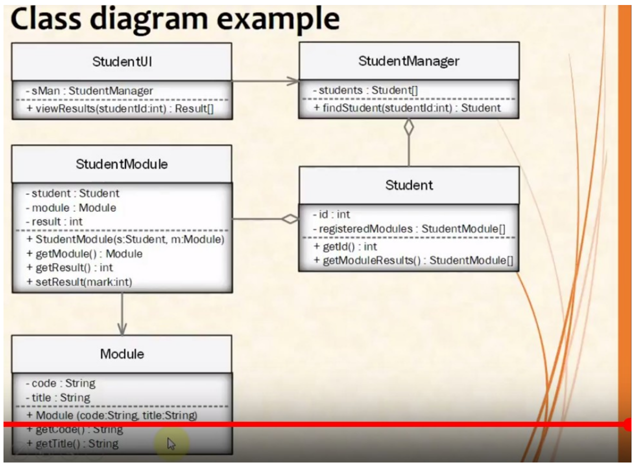


Analysis model example

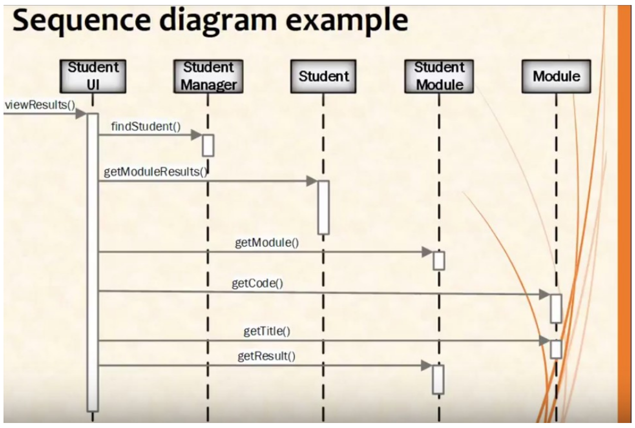


Analysis model example

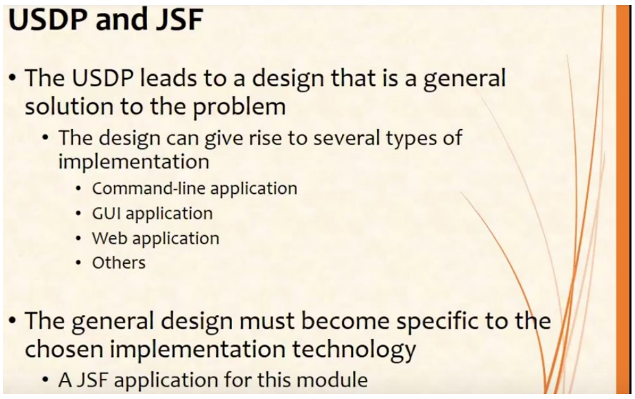
Manages the collection of Student objects
First message: Use case name
Student
1. View results
Student UI
2. Find student
Student Manager
3. Get module results
Student
Coordinates the system's response to the actor's request



Class diagram example

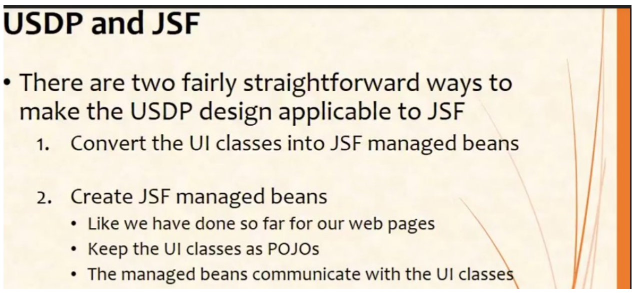


Sequence diagram example



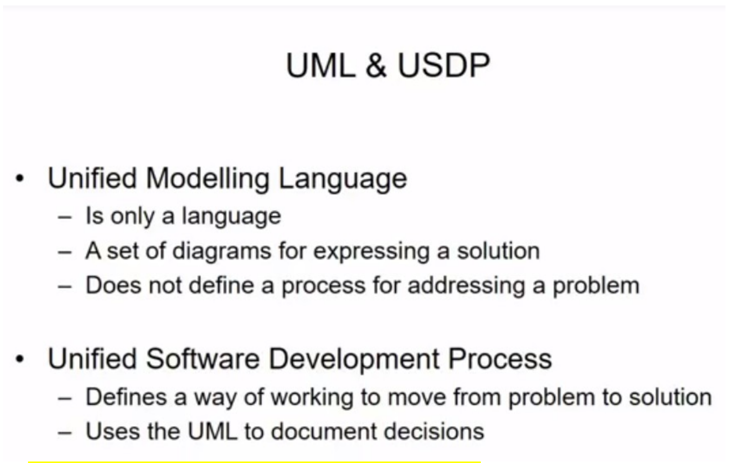
USDP and JSF

- The USDP leads to a design that is a general solution to the problem
  - The design can give rise to several types of implementation
    - Command-line application
    - GUI application
    - Web application
    - Others
- The general design must become specific to the chosen implementation technology
  - A JSF application for this module



USDP and JSF

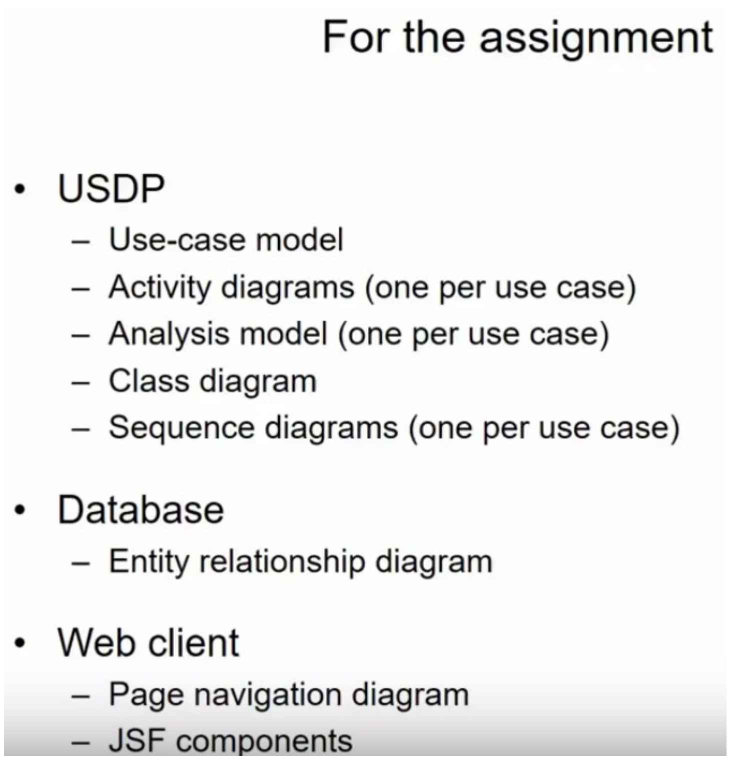
- There are two fairly straightforward ways to make the USDP design applicable to JSF
  1. Convert the UI classes into JSF managed beans
  2. Create JSF managed beans
     - Like we have done so far for our web pages
     - Keep the UI classes as POJOs
     - The managed beans communicate with the UI classes



UML & USDP

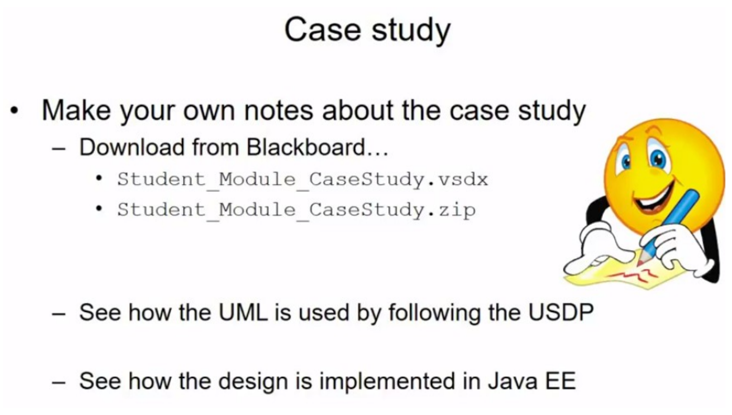
- Unified Modelling Language
  - Is only a language
  - A set of diagrams for expressing a solution
  - Does not define a process for addressing a problem
- Unified Software Development Process
  - Defines a way of working to move from problem to solution
  - Uses the UML to document decisions

How to derive use case names.
What the user uses the System to do, that is the use case name.
Tips: The Admin uses the System to
E.g
1. Create or delete user.



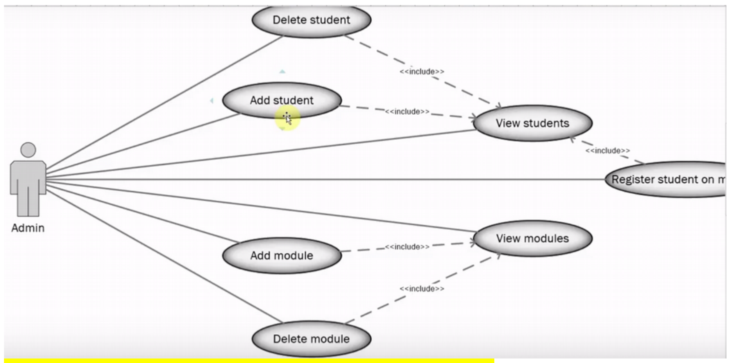
For the assignment

- USDP
  - Use-case model
  - Activity diagrams (one per use case)
  - Analysis model (one per use case)
  - Class diagram
  - Sequence diagrams (one per use case)
- Database
  - Entity relationship diagram
- Web client
  - Page navigation diagram
  - JSF components

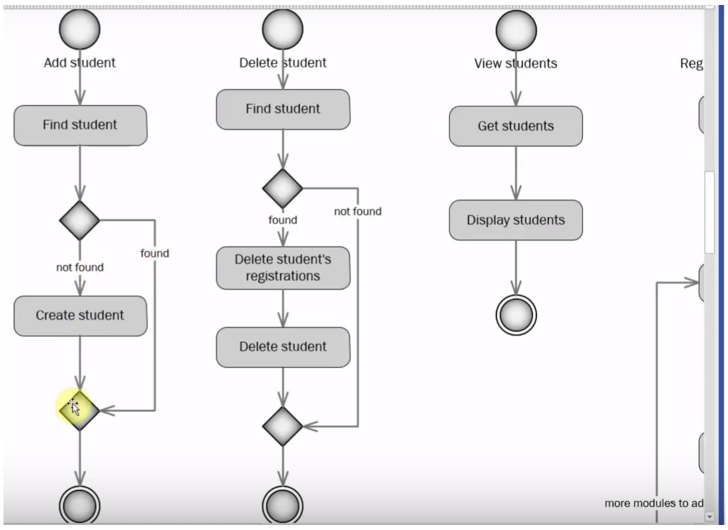


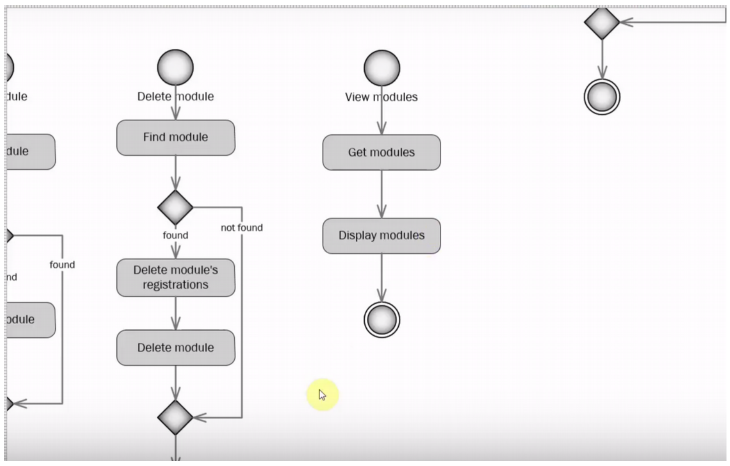
Case study

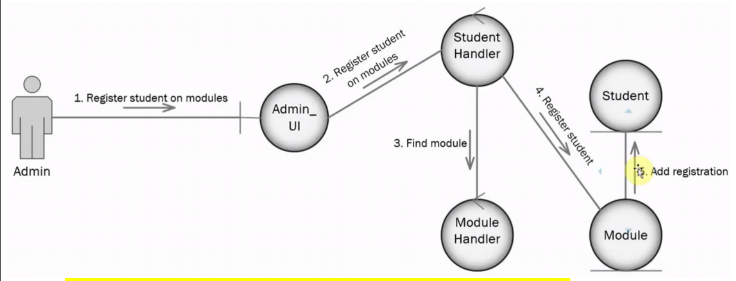
- Make your own notes about the case study
  - Download from Blackboard…
    - `Student_Module_CaseStudy.vsdx`
    - `Student_Module_CaseStudy.zip`
  - See how the UML is used by following the USDP
  - See how the design is implemented in Java EE



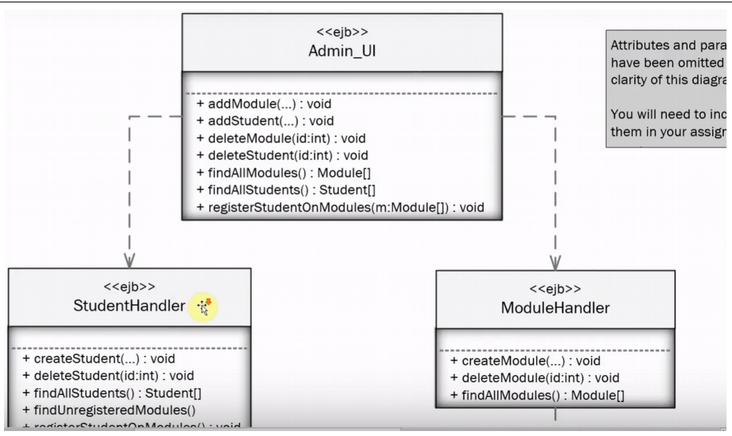
The Admin uses the System to …….
This is how to derive the usecase names.



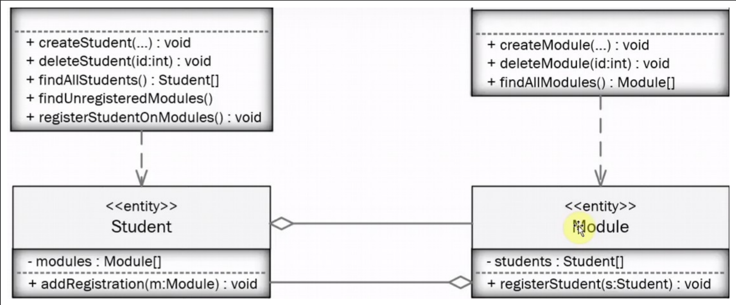




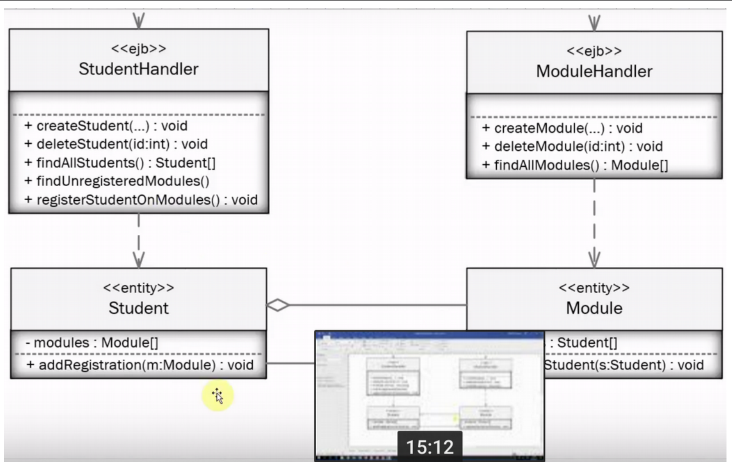
The message becomes a method in the class.
E.g The message register student on modules, becomes a method in the Admin_UI class



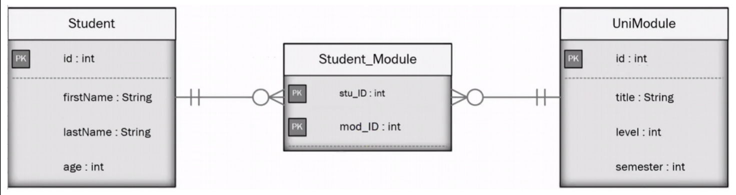
<<ejb>>
Admin_UI
+ addModule(...) : void
+ addStudent(...) : void
+ deleteModule(id:int) : void
+ deleteStudent(id:int) : void
+ findAllModules() : Module[]
+ findAllStudents() : Student[]
+ registerStudentOnModules(m:Module[]) : void
Attributes and para
have been omitted
clarity of this diagra
You will need to inc
them in your assign
<<ejb>>
StudentHandler
+ createStudent(...) : void
+ deleteStudent(id:int) : void
+ findAllStudents() : Student[]
+ findUnregisteredModules()
<<ejb>>
ModuleHandler
+ createModule(...) : void
+ deleteModule(id:int) : void
+ findAllModules() : Module[]

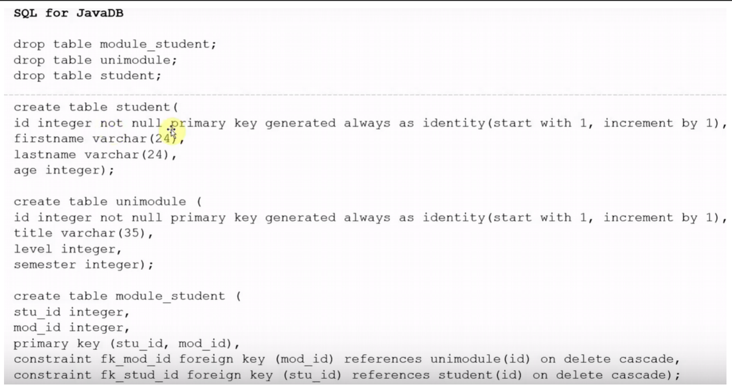


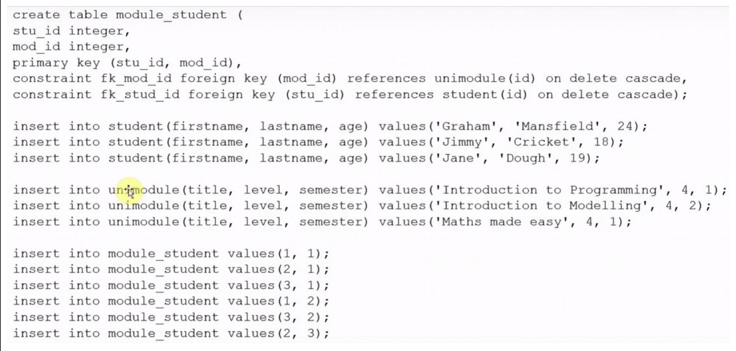
+ createStudent(...) : void
+ deleteStudent(id:int) : void
+ findAllStudents() : Student[]
+ findUnregisteredModules()
+ registerStudentOnModules() : void
+ createModule(...) : void
+ deleteModule(id:int) : void
+ findAllModules() : Module[]
<<entity>>
Student
- modules : Module[]
+ addRegistration(m:Module) : void
<<entity>>
Module
- students : Student[]
+ registerStudent(s:Student) : void

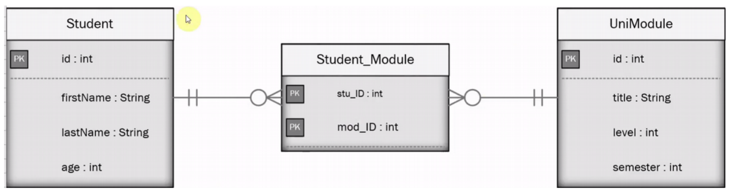


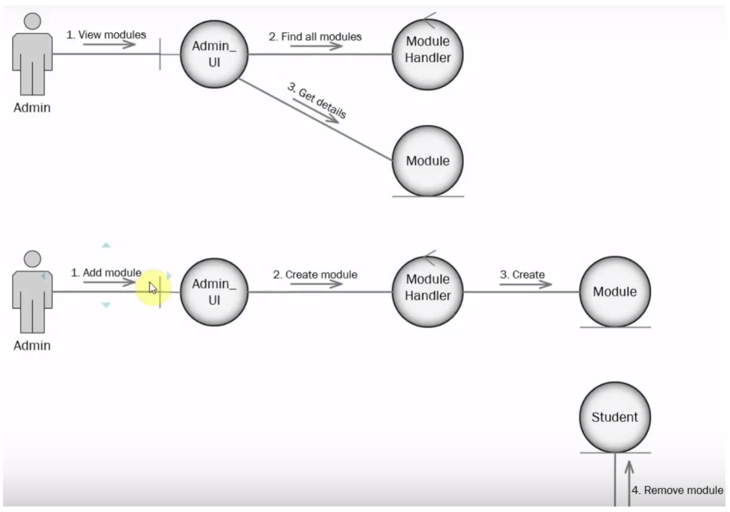
<<ejb>>
StudentHandler
+ createStudent(...) : void
+ deleteStudent(id:int) : void
+ findAllStudents() : Student[]
+ findUnregisteredModules()
+ registerStudentOnModules() : void
<<ejb>>
ModuleHandler
+ createModule(...) : void
+ deleteModule(id:int) : void
+ findAllModules() : Module[]
<<entity>>
Student
- modules : Module[]
+ addRegistration(m:Module) : void
<<entity>>
Module
: Student[]
Student(s:Student) : void
15:12

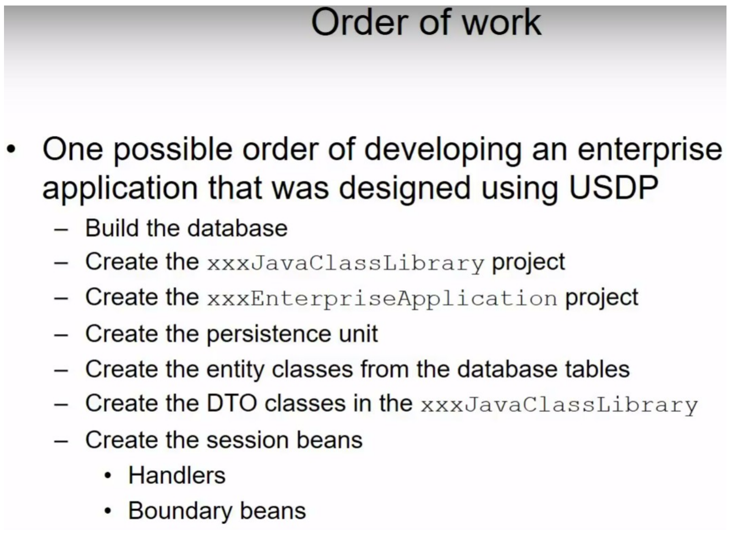



```
SQL for JavaDB

drop table module_student;
drop table unimodule;
drop table student;

create table student(
id integer not null primary key generated always as identity(start with 1, increment by 1),
firstname varchar(24),
lastname varchar(24),
age integer);

create table unimodule (
id integer not null primary key generated always as identity(start with 1, increment by 1),
title varchar(35),
level integer,
semester integer);

create table module_student (
stu_id integer,
mod_id integer,
primary key (stu_id, mod_id),
constraint fk_mod_id foreign key (mod_id) references unimodule(id) on delete cascade,
constraint fk_stud_id foreign key (stu_id) references student(id) on delete cascade);
```


```
create table module_student (
stu_id integer,
mod_id integer,
primary key (stu_id, mod_id),
constraint fk_mod_id foreign key (mod_id) references unimodule(id) on delete cascade,
constraint fk_stud_id foreign key (stu_id) references student(id) on delete cascade);

insert into student(firstname, lastname, age) values('Graham', 'Mansfield', 24);
insert into student(firstname, lastname, age) values('Jimmy', 'Cricket', 18);
insert into student(firstname, lastname, age) values('Jane', 'Dough', 19);

insert into unimodule(title, level, semester) values('Introduction to Programming', 4, 1);
insert into unimodule(title, level, semester) values('Introduction to Modelling', 4, 2);
insert into unimodule(title, level, semester) values('Maths made easy', 4, 1);

insert into module_student values(1, 1);
insert into module_student values(2, 1);
insert into module_student values(3, 1);
insert into module_student values(1, 2);
insert into module_student values(3, 2);
insert into module_student values(2, 3);
```







Order of work

- One possible order of developing an enterprise application that was designed using USDP
  - Build the database
  - Create the `xxxJavaClassLibrary` project
  - Create the `xxxEnterpriseApplication` project
  - Create the persistence unit
  - Create the entity classes from the database tables
  - Create the DTO classes in the `xxxJavaClassLibrary`
  - Create the session beans
    - Handlers
    - Boundary beans

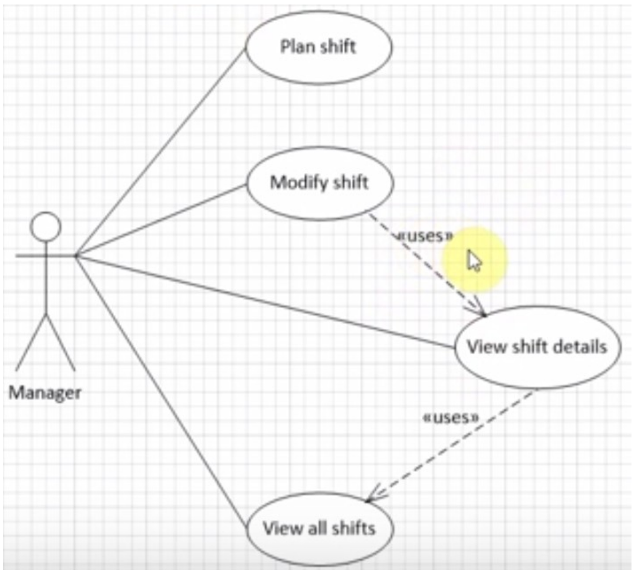
There are two fairly straightforward ways to make a USDP design applicable to JSF the first is just to take the UI classes and convert them into the managed beans so in your USDP design you've got your boundary class with all those methods coming in those effectively will become action methods that the views will call when the submit button on that view is clicked so the managed bean becomes the interface class.

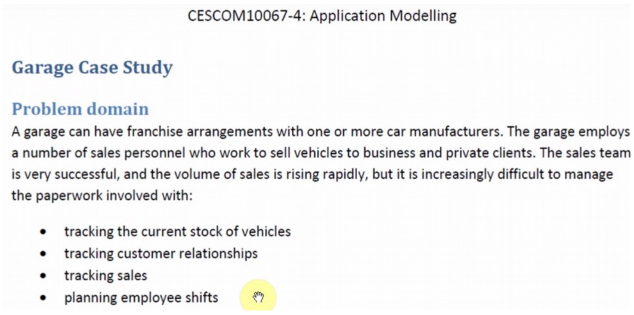
A better approach is to create JSF managed beans , have the beans sit behind the views that we want, and then have those beans in their action methods call the UI classes from the USDP design so those **UI classes** are essentially **pojos** that sit behind the managed beans and the managed beans interact with those **UI classes** .

Note: Control and Entity class are business class.

Entity class are used for data storage in the system.

Control class manages the communication between the boundary, entity and database.
Boundary or interface classes are ManagedBeans



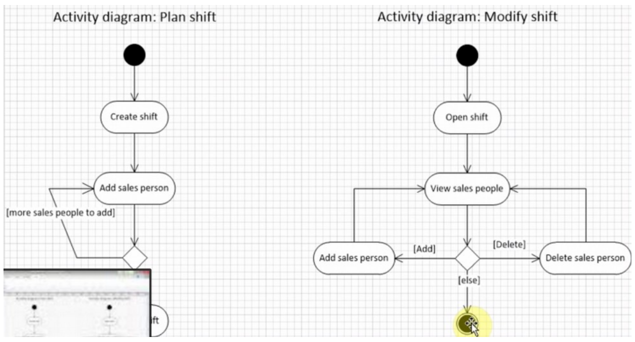

CESCOM10067-4: Application Modelling

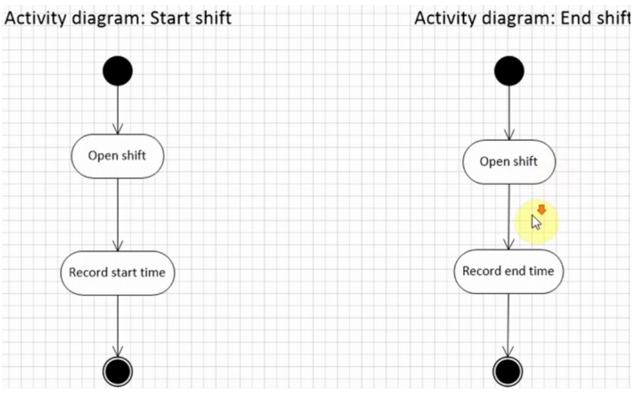
Garage Case Study

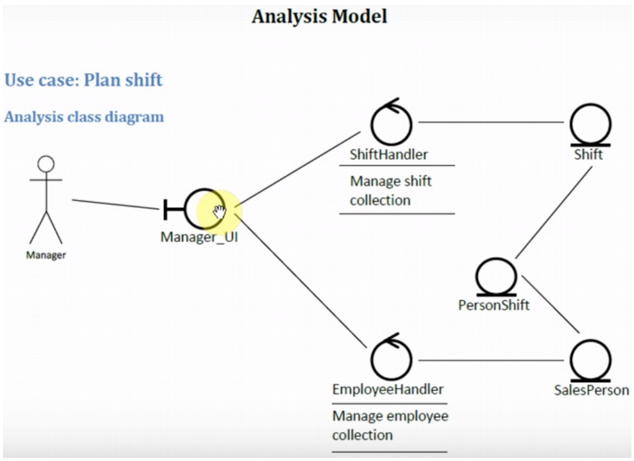
Problem domain

A garage can have franchise arrangements with one or more car manufacturers. The garage employs a number of sales personnel who work to sell vehicles to business and private clients. The sales team is very successful, and the volume of sales is rising rapidly, but it is increasingly difficult to manage the paperwork involved with:

- tracking the current stock of vehicles
- tracking customer relationships
- tracking sales
- planning employee shifts

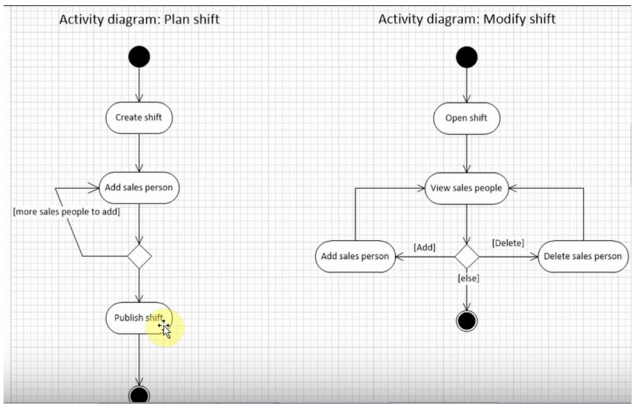


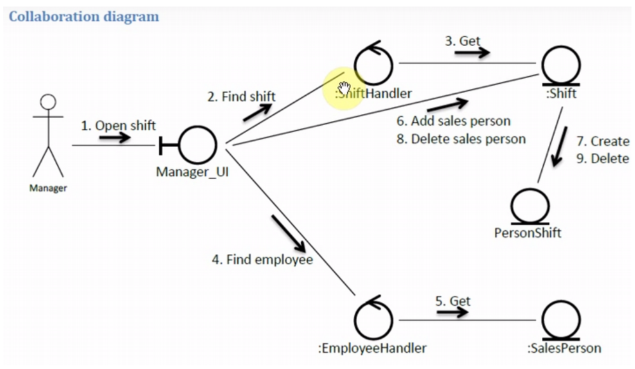



Analysis Model

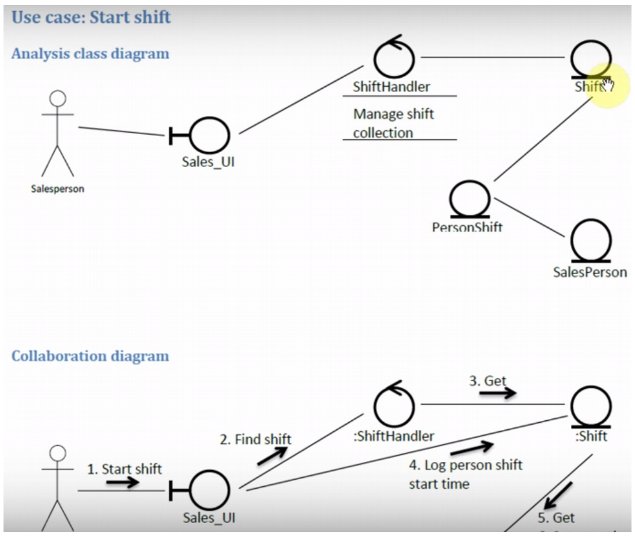
Use case: Plan shift

Analysis class diagram





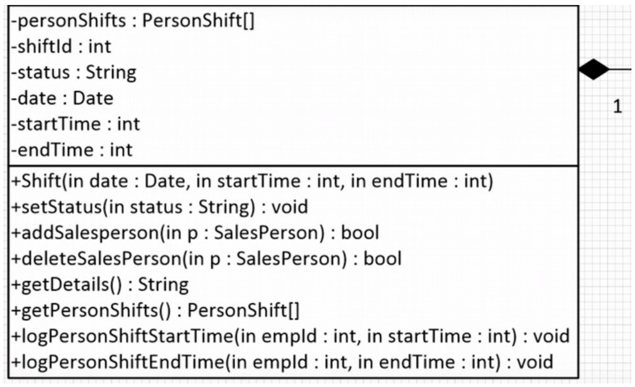
Collaboration diagram



Use case: Start shift

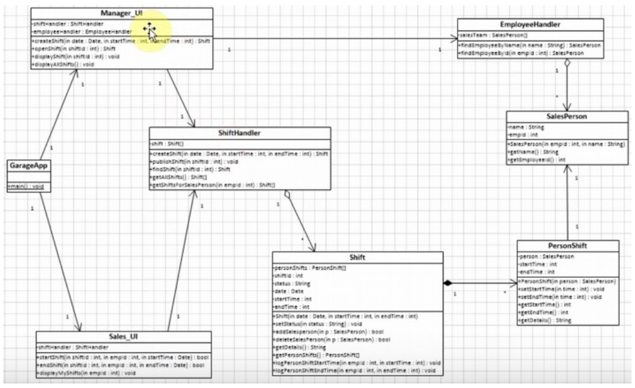
Analysis class diagram

Collaboration diagram

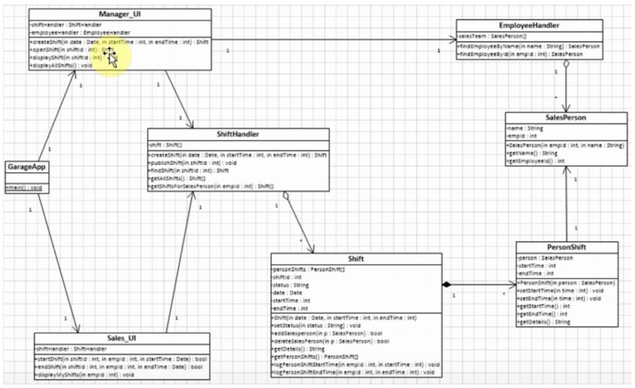


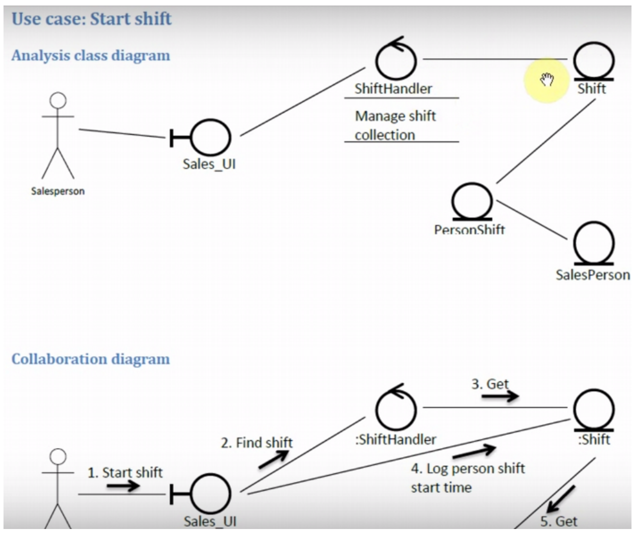
-personShifts : PersonShift[]
-shiftId : int
-status : String
-date : Date
-startTime : int
-endTime : int

+Shift(in date : Date, in startTime : int, in endTime : int)
+setStatus(in status : String) : void
+addSalesperson(in p : SalesPerson) : bool
+deleteSalesPerson(in p : SalesPerson) : bool
+getDetails() : String
+getPersonShifts() : PersonShift[]
+logPersonShiftStartTime(in empId : int, in startTime : int) : void
+logPersonShiftEndTime(in empId : int, in endTime : int) : void



Manager_UI
-shiftHandler : ShiftHandler
-employeeHandler : EmployeeHandler
+createShift(in date : Date, in startTime : int, in endTime : int) : Shift
+openShift(in shiftId : int) : Shift
+displayShift(in shiftId : int) : void
+displayAllShifts() : void
EmployeeHandler
-salesTeam : SalesPerson[]
+findEmployeeByName(in name : String) : SalesPerson
+findEmployeeById(in empId : int) : SalesPerson
GarageApp
+main() : void
ShiftHandler
-shift : Shift[]
+createShift(in date : Date, in startTime : int, in endTime : int) : Shift
+publishShift(in shiftId : int) : void
+findShift(in shiftId : int) : Shift
+getAllShifts() : Shift[]
+getShiftsForSalesPerson(in empId : int) : Shift[]
SalesPerson
-name : String
-empId : int
+SalesPerson(in empId : int, in name : String)
+getName() : String
+getEmployeeId() : int
Shift
-personShifts : PersonShift[]
-shiftId : int
-status : String
-date : Date
-startTime : int
-endTime : int
+Shift(in date : Date, in startTime : int, in endTime : int)
+setStatus(in status : String) : void
+addSalesperson(in p : SalesPerson) : bool
+deleteSalesPerson(in p : SalesPerson) : bool
+getDetails() : String
+getPersonShifts() : PersonShift[]
+logPersonShiftStartTime(in empId : int, in startTime : int) : void
+logPersonShiftEndTime(in empId : int, in endTime : int) : void
PersonShift
-person : SalesPerson
-startTime : int
-endTime : int
+PersonShift(in person : SalesPerson)
+setStartTime(in time : int) : void
+setEndTime(in time : int) : void
+getStartTime() : int
+getEndTime() : int
+getDetails() : String
Sales_UI
-shiftHandler : ShiftHandler
+startShift(in shiftId : int, in empId : int, in startTime : Date) : bool
+endShift(in shiftId : int, in empId : int, in endTime : Date) : bool
+displayMyShifts(in empId : int) : void





Use case: Start shift

Analysis class diagram

Collaboration diagram