UECS2344 Software Design: Lecture 5

Object-Oriented Design Activities

Software Design Activities (not necessarily in the order given below)

- Perform **use case realisations**
  - Use case realisation refers to design of software that implements each use case, focusing on interactions among the different software objects required to accomplish a particular use case

- Design the **software architecture**
  - Architectural design for software is equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows. The floor plan gives an overall view of the house. Architectural design of software gives a high-level view of the software

- Perform **detailed design**
  - Detailed design of software is equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings show wiring and plumbing within each room, the address of wall switches, sinks, etc. They also describe the flooring to be used, and every other detail associated with a room. The detailed design for software fully describes the details of each software module and gives a low level view of the software.

- Design the **database**
  - Usually relational database used – need to design database tables. Alternatively object-oriented database or files can be used

- Design the **support services architecture** and **deployment environment**
  - Support services architecture and deployment environment covers computer hardware (servers, client workstations, etc.), networks, and software systems (operating systems, database management systems, etc.)

- Design the **user and system interfaces**
  - Design the user interface in such a way that is helps the user accomplish his/her tasks easily. The new system may interact or work with other systems – need to design system interfaces to ensure all systems work together

- Design the **controls for the system**
  - Controls include:
    - user access controls – may need to limit access to authorised users
    - system controls - ensure other systems cause no harm to this system
    - application controls – ensure transactions are recorded and work done by system are done correctly

- database controls – ensure data are protected from unauthorised access and from accidently loss due to software or hardware failure
- network controls – ensure communication through networks is protected

Domain-Driven Design – created by Eric Evans

Making sense of user requirements is sometimes hard. Also, the inability to completely understand user requirements is probably the primary cause of misunderstandings between business and development teams.

- Developers write software for a specific business but they might not be experts of the specific business domain.
- Likewise, domain or business experts might have some knowledge of software development but probably not enough to avoid misunderstandings and incorrect assumptions.

Domain-Driven Design is about gaining knowledge about a given problem domain in order to better understand the user requirements.

- The problem domain is how an organisation conducts its business.
- The ultimate objective is that the system to be built is based on the problem domain.

Domain-driven design results in a model (domain model) that mirrors the problem domain.

- In the domain model, you focus on business concepts and business processes.
- The domain model provides a conceptual view of the business of the organisation.
- It may be represented with an Analysis Class Diagram.

Use Case Realisation - Moving from Analysis to Design

Use case realisation can be used to move from analysis to design.

- It starts with use cases.

General Steps for Use Case Realisation:

1. Create an **Analysis Class Diagram** to represent the Domain Model.

2. Create a **Use Case Diagram** to identify the actors and the use cases.

3. **For each use case**:
   - Develop a **Prototype Screen** (which could be a paper-based sketch of the user interface) to define the inputs and outputs for the use case. Remember that the user interface is the means through which the actors accomplish their tasks and interact with the system.
   - Develop a **Use Case Description**. The use case description focuses on the interactions between the actor and the system. It documents the user's actions and the system responses.
   - Develop a **System Sequence Diagram**. The system sequence diagram shows the flow of events in a use case description at an abstract (system) level. (Alternatively, an **Activity Diagram** may be used to represent the flow of events in a use case.)
   - Identify the objects that collaborate to accomplish the use case.
   - Draw a **Sequence Diagram** to show how the objects collaborate. (Alternatively, a **Communication Diagram** can be drawn.)

- Draw a **Design Class Diagram** showing the classes of the objects that collaborate in that use case and the operations of the classes corresponding to the messages in the sequence diagram.

Repeat the process for each use case and expand the Design Class Diagram with new operations and attributes as needed.

Design Object Categories – developed by Ivar Jacobson

During design, the objects that will accomplish the use case must be identified.

To help identify the objects and their classes, objects can be categorised into 3 types:

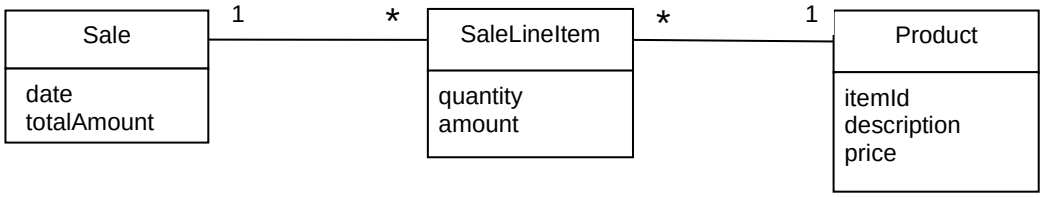
- **Entity objects** – objects that are represented in the domain model; they represent real-world objects.
- **Boundary objects** – objects that are used to model the interactions between the system and its actors.
- **Control objects** – objects that are used in the coordination of the use cases.

The UML notation uses the following stereotypes for the 3 types of classes:

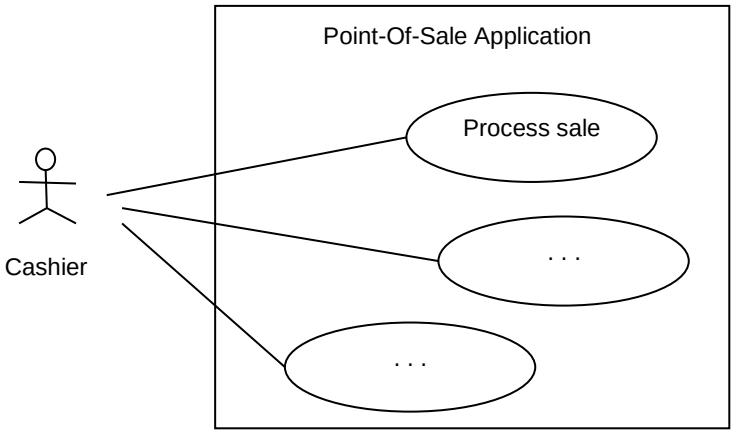
Use Case Realisation Examples

Example 1: Point-Of-Sale (POS) Application

Analysis Class Diagram (Domain Model)



Use Case Diagram (partial / incomplete)



Use Case: Process sale

Prototype Screen

```
====New Sale====
Enter item identifier: xxxxx
Enter quantity: xx
Item Description: xxxxxxxxxxxxxxx
Price: xxx.xx
Amount: xxxx.xx
Total: xxxxxx.xx
Enter item identifier: xxxxx
. . .
Total: xxxxxx.xx
. . .
```

Use Case Description (partial - with main success scenario only)

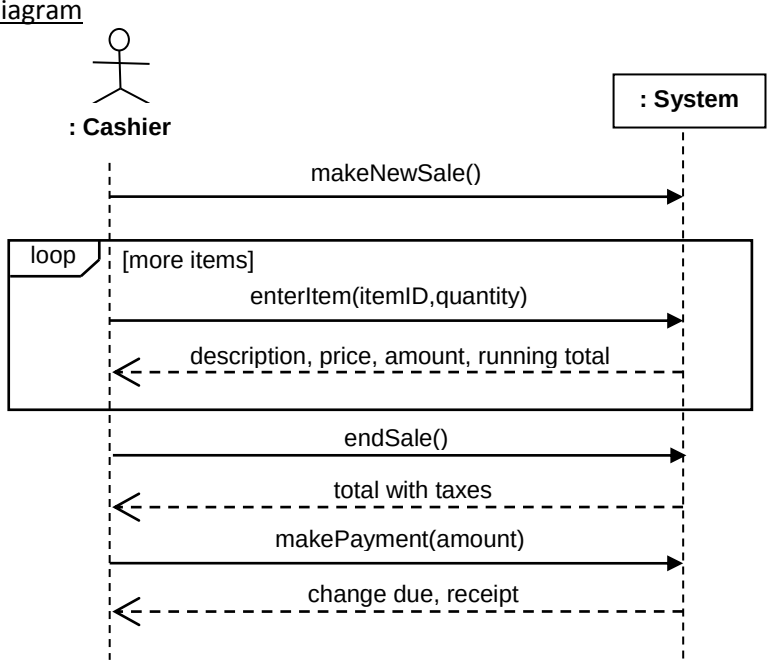
Flow of Events:

1. Cashier starts a new sale.
2. Cashier enters item identifier and quantity.
3. System presents item description, price, amount, and running total and records sale line item.

Cashier repeats steps 2-3 while there are more items.

4. Cashier ends the sale.
5. System presents total with taxes calculated.
6. Cashier handles payment.
7. System presents change due and receipt.

System Sequence Diagram

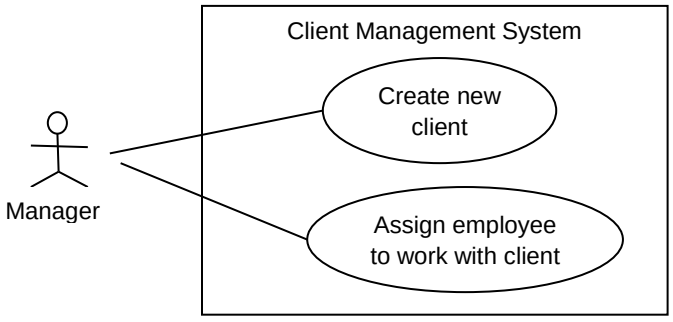


(Remaining steps of Use Case Realisation are not shown)

Example 2: Client Management System

Analysis Class Diagram (Domain Model)

Use Case Diagram



Use Case: Create new client

Prototype Screen

```
====Create New Client====
Enter client name: xxxxxxxxxx
Enter client location: xxxxxxxxxxxxxxxxxxxxxxxxxx
New client added
```

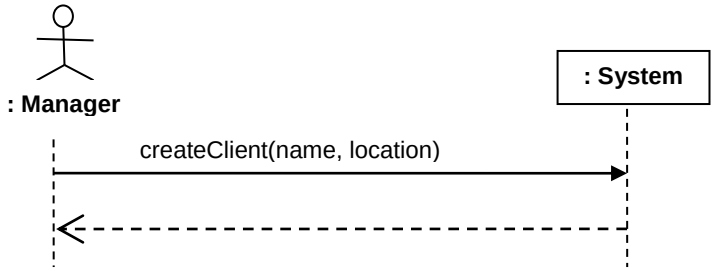
```
======Create New Client======
Client name:      xxxxxxxxx
Client location:  xxxxxxxxxxxxxxxxx
                  Create Client
New client added
```

Use Case Description (partial - with main success scenario only)

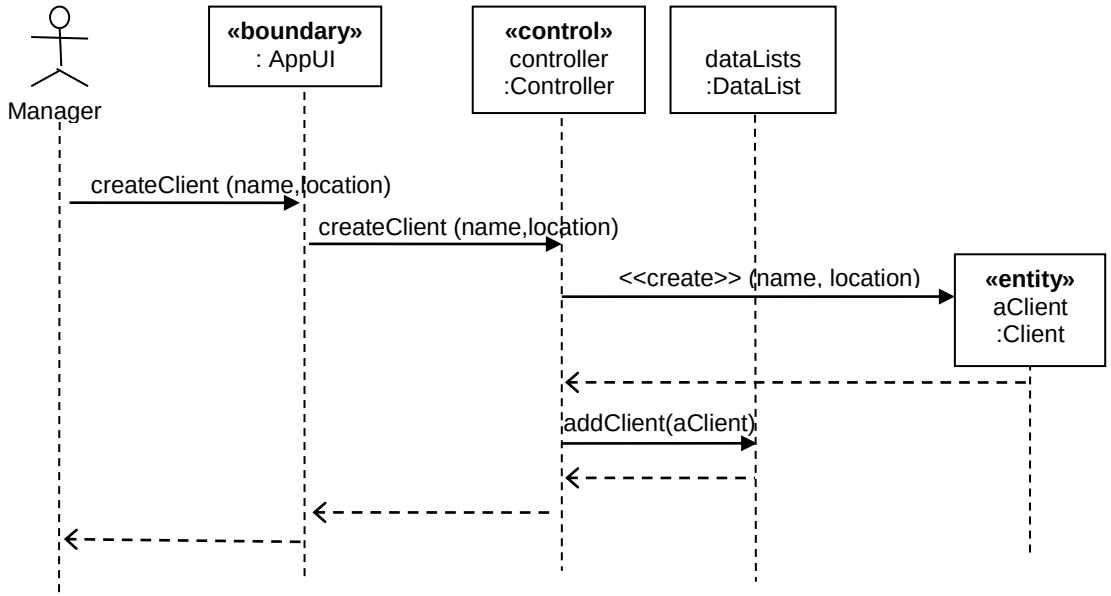
Flow of Events:

1. Manager enters new client name and location.
2. System records new client details.

System Sequence Diagram



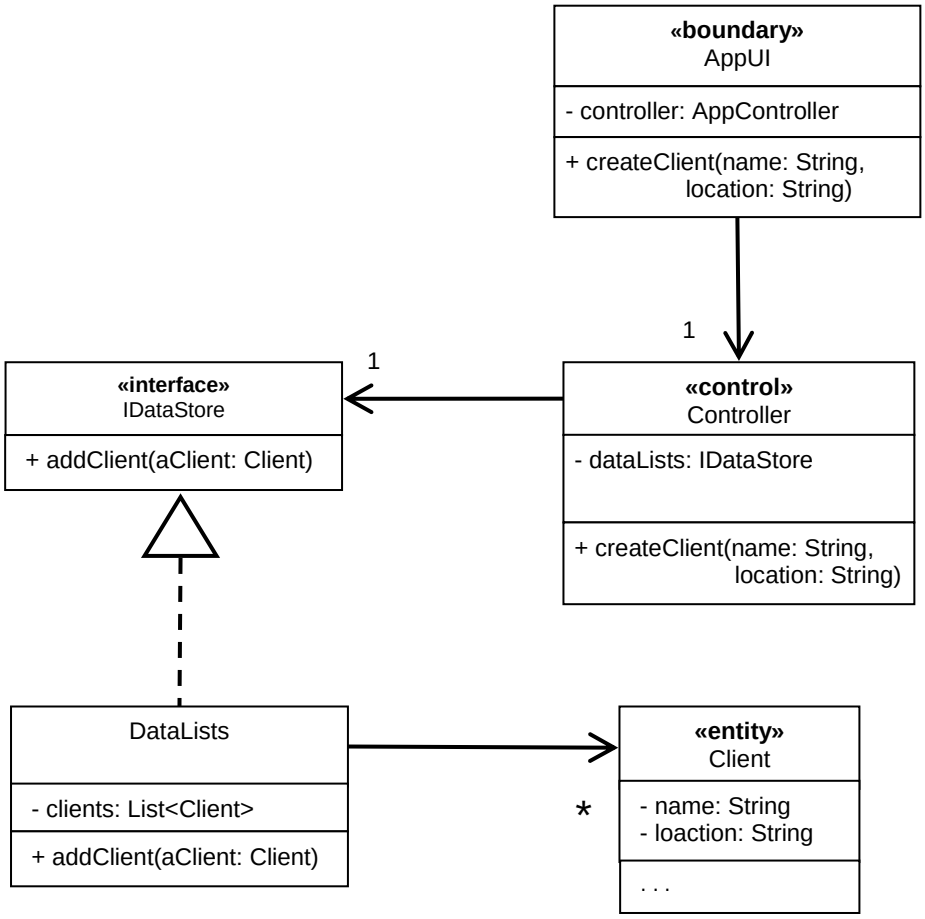
Sequence Diagram



Note:

- The boundary object receives requests and delegates requests to the control object.
- The Scanner and ArrayList objects are not shown.

Initial Design Class Diagram



Note: The Operations in the Classes must correspond to the messages in the Sequence Diagram.

Code

```
// boundary object class
public class AppUI . . .

   // reference to the control object
   private Controller controller;


   . . .

   public void createClient() {
       // get data of a new client
       String name = . . .
       . . .
       String address = . . .

       // request Controller object to create client
       controller.createClient(name, address);
    }
   . . .
}
```

```
// control object class
public class Controller {

   // reference to list of objects
   private IDataStore dataLists;

   . . .

   public Controller() {
       dataLists = new DataLists();
   }

   . . .

   public void createClient(String name, String location) {
       Client aClient = new Client(name, location);
       dataLists.addClient(aClient);
   }
   . . .
}

public interface IDataStore {
    public void addClient(Client client);

    . . .
}

public class DataLists implements IDataStore {

      private List<Client> clients;

      public DataLists() {
            clients = new ArrayList<Client>();
      }

      public void addClient(Client aClient) {
            clients.add(aClient);
      }

      . . .
}
```

Use Case: Assign employee to work with client

Prototype Screen

```
====Assign Employee for Client====
Enter client name: xxxxxxxxxx
Enter employee name: xxxxxxxxxxxxx
Employee assigned to client
```

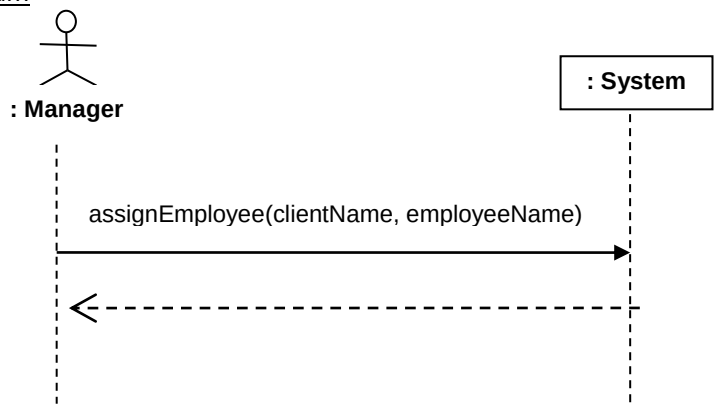
```
=====Assign Employee to Client=====
Client name:      xxxxxxxxx
Employee name:    xxxxxxxxxxxxxxxxxx
                  Assign Employee
Employee assigned to client
```

Use Case Description (partial - with main success scenario only)

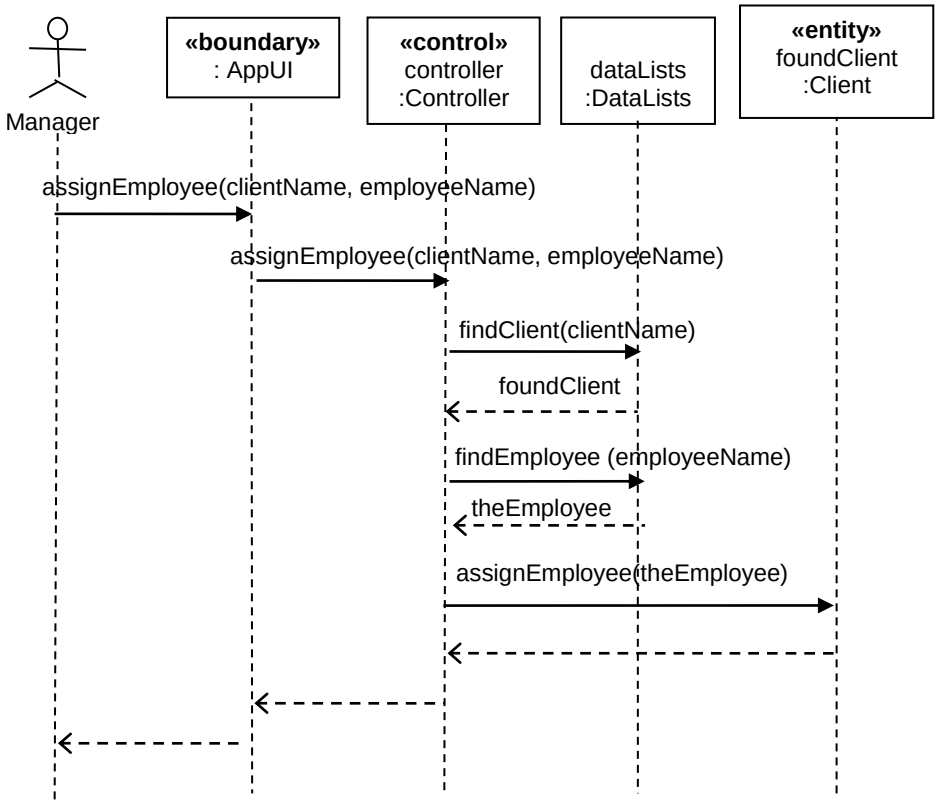
Flow of Events:

1. Manager enters name of client.
2. System finds client record.
3. Manager enters name of employee to assign to client.
4. System finds employee record and assigns employee to client.

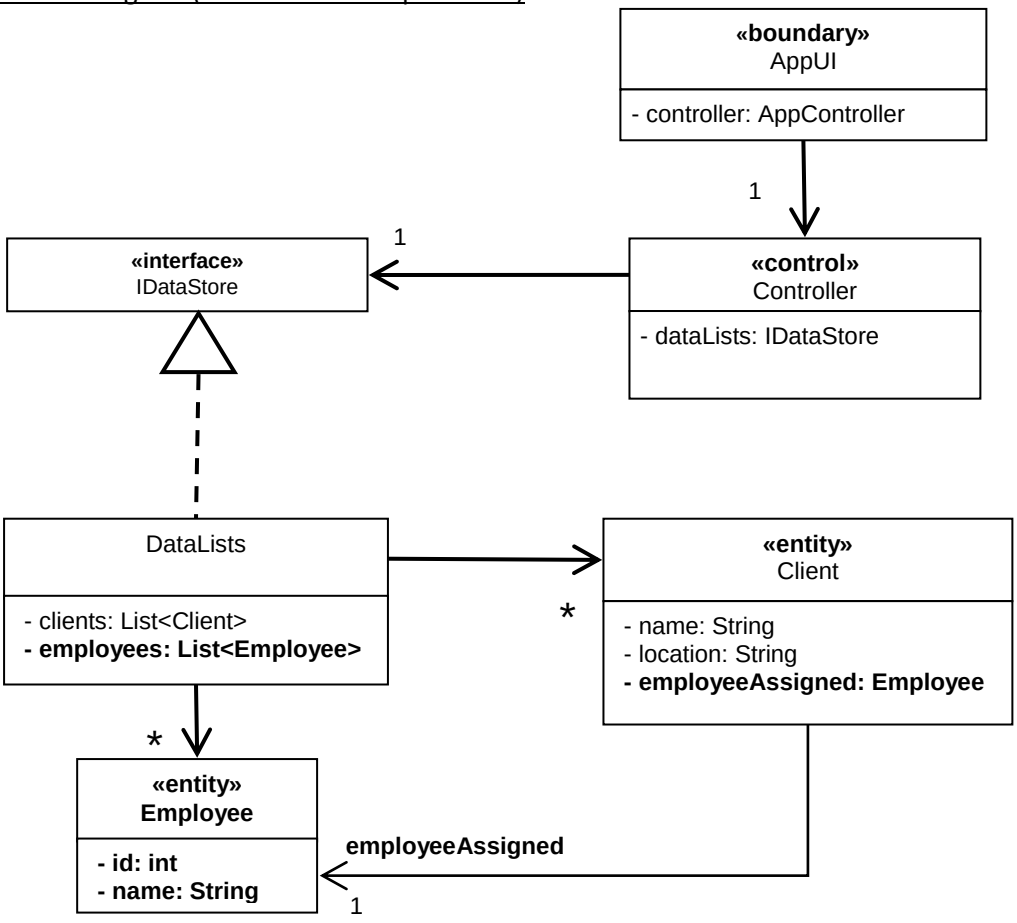
System Sequence Diagram



Sequence Diagram



Design Class Diagram (shown without operations)



Code

The IDataStore interface and DataLists class are modified to manage a list of clients as well as a list of employees.

```
public interface IDataStore {

    public void addClient(Client client);

    public Client findClient(String name);

    public Employee findEmployee(String name);

    . . .
}

public class DataLists implements IDataStore {

    private List<Client> clients;

    private List<Employee> employees;

     public DataLists() {
          clients = new ArrayList<Client>();
          employees = new ArrayList<Employee>();
     }

     public void addClient(Client client) {
          . . .
     }

     public Client findClient(String name) {
          . . .
     }

     public Employee findEmployee(String name) {
          . . .
     }


      . . .
}
```