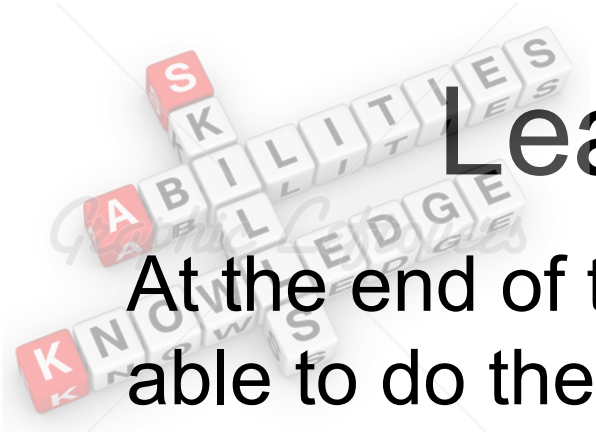
50.003
Elements of Software Construction
Lecture 6

UML Design Model/Solution Class Diagrams

Scope

- Design (Solution) Model
- convert the responsibilities of the analysis objects in the sequence diagrams into operations (methods) in the design model
- 2 fundamental design principles of cohesion and coupling
- Completing the Solution Class Diagram
- Mapping to code



Learning Outcomes

At the end of this session, you should be able to do the following:

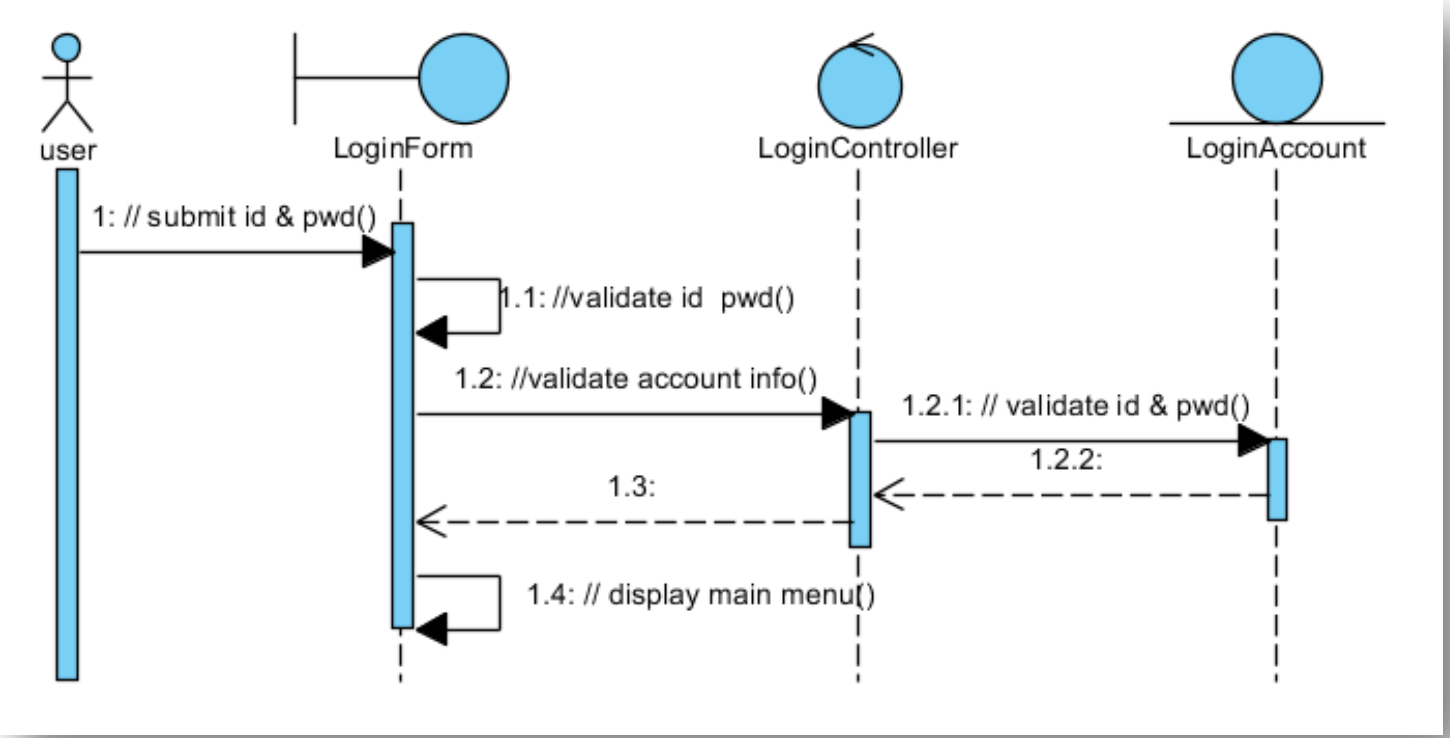
- make use of all the detail sequence diagrams of all the use cases of the intended system to consolidate the Design (Solution) Model
- convert the responsibilities of the analysis objects in the sequence diagrams into operations (methods) in the design model
- explain and differentiate the 2 fundamental design principles of cohesion and coupling
- explain the role of the solution classes in the implementation stage

Learning Outcomes

- apply the 2 fundamental design principles of cohesion and coupling to obtain a more efficient design model
- complete the class design by refining the attributes and the operations
- construct the overall design class diagram of the intended system
- package the overall design class diagram

Software Models for Use Case Realization

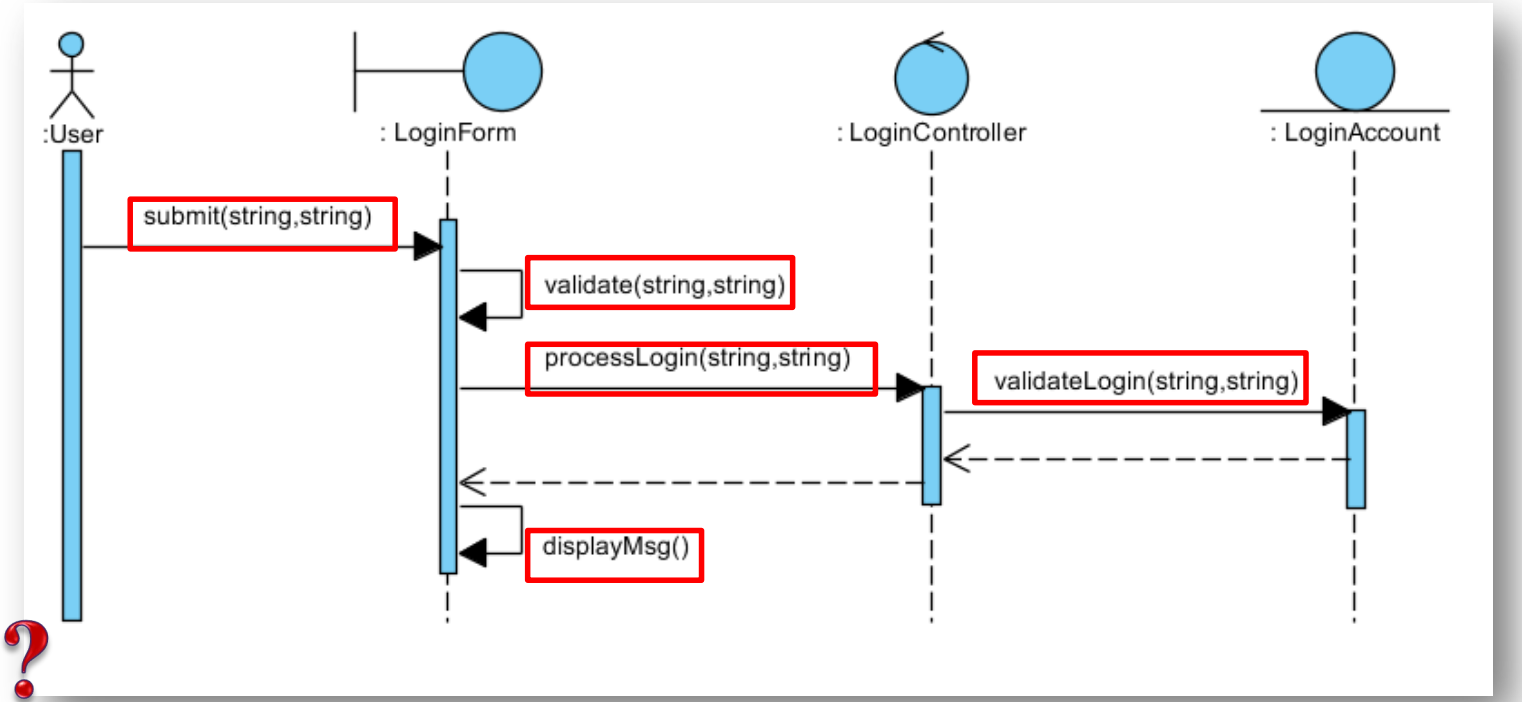
- **Analysis model**
  - Analysis classes with essential attributes , high-level responsibilities and brief relationships among classes.



Software Models for Use Case Realization

- **Design model**
  - Design classes (a.k.a. software classes) with details of attributes and operations, and detailed relationships between software classes.

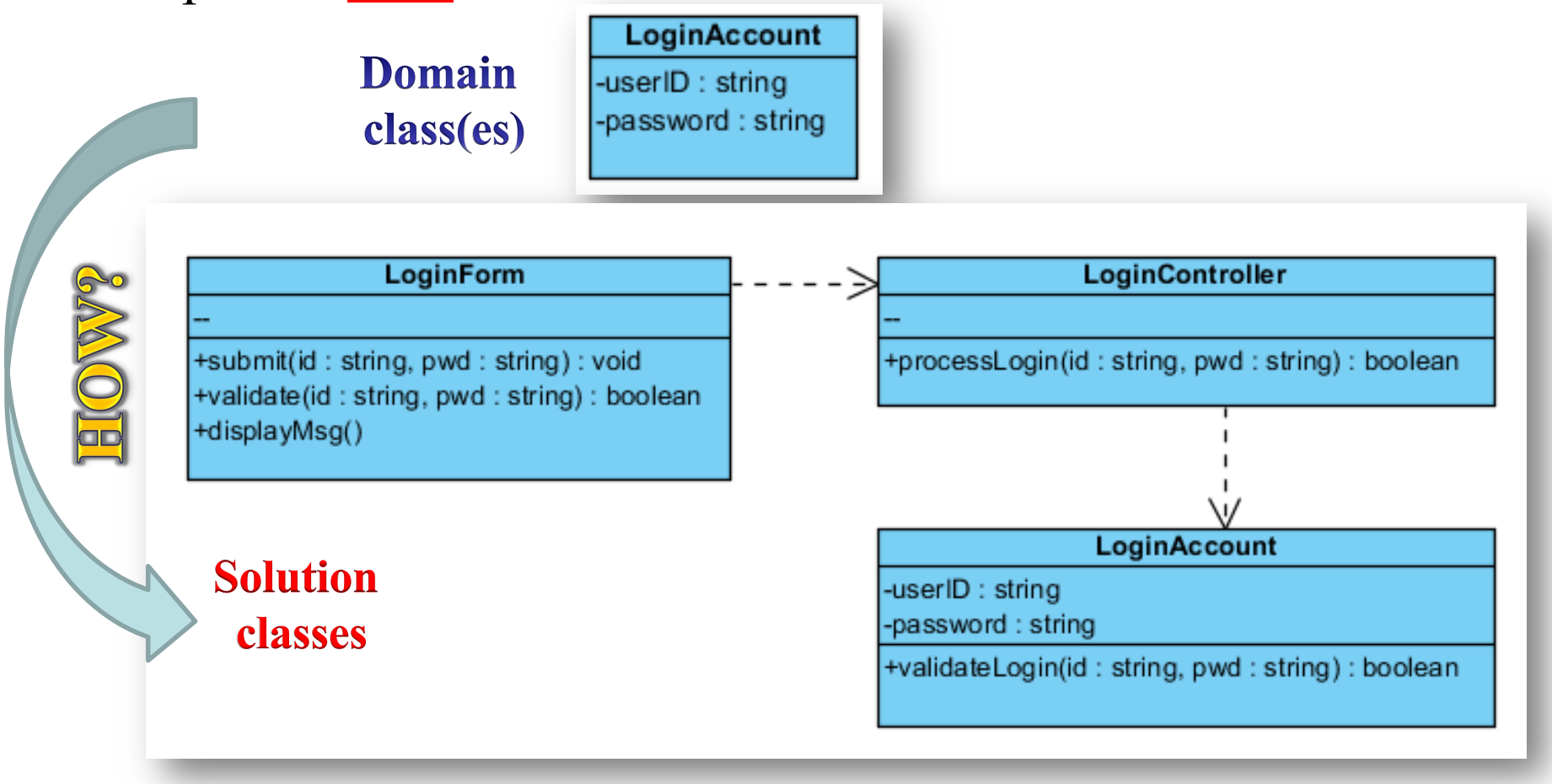
Observe the change from responsibilities to actual method definitions



Purpose??

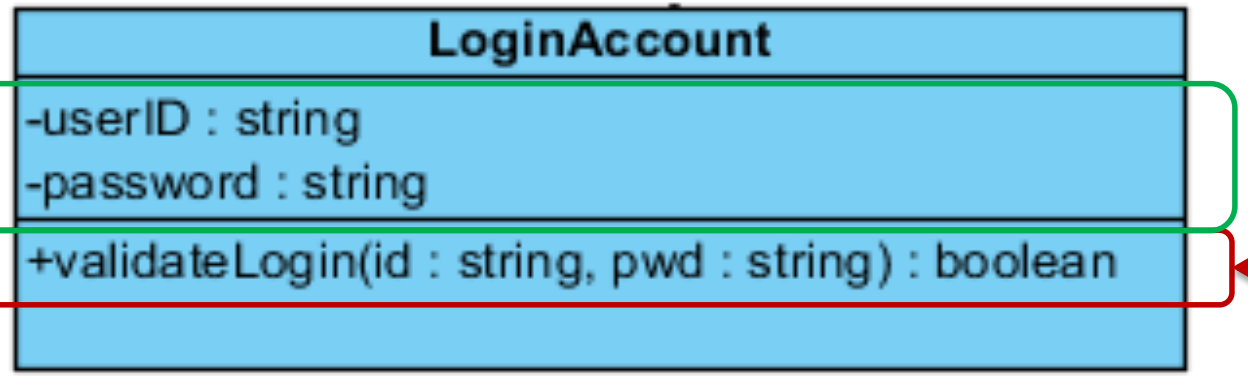
From Domain class diagram to Design/Solution class diagram

Ultimately we want to have a design class diagram with details that can help us to code.



Details needed when specifying a design (solution) class

From domain model and sequence diagram

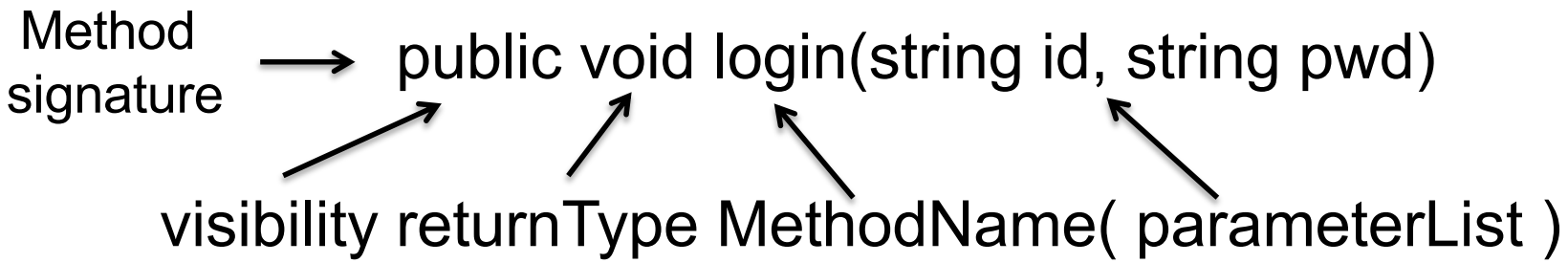


From sequence diagram

- Visibility
  - \+ for Public class members and
  - – for Private class members
- Name of operations
- Type of attributes
  - float, integer, string, boolean etc

- Parameter_List
  - (id, pwd)
- Return types of operations

Details needed when converting high level responsibility to operation



- Visibility
  - + for Public class members and
  - – for Private class members
- Name of operations
- Type of attributes
  - float, integer, string, boolean etc.

- Parameter_List
  - (id, pwd)
- Return types of operations

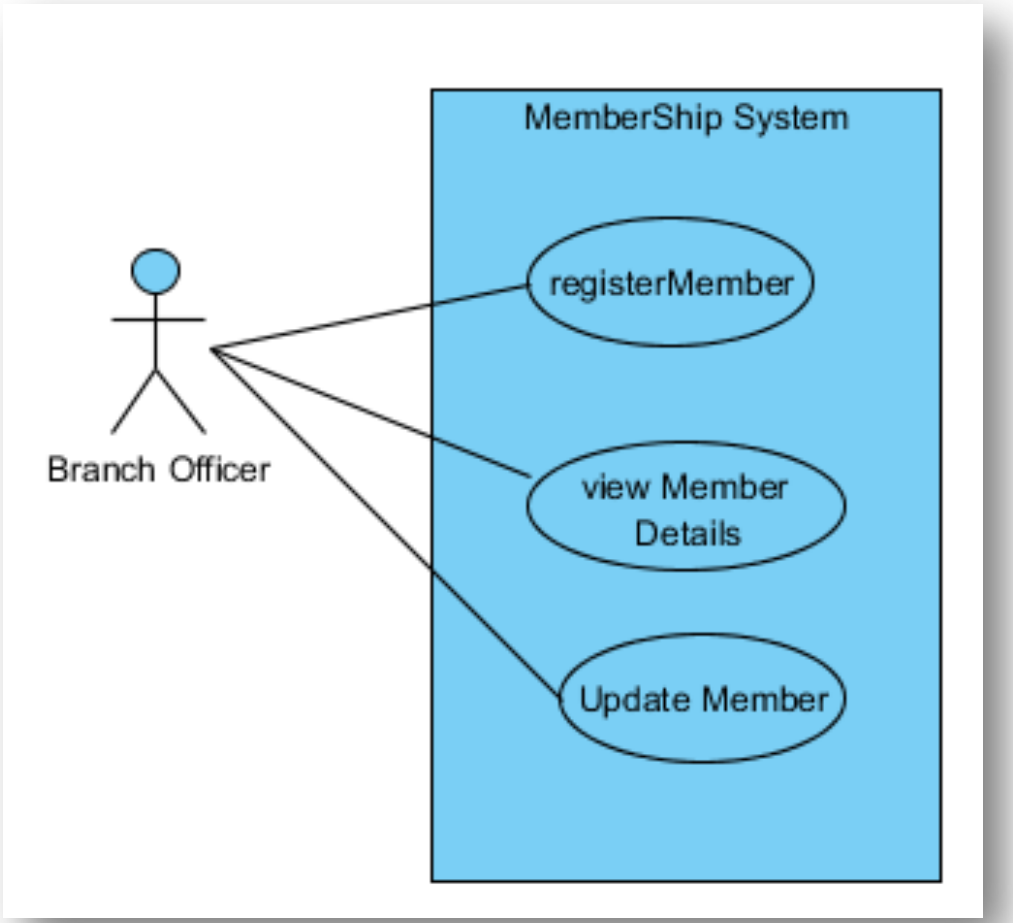
Steps to derive design (solution) class diagram from the sequence diagrams

1. Convert analysis objects into design classes by mapping the responsibilities into operations
2. Consolidate the details from all design class diagrams to form one overall solution (system) class diagram
3. Refine overall system class diagram using 2 fundamental design principles : cohesion and coupling

Note that these steps may be iterative.

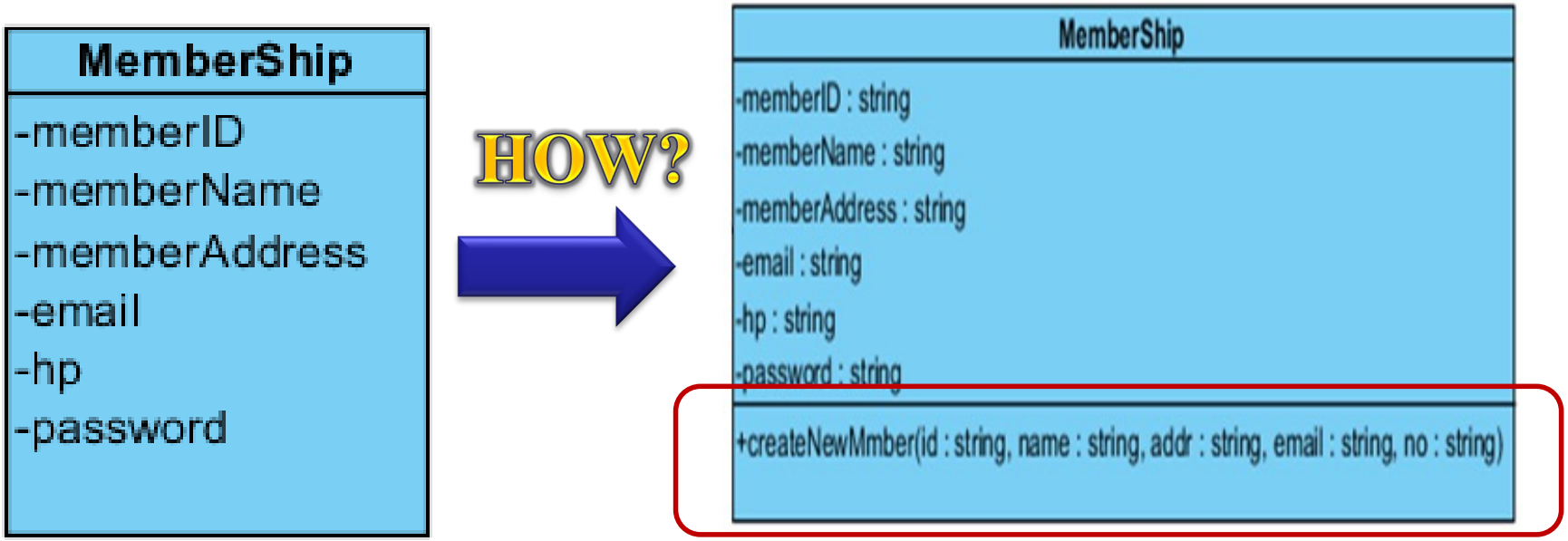
Example – Membership system

- Assuming we have to design and build a MemberShip system with the functionalities as shown in the use case diagram on the right



Use case diagram

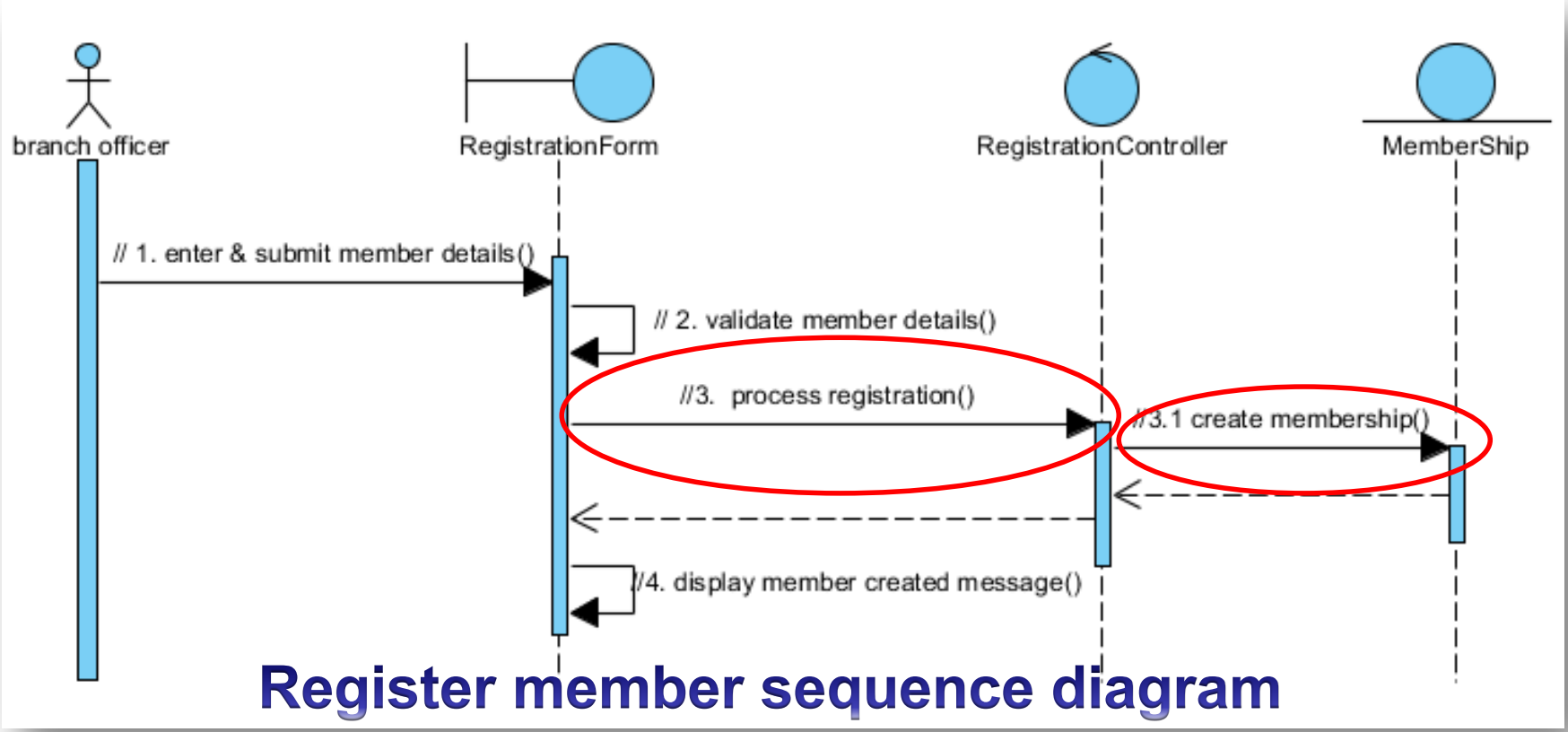
Example – Membership system



Domain class diagram

Design class diagram

Register Member Use Case



Register member sequence diagram

Let's consider the responsibilities assigned to the **controller** and the **entity** class.

Step 1 – convert analysis objects into design classes and map the responsibilities into methods for each use case

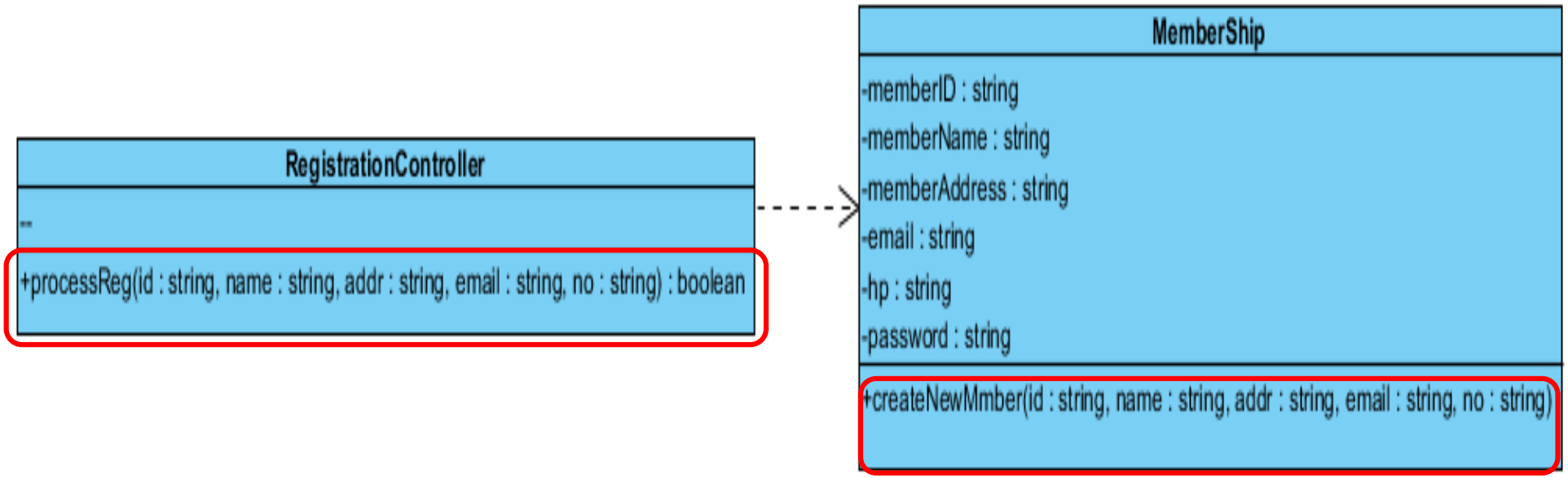
RegistrationController

Responsibilities	Operation definition
//process registration()	

MemberShip

Responsibilities	Operation definition
//create membership()	

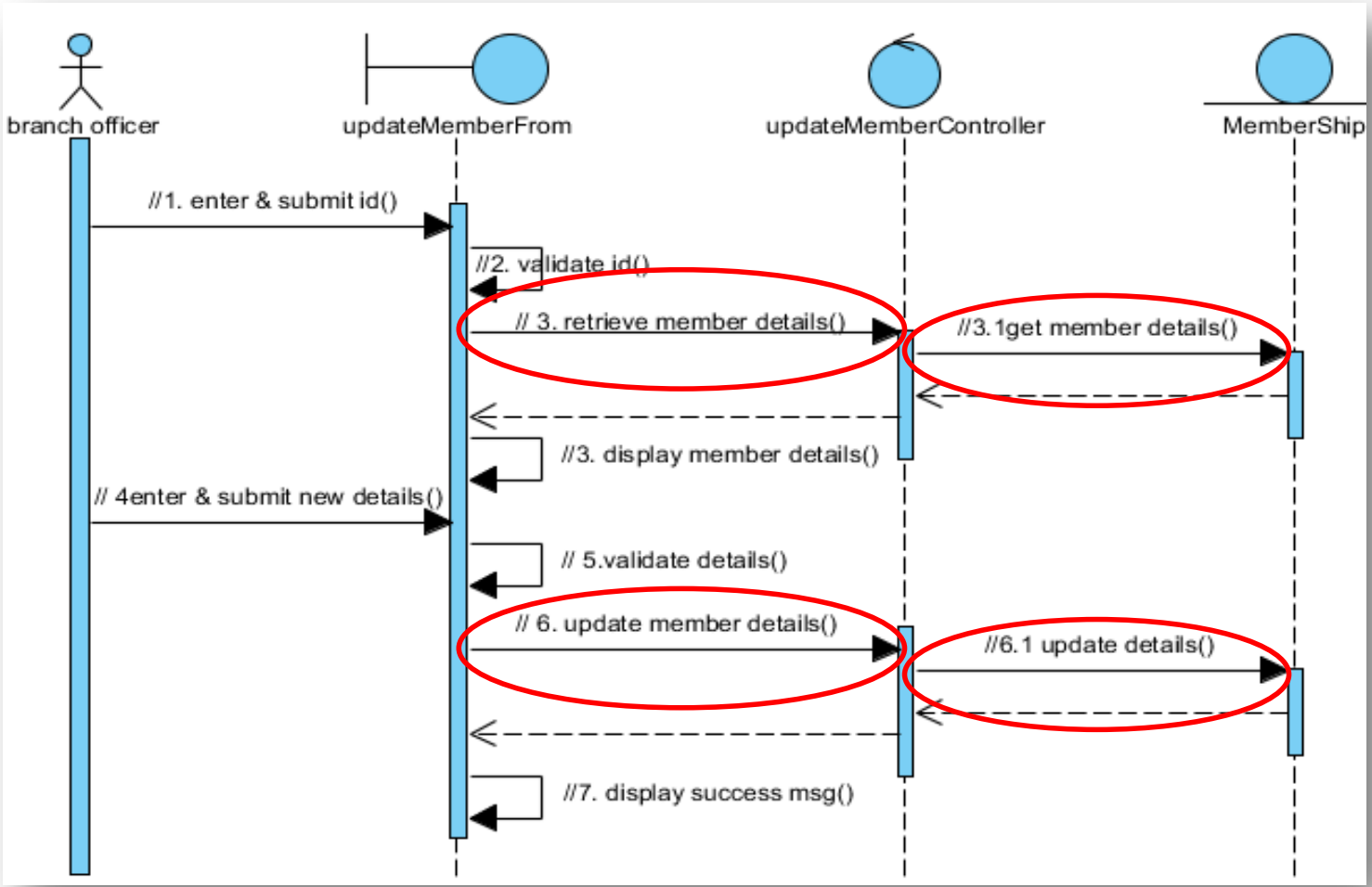
Step 1 – convert analysis objects into design classes and map the responsibilities into methods for each use case



Register member design classes

Now we do the same for the other use cases: update member and view member details

Update Member Use Case



Update member sequence diagram

Step 1 – convert analysis objects into design classes and map the responsibilities into methods for each use case

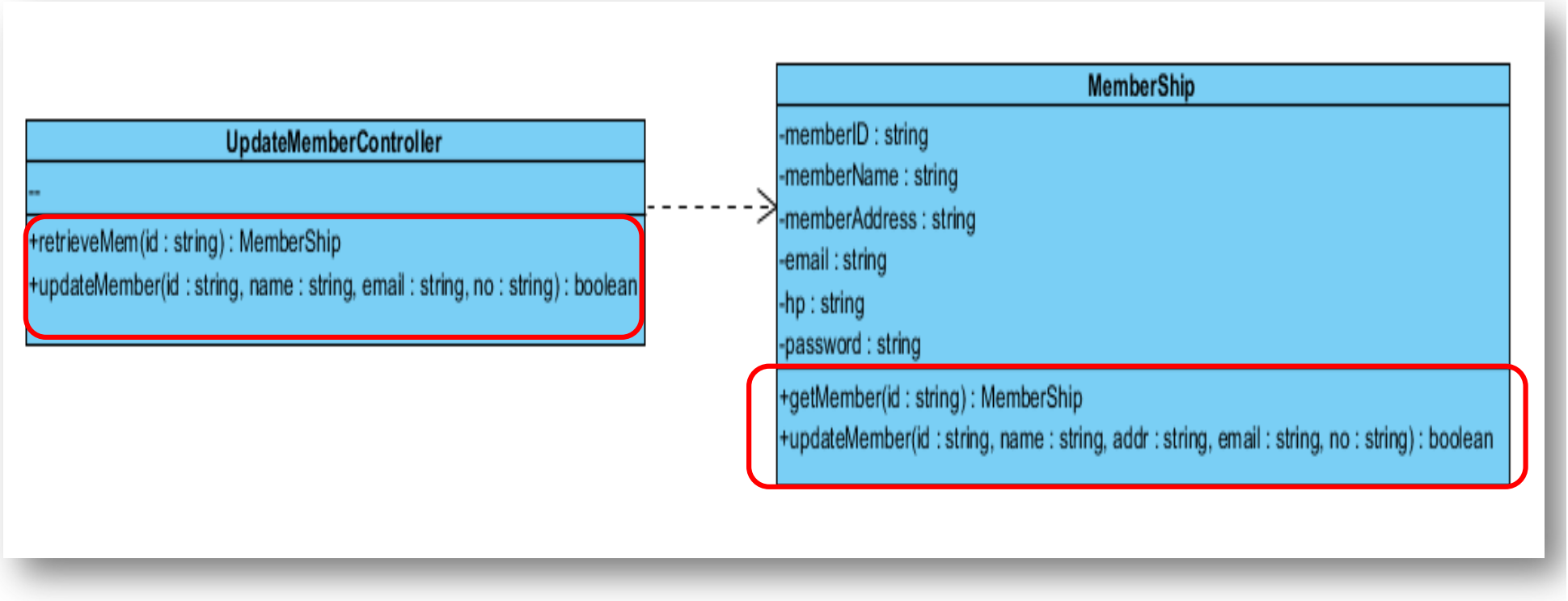
UpdateMemberController

Responsibilities	Operation definition
//retrieve member details()	retrieveMem(string id) : Member
// update member details()	updateMember(string id, string name, string addr, string email, string no) : boolean

MemberShip

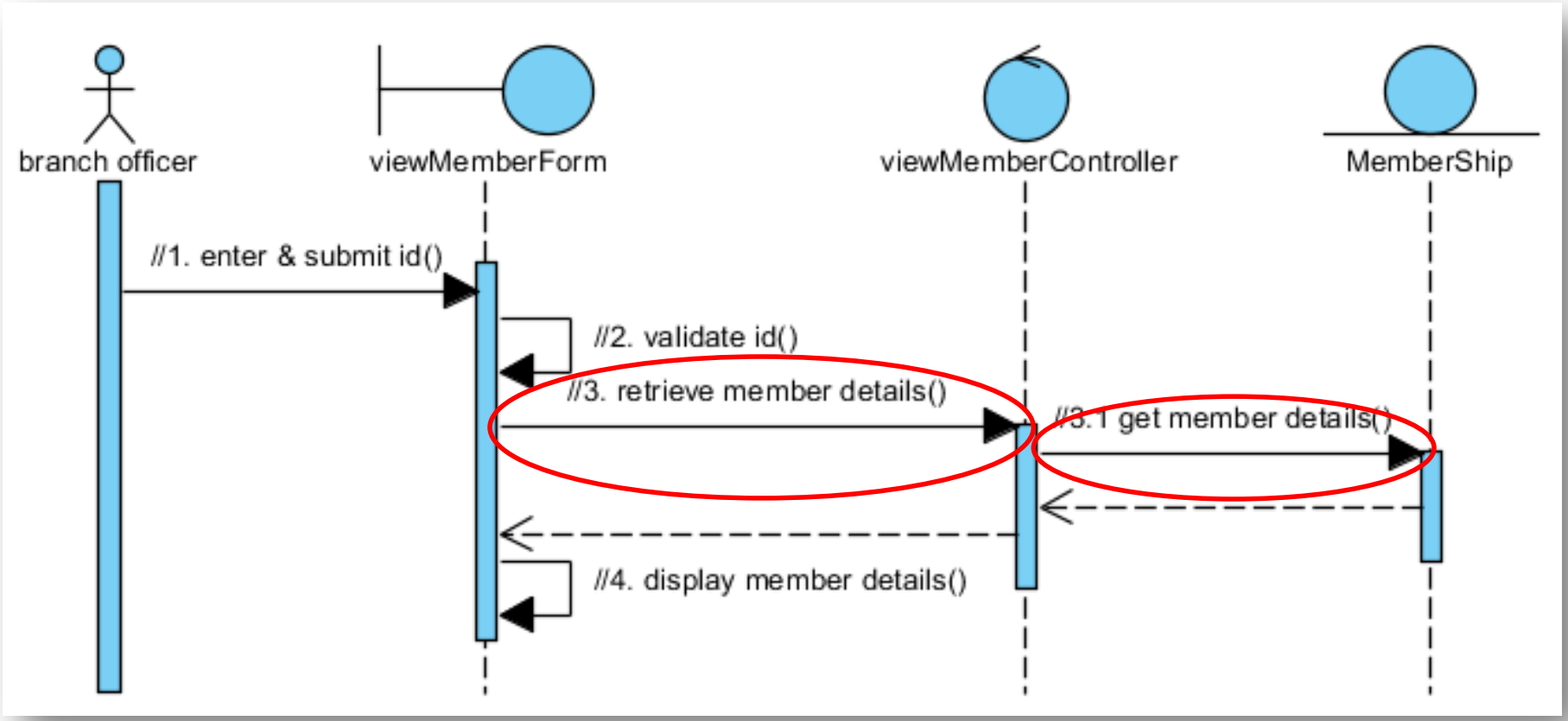
Responsibilities	Operation definition
//get member details()	getMember(string id) : Member
//update details()	updateMember(string id, string name, string addr, string email, string no) : boolean

Step 1 – convert analysis objects into design classes and map the responsibilities into methods for each use case



Update member design classes

View Member Details Use Case



View member details sequence diagram

Step 1 – convert analysis objects into design classes and map the responsibilities into methods for each use case

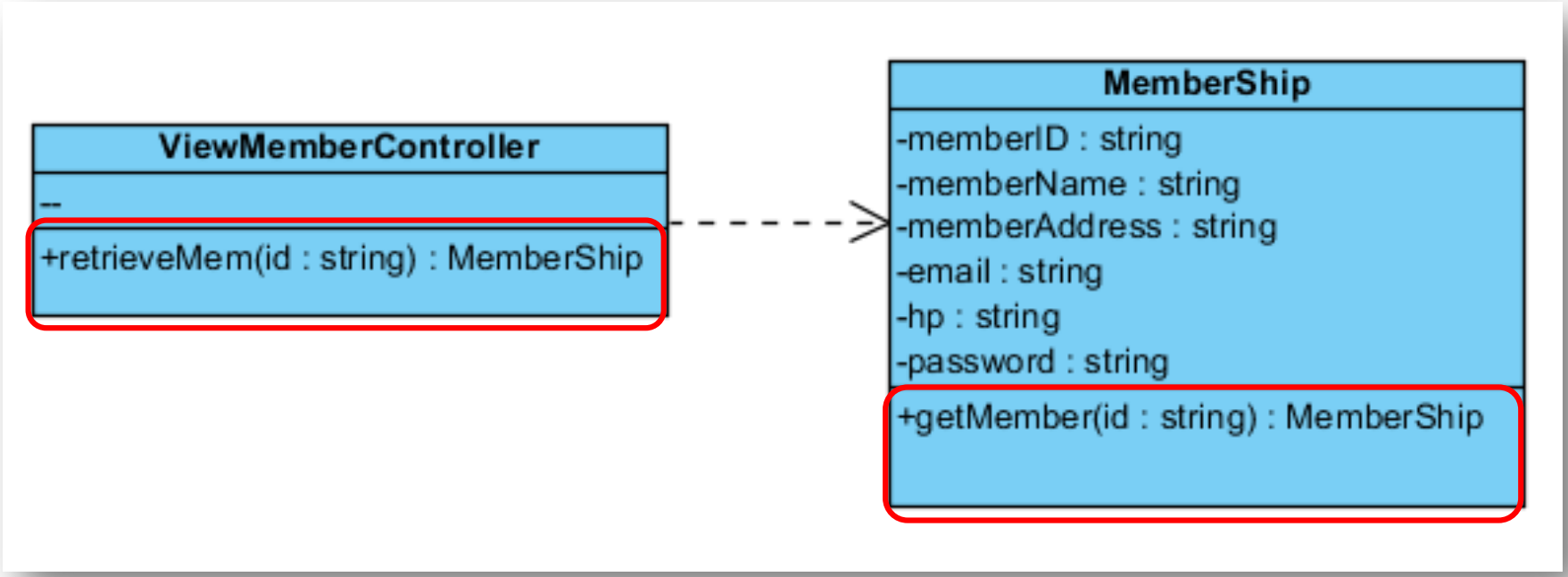
ViewMemberController

Responsibilities	Operation definition
//retrieve member details()	retrieveMem(string id) : Member

MemberShip

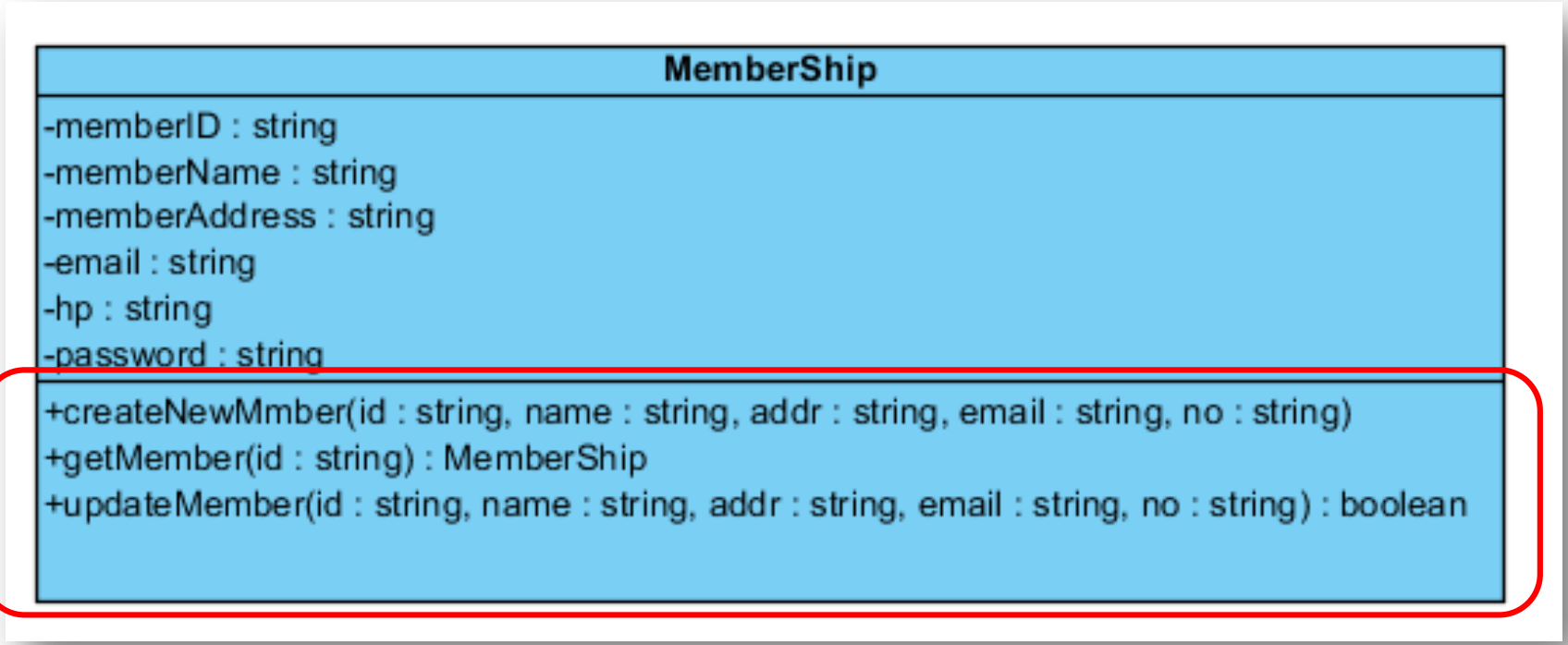
Responsibilities	Operation definition
//get member details()	getMember(string id) : Member

Step 1 – convert analysis objects into design classes and map the responsibilities into methods for each use case



View member design classes

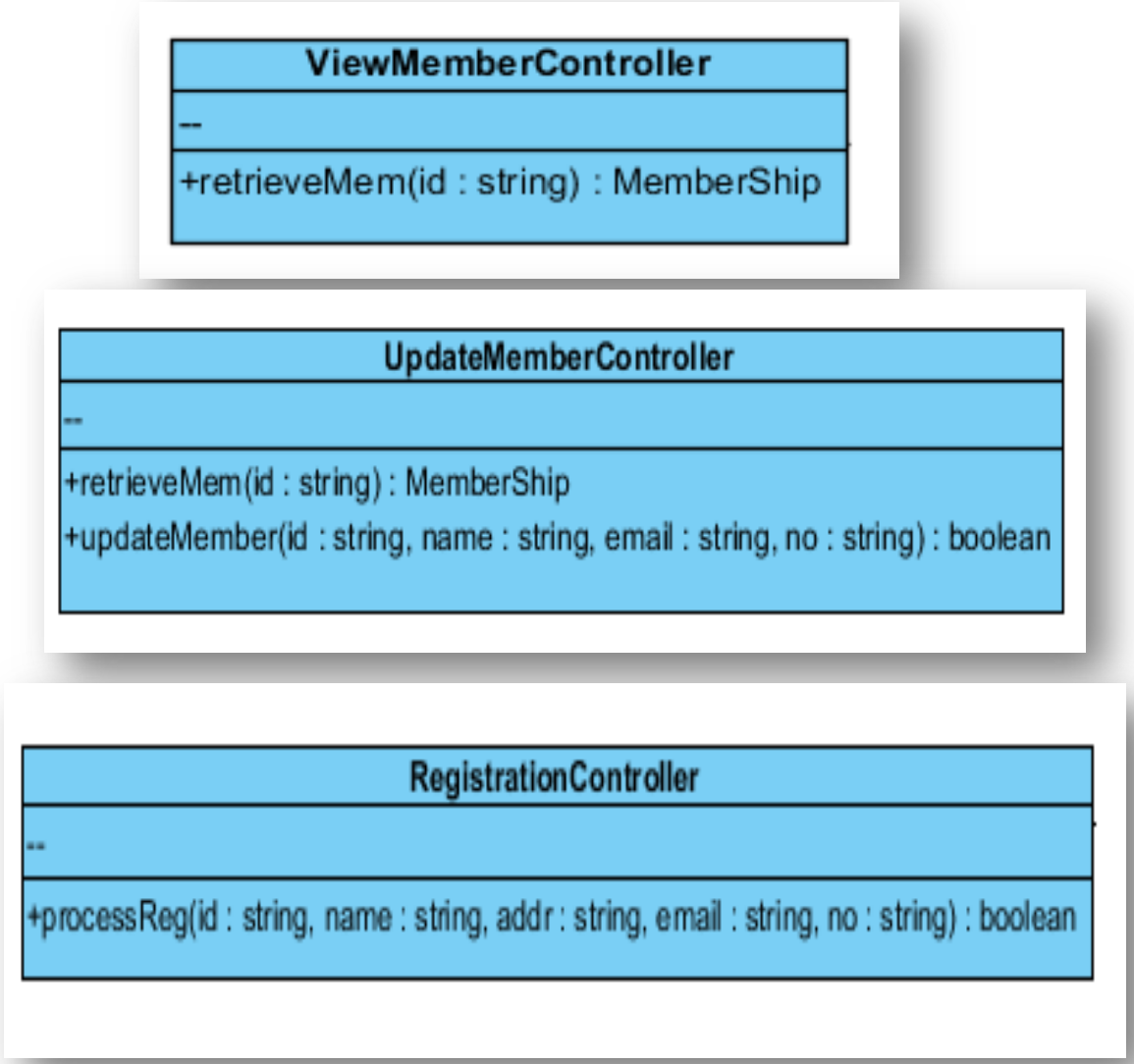
Step 2 –consolidate the details from all class diagrams to form one overall system class diagram



All the MemberShip operations from the 3 use cases are combined into the MemberShip Design Class

The Membership class looks ok

Step 2 –consolidate the details from all class diagrams to form one overall system class diagram



But how about the controller classes? Each one has only one method.

Can we combine all these methods into one controller class? After all, they are handling Membership records.

This will lead us to next issue: cohesion and coupling.

Step 3 – Apply 2 fundamental design principles : cohesion and coupling

- Cohesion:
  - Measures how focused a class is, that means; is the class doing what it is supposed to do?
  - The higher the functional cohesiveness, the better is the class design.
- Separation of responsibility
  - Related to cohesion.
  - Object responsibilities are identified when drawing the sequence diagrams

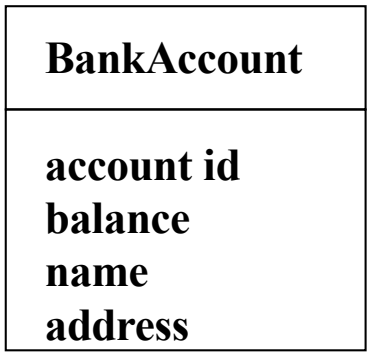
Step 3 – Considering 2 fundamental design principles : cohesion and coupling

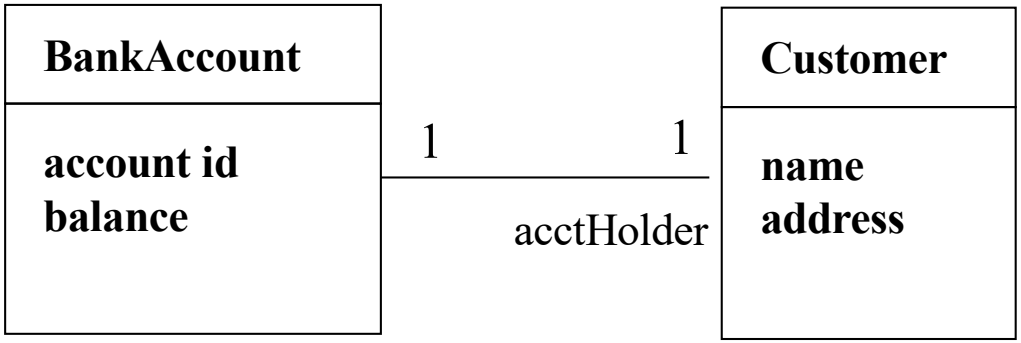
- **Coupling**:
  - Coupling indicates how one class is dependent on other classes.
  - The more dependent a class is on other classes, the higher the coupling.
  - High coupling creates ripple effect when changes are introduced which is **NOT** desired.
- We want to achieve **HIGH cohesion** and **LOW coupling**.

Separation of concerns (responsibilities)

- The process of separating the design of classes into distinct features that minimalize overlap in functionality as much as possible.
- Concerns (responsibilities) refers to features or behaviours of a class.
- An example of separation of concerns is the 3-tier system architecture:
  - Presentation layer : concerns of this layer : user interface
  - Business logic layer: concerns of this layer- processing logic
  - Data layer: concerns of this layer is handing data manipulation

Example of High Cohesion





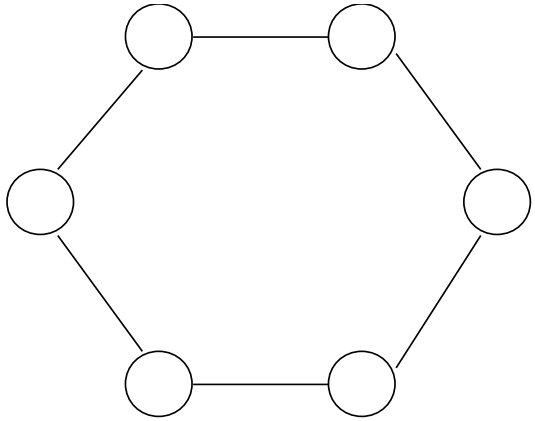
Low cohesion (bad design, why?)

High cohesiveness (better design)
Don't worry about how to know who owns the account.
It is taken care of by the association relationship.

Another good reason for achieving low coupling

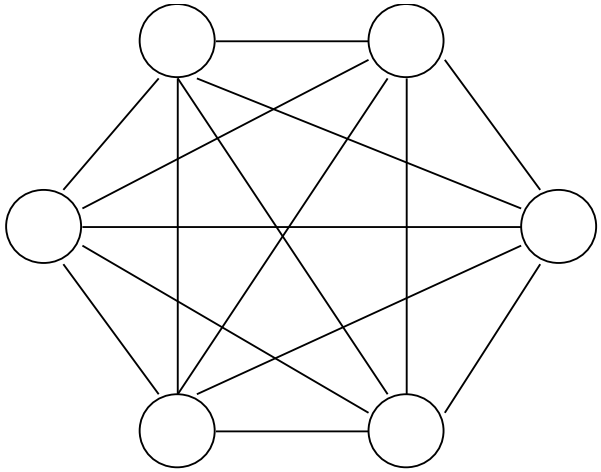
- A class with high (or strong) coupling relies on many other class. It is not desirable because:
  a) Changes in related classes force local changes
  b) Harder to understand in isolation
  c) Harder to reuse because its use requires the additional presence of the classes on which it is dependent
- But it does not mean no coupling is good.
  – in OO system, objects are supposed to interact with one another to execute the tasks in a system.

Another good reason for achieving low coupling



(a)

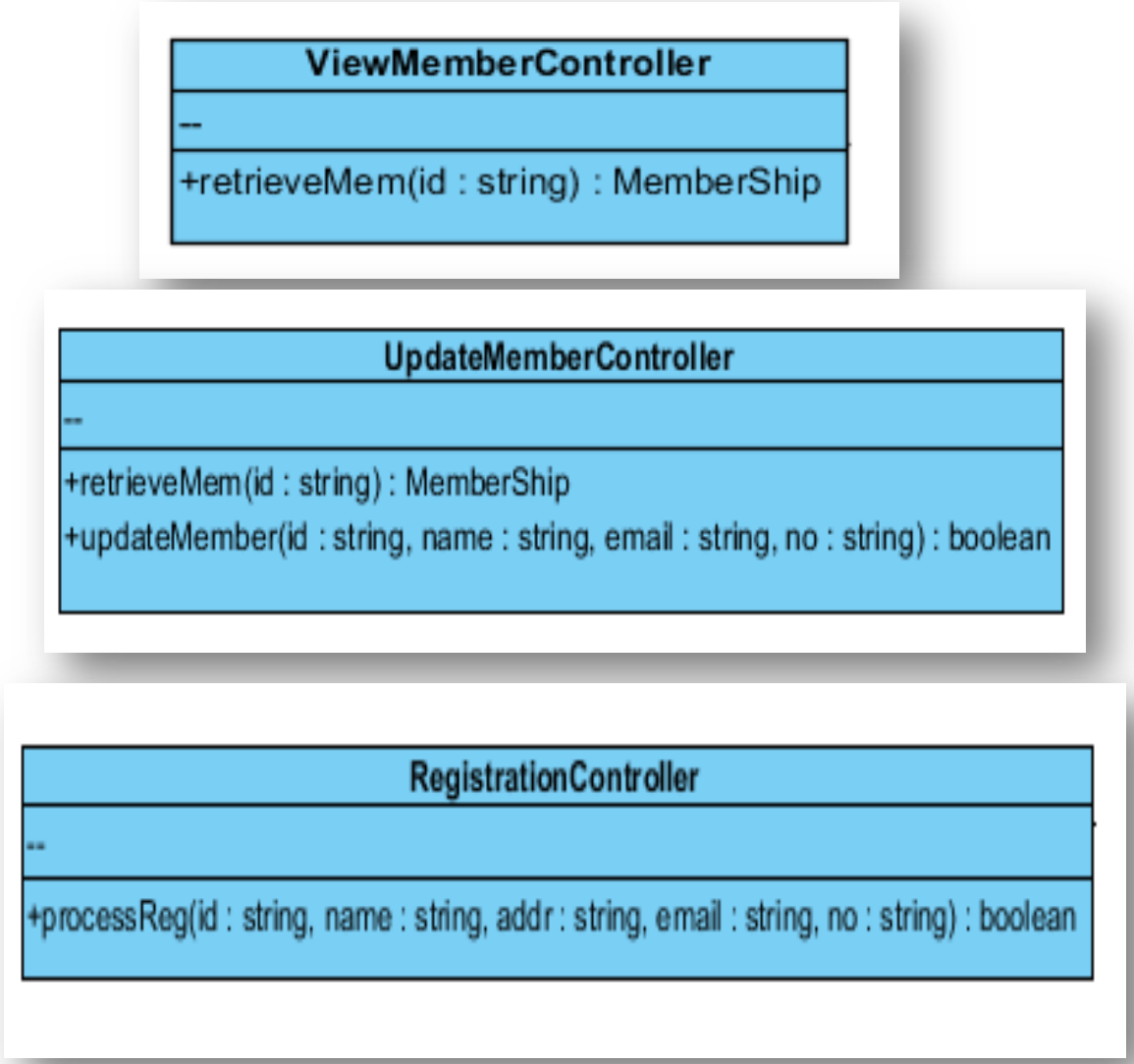
(a) is what we want to achieve: a system with minimum coupling.



(b)

(b) shows a system unnecessarily complex and very hard to maintain.

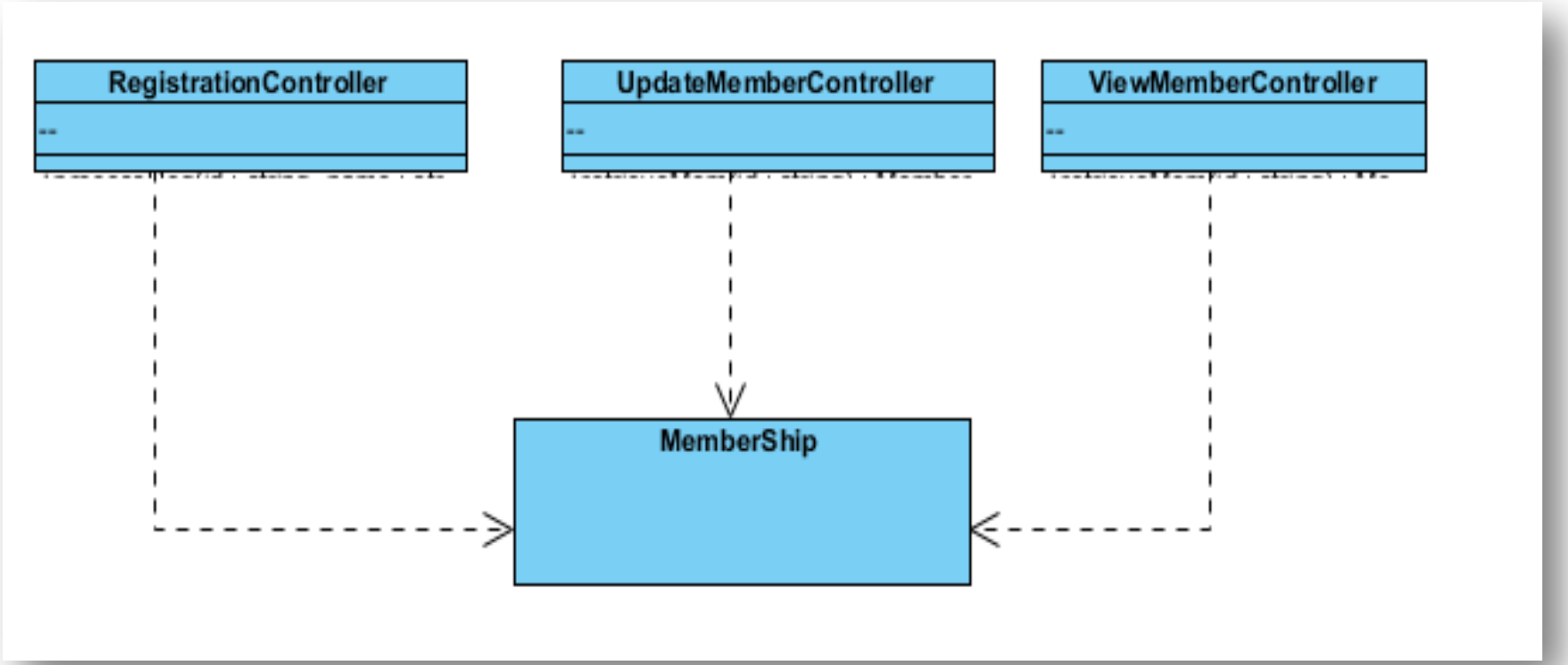
Let's consider our 3 use cases



Suggestion:
Instead of having 3 controller classes each handling related operations such as registration, updating and retrieving, we have just ONE controller class to handle
all of them.

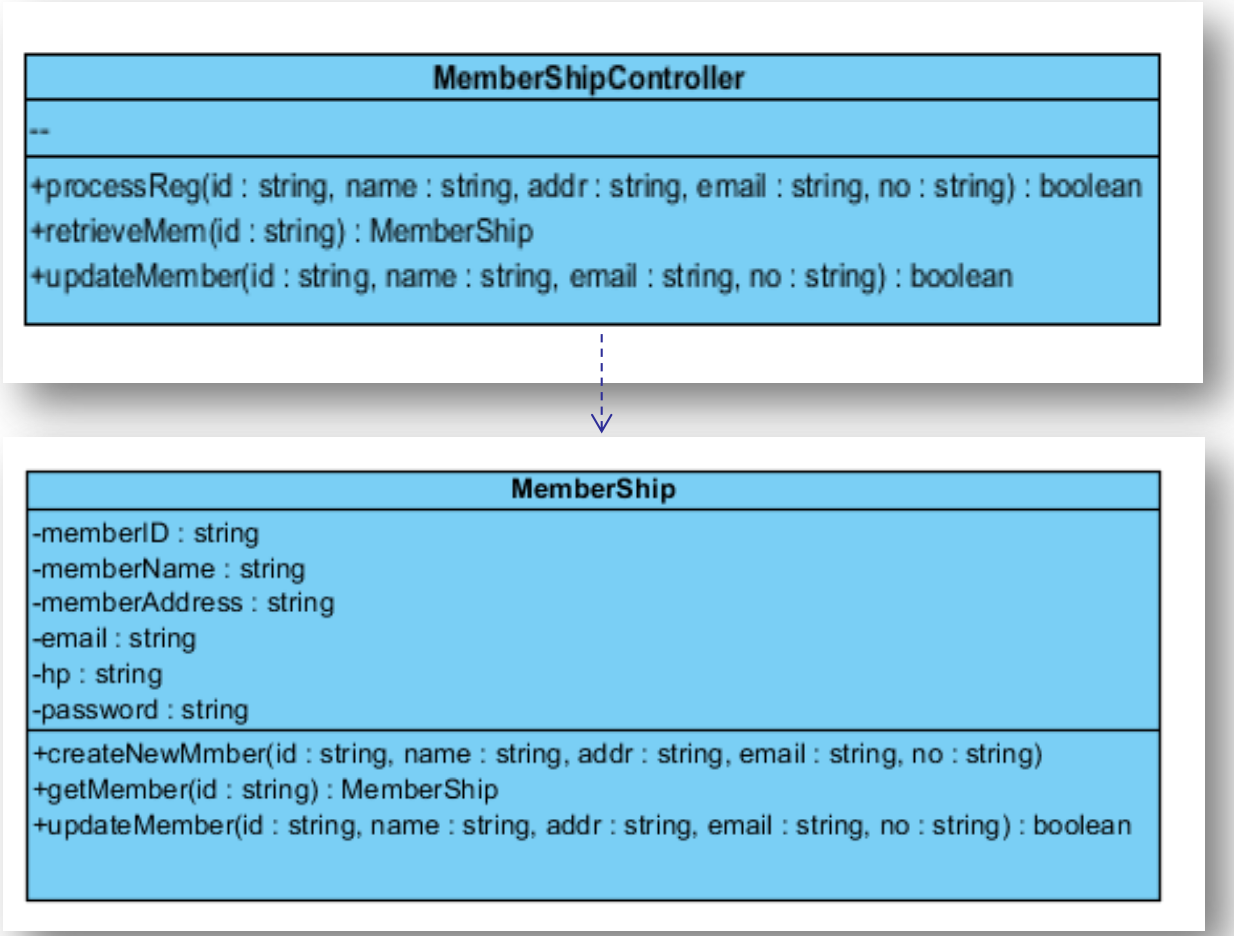
This helps us to achieve low coupling. So instead of depending on 3 classes, now we have one class
to manage the membership records
For registration, updating and retrieving.

Initial solution class diagram



High coupling

Revised solution class diagram



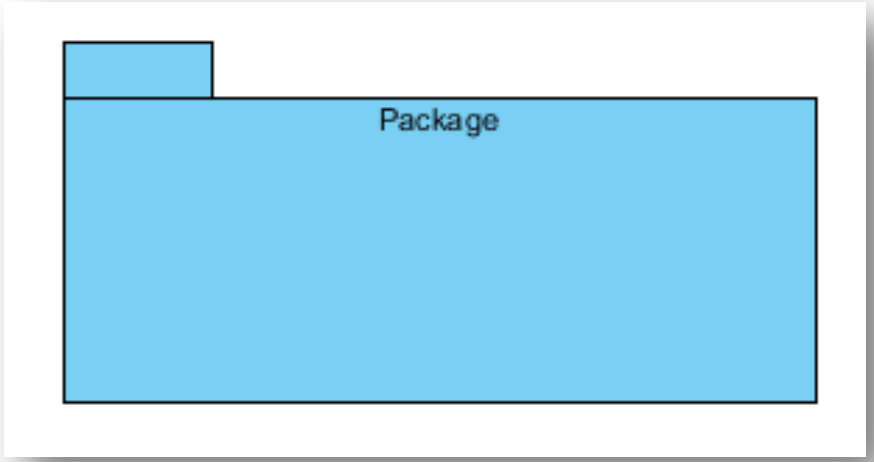
Advantage?

Low coupling
&
High cohesion

This improvement also helps to achieve high cohesion for the Membership controller class as we focus all membership operations in this class.

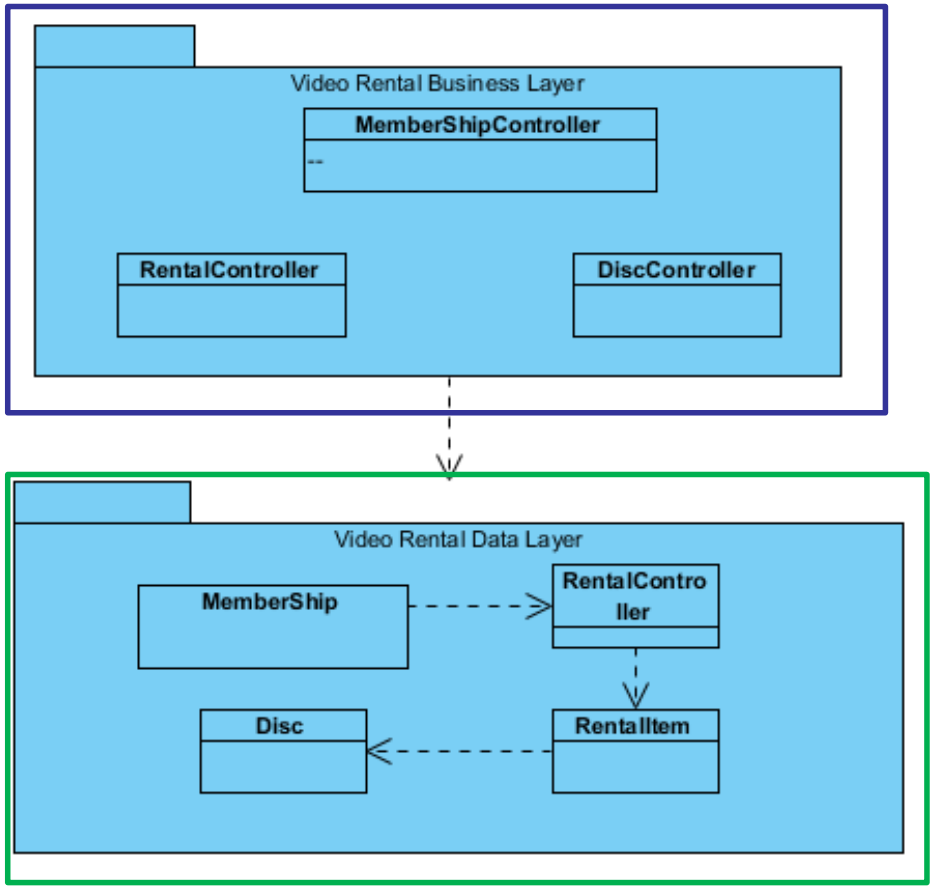
Package Diagram

- You can consider grouping **related classes** into *packages* based on their functionality and present the overall system design as **subsystems**.
- A **Package** is a general purpose mechanism for organizing model elements & diagrams into groups.



UML Package notation

Example: Package diagram for a Disc Rental System



We could have 2 packages to group these classes into:

Business Logic Layer (or subsystem) that contains:

- MembershipController
- RentalController, DiscController

Entity (Data) Layer (or subsystem) that contains:

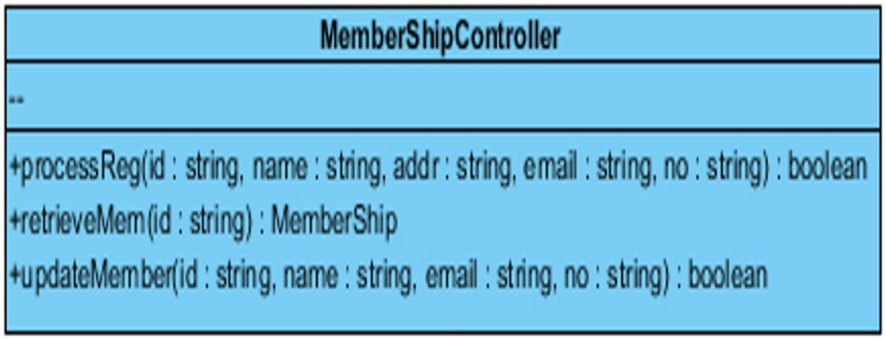
- Membership
- Disc
- Rental
- RentalItem

This is another way to represent the overall class diagram in design

What is next?

- Once the sequence diagram and the class diagram are completed, we are ready to go into coding the intended system
- We will be very clear in what to code based on these models
- That will be the implementation phase of the SDLC

Mapping solution classes to code



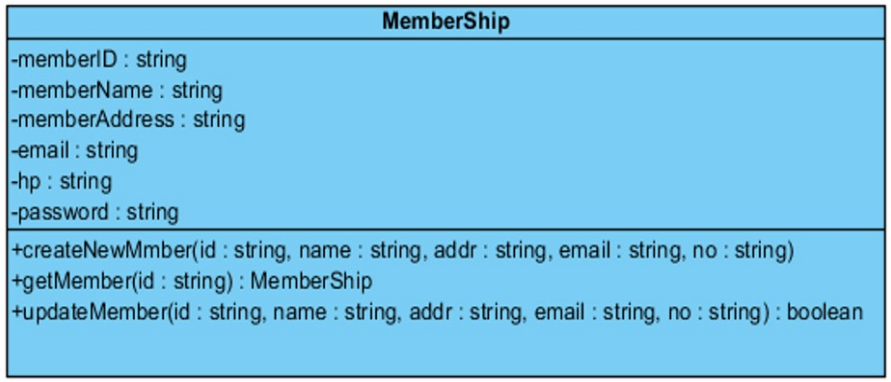

```
public class MemberShipController {

  public boolean processReg(String id, String
    name, String addr, String email, String no)
    {   }

  public MemberShip(String id)
    {   }

  public boolean updateMember(String id, String
      name, String email, String no)
    {   }
}
```

Mapping solution classes to code




```
public class MemberShip {

  private String memberID;
  private String memberName;
  private String memberAddress;
  private String email;
  private String hp;
  private String password;

  public boolean createNewMember((String id, String
name, String addr, String email, String no) {   }

public MemberShip getMember(String id) {   }

  public boolean updateMember(String id, String name,
String addr, String email, String no) {    }
}
```

Summary

- We discussed how to convert the responsibilities in the sequence diagram to method definitions in the design class
- We learnt how to construct an overall class diagram and apply low coupling, high cohesion principles to make the class diagram a better one
- We also learnt that the package diagram is another alternative to represent the overall class diagram

Cohort Exercise

1. Complete the solution class diagram by converting the responsibilities you had identified previously to method definitions.
2. As a team combine all the solution classes you had derived into an overall solution class diagram.
3. Apply the 2 design principles and re-design your team solution class diagram.