Software Design and Analysis
CS-3004
Lecture#10

Dr. Javaria Imtiaz,

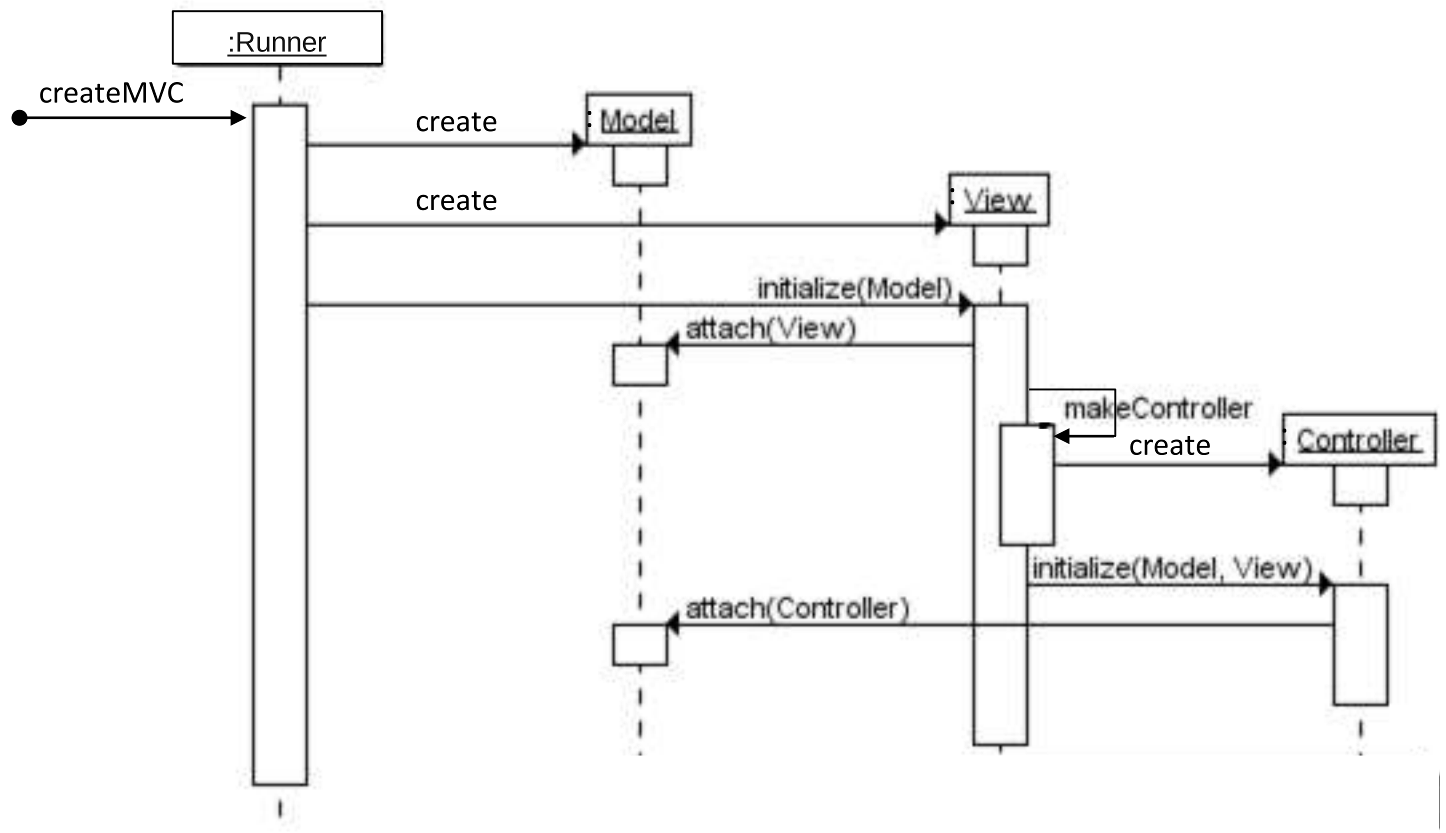
Mr. Basharat Hussain,

Mr. Majid Hussain

quest lab

Quiz # 4

Write code for the given sequence diagram.



```
public class Runner {
private Model m;
private View v;
public void createMVC() {
        m = new Model();
        v = new View();
        v.initialize(m);
       }
}
```

```
public class View {
private Model m;
private Controller c;
public void initialize(Model model){
        m = model;
        m.attach(this);
        makeController();
        c.initialize(m,this);    }

public void makeController() {
        c = new Controller();    }
        }
```

```
public class Model {
private View v;
private Controller c;
public void attach(View view) {
        v = view;     }
public void attach(Controller controller) {
c = controller;     }
}
```

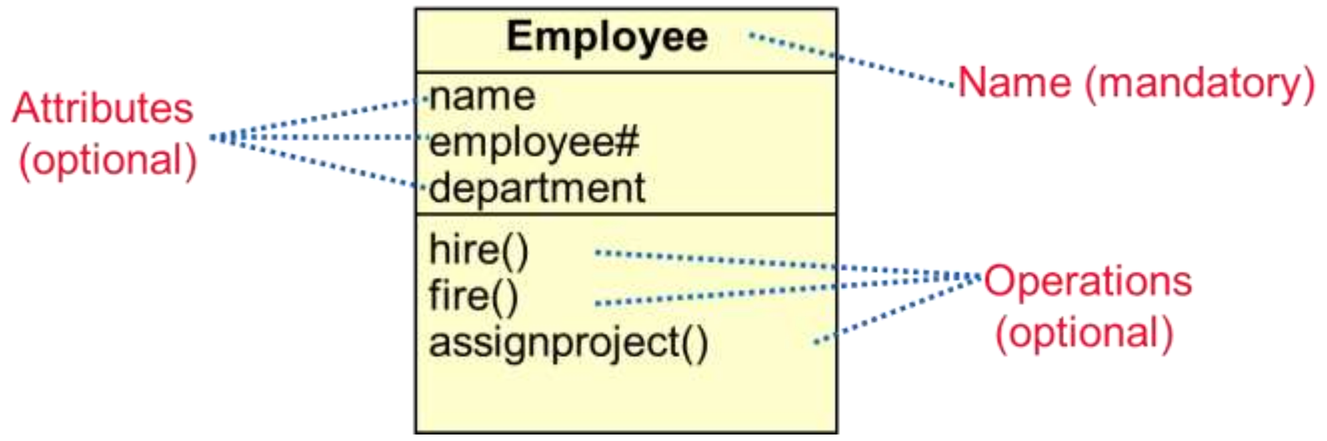
```
public class Controller {
private Model m;
private View v;
public void initialize(Model model, View view){
                m = model;
                v = view;
                m.attach(this);    }
}
```

```
public class MainClass {
public static void main(String[] args) {
       Runner run = new Runner();
       run.createMVC(); } }
```

Object-Oriented Design

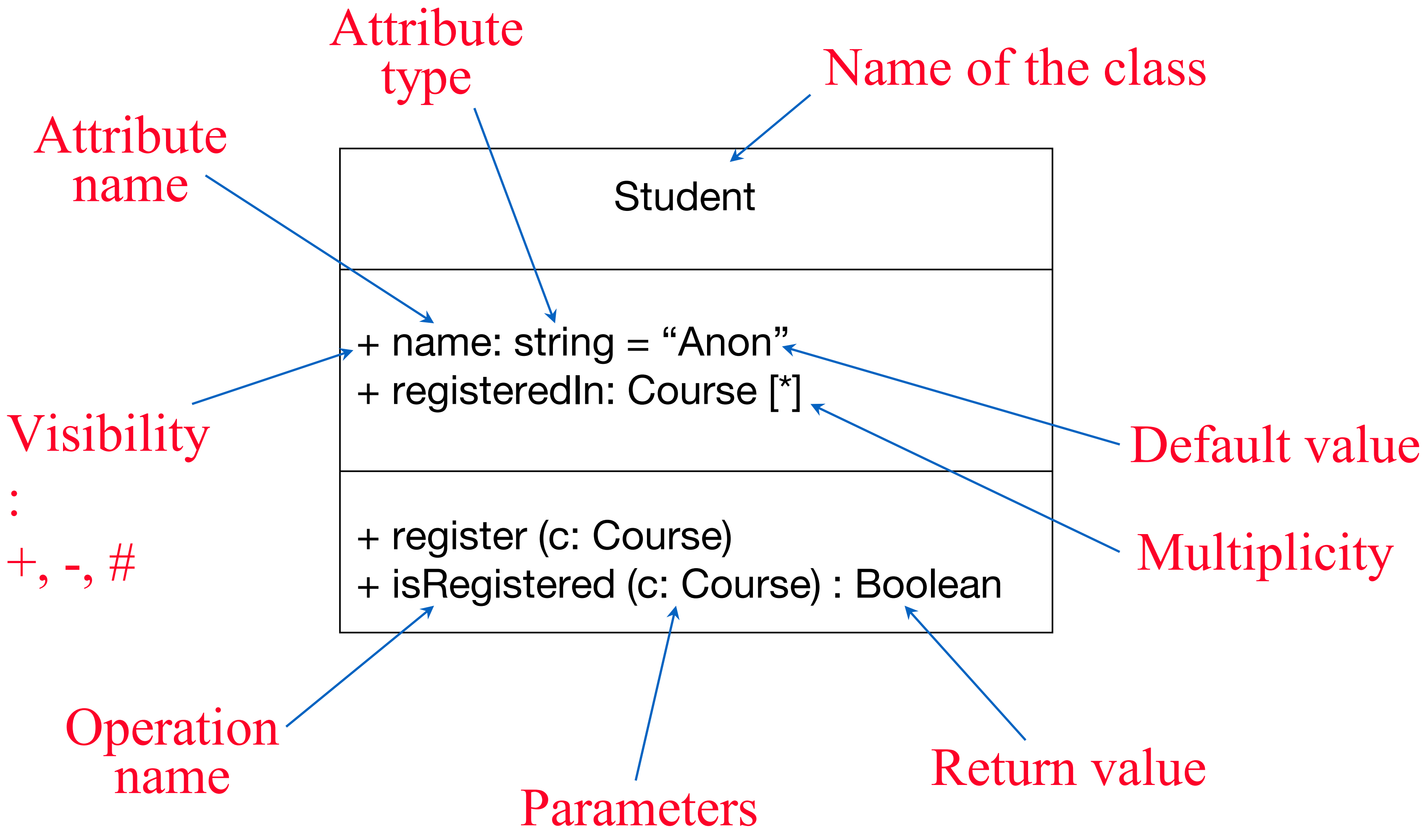
- "After identifying your requirements and creating a domain model, then create software classes, and define the messaging between the objects to fulfill the requirements."
- Often make Sequence diagram and Class Diagram in parallel using the domain model and SSD

Class Diagram

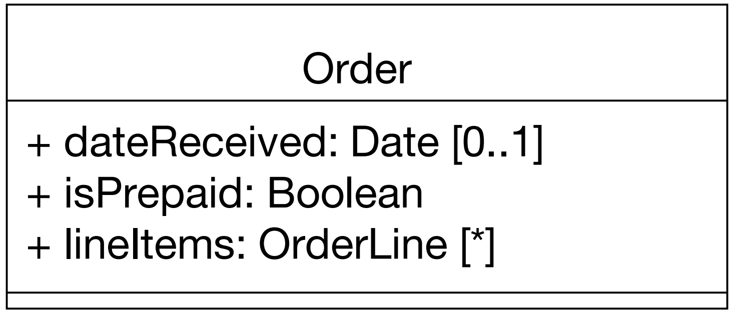


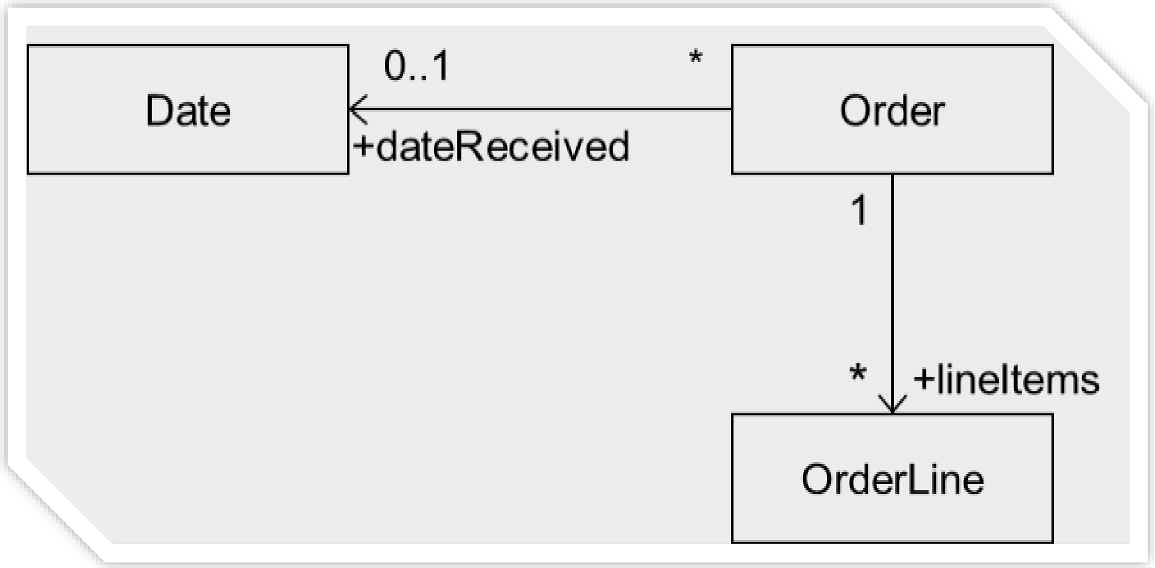
- A class consists of
  - properties (attributes),
  - behavior (operations),
  - relationships to other objects,
- Examples
  - Employee: has a name, employee# and department; an employee is hired, and fired; an employee works in one or more projects

Class Diagram Notation

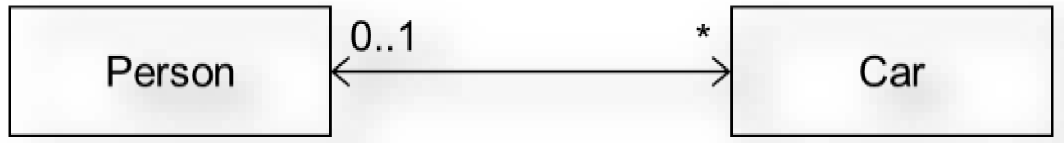


Navigability

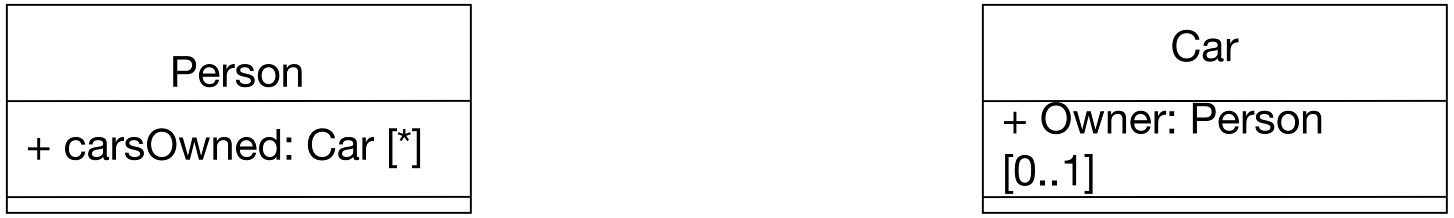




Bidirectional Association

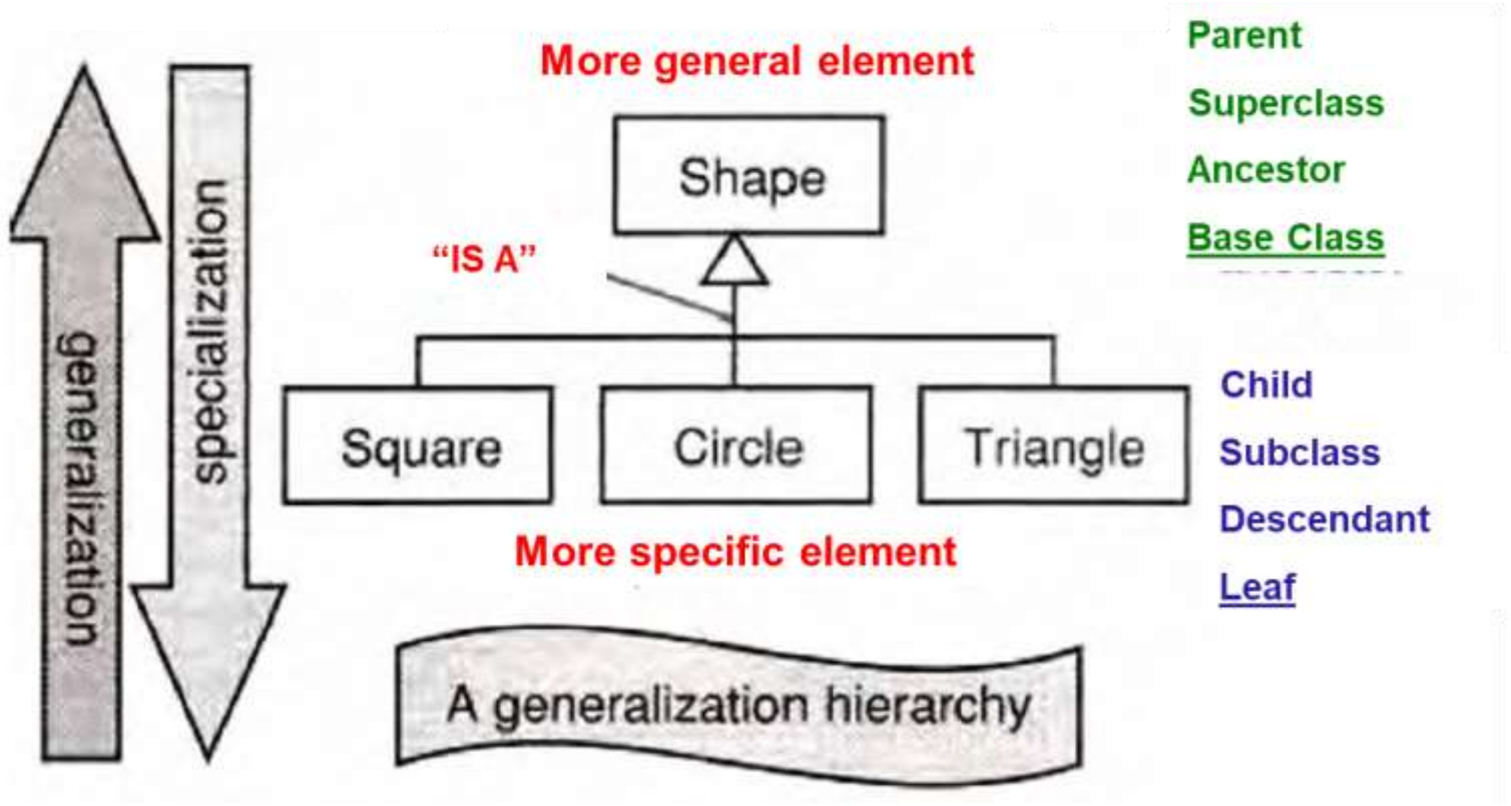


How implement it?



Generalization/Specialization

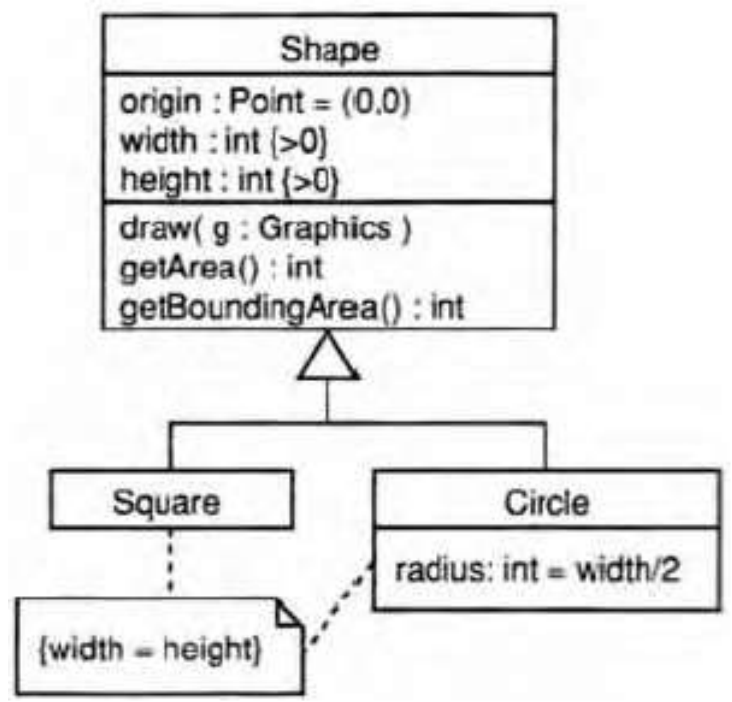
- Generalization hierarchies may be created by generalizing from specific things or by specializing from general things.



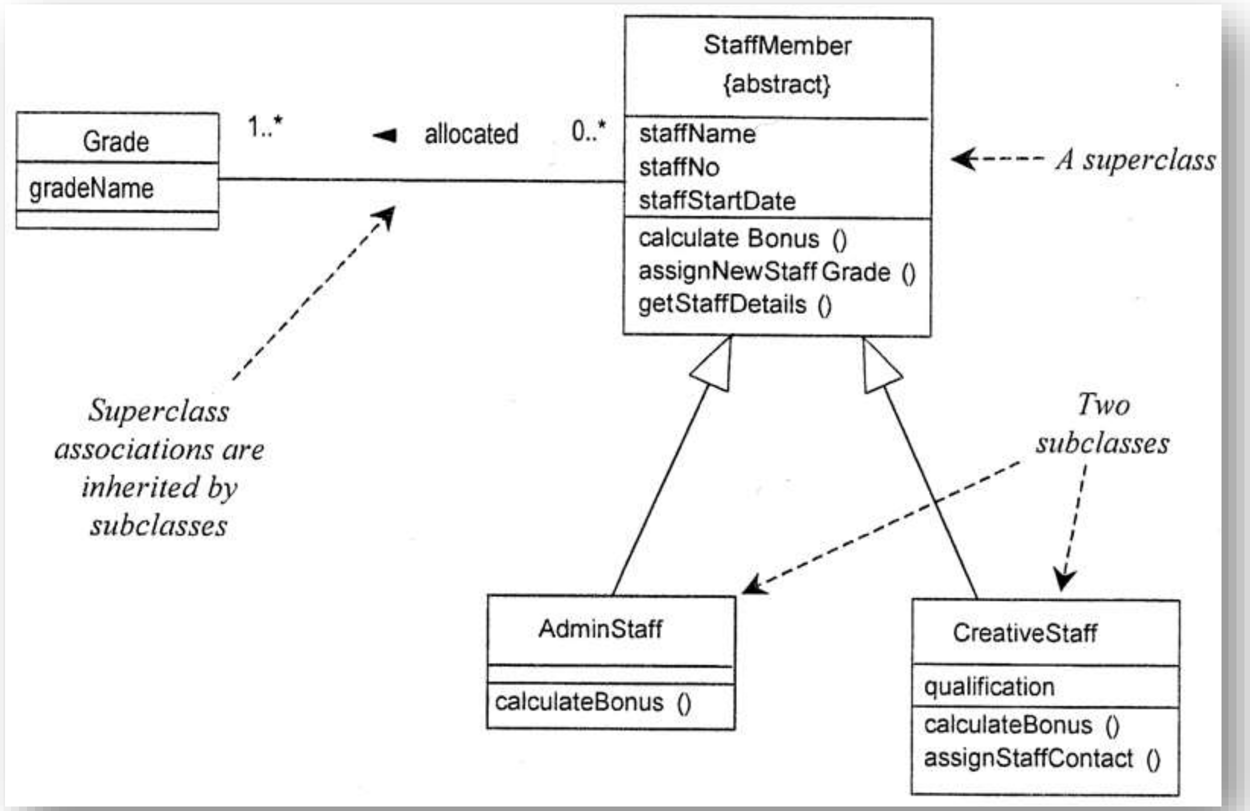
Inheritance

- Class inheritance is implicit in a generalization relationship between classes.
- Subclasses inherit attributes, associations, & operations from the superclass

What is the inheritance mechanism in Java?



Inheritance

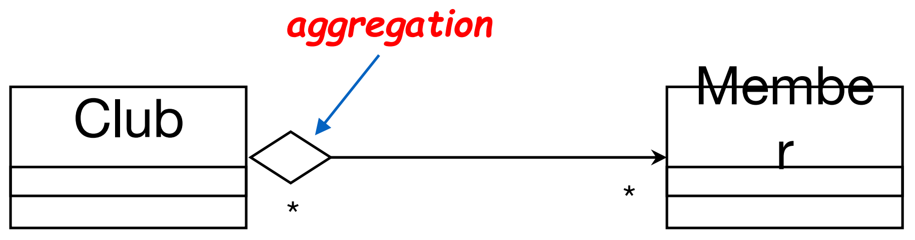


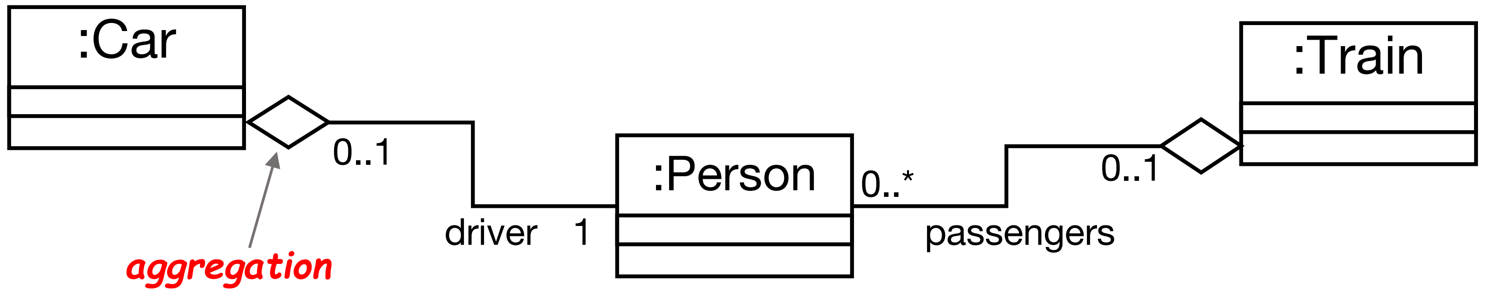
Java Inheritence

- A subclass may override an inherited aspect
  - e.g. AdminStaff & CreativeStaff have different
  - methods for calculating bonuses
- A Subclass may add new features
  - qualification is a new attribute in CreativeStaff
- Superclasses may be declared {abstract}, meaning they have no instances
  - Implies that the subclasses cover all possibilities
  - e.g. there are no other staff than AdminStaff and CreativeStaff

Aggregation

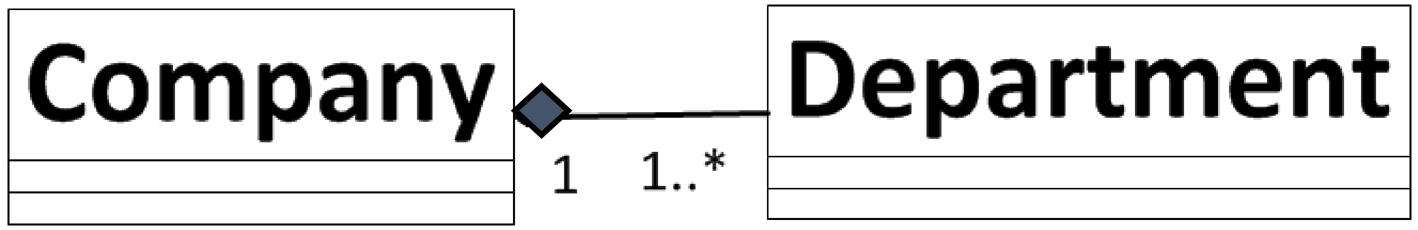
- This is the "Has-A" relationship

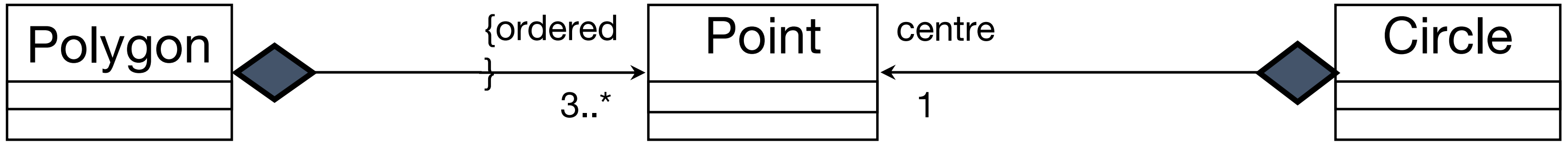




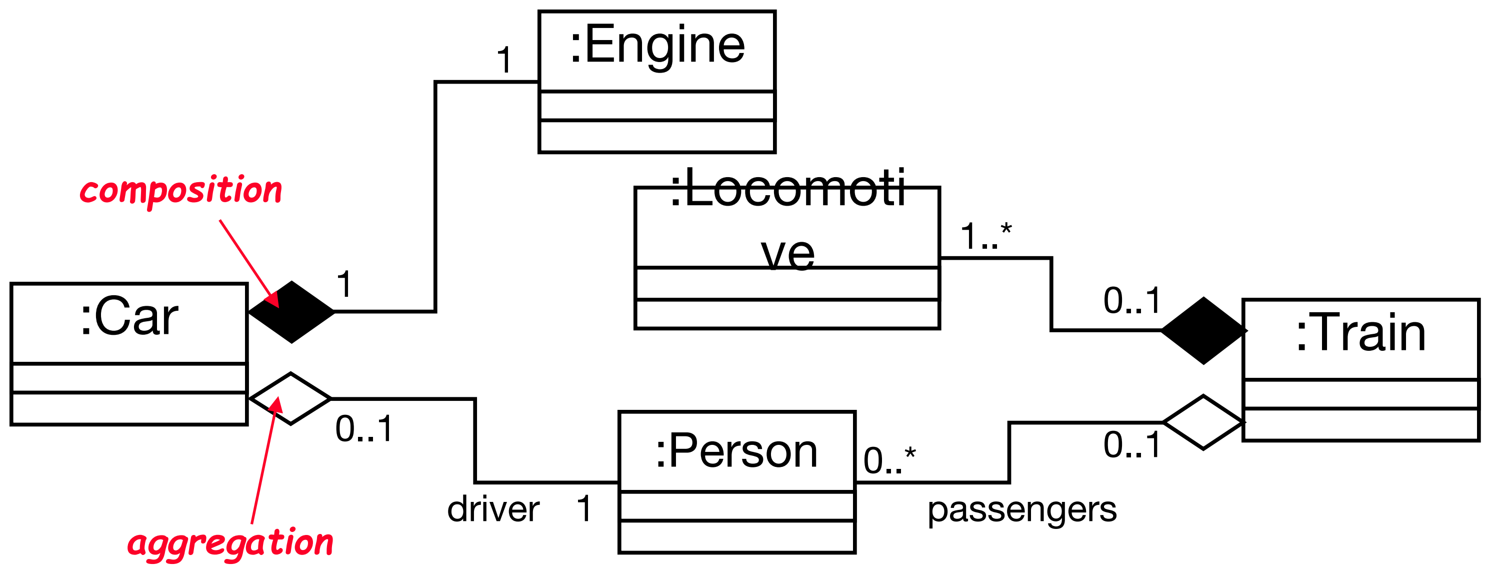
Aggregation and Composition

- Composition is strong form of aggregation that implies ownership:
  - if the whole is removed from the model, so is the part.
  - the whole is responsible for the disposition of its parts
  - Note: Parts can be removed from the composite (where allowed) before the composite is deleted





Aggregation and Composition



Object-Oriented Design

- “After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.”
- ***But how?***
  - How should concepts be implemented by classes?
  - What method belongs where?
  - How should the objects interact?
  - This is a critical, important, and non-trivial task

Design Patterns - GRASP

- Responsibility Driven Development

- General Responsibility Assignment Software Patterns

- Controller
- Information Expert
- Creator
- Low Coupling
- High Cohesion
- Pure Fabrication

A design pattern is a general repeatable solution to a commonly occurring problem in software design

Design patterns can speed up the development process by providing tested, proven development paradigms

Controller

- A simple layered architecture has a user interface layer (UI) and a business logic layer.
- Actors, such as the human user, generate UI events (such as clicking a button).
- The UI software objects (such as a JFrame window and a JButton) must process the event.
- When objects in the UI layer pick up an event, they must delegate the request to an object in the domain layer.

- ***Problem: What first object in the business logic layer should receive the message from the UI layer?***

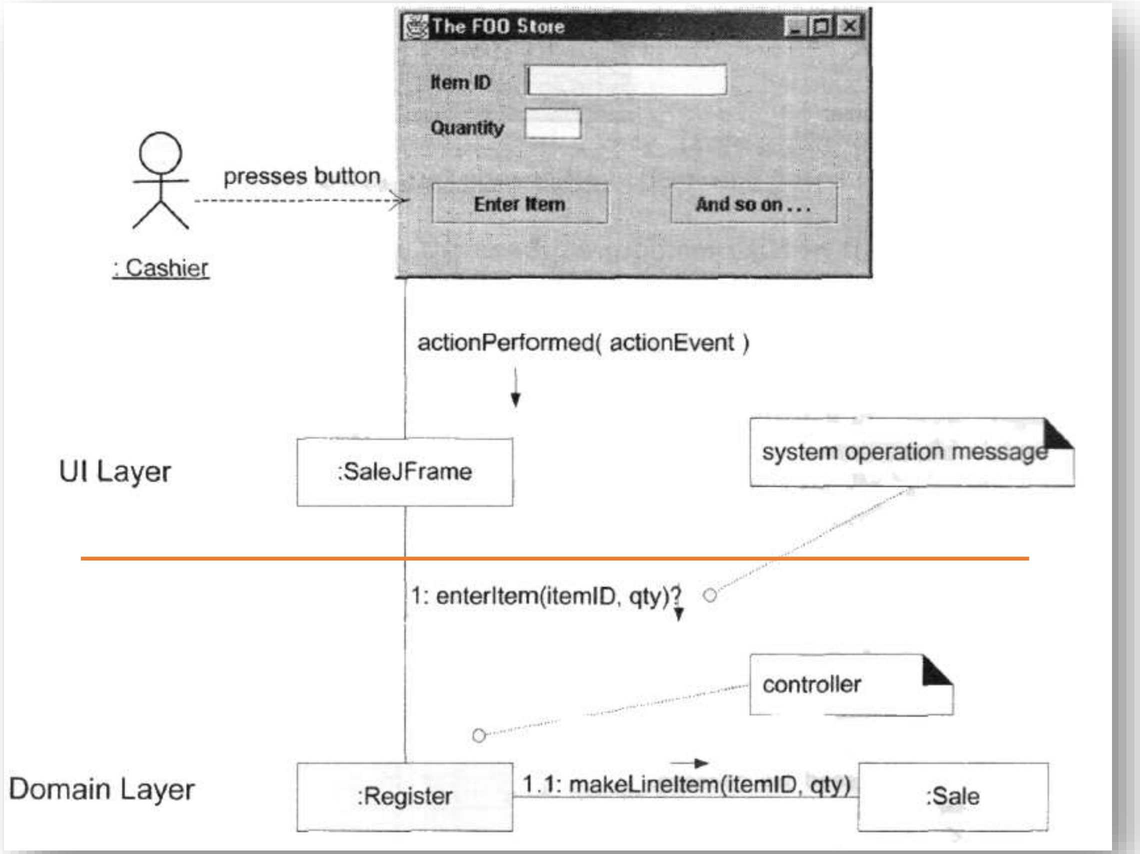
  OR in other words

- ***Who should be responsible for handling system events ?***

Controller

- The Controller is also an important idiom in modern web development frameworks, in forming a pillar of the Model-View-Controller architectural pattern. Controllers are used in AngularJS, Ruby on Rails, Sails and more.

Problem: What first object in the business logic layer should receive the message from the UI layer?



Solution:

Façade Controller to hide the complexity

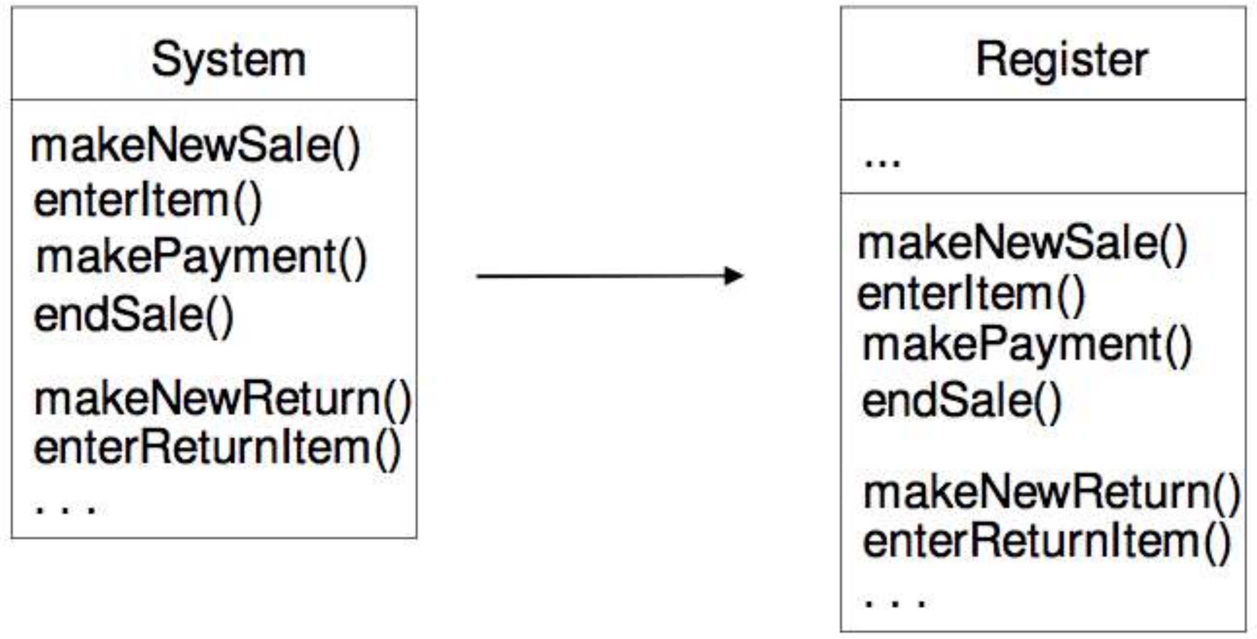
- Works as a wrapper
- Should not contain business logic

Controller

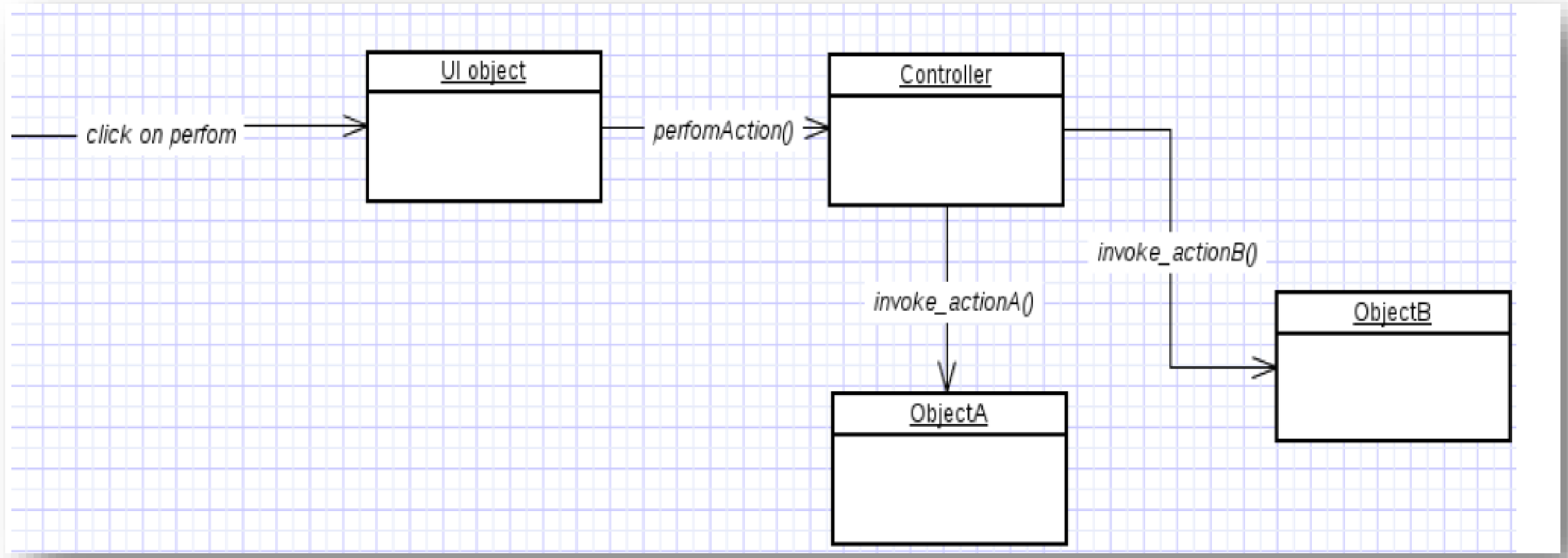
- Which class will receive the first system call?
  - Deals with how to delegate the request from the UI layer objects to domain layer objects.
  - When a request comes from UI layer object, Controller pattern helps us in determining what is that first object that receive the message from the UI layer objects.
  - This object is called controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.
  - It delegates the work to other class and coordinates the overall activity.

Façade Controller

- All system operations are assigned to one controller.



Controller



- We can make an object as Controller, if
  - Object represents the overall system (facade controller)
  - Object represent a use case, handling a sequence of operations (session controller).
- Benefits
  - can reuse this controller class.
  - Can use to maintain the state of the use case.
  - Can control the sequence of the activities

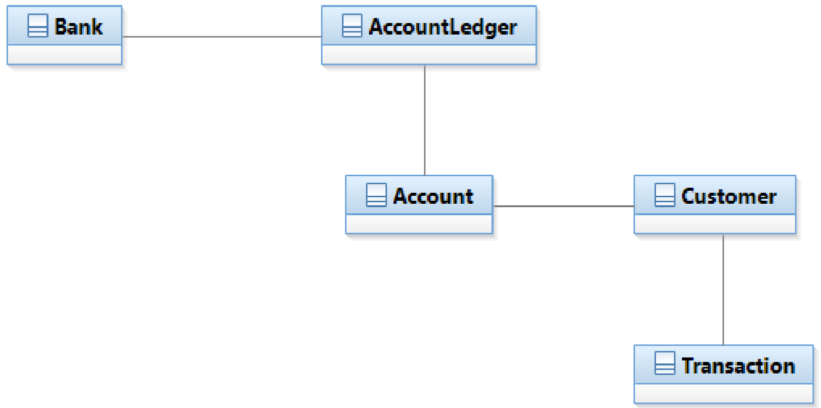
Bloated Controller

- Controller class is called bloated, if
  - The class is overloaded with too many responsibilities.
  - Solution: Add more controllers

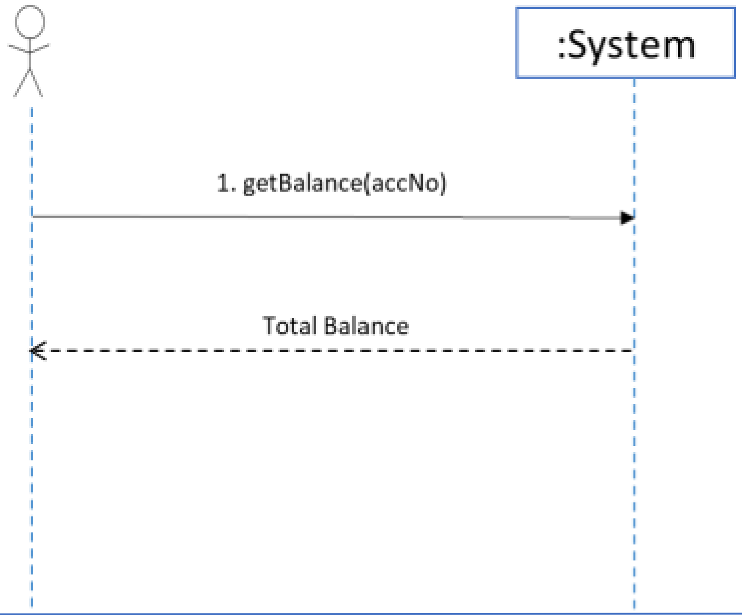
- The responsibility of controller class is to delegate things to others.
  - It will not perform any kind of business logic/ calculations.

Example

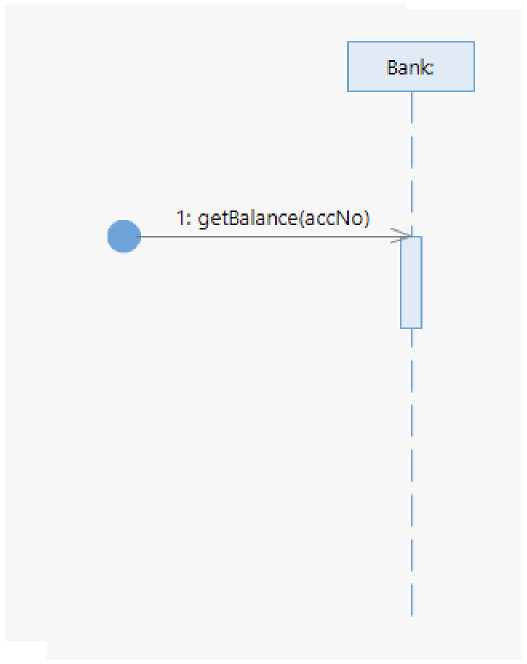
Domain Model



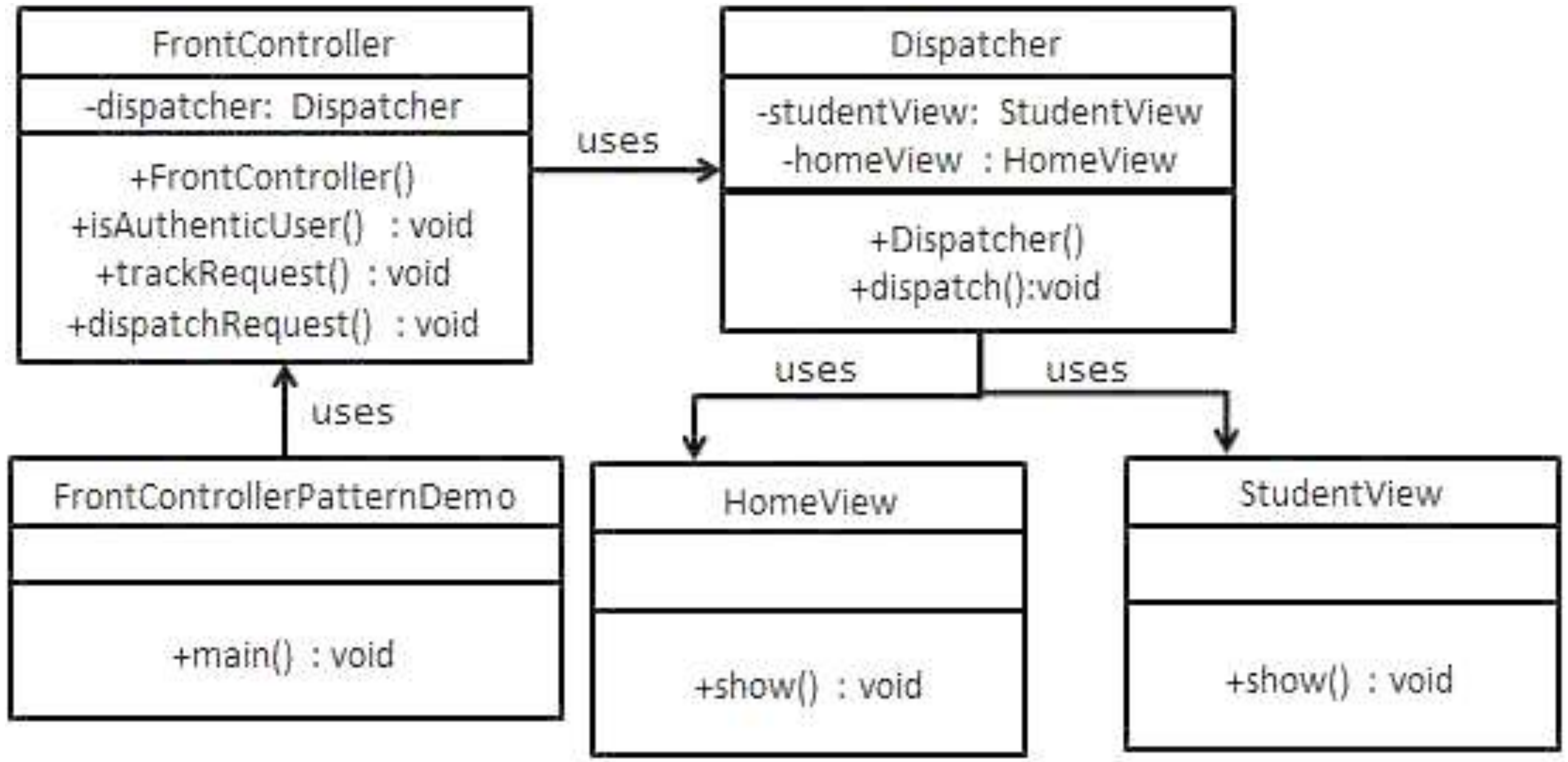
System Sequence Diagram

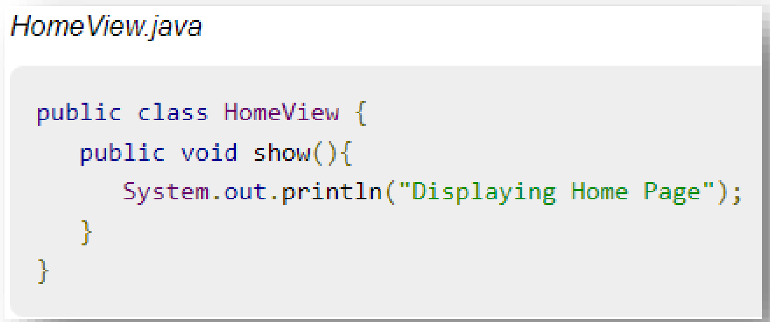


Sequence Diagram

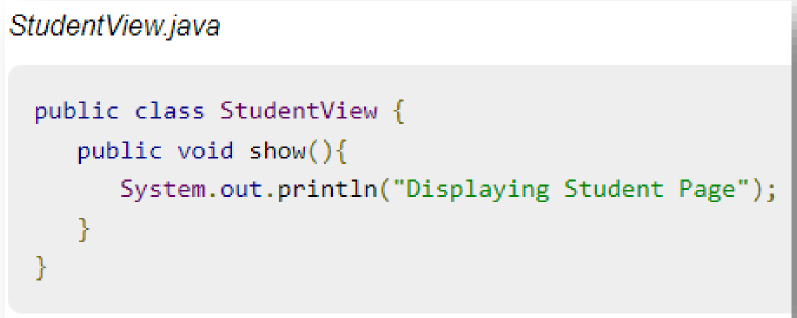


Controller Demo

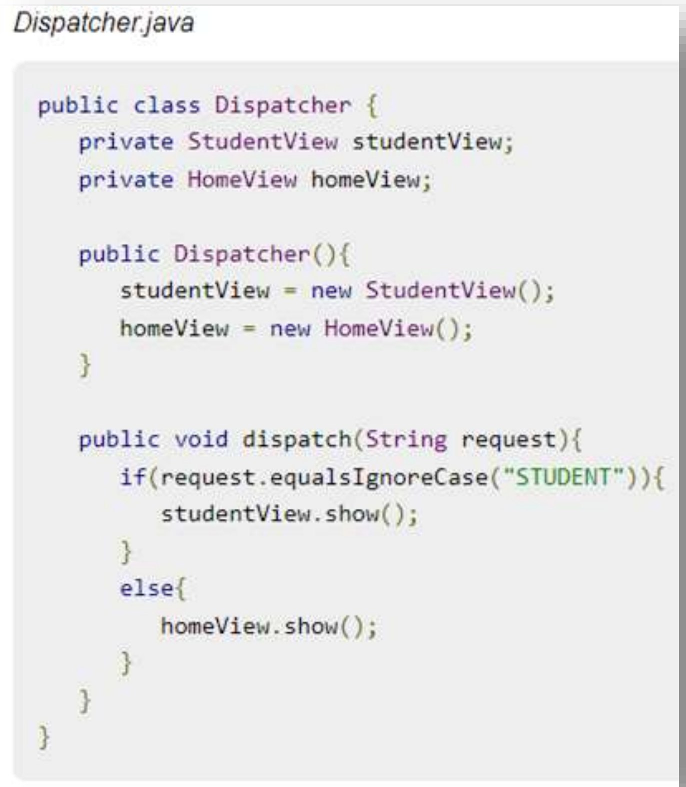



HomeView.java

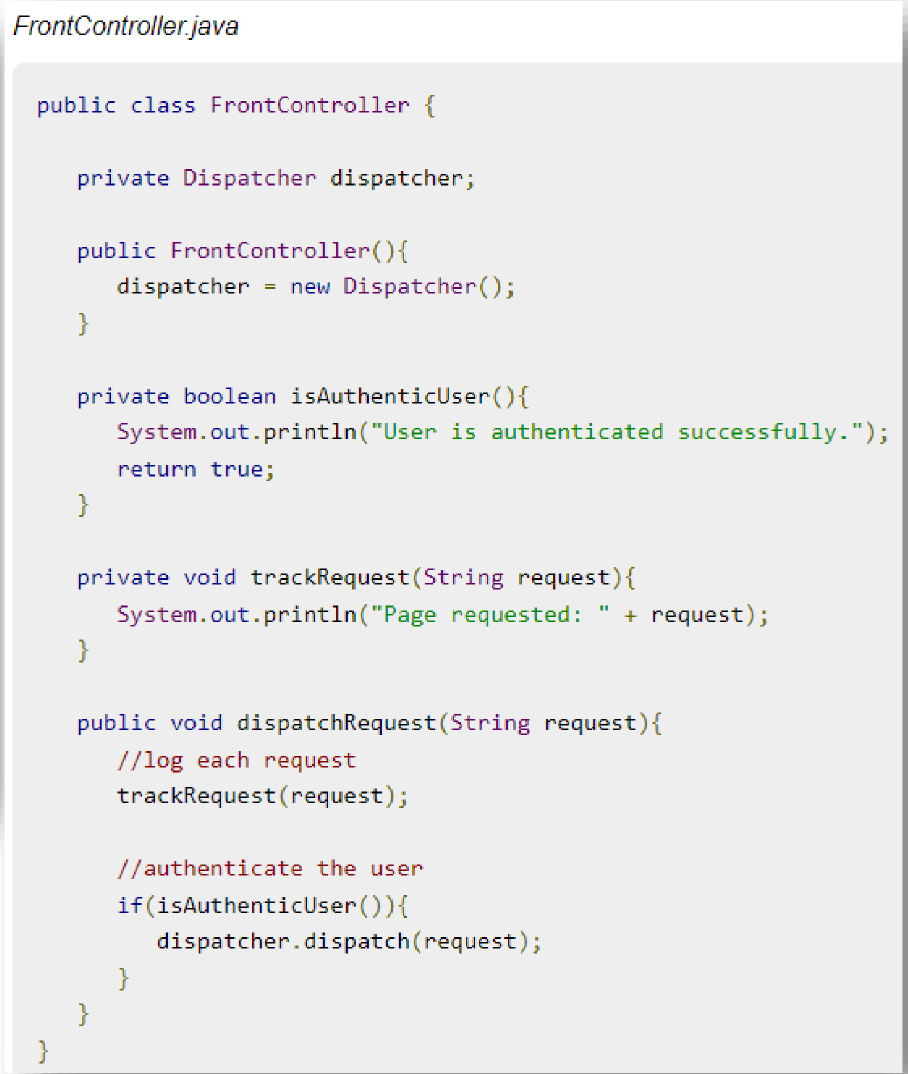
```
public class HomeView {
   public void show(){
      System.out.println("Displaying Home Page");
   }
}
```


StudentView.java

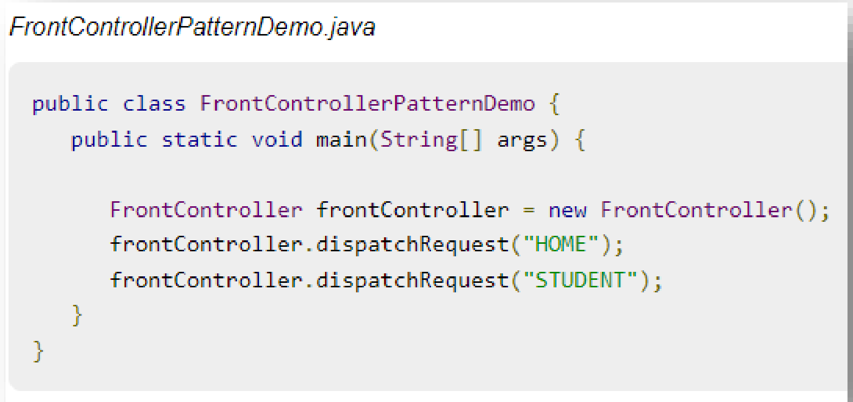
```
public class StudentView {
   public void show(){
      System.out.println("Displaying Student Page");
   }
}
```


Dispatcher.java

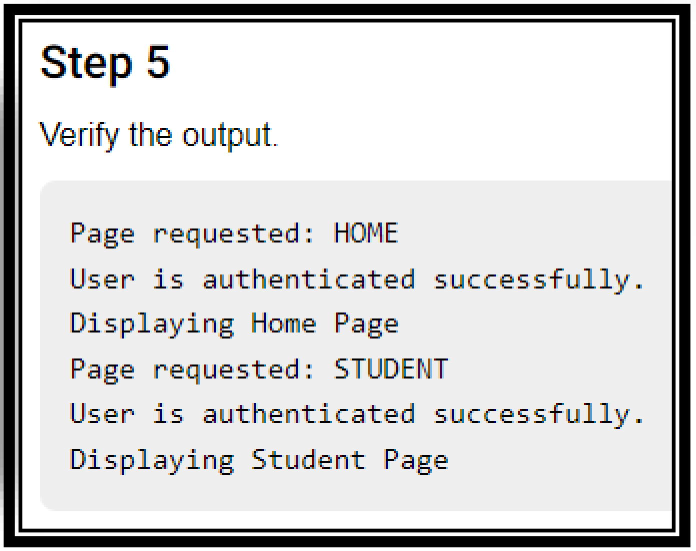
```
public class Dispatcher {
   private StudentView studentView;
   private HomeView homeView;

   public Dispatcher(){
      studentView = new StudentView();
      homeView = new HomeView();
   }

   public void dispatch(String request){
      if(request.equalsIgnoreCase("STUDENT")){
         studentView.show();
      }
      else{
         homeView.show();
      }
   }
}
```


FrontController.java

```
public class FrontController {

   private Dispatcher dispatcher;

   public FrontController(){
      dispatcher = new Dispatcher();
   }

   private boolean isAuthenticUser(){
      System.out.println("User is authenticated successfully.");
      return true;
   }

   private void trackRequest(String request){
      System.out.println("Page requested: " + request);
   }

   public void dispatchRequest(String request){
      //log each request
      trackRequest(request);

      //authenticate the user
      if(isAuthenticUser()){
         dispatcher.dispatch(request);
      }
   }
}
```


FrontControllerPatternDemo.java

```
public class FrontControllerPatternDemo {
   public static void main(String[] args) {

      FrontController frontController = new FrontController();
      frontController.dispatchRequest("HOME");
      frontController.dispatchRequest("STUDENT");
   }
}
```


Step 5

Verify the output.

```
Page requested: HOME
User is authenticated successfully.
Displaying Home Page
Page requested: STUDENT
User is authenticated successfully.
Displaying Student Page
```

Information Expert

- ***Problem : What is general principle of assigning responsibilities to Objects?***

- Solution : Assign Responsibility to class that has the information to fulfil the responsibility

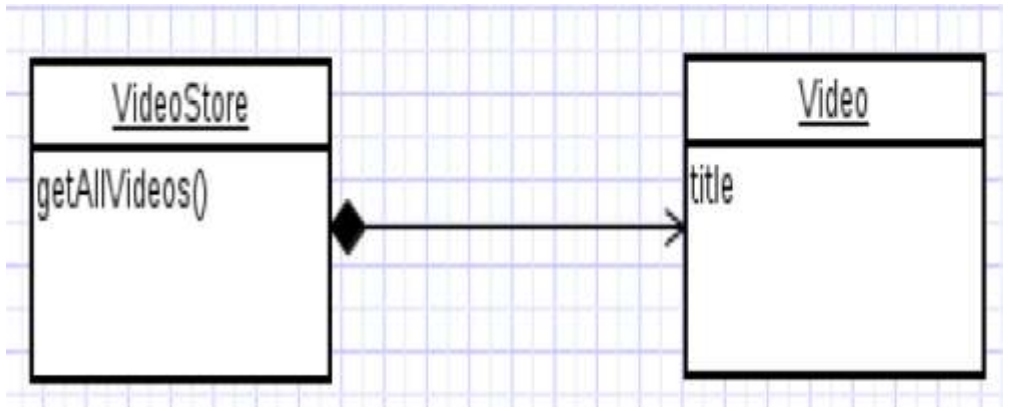
- Decision of which class to call? The class that has relevant data
  - Can be current class or some other class

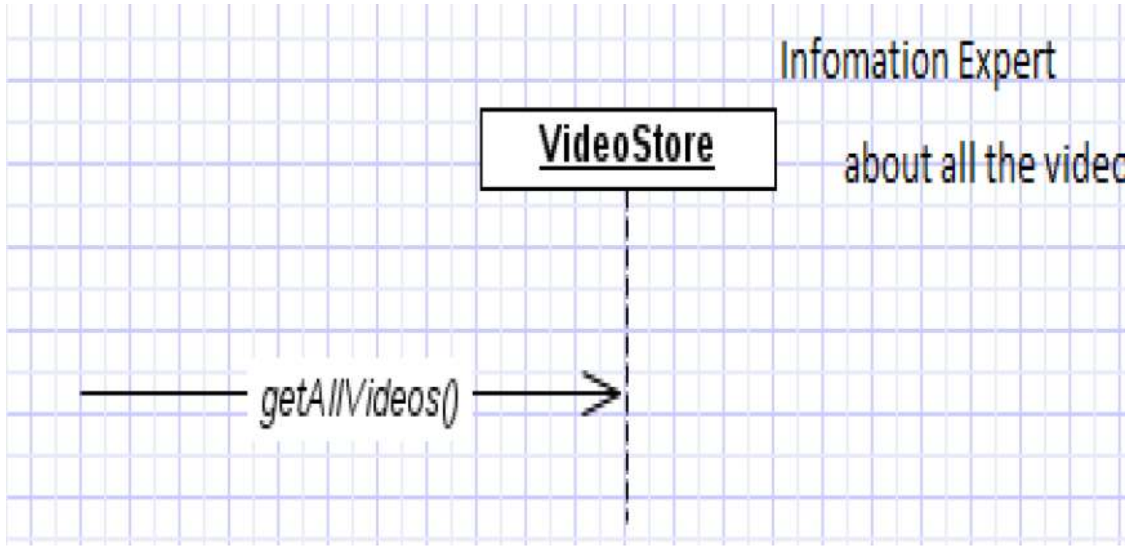
Benefits

- Information encapsulation is maintained since objects use their own information to fulfill tasks.
- This usually supports low coupling, which leads to more robust and maintainable systems.
- Behavior is distributed across the classes that have the required information, thus encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain.

Example

- Assume we need to get all the videos of a VideoStore.
- Since VideoStore knows about all the videos, we can assign this responsibility of giving all the videos can be assigned to VideoStore class.
- VideoStore is the information expert.

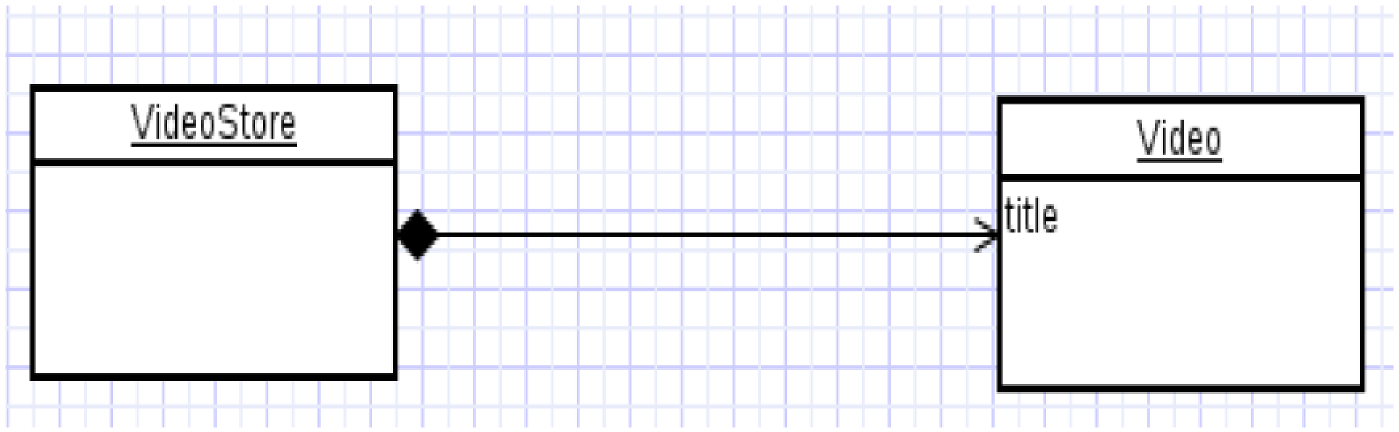




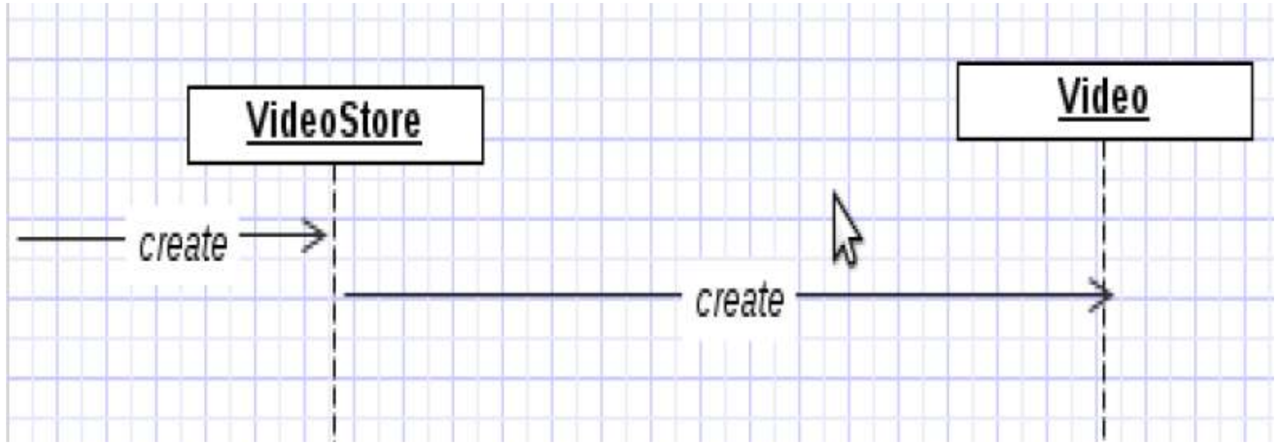
Creator

- ***Problem: Who should be responsible for creating new instances of a class?***
- “Container” object creates “contained” objects.
- Decide who can be creator based on the objects association and their interaction.

- Solution: B creates If,
  - B aggregates A
  - B contains A
  - B records A
  - B closely uses A
  - B initializes A



Sequence Diagram – illustrates Creator Pattern



Low Coupling

- ***Problem: How to support Low dependency, low change impact, increase re-use ?***

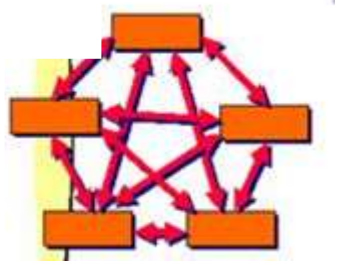
Coupling

- How ***dependent*** one element (e.g. class) is on other elements (e.g. classes)

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

- High coupling is problematic

High coupling

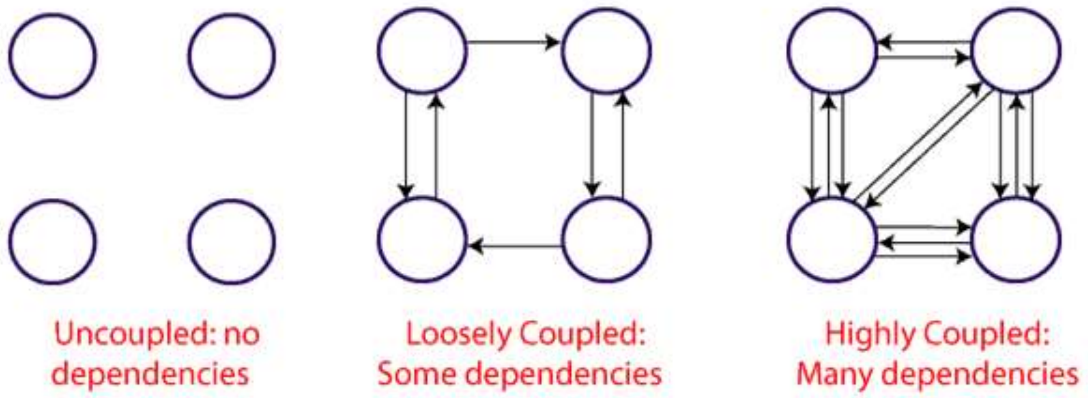


Problems with High Coupling

- High coupling would mean that your class knows the way too much about the inner workings of other classes.

- classes that know too much about other classes make changes hard to coordinate and make code brittle and difficult to reuse.

- If class A knows too much about clas
  the functionality in class A.


Uncoupled: no dependencies
Loosely Coupled: Some dependencies
Highly Coupled: Many dependencies

Low Coupling

- ***Problem: How to achieve low dependency, low change impact, increase re-use ?***

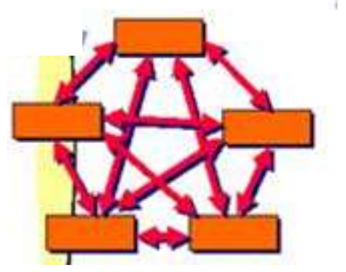
- Solution : Assign responsibilities so that coupling remains low.

Low Coupling

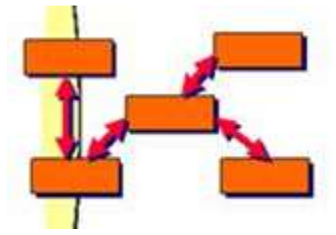
Low coupling is an evaluative pattern that dictates how to assign responsibilities for the following benefits:

- lower dependency between the classes
- change in one class having a lower impact on other classes
- higher reuse potential.

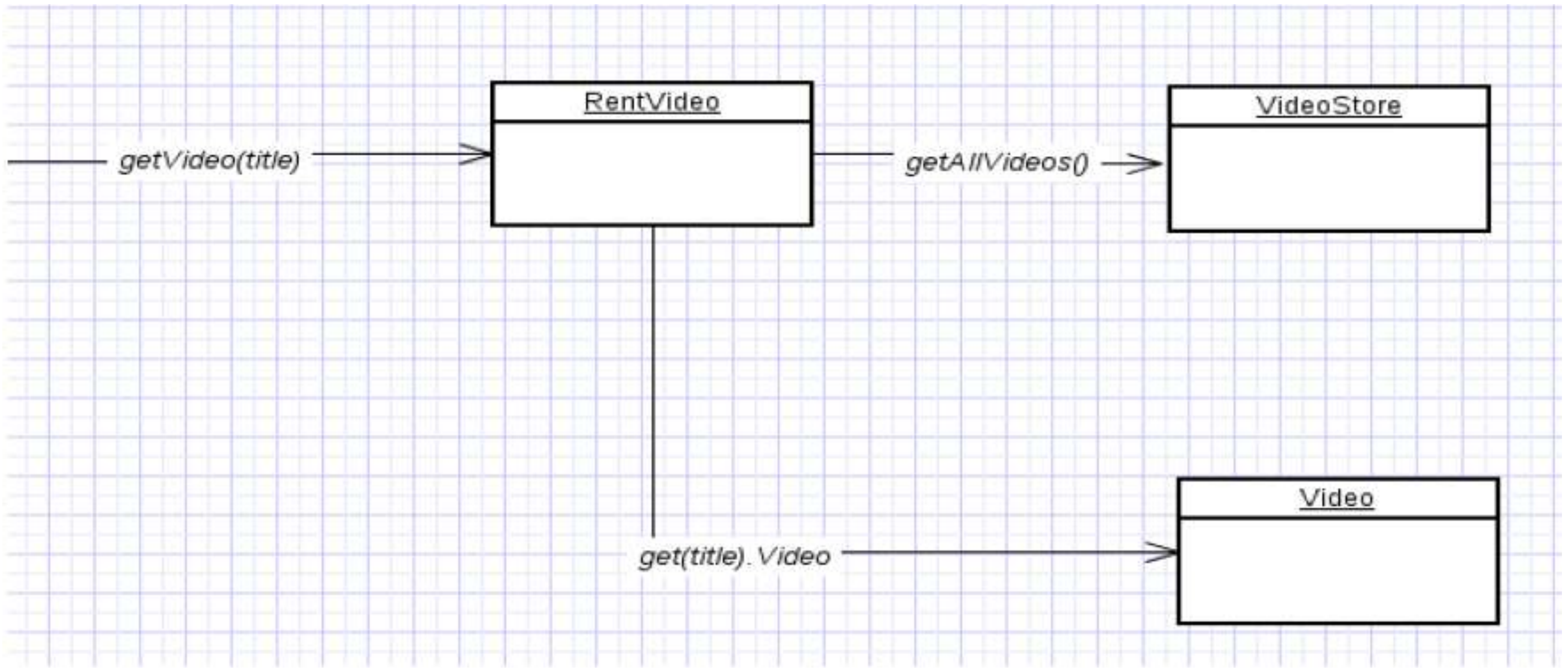
High coupling



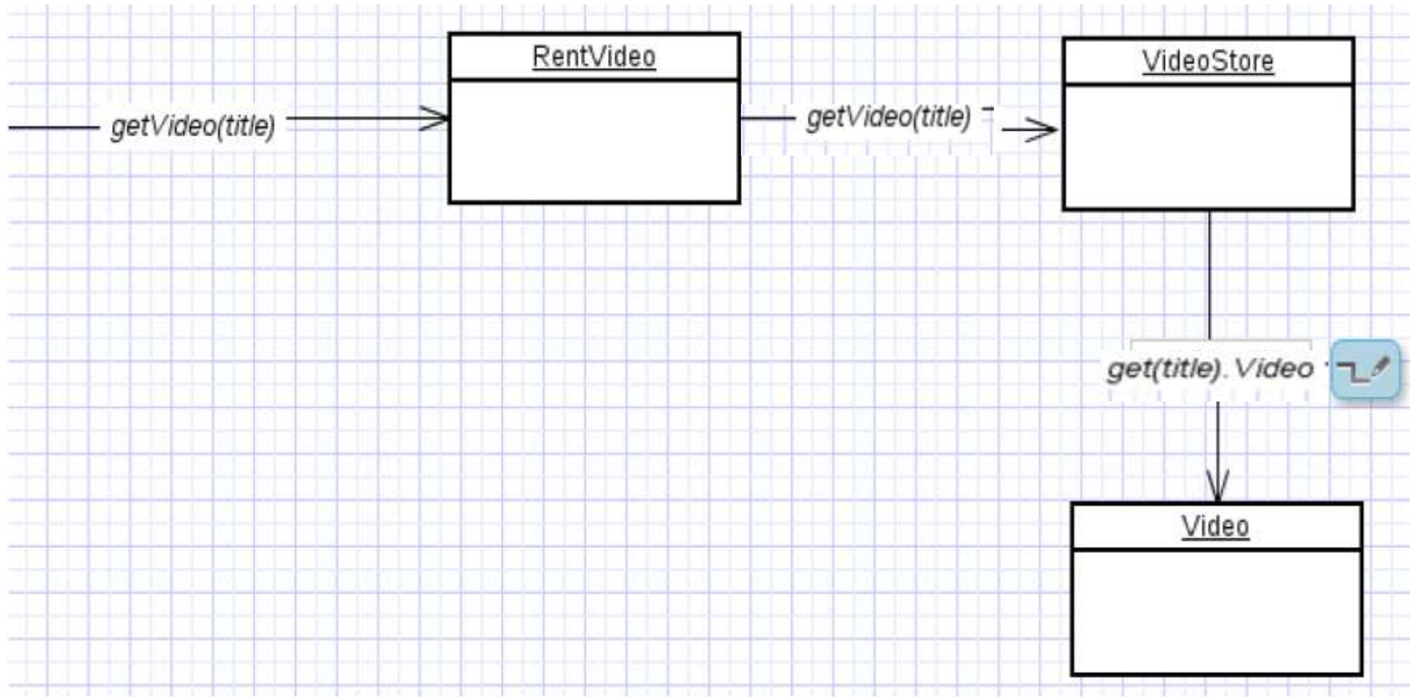
Low coupling



Example (High Coupling)

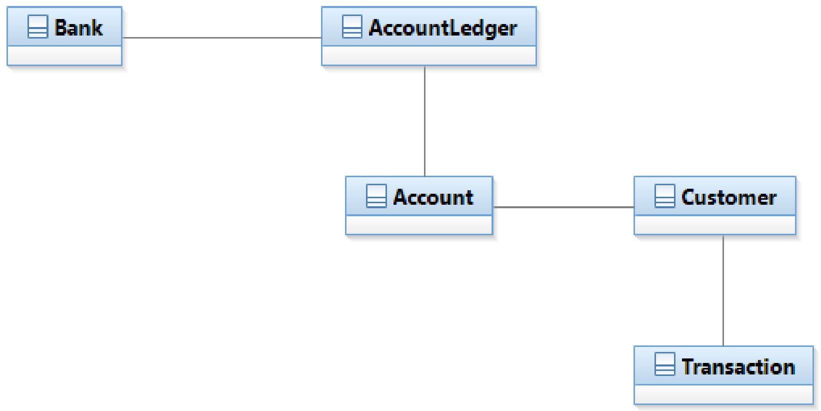


Example (Low Coupling)

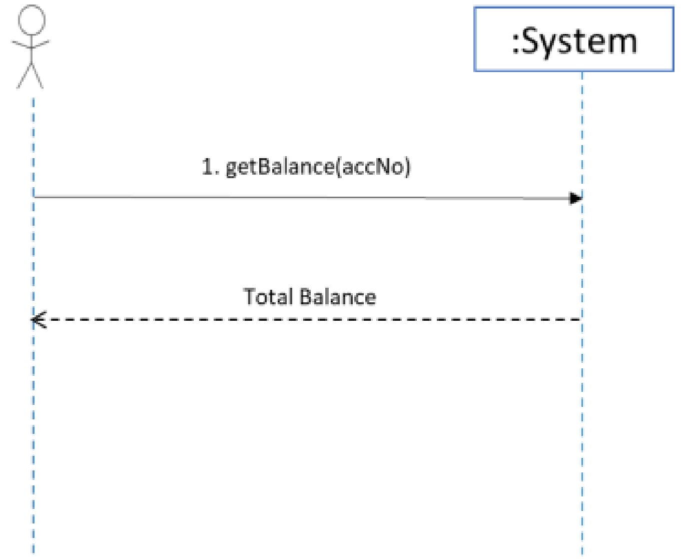


Example

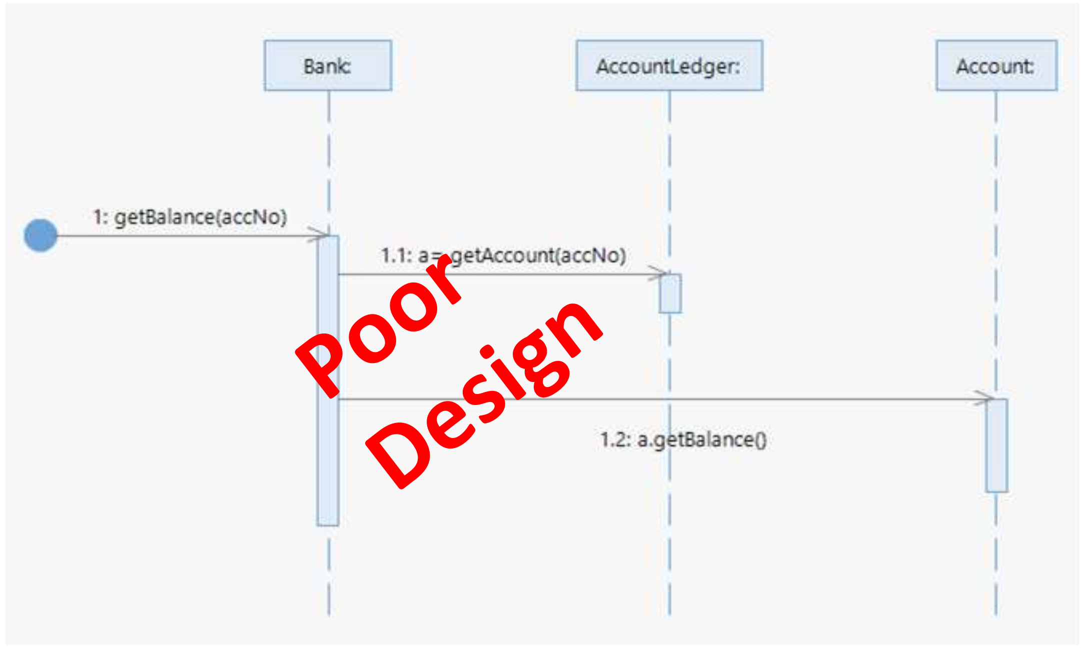
Domain Model



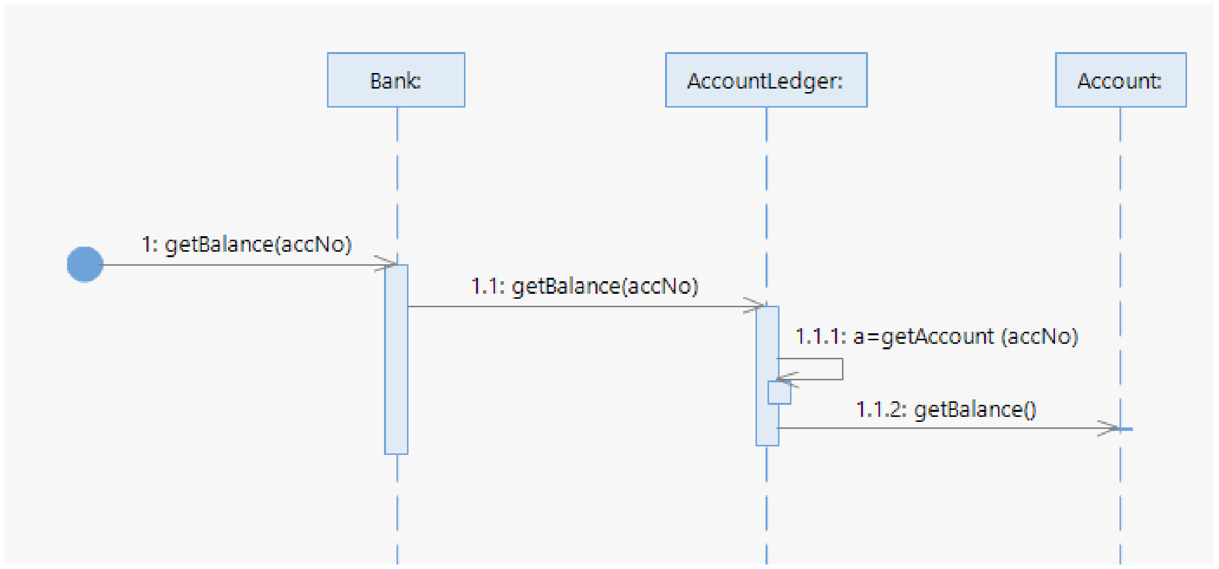
System Sequence Diagram



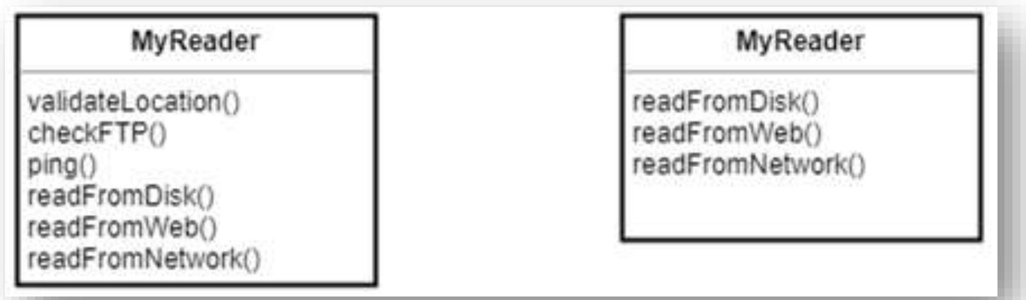
Sequence Diagram



Design Alternative



High Cohesion

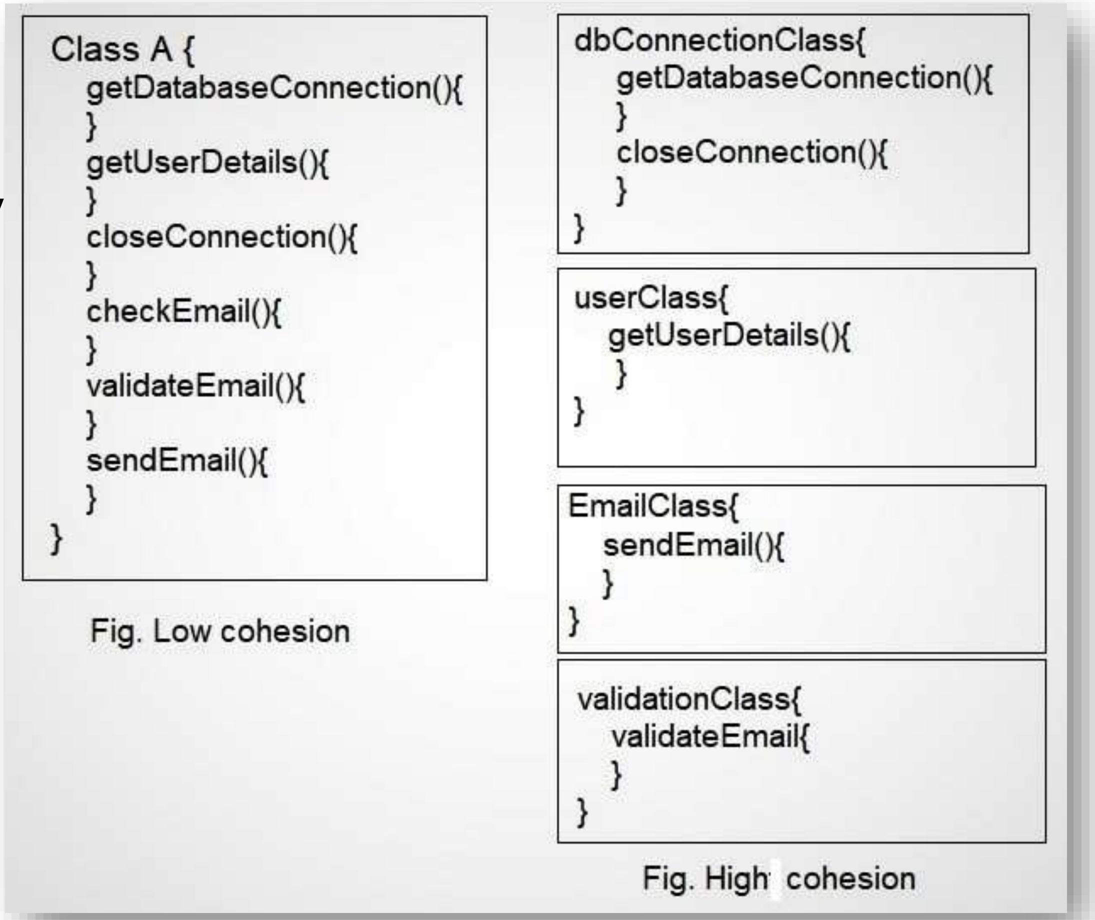


- Cohesion refers to how the functions of a class belong together. Related code should be close to each other to make it highly cohesive.

- ***Problem: How to keep complexity manageable ?***
  - Solution : Assign responsibilities so that cohesion remains high.
  - Note low cohesion and high coupling often go together.

High Cohesion

- How are the operations of any element functionally related?
- Related responsibilities in to one manageable unit
- Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
  - Easily understandable and maintainable.
  - Code reuse
  - Low coupling



```
Class A {
    getDatabaseConnection(){
    }
    getUserDetails(){
    }
    closeConnection(){
    }
    checkEmail(){
    }
    validateEmail(){
    }
    sendEmail(){
    }
}
```

Fig. Low cohesion

```
dbConnectionClass{
    getDatabaseConnection(){
    }
    closeConnection(){
    }
}
```

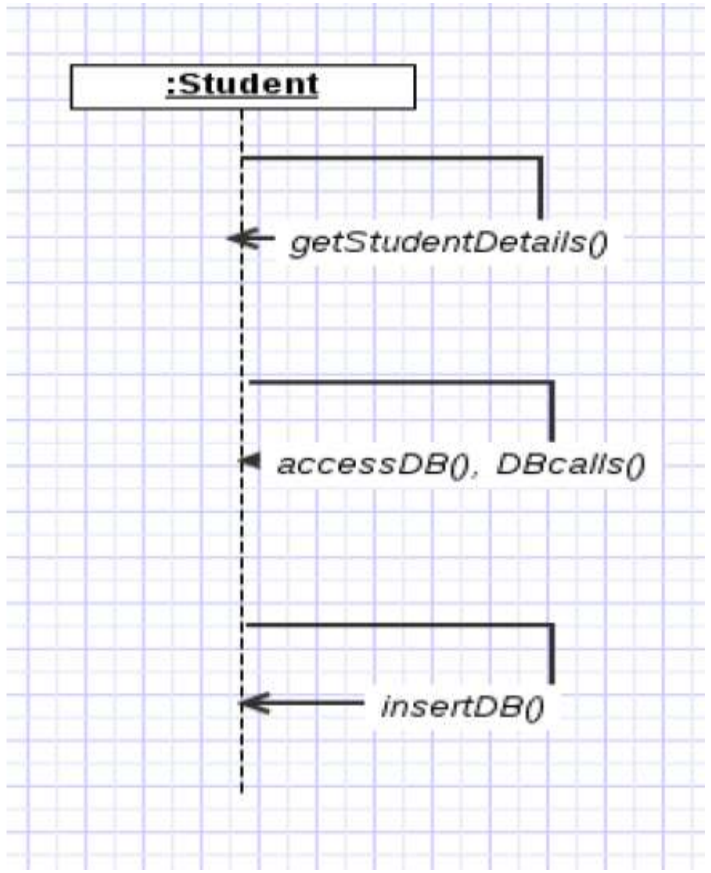
```
userClass{
    getUserDetails(){
    }
}
```

```
EmailClass{
    sendEmail(){
    }
}
```

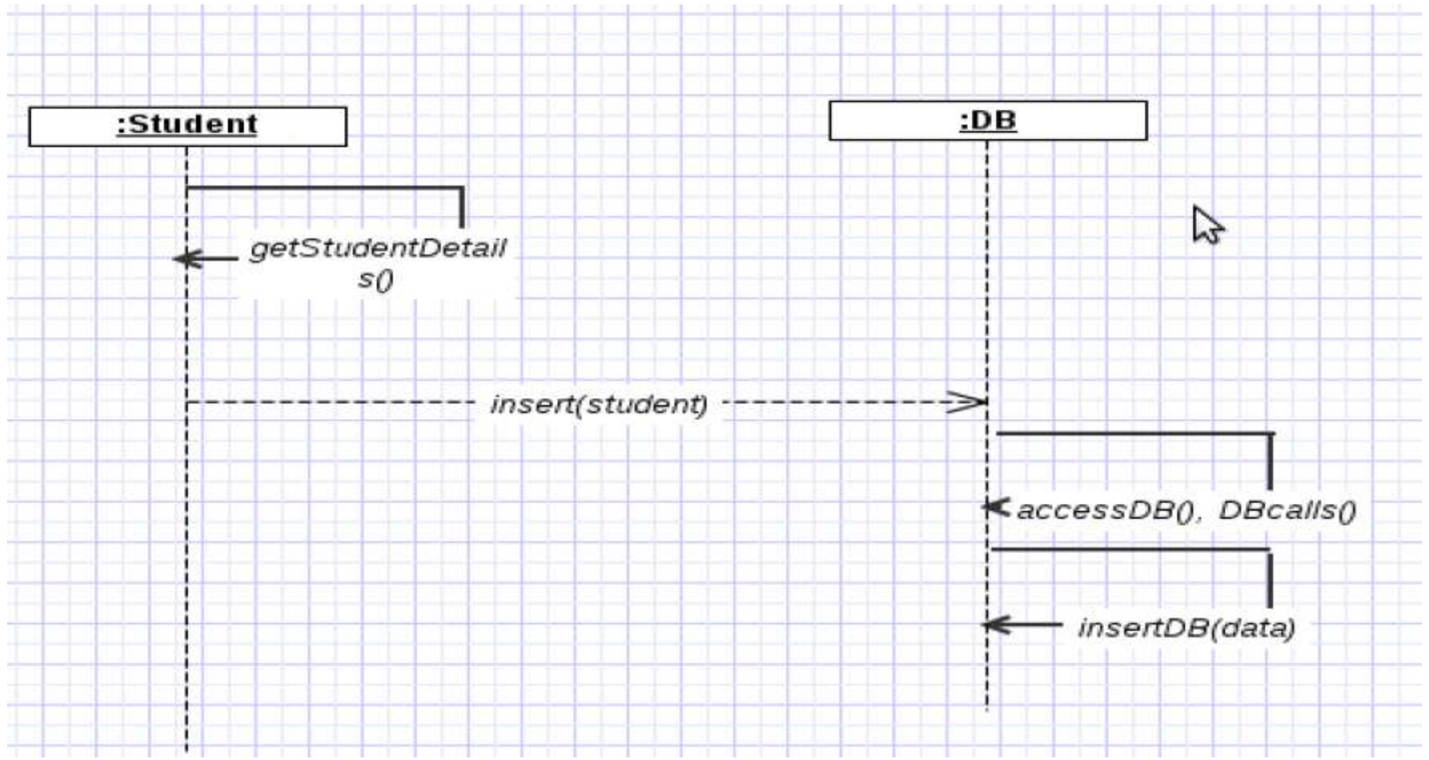
```
validationClass{
    validateEmail{
    }
}
```

Fig. High cohesion

Example of Low Cohesion



Example of High Cohesion



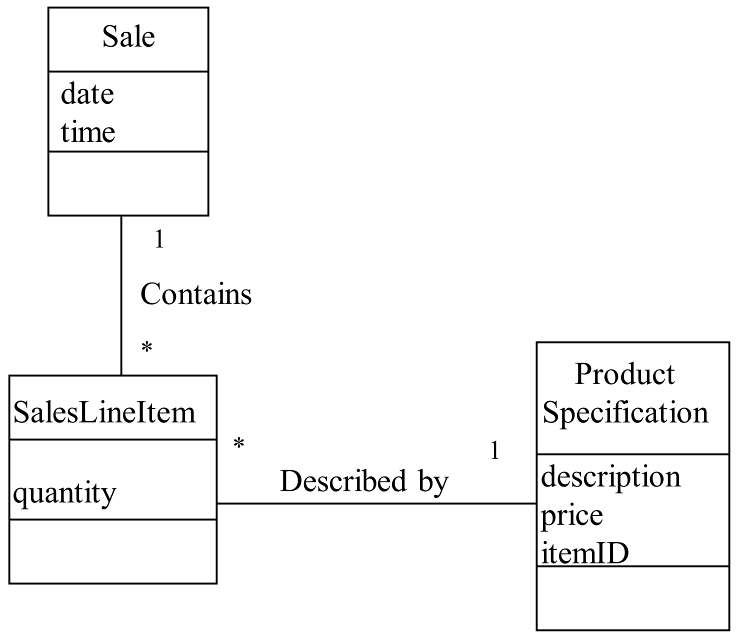
Pure Fabrication

- If domain model provides no reasonable concept to assign responsibility without violating cohesion/coupling -> create a new abstraction (e.g., PersistantStorage).

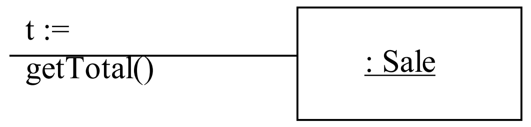
- Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept—something made up, to support high cohesion, low coupling, and reuse

GRASP Pattern

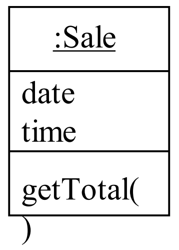
Initial Domain Model



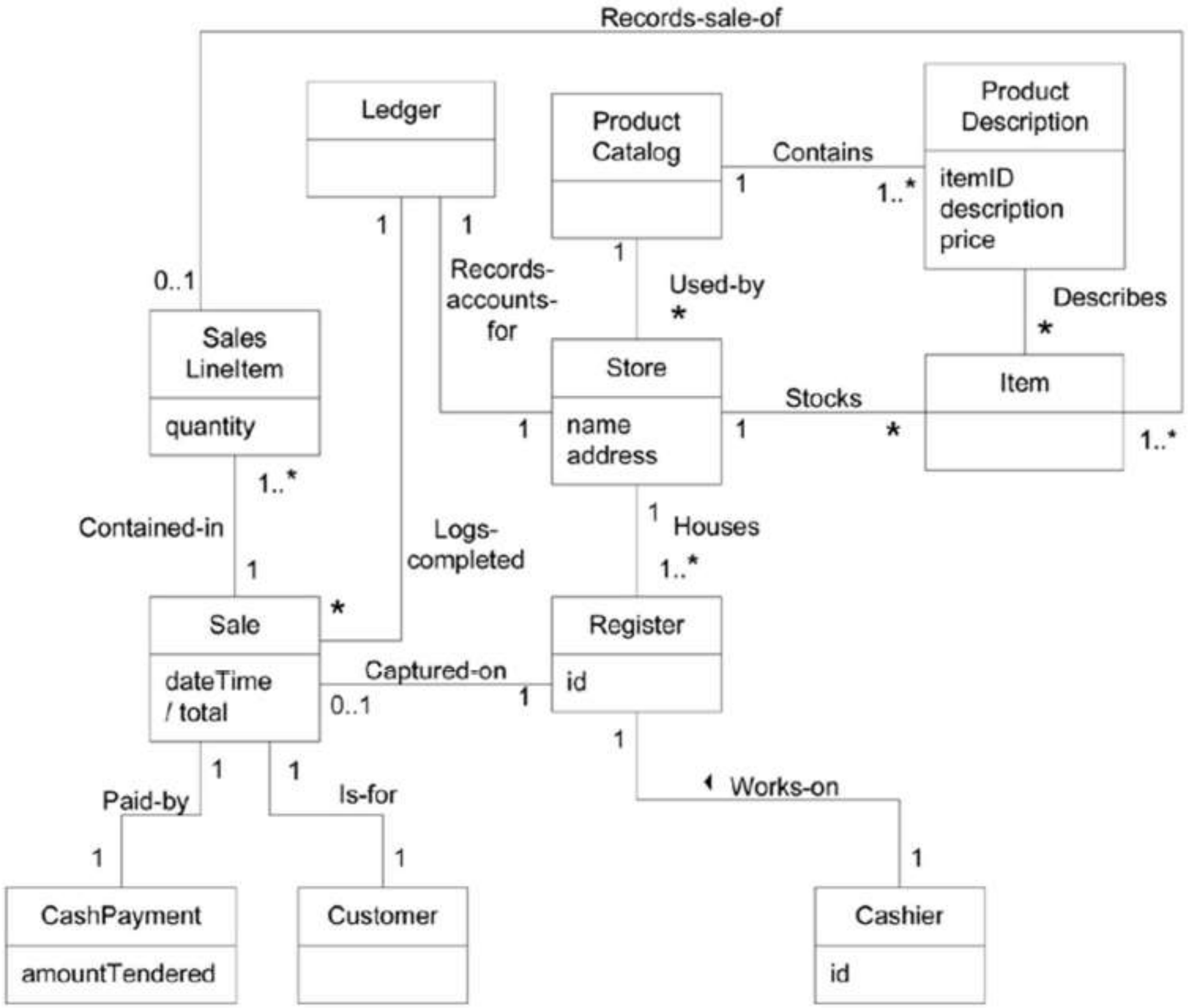
Sequence Diagram



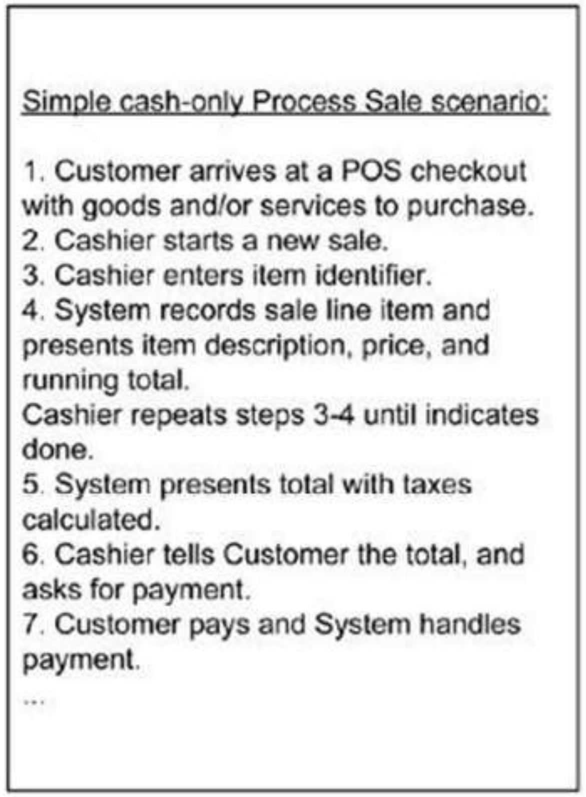
Software Class



Domain Model

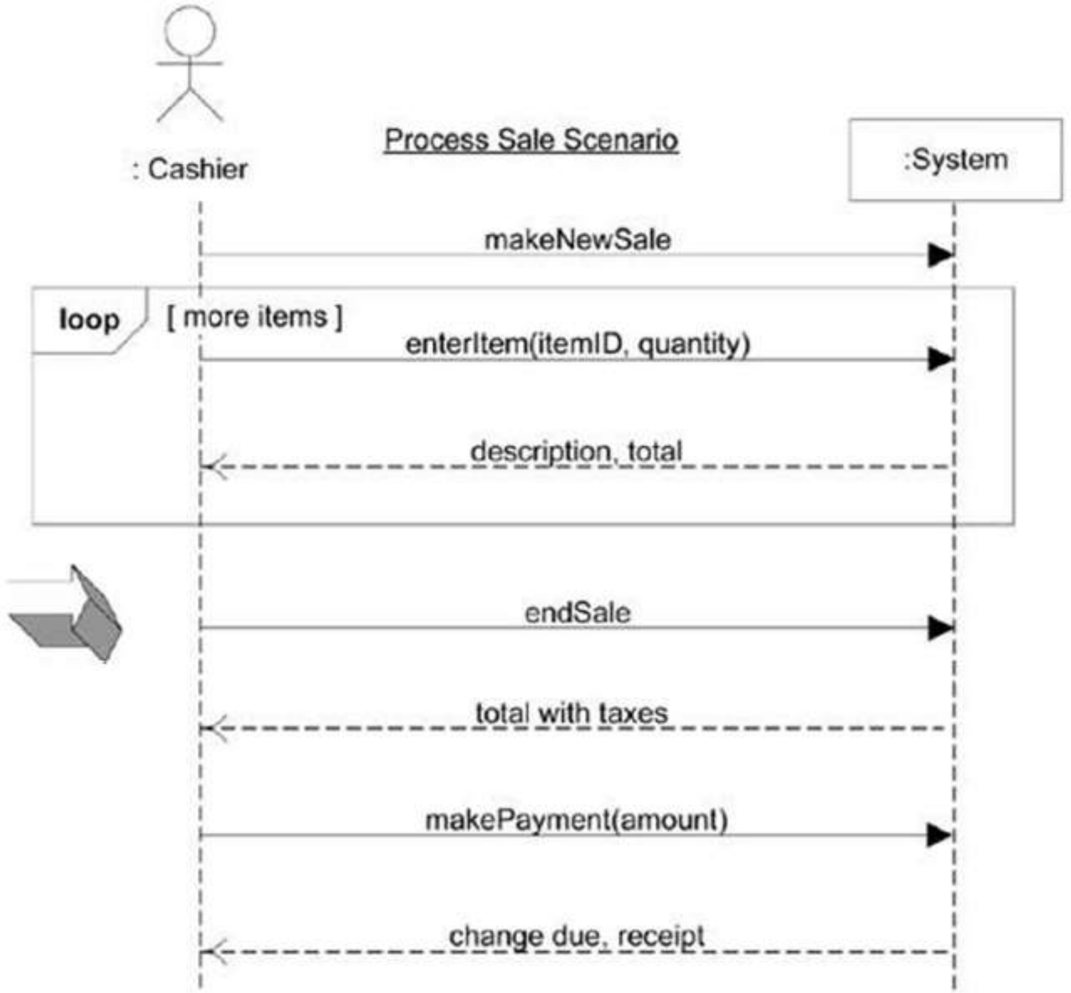


System Sequence Diagram



Simple cash-only Process Sale scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
...

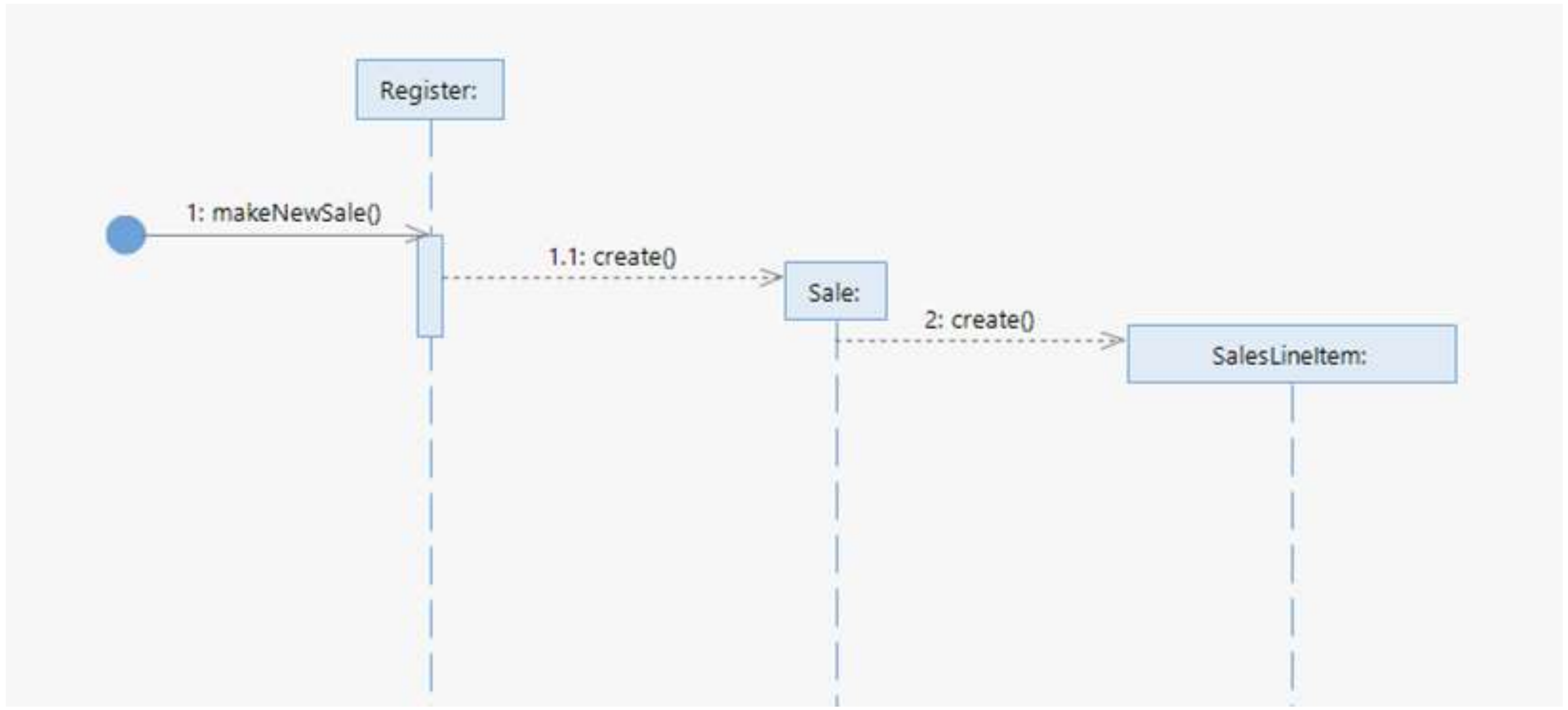


makeNewSale()

Controller ?

Information Expert ??

Creator ??



Explanation

- *Register* may be thought of as recording a *Sale*
  - *Register* is a reasonable candidate for creating a *Sale.*
  - *By* having the *Register* create the *Sale,* the *Register* can easily be associated with it over time,
  - During future operations within the session, the *Register* will have a reference to the current *Sale* instance.

- When the *Sale* is created
  - it must create an empty collection (container*)* to record all future *SalesLineItem* instances that will be added.
  - This collection will be contained within and maintained by the *Sale* instance,

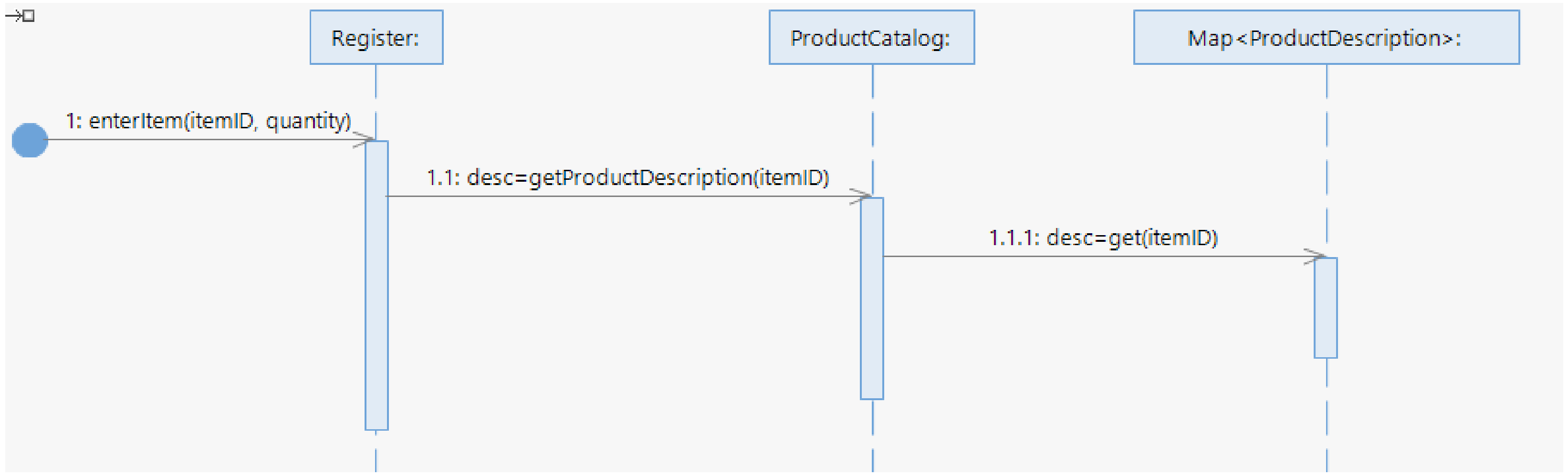
- Therefore:
  - the *Register* creates the *Sale*
  - the *Sale* creates an empty collection, represented by a multiobject in the interaction diagram.

enterNewItem(itemID, quantity)

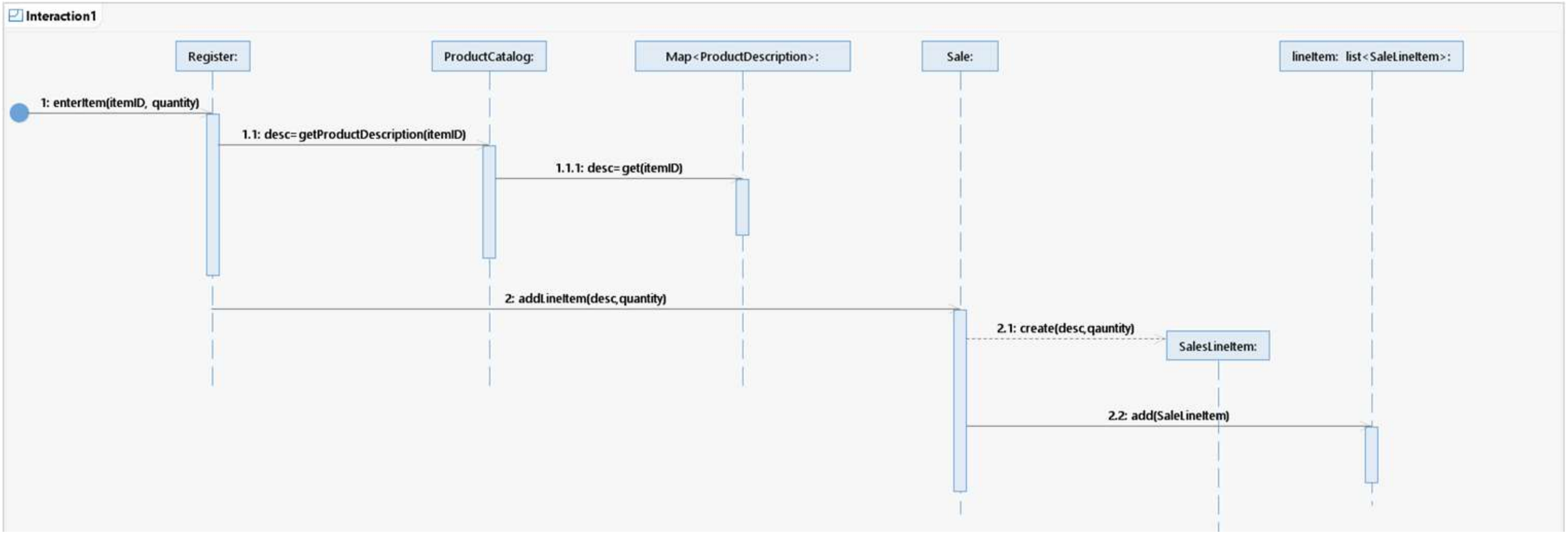
Controller ?

Information Expert ??

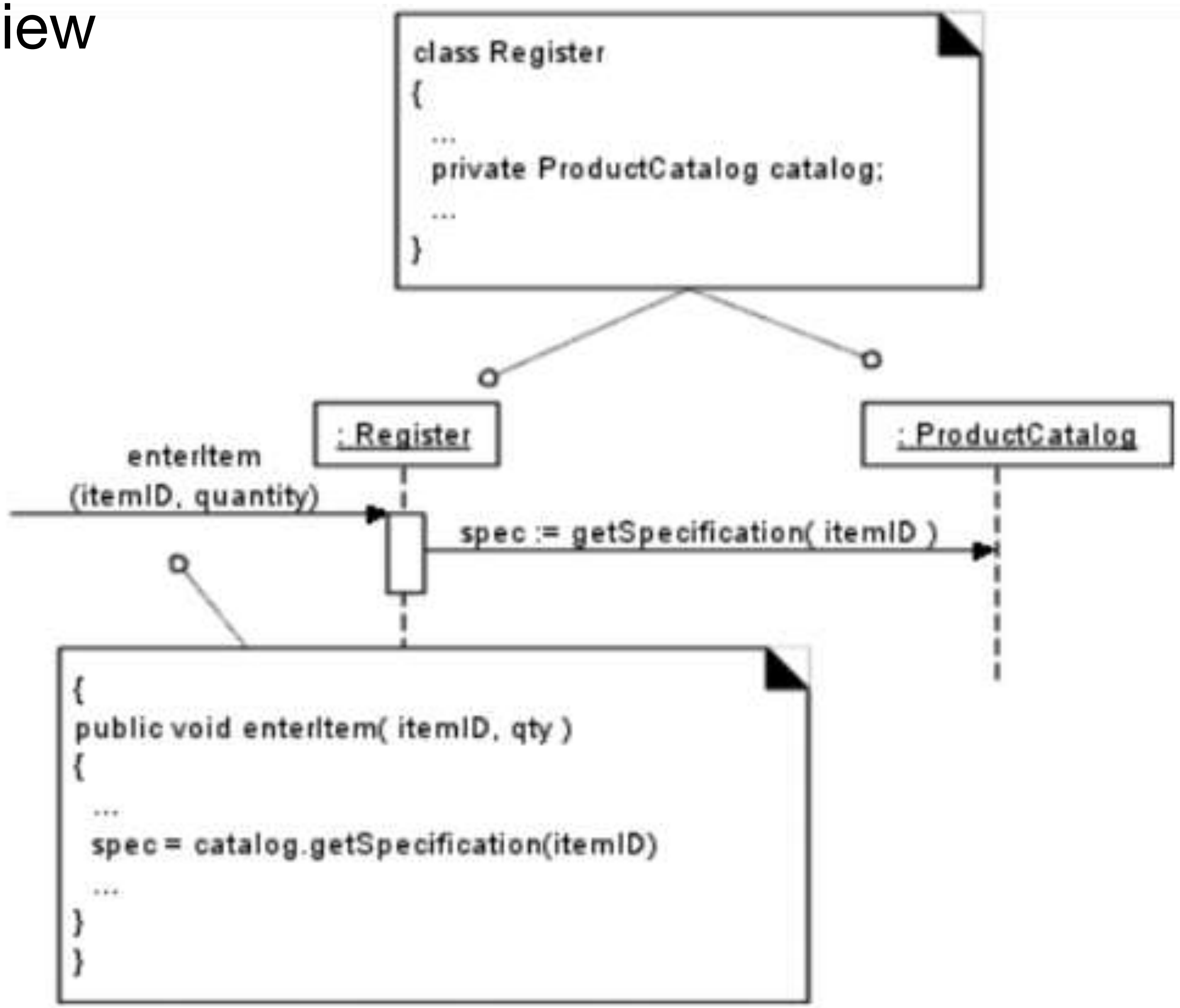
Creator ??



First step: access **ProductDescription** based on the itemID
ProductCatalog is information expert of ProductDescriptions


Interaction1
Register:
ProductCatalog:
Map<ProductDescription>:
Sale:
lineItem: list<SaleLineItem>:
1: enterItem(itemID, quantity)
1.1: desc=getProductDescription(itemID)
1.1.1: desc=get(itemID)
2: addLineItem(desc,quantity)
2.1: create(desc,qauntity)
SalesLineItem:
2.2: add(SaleLineItem)

Code level view

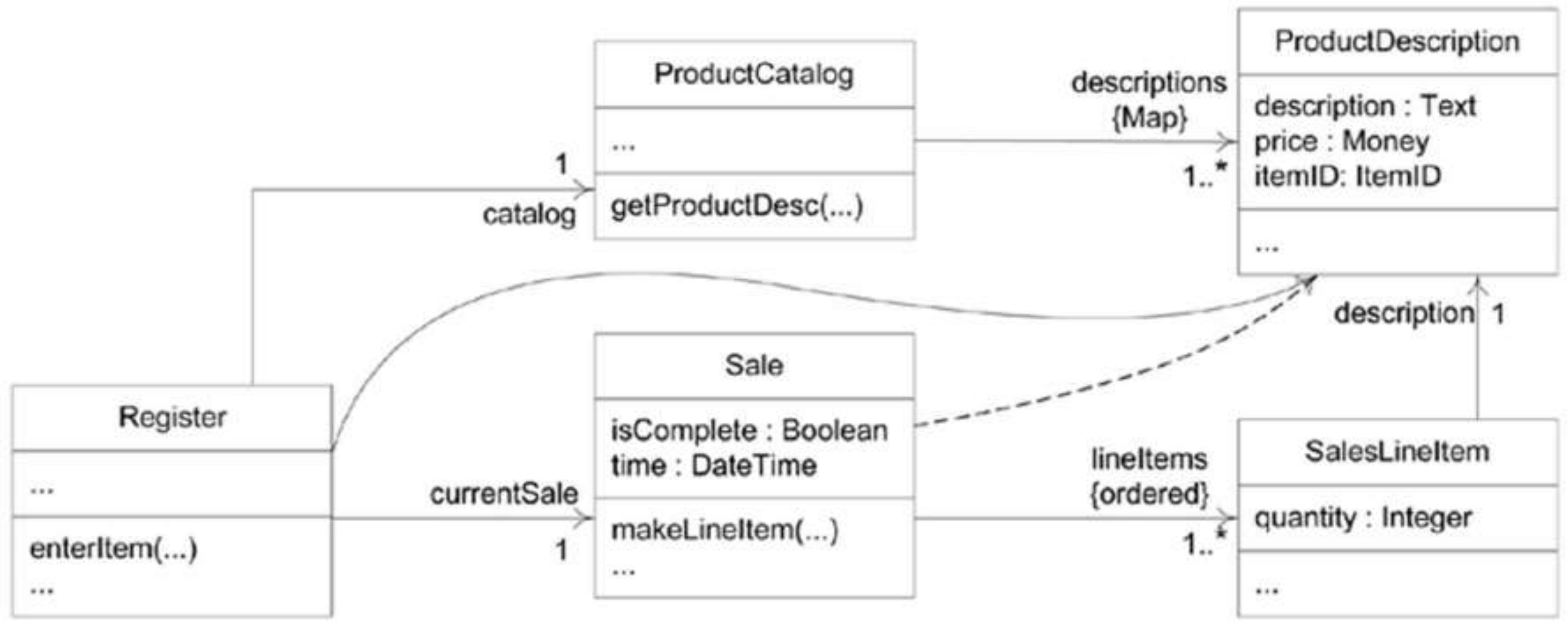


: Register
: ProductCatalog
enterItem (itemID, quantity)
spec := getSpecification(itemID)

```
class Register
{
  ...
  private ProductCatalog catalog;
  ...
}
```

```
{
public void enterItem( itemID, qty )
{
  ...
  spec = catalog.getSpecification(itemID)
  ...
}
}
```

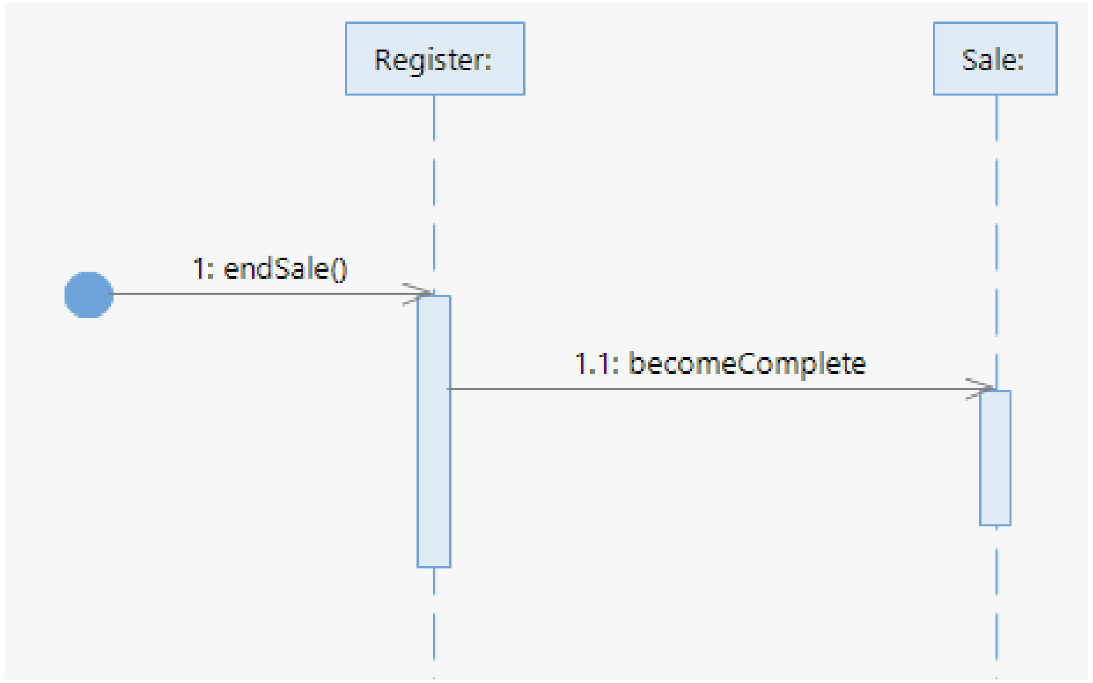
Class Diagram, So far

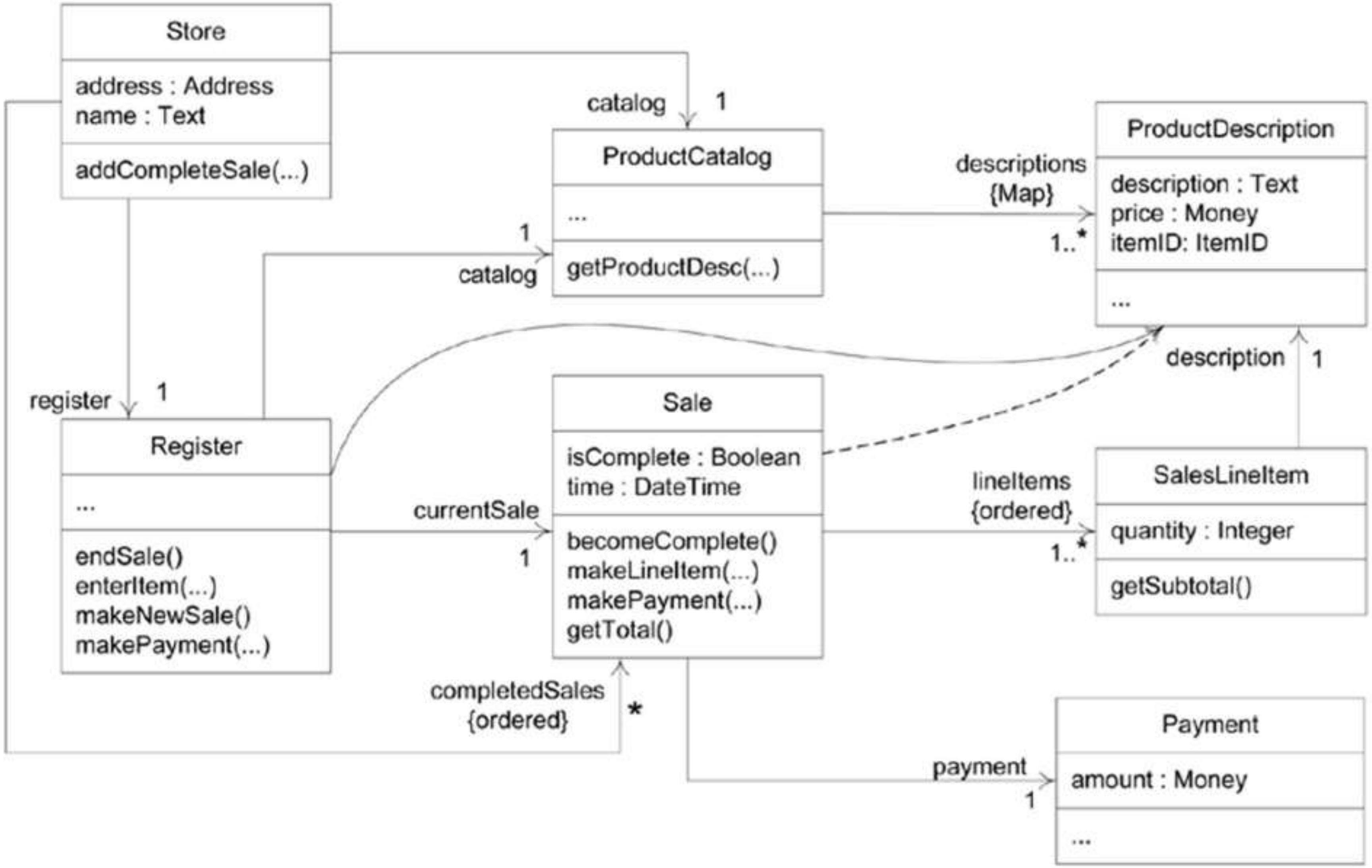


endSale – Design Decisions

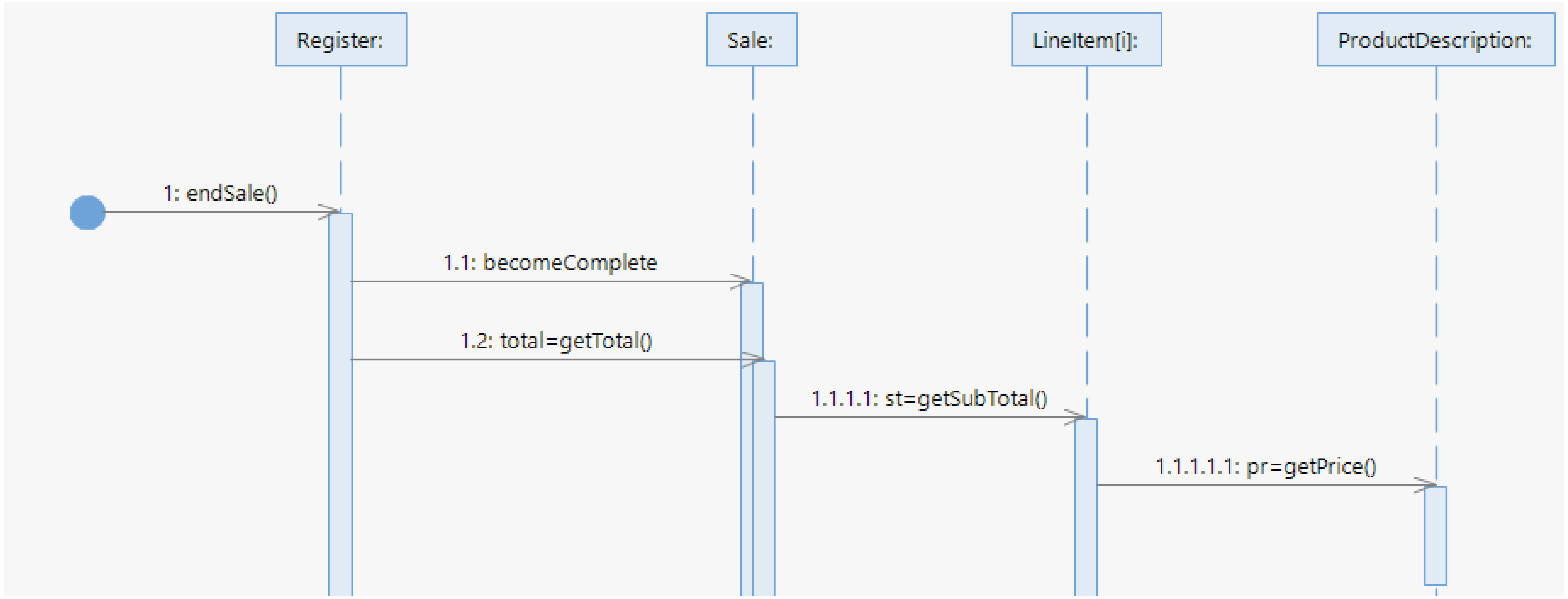
- Sale is to be completed
- Total with tax calculated is presented

endSale()

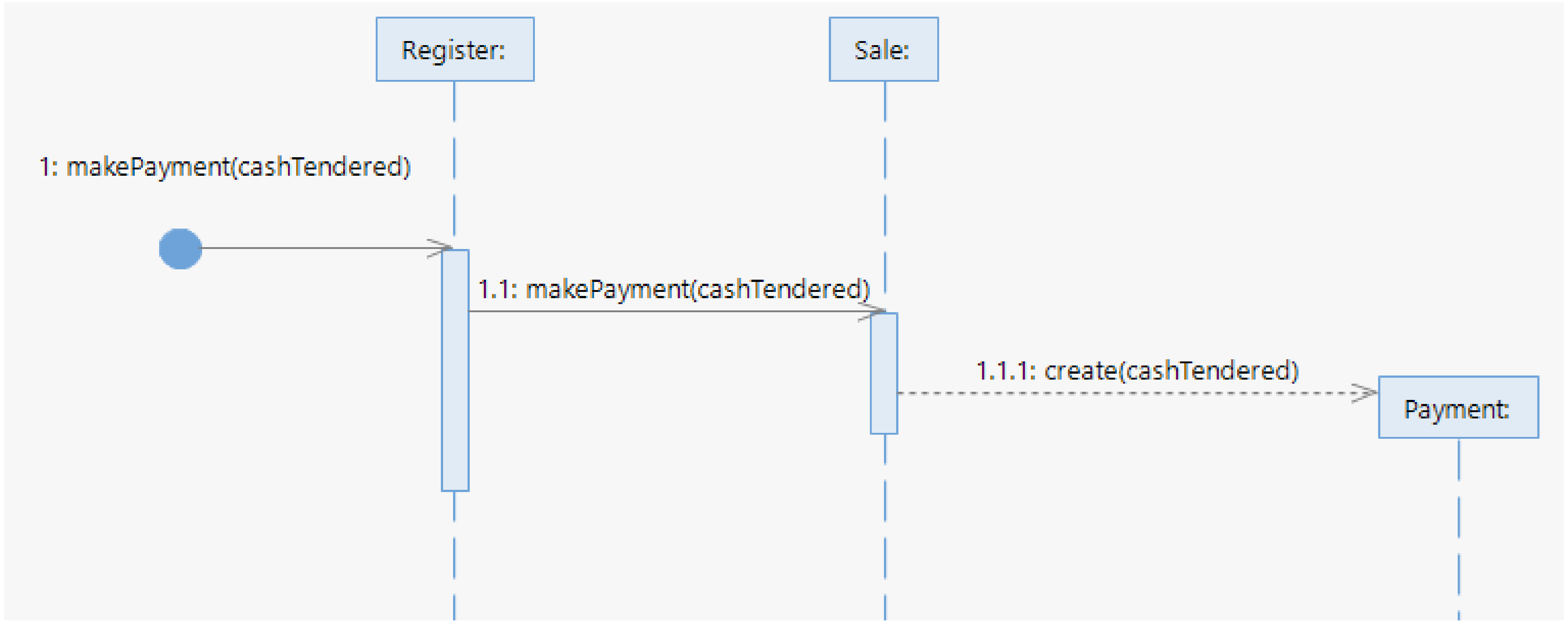


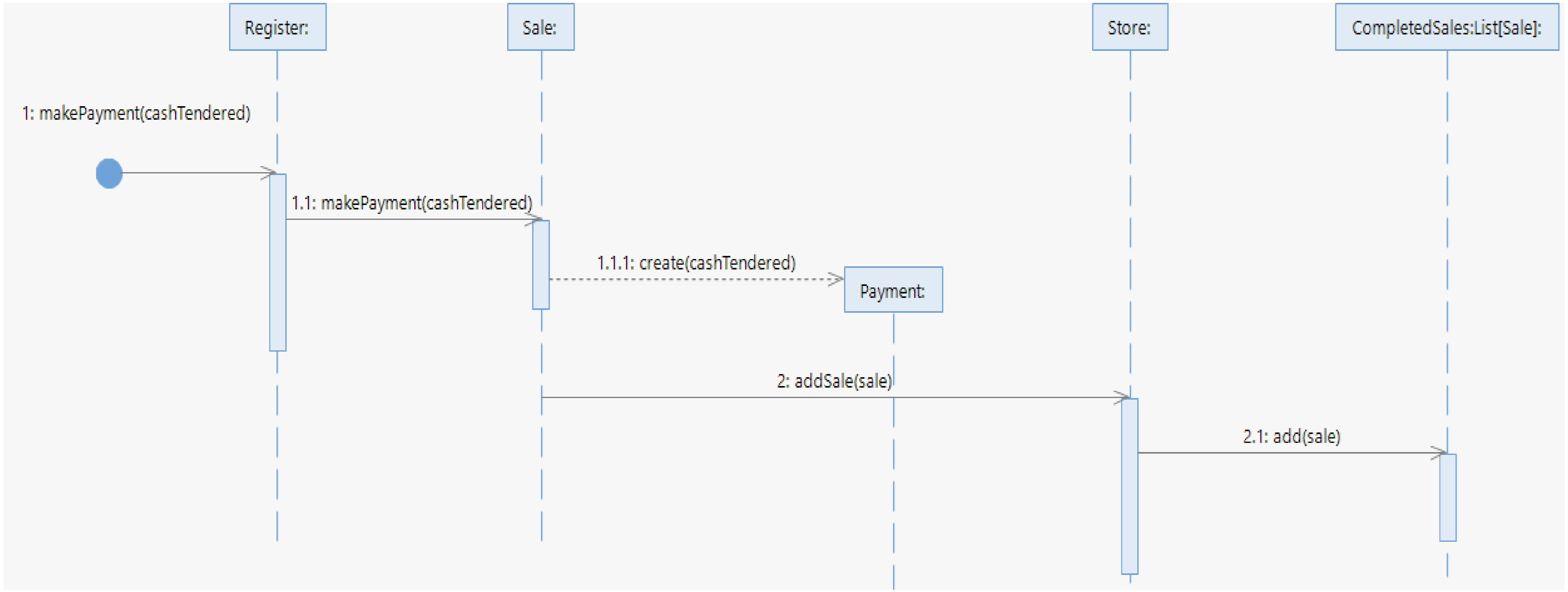

Store
address : Address
name : Text
addCompleteSale(...)
catalog
1
ProductCatalog
...
getProductDesc(...)
descriptions
{Map}
1..*
ProductDescription
description : Text
price : Money
itemID: ItemID
...
1
catalog
description
1
register
1
Register
...
endSale()
enterItem(...)
makeNewSale()
makePayment(...)
currentSale
1
Sale
isComplete : Boolean
time : DateTime
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()
lineItems
{ordered}
1..*
SalesLineItem
quantity : Integer
getSubtotal()
completedSales
{ordered}
*
Payment
amount : Money
...
payment
1

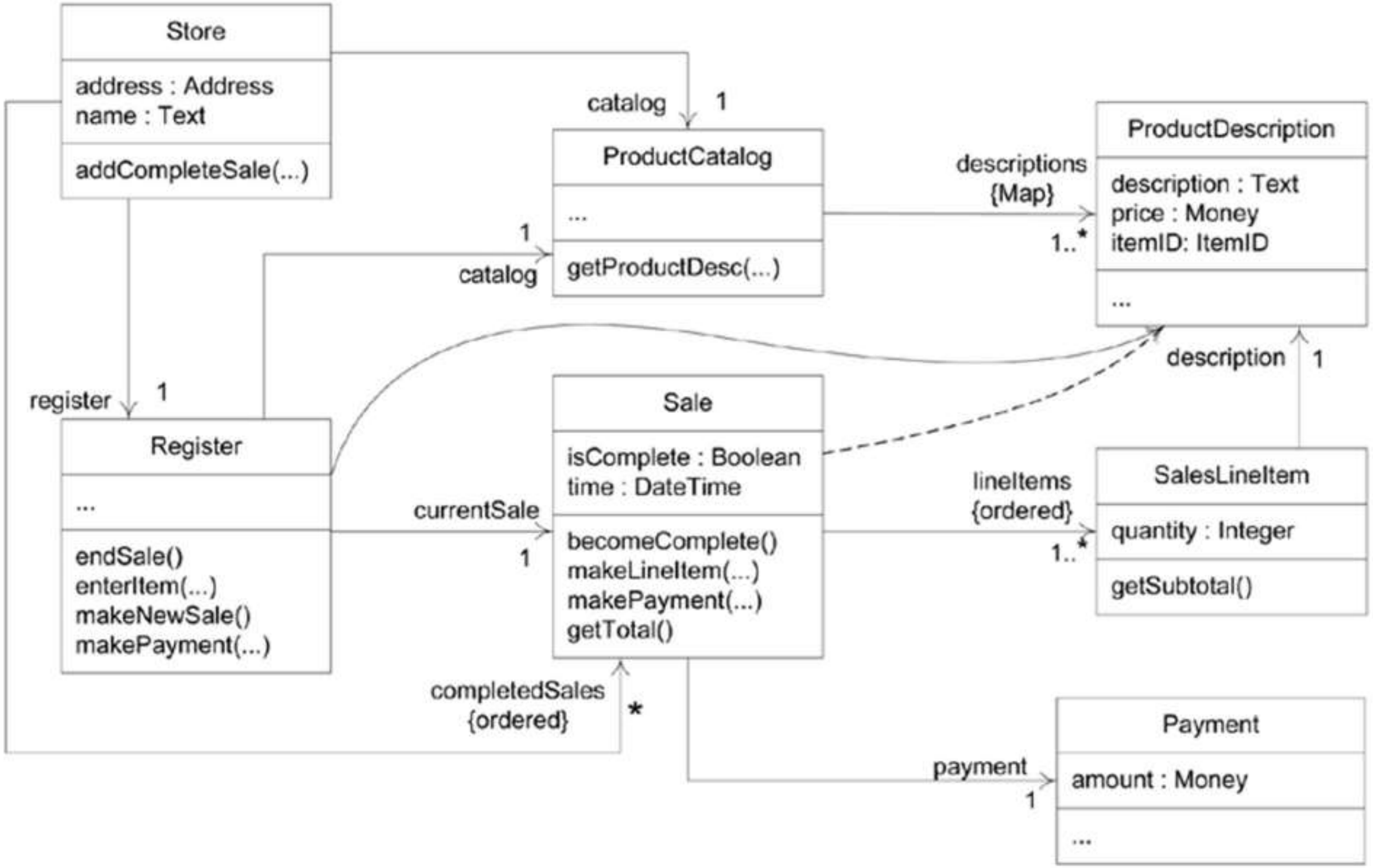
getTotal()


Register:
Sale:
LineItem[i]:
ProductDescription:
1: endSale()
1.1: becomeComplete
1.2: total=getTotal()
1.1.1.1: st=getSubTotal()
1.1.1.1.1: pr=getPrice()

makePayment(cashTendered)


Register:
Sale:
1: makePayment(cashTendered)
1.1: makePayment(cashTendered)
1.1.1: create(cashTendered)
Payment:


Register:
Sale:
Store:
CompletedSales:List[Sale]:
1: makePayment(cashTendered)
1.1: makePayment(cashTendered)
1.1.1: create(cashTendered)
Payment:
2: addSale(sale)
2.1: add(sale)


Store
address : Address
name : Text
addCompleteSale(...)
catalog
1
ProductCatalog
...
getProductDesc(...)
descriptions
{Map}
1..*
ProductDescription
description : Text
price : Money
itemID: ItemID
...
1
catalog
register
1
Register
...
endSale()
enterItem(...)
makeNewSale()
makePayment(...)
description
1
Sale
isComplete : Boolean
time : DateTime
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()
currentSale
1
lineItems
{ordered}
1..*
SalesLineItem
quantity : Integer
getSubtotal()
completedSales
{ordered}
*
Payment
amount : Money
...
payment
1