

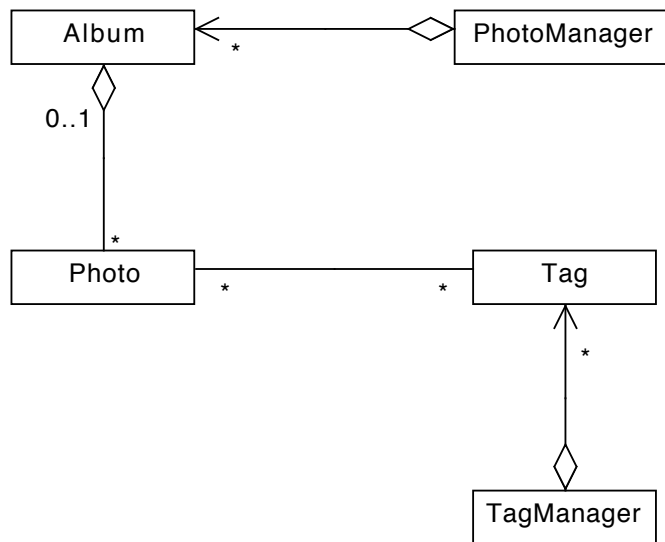
## CPSC 210

### Sample Final Exam Questions - Solution

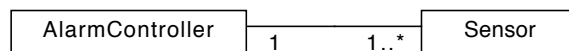
---

**Don't even think about** looking at these solutions until you've put significant effort into developing your own solution to these problems!!!

Q1.



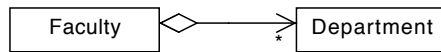
Q2.



- i. Name used to describe this relationship: **bidirectional association**

Point form description of what diagram communicates about relationship between classes:

- **each AlarmController object is associated with at least one Sensor object**
- **each Sensor object is associated with only one AlarmController object**
- **AlarmController can access ("knows about") services/behaviours provided by the Sensor class and vice-versa**



- ii. Name used to describe this relationship: **aggregation**

Point form description of what diagram communicates about relationship between classes:

- each **Faculty** object is associated with many **Department** objects
- we consider **Faculty** to be the "whole" and **Department** objects to be the "parts"
- **Faculty** can access ("knows about") services/behaviours provided by the **Department** class but *not* vice-versa

- Q3a)** i. You are writing a system to manage a hockey pool. For each participant in the pool, you must be able to track a team of players. What data structure will you use to represent the team of players? Why?

**HashSet<Player>**

We assume we have a **Player** class to represent a hockey player. We won't want to add a particular player to the team more than once and so we use a **Set** which does not allow for duplicate entries. **HashSet** is an implementation of the **Set** interface which provides efficient implementations of the **add**, **remove** and **contains** methods - all in  $O(1)$  time.

- ii. You are writing a system to model line-ups at the bank. Each teller has their own line-up. What data structure will you use to store all the people in line at all of the tellers?

**ArrayList<LinkedList<Customer>>**

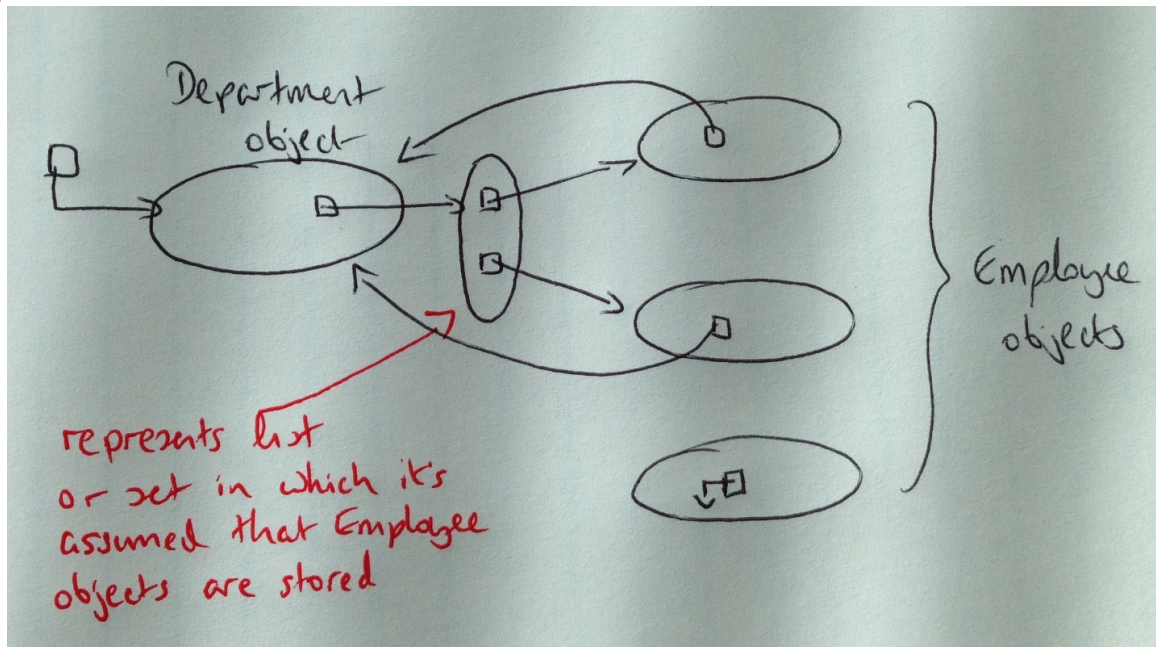
We assume that we have a **Customer** class to represent a customer at the bank. We represent each line-up using a **LinkedList** as **LinkedList** implements the **Queue** interface and can therefore be used to maintain customers in first-in, first-out (FIFO) order. Given that there is more than one teller (and therefore more than one line-up), we use an **ArrayList** to store each of the **LinkedLists**.

- b) Suppose you are designing a course registration system. Assume that there is a `Student` and a `Course` class in the system. How would you store the students and courses so that you can quickly retrieve the courses in which a given student is registered?

`HashMap<Student, HashSet<Course>>`

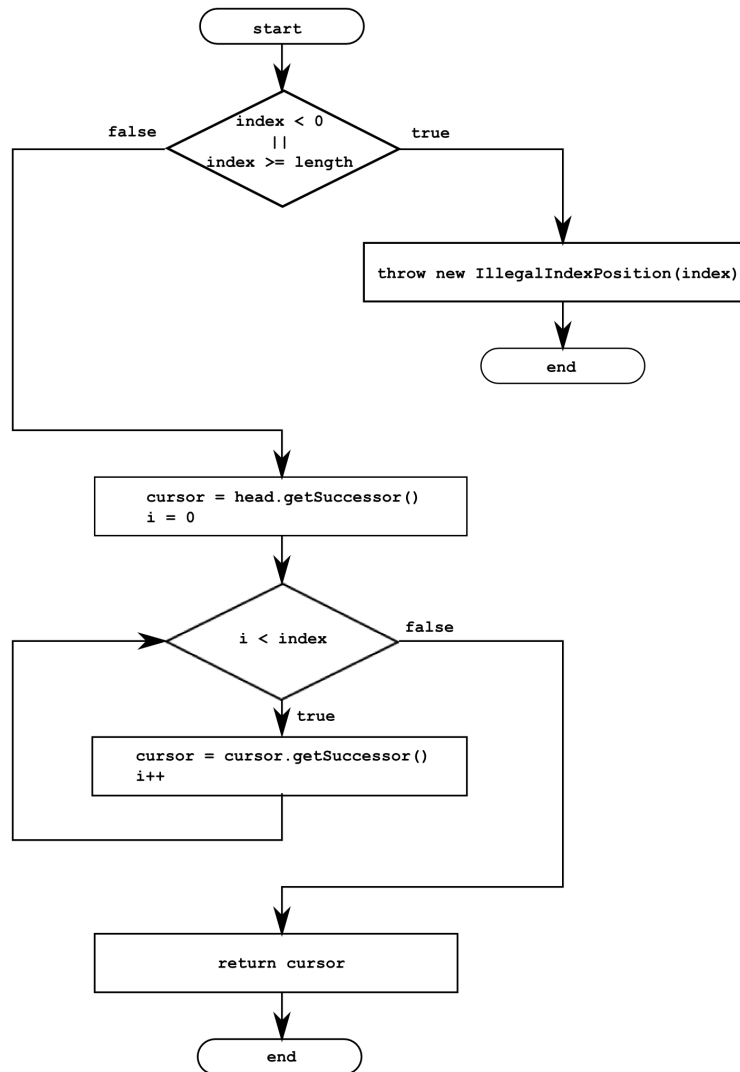
For each student we want to be able to quickly retrieve the courses in which he/she is registered. We therefore use a map with the student as key and the courses in which the student is registered as the corresponding value. Given that a student won't register in a course more than once at any given time, we represent the collection of courses using a `Set`.

Q4a)



- b) See the `HRSystemComplete` repository on GitHub.

**Q5.** Draw an intra-method control flow diagram for the *iterative version* of the method:  
`MyListNode<E> find(int index) throws IllegalIndexPosition;`  
 in the `ubc.cpsc210.list.linkedList.MyLinkedList` class of the project  
 P-LinkedListComplete



**Q6a)** What does it mean for the design of a class to be robust?

**First, all of the methods in the class must be robust. A method is robust if its specification covers all possible input values to that method.**

**In addition, we must specify class invariants and ensure that they hold before any method in the class is called and immediately after any of those methods executes. The invariants specify what an operation can assume about the state of an instance of that class at any time.**

- b)** Design and implement a class that represents an *unchecked* exception that will be thrown by the following method of a `MyArrayList<E>` class when the index is not valid:

```
public E get(int index);
```

The class must provide a constructor that takes the invalid index as a parameter and uses it to construct a meaningful error message that can be displayed when the exception is caught.

```
/**
 * Represents an exception raised when an illegal index
 * is used.
 */
public class IllegalIndexPosition extends RuntimeException {
    /**
     * Constructor
     * @param index the illegal index
     */
    public IllegalIndexPosition(int index) {
        super("The index " + index + " is not valid.");
    }
}
```

**Q7.** This question refers to the class `ubc.cpsc210.list.linkedList.MyLinkedList` from the `LinkedListComplete` project.

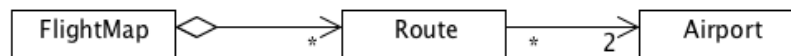
Complete the implementation of the following member of the `linkedList.MyLinkedList` class. Assume that `Collection<E>` is from the `java.util` package.

```
/**
 * Returns true if this list contains all the elements in the
 * collection c, false otherwise.
 */
public boolean containsAll(Collection<E> c) {
    for (E next : c) {
        if (!contains(next))
            return false;
    }

    return true;
}
```

- Q8.** Provide an implementation for the classes shown in the UML diagram below. You must include any fields or methods necessary to support the relationship between the classes in addition to appropriate getters and setters. Note that a route has two associated airports: the departure airport and the arrival airport. Each airport has a unique code (e.g. "YVR" represents Vancouver International, "LHR" represents London Heathrow and "PEK" represents Beijing) which cannot be changed.

Assume that once set, the arrival and departure airports for a particular route cannot be changed. Further assume that routes can be added to or removed from a flight map but the same route cannot be added to the flight map more than once. We consider two routes to be the same if they have the same departure and arrival airports. Two airports are the same if they have the same code.



```
public class Airport {
    private final String code;

    public Airport(String code) {
        this.code = code;
    }

    public String getCode() {
        return code;
    }

    // Generate these methods using Eclipse!
    @Override
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this.getClass() != o.getClass())
            return false;

        Airport other = (Airport) o;

        return (code.equals(other.code));
    }

    @Override
    public int hashCode() {
        return code.hashCode();
    }
}
```

```

public class Route {
    private final Airport departure;
    private final Airport arrival;

    public Route(Airport dep, Airport arr) {
        departure = dep;
        arrival = arr;
    }

    public Airport getDepartureAirport() {
        return departure;
    }

    public Airport getArrivalAirport() {
        return arrival;
    }

    // Generate these methods with Eclipse!
    @Override
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this.getClass() != o.getClass())
            return false;

        Route other = (Route) o;

        return (other.arrival.equals(this.arrival)
            && other.departure.equals(this.departure));
    }

    @Override
    public int hashCode() {
        return arrival.hashCode() * 13
            + departure.hashCode();
    }
}

public class FlightMap {
    private Set<Route> routes;

    public FlightMap() {
        routes = new HashSet<Route>();
    }

    public void addRoute(Route r) {
        routes.add(r);
    }

    public void removeRoute(Route r) {
        routes.remove(r);
    }
}

```



**Q9.** You have been asked to alter the `PhotoAlbumBase` system to make it possible for a method containing the following code to compile and execute correctly:

```
Album anAlbum = new Album("My Album");
// Put lots of photos into album
//...
for (Photo p: anAlbum) {
    // do something with each photo
}
```

For each interface, class, field or method that must be changed or added, describe the change or addition in as close to correct Java syntax as you can.

**The class declaration must change:**

```
public class Album implements Iterable<Photo> {
```

**We must add the method iterator to Album...**

```
public Iterator<Photo> iterator() {
    return photos.iterator();
}
```

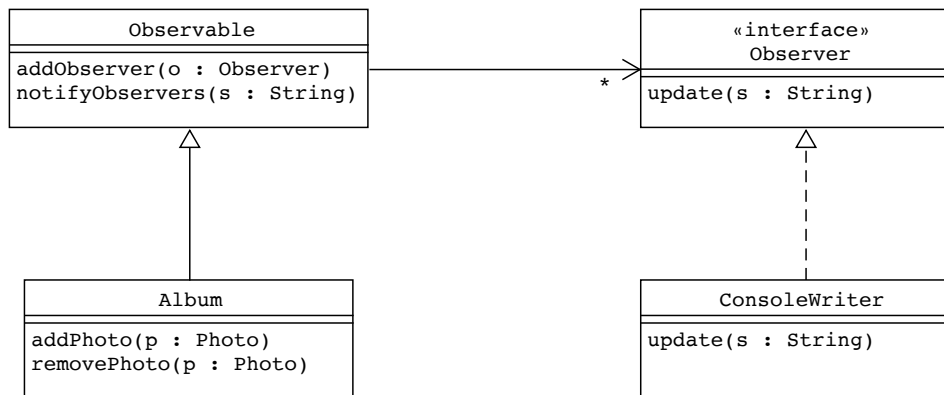
**Q10.** You have been given a class `ConsoleWriter` that can write a given string to the console.

```
public class ConsoleWriter {

    private void writeToConsole(String s) {
        System.out.println(s);
    }
}
```

You have been asked to alter the functionality of the `PhotoAlbumBase` system to write to the console the name of any photo added or deleted from an album in the system. You have been told you must use the Observer design pattern to implement this new functionality.

- a) Draw a UML class diagram that provides an overview of the changes and additions needed to `PhotoAlbumBase` to support the use of the Observer design pattern to provide the desired functionality. You need not reproduce the entire UML class diagram for the system. Just show those parts of the UML class diagram that must change. Indicate fields and methods that must be changed or added in the UML class diagram.



b) For each interface, class, field or method that must be changed or added, describe the change or addition in as close to correct Java syntax as you can.

### 1) Implement Observable.

```

public class Observable {

    List<Observer> observers;

    public Observable() {
        observers = new ArrayList<Observer>();
    }

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void notifyObservers(String s) {
        for (Observer o: observers)
            o.update(s);
    }
}
  
```

### 2) Implement Observer.

```

public interface Observer {
    public abstract void update(String s);
}
  
```

### 3) Declaration of Album must change:

```

public class Album extends Observable {...
  
```

**4) Album's constructor must change to include:**

```
public Album() {  
    ...  
    ConsoleWriter cw = new ConsoleWriter();  
    addObserver(cw);  
}
```

**5) Change ConsoleWriter's declaration:**

```
public class ConsoleWriter implements Observer {...
```

**6) Must alter addPhoto(...) and removePhoto(...) to add notification to observers:**

```
public void addPhoto(Photo p) {  
    ...  
    notifyObservers(p.getName());  
}  
  
public void removePhoto(Photo p)  
    ...  
    notifyObserver(p.getName());  
}
```

**7) Must provide implementation for update on ConsoleWriter:**

```
public void update(String s) {  
    writeToConsole(s);  
}
```

**Q11.** Consider the following Java class.

```
public class Clock {  
    private Integer startTime;  
    private Integer numHours;  
  
    public Clock( Integer start, Integer hrs ) {  
        startTime = start;  
        numHours = hrs;  
    }  
}
```

Modify this class and introduce any other code required, but without using any standard Java Collection Framework classes or interfaces (e.g., `List`), so that you can iterate over an instance of `Clock` using a for-each loop. For example, the following code should print the numbers 2, 3, 4, and 5. You do not have to consider the case where the hours go past 23. Assume the input you're given will produce output within the same day (i.e., values between 0-23 only).

```
for (Integer i: new Clock(2,4)) {
    System.out.println(i);
}

import java.util.Iterator;

public class Clock implements Iterable<Integer> {

    private Integer startTime;
    private Integer numHours;

    public Clock(Integer start, Integer hrs) {
        startTime = start;
        numHours = hrs;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new ClockIterator();
    }

    private class ClockIterator implements Iterator<Integer> {

        private int currTime;

        public ClockIterator(){
            currTime = startTime;
        }

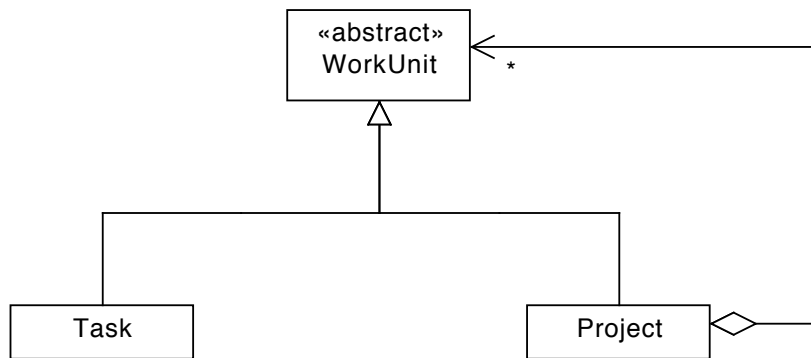
        @Override
        public boolean hasNext() {
            return currTime < (startTime + numHours);
        }

        @Override
        public Integer next() {
            Integer nextTime = currTime;
            currTime++;
            return nextTime;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

**Q12.** Suppose you are designing an Android app that will allow the user to perform task management. The user can add tasks to the system and can group tasks together into projects. Projects can be added as sub-projects of other projects, nested arbitrarily deep. Each task has an estimated time for completion that is specified when the task is constructed. You want to be able to treat individual tasks and projects in the same way. In particular, you want to be able to get the time needed to complete a task or a project. The time taken to complete a project is the total time needed to complete all the tasks in the project or in sub-projects of that project.

- a) Consider the partially completed code in the `TaskManager` project. Draw a UML diagram that includes all the classes in the `ca.ubc.cpsc210.taskmanager.model` package and that shows how you will apply the composite pattern to the design of this system.



- b) Write the complete implementation of the `Task` and `Project` classes below. Note that if your implementation is correct all the tests provided in `ca.ubc.cpsc210.taskmanager.tests.TestProject` should pass. Space is also provided on the following page for your answer to this question.

```
public class Task extends WorkUnit {
    private int hours;

    public Task(int hours) {
        this.hours = hours;
    }

    public int hoursToComplete() {
        return hours;
    }
}
```

```

public class Project extends WorkUnit {
    private List<WorkUnit> units;

    public Project() {
        units = new ArrayList<WorkUnit>();
    }

    public void add(WorkUnit wu) {
        units.add(wu);
    }

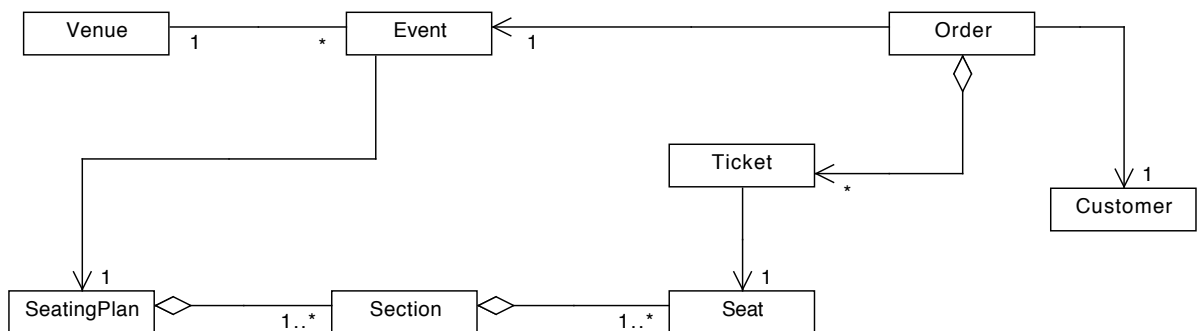
    public int hoursToComplete() {
        int total = 0;

        for( WorkUnit wu : units )
            total += wu.hoursToComplete();

        return total;
    }
}

```

**Q13.** The following UML class diagram represents the design of a "TicketWizard" system that allows tickets to be sold for different events held at venues across the country.



Answer the following questions based on the design presented in the UML diagram above and **not** on what you think the design should be! Circle **True** or **False** and briefly explain your choice. Note that you will earn no marks if your explanation is not correct.

- a) True or False: given a customer object, you can retrieve the orders placed by that customer.

**False: the link between Order and Customer is uni-directional from Order to Customer.**

- b) True or False: it is possible that an order has no tickets associated with it.

**True: the multiplicity on the Order end is \*, meaning 0 or more.**

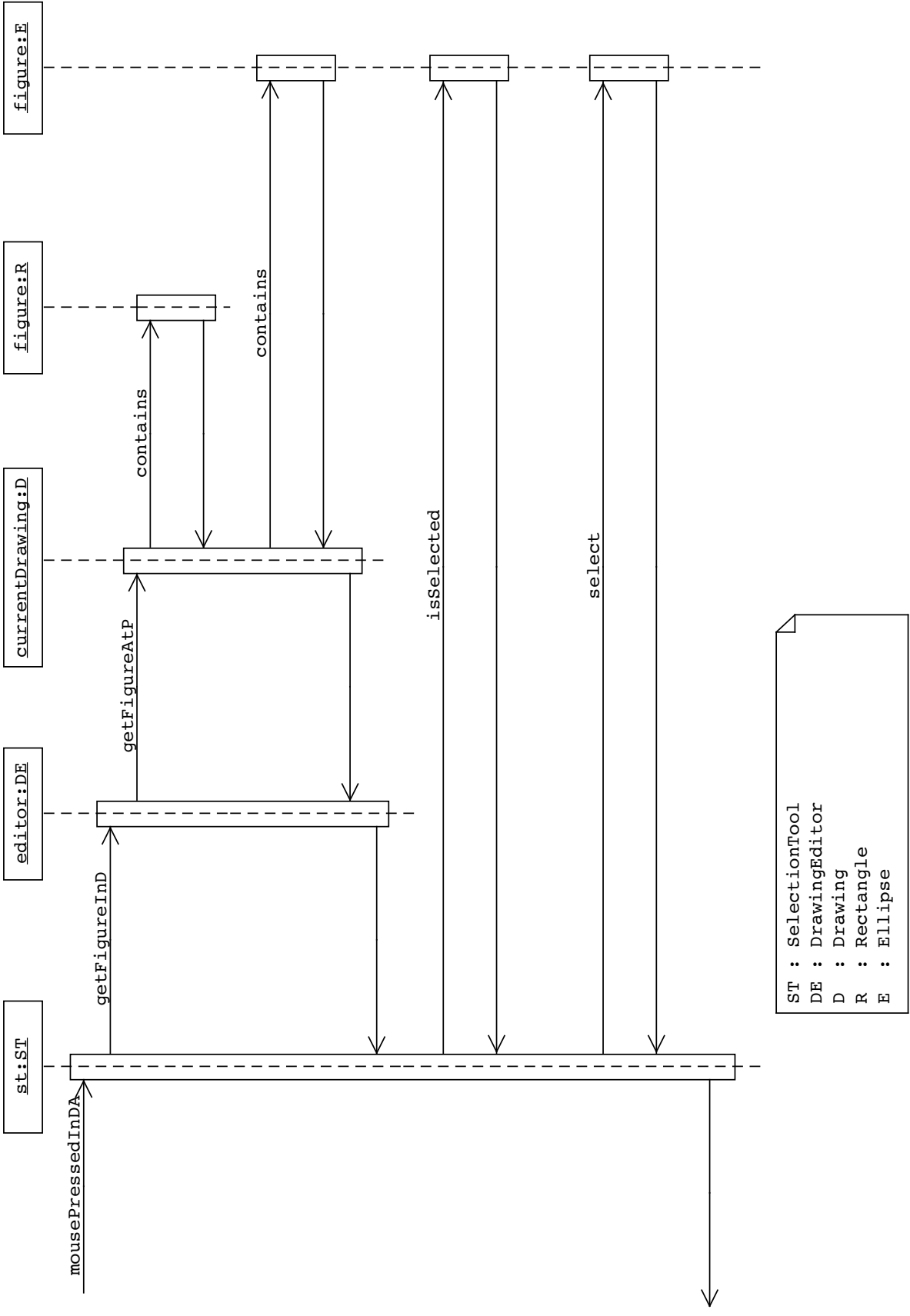
- c) True or False: given a ticket object, it is possible to retrieve the section object to which the associated seat belongs.

**True: you can get the corresponding seat directly and the section via the bi-directional association between Seat and Section.**

**Q14.** This question refers to the `CompositeDrawingEditorComplete` project. Note that a `Drawing` stores the figures drawn by the user in a list in the order in which they were drawn. In this question you must assume that the user has drawn three figures in the order listed here: a rectangle, an ellipse and a square. Now assume that the user chooses the "select" tool and clicks the ellipse figure. In so doing, the ellipse figure changes from "not selected" to "selected".

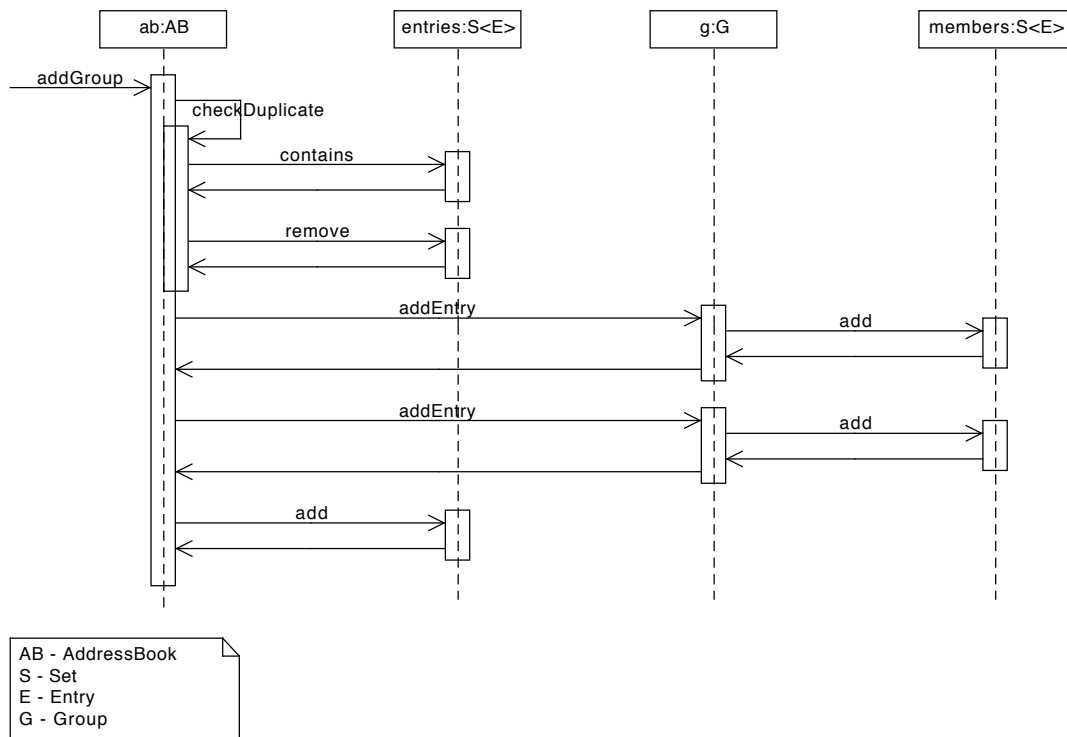
Draw a *sequence diagram* for the method

`SelectionTool.mousePressedInDrawingArea`, called as a result of the user clicking the ellipse figure with the "select" tool. It is not necessary to include calls to any methods in the Java library. Use the fact that you know which figures have been drawn to resolve calls to abstract methods. If you encounter a branch in the code, it is necessary to include only those methods that will actually execute in the scenario described above. You may abbreviate the names of types and methods provided you do not introduce any ambiguity into your diagram.

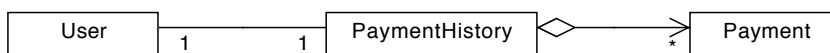




**Q15.** This question uses the `EmailManager` project found on GitHub. Draw a sequence diagram for the method `AddressBook.addGroup` within the package `ca.ubc.cs.cpsc210.addressbook`. If you have to loop over a collection, you must assume that there are exactly two objects in that collection. Include calls to *all* methods in the `EmailManager` project *except* constructors. You must also include the first level of calls to the Java library, if any. Be sure to include a legend if you abbreviate class names.



**Q16.** Consider the UML class diagram shown below:



It represents the design for a small part of an online storage system. Users have to pay for the service and a history of payments is maintained in the system. Write the code for the `PaymentHistory` class. You must include fields and methods that are necessary to support relationships between the other classes shown on the UML diagram but it is not necessary to include any others. Assume that you can add a `Payment` to the `PaymentHistory` but a `Payment` cannot be removed. Further assume that the payments must be stored in the order in which they were added and that the `PaymentHistory` must not contain duplicate `Payment` objects. Assume that the constructor of the `User` class creates the corresponding `PaymentHistory` object.

```

public class PaymentHistory {
    private List<Payment> payments;
    private User user;

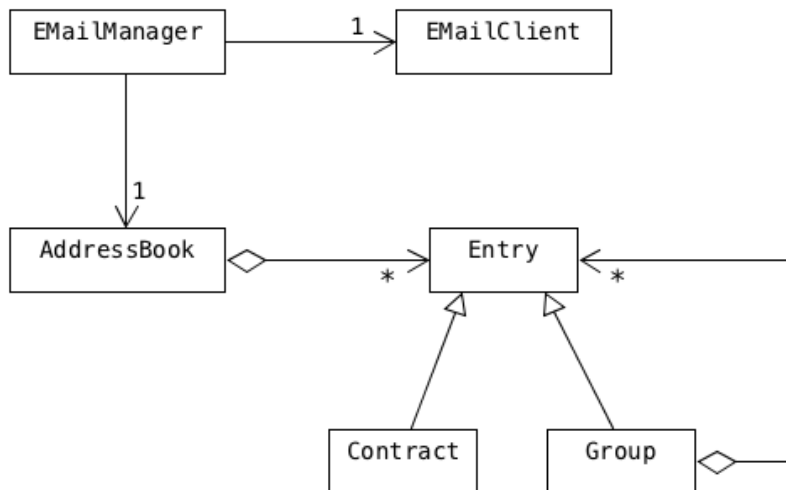
    public PaymentHistory(User user) {
        this.user = user;
        payments = new LinkedList<>(); // or ArrayList
    }

    public void addPayment(Payment p) {
        if (!payments.contains(p))
            payments.add(p);
    }
}

```

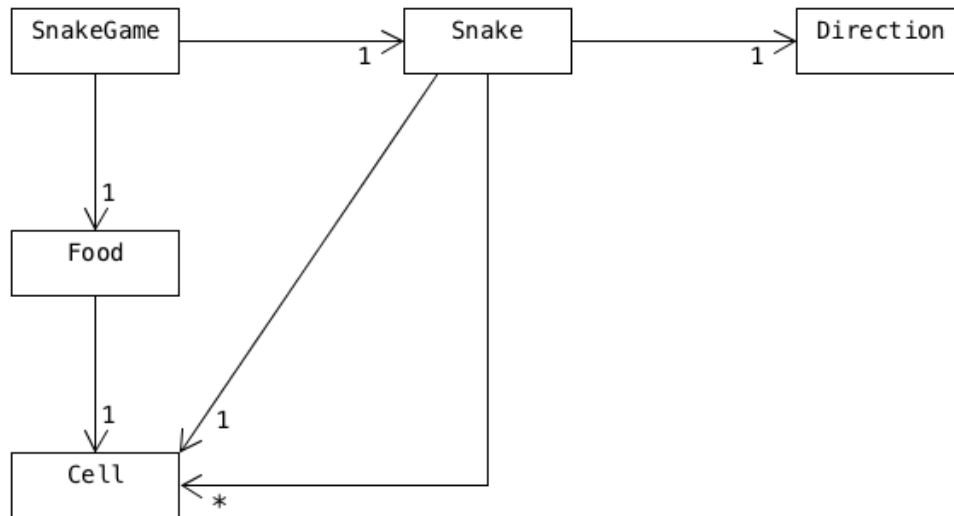
**Note:** this solution assumes that the user associated with a `PaymentHistory` object cannot be changed after the `PaymentHistory` object has been constructed.

**Q17.** Draw a UML class diagram to represent the design of all the types in the `addressBook` and `email` packages of the `EMailManager` system checked out from the lectures repository.



**Q18.**

- a) Draw a UML class diagram to represent the design of all the types in the `model` package of the `SnakeStarterLecLab` project found on GitHub. It is recommended that you check out a new copy of this code and assume that the `Food` class has been implemented according to the given specification.



b) Draw a UML sequence diagram to model the call to `Snake.move`. Include the first level of calls to the Java library. Model only the code found in the *first* case in any `switch` statement that you encounter.

