PA1415 Programvarudesign

Main Exam

Mikael Svahnberg (0455-385811), Ludwik Kuzniarz

2016-05-29

Points

(Filled in by the Marker)

Question:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	SUM
Max Points:	4	4	4	4	4	4	4	4	4	4	6	4	4	4	4	4	66
Points:																	

Grade:	

Instructions

- **Please Remember** to provide an answer in *each* checkbox [ ].
- **Marking** All multiple choice questions give four points. Each wrong answer in a question subtracts one point, but never below zero.
- **Diagrams** In some questions we ask you to draw a diagram (for example a class diagram or an interaction diagram). Please use a separate paper to make a draft first, and then redraw them in the marked area on the exam paper. Try to arrange the elements (and especially connecting lines) so that it is easy to read.
- **Allowed Books**: English to Swedish Dictionary.
- **Allowed Material:** Pen, Eraser, Ruler, Candy.

Good Luck!

Question 1. Requirements Engineering 4P

Please study these requirements, and then assess each of the claims below:

ID: R1
Title: Visual Feedback
Description: As a user I want to see a response from the system within 3 seconds so that I know it is working.

ID: R2
Title: Send Message
Description: The system shall immediately deliver messages between users of the system.

ID: R3
Title: Autologout
Description: The system shall automatically log out inactive users.

Claims: (Please mark *each* claim as true **T** or false **F**)

- [T] Requirement R1 is testable
- [T] Requirement R1 is measurable
- [T] Requirement R2 is testable
- [F] Requirement R2 is measurable
- [T] Requirement R3 is testable
- [F] Requirement R3 is measurable

Question 2. Development Methodologies 4P

Please mark *each* of the following statements as true **T** or false **F**:

- [F] An iteration in SCRUM usually takes two years.
- [T] If you are using a strict Waterfall process, you are not allowed to go back to previous development phases and add more information.
- [T] A *backlog* is a prioritised list of requirements.
- [F] The maintenance phase of a software system is usually short and can be ignored if you need more time.
- [T] The Unified Process (RUP) is an *iterative* and *incremental* development methodology.
- [T] In *Object-Oriented Analysis*, you focus on the domain concepts, and not how you are going to implement your system.

Question 3. Interaction Diagrams 4P

Please mark *each* of the following statements as true **T** or false **F**:

- [F] A collaboration diagram describes the interaction between classes.
- [T] The entity `dbs:Bicycle` is an object of the type `Bicycle` and the name `dbs`.
- [F] The entity `:Bicycle` is a class with the name `Bicycle`.
- [F] In a sequence diagram you list the attributes and their current values of each object under each object's lifeline.
- [T] An interaction diagram shows the messages sent between a set of objects.
- [F] You base your interaction diagrams on information you get from use case diagrams.

Question 4. Class Diagrams 4P

Please mark *each* of the following statements as true **T** or false **F**:

- [F] A class diagram describes the current values of the attributes in a class.
- [F] An association between two classes means that both classes have at least a pointer to the other class.
- [F] Classes must have attributes.
- [T] In `[Class A]`◆— `[Class B]` , the ◆— means that no instances of `B` may exist without belonging to an instance of `A`.
- [T] `«Interface» PrinterDriver` can not have any implemented methods.
- [T] The class `FileUtils::SimpleCache` belongs to the `FileUtils` package.

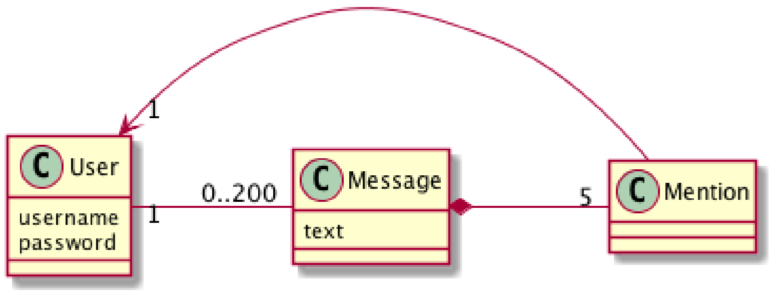
Question 5. State Machine Diagrams 4P

Please mark *each* of the following statements as true **T** or false **F**:

- [T] State Machine Diagrams contains *states*, *events*, and *transitions* between states.
- [T] States can be nested.
- [T] When you are in one particular state, one single event may trigger a transition to more than one new state.
- [T] When an event occurs, it is possible to change the value of an attribute.
- [T] The `[rooms left] makeBooking / notifyPorter()` transition only happens when there are rooms left and someone makes a booking.
- [F] For each state machine diagram you must make one sequence diagram to show how the involved objects calls methods on the other objects.

Question 6. Relations between Classes 4P

Please study the following diagram, and then assess each of the claims below:



Claims: (Please mark *each* claim as true **T** or false **F**)

- [F] A `User` may have any number of messages.
- [F] There can only be 200 `Messages` in this system.
- [F] Each `Message` may contain five `Mentions` or less.
- [T] One `User` may mention a maximum of 1000 `Users`.
- [F] Two `Messages` may have the same `Mention`.
- [T] Each `Mention` mentions precisely one `User`.

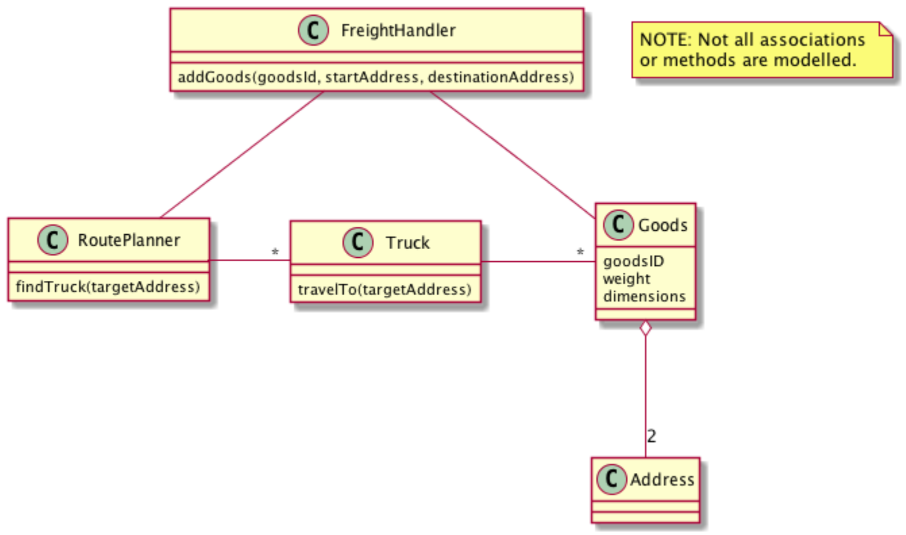
Question 7. Coupling and Cohesion 4P

Please mark *each* of the following statements as true **T** or false **F**:

- [F] It is bad to have many associations to other classes, so it is better to do everything inside a small set of classes.
- [F] The more responsibilities I give a class, the higher cohesion it will have.
- [F] No class should have more than three associations.
- [F] With fewer associations, it is easier to understand what each individual class does.
- [T] To maintain high cohesion, it is sometimes necessary to add more classes and associations into the design.
- [F] If I have high coupling in a system I will automatically also have high cohesion.

Question 8. GRASP Patterns 4P

Please study the following partial class diagram, and then assess each of the claims below:



Claims: (Please mark *each* claim as true **T** or false **F**)

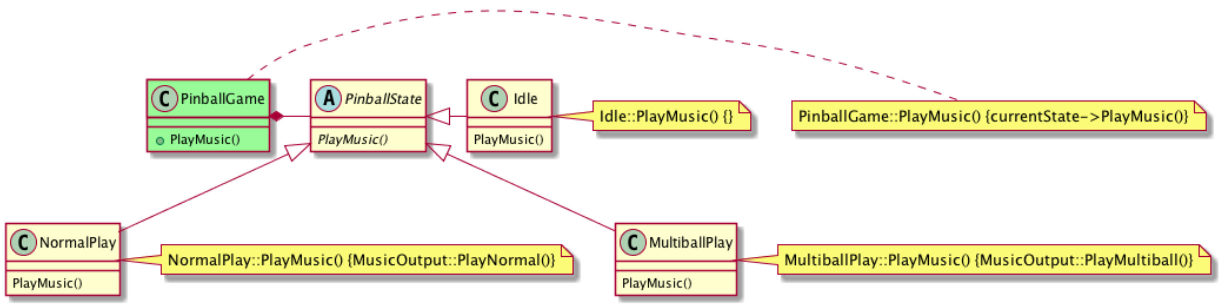
- [T] According to the *Creator* pattern, `FreightHandler` is best equipped to create new `Goods` items.
- [F] According to the *Information Expert* pattern, a `Truck` should contain a list of `Addresses` where it should stop.
- [T] The `FreightHandler` is a *Controller* for ordering transportation of goods.
- [F] According to the *Creator* pattern, `Goods` should be responsible for creating `Address` objects.
- [F] `FreightHandler` is an *Information Expert* on ensuring that each `Truck` does not get too heavy.
- [T] A `Truck` is an *Information Expert* on keeping track of the `Goods` it should contain.

Question 9. State Pattern 4P

A `PinballGame` can be in three states. In each of the states, a different music loop is played as follows:

- *Idle*: no music is played.
- *NormalPlay*: music is played by calling the static method `MusicOutput::PlayNormal()`.
- *Multiball*: music is played by calling the static method `MusicOutput::PlayMultiball()`.

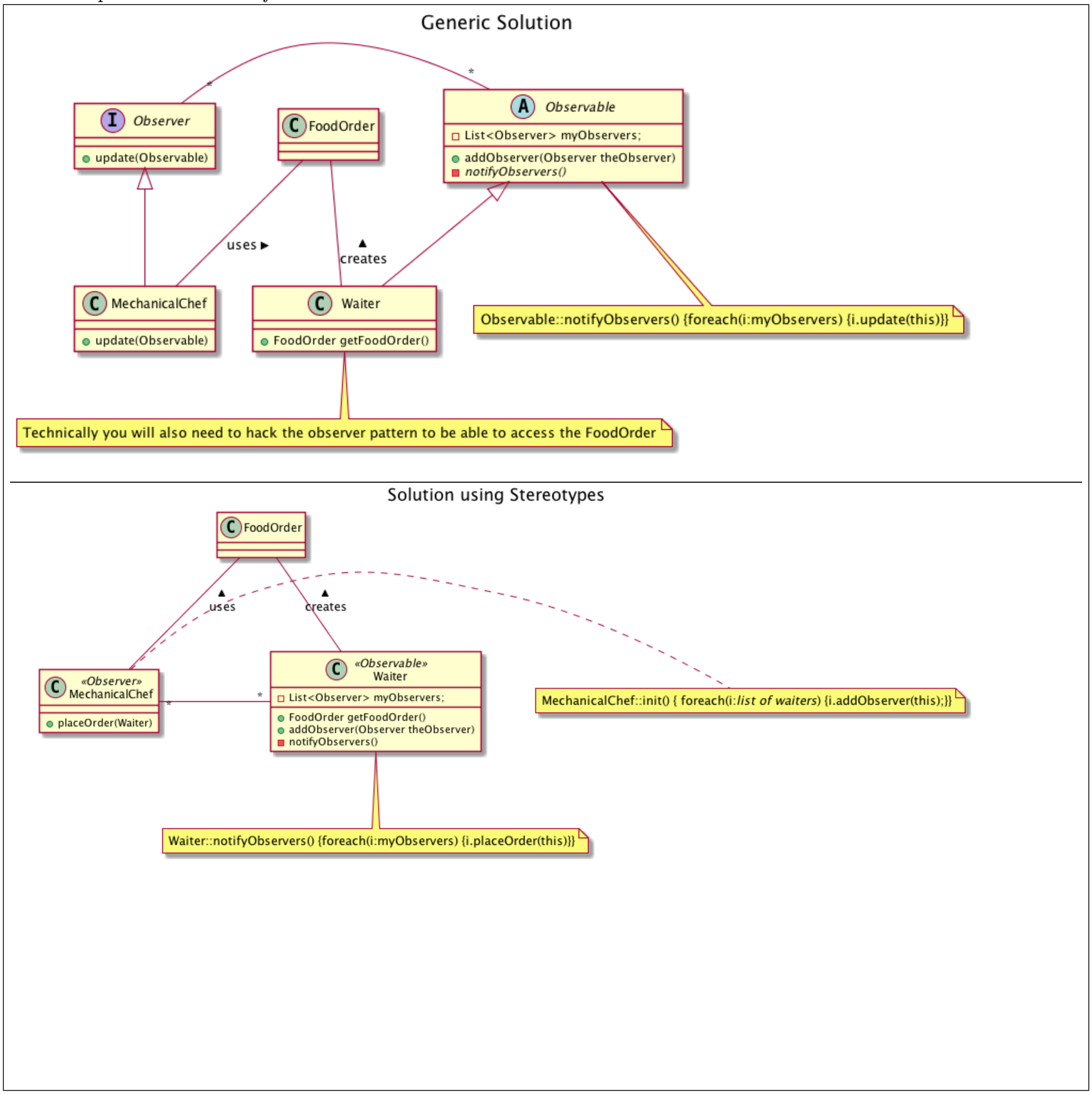
Please complete the following class diagram using the *State* pattern. Please also fill in the method bodies.



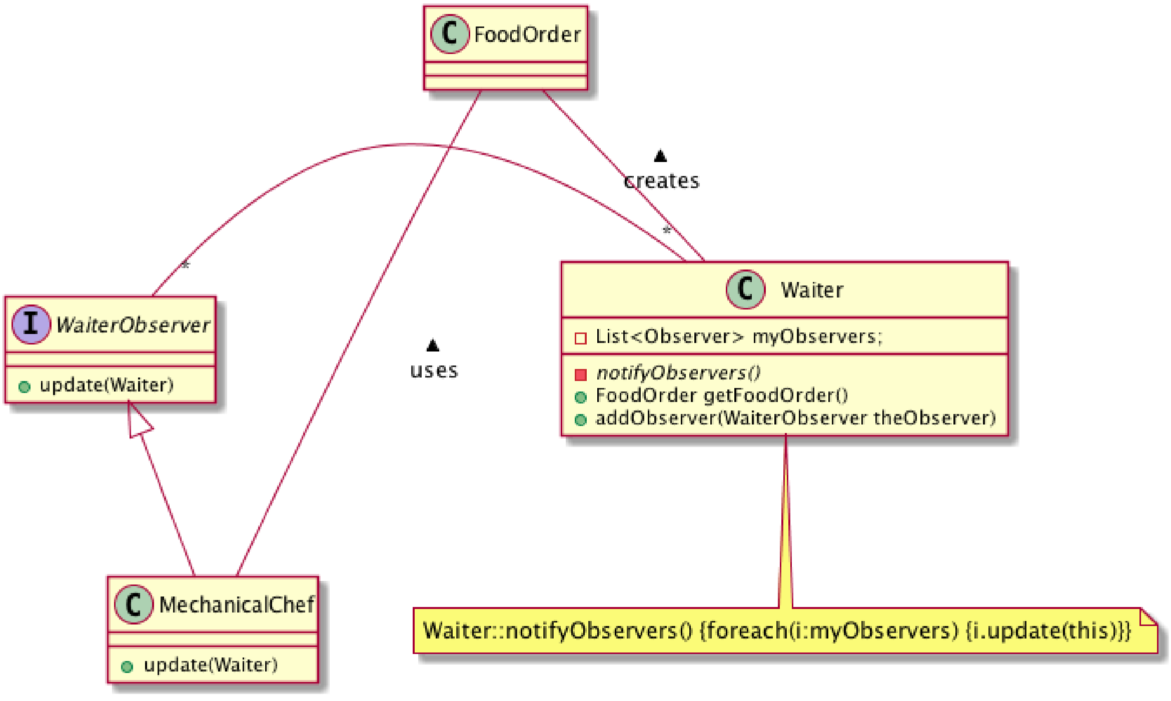
PinballGame
PlayMusic()
PinballState
PlayMusic()
Idle
PlayMusic()
Idle::PlayMusic() {}
PinballGame::PlayMusic() {currentState->PlayMusic()}
NormalPlay
PlayMusic()
NormalPlay::PlayMusic() {MusicOutput::PlayNormal()}
MultiballPlay
PlayMusic()
MultiballPlay::PlayMusic() {MusicOutput::PlayMultiball()}

Question 10. Observer Pattern 4P

A `MechanicalChef` is waiting for `FoodOrder` from one or several `Waiter`. Please model this in a class diagram using the *Observer* pattern. Use *interfaces* as well as concrete classes.


Generic Solution
I Observer
update(Observable)
C FoodOrder
A Observable
List<Observer> myObservers;
addObserver(Observer theObserver)
notifyObservers()
uses ▸
▴ creates
C MechanicalChef
update(Observable)
C Waiter
FoodOrder getFoodOrder()
Observable::notifyObservers() {foreach(i:myObservers) {i.update(this)}}
Technically you will also need to hack the observer pattern to be able to access the FoodOrder
Solution using Stereotypes
C FoodOrder
▴ uses
▴ creates
«Observer» MechanicalChef
placeOrder(Waiter)
«Observable» Waiter
List<Observer> myObservers;
FoodOrder getFoodOrder()
addObserver(Observer theObserver)
notifyObservers()
MechanicalChef::init() { foreach(i:list of waiters) {i.addObserver(this);}}
Waiter::notifyObservers() {foreach(i:myObservers) {i.placeOrder(this)}}

Partially Generic, more correct Solution


FoodOrder
creates
*
*
WaiterObserver
+ update(Waiter)
Waiter
- List<Observer> myObservers;
- notifyObservers()
+ FoodOrder getFoodOrder()
+ addObserver(WaiterObserver theObserver)
uses
MechanicalChef
+ update(Waiter)
Waiter::notifyObservers() {foreach(i:myObservers) {i.update(this)}}

Question 11. Conceptual Model **6P**

Please read the description below and then construct a *Conceptual Model* for this system. If and where applicable, use associations, aggregation, composition, inheritance, and association attributes. Please also consider multiplicity where stated and applicable.

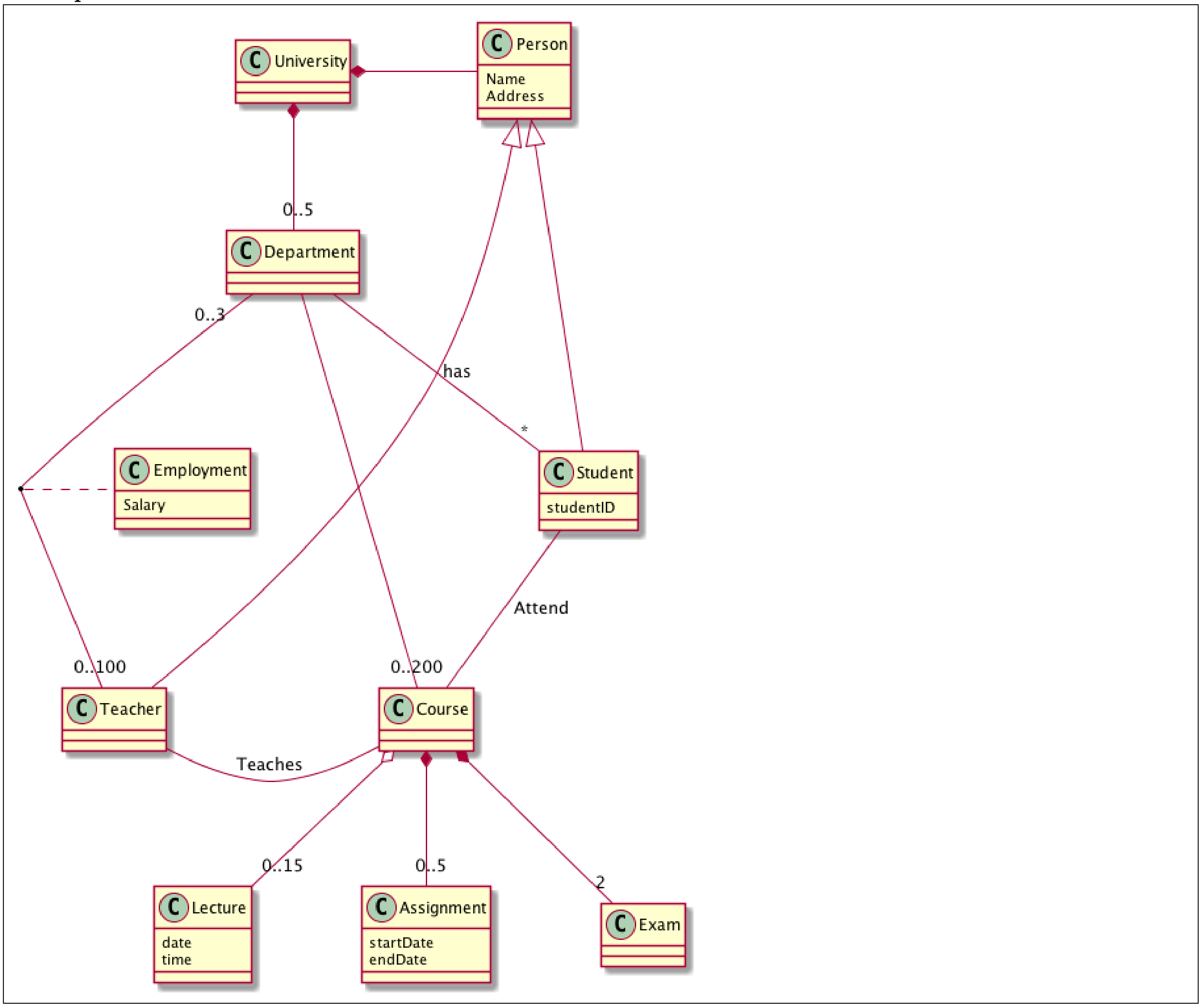
At a university you have students and teachers. Both students and teachers have a name and an address. Students also have a student-id. Students attend courses, and teachers teach courses.

The university consists of between zero and five departments, that each has up to 200 courses, up to 100 teachers, and any number of students.

The department pays a salary to teachers. Teachers can be employed by between one and three departments, and would then get a separate salary from each department. Each teacher negotiates their own unique salary.

A course consists of up to 15 lectures, one to five assignments, and two exams (main exam, and resit). Lectures and exams have a date and time, and assignments have a start date and an end date.

Conceptual Model:



Case Description

The remaining questions uses the following case. Each paragraph is numbered for easy reference in the questions. Please read this description carefully before answering the questions below.

Dash

(1) *Dash* is a tool to quickly and easily transfer money to friends and to pay for things using your mobile phone. It is developed in collaboration with a collection of swedish banks. In this description we focus on what you can do with the application once it is installed Dash on your phone and connected it with your bank account, and that you have *BankID* installed and configured. Your task is to design the features of the system that are described below.

(2) With Dash you can transfer money to other Dash users, receive money, and view a list of previous transactions.

(3) You transfer money by entering the *mobile phone number* of the recipient, the *amount* you want to transfer, and a *message*.

(4) In order to actually execute the transaction you identify yourself using your *BankID*. This starts a transfer of the specified amount from your account to the bank and account associated with the recipient's mobile phone number.

(5) At any point, you can abort a transaction.

(6) If there is not enough money in our account to complete the transaction, the transfer is aborted.

(7) If you want to see if you have received any money you first identify yourself yourself using your *BankID*. The Dash app then displays the last transaction, including the *sender* of the money, the *amount* received, and the *message*.

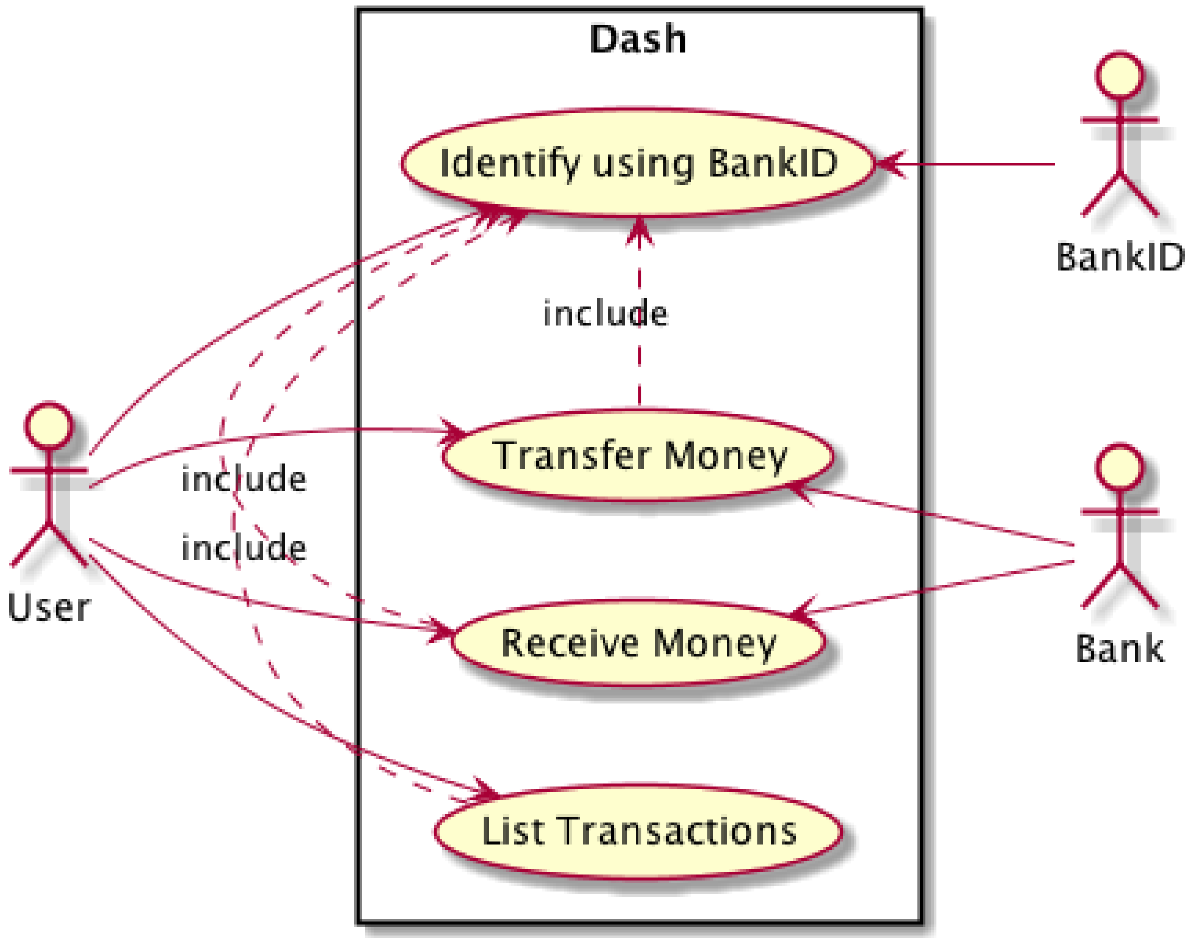
(8) You can get an overview of all transactions. You first have to identify yourself using your *BankID*, and then the Dash app displays an overview of previous transactions. This includes both *incoming* and *outgoing* transactions.

(9) To identify yourself using *BankID*, you enter a *pin code*. The system then connects to the BankID server to verify your identity.

(10) If you enter the wrong pin code, the current operation is aborted.

Question 12. Use Case Diagram **4P**

Please draw a *Use Case Diagram* for the *Dash* system. Make sure you identify all *Actors* as well as all *Use Cases*. If applicable, you may structure your use case diagram so that some use cases *include* or *extends* other use cases.



Question 13. Expanded Use Case 4P

Please write an *Expanded Use Case* for the `Send Payment` Use Case. This use case covers the case where a Dash user wants to send money to another Dash user (paragraphs 3 to 6 in the case description above).

Use Case: Send Payment

Primary Actor: User

Actors: Bank, BankID, receiving User

Preconditions: the User has registered with Dash.

Postcondition: Money is transferred.

Brief Description: A user initiates a transfer, enters the mobile phone number of the recipient, the amount to transfer, and a message.

Continued on the next page

Basic Flow:

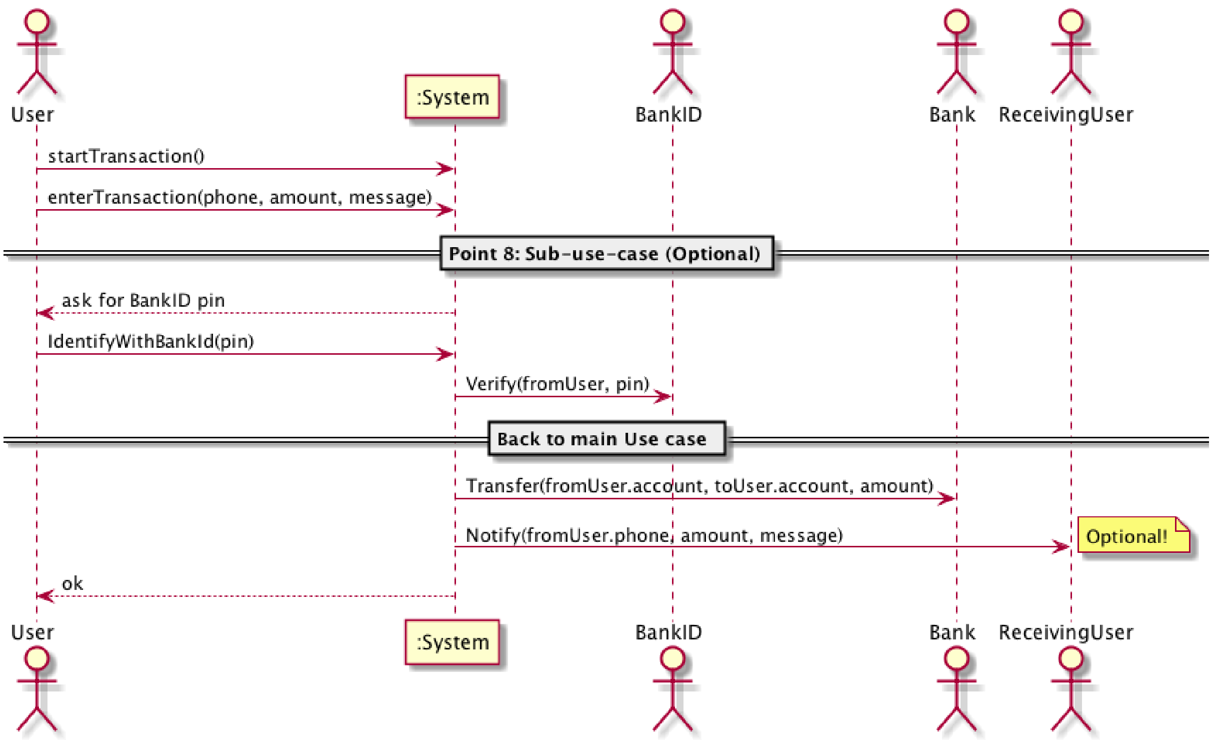
Actor	System
1. A User initiates a transaction	2. The System indicates that it is ready to receive information.
3. The User enters the phone number of the recipient	
4. The User enters the amount	
5. The User enters a message for the recipient	
6. The User tells the system to start the transaction	7. The system verifies that the given number, amount and message are ok.
	8. The system initiates the Identify Using BankID use case
	9. The system contacts the bank to transfer the money.
	10. The system notifies the receiving User about the transaction. **NOTE: Optional!**

Alternative Flows:

- 7. Incorrect phone number is entered. The transaction is aborted and an error message is shown.
- 8. The user fails to identify themself using BankID. The transaction is aborted and an error message is shown.
- 1–8. The user decides to abort the transaction.
- 9. There is not enough money to complete the transaction. The transaction is aborted and an error message is shown.

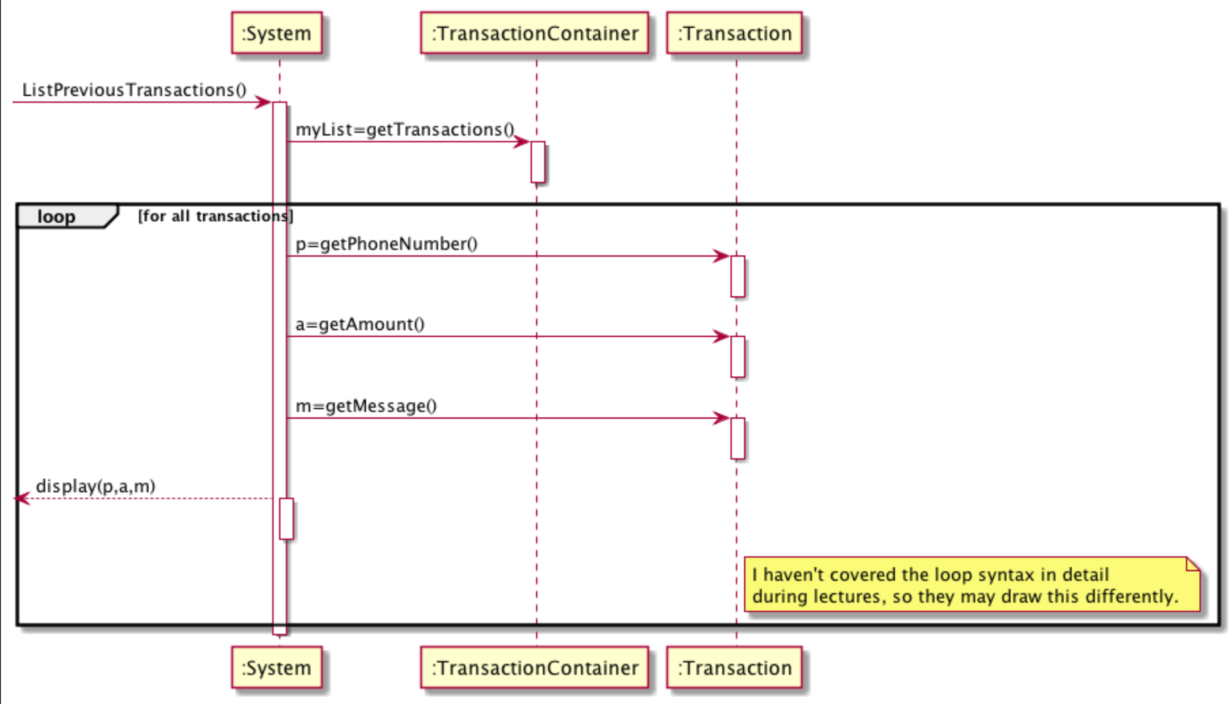
Question 14. System Sequcence Diagram 4P

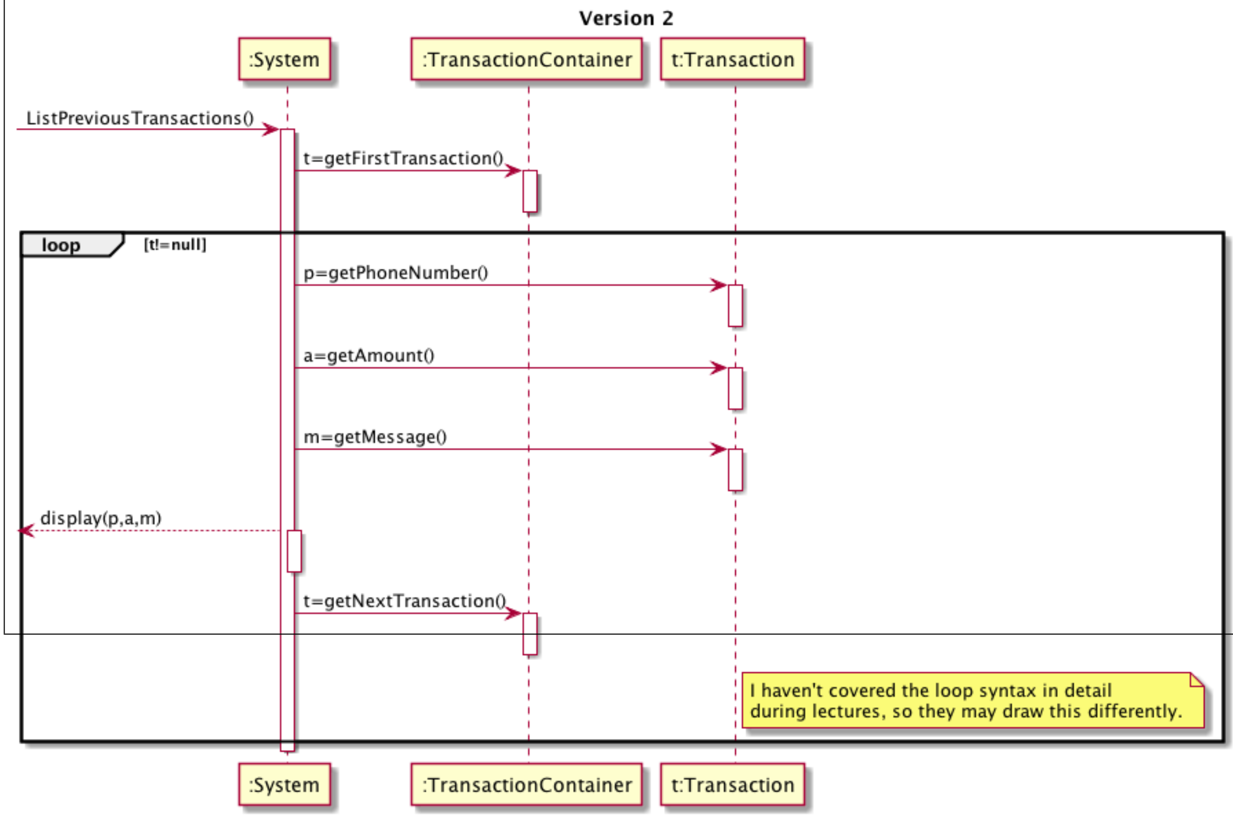
Please draw a *System Sequence Diagram* for the `Send Payment` use case. Make sure that all actors involved in the use case are represented, including those that the system may need to contact in the course of the use case.



Question 15. Interaction Diagram 4P

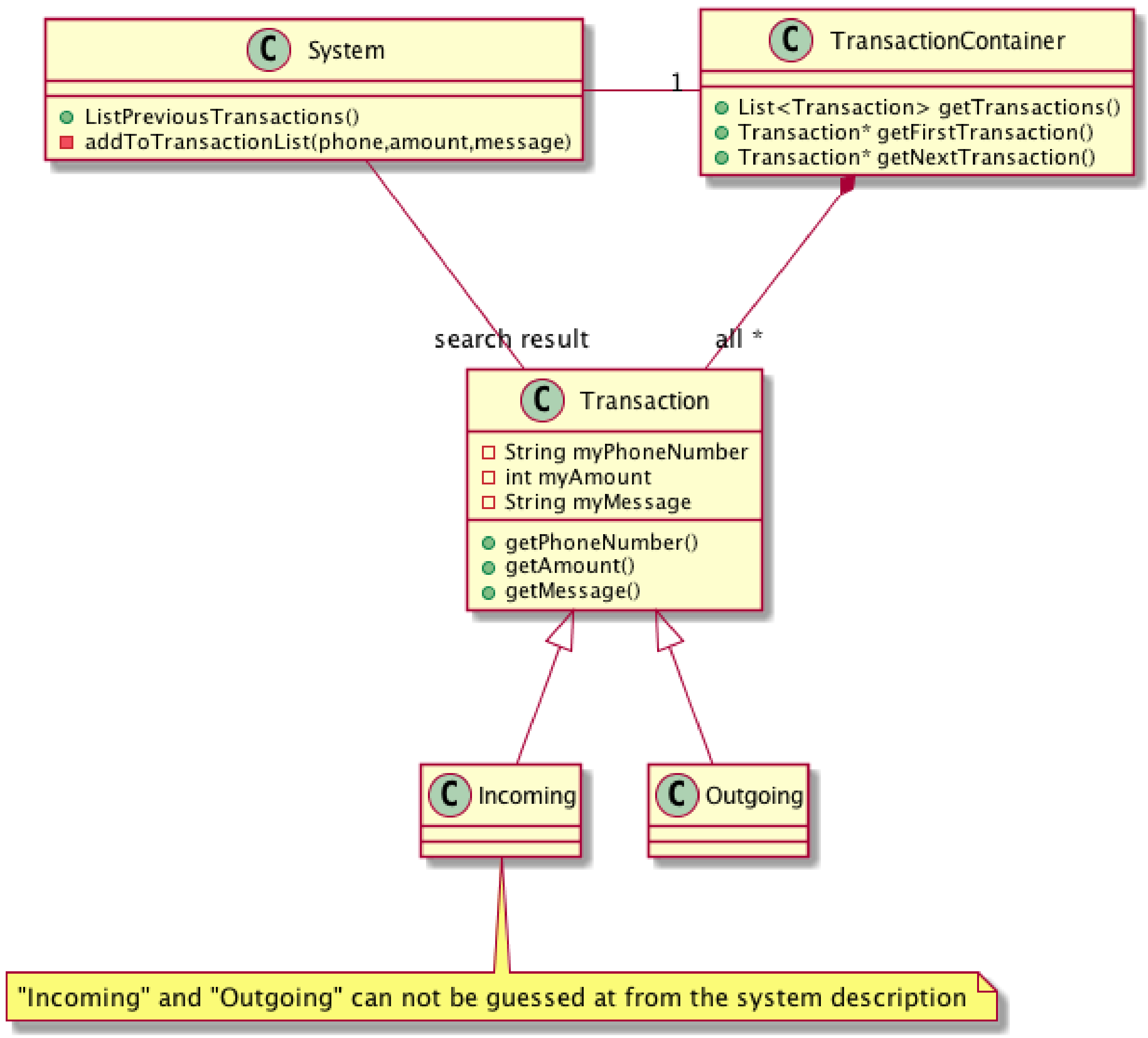
Please draw an *Interaction Diagram* (Sequence or Communication Diagram) for the `ListPreviousTransactions()` system operation.





Question 16. Class Diagram 4P

Please draw a partial class diagram that includes the classes, methods, and attributes that you used in the interaction diagram in the previous question. Remember to also consider the associations between classes, and add (if and where applicable) base classes and inheritance hierarchies.



System
ListPreviousTransactions()
addToTransactionList(phone,amount,message)
1
TransactionContainer
List<Transaction> getTransactions()
Transaction* getFirstTransaction()
Transaction* getNextTransaction()
search result
all *
Transaction
String myPhoneNumber
int myAmount
String myMessage
getPhoneNumber()
getAmount()
getMessage()
Incoming
Outgoing
"Incoming" and "Outgoing" can not be guessed at from the system description