Inf2-SEPP 2023-24

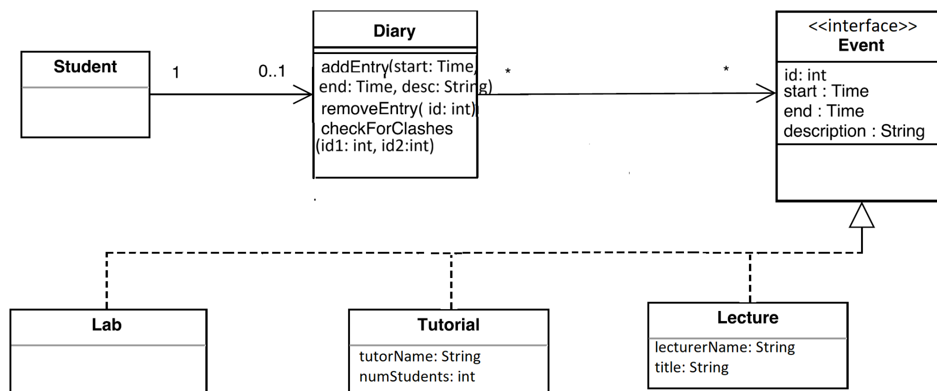
Tutorial 3 (Week 5)

Notes on Answers

1 Creating class diagrams

1.1 For a diary system

Answers expected similar to the ones below:



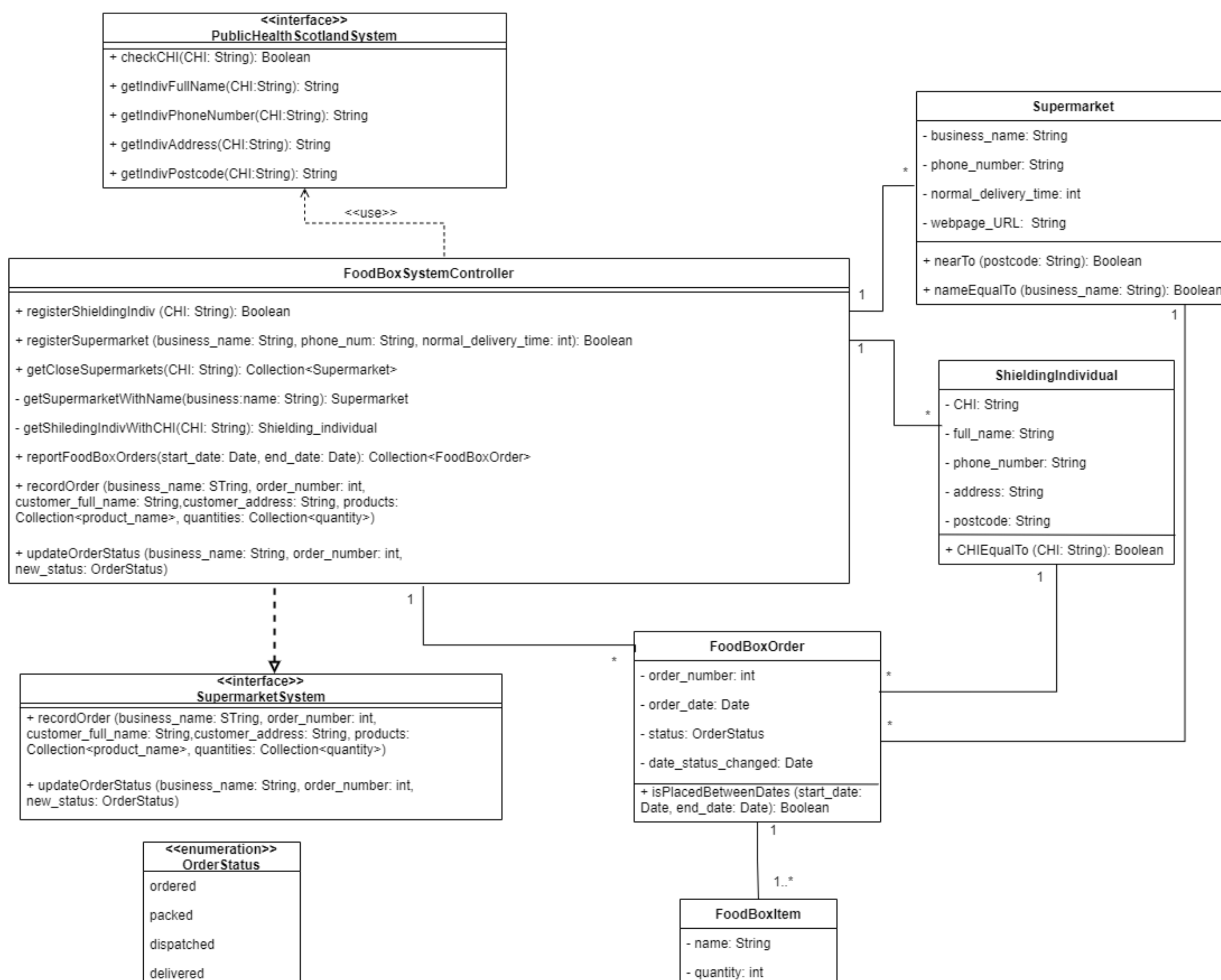
It would have also been acceptable for the Event to be a class and Lab, Tutorial, Lecture its subclasses, however the provided solution with Event as an interface makes sure that Event cannot be instantiated. This makes sense in the current context: only specific events would be added to the diary. Also, we could have used a use dependency arrow from Diary to Event, but the association makes things more specific because it can include multiplicities: a Diary object has associated a collection of Event objects (because diaries contain events), and an Event object has associated a collection of Diary objects (because the same event can be added to multiple diaries). In these multiplicities, we assumed that a diary may be created as empty or become empty if all of its events are removed (multiplicity of * equivalent to $0 \ldots *$ next to Event class), and that an event may keep existing even if it was removed from a diary (multiplicity of * equivalent to $0 \ldots *$ next to Diary class). We also assumed that a diary belongs to one student (multiplicity of 1 next to Student), which makes sense as diaries are usually personal. The fact that diaries maintain ordered lists of events is an implementation detail, and so could be ommitted from this high level class diagram.

Important advice for the following tasks

Try to structure your design for the *problem* rather than a particular *solution*. In these exercises, you are not attempting to design the implementation classes, but describe the *design* of the implementation. Implementation will have more classes and more detailed interfaces than a class diagram. In short, the class diagram is an intermediate point between the unavoidably ambiguous natural language of the requirements specifications and the use-cases, and the unambiguous implementation source code. Here you are making the specification of the solution *more* concrete, without as much commitment as in a full implementation. The goal is to clean up mistakes in the requirements before implementation is begun. Because of this, class diagrams should still be understandable to someone who is not a software developer.

1.2 For a food box system

A possible solution to the class diagram is provided in the following figure:



For defintions and explanations about design principles, check Lecture 9 part 1 as well as its resources. Googling also helps!

The following are notes on this design:

- We split the system into classes (corresponding to conceptual entities), each with distinct responsibilities and well-defined interfaces given by their public operations (decomposition, modularisation)
- We need a controller/manager class to coordinate the other classes but otherwise the classes do actions that concern them themselves (high cohesion)
- We must use an interface for Public Health Scotland's system which would be implemented by this system, because we want their system to carry out actions like checking the CHI number and retrieving the individual's details for us. We don't show the class implementing the interface, because this would be part of Public Health Scotland's system.
- We must also use an interface for the supermarket system, but this time the interface would be used for it to notify our system of new orders and changes to order statuses. Therefore, our system must implement the interface this time. Please observe the notation with the dashed arrow with empty arrowhead pointing towards interface.
- The two points above help respect the design principle of separation of interface and implementation
- We assumed that the supermarket also sends information on the contents of the order when notifying of the order being placed, so that we could record them for more accurate future reporting.
- Some of the associations are not clear from the system description, but were included because we envisage the potential need to extract more types of reports (e.g. association ShieldingIndividual- FoodBoxOrder, Supermarker-FoodBoxOrder)
- A representation of an enumeration is shown in OrderStatus
- It is good practice to have attributes as private and access/change them only through public getter and setter operations. Here, these were not represented as required. Other 'get' methods represented do more than just retrieving attribute values.
- Only operations that can be accessible from other classes must be public.
- The above 2 points respect the design principle of abstraction and encapsulation.
- Deciding on multiplicities was sometimes a matter of thinking whether an object of a class would always have access to objects of another. If this is not the case as they may not exist yet, but they could also be more than 1, we used * (same as 0...*).
- **Important!** When associations are to be impemented as attributes, these attributes must not also be represented on the diagram as it would be redundant. The only reasons to show attributes on the class diagram are if they are of a primitive type (e.g.

int, boolean, etc.), of a built-in Java class type (e.g. String, LinkedList, etc.), or of a library class type if a library is used. To avoid clutter, associations with enumerations may not be represented.

1.3 (Advanced) For a lift system

See Figure 1 on page 5 for an example solution.

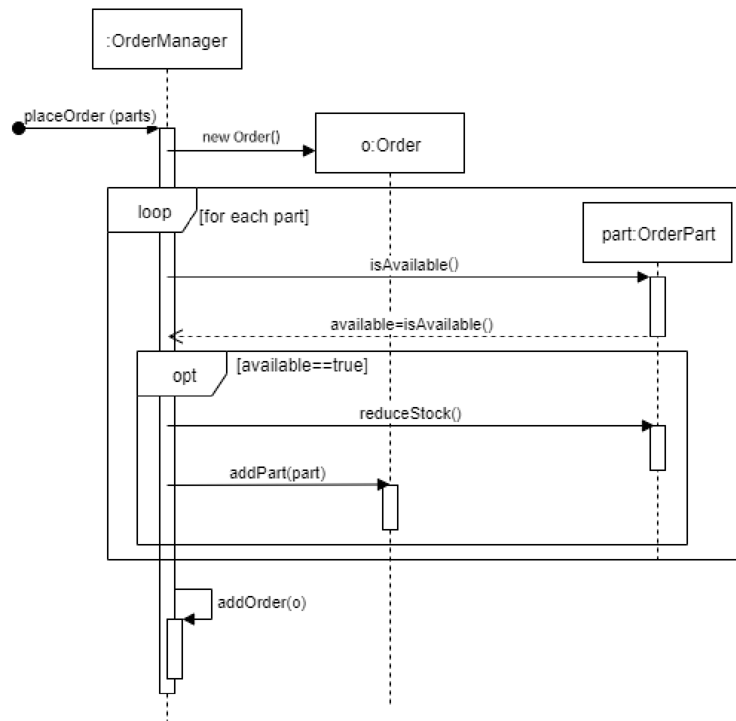
Here we imagine there are N floors. For simplicity there is only one PositionDisplay class for both the displays above the doors and inside the lift. One could imagine having more specialised classes for each, as with the buttons.

Objects in the LiftPosition class record which floor the lift is closest to, along with whether it very near or at a floor. The diagram assumes that the software system anticipates the lift arriving at some destination floor and sends the lift motor a stop message at just the right moment for the lift to slow down and stop exactly at the floor. This is perhaps not very plausible. A more realistic setup would be to have a separate LiftMotionControl class integrating both the PositionSensor and LiftMotor classes which accepts messages like moveToFloor(floor : int) and has an operation reachedDestinationFloor() to model input events signalling move completion.

2 Creating sequence diagrams

2.1 Warm-up sequence diagram

The sequence diagram should closely resemble (please note that the notation used for the first message is that of a found message, as we don't know where it comes from):



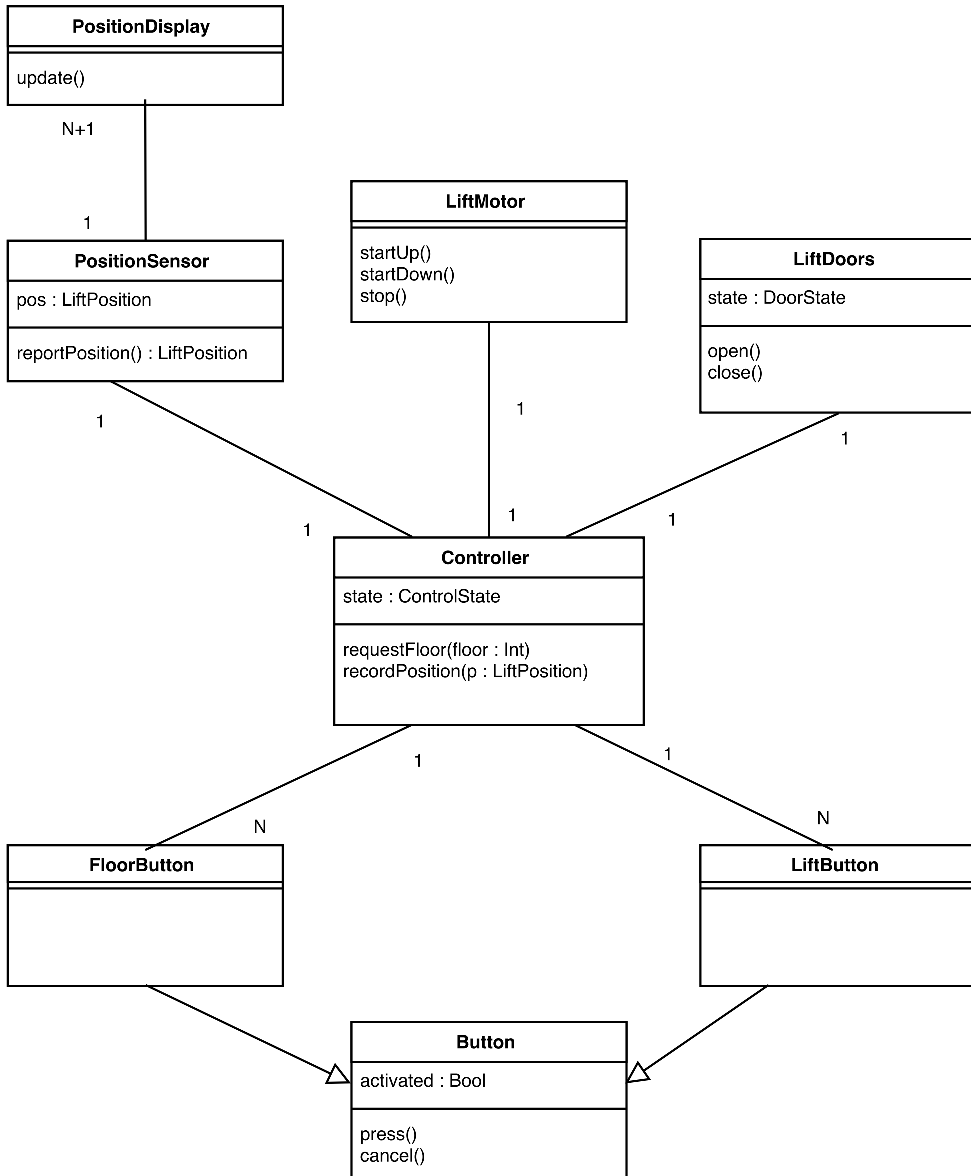


Figure 1: Lift Class Diagram

Issues to look for in the diagram include:

- Basic diagram syntax captured (object rectangles with dashed lifelines descending below. Horizontal solid arrow messages. Dashed arrow responses for when operations return something (i.e. they are not of type void).)
- Correct ordering of messages and messages going between correct objects.
- Use of solid arrow pointing towards object for the Order constructor.
- Use of UML loop frame with appropriate condition for the for region.
- Use of UML opt frame with appropriate condition for if region.
- Local variable value available set on method return for isAvailable() method.
- Activation bar for addOrder() call half overlaps the activation bar for placeOrder() invocation.

2.2 Advanced sequence diagram

See Figure 2 on page 7 for an example sequence diagram solution for the 'Call lift' use case.

This has examples of both approaches noted on the question sheet to handling real time: we assume the open() and close() operations only return on doors fully opening or fully closing respectively. On the other hand, we have the lift startUp() operation returning immediately, and on reportPosition() operations on the PositionSensor to signal how the lift is moving and when it reaches the destination floor.

Exactly as written, the sequence is not realistic: we only exit the loop when the lift reaches the destination floor, and assume then the lift can be stopped immediately.

In the sequene diagram, reportPosition() is an example of a found message, a message whose source we do not make explicit. Found messages are useful because they can avoid the clutter of an extra lifeline for the message source. In this case the actor generating this message is the lift itself. We imagine the lift generating such a message whenever it moves past some physical position sensing device.

We imagine the reportPosition() operation of the PositionSensor class being invoked whenever a reportPosition() message is sent to the PositionSensor object from the lift actor. The pos attribute is for recording the last observed position of the lift according to the reportPosition() messages coming in. Not shown in the class or sequence diagrams is how this attribute might get used. A more detailed version of the Figure 2 sequence diagram would show the controller object querying this attribute in order to determine whether to send a startUp() or a startDown() message to the LiftMotor object.

In this sequence diagram, we represented all return messages, even if nothing is returned (i.e. the operation in the class of the reached object is of type void). This makes it easier to ensure the sequencing of message calls in such a non-parallel scenario (i.e. that the next message doesn't start until the previous one received a return). However, we can omit return messages when nothing is returned as long as we take care of where we start the next message on the diagram.

Cristina Adriana Alexandru. 7th Feb 2024.

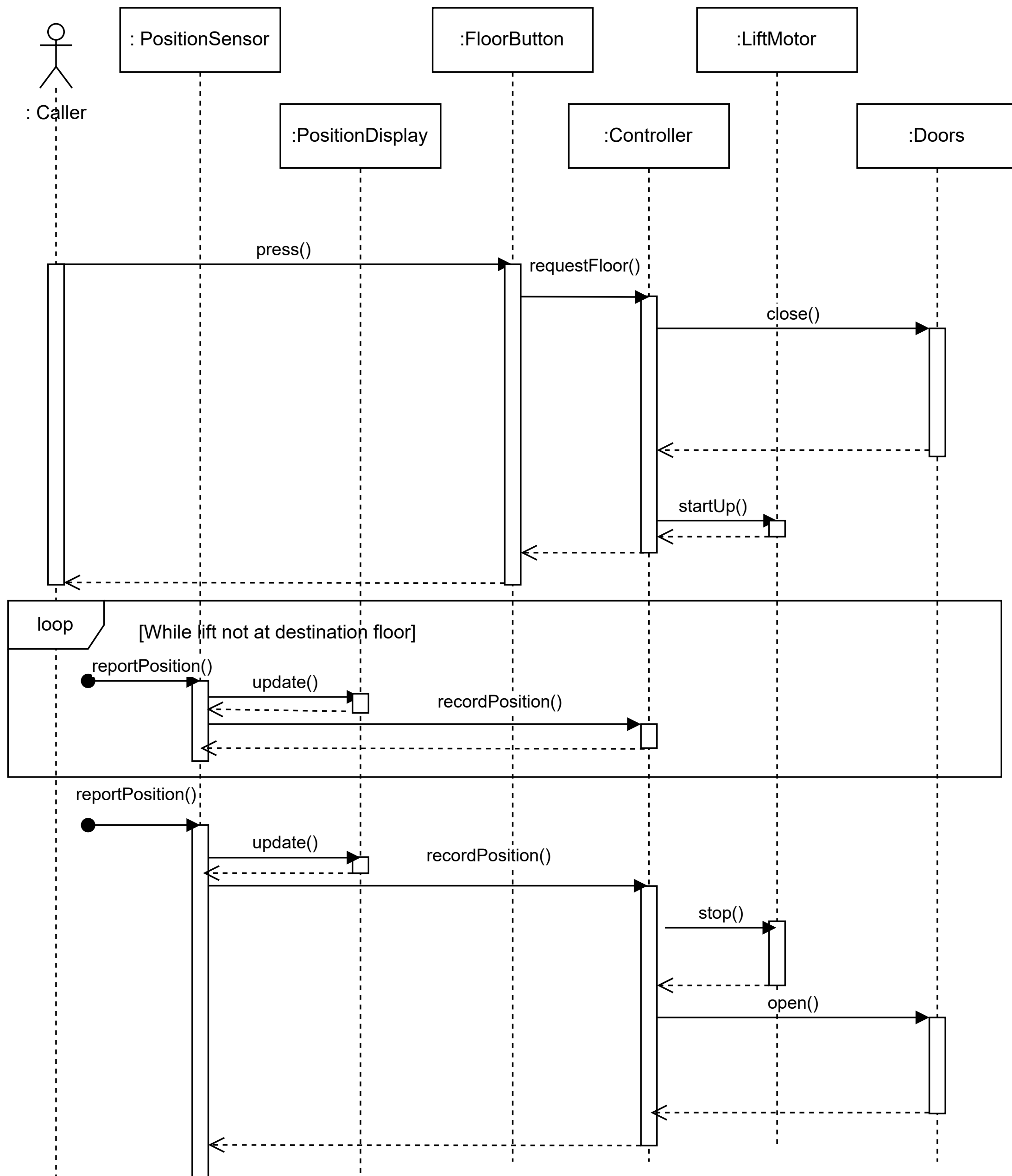


Figure 2: Lift Sequence Diagram