Inf2C Software Engineering 2017-18

Tutorial 2 (Week 6)

UML Class and Sequence Diagrams

Study this tutorial sheet and make notes of your answers BEFORE the tutorial.

1 Introduction

In this tutorial you will practice creating UML Class and Sequence diagrams.

2 Creating Class Diagrams

Create a Class diagram for the Lift system described in the Tutorial 1 question sheet. Add sufficient operations to your classes that you can trace through the message sequences involved in realising the Lift use-cases. Annotate associations between classes with multiplicities. Can you identify any opportunities for using inheritance?

If you have time, also have a go at producing a Class diagram for the Shopping List App also described there.

Designing, not implementing

Try to structure your design for the *problem* rather than a particular *solution*. In this exercise, you are not attempting to design the implementation classes, but describe the *design* of the implementation. In particular if all we wanted was a list of the classes in the implementation, then why not simply write down that description in the implementation language? We could even then write software to extract a UML Class diagram from the source code.

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, the Class diagram should still be understandable to someone who is not a software developer.

Handling interfaces

When designing the Shopping List App Class diagram, ignore completely the details of the interactions with the touch-screen interface of the smart-phone or tablet running the app. Think in terms of the user directly having handles on some of the system objects, and then being able to send messages to these objects and receive return responses back.

The Lift system has a number of physical interfaces to the outside world: to the various buttons, displays showing the lift position, the lift motor for raising and lowering the lift, for example. The Class diagram for the Lift system would be rather impoverished if these interfaces were not explicitly recognised somehow in the diagram. In the *library* example discussed in lecture, some classes like Copy or LibraryMember are *avatars* or *proxies* for real world entities, holding just that information the system needs about each entity. Here, we can do something similar, introducing classes for each electrical/physical interface. We could introduce CallButton and LiftMotor classes, for example. The difference is that we then add specific operations to these classes for modeling interface events: for a class modeling an input interface, we add an operation whose calls model input events. A CallButton class might have a press() operation that we imagine invoked on an CallButton object representing a particular button, whenever the lift user presses that button. For a class modeling an output interface, we add an operation whose calls model output events being generated. A LiftMotor class might have a startDown() operation whose calls on a LiftMotor object model the system issuing commands to the lift motor to start the lift moving down.

3 Creating Sequence Diagrams

Draw a UML Sequence diagram for one of the main Lift system use cases identified in the Tutorial 1 Answer Notes.

Handling real time

One issue to consider here is the real-time nature of the system. For example, a LiftDoors class might have an close() operation for closing the doors. Does the close() operation return immediately, even though it may take a couple of seconds for the doors to close? It would be very bad if the close() operation did return immediately and then the system immediately after this invoked an operation to start the lift moving: we could have the lift moving while the doors are still partially open. Two alternatives to think about are

1. Have the close() operation wait for the doors to fully close before returning

2. have the close() operation return immediately, but then have the doors generate a doorsFullyClosed() event when the doors are indeed fully closed. The Doors class would then include a doorsFullyClosed() for modeling such events.

You might try one approach for the doors, another for the lift motion control system.

With the second alternative, the doors or the lift motion control system effectively become actors, and you could describe them as such in your diagram. Alternatively, you could use the Sequence diagram message notation for the events generated by these actors where the message arrow head points to the object receiving the message and the arrow tail is a filled-in circle. This avoids cluttering the diagram with e.g. lifelines for both a Doors actor and a Doors object. Such messages are called *found messages* in UML.

4 Advanced - Negative Commenting

This section is mostly to promote discussion if there is time during the tutorial. One problem with Class diagrams and modelling design in general is the possibility to be overly complex and hence too prescriptive.

Now that you are finished with your Class diagram try to comment on each class describing what would happen if you were to remove this from the Class diagram. This is known as *"Negative Commenting"*. Hopefully for most of your classes the effect of removing the class from the Class diagram should be quite drastic.

Finally then, consider whether negative commenting is useful? Is it useful to the clients, the developers, both or neither?

Paul Jackson. 18th Oct 2017.