

DT228-3 Software Engineering III - Lab 2 (Week 3)

Rational Software Architect – Sequence Diagrams & Code Generation

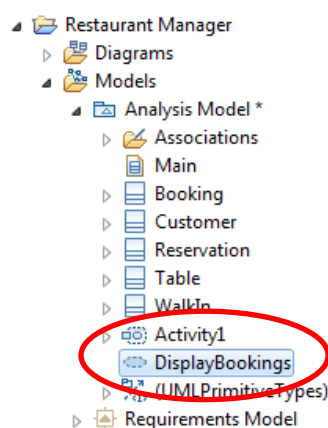
PART I – RSA Sequence Diagrams

In part I of this lab session we will develop some sequence diagrams for the Restaurant Booking Management system we have been working on to illustrate the features of RSA. We will see how working through the sequence diagrams helps us to refine our object model and build our class diagram. Our sequence diagrams will show how the classes and associations we have defined will collaborate in order to implement specific use cases.

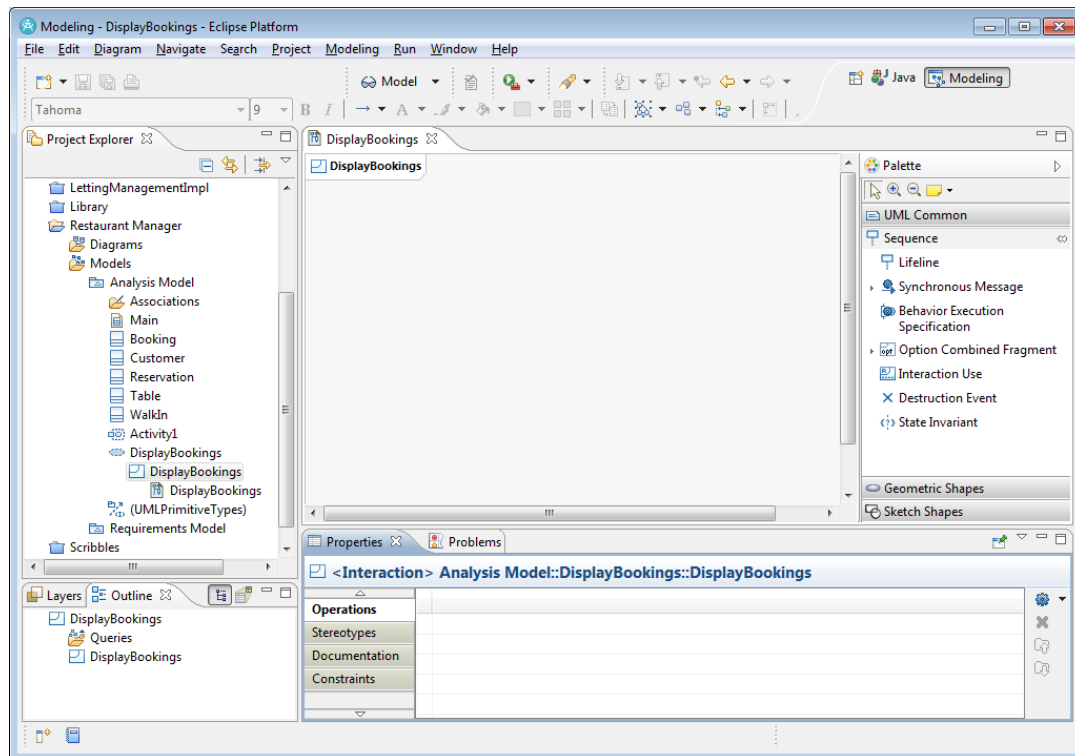
Task 1 – Add a *Collaboration* and *Sequence Diagram* to our Analysis Model

Firstly in RSA, we will create a *Collaboration*. The collaboration will identify what use case we are going to realise.

- Right-click on the Analysis Model in your restaurant system project and choose *Add UML -> Collaboration* – name the collaboration *DisplayBookings*. You should now see a Collaboration in your model like below:



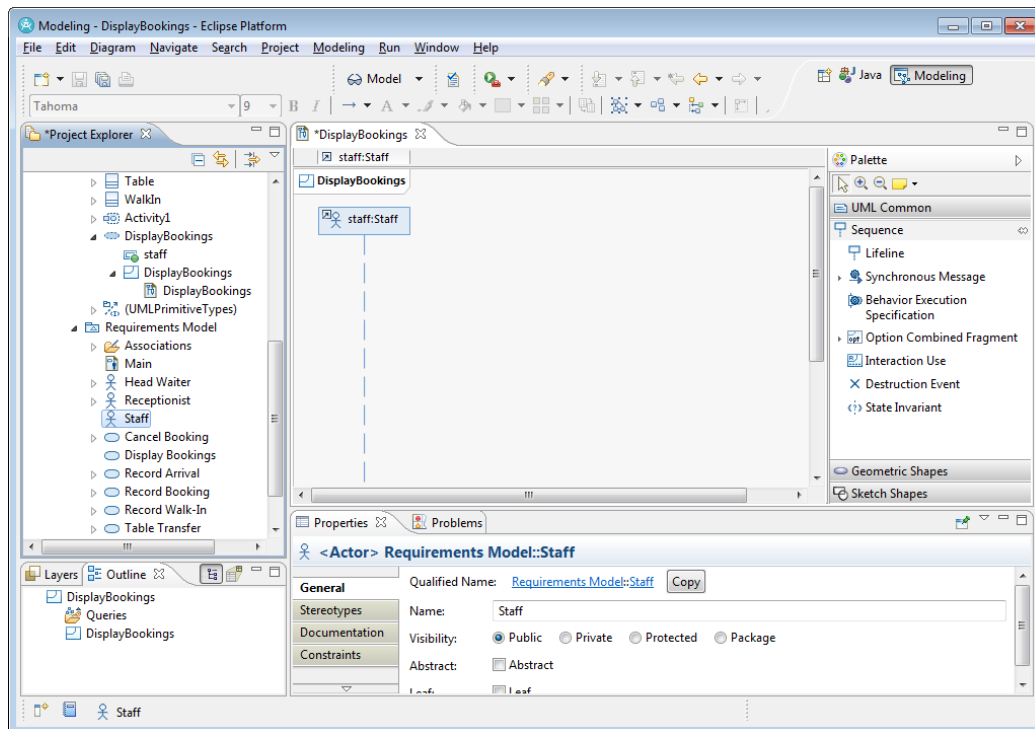
- Next, right-click on the *DisplayBookings* collaboration and choose *Add Diagram -> Sequence Diagram* – name the sequence diagram *DisplayBookings*. This creates an *Interaction* in the Project Explorer under the collaboration – you can right-click on this and choose *Edit->Rename* to change its name to *DisplayBookings*. You should now have something like the screenshot below with an empty sequence diagram open in the editor.



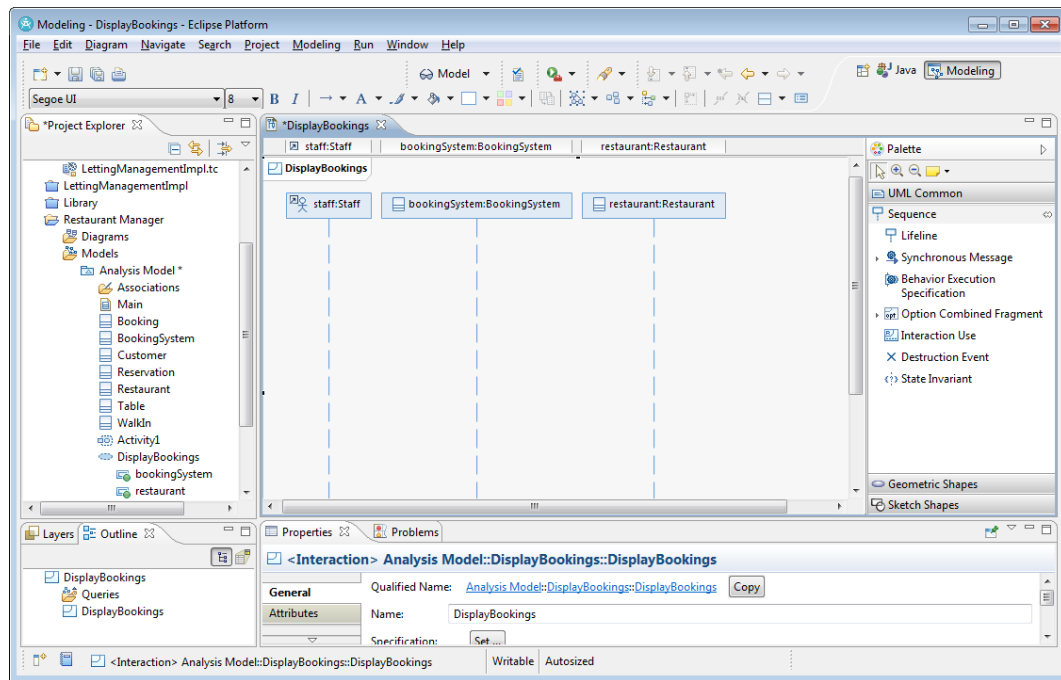
Task 2 – Add some instance *Lifelines* to our Sequence Diagram

In this task, we model the interactions between the objects that are required to realise the *Display Bookings* use case. Note, as we do this we will be creating additional classes and defining messages that go between the objects. RSA automatically updates our object model (classes) as we do this.

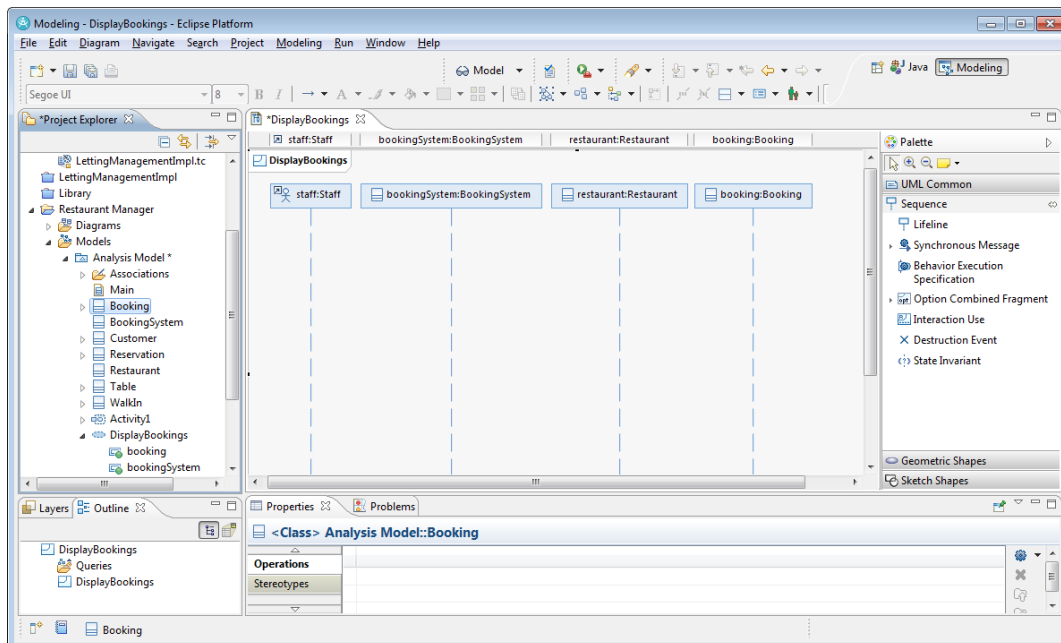
- First we need to add an instance of an *Actor* that is interacting with the system in this particular use case. Drag your *Staff* actor from your requirements use case into the sequence diagram that is open in the editor – you should have something like this:



- At this point we can decide to add a *Controller* object that will act as the initial interaction point between the actor and the system. To do this, drag a *Lifeline* from the palette on the right in to the diagram (place it to the right of the actor) – when you release the mouse you will be given some options, choose *Create Class* and name it *BookingSystem*. It then draws the lifeline and gives the instance a default name.
- **Note:** This does two things: it creates a new class in our analysis model called **BookingSystem** which we can see in the Project Explorer; secondly it adds an instance (object) of that type to our sequence diagram. In this way, we have built on our initial model through the development of our sequence diagram.
- Next, we can decide to introduce another class called *Restaurant* which will hold references to all *Bookings*. As before, drag a *Lifeline* from the palette to the diagram and call the new class *Restaurant*. You should have something like:



- Next we will add an instance of our *Booking* class to the diagram (instances of these will be required so that we can display their information). As the *Booking* class already exists in our model, we can drag it from our model (in the Project Explorer) into the diagram and drop it to the right of *Restaurant* instance. This creates the *Booking* instance(s) we need in our diagram. You should finally have something like:

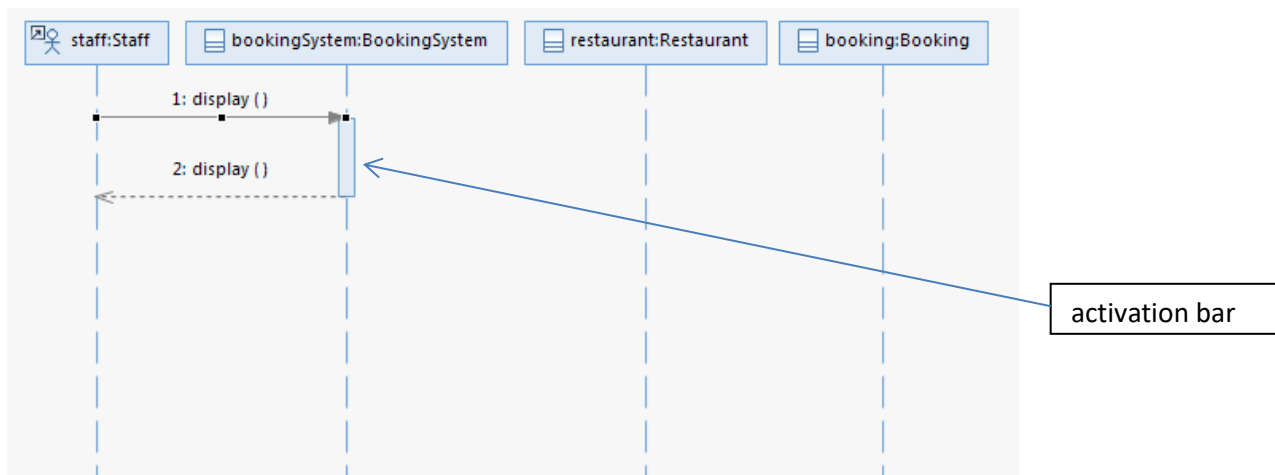


Task 3 – Adding messages from one instance to the other

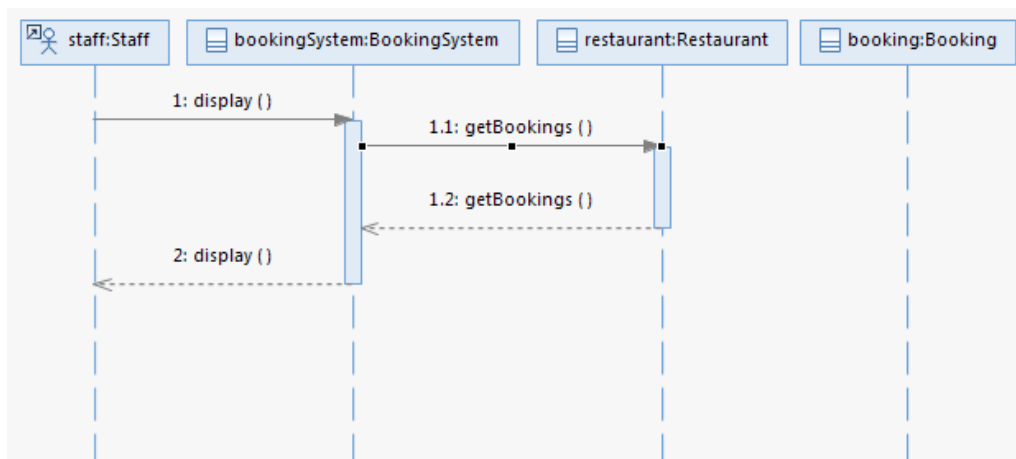
So far we have drawn lifelines that represent instances (objects at runtime) that will be needed to somehow perform the *Display Bookings* use case. We now need to show how those instances will communicate with each other in order to do that (from an implementation perspective these

messages show what methods each object will invoke on the other at runtime and in what sequence).

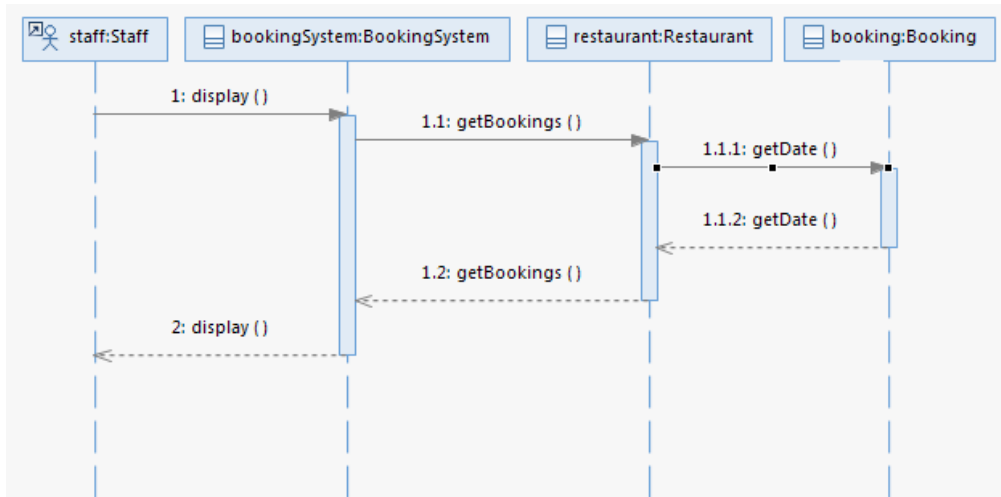
- Click on the *Synchronous Message* element in the palette on the right (release the mouse button). Place the mouse pointer at the top of the *staff* actor lifeline (over the dashed line) and drag it across to the dashed line of the *BookingSystem*'s lifeline – when you release the mouse button you will be asked to give the operation (message) a name, type in *display*. You should now have something like this:



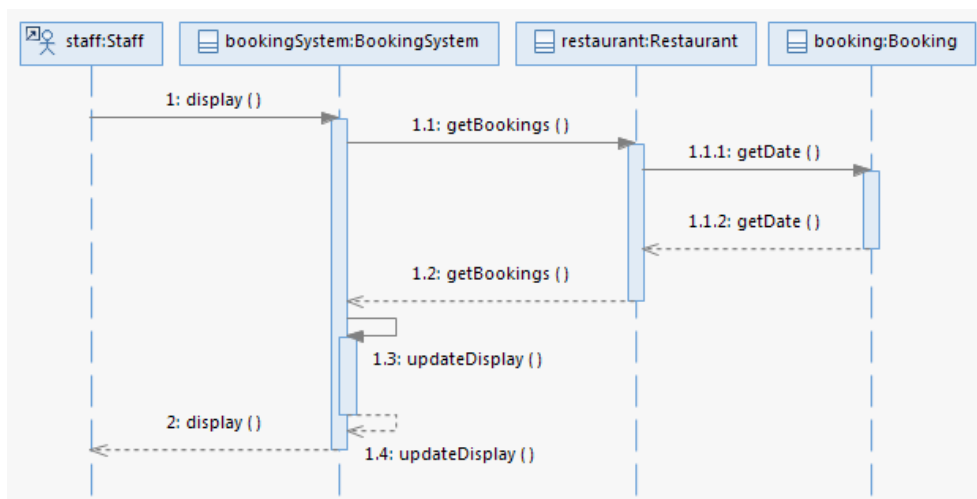
- Starting from the message activation bar on the *bookingSystem* lifeline, add another message between *bookingSystem* and *restaurant* called *getBookings*. You should have:



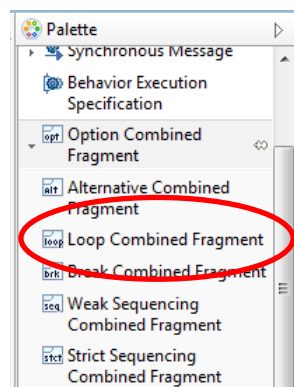
- Similarly add a message from *restaurant* to *booking* to get:



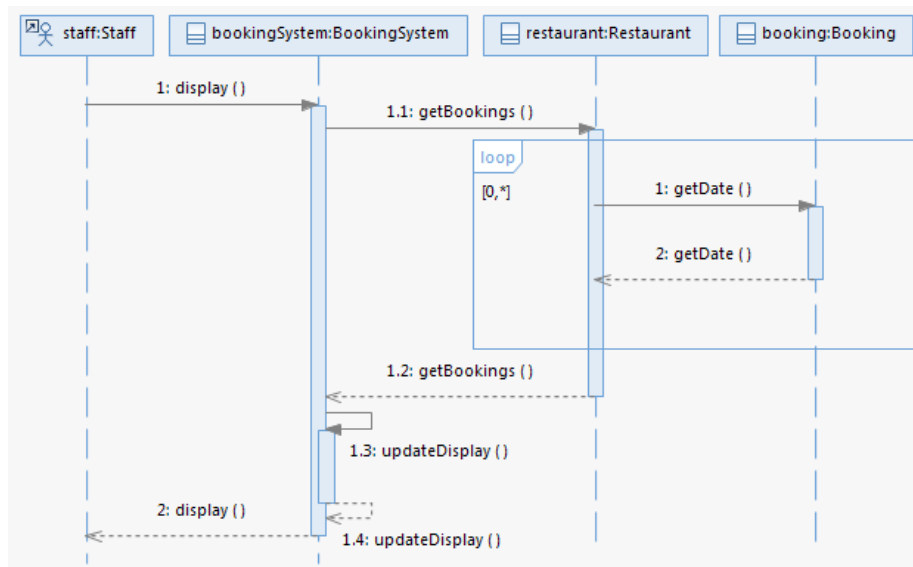
- Next we add an internal (also called reflexive or self) message to the *bookingSystem* lifeline. We do this by dragging a *Synchronous Message* from the *bookingSystem* activation bar back to itself. When you do this you will be given some options: *create a new operation* or *pick an existing one*. Here we want to create a new one called *updateDisplay*. Do this and you should have:



- Next we will add a *Loop* frame around the *getDate()* message to denote that this message is passed for each *booking* object that it has a reference for. To do this, click on the *Loop Combined Fragment* in the palette

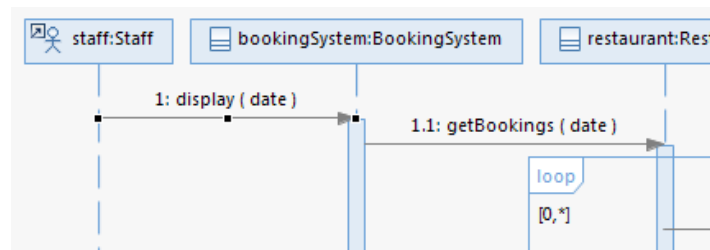


and draw a rectangle around the getDate() message and its return message to get this:



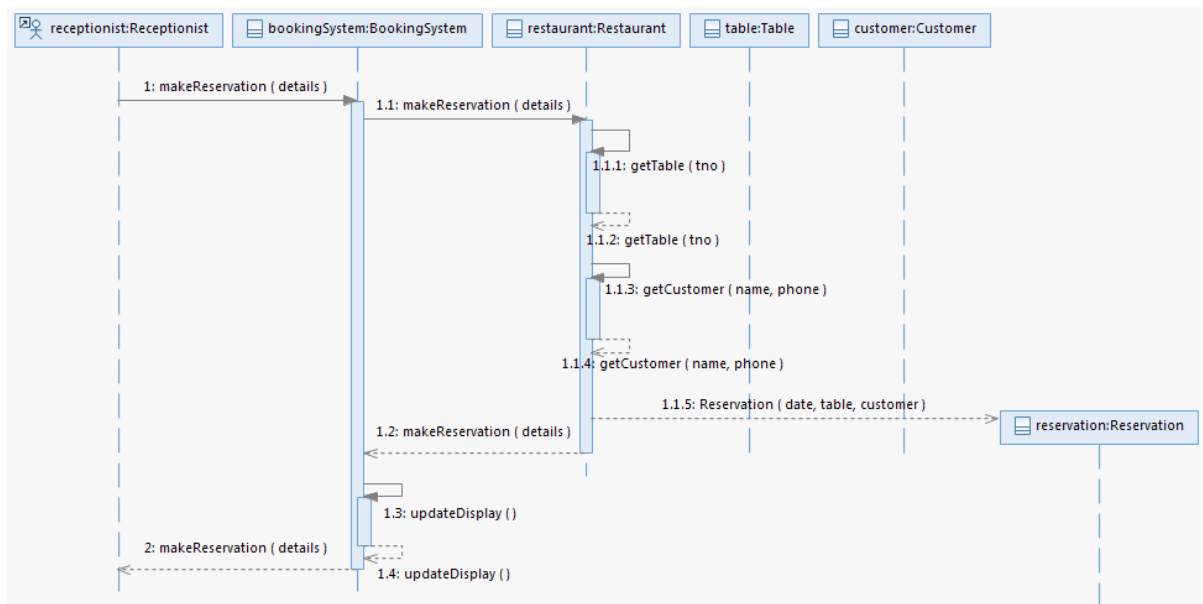
Task 4 – Update the operations on the new Classes

- Look at the new *BookingSystem* and *Restaurant* classes in the Project Explorer – by expanding them out you will see the operations. Click the *display()* operation on the booking system and add an input parameter called *date* leaving the *Type* blank for the moment (you can use the Properties view to add parameters).
- Once complete, have a look at the sequence diagram, you should now have:



Task 5 – Add a new collaboration and sequence diagram for the Create Booking (Reservation) use case

- Create a new collaboration and sequence diagram to show how the system will realise the *Create Booking* use case. You should end up with something like the screenshot below.
- Note 1:** You can select the message or return message arrows to move them up and down.
- Note 2:** You can select a *Create Message* arrow in the palette when you want to show a new object is instantiated (in this case it's the *Reservation* instance).
- Note 3:** When you add the *updateDisplay()* message in the new sequence diagram, you should see that when you are prompted to either create a new operation or use an existing one – there will already be such an operation on the class from when we created it during the last sequence diagram.
- Note 4:** Remember you can add the parameters to the class operations by clicking on them in the Project Explorer and adding them using the Properties view.



PART II – Object Model for the Department / Student / Account

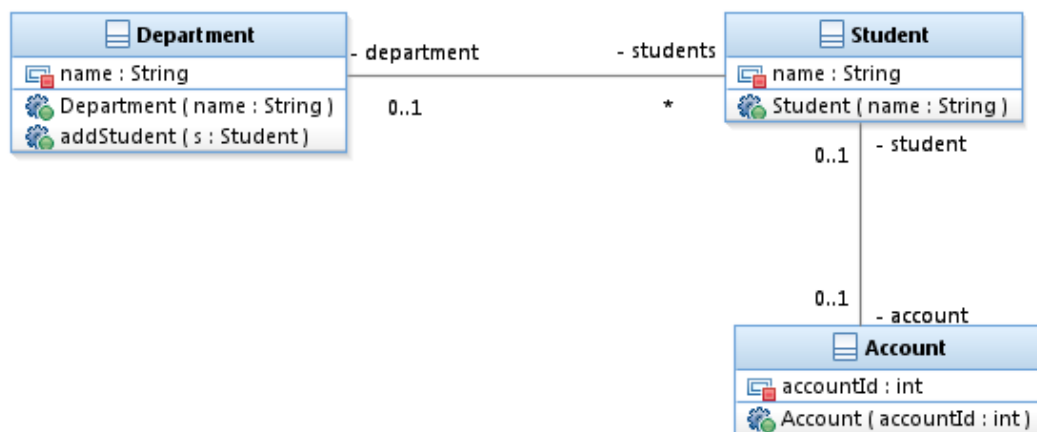
Example & Code Generation

In this part of the lab session we will create a new UML project and specify the analysis model discussed in class with a view to using RSA to generate the code. We can then investigate the code and understand the initial implementation of our model. As an exercise, you will add source and test java code to further implement and test the model.

Task 1 – Create a new analysis model

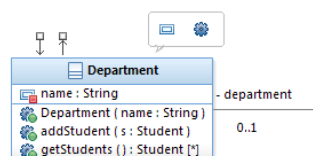
Consider the class diagram and its associations discussed in class – *Department*, *Student* and *Account*. Follow the instructions below.

- As you did in the first lab sheet, create a new UML project in RSA and call it *College*. Use a default *Blank Analysis Package* template as before (**recall, depending on the version of RSA, you may not see *Blank Analysis Package*, if this is the case, choose *Blank Model* and then choose *Class Diagram* as the default diagram**).
- Add to the default model as appropriate to create the following class diagram. Ensure all the details are accurate.
- See note below on adding operations *e.g. AddStudent(s:Student)*.

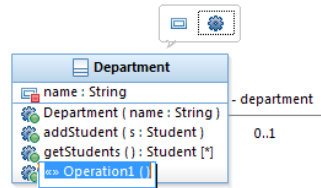


Note: To add an operation to a class:

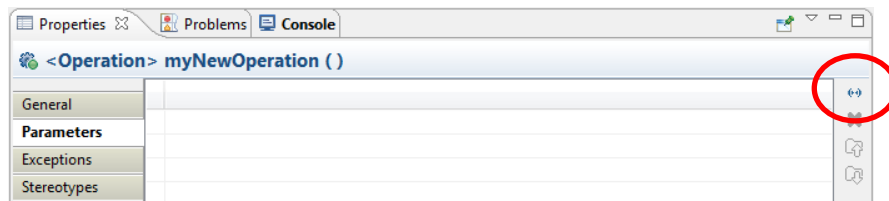
- Hover the mouse over the class to which you want to add an operation and you will see two icons above the top right corner as below




- Click the icon on the right to add a new operation (the operation will become a method in our class later) – you should see something like this:



- Click into the highlighted operation and give it the name you want and press enter. Have a look at your *Properties View* – it will show the various properties for the new operation.
- In your *Properties View*, click the *Parameters* tab. This shows you a list of the parameters for this operation – there will be no parameters listed.



- Click on the *Insert New Parameter* icon (highlighted above) – you should now see a parameter listed. The important information for now is:
 - **Direction:** For now this will either be *In* or *Return* (input parameter / argument and return type respectively).
 - **Name:** A name for the parameter (e.g. theStudent).
 - **Type:** The type of the parameter (e.g. String, Student, Account etc.). When you click the *Type* cell, you can then choose the  button to search for or pick a specific type. When searching for *types* – if you want to make it a primitive (e.g. int, double, long etc.) choose a *JavaPrimitiveType* from the search results otherwise use a *UMLPrimitiveType* (e.g. String).
 - **Multiplicity:** Is it a single reference or collection of references. In the case of *addStudent()* in the *Department* class, this will be '1' for the argument.
- By default, RSA does not show the operation signatures (parameters and return type) in the diagram. If you click on the class to select it and then right-click -> *Filters* -> *Show signatures*, the signatures will then appear in the class diagram.

Tip: If you know the signature, you can type it directly when naming the operation and RSA will update the model accordingly e.g. *addStudent (s : Student)*. RSA uses the following notation:

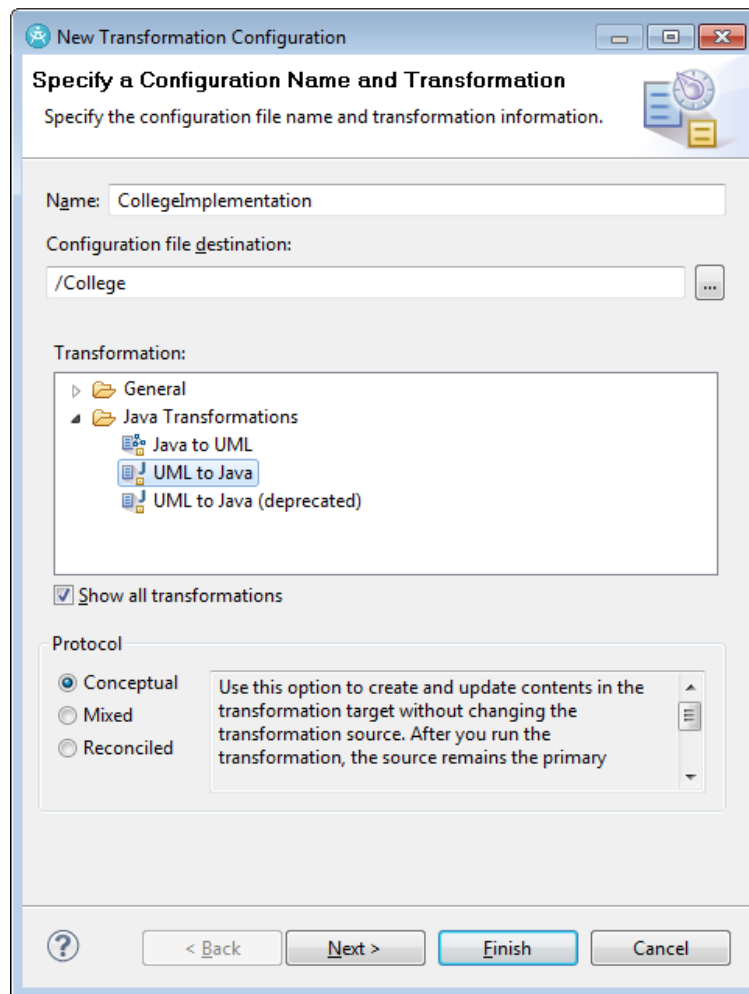
```
methodName ( parameterName : Parameter Type ) : Return Type [multiplicity]
```

Task 2 – Create a Transformation Configuration file

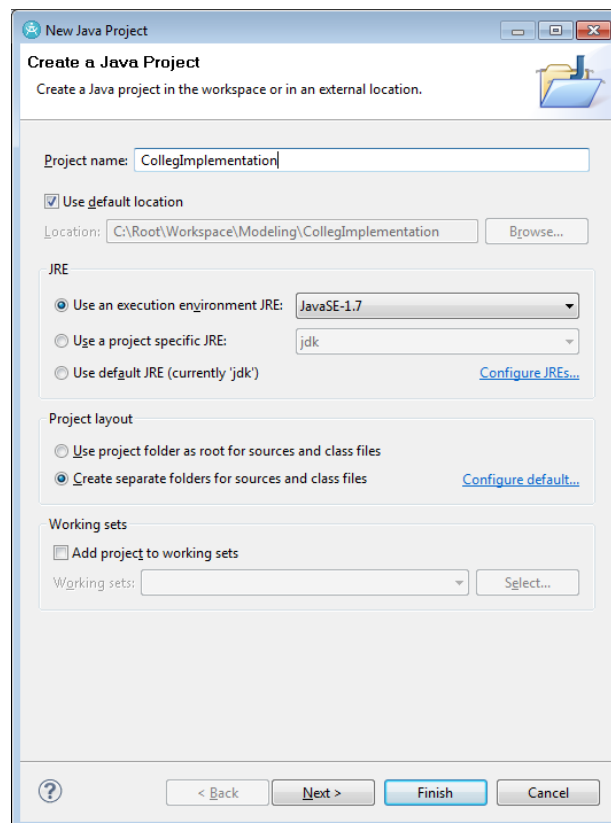
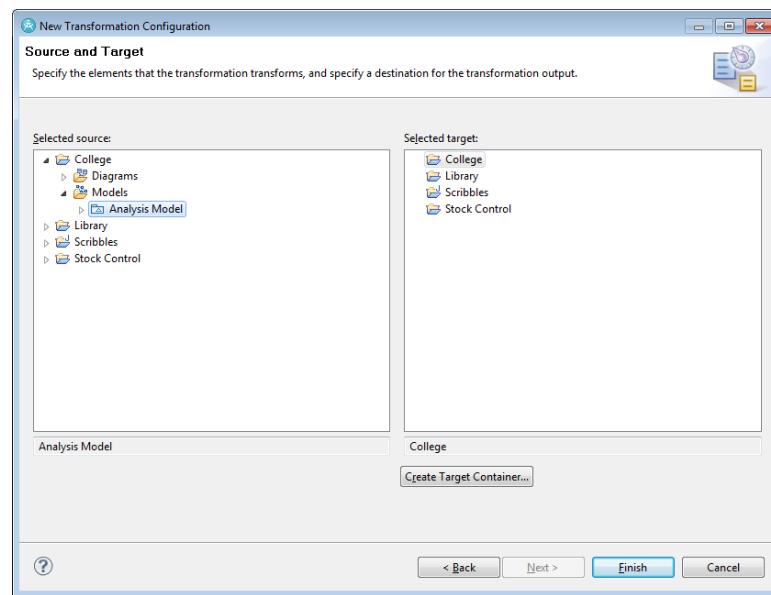
In this task we will create a Transformation Configuration file. We will use this configuration file to transform our analysis model into a Java project which will contain implementations of the three classes in our model.

- From the menu bar across the top of the window, choose *Modelling* -> *Transform* -> *New Configuration*.

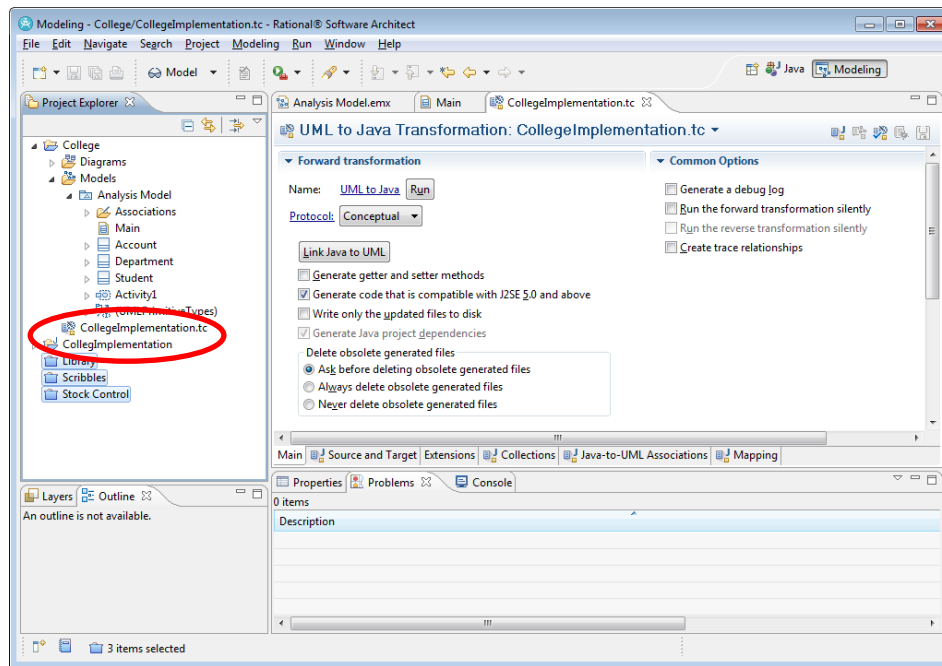
- Fill out the screen as below (you may need to check *Show all transformations* to see the *UML to Java* transformation).



- Click *OK* if you are prompted with the screen asking to enable Java Modelling. Click *Next* to proceed.
- You should get the following screen – choose *Create Target Container...* to create the new *java project* for our implementation code.



- Click *Finish*
- You should now have the screen below:



- Make sure the *Generate getter and setter methods* is checked (default is unchecked as in diagram above).
- Right-click on the *CollegelImplementation.tc* file in the Project Explorer and choose *Transform -> Run UML to Java*.
- You should now have a *CollegelImplementation* java project in the Project Explorer with a default package containing the three classes' implementations.

Task 3 – Investigate the Generated Code

- Have a look at the generated code – note the references (attributes) that were added to implement the associations (*department*, *students*, *student* and *account*).
- Ensure you are happy with how the generated code provides the basis for implementing our model.

Task 4 – Add some code to maintain the links between objects at runtime

Look at the class slides on webcourses for week 2 lecture. On page 13 and 14 there is code that will maintain the runtime links between instances of these classes.

- Enter the code at the appropriate locations in the implemented java classes.
- Ensure there are no compiler errors.
- Right-click on the *CollegelImplementation* project and choose *New -> Source Folder*. Name the new source folder *test*.
- Create a new class in this folder called *DepartmentTest.java* (right-click the folder and choose *New -> Class* – ensure you choose to create a static *main* method).
- See if you can write some test code within the main method to test that the code is maintaining the links as expected – keep the following in mind
 - You will need to instantiate the appropriate objects

- We want to check that if, for example, we add a student to the collection of students within a department object then, automatically, the student object should have its department reference set to point to that department object.
- Note, so far, we have only addressed the aspect of setting up the links and not removing links.