**Software Engineering Lab #11**

**Software Analysis and Design with UML (Part 3)**

**System Design**

During system analysis, use case realizations show how the business functions of the system can be implemented with objects in the application layer. The design task however is to extend this level of modeling to the other layers (normally presentation layer and storage layer). All classes in the design stage will be further reﬁned so that they contain fuller information about data types, message parameters and so on. By the end of the design stage, we aim to have an understanding of the system that is detailed enough to permit implementation to begin.

**Applying a Design Pattern (for the presentation layer)**

In the MVC architecture, the application layer should be independent of the presentation layer. When there is an interaction from the user, the presentation layer usually request data or send command to the application layer. The problem is that a class in the application layer needs a mechanism to notify a class in the presentation layer when a view update is required. An object of a class in the application layer should know nothing about objects that need to be updated but provide an interface to notify them given that those objects register themselves to receive the notification. In this case, a suitable solution is provided by the **Observer** pattern.

The following simple example code demonstrates how to implement classes that apply the Observer pattern:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | **class BookingSystem(object):**  def \_\_init\_\_(self):  self.observers = []  def **addObserver**(self, o):  self.observers.append(o)  def **notifyObservers**(self, data):  for o in self.observers:  o.update(data)  def **getBookings**(self):  data = "booking data"  self.notifyObservers(data)  return data  **class BookingObserver(object):**  def **update**(self, data):  pass  **class StaffUI(BookingObserver):**  def \_\_init\_\_(self, name):  self.name = name  def **update**(self, data):  print(self.name + ": StaffUI.update() -- " + data)  # ...  s = BookingSystem()  ui1 = StaffUI(s,"UI#1")  ui2 = StaffUI(s,"UI#2")  s.addObserver(ui1)  s.addObserver(ui2)  # ...  s.getBookings() |

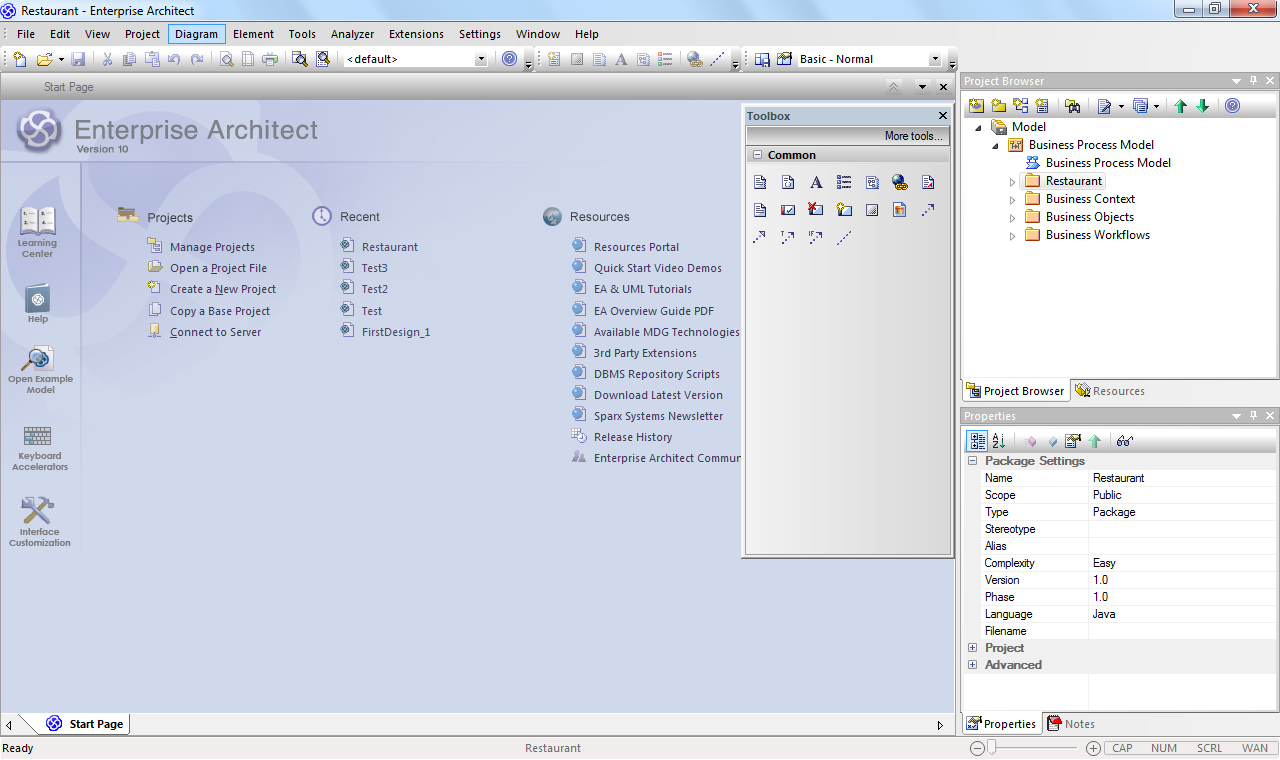
From the above example, the BookingSystem class maintains references to a set of observers and, when its state changes, sends the “update” message to each observer. The observers could be of any class, but the booking system only accesses them through the interface defined in the BookingObserver class. The BookingObserver class is defined in the application layer. The BookingSystem knows nothing about the existence of the StaffUI class. The StaffUI class implements the BookingObserver’s interface and can be added as an observer of the BookingSystem object.

For a PyQt application, you can take advantage of using signals and slots to apply the Observer pattern instead of implementing your own solution to your system.

1. **Creating a state chart diagram in Enterprise Architect (EA)**

A State chart or State Machine diagrams illustrate how an element can move between states, classifying its behavior according to transition triggers and constraining guards. In this tutorial we will create a new state chart diagram using EA as the following steps.

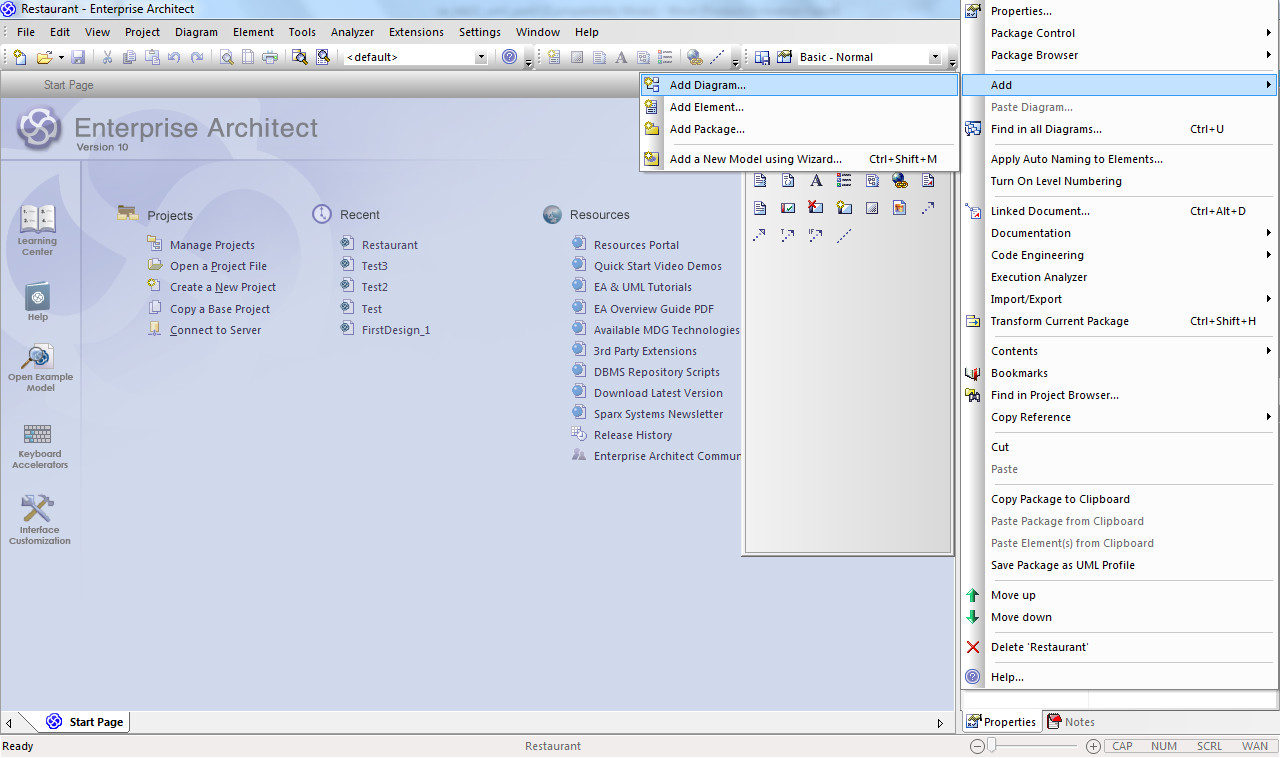
1. From the Enterprise architect browser window, right click on the **Restaurant package** created earlier.



**Right click** on the **Restaurant package**

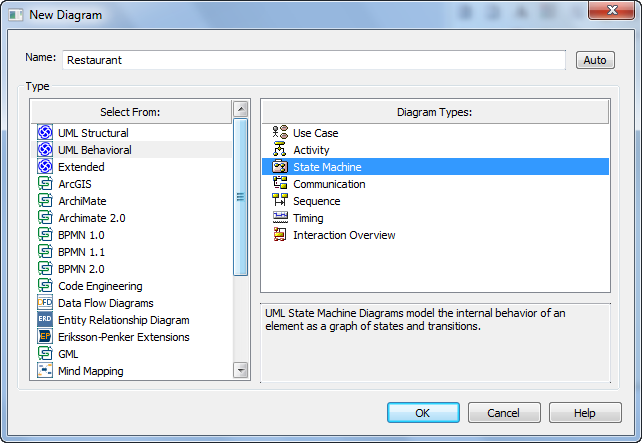
**Figure 1:** Enterprise architect browser window

1. Select  **“Add > Add Diagrams...”**. or click at  icon on Project Browser.

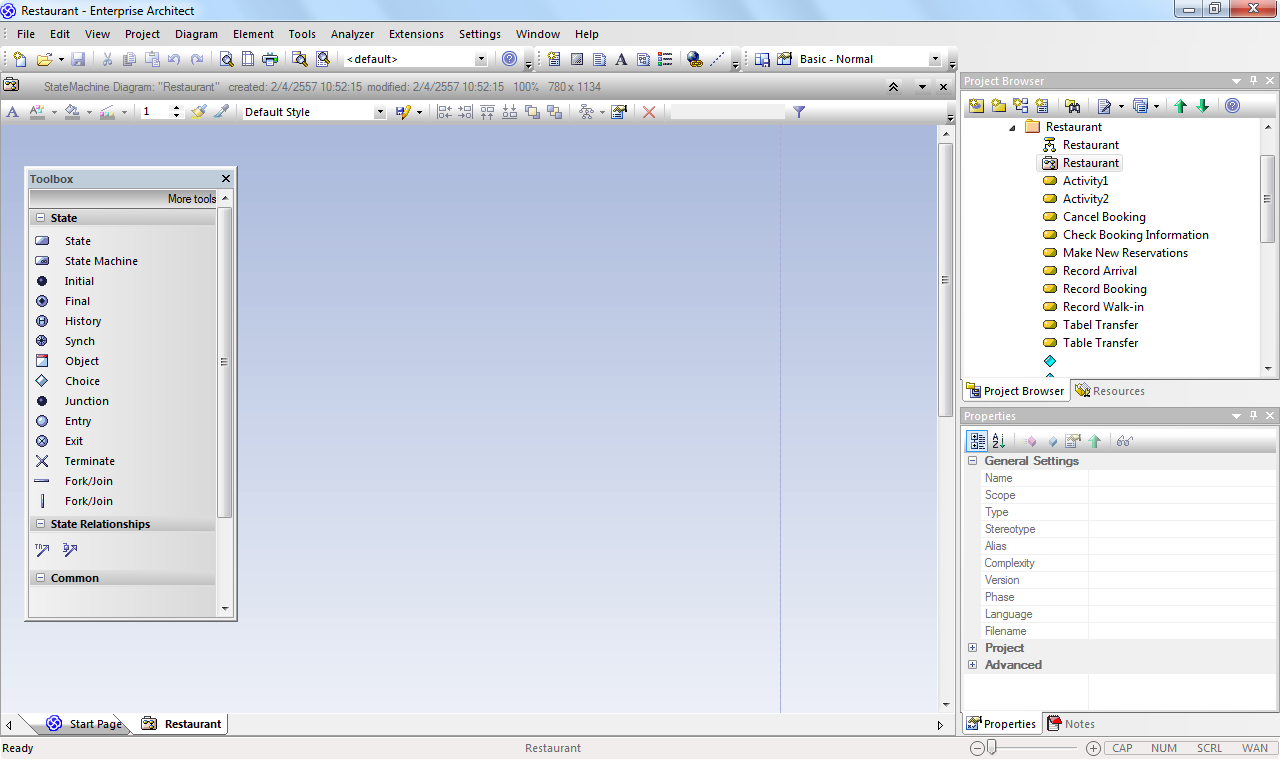


**Figure 2:** Add New State chart diagram from **Add Diagram…** menu

1. Choose "**UML Behavioral -> State Machine**" on "**New Diagram**" window.



**Figure 3:** Select State Machine on Diagram Types.



**Drawing Toolbox**

**Properties**

**Figure 4:** Drawing Toolbox component of Sate Machine

1. After creating a state chart diagram, you can right click on it and select **“Properties...”** to change the diagram features such as name, description, and properties.
2. To start adding components into the state chart diagram, if the drawing toolbar is not already shown, open the drawing toolbar by press **"Alt+5".** You can select a component, from the diagram tools in the drawing toolbar, to add into the state chart diagram.
3. Right-click on a component in the diagram and then select **“Appearance...”** from the menu if you want to customize the component appearance and select **“Properties...”** to add to it any further detail description. You can select a component, from the diagram tools in the drawing toolbar, to add into the statechart.
4. Use the Initial tool to create an Initial Pseudo state. Click OK in the State dialog.
5. Use the State tool  to create States named “Created”, “Identifying” and “Moving” respectively.
6. Use the Transition tool  to add Transitions from the Initial state to the Created state, from “Created” to “Identifying”, from “Identifying” to “Moving”, and from “Moving” to itself (click and release within Moving). Arrange your diagram to look like this:

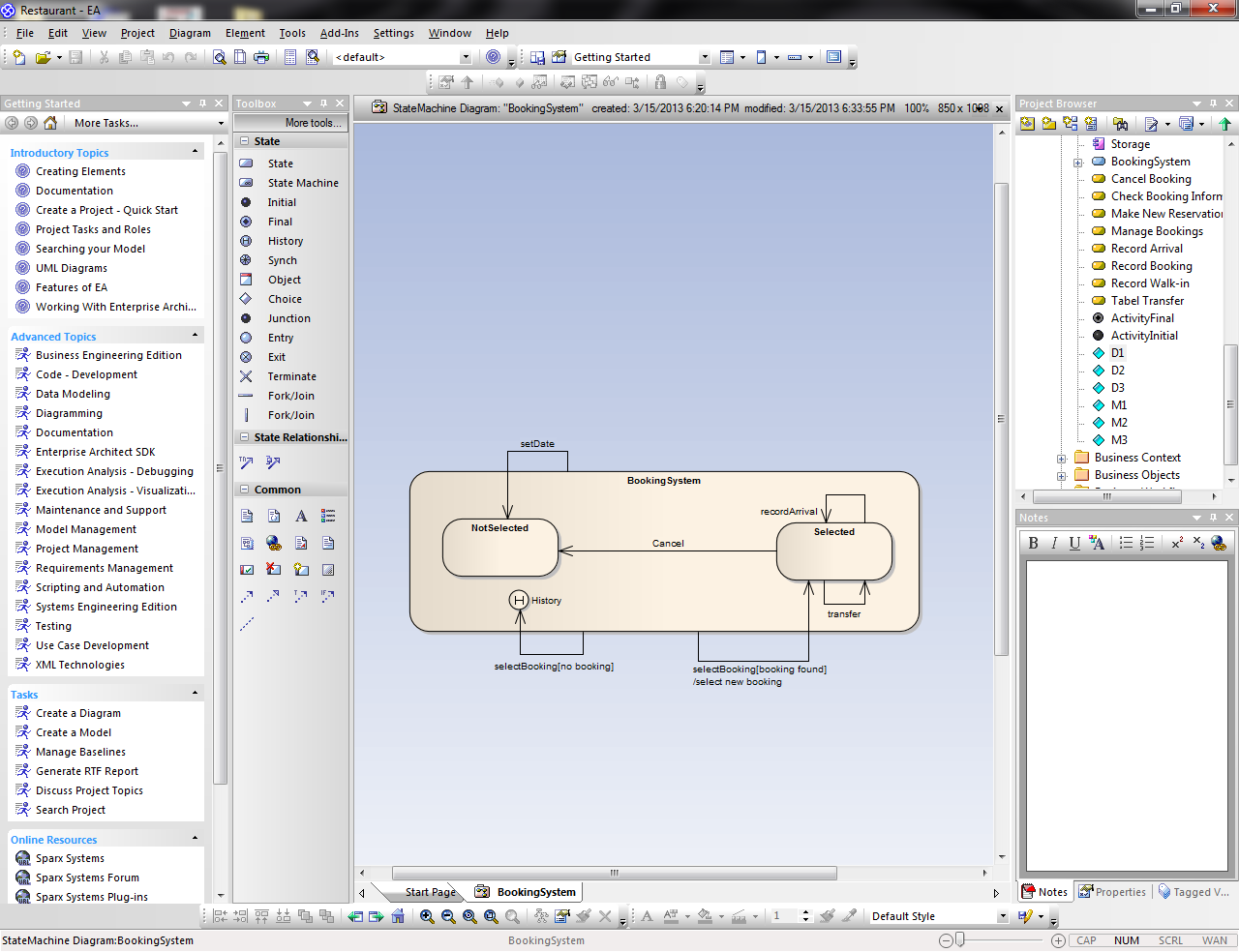
****

1. Select the self Transition (transition to itself) of the Moving state, right click and select Line Style --> Bezier.

**Note :** For more details about this , please follow the link below.

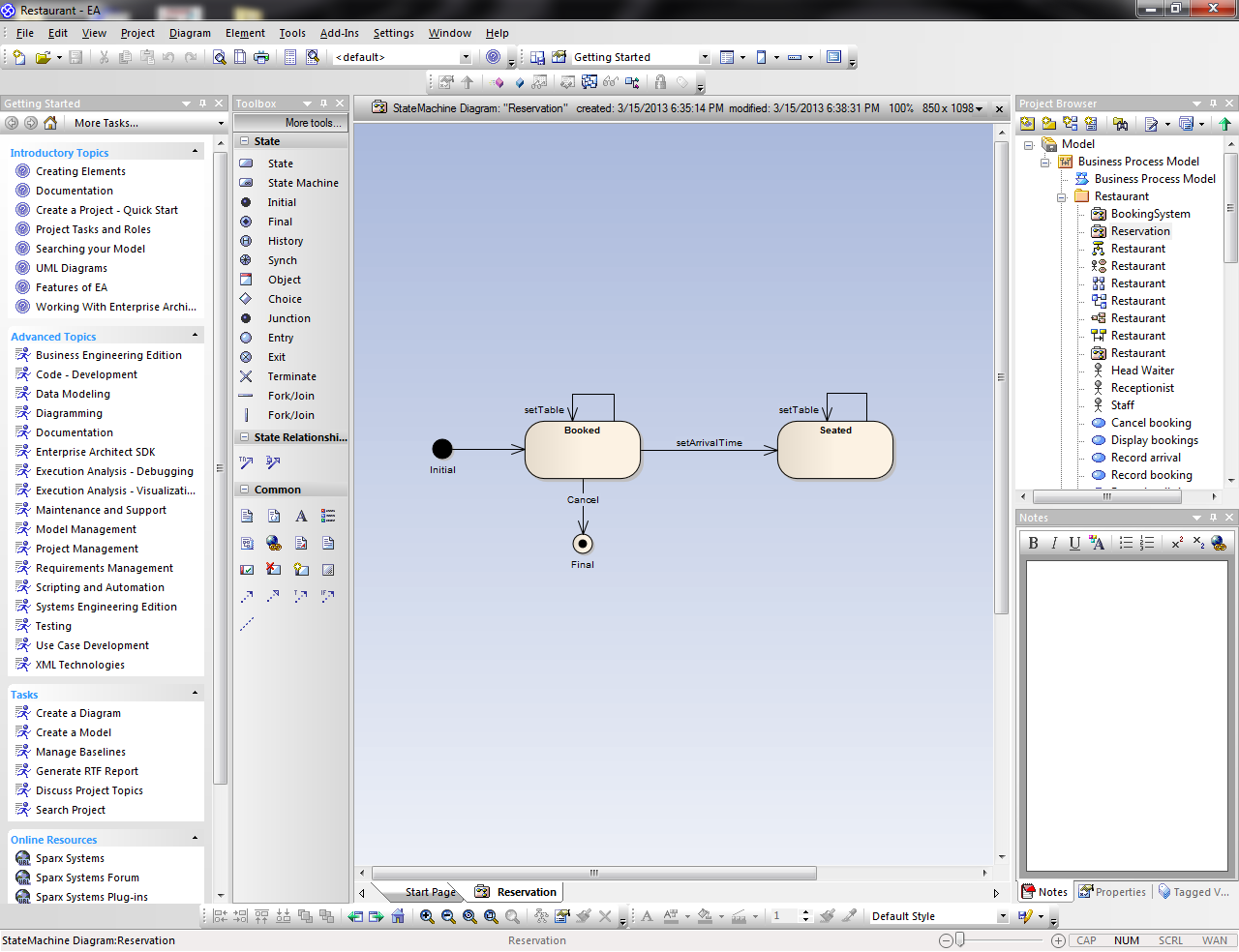
<http://www.sparxsystems.com/resources/index.html>

* **Draw the following state chart for the “BookingSystem” class:**



**Figure 6**

* **Draw the following state chart for the “Reservation” class:**



**Figure 7**

**When Not to Draw a State chart**

It is not necessary to draw a state chart for every class in a system. Normally, state charts are only drawn for classes with ‘interesting’ behavior: typically, these will be classes that expect to receive messages in a certain fixed order or which exhibit state-dependent behavior, responding to the same message in different ways at different times.

For example, it might seem that a state chart could be drawn for the customer class, to show, for example, a distinction between a customer who has a reservation made for some time in the future and one who hasn’t. The difference likes this between customers can certainly be identified, but from the perspective of the system it is entirely irrelevant. The operations in the interface of the customer class can be called at any time, and have the same effect at all times. A state chart for the customer class specifying this would therefore not be a useful addition to the system design documentation.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_