VASAVI COLLEGE OF ENGINEERING (Autonomous)
IBRAHIMBAGH, HYDERABAD – 500 031
Department of Computer Science & Engineering

INNOVATION IN TEACHING

Course Name: Software Engineering **Faculty: M Jithender Reddy**

Topic: UML Diagram **Year/Sem: VI Sem**

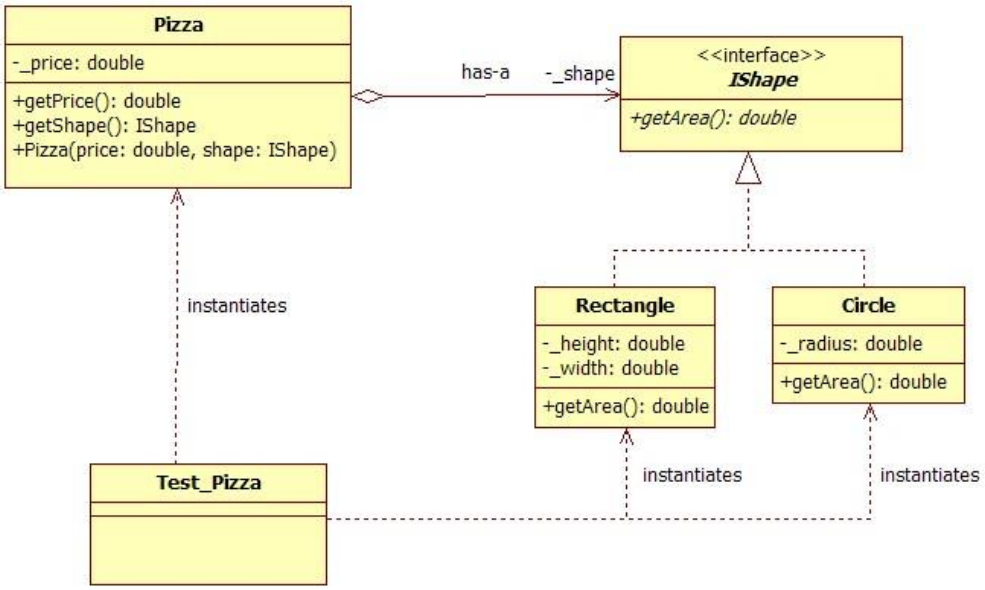
Teaching Aid / Tool Used: StarUML

Description of the Tool:

StarUML (SU) is a tool to create UML class diagrams and automatically generate Java "stub code". SU can also reverse engineer Java source/byte code to produce the corresponding UML diagrams.

Tools Usage in Teaching:

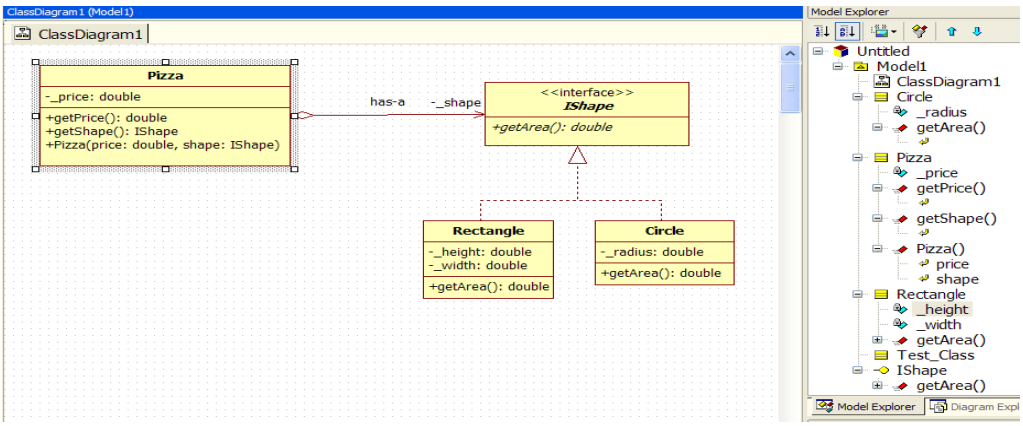
StarUML will generate code that reflects the class structure, but not the specific actions on any objects. For that, after creating the diagram using SU, you'll edit the resulting stub code to add the rest of the functionality to the code, filling in what each method should do.



1. On the "Model Explorer" pane on the upper right side, select the (as-yet) "Untitled" model.
2. Either on the main menu under "Model", or by right-clicking the selected model, got to "Add/Design Model"
3. Either on the main menu under "Model", or by right-clicking the selected model, got to "Add Diagram/Class Diagram":
4. Save the project now so you do not lose any progress if some problem arises. From the "File" menu, choose "Save", and select a location to save the project. Your StarUML project should now look something like this:

5. Now to begin actually creating the diagram, from the "Toolbox" which starts by default on the left side of the screen, select the "Class" icon, and left-click somewhere on the diagram window. This should create a new class with a generic name. Rename the class to Circle by double clicking on the name.
6. Add an "Attribute" (or field) to Circle by right-clicking the object on the diagram, expanding the "Add" menu, and pressing the green "Attribute" button. Type in the desire name of the field, "_radius".
    - Specify the data type in the Properties panel (lower right side of window) by typing double in the "Type" slot.
    - Internal data of a class (field/attributes) are always private because they are strictly for personal use by the class to help it determine its behavior. So, in the Properties panel for the _radius field, select PRIVATE for its Visibility.
7. Repeat the same process to create a class called Rectangle with private _width and _height fields of type double. You may notice using the "Model Explorer" on the right is faster to add these, but do however note that adding the classes and interfaces themselves in this toolbox (instead of using the toolbox on the left and clicking on the palette to create the object) will not create the objects in the diagram. If you choose to use the "Model Explorer", the area we will be interested in is visible after expanding the "Design Model" group.
8. Create an interface called IShape
    - From the toolbox, choose "Interface" and click somewhere on the palette. Rename the generic name to IShape. Select the interface by left-clicking the item after it is created.
    - On the top toolbar, select the dropdown "Stereotype Display" and change the value to "None". This will change the previous circular shape into a rectangular shape.
    - Also on the toolbar, de-select the "Suppress Operations" box. This will allow us to see what operations the interface has in the diagram view.
    - Add a getArea method of type double to the IShape interface.
        - This can be accomplished by right clicking the interface, expanding the add menu, and pressing the red "Operation" button. Enter the name as: getArea.
        - To set the return type, expand IShape in the "Model Explorer", right click the getArea method you just created, and select "Add Parameter". In the "Properties" box, change the parameter's name to nothing, "", change the "DirectionKind" to "RETURN", and change the "Type" to double.
    - On both the IShape interface itself as well as its getArea method, check the IsAbstract box in the Property pane. This will make their titles appear as italics, as per the UML standard for interfaces and other purely abstract entities.
9. Make Circle and Rectangle implement IShape by selecting the "Realization" arrow from the toolbox, clicking on Circle and dragging the line to IShape. Repeat the same process for Rectangle. This is adding the relationship that Circle and Rectangle will implement the IShape interface.
    - To make the connector line make nice right-angle bends, right-click the line and select "Format/Line Style/Rectilinear". You can make your diagram look cleaner by simply laying arrowheads that point to the same place right on top of each other, making it it look as if there is only one arrowhead.

10. Since the Circle and Rectangle class both implement the IShape interface, they must have the same behaviors (methods) as IShape.
    - In the Model Explorer pane, copy the getArea method (Ctrl-C or right-click and select Copy) from IShape to both Circle and Rectangle.
    - The implemented methods in the Circle and Rectangle classes are not abstract, but concrete because they actually perform some particular action (i.e. calculate the area for a circle and rectangle respectively). So, uncheck the IsAbstract box for those methods.
11. Your diagram should now look something like this:
12. Add a class called Pizza.
    - Add a private _price field of type double.
    - Add a public getPrice operation that returns type double.
13. To make Pizza reference an IShape, select class Pizza.
    - Select the "DirectedAssociation" arrow in the toolbox, click on Pizza, and drag to IShape.
        - Now select the arrow, and in the "Properties" box on the right, change the name to "has-a", change "End1.Aggregation" to "AGGREGATE" (this is a formal diagrammatic statement that a pizza is made up, i.e. "aggregated", with another object, a shape object).
        - Change the "End2.Name" to _shape. This will automatically add a private field called _shape of type IShape to Pizza.
        - change the End2.Visibility to PRIVATE.
    - Create a "gettor" method (Routine) for _shape called getShape that returns IShape. That is, create an operation called getShape that returns IShape.
14. Constructors are special pieces of code used to initialize an instance of a class when it comes into existence.
    - To add a constructor for Pizza, right click on Pizza, expand the "Add" menu, and select "Operation". From here, add a normal operation as usual, with input parameters double price and IShape shape.
        - Adding an input parameter is just like adding an output parameter for the return type earlier, except you specify the desired parameter name, such as price and shape, and the appropriate data type.
    - Add a Circle constructor with parameter double radius.
15. Add a Rectangle constructor with parameters double width and double height. Your diagram should now look something like this:



16. To illustrate one more type of UML class diagram feature, add another class to your diagram called "Test_Pizza". This would be a class that uses the Pizza and IShape-derived classes, say, for testing purposes.
    - Dependency lines help show relations between classes that occur more dynamically. For instance, one class may instantiate another class but not hold permanent a reference to it by using a field. Or a class's method may take another class as an input parameter, retaining a reference to it only for the duration of the execution of that method.
    - Add dependencies between different classes by selecting the "Dependency" arrow from the toolbox, selecting a dependent class, and dragging the arrow to the class it is dependent upon. In this example, Test_Pizza "depends" on Pizza, Circle, and Rectangle because it instantiates them.
    - Enter a label for a dependency by changing the "Name" property in the Properties box or by double-clicking the dependency line. Typically when one class instantiates another class, we label the dependency line "instantiates" (surprise, surprise!).
    - You can move the label of the dependency line around to a more aesthetic location by selecting the label on the diagram and dragging it.
    - Dependencies have no effect on code generation.