ECS505U Software Engineering

Laboratory Exercise 2

Generating Java Code from UML Diagrams Using Visual Paradigm

Version 1.3, 16/10/2016

1. Prerequisites

- You should be able create simple UML class diagrams using Visual Paradigm.
- You should be able to run NetBeans. For campus students version 8 or newer is pre-installed on all machines. For non-campus students check the QMplus course page for instructions (the "Using the NetBeans Environment" link is the one you need).
- You should watch the tutorial video titled "Java multiple inheritance with UML" available on QMplus course page.

2. Additional Reading

- Java Interfaces http://docs.oracle.com/javase/tutorial/java/concepts/interface.html
- Java Enumerations (Enum type) http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
- UML realization relationship http://www.uml-diagrams.org/abstraction.html
- UML realization relationship in VP http://www.visual-paradigm.com/VPGallery/diagrams/Class.html#realization

3. Aims

The aim of this assignment is to get you used to design and implement well-defined classes using Visual Paradigm. The skills you will learn and practice are:

- UML class design using Visual Paradigm.
- Learn about multiple inheritance in Java.
- Learn how to design Java interfaces and Enum types in UML.
- Learn how to auto generate Java code from a UML design.
- Practice modifying the auto generated Java code using NetBeans.

4. Clone Git repository

1. Clone Git repository https://github.research.its.qmul.ac.uk/eew954/Lab2.git. If you haven't done the Git clone tutorial from the Lab1 you can download the tutorial and follow it to clone the repository. You need to replace all "Lab1"s with "Lab2" in order to clone the repository correctly.

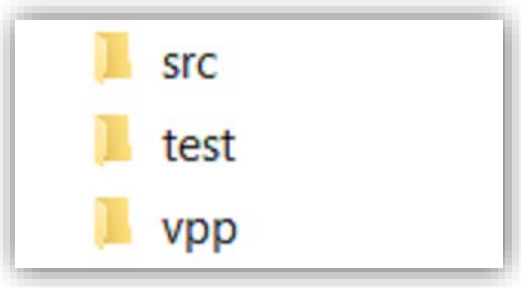

Figure I: Folder contents

2. Create a new project named "**Lab2**" and add the cloned folders (src and test) to this new project. If you create the project on the folder that you cloned the repository the folders will be added by default.
3. Open lab2.vpp in the vpp folder with Visual Paradigm.
4. You should have a class diagram in the project with the elements depicted in Figure II.

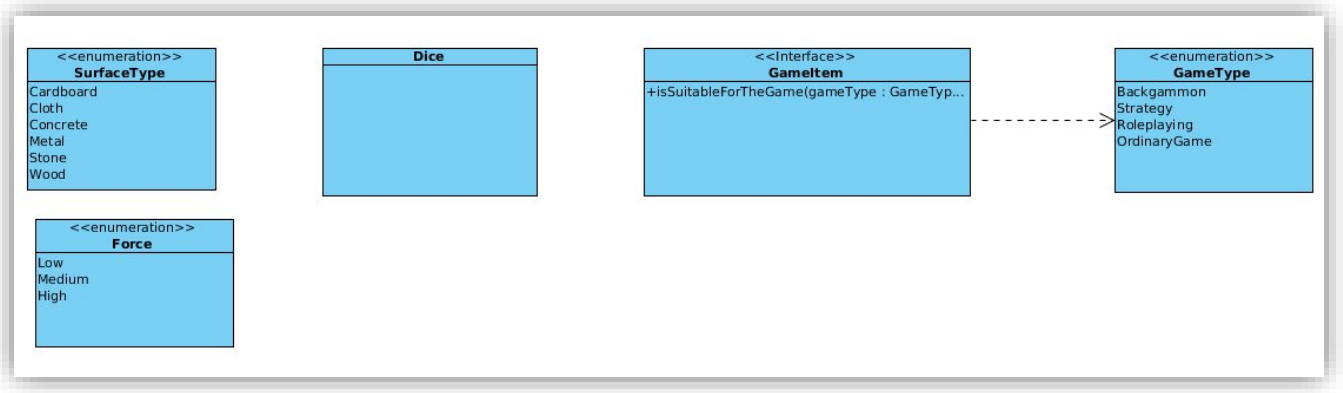

Figure II: Overview of the initial class diagram

5. You are expected to modify this class diagram. Add the required attributes and methods to Dice and other necessary classes based on the specifications in the next Section.

5. Specification of the classes

Don't worry too much **if you do not completely understand** the following specifications because **some parts of the specifications are intentionally described in an indirect manner** in order not to reveal the details of the expected design. Don't hesitate to ask questions to clarify the parts that seem confusing.

1. The Dice class is a class that represents the common behaviour and attributes of all existing dice entities. The details of the types of dices is presented in Table I. You have to build the Dice class considering the fact that you don't need to create an instance of a Dice but instances of the type of dices mentioned in Table I need to be created.

Number of Sides	Name	Number Range
4	Tetrahedron	1-4
6	Cube	1-6
8	Octahedron	1-8
10	Pentagonal trapezohedron	1-10
12	Dodecahedron	1-12
20	Icosahedron	1-20

Table 1: Dice types and their properties. Number range represents the range of possible values produced by the *roll* method of the corresponding dices.

2. The Dice class has two attributes: ***numberOfSides*** and ***numberRolled***. The latter represents the current number a dice is showing (number on upper surface) which is an integer between 1 and *numberOfSides*.

   HINT: The attributes should be private, but there should be public 'getter' and 'setter' methods for both *numberRolled* and *numberOfSides* (e.g. *setNumberRolled*).

3. The Dice class must have a behaviour ***roll*** (which is responsible for setting the *numberRolled*). The behaviour ***roll*** represents general behaviour of all dices (all dices should behave the same).

4. All classes representing the dices on Table I should be a specialization or realization of both the ***Dice*** class and ***GameItem*** interface (There are multiple solutions to this design problem so you can use any solution that fulfils this requirement).

   HINT: Beware, the generalisation element (the arrow) that represents the relation between a class and subclasses is not used for describing the relation between an interface and other classes. If you don't know which UML element is used to describe the later relation, please watch the tutorial video.

5. All dice classes have a ***trickRoll*** behaviour. The function of this behaviour is based on two inputs: first type of the surface dice is rolled on (represented as SurfaceType) and second force the dice rolled/thrown with (represented as Force).

   HINT: These two parameters are Enum data types (the classes with "<<enumeration>> identifier) and they are already included in the class diagram.

6. The behaviour ***trickRoll*** does not represent a general behaviour and all dice types (Table I) should have their own specialised ***trickRoll*** behaviour.

   HINT: The existence of the ***trickRoll*** method must be enforced by the Dice class on all other dice classes. The method should not return any value. Solution to this requires you to make a design decision so think about it!

6. Modify the Generated Java Code with NetBeans

1. Generate the Java code from the UML diagram. The Java code should appear in the same folder as your project "Lab2.vpp" following the package name under "ecs505/lab2/dice/"

2. Copy and paste the "ecs505" folder into the source code folder in your NetBeans project, the path is "<your-project-folder>/src/". If the **src** folder doesn't exist create it.

3. Open the Lab2 NetBeans project from the folder you unzipped the initial "Lab 2 exercise files" file. You should be able to open the project using NetBeans.

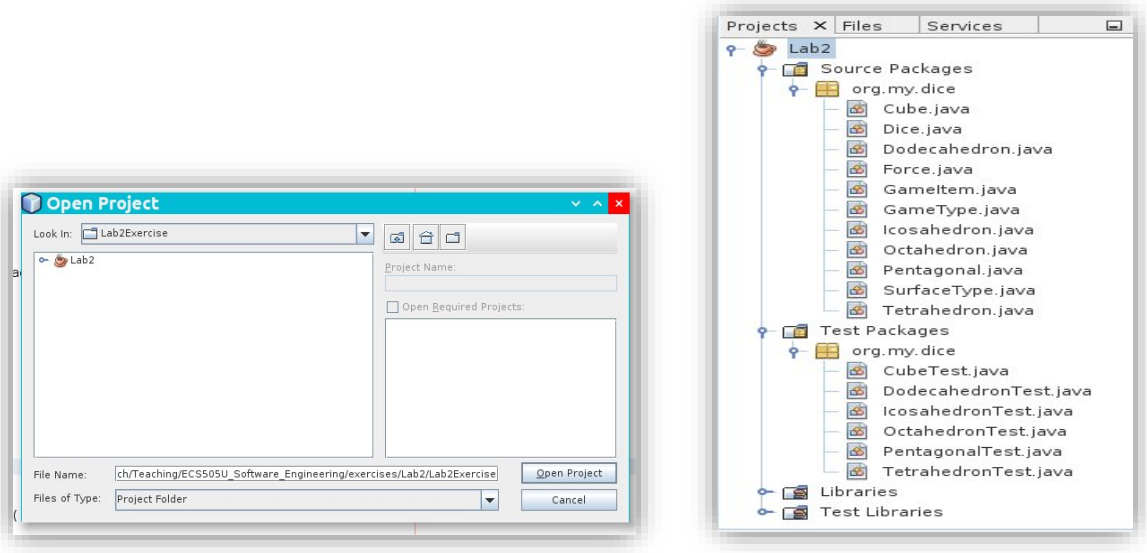

Figure III: NetBeans project in the exercise folder (left) and Dice exercise code structure (right)

4. You should see a similar code structure as Figure IV if you correctly followed the instructions above.

5. If you realised the interfaces and add the default constructor in your class diagram each of the generated dice classes should look like the code snippet Figure VI. How to realise interfaces and abstract methods is explained in the tutorial video. Don't forget to remove the auto generated exception.

```java
package org.my.dice;

public class Pentagonal extends Dice implements GameItem {

        public Pentagonal() {
                // TODO - implement Pentagonal.Pentagonal
                throw new UnsupportedOperationException();
        }

        /**
         *
         * @param surfaceType
         * @param force
         * @return
         */
        @Override
        public void trickRoll(SurfaceType surfaceType, Force force) {
                // TODO - implement Pentagonal.biasedRoll
                throw new UnsupportedOperationException();
        }

        /**
         *
         * @param gameType
         * @return
         */
        @Override
        public boolean isSuitableForTheGame(GameType gameType) {
                // TODO - implement Pentagonal.isSuitableForTheGame
                throw new UnsupportedOperationException();
        }

}
```

Figure IV: Auto-generated Java code snippet

6. If you haven't realised the interfaces in VP you can do it in NetBeans. In any class file by clicking the warning icon (depicted in Figure VII) next to your class definition and select the first option "Implement all abstract methods". This will create the abstract methods from the inherited class and interfaces but you need to implement the default constructor yourself. (Make sure you performed this for all classes before the nest step)

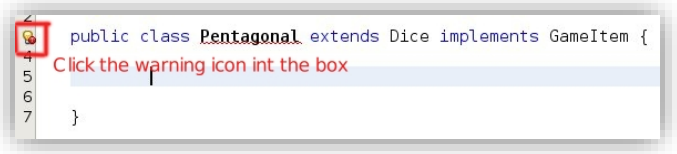

Figure V: Realization in NetBeans

7. Each dice class (except the Dice class) should set the number of sides in the constructor. For example if you created a class for petagonal dice you should set the *numberOfSides* variable in that class to 10. Beware, you set the visibility of the *numberOfSides* to private in the abstract class as a result you cannot access the variable directly in the subclasses. You should you the "setter" method of the variable instead.

8. You need to set the general behaviour of ***roll*** method for all dice classes. Beware the behaviour of the *roll* method is same for the all dice classes and can be done for all of them at once. If you don't know how to implement generalised behaviour for all subclasses of a class please watch the tutorial video.

9. The behaviour of the roll method should be as followed.
    * Generate a random number within the range of the possible values (on Table I) of that dice type. You should use the *numberOfSides* to generate the value. Remember the "getter" method for thevariable.
    * You should set the value of *numberRolled* to this randomly generated value.

To implement the roll method you need to know that there is a standard java function ***Math.random()***. Random method returns a random number between 0 and 1. But the range 0.0...0.999999+ is not what you need, so it is necessary to scale the range by multiplying and translate the values by addition. Moreover, since an int is needed, casting is required. For example, if you need an int in the range 1 to 10, the following code could be used.

int n = (int) (10.0 * Math.random()) + 1;

The multiplication scales the range to 0.0 to 9.9999+. Adding the cast (int) gets it into the integer range 0 to 9 (truncation, not rounding), then addition of 1 translates it into the range 1 to 10.
(There are a number of other ways you can do this, including creating an instance of the java Random class and using the *nextInt* method, but the above approach is the easiest.)

new Random().nextInt(10) + 1;

10. In order to implement ***trickRoll*** method we will use the table below. *trickRoll* is different than regular roll. In this method, rolling of the dice depends on the surface type that the dice rolls on and the force of the throw.

Sides	Dice	Surface Type					
		Cardboard	Cloth	Concrete	Metal	Stone	Wood
4	Tetrahedron	L/M	L/M	L/M	L/M	L/M	L/M
6	Cube	L/M	L/M	L	L	L	L/M
8	Octahedron	L	L	-	-	-	L
10	Pentagonal	L	L	-	-	-	L
12	Dodecahedron	L	L	-	-	-	L
20	Icosahedron	L	L	-	-	-	L

Table 2: The trick roll conditions for each dice. The letters in each box represent the force L for Low, M for Medium and H for High. Boxes with multiple letters means the condition is same for both force types. For example, L/M means for both low and medium force the outcome of the roll is the same.

The values in each surface type column (cardboard, cloth, etc…) on Table II represent the condition where the given dice type **does not roll** (*numberRolled* value does not change). The letters in each box represent the force: L for Low, M for Medium. The boxes with no letters for a dice represent the fact that on the given surface type the dice will always roll regardless of the force. The surface types used in our application are the existing values from SurfaceType Enum datatype. Few examples:

* A cube dice rolled using ***trickRoll*** on a cardboard surface with low or medium force should not roll (Cube-Cardboard : L/M) however if the force is high it should roll .
* A pentagonal dice ***trickRoll***(ed) on a stone surface should always roll.
* A octahedron dice ***trickRoll***(ed) on a wood surface with low force should not roll, but with medium or high force it should roll.

```
public void trickRoll(SurfaceType surfaceType, Force force) {
        // TODO - implement Pentagonal.biasedRoll
        throw new UnsupportedOperationException();
}
```

Figure III: Auto-generated trickRoll method

11. The values for force and surface type passed to the ***trickRoll*** method throught the two input parameters; surfaceType and force. Enum data types often used as flags in programming and in Java below are two examples of use of an enum.

```java
public void trickRoll(SurfaceType surfaceType, Force force) {
    switch (surfaceType) {
        case Cardboard:
            break;
        case Cloth:
            break;
        case Concrete:
            break;
        case Metal:
            break;
        case Stone:
            break;
        case Wood:
            break;
    }
}
```

```java
public void trickRoll(SurfaceType surfaceType, Force force) {
    if (surfaceType == SurfaceType.Cardboard) {

    } else if (surfaceType == SurfaceType.Cloth) {

    } else if (surfaceType == SurfaceType.Concrete) {

    } else if (surfaceType == SurfaceType.Metal) {

    } else if (surfaceType == SurfaceType.Stone) {

    } else { // case for wood suface type

    }
}
```

Figure IVII: Example trickRoll method implementations

The above images only present two different coding for one of the possible ways of implementing the ***trickRoll*** method. You are free to implement it the way you want.

12. You should implement the ***isSuitableForTheGame*** method using the table below. The values in each box represent the return value for a dice with the given game type. For example, only cube dice is suitable for backgammon and all other dices should return false if asked they are suitable for backgammon. You should find out if a dice is suitable for a game type by calling ***isSuitableForTheGame***.

Sides	Dice	Game Type			
		Backgammon	**Strategy**	**Roleplaying**	**Ordinary game**
4	Tetrahedron	False	False	False	False
6	Cube	True	False	False	True
8	Octahedron	False	True	False	False
10	Pentagonal	False	True	False	False
12	Dodecahedron	False	True	True	False
20	Icosahedron	False	True	True	False

Table 3: Suitability of each dice type for different board game categories

7. Test your Code

Add the available JUnit library to the project libraries. Run the JUnit test for each class you generated and verify their correctness. If you used class names other than the ones specified in this document, you need to modify the unit test files in order to be able to run them.

You can run a Junit test case separately in NetBeans by right clicking on it and selecting “Run file” option or run all test cases by right-clicking the project name (on the project layout window) and selecting “Test” option.

After running the test cases you can see the results at the bottom of the IDE as depicted in Figure VIII.

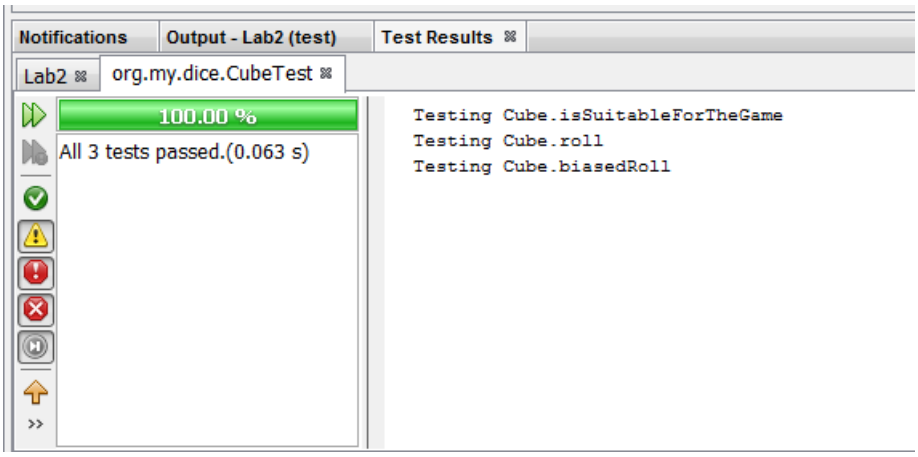

Figure VIII: Sample test result