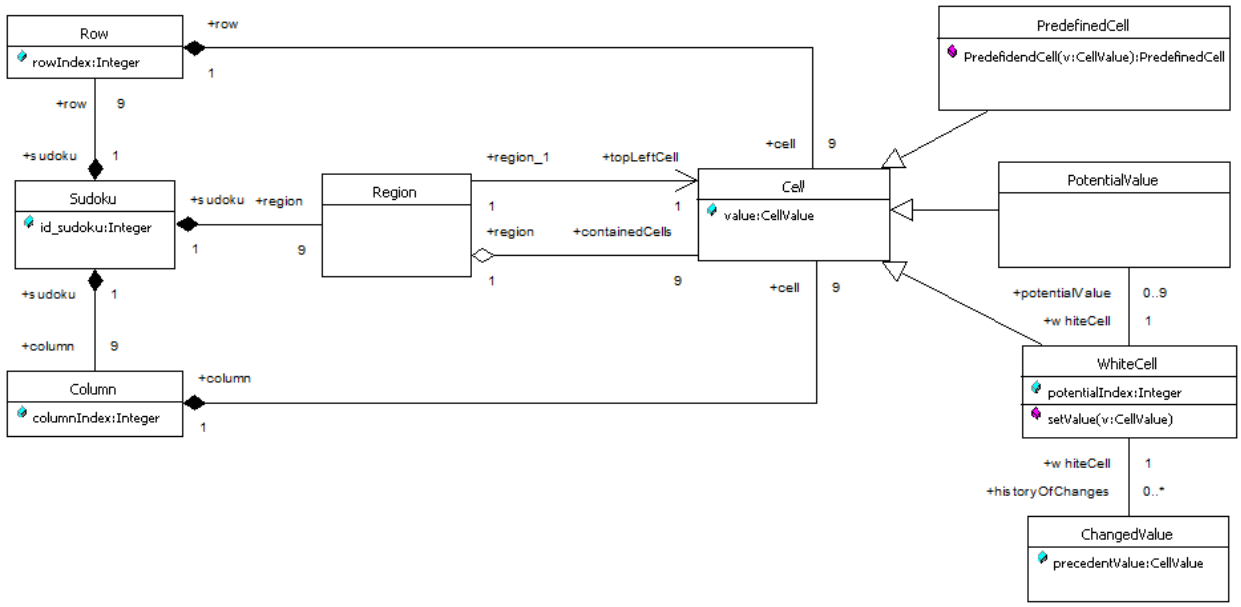
ISS 24-06-2017- IE

I. A Sudoku puzzle is shown in the figure below. To complete this puzzle requires the puzzler to fill every empty cell with an integer between 1 and 9 in such a way that every number from 1 up to 9 appears once in every row, every column and every one region of the small 3 by 3 boxes highlighted with thick borders. Each cell corresponds to a rowIndex, which represents the row of the cell, a columnIndex, which represents the column of the cell and has a value, which represents the content of the cell. A cell belongs to a region; each region is defined by the topLeft cell and by the contained cells. Initially, the puzzle contains predefined cells whose values remain unchanged and by empty cells, also named whiteCells. During the game, the puzzler starts by setting the value of empty cells trying to comply with the constraints mentioned in the beginning. Once the value of an empty cell was set, the type of the cell is considered to be a new kind, named PotentialValue. If later in the game the puzzler notices that the value of the PotentialValue cell is incorrect, then this must be changed. In order to support new potential changes of the value, the player keeps changes history in a sequence of so named ChangedValue cells.

a. Using the UML, please construct a model containing the concepts needed to support the player. 2 pt

b. Using OCL please specify in the context of a cell an invariant constraining the value of the cell which is not empty, to be unique both on his row, column and region. 1 pt

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4



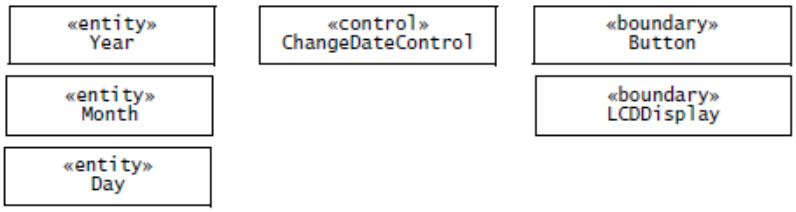

```
context Cell
    inv uniqueValue:
        if self.value.isUndefined
            then true
            else
                  (self.row.cell->reject(c:Cell     |     c.value.isUndefined)-
>select(c:Cell | c.value = self.value)->size = 1 and
                  self.column.cell->reject(c:Cell    |    c.value.isUndefined)-
>select(c:Cell | c.value = self.value)->size = 1 and
                  self.region.containedCells->reject(c:Cell |
c.value.isUndefined)->select(c:Cell | c.value = self.value)->size = 1)

        endif
```

II. Please explain what do you mean by: entity, boundary and control objects. In which model these objects are used? How are these represented in UML? 1pt

The analysis object model consists of **entity, boundary**, and **control** objects. **Entity objects** represent the persistent information tracked by the system. **Boundary objects** represent the interactions between the actors and the system. **Control objects** are in charge of realizing use cases. Modeling the system with entity, boundary, and control objects provides developers with simple heuristics to distinguish different, but related concepts.

To distinguish between different types of objects, UML provides the stereotype mechanism to enable the developer to attach such meta-information to modeling elements. For example, in figure above, we attach the «control» stereotype to the ChangeDateControl object. In addition to stereotypes, we may also use naming conventions for clarity and recommend distinguishing the three different types of objects on a syntactical basis: control objects may

have the suffix Control appended to their name; boundary objects may be named to clearly denote an interface feature (e.g., by including the suffix Form, Button, Display, or Boundary); entity objects usually do not have any suffix appended to their name. Another benefit of this naming convention is that the type of the class is represented even when the UML stereotype is not available, for example, when examining only the source code.



III. What is the purpose on modeling in Software Engineering? 0.5pt

To enable a better management of the complexity by supporting:

- an easier and better understanding of the problem,
- an easier communication among developers and clients, a clearer communication inside developers,
- reusing at a higher abstraction level because modeling languages are independent from different technologies,
- simulation of different parts of the system in order to validate some ideas and solutions, a.s.o.

IV. Please explain what do you mean by "Design by Contract" (shortly DBC)? Which are the assertions used in DBC? What are the advantages of DBC vs. Defensive Programming?. 1pt

As the name suggests, **DBC** is **a design methodology** considering that relationships between two software components are **very similar to relationships governing the world of business**, where the relationships are clearly stated in **contracts**. In each contract, both the rights and duties of the two parties: client and server are mentioned. In the software domain, client constraints (duties) are named **preconditions** and server constraints **postconditions**. DBC offers many advantages compared with defensive programming:

- the constraints are not restricted to the parameter values,
- all constraints are public, being located outside of the method body. So, the methods' body are smaller and by consequence easier to understand.
- pre & postconditions support the understanding of types because these are located in the interface
- pre & postconditions support testing.

V. From the point of view of granularity level, how many kinds of design to you know? Please describe shortly each of them. 1 pt

There are two types of design:

- **the system design**, concerning the design of the system architecture in large. The components of system design are: subsystems, partitions, layers. At this level there are some architectural styles: multi layer, client-server, repository, pipe&filter, MVC a.s.o
- **the object design**, concerning the design of the subsystem components inside the subsystem. At this levels we discuss about design patterns.

VI. Please analyze the code above and represent in UML the class diagram corresponding to the code. Please justify if the architecture contains a design pattern. If exists, name the design pattern and describe the kind of problems that this pattern solve and how. 2.5pt

```
public class NewPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaCode(subject);
      new OctalCode(subject);
      new BinaryCode(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
   }

public abstract class Represent {
   protected Subject subject;
   public abstract void update();
   }

   import java.util.ArrayList;
   import java.util.List;
   public class Subject {

      private List<Represent> represents = new ArrayList<Represent>();
      private int state;

      public int getState() {
         return state;
      }
      public void setState(int state) {
         this.state = state;
         notifyAllRepresents();
```

```
      }
      public void attach(Represent represent){
         represents.add(represent);
      }
      public void notifyAllRepresents(){
         for (Represent represent : represents) {
            represent.update();
         }
      }
   }

public class BinaryCode extends Represent{

   public BinaryCode(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString(
subject.getState() ) );
   }
   }

public class HexaCode extends Represent{

   public HexaCode(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }
   @Override
   public void update() {
      System.out.println( "Hex String: " + Integer.toHexString( subject.getState()
).toUpperCase() );
   }
   }

public class OctalCode extends Represent{
   public OctalCode(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }
   @Override
   public void update() {
     System.out.println( "Octal String: " + Integer.toOctalString( subject.getState()
) );
   }
   }
```

The observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern uses three classes. Represent, Subject and NewPatternDemo, the Client. Represent is an object having methods to attach and detach observers to a client object. We have created an abstract class *Represent* and a concrete class Subject that is extending the class *Represent*.

NewPatternDemo, our demo class, will use *Subject* and concrete class object to show observer pattern in action.