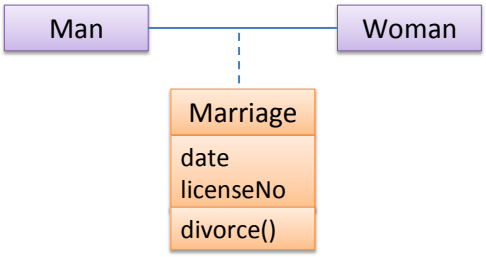
[L4P1]

Object-Oriented Programming: Intermediate Concepts

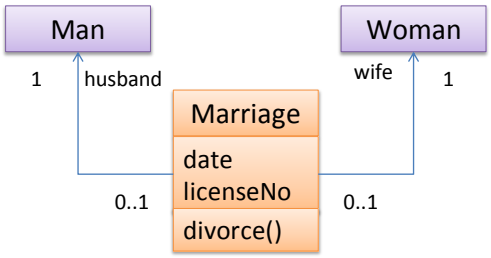
The analysis process for identifying objects and object classes is recognized as one of the most difficult areas of object-oriented development.
--Ian Sommerville, in the book 'Software Engineering'

Association classes

At times, there is a need to store additional information about an association. For example, a Man class and a Woman class linked with a 'married to' association might also require the date of marriage to be stored. However, that data is related to the association but not specifically owned by either the Man object or the Woman object. In such situations, an additional class can be introduced, e.g. a Marriage class, to store such information. These classes are called *association classes* and they are denoted as a connection to an association link using a dashed line as shown below.

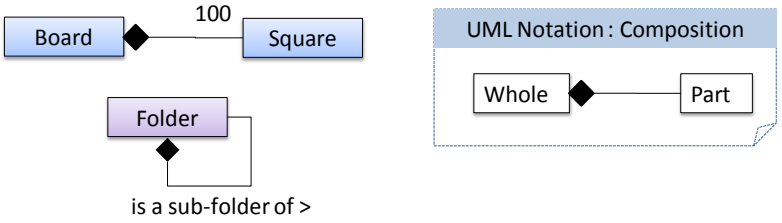


Note that while a special notation is used to indicate an association class, there is no special way to implement an association class. At implementation level, an association class is most likely implemented as follows.



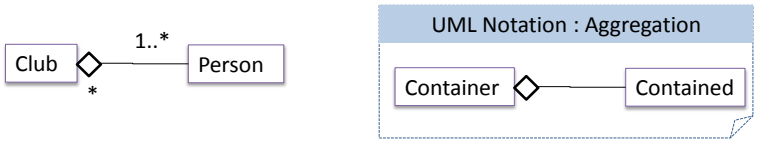
Composition and aggregation

A composition is an association that represents a strong "whole-part" relationship. When the "whole" is destroyed, "parts" are destroyed too. UML uses a solid diamond symbol to denote *composition*.



In addition, composition also implies that there cannot be cyclical links. In the example above, the notation represents a 'sub-folder' relationship between two folders while implying that a Folder cannot be a sub-folder of itself. If the diamond is removed, it is no longer a composition relationship and technically, allows a folder to be subfolder of itself.

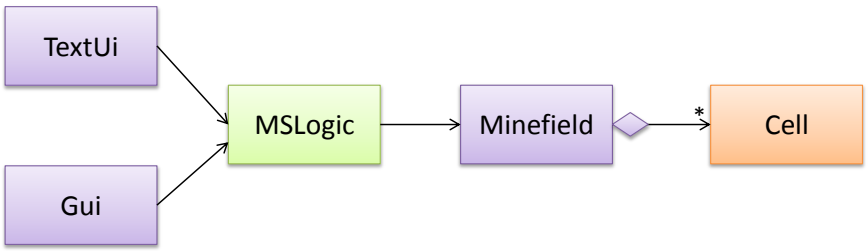
Aggregation represents a 'container-contained' relationship. It is a weaker relationship than composition. In UML, a hollow diamond is used to indicate an aggregation.



The distinction between composition and aggregation is rather blurred. Some practitioners (e.g., Martin Fowler, in his famous book *UML Distilled*) advocate not using the aggregation symbol altogether as using it adds more confusion than clarity.

Object interactions

Suppose we are planning to implement a simple minesweeper game that has a text based UI and a GUI. Given below is the OOP design we are considering for the application.



Before jumping into coding, we may want to decide if this class structure is able to produce the behavior we want. We also need to decide what methods each class need to have. To make those decisions, we need to analyze the how the objects of these classes will interact with each other. Both class diagrams can object diagrams we encountered in previous handouts can only depict the structure of an OOP design. To depict the behavior, we can use UML *sequences diagrams*.

> A UML sequence diagram *captures the interactions between multiple objects for a given scenario*.

The following is a sequence diagram (SD) that describes the interactions between the player (an actor) and the TextUi object. Note that newgame and clear x y represent commands typed by the Player on the TextUi.

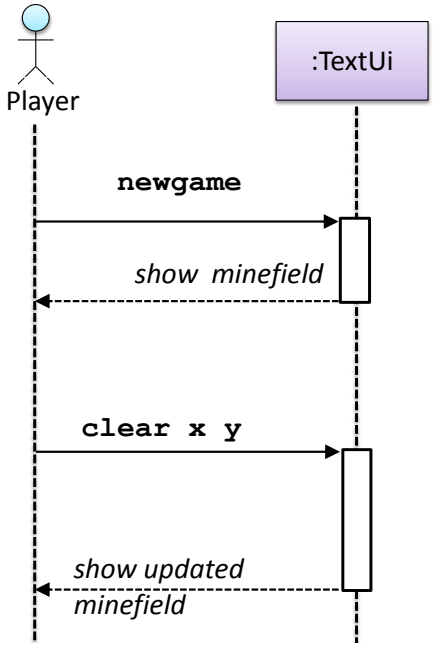


Figure 2. SD for newgame and clear

An explanation of the some basic elements of an SD is given below.

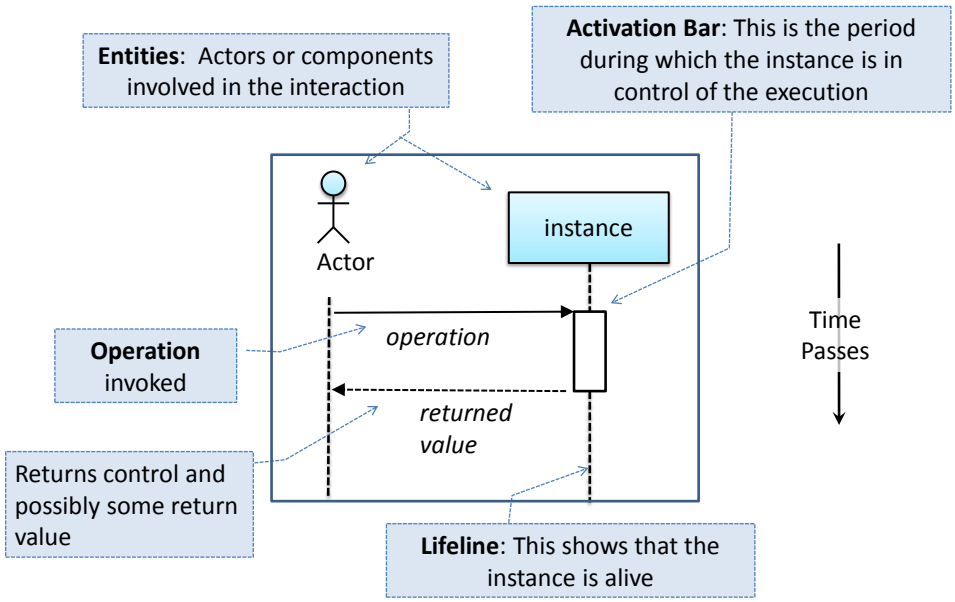


Figure 3. Basic elements of an SD

The notation :TextUi denotes 'an unnamed instance of the class TextUi'. If there were two instances of TextUi in the diagram, they can be distinguished by naming them as TextUi1:TextUi and TextUi2:TextUi.

In the diagram below, an SD shows the interaction between the player and the TextUi for the full game.

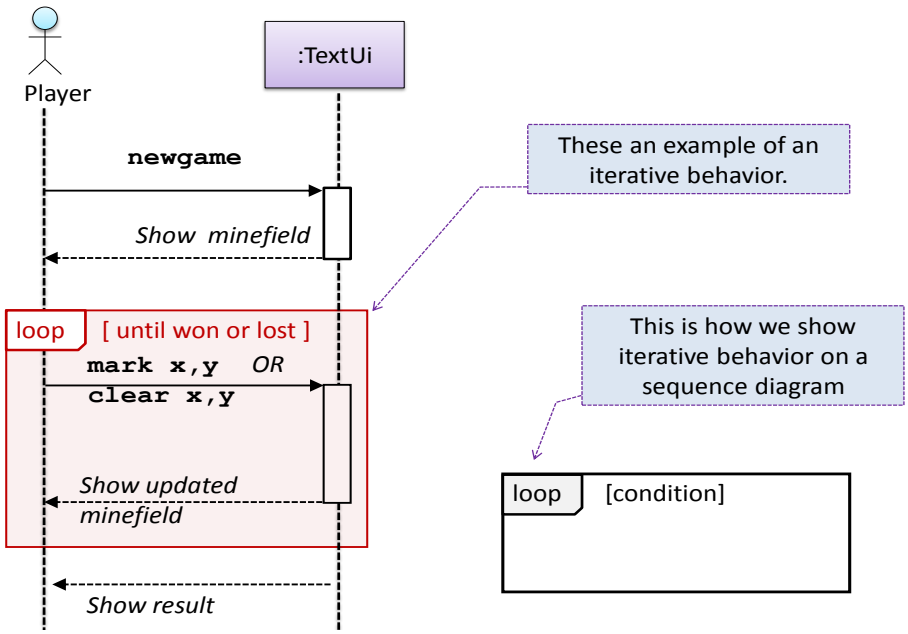


Figure 4. Showing loops in an SD

How does the TextUi object carry out the requests it has received from player? It would need to interact with other objects of the system. Since the MSLogic class is the one that controls the game logic, the TextUi needs to collaborate with MSLogic to fulfill the newgame request. This may be represented by extending the SD as shown below.

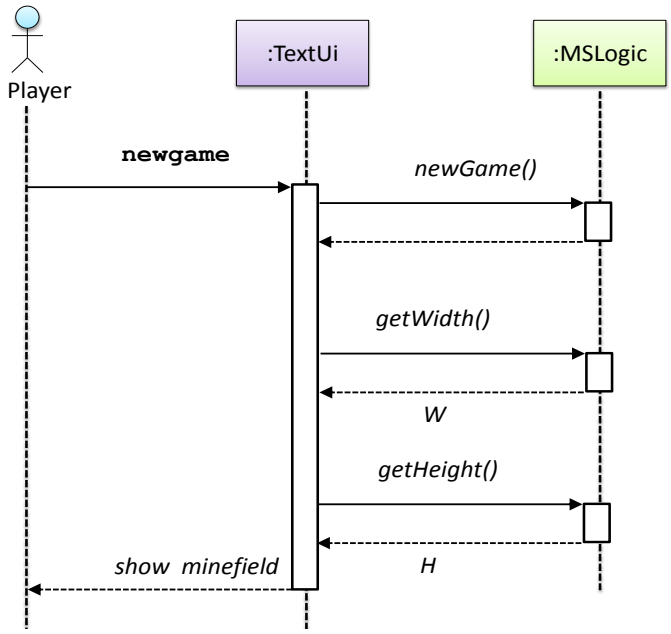


Figure 5. TextUi and MSLogic interactions for newgame

In the above diagram, it is assumed that W and H (Width and Height of the minefield, respectively) are the only information TextUi requires to display the minefield to the Player. Note that there could be other ways of doing this.

The MSLogic methods that showed up in this SD are:

- newGame():void
- getWidth():int

- getHeight():int

The next task focuses on the mark or clear operations performed until the game is won or lost.

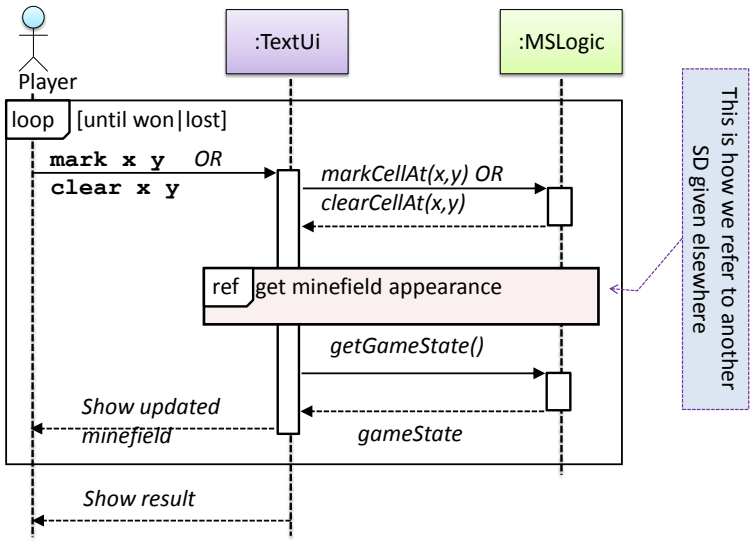


Figure 6. Using a ref frame in an SD

This interaction adds the following methods to the MSLogic class

- clearCellAt(int x, int y)
- markCellAt(int x, int y)
- getGameState() :GAME_STATE (GAME_STATE: READY, IN_PLAY, WON, LOST, …)

Note the use of a *ref frame* to allow a segment of the interaction to be omitted and detailed in another diagram. Given below is the SD that elaborates on the retrieval of cell appearance from MSLogic.

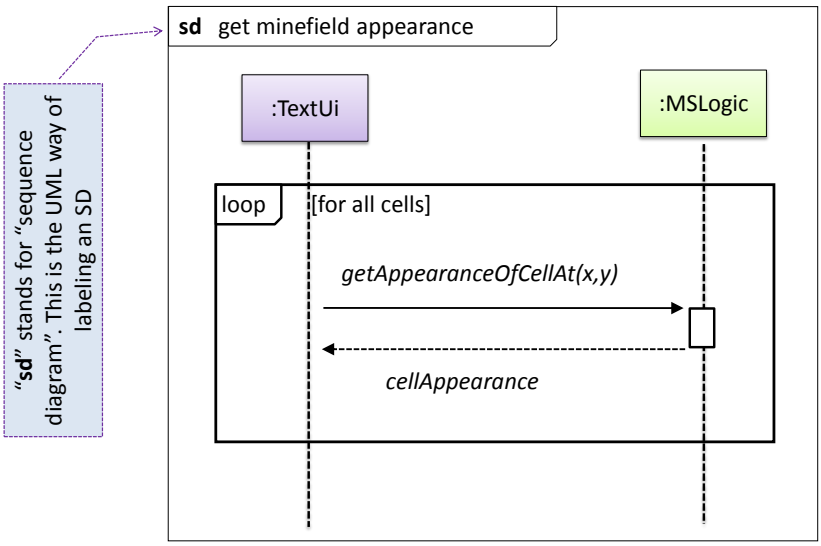


Figure 7. The SD for the ref frame

Correspondingly, the following operation gets added to MSLogic API:

- getAppearanceOfCellAt(int,int):CELL_APPEARANCE (CELL_APPEARANCE: HIDDEN, ZERO, ONE, TWO, THREE, …, MARKED, INCORRECTLY_MARKED, INCORRECTLY_CLEARED)

In the above design, TextUi does not access Cell objects directly. Instead, it gets values of type CELL_APPEARANCE from MSLogic to be displayed as a minefield to the player. Alternatively, each cell or the entire Minefield can be passed directly to TextUi.

To reduce clutter, activation bars and return arrows may be omitted if they do not result in ambiguities or loss in information. Informal operation descriptions such as those given in the SD below can be used, if the purpose is to brainstorm and not to specify the API.

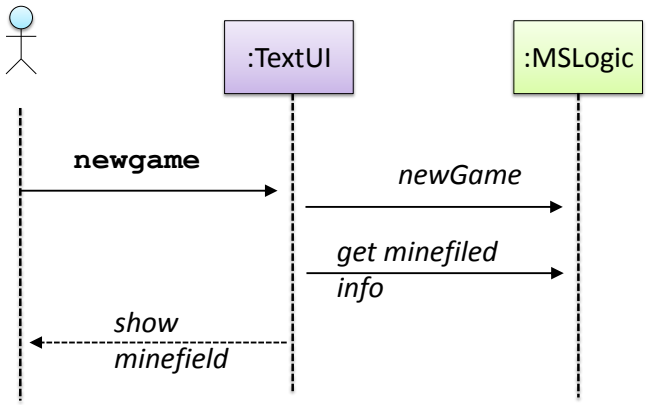


Figure 8. A 'no frills' SD

Here is the full list of MSLogic methods discovered thus far:

newGame(): void

getWidth():int

getHeight():int

clearCellAt(int x, int y)

markCellAt(int x, int y)

getGameState() :GAME_STATE

getAppearanceOfCellAt(int x, int y): CELL_APPEARANCE

The above is for the case when Actor Player interacts with the system using a text UI. Additional operations (if any) required for the GUI can be discovered similarly.

More details can be included to increase the precision of the method definitions before coding. Such precision is important to avoid misunderstandings between the developer of the class and developers of other classes that interact with this class.

Operation: *newGame(): void*
Description: Generates a new *WxH* minefield with *M* mines. Any existing minefield will be overwritten.
Preconditions: none.
Postconditions: A new minefield is created. Game state is READY.

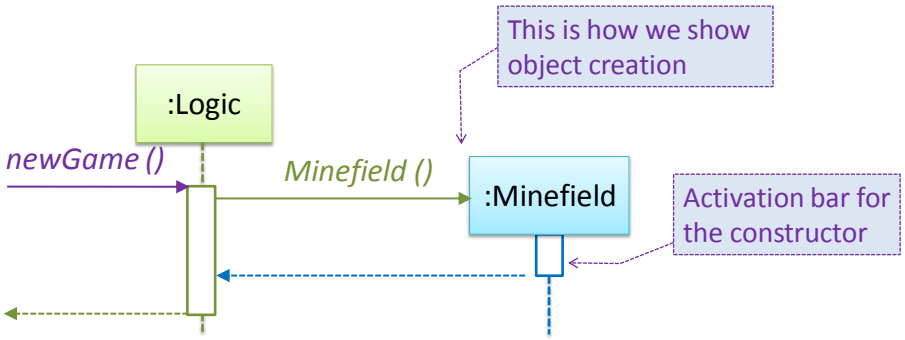
Preconditions are the conditions that must be true before calling this operation. Postconditions describe the system *after* the operation is complete. Note that post conditions do not say what happens *during* the operation. Here is another example:

Operation: *clearCellAt(int x, int y): void*
Description: Records the cell at x,y as cleared.

Parameters: x, y coordinates of the cell
Preconditions: game state is READY or IN_PLAY. x and y are in 0..(H-1) and 0..(W-1), respectively.
Postconditions:
Cell at x,y changes state to ZERO, ONE, TWO, THREE, ..., EIGHT, or INCORRECTLY_CLEARED
Game state changes to IN_PLAY, WON or LOST as appropriate.

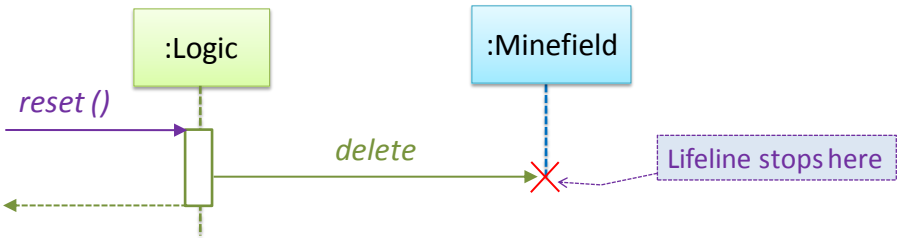
Other operations can be specified similarly. Note that preconditions and postconditions can be described as part of the operation 'description' instead of stating them separately. That is the approach taken by Java method descriptions.

Now, let us look at what other objects and interactions are needed to support the newGame() operation. It is likely that a new Minefield object is created when the newGame() method is called, as depicted in the SD below.



Note that the behavior of the Minefield constructor has been abstracted away. It can be designed at a later stage.

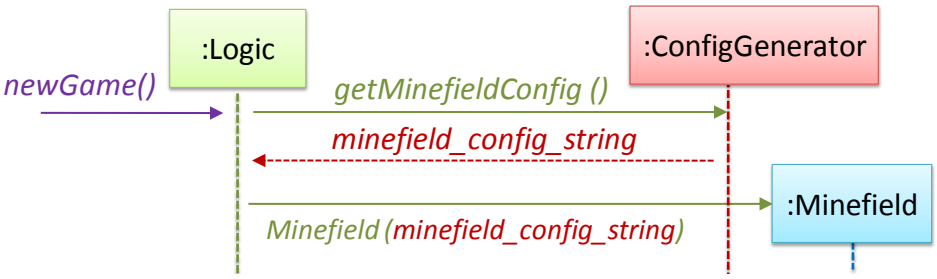
To illustrate object deletion in an SD, suppose MSLogic supports a reset() operation.



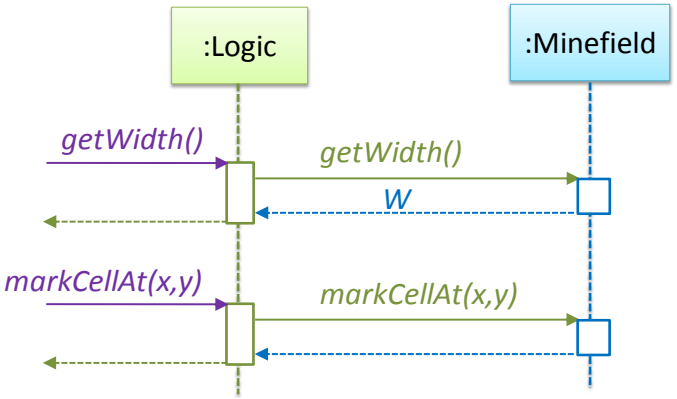
In languages such as Java that supports automatic memory management, although object deletion is not that important, the above notation can still be used to show the point at which the object ceased to be used.

Moreover, the diagram below assumes that the Minefield object has enough information (i.e. H, W, and mine locations) to create itself.

An alternative is to have a ConfigGenerator object that generates a string containing the minefield information as shown below.



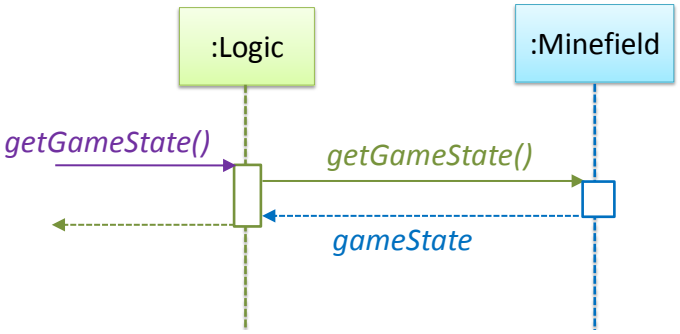
In addition, getWidth(), getHeight(), markCellAt(x,y) and clearCellAt(x,y) can be handled like this.



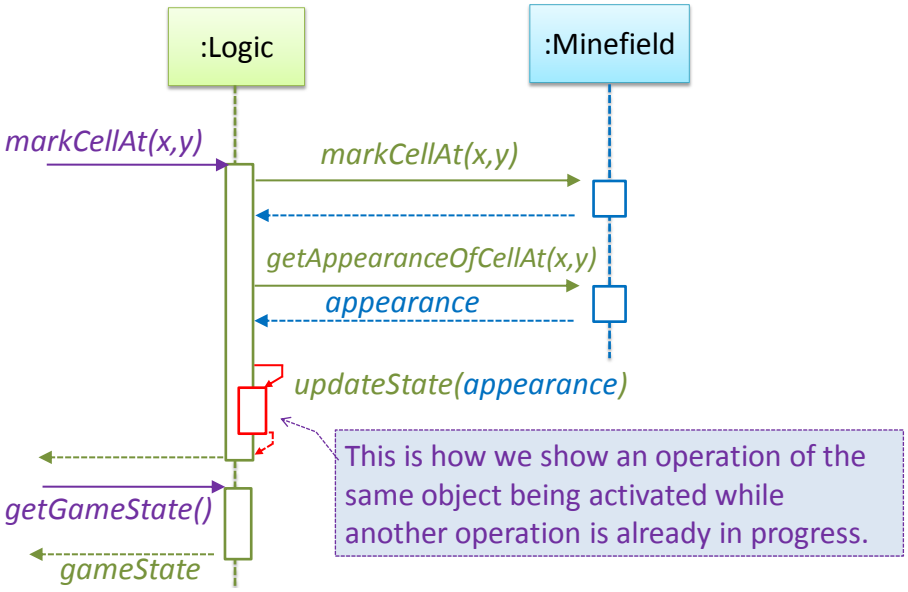
How is getGameState() operation supported? Given below are two ways (there could be other ways):

1. Minefield class knows the state of the game at any time. Logic class retrieves it from the Minefield class as and when required.
2. Logic class maintains the state of the game at all times.

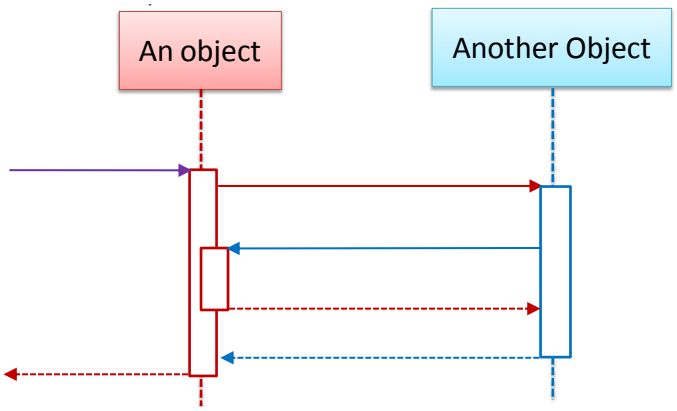
Here's the SD for option 1.



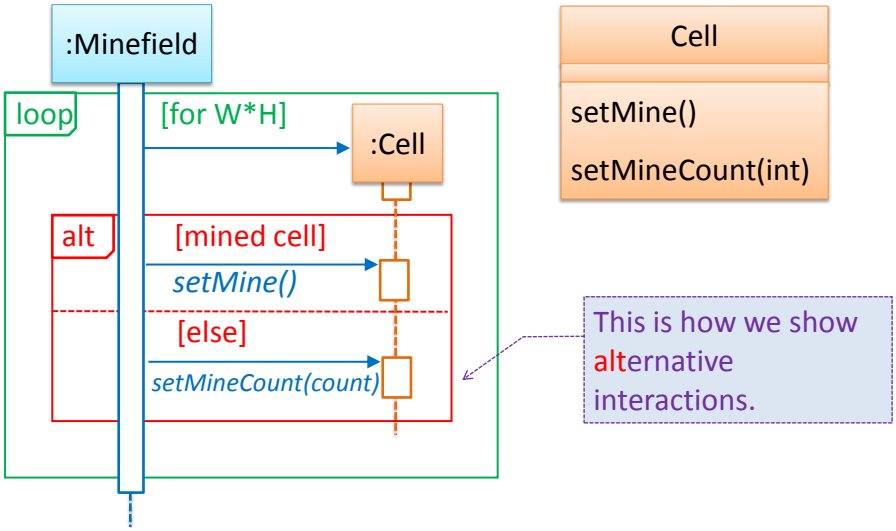
Here's the SD for option 2. Here, assume that the game state is updated after every mark/clear action. Note the use of a nested activation bar.



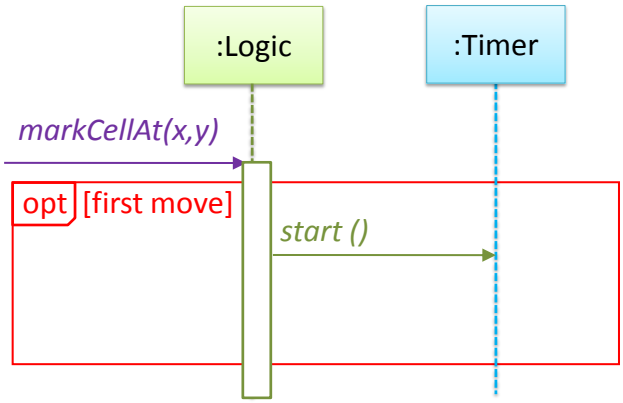
Here is another example showing the use of a nested activation bar.



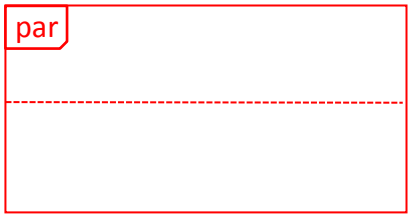
It is now time to explore what happens inside the Minefield constructor? One way is to design it as follows.



To illustrate optional interactions using the 'opt' frame, assume that Minesweeper supports the 'timing' feature.



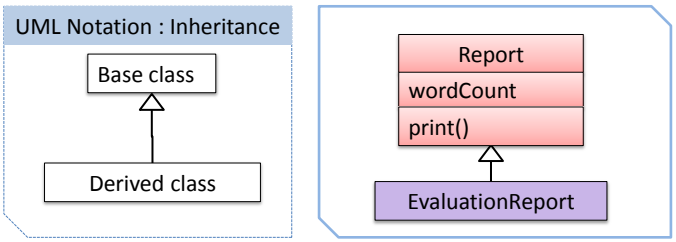
Similarly, parallel behavior can be shown using a 'par' frame ('par' frames are not examinable).



When designing components, it is not necessary to draw elaborate sequence diagrams for all interactions among objects. They can be done as rough sketches. Draw sequence diagrams only when you are not sure which operations are required by each class, or when you want to verify that your class structure can indeed support the required operations.

Inheritance

Sometimes, it helps to be able to define a new class based on another class. For example, to be able to define an EvaluationReport class based on a Report class so that the EvaluationReport class does not have to duplicate code that is already implemented in the Report class. This can be achieved using the object-oriented concept of *inheritance* .

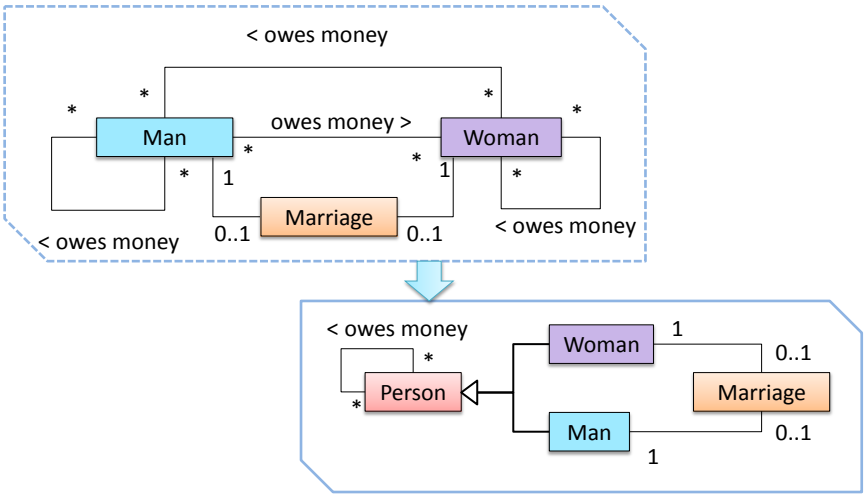


In the example above, The EvaluationReport inherits the wordCount variable and the print() method from the base class Report.

- Other names for Base class: *Parent class*, *Super class*
- Other names for Derived class: *Extended class*, *Child class*, *Sub class*

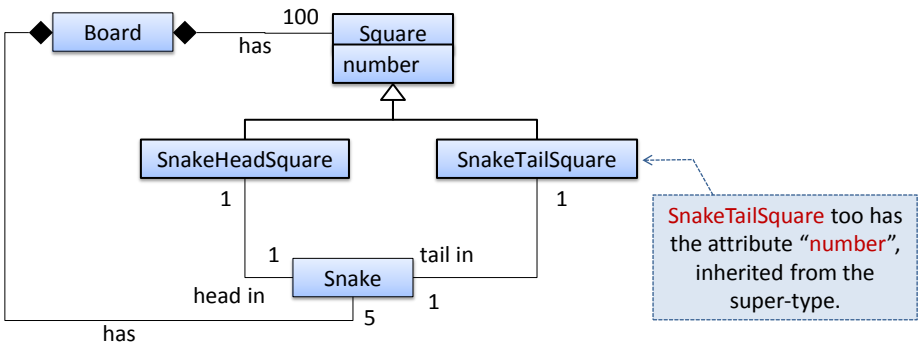
Applying inheritance concept can result in the common parts among classes being extracted into more general classes. In the example below, Man and Woman behaves the same way for the 'owes money' association. However, the two classes cannot be simply replaced with a more general class Person because of the need to distinguish between Man and Woman for the

'marriage' association. A solution is to add the Person class as a super class and let Man and Woman inherit from Person.



Inheritance implies the derived class can be considered as a *sub-type* of the base class (and the base class is a *super-type* of the derived class), resulting in an 'is a' relationship[2]; for example, in the class diagrams of a *Snakes and Ladders* board game given below:

- SnakeHeadSquare *is a* Square.
- SnakeTailSquare *is a* Square.



Substitutability

Every instance of a subclass is an instance of the superclass, but not vice-versa. For example, an Academic is an instance of a Staff, but a Staff is not necessarily an instance of an Academic. As a result, inheritance allows *substitutability*. i.e. wherever an object of the superclass is expected, it can substituted by an object of any of its subclasses. For example, the following code is valid because a SnakeHeadSquare object is substitutable as a Square object.

```
Square square = new SnakeHeadSquare();          // OK
```

But the following code is not because a square is declared as a Square type (see above) and therefore its value may or may not be of type SnakeHeadSquare, which is the type expected by variable snakeSquare .

```
SnakeHeadSquare snakeSquare = square;           //Compile error
```

[2] Inheritance does not necessarily mean a sub-type relationship exists. However, the two often go hand-in-hand. In this handout, we assume inheritance implies a sub-type relationship.

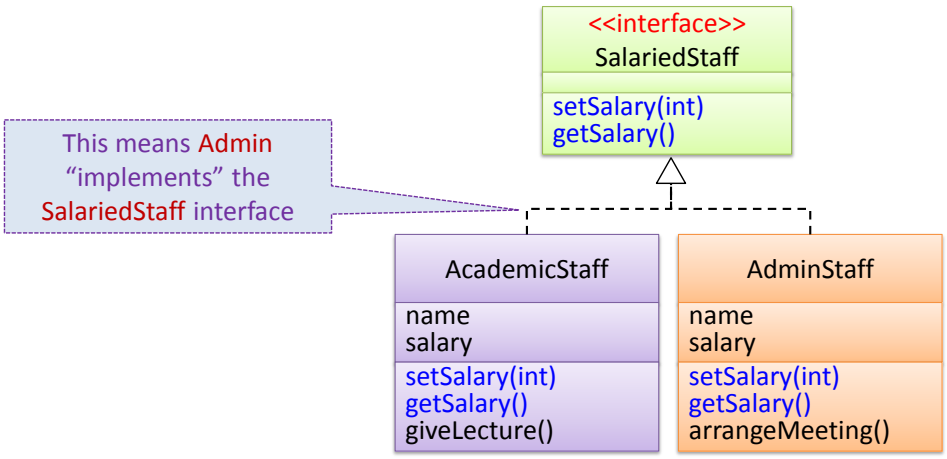
Interfaces

In an OO solution, an *interface* is a behavior specification. In Java, a class can be explicitly declared as an interface which can then be used to specify the available operations without implementing any. As shown in the code below.

```java
/**
 * Represents an Employee who is paid
 * a salary.
 */
public interface SalariedStaff {
    void setSalary(int newSalary);
    int getSalary();
}
```

```java
/**
 * Represents a staff member holding
 * an admin position.
 */
public class AdminStaff implements SalariedStaff {
    private int salary;

    @Override
    public void setSalary(int newSalary) {
        this.salary = newSalary;
    }

    @Override
    public int getSalary() {
        return salary;
    }
}
```

In UML, an interface is depicted with the keyword <<interface>>.



If a class implements all operations specified in an interface, it is said that the class 'implements' the interface. Another term for this is *interface inheritance*. In UML, the relationship is shown with a dashed line and a triangle similar to the one used for inheritance relationship. Similar to the class inheritance explained earlier, interface inheritance too results in an *is-a* relationship.

Interface Segregation Principle

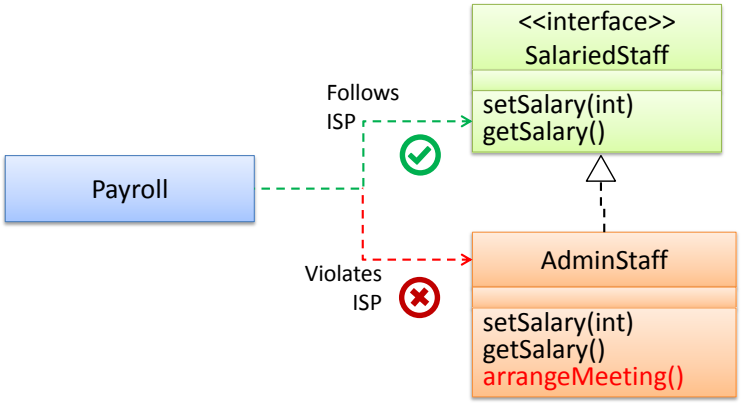
The *integration Segregation Principle (ISP)* states:

> *No client should be forced to depend on methods it does not use.*

For example, the Payroll class should not depend on the AdminStaff class because it does not use the arrangeMeeting() method. Instead, it should depend on the SalariedStaff interface.

```
public class Payroll {
    //...
    private void adjustSalaries(AdminStaff adminStaff){ //violates ISP
        //...
    }

}
```

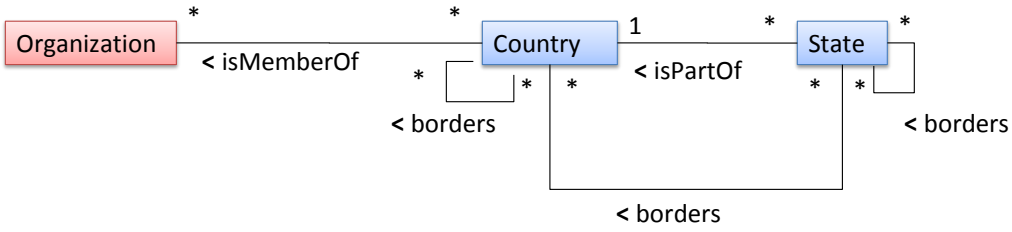
```
public class Payroll {
    //...

    private void adjustSalaries(SalariedStaff staff){ //does not violate ISP
        //...
    }
}
```



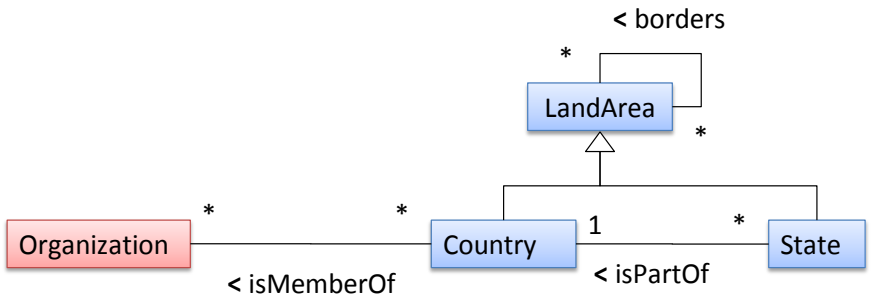
Worked examples

[Q1]

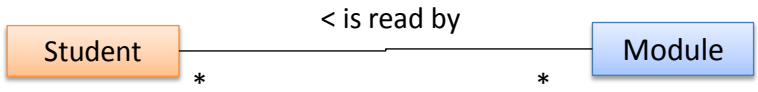
Refine the class diagram model (given in L5P2 handout) so that it uses inheritance.



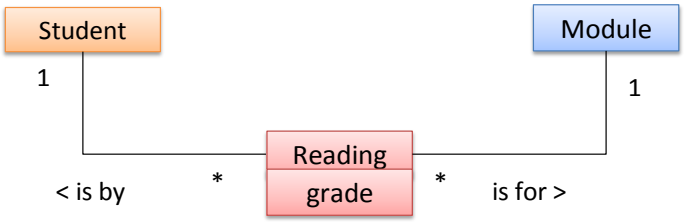
[A1]



[Q2]
Modify the class diagram below so that we can keep track of the grade a Student earned every time he/she reads a module. Here, 'reads' means taking the module.



[A2]
It appears that now we are required to store some data (i.e. grade) about an association (i.e. 'is read by'). Note that storing the grade inside the Student or the Module is not appropriate as the data is about a particular association between the two objects. Besides, the Student can read a module multiple times. To tackle this situation, we can introduce a class called Reading to represent the association.



Based on the new class diagram, a `Reading` object represents exactly one student reading exactly one Module and the grade he/she obtained for the Module in that reading. A module can have any number of `Reading` objects associated with it and a Student can have any number of Reading objects. Furthermore, a Student can have multiple Reading objects for the same Module.

Note: Reading class can be shown as an association class too.

[Q3]
Create an OO class diagram to represent the following description:

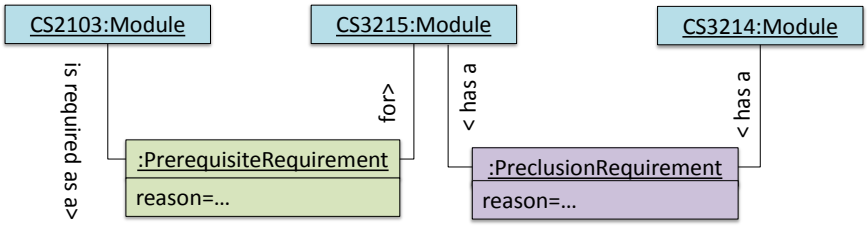
> Some modules require other prerequisite modules to be taken first. If two modules cover nearly same material, taking one of them would preclude a student from taking the other. Such modules are said to be mutually exclusive. The reasons or descriptions for preclusion as well as for the prerequisite also need to be captured.

[A3]

We cannot represent a prerequisite as a mere association between two modules because there is some data (i.e. the reason for the prerequisite) we have to store about the association. Therefore, we introduce a class to represent a prerequisite. Note that the PrerequisiteRequirement class does not represent a module that is a prerequisite for another module. It simply represents the fact/requirement/constraint that a certain module is a prerequisite for a certain other module. A similar reasoning is behind the PreclusionRequirement class.

A PrerequisiteRequirement has two associations with the Module class because the roles played by each of the two Module objects connected to a PrerequisiteRequirement object are different (one is the module that requires the other module as the prerequisite). In the case of the PreclusionRequirement, both Module objects play the same role (either one is a preclusion with respect to the other), letting us show it as one association with multiplicity of 2.

In the object diagram below (for your info only, not required by the question), CS2103 is a prerequisite for CS3215 while CS3215 is a preclusion for CS3214 (and vice versa).



[Q4]

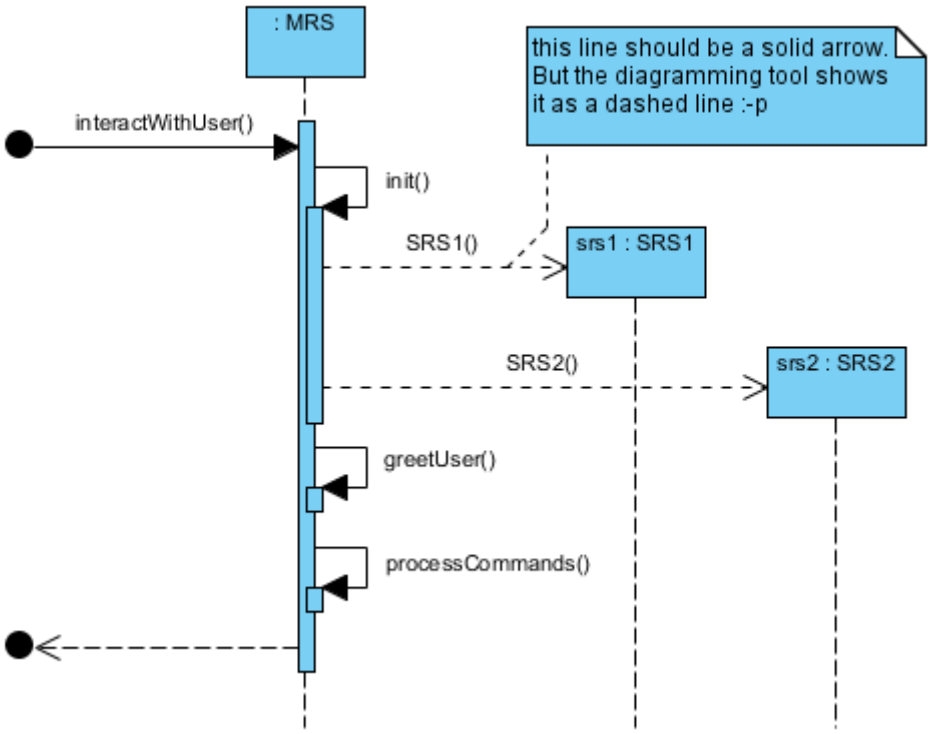
(a) Draw the corresponding sequence diagram for the interactions resulting from the highlighted statement below.

(b) Draw a class diagram for the code shown in the previous question. Show associations, dependencies, navigabilities, and multiplicities.

```
class MRS {
        private int BUFFER_SIZE = Config.BS;
        //BS is a class-level constant of the Config class
        private SRS1 srs1 = null;
        private SRS2 srs2 = null;

        public static void main(String[] args){
                MRS mrs = new MRS();
                mrs.interactWithUser();
        }
        private void interactWithUser(){
                init();
                greetUser();
                processCommands();
```

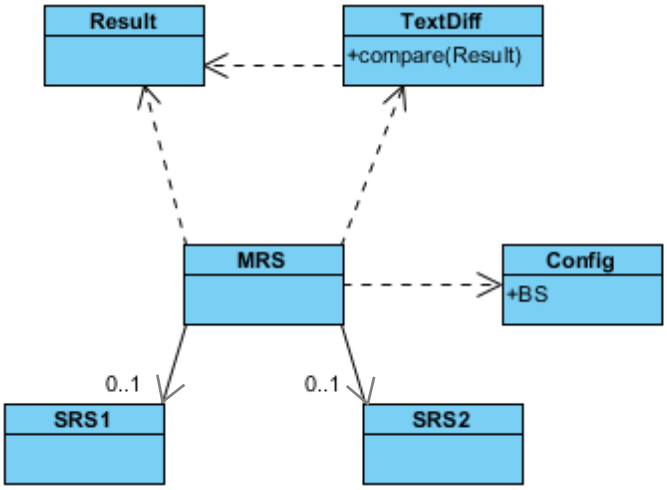
```
        }
        private void init(){
                srs1 = new SRS1();
                srs2 = new SRS2();
        }

        // TextDiff and Result are part of the solution (they are not library classes).
        private boolean compareResults(Result[] results){
                TextDiff diff = new TextDiff();
                return diff.compare(results);
        }
    }
```

[A4]
(a)



Note slight variations in the notation. Unfortunately, no two UML tools draw diagrams exactly alike and apparently, none follow the UML standard precisely. It is better for you to get used to such slight variations.

(b)

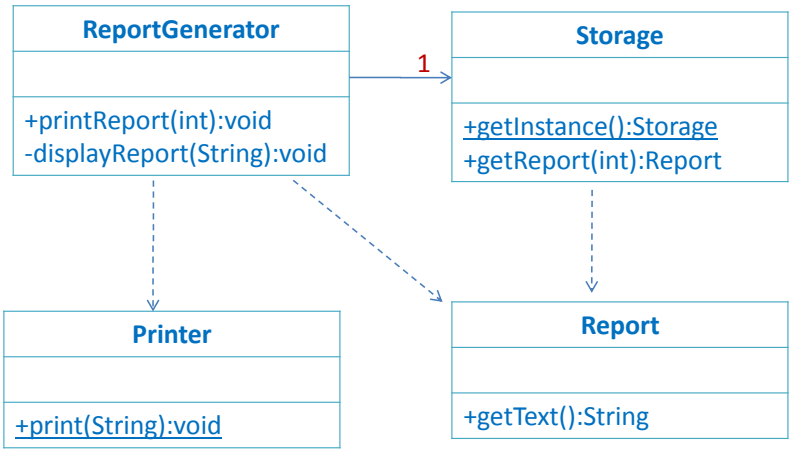


Notes: Result is simply a parameter to a method of MRS. It does not indicate a structural relationship between MRS and Result. As far as we can see from the given code, it is merely a dependency. Similarly, TextDiff is just a local variable inside a method, not a structural relationship.

[Q5]
Consider the code given below. Draw a class diagram to match the code. Include attributes, operations, visibilities, navigabilities, multiplicities, and dependencies.

```
class ReportGenerator{
        //getInstance always returns a valid Storage object.
        private Storage storage = Storage.getInstance();
        public static void main(){
                ReportGenerator generator = new ReportGenerator();
                generator.printReport(342);
        }
        public void printReport(int reportID){
                //getReport(int) returns a Report object.
                displayReport ( storage.getReport(reportID).getText() );
        }
        private void displayReport(String s){
                //call the static method print() of Printer class
                Printer.print(s);
        }
}
```

[A5]
Note how only the link between ReportGenerator and Storage is an association. The other links are merely dependencies because they do not represent structural links between classes. The multiplicity 1 represents the fact that a ReportGenerator will always be linked to exactly one Storage object.



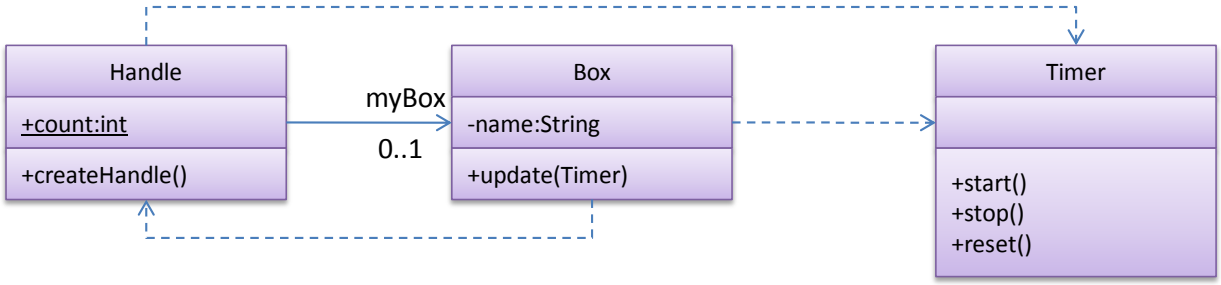
[Q6]

Draw the corresponding class diagram for the following code. Include navigabilities, visibilities, dependencies, multiplicities, attributes, association role names, and operations.

```
class Handle {
        public static int count;
        private Box myBox;

        public void createHandle() {
                Timer timer = new Timer();
                timer.start();
                myBox = new Box();
                myBox.update(timer);
                timer.stop();
        }
}
```

```
class Box {
        private String name;

        public void update(Timer timer) {
                timer.reset();
                Handle.count++;
        }
}
```

[A6]



Points to note:

- Associations are structural links between objects. An association link should be supported by an instance-level variable that holds the link. Box accessing a static variable in Handle class simply means a dependency, not an association link. Local variables (e.g. timer variable inside createHandle method) and parameters (e.g. Timer parameter in the update method) too indicate dependencies.
- An association link already implies a dependency; there is no need to add a dependency link from Handle to Box.

-- End of handout --