[Handout for L6]
Designing component internals using OO

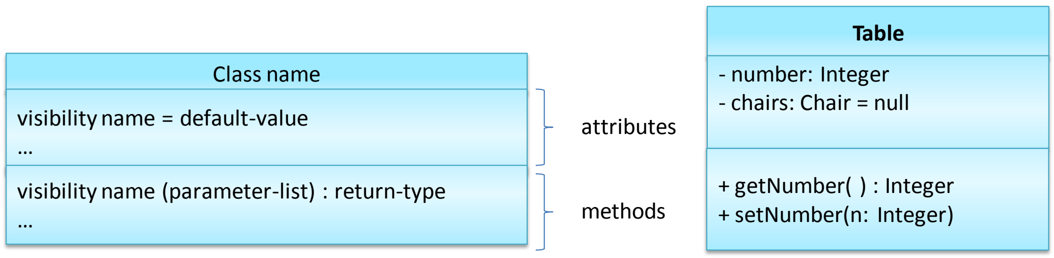
Introduction to Object-Orientation

Instead of writing our own handout, we refer you to read (compulsory reading) pages 11-20 of the document *Object-Oriented Programming with Objective-C* released by Apple Inc. In spite of the title, the document is mostly programming language independent. When reading it you may ignore any references to Objective C. The document is available online at http://tinyurl.com/oopwithobjc (both HTML and pdf formats).

Creating an OO solution

Class diagrams

A UML *class diagram* describes the structure (i.e. not behavior) of an OO solution. They are possibly the most used diagram type in the industry and an indispensible tool for an OO programmer.



The diagram above gives the basic notation to show a class in a class diagram. Attributes represent the data of the class.

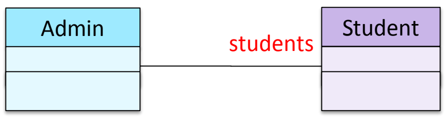
Visibility

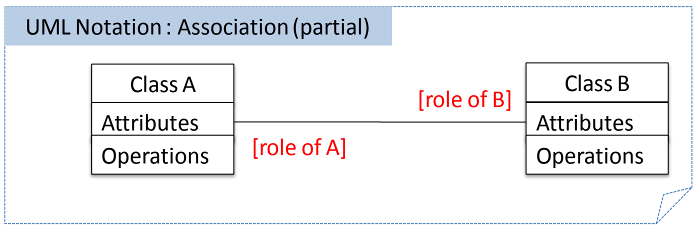
We can also indicate the *visibility* of attributes and operations. That is, level of access allowed for each attribute or operation. Names used for visibilities and their exact meaning depends on the programming language used. Here are some common visibilities and how they are indicated in a class diagram:

+ public   - private   # protected   ~ package

Associations

Classes do not exist in isolation. They are connected to other classes that represents how objects collaborate with each others in our OO solution. These relationships between classes are called associations. A UML class diagram shows not only the classes, but also how they are connected to each other. We indicate an associations as a line between two classes. For example, the Admin objects need to have a link to Student objects, so as to access information, and we show that association as follows.

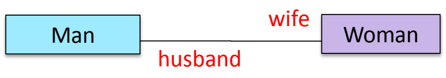




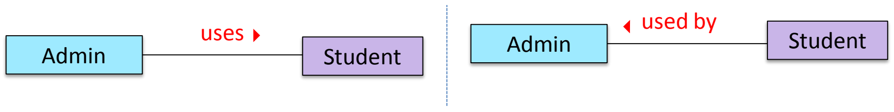
Note that an association line in a class diagram denotes a permanent or semi-permanent structural link between objects of two classes. We don't use association links to show temporary contacts between objects.

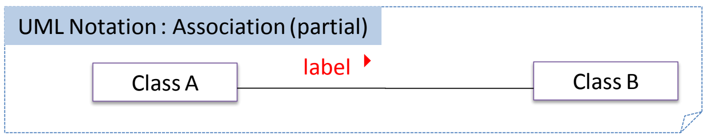
It is OK to omit the 'Operations' compartment or both 'Attributes' and 'Operations' compartments if showing those details are not important for the task at hand. For simplicity, we omit those two compartments from most of the subsequent class diagrams.

Optionally, we can use 'role' labels to indicate the role played by the class in the relationship. For example, the association given below represents a marriage between a Man object and a Woman object, in which the roles played by objects of the two classes are 'husband' and 'wife', respectively.



We can use an *association label* to describe that the association is. We read the diagram below as "an Admin object uses Student objects" or "Student objects are used by an Admin object". The arrow head tells us which direction we should read the label.



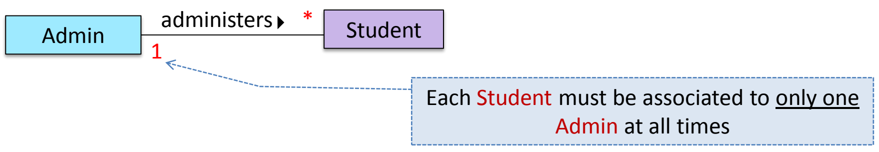


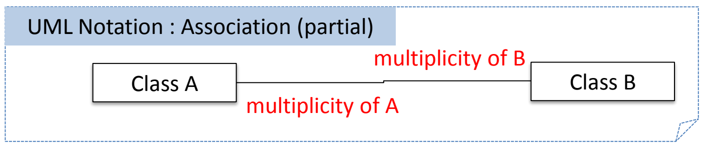
A class diagram can also indicate the *multiplicity* of an association. Multiplicity is the number of objects of a class that participate in the association.

Commonly used multiplicities:

- 0..1 : optional, can be linked to 0 or 1 objects
- 1 : compulsory, must be linked to one object at all times.
- * : can be linked to 0 or more objects.
- n..m : the number of linked objects must be n to m inclusive

In the diagram below, an Admin object administers (in charge of) any number of students but a Student object must always be under the charge of exactly one Admin object.

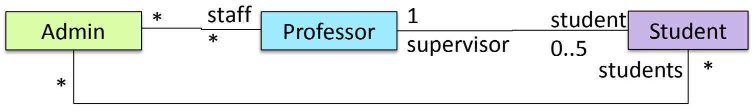

Admin
administers ▸
*
1
Student
Each Student must be associated to only one Admin at all times



Next, let us assume we have the following additional details:

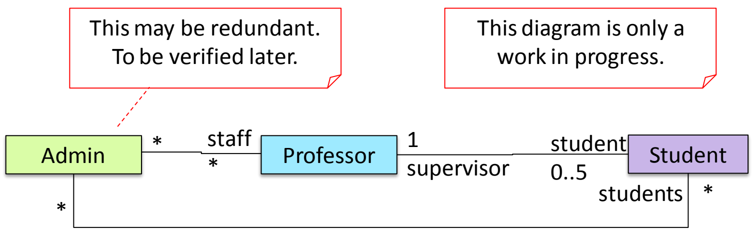
- Each Student must be supervised by a Professor.
- Students have matriculation numbers. A Professor cannot supervise more than 5 students.
- All personnel have names.
- Admin staff handles Professors as well.

Here's the updated class diagam:

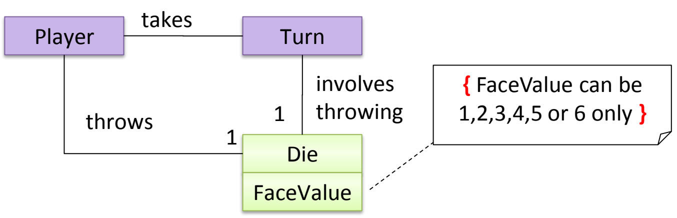


UML notes and constraints

We can use UML notes to add additional information to a UML diagram. These notes can be connected to a particular element in the diagram or can be shown without such a connection. The diagram below shows examples of both.

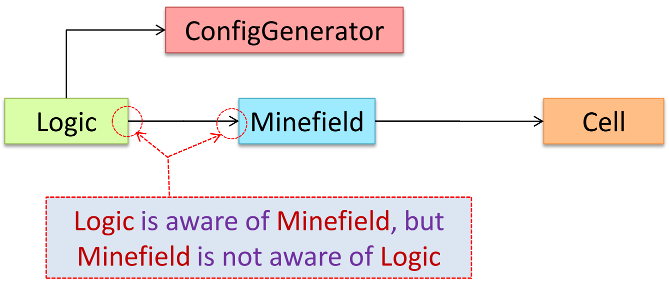

This may be redundant. To be verified later.
This diagram is only a work in progress.
Admin
*
staff
*
Professor
1
supervisor
student
0..5
Student
students
*
*

A *constraint* can be stated inside a note. A constraint is specified within curly brackets. We can use natural language or a formal notation such as OCL (Object Constraint Language) to specify constraints. OCL is beyond the scope of this handout.

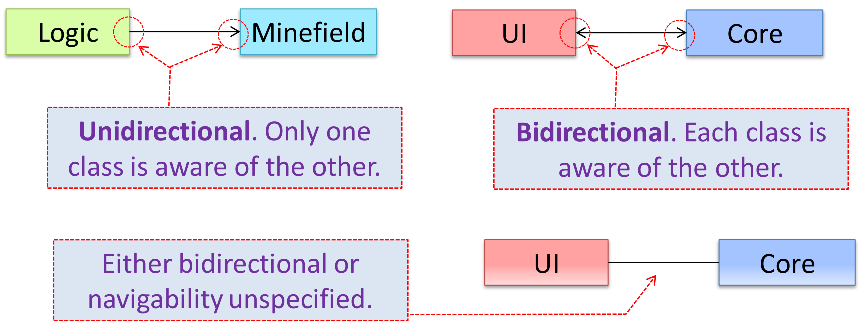


Navigability

Navigability is another detail we can show in class diagrams. Navigability (shown as an arrowhead in the association link) indicates whether a class involved in an association is "aware" of the other participating class.

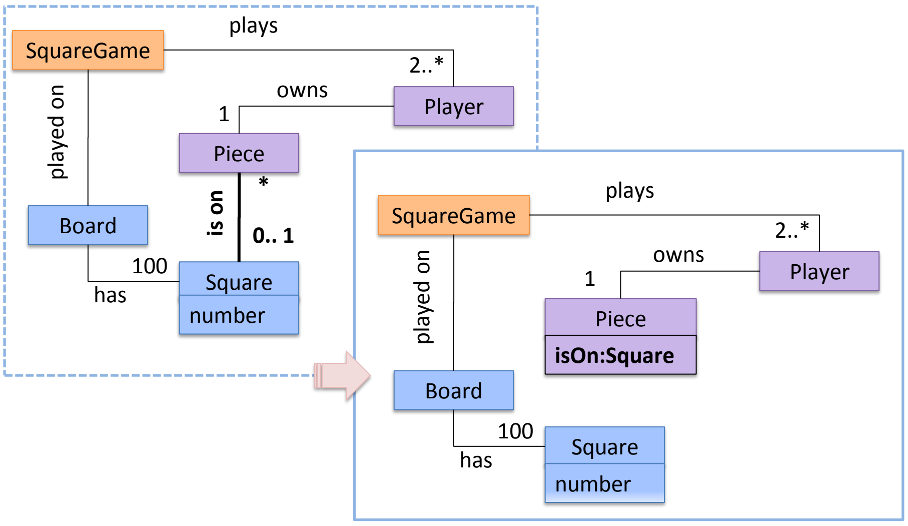

ConfigGenerator
Logic
Minefield
Cell
Logic is aware of Minefield, but Minefield is not aware of Logic

At code level, navigability implies which type of object is holding a reference to which type of object. In the example above, a Logic object will hold a reference to a ConfigGenerator object, but not the other way around. Navigability can be *unidirectional* or *bidirectional*. For convenience, we can use a line without arrow heads to indicate a bidirectional link. However, if none of the associations in a class diagram has navigability arrows, it usually means navigability is "unspecified".


Logic
Minefield
Unidirectional. Only one class is aware of the other.
UI
Core
Bidirectional. Each class is aware of the other.
Either bidirectional or navigability unspecified.
UI
Core

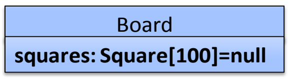
Associations as attributes

We sometimes use an attribute to represent an association, as shown in the diagram given next.

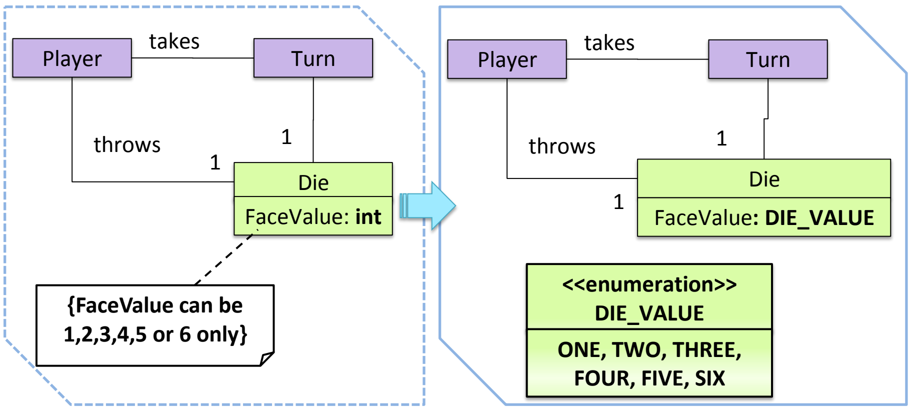


In such a situation, we can show the multiplicity of the association as part of the attribute.

name: type [multiplicity] = default value

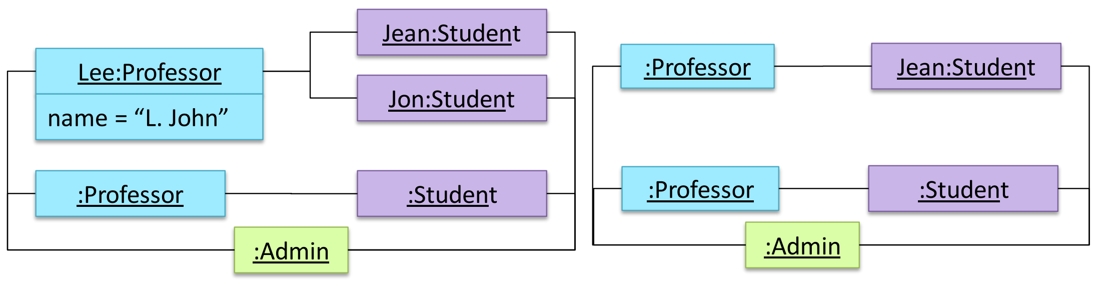


Enumerations
We can also use the <<enumeration>> to indicate a fixed set of values that an attribute can have.



Object diagrams

Sometimes it is useful to show objects instead of classes. Those are called *object diagrams*. An object diagram shows an object structure at a given point of time while a class diagram represents the general situation. Given below is an example.



Note how object names are shown differ from how class names are shown. First, object names are underlined. Second, it is possible to give each object an 'instance name' in addition to the class name, in the format 'instance name: class name'. For example, Jean:Student means Jean is the object instance name while Student is the class name. Also note how we have omitted association roles, most attributes, and some object names. It is ok to omit information when they do not add value to the diagram. It also does not make sense to show methods in an object diagrams.

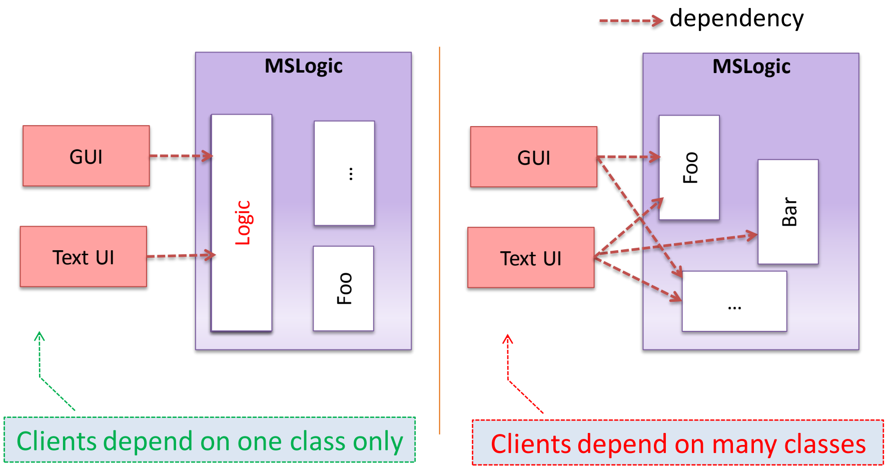
Also note that both object diagrams come from the same class diagram. That is, each object diagram shows 'an instance of' the class diagram.

Discovering class structure and behavior

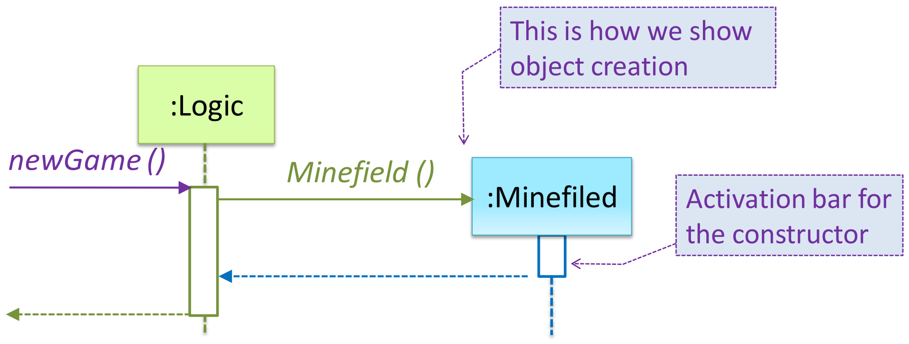
The analysis process for identifying objects and object classes is recognized as one of the most difficult areas of object-oriented development.
--Ian Sommerville, in Software Engineering

Since we already know the API (i.e. external behavior) of the component, now it is a matter of figuring out what smaller components should be inside it and how should they interact with each other to support that API. That is, we want to figure out the internal structure and behavior of that component that can support the external behavior expected from it. By applying OO thinking, we can roughly figure out what classes are required and how they relate to each other. Then, similar to how we discovered component APIs, we can use sequence diagrams to discover APIs of the classes, while refining the class structure as we go.

For example, let us take the MSLogic component. What internal behavior is required to support the MSLogic API? First, let us assume that MSLogic API is implemented in a single class called Logic (as shown on the left below) instead of spreading it among many classes. This approach (shown on the left of the diagram below) simplifies the interactions between MSLogic component and clients of MSLogic such as the GUI.

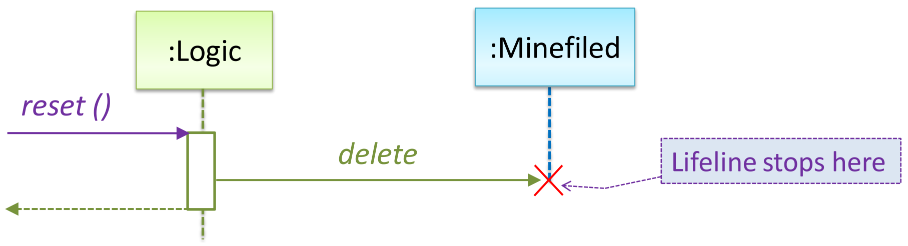


Now, what internal behavior is needed to support the newGame() operation? It is likely that we create a new Minefield object when the newGame() operation is called. This is how we show that in an SD.

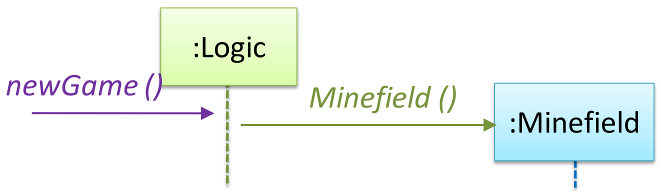

This is how we show object creation
:Logic
newGame ()
Minefield ()
:Minefiled
Activation bar for the constructor

Also note that we have abstracted away the behavior of the Minefield constructor. We plan to design it at a later stage.

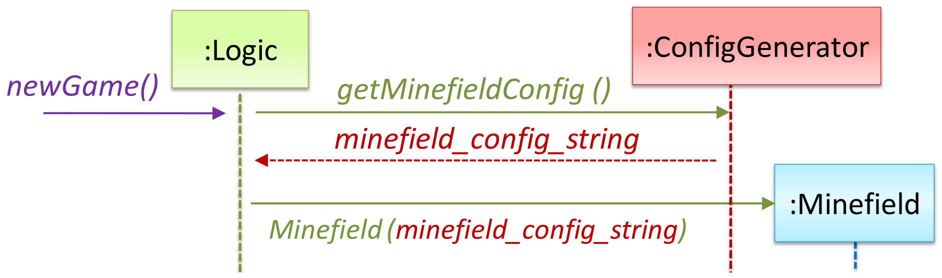
To illustrate how we show object deletion in an SD, let us assume MSLogic also supports a reset() operation.



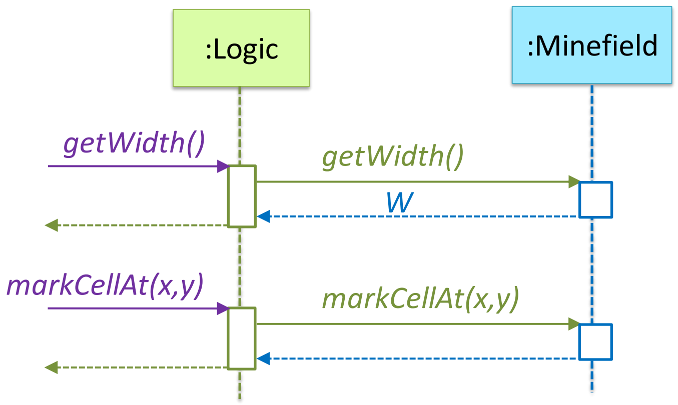
Note that in languages such as Java that supports automatic memory management, object deletion is not that important. However, you can still use the above notation to show at which point the object stops being used.



The above diagram assumes that the Minefiled object has enough information to create itself (i.e. H, W, and mine locations). An alternative is to have a ConfigGenerator object that generates a string containing the minefield information, as shown below.



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

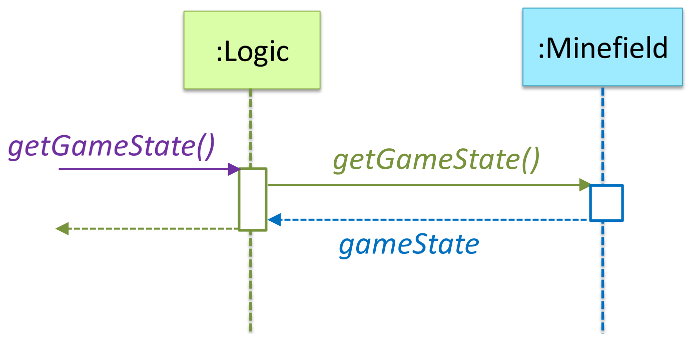


It appears as if MSLogic is simply redirecting operations to other internal components. This is normal for classes that play the role of a "façade" whose main job is to shield the component's clients from its internal complexities.

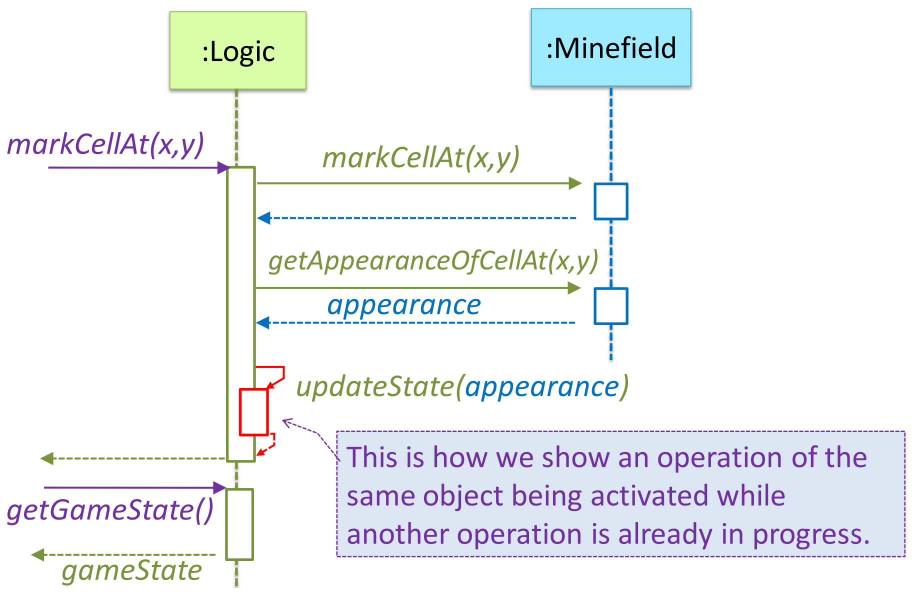
How is getGameState() operation supported? Given below are two ways of supporting it (there could be other ways):

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.

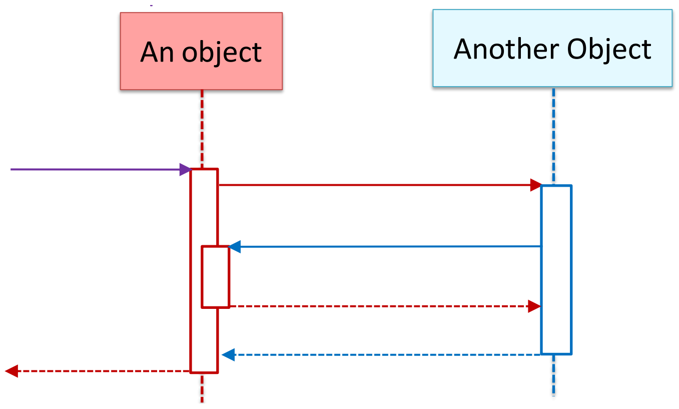
Here's the SD for option 1.



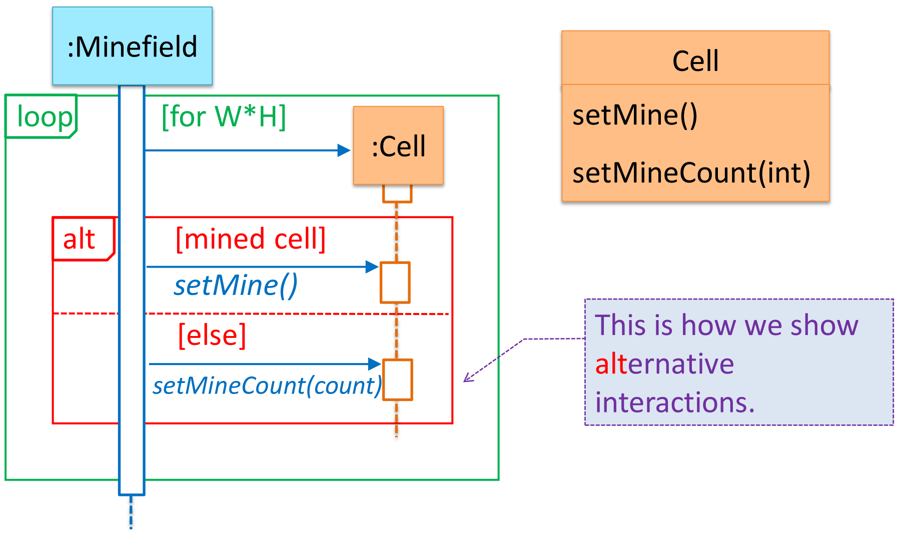
Here's the SD for option 2. Here, we assume that the game state is updated after every mark/clear action.


:Logic
:Minefield
markCellAt(x,y)
markCellAt(x,y)
getAppearanceOfCellAt(x,y)
appearance
updateState(appearance)
This is how we show an operation of the same object being activated while another operation is already in progress.
getGameState()
gameState

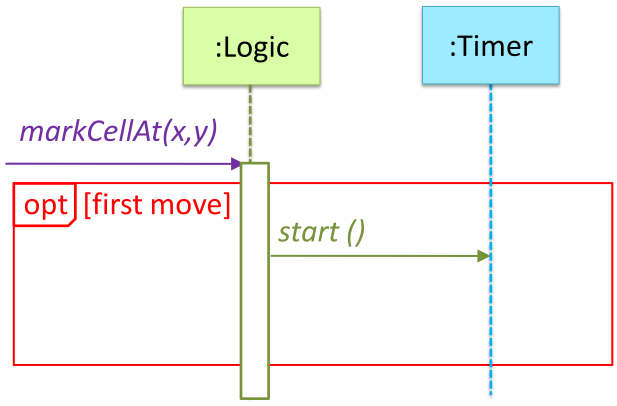
Here is another example of that notation.



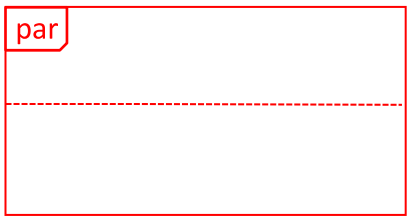
At this point, we know the API of Logic and have some idea about its internal behavior. We also know the API of the Minefield class. Now let us explore the internal behavior of Minefield. What happens inside its constructor? We can design it in this way.



To illustrate how we show optional interactions, let us assume Minesweeper supports the ‘timing’ feature.



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



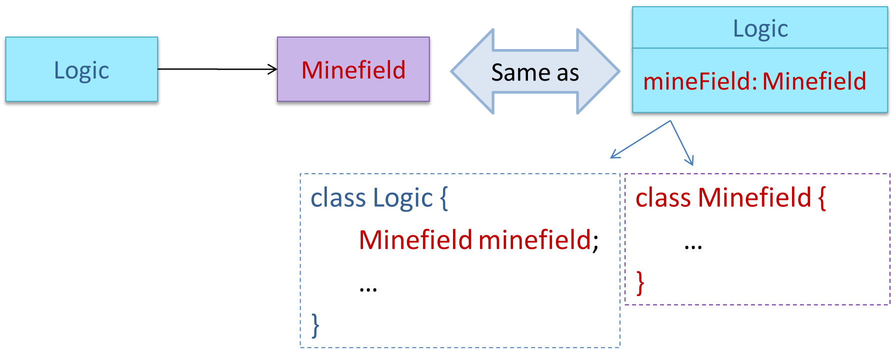
Note that it not necessary to draw elaborate sequence diagrams for all interactions inside the component you are designing. They can be done as rough sketches, and only when you are not sure which operations are required by each class, or to verify that your class structure can indeed support the required operations.

Implementing basic class structures

Most OO languages have direct ways of implementing these.

Java	Classes → Java classes Attributes → Java variables Operations → Java methods
C++	Classes → C++ classes (and header files) Attributes → C++ variables Operations → C++ functions

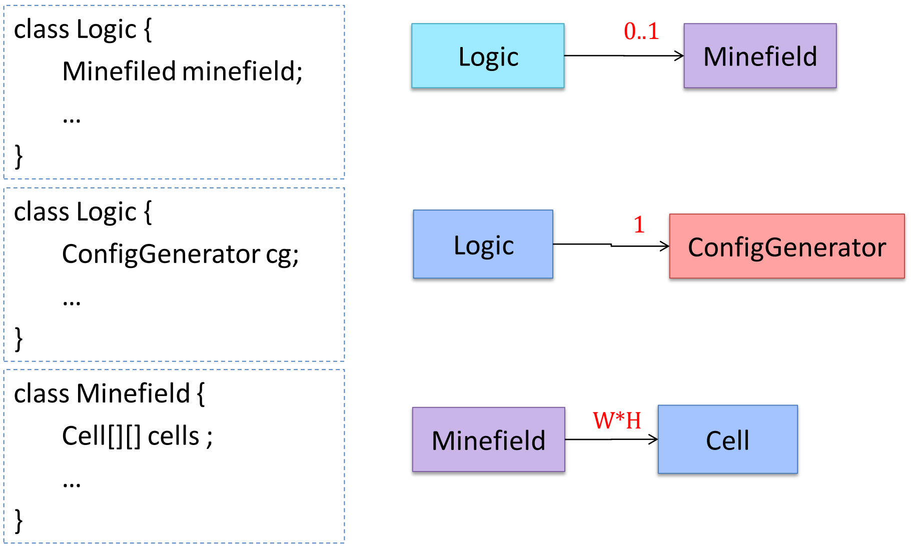
Note that we are referring to reference type variables here, not primitive variables such as int. We use variables to implement associations.



```
class Logic {
      Minefield minefield;
      ...
}
```

```
class Minefield {
      ...
}
```

A variable gives us a 0..1 multiplicity (also called *optional associations*) because a variable can hold a reference to a single object or null. A variable can be used to implement a 1 multiplicity too (also called *compulsory associations*). To implement other multiplicities, we have to choose a suitable data structure such as Arrays, ArrayLists, HashMaps, Sets, etc.



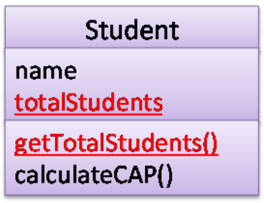
```
class Logic {
      Minefiled minefield;
      ...
}
```

```
class Logic {
      ConfigGenerator cg;
      ...
}
```

```
class Minefield {
      Cell[][] cells ;
      ...
}
```

Class-level members

Most OO languages allow defining class-level (also called static) attributes and operations. They are like 'global' variables/functions but attached to a particular class. For example, the variable totalStudents of the Student class can be declared static because it should be shared by all Student objects. However, the variable name should not be static as each Student should have its own name. Similarly, getTotalStudent() can be a static operation. In UML class diagrams, we use underlines to denote class-level attributes and variables.



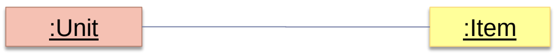
References

[1] Object-Oriented Programming with Objective-C , A document by Apple inc., retrieved from http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/OOP_ObjC/OOP_ObjC.pdf

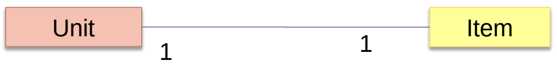
Worked examples

[Q1]

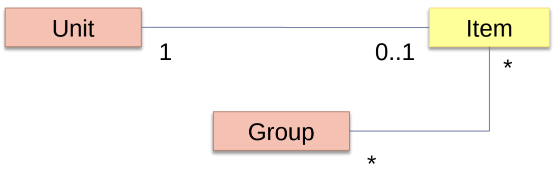
(a) Which of the following class diagrams match with the object diagram below? For example, class diagram (1) matches with the object diagram because the object diagram could be an instance of the class diagram given.



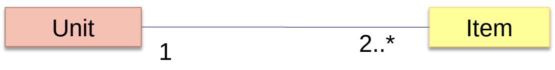
(1)



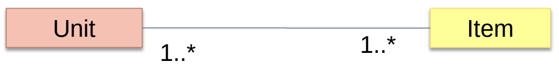
(2)



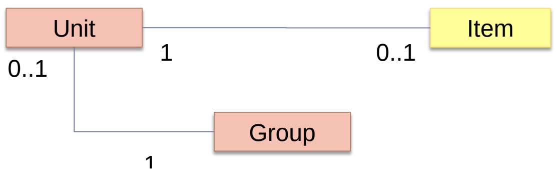
(3)



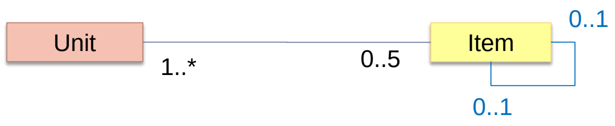
(4)



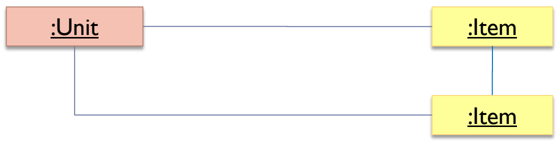
(5)



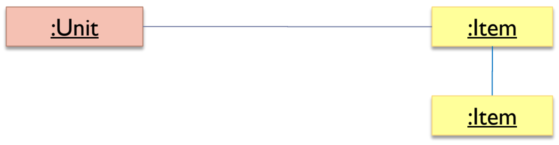
(b) Which of the following object diagrams are allowed by the class diagram below?



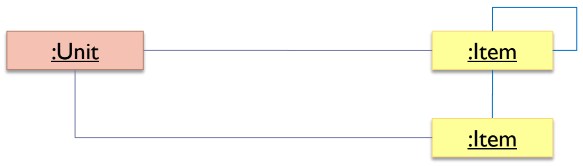
(1)



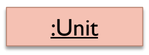
(2)



(3)



(4)



[A1]
(a)

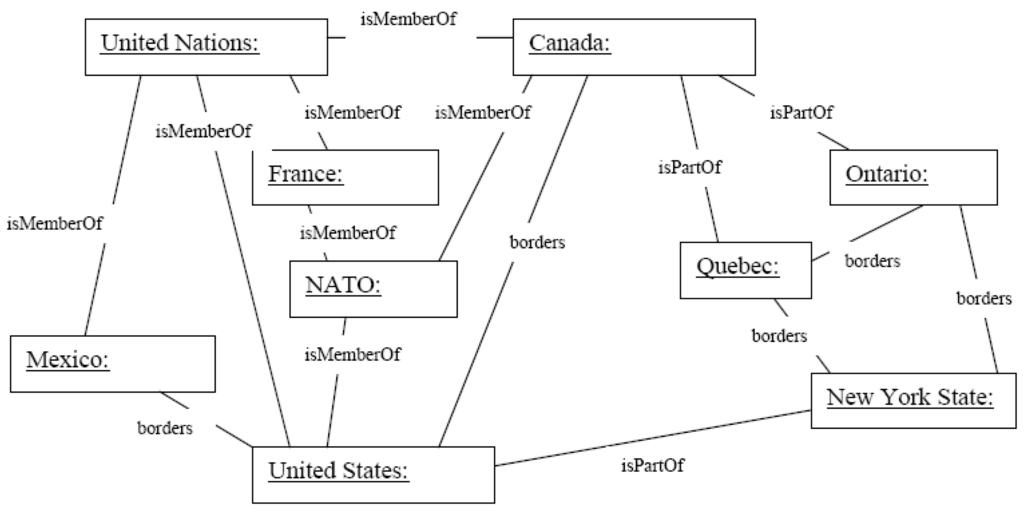
(1) matches
(2) matches

(3) does not match – According to this model, there should be at least 2 Items per Unit.
(4) matches
(5) does not match – According to this model, a Unit must have a link to a Group.

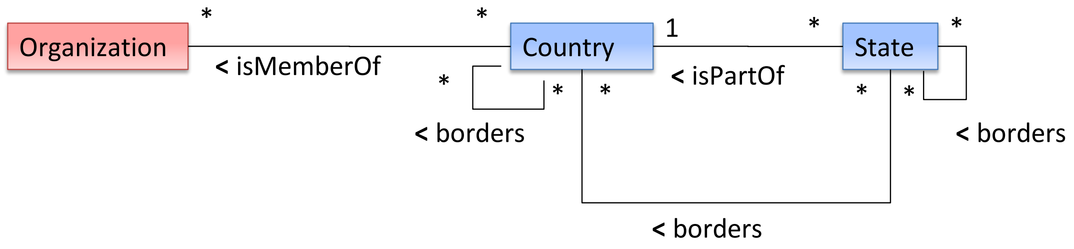
(b)

(1) allowed
(2) not allowed. One item is not linked to a Unit
(3) Not allowed. An Item cannot be linked to more than one other Item.
(4) Allowed. A unit can exist without an Item.

[Q2]
Infer the class names that are missing in the following object diagram. Draw a class diagram that could possibly generate this object diagram.



[A2]

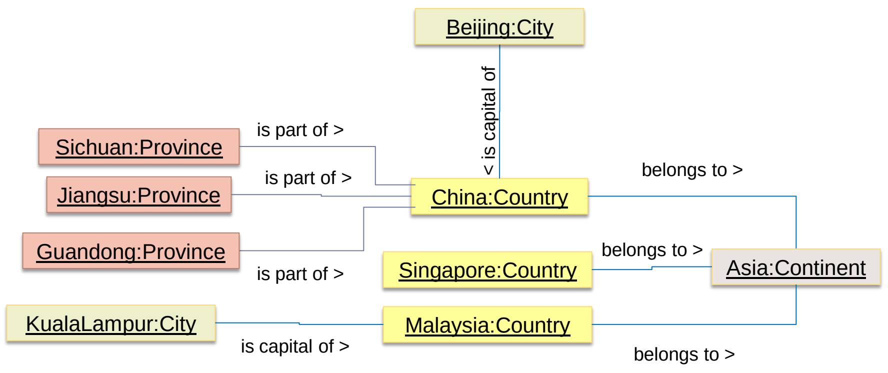


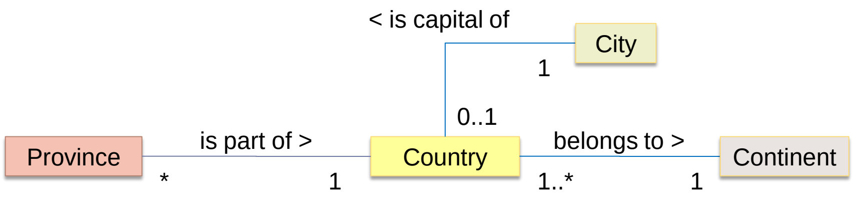
Note: This answer does not use inheritance as it was not covered in this handout.

[Q3]
First, draw an object diagram to represent the following description. Then, use the object diagram to draw a class diagram. Be sure to indicate the multiplicity and label the associations.

Sichuan, Jiangsu, and Guandong are all provinces in China. Singapore, Malaysia, and China are all countries in Asia. Beijing is the capital of China. Kuala Lumpur is the capital of Malaysia.

[A3]





---End of Document---