UML

Class diagrams

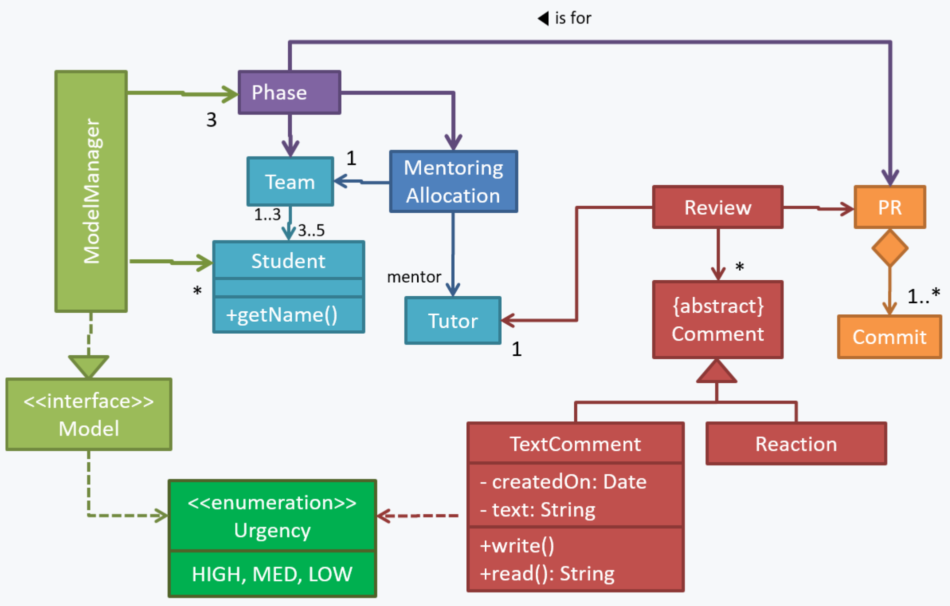
Introduction

What

★☆☆☆ Can explain/identify class diagrams

UML *class diagrams* describe the structure (but not the behavior) of an OOP solution. These are possibly the most often used diagrams in the industry and are an indispensable tool for an OO programmer.

An example class diagram:

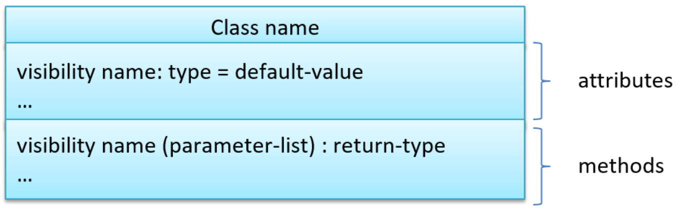


Classes

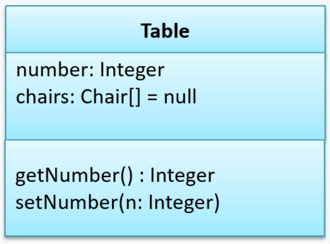
What

★☆☆☆ Can draw UML classes

The basic UML notations used to represent a *class*:

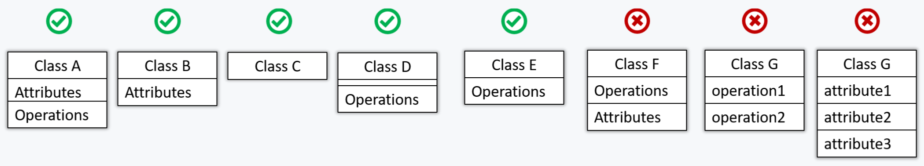

Class name
visibility name: type = default-value
...
visibility name (parameter-list) : return-type
...
attributes
methods

A `Table` class shown in UML notation:



> The equivalent code

The 'Operations' compartment and/or the 'Attributes' compartment may be omitted if such details are not important for the task at hand. Similarly, *some* attributes/operations can be omitted if not relevant. 'Attributes' always appear above the 'Operations' compartment. All operations should be in one compartment rather than each operation in a separate compartment. Same goes for attributes.

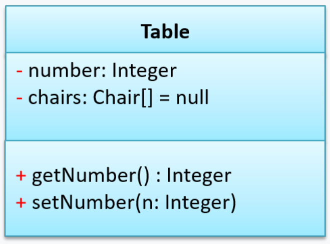


The *visibility* of attributes and operations is used to indicate the level of access allowed for each attribute or operation. The types of visibility and their exact meanings depend on the programming language used. Here are some common visibilities and how they are indicated in a class diagram:

- `+` : public
- `-` : private
- `#` : protected
- `~` : package private

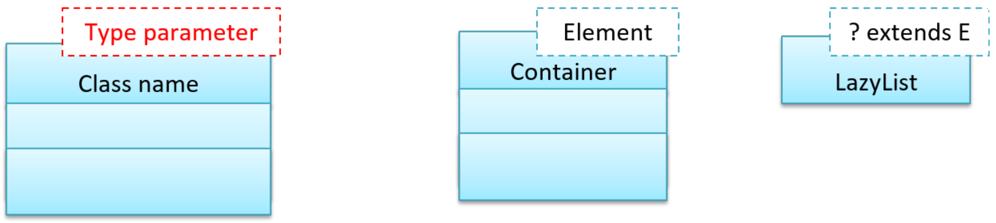
> How visibilities map to programming language features

`Table` class with visibilities shown:



> The equivalent code

Generic classes can be shown as given below. The notation format is shown on the left, followed by two examples.



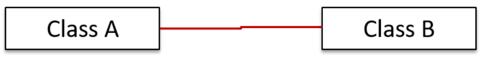
Exercises

Associations

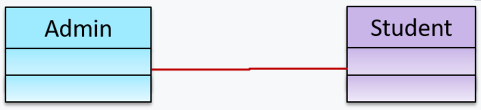
What

★☆☆☆ Can interpret simple associations in a class diagram

You should use a solid line to show an association between two classes.



This example shows an association between the `Admin` class and the `Student` class:

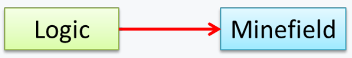


Navigability

★★☆☆ Can interpret association navigabilities in class diagrams

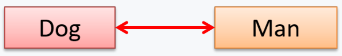
Use arrowheads to indicate the navigability of an association.

In this example, the navigability is unidirectional, and is from the `Logic` class to the `Minefield` class. That means if a `Logic` object `L` is associated with a `Minefield` object `M`, `L` has a reference to `M` but `M` doesn't have a reference to `L`.



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

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

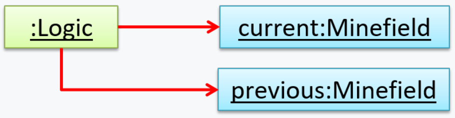
Here is an example of a bidirectional navigability; i.e., if a `Dog` object `d` is associated with a `Man` object `m`, `d` has a reference to `m` and `m` has a reference to `d`.



```
class Dog {
    Man man;
    // ...
}

class Man {
    Dog dog;
    // ...
}
```

Navigability can be shown in class diagrams as well as object diagrams.

According to this object diagram, the given `Logic` object is associated with and aware of two `MineField` objects.

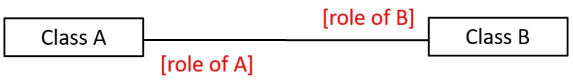


Exercises

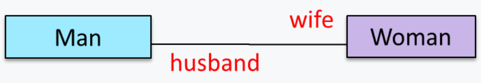
Roles

★★★☆ Can explain/use association roles in class diagrams

Association Role **labels are used to indicate the role played by the classes in the association.**



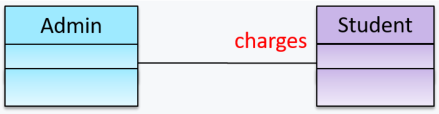
This association represents a marriage between a `Man` object and a `Woman` object. The respective roles played by objects of these two classes are `husband` and `wife`.



Note how the variable names match closely with the association roles.

```
class Man {
    Woman wife;
}

class Woman {
    Man husband;
}
```

The role of `Student` objects in this association is `charges` (i.e. Admin is in charge of students)

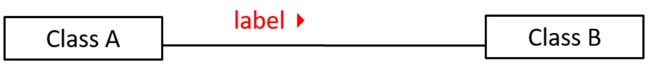


```
class Admin {
    List<Student> charges;
}
```

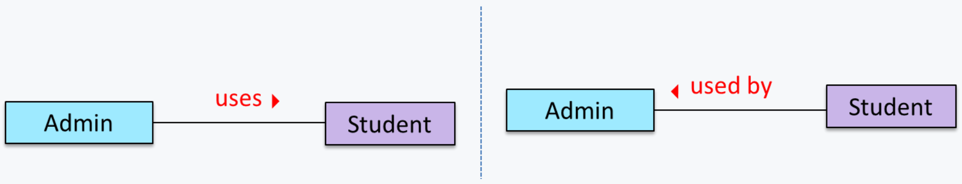
Labels

★★☆☆ 🏆 Can explain/use association labels in class diagrams

Association labels **describe the meaning of the association.** The arrow head indicates the direction in which the label is to be read.



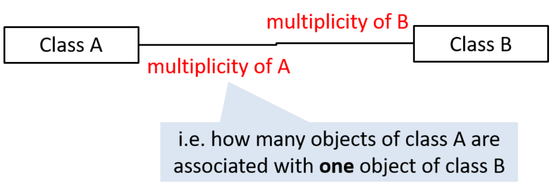
In this example, the same association is described using two different labels.



- Diagram on the left: `Admin` class is associated with `Student` class because an `Admin` object *uses* a `Student` object.
- Diagram on the right: `Admin` class is associated with `Student` class because a `Student` object is *used by* an `Admin` object.

Multiplicity

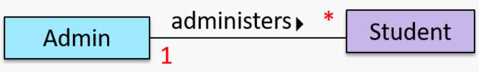
★★★☆ 🏆 Can explain what is the multiplicity of an association


Class A
multiplicity of B
Class B
multiplicity of A
i.e. how many objects of class A are associated with **one** object of class B

Commonly used multiplicities:

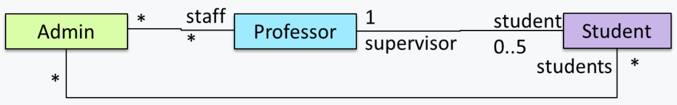
- `0..1` : *optional*, can be linked to 0 or 1 objects.
- `1` : *compulsory*, must be linked to one object at all times.
- `*` : can be linked to 0 or more objects.
- `n..m` : the number of linked objects must be within `n` to `m` inclusive.

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



In the diagram below,

- Each student must be supervised by exactly one professor. i.e. There cannot be a student who doesn't have a supervisor or has multiple supervisors.
- A professor cannot supervise more than 5 students but can have no students to supervise.
- An admin can handle any number of professors and any number of students, including none.
- A professor/student can be handled by any number of admins, including none.



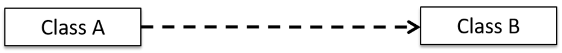
Exercises

Dependencies

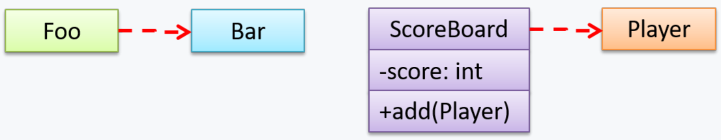
What

★★★☆ Can use dependencies in a class diagram

UML uses a dashed arrow to show dependencies.



Two examples of dependencies:



Dependencies vs associations:

- An association is a relationship resulting from one object keeping a reference to another object (i.e., storing an object in an instance variable). While such a relationship forms a *dependency*, we need not show that as a dependency arrow in the class diagram if the association is already indicated in the diagram. That is, showing a dependency arrow does not add any value to the diagram. Similarly, an inheritance results in a dependency from the child class to the parent class but we don't show it as a dependency arrow either, for the same reason as above.
- **Use a dependency arrow to indicate a dependency only if that dependency is not already captured by the diagram in another way** (for instance, as an association or an inheritance) e.g., class `Foo` accessing a constant in `Bar` but there is no association/inheritance from `Foo` to `Bar`.

Associations as attributes

What

★★★☆ Can show an association as an attribute

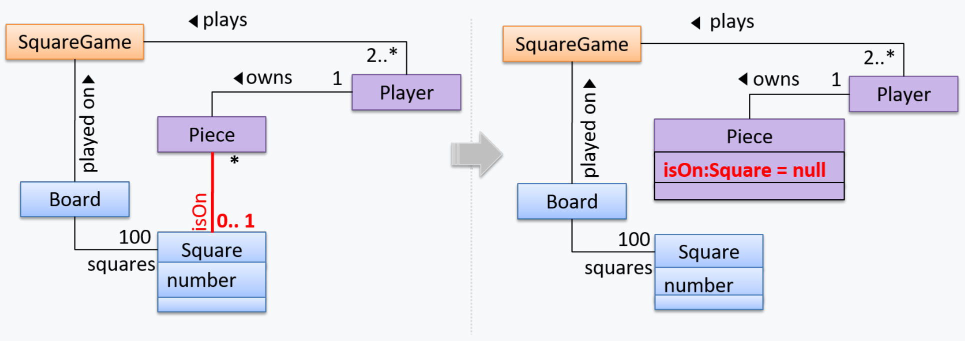
An association can be shown as an attribute instead of a line.

Association multiplicities and the default value can be shown as part of the attribute using the following notation. Both are optional.

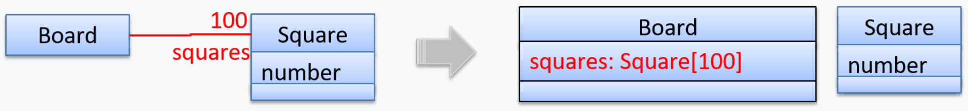
`name: type [multiplicity] = default value`

The diagram below depicts a multi-player *Square Game* being played on a board comprising of 100 squares. Each of the squares may be occupied with any number of pieces, each belonging to a certain player.

A `Piece` may or may not be on a `Square`. Note how that association can be replaced by an `isOn` attribute of the `Piece` class. The `isOn` attribute can either be `null` or hold a reference to a `Square` object, matching the `0..1` multiplicity of the association it replaces. The default value is `null`.



The association that a `Board` has 100 `Square`s can be shown in either of these two ways:



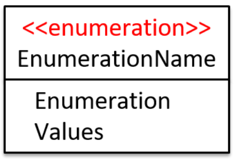
Show each association as **either an attribute or a line but not both**. A line is preferred as it is easier to spot.

Enumerations

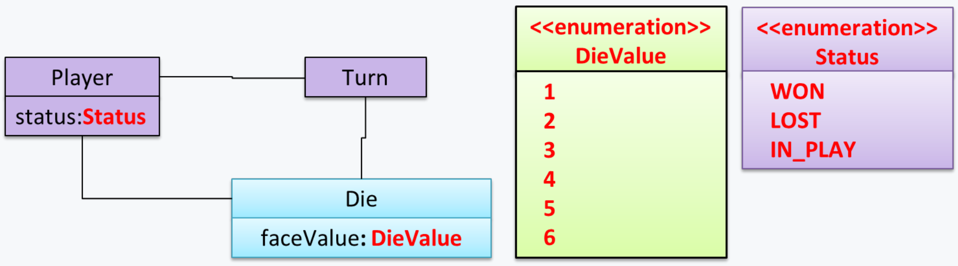
What

★★★☆ Can interpret enumerations in class diagrams

Notation:



In the class diagram below, there are two enumerations in use:



Exercises

Define WeekDay Enum

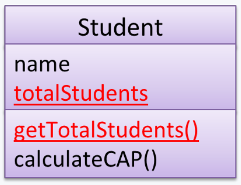
Class-level members

What

★★★☆ Can interpret class-level members in class diagrams

In UML class diagrams, **underlines denote class-level attributes and methods.**

In the class diagram below, the `totalStudents` attribute and the `getTotalStudents` method are class-level.

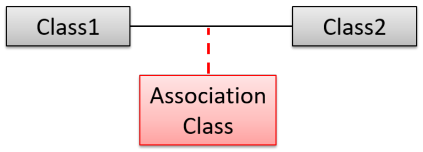


Association classes

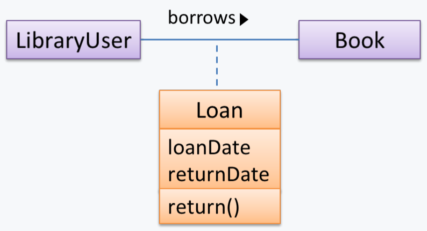
What

★★★☆ 🏆 Can interpret association classes in class diagrams

Association classes are denoted as a connection to an association link using a dashed line as shown below.



In this example `Loan` is an association class because it stores information about the `borrows` association between the `User` and the `Book`.



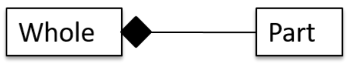
Composition

What

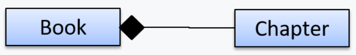
★★☆☆ 🏆 Can interpret composition in class diagrams

UML uses a solid diamond symbol to denote composition.

Notation:



A `Book` consists of `Chapter` objects. When the `Book` object is destroyed, its `Chapter` objects are destroyed too.



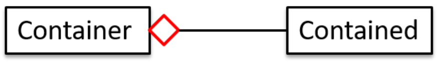
Aggregation

What

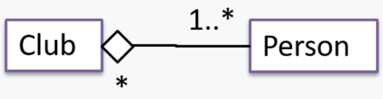
★★★☆ Can interpret aggregation in class diagrams

UML uses a hollow diamond to indicate an aggregation.

Notation:



Example:



Aggregation vs Composition

The distinction between composition (◆) and aggregation (◇) is rather blurred. Martin Fowler's famous book *UML Distilled* advocates omitting the aggregation symbol altogether because using it adds more confusion than clarity.

Exercises

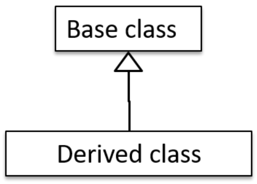
Class inheritance

What

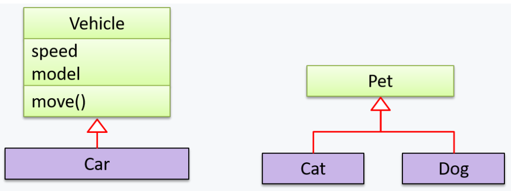
★★☆☆ Can interpret class inheritance in class diagrams

You can use a triangle and a solid line (not to be confused with an arrow) to indicate class inheritance.

Notation:



Examples: The `Car` class *inherits* from the `Vehicle` class. The `Cat` and `Dog` classes *inherit* from the `Pet` class.



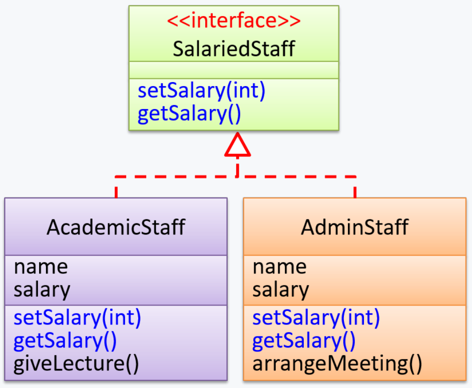
Interfaces

What

★★☆☆ 🏆 Can interpret interfaces in class diagrams

An interface is shown similar to a class with an additional keyword `<<interface>>`. When a class implements an interface, it is shown similar to class inheritance except a dashed line is used instead of a solid line.

The `AcademicStaff` and the `AdminStaff` classes *implement* the `SalariedStaff` interface.

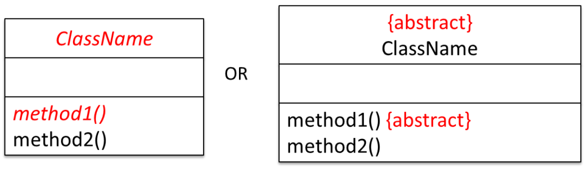


Abstract classes

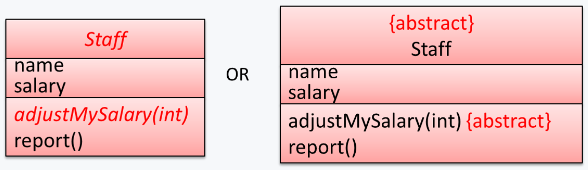
What

★★★☆ 🏆 Can interpret abstract classes in class diagrams

You can use *italics* or `{abstract}` (preferred) keyword to denote abstract classes/methods.



Example:



Combine

Basic

★★☆☆ Can combine different basic aspects of class diagrams

Exercises

Sequence diagrams

Introduction

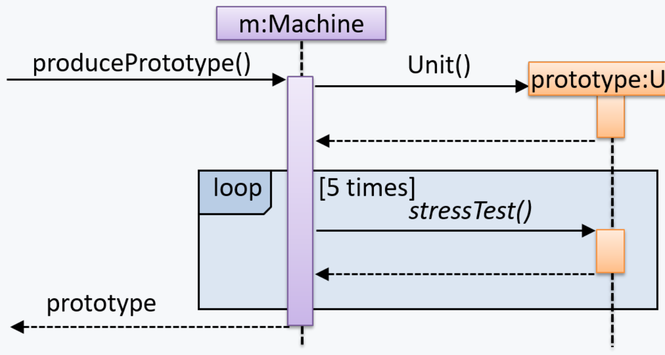
★☆☆☆ Can explain/identify sequence diagrams

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

Consider the code below.

```
class Machine {

    Unit produceType() {
        Unit prototype = new Unit();
        for (int i = 0; i < 5; i++) {
            prototype.stressTest();
        }
        return prototype;
    }
}

class Unit {

    public void stressTest() {

    }
}
```

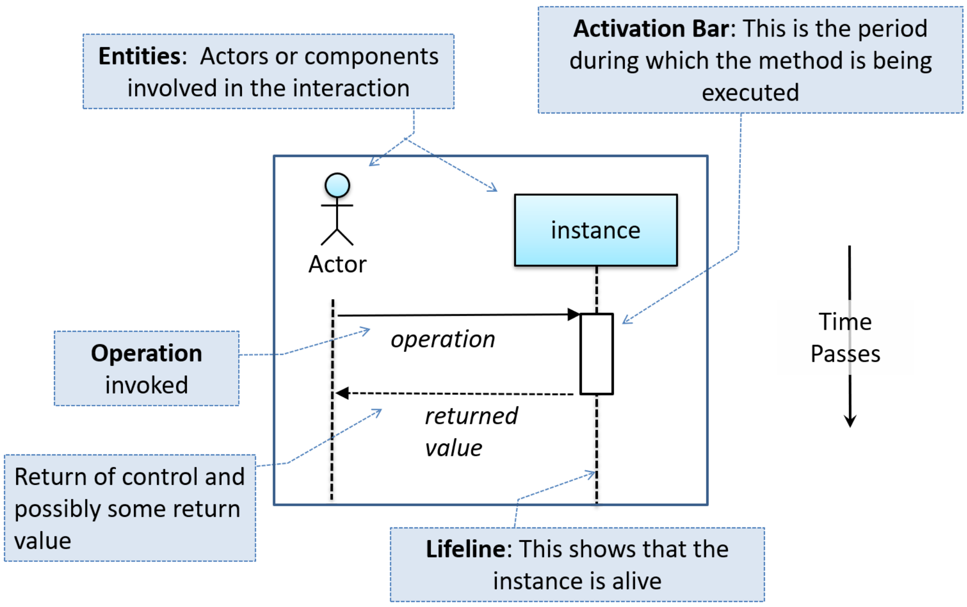
Here is the sequence diagram to model the interactions for the method call `produceProtoype()` on a `Machine` object.



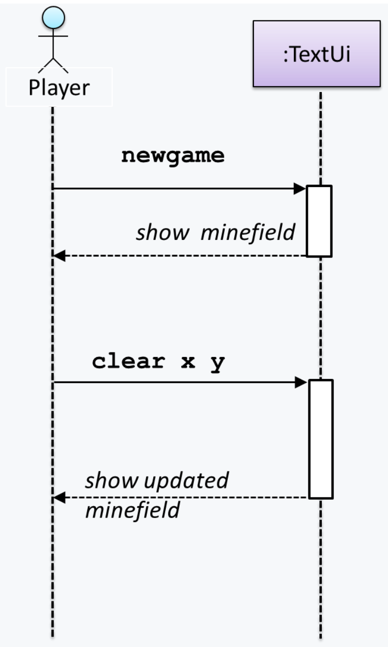
Basic

★☆☆☆ 🏆 Can interpret sequence diagrams with basic notation

Notation:


Entities: Actors or components involved in the interaction
Activation Bar: This is the period during which the method is being executed
Actor
instance
operation
returned value
Time Passes
Operation invoked
Return of control and possibly some return value
Lifeline: This shows that the instance is alive

This sequence diagram shows some interactions between a human user and the Text UI of a CLI Minesweeper game.



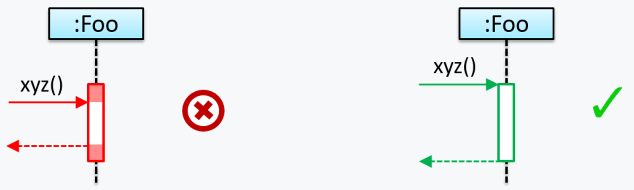
The player runs the `newgame` action on the `TextUi` object which results in the `TextUi` showing the minefield to the player. Then, the player runs the `clear x y` command; in response, the `TextUi` object shows the updated minefield.

The `:TextUi` in the above example 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 e.g. `TextUi1:TextUi` and `TextUi2:TextUi`.

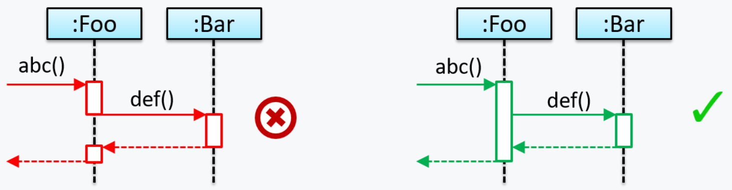
Arrows representing method calls should be solid arrows while those representing method returns should be dashed arrows.

Note that unlike in object diagrams, the **class/object name is not underlined in sequence diagrams**.

❌ **[Common notation error] Activation bar too long:** The activation bar of a method cannot start before the method call arrives and a method cannot remain active after the method has returned. In the two sequence diagrams below, the one on the left commits this error because the activation bar starts *before* the method `Foo#xyz()` is called and remains active *after* the method returns.



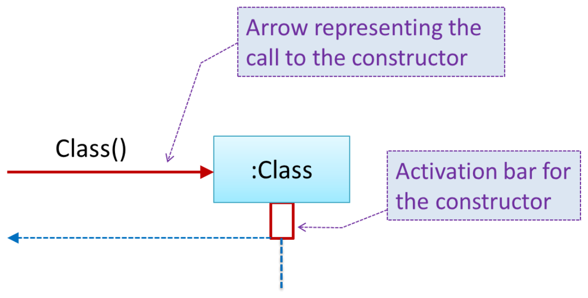
❌ **[Common notation error] Broken activation bar:** The activation bar should remain unbroken from the point the method is called until the method returns. In the two sequence diagrams below, the one on the left commits this error because the activation bar for the method `Foo#abc()` is not contiguous, but appears as two pieces instead.



Object Creation

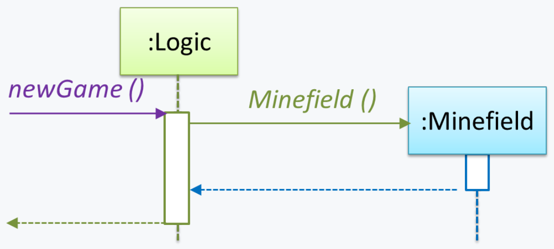
★★☆☆ Can interpret sequence diagrams with object creation

Notation:


Arrow representing the call to the constructor
Class()
:Class
Activation bar for the constructor

- The arrow that represents the constructor arrives at the side of the box representing the instance.
- The activation bar represents the period the constructor is active.

The `Logic` object creates a `Minefield` object.



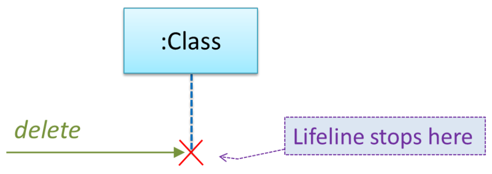
Object Deletion

★★★☆ Can interpret sequence diagrams with object deletion

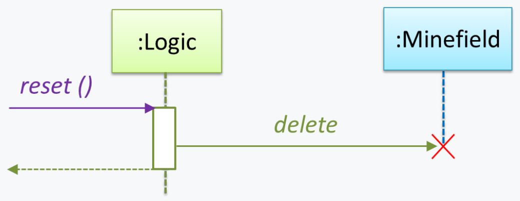
UML uses an `X` at the end of the lifeline of an object to show its deletion.

Although object deletion is not that important in languages such as Java that support automatic memory management, you can still show object deletion in UML diagrams to indicate the point at which the object ceases to be used.

Notation:



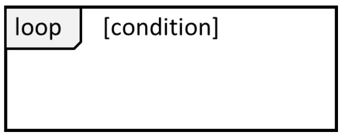
Note how the below diagram shows the deletion of the `Minefield` object.



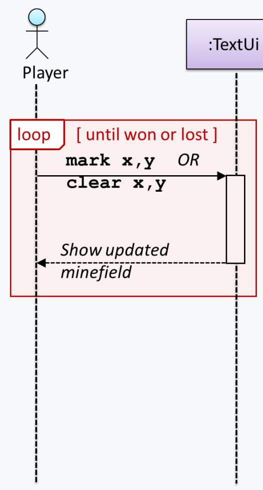
Loops

★★☆☆ Can interpret sequence diagrams with loops

Notation:



The `Player` calls the `mark x,y` command or `clear x y` command repeatedly until the game is won or lost.

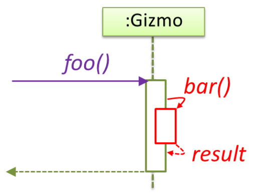


Self Invocation

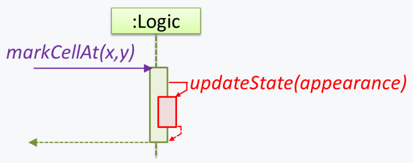
★★★☆ Can interpret sequence diagrams with self invocation

UML can show a method of an object calling another of its own methods.

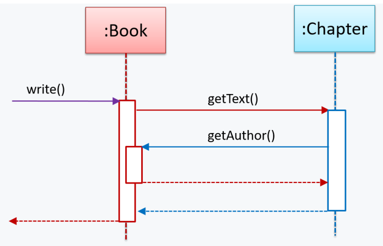
Notation:



The `markCellAt(...)` method of a `Logic` object is calling its own `updateState(...)` method.



In this variation, the `Book#write()` method is calling the `Chapter#getText()` method which in turn does a *call back* by calling the `getAuthor()` method of the calling object.

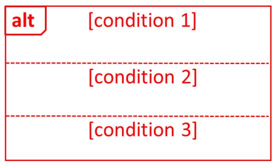


Alternative Paths

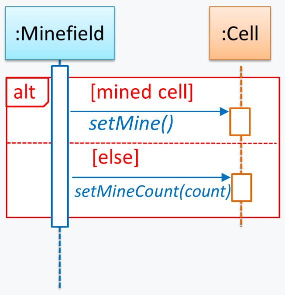
★★★☆ 🏆 **Can interpret sequence diagrams with alternative paths**

UML uses `alt` frames to indicate alternative paths.

Notation:



`Minefield` calls the `Cell#setMine` method if the cell is supposed to be a mined cell, and calls the `Cell:setMineCount(...)` method otherwise.



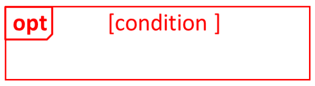
No more than one alternative partitions be executed in an `alt` frame. That is, it is acceptable for none of the alternative partitions to be executed but it is not acceptable for multiple partitions to be executed.

Optional Paths

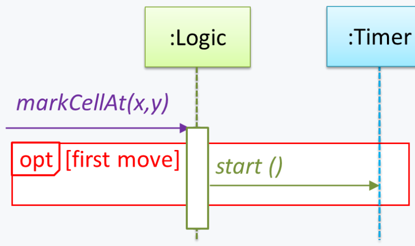
★★★☆ 🏆 **Can interpret sequence diagrams with optional paths**

UML uses `opt` frames to indicate optional paths.

Notation:



`Logic#markCellAt(...)` calls `Timer#start()` only if it is the first move of the player.

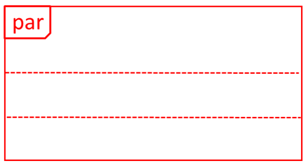


Parallel Paths

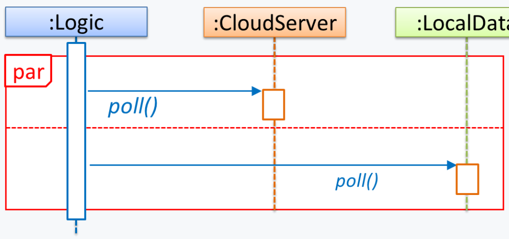
★★★☆ Can interpret sequence diagrams with parallel paths

UML uses `par` frames to indicate parallel paths.

Notation:



`Logic` is calling methods `CloudServer#poll()` and `LocalData#poll()` in parallel.



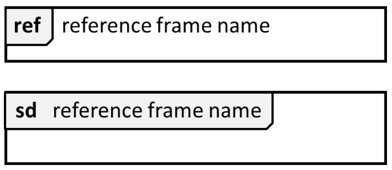
If you show parallel paths in a sequence diagram, the corresponding Java implementation is likely to be *multi-threaded* because a normal Java program cannot do multiple things at the same time.

Reference Frames

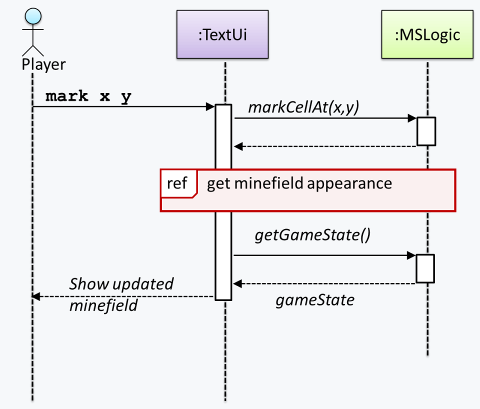
★★★☆ Can interpret sequence diagrams with reference frames

UML uses *ref frame* to allow a segment of the interaction to be omitted and shown as a separate sequence diagram. Reference frames help you to break complicated sequence diagrams into multiple parts or simply to omit details you are not interested in showing.

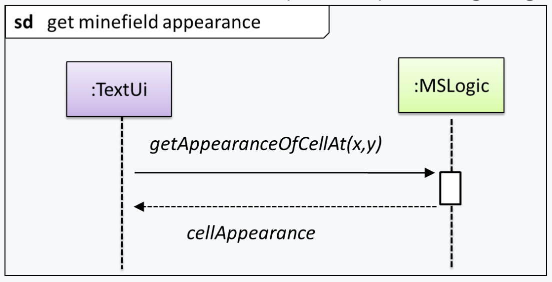
Notation:



The details of the `get minefield appearance` interactions have been omitted from the diagram.



Those details are shown in a separate sequence diagram given below.

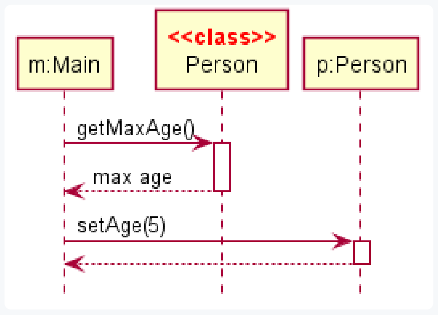


Calls to Static Methods

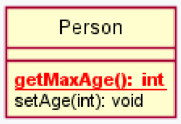
★★★☆ Can show calls to static methods

Method calls to `static` (i.e., class-level) methods are received by the class itself, not an instance of that class. You can use `<<class>>` to show that a participant is the class itself.

In this example, `m` calls the static method `Person.getMaxAge()` and also the `setAge()` method of a `Person` object `p`.



Here is the `Person` class, for reference:

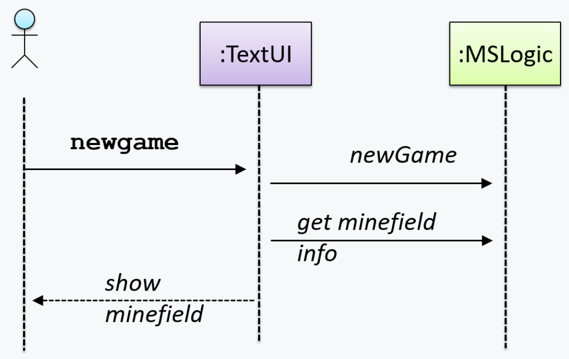


Minimal Notation

★★★☆ 🏆 Can interpret sequence diagrams with minimal notation

To reduce clutter, **optional elements (e.g, activation bars, return arrows) may be omitted** if the omission does not result in ambiguities or loss of *relevant* information. Informal operation descriptions such as those given in the example below can be used, if more precise details are not required for the task at hand.

A minimal sequence diagram



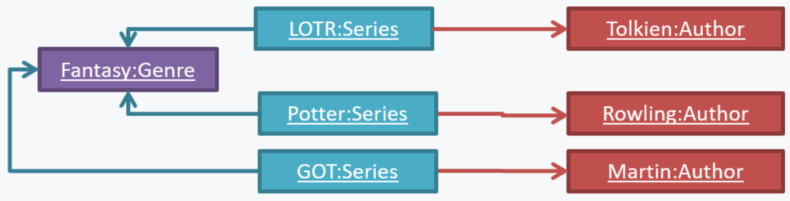
Object diagrams

Introduction

★☆☆☆ 🏆 Can explain/identify object diagrams

An object diagram shows an object structure at a given point of time.

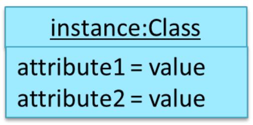
An example object diagram:



Objects

★☆☆☆ Can draw UML objects

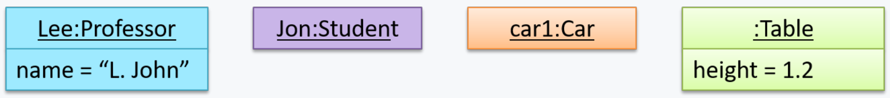
Notation:



Notes:

- The class name and object name e.g. `car1:Car` are underlined.
- `objectName:ClassName` is meant to say 'an instance of `ClassName` identified as `objectName`'.
- Unlike classes, there is no compartment for methods.
- *Attributes* compartment can be omitted if it is not relevant to the task at hand.
- Object name can be omitted too e.g. `:Car` which is meant to say 'an *unnamed* instance of a Car object'.

Some example objects:

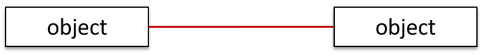


Exercises

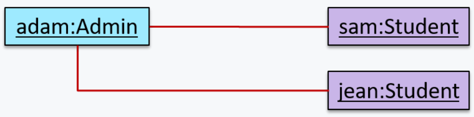
Associations

★☆☆☆ Can interpret simple associations among objects

A solid line indicates an association between two objects.



An example object diagram showing two associations:



Activity diagrams

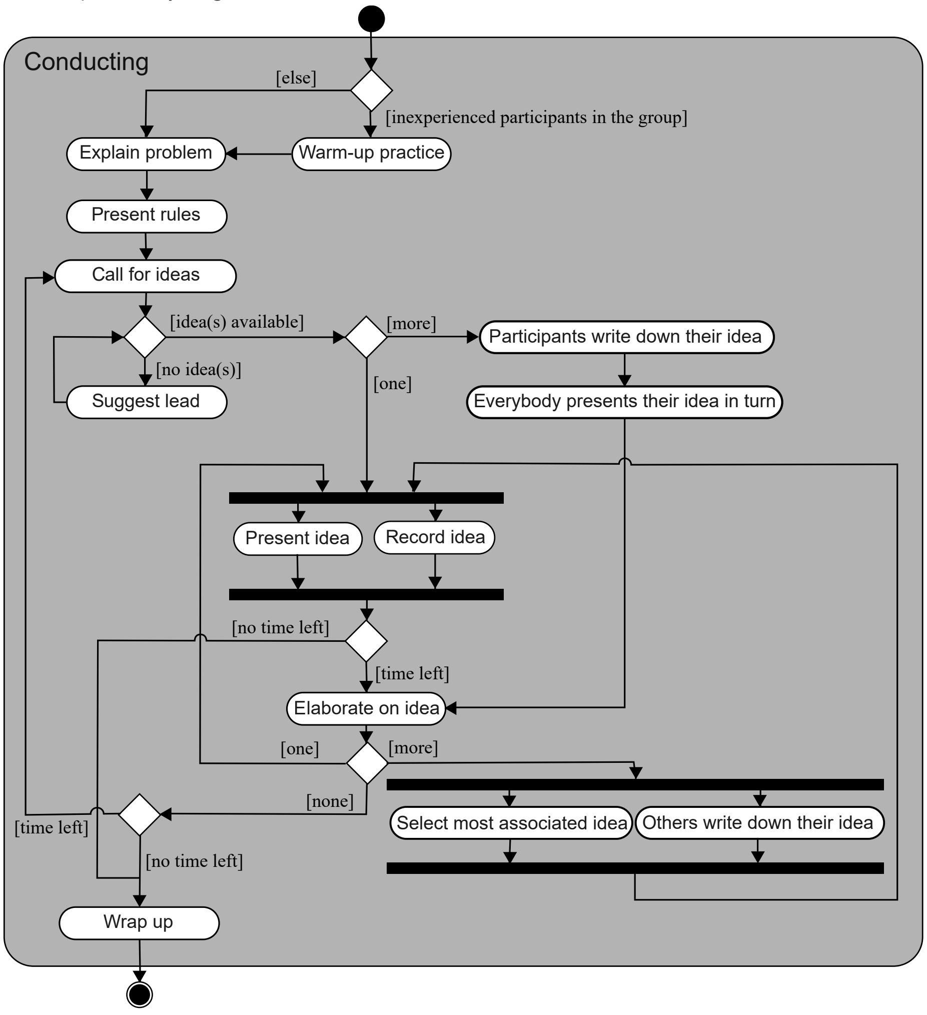
Introduction

What

★☆☆☆ 🏆 Can explain activity diagrams

UML ***activity diagrams*** **(AD) can model workflows.** *Flow charts* are another type of diagram that can model workflows. Activity diagrams are the UML equivalent of flow charts.

An example activity diagram:



[source:wikipeida]

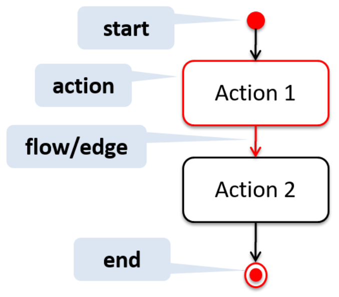
Basic notations

Linear Paths

★★☆☆ 🏆 Can interpret linear paths in activity diagrams

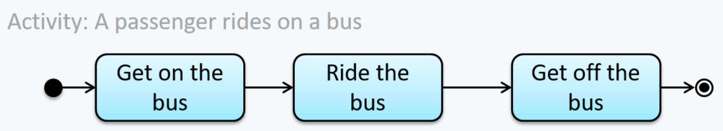
An activity diagram (AD) captures an *activity* through the *actions* and *control flows* that make up the activity.

- An *action* is a single step in an activity. It is shown as a rectangle with rounded corners.
- A *control flow* shows the flow of control from one action to the next. It is shown by drawing a line with an arrow-head to show the direction of the flow.



Note the slight difference between the *start node* and the *end node* which represent the start and the end of the activity, respectively.

This activity diagram shows the action sequence of the activity *a passenger rides the bus*:



Exercises

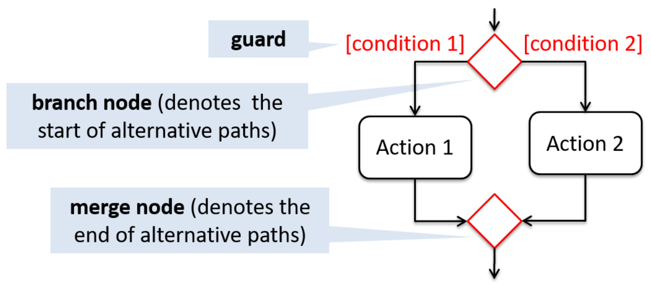
Alternate Paths

★★☆☆ Can interpret alternate paths in activity diagrams

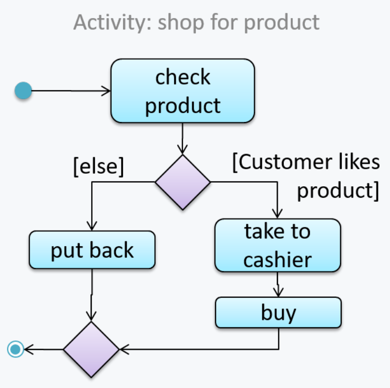
A *branch node* shows the start of alternate paths. Each control flow exiting a branch node has a *guard condition*: a boolean condition that should be true for execution to take that path. ***Exactly one*** **of the guard conditions should be true** at any given branch node.

A *merge node* shows the end of alternate paths.

Both branch nodes and merge nodes are diamond shapes. Guard conditions must be in square brackets.


guard
branch node (denotes the start of alternative paths)
merge node (denotes the end of alternative paths)
[condition 1]
[condition 2]
Action 1
Action 2

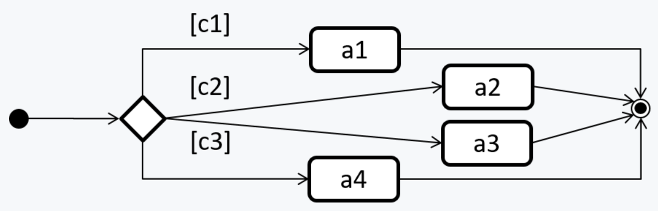
The AD below shows alternate paths involved in the workflow of the activity *shop for product*:



Some acceptable simplifications (by convention):

- Omitting the merge node if it doesn't cause any ambiguities.
- Multiple arrows can starting from the same corner of a branch node.
- Omitting the `[Else]` condition.

The AD below illustrates the simplifications mentioned above:



Exercises

Parallel Paths

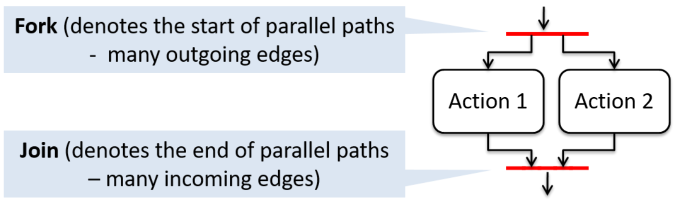
★★☆☆ Can interpret parallel paths in activity diagrams

***Fork* nodes indicate the start of concurrent flows of control.**

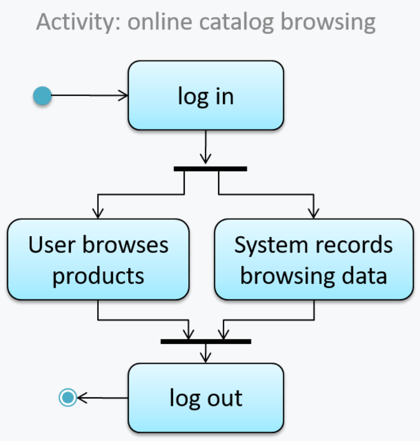
***Join* nodes indicate the end of parallel paths.**

Both have the same notation: a bar.

In a set of parallel paths, execution along **all parallel paths should be complete before the execution can start on the outgoing control flow of the *join*.**


Fork (denotes the start of parallel paths - many outgoing edges)
Action 1
Action 2
Join (denotes the end of parallel paths – many incoming edges)

In this activity diagram (from an online shop website) the actions *User browses products* and *System records browsing data* happen in parallel. Both of them need to finish before the *log out* action can take place.


Activity: online catalog browsing
log in
User browses products
System records browsing data
log out

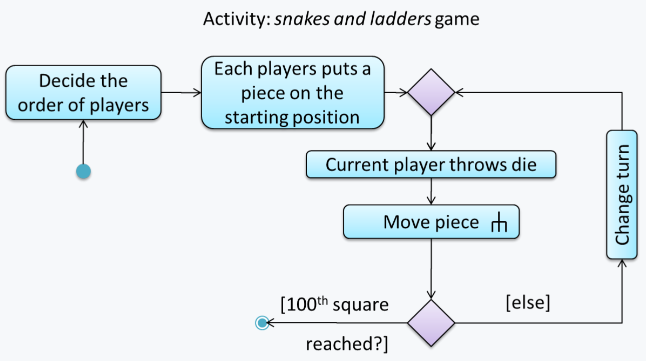
Exercises

Rakes

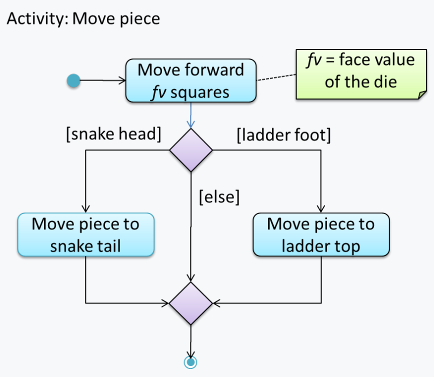
★★★☆ Can use rakes in activity diagrams

The rake notation is used to indicate that a part of the activity is given as a separate diagram.

Here is the AD for a game of 'Snakes and Ladders'.


Activity: *snakes and ladders* game
Decide the order of players
Each players puts a piece on the starting position
Current player throws die
Move piece
Change turn
[100th square reached?]
[else]

The *rake* symbol (in the `Move piece` action above) is used to show that the action is described in another subsidiary activity diagram elsewhere. That diagram is given below.

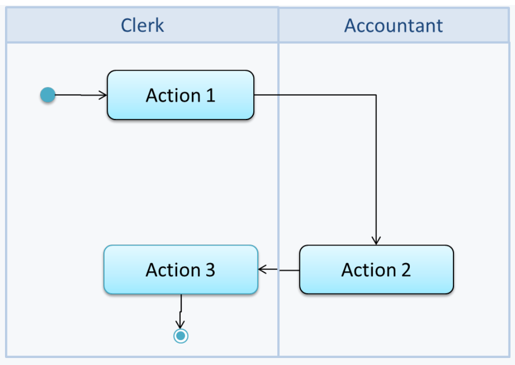


Swimlanes

★★★☆ Can explain swimlanes in activity diagrams

It is possible to *partition* an activity diagram to show who is doing which action. Such partitioned activity diagrams are sometime called *swimlane diagrams*.

A simple example of a swimlane diagram:



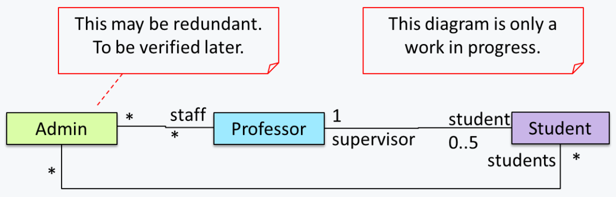
Notes

Notes

★★★☆ 🏆 Can use UML notes

UML notes can augment UML diagrams with additional information. These notes can be shown connected to a particular element in the diagram or can be shown without a connection. The diagram below shows examples of both.

Example:

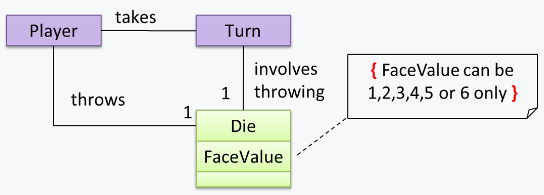

This may be redundant. To be verified later.
This diagram is only a work in progress.
Admin
*
staff
*
Professor
1
supervisor
student
0..5
Student
students
*
*

Constraints

★★★★ 🏆 Can specify constraints in UML diagrams

A *constraint* can be given inside a note, within curly braces. Natural language or a formal notation such as *OCL (Object Constraint Language)* may be used to specify constraints.

Example:



Miscellaneous

Object vs Class Diagrams

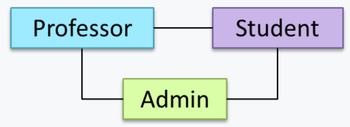
★★☆☆ 🏆 Can distinguish between class diagrams and object diagrams

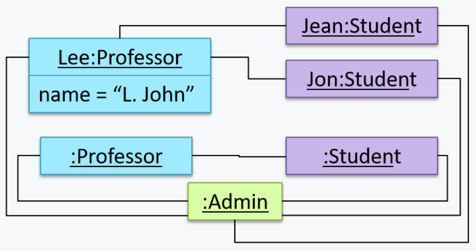
Compared to the notation for class diagrams, object diagrams differ in the following ways:

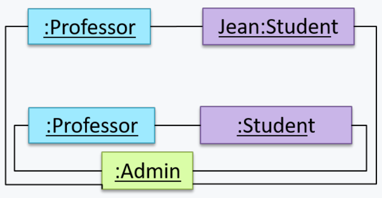
- Show objects instead of classes:
  - Instance name may be shown
  - There is a `:` before the class name
  - Instance and class names are underlined
- Methods are omitted
- Multiplicities are omitted. Reason: an association line in an object diagram represents a connection to exactly one object (i.e., the multiplicity is always 1).

Furthermore, **multiple object diagrams can correspond to a single class diagram**.

Both object diagrams are derived from the same class diagram shown earlier. In other words, each of these object diagrams shows 'an instance of' the same class diagram.

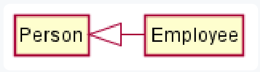




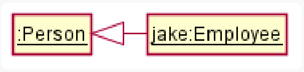


When the class diagram has an inheritance relationship, **the object diagram should show either an object of the parent class or the child class, but not both**.

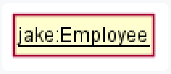
Suppose `Employee` is a child class of the `Person` class. The class diagram will be as follows:



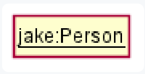
Now, how do you show an `Employee` object named `jake`?

- ✗ 

This is not correct, as there should be only one object.

- ✓ 

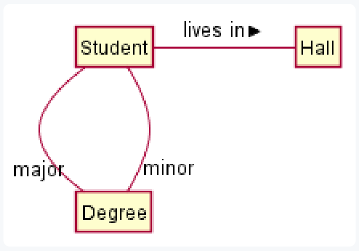
This is OK.

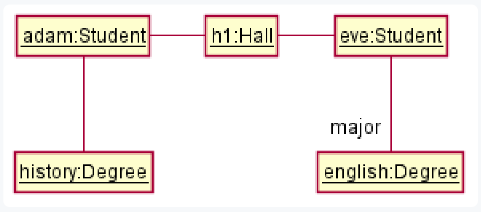
- ✓ 

This is OK, as `jake` *is a* `Person` too. That is, we can show the parent class instead of the child class if the child class doesn't matter to the purpose of the diagram (i.e., the reader of this diagram will not need to know that `jake` is in fact an `Employee`).

Association labels/roles *can* be omitted unless they add value (e.g., showing them is useful if there are multiple associations between the two classes in concern -- otherwise you wouldn't know which association the object diagram is showing)

Consider this class diagram and the object diagram:





We can clearly see that both Adam and Eve lives in hall h1 (i.e., OK to omit the association label `lives in`) but we can't see if History is Adam's major or his minor (i.e., the diagram should have included either an association label or a role there). In contrast, we can see Eve is an English major.

Exercises