Introduction to UML

Yi-ting Chiang

Department of Applied Mathematics
Chung Yuan Christian University

October 24, 2023

About UML

This topic introduces Unified Modeling Language (UML)[1]. UML is used to analyze, specify, and design software systems. It can represent the system that will be constructed.
UML diagrams can be classified in to two groups:

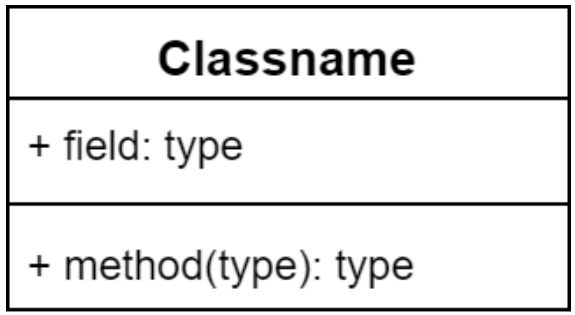
- Structure diagrams: show the static structure of elements in the system
  - class diagram, package diagram, component diagram, deployment diagram, object diagram, composite structure diagram
- Behaviour diagrams: express the dynamic behavioural semantics of a problem or its implementation
  - use diagram, activity diagram, state machine diagram, interaction diagrams (sequence diagram, communication diagram, interaction diagram, timing diagram)

Only class diagram, package diagram, activity diagram, and sequence diagram will be introduced in this lecture

[1] Most figures in this slides is created by using the diagrams.net

Class Diagrams

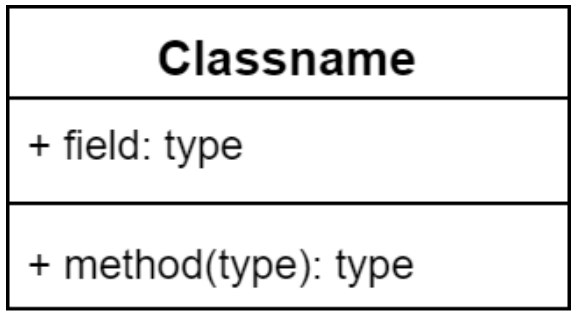
A class diagram is used to show the existence of classes and their relationships in the logical view of a system. That is, it can represent the view of a class and the relationships between classes in a system. Here is an example of a class icon of class diagrams:



This example represents a class. A class icon consists of three compartments:

- class name
- attributes (fields) of the class
- operations (methods) of the class

Class Diagrams



By convention, the class name begins in capital letter (For example "Account"), and the attribute and the operation names' first letter is lowercase with subsequent words starting in uppercase (For example, "name" and "getName()").
Generally, the format of the attribute and operation specification are[2]:

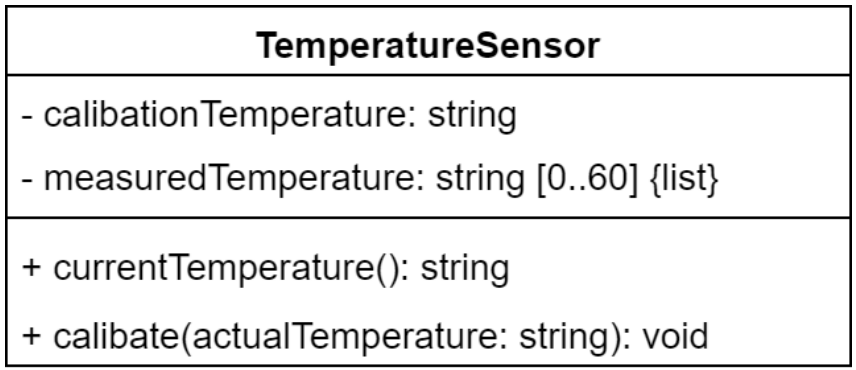
visibility attributeName: type [multiplicity] = DefaultValue {property string}
visibility operationName (parameterName: Type) : ReturnType {property string}

- Visibility: who can directly access the attribute or the operation
- Type and return type: data type of attributes, parameters, or returned data
- DefaultValue: the value to be given at creation time
- Property string: provide additional properties of the attribute or operation

You can select and list only part of the attributes and operations as necessary.

[2]Or use the syntax of the programming language chosen to implement the system.

Class Diagrams



Here are some property strings[3]

Property string	Description
id	is part of the identifier for the class which owns the property
readOnly	is read only
ordered	is ordered
unique	is multi-valued property with no duplicate values
nonunique	is multi-valued property which may have duplicate values
sequence/seq/list	is an ordered bag (is ordered but is not unique)

[3]Selected From: https://www.uml-diagrams.org/property.html

Class Diagrams

The symbol and the description of visibility is:

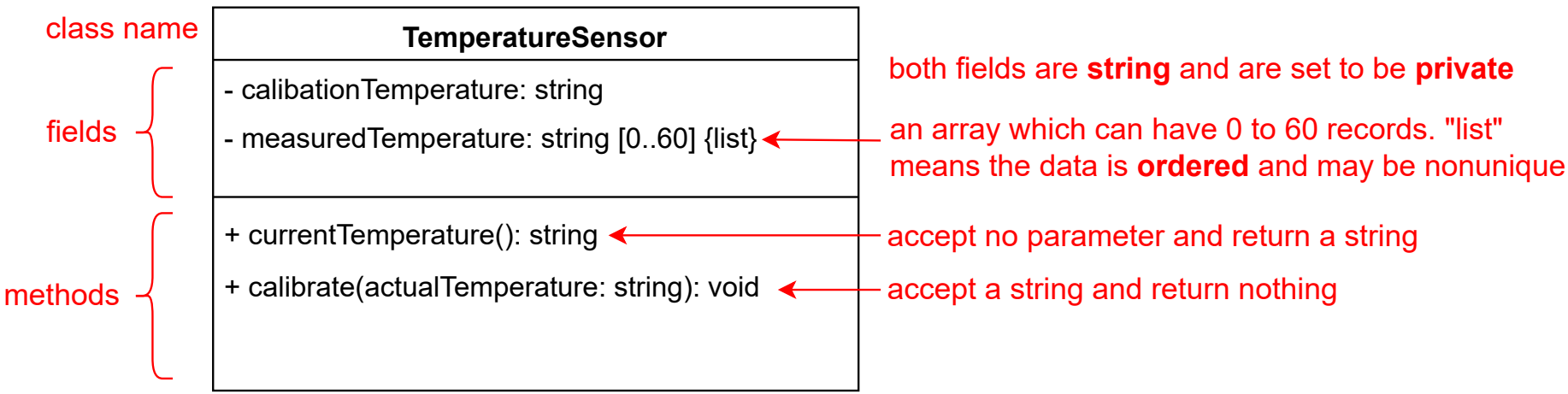
Symbol	Visibility	Description
+	Public	Visible to any element that can see the class
♯	Protected	Visible to other elements within the class and to subclasses
-	Private	Visible to other elements within the class
∼	Package	Visible to elements within the same package

Class Diagrams

Using the hydroponics gardening system as an example. Recall that the temperature sensor has the following responsibilities:

- report current temperature
- calibrate

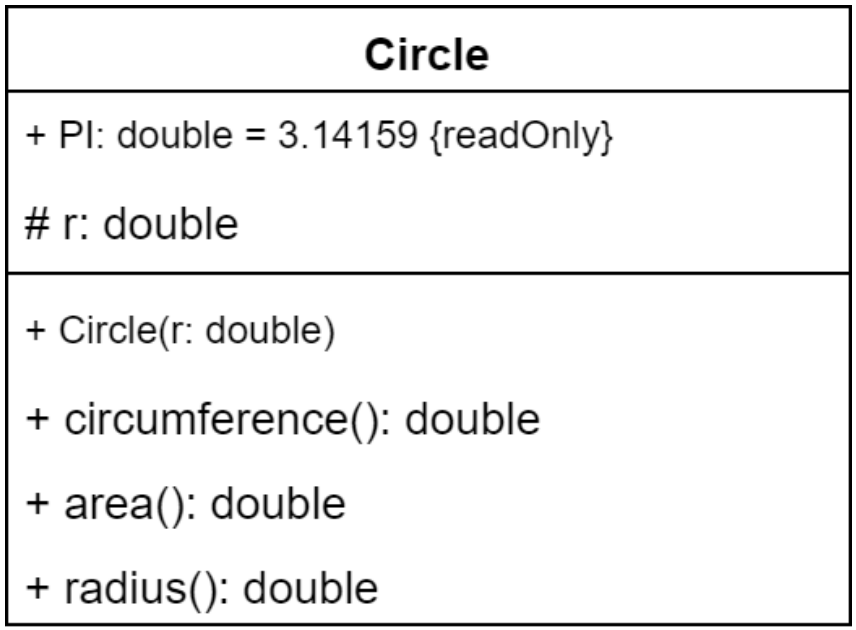
This class can be represented as:



Class Diagrams

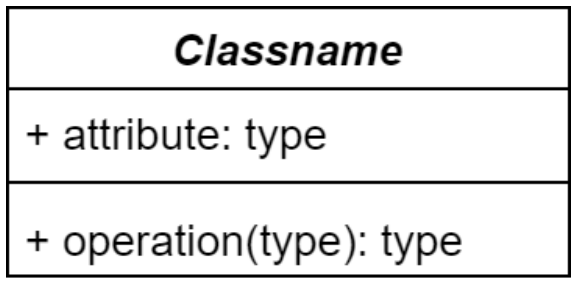
Here is another example of the Java class Circle and its class diagram:

```
public class Circle {
    public static final double PI = 3.14159;
    protected double r;
    public Circle(double r) { this.r = r; }
    public double circumference() { return 2*PI*r; }
    public double area() { return PI*r*r; }
    public double radius() { return r; }
}
```



Class Diagrams

Recall that abstract classes is used to describe abstractions. They are used to be the superclass of other classes and will not have any instance. The class icon for abstract classes is:



That is, the name of an abstract class in the class diagram is shown in italic[4]. Similarly, the name of abstract methods are also in italic type.

[4] For an abstract class which only defines operations, e.g. Java interface, you can annotate <<interface>> above the class name without italicizing anything

Class Diagrams

We can also use class diagram to represent the relationship between classes in the system. In UML, we can define the following relationships between classes:

- Association: just represents that two classes are related in the early stage of system analysis and design
- Generalization: "is a" (inheritance) relation between classes. Denote the superclass and the subclass
- Aggregation: the "part of" or "whole/part" relation
- Composition: physical containment. a stronger type of "part of"

Class Diagrams

The following figure shows an association relationship[5]:

The two classes are connected by a line. The number "1" and the asterisk($*$) symbol shows the multiplicity across the relationship, which means unlimited number of wheels can associated with just one vehicle. Here are some example of the multiplicity:

- 1: Exactly one
- $*$: Unlimited number, including zero or more
- 0..$*$: Zero or more
- 1..$*$: One or more
- 3..7: Three to seven. A specified range.

Note that in addition to the multiplicity, you can also include a textual label on the line to denote the name or the purpose of the relationship[6].

[5] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

[6] The textual label can also be used on other kinds of class relationships.

Class Diagrams

Association can have direction. In addition, the role of the two classes in an association can be denoted on the two ends. The direction of the association can illustrate the service provider and the client. See the following example:

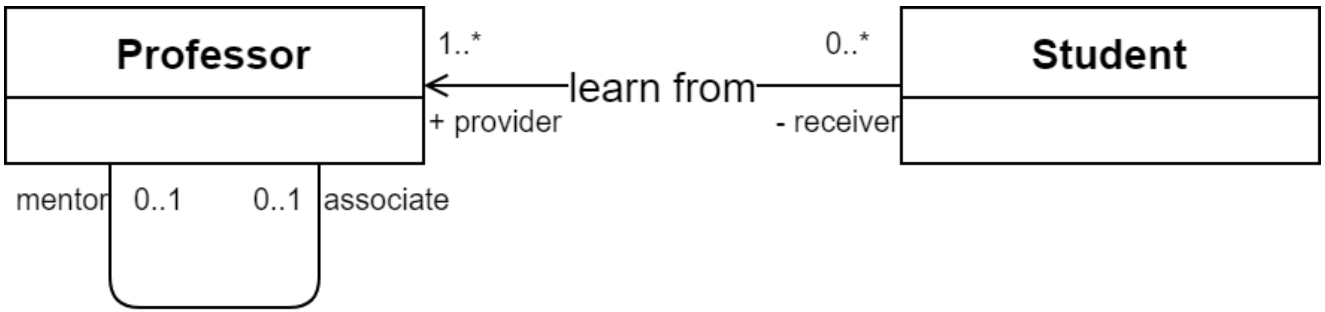


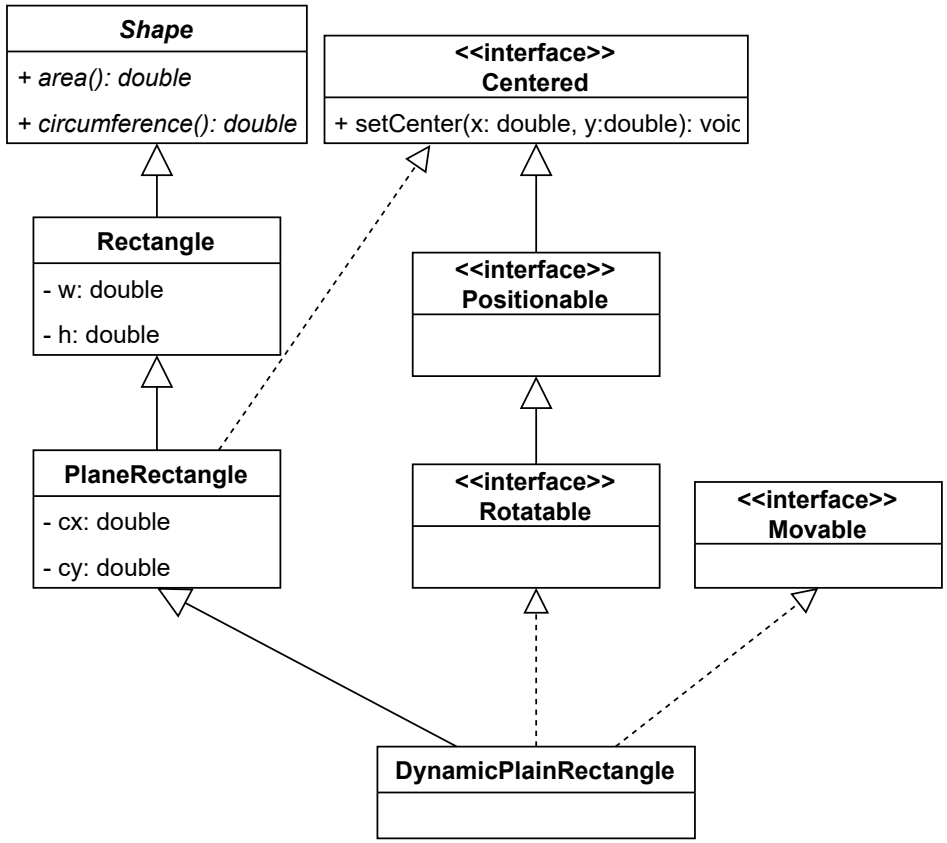
Figure: The two classes, Professor and Student, have association relationship.

In this example, we can see

- Kind of class relationship: learn from. "student" learns from "professor".
- The role name (associate end name): "provider" and "receiver".
- Visibility of the roles: "receiver" is private to the teacher.
- Reflexive association: A professor can be the mentor of another professor.

Class Diagrams

Generalization defines the hierarchy of the classes in the system. Here is an example[7]:



Generalization is denoted with a closed arrowhead. The arrowhead points to the superclass, and the opposite end is the subclass. For the case of interface implementation, you can use a dashed arrow. Note that there is no multiplicity on this kind of relationship.

[7] Refer to the topic of Java for the source code of this example.

Class Diagrams

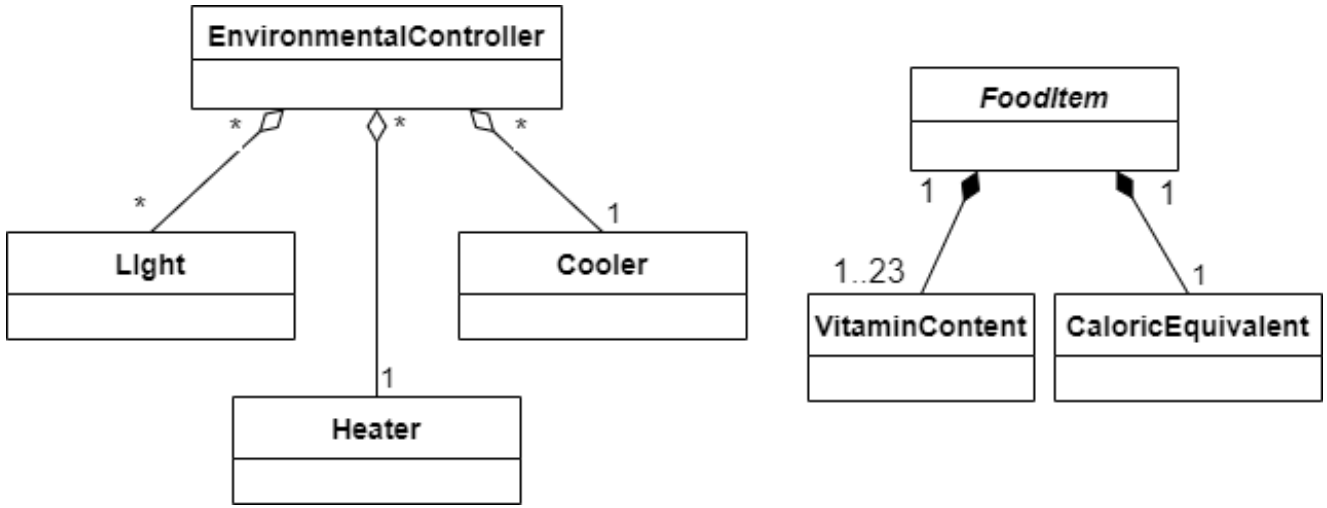


Figure: Aggregation(left) and Composition(right)[8]

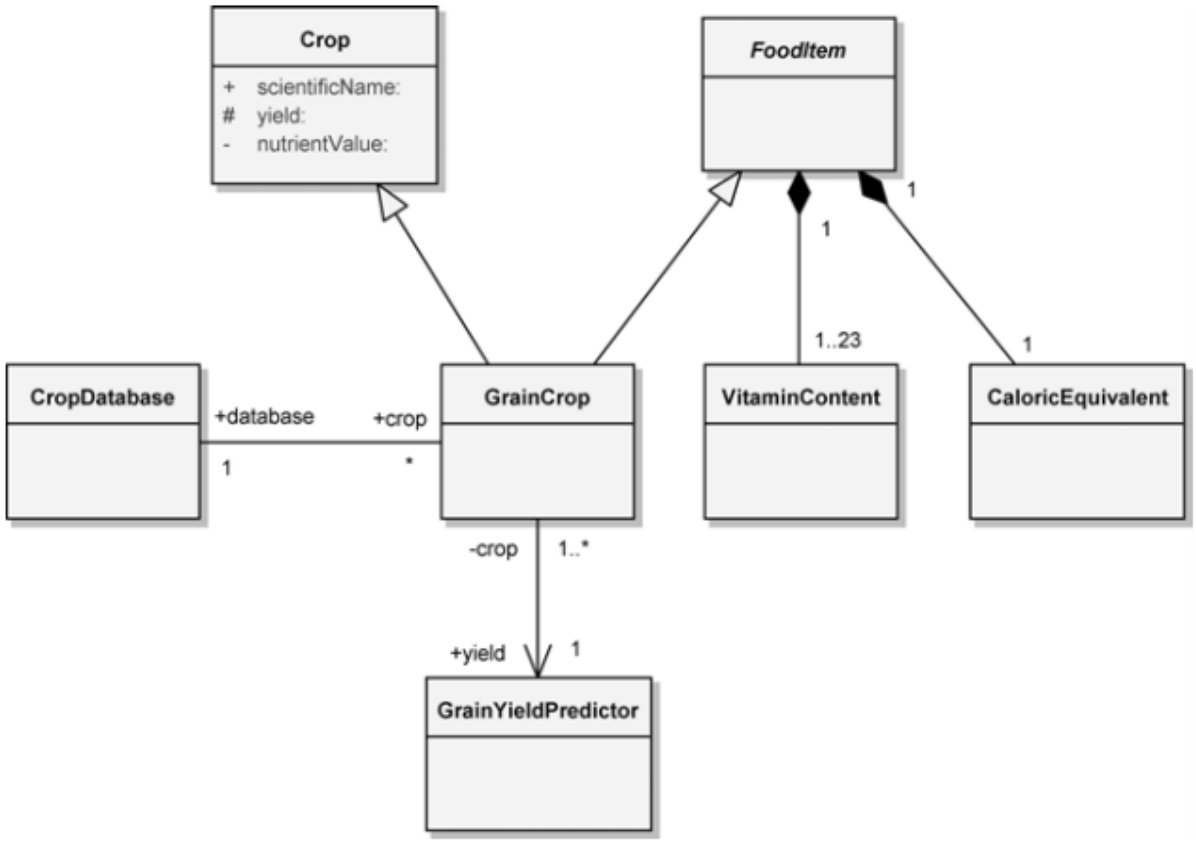
This figure shows the two kinds of aggregation relationships. The (shared) aggregation icon is denoted with an unfilled diamond at the end of the aggregate (the whole) with another end whose instance is part of the aggregate object. The composition icon appears with a filled diamond at the end of the aggregate. The multiplicity appears on the two ends.

Note that the $*$ in the aggregate (the whole) also means the two classes is not physical containment. On the other hand, for composition, the multiplicity on the end of aggregate cannot be zero.

[8] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Class Diagrams

You can draw more than one kind of relationships on one class diagram to illustrate the relationship of the classes in your system. Here is an example of the relationships between some of the classes in a hydroponics gardening system[9]:



[9] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

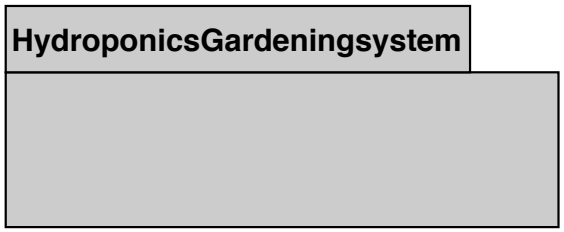
Package Diagrams

The UML package diagram is the primary means to organize the OOAD artifacts to represent the analysis of the problem space and the associated design. The benefit of organizing the OOAD artifacts includes:

- Provide clarity and understanding in a complex system development
- Support the concurrent model used by multiple users
- Support version control
- Provide abstraction at multiple levels (from systems to classes in a component)
- Provides encapsulation and containment

Package Diagrams

The figure below[10] gives a package diagram:

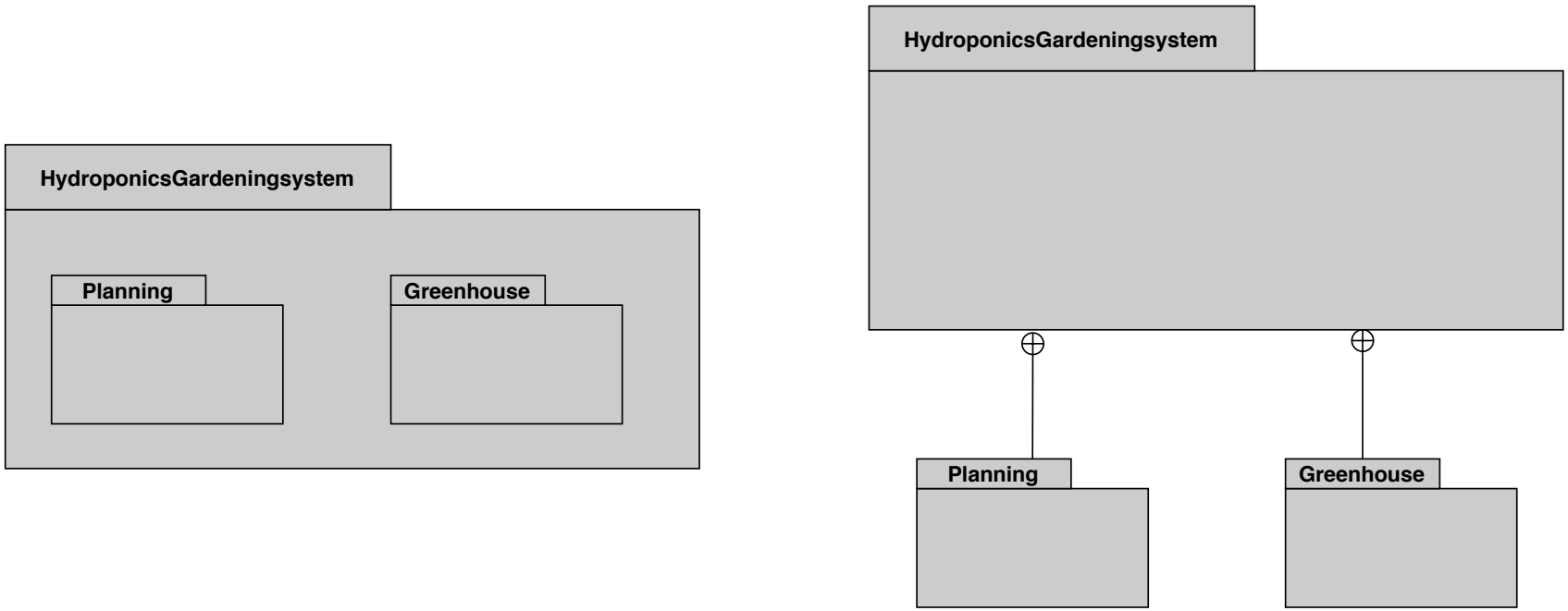


Package diagram has a rectangle with a tab on the top left. This figure provides a black-box perspective of this package without giving any components in it. In UML 2.0, the name of the package is placed within the rectangle if there is no UML elements in the package.

[10] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Package Diagrams

This figure[11] gives a package diagram which shows an example of two different ways to represent a package contains two elements which are also packages.



In the left one, the two packages, Planning and Greenhouse, are put inside the HydroponicsGardeningSystem; the right one is an alternate notation for the containment relationship.

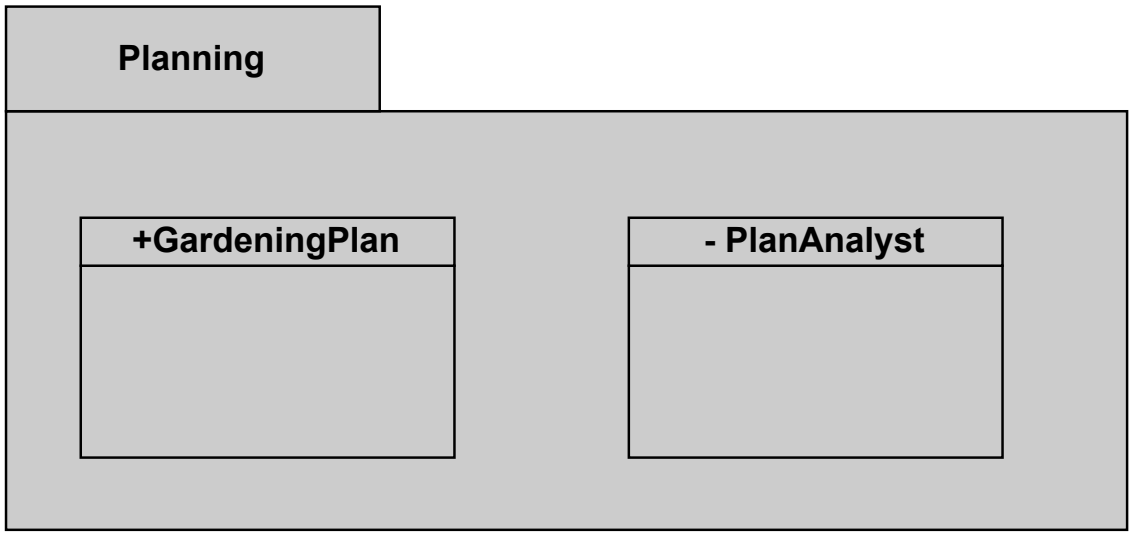
[11] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Package Diagrams

Some elements in a package may not be visible from the outside. The notations to show the visibility of the elements are:

- $+$: Public. This element is visible to the elements withing the package and to external elements
- $-$: Private. This element is only visible to the elements within the package

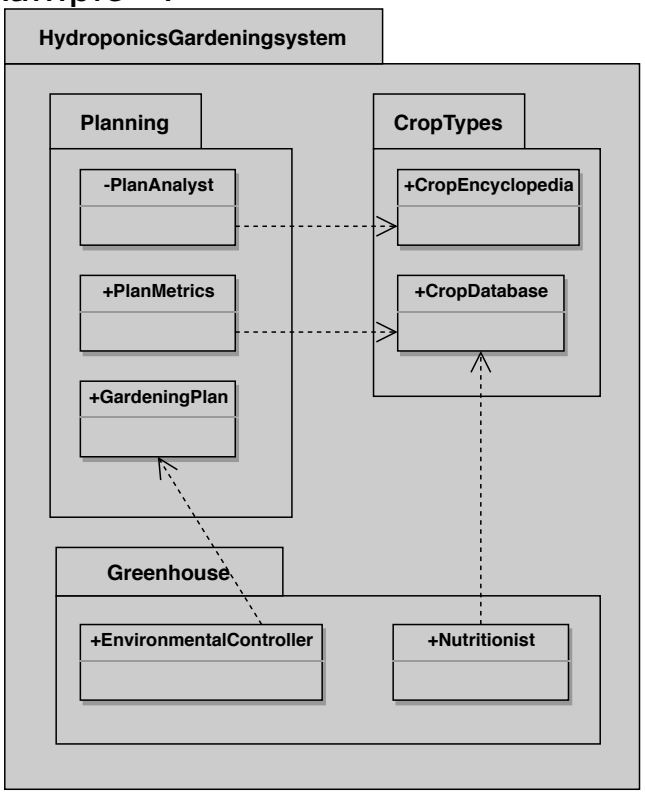
Here is an example[12]:



[12] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Package Diagrams

Package diagrams can show dependency relationships between elements. A dependency shows that an element requires another elements to fulfill its responsibilities in the system. Here is an example[13]:



This figure shows three packages and dependencies in the HydroponicsGardeningSystem:

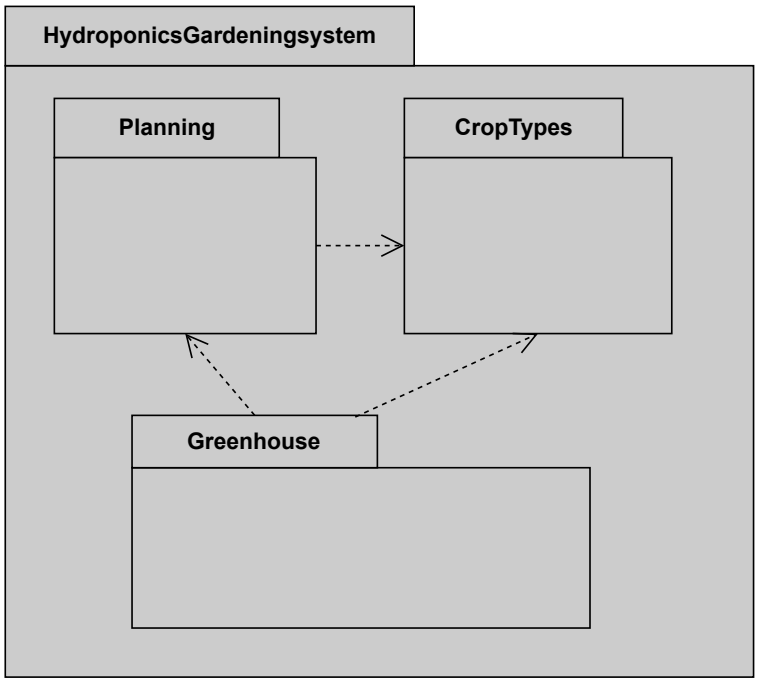
- Package Planning with one private class (PlanAnalyst) and two public classes (PlanMetrics and GardeningPlan) in it
- Package CropTypes with two public classes (CropEncyclopedia and CropDatabase) in it
- Package Greenhouse with two public classes (EnvironmentalController and Nutritionist) in it
- PlanAnalyst depends on CropEncyclopedia
- PlanMetrics depends on CropDatabase
- EnvironmentalController depends on GardeningPlan
- Nutritionist depends on CropDatabase

Dependencies are represented as a dashed arrow with an open arrowhead. The tail is the client element, and the head is the supplier. You can denote the type of dependencies inside the guillemets (<< >>). Package-level dependencies include "import", "access", and "merge"; dependencies between elements of packages includes "trace", "derive", "permit", "refine", and "use".

[13] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Package Diagrams

You can aggregate several dependencies between the elements of packages into a package-level dependency. As the figure below. Elements in Planning import some elements in CropTypes, so there is a label <<import>>[14].



15

[14] If the type of the dependencies are different, it will not be labeled

[15] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Activity Diagrams

Activity diagrams provide visual depictions of the flow of activities. This diagram shows what activities are performed and who is responsible for the performance of the activities.
The elements of an activity diagrams include

- action nodes
- control nodes
  - initial and final nodes
    - final nodes include activity final and flow final nodes
  - decision and merge
  - fork and join
- object nodes

Activity Diagrams: actions

An action is represented using a rounded rectangle in an activity diagram. See the following example:

Figure: An action[16].

Activities can contain many actions, and an action can be decomposed into sub-activities. This figure shows a "check tank levels"(檢查儲存槽內存量) action in the hydroponics gardening system. Note that there is a rake symbol in the lower right corner. This symbol indicates that this action calls an activity. In practice, this symbol is used only when that activity has been actually defined. That is, you can draw another activity diagram for it.

[16] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Activity Diagrams: Starting and Stopping

We have to show the start and stop points of the process flow of an activity. In activity diagrams, the starting point is denoted as a solid dot, and the stopping point (activity final node) is shown as a bull's eye. See the example:

Figure: An activity diagram with initial and final nodes[17].

Another type of final node, the flow final node, is denoted by a cycle with an "X" in it:

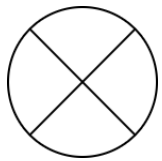


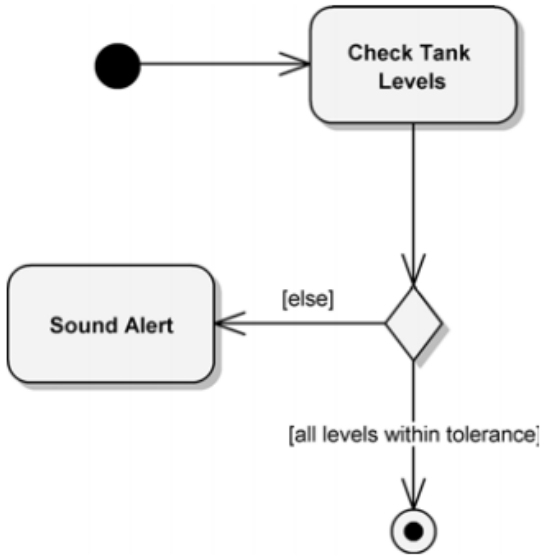
Figure: The flow final node.

The flow final node is used to stop a single flow without stopping the entire activity. For example, the activity may have many inputs but only some of them will be accepted. By directing the rejected inputs to the flow final node, the activity can continue.

[17] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Activity Diagrams: Decision and Merge Nodes

Decision and merge nodes represent the flow under decisions in an activity. The icon is represented by a diamond shape with incoming and outgoing flows. Here is an example[18]:

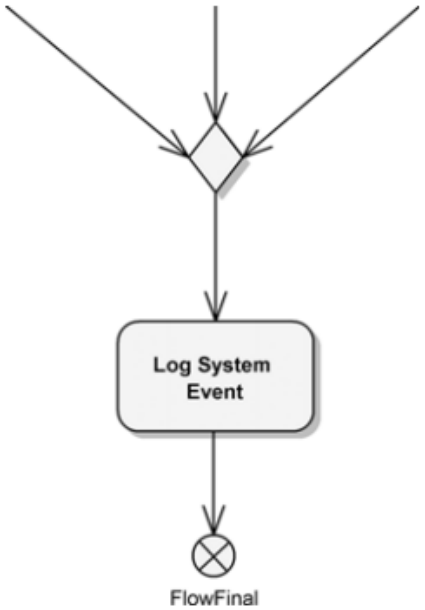


A decision has a single incoming flow and several outbounding flows. Each outbounding flow has a boolean condition, or guard condition, placed inside brackets indicating which outgoing path is selected. As the figure shows, [all levels within tolerance] and [else] are the two conditions. This figure indicates that if the levels in the tank is not within tolerance, the flow goes to the [else] path.

[18] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Activity Diagrams: Decision and Merge Nodes

The merge node is also represents by a diamond shape. It has multiple input and only one output flow[19]:

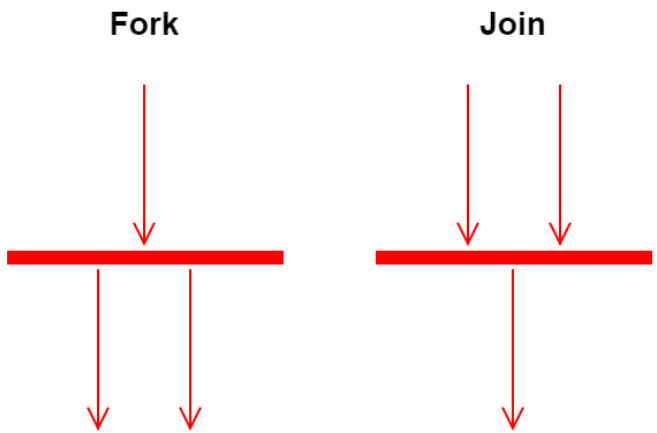


This figure shows a merge node with three incoming flows, an action, and a flow final node.
Note that when reaching a decision node, only one of the outbounding flow can be taken. In addition, there is no waiting or synchronization at a decision and a merge node.

[19] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

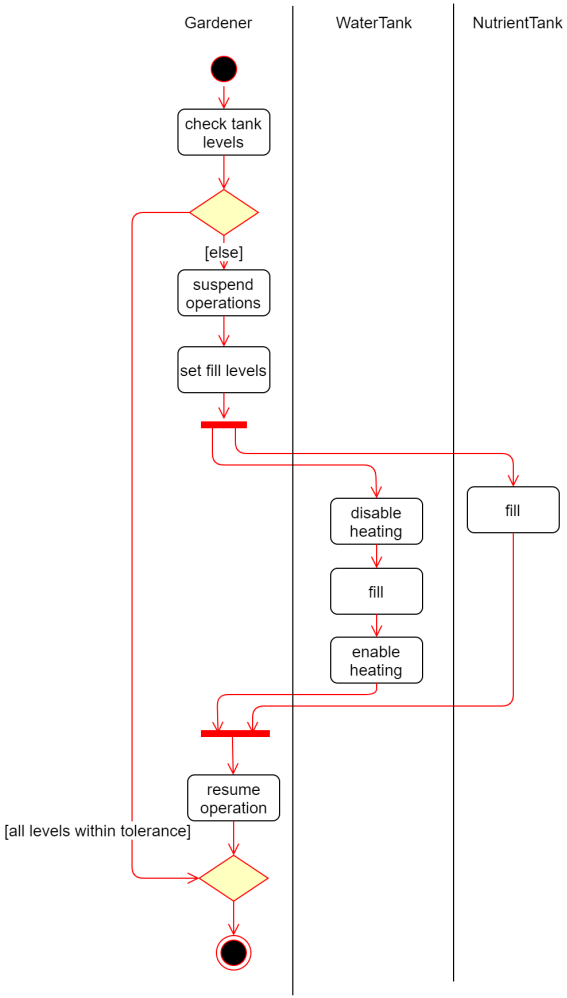
Activity Diagrams: Forks and Joins

Fork and join nodes are similar with decision and merge nodes respectively. The difference is that a single flow goes into fork will result in multiple outbound flows, and these flows occur concurrently.
Similarly, both join nodes and merge nodes have multiple incoming flows and single outbound flow. However, all the incoming flows must be complete before the outgoing flow stars. That is, fork and join are with concurrency.
The fork and join nodes are shown in the figure below:



In this figure, we only put two incoming/outbounding flows.

Activity Diagrams: Partitions



A partition is used to indicate where the responsibility lies for performing specific activities. The partition can be an organization, other systems or subsystems, or objects in the application.

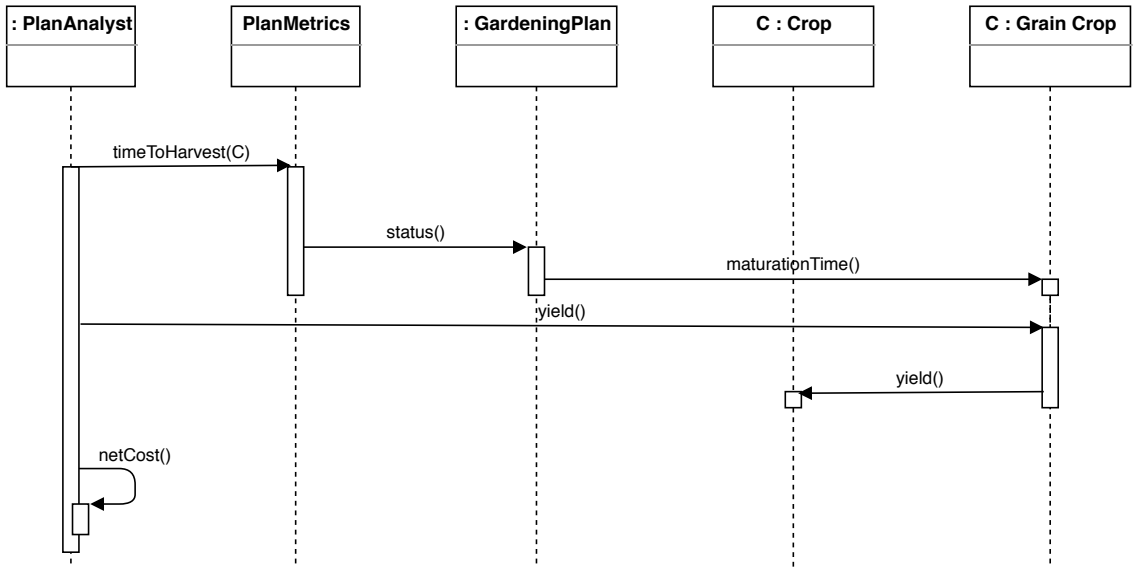
The figure in the left shows the activity with three partitions: Gardener, WaterTank, and NutrientTank. The flow of this activity is clear: first check the tank levels. If the all the levels are within tolerance, the activity ends. Otherwise, the gardener sets sets the required level of the water and nutrient, and ask WaterTank and NutrientTank to fill the tank. From activity diagrams, we can easily find not only what but also who performs specific actions in the activity.

20

[20] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Sequence Diagrams

A sequence diagram is used to trace the execution of a scenario. The advantage of sequence diagrams is that it is easy to read the passing of messages in relative order. See the following example:

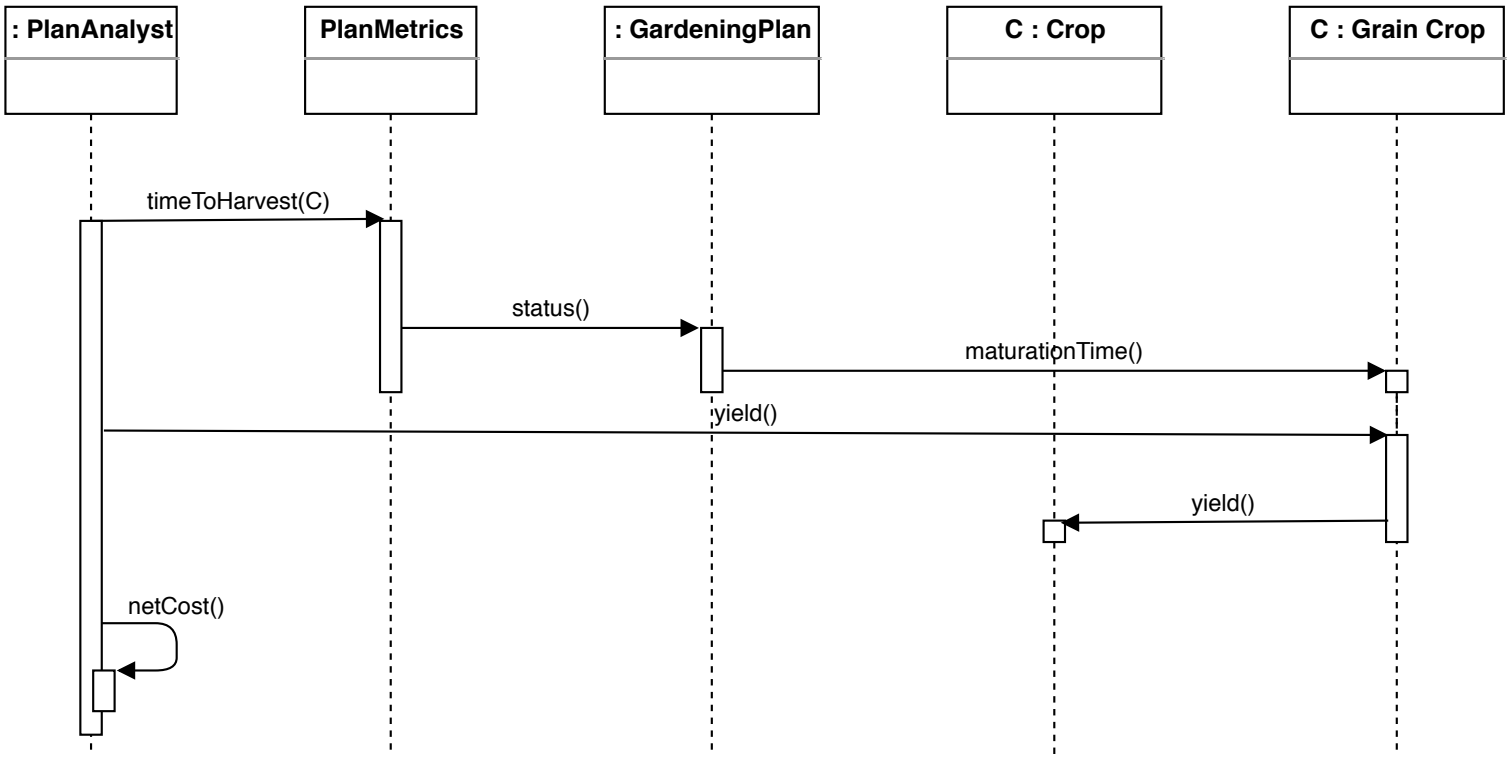


There are five entities (objects) of interest in this figure[21]. The name of objects can be written as:

- objectName:ClassName: give both the object name and the Class of this object
- objectName: only give the object name. Its class is considered anonymous
- ClassName: only gives the class name. This object is said to be anonymous

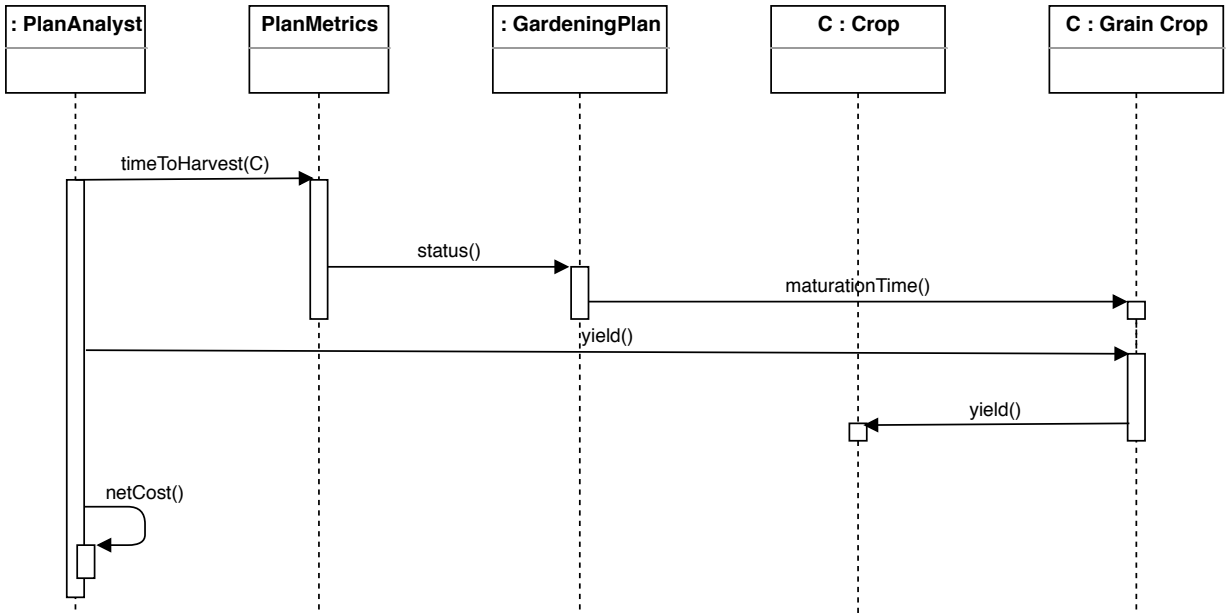
[21] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Sequence Diagrams



The five entities are: planAnalyst, PlanMetrics, GardeningPlan, Crop, and Grain Crop. You can see the interactions between these entities. These interactions are shown horizontally across the top of the diagram.

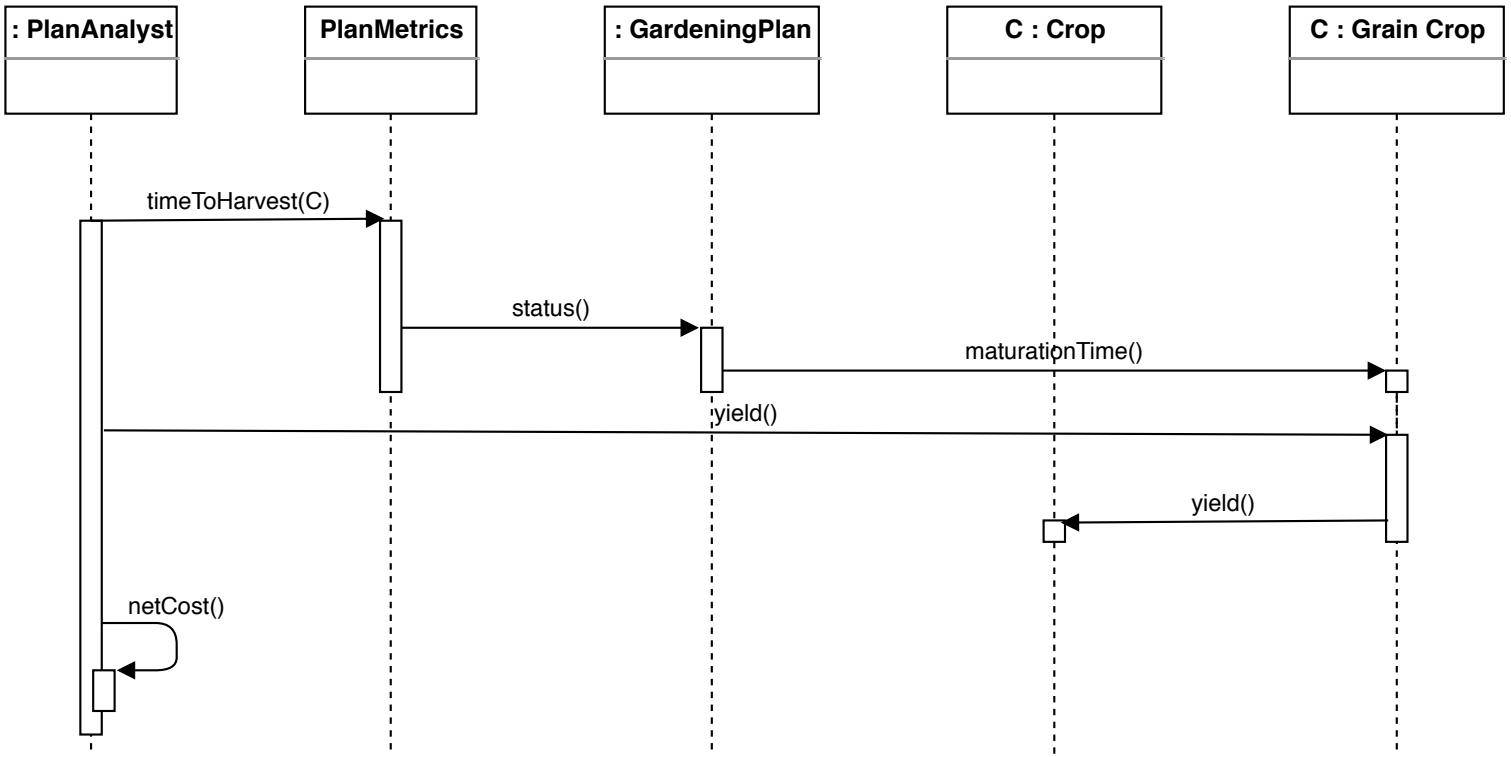
Sequence Diagrams



Besides the entities(objects), this figure shows the basic elements in a sequence diagram:

- Lifelines: the dashed vertical lines indicate the existence of the object
- Message: the horizontal lines with arrow drawn from the sender to the receiver denote events or the invocation of operations. The ordering of messages is shown by their vertical positions
- Activation bar: the bar on the lines shows when the object is active in the interaction

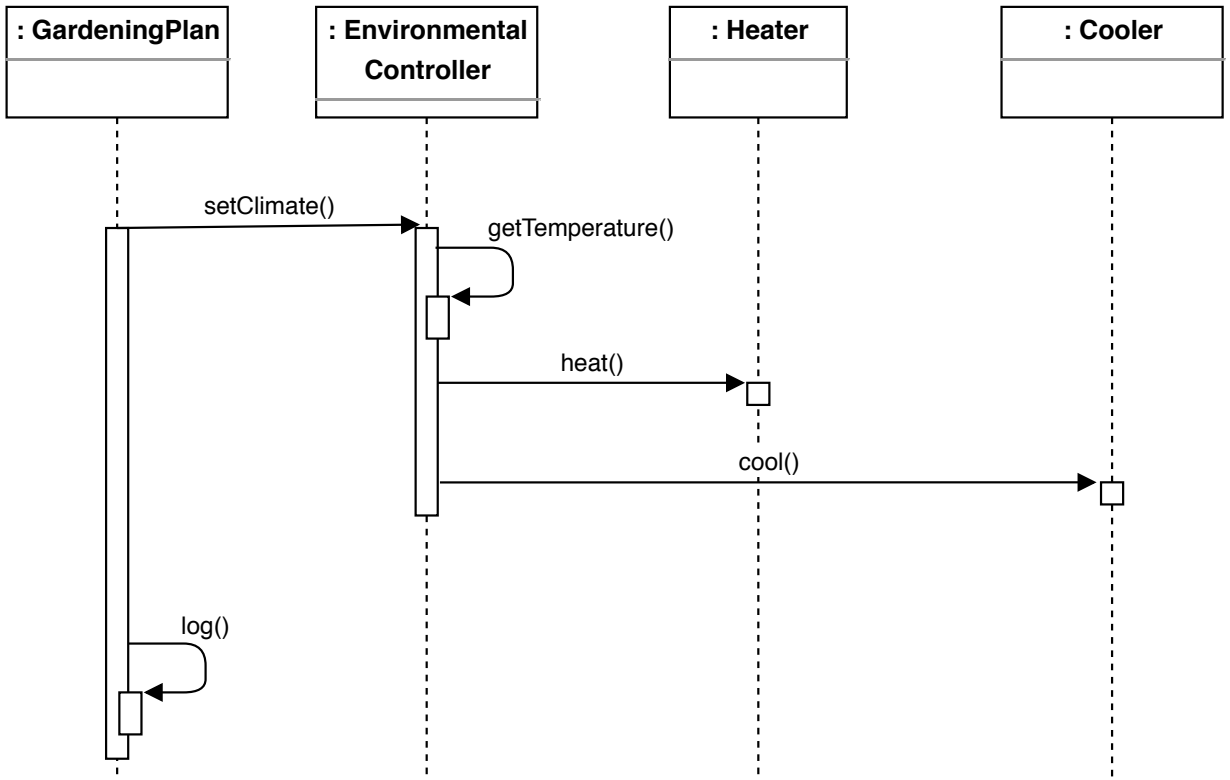
Sequence Diagrams



Here is the first part of the sequence: An anonymous object of PlanAnalyst calls PlanMetrics's operation, timeToHarvest. PlanMetrics uses another anonymous object of GardeningPlan's operation status. This anonymous object will make an object of GrainCrop to be created.

Sequence Diagrams

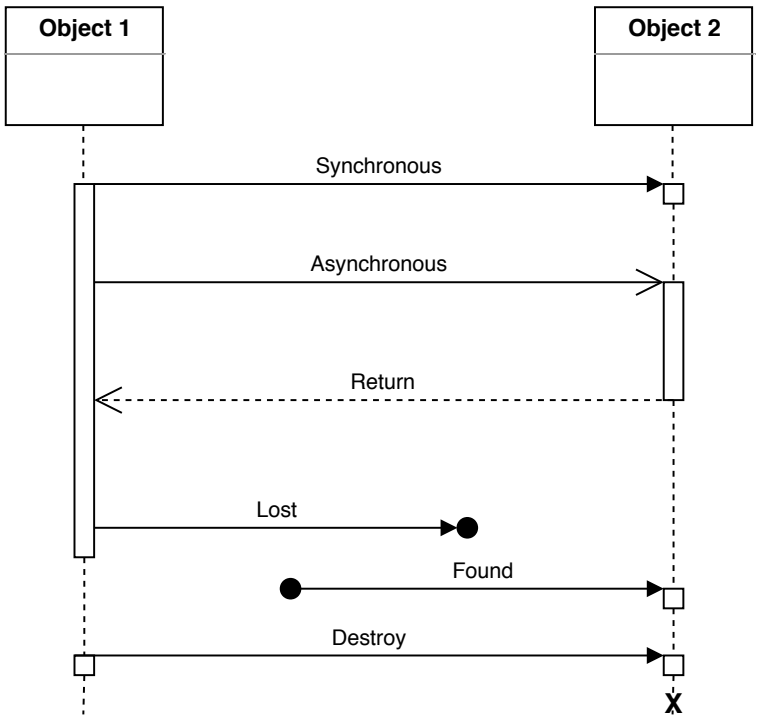
Here is another example shows the object GardeningPlan carries out a climate plan:



This example[22] shows the orderings of the operations invoked. We can see operation setClimate() invokes other methods, which in turn call other methods. The control is eventually return back to the GardeningPlan object.

[22] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Sequence Diagrams



The figure in the left gives the notation of different types of messages:

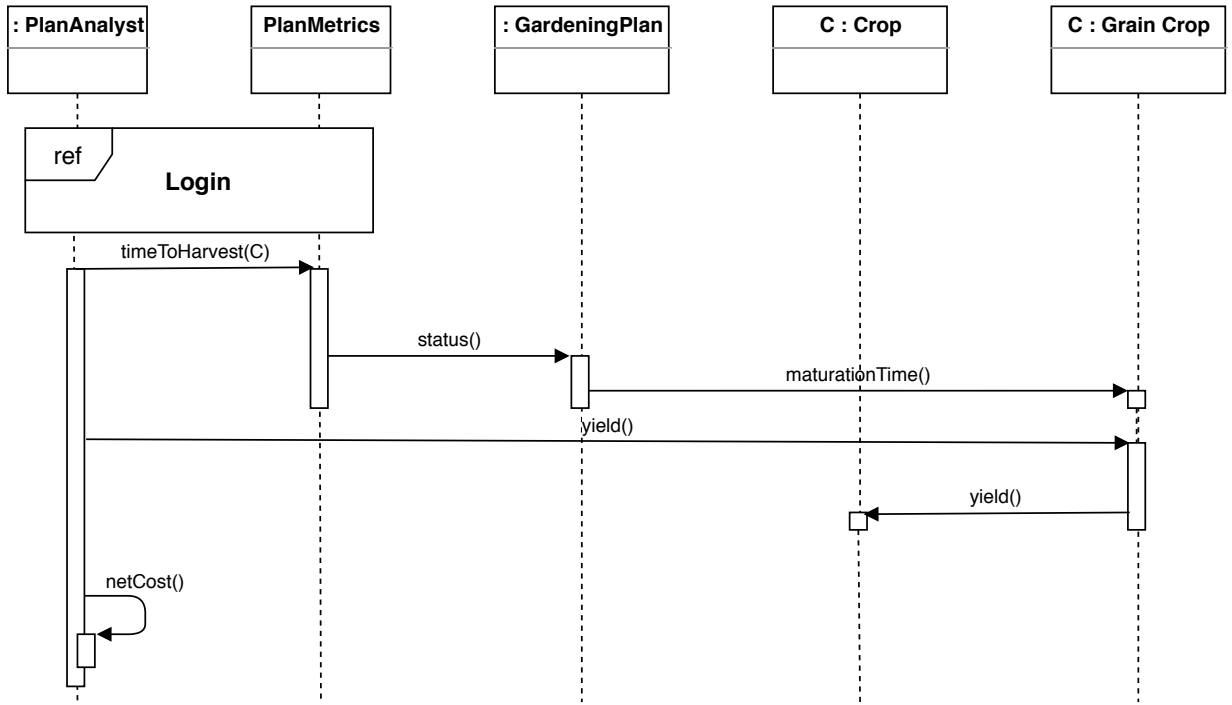
- Synchronous message (usually an operation call): a solid line with a filled arrowhead
- Asynchronous message: a solid line with an open arrowhead
- Return message: a dashed line with an open arrowhead
- Lost message (a message that doesn't reach its destination): a synchronous message that terminates at and endpoint
- Found message (a message whose sender is not known): a synchronous message that originates at an endpoint symbol
- Destroy: a synchronous message with an $\mathbb{X}$ at the end of the lifeline

If the object to be destroyed is involved in a composition relationship, the other involved objects may also be destroyed.

[23] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Sequence Diagrams

A complex scenario may involve too many objects and operations to shown in a sequence diagram. UML (2.0) provides constructs to simplify complex sequence diagrams. See the following example:

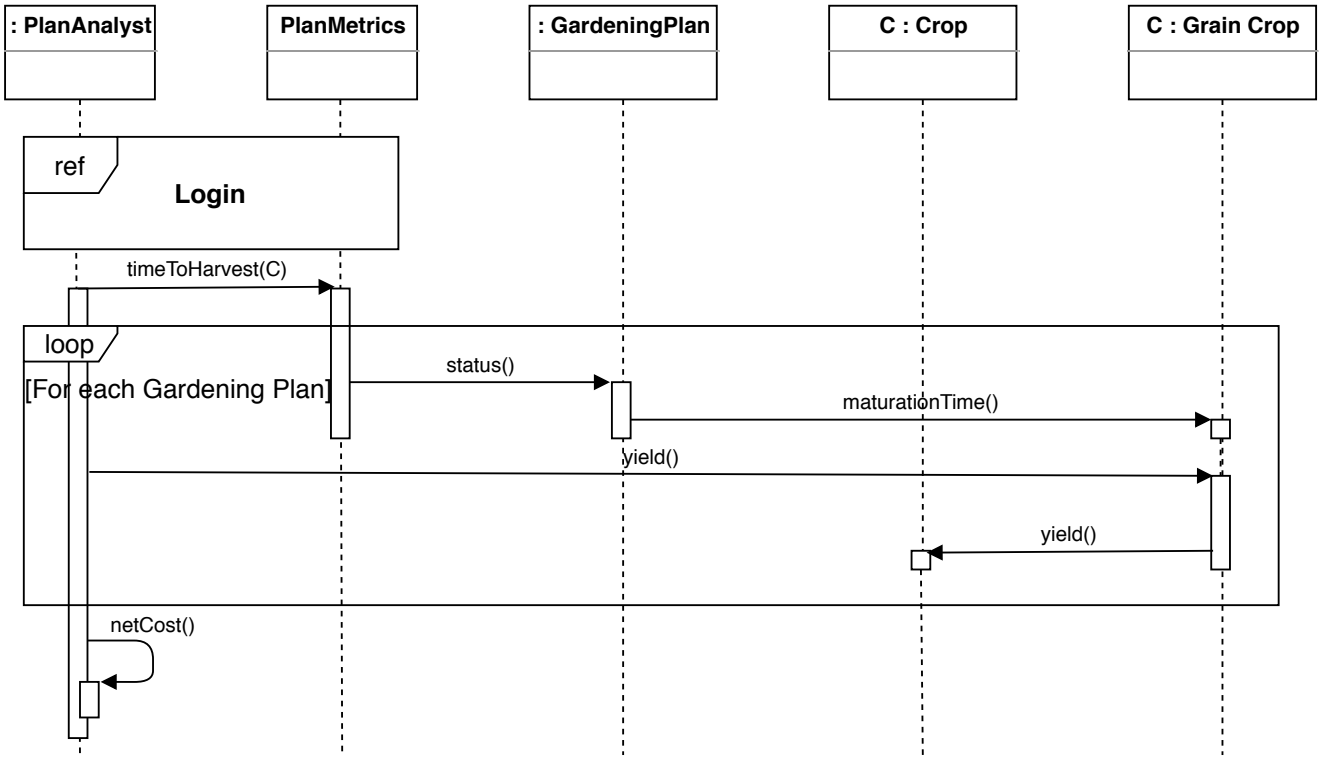


This figure[24] shows an interaction use, which is a frame with label "ref". An interaction use is a way to indicate that we reuse an interaction that is defined elsewhere. In this example, a login sequence is required in this scenario before PlanAnalyst uses the system.

[24] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Sequence Diagrams

You can also define flow control structures in sequence diagrams. Here is an example:

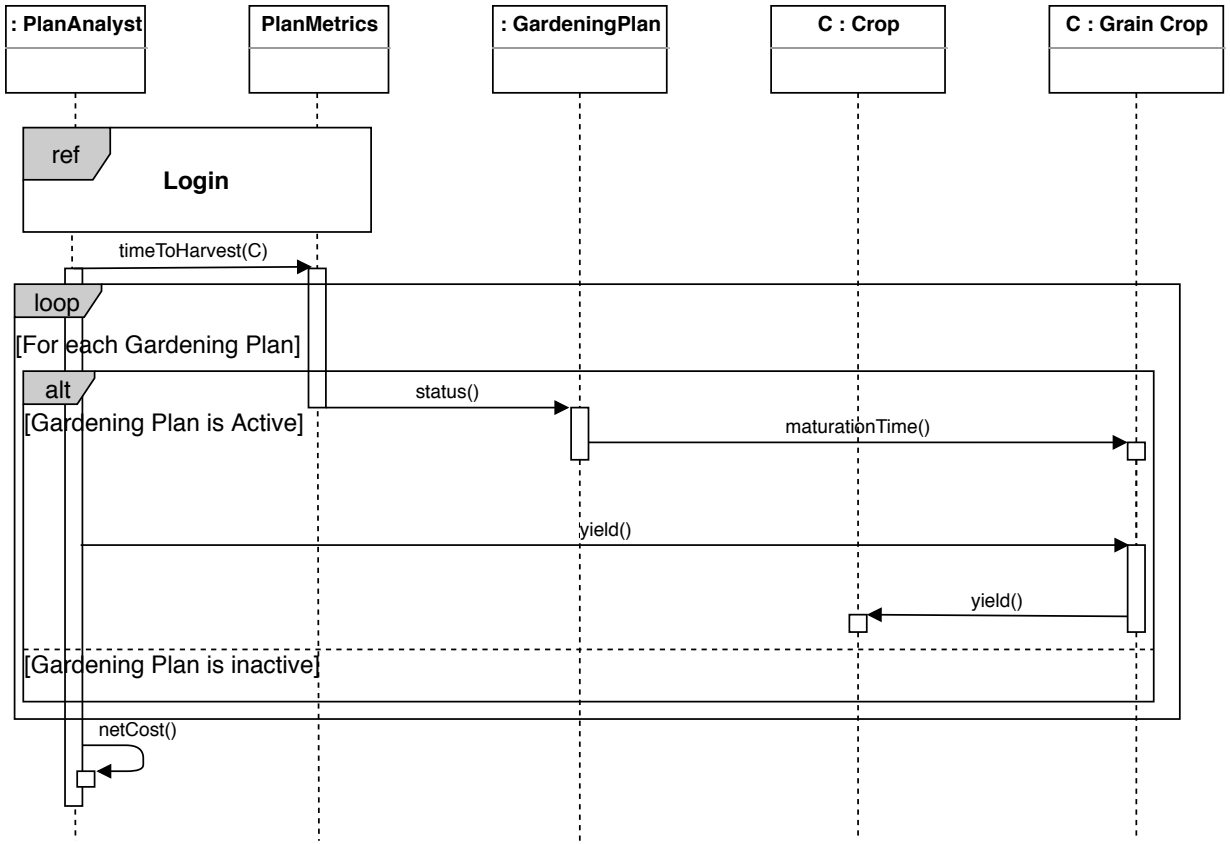


This example[25] defines a loop that is executed by each gardening plan.

[25] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Sequence Diagrams

If there is more than one gardening plan, and we only want to apply the sequence to those who are active. The sequence diagram can be modified as follows[26]:



The alt frame defines an alternative operation. The sequence is only applied to active gardening plans. Note that the alt frame is separated into two regions, each with its own condition. The operations are invoked (or messages are sent/received) only on the region that the gardening plan is active.

[26] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007