PlantUML: Your Flexible Friend

Neville Churcher
2022

Introduction

Once upon a time[1], in a land far, far away, there was a library called Graphviz and a bunch of tools with cool names like “dot”, “neato” and “lefty”. This software was initially developed by Stephen North and others at AT&T Labs and continues to evolve. Graphs — the node and edge kind — can be specified in a simple textual language called DOT[2]. The various tools can then process the graph descriptions to produce “nice” layouts for graph visualisation applications[3]. The range of output formats (PDF, SVG, …) makes this a convenient way to generate images for publication.

The success of DOT and the associated software has led to it being incorporated in many other systems. In our case, we’ll be most interested in PlantUML which — as its name suggests — can generate the various diagram types that make up the Unified Modelling Language (UML) from a DOT-like textual description. PlantUML can also generate a number of other (non-UML) diagram types. If you took COSC265 last year then you will have seen Entity-Relationship diagrams produced by PlantUML.

Why should I care?

PlantUML is a handy tool — and the price is right!

At this point, it seems certain that this semester’s exams will be online. It is likely that the SENG301 exam will include questions requiring you to understand, create or modify UML class diagrams. Familiarity with PlantUML will be assumed. We’ll also encounter it in labs and lectures.

PlantUML context

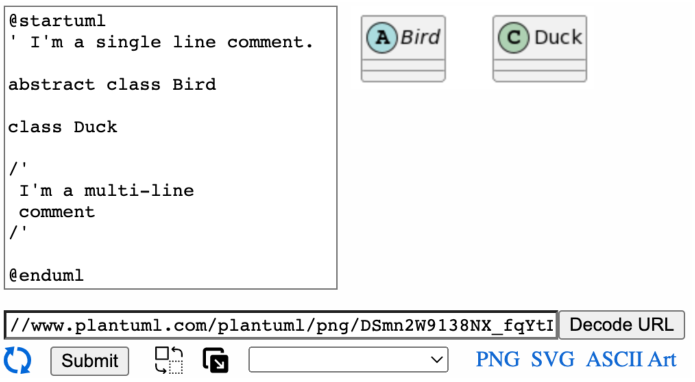
Extensive documentation is available in PDF format at https://plantuml.com/guide but the online info at https://plantuml.com is sufficient for most purposes. We will be primarily interested in class diagrams but you may find some of the other diagram types useful too.

PlantUML can be downloaded, and used in a variety of ways. However, for our purposes it is generally easiest to use the online server — click the link at https://plantuml.com — and do everything in the browser. A more sophisticated web editor, and an IntelliJ plugin and a VSCode plugin are also available.

[1] We’re talking around 1990 — wouldn’t it be great to think that the software we write would still be used decades later!
[2] A great example of a Domain Specific Language (DSL)
[3] See https://graphviz.org/gallery/ for some examples

You will see:

- A text area where you enter your diagram specification
- A "Submit" button to tell PlantUML to perform layout with the current specification
- A "Change layout" button that determines whether the resulting diagram appears below or beside the specification
- Buttons to select the desired output format so you can save the result (usually PNG for us)



```
@startuml
' I'm a single line comment.

abstract class Bird

class Duck

/'
 I'm a multi-line
 comment
/'

@enduml
```

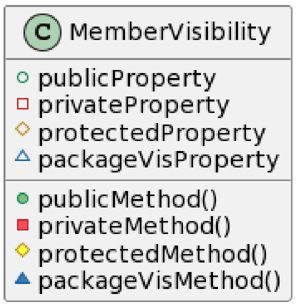
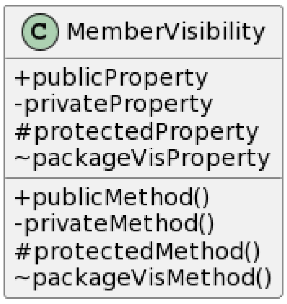
PlanUML has limited error reporting functionality but you'll soon learn the common errors.

Things to note include:

- Our typical workflow will be to enter the diagram specification (or paste a previously-saved one) and tweak it until we are satisfied with the diagram. We'll then save the diagram (or paste it into another document) and also save the specification (typically with the aid of a simple text editor) for later re-use.
- The diagram specification goes between the @startuml and @enduml tags.
- Single line comments begin with a single quote. Multi-line comments begin and end with a slash and single quote (/').
- The keyword "class" creates a basic diagram element with a label, boxes for properties and methods, and a "C" icon to show it's a class — there are corresponding icons for enums, interfaces and abstract classes too.
- The diagram specification includes two main categories of statements: some (e.g. "class") describe the UNL elements and their associations; others (e.g. "hide," "show," and "skinparam") control presentation detail.

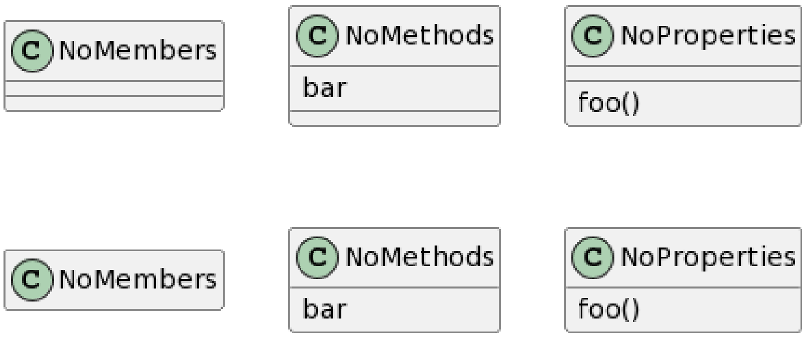
Visibility

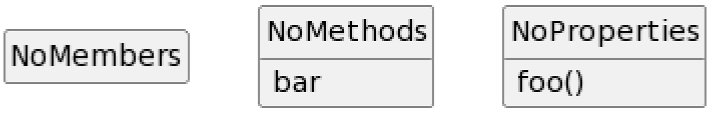
Member visibility can be indicated using either coloured icons (hollow for properties, filled for members) or classic UML characters. The option is controlled by the "skinparam" statement — see the PlantUML documentation for other display options.

'skinparam classAttributeIconSize 0
 	
class MemberVisibility {
 + publicProperty
 - privateProperty
 # protectedProperty
 ~ packageVisProperty
 + publicMethod()
 - privateMethod()
 # protectedMethod()
 ~ packageVisMethod()
}
	 MemberVisibility
	publicProperty
	privateProperty
	protectedProperty
	packageVisProperty
	publicMethod()
	privateMethod()
	protectedMethod()
	packageVisMethod()
		 MemberVisibility
		+publicProperty
		-privateProperty
		#protectedProperty
		~packageVisProperty
		+publicMethod()
		-privateMethod()
		#protectedMethod()
		~packageVisMethod()

Hiding things

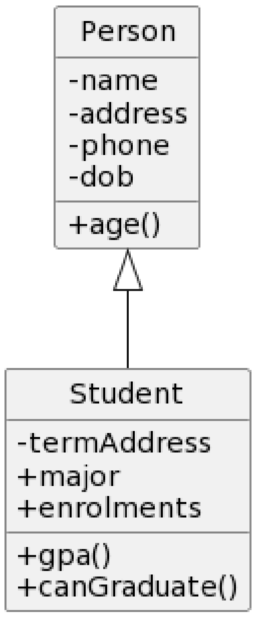
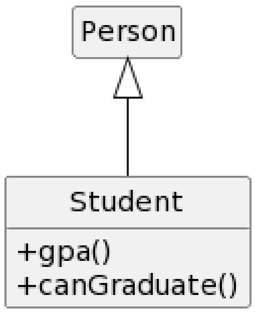
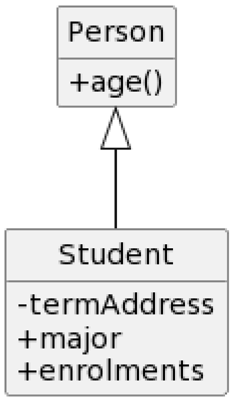
We generally prefer to hide empty property/method boxes. In the first row of the table below we see the effect of uncommenting a "hide empty members" statement. Another common use of "hide" is to remove the icons denoting classes, interfaces etc. with "hide circle" as in the second row. See the PlantUML documentation for more detail on "hide".

'hide empty members
class NoMembers {
}
 	
class NoMethods {
 bar
}
 	
class NoProperties {
 foo()
}
	 NoMembers
	NoMethods
	bar
	NoProperties
	foo()
	NoMembers
	NoMethods
	bar
	NoProperties
	foo()

hide circle
hide empty members
class NoMembers {
}
class NoMethods {
 bar
}
class NoProperties {
 foo()
}
	

A cluttered diagram with too much text severely limits the ability of a class diagram to communicate essential design features to the reader. Consequently, we may want to hide attributes (fields) and/or methods that are not essential for a diagram. For fine control we can hide/show attributes or methods — either globally or for particular classes. The following example illustrates some of the possibilities.

		hide attributes hide Person methods	hide fields show Student fields hide Student methods

class Person {
 - name
 - address
 - phone
 - dob
 + age()
}
 	
class Student {
 - termAddress
 + major
 + enrolments
 + gpa()
 + canGraduate()
}
 	
Person <|-- Student
	 Person
	-name
	-address
	-phone
	-dob
	+age()
	Student
	-termAddress
	+major
	+enrolments
	+gpa()
	+canGraduate()
			 Person
			+age()
			Student
			-termAddress
			+major
			+enrolments

Adding detail

The following example illustrates specifying stereotypes, static & abstract members, placement of notes on classes & members, and the specification of parameter & return types.

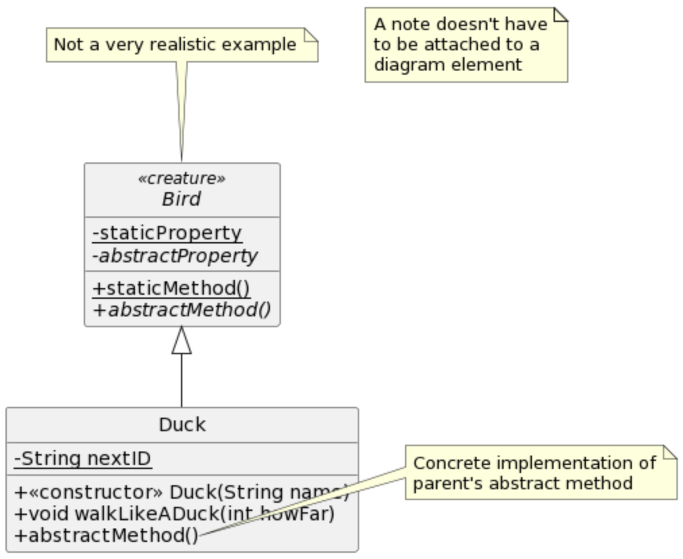
Some experimentation may be needed to achieve a “nice” layout if there are a large number of notes. Notes may be attached to attributes or methods, classes, associations or may float freely. See the PlantUML documentation for more detail.

```
abstract class Bird <<creature>> {
 - {static} staticProperty
 - {abstract} abstractProperty
 + {static} staticMethod()
 + {abstract} abstractMethod()
}
note top
Not a very realistic example
end note

class Duck {
 - {static} String nextID
 + <<constructor>> Duck(String name)
 + void walkLikeADuck(int howFar)
 + abstractMethod()
}
note right of Duck::abstractMethod
Concrete implementation of
parent's abstract method
end note

note as FloatingNote
A note doesn't have
to be attached to a
diagram element
end note

Bird <|-- Duck
```

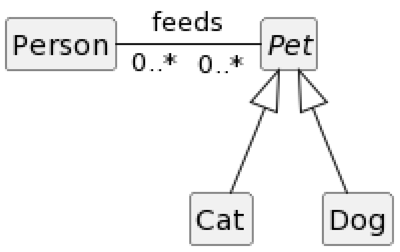

Not a very realistic example
A note doesn't have to be attached to a diagram element
«creature» Bird
-staticProperty
-abstractProperty
+staticMethod()
+abstractMethod()
Duck
-String nextID
+«constructor» Duck(String name)
+void walkLikeADuck(int howFar)
+abstractMethod()
Concrete implementation of parent's abstract method

Associations

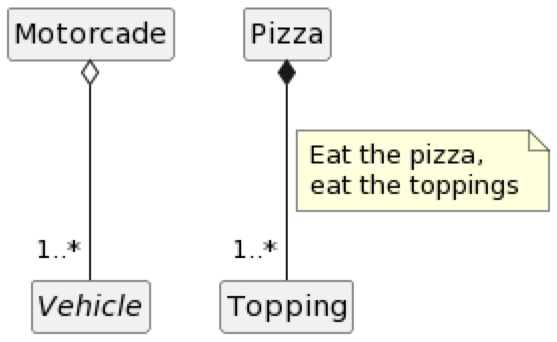
The last line in the previous example specifies that Duck is a child of Bird. Binary associations are specified by stating the element at each end, whether the line is solid (“--”) or dashed (“..”), and — optionally — a symbol at each end. Labels and notes may also be attached to associations.

The following examples show some common constructs.

```
class Person
abstract class Pet
class Cat
Pet <|-- Dog
Pet <|-- Cat
Person "0..*"- "0..*" Pet : feeds
```



```
class Motorcade
abstract class Vehicle
class Pizza
class Topping
Motorcade o-- "1..*" Vehicle
Pizza *-- "1..*" Topping
note on link
Eat the pizza,
eat the toppings
end note
```



Packages and interfaces

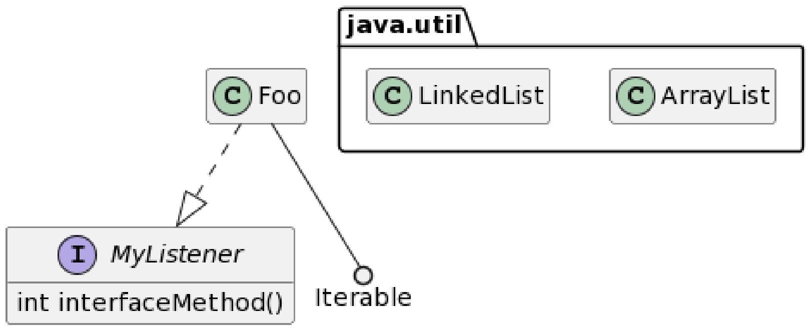
Packages can be specified to group together corresponding elements. Associations between package members or packages themselves can be specified.

Interfaces can be specified similarly to classes for a full description. Alternatively, the "lollipop" notation can be used.

The following example shows some possibilities.

```
package java.util {
 class ArrayList
 class LinkedList

}
interface MyListener {
 int interfaceMethod()
}
class Foo
Foo --() Iterable
Foo ..|> MyListener
```



Layout control

One of the (good or bad, depending on your point of view) consequences of using a layout algorithm is that we don't have total control over the placement of diagram elements. However, there are a few things we can do to improve — or at least change — the result. These should be considered a last resort — our best plan is to let the layout algorithm do its thing and only make minor adjustments if absolutely necessary.

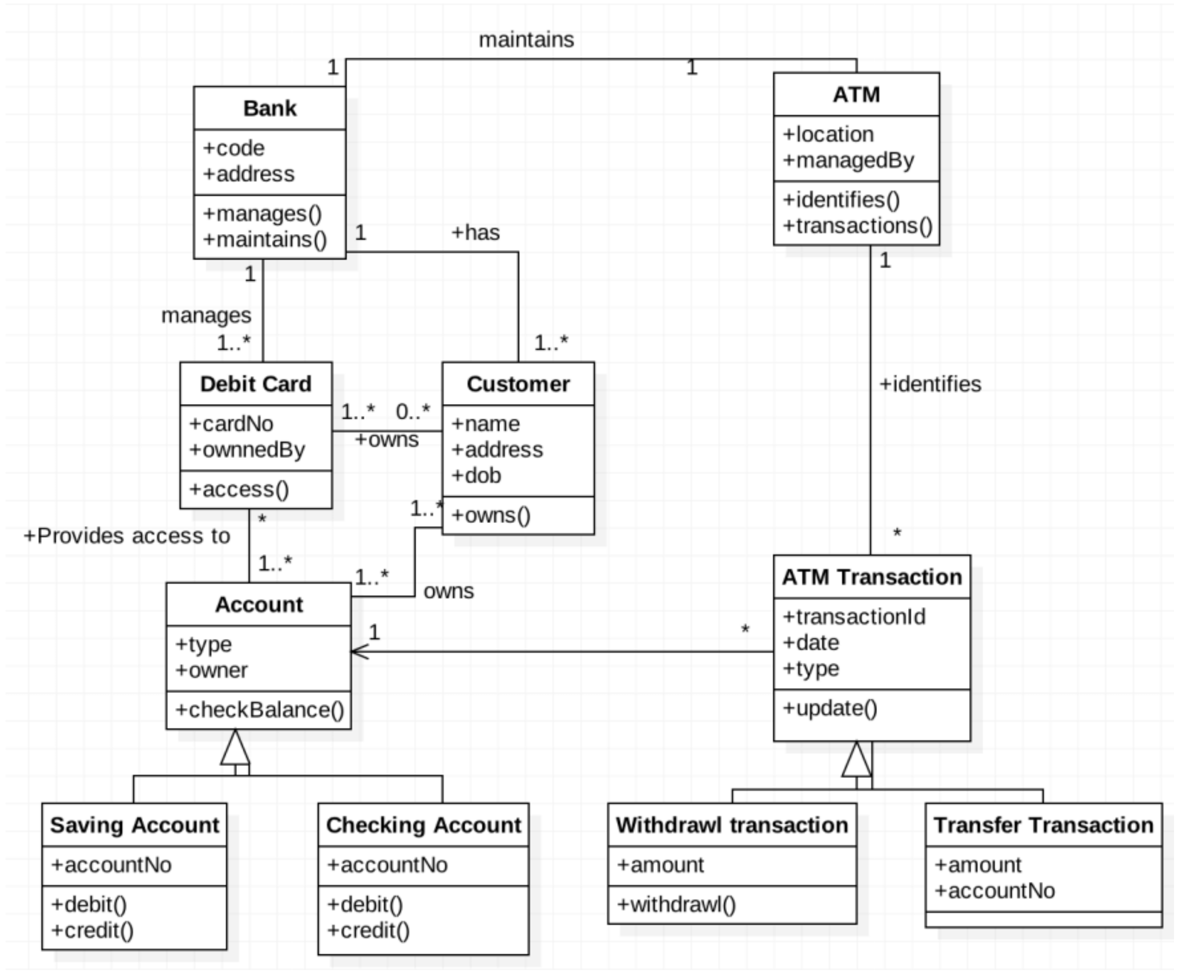
- Connections can be made longer or shorter by changing the number of symbols in the corresponding specification (e.g. "---" will tend to produce a longer edge than "--"). This also makes it easier for connections to bend to avoid crossing other elements.
- Changing the order of parts of the specification can also influence the results.
- You can use "together" to signal that objects should be grouped nearby if possible (e.g. "together { A B C}")
- Another technique that can be useful is the use of "hidden" connections (e.g. "Foo -[hidden] Bar"). This can help when vertical/horizontal layout changes are desired. Hidden edges contribute to the layout constraints but have no visible presence.

For further information, see the PlantUML documentation.

Exercises

If you haven't already done so, now's the time to try PlantUML for yourself.

1. Consider the following UML class diagram, taken from https://www.edrawsoft.com/example-uml-class-diagram.html (originally from University of Michigan-Dearborn).



   a. Use PlantUML to reproduce the diagram.
   b. What, if any, changes to the original diagram would you suggest? Discuss your suggestions with other students and/or your tutor and update your diagram as appropriate.
2. A motorcade consists of a pair of motorcycle outriders, one or more police cars, and one or more limousines. A motorcade is assembled for important occasions, such as delivering the president from the airport to the opera house. After the journey, the motorcade breaks up and its constituents are available for inclusion in subsequent motorcades.
3. Choose an OO design you are familiar with — your SENG202 assignment, SENG202 project or current SENG302 project are likely to be suitable — and use PlantUML to produce the corresponding UML class diagram.
   a. How do you decide what to include/omit from the diagram?

   b. What are the advantages & disadvantages of PlantUML compared to other UML tools you are familiar with, general drawing applications, or pencil & paper?

4. Experiment with the third-party web editor and IDE plugins mentioned on page 1. Which (if any) will you use in your own future work?