Lecture - Modular design and UML

Modularity: What and why?

(Most of) this lesson is about generate principles of writing software, not specific to Java.

One of the principal strengths of class inheritance is that it helps in enforcing *modularity*.

Modularity is a way to manage complexity and improve quality.

A **module** is the basic unit of decomposition of our systems. It will often correspond to a Java class, a file containing multiple classes, or a "package", which is a set of Java classes that cooperate to achieve a task.

Modular design and programming is designing or constructing software based on modules.

Modules typically form a hierarchy: modules can consist of smaller modules. For example, a java program may consist of multiple packages. A package consists of multiple classes. Each class consists of multiple methods, or even of sub-classes.

Modular design criteria

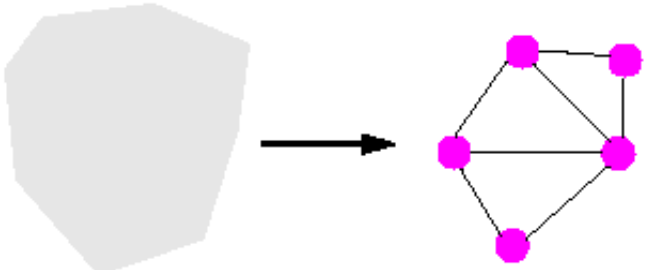
There are five criteria that underlie modular design.

1. Decomposability.
2. Composability.
3. Understandability.
4. Continuity.
5. Protection.

1. Modular decomposability

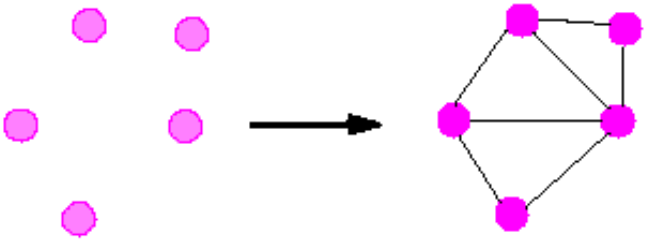
A modular design method makes it easier to decompose a problem into a small number of less complex sub-problems, connected by a simple structure, and ***independently enough*** to allow further work to proceed separately on each of them.

In the figure below, it isn't clear how to break down the shape on the left. Modular design is about a way of converting that into something like the figure on the right, with multiple modules, each interacting in well-defined ways with some of the other modules. If this is done well, each of the five modules on the right is simpler to implement than the big block.



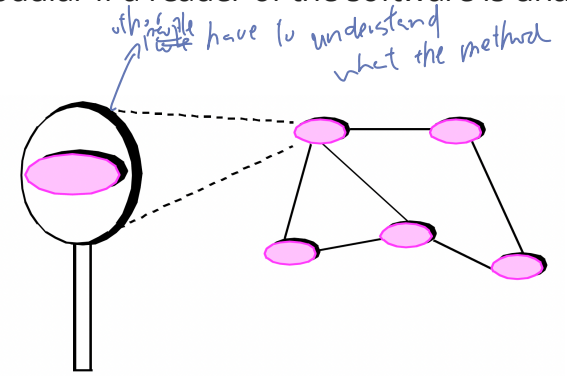
2. Modular composability

A modular design method makes it easy to produce software components that can then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.



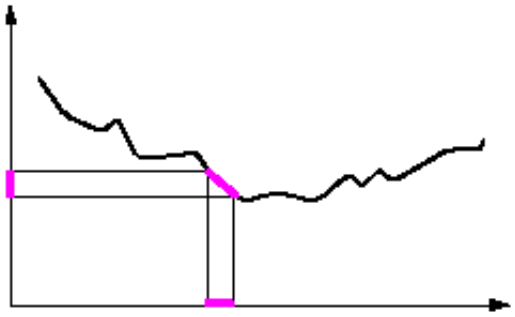
3. Modular understandability

Modular design helps produce software in which a human reader can understand each module without having to know the others, or at worst, by having to examine only a few of them. A design method can hardly be called modular if a reader of the software is unable to understand its elements separately.



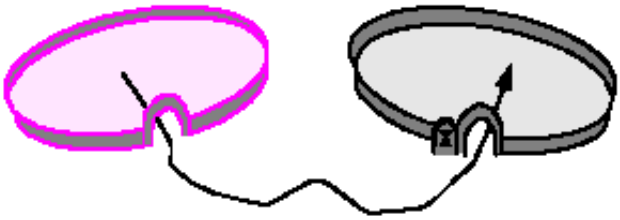
4. Modular continuity

A design method satisfies Modular Continuity if a small change in a problem specification will trigger a change of just one module, or a small number of modules. Continuity means that small changes should affect individual modules in the structure of the system, rather than the structure itself.



5. Modular protection

A design method provides protection if it gives architectures in which the effect of an abnormal condition occurring at run time in one module will remain confined to that module or at worst will only propagate to a few neighbouring modules.



References:

[1] Meyer, Bertrand. “Object-Oriented Software Construction, 2nd Edition.” (1997).

Modular design rules

The guiding principles that allow modular design to meet the above criteria are as follows.

1. Direct mapping
2. Few interfaces
3. Small interfaces
4. Explicit interfaces
5. Information hiding

1. Direct mapping

Any software system is designed to address the needs of some real-world problem. If we have a good model for describing that real-world problem, it is desirable to keep a clear mapping between the structure of the solution, as provided by the software to be developed, and the structure of the problem to be solved.

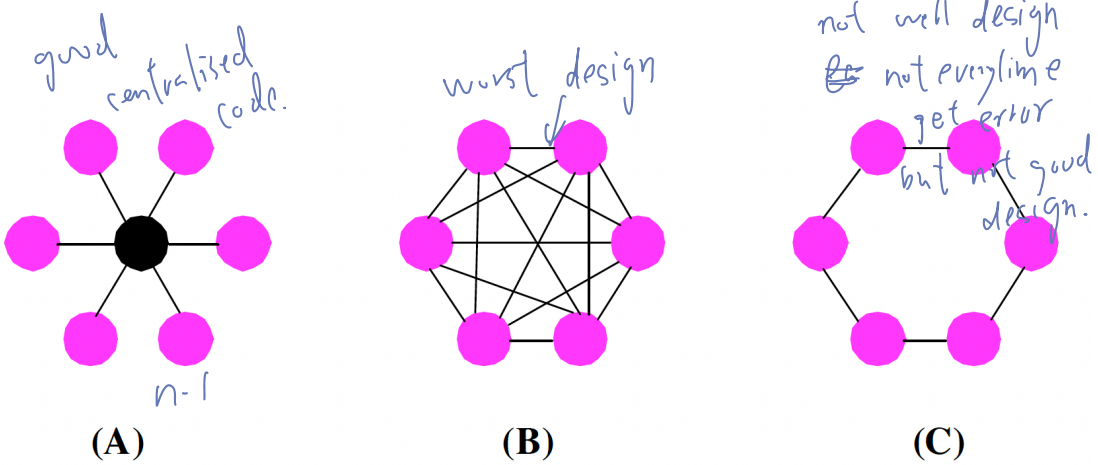
The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modelling the target problem.

- Modular Criteria: Continuity, Decomposability

2. Few interfaces

This rule restricts the overall number of communication channels between modules in a software architecture. Every module should communicate with as few others as possible. More specifically, if a system is composed of n modules, then the number of intermodule connections should remain as closer to the minimum, n - 1, as shown in the figure A, than to the maximum, n * (n - 1) /2, as shown in the figure B.

Types of module interconnection structures



Here "interface" refers to things like the set of public methods and variables that a Java class has. Modules "communicate" if, for example, one class imports the other and uses its methods.

In the figure A, this is an extremely centralized structure in which there is one "master" module; all other modules talk to it and it only. In the figure C, every module just talks to its two immediate neighbors, and there is no central authority.

- Modular criteria: Continuity, Protection, Understandability, Composability and Decomposability

3. Small interfaces

This rule relates to the size of intermodule connections rather than to their number. If two modules communicate, they should minimize their interaction (i.e., they should exchange as little information as possible). For example

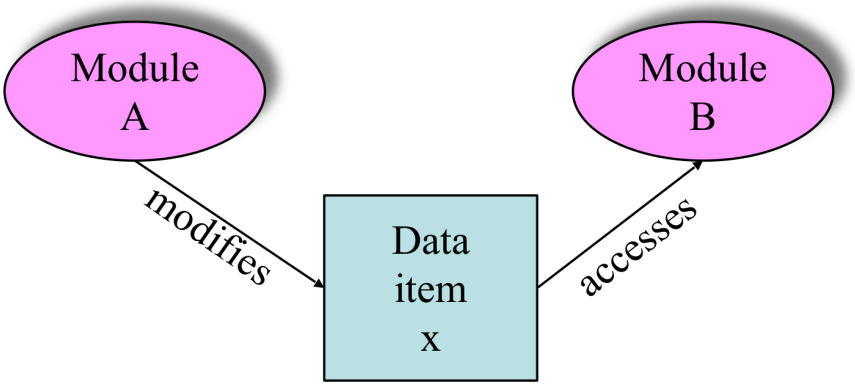
- it is better to provide as fewer public methods as possible to achieve the goal
- it is better to require a method to be called only once instead of many times
- it is better if calls don't need to be performed in a complex sequence. Sometimes you will need to call method A before method B. That is OK, but if you need to call methods A, B, ..., M from some class, and the behaviour of method N depends on the order in which you called them, then you may want to redesign your interface.

These all say "simple is better". However, Einstein said that things should be "as simple as possible but no simpler". Sometimes the complexity of a problem will require many interfaces or large interfaces. That is OK, but you should still look for ways to make it as simple as possible.

- Modular criteria: Continuity, Protection.

4. Explicit interfaces

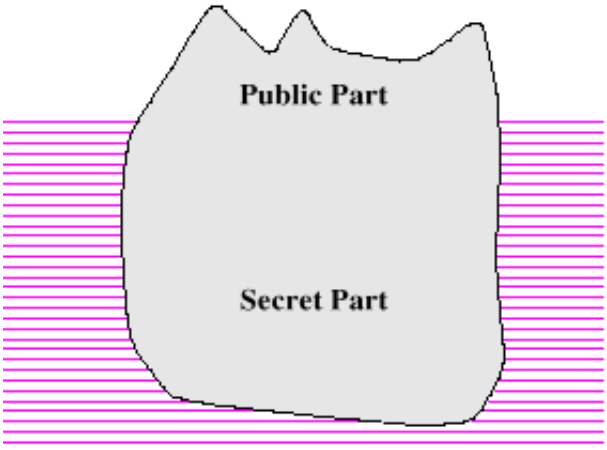
Whenever two modules A and B communicate, this must be obvious from the text of A or B or both.



- Modular Criteria: Decomposability, Composability, Continuity and Understandability.

5. Information hiding

The designer of every module must select a subset of the module's properties to be made visible to authors of client modules as the official information about the module. This is related to keeping the interfaces small.



- Modular Criteria: Continuity

References:

[1] Meyer, Bertrand. “Object-Oriented Software Construction, 2nd Edition.” (1997).

Summary of modularity

The biggest challenge in writing modern software is usually managing the complexity

- A web browser is about 10,000,000 (ten million) lines of code.

Modularity is an important tool to help in that management.

It isn't a strict method, but a set of goals, and basic (fuzzy) rules that help to achieve those goals.

Unified modelling language (UML)

If modules are so important, how do we describe them? We want a description that is simpler than the code that makes up the modules. We want to be able to design the module interactions at the "level" of modules.

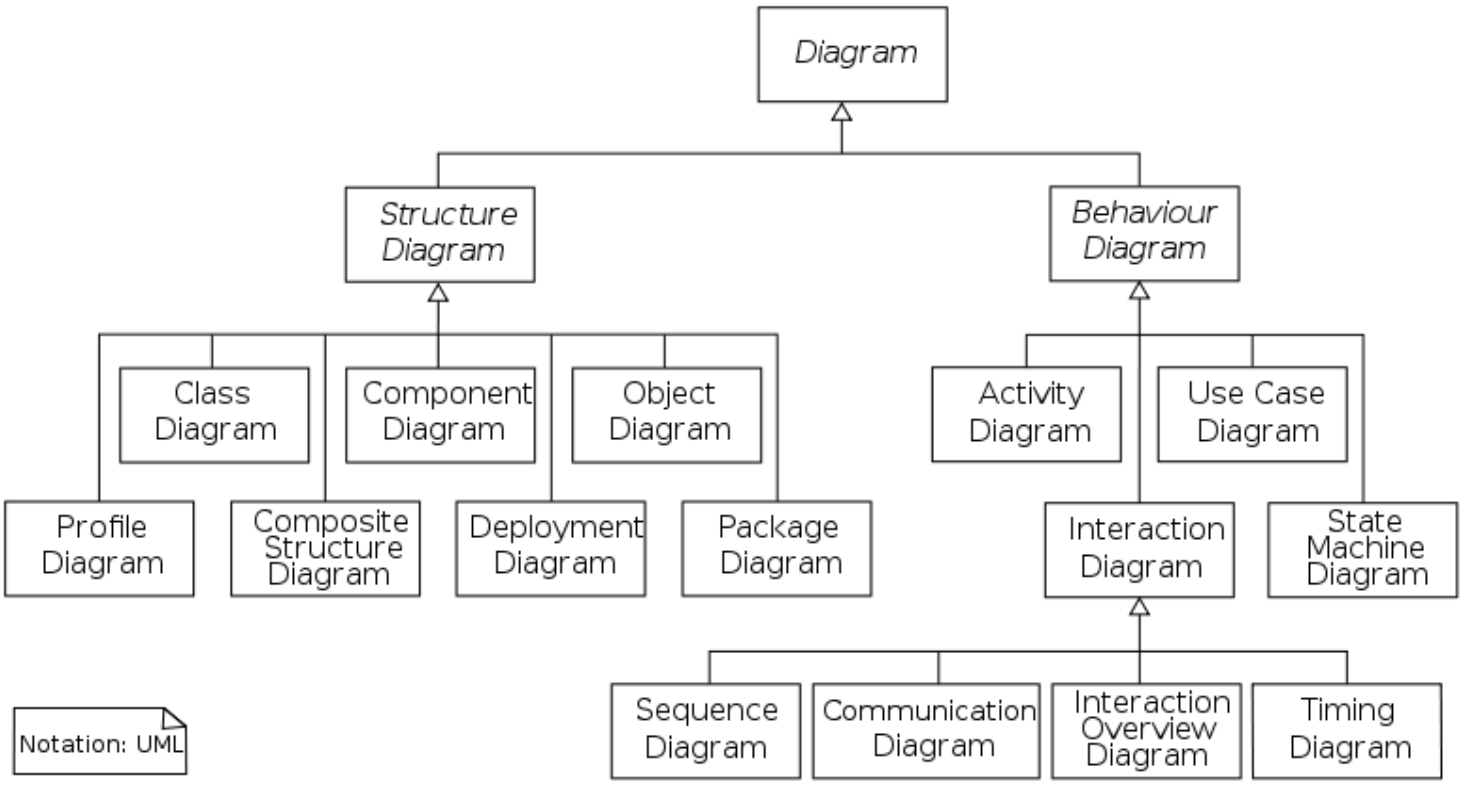
The Unified modelling language (UML) is a graphical representation of modules and their interactions.

Graphical representation systems for program design have been used for over half a century. Flowcharts are the best known example. Structure diagrams have also been used.

UML is designed to reflect modularity, and the object-oriented programming philosophy.

UML diagrams

UML diagrams show relationships between classes.



They are a type of *structure diagram*. They describe the structure of a system by showing the system's:

- Classes
- Their attributes (variables) and the accessibility
- Methods
- The relationships among the classes

Classes are central to OOP, and the **class diagram** is the easiest of the UML graphical representations to understand and use.

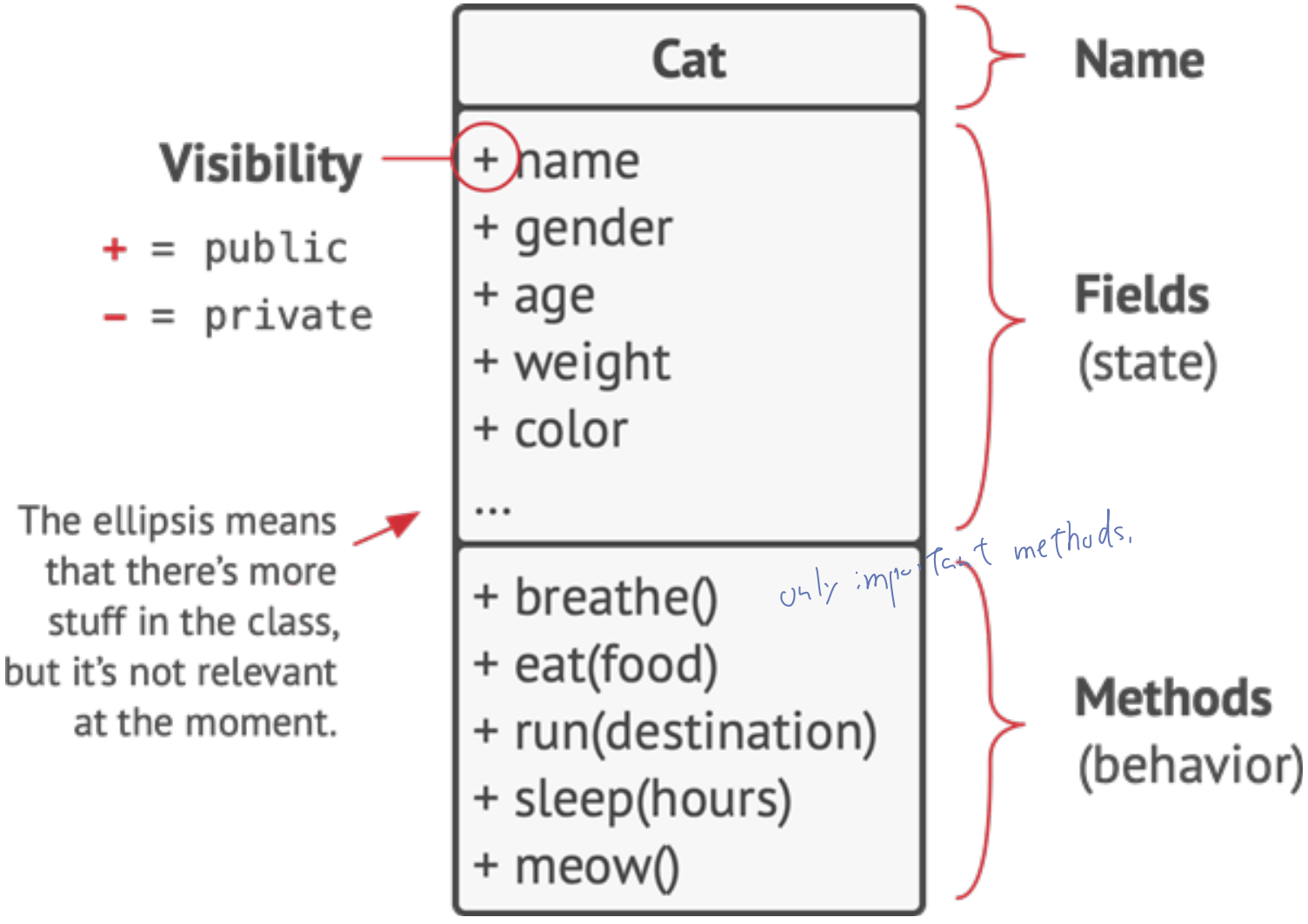
A class diagram is divided up into three sections:

- The top section contains the class name
- The middle section contains the data specification for the class
- The bottom section contains the actions or methods of the class

The data specification for each piece of data in a UML diagram consists of its name, followed by a colon, followed by its type

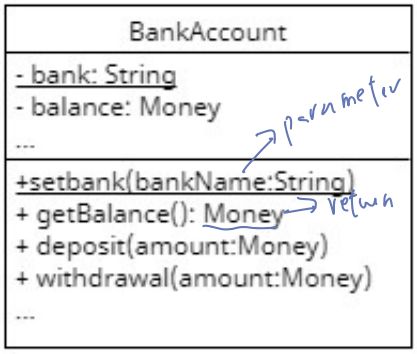
Each name is preceded by a character that specifies its access type:

- A minus sign (-) indicates private access
- A plus sign (+) indicates public access
- A sharp (#) indicates protected access
- A tilde (~) indicates package access



A class diagram need not give a complete description of the class. If a given analysis does not require

that all the class members be represented, then those members are not listed in the class diagram. Missing members are indicated with an ellipsis (three dots).



Class interactions

Rather than show just the interface of a class, class diagrams are primarily designed to show the interactions among classes.

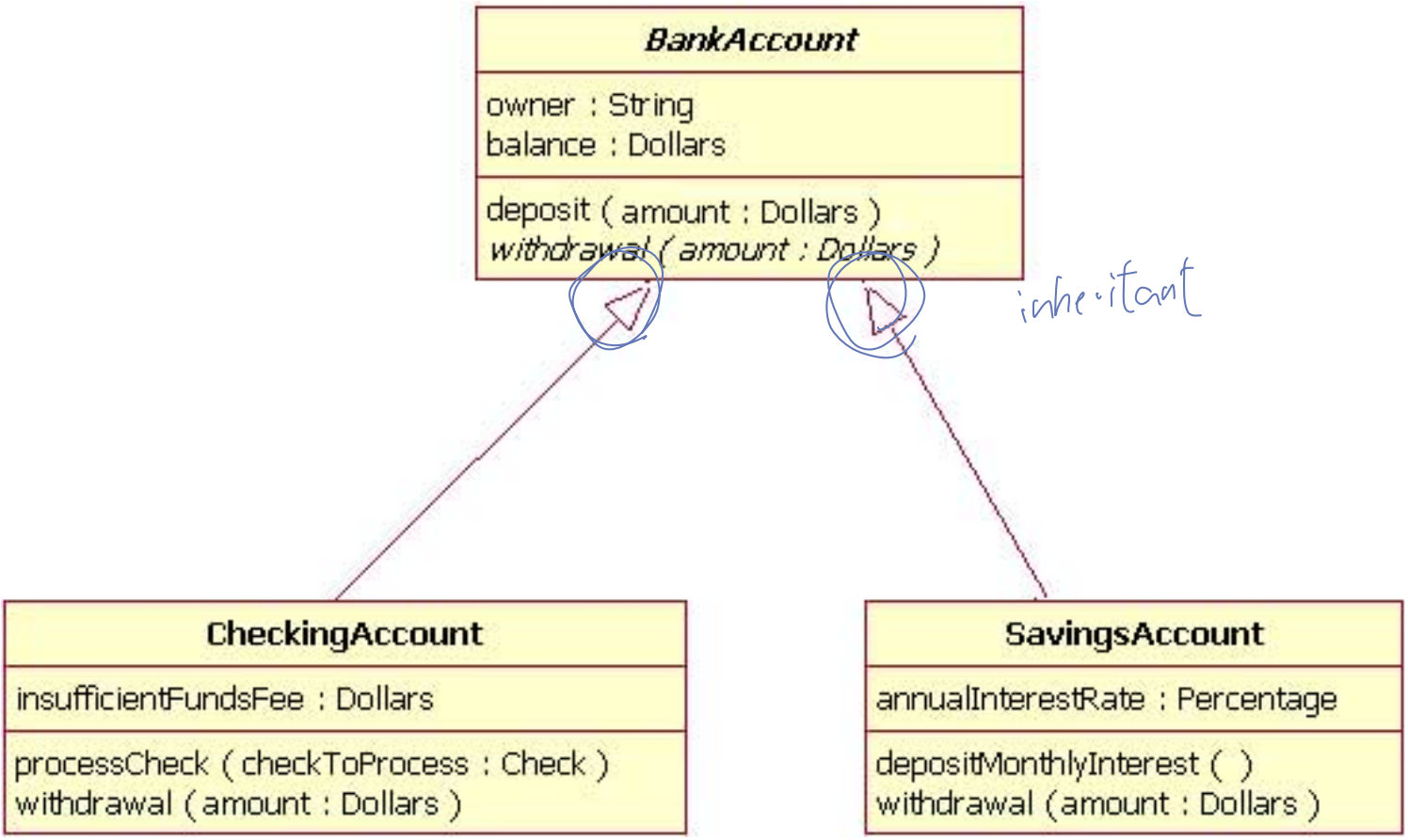
UML has various ways to indicate the information flow from one class object to another using different sorts of annotated arrows.

UML has annotations for class groupings into packages, for inheritance, and for other interactions

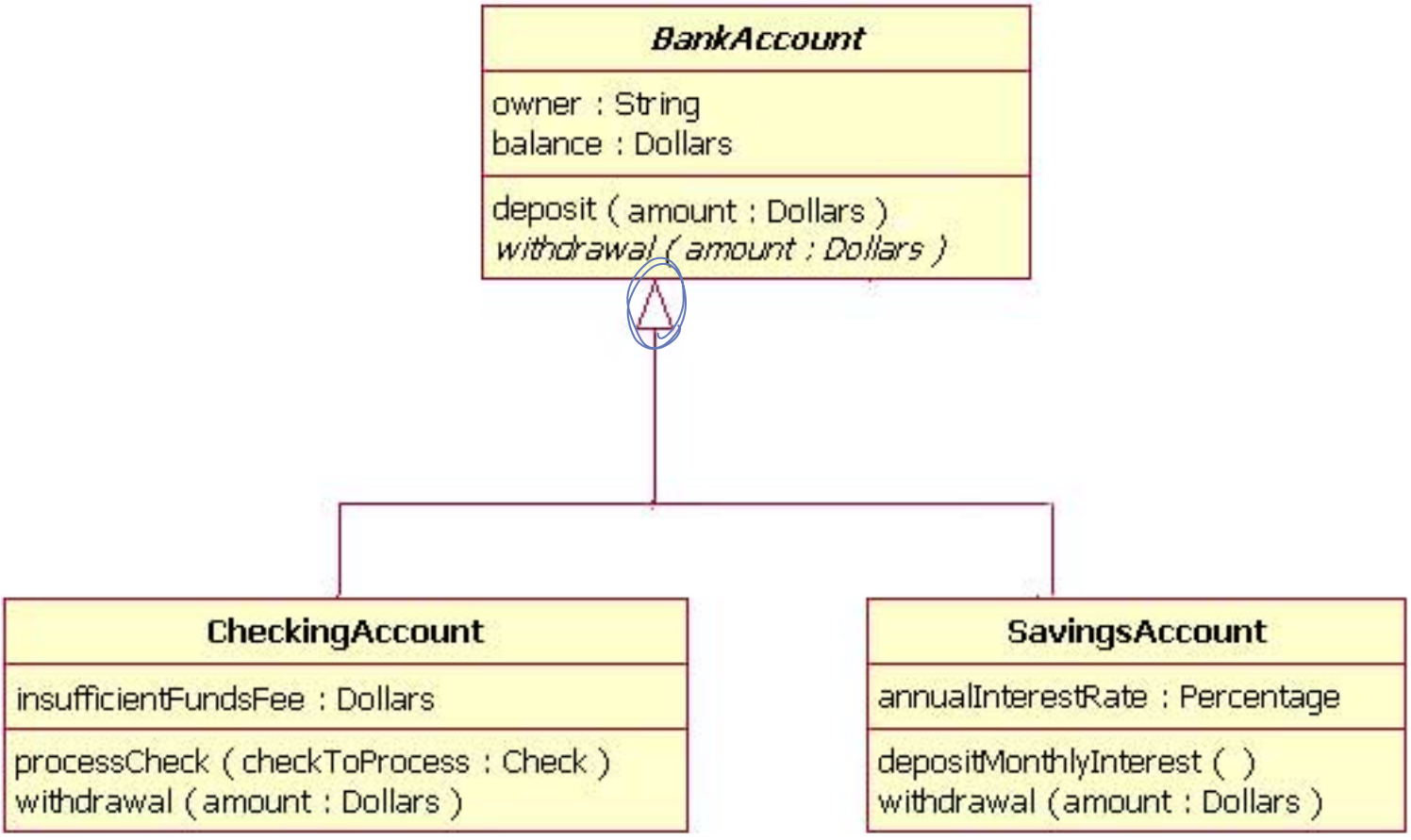
In addition to these established annotations, UML is extensible.

Inheritance (Called Generalisation in UML)

One important relationship that can be shown by UML is which classes are derived from which others. This is shown by an arrow with an *open head* leading from the child class to the parent class.



Sometimes, the arrows are not straight lines, but are made of horizontal and vertical segments, which can overlap if they are going to the same place.

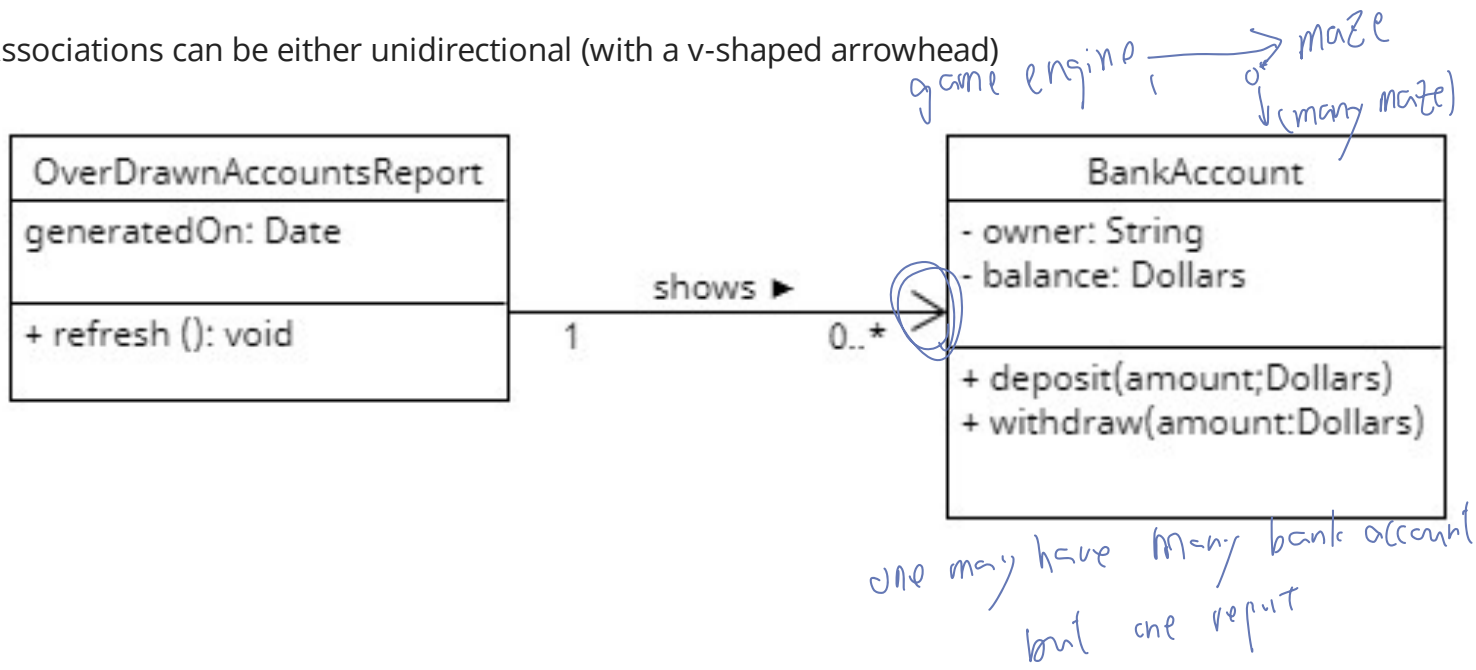


Associations

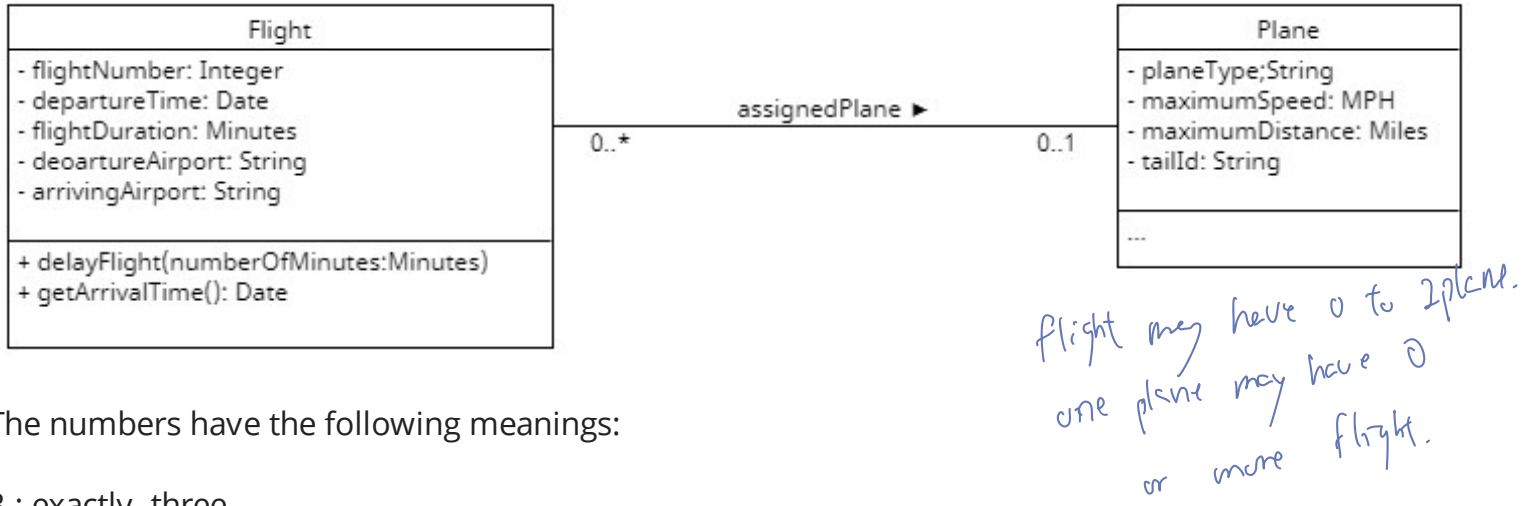
different arrows.

Another type of arrow indicates associations. In this case, there is something (unspecified) linking to two classes or objects. In other words, one objects uses or interacts with another object. A common case is that one object of one class has an identifier (like an array index) that specifies an objects of another class. The array doesn't have to be a member of either of those classes.

Associations can be either unidirectional (with a v-shaped arrowhead)



or non-directional with no arrow head



The numbers have the following meanings:

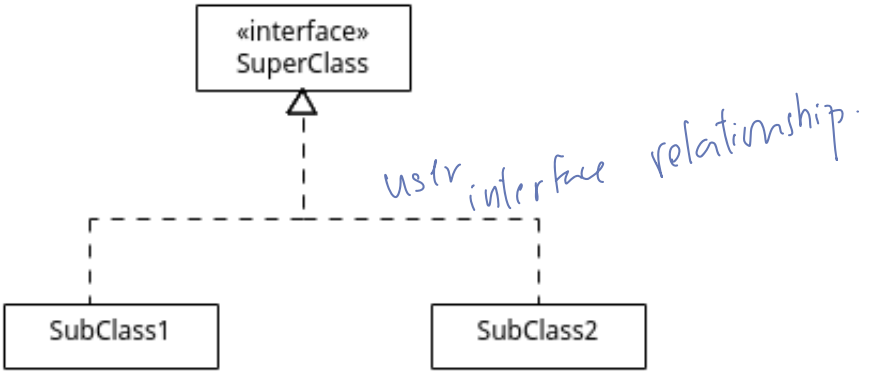
3 : exactly three

* : zero or more

0..* : any number from zero to many

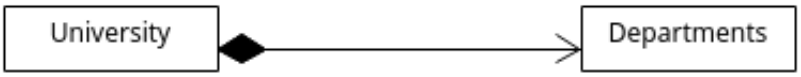
1..*: one to many.

The Java interface relationship (also called implements or realisation) is shown with a dotted line in UML. Here is the example.



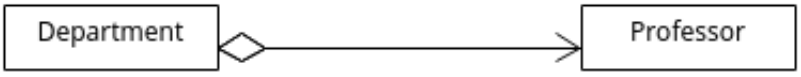
Aggregation (Shared and Composed)

There are two types of aggregation relationships exist in UML, the shared and composed (also called composition) aggregations. Composition is a whole part relationship between objects, where one is composed of one or more instances of the other. The distinction between this relation and others is that the component can only exist as a part of the container. In UML the composition relationship is shown by a line with a filled diamond at the container end and an arrow at the end pointing toward the component.



UML Composition. University consists of Departments.

Shared aggregation (or simply aggregation) is a less strict variant of composition, where one object merely contains a reference to another (that is conceptually they share the objects). The container doesn't control the life cycle of the component. The component can exist without the container and can be linked to several containers at the same time. In UML the aggregation relationship is drawn the same as for composition, but with an empty diamond at the arrow's base.

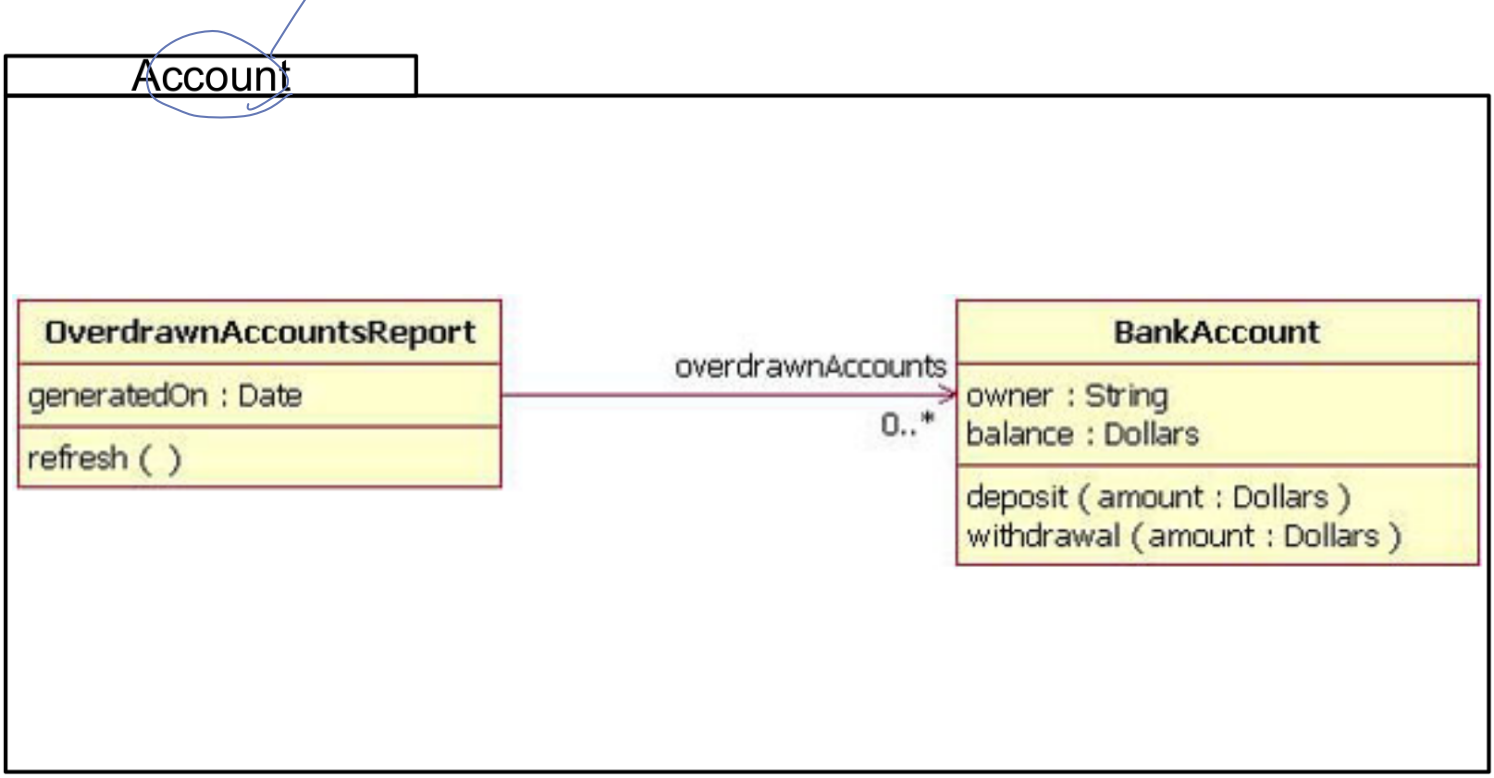


UML aggregation (shared). Department contains professors.
The professors may belong to more than one departments.

There are many other relationships in UML such as dependency etc; we are not covering them in this unit as they are part of other unit offered at Unimelb.

UML Packages

UML can show how classes are gathered into packages, by surrounding them by a rectangle with a name tab.



Tools

UMLET accessible online at https://www.umletino.com/umletino.html. The standalone version can be downloaded at https://www.umlet.com/