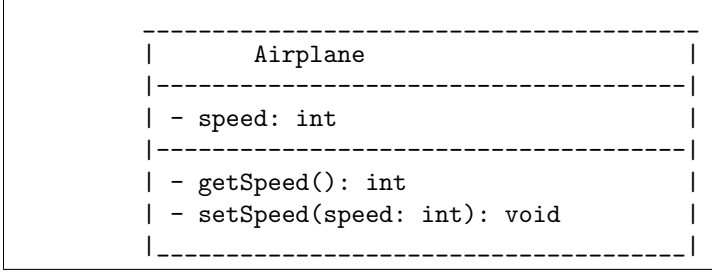
1 Reading 6: HFSD Appendix A, #1 (UML Class Diagrams), pp. 434-435

A UML diagram has

- The name of the class. Always in bold, at the top of class diagram.
- Member variables are in the second part. Each one has a name, then a type after colon.
- Methods are in the third part. Each has a name, and then any parameters the method takes, and then a return type after the colon.

The `+` and `-` signs describe the visibility of member variables and the methods. `+` is public. `-` is private.

(Example.)

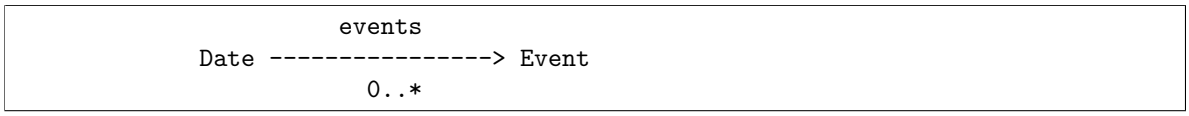


A class diagram describes the static structure of your classes. It makes it easy to see the big picture: you can easily tell what a class does at a glance. You can even leave out particular variables and/or methods if it helps you communicate better.

1.1 Class Diagrams Show Relationships

Association

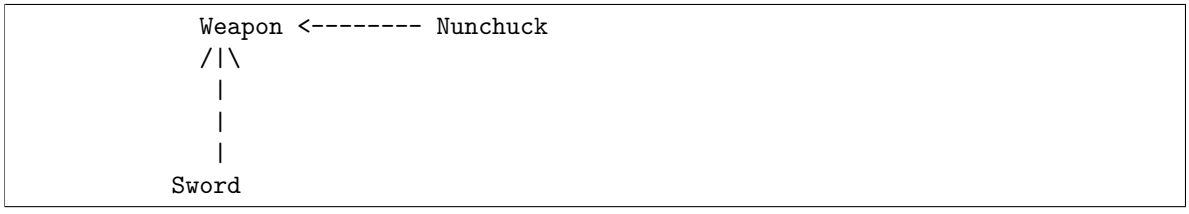
- One class is made up of objects of another class.
- For example, a Date is associated with a collection of Events.



Note that `events` is the name of the member variable in the `Date` class. There could be any number of `Events` on a `Date`.

Inheritance

- Useful when a class inherits from another class.
- For example, a Sword inherits from Weapon.



The inheritance relationship is defined by the arrow.

Some things to remember

- UML can be drawn using paper and pencil. No need for big expensive set of tools
- Class diagram isn't a very complete representation of a class. It's just a way to communicate basic details of a class's variables and methods. Also easy for you to talk about.
- UML is like a standard – everyone should know what it is.
- UML has diagrams for state of objects, sequence of events in application, etc. (alongside just class diagrams).

2 HFSD Appendix A, #5 (Refactoring), p. 441

- Process of modifying structure of code without modifying its behavior.
- Usually related to specific improvement in your design.
- Increases cleanness, flexibility, and extensibility of code

3 HFSD Ch 5 (Good Enough Design), pp. 149-163, 168-169, 172

- Well-designed classes are singularly focused.
  - If you have to change a bunch of classes just to change one thing for one class, time to redesign.
- **Single Responsibility Principle**
  - Every object in your system should have a single responsibility, and all the object's services should be focused on carrying out that single responsibility.
- Spotting multiple responsibilities in your design
  - Most of the time, you can spot classes that aren't using SRP with a simple test.
    - On a blank sheet of paper, write down a bunch of lines like this: "The [blank] [blanks] itself." You should have a line like this for every method in the class you're testing for the SRP
    - In the first blank of each line, write down the class name. In the second blank, write down one of the methods in the class. Do this for each method in the class.
    - Read each line out loud (adding a letter or word as needed to get it to read normally). Does what you just said make any sense? Does your class really have the responsibility that the method indicates it does?
  - You may need to make some judgement calls yourself.
- Going from multiple responsibilities to a single responsibility
  - Move methods that don't make sense on a class to a different class.
- Your design should obey the SRP, but also be DRY
  - DRY: don't repeat yourself!
  - Avoid duplicate code by abstracting or separating out things that are common and placing those things in a single location.
  - DRY is about having each piece of information and behavior in your system in a single, sensible place.
  - SRP: making sure a class only does one thing and does that one thing well. DRY: putting a piece of functionality in one place.
  - Cohesion is another way of saying SRP
- A great design helps you be more productive as well as making your software more flexible.