3F6 - Software Engineering and Design

Handout 4

The Unified Modeling Language

With Markup

Ed Rosten

Contents

Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching`.

The Unified Modeling Language

- a formal graphical language comprising a set of diagrams for describing software systems.
- used for designing, documenting and communicating various *views* of the software architecture and program behaviour.
- these different views of the system can be used at varying scales, presenting the key information and suppressing unimportant detail as desired.

Historical Note
UML evolved from three earlier rival approaches independently developed between 1989 and 1994.

- **Booch method** developed by Grady Booch of the Rational Software Corporation.
- **Object Oriented Software Engineering (OOSE)** developed by Ivar Jacobson of Objectory.
- **Object Modeling Technique (OMT)** developed by James Rumbaugh of General Electric.

The first complete version of UML (V1.1) was released in 1997 by a consortium which included DEC, HP, IBM, Microsoft, Oracle, and Texas Instruments. The current release is V2.0 which is used in this course.

Diagram Types

Structural Diagrams which describe the architecture or layout of the system.

1. **Class**. These describe the software architecture of the system by specifying what classes are present in the system, and their relationships.
2. **Object**. These describe a snapshot of the system while it is running and identify what objects are present at a given instant, and their relationships.

Behavioural Diagrams which describe dynamic aspects of the software behaviour by showing sequences of activity.

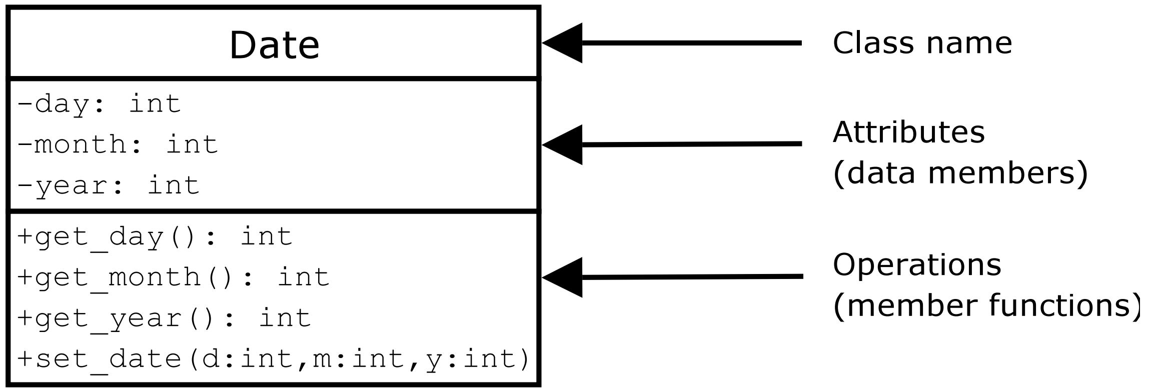
3. **Sequence**. These describe the time ordered activity of the system as it performs a particular task.
4. **Communication**. These show the structural organization of objects that pass sequences of messages to each other.
5. **State**. These are particularly useful for describing software that can be modeled as a state machine.

Note that the UML specification contains 13 kinds of diagram in total. Here we describe 5, but in fact, class and sequence diagrams are the most commonly used.

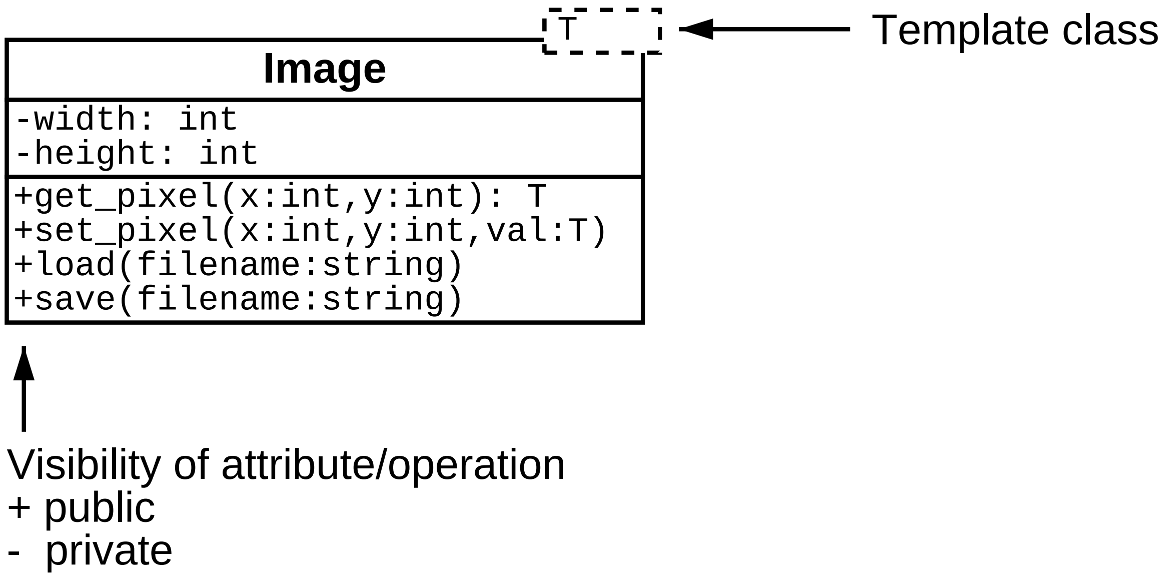
Class Diagrams

These show the classes in a software system, their attributes and operations and their relationships.

Each class is represented by a box split into three sections:

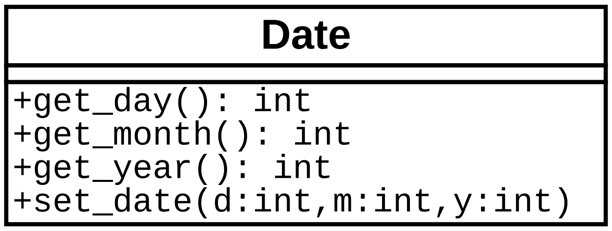


Template classes like our image class can also be shown:



Suppressing detail

It is common to suppress unwanted or redundant levels of detail in these diagrams. In particular, we often wish to suppress the attributes of the class since these are part of the implementation detail, and not available in the public interface.



Also, in the earlier example, the **`Image`** class does not show the **`get_height()`** and **`get_width()`** functions because we can see that the class has **`height`** and **`width`** as attributes.

Formally the specification for the layout of an attribute is:

```
<visibility> <name> :  <type>
```

and for an operation:

```
<visibility> <name> ( [<argname> :  <argtype>] ) :  <return type>
```

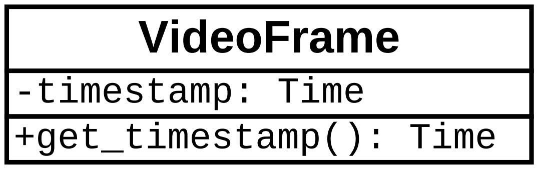
It is common to drop the visibility indicator and assume that all operations are public and all attributes are private. and we will do this

Class Relationships

Objects will often contain other objects. For example a `VideoFrame` might contain a `Time` rather than just an `int` to represent the timestamp:

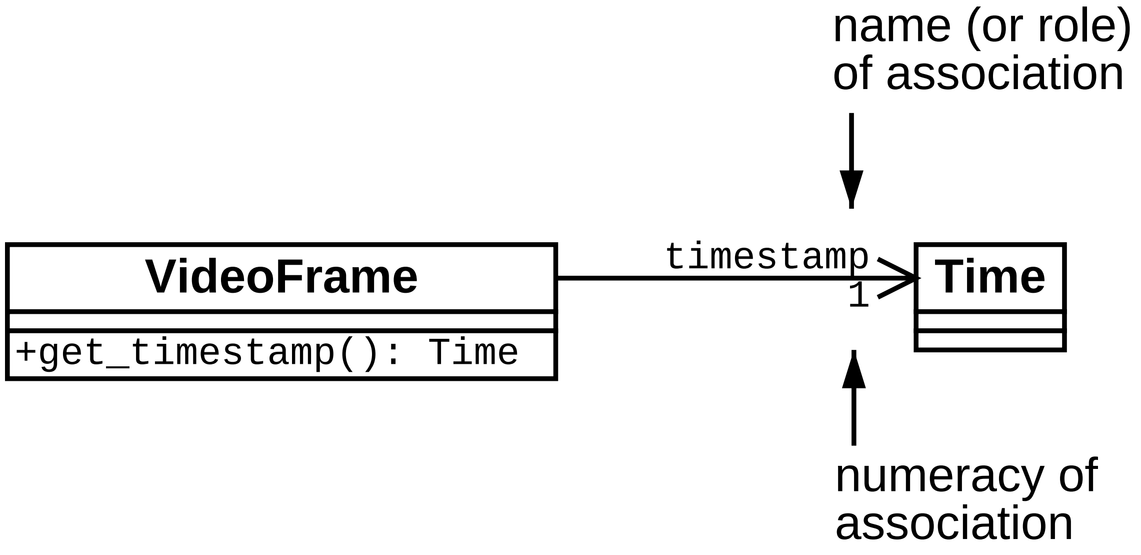
```
class VideoFrame : public Image {
public:
  VideoFrame(int w, int h, Time t);
  Time get_timestamp();
private:
  Time timestamp;
};
```

This information can be represented in one of two ways in UML. We can record timestamp as an attribute in the `VideoFrame` class



Association Arrows

or we can draw an association between the two classes in a UML diagram.

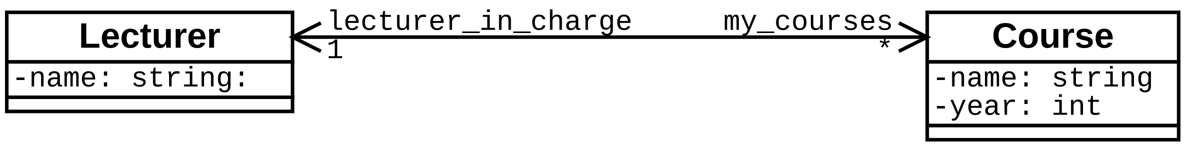


The line showing this association has:

1. An arrowhead showing navigability. A `VideoFrame` knows what its timestamp is but the `Time` object doesn't know which `VideoFrame` it belongs to (indeed not all `Time` objects do belong to `VideoFrame`s).
2. A name for the role that the `Time` object plays within `VideoFrame`, in this case "timestamp".
3. A number stating how many `Time` objects a `VideoFrame` has, in this case one. This can be a fixed number or a range (e.g. "`0..3`" or "`*`" or "`1+`").

Navigability

Associations can also show bidirectional navigability. For example, consider the relationship between lecturers and lecture courses in a timetabling application.



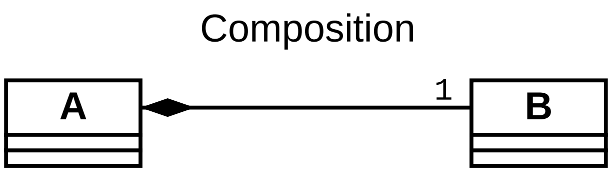
This shows that each `Lecturer` object knows about all the courses they are teaching and that each `Course` knows which lecturer is in charge. The * indicates that a `lecturer` may be in charge of any number of courses (including none).

```
class Lecturer {
  // ...
private:

  vector<Course*> my_courses;

};

class Course {
  // ...
private:

  Lecturer* lecturer_in_charge;

};
```

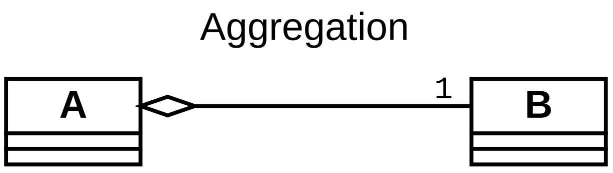
It is also permissible to show no arrows on an association. This indicates that the navigability is unknown, hence it usually appears during the design process.

Composition and Aggregation

UML also allows a class association to be adorned with a diamond. Not all users of UML employ this, but where it appears, it has the following meaning:



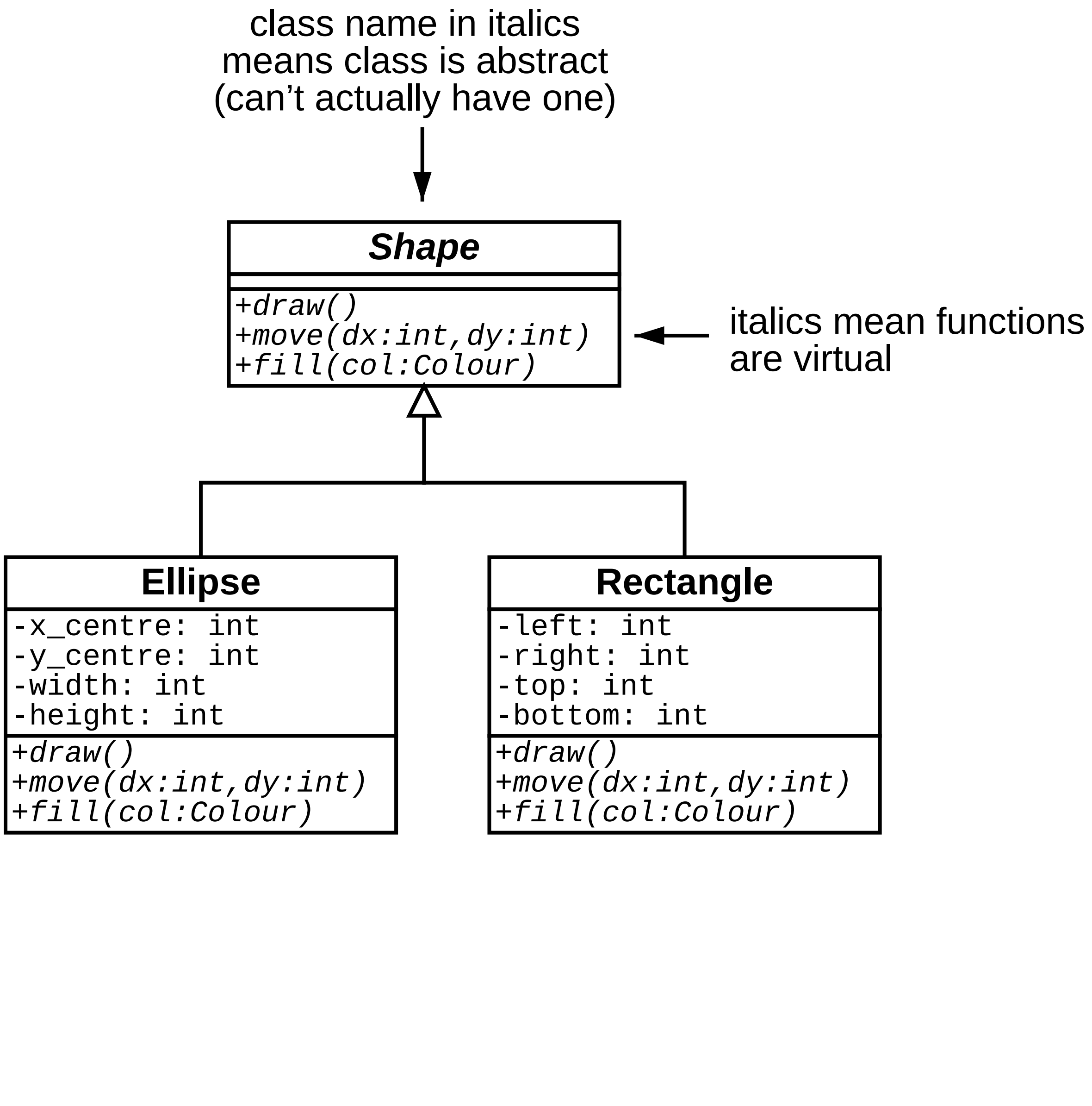
A solid diamond indicates that objects of class A own an instance of objects of class B. The object of class A is responsible for creating and destroying the instance of class B.



An empty diamond indicates that objects of class A know about (have a reference or pointer to) objects of class B. It also implies that A is a higher level construction than B, thus class B will not need to know about class A. By contrast, a plain line indicates that the two classes exist at the same level in the code.

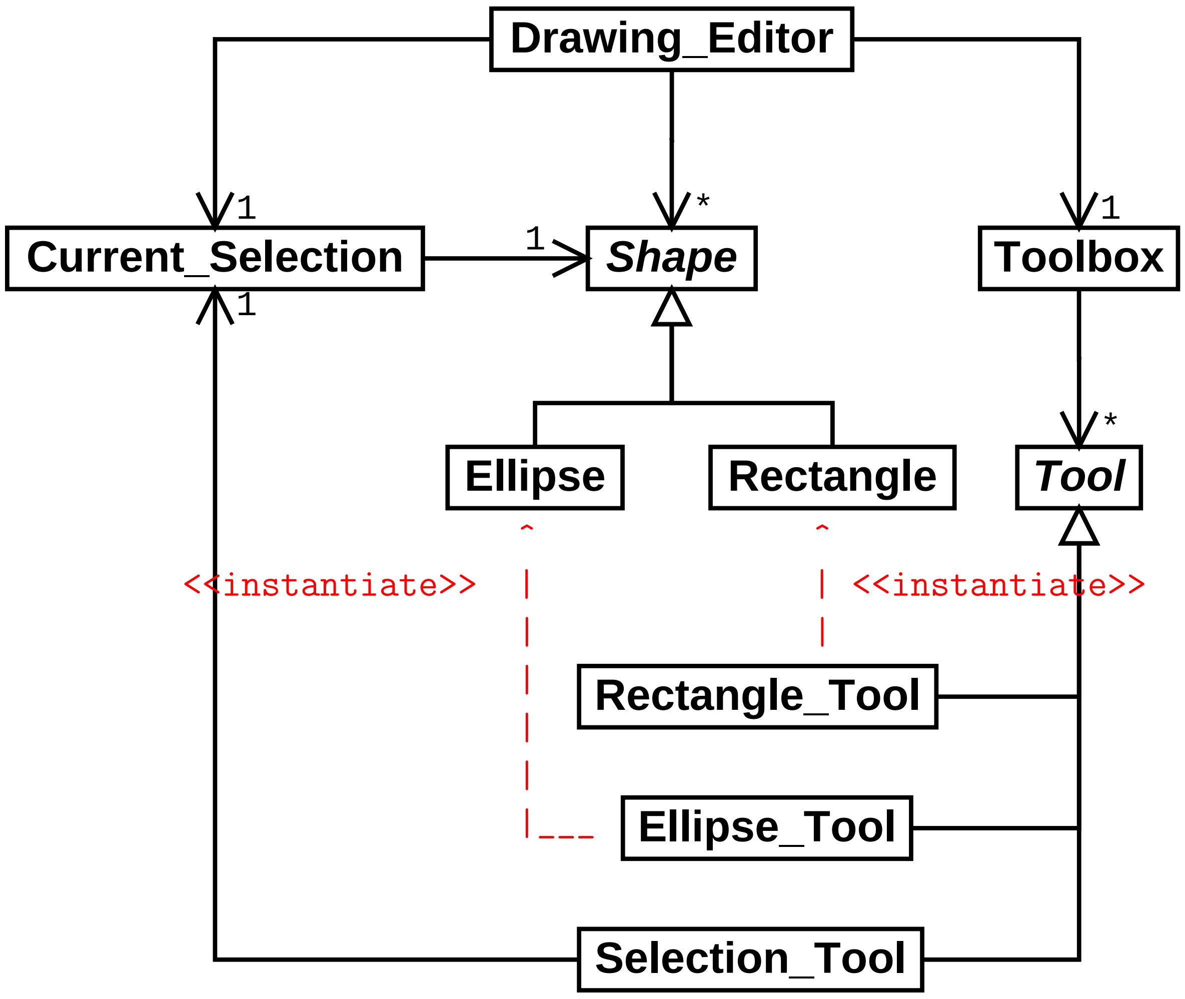
Class Derivation

Derived classes are shown using a triangular arrowhead pointing to the base class (or superclass). Virtual functions are shown in italics.



Drawing Editor

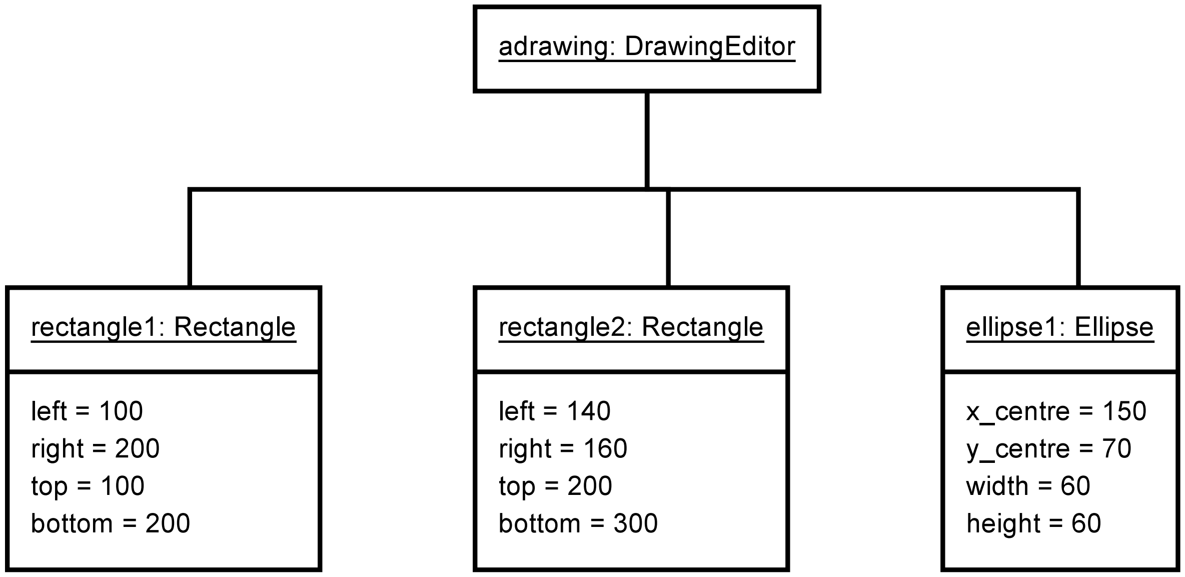
We can now show the drawing editor using UML (suppressing all attributes and operations):



This is a simple diagram but it conveys a lot of useful information about the architecture of the software.

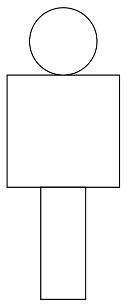
Object Diagrams

Object diagrams show the *objects* that exist at some moment of time when the system is running.



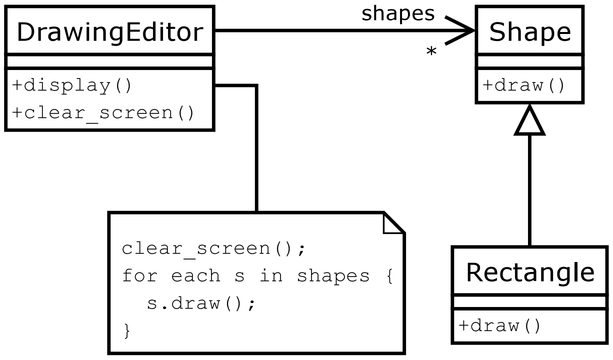
The name and type of the object are shown in the top section of the box and are underlined. This diagram shows four objects, `adrawing` which is of type `Drawing_Editor`, `rectangle1` and `rectangle2` which are two objects of type `Rectangle`, and an `Ellipse` called `ellipse1`.

The actual values of the attributes of the rectangles and the ellipse are shown in the bottom section of the box. Using this information we could construct the following figure:

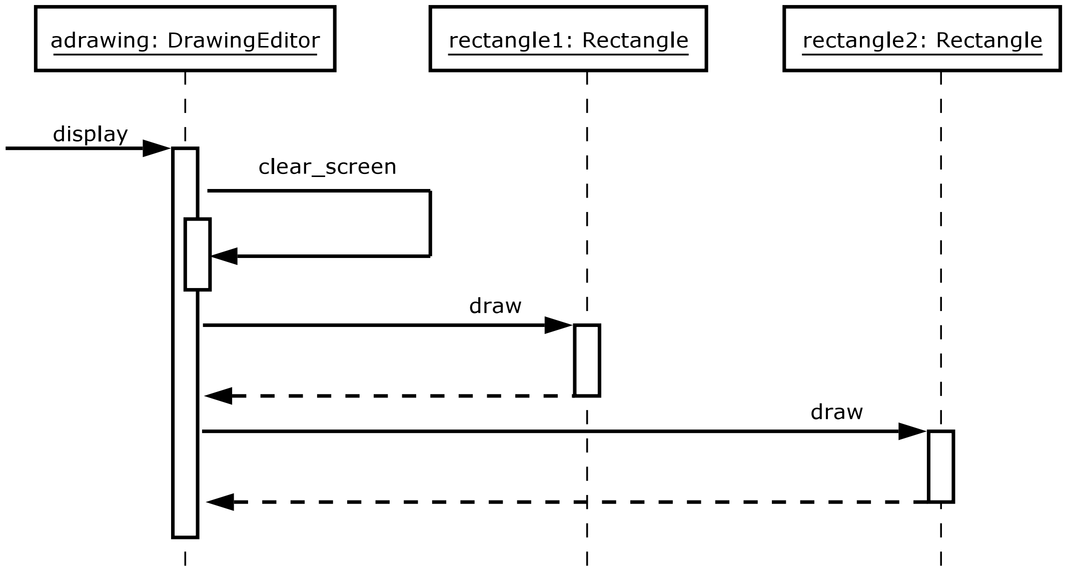


Sequence Diagrams

By contrast to class diagrams which show the static software architecture, sequence diagrams show the dynamic behaviour of a software system as it runs. For example, consider the drawing editor - part of which is shown in this class diagram:



Suppose the drawing consists of two rectangles and the `display()` function is called on the `DrawingEditor`. The series of function calls that takes place can be shown in a *sequence diagram*:



Interpreting the diagram

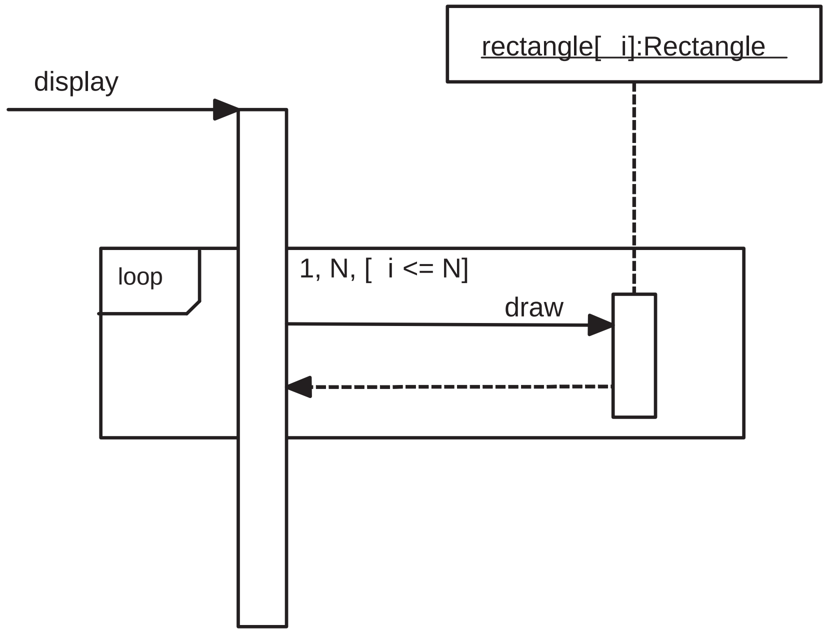
This diagram shows three objects. These are: one object of type `Drawing_Editor` called `adrawing` and two objects of type `Rectangle` called `rectangle1` and `rectangle2`. The vertical axis of this diagram corresponds to time (traveling downward). The white boxes show the duration of each call. This diagram explicitly shows the return from the two `draw()` calls with dashed lines although these are often omitted.

The sequence of events is:

1. The `display()` function is called in `adrawing`.
2. `adrawing` calls the `clear_screen()` function in itself.
3. This function returns.
4. `adrawing` calls the `draw()` function in `rectangle1`.
5. This function returns.
6. `adrawing` calls the `draw()` function in `rectangle2`.
7. This function returns.
8. The `display()` function is completed and returns.

Sequence Fragments

Sequence fragments allow procedural control functions such as if, while, for, etc to be represented. They consist of a box around the controlled action or sequence of actions labelled with the fragment type in the corner and parameters along the top edge. For example, if all of the shapes to draw was stored in an array, the previous example might be redrawn as:



The types of sequence fragment include:

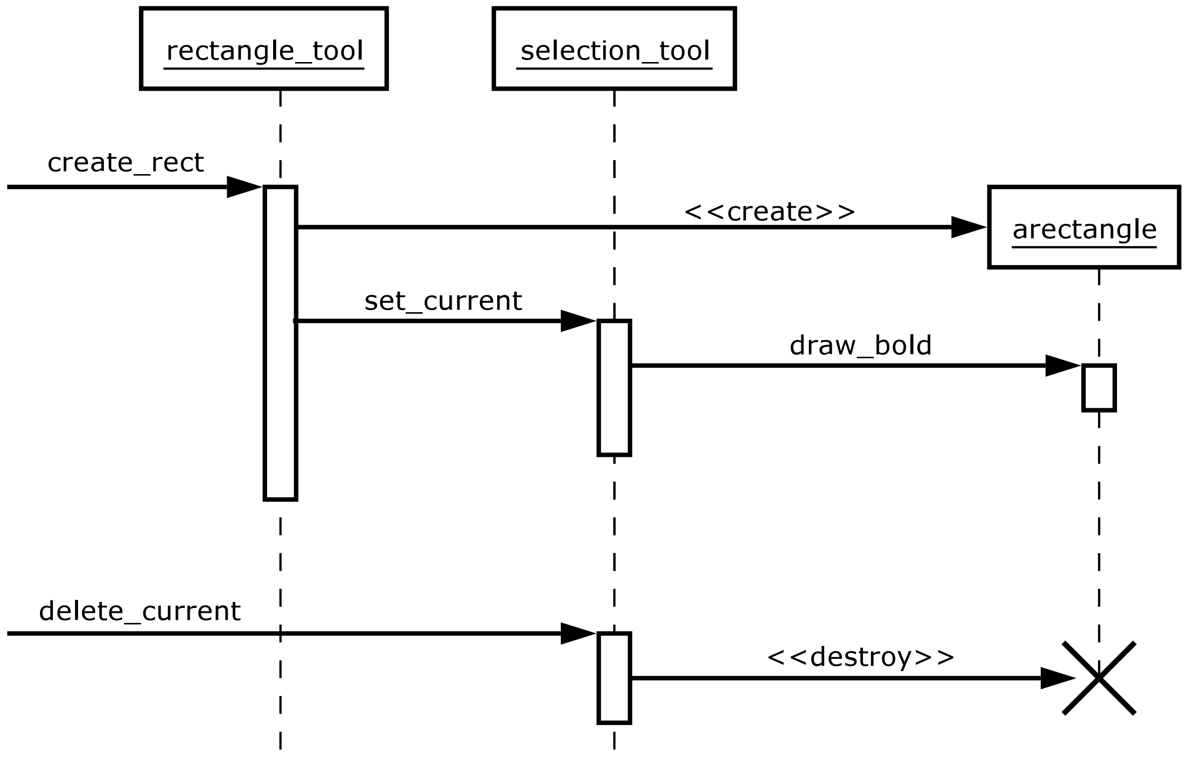
Type	Parameters	Purpose
ref	None	decomposing diagrams
loop	min,max, [cond]	equivalent to for loop
opt	[cond]	equivalent to if ..
alt	[cond1] ... [cond2] ... [else] ...	equivalent to if .. else if ... else
break	None	used to break out of loops

Creation and Destruction

Sequence diagrams can also show the creation and destruction of objects in the system:

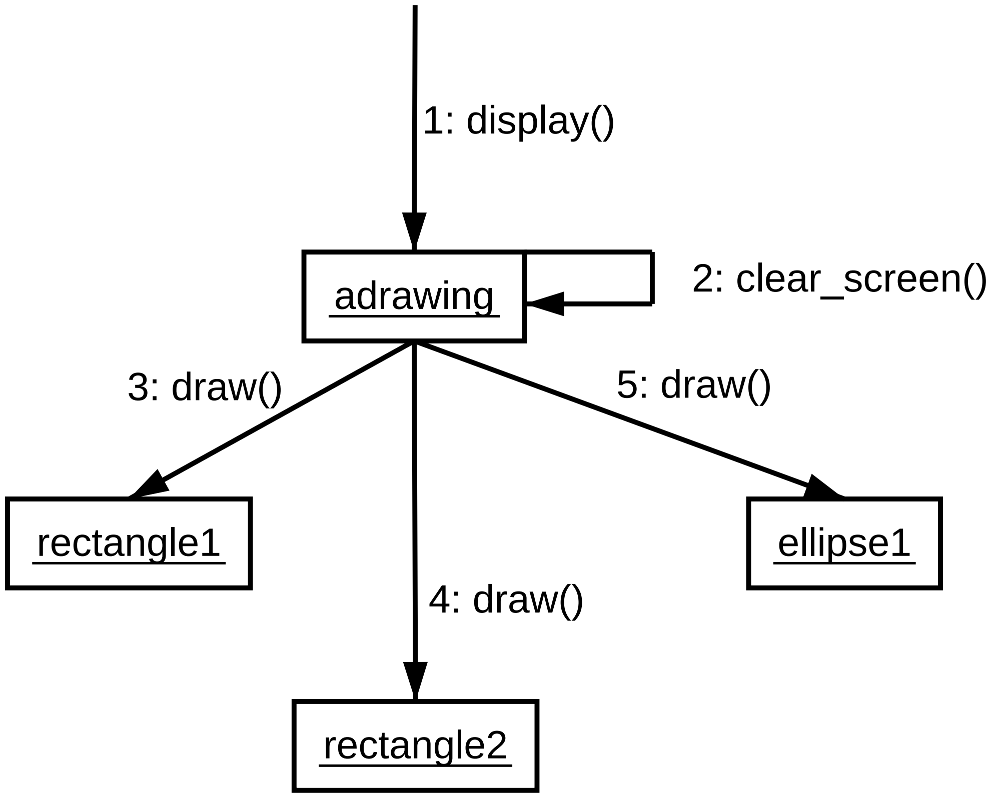
Scenario: The rectangle tool is used to draw a rectangle. This will result in the creation of a new `Rectangle` object - this object will then automatically set to be the current object in the selection tool. Then suppose the user doesn't like the placement of the rectangle and hits the delete button and destroys the rectangle.

The sequence diagram that illustrates this scenario must show the construction and destruction of the rectangle:



Communication Diagrams

Communication diagrams are an alternative way of showing sequences of events.

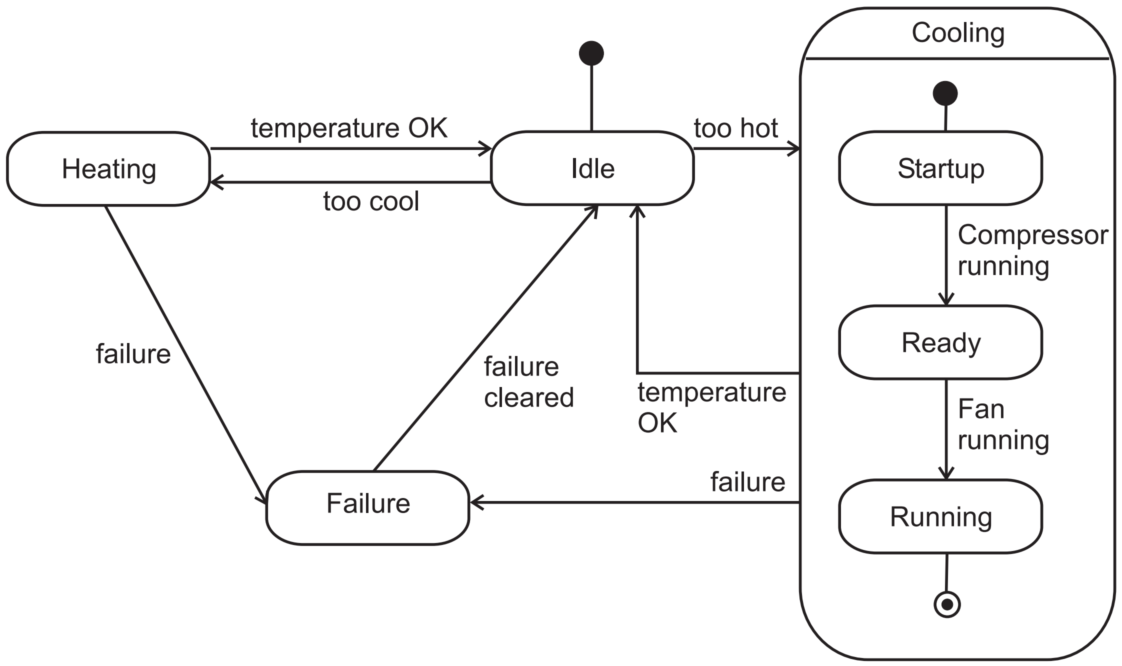


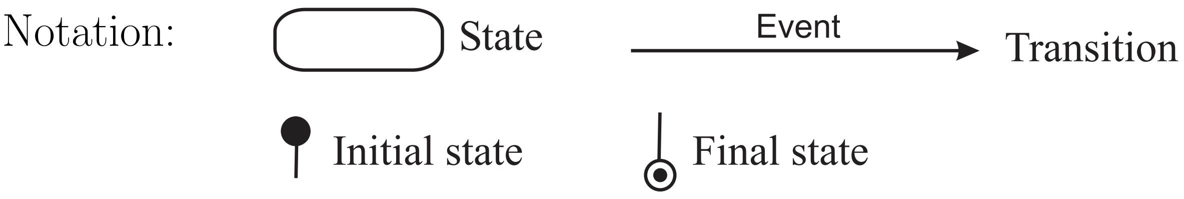
This diagram uses the same layout as the object diagram and overlays functions calls on it. In order to make the sequence of events explicit, the calls are numbered with a sequence number.

State Diagrams

As well as thinking about the relationship between objects, the behaviour of each individual object needs to be considered. In function-oriented design, the behaviour of each function also needs specifying. In either case this can be done by drawing a *state transition diagram*, which shows how the system or object state changes with different stimuli. UML specifies a notation for these, which it calls *statecharts*.

Example: Air conditioning unit





Other Types of UML Diagrams

In addition to the 5 diagram types listed above, UML also supports

6. Use Case Diagrams: for describing the interactions between your program, users and other systems.
7. Activity Diagrams: for describing work flow in terms of sequential and parallel activities.
8. Timing Diagrams: for showing the execution of a program along a time line. These are particularly useful for real time event-driven programs.
9. Interaction Overview Diagrams: provide a high level view of specific program interactions. Often used give more detail to use case diagrams.
10. Composite Structure Diagrams: show the internal structure of a class
11. Component Diagrams: for describing software components as building bricks.
12. Deployment Diagrams: show how software components map onto real hardware.
13. Package Diagrams: show how a system is composed of chunks with interdependencies
14. Profile Diagrams: show extensions to new domains as stereotypes relationships between them