4M21: Software Engineering and Design

Kazal Oshodi

January 2023

Contents

1 Lecture 1

Software Engineering: a collection of methods,techniques and tools that could be applied to design, build and maintain the instructions for telling a computer what to do and how to do it

1.1 Software Process

- Analysis: process of understanding the requirements, constraints and goals of a software development project in order to identify the best solution + alignment with stakeholder's needs
- Design: creating a definition of how project goals will be achieved
- Implementation: process of writing code - normally split into many subprojects
- Building: creating a complete version of software - putting all code together + custom configurations for target deployment
- Testing:
  - Unit Tests: making sure that small independent parts of code work correctly
  - Integration Tests: all code parts work together
  - Acceptance: functionality meets requirements
  - Regression: New code doesn't break old
- Deployment: release of software into end user environment
- Maintenance: supporting software during its lifetime i.e new software updates

Normally, 80-90% of software system Total Cost Of Ownership (all costs associated with a product over its life cycle) is due to maintenance. This is because the cost of change is high + better to extend existing system to get better return on investment

1.2 Requirement Analysis

> The hardest single part of building a software system is deciding precisely what to build
>
> (Fred Brooks, "No Silver Bullet - Essence and Accident in Software Engineering")

It is impossible to know everything in advance required to build a system:

- Users don't know 100% what everything should do
- Developers don't know 100% how users would use it
- Only 100% certainty is that things will change

1.3 Software Engineering Aims

Software Engineering aims to manage complexity + minimise risks, as well as designing systems that can meet requirements, adapt to changes + withstand any unexpected use cases

2 Lecture 2 - Object Oriented Programming Design and Analysis

2.1 Object Oriented Programming

"Any computable function" can be achieved by:

- "Sequencing" → ordered statements
- "Selection" → conditions, if/else
- "Repetition" →, iterations

Object-Oriented Programming: a method of implementation in which programs are organised as cooperative collections of objects.
Object-Oriented Design: a method of design encompassing the process of object-oriented decomposition, i.e using objects + classes to describe the system
Object-Oriented Analysis: a method of analysis that examines requirements from the perspective of the classes and objectives found in the vocabulary of the problem domain

2.2 Abstraction

We can use an object-oriented approach to describe a system in terms of meaningful abstractions (classes), relationship and interactions between them.
Abstractions are generalisations that define certain key characteristics and behaviours e.g all cars have 4 wheels.
In Object-oriented design, recognising the sameness amongst things can allow us to create abstractions which cause smaller applications and simpler architectures.
Abstraction is about:

- Dealing with ideas
- Generalisation
  - To focus on what matters - on key characteristics and behaviours
  - To simplify and make complexity more manageable

2.2.1 Class

A class encapsulates data and behaviour. When thinking about object-oriented design, need to consider:

- What classes of objects are present in the problem?
- What behaviour should each class provide?
- What should happen when an action is requested of an object?

Concept	Data	Behaviour
car	speed	move/stop
bank account	balance	debit/credit
message	text	send

Figure 1: Example of classes

Note however, that we don't want an abundance of classes. Each class might only be useful for a particular application, and thus should be used for that purpose. For example, a student in the view of the Teaching Office is characterised by the modules they are taking, whereas from the tutor viewpoint they are characterised by their wellbeing.

2.2.2 Object

Each class can create multiple instances (objects), which can contain data/behave like the class it is made from. For example the bank account class can be used to create two bank accounts which can credit/debit cash

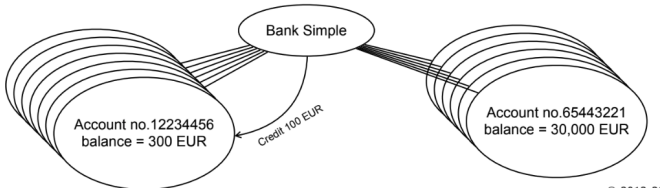


Figure 2: Two Objects derived from the same class

2.2.3 Encapsulation

Classes can provide abstraction. This means that an object can be used without the knowledge of how it works. We want to expose only what it can do and not how, e.g that a steering wheel will make a car turn left if we turn it left, without any knowledge about how this occurs. This is encapsulation

Suppose a person has a Name attribute. If we want to access the Name, we could do this directly:

```
class Person
{
    public String FIRST_NAME = "John"
    public String LAST_NAME = "Smith"
}
print Person.FIRST_NAME + Person.LAST_NAME
```

But, if we wanted to print the last name first e.g in France, then we could:

```
if (in France)
    print Person.LAST_NAME + Person.FIRST_NAME
else
    print Person.FIRST_NAME + Person.LAST_NAME
```

But, it would be easier to keep Name variables hidden (encapsulation), and just use a method to access them instead:

```
class Person
{
    private String FIRST_NAME = "John"
    private String LAST_NAME = "Smith"

    public String NAME
    {
```

```
    if (in France)
        return Person.LAST_NAME + Person.FIRST_NAME
    else
        return Person.FIRST_NAME + Person.LAST_NAME
    }
}
print Person.NAME
```

We use encapsulation to:

- Hide implementation
- Reduce dependencies between different parts of the application - decoupling
- Make changes safely
- Debug easier
- Make everything more manageable

Thus we should try to encapsulate as much as possible.

2.2.4 Getters/Setters

We use getters and setters to allow for controlled access to internal data fields.

```
class Person
{
    private String email;
    public String getEmail {
        return email;
    }
    public setEmail( String newEmail) {
    if ((newEmail != null) && (newEmail.contains('@'))){
        email = newEmail;
    }
    }
}
```

We can do this to implement constraint checking, as well as control concurrent access - allowing only one thread to access an object at a time. It also allows for the actual data source to be hidden so it can't be access.

> The point of encapsulation isn't really about hiding the data, but hiding design decisions, particularly in areas where those decisions may have to change.
>
> (Martin Fowler)

For example, if you want to communicate with an external data source, use encapsulation, because we want to focus on the messages delivered to the data source rather than the data source itself.
If we don't have controlled access to internal data fields, then this means that:

- We can't abstract the class
- We can't remove/reduce dependencies between different parts of the system

- We can't make changes safely
- We can't debug quickly

Thus, we should hide everything except what we want to expose.

2.3 Inheritance

Classes can be related:

- Superclass/base class (parent) can be extended/inherited from
- Subclass (child) extends/inherits from superclass

Each subclass can do everything a superclass can, but they can also:

- Hold additional data
- Perform new actions
- Perform original actions differently

For example with a shape:

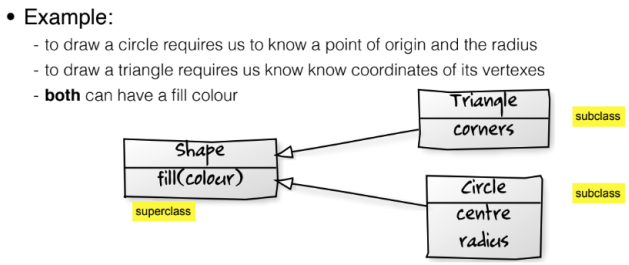


Figure 3: Inheritance

2.4 Polymorphism

Polymorphism allows us to request the same action from objects, but execute it in different ways. For example, if we have different bank accounts, but we want to find out how much money is in each account, then the action is the same - request the current balance. However, the way the balance could be calculated could be different. But we can still use polymorphism to simplify our code. This is an example of class inheritance hierarchy, which lets us choose the level of abstraction at which we interact with an object.
For example, compare this code:

```
for CurrentAccount in Bank
{
    TotalMoney += CurrentAccount.balance()
}
for SavingAccount in Bank
{
    TotalMoney += SavingAccount.balance()
}
for SuperHighInterestAccount in Bank
```

```
{
    TotalMoney += SuperHighInterestAccount.balance()
}
```

To this code:

```
for Account in Bank
{
    TotalMoney += Account.balance()
}
```

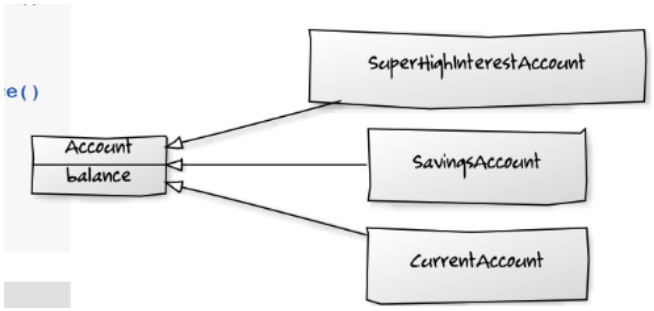


Figure 4: Polymorphism

Polymorphism seperates the declaration of functionality from the specifics of its implementation.

2.5 Identifying good Classes

Remember: Inheritance "IS A" Relationship - Subclass "IS A" Superclass i.e Coffee IS A Drink, Car IS A Vehicle

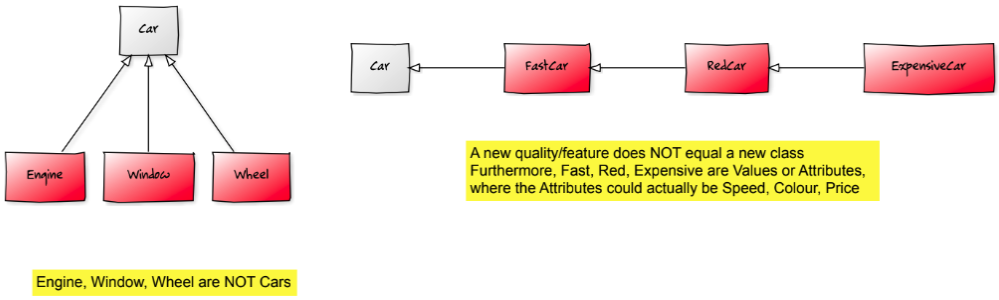


Figure 5: Bad Classes

2.6 Abstract Classes

We want our classes to be abstract - a class which can't be instantiated. This means that we can't define the class without going into more detail. This is good for capturing the higher level view of a system

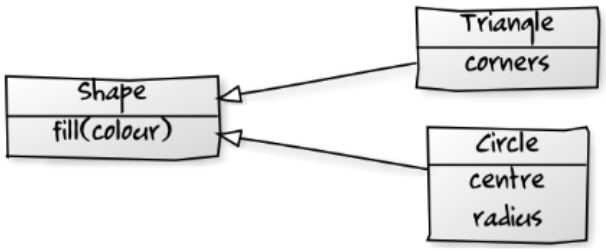


Figure 6: Abstract Class

2.7 Interfaces

An interface is a purely abstract class which defines only behaviour - not data. This is a point where two entities meet and interact. It defines a set of ways to interact with a class, or a contract of what must be avaliable. Interfaces help to add specific behaviour to classes. Most classes with extend one superclass, but will implement multiple interfaces.

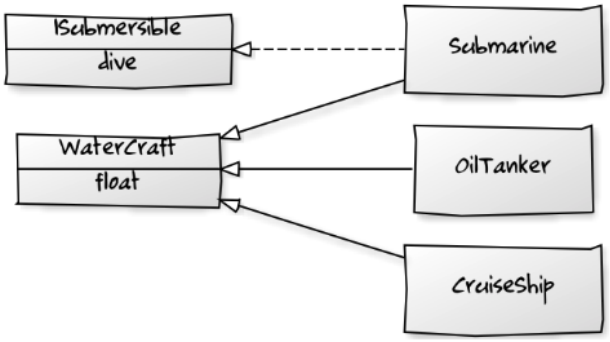


Figure 7: Interfaces

3 Lecture 3 -Introduction to UML

UML is a formal graphical language which has a set of diagrams for describing software systems. This can be used for designing, documenting and communicating software architecture + program behaviour.
There are 14 types of diagrams defined:

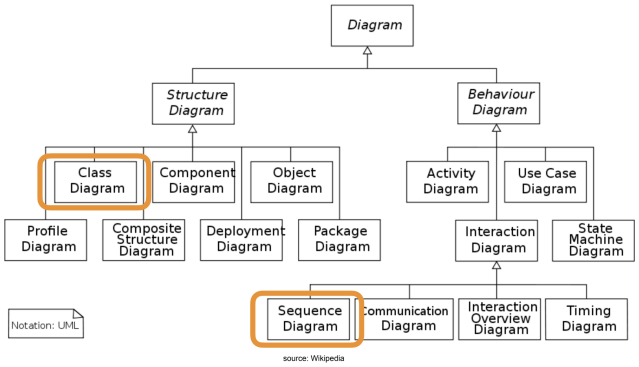


Figure 8: UML Types

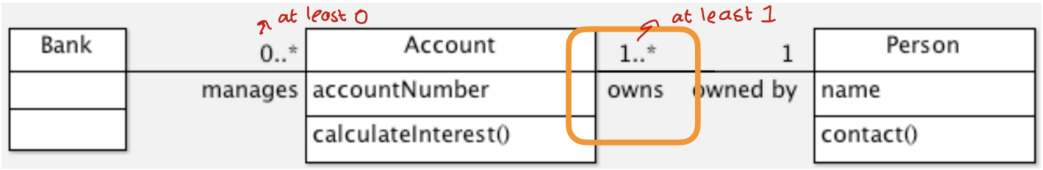
To describe a system we need:

- Structures: Items + how they are connected
- Behaviour: How items interact

UML, we use Class Diagrams to represent structures. These describe the software architecture by explaining what classes are in a system + their relationships.
Behaviour diagrams such as Sequence and Communication Diagrams show interaction between objects.
We want to use UMLAsSketch to communicate parts of the system to a group of people.

3.1 Class Diagram

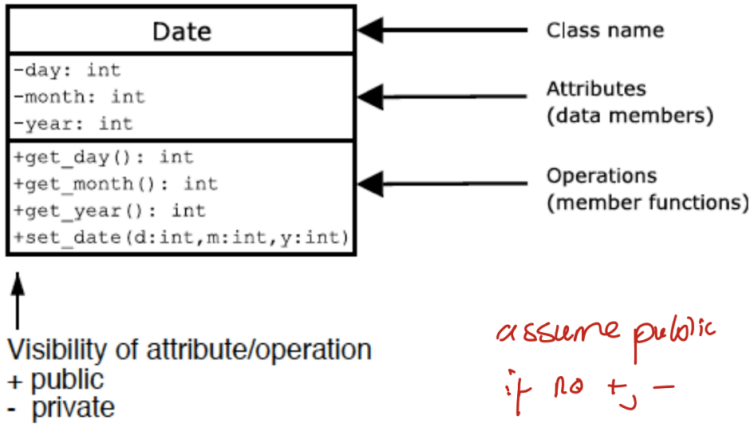
These show the classes in a software system, along with their attributes, operations and relationships:



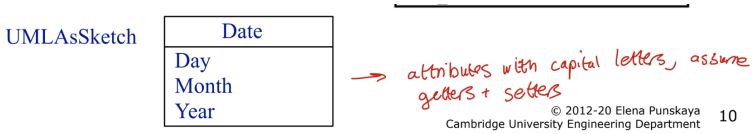
Explaining this diagram:

- Bank can manage at least 0 accounts
- An Account has an account number and can calculate interest
- Each Account can be owned by one Person
- One Person can own multiple accounts
- Each Person has a name and can be contacted

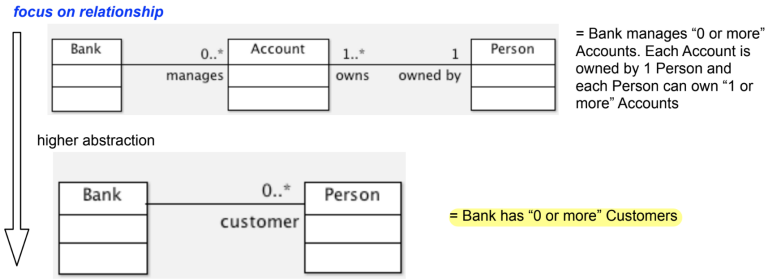
Each class is represented by 3 different sections: name, attributes and operations/methods:



We try to only include public features of a class in UML as they can be used in relation with other classes.



We want to have the right level of abstraction to keep the diagrams concise and useful:

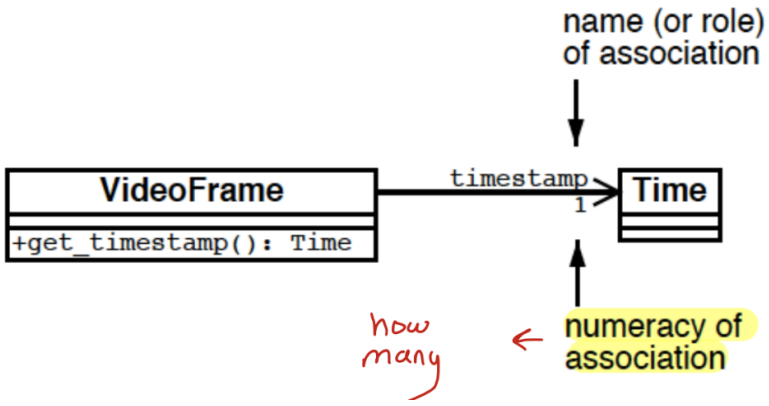


At some points, might be best to get rid of all methods and attributes to just focus on relationships.

3.2 Association

In class box, we represent attributes as a line of text i.e a reference to a Time Object:

But, we can also show that using association, with a solid line with an arrow. This indicates that the class uses an attribute of that object. This is mostly used for more significant classes.

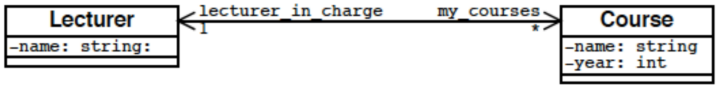


This line is:

- Directional - unidirectional association
- Meaningful - need to name what the attribute is defined as in the class
- Constrained - it is stated how many Time objects a VideoFrame can be in this relationship at the same time i.e 1. This can also be a range e.g 0..1, * (infinite amount) or 1..* (at least 1)

3.3 Navigability

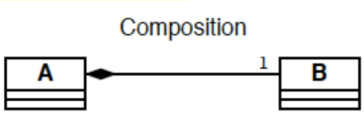
These associations can also do bidirectional navigability, as seen below.



From this we can see that each Lecturer object knows about all the courses they are teaching, and * indicates that they can be in charge of any number of courses. For each Course, there is only 1 lecture in charge.
We might not show the directional of associations if it is not relevant to the meaning of the diagram.

3.4 Composition - don't need to draw this in exam

We can include a diamond when having class association. A filled diamond indicates composition



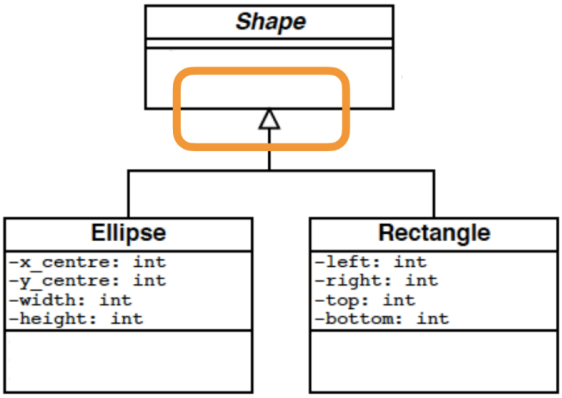
This means that if A is destroyed, then B will be destroyed too. These are good to show the strength of relationship between classes i.e exclusive ownership.

3.5 Aggregation - don't need to draw this in exam

An empty diamond means objects of class A contain objects of class B. But A doesn't control the lifecycle of B i.e destroying A doesn't destroy B.
For example, a college society has students, but closing the society won't destroy the students.

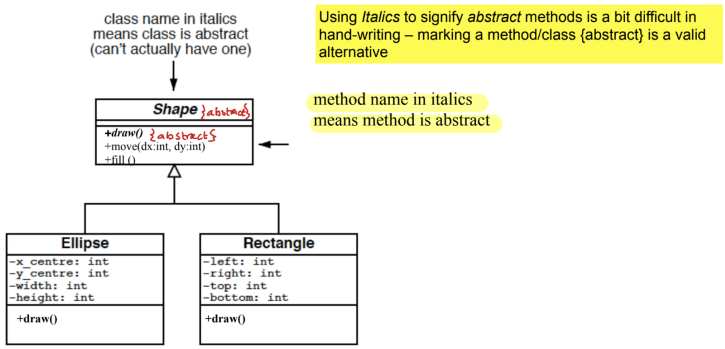
3.6 Inheritance

We can represent inheritance between classes as:



3.7 Abstract Methods

We can include abstract classes and methods in UML using italics. This can be hard to do in handwriting, so instead, can mark a class/method as {abstract}. If a class has an abstract method, then it must be declared as an abstract class.



3.8 Interfaces

Inheritance is useful when:

- Implementation inheritance - superclass implements some functionality which can be re-used by all subclasses
- Behaviour inheritance - class can expose a certain set of functionality

We can use interfaces for methods which might be access by different subclasses, as seen in Interfaces.

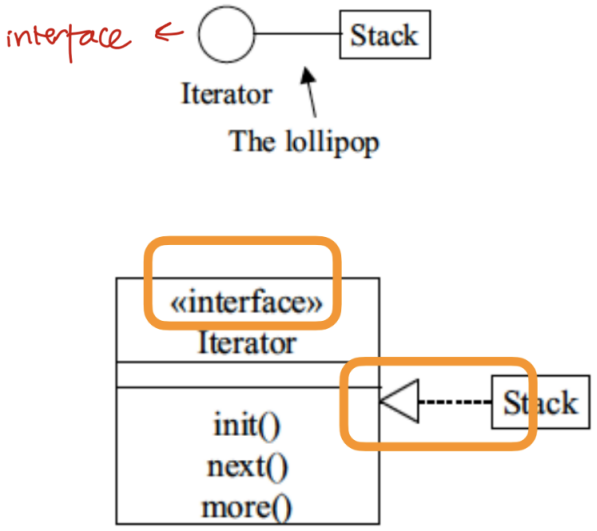
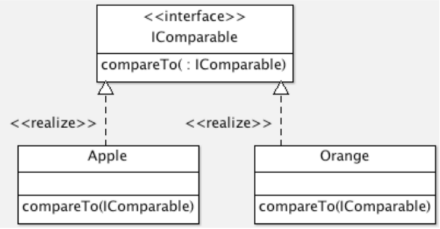


Figure 9: UML Interface

It is also common to start interface names with the letter I. Interfaces are great to apply the same functionality to very different concepts, like comparing apples to oranges.



Can start with
I if an interface

if Apples and Oranges "know" how to compare

```
IComparable apple = new Apple();
IComparable orange = new Orange();

apple.compareTo(orange);
```

3.9 Class Diagram Example

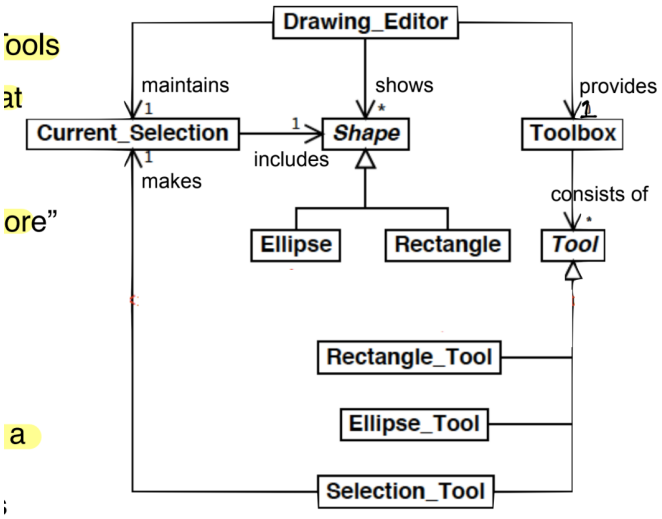


Figure 10: Drawing Editor Class

Explaining this diagram:

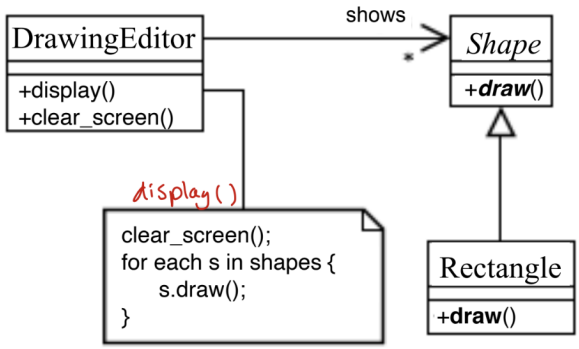
- Drawing Editor has 1 Toolbox
- Toolbox has any number of Tools
- There are 3 types of Tools: Rectangles, Ellipses and make a Selection
- The Drawing Editor can show any number of shapes
- There are 2 types of shapes: Ellipses and Rectangles
- Drawing Editor maintains the current Selection
- Only 1 shape can be selected at once

3.10 Dynamic View

Class Diagrams are good for capturing the relationships, but we need to use dynamic diagrams to show how objects interact with each other for a particular action. There are 2 choices:

- Sequence diagrams - they present a more clear view of the timeline. They are easier to read when there are a few objects, with a lot of calls.
- Communication diagrams - they focus more on the nature of collaboration between the object to perform an action. They are better if there are a few method calls between many objects.

To look at both of them, we can use a drawing editor example:



If we call the display() method, then we can look at each diagram and how it works:

3.11 Sequence Diagram

A Sequence diagram has a vertical axis, representing time, and white boxes to show the duration of each call.

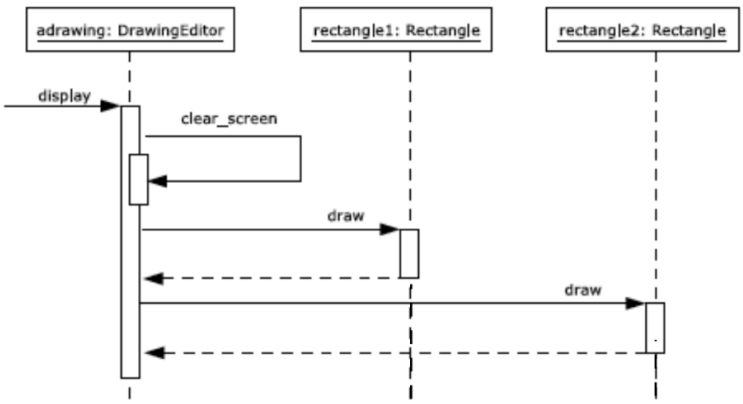


Figure 11: Sequence Diagram

This works by:

- display() is called in adrawing

- adrawing calls the clear_screen() method on itself
- adrawing calls the draw() method in rectangle1, which returns
- adrawing calls the draw() method in rectangle2, which returns
- the display() method is completed

We don't always need to include the return from the draw() class (in dashed lines).

3.12 Communication Diagram

These show the method flow between objects, by numbering method calls:

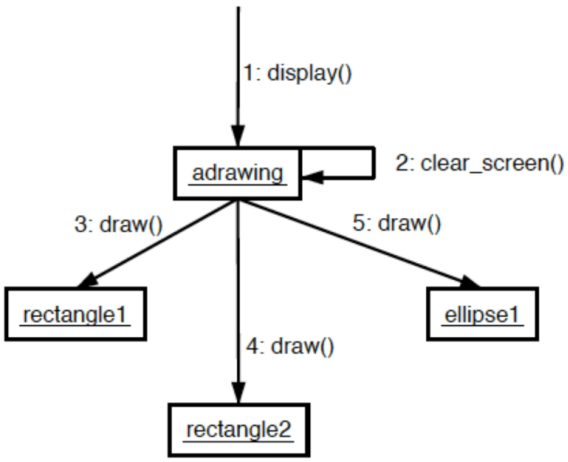


Figure 12: Communication Diagram

3.13 State Diagrams

If the behaviour of an object depends on its state e.g in embedded software, then we can use UML State Diagrams to show it

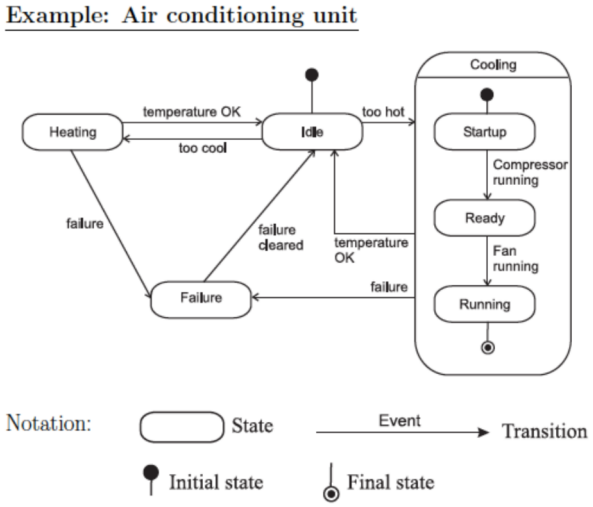


Figure 13: State Diagram

3.14 New Notation

Project Teams can expand UML to their own needs, as long as everyone on the project can understand it, and everyone's understanding is the same