Object Oriented Methodology

Week – *1*, Lecture – *1*

SDLC Basics

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

What is a software?

There are no hard definitions of software, but there are many colloquial ones

- For instance, in the early days, a computer was said to be made up of “hardware” and “software”
- Everything that “can be touched physically” was considered as “hardware”…
- … and everything that “cannot be touched physically” was “software”
- Of course, today, this definition seems a bit creepy !!

In any case, an informal definition that is good enough for our purpose is given below
“*A software is a collection of programs along with suitable documentation*”

This simple definition stresses on two aspects

- First, it is a “collection” of programs; in general, a software consists of multiple collaborating programs
- Second, it is accompanied by some “documentation”; you will create some of it for your term projects

P.S. – the plural of software is also software, so don’t be confused !!

Developing Software

There are many steps involved in building commercial software products

While the actual number of steps discussed in literature vary, there are three major phases

- These phases usually happen in a cycle over multiple iterations

In the first phase, we analyse the problem at hand, and prepare prospective plans

- Typically, we analyse if the project is feasible subject to the available people and resources
- If so, we chalk out detailed plans to for the complete development process
- We also prepare some sketches and documents, which can guide the upcoming phases

Next, we begin the process of implementation

- This may involve finding and using suitable libraries or reusing code fragments from previous projects
- It usually also involves writing new code fragments to supplement existing parts

Last, we release or deploy the software, and provide support to the users

- The process is also called as “Maintenance” – aka keeping the software free of problems

Software Development Lifecycle – 1/2

The Software Development Lifecycle (SDLC) is a common term used to refer to the development process

- The actual steps of development may not walk through these phases religiously…
- … but the general development process may resemble SDLC closely

The Planning Phase

- We begin by analysing the requirements of the project and evaluating if they are feasible
- Two most important aspects of feasibility are the development cost and skills of the developers
- The next step is to create some initial design – a bird's eye view of the software
- The designing process continues till we have "sufficiently detailed" design documents

Software Development Lifecycle – 2/2

The Implementation Phase

- Implementation refers to actually developing the software at the level of code and binaries
- This includes writing new code, reusing code from previous projects and using third-party libraries…
- … to achieve the functionality expected from the software
- The development is tightly coupled with Testing – the process of checking code fragments for problems
- Testing is performed at lower (or Module) levels as well as higher level (for the system or a group of Modules)

The Maintenance Phase

- At the end of development, the software is finally deployed or released
- But the job of the development team doesn't end there – it is only the beginning of a long journey
- Throughout a software's planned lifecycle, the developers keep performing Maintenance
- Maintenance may be performed to weed out existing problems…
- … or, to enhance the software with new features

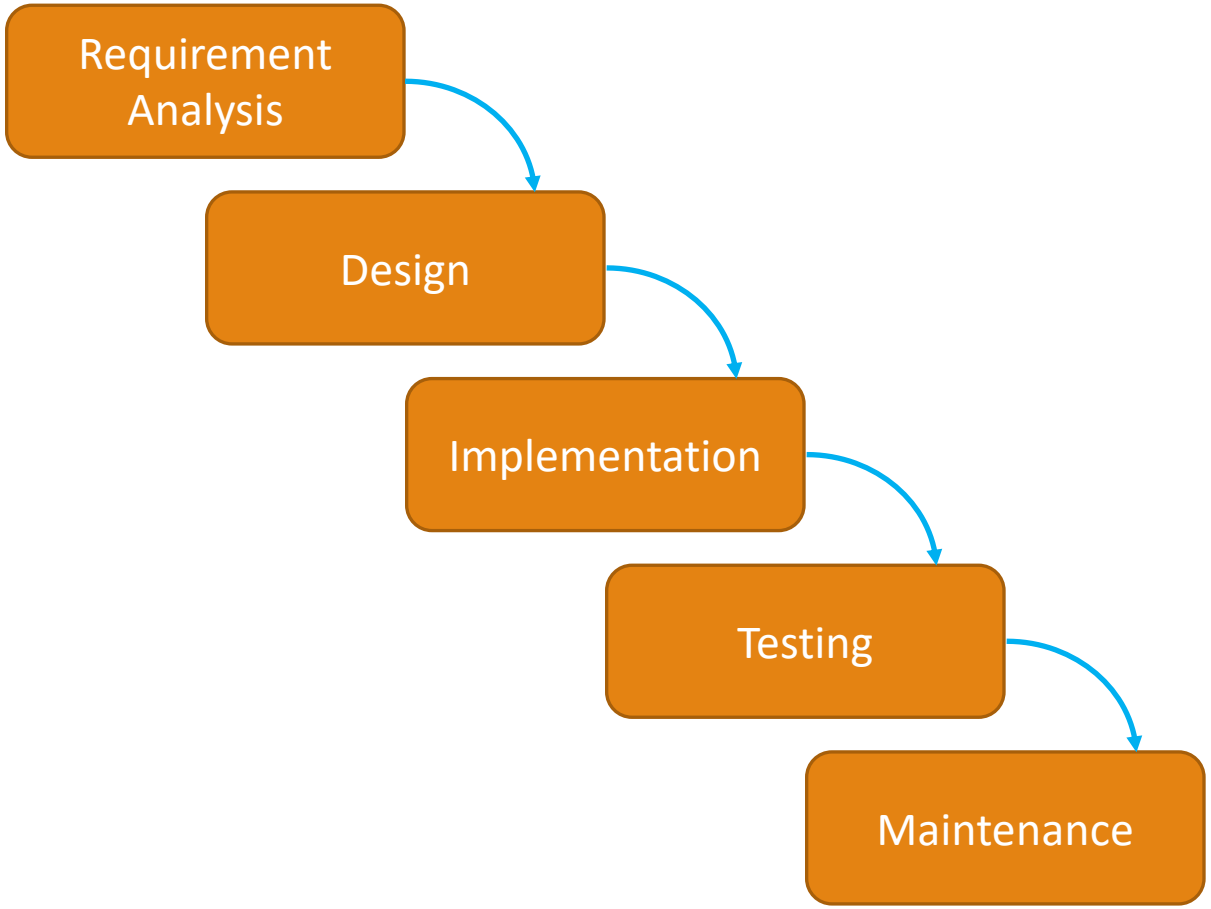
The Waterfall Model of development

The earliest method of developing software was probably the Waterfall Model

The Waterfall model defines the steps involved in software development in a sequential order

- It is akin to the fall of water from the top to bottom, hence, the “Waterfall” model

The Waterfall Model of development



The Waterfall Model of development

The earliest method of developing software was probably the Waterfall Model

The Waterfall model defines the steps involved in software development in a sequential order

- It is akin to the fall of water from the top to bottom, hence, the “Waterfall” model

The steps of the Waterfall model generally include

- Requirement Analysis – collecting and analysing the requirements related to the software
- Design – coming up with an Architecture for the software (a blueprint of the finished product)
- Implementation – writing or absorbing code fragments from third-parties to achieve functionality
- Testing – testing the software at module level, after inter-module integrations and the overall software
- Maintenance – providing post-deployment support for the software

The Design Phase

In this course, we will have a look at some of the activities performed in the Design phase

In the Design phase, we come up with some *Design Documents*…

- … which act like guidelines for the implementation phase

Typically, we come up with *Architectural Views* and *UML Diagrams*

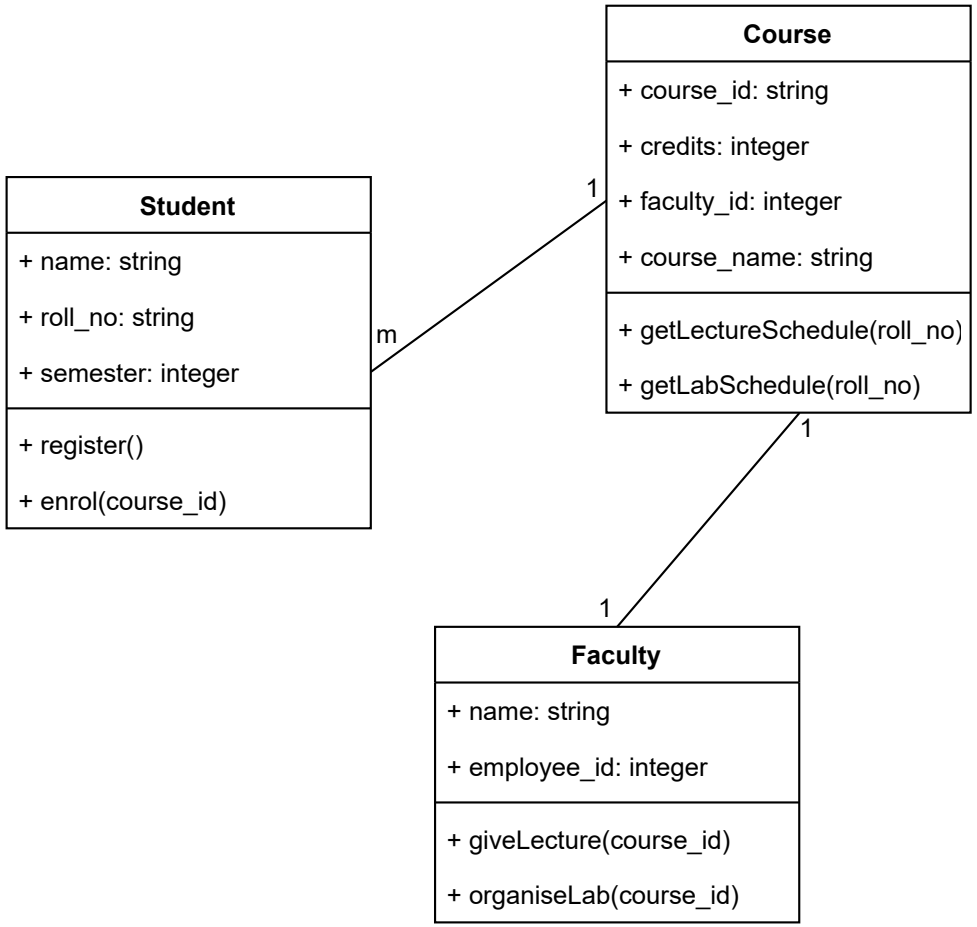
Architecture can be considered to be a kind of “higher-level” Design

- For now, don’t worry about it… assume that they are the same thing !!

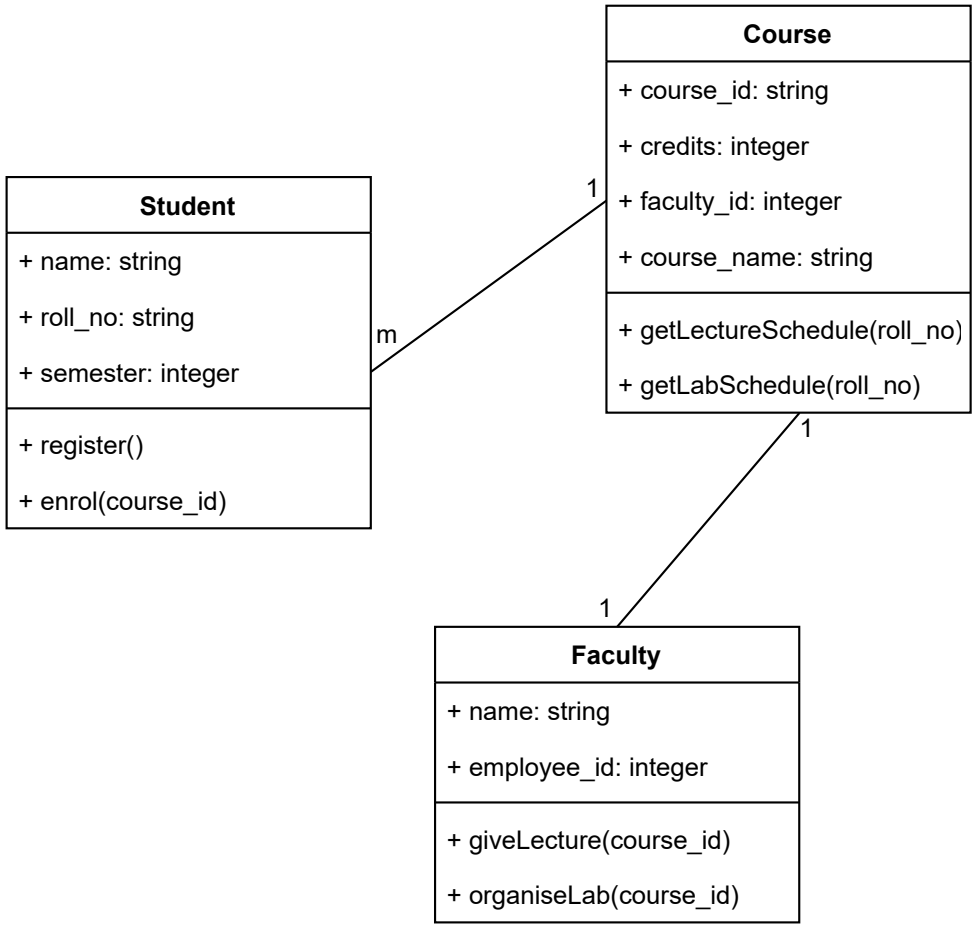
Design documents map the requirements of a software to some formal elements

- One such element is a *Class* – we will discuss them later this week
- The diagram represents the relationship between the classes is known as a *Class Diagram*

A “not-so-perfect” Class Diagram



A “not-so-perfect” Class Diagram



We will see more examples as we move further in the course

Imperative vs Declarative Paradigm

Imperative Paradigm focusses on *how* the goals are achieved

- We provide all the details of computation required to perform a task

Declarative Paradigm focusses on defining *what* the goals are

- We assume that the goals can be achieved “somehow”
- We don’t care about the details, as long as we get the required results

How are these code fragments different?

```
...
printf("Enter a positive integer:");
scanf("%d", &num);


long factorial = 1;
int copy_of_num = num;
while(copy_of_num > 0)
    factorial *= copy_of_num--;


printf("The factorial of %d is %ld", num, factorial);
...
```

```
...
printf("Enter a positive integer:");
scanf("%d", &num);


long factorial = factorial(num);


printf("The factorial of %d is %ld", num, factorial);
...
```

How are these code fragments different?

```
...
printf("Enter a positive integer:");
scanf("%d", &num);


long factorial = 1;
int copy_of_num = num;
while(copy_of_num > 0)
    factorial *= copy_of_num--;


printf("The factorial of %d is %ld", num, factorial);
...
```

```
...
printf("Enter a positive integer:");
scanf("%d", &num);


long factorial = factorial(num);


printf("The factorial of %d is %ld", num, factorial);
...
```

How are these code fragments different?

```
...
printf("Enter a positive integer:");
scanf("%d", &num);

long factorial = 1;
int copy_of_num = num;
while(copy_of_num > 0)
    factorial *= copy_of_num--;

printf("The factorial of %d is %ld", num, factorial);
...
```

```
...
printf("Enter a positive integer:");
scanf("%d", &num);

long factorial = factorial(num);

printf("The factorial of %d is %ld", num, factorial);
...
```

We specify *how* our goals are achieved?

How are these code fragments different?

```
...
printf("Enter a positive integer:");
scanf("%d", &num);
```

We specify *how* our goals are achieved?
This is a more *Imperative* approach

```
long factorial = 1;
int copy_of_num = num;
while(copy_of_num > 0)
    factorial *= copy_of_num--;
```

```
printf("The factorial of %d is %ld", num, factorial);
...
```

```
...
printf("Enter a positive integer:");
scanf("%d", &num);
long factorial = factorial(num);
printf("The factorial of %d is %ld", num, factorial);
...
```

How are these code fragments different?

```
...
printf("Enter a positive integer:");
scanf("%d", &num);


long factorial = 1;
int copy_of_num = num;
while(copy_of_num > 0)
    factorial *= copy_of_num--;


printf("The factorial of %d is %ld", num, factorial);
...
```

```
...
printf("Enter a positive integer:");
scanf("%d", &num);

long factorial = factorial(num);


printf("The factorial of %d is %ld", num, factorial);
...
```

We specify *what* our goals are…
We don't care *how* they are achieved?

How are these code fragments different?

```
...
printf("Enter a positive integer:");
scanf("%d", &num);


long factorial = 1;
int copy_of_num = num;
while(copy_of_num > 0)
    factorial *= copy_of_num--;


printf("The factorial of %d is %ld", num, factorial);
...
```

```
...
printf("Enter a positive integer:");
scanf("%d", &num);

long factorial = factorial(num);


printf("The factorial of %d is %ld", num, factorial);
...
```

We specify *what* our goals are…
We don't care *how* they are achieved?
This aligns more with a *Declarative* approach

Imperative vs Declarative Paradigm

Imperative Paradigm focusses on *how* the goals are achieved

- We provide all the details of computation required to perform a task

Declarative Paradigm focusses on defining *what* the goals are

- We assume that the goals can be achieved “somehow”
- We don’t care about the details, as long as we get the required results

Levels of *abstraction* are “relative”

- Even the ”imperative example” uses abstractions for input/output (`printf` and `scanf`)
- Thus, there are no hard definitions for *imperative* and *declarative* approaches

Java is somewhere ”in between” on the ”abstraction hierarchy”

- C is “arguably” below Java
- MATLAB is “arguably” above Java

Homework !!

Read more about SDLC and the Waterfall Model

- These two links may be enough:
  https://www.tutorialspoint.com/sdlc/sdlc_overview.htm
  https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm
- While you may read about other development models as well, it is better if you stick to Waterfall for now !!