Object Oriented Analysis & Design

Week 3

Programming Philosophies

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 2

- The Software Development Lifecycle (SDLC) forms the basis for all software development methodologies (processes) (SDM). It has 4 simple steps namely planning, analysis, design and implementation.
- The main SDMs are waterfall, iterative, incremental and spiral. These are applied to Rapid Application Development (RAD), Agile methodologies, Extreme Programming (XP) and Scrum.
- The two approaches to system analysis and design are structured (SSADM) and object oriented (OOAD). The latter offers distinct advantages over the former.
- There are 3 key elements to every OO methodology: notation, process and tools.
- The Unified Modelling Language (UML) is the language used used to specify, visualize, construct, and document the artifacts (major elements) of the software system.

Content

- Introduction
- Object Based vs Object Oriented programming
- History of Programming Languages
- Implementation of Object Oriented Principles in Programming
- Example Implementations in Select Popular Languages

Part 1

Introduction

Introduction

- In lesson one the fundamental concepts of object oriented programming were introduced. Further the evolution of different paradigms building up to the object oriented paradigm was also discussed.
- Table 1 in lesson 1 described the differences between structured programming and object oriented programming. This was actually a limited introduction to what structured programming is all about (this course is, after all, about object oriented stuff right?). Well, not quite.
- The comparison of structured programming with object oriented programming is a lot like contrasting philosophies; without understanding one you will not fully grasp the other.
- Let us examine this from a day to day perspective. How do you know day has broken? It is because it is no longer night. In equatorial parts of the world day and night are usually balanced in terms of hours (roughly 12 hours of daylight and 12 hours of darkness). However, this is not the case with the rest of the world (depending on which season it is). When this author visited America for the first time it was pure confusion; there was daylight up to around 8 pm. So I almost missed supper on the first day! Around the equator it gets dark before 7 pm, and so this was a new experience.

Introduction (cont'd)

- The principle here is that in order to know one you must know the other. You can't know it is dark until you experience light and vice versa. Similarly unless you experience winter you won't know what extreme cold is. This concept is even extended to spirituality; there is good and evil, honesty and dishonesty, peace and war, and so on.
- In this introduction we revisit the principles of structured programming before delving into more details regarding object oriented programming. Perhaps the question the learner has is "why learn anything about programming when this course is about object oriented analysis and design?"
- The answer to this is that you can not learn about analysis and design without knowing something about programming. This is the next step after making out your system design and they go hand in hand in the overall methodology (even though programming has its bigger foot in the implementation phase of the SDLC).
- Recall that one of the major differences between structured and object oriented programming is that the former solves specific programs and thus code can't be reused; however, it uses modularity (the modular paradigm) to solve bigger problems using a top down approach.
- In structured programming the program development process may be broken into several steps in order to achieve the final solution. This is demonstrated in fig 1.

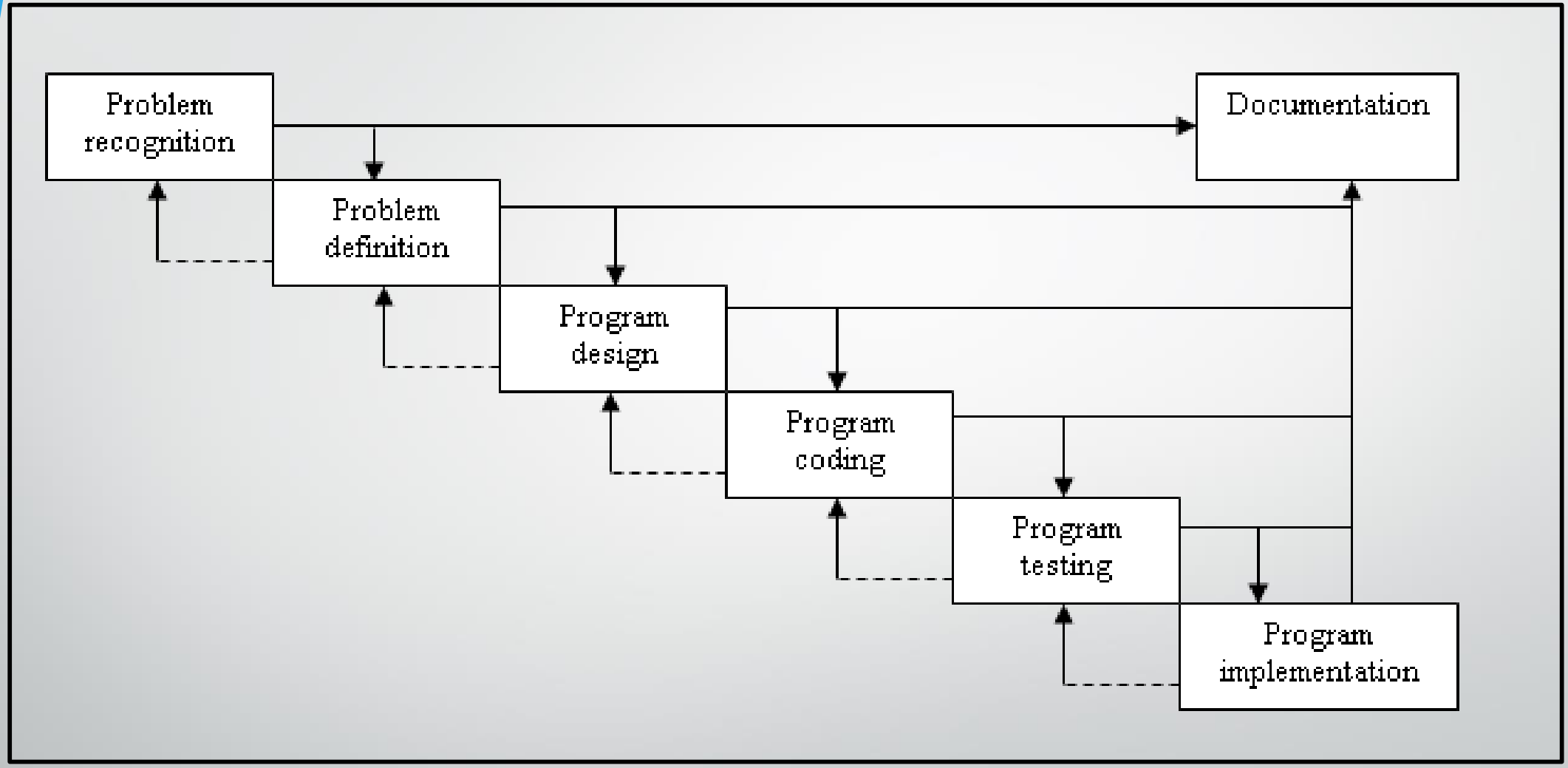

Fig 1. Program development process (Hawi, 2014)

Introduction (cont'd)

- No doubt the resemblance between fig 1 and the SDLC is not lost on the learner. This is because every software development process draws from the SDLC; the steps are just logical. A brief description of the process:
- Note that the process is follows the waterfall model in terms of implementation, that is, one step is completed then the next step begins. However, the difference is that iteration is allowed in that at the end of a step it is possible to go back to the previous step (the original waterfall model was rigid in this area). There are 6 steps involved in this development cycle:
- Problem recognition – this is the first step. In this step the programmer recognizes there is a problem that can be solved using a computer program. This may involve the simple automation of a process, or something more complex like the computation of a sales invoice. Thus in the first step we seek to solve a problem or exploit an opportunity.
- Problem definition – in this step the identified problem is moved from the recognition domain to the definition domain (the computer world so to speak). This essentially means determining the following: input, process, output. For example in the case of a sales invoice the input is the different sales items and quantities, the process is the calculation of individual sales items (for example, 3pcs of hammers at USD 5 each gives 3 X5 = USD 15 and so on) and the output is the total cost of the sale (hammers + super glue + masking tapes, and so on: total sale = USD 42). It is after this stage that the programmer writes a requirements document (i.e. what is needed to design the program).

Introduction (cont'd)

- Program design – this is the phase where the actual design is done by use of algorithms. An algorithm is a series of steps that show how the problem will be solved. It is here that the design will show how the data will flow and the series of instructions needed (recall that structured programming focuses on processes then data). Algorithms can be written in plain English or pseudocode. Aids like flowcharts also assist in this phase.
- Program coding – this is the phase where the design is converted into a suitable programming language such as Pascal or C. The different programming languages belonging to different groups will be discussed later in this lesson.
- Program testing – the program is now tested for any errors at this stage. Different kinds of errors may arise such as syntax or logical errors, hence the need for thorough testing. The process of removing these errors is called debugging. This is one phase where the need to iterate between coding and testing is essential.
- Program implementation – this is the actual delivery and installation of the new program ready for use. The new program will obviously change the way things are done when implemented hence the need for review and maintenance.
- ***Review and maintenance**: this is important because of the errors that may be encountered after the program has been implemented or exposed to extensive use. A program may also fail not because of poor development but also due to poor use.

Introduction (cont'd)

- Documentation - This is the writing of support materials explaining how the program can be used by users, installed by operators or modified by other programmers. All stages should be documented so as to help during future modifications.
- It can either be ***internal*** or ***external***:
- ***Internal***: is written in non-executable lines (comments) in the source code that help other programmers to understand some code statements.
- ***External*:** reference materials such as user manuals printed as booklets
- There are 3 types of external documentation: User oriented, operator oriented, programmer oriented.

Introduction (cont'd)

- User-oriented - enables user to learn how to use the program as quickly as possible and with little help from the program developer (has no technical terms).
- Operator-oriented - meant for computer operators such as the technical staff. It helps them to install and maintain the program.
- Programmer-oriented - detailed documentation written for skilled programmers. Provides necessary technical information to help in future modification of the program.
- Thus we notice the subtle differences between this process and the object oriented one due to the differences between structured programming and object oriented programming. Let us now refresh on the foundation principles of object orientation.

Introduction (cont'd)

- The foundations of object orientation are built upon abstraction, encapsulation, modularity and hierarchy. The need to review the meaning of these terms is that whereas they were introduced from an analysis and design perspective, we wish to examine them from an object oriented programming perspective in this lesson.
- Abstraction – leaving out details that aren't important in finding the solution to a problem. In a given case study a lot of information is given regarding the problem domain. As the programmer it is important to determine what information is relevant to solving the problem; without doing this the program development will be extremely tedious and a waste of man hours (remember the idea is to manage complexity).
- Encapsulation – in programming encapsulation is the process of hiding the inner workings of the object from the environment; that is to say, what is private remains hidden and what is public can be viewed and used by all. The term used for this is the black box.

Introduction (cont'd)

- Modularity – this simply means breaking down the problem into smaller manageable units. Writing codes for these smaller units is easier than tackling the whole problem as one (what is referred to as a monolithic approach in programming.
- Hierarchy - Booch et al. (2007) define hierarchy as simply an ordering of abstractions. By arranging the abstractions in order the development of the system is made much easier and the problem can be understood much better. Using hierarchy the system can be arranged in order of increasing abstraction. Hierarchies can be described as singular inheritance (ISA) or multiple inheritance. They can also be described as an object structure (part –of relationship). In programming hierarchy allows for reuse of code and better structure of the program.
- Having discussed the intricacies (at least from an introductory perspective) of structured programming, and refreshed ourselves on the 4 principles that make the foundation of object orientation we can proceed to the rest of the lesson.

Introduction (cont'd)

- In the remaining parts we wish to examine 4 concepts related to object oriented programming. Of course there is a separate course dedicated to object oriented programming and so in this course we just look at an introduction to put your feet on the ground as far as this area is concerned.
- First an examination of object based languages and how they differ from object oriented languages. This is followed by a history of programming languages (essential to understand where programming started from to where it is now). Thereafter is a discussion of the implementation of object oriented principles in programming. Lastly some example implementations in popular languages (we examine 2 popular languages taught at university level, Java and Python).

Part 2

Object based vs Object oriented Programming

Object based vs Object oriented

- In the object oriented world there are many terms that are used, sometimes even inadvertently interchangeably, to mean different things.
- The terms object based and object oriented are used in different literature. Do they mean the same thing? The answer is no. They are different.
- A language that is object oriented is also object based; however, an object based language is not object oriented. Understand?
- Well what this means is that object based languages lack some of the characteristics of object oriented languages. The differences can be summarized as follows:
- Object based languages have the following characteristics (tutorialpoint.com):
- Object based languages supports the usage of object and encapsulation.
- They does not support inheritance or, polymorphism or, both.
- Object based languages does not supports built-in objects.
- Javascript, VB are the examples of object bases languages. Other examples include Ada and Smalltalk.

Object based vs Object oriented (cont'd)

- Further, "The term object-based language may be used in a technical sense to describe any programming language that uses the idea of encapsulating state and operations inside objects." (wikiwand.com). In addition to not supporting inheritance they also do not support sub typing (almost like inheritance but mostly refers to interfaces; however, it can also be used in the context of inheritance. See https://www.cs.princeton.edu/courses/archive/fall98/cs441/mainus/node12.html for further detailed explanation)
- Object oriented languages on the other hand have the following characteristics (tutorialspoint.com):
- Object Oriented Languages supports all the features of Oops including inheritance and polymorphism.
- They support built-in objects.
- C#, Java, VB. Net are the examples of object oriented languages. Other examples include Python and Ruby.

Object oriented vs Object based (cont'd)

- Another term that you may come across is prototype programming. This belongs in the domain of object oriented programming methodology. The idea with this type of programming is to reuse code based on existing objects called prototypes. The objects are generalized and thus can be cloned or extended (meaning other objects can inherit from them). The difference between this and the class based paradigm is the cloning aspect where an object can be cloned from another "generic" similar object.
- Table 1 summarizes the differences between object based programming languages and object oriented programming languages.

Point of Distinction	Object Oriented Language	Object Based Language
OOP Features	Support All Features (Abstraction, Encapsulation, Inheritance, Polymorphism).	Does not support all features. Example: Polymorphism and Inheritance are not supported.
Build in Object	No	Yes, E.g. JavaScript window Object.
Languages	Java, C++, C#, Smalltalk etc.	JavaScript, Visual Basic.

Table 1. Object based vs object oriented (programsbuzz.com)

Part 3

History of Programming Languages

Introduction

- Most programmers are thrown into the world of programming vide some language. At university level most programming courses begin with teaching the learner a language such as C.
- The subsequent courses then advance the learner through a sequence such as C →C++ → Java → Python, or something similar. Why is this the case? Why not just start you off directly on Python or Ruby from the word go?
- The answer to this lies in the concepts that the user is exposed to progressively. The user starts off with learning C (which is considered the simplest) and progresses by learning new programming constructs and styles over time.
- At some point you realize that with the foundation you have learnt it gets easier to learn new programming languages. Programming has come a long way and it goes without saying that the student of software engineering needs to have a grasp and the different advancements in programming from the days of writing machine code to writing solutions using object oriented programming.
- This section purposes to do exactly that; the distinction between high level programming and low level programming is also explored and explained with successive generations.

3.1 Types of programming languages

- Many programming languages have been developed over the years. However, these languages are classified into 2 major types namely:
  - Low-level languages
  - High-level languages
- These levels are spread over 5 generations. The first and second generations consists of low-level languages while the third through to fifth generation consist of high-level languages.
- Low-level languages are considered low because they can easily be understood by the computer directly (machine language) or require little effort to translate into machine readable form (assembly language); this explains why they constitute the first two generations of computer languages, since back then the concept was simply to get the computer to understand what you want it to do in a language as close to 0s and 1s as possible.
- High level languages are called so because they are very close to human languages (English like) and they can be read and understood even by people who are not experts in programming. There are many types of high –level languages and each of them was developed to address a particular problem solving domain while others came about due to advancement in technology.

3.2 First generation languages

- These languages consist mainly of binary logic (i.e. os and 1s). A program written in machine language might look like this:
- To calculate `wages = rates * hours` in machine language:
- **11100011 00000001 10000011 // load (wages)**
- **00011100 10001101 // multiply (rates * hours)**
- **10001111 11111000 10000001 // store (the computed wages)**
- The // sign is a common sign used to indicate comments (not part of the program). It was not in use back then but is used here for easier understanding of the program.

3.3 Second generation languages

- These were developed to overcome the short comings of machine languages. In an attempt to make them more readable, they allowed programmers to use names instead of binary numbers. The programmers could thus write programs as a set of symbolic operation codes called mnemonics
- Mnemonics are basically shortened 2 or 3 letter words. A program written in assembly language may look as follows:
- MOV CT, 25 (move 25 to register AX)
- SUB CT, 15 (subtract 150 from the value in AX)
- Key: MOV – Move; SUB – Subtract; AX – data register
- First line – move the value 15 into the processor register with the name AX.
- Second line – subtract 10 from the value in register AX
- Programs written using mnemonics need to be converted to machine code which the computer understands. Assemblers are used to make this conversion from assembly language to machine code.

3.4 Third generation languages

- Also known as ***structured/procedural languages***
- A procedural language enables a program to be broken down into smaller components called *modules* each performing an independent task. This is referred to as *structured programming* (to be discussed later in detail).
- Examples of 3 GLs include C, Fortran and Pascal
- All high level languages require either a compiler (translates before running) or an interpreter (translates at runtime) to translate code from high level language to machine code.
- Let us examine some characteristics of a few languages that fall in this category.

3.4.1 Fortran

- It was designed by John Backus in 1954.
- It was designed at IBM by John Backus to efficiently translate mathematical formulas into IBM 704 machine code. Wanted code at least as efficient as hand-coded.
- Language design was secondary to compiler design for optimization
- 1954 Report for a proposed Formula Translating System
- 1957 FORTRAN language manual published
- Translator produced code that in some cases was more efficient than the equivalent hand-coded program.
- Fortran introduced the following innovations:

3.4.1 Fortran (cont'd)

- A language based on variables, expressions, statements
- It also introduce the form of the arithmetic-assignment statement
- Conditional and repetitive branching control structures were introduced
- Arrays with maximum size known at compile time
- Fortran also introduced a provision for comments, which was hitherto not possible.

3.4.2 Lisp

- It is an interactive functional language
- Designed for IBM 704 by John McCarthy at Dartmouth 1956-1958
- Implemented at MIT. First reference manual published in 1960.
- Language based on lambda calculus. (Mathematical notation for expressing functions.)
- LISP was designed for symbolic formula manipulation. Stands for LISt Processor.
- Has become standard language of the AI community (which is why it is of interest).
- LISP introduced the following innovations:

3.4.2 Lisp (cont'd)

- The function as the basic program unit
- The list as the basic data structure
- Dynamic data structures
- Facilities for "garbage collection" of unused memory (taken up by other languages such as Java, after this)
- Use of symbolic expressions as opposed to numbers
- Recursion and the conditional expression as control structures
- The "eval" function for interactive evaluation of LISP statements

3.4.3 C

- It was designed by Kenneth Thompson and Dennis Ritchie at Bell Labs in 1972.
- Designed for coding the routines of the UNIX operating system; it is the language used to write most operating systems.
- "High level" systems programming language which created the notion of a portable operating system
- Concise syntax – programs somewhat hard to read, understand, debug, maintain
- No built-in operations for handling composite data types such as strings, sets, and lists.
- Not strongly typed. No run-time type checking. Easily leads to programming errors.
- Provides ability to code low-level operations in a high-level language.
- C opened the door for the development of other high level languages such as C++ and Java

3.5 Fourth generation languages

- 4 GLs make programming even easier. They present programmer with more programming tools such as command buttons, forms.
- Instead of writing lines upon lines of code, the programmer selects graphical objects on the screen called controls then uses them to create designs on a base form. The programmer may also use an application generator that works behind the scenes to generate the necessary code, hence the programmer is freed from the tedious work of writing the code.

3.6 Fifth generation languages

- These languages are designed around the concept of solving problems by enabling the computer to depict human like intelligence. They are designed to make the computer solve the problem for the programmer rather than the programmer spending a lot of time to come up with a solution. With such languages, the programmer only worries about what the problem needs to be solved i.e. the conditions that need to be met.
- Examples of 5 Gls: PROLOG, Mercury, LISP and OCCAM

3.7 Object oriented programming languages

- The concept of OOP is to look at a program as various objects interacting to make up a whole. Each object has specific data values that are unique to it called (*state*) and a set of the things it can accomplish called (*functions or behavior*). Several objects can then be linked together to form a complete program. OOP has greatly contributed to development of graphical user interface systems and application programs
- Examples of 5 GLs: C++, Java, SmallTalk, Simula.

3.8 Web scripting languages

- These languages are used to develop or add functionalities on web pages.
- *Web pages* are hypertext documents created in a language called Hypertext Markup Language (HTML). Another example of scripting language is Javascript, PHP and VBScript

3.9 Characteristics of good programming languages

- Having described the characteristics of each group of programming languages let us wind up this section by describing the characteristics of a good programming language:
- **Readability**: A good high-level language will allow programs to be written in some ways that resemble an English-like description of the underlying algorithms.
- **Portability**: High-level languages, being essentially machine independent, should be able to develop portable software i.e. can be transferred from one machine to another
- **Generality**: Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.
- **Brevity**: Language should have the ability to implement the algorithm with less amount of code. Programs expressed in high-level languages are often considerably shorter than their low-level equivalents.

3.9 Characteristics of good programming languages (cont'd)

- **Error checking**: Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.
- **Familiar notation**: A language should have familiar notation, so it can be understood by most of the programmers.
- **Quick and Efficient translation**: It should admit quick translation and permit the generation of efficient object code.
- **Modularity**: It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.
- **Widely available**: Language should be widely available and it should be possible to provide translators for all the major machines and for all the major operating systems.

Part 4

Implementation of OO Principles in Programming

Introduction

- Booch et al. (2007) state that the conceptual framework for all things object oriented is the object model. This model consists of 4 major concepts:
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
- This means that for anything to be called object oriented it MUST have the 4 properties. A minor requirement of the object model is to have (essential but not a must):
  - Typing
  - Concurrency
  - Persistence
- This section examines how the MUST haves are used in object oriented programming (OOP).

4.1 Abstraction

- Booch et al. (2007) define abstraction as: "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."
- The key challenge in applying abstraction in an object oriented domain is to determine which characteristics of an object should be used and which ones should be left out (ignored). This is done by the designer and so it depends on the nature of the problem that requires to be solved.
- As discussed in lesson 1 every object has state, behavior and identity. Abstraction is therefore applied in determining what the identity, state and behavior of an object is of interest to us. Let us apply this to a simple problem like in the design of a university classroom attendance application.
- In such a problem we are interested in the process of attendance and what goes on: A university professor will go to class to give a lecture. Students are expected to attend all lectures. Lectures consist of different courses. What information about the university professor do we wish to capture? What of the student? What of the course? And what of the university lecture rooms?

4.1 Abstraction

- As can be seen a lot of information is available, but we are interested in just specific information for the purpose of designing our system.
- Let us consider the professor as an example: we determine that we are interested in their staff number, name, course taught, lecture room. We are also interested in the following actions to be done by him/her: open room, switch on projector, switch off projector, and close room. We can capture this in OOP as follows:
- Class name: Professor
- Attributes: staff_number, course_taught, lecture_room
- Methods: open_room, switch_on_projector, switch_off_projector, close_room

4.1 Abstraction (cont'd)

- When writing the class this what we shall capture in the chosen language.
- In python for example the class will be declared as follows:
- class Professor():

```
def __init__(self, staff_number, course_taught, lecture_room):
        #the constructor of the class
        self.staff_number = staff_number
        self.course_taught = course_taught
        self.lecture_room = lecture_room
#methods will be declared after...this is a comment line
```

4.2 Encapsulation

- Encapsulation works hand in hand with abstraction. Abstraction deals with only the important details while encapsulation is about hiding details of the implementation of an object.
- Encapsulation is applied a lot in daily life; take for example the example of the classroom attendance application. As a professor I simply switch on the projector, hook it up to my laptop, and present my slides. However, the inner working of how this takes place does not bother me since the implementation details are hidden. I only see the result.
- In OOP encapsulation is implemented by hiding the state or the methods of a class from direct access; normally the term private and public are used to indicate this.
- In our previous example suppose I wish to hide the method open_room from direct access, and allow direct access to switch_on_projector. The two methods will be declared as follows:

```
private void open_room()  // the method returns nothing but is now private to only members of this class

public void switch_on_projector() // the method is available to all members outside of this class.
```

4.3 Modularity

- Modularity means breaking down the problem into many smaller manageable parts. This works very well where the program is big and complex.
- In such scenarios after doing the analysis and design thus determining all the classes, attributes and methods that are required the system analyst can work with the programming team to assign different classes and methods to be written by different members of the team. Practically sub-teams can write whole classes with one team writing the code that integrates the system.
- Supposing we were building an enterprise wide university system consisting of different departments say, faculty, administration and casuals. There will be teams to write code for the faculty group, another for administration, another for casuals, and one for system integration.

4.4 Hierarchy

- Hierarchy is the ordering or ranking of abstractions (Booch et al., 2007). Recall that in lesson one (see fig 3) it was determined that the lower you go in the hierarchy the lower the level of abstraction (hiding of details)?
- Hierarchy in the object model is expressed as an ISA relationship. Certain object oriented programming languages implement single hierarchy (a class can only inherit from one and only one other class) such as Java (at least not with classes) while others allow for multiple inheritance (a class can inherit from more than one class) such as C++ and Python.
- In java the "extends" word is used to represent inheritance. For example in our enterprise wide information system faculty is made up of tutorial fellows, lecturers, and graduate assistants. This means that a lecturer ISA faculty. In Java this would be written as:

```
lecturer extends faculty.
```

In Python it would be written as class lecturer (faculty). If we wished to create a class that will inherit from both faculty and administration in python (call it casual) we would declare it as follows:

```
class casual (faculty, administration):
```

Part 5

Example Implementations in popular languages

Introduction

- We have seen how object oriented principles are applied in object oriented programming languages.
- The only thing we haven't seen are complete programs written in an object oriented programming language.
- In this last section we tackle 2 problems in the most common object oriented programming languages. Hopefully the learner can appreciate the power of using the object oriented domain to solve problems.

5.1 Problem 1

- You wish to design a software that will be able to differentiate between molecules, proteins and aminoacids. As a bioinformatician you know that proteins are a type of molecule, while an aminoacid is associated with protein. The software should return the names of either molecule or protein; as for the aminoacid we are interested in displaying its code. All protein sequences must be displayed in upper case letters, while in molecular sequences it's only instances of 'U' that must be in upper case letters. Write the python program that will achieve this.

Solution to problem 1

- class Molecule:
def __init__(self, name):
if not name:
raise Exception('name must be set to something')
self.name = name
def getName(self):
return self.name
def getCapitalisedName(self):
name = self.getName()
return name.capitalize()

- class Protein(Molecule):
def __init__(self, name, sequence):
Molecule.__init__(self, name)
self.aminoAcids = []
for code in sequence:
aminoAcid = AminoAcid(code)
self.aminoAcids.append(aminoAcid)
def getAminoAcids(self):
return self.aminoAcids
def getSequence(self):
return [aminoAcid.code for aminoAcid in self.aminoAcids]
def getMass(self):
mass = 18.02

# N-terminus H and C-terminus OH
aminoAcids = self.getAminoAcids()
for aminoAcid in aminoAcids:
mass += aminoAcid.getMass()
return mass
class AminoAcid:

5.2 Problem 2

- You are required to write a Java program for checking account balances in a bank. When a customer deposits money into their account the new account balance is output on the screen. When a customer attempts to withdraw more than what they have in the account they are given a message that they are attempting to overdraw their account. Once a customer attempts to overdraw three times the system will advise them to visit the customer care representative to open a current account which allows for overdraft facility. The program should also allow the customer to query their balance. Write the program showing all necessary constructors that will implement these requirements

Solution to problem 2

- public class Account {
- // private data
- private double balance;
- // constructor
- public Account(double init_deposit) {
- balance = init_deposit;
- }
- public Account() {
- // no need to do anything, _balance will default to o
- }
- // deposit monies into account
- public void depositFunds(double amount) {
- balance = balance + amount;
- }
- // query the balance
- public double getBalance() {
- return balance;
- }
- // withdraw funds from the account
- public double withdrawFunds(double amo) {
- if(amount > balance) { // adjust the amoun
- amount = balance;
- }
- balance = balance - amount;
- return amount;
- }//modify this program for the extra requirements
- }

Summary

- The steps in the program development cycle are problem recognition, problem definition, program design, program coding, program testing, program implementation. Documentation should also be done at every stage.
- The key difference between object based languages and object oriented languages is that the former do not support inheritance and polymorphism.
- Programs can be classified as being either low level or high level. The first 2 generations were low level while all subsequent generations were high level.
- The object model consists of 4 major concepts: abstraction, encapsulation, modularity and hierarchy. For anything to be called object oriented it must have all 4.
- The 4 major concepts are implemented in every object oriented language, though some support multiple inheritance (like python) while others do not (like java).

References

Barbey, S.; Kempe, M.; Strohmeier, A. (1993). "Object-Oriented Programming with Ada 9X". *Draft Technical Report*. Swiss Federal Institute of Technology in Lausanne Software Engineering Laboratory. Retrieved 15 December 2013.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., & Houston, K. A. (2007). *Object-oriented analysis and design with applications* (Third). Addison-Wesley.

Hawi, R. (2013). *Introduction to programming*. *Class Notes*.

iVagus Services Pvt. Ltd. . (n.d.). *What is the difference between an object-oriented programming language and object-based programming language?* Online Technical Courses. Retrieved October 3, 2022, from https://www.programsbuzz.com/interview-question/what-difference-between-object-oriented-programming-language-and-object-based

Sebesta, R. W. (2022). *Concepts of programming languages*. Pearson.

Wegner, Peter (December 1987). Meyrowitz, Norman (ed.). "Dimensions of Object-Based Language Design" (PDF). OOPSLA'87 Conference Proceedings. ***22*** *(12): 168–182. doi:10.1145/38765.38823. ISBN 0897912470. S2CID 819420.*