1 PROGRAMMING PARADIGMS

1.1 DEVELOPMENT OF THE DIFFERENT PARADIGMS

1.1.1 limitations of the imperative paradigm

1.1.1.1 difficulty with solving certain types of problems

- The imperative paradigm was intended to solve **mathematical and arithmetical problems**
    - Many problems do not have real answers or have a variety of different answers
    - Many problems require solutions that are **not feasible** to develop in strict mathematical or arithmetical terms

1.1.1.2 the need to specify code for every individual process

- **Requires the developer to understand all details** of the problem and be able **to solve the problem completely**
    - Sometimes **the entire problem cannot be easily understood**
- **The entire solution needs to be developed** before it can operate
    - Testing is extremely difficult or not possible until all aspects of the solution are completed down to the lowest level

1.1.1.3 difficulty of coding for variability

- Many parts of code are repeatedly re-used for different parts of the solution
- Imperative programs do not completely support re-usable code
    - **Subprograms and copying/pasting code offers limited reusability** and will often require more work to make the code function properly

1.1.2 emerging technologies

- Machine languages (E.g. Assembler) were **specific to different types of machines**
    - Code cannot be used on different machines when using machine specific languages
- **Humans were required to think in machine language** (binary digits) to work with them
- At the time, this was considered a breakthrough and separated the human from the computer (which does all the hard work), however it is now seen as extremely primitive
- The exponential increase in the speed of technoLogical advancement has allowed the programmer to be **free of knowledge of machine code**
    - The final solution is **not as efficient as a machine code product**, however the **development is significantly easier and faster**
    - Programs are now able to be written in languages that
        - Are **human readable** (to a certain extent)
        - Are able to be used on **multiple types of machines**

1.1.3 simplifying the development and testing of some larger software projects

- **Speed of code generation**
    - Programming languages that increase the speed of code generation increase productivity, as **programmers can write more effective code in less time by choosing the most appropriate paradigm**
- Approach to **testing**
    - Programming languages that **reduce the time or effort required for testing are desirable to hasten development**
- Effect on **maintenance**

  - **Modular programming reduces maintenance time as errors are easier to locate and correct** in smaller modules
- **Efficiency of solution** once coded
  - Programming languages **vary in their level of efficiency depending on the computer processor and the level of modularity in the code**

1.1.4 strengths of different paradigms

PARADIGM	STRENGTHS	WEAKNESSES
IMPERATIVE	• Efficient • Close to the machine • Popular • Familiar	• Makes debugging harder • Abstraction is more limited • Order is crucial, which doesn't always suit itself to problems • Code is not ideal for re-use
LOGIC	• The system solves the problem, so the programming steps themselves are kept to a minimum • Proving the validity of a given program is simple	• Difficult to code complex programs
OBJECT-ORIENTATED	• Very easy to re-use code and extend it • High degree of modularity – easier to understand and maintain • **Inheritance** saves the re-writing of inherited attributes that are already defined in classes and subclasses	• Only **benefits problems with the need for re-usable code** and those and are **not sequentially driven**
FUNCTIONAL	• The high level of abstraction, especially when functions are used, **supresses many of the details of programming** and thus **removes the possibility of committing many classes of errors** • The lack of dependence on assignment operations allows programs to be evaluated in many different orders o This evaluation order independence makes Functional languages **good for programming massively parallel computers** • The absence of assignment operations makes the function-oriented programs much **more amenable to mathematical proof and analysis** than are imperative programs, because Functional programs possess referential transparency	• **Less efficient** • Problems involving many variables or a lot of sequential activity are **sometimes easier to handle imperatively or with object-oriented programming**

1.2 LOGIC PARADIGM

1.2.1 concepts

1.2.1.1 variables

- A variable in Logic programming is used to refer to **an unspecified individual** rather than a stored value of characters
- They can be used to substitute atoms, e.g.

```
likes(X,pizza)
```

- In this case, **"X" is the variable**, and the **fact** translates to **"X likes pizza"** where X can be **substituted** for anything
- **Querying** the following would result in "bob", as X can be substituted for "bob"

```
?-likes(X,pizza)
bob.
```

- Having multiple facts or rules using the same variable name requires the variable to stay **constant** for all facts or rules
  - E.g.

```
dog(X) <- mammal(X), furry(X), barks(X).
```

Will only work if all "X" are the same, e.g.

```
dog(fluffy) <- mammal(fluffy), furry(fluffy), barks(fluffy).
```

1.2.1.2 rules

- Rules assert something **if a specified condition is true**
- E.g.

```
dog(X) <- mammal(X), furry(X), barks(X).
```

- This means that if "X" apples to "mammal", "furry" and "barks", then "X" is a "dog"

1.2.1.3 facts

- Exactly what it appears to be
- E.g.

```
funny(bob).
```

Or

```
likes(bob,cake).
```

- In PROLOG, a database of facts and rules must be supplied to the program
  - From this database, queries can be performed
- The basic unit of PROLOG is the **predicate**, which is defined to be true
  - Predicate: **a head and a number of arguments**
  - In the example, "funny" or "likes" is the predicate
  - "bob" and "cake" are **atoms** (simple data items)
    - Atoms must commence with a lowercase character

1.2.1.4 heuristics

- A **rule of thumb** based on previous experience
- The criteria for **deciding which alternative course of action would be most effective** to achieve a goal

- Usually results in more than one possible solution

1.2.1.5 goals (queries)

- A query that can result in either being **fulfilled**, in which case the result is "Yes" or **not being fulfilled**, in which case the result is "No"
- Queries must begin with "?-"
- E.g. The following query must result in "Yes", because according to the "dog(X)" rule, all dogs are mammals, furry and bark. When the program is asked whether "fluffy" is a mammal, it knows that all dogs are mammals, "fluffy" is a dog, and therefore "fluffy" is a mammal and will result in "Yes". (Note: The entire code is not shown. The rules for a dogs to be mammals, furry and barking need to be mentioned before)

```
dog(fluffy)

?- mammal(fluffy)
```

1.2.1.6 inference engine

- The control mechanism that applies **knowledge that is contained in a knowledge base to resolve goals**, with the result being fulfilment or failure of the goal
    - Knowledge base: **A database containing all facts and rules**
- The inference engine is **the processing unit** of Logical programming
- Inference engines **apply knowledge gained from facts and rules** in a knowledge base to **reach conclusions about goals** in an organized, systematic manner

1.2.1.7 backward/forward chaining

- Backward chaining
    - **Start with a goal and prove it is true or false**
    - Requires the answer to be in the knowledge base

```
?-wizard(ron).

true.
```

- Forward chaining
    - **Provide a goal and find the values for which it is true**
    - Needs to follow a path through rules and facts to find answers

```
?-wizard(X).

X = ron;

X = hermione;

X = harry;
```

1.2.2 language syntax

1.2.2.1 variables

- Variables commence with a capital or underscore
- E.g.

```
"X"
```

```
"Count"

"_sum"
```

1.2.2.2 rules

- Like an IF statement which consists of facts which must be true for the rule to be true
- E.g. If A can eat B and B can eat C, then A can eat C

```
eat(A,C) :- eat(A,B), eat(B,C)
```

1.2.2.3 facts

- A fact expresses a relation which holds of objects
- Consists of a predicate and an atom or a variable
- E.g.

```
likes(sally,pizza).

boring(school).
```

1.2.3 appropriate use, such as:

1.2.3.1 pattern matching

- Often the **reasoning performed by the inference engine is pattern matching**
- E.g.

```
parent(Parent,Child).
```

  matches with

```
parent(joe,sue).
```

  where the program can match patterns with facts to find solutions
- Programming capabilities like this are simply **too difficult to program in other paradigms** or are extensively complicated to do so

1.2.3.2 AI

- **Pattern matching abilities are used to develop AI applications and research**
- Grammar and spelling check applications are an example of AI applications that use pattern matching
- AI makes **extensive use of the inference engine and heuristics** which are very difficult to program in other paradigms

1.2.3.3 expert systems

- An expert system is **used to perform functions that would normally be performed by a human expert** in that field
- An **expert system shell** is a software product that can be used to create an expert system. **Facts, rules and probabilities** (0 ←→ 1, where 0 is never and 1 is definite) of events occurring are **entered into a knowledge base**
  - Expert system shells **provide the framework for an expert system to which specialised knowledge must be added by knowledge engineers**
  - Reasoning is stored in the knowledge base which is then **interrogated** by the expert system shell

- Simulating the experience of a human expert is **difficult and data intensive**
- Expert systems **cannot learn new information**, as opposed to AI systems

1.3 Object Oriented paradigm

1.3.1 concepts

1.3.1.1 classes

- The definition of a category of objects
- Defines all the **common attributes and methods of the different objects that belong to it**

```
public class Point

{

   Data and methods of the class are declared here

}
```

1.3.1.2 objects

- Contain **attributes** and **methods**
- Is contained within the respective class and **inherits** its attributes and methods

1.3.1.3 attributes

- What an object knows and remembers
- **Can only be accessed and altered by the objects own methods**

```
private double x;

private double y;
```

1.3.1.4 methods/operations

- Methods are **housed within classes**
- The **executable part of a class**
- **Actions an object can do**
- Provides an **interface through which the object can communicate with other objects**

```
public Point(double xinit, double yinit)

{

   x = xinit;

   y = yinit;

}

public double xcoord ()

{
```

```
   return x;
}
```

1.3.1.5 variables and control structures

- Used to **define attributes and methods**
- Work similarly to those in the Imperative Paradigm
  - Variables such as integers, strings, Booleans, etc.
  - Control structures such as loop structures, binary and multiway selection, etc.
- Control Structures are statements which are used to control execution flow in the scripts
- They are sequences of scripting code which help to control complex procedure
- Control structures can define code which is only executed under certain conditions or repeated for a couple of times (iteration and selection)

1.3.1.6 abstraction

- The process of designing objects by breaking them down into component classes **allows more concentration on the details of the object**
- The hierarchy of classes is designed in such a way that **each class is reduced so as to include only its necessary attributes and methods**
- Abstraction allows us to **isolate parts of the problem** and consider its solution apart from the main problem
- **Encapsulation and inheritance greatly assist in the abstraction process**
  - They allow is to **put the overall problem aside while sub-problems are dealt with**

1.3.1.7 instantiation

- Creating an object (instance) based on a class
- **The object will inherit the attributes and methods of the class, as well as having its own specific attributes and methods**

1.3.1.8 inheritance

- The **ability of objects to take on the characteristics of their parent class or classes**
- **Encourages modularity and robust code**
- Development of new objects and child classes is **greatly simplified** using inheritance
- This **allows different classes of objects to be built hierarchically**, with the most general class on top and the more specific classes at the bottom of the hierarchy
  - A class does not have to define all of the methods itself but rather **it can reuse the methods from classes higher up in the hierarchy**

```
public class Circle extends Point
{
   private double r; /* radius of circle */
   public Circle (double xinit, double yinit, double rinit)
   {
      super(xinit, yinit);/* superclass */
      /* constructor */
```

```
        r = rinit;
    }
    public double area()
    {
        return Math.PI*r*r;
    }
}
```

1.3.1.9 polymorphism

- The **ability to appear in many forms**
- At runtime **a method can process data differently depending on circumstances**
    - The system chooses the precise method to execute based on the subclass of each particular object being processed
- Helps reduce the complexity of code
    - The **programmer does not need to include decisions** within the code to make decisions as **the system will decide which method to run during runtime**
    - Results in cleaner, more maintainable and faster execution of code because decisions are made by a built in part of the system rather than the programmers logic

```
Public class Rectangle extends point
{
    private double h; /* the height of rectangle */
    private double w; /* the width of rectangle */
    public Rectangle (double xinit, double yinit, double width, double height)
    {
        super (xinit, yinit); /* superclass */
        w = width;
        h = height;
    }

    public double area()
    {
        return w*h;
```

```
    }

}
```

1.3.1.10 encapsulation

- The process of **hiding an object's data and processes from its environment**
- **Only the object can alter its own data**
- Objects control their own private attributes using their **methods**
- No other object or class can alter another objects attributes directly, but rather must use the objects public methods
- Allows the creation of **robust and reusable classes of objects**
- Helps with testing and debugging as **the problem can only exist within the objects method**

1.3.2 language syntax

1.3.2.1 classes

- A group of objects sharing some **common characteristics and performing similar operations**
- The class declares all the common attributes and methods of the different objects that belong to it
- Parent classes **inherit** attributes and methods from superclasses

1.3.2.2 objects

- An individual thing that has its own unique methods and attributes
- **Inherits** methods from parent and superclasses

1.3.2.3 attributes

- Present in classes and objects
- Can be **inherited** from parent and superclasses
- Can be **encapsulated** within classes and objects
- Can be **overridden** through **polymorphism**

1.3.2.4 methods/operations

- Present in classes and objects
- Can be **inherited** from parent and superclasses
- Can be **overridden** through **polymorphism**

1.3.2.5 variables and control structures

- Variables need to be **declared** before use
  - E.g.

```
private int CurrentCount;
```

declares the variable CurrentCount as an integer

- After variables are declared, they can be made use of through control structures to obtain results

1.3.3 appropriate use, such as

1.3.3.1 computer games

- **Data Management**
  - Objects, structures, and advanced data types like linked lists and trees found in Object Oriented languages are often extremely helpful

  - These structures can be built manually, but it is much **faster and easier to use them when they are already part of the language**
- Objects
  - Objects in games can be thought of objects in programing languages, allowing them to have their own **features and attributes**
- **Event-driven**
  - Games are about events: the passage of time, user input, objects bonking into each other, robot zombie opossums falling out of the sky, etc.
  - Most OOP languages already have **robust support for event-handling**
- **Access to libraries**
  - Access to libraries for 3D rendering **simplifies the process for rendering GUI's and 3D objects**
  - Most Object Oriented languages **have a binding to at least the most basic libraries** (DirectX / Direct3D for Windows, SDL / OpenGL for everything else)
- **Ease of use**
  - Polymorphism
  - Abstraction
  - Inheritance
  - Instantiation
  - Encapsulation

1.3.3.2 web-based database applications

- An object database (also object-oriented database management system) is a **database management system in which information is represented in the form of objects as used in object-oriented programming**
- Use of Object Oriented languages **allow programmers to develop the product, store them as objects, and replicate or modify existing objects to make new objects (Polymorphism)**
- Using a DBMS that has been specifically designed to store data as objects **gives an advantage to those companies that are geared towards multimedia presentation or organizations that utilize computer-aided design (CAD)**

1.4 ISSUES WITH THE SELECTION OF AN APPROPRIATE PARADIGM

1.4.1 nature of the problem

- **Different problems require a different set of tools** to enable the production of efficient and reliable solutions
- Using a programming language more suited to a specific problem **increases productivity**

1.4.2 available resources

- Decisions about programming paradigms **will need to consider constraints**
  - Some programming languages will often require **more money** to produce a solution and to maintain it
  - They may also take **more time** to build a better solution
  - They may require **more programmers** to work on the solution

1.4.3 efficiency of solution once coded

- Efficiency is measured in **speed**
- **Imperative programs are always the most efficient as processors are designed for them**
  - Imperative programs are based on the **Von Neumann architecture**
  - **They have evolved along with developments in hardware technology**

1.4.4 programmer productivity

1.4.4.1 learning curve (training required)

- Imperative Languages
  - Relatively **low learning curve due to logical sequential manner**
- Logical Languages
  - **Easier to teach to someone new at programming** that someone who has already worked with another paradigm due to simplicity
- Object Oriented Languages
  - Object and class variables and control structures function much the same way as in Imperative and is therefore **easier to learn for someone already proficient in using Imperative languages**
  - Widely accepted and **used by a large portion of software developers**, and thus there are **more resources available to learn in this paradigm**

1.4.4.2 use of reusable modules

- **Reusability of modules may be required** for the development of a solution
- In such cases, it may be wise to use an Object Orientated program since they support **inheritance**

1.4.4.3 speed of code generation

- The speed at which code is generated is a traditional measure of programmer productivity
  - i.e. faster generation → more productivity
- **Languages that increase the speed of code generation increases the productivity of programmers**
- Using a language based on a more suitable paradigm also **increases the speed of code generation**
- Different **problems are suited to solutions using different paradigms**
- Choosing the most suitable paradigm can make the process of software design and code generation **more efficient and result in a more elegant and usable final solution**
- High-level programming languages have a shorter development time than a program written in a low-level language, given the programs were of same complexity
  - This is due to high-level languages ability to be **easily understood by humans** and **the ability for the programmer to ignore details such as memory locations and storage of variables**
    - The programmer can concentrate on the steps to **solve the problem**
- Objects and reusable functions lead solutions to be developed in a **rapid application development environment**, thus **speeding up project development time significantly and much more efficiently** than the limited low-level languages

1.4.4.4 approach to testing

- Different types of testing are faster in certain paradigms
  - Imperative Languages
    - Procedural and sequential structuring → **locating errors in the code easier**
  - Functional Languages
    - Employ **simple syntax → reduces the likelihood of syntax errors**
  - Logic Languages
    - Because they don't need complicated control structures → **contain less code to test**
  - Object Oriented Languages
    - Abstraction allows different functions and methods to be tested individually and allows the isolation of any problems → **easier testing**