ICT365 Ans 1

Library is a collection of method or functions which your code will call when it requires the function

JQuery and React

Software development <u>frameworks</u> (SDF)

- Provides basic programming structure for building an application. Your code will define the operations in the various required empty framework spots
- Collection of allowable programming languages, reusable software components and a set of software tools
- Allows clients to produce applications more quickly and efficiently through component reuse
- There are many different software development frameworks such as
  - Microsoft .NET Framework
  - Xamarin

.NET Framework was developed in year 2000, it an open source framework or platform (programming language + libraries) for making different types of application and is where it is not bounded by a specific programming language. The <u>programming language</u> running on top of the platform, once compiled by compiler it is converted to a *.NET assembly code.* Thus programming language independent because all programming languages will eventually compile the universal .NET assembly code. Contains

The **common language runtime** software is found in the .NET framework. It's Microsoft implementation of common language infrastructure (CLI). It will run the .NET assembly code (CIL) by using just-in-time compiler to compile the .NET assembly code on the fly to *native machine code*. The CLR will also be in charge of managing the memory of the running program.

Thus hardware/platform independent because as long as the hardware has CLR and Framework class libraries (FCL) it can run .NET assembly code.

Framework class library (FCL) contains the collection of

ICT365 Ans 2

ICT365 Ans 3

Every class is a child of System.Object (root) class i.e EVERY new class is System.Object [CLASS]

Overriding methods:

- Where a child class method overrides the implementation/definition of parent class method
- Intention of method MUST be clear-
    - Base class method TO BE overridden declare keyword ***virtual***
    - Child/derived class method TO override base class declare keyword ***override***
- Allows you to define similar operation in different ways for different OBJECT TYPES i.e. parameter must be same

```
class Base {
    public virtual int foo() {} }
class Derived {
    public override int foo() {}}
```

Overloading methods:

- Multiple methods in SAME class with SAME method name but different implementation
- Allows you to define similar operation in different ways for different DATA TYPES i.e. parameter can be different

```
int foo(string[] bar);
int foo(int bar1, float a);
```

Abstract classes: (Think interface class but with data members + Implementation)

Is a relationship

- Not fully complete and is meant to be used as a base class
  - [ASK FOR INHERITANCE BASE CLASS? YES]
- Those that provide an interface (set of methods) and fields/data members with SOME implementation
- Provide common interface among related classes or hierarchy of classes
- Contains at least one ABSTRACT method w/o implementation

- Class can inherit only ONE abstract class
- Can't Instantiate or create abstract class object. Very useful

`NOTE: Considered inheritance so inheritance rules apply`

Structure of C#

```
public abstract class A { //Needed if abstract method
      public abstract void M();
      …
      protected void AddTax() {…};
      protected virtual int convertToCurrency() {..}
}
~~public~~ class B: A {
      public override void M() { … };
      …
      protected override int convertToCurrency() {..}
}
```

- Public Abstract/Virtual → We want client to access
- Protected Abstract/Virtual → We don't want client to access but want program to access

Interface class: (Think English- Iam[IntefaceName] OR Ican[interface] or IHave[InterfaceName])

Implements or Can Do relationship

- Defines a contract that classes must implement. Must not inheritance rules apply
- Contain ZERO data members/fields and NO method implementation
- Only consist of an interface (set of methods prototypes) that other classes must implement

- Class can implement MULTIPLE interfaces
- Can’t Instantiate or create abstract or interface class object. Very useful

```
interface I[IntefaceName] {

      void Paint();

      void SetItems(string[] items);

}

public class EditBox: I[IntefaceName] {

      public void Paint() { … };

      public SetItems(string[] items) { … };

      …

}
```

Abstract classes vs Interface:

- Class can inherit ONE abstract class VS Class can implement MULTIPLE interfaces
- Class contains some implementation VS Interface class contains no implementation
- Class can contain fields/data members VS Interface class contains ZERO fields/data members
- Class can inherit abstract class VS Interface class a struct or class can implement interface class

- Abstract class is inheritance Vs Interface is contract (strict inheritance rule doesn’t apply)

Three ways to pass listbox:

1. Pass ListBox to method

```
public void DisplayAllItems(ListBox newListBox) {
      foreach(var element in tempDatabase) {
            newListBox.Items.Add(element);
      }
}
```

Disadvantage- Not generic enough so breaks abstraction

2. Pass interface to method

```
public void DisplayAllItems(IList newListBox) {
      foreach(var element in tempDatabase) {
            newListBox.Add(element);
      }
}
```

Advantage- More generic and promotes abstraction by hiding specific implementation such as a listbox. Instead it is Ilist which is more general and applies to many other objects thus reduces code duplication

3. Pass objects to method

```
public List<Int> DisplayAllItems() {
      List<Int> TempList
      foreach(var element in tempDatabase) {
            TempList.Add(element);
      }
      Return TempList;
```

```
}
```

Disadvantage- Overhead

ICT365 Ans 4

ADO

Active Data Objects (ADO) .NET looks at creating a database connection, sending query request for the database, and defining data models for its tables. For creating .NET database software. Included in the .NET Framework and provides access to a wide variety of different database and sources. Having a database makes the application more scalable as it allows for long term storage.

SQL server, Oracle, Azure, and Microsoft Access

ADO .NET Object Model has many different objects.

Database will establish a connection with the **connection object** which is in charge of talking with the physical database.

Data adapter object passes query results to the **dataset** and acts as bridge between data set and data source. The data adaptor controls when the connection can be closed with data. Clients will interact with cache or short term storage rather than physical database. This is used when the query requires the modification of data in database.

Both data reader object and command object work together to retrieve read only data from database in order when query executes. Not used when the query requires the modification of data in database

Data set object memory copy of a subset of the database. Consist of set of tables and data relations

Entity framework is open source object relational mapping for ADO .NET. Allows object oriented programming logic applied to database logic

ASP .NET

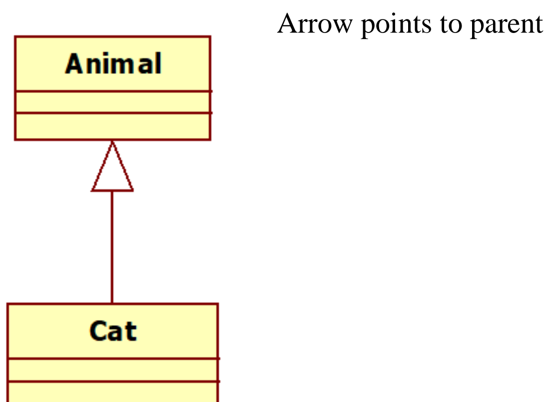
ICT365 Ans 5

Class relationships-

Specialisation: ~~(think a square is a rectangle or oval is a circle. BAD because rectangle has width but square doesn't square has 1 side length variable for all. Also rectangle has setLength and setWidth while square has setSideLength)~~

- Class *extends* another class (adds information)
- When a class (Cat) has all the characteristics of another class (Animal) but then adds information it is called specialisation
- Derive a child class (cat) from parent class (animal) and add to the child (cat)
- If you don't follow you break L in SOLID principle

- Rule: You can substitute objects of parents (animal) for → objects of child (cat) w/o effecting program behaviour
- For: The child class (cat) uses *every* member variable declared in the parent class (animal)
- For: The child class (cat) requires *every* method defined in the parent class (animal)
- For: you can relate them in English using x (Parent) 'is a' y (Child) in a behavioural sense
  - Animal is a cat
  - Transport vehicle is a car
  - NOT money is a wallet (makes no sense)

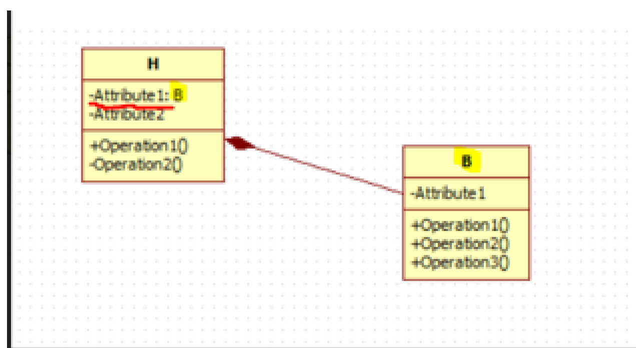


Arrow points to parent

Composition: (logical design. Logical makes sense)

- When the object (parent class/wallet) disappears all the parts (child class/money) disappears. Child object does not have their own life cycle. Depends on parents object lifecycle
- Include multiplicity unless relationship 1 to 1 (most likely has multiplicity)

- For: you can relate them in English using x (Parent class) 'has a' y (Child class)
    - Wallet has money
    - Apartment has room
    - Class has student (although not a composition but aggregation)
    - Company has location
- For: The parent class* has member variables (instance variables) that is the datatype child

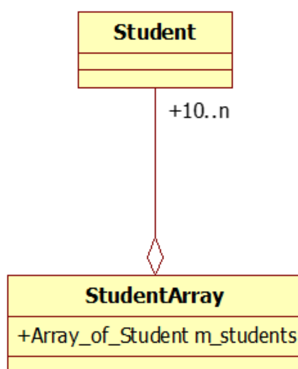


Arrow points to parent (bigger class)

Aggregation: (still a container)

- When the <u>object</u> (parent class/Class) <u>disappears</u> all the parts (child class/student) <u>DOESN'T</u> disappears

- For: you can relate them in English using x (Parent class) 'has a' y (Child class)
    - Class has student (although not a composition but aggregation)
- For: one could say that one class *refers* to another class. I.e student → studentList or unit → unit coordinator Not studentlist →unit
- For: The parent class* has member variables (instance variables) that is the datatype child



Arrow points to parent (bigger class)

```
Class PartClass{
//instance variables
//instance methods
}
class Whole{
PartClass* partclass;
}
```

(inside private/public demo doesn't show it. Also not sure about the *)

Realisation:

- Refers to abstract, template, and interface classes
- One class implementers the functionality belonging to another class

Dependency: bad practice since dependencies create high coupling. Where one class depends on another class. Aim to reduce

- Changes to the supplier affects the consumer since the consumer depends on the supplies but not vice versa
- One object invokes another object in order to complete a task (method)
- The method for one class (consumer) has the other class in its parameter

Design patterns

Design patterns are common problems that programmers have found an efficient solution for. It reduces the need for a developer to spend time on finding a solution to the standard problem, through code reuse. Design patterns provide a rough or general template thus allow developers to implement the pattern to solve a standard problem in our context. They are also language independent meaning they can be applied to all programming languages

Creational patterns deal with object instantiation and hiding creating object with new

Singleton pattern is a creational pattern where it deals with object instantiation. This pattern ensures that a class will only be able to be instantiated to an object once during runtime. The single object can be referred to throughout the system providing a safe global access point. Eliminates the need to constantly instantiate a frequently used object

Singleton pattern EG

```
public sealed class EventDatabase //Sealed tries to prevent accidental multiple instances of class. Thread safe

    {

        private static readonly EventDatabase instance = new EventDatabase();

        private Dictionary<string, IAmEvent> DatabaseForEvents = new Dictionary<string, IAmEvent>();


        private EventDatabase()

        {


        }

        public static EventDatabase Instance

        {

            get

            {

                return instance;

            }

        }
```

Factory method is a creational pattern where it deals with object instantiation. The subclasses are responsible for determining which class to instantiate and thus return as object. Also instead all the object instantiation is kept inside the factory method. Often used when a class doesn't know which subclass object is needed to be instantiated often decision is made at run time. Stop the client having to remember the class name for each new objects. Also encapsulates the creating of the object thus allowing controlled access to when an object can be instantiated.

Factory method pattern EG

```
public abstract class IAmAccoutType

    {

        public string Balance { get; set; }

    }


    public class SavingsAccount : IAmAccoutType

    {

        public SavingsAccount()

        {

            Balance = "10000 Rs";

        }

    }


    public class CheckingAccount : IAmAccoutType

    {

        public CheckingAccount()

        {

            Balance = "20000 Rs";

        }

    }


    public interface IAccountFactory

    {

        AccoutType GetAccoutType(string accountName);

    }


    public class AccountFactory : IAccountFactory

    {

        public AccoutType GetAccoutType(string accountName)

        {

            if (accountName.Equals("SAVINGS")

            {

                return new SavingsAccount();

            }

        }

    }
```

Abstract factory pattern

Abstract factory pattern is a creational pattern where it deals with object instantiation. Creates and return an encapsulated family of related objects ~~through an interface~~. Contains multiple factories where each factory creates their own specific object. An interface is in charge of creating an encapsulated family of related objects.

> https://stackoverflow.com/questions/5739611/what-are-the-differences-between-abstract-factory-and-factory-design-patterns

https://refactoring.guru/design-patterns/abstract-factory

Builder design pattern

Builder design pattern is for the creation of a complex object step by step. Where a factory method constructs simple object. They often involve the object to be constructed is composed of many other objects. Thus, this pattern helps promote abstraction since where the internal implementation the construction of many other objects is hidden to the client.

Structural patterns deal with composition and relations in terms of how composing and relate different objects together

Adapter design pattern translates an interface of a class to the client expected interface. Essentially promotes abstraction by allowing the client to use his preferred interface by creating an adaptor class which handles communication between the incompatible interfaces. The adaptor class calls the relevant methods of the incompatible interface class. Also allows different classes with interfaces to essentially work together through implementing one adaptor interface and an adaptor class for all the classes we want working together

Adapter design pattern EG

```
public interface IAmSinger {
        void Sing();
    }


   Class AdapterBird : IAmSinger {
       private readonly Bird currentBird;
       public Adapter(Bird newBird) {
               currentBird = newBird;
       }
       Public void Sing() {
               currentBird.Chirp();
       }
   }
   Class Bird {
       Public Bird() {
       }
       Public void Chirp() {


       }
    }
```

Façade design pattern is way to essentially promote abstraction and hide the internal implementation. A single class will be responsible for providing an interface for a group of complex sub systems or classes. Way to centralise the dependencies of the sub system or classes and improve method interface so it's more readable to client. This involves wrapping the method with the client assumed method name. Furthermore, the single class providing the interface for the group of complex sub systems or classes is abstracted so that each of the subsystems can be isolated and tested so issues can be easier to identify.

Façade design pattern EG

```
Class SubSystemA {
      Public SubSystemA() {
      }
      …
}
Class SubSystemB {
      Public SubSystemA() {
      }
      …
}
Public class Façade {
      SubSytemA a = new SubsystemA();
      SubsystemB b = new SubsystemB();
}
```

Decorator pattern

Behavioural patterns

Template method pattern is a behavioural pattern where it identifies the common patterns in relation to communication between different objects. This pattern contains a base class that has the template method which defines the algorithmic steps in order. The subclasses of the base class have the ability redefine the algorithm steps and implement their own implementation without changing the algorithm method structure. Reduces code duplication since sub classes inherit methods from base class.

Template method pattern EG

```
Class One : TemplatePattern  {
      Public override void StepOne() {
            …
      }
      …
}
Class Two : TemplatePattern {
      Protected override void StepTwo() {
            …
      }
      …
}
Public abstract class TemplatePattern{
      Public void TemplateMethod() {
            StepOne();
            StepTwo();
            StepThree();
      }
      private void StepThree() {
            …
      }
      public abstract void StepOne();

      protected virtual void StepTwo() {
            …
      }



}
```

ICT365 Ans 6

Design principals (SOLID)

Cohesion relates to how closely the contents of the class are with each other

Single responsibility principle

Single responsibility principle is where a class and method should have one responsibility and one reason to change.

Not abiding by single responsibility makes testing the class more difficult since there are more test cases that need to be written and retested after implementing changes.

Also the impact of any changes is likely to be greater compared to code that is written with single responsibility in mind because code that abides by single responsibility are isolated and abstracted to separate classes. Thus, changes to the implementation would only affect the abstracted or isolated class when single responsibility is followed.

Another impact of not following single responsibility is that reusability is decreased. For example, when a class wants to use a single method found another class and the method is buried within a bunch of other methods then reusability is decreased. The reason why is because the client is exposed to more unused and irrelevant methods thus violating abstraction and also the class is not encapsulated as tight as possible.

The violation of this principle leads to code that is harder to read and maintain. Because as the class has more responsibility then needed the complexity will grow thus it makes it harder to read

```
Class Dog {
      …
}
Class Printer {
      public void PrintDog(Dog newDog) {
            …
      }
}
```

In this example the Dog class should not be responsible for printing the dog. The reason being is that client should be able to specify the format he wants the object printed. Also there could be different ways the object can be printed such as on a file or to the console thus it is difficult to justify that dog object should be responsible for the different ways in which it can be printed. ~~Thus, any changes to the printing method violates single responsibility~~. So the dog should be responsible for its content but another class should be responsible for printing the content.

Open-closed principle

Open closed principle is where a method, class and modules should be open for extension but closed for modification. This means new features can be added without changing existing code. Also existing code can be refactored but the behaviour should not be modified.

Violating open closed principle makes extending existing code more difficult. For example below

```
Class Dog : IAnimal {
        …
}
Class Cat : IAnimal {
    …
}
    Class Printer {
        public void Print(IAnimal newAnimal) {
            if(newAnimal is Dog) {
                …
            } else if(newAnimal is Cat) {
                …
            }
        }
    }
```

The above code violates the open closed principle. If more IAnimal were to be added, then the existing if statement code would need to be modified to support this. And if the code base is very large then it would be too difficult for the developer to know where in the existing code needs to be modified to support a IAnimal. Also if that section of code was duplicated everywhere then the developer would need to apply the change everywhere as well. Thus this makes extending class difficult

Secondly violating open closed principle may make maintaining the code more difficult. When code is not closed for modification and ends up being modified. The code would need to be retested. Furthermore, this opens up the possibility of additional bugs being introduced since we are modifying more sections of the code which could cascade down

Liskov substitution principle

Interface segregation principle

Interface segregation principle is the premise that classes and clients that implement interface will use all methods of the interface. This principle favours many small interface cohesive over large interfaces. Each smaller interface will have different specific roles and responsibility they fulfil.

The violation of ISP will result in decreased code reuse. For instance

```
public interface IAmAnimal

{

        public void Eat();

        public void Sleep();

}
```

While a dog class can implement the IAMAnimal interface. A future plant class cannot implement the IAMAnimal interface because plants can't sleep. However, if we were to only have two interfaces with interface one being ICanEat and interface two being ICanSleep then the plant class can implement the ICanSleep class. The dog class can implement both the two interfaces. Therefore, code reuse is promoted

The violation of ISP will also result in having the client implement methods they don't intend to use. There is a chance an inexperienced client implemented the empty interface methods with a throw exception. This makes the error message hard to track down and solve.

Thirdly violating integration segregation principle leads to worse maintainability. When it comes to maintainability code needs to be testable and readable. When an interface contains unused methods it will confuse the clients and make it existing unit testing not reusable. Also when bug appear there is a possibility the client would try to use empty methods which would further confuse the client. Also there will be more code implemented thus greater chance of there being more bugs

Linq

LINQ provides a standardised way of querying any LINQ compatible data source and thus allows us to retrieve the data from data source. Provides a simple way to retrieve sorted, filtered, ordered and grouped from data source. In addition, a single LINQ statements is reusable across many different data sources such as an array or list. Benefit it makes code more readable through

Query expression is operations you want to perform on the data source

General LINQ

```
//Get Data source
…
List<string> dataSourceName = new List<string>(){ "Bob", "Marley"};

//Create query

var queryResults = from item in dataSourceName
                   where item > 5 OR item.Contains(5)  //CONDITION OR FILTER
                   select item;

//Execute Query
foreach(var tempElement in queryResults) {
      …
}
```

LINQ TO XML (Read/Write to XML)

Deals with retrieving and writing data from XML data source

Designed for storing and transporting data that will be same throughout different machines so configuration files

Alternative to an implementing a full blown database when a light weight solution like XML is more appropriate. (Light weight)

Portable

General LINQ To XML

Reading from XML

```
//Get Data source
…
XDocument dataSourceName = XDocument.Load("MyFile.xml");
//Create query

var queryResults = from item in dataSourceName.Descendants("Section")
                   where (datatype) item.//CONDITION OR FILTER
                   select (datatype) item.Attribute("insideSection");

//Execute Query
foreach(var tempElement in queryResults) {
      …
}
```

Writing to XML

```
//Get Data source
…
XDocument dataSourceName = XDocument.Load("MyFile.xml");

//Create query

var queryResult = new XElement("Event",
                new XElement("eventid", newTweet.EventID),
                new XElement("tweet", newTweet.TweetText));

//Execute Query
dataSourceName.Root.Add(queryResult);
dataSourceName.Save("MyFile.xml");
```

ICT365 Ans 8

Refactoring

Refactoring seeks to fix and improve existing code design in a systematic way without changing the behaviour or features. Making sure refactoring doesn't change the behaviour or features can be checked through prepared unit tests. ~~The process of refactoring promotes easier code readability~~

~~and maintainability by writing clean code.~~ The process of refactoring aims to make existing code more readable and maintainable by ensuring that existing code is clean code. This ensure that in the future, additional features or behaviour can be implemented fast.

> For example, when the developer wants to figure out why a new feature they added isn't working, all they would need to look at is the code they have written. Because the likelihood the issue is caused by the previous code is significantly reduced when the previous code has been refactored. Thus, refactoring ultimately narrows the scope of where the problem lies.

Refactoring also promotes code that is more testable. Because without testable code the process of making sure refactored code doesn't change the existing behaviour of the code is more difficult. Not doing any code refactoring often will result in accumulating technical debt. The process of refactoring should ideally occur throughout the software development cycle and when you identify code that smells or not clean code.

Clean code is-

Code smell is the signs or characteristic with existing code design that often indicate that refactoring should occur-

Duplicate code EG

Duplicated code is where lines of code that perform the same task appear in multiple places. The issue with duplicate code is that it makes the program harder to maintain. For instance, required changes to duplicate code will not apply to other instances of duplicate code meaning the change needs to be manually applied to all instances of duplicate code.

```
public int CalcAverage(List<int> currentList) {

        int total = 0;

        foreach(int num in currentList) {

                total = total + num;

        }

        total/currentList.Count;

}


public void PrintTotal(List<int> currentList) {

        int total = 0;

        foreach(int num in currentList) {

                total = total + num;

        }

        …

}
```

Extract method refactoring EG

Solution refactoring method/Technique **is Extract method refactoring** is where duplicate code is grouped moved to separate new method and call the new method from where the old code was.

```
public int CalcAverage(List<int> currentList) {

        FindTotal(currentList)/currentList.Count;
```

```
}

public void PrintTotal(List<int> currentList) {

          FindTotal(currentList);

          …

}

public int FindTotal(List<int> currentList) {

          int total = 0;

          foreach(int num in currentList) {

                    total = total + num;

          }

          return total;

}
```

Alternative code with different interface EG

Alternative classes with different interfaces is where classes have methods that perform the same behaviour but have different method names. This often leads to unnecessarily duplicate code and poor abstraction.

```
public interface IAmClass {

          ...

}

public class FootyClass : IAmClass {

          ...

          private int CalcFootyClassSize() {

                    ...

          }

}

public class SoccerClass : IAmClass{

          ...

          private int NumberOfSoccerClassSize {

                    ...

          }

}
```

Rename (method) refactoring EG

Solution refactoring method/Technique **is Rename (method) refactoring** where the method name will be change to better describe the behaviour of the method. Thus providing a solution to this code that smells

```
public interface IAmClass {
```

```
        public int CalcClassSize();
}
public class FootyClass : IAmClass {
        ...
        public int CalcClassSize() {
                ...
        }
}
public class SoccerClass : IAmClass{
        ...
        public int CalcClassSize {
                ...
        }
}
```

Speculative generality EG

Speculative generality is where additional parameters, methods, or class are implemented because we predict the client might request that feature in the future. This often leads to more difficult to maintain code since it needs to be tested and more unnecessary complexity.

```
public float GetNetWorth(string currency) {
        ...
        return netWorth;
}
```

Remove parameter refactoring

Solution refactoring method/Technique **is Remove parameter refactoring** which is where the unused parameter is removed. In this case, we speculated a feature that the client might want is get net worth but in his desired currency. But the implementation has not been completed since the feature is not requested. So remove it

```
public float GetNetWorth() {
        ...
        return netWorth;
}
```

Comments EG

Comments is where comments are included inside a method body that sole purpose is to badly written code and explain something. This often leads to the code being more difficult to read by developers.

```
class Check {
        ...
```

```
        public int amount;
}
class Human {

        ...
        public void WriteCheck(int newAmount) {
                Check tempNewCheck = new Check();
                Check.amount = newAmount //COMMENT: Can't be negative amount. Negative amount breaks code
                ...
        }
}
```

Encapsulate field refactoring

Solution refactoring method/Technique is **Encapsulate field refactoring** which is where the data member is provided with controlled access. In this case, the comments that smells are able to be removed through implementing this technique.

```
class Check {
        ...
        private int amount;
        Public int Amount {
                …
                Set {
                        if(value > 0) {
                                amount = value;
                        }
                }
        }
}
class Human {

        ...
        public void WriteCheck(int newAmount) {
                Check tempNewCheck = new Check();
                Check.Amount = newAmount
                ...
        }
}
```

Long method EG

Long method is where a single method contains too much lines of code. This often indicates the method has too much responsibility thus breaking single responsibility principle. The issue is that often the responsibilities inside the long method are useful to other methods but since it's all in one long method we can't reuse it.

```
public void PrintDetails() {
```

```
        foreach(var element in houseList) {

                Dog tempDog = element.Dog;

                Console.Write("Dog is: {0}" + tempDog.Age);

                Console.Write("Dog name: {0}" + tempDog.Name);

                Person tempPerson = element.Person;

                Console.Write("Peson is: {0} and {1}" + tempPerson.Age + tempPerson.Name);

                Cat tempCat = element.Cat;

                ...

                Pigeon tempPigeon = element.Pigeon;

                ...

                Goat tempGoat = element.Goat;

                ...

        }

}
```

Extract method refactoring EG

Solution refactoring method/Technique **is Extract method refactoring** is where duplicate code is grouped moved to separate new method and call the new method from where the old code was.

```
public void PrintDetails() {

        foreach(var element in houseList) {

                PrintDog(element.Dog);

                PrintPerson(element.Person);
```

```
                    PrintCat(element.Cat);
                    PrintPigeon(element.Pigeon);
                    Goat tempGoat = element.Goat;
          }
}
private void PrintDog(Dog newDog) {
          Console.Write("Dog is: {0}" + tempDog.Age);
          Console.Write("Dog name: {0}" + tempDog.Name);
}
private void PrintPerson(Person newPerson) {
          Console.Write("Peson is: {0} and {1}" + tempPerson.Age + tempPerson.Name);
}
private void PrintCat(Cat newCat) {
          ...
}
private void PrintPigeon(Pigeon newPigeon) {
          ...
}
```

Class too large

Feature envy

Switch statements

Data clumps

Lazy class

Refactoring Techniques/Types

Refactoring Techniques/Types addresses code smells

Extract method refactoring is where group code and move to separate new method and call the method from where the old code was.

Rename (method) refactoring is where the method name is changed to better describe the behaviour of the method

Remove parameter refactoring is where an unused parameter is removed from the method's parameter. This unused parameter is not used on the body of the method

Encapsulate field refactoring is where a data member does not have controlled access. Also future implementation of controlled access would break the client code. So use this technique to ensure data member is encapsulated and provided with controlled access

Extract interface refactoring is where two or more classes share common behaviour so extract to an interface.

Reorder parameter refactoring

Abstraction makes it more generalizable which allows it to be overridden and re-implemented

ICT365 Ans 9

Non-generic collections

Are (System.Collections) they store System.object type. Given non-generic types store System.object the process to convert items that are added to the collection to System.Object is called boxing and unboxing. Boxing and unboxing allows any datatype item to be added to the collection. One disadvantage of this is inferior performance compared to generic collection. The reason is because the need to box and unbox items every time during different operations is an expensive process. It slows runtime execution speed

ArrayList [] tempArray = new ArrayList();

Queue tempQueue = new Queue();

Stack tempStack = new Stack();

Hashtable

Generic collections

Generic collections (System.Collections.Generic) ensure safety because the programmer will specify the type the collection will hold. Without this safety net, we can't guarantee all items in the collection will be of the same type. Thus, error would appear during processing of the data if unexpected data type is contained in the collection. One advantage is it provides compile time error rather than run time error. This means upon compilation the programmer will be easily able to identify the error and fix it before it is deployed. Another advantage is that we don't have to create custom overloads and provide the overloaded functions unlike with non-generic. ~~A third advantage with generic is performance~~

List<T>

Queue<T>

Dictionary<T>

Exceptions

- Are Unusual errors during runtime not as a result of programming errors
- User specific excepts first then general exceptions at end
- Finally block always gets executed even after catch
- Only provide exception if we predict they will occur and we know what to when it occurs. Prefer application to crash over bad data

Disadvantages of non-exceptions

- May be difficult to outline normal process to reader since it may be cluttered with if statements
- The lack of libraries to implement common exceptions so much take the time to code
- Some errors are difficult to check and identify

Advantages of exceptions

- Makes code easier to read due to how try catch are organised
- Clearly outlines the normal process to the readers. Any unusual errors are organised at bottom away from normal process
- Many of the exceptions or issues programs face have already been identified and been provided a solution. Thus, reduce the need to excess code

The exception method-

```
try
{
System.IO.StreamReader inFile = System.IO.File.StreamReader(@"Hello.txt");
string test = inFile.ReadlLine();
test = test.ToUpper();
inFile.Close();

System.IO.StreamWriter outFile = System.IO.File.StreamWriter(@"NewHello.txt");
outFile.WriteLine(test);
outFile.Close();


}
Catch(FileNotFoundException e)
{
```

```
    …
}
Catch(DivideByZeroException e)
{


}
Catch(IOException e)
{
    e.Message
}
Catch(FormatException e)
{
    e.Message
}
```

http://vb.net-informations.com/framework/what_is_net_framework.htm

ICT365 Ans 10

Platform independence describes how application should be able run on different architectures. The .NET program compiled will be in .NET assembly. Where platforms would require only FCL and CLI to execute the .NET program thus it is platform independent.

Language interoperability allows the use of different programming languages in one application. Either through using a .NET language in another .NET program by calling the libraries

Both are promoted through below

Common Language Infrastructure (CLI) is a standard that ensures that applications programmed in multiple high level languages are compatible on different computer platforms or architecture.

The **common language runtime (CLR)** software is found in the .NET framework. It's Microsoft implementation of common language infrastructure (CLI). It will run the .NET assembly code (CIL) by using just-in-time compiler to compile the .NET assembly code on the fly to *native machine code*. The CLR will also be in charge of managing the memory of the running program.

Thus hardware/platform independent because as long as the hardware has CLR and Framework class libraries (FCL) it can run .NET assembly code.

.Net assembly

.Net assembly is a logical deployable unit of code produced when the .NET language is compiled. NET assembly is composed of manifest, metadata and common intermediate language

Common intermediate language (CIL) is the code that is produced when any .NET language is compiled. It is stored in a .NET assembly. This language provides the standard low level language that is compatible with different computer platforms or architecture regardless of the specific high level .NET programming language that it was produced with. The CIL can be translated by the Just-in-time compiler to machine code to execute.

Metadata is the information about the content inside .NET assembly. Provides information such as declarations, description of the assembly references of types and security permission.

Manifest is the information about the .NET assembly itself. Provides information such as assembly version and name.

Framework class library (FCL) contains the collection of components, classes, data that is provided by the .NET framework and available to be used in a .NET application.

Common type system (CTS) is outlines the standards of how a common type should behave and be defined. This standard ensures once the .NET gets compiled any high level .NET language can be mapped to an equivalent data type. Thus this promotes *language interoperability*. It facilitates a common set of libraries to be compatible with all .NET language.

XAML

ASP .NET

ASP .net This technology is primarily used for creating web applications. Makes it easier to work with the backend and communicate with web services.

Mobile web application approach targets the browser NOT the mobile operating system. So the application would be accessed using the browser. This allows the application to be cross platform across mobile and even desktop operating system.

> Positives is the developer would only need to maintain and extend the application for a single codebase. Inexpensive to develop given developer can use just HTML and CSS. Also compatible with nearly all devices mobile and desktop as long as they have web browsers.
>
> Disadvantages is that since it's not fully native the full features of a specific platform are not utilised. Given the app is developer for the browser the UI will not fully feel like a native application

ICT365 Ans 12

SILO approach (native) was used where an application would be developed for a specific mobile operating system such as IOS, or Android. So for IOS objective C and XCode was used while Java and Eclispe for android. There would be a separate codebase for each platform.

> Positives is this approach allows developer to utilise the full features of a specific platform to the full extent since the application is built natively.
>
> Negatives of this approach is that an organisation would need to maintain and extend the same application over different codebases, which would be expensive task in terms of cost and resource utilisation. Also not being able to reuse the code or specific/important libraries over different platforms

Mobile web application approach targets the browser NOT the mobile operating system. So the application would be accessed using the browser. This allows the application to be cross platform across mobile and even desktop operating system.

> Positives is the developer would only need to maintain and extend the application for a single codebase. Inexpensive to develop given developer can use just HTML and CSS. Also compatible with nearly all devices mobile and desktop as long as they have web browsers.

Disadvantages is that since it’s not fully native the full features of a specific platform are not utilised. Given the app is developer for the browser the UI will not fully feel like a native application

Cross platform is about creating a single mobile application for all mobile operating ~~system and all desktop operating system~~

XAMARIN APPROACH is for native cross platform mobile application with share backend code for all platforms ~~development and desktop application development~~. Previously in order to develop mobile application a SILO approach. This approach is designed to be compatible for cross mobile operating system ~~and desktop operating system~~. Helps create a single base code for the ~~User Interface~~ backend for all mobile operating system. Provides the benefits of creating native mobile application because the UI utilise each platforms features. Allows for the distribution of application on all mobile OS stores thus increasing the potential market share of application.

Positives is that it provides code reuse for different platforms in terms of the backend. Allows the developer to utilise the full features of a specific platform to the full extent since UI has to be made for each platforms. Compared to SILO approach easier to maintain and extend since the backend codebase is shared among different platforms

Disadvantage is that the developer would still need to maintain and extend the same application UI over different platform codebases. Once a new operating system update is released needs to wait for XAMARIN to add and make the updated operating system available as a target. Tied to XAMARIN as a vendor and difficult to switch. Thus, one issue could be XAMARIN may increase the cost of their services thereby affecting the viability of an application for smaller companies.

XAMARIN FORMS provides developers the ability to shared backend code and shared UI code for all the mobile platforms.

WINDOWS FORMS

ASP NET FORMS