**Class Diagrams Updated 2024**

1. Introduction

Candidates taking Information Technology must be able to read and create class diagrams. Class diagrams will be used in IT Paper 1 and Paper 2, SBA assessments, and the PAT in varying ways. Standardising a language-independent class diagram structure is vital to representing a class design consistently.

Please read this document together with "**Algorithms Updated 2024**".

2. Class Diagram Structure

Class diagrams consist of three rows in a single-column table with the following sections:

|  |
| --- |
| **Class Name** |
| **Fields** |
| **Methods** |

**Note**: All class diagrams have these three sections, even if the section is empty. A class diagram must have a class name but might not have any fields. This scenario is most likely to occur if the class is a subclass whose methods override the methods of a superclass.

2.1 Access Modifiers

All fields and methods must be prefixed with one of three access modifiers in a class diagram:

* **-** for private (not accessible outside the class)
* **+** for public (accessible outside the class)
* **#** for protected (accessible to subclasses and classes in the same package)

2.2 Types

In a class diagram, fields and typed methods must use one of the following data types:

* string
* integer
* real
* character
* boolean
* [ ] (array)
* Date
* Time
* DateTime
* User-Defined class

A user-defined class consists of multiple fields. For example, the **Fruit** class (**see section 6**) consists of fields to describe a **Fruit** object, such as description, qty, costPrice. Each field has an associated data type such as string, integer, real, etc.

**Note:** All primitive and string types start with a lowercase letter. The **Date**, **Time**, and **DateTime** types start with capital letters indicating they are objects.

3. Class Name

The class name must appear in the first row of the class diagram. If a name is not provided in the assessment, an appropriate meaningful name must be chosen. It must always start with a capital letter. If additional words are included, they should be capitalised.

4. Fields

In the class diagram, list the fields in the order they were represented in the assessment using the provided identifier names. The field name cannot contain spaces and must be in camel case (first letter in lowercase with the subsequent first letter of words capitalised). Each field must have an **access modifier** and **type**.

4.1 Non-Static Fields

Non-static (instance) fields and methods belong to an OBJECT. Each object will have its own copy of the fields.

4.1.1 General Format of Non-Static Field:

<access modifier> fieldName : type

4.1.2 Examples of Non-Static Fields:

- name : string

+ num : integer

- initial : character

+ valid : boolean

# invoiceCost : real

- subjectArr [ ] : string

- dob : Date

- arrival : DateTime

+ raceTime : Time

Most non-static fields are either private or protected (**see section 6.1.1**). A public non-static field violates information-hiding rules. A non-static field should be declared as private and an accessor method provided only if necessary.

The private field is still 'visible' if it has an accessor and mutator method.

4.2 Static Fields

Static fields and methods belong to a CLASS. A static field is not created for each object but shared among all class instances.

Static fields are named and typed similarly to non-static fields but must be underlined.

4.2.1 General Format of Static Field

<access modifier> fieldName : type

4.2.2 Examples of Static Fields

- totalQty : integer

- totalCost : real

- numFriends : integer

+ combinedMessage : string

- vat : real

- discount : real

Static fields are typically incremented or changed in the object's constructor method and do not have an associated constructor parameter. These fields can be altered in other class methods.

In the above examples, the variable **vat** and **discount** fields belong to the class but might need to be changed so they are not declared as constant.

4.2.2.1 Static Fields Incremented/Changed in a Constructor Method

The constructor for the **Fruit** class (refer to class diagram in **section 6**) has four private non-static fields description, qty, costPrice and sellingPrice. The constructor method has three parameters to assign values to the first three private non-static fields. The private non-static sellingPrice field of each **Fruit** item is based on the costPrice and MARK\_UP values, hence sellingPrice is not in the parameter list.

The private static fields, totalQty and totalCost, are calculated using the private non-static costPrice and qty fields.

public Fruit(String d, int q, double c)

{

description = d;

qty = q;

costPrice = c;

**sellingPrice = costPrice + costPrice \* MARK\_UP / 100.0;**

**totalQty = totalQty + qty;**

**totalCost = totalCost + costPrice \* qty;**

}

4.2.2.2 Static Fields Incremented/Changed in Other Methods

Static fields can be altered in methods other than the constructor method. In the **Fruit** class (class diagram **section 6**), the public void method sell reduces the **Fruit** object's qty field and reduces the static totalQty field by the value of the parameter q (e.g., when the number of apples is reduced the fruit total must also be reduced). The static totalSales field is also increased using the parameter q multiplied by the **Fruit** object's sellingPrice. When a particular fruit sells, the totalSales will increase.

public void sell(int q)

{

qty = qty - q;

**totalQty = totalQty - q;**

**totalSales = totalSales + q \* sellingPrice;**

}

A public static field violates information-hiding rules. Define a static field as private and provide an accessor method if necessary. Providing a mutator method if the class controls the static field's value does not protect the field.

4.3 Constant Fields

**Constants** whose value cannot change are written in uppercase. Separate words in a constant identifier name with the underscore character, such as MARK\_UP.

In most cases, the constant is also static, as each object does not need a copy of the constant.

4.3.1 General Format of a Constant Field

<access modifier> FIELDNAME = value : type

4.3.2 Examples of a Constant Fields

+ MARK\_UP = 75 : integer

- MIN\_VALUE = 150.0 : real

*//Ensure the underscore is visible above the underline*

**Note**: The underscore in a constant field name must be visible and not covered by the underline (see how to format an underlined underscore in the **Appendix**).

If a **constant** is intended to be used by other classes, it should be declared **public** and **static** so it can be accessed from another class using the format ClassName.CONSTANT. A public constant will NOT need an accessor or mutator method. A public constant does not violate information-hiding rules as it cannot be changed.

A private constant can improve a class's readability. If the constant is required outside the class, make it **public** and **static**. A mutator is not needed for a constant, as its value never changes.

4.4 Array Fields

Add square brackets after the field name to indicate an array. If the number of array elements is relevant, include the array size in brackets after the type. Including the word **array** or **arr** in the identifier name is good practice to indicate that it is an array type.

4.4.1 General Format of an Array Field

<access modifier> fieldName [ ] : type

<access modifier> fieldName [ ] : type [number of elements]

4.4.2 Examples of Array Fields

- strArray [ ] : string [100]

- realArray [ ] : real [50]

- someArray [ ] : character *//The number of elements is not included*

- weekDaysArr [ ] ← {"Monday","Tuesday","Wednesday",

"Thursday","Friday"}

- hourlyRatesArr [ ] ← {100,200,300,400}

The arrays weekDaysArr and hourlyRatesArr are array literals. An array literal initialises the array with values and does not specify the array size. The array type does not need to be included. The left arrow is used to indicate the array elements are assigned values. Since the values of an array literal can be changed, the array should be declared as **private**. This allows for controlled changes to the array's values using mutator methods.

5. Methods

In the class diagram, list the methods in the order they were represented in the assessment, maintaining the same method and parameter names. Methods will have:

* an access modifier (+, - or #),
* a method name,
* possibly a parameter list, and
* a type (if values are retuned).

Method names are always followed by a set of parentheses, whether there are parameters or not. See **sections 11.1** and **11.2** in the **Algorithms Updated 2024** notes regarding the format of method headers. The only distinction lies in class diagrams using +,-, or # to represent access modifiers and omitting the word "method."

5.1 Parameters

All parameters follow the same format as field declarations discussed in **section 4** without access modifiers. Use the same data types provided in **section 2.2**. Commas separate multiple parameters.

5.1.1 General Form of a Parameter

parameterName : type

5.1.2 Examples of Parameter Lists

+ sell (q : integer)

+ addValues(num1 : real, num2 : real, num3 : real) : real

*//Typed method returns a real value*

+ changeFlag(flag : integer)

5.1.3 Array Parameters

Arrays can be sent to a method in a parameter list. As shown in **section 4.4**, indicate an array with square brackets [ ] after the parameter name. The number of elements in the array is not indicated in the array parameter list.

+ upDateStudentSubjects(subjArr [ ] : string)

+ displayBothLists(friendList1 [ ] : Friend,

friendList2 [ ] : Friend)

**Note**: The format of an array parameter in a **method header** is the same as the array declaration in **4.4.1** without the access modifier.

5.2 Void Methods (Procedures in Delphi)

Void methods do not return a value. The method header in the class diagram will not include a colon and a return type at the method's end (after the method's parenthesis).

5.2.1 General Form of a Void Method

<access modifier> methodName (parameter list) *//No ':' and return type after ')'*

5.2.2 Examples of Void Methods

+ sell (q : integer)

+ sort ( )

- doubleFieldValue ( )

+ setDob (inDob : Date)

+ findIntersection (arr [ ] : real)

A **helper** method (e.g., doubleFieldValue()) is inaccessible outside the class. This method will be used by other methods in the same class and must be declared private.

5.2.3 Void Methods that Operate on Static Fields

Void methods operating on static fields must be static and underlined in a class diagram.

+ setTotalTrees (num : integer)

+ setVat (newVat : integer) *//Vat is static but not constant as Vat can   
 //change*

- reduceDiscountBy10Perc () *//No parameter is needed as the discount is   
//reduced by 10% in the method; helper method*

5.3 Typed Methods (Functions in Delphi)

Typed methods return a value indicated by a colon and the value's data type at the method's end (after the method's parenthesis).

5.3.1 General Form of a Typed Method

<access modifier> methodName (parameter list) : returnType

5.3.2 Examples of Typed Methods

+ getName () : string

- calculateAverage () : real

- search (name : string) : Friend *//The method returns a Friend object*

*//The class name starts with a capital letter*

5.3.3 Typed Methods that Operate on Static Fields

Typed methods operating on static fields must be static and underlined in a class diagram.

+ getTotalQty () : integer

+ getVat () : real

+ addVatToCost () : real

5.3.4 Array Return Types

Methods that return an array must include square brackets after the return type.

+ findStudentSubjectsAbove80() : real [ ]

+ findAllFriends(age : integer) : Friend [ ]

5.4 Constructor Method

A constructor method assigns values to the private fields with default values (default constructor) or parameter values (parameterised constructor). The constructor method is named **Constructor**, with an uppercase 'C'. The constructor method's parameters (if any) must be in the supplied order provided in the assessment and named and typed accordingly. Constructor methods have no return type (not even a 'void' in Java).

5.4.1 Constructor Method Examples

+ Constructor (d : string, q : integer, c: real) *//Parameterised constructor*

+ Constructor () *//Default constructor*

5.5 Accessor and Mutator Methods

Accessor and mutator methods retrieve and assign values to an object's private fields. Since these fields are private, the accessor and mutator methods must be marked with the public access modifier, allowing them to be called from other classes. In the class diagram, they should be denoted with a '+' symbol.

5.5.1 Accessor Methods

An accessor (getter) method retrieves a value from an object's field. The names of accessor methods must be prefixed with 'get' unless the method returns a boolean, in which case it is prefixed with 'is'. Accessor methods are always typed with a colon and return type added at the end of the method header.

5.5.2 Examples

+ getDescription() : string

+ getQty() : integer

+ isValid () : boolean *//Accessor methods for boolean fields   
 //should start with* ***is****.*

+ getTotalTrees () : integer *//Underline static fields accessor methods*

These methods do not require parameters, as the field's value is available in the class.

A private static field (e.g. totalTrees) will require a static accessor method if it is needed outside the class (e.g. getTotalTrees() method above). This method must be underlined.

5.5.3 Mutator Methods

A mutator (setter) method assigns values to an object's fields. The names of mutator methods must be prefixed with **'set'**. Mutator methods always require one or more parameters to assign the new value to the field. These methods are always void and do not include a colon and a return type at the method's end.

5.5.4 Examples

+ setPrice(p : real)

+ setSellingPrice(sp : real)

+ setPizzaOrder (pO [ ] : Pizza)

+ setPostion (newX : integer, newY : integer)

+ setTotalTrees (num : integer) *//Underline the static fields accessor methods*

Generally, a private static field should NOT have a mutator method solely for modifying its value. These fields are typically calculated within the class and should not be directly changeable by class users. Private static fields, like totalQty in the sell method (**section 4.2.2.2**), can be modified within methods of the class they belong to, such as the Fruit class.

5.6 toString method

The **toString** method concatenates the object's field values into a single string. If the field is an array, the **toString** method combines its values into a single string using a loop.

5.6.1 toString Method Example

+ toString() : string

6. Complete Class Diagram Examples

|  |
| --- |
| **Fruit** |
| - description : string  - qty : integer  - costPrice : real  - sellingPrice : real  - totalQty : integer  - totalCost : real  - totalSales : real  + MARK\_UP = 75 : integer *//Leave a line after the static field to clearly   //show it is underlined.*  *//Ensure the underscore is visible.* |
| + Constructor(d : string, q : integer, c: real)  + getDescription() : string  + getQty() : integer  + getPrice() : real  + getSellingPrice() : real  + setDescription(d : string)  + setPrice(p : real)  + sell(q : integer)  + getTotalQty() : integer *//Underline static accessor methods*  + getTotalCost() : real  + getTotalSales() : real  + toString() : string |

6.1 Inheritance

In inheritance, there is an **IS-A** relationship between objects. The **subclass** object **IS-A** **superclass** object since the subclass contains all the superclass fields and methods along with its own fields and methods. For example, a German Shephard **IS-A** Dog and a Rectangle **IS-A** Shape.

6.1.1 Protected Fields

Unlike a private field, accessible only from within a class, a **protected** field (indicated with a #) can be accessed from any subclass or class within the same package.

6.1.2 Generic Layout

|  |
| --- |
| Parent class |
| Parent class fields |
| Parent class methods |

|  |
| --- |
| Child class |
| Child class fields |
| Child class methods |

**NB:** The arrowhead is shown as clear, not filled.

6.1.3 Example

6.1.3.1 Players and Provincial Players

The following example determines a soccer player's commitment rating based on the number of school practices they have attended. Some players are part of the provincial team and attend both provincial and school practices. The **Player** class stores the soccer player's name, date of birth, position on the soccer field, and shirt size. The protected field numPracticesAttended records the number of school soccer practices. The **ProvPlayer** class includes the field numProvPractices to store the number of provincial practices attended by a provincial player.

A **ProvPlayer** object **IS-A** **Player** object because the **Player** is the parent object, and **ProvPlayer** **IS-A** child object that inherits the fields and methods from the **Player** superclass.

|  |
| --- |
| **Player** |
| - name: string  - dob : Date  - position : string  - shirtSize : string  # numPracticesAttended : integer  + PRACTICES\_PER\_TERM : integer = 20  + COMMITMENT\_CUT\_OFF\_PERCENTAGE : real = 80.0 |
| + Constructor(inN : string,  inD : Date,  inP : string,  inSS : string,  inNP : integer)  + getName() : string  + getAgeCategory() : character  + getCommitmentRating() : real  + getKitCost() : real  + toString() : string |

|  |
| --- |
| **ProvPlayer** |
| - numProvPractices: integer |
| + Constructor(inN : string,  inD : Date,  inP : string,  inSS : string,  inNP : integer,  inNumPP : integer)  + getCommitmentRating() : real  + toString() : string |

The field numPracticesAttended is set as protected and visible in both classes. Both classes have an **overridden** method named getCommitmentRating(), and both methods use the numPracticesAttended value in calculating a player's commitment rating. The getCommitmentRating() method in the subclass differs from the superclass because the private subclass field numProvPractices is considered in the commitment rating calculation. Soccer players who have attended provincial practices must be given credit along with their school attendance.

The **toString** method is always overridden in inherited classes.

6.2 Composition

Composition is a programming construct in which an object's field type is another object. Each object's field type must be previously defined in its own class definition. Create diagrams for each class.

In composition, a **HAS-A** relationship exists between objects where one class, the **Composition** class, contains (**HAS-A**) a field of another class, the **Component** class. In the example below, fieldX in the **Composed** class is a **Component** object.

6.2.1 Generic Layout

|  |  |  |
| --- | --- | --- |
| Composed class |  | **Component** class |
| Composed class fields  - fieldX : **Component** |  | Component class fields |
| Composed class methods |  | Component class methods |

In composition, the relationship between classes is shown with a solid line and an arrow pointing from the **Composed** class to the **Component** class[[1]](#footnote-1). The **Component** class forms part of the **Composed** class.

**Note**: The composition class diagrams can be written next to each other or one below the other.

6.2.2 Examples

6.2.2.1 Players and Kit

Continuing with the **Player** example (**6.1.3.1**), the **Player** class has a string field called shirtSize. Instead of only storing the size of a shirt, the field could be replaced with one that stores more precise details of a player's clothing requirements. These details will be stored in a **Kit** object. Replace the shirtSize field with a **Kit** field called clothing to record the kit details (shirt size, shorts size, shorts colour and boot size).

The **Player** object **HAS-A** relationship with a **Kit** object. A **Kit** object must be instantiated BEFORE the **Player** object is instantiated, as the **Kit** object is a component of the **Player** object.

The **Player** class includes a **Kit** field. A player has the same basic fields (name, date of birth and position) with a new **Kit** field called clothing. As previously mentioned, instantiate a component **Kit** object before a composite **Player** object.

|  |
| --- |
| **Kit** |
| - shirtSize: character  - shortsSize : character  - shortsColour : string  - bootSize: integer |
| + Constructor(inSS : character,  inShS : character,  inShC : string,   inBS : integer)  + getShirtSize() : character  + setBootSize(inBS : integer)  + toString() : string |

|  |
| --- |
| **Player** |
| - name: string  - dob : Date  - position : string  - clothing : **Kit** |
| + Constructor(inN : string,  inD : Date,  inP : string,   inC : **Kit**)  + getName() : string  + getClothing() : **Kit**  + toString() : string |

6.2.2.2 Pizza Orders

This example uses three classes to show a more complex composition example. A pizza restaurant offers a delivery service to customers who order pizzas. The orders are delivered by agents. An order can contain one or many pizzas. The relationship between the **Agents**, **Pizzas** and **Orders** classes is shown in the class diagrams.

An **Agent** object has primary fields for the name, gender and phone number of the person delivering the pizzas.

|  |
| --- |
| **Agent** |
| - agentName : string  - agentGender : character  - agentPhoneNumber : string |
| + Constructor(inAN : string,  inAG : character,  inAPN: string )  + getAgentName() : string  + setAgentName (inAN : string)  + toString() : string |

A **Pizza** object has fields describing the base type, three ingredients, and up to 5 toppings. The static constant field TOPPING\_COST records the cost for each topping, and the static constant field PIZZA\_BASE\_COST records the cost of the pizza base.

|  |
| --- |
| **Pizza** |
| - pizzaBase : string  - basicIngredientsArr [ ]: string [3]  - extraToppingsArr [ ] : string [5]  + TOPPING\_COST = 2.50 : real  + PIZZA\_BASE\_COST = 10.50 : real |
| + Constructor(inPB : string,  inBI [ ] : string,  inET [ ] : string)  + getPizzaBase() : string  + getBasicIngredientsArr() : string [ ]  + getExtraToppingsArr() : string [ ]  + calculatePrice() : real  + toString() : string |

The **Order** class has fields to store the unique order number, the customer's name, an array of **Pizza** objects and the assigned **Agent**.

The **Order** **HAS-A** delivery **Agent** and the **Order HAS-A** (in fact many) **Pizzas**.

|  |
| --- |
| **Order** |
| - orderNumber : integer  - customerName : string  - pizzaOrder [ ] : **Pizza** [10]  - deliveryAgent : **Agent** |
| + Constructor (inON : integer,  inCN : string,  inPO [ ] : **Pizza**,  inDA : **Agent**)  + getOrderNumber() : integer  + getDeliveryAgent() : **Agent**  + setPizzaOrder (inPO [ ]: **Pizza**)  + setDeliveryAgent (inDA : **Agent**)  + toString() : string |

The relationship between the three class diagrams is shown with a solid line and an arrow pointing from the composite **Order** class to the **Pizza** and **Agent** component classes.

|  |
| --- |
| **Order** |
| - orderNumber : integer  - customerName : string  - pizzaOrder [ ] : **Pizza** [10]  - deliveryAgent : **Agent** |
| + Constructor (inON : integer,  inCN : string,  inPO [ ] : **Pizza**,  inDA : **Agent**)  + getOrderNumber() : integer  + getDeliveryAgent() : **Agent**  + setPizzaOrder (inPO [ ]: **Pizza**)  + setDeliveryAgent (inDA : **Agent**)  + toString() : string |

|  |
| --- |
| **Agent** |
| - agentName : string  - agentGender : character  - agentPhoneNumber : string |
| + Constructor(inAN : string,  inAG : character,  inAPN: string )  + getAgentName() : string  + setAgentName (inAN : string)  + toString() : string |

|  |
| --- |
| **Pizza** |
| - pizzaBase : string  - basicIngredientsArr [ ]: string [3]  - extraToppingsArr [ ] : string [5]  + TOPPING\_COST = 2.50 : real  + PIZZA\_BASE\_COST = 10.50 : real |
| + Constructor(inPB : string,  inBI [ ] : string,  inET [ ] : string)  + getPizzaBase() : string  + getBasicIngredientsArr() : string [ ]  + getExtraToppingsArr() : string [ ]  + calculatePrice() : real  + toString() : string |

6.3 Composition and Inheritance Example

In the Player and Provincial Player example (**6.1.3.1**), the **Player** class had a component field called **Kit** (**6.2.2.1**) and the **ProvPlayer** inherited from the **Player** class. All the soccer players are stored in an array field called playerArr in a **PlayerArray** class. The **Player** class is a component of the composed **PlayerArray** class.

The constructor in the **PlayerArray** class reads the soccer players' data from a text file and instantiates the playerArr field. The size field records the number of elements in the array. The **PlayerArray** class has a void method to sort the playerArr and a typed method to search for a **Player** object given the player's name.

|  |
| --- |
| **PlayerArray** |
| - playerArr [ ] : **Player**  - size : integer |
| + Constructor(fileName : string)  + sortPlayers()  + search(inName : string) : **Player**  + toString() : string |

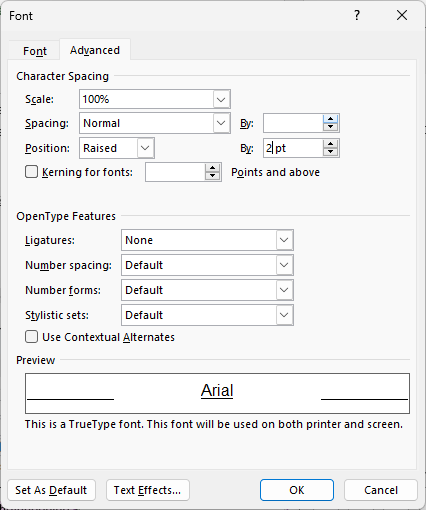
|  |
| --- |
| **Kit** |
| - shirtSize: character  - shortsSize : character  - shortsColour : string  - bootSize: integer |
| + Constructor(inSS : character,  inShS : character,  inShC : string,   inbS : integer)  + getShirtSize() : character  + setBootSize(inBS : integer)  + toString() : string |

|  |
| --- |
| **Player** |
| - name: string  - dob : Date  - position : string  - clothing : **Kit** |
| + Constructor(inN : string,  inD : Date,  inP : string,   inC : **Kit**)  + getName() : string  + getClothing() : **Kit**  + toString() : string |

|  |
| --- |
| **ProvPlayer** |
| - numProvPractices: integer |
| + Constructor(inN : string,  inD : Date,  inP : string,   inNP : integer,  inSS : string,  inNumPP : integer)  + getCommitmentRating() : real  + toString() : string |

Appendix

Ensure the underscore in a constant field name is not covered when underlined, giving the appearance of an identifier with spaces. In MS Word, on the **Home** tab, click the diagonal arrow in the bottom left corner of the **Font** group, and in the **Advanced** tab, select **Position: Raised By:** and **2 pt** to separate the underscore from the underline.



1. In UML, this type of composition is known as one-way association. Other composition types (such as as aggregation) are beyond the scope of the syllabus. [↑](#footnote-ref-1)