CSC4311: SOFTWARE ENGINEERING
COUSE OUTLINE

- Software Design: Object-Oriented Analysis & Design, Software architecture, Design Patterns;
- Design for Re-use;
- Using APIS: API programming Class browsers and related tools;
- Component based computing;
- Software Tools and Environment: Requirements Analysis and Design
- Modelling Tools, Testing tools, Tool integration mechanism

1.0 Object Oriented Analysis and Design

Object-Oriented Analysis and Design (OOAD) is a Software Engineering Style that uses "real-world" objects to design computer programs. In essence, it is a way of breaking down complex systems into smaller, more manageable "objects" that work together to achieve the desired functionality.

The following are the two phases of Object-Oriented Analysis and Design:

i. **Object-Oriented Analysis (OOA):** This phase involves studying the problem domain, identifying objects, understanding their relationships, and capturing requirements in an object-oriented model. This model uses visual representations like UML diagrams for clear communication.

ii. **Object-Oriented Design (OOD):** Based on the analysis, this phase translates the conceptual model into a concrete design, considering implementation constraints like software and hardware platforms. This involves defining class structures, interfaces, and algorithms for efficient implementation.

1.1 Why OOAD?

- To manage complexity and improve productivity in software development;
- To reduce the conceptual gap from the design space to the implementation space, because the program is conceptualized as a collection of objects interacting to achieve a common goal.

1.2 Benefits of OOAD

- **Modularity:** By isolating functionalities within objects, code becomes more modular and reusable, promoting maintainability and extendibility.
- **Understandability**: Object-oriented models closely resemble real-world entities, making the system easier to visualize and understand for developers and stakeholders.
- **Reusability:** Well-designed classes and objects can be reused across different projects, saving development time and effort.

OOAD has paved way for several matured software engineering practices such as:

- **Object Relational Mapping (ORM):** ORMs act as a bridge between object-oriented programming languages and relational databases, which store data in tables. By mapping

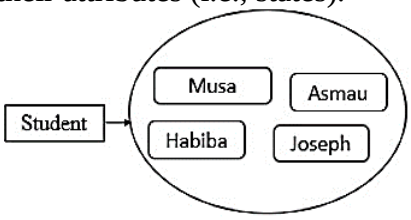
objects to database tables and vice versa, ORMs simplify data access and manipulation, reducing the need for writing complex SQL queries.

- **Design patterns**: design patterns are reusable solutions to common software design problems, documented and catalogued for easy reference. They provide proven approaches to handle specific scenarios, promoting code quality, consistency, and maintainability across projects.
- **Model Driven Development (MDD)**: MDD emphasizes creating and using models throughout the development process, from requirements to implementation. These models capture the system's functionality, structure, and behavior, serving as a central artifact for communication, analysis, and code generation. MDD tools can generate code based on these models, potentially reducing development time and errors.

1.3 Objects and classes

1.3.1 Object

Literally, an object is anything that can be seen and touched, such as a table, a phone, a chair,etc. However, the concept of an object in software design goes beyond this literal definition. In software design, an object is a real-world entity, either tangible or intangible, that has states and behaviors. Tangible objects are objects that can be seen physically. For example, in the context of tertiary institutions, there can be student objects such as Musa, Joseph, Asmau, and Habiba.On the other hand, intangible objects are objects that cannot be seen but can be described logically. For example, courses (modules) taken by students at tertiary institutions such as *CSC 211*, *BIO 112*, and *CSC 321* are intangible objects. Even the semester registration of the courses is logically an object. An object has two main members, namely: state and behavior. To understand what a state and a behavior mean, we need to associate each of these objects mentioned above with a particular group based on their attributes (i.e., states).



State	Behavior
Name	Can pay fees
Admission No.	Can sit for exams
Gender	Can graduate
Course	Can register for semester

Figure 1: Representing Student as a class with its possible objects

In Figure 1, we can see that the objects "Musa, Joseph, Asmau, and Habiba" are grouped together under a class called *Student* because they have common *attributes*.

1.3.2 Class

A class is a general specification of the attributes and behavior of a set of objects. It can also be seen as a template that defines the properties/states and behaviors that are common to set of objects. Each object is said to be an *instance* of a class. It shares the behavior of the class but has specific values for the attributes. In the above example, Studentis a class, while Musa, Joseph, Asmau, and Habiba are the specific instances (objects) of theStudent class. Another example of objects described by a common class because they share common attributes and behaviors is Courseas demonstrated in Figure 2.

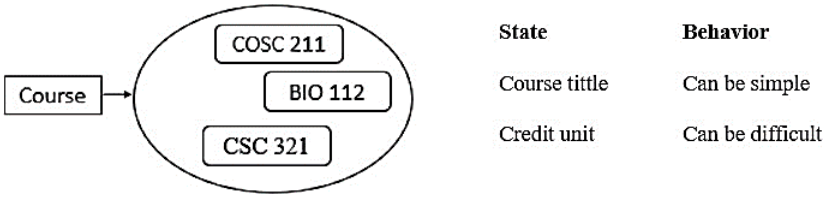


Figure 2: Representing course as a class with its possible objects

2.0 Methods of OO Analysis

Object-oriented programming requires some design effort before the actual coding starts. Because finding out the objects and deciding on the task allocation among them is not a programming job. As such, there is need for systematic method finding answers to questions like:

- What will the program achieve when it is completed?
- What classes and objects will be needed?
- What will be the responsibilities of the classes and objects?
- What will be the information content and functionalities of each object?
- How will the objects handle their responsibilities?
- How will the objects communicate?
-

2.1 Noun/ Noun Phrase Analysis

Noun/ Noun Phrase Analysis is a technique used during the initial stages of system analysis to identify and extract potential objects and classes from the problem domain. The analysis begins by examining the problem statement, requirements documents, or other sources of information related to the system under consideration. Analysts look for nouns and noun phrases, which often represent entities, concepts, or things in the problem domain. Nouns and noun phrases that represent tangible entities or abstract concepts within the problem domain are potential candidates for classes in the object-oriented model. These could include things like "customer," "order," "product," "transaction," etc. The identified classes, relationships, and attributes undergo refinement and validation through discussions with stakeholders, domain experts, and other relevant parties to ensure that the model accurately reflects the problem domain and meets the requirements of the system.

Example 1

In the online supermarket ordering process, a **customer** navigates to the supermarket's website or mobile app. The customer browses through the available **product** categories and selects the **items** they wish to purchase by adding them to their virtual shopping **cart**. Upon completing their selection, the customer proceeds to the checkout section. The system prompts the customer to log in to their account or create a new account if they haven't already done so. Once logged in, the customer provides delivery details such as address and preferred delivery time slot. The system

calculates the total cost including any applicable taxes or delivery fees and presents it to the customer for confirmation. The customer confirms the order and proceeds to **payment**. The system securely processes the payment using the customer's preferred payment method (e.g., credit card, PayPal). Upon successful payment, the system sends a confirmation email or notification to the customer, containing the details of the order including items purchased, total cost, delivery address, and scheduled delivery time. The supermarket's **staff** receives the order and prepares it for delivery within the specified time slot.

Engineers may use noun/noun phrase analysis to identify objects and classes and then develop a Domain model. Notice that the key nouns in the statement are in bold face. Table 1 presents four potential classes from the domain of supermarkets and their attributes.

Table 1: Classes of objects from the supermarket domain

Product	**Customer**	**Staff**	**Invoice**
Product Name ManufacturerDescription Price	Name Address	Name Staff Id No DepartmentRank	Invoice Id Customer IdCreated by Amount

Example 2:
Imagine you want to automate a trivial problem of manual booking of an appointment with a **dentist**. In the manual process, a **patient** calls the dental clinic. The **receptionist** receives the call and guides the patient on the available **slots**. The patient selects one of the available slots, which the receptionist will reserve and informs the patient of the appointed **schedule**. For brevity, we assume that the patient has already registered with the clinic.

Exercise 1
Identify the classes and their attributes in the Dental Booking System

2.2 Use Case

A use case model describes a software product from the user's perspective. It is a representation of the system's high-level functionalities. Ivar Jacobson popularized use case diagrams in the late '80s and early '90s. It is essential to analyze the whole system to identify every single functionality and then transformed them into the use cases to be used in the use case diagram.

2.2.1 Elements of a Use case

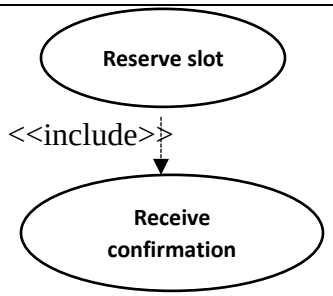
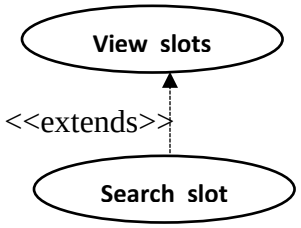
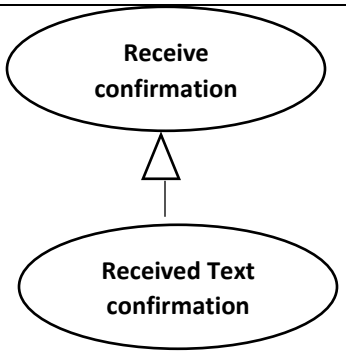
Table 2: Elements of Use Case

Element	**Description**	**Notation**
Actor	An actor represents an entity that performs certain roles in a given system	
Use Case	Representation of a distinct functionality in a system	

System boundary	A system boundary defines the scope of what a system will be. The system boundary is shown as a rectangle spanning all the use cases in the system	

2.2.2 Relationships in Use Cases

Table 3: Relationships between Use Cases

Relationship	Description	Notation
Include	When a use case uses the functionality of another use case	
Extend	In an *extend* relationship between two use cases, the child use case adds to the existing functionality	
Generalizations	A *generalization* relationship is also a parent-child relationship between use cases; It is used to represent the inheritance between use cases A child use case specializes some functionality it has already inherited from the parent use case	

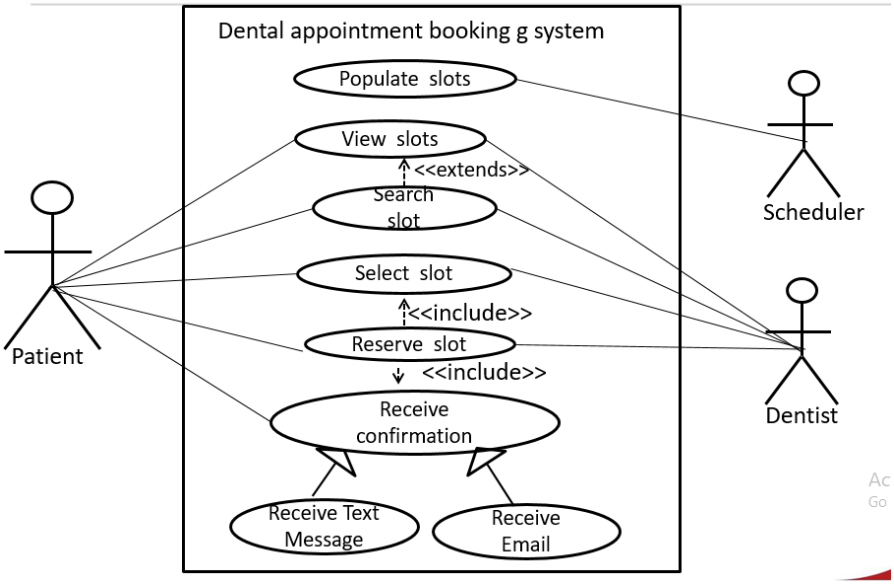


Figure 3: Use Case diagram for Dental Appointment Booking System

2.2.3 Use Case Description

Internally, a use case includes a whole sequence of steps to complete a process

- **use case description**
  - a textual description of processing details within a use case
- **scenarios**
  - unique sets of internal activities within use cases
- **precondition**
  - a condition that must be true before a use case begins
- **postcondition**
  - what must be true upon the successful completion of a use case

2.2.4 Use Case Description Template

- **Use case name**: a unique name for this use case
- **Actors**: all those actors who use this use case
- **Brief Description**: a one or two sentences description of the results of the use case
- **Triggering Event**: the business event that initiates or triggers this use case
- **Scenario**: the instance of the use case being documented
- **Exceptions:** alternative paths that would make the use case unsuccessful
- **Related use case**: any related use case
- **Precondition:** any required prior conditions
- **Post condition:** the state of the system at the successful completion of the use case

Examples of use case description: Populate slots

Use case name	Populate slots
Actors	Scheduler
Brief Description	Populating slots to make available for booking

Triggering Event	New shift schedule is available
Scenarios	Assign an id to the slot Assign time for the slot Assign one of the available dentists to the slot
Exceptions	Scheduling a dentist more than once at the same time Specifying time outside working hours
Related use case	
Pre condition	Shift schedule is verified
Postcondition	Appointment slots are stored in permanent storage

Examples of use case description: Reserve Slot

Use case name	Reserve slot
Actors	Patient, Dentist
Brief Description	A patient reserves an appointment with a dentist
Triggering Event	A patient has requested for appointment
Scenarios	1. Determine if the requested slot is available 2. Reserve the slot
Exceptions	1. Requested slot is not available
Related use case	Select slot, Receive confirmation
Pre condition	The patient is verified
Post condition	Reserved slot is stored in permanent storage

2.3 Class Responsibility Collaboration (CRC) Cards

Class Responsibilities Collaborators (CRC) was invented by Ward Cunningham and Kent Beck as an approach to discovering and documenting objects in OO design. CRC was initially designed to simplify learning OOP but has also been used in professional software development such as Agile's eXtreme Programming (XP). A class represents a template from which similar objects are created, responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.

CRC card is a 3x5 index card and is partitioned into three: the first and the topmost portion is a row in which the name of the class being considered is represented; the second and the leftmost portion, represent the responsibilities of that class; the third portion represents the collaborators that the class will need to complete its responsibilities. Figure 4 presents the major CRC cards derivable from the Dental Appointment Booking System

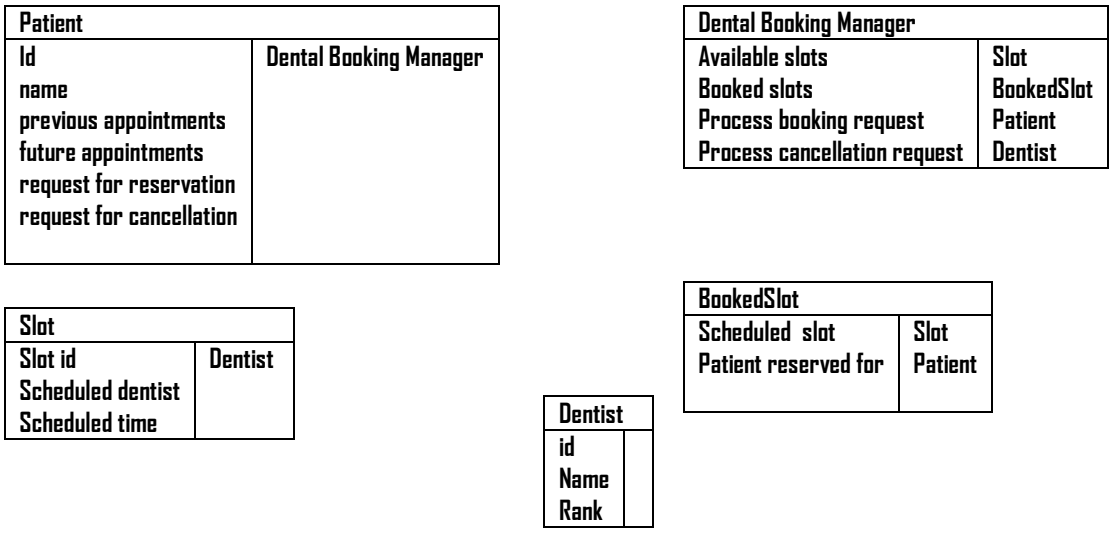


Figure 4: CRC diagram of Dental Booking System

2.3 Activity Diagram

The activity diagram is one of the behavioural diagrams of Unified Modelling Language (UML). It is similar to the flowchart as it can be used to diagrammatically represent a series of actions or flow of control to solve a given problem. Similarly, it can be used for other things such as modelling business processes or behavioural descriptions of a use case diagram.

Swim lane activity is an activity diagram that is used to show which system actor is responsible for what, in addition to the representation of the series of actions or flow of control to solve the problem. Thus, the presence of objects as well as their high level responsibilities can be captured explicitly as system actors on the diagram. Figure 5 is an activity diagram that represents the same solution depicted in Figure 4 as CRC. It is a swim lane activity diagram because the responsibilities are indicated under the System and Patient as the main actors.

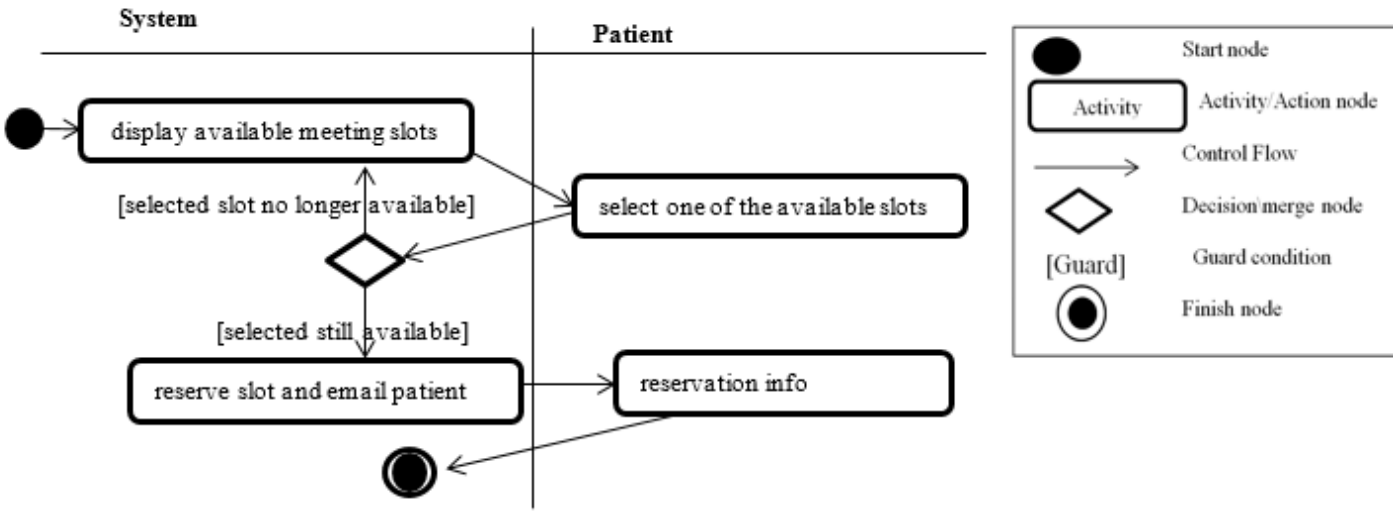


Figure 5: Swim lane activity diagram representing the design of a Dental Booking System

3.0 Unified Modeling Language (UML)

UML has become the de facto standard for modeling software applications and is growing in popularity in modeling other domains. Its roots go back to three distinct methods influenced by Ivar Jacobson. UML is a language; it has both syntax and semantics.

UML attempts to bridge the gap between the original idea for a piece of software and its implementation. UML 2.0 is central to the MDA/MDD effort: A relatively new way to develop executable models that tools can link together and to raise the level of abstraction above traditional programming languages.

The Diagram Interchange Specification was written to provide a way to share UML models between different modeling tools.

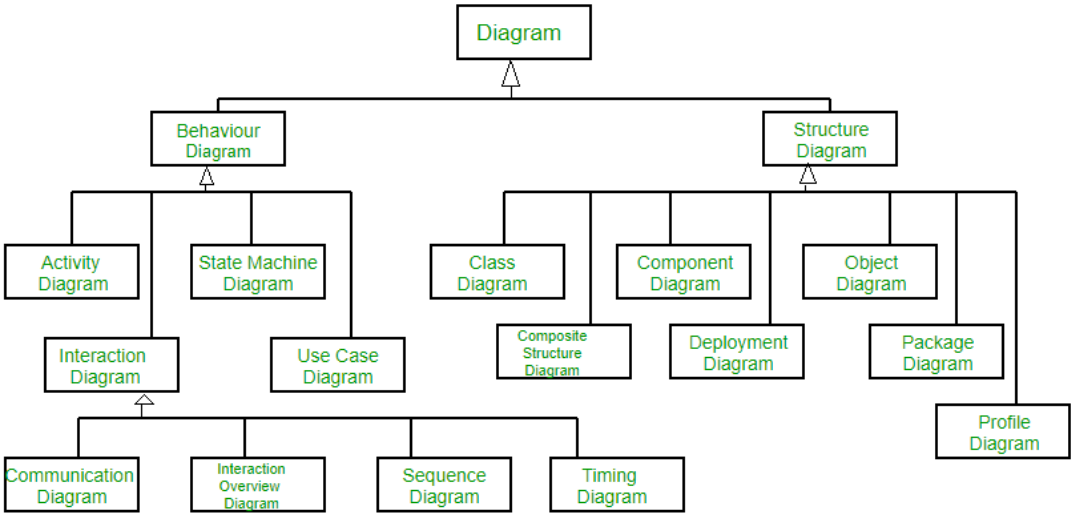
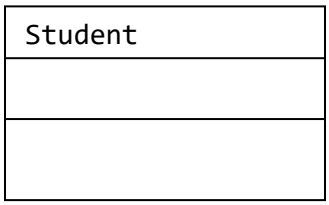


Figure 6: Taxonomy of UML models

3.1 Class

In UML, a class is represented by a rectangular box divided into compartments. A compartment is simply an area in the rectangle to write information. The first compartment holds the name of the class, the second holds attributes, and the third is used for operations. You can hide any compartment of the class if that increases the readability of your diagram. A model is never meant to be 'complete'. When reading a diagram, you can make no assumptions about a missing compartment; it doesn't mean it is empty.



3.1.1 Attribute and Operations

An attribute can be shown using two different notations: inlined or relationships between classes. You can list a class's attributes right in rectangle notation; these are typically called inlined attributes.

Syntax:

Visibility (-/+) name: type

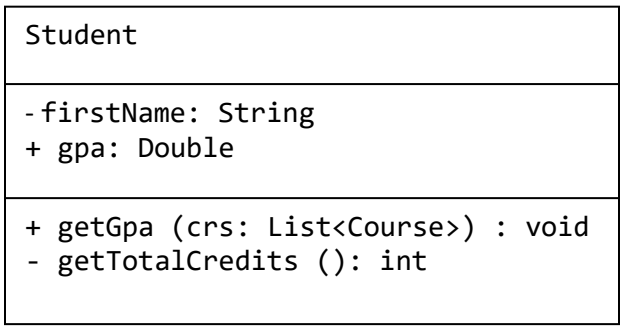
e.g. – firstName: String, + gpa: double

Operations are features of classes that specify how to invoke a particular behavior

Syntax:

visibility name (parameters) : return-type

E.g + computeGpa (std: Student) : void



There are many other notations but in this course we need only few + relationships

4.0 Implementation of class and object

So far, we have introduced the concepts of class, object, and behavior in an abstract manner.We will now demonstrate how to implement these concepts in the Java programming language. Figure 7 represents the syntax of class declarations in Java.

```
1  accessModifier class className {
2    // states/attributes declarations
3    // constructor
4    // method declarations
5  }
```

Figure 7: Syntax of class declaration in Java

In Figure 7 the `accessModifier` specifies the *visibility* of the class to other programming entities. The word `class` is required; it is a keyword. The `className` is an identifier. Between the opening and closing braces, we can have state/attribute declarations, class constructors, and method declarations. The attributes, constructors, and methods are collectively known as *members* of the class. As an example, Figure 8 illustrates a bare minimum implementation of the `Staff` class from the supermarket domain.

```
8   public class Staff {
9       String staffId;
10      String department;
11      String rank;
12
13      public static void main(String[] args) {
14          Staff s1 = new Staff();
15          s1.takeOrder();
16      }
17
18      public void takeOrder() {
19          System.out.println("Taking Order");
20      }
21
```

```
Output - OOPBook (run)
run:
Taking Order
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 8: Bare minimum implementation of staff class in Java

In Figure 8, you may have noticed that the class has four attributes, all of which are of the `String` data type. The `takeOrder()` method has only one statement: printing "`taking order`". The class also includes a `main` method (`Lines 13-16`) in which an object of the `class` is created and assigned the name `s1` with the statement `Staff s1 = new Staff();`. In this statement, the first appearance of `Staff` refers to the `Staff` class, `s1` is the new object from the `Staff` class, `new` is the required keyword, and `Staff()` is used to call for the actual creation (instantiation) of the `Staff` object. After the object is created, the `takeOrder()` method is called, resulting in the outcomes shown at the bottom of Figure 8.

Let's create a method, `printStaffDetail()`, to print the details of the object *s1* that has been

created from the Staff:

```
18      public void printStaffDetail() {
19          System.out.println("Staff ID: " + staffId);
20          System.out.println("Staff Name: " + name);
21          System.out.println("Staff Department: " + department);
22          System.out.println("Staff Rank: " + rank);
23      }
```

Figure 9: Additional method, `printStaffDetail()`, in the `Staff` class

The method, `printStaffDetail()` in Figure 9 shall print the string literal enclosed in double quotation marks, such as "`Staff ID`:", followed by the actual value assigned to the attributes. The plussymbol (+) is used for concatenating the string literals with the actual value of the attributes.

Now, let's also assign specific values to the attributes of the instantiated object. We will do this within the main method after the statement `Staff s1 = new Staff();` and before the statement `s1.takeOrder();`. The dot operator (.) is used to access properties of an object if they are visible (See Figure 10).

```
s1.staffId = "0001";
s1.name = "Blessing";
s1.department = "Sale";
s1.rank = "Sales girl";
```

Figure 10: Accessing properties of an object using dot operator

Before we are done, we will also want to execute the method `printStaffDetail()` before we execute the `takeOrder()` method as follows:

```
s1.printStaffDetail();
s1.takeOrder();
```

Figure 11: Method to print staff details before the `takeOrder` method

The complete program and its output, when compiled and executed, are shown in Figure 12.

```
8   public class Staff {
9       String staffId;
10      String name;
11      String department;
12      String rank;
13      public void printStaffDetails() {
14          System.out.println("Staff Id" + staffId);
15          System.out.println("Name" + name);
16          System.out.println("Department" + department);
17          System.out.println("Rank" + rank);
18      }
19      public static void main(String[] args) {
20          Staff s1 = new Staff();
21          s1.staffId = "001";
22          s1.name = "Blessing";
23          s1.department ="Sale";
24          s1.rank = "Sales Girl";
25          s1.printStaffDetails();
26          s1.takeOrder();
27      }
28      public void takeOrder() {
29          System.out.println("Taking Order");
30      }
31
32  }
```

```
run:
Staff Id001
NameBlessing
DepartmentSale
RankSales Girl
Taking Order
BUILD SUCCESSFUL (tota
```

Figure 12: Implementation of revised `staff` class in Java

4.1 How objects interact in Java programming

Objects from different classes can interact **by sending messages to each other**. This interaction involves objects calling each other's methods, also known as *method invocation*. Let's consider an example from the real-world scenario of a supermarket. Suppose there's a supermarket named Ladan Wapa, with Blessing as one of the saleswomen. Two objects of a `Customer` class, Muhammad and Kabir, intend to buy bags of rice from the supermarket at a particular time. They will place their orders with Blessing, the saleswoman.

When sending a message to an object, which essentially means calling methods of that object, the caller may provide additional information. The information is specified in the method'sdefinition within the parentheses after the method name and is referred to as its ***parameters***. For instance, to make the `takeOrder()` method of the `Staff` object more useful, the method expects specific information from the caller, such as the type of product they wish to buy (e.g., rice) and the quantity of the item (e.g., 2 bags).

This information can be defined as parametersof the `takeOrder()` method, as shown in Figure 13:

```
void takeOrder(Product product,int quantity{... }
```

Figure 13: Parameters of the `takeOrder` method

Here, `product` (with a lowercase `p`) is a parameter, and `Product` (with an uppercase `P`) is the data type, specifically a reference to the Product object. Similarly, `quantity` is another parameter representing the amount of the product to be ordered, and its data type is `int` (integer).

In response to Muhammad's message as a customer, Blessing, the saleswoman, replies byreturning the result of his request. To simplify, let's modify the `takeOrder()` method to return the value of the order as a response from Blessing (See Figure 14):

```
float takeOrder(Product product, int quantity) {
return product.price * quantity;
}
```

Figure 14: The `takeOrder` method that returns the value of the order

In this modified method of Figure 14, the `float` data type replaces the `void` keyword to indicate that the method is returning a result, specifically of type floating-point, to the calling object. The `return` keyword is used, and the method returns the price of the product multiplied by the requested quantity.

We will now implement the Ladan Wapa supermarket scenario with the following objects: `Staff`, `Product`, and `Customer`.

```java
public class Staff {
    String staffId;
    String name;
    String department;
    String rank;

    public void printStaffDetail() {
        System.out.println("Staff ID: " + staffId);
        System.out.println("Staff Name: " + name);
        System.out.println("Staff Department: " + department);
        System.out.println("Staff Rank: " + rank);
    }

    public float takeOrder(Product p, int quantity) {
        System.out.println("taking order");
        return p.price * quantity;
    }
}
```

Figure 15: Implementation of `Staff` class

Notice that the main method has been removed from the `Staff` class -`Staff` class no longer has an entry point to the program. Also, the `takeOrder()` method has been modified with additional parameters to return a result. Next is the implementation of the `Product` class (See Figure 16).

```java
public class Product {
      String productName;
      String description;
      String manufacturer;
      float price;
 }
```

Figure 16: Implementation of the `Product` class

Nothing fancy in the `Product` class in Figure 16. All attributes are declared with the appropriate data type. Next is the implementation of the `Customer` class.

```java
public class Customer {

    String name;
    String address;
    double budget;

    void makePurchase(Product p, int qty, Staff staff) {
        float result = staff.takeOrder(p, qty);
        System.out.println("The value recived is: " + result);
    }
}
```

Figure 17: Implementation of the `Product` class

The `Customer` class in Figure 3.16 also has its attributes declared with the appropriate data type. In addition, there is a method, `makePurchase()`, that is declared to have `Product`, `quantity`, and

Staff as its parameters. In addition, the method is the point where the Customer object sends a message to the Staff object using the staff.takeOrder(p, qty) statement. The response returned by the Staff object is stored in the variable result of float data type.

Next,we implement the SuperMarket class to test the interaction between objects.

```
public class SuperMarket {
    public static void main(String[] args) {
//creating the first Product object
        Product product = new Product();
        product.productName = "Rice";
        product.description = "Bag";
        product.price = 30000;
//creating the second Product object
        Product p2 = new Product();
        p2.productName = "Spagetti";
        p2.description = "Carton";
        p2.price = 7000;
//Creating the Staff Object
        Staff s1 = new Staff();
        s1.staffId = "001";
        s1.name = "Blessing";
        s1.department = "Sales";
        s1.rank = "Sales girl";
//creating customers objects
//creating the first Customer
        Customer c1 = new Customer();
        c1.name = "Muhammad";
        c1.address = "Hamada Carpet Katsina";
        c1.budget = 100000;
//creating the first Customer
        Customer c2 = new Customer();
        c2.name = "Kabir";
        c2.address = "Hayin Gada Dutsinma";
        c2.budget = 120000;
        c1.makePurchase(p2, 2, s1);
        c2.makePurchase(product, 1, s1);

    }
}
```

Figure 18: Implementation of the SuperMarket class

The Supermarket class in Figure 18 contains the main method. Inside the main method, the following steps are executed:

First, two Product objects, product and p2, are instantiated to represent the products "Rice" and "Spaghetti," respectively.

- Next, a Staff object is created to represent Blessing, one of the saleswoman.
- After that, two customer objects are instantiated to represent Muhammad and Kabir.

- Finally, Muhammad, as a `customer` object, purchases two cartons of spaghetti, and Kabir, as another `customer` object, purchases one bag of rice.

When the program is compiled and executed successfully, the output shown in Figure 19 will be displayed.

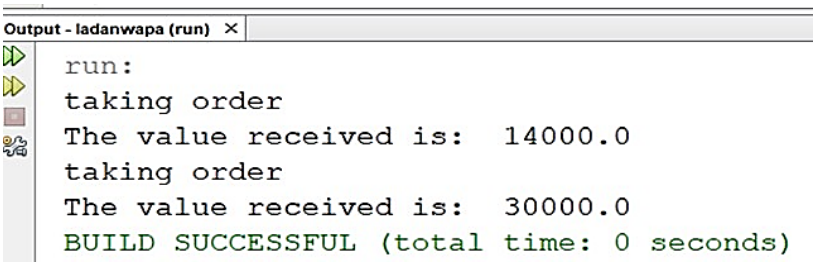


Figure 19: Output of Customer objects interacting with Staffobject

The interactions between Muhammad and Blessing in the aforementioned real-world scenario can be effectively represented using the concepts of object-oriented programming. In this context, Muhammad and Blessing are instances of different classes (namely, the `Staff` and `Customer` classes) that engage in communication through messages.
The act of Muhammad or Kabir placing a `takeOrder` request to Blessing exemplifies the idea of sending a message. This message could potentially include accompanying information referred to as *actual parameters* or *arguments*. Muhammad understands that if his message is valid, Blessing will undoubtedly respond appropriately. However, the specific mechanism employedby Blessing to fulfill this request might remain unknown to Muhammad. This concept underlines a crucial principle in object-oriented programming known as "information hiding," where the sender of a message is unaware of the precise method by which the receiver will fulfill the request in the message.

In summary, methods are the actions that an object can execute, facilitating interactions and exchanges of messages between objects. A straightforward method is one that performs certain actions but does not provide any output to the calling object, nor does it necessitate anyadditional input from the caller. An illustration of this is the `printStaffDetail()` method within the `Staff` class, which is a simple method that neither returns a result nor requires parameters. However, methods can also return results to the calling object and can accept parameters. For instance, the void `makePurchase(Product p, int qty, Staff staff)` method within the `Customer` class takes parameters but does not yield a result. Conversely, the `float takeOrder(Product p, int qty)` method within the `Staff` class possesses both parameters and returns a result.

In this context, the general syntax of methods can be depicted as follows in Figure 3.20:

```
1  accessModifier returnType methodName ([parameterType  parameter,]) {
2  statements;
3  return value/expression
4  }
```
Figure 20: General syntax of methods