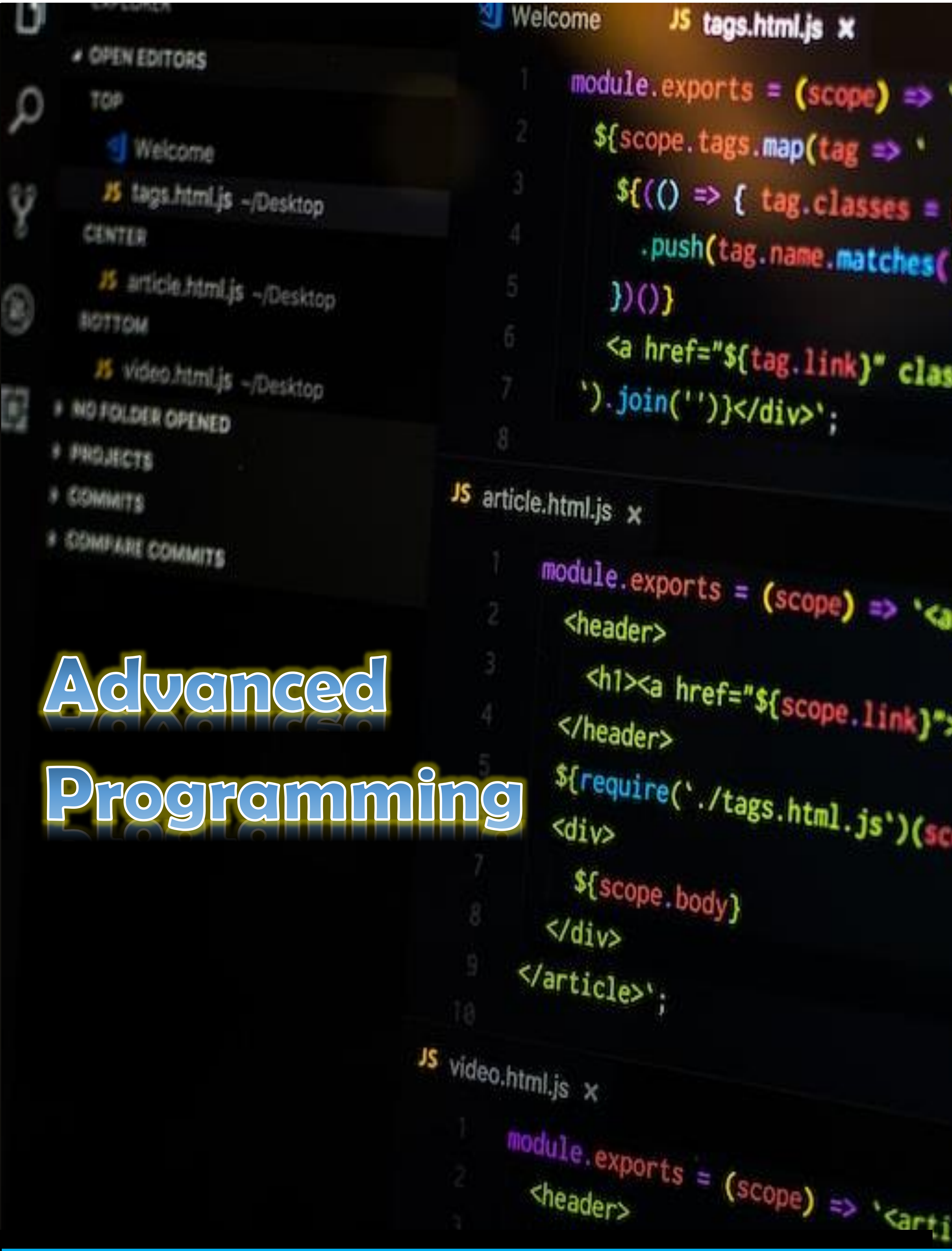


OPEN EDITORS
TOP
Welcome
tags.html.js ~/Desktop
CENTER
article.html.js ~/Desktop
BOTTOM
video.html.js ~/Desktop
NO FOLDER OPENED
PROJECTS
COMMITS
COMPARE COMMITS

Welcome tags.html.js ×

```
module.exports = (scope) =>
  ${scope.tags.map(tag => `
    ${(() => { tag.classes =
      .push(tag.name.matches(
    })()}
    <a href="${tag.link}" clas
  `).join('')}</div>`;
```

article.html.js ×

```
module.exports = (scope) => `<a
  <header>
    <h1><a href="${scope.link}">
  </header>
  ${require('./tags.html.js')(sc
  <div>
    ${scope.body}
  </div>
</article>`;
```

video.html.js ×

```
module.exports = (scope) => `<arti
  <header>
```

Advanced Programming

Ryan Wickramaratne | COL 00081762

Higher Nationals

Internal verification of assessment decisions – BTEC (RQF)

INTERNAL VERIFICATION – ASSESSMENT DECISIONS			
Programme title	HND in Computing - Application Dev / Software Eng. Pathway		
Assessor	Mr. Steve	Internal Verifier	
Unit(s)	Unit 20 – Advance Programming		
Assignment title			
Student's name	Ryan Wickramaratne (COL 00081762)		
List which assessment criteria the Assessor has awarded.	Pass	Merit	Distinction

INTERNAL VERIFIER CHECKLIST		
Do the assessment criteria awarded match those shown in the assignment brief?	Y/N	
Is the Pass/Merit/Distinction grade awarded justified by the assessor's comments on the student work?	Y/N	
Has the work been assessed accurately?	Y/N	
Is the feedback to the student: Give details: • Constructive? • Linked to relevant assessment criteria? • Identifying opportunities for improved performance? • Agreeing actions?	Y/N Y/N Y/N Y/N	
Does the assessment decision need amending?	Y/N	

Assessor signature		Date	
Internal Verifier signature		Date	
Programme Leader signature (if required)		Date	

Confirm action completed			
Remedial action taken Give details:			
Assessor signature		Date	

Internal Verifier signature		**Date**	
Programme Leader signature (if required)		**Date**	

Higher Nationals - Summative Assignment Feedback Form

Student Name/ID	Ryan Wickramaratne (COL 00081762)		
Unit Title	**Unit 20 – Advance Programming**		
Assignment Number	1	**Assessor**	Mr. Steve
Submission Date	15/08/2023	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	

Assessor Feedback:

LO1. Examine the key components related to the object orientated programming paradigm, analysing design pattern types.

Pass, Merit & Distinction Descripts P1 ☐ M1 ☐ **D1** ☐

LO2. Design a series of UML class diagrams

Pass, Merit & Distinction Descripts P2 ☐ M2 ☐ D2 ☐

LO3. Implement code applying design patterns

Pass, Merit & Distinction Descripts P3 ☐ M3 ☐ D3 ☐

LO4. Investigate scenarios with respect to design patterns

Pass, Merit & Distinction Descripts P4 ☐ M4 ☐ D4 ☐

Grade:	**Assessor Signature:**	**Date:**

Resubmission Feedback:

Grade:	**Assessor Signature:**	**Date:**

Internal Verifier's Comments:

Signature & Date:

* Please note that grade decisions are provisional. They are only confirmed once internal and external moderation has taken place and grades decisions have been agreed at the assessment board.

Assignment Feedback

Formative Feedback: Assessor to Student			
Action Plan			
Summative feedback			
Feedback: Student to Assessor			
Assessor signature		**Date**	
Student signature		**Date**	

Pearson Higher Nationals in Computing

Unit 20 – Advance Programming

General Guidelines

1. A Cover page or title page – You should always attach a title page to your assignment. Use previous page as your cover sheet and make sure all the details are accurately filled.
2. Attach this brief as the first section of your assignment.
3. All the assignments should be prepared using a word processing software.
4. All the assignments should be printed on A4 sized papers. Use single side printing.
5. Allow 1" for top, bottom , right margins and 1.25" for the left margin of each page.

Word Processing Rules

1. The font size should be **12** point, and should be in the style of **Time New Roman**.
2. **Use 1.5 line spacing**. Left justify all paragraphs.
3. Ensure that all the headings are consistent in terms of the font size and font style.
4. Use **footer function in the word processor to insert Your Name, Subject, Assignment No, and Page Number on each pag**e. This is useful if individual sheets become detached for any reason.
5. Use word processing application spell check and grammar check function to help editing your assignment.

Important Points:

1. It is strictly prohibited to use textboxes to add texts in the assignments, except for the compulsory information. eg: Figures, tables of comparison etc. Adding text boxes in the body except for the before mentioned compulsory information will result in rejection of your work.
2. Carefully check the hand in date and the instructions given in the assignment. Late submissions will not be accepted.
3. Ensure that you give yourself enough time to complete the assignment by the due date.
4. Excuses of any nature will not be accepted for failure to hand in the work on time.
5. You must take responsibility for managing your own time effectively.
6. If you are unable to hand in your assignment on time and have valid reasons such as illness, you may apply (in writing) for an extension.
7. Failure to achieve at least PASS criteria will result in a REFERRAL grade.
8. Non-submission of work without valid reasons will lead to an automatic RE FERRAL. You will then be asked to complete an alternative assignment.
9. If you use other people's work or ideas in your assignment, reference them properly using HARVARD referencing system to avoid plagiarism. You have to provide both in-text citation and a reference list.
10. If you are proven to be guilty of plagiarism or any academic misconduct, your grade could be reduced to A REFERRAL or at worst you could be expelled from the course

Student Declaration

I hereby, declare that I know what plagiarism entails, namely to use another's work and to present it as my own without attributing the sources in the correct form. I further understand what it means to copy another's work.

1. I know that plagiarism is a punishable offence because it constitutes theft.
2. I understand the plagiarism and copying policy of Edexcel UK.
3. I know what the consequences will be if I plagiarise or copy another's work in any of the assignments for this program.
4. I declare therefore that all work presented by me for every aspect of my program, will be my own, and where I have made use of another's work, I will attribute the source in the correct way.
5. I acknowledge that the attachment of this document signed or not, constitutes a binding agreement between myself and Pearson, UK.
6. I understand that my assignment will not be considered as submitted if this document is not attached to the assignment.

[illegible]

15/08/2023

ryandilthusha@gmail.com

Student's Signature: **Date:**

(*Provide E-mail ID*) **(*Provide Submission Date*)**

Higher National Diploma in Business

Assignment Brief

Student Name /ID Number	Ryan Wickramaratne (COL 00081762)
Unit Number and Title	**Unit 20 – Advance Programming**
Academic Year	2021/22
Unit Tutor	Mr. Steve
Assignment Title	
Issue Date	
Submission Date	
IV Name & Date	

Submission format

The submission is in the form of an individual written report about. This should be written in a concise, formal business style using single spacing and font size 12. You are required to make use of headings, paragraphs and subsections as appropriate, and all work must be supported with research and referenced using the Harvard referencing system. Please also provide an end list of references using the Harvard referencing system. Please note that this is an activity-based assessment where your document submission should include evidences of activities carried out and of team working. To carry out activities given on the brief, you are required to form groups, comprising not exceeding 15 individuals.

The recommended word count is 4,000–4,500 words for the report excluding annexures. Note that word counts are indicative only and you would not be penalised for exceeding the word count.

	Unit Learning Outcomes:
	Learning Outcomes By the end of this unit students will be able to: LO1. Examine the key components related to the object-orientated programming paradigm, analysing design pattern types. LO2. Design a series of UML class diagrams. LO3. Implement code applying design patterns. LO4 Investigate scenarios with respect to design patterns.

Assignment Brief and Guidance:

BOC Software solutions is leading software company in Kandy as system analyst you have to build an application for **Family Dental Care (FDC)** considering given scenario.

Family Dental Care (FDC) is a leading up market dental surgery located in Kandy. It provides all types of dental treatments to patients which include extractions, nerve fillings, maxillofacial surgeries (i.e. surgeries involving jaw bone) and sophisticated dental implants. It is visited by prominent dentists and dental consultants with post graduate qualifications, some of whom are working at the Faculty of Dental Science at the University of Peradeniya.

Patients consult doctors by appointment. On their first visit, patients are required to register by entering their personal details such as name, address, national identity card number and contact number. A small fee is charged from the patient during registration. A separate fee is charged for each treatment given.

Doctors too must get registered at FDC by providing personal details such as name, address, date of birth, national ID number and contact number. In addition, consultants must provide the name of their post graduate qualification along with the country of the University that granted it and ordinary dentists should indicate the number of years of experience.

FDC consists of four fully equipped surgery rooms so that four patients can be accommodated at any given time. FDC also contains a dental scan room which can be attended by one patient at a time. The dental scan machine is operated by one of the dentists of the FDC facility. Normally, a dentist without appointments for a given time slot (say, between 5 PM and 6 PM) is assigned to the machine by the manager. When that time slot finishes, another doctor who is free will be assigned.

The staff of FDC is made up of a manager, four nurses (one for each of the four surgery rooms) and a receptionist who handles registrations and appointments.

An information system is required to keep track of patients, doctors, appointments, treatments given to patients and payments. The system must also maintain information about the staff. It has been decided to use an object oriented approach to design and implement the system.

Task 1

Examine the Object oriented concepts given below. Provide diagrams and code snippets from suitable specific programming language to supplement your explanations.

i) Class
ii) Object
iii) Message
iv) Encapsulation
v) Inheritance
vi) Polymorphism
vii) Aggregation/composition

Task 2

Design and build the detailed UML class diagram for the Family Dental Care system. Your solution should demonstrate all inter-class relationships namely Association, Inheritance and Aggregation/composition. The classes should include attributes and methods needed.

Draw the class diagram for the explained system. Including all notations and details and ensure that the diagram has the required functionalities. Analyze the class diagram provided above and derive code scenarios related to the UML diagram.

Task 3

Determine and briefly discuss the range of design patterns and describe at least one design pattern from the three available types of design pattern. Provide suitable UML diagrams for the given patterns and analyze the relationship between object-oriented paradigm and design patterns providing a suitable example.

Task 4

Scenario 1

FDC owns a very expensive, state of the art dental scan machine (a device far superior to a traditional dental X-ray machine) manufactured by Toshiba, Japan. FDC will be own just one such machine in the foreseeable future. When modeling and implementing FDC system in software, you must ensure that only one instance of that machine is created.

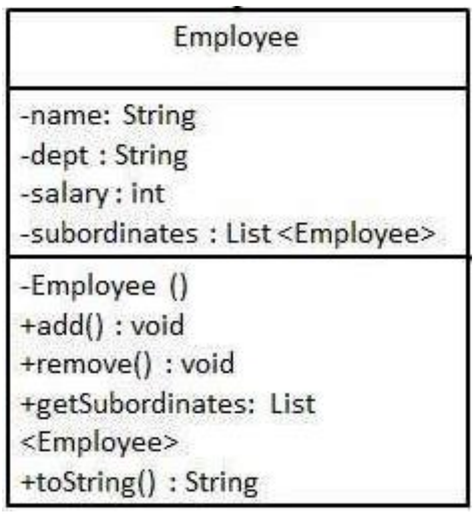
You may include suitable attributes for the machine such as serial number, make, country of origin and cost. Implementation should allow the user to enter details of the dental scanner and create the sole instance of that machine.

Scenario 02

Below table provides the hierarchy of the Employees and their monthly salary in FDC.

Emp Id	Name	Position	Salary
A001	Anton	Director	1,000,000 LKR
A002	Chamod	Dentist	600,000 LKR
A003	Supuni	Dentist	600,000 LKR
A004	Madhavi	Dentist	600,000 LKR
A005	Piyal	Nurse	200,000 LKR
A006	Kamal	Nurse	200,000 LKR
A007	Kapila	Nurse	200,000 LKR

All the Nurses are working under the Dentists and Chamod(Dentist) is working under the Supuni(Dentist) and Madhavi(Dentist) is working under the Anton(Director).



Develop a system to display the details of all employees and your system should display which employee is working under which employee clearly.

Scenario 03

Patients who need dental scans are kept in a First in First Out queue. Assume that you have found an already developed Queue container in a software library. It provides standard queue operations to insert and remove data (known as enqueue and deque respectively). However, you need some specific operations such as search() to look up a particular patient and showAll() to list all the patients in the queue. These additional operations are not provided by the library unit.

For each of the above scenarios:

Select and Justify the most appropriate design pattern for each of the above given scenarios then Define and Draw class diagrams for above mentioned design patterns and develop code for the above scenarios (except for the 3rd Scenario) using an appropriate programming language. Critically evaluate why you selected the above design patterns and compare your answer with the range of design patterns available.

Grading Rubric

Grading Criteria	**Achiev e d**	**Feedback**
LO1 Examine the key components related to the object-orientated programming paradigm, analyzing design pattern types		
P1 Examine the characteristics of the object-orientated paradigm as well as the various class relationships.		
M1 Determine a design pattern from each of the creational, structural and behavioral pattern types.		
D1 Analyse the relationship between the object-orientated paradigm and design patterns.		
LO2 Design a series of UML class diagrams		
P2 Design and build class diagrams using a UML tool.		
M2 Define class diagrams for specific design patterns using a UML tool.		
D2 Analyse how class diagrams can be derived from a given code scenario using a UML tool.		
LO3 Implement code applying design patterns		
P3 Build an application derived from UML class diagrams.		
M3 Develop code that implements a design pattern for a given purpose.		
D3 Evaluate the use of design patterns for the given purpose specified in M3.		
LO4 Investigate scenarios with respect to design Patterns		

P4 Discuss a range of design patterns with relevant examples of creational, structure and behavioral pattern types.		
M4 Reconcile the most appropriate design pattern from a range with a series of given scenarios.		
D4 Critically evaluate a range of design patterns against the range of given scenarios with justification of your choices.		

Acknowledgement

I would like to express my special thanks of gratitude to my Advance Programming lecturer Mr. Steve for providing invaluable guidance and giving immense amount of knowledge to work on this assignment perfectly. I specially thanks her because she helped us in doing a lot of research and I came to know about so many new things about the Systems Analysis & Design.

Secondly, I would like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Executive Summary

Throughout this comprehensive study, I delved deep into the intricate facets of the Object-Oriented Paradigm, specifically within the context of the Family Dental Care (FDC) scenario. Beginning with the foundational concepts of classes and objects, I systematically explored the dynamics of messages, encapsulation, inheritance, and polymorphism. Our emphasis on explaining code, paired with UML diagrams, ensured a practical and visual grasp of these abstract concepts.

Further, the exploration of class relationships underscored the inherent flexibility and expressiveness of object-oriented design. Our analysis then transitioned to a meticulous examination of design patterns. By illuminating patterns such as Singleton, Composite, and Observer, I highlighted the synergy between these design templates and OOP principles. Each pattern's application to the FDC scenario illustrated its real-world relevancy and potential to enhance system design.

Finally, the in-depth scenarios in Task 4 underscored the potency of these patterns. Through detailed class diagrams, code, and comparative analyses, I determined the optimal design strategy for each unique requirement.

In essence, this assignment has not only fortified our understanding of OOP and design patterns but also showcased their pivotal role in architecting robust, maintainable, and efficient software solutions, especially for intricate domains like healthcare.

List of figures

TABLE OF CONTENTS

Task 1

1.1 Class

1.1.1 Understand what a class is

In Object Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. This blueprint outlines a set of attributes (also known as fields or variables) and methods (functions) that define the characteristics and behaviors of an object of that class.

A class in essence, is a custom data type that encapsulates data and operations on that data within a single unit. The data encapsulated within the class is represented as attributes, and the operations are represented as methods.

For example, consider a real-world object like a car. If we were to model a car as an object in OOP, we would define a class, say "Car". This class might have attributes such as "color", "model", "engine", and so forth, which describe the properties of the car. It may also have methods like "start", "stop", "accelerate", and "brake", which describe the behaviors or actions a car can perform.

It's also important to note that a class itself is not an object but a definition or prototype from which objects can be created. In the "Car" example, we could use the "Car" class to create specific car objects, like a red Ferrari or a blue BMW. Each of these car objects is an instance of the "Car" class.

1.1.2 Define a class relevant to the FDC scenario

In the Family Dental Care (FDC) system, various classes can be identified based on the details of the assignment brief. For instance, we can consider "Patient" as a class.

The "Patient" class could encapsulate attributes such as "name", "address", "nationalIdentityCardNumber", "contactNumber", and "registrationFee". The "name", "address", "nationalIdentityCardNumber", and "contactNumber" attributes would store the personal details of the patient which are collected at the time of registration. The "registrationFee" attribute would represent the fee charged to the patient at the time of registration.

In terms of methods, the "Patient" class could include methods such as "register", which could be used to register a new patient, and "getDetails", which could be used to retrieve the details of a patient.

This class encapsulates all the necessary data fields and operations related to a patient, adhering to the principles of object-oriented programming and representing a crucial part of the FDC system. In the context of the full FDC system, additional classes like "Doctor", "Treatment", "Appointment", etc., would be defined and used alongside the "Patient" class.

1.1.3 Code for the “Patient”class

In Java, the class for "Patient" could be written as follows:

```java
public class Patient {
    // Attributes
    private String name;
    private String address;
    private String nationalIdentityCardNumber;
    private String contactNumber;
    private double registrationFee;

    // Constructor
    public Patient(String name, String address, String
nationalIdentityCardNumber, String contactNumber, double
registrationFee) {
        this.name = name;
        this.address = address;
        this.nationalIdentityCardNumber = nationalIdentityCardNumber;
        this.contactNumber = contactNumber;
        this.registrationFee = registrationFee;
    }

    // Getters and Setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getNationalIdentityCardNumber() {
        return nationalIdentityCardNumber;
    }

    public void setNationalIdentityCardNumber(String
nationalIdentityCardNumber) {
        this.nationalIdentityCardNumber = nationalIdentityCardNumber;
    }

    public String getContactNumber() {
        return contactNumber;
    }

    public void setContactNumber(String contactNumber) {
        this.contactNumber = contactNumber;
    }

    public double getRegistrationFee() {
        return registrationFee;
    }

    public void setRegistrationFee(double registrationFee) {
        this.registrationFee = registrationFee;
    }
}
```

1.1.4 Explanation of the code

In the provided Java code, the class named `Patient` has been defined. The class contains five private attributes: `name`, `address`, `nationalIdentityCardNumber`, `contactNumber`, and `registrationFee`.

These attributes represent the data that we want to store for each patient. The `private` access modifier is used to ensure that these attributes cannot be accessed directly from outside the class, which is a principle known as encapsulation. This way, we can control how the attributes are set and retrieved.

A constructor is defined to initialize a new object of the `Patient` class. This constructor takes five parameters: `name`, `address`, `nationalIdentityCardNumber`, `contactNumber`, and `registrationFee`. When a new `Patient` object is created, the constructor will be called with the corresponding arguments, and these arguments will be used to set the initial state of the object.

Getter and setter methods have been defined for each of the attributes. These methods allow us to retrieve (get) and modify (set) the values of the attributes. The getter method for an attribute returns the attribute's value, while the setter method takes a parameter and assigns it to the attribute. For example, the `getName` method returns the `name` of the patient, and the `setName` method sets a new `name` for the patient.

By using getter and setter methods instead of accessing the attributes directly, we can add additional logic if needed. For example, we could add validation rules in the setter methods to prevent invalid data from being set.

This is a very basic version of the `Patient` class. In a real-world application, this class might also include additional methods to perform operations relevant to a patient, such as scheduling an appointment, paying fees, etc.

<u>1.1.5 Create a UML diagram for the above class.</u>

A UML class diagram for the above Patient class can be depicted as follows:

```
---------------------
|       Patient      |
---------------------
| -name: String      |
| -address: String   |
| -nationalIdentityCardNumber: String |
| -contactNumber: String |
| -registrationFee: double |
---------------------
| +Patient(name: String, address: String, nationalIdentityCardNumber:
String, contactNumber: String, registrationFee: double) |
| +getName(): String |
| +setName(name: String): void |
| +getAddress(): String |
| +setAddress(address: String): void |
| +getNationalIdentityCardNumber(): String |
| +setNationalIdentityCardNumber(nationalIdentityCardNumber: String):
void |
| +getContactNumber(): String |
| +setContactNumber(contactNumber: String): void |
| +getRegistrationFee(): double |
| +setRegistrationFee(registrationFee: double): void |
---------------------
```

The UML class diagram includes the class name (`Patient`), the attributes, and the methods (including the constructor). The `-` sign before the attributes signifies that the attributes are private. The `+` sign before the methods signifies that the methods are public.

Each attribute's line includes its name and type. Similarly, each method's line includes its name, parameters, and return type.

The constructor is represented as a method with the same name as the class. In this case, it's `Patient`. It includes all the parameters required for creating a new `Patient` object.

Each attribute and method has been represented based on the visibility, name, type, and parameters (if any). This way, it gives a comprehensive overview of what the `Patient` class is about and how it functions.

The UML class diagram is a visual representation of the structure of the `Patient` class, and it's a crucial part of object-oriented programming and design. It helps in understanding the class and its functionality without diving into the code.

1.2 Object

1.2.1 Understand what an object is

In Object-Oriented Programming (OOP), an object is a particular instance of a class where the class is a blueprint that describes the type of information and behavior that the object will have. Simply put, an object is a real-world entity that has a state and behavior. The state of an object is stored in fields (also referred to as variables), while methods (often known as behaviors) depict what actions an object can take.

For example, consider the real-world concept of a car. The car's make, model, color, and engine type are its attributes, and they make up the state of the car. The behaviors of a car might include starting the engine, accelerating, braking, and turning. All of these attributes and behaviors collectively define the car object.

The power of OOP lies in its ability to simulate real-world objects and interactions within a programming environment. Each object in an OOP language like Java can be manipulated and controlled independently, just like objects in the real world.

1.2.2 Create an object of the class defined in the previous step

After defining a class, we can use it to create objects. An object is created from a class using the `new` keyword followed by a call to a constructor method (which has the same name as the class). The constructor initializes the new object.

To create an object of the `Patient` class we defined earlier, we need to provide data for the patient's name, address, national identity card number, contact number, and registration fee. These values will be passed to the `Patient` constructor when creating a new object.

For instance, we could create a new `Patient` object for a patient named "Ryan Wick", living at "123 Street, Kandy", with the NIC number "123456789V", the contact number "0777123456", and a registration fee of 500 LKR. The process of creating an object using these details is referred to as instantiation, and the resulting object is an instance of the `Patient` class.

Remember that each object has its own set of values for the attributes of the class, so another `Patient` object could have different values. This is how we can represent multiple unique patients in the Family Dental Care system using the `Patient` class.

1.2.3 Code for the above object creation

To create a new Patient object in Java, we can use the following code:

```
Patient RyanWick = new Patient("Ryan Wick", "123 Street, Kandy",
"123456789V", "0777123456", 500);
```

1.2.4 Explanation of the code

Let's break down this line of code:

- `Patient` - This is the class type. When creating an object, we have to specify the type of the object, which is the class we're using to create it.
- `RyanWick` - This is the name of the variable we're creating to hold our `Patient` object. Variable names can be almost anything we want, but it's good practice to make them descriptive of what they hold. In this case, ` RyanWick` is holding a `Patient` object that represents a patient named Ryan Wick.
- `new` - The `new` keyword in Java is used to create a new instance of a class.
- `Patient("Ryan Wick ", "123 Street, Kandy", "123456789V", "0777123456", 500);` - This is a call to the `Patient` class constructor. The constructor is a method in a class that is used to create new objects of that class. In the `Patient` class we defined

earlier, the constructor takes five parameters: name, address, national identity card number, contact number, and registration fee. The values of these parameters are used to initialize the new `Patient` object.

So, the entire line of code `Patient RyanWick = new Patient("Ryan Wick ", "123 Street, Kandy", "123456789V", "0777123456", 500);` is creating a new instance of the `Patient` class, initializing it with the provided details for Ryan Wick, and storing that instance in the ` RyanWick` variable.

The ` RyanWick` object now has all the properties and methods defined in the `Patient` class. We can access these properties and methods using the dot notation. For instance, ` RyanWick.getName()` would return the name of the patient, which in this case would be " Ryan Wick ".

1.3 Message

1.3.1 Understand what a message is in OOP

In the context of object-oriented programming, a message is a mechanism by which an object is able to call a method of another object. Essentially, a message is a function or method call from one object to another object.

The concept of sending a message is slightly different from the traditional function calling mechanism. In OOP, sending a message is like sending a request to another object to execute a method or a group of methods. The object that receives the message is free to decide which method to use to respond to the message.

This form of communication allows objects to interact with each other while keeping their state and behavior private, thus preserving the principles of encapsulation.

1.3.2 Identify a scenario for inter-object communication in the FDC scenario

One common instance of inter-object communication in the Family Dental Care (FDC) system could be the interaction between a `Patient` object and a `Doctor` object for an appointment.

In this case, a `Patient` may send a request or a "message" for an appointment with a `Doctor`. Here, the `Patient` object would need to interact with the `Doctor` object. Specifically, the `Patient` object might call a `makeAppointment()` method on the `Doctor` object, passing along necessary information such as preferred date and time for the appointment.

In response, the `Doctor` object might use that information to check its own schedule (an internal detail encapsulated within the `Doctor` object) and then either confirm the appointment, suggesting a new time slot or redirect the `Patient` to another `Doctor` based on the information provided and its own internal state (like the `Doctor's` schedule and specialty).

The `Patient` object doesn't need to know how the `Doctor` object manages its schedule or decides whether to accept or decline an appointment. All it needs to know is that it can send a "makeAppointment" message to the `Doctor` object and expect a response. This is an example of encapsulation, one of the key principles of object-oriented programming.

The encapsulation is preserved because the `Doctor` object manages its own state and the `Patient` object doesn't have to manage the schedule of the `Doctor`. This leads to a more robust system where each object manages its own state, thus reducing complexity. The objects communicate through well-defined interfaces (like the `makeAppointment()` method), leading to loose coupling between objects, which is a desirable attribute in system design.

1.3.3 Code for the inter-object communication (method calls)

Let's implement the scenario we described in the previous step with code. Here's some Java code for the interaction between a `Patient` object and a `Doctor` object.

Patient Class:

```java
// This is the Patient class
public class Patient {
    private String name;

    // constructor
    public Patient(String name) {
        this.name = name;
    }

    // method to get patient's name
    public String getName() {
        return this.name;
    }

    // method to send a message to Doctor object to make an appointment
    public void makeAppointment(Doctor doctor, String date) {
        doctor.setAppointment(this, date);
    }
}
```

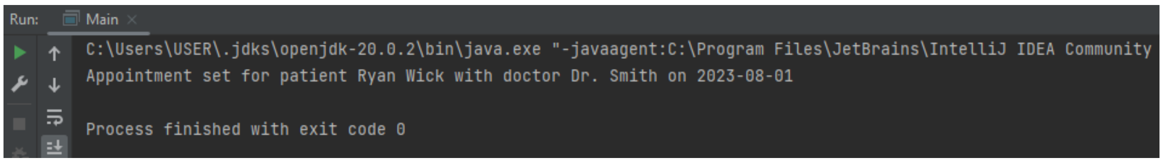
Doctor Class:

```java
// This is the Doctor class
public class Doctor {
    private String name;

    // constructor
    public Doctor(String name) {
        this.name = name;
    }

    // method to accept the appointment
    public void setAppointment(Patient patient, String date) {
        System.out.println("Appointment set for patient " +
patient.getName() + " with doctor " + this.name + " on " + date);
    }
}
```

The main method to initialize these classes and make the call might look something like this:

```java
// The Main class
public class Main {
    public static void main(String[] args) {
        Patient patient = new Patient("Ryan Wick");
        Doctor doctor = new Doctor("Dr. Smith");
        patient.makeAppointment(doctor, "2023-08-01");
    }
}
```

The output:



```
C:\Users\USER\.jdks\openjdk-20.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community
Appointment set for patient Ryan Wick with doctor Dr. Smith on 2023-08-01

Process finished with exit code 0
```

<u>1.3.4 Explanation of the code</u>

In this code, we have two classes, `Patient` and `Doctor`. The `Patient` class has a constructor that takes a `name` parameter and a `makeAppointment()` method. This method takes two parameters, an instance of `Doctor` and a `date`. This method is where the Patient sends a message to the Doctor object.

The `Doctor` class has a constructor that takes a `name` parameter and a `setAppointment()` method. This method takes two parameters, an instance of `Patient` and a `date`. The `setAppointment()` method is the Doctor's response to the message sent by the Patient object.

In the `main()` method, we create a new `Patient` object and a new `Doctor` object. We then call the `makeAppointment()` method on the `Patient` object, passing the `Doctor` object and a date as arguments. This is an example of inter-object communication. The `Patient` object is sending a **message** to the `Doctor` object by calling the `setAppointment()` method on it.

The output of the above code will be: `Appointment set for patient Ryan Wick with doctor Dr. Smith on 2023-08-01`.

1.4 Encapsulation

1.4.1 Understanding Encapsulation

Encapsulation is a key principle of object-oriented programming (OOP) and it is used to bundle the data (variables) and methods (functions) that operate on the data into a single unit known as a class. This concept of wrapping data and methods together into a single unit is also known as data hiding. It means that the data of a class is hidden from other classes, and can be accessed only through the methods of their current class.

Encapsulation provides a way to protect data from accidental corruption and it is the mechanism that allows us to control the accessibility of the components of an object. The main purpose of encapsulation is to provide a way to implement abstraction.

In practical terms, encapsulation is often accomplished using classes (or structures) with private data and public methods, which are the only way to access the private data. Encapsulation enables an object to be seen as a "black box" that can perform a task without the need to understand the internal workings of the task.

1.4.2 Demonstration of Encapsulation in the FDC Scenario

For the Family Dental Care (FDC) system, consider the `Patient` class as a good candidate to demonstrate the concept of encapsulation.

The `Patient` class may have private attributes such as `name`, `address`, `nationalId`, and `contactNumber`. These attributes are the data of the `Patient` object and are hidden from other classes, meaning they can't be directly accessed.

To access or modify these private attributes, we should use public methods. For example, `getName()`, `getAddress()`, `getNationalId()`, and `getContactNumber()` can be used to access the values of these attributes, while `setName(String name)`, `setAddress(String address)`, `setNationalId(String nationalId)`, and `setContactNumber(String contactNumber)` can be used to modify the values. These methods are often called getters and setters.

This is a demonstration of encapsulation, as the `Patient` class's actual data (the private attributes) are encapsulated within the class, and only accessible and modifiable through the class's public methods. Any other parts of the program, or other classes, would interact with a `Patient` object through these public methods, without needing to understand the internal workings of the class.

1.4.3 Encapsulation in Code

Here's a simple example of a `Patient` class in Java, demonstrating the principle of encapsulation:

```
public class Patient {
    // Private attributes
    private String name;
    private String address;
    private String nationalId;
    private String contactNumber;

    // Public constructor
    public Patient(String name, String address, String nationalId,
String contactNumber) {
        this.name = name;
        this.address = address;
        this.nationalId = nationalId;
        this.contactNumber = contactNumber;
    }

    // Public getter methods
    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public String getNationalId() {
        return nationalId;
    }

    public String getContactNumber() {
        return contactNumber;
    }

    // Public setter methods
```

```java
    public void setName(String name) {
        this.name = name;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public void setNationalId(String nationalId) {
        this.nationalId = nationalId;
    }

    public void setContactNumber(String contactNumber) {
        this.contactNumber = contactNumber;
    }
}
```

1.4.4 Explanation of the Code

In the above `Patient` class, the attributes `name`, `address`, `nationalId`, and `contactNumber` are declared as `private`. This means they can only be accessed and modified within this class, thus ensuring the data is safe from accidental modification.

The `Patient` class includes a constructor that takes four parameters - `name`, `address`, `nationalId`, and `contactNumber`. This is a public method used to create a new `Patient` object with initial values for the private attributes.

To provide access to these private attributes, public getter methods (e.g., `getName()`, `getAddress()`, `getNationalId()`, `getContactNumber()`) are used. These methods simply return the value of the corresponding attribute.

To allow the values of these private attributes to be changed, public setter methods (e.g., `setName(String name)`, `setAddress(String address)`, `setNationalId(String nationalId)`, `setContactNumber(String contactNumber)`) are used. Each of these methods takes a parameter and assigns its value to the corresponding private attribute.

This way, the `Patient` class maintains control over its own data, only allowing it to be accessed or changed through its own methods. Other parts of the program, or other classes, would interact with a `Patient` object through these public methods, without needing to understand the internal workings of the class. This is a demonstration of the encapsulation principle of object-oriented programming.

1.5 Inheritance

1.5.1 Understanding Inheritance

Inheritance is a fundamental principle in Object-Oriented Programming (OOP). It refers to the ability of one class (child class or subclass) to inherit properties (attributes) and behaviors (methods) from another class (parent class or superclass). This principle promotes code reusability, as common properties and behaviors only need to be defined once in the parent class and can be used by any number of child classes. It also provides a mechanism for creating more specific classes from general ones, effectively creating a type hierarchy, which can lead to more organized and manageable code.

Inheritance represents an "is-a" relationship between classes. For example, if we have a `Vehicle` class and a `Car` class, we could say that a `Car` is a `Vehicle`, so the `Car` class should inherit from the `Vehicle` class.

1.5.2 Inheritance in the Family Dental Care Scenario

In the Family Dental Care (FDC) system, the concept of inheritance can be demonstrated through the relationships between several classes.

The most fundamental class in our scenario is the `Person` class, which contains attributes and methods that are common to any person in the system, such as name, address, national identity card number, and contact number. This class acts as a parent for other classes.

- The `Patient` class inherits from the `Person` class. This represents that a patient is a person but with additional attributes and behaviors unique to patients, such as a registration status.
- The `Doctor` class also inherits from the `Person` class, as a doctor is a person with additional attributes and behaviors, such as years of experience.

In our hierarchy, the `Doctor` class itself is a parent class to `Consultant` and `Dentist` classes, making these subclasses of `Doctor`. This implies that a `Consultant` and a `Dentist` are both doctors but with distinct attributes and behaviors. For instance, a `Consultant` possesses a post-graduate qualification and the country of the university that granted it, while a `Dentist` indicates the number of years of experience.

- The `Staff` class also inherits from the `Person` class. Staff members are persons with additional attributes and behaviors unique to the staff of FDC. The `Staff` class is a parent class to the `Manager`, `Nurse`, and `Receptionist` classes. This suggests that a manager, a nurse, and a receptionist are all staff members with specific roles.
- The `Room` class is a parent to the `ScanRoom` class, indicating that a `ScanRoom` is a type of `Room` with additional behaviors or attributes unique to it.

This design reduces code redundancy by utilizing common attributes and methods from parent classes and ensures that each class only contains data and logic that are appropriate to it, enhancing code maintainability and clarity.

1.5.3 Implementing Parent and Child Classes in Java

Java provides a straightforward and effective means to implement inheritance through the `extends` keyword. For this example, let's focus on the `Person` class and one of its child classes, the `Patient` class.

The Code:

Let's start with the parent class `Person`. Here's a simple representation:

```java
public class Person {
    // Private attributes
    private String name;
    private String address;
    private String idCardNumber;
    private String contactNumber;

    // Constructor
    public Person(String name, String address, String idCardNumber,
String contactNumber) {
        this.name = name;
        this.address = address;
        this.idCardNumber = idCardNumber;
        this.contactNumber = contactNumber;
    }

    // Getter methods
    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public String getIdCardNumber() {
        return idCardNumber;
    }

    public String getContactNumber() {
        return contactNumber;
    }
}
```

Now let's create the `Patient` class, which inherits from `Person`:

```
public class Patient extends Person {
    // Additional attribute unique to Patient
    private boolean isRegistered;

    // Constructor
    public Patient(String name, String address, String idCardNumber,
String contactNumber, boolean isRegistered) {
        // Call to the parent's constructor
        super(name, address, idCardNumber, contactNumber);
        this.isRegistered = isRegistered;
    }

    // Getter method for isRegistered
    public boolean getIsRegistered() {
        return isRegistered;
    }

    // Setter method for isRegistered
    public void setIsRegistered(boolean isRegistered) {
        this.isRegistered = isRegistered;
    }
}
```

1.5.4 Explanation of the Code

In the code above, the `Person` class is defined with attributes that are common to all persons (name, address, idCardNumber, contactNumber). Each of these attributes is private, meaning they can only be accessed directly within the `Person` class. Public getter methods are provided to access these attributes, as is common in Java to maintain the principle of encapsulation.

The `Patient` class, on the other hand, is defined with the keyword `extends`, followed by `Person`. This indicates that `Patient` is a subclass of `Person` and will inherit all of `Person`'s attributes and methods. The `Patient` class also has an additional attribute unique to it, `isRegistered`, which indicates whether the patient is registered.

In the `Patient` class's constructor, the `super` keyword is used to call the parent class's constructor, passing the necessary parameters. This ensures that the `Person` part of the `Patient` object is properly initialized. The unique attribute `isRegistered` is then initialized.

Finally, getter and setter methods for `isRegistered` are provided. The getter allows us to check the registration status of a patient, and the setter allows us to change it.

In this way, the `Patient` class successfully extends the `Person` class, demonstrating the concept of inheritance. This allows the `Patient` class to reuse code from the `Person` class, promoting code efficiency and organization.

1.5.5 Inheritance: UML Diagram of Class Hierarchy

Creating a UML (Unified Modeling Language) diagram helps visualize the structure of our system and the relationships between classes. The diagram can show inheritance (using a generalization arrow), along with the classes' attributes and methods. Below I've created a simple textual representation of a UML diagram for the `Person` and `Patient` classes based on the code written above.

UML Diagram:

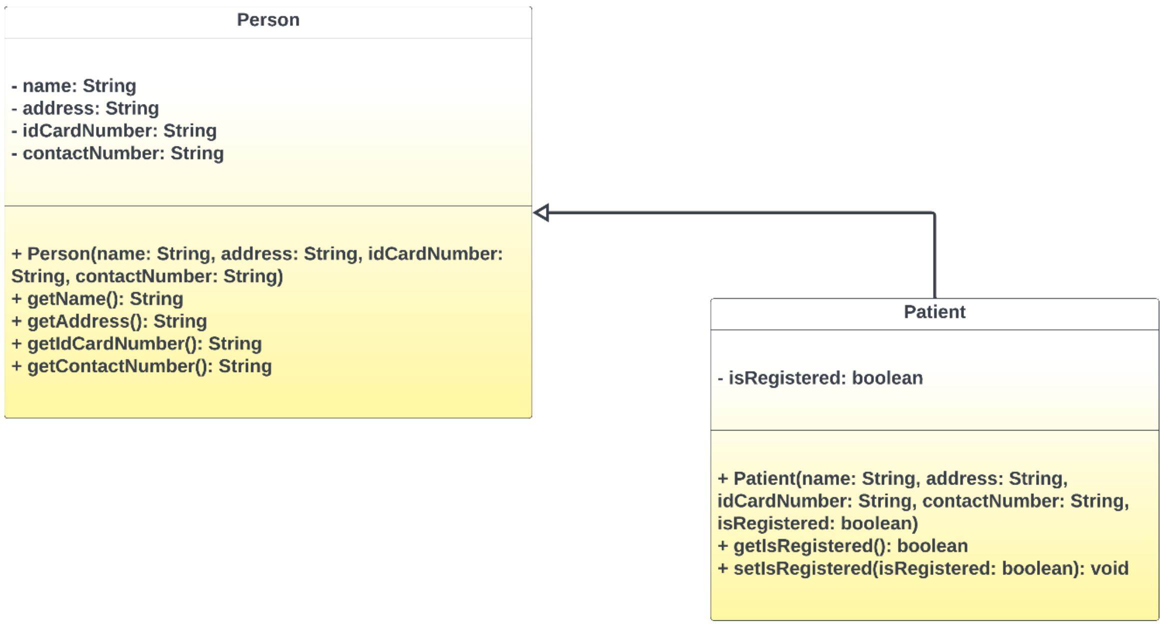

Person
- name: String
- address: String
- idCardNumber: String
- contactNumber: String
+ Person(name: String, address: String, idCardNumber: String, contactNumber: String)
+ getName(): String
+ getAddress(): String
+ getIdCardNumber(): String
+ getContactNumber(): String
Patient
- isRegistered: boolean
+ Patient(name: String, address: String, idCardNumber: String, contactNumber: String, isRegistered: boolean)
+ getIsRegistered(): boolean
+ setIsRegistered(isRegistered: boolean): void

Figure 1. 1 Inheritance: UML Diagram of Class Hierarchy

1.5.6 Explanation of the Diagram

In the UML diagram, `Person` is the superclass, and `Patient` is the subclass (or child class). This is shown with the arrow pointing from `Patient` to `Person`, indicating that `Patient` extends `Person`.

The attributes of the classes are shown below the class names. They're preceded by a minus sign (-), which means they're private.

Below the attributes, I've listed the methods for each class. They're preceded by a plus sign (+), indicating they're public. The methods include the constructors (which have the same name as the class) and the getter and setter methods.

The `Patient` class includes additional methods and attributes beyond those of `Person`, demonstrating that a subclass can extend the functionality of its superclass.

1.6 Polymorphism

1.6.1 Understanding Polymorphism

Polymorphism is a core concept in Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a parent class. The term polymorphism is derived from Greek words "poly" meaning many, and "morphos" meaning forms, hence the term 'many forms'. In programming, polymorphism allows a subclass to inherit the methods and attributes of a superclass, but also to override or add new methods, providing more specific implementations. This feature adds a lot of flexibility and dynamism to OOP languages like Java.

There are two types of polymorphism in Java:

1. Compile-time Polymorphism (or Static Polymorphism): This type of polymorphism is achieved through method overloading. In method overloading, multiple methods will have the same name but different parameters.
2. Runtime Polymorphism (or Dynamic Polymorphism): This type of polymorphism is achieved through method overriding. In this case, the subclass has the same method as declared in the parent class.

1.6.2 Identifying a Method for Overriding

Given our Family Dental Care (FDC) scenario, let's consider the `Person` class and its subclasses `Patient`, `Doctor`, and `Staff`. All these classes will have a `getName` method as they all extend the `Person` class. However, the way we represent the name might be different for each class.

For example, a `Doctor` might have their title (e.g., "Dr.") prefixed to their name, a `Staff` member might have their job role suffixed (e.g., "John - Receptionist"), while a `Patient` may just have their name displayed plainly. This presents an opportunity for the `getName` method to be overridden in each subclass to provide a more specific implementation.

To be clear, in this scenario, the `getName` method in the `Person` class can be overridden in the `Doctor`, `Patient`, and `Staff` subclasses. This is a perfect illustration of polymorphism, where a single method name, `getName`, is used to perform a related but slightly different function in each subclass.

<u>1.6.3 Code Demonstrating Polymorphism</u>

The following Java code shows the `Person` superclass and the `Patient`, `Doctor`, and `Staff` subclasses. The `getName` method in the `Person` class is overridden in each subclass to provide specific implementation:

```java
// Parent class - Person
public class Person {
    protected String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

// Child class - Patient
public class Patient extends Person {

    public Patient(String name) {
        super(name);
    }

    // Overriding getName method from Person
    @Override
    public String getName() {
        return "Patient: " + name;
    }
}

// Child class - Doctor
public class Doctor extends Person {

    public Doctor(String name) {
        super(name);
    }
    // Overriding getName method from Person
    @Override
    public String getName() {
        return "Dr. " + name;
    }
}

// Child class - Staff
public class Staff extends Person {

    public Staff(String name) {
        super(name);
    }

    // Overriding getName method from Person
    @Override
    public String getName() {
        return "Staff: " + name;
    }
}
```

1.6.4 Explanation of the Code

This code demonstrates the concept of polymorphism. The parent class, `Person`, has a field `name` and a method `getName()` that returns the name of the person.

The `Person` class is extended by three classes - `Patient`, `Doctor`, and `Staff`. Each of these subclasses represents a different type of person in the context of the Family Dental Care system.

Each subclass has a constructor that calls the `super(name)` constructor of the parent `Person` class to initialize the `name` attribute.

The `getName()` method is overridden in each subclass. For a `Patient`, the name is prefixed with "Patient: ". For a `Doctor`, the name is prefixed with "Dr. ", and for a `Staff`, the name is prefixed with "Staff: ". This method overriding, a form of polymorphism, allows us to use the same method name to perform similar but slightly different actions in each subclass.

So, when the `getName()` method is called on an object, the specific implementation of the method in the object's class will be used. This is the essence of polymorphism - the ability of different objects to respond, each in its own way, to the same message, or in this case, the same method call.

1.7 Aggregation/Composition

1.7.1 Understanding Aggregation and Composition

In the world of object-oriented programming, objects often have relationships with other objects. These relationships can be categorized in different ways, including aggregation and composition. Both of these concepts represent a part-whole relationship, but they do it in slightly different ways.

- **Aggregation:** Aggregation is a relationship where one object (the 'whole') can exist without the other (the 'part'), but the 'part' object may be part of multiple 'whole' objects. In aggregation, the 'part' object can exist independently of the 'whole' object. For example, a team (whole) can exist without a player (part), and a player can be part of multiple teams.
- **Composition:** Composition, on the other hand, is a more restrictive form of aggregation. In composition, the 'part' object cannot exist independently of the 'whole' object. If the 'whole' object is destroyed, the 'part' object is destroyed as well. An example of composition is a car (whole) and its engine (part). The engine cannot exist independently of the car, and if the car is destroyed, the engine is also destroyed.

1.7.2 Aggregation and Composition in the FDC Scenario

In the Family Dental Care (FDC) system, we can identify several instances of aggregation and composition:

Aggregation:

- An \`Appointment\` is an aggregation of a \`Doctor\` and a \`Patient\`. The \`Appointment\` cannot exist without a \`Doctor\` or a \`Patient\`, but both the \`Doctor\`

and the `Patient` can exist independently of the `Appointment`. Therefore, the relationship between an `Appointment` and a `Doctor` or `Patient` is an example of aggregation.

Composition:

- The `Doctor` class is composed of either a `Consultant` or a `Dentist`. This is an example of composition because a `Consultant` or a `Dentist` cannot exist independently of a `Doctor` – they are specific types of `Doctor`, and their existence is entirely dependent on the `Doctor` object.
- The `Staff` class is composed of a `Manager`, `Nurses`, and a `Receptionist`. These roles cannot exist independently from the `Staff` – they are specific types of `Staff`, and their existence is completely dependent on the `Staff` object.

These examples illustrate the concepts of aggregation and composition in the FDC system. By understanding these relationships, we can build a more effective and accurate object-oriented model of the system.

1.7.3 Code Demonstrating Aggregation and Composition

Let's take a look at a simplified representation of how aggregation and composition can be implemented in Java code.

```
// The Doctor class
public class Doctor {
    private String name;

    // constructor
    public Doctor(String name) {
        this.name = name;
    }

    // getter method
    public String getName() {
        return name;
    }
}
```

```
// The Patient class
public class Patient {
    private String name;

    // constructor
    public Patient(String name) {
        this.name = name;
    }

    // getter method
    public String getName() {
        return name;
    }
}

// The Appointment class representing aggregation
public class Appointment {
    private Doctor doctor; // The Doctor object is part of the
Appointment
    private Patient patient; // The Patient object is part of the
Appointment

    // constructor
    public Appointment(Doctor doctor, Patient patient) {
        this.doctor = doctor;
        this.patient = patient;
    }

    // getter methods
    public Doctor getDoctor() {
        return doctor;
    }

    public Patient getPatient() {
        return patient;
    }
}

// The Staff class
public class Staff {
    private String name;

    // constructor
    public Staff(String name) {
        this.name = name;
    }

    // getter method
    public String getName() {
        return name;
    }
}

// The Manager class representing composition
public class Manager {
    private Staff staff; // The Staff object is part of the Manager

    // constructor
    public Manager(Staff staff) {
        this.staff = staff;
    }

    // getter method
    public Staff getStaff() {
        return staff;
    }
}
```

1.7.4 Explanation of the Code

The above code demonstrates both aggregation and composition within the context of the Family Dental Care system.

In the `Appointment` class, we see an example of **aggregation**. The `Appointment` class has two member variables: a `Doctor` object and a `Patient` object. In the constructor, we see that an `Appointment` object is created by passing in a `Doctor` and a `Patient` object. This represents a has-a relationship – an `Appointment` has a `Doctor` and a `Patient`. However, the `Doctor` and `Patient` objects can exist independently of the `Appointment` object. This means that if the `Appointment` object is destroyed, the `Doctor` and `Patient` objects will not be destroyed.

In the `Manager` class, we see an example of **composition**. The `Manager` class has one member variable: a `Staff` object. In the constructor, we see that a `Manager` object is created by passing in a `Staff` object. This represents a part-of relationship – a `Manager` is part of the `Staff`. In this case, the `Staff` object cannot exist independently of the `Manager` object. This means that if the `Manager` object is destroyed, the `Staff` object will also be destroyed.

These two examples clearly illustrate the difference between aggregation and composition in object-oriented programming. Aggregation represents a has-a relationship where the 'part' can exist independently of the 'whole', while composition represents a part-of relationship where the 'part' cannot exist independently of the 'whole'.

1.8 Class Relationships in Object-Oriented Design

Delving into the various class relationships is integral to understanding the nuances of object-oriented design and development. These relationships dictate how classes interact with each other, allowing for the creation of robust and modular systems. Here is an examination of the primary class relationships in the object-oriented paradigm:

1.8.1 Association

Association represents a “using” relationship between two or more objects. It indicates that one class makes reference to another class. An association can be bi-directional, where both classes are aware of each other, or uni-directional, where only one class is aware of the other. Multiplicity, which is often added to associations, indicates how many instances of one class relate to a single instance of the other class. For example, a `Doctor` class might have an association with a `Patient` class, denoting that a doctor can have many patients, and vice-versa.

Think of a `University` and `Student` relationship. A university can have numerous students enrolled, and each student can be associated with one specific university. This represents a one-to-many association.

```
class University {
    // A list containing students enrolled in the university.
    private List<Student> students;

    // Function to add a student to the university's list.
    void addStudent(Student student) {
        this.students.add(student);
    }
}

class Student {
    // A reference to the university to which the student is enrolled.
    private University university;

    // Function to set the university for the student.
    void setUniversity(University uni) {
        this.university = uni;
    }
}
```

1.8.2 Aggregation

Aggregation is a specialized form of association, representing a "whole-part" relationship. It signifies that one class (the whole) contains objects of another class (the parts) but doesn't necessarily dictate their lifetime. An aggregate object can exist independently of its parts. An example can be a `Library` class containing an aggregation of `Books`. If the library ceases to exist, the books might still remain.

A `Library` and `Book` relationship demonstrates below code. A library houses multiple books, but if the library ceases to exist, the books still remain.

```
class Library {
    // List containing books present in the library.
    private List<Book> books;

    // Function to add a book to the library's collection.
    void addBook(Book book) {
        this.books.add(book);
    }
}

class Book {
    // Attributes like title and author.
    String title;
    String author;
}
```

1.8.3 Composition

Composition is a stricter form of aggregation. Like aggregation, it represents a whole-part relationship. However, in composition, the lifetimes of the part and the whole are tightly coupled. When the whole is destroyed, so are its parts. For instance, a `Human` class might have a composition relationship with a `Heart` class. If the human object is destroyed, the heart object will be too.

Examine a `Computer` and `CPU` relationship. A computer contains a CPU. If the computer is discarded, the CPU also gets discarded.

```
class Computer {
    // CPU is an integral part of the computer.
    private CPU cpu;

    Computer() {
        // CPU is instantiated when the computer is created.
        this.cpu = new CPU();
    }
}

class CPU {
    // Attributes such as clock speed and brand.
    double clockSpeed;
    String brand;
}
```

1.8.4 Inheritance (or Generalization)

This relationship establishes a hierarchy between a parent (or base) class and a child (or derived) class. A child class inherits attributes and behaviors from the parent class, which means it can inherit its data and methods. This promotes code reuse and establishes a relationship based on the principle that the derived class "is a type of" the base class. For example, a `Bird` class might be a parent class, and `Eagle` or `Penguin` might be child classes that inherit from it.

Take a base class `Vehicle` and a subclass `Car`. A car "is-a" type of vehicle and thus inherits properties from the `Vehicle` class.

```
class Vehicle {
    // Common attributes for all vehicles.
    int speed;
    int wheels;
}

class Car extends Vehicle {
    // Specific attribute for cars.
    int numberOfDoors;
}
```

1.8.5 Dependency

Dependency indicates that one class depends on another, meaning that a change in one class might force a change in the other. However, this relationship is more transient than others, often existing for a very short duration - like the duration of a method operation. For example, a `PaymentProcessing` class might have a dependency on an `Invoice` class if, within a method, it utilizes the `Invoice` class for generating a bill.

Below code reflect on a `PaymentGateway` class and an `Order` class. The `Order` class needs the `PaymentGateway` to process payments but isn't perpetually tied to it.

```
class Order {
    // Processes payment through the passed payment gateway.
    void processPayment(PaymentGateway gateway) {
        gateway.pay();
    }
}

class PaymentGateway {
    // Processes the payment.
    void pay() {
        // Logic to process the payment.
    }
}
```

1.8.6 Realization

This relationship exists between classes and interfaces. A class realizes an interface by implementing its methods. Realization is a way to describe contracts in object-oriented systems. The class that realizes the interface agrees to fulfill the contract presented by the interface.

Take an interface `Drawable` and a class `Circle` that implements this interface. The `Circle` class provides a concrete method to draw, as declared in the `Drawable` interface.

```
interface Drawable {
    // Declares a contract to draw.
    void draw();
}

class Circle implements Drawable {
    @Override
    void draw() {
        // Provides a concrete implementation of the draw method.
    }
}
```

In conclusion, the relationships between classes form the backbone of object-oriented design, establishing clear interactions and hierarchies between classes. Whether it's about generalization through inheritance, modular designs via composition, or establishing contracts through realization, these relationships help in creating systems that are modular, scalable, and maintainable. Understanding these relationships is crucial for any software developer working within the object-oriented paradigm, as they dictate the structure and interactivity of the system components.

Task 2

2.1 Understand the Problem Scenario

2.1.1 Analysis of the Family Dental Care System Requirements

Introduction:

Family Dental Care (FDC) is an upscale dental clinic located in Kandy, offering a range of dental services. The clinic collaborates with distinguished dentists and dental consultants, some affiliated with the Faculty of Dental Science at the University of Peradeniya. With the aim of streamlining and enhancing its operations, FDC wishes to implement an information system, using an object-oriented approach, to manage its patients, doctors, appointments, treatments, and payments, along with its staff details.

Patient Management:

- Registration: All first-time patients need to register by providing personal details including name, address, national ID, and contact number. Registration involves a nominal fee.
- Appointments: Treatments are administered based on appointments, allowing patients to consult doctors at a scheduled time.
- Treatments and Payments: FDC offers a wide range of dental treatments, each incurring a separate fee. The system needs to record the treatments provided to each patient and track the associated payments.

Doctor Management:

- Registration: Both dentists and dental consultants are required to register at FDC. The registration process involves collecting personal details such as name, address, date of birth, national ID, and contact number.
    - Dentists: Must provide their years of experience during registration.
    - Consultants: Need to provide details about their post-graduate qualifications, including the name of the qualification and the country of the university that awarded it.

Facility Details:

- Surgery Rooms: FDC houses four fully equipped surgery rooms, each capable of accommodating one patient at a time.
- Dental Scan Room: This special room, which can accommodate one patient at a time, is equipped with a dental scan machine operated by one of FDC's dentists. The system should allow the manager to assign available doctors to the scan room based on their free slots.

Staff Management:

- Composition: The staff at FDC comprises a manager, four nurses (each designated to one of the four surgery rooms), and a receptionist responsible for managing registrations and appointments.
- Roles and Responsibilities:
    - Manager: Assigns available doctors to the dental scan room based on their availability.
    - Nurses: Each nurse is responsible for one surgery room.
    - Receptionist: Handles patient registrations and appointment bookings.

System Requirements:

- Object-Oriented Design: The design and implementation of the system should leverage the principles of object-oriented programming.
- Tracking: The system should keep a log of patients, doctors, appointments, treatments provided to patients, and payments.
- Staff Information: A comprehensive database of staff, including their roles and responsibilities, should be maintained.

Relationships in the System:

- Inheritance: The system uses inheritance to distinguish between different roles like patients, doctors (further divided into dentists and consultants), and staff members.
- Association: Several classes have basic relationships, like the association between patients and appointments or doctors and scan rooms.
- Aggregation: This relationship is evident in scenarios like a doctor having a list of their appointments.
- Composition: This strong form of aggregation can be seen in scenarios like a patient having a registration record, indicating a strong ownership.

2.1.2 Overview of Family Dental Care UML Class Diagram

The UML Class Diagram for the Family Dental Care (FDC) provides a detailed structural representation of the system's classes, their attributes, methods, and the relationships among them. This section will provide an overview of the main classes and their interconnections, laying out the framework for how FDC will function.

Core Classes:

- Person: This is the base class from which several other classes inherit common attributes and methods related to personal details like name, address, contact information, and date of birth.
- Patient: Derived from the `Person` class, it represents the individuals receiving treatments at FDC. They have a unique registration attribute and are associated with appointments and payments.
- Doctor: Also a subclass of `Person`, doctors can be general dentists or specialized consultants. They are integral to the treatment process and are associated with appointments. Doctors also have a unique registration and can be associated with the scan room.
- Consultant: A specialized form of doctor with added attributes pertaining to their postgraduate qualifications and the university that awarded it.
- Dentist: Represents general practitioners without the specialization of consultants.
- Staff: Derived from the `Person` class, this represents the non-medical employees of FDC, including the manager, nurses, and receptionists. They have a distinct position within the organization.

Operational Classes:

- Appointment: This class embodies the scheduled meetings between doctors and patients. Appointments are linked with specific doctors, patients, treatments, and appointment slots.

- Treatment: Represents the various dental procedures offered by FDC. Each treatment has its own cost and description.
- Payment: Captures the financial transactions at FDC, including the amount and the date of payment.
- Registration: Every doctor and patient must register before availing or providing services. This class holds details about the registration, including the associated fee.
- AppointmentSlot: Represents specific time periods reserved for appointments. Each slot is connected to a doctor and a specific surgery room.

Infrastructure Classes:

- SurgeryRoom: Represents the physical spaces where treatments take place. Each room has a unique number and can be associated with a nurse.
- ScanRoom: A specialized form of surgery room dedicated to dental scans. This room can be associated with a doctor, and the assignment of doctors to this room is overseen by the manager.

2.1.3 Relationships Overview

Inheritance:

The classes `Patient`, `Doctor`, `Staff`, `Consultant`, and `Dentist` all inherit attributes and methods from the `Person` class. Similarly, `Manager`, `Nurse`, and `Receptionist` inherit from `Staff`, while `Consultant` and `Dentist` inherit from `Doctor`. Lastly, `ScanRoom` inherits from `SurgeryRoom`.

Association:

Many classes are interconnected via associations. For example:

- Doctors are associated with appointments and the scan room.
- Patients are linked with appointments and payments.
- The manager oversees the scan room's operations.
- Nurses are associated with specific surgery rooms.

Aggregation:

Some classes form part-whole relationships, indicating a weaker form of ownership. Examples include:

- The relationship between `Appointment` and `Doctor`, `Patient`, and `AppointmentSlot`.
- The link between `AppointmentSlot` and `Doctor` and `SurgeryRoom`.

Composition:

Indicating a strong form of ownership, examples of this relationship include the link between `Patient` and `Registration` and the association between `Doctor` and `Registration`.

2.2 Develop a Conceptual Class Diagram

<u>2.2.1 Conceptual Class Diagram Structure</u>

Basic Entities:

`Person` Class → This is the foundational class that encapsulates the general attributes of a person: their name, address, national ID, and contact number. It also provides the corresponding methods to set and get these attributes. Various roles in the hospital, like patients, doctors, and staff, will inherit from this class, ensuring that they all share these foundational attributes.

Patient-related Entities:

`Patient` Class → The `Patient` class represents individuals who seek medical attention. Apart from the basic attributes inherited from the `Person` class, it has an attribute to determine if the patient is registered. A patient can have multiple appointments and can make various payments, hence the associations to the `Appointment` and `Payment` classes.

`Appointment` Class → Each appointment has a date and is associated with a specific doctor and patient. Appointments are central to the functioning of the hospital as they dictate when a patient sees a doctor. Each appointment can also have an associated treatment.

`Treatment` Class → Treatments represent medical procedures or advice given during an appointment. Each treatment has a description and a cost.

`Payment` Class → Payments represent the transactions made by patients for the services they receive. Each payment has an associated amount and date.

Doctor-related Entities:

\`Doctor\` Class → Doctors are specialized persons who offer medical advice and treatments to patients. Apart from the basic attributes, doctors have experience in years. They can be associated with various appointments and can operate in scan rooms.

Specialized Doctors → \`Consultant\` Class: Consultants are specialized doctors. Apart from having the attributes of a general doctor, they possess postgraduate qualifications from specific universities in certain countries.

\`Dentist\` Class: Dentists are doctors specialized in dental care. In this diagram, they don't have any unique attributes or methods, but their presence implies specialized dental treatments.

Other Hospital Staff:

\`Staff\` Class → Staff is a generalized class that encompasses all the non-doctor employees of the hospital. They have a specific position in the hospital, which is an attribute of this class.

Specific Staff Members → \`Manager\` Class: Managers oversee certain operations and facilities. They specifically have an association with scan rooms, perhaps indicating their role in managing these specialized rooms.

\`Nurse\` Class: Nurses are vital in any healthcare system. Here, they have an association with rooms, which may imply that they are in charge of or operate within specific rooms.

\`Receptionist\` Class: While no unique attributes or methods are given, the presence of receptionists indicates their role in front-desk operations, appointment scheduling, and patient interactions.

Hospital Infrastructure:

\`Room\` Class → Rooms are fundamental units of a hospital's infrastructure. Each room has a unique room number. Nurses have associations with these rooms.

`ScanRoom` Class → A specialized type of room where scans (like X-rays or MRIs) take place. Doctors and managers have specific associations with these rooms, indicating their role in operating or managing them.

This UML diagram effectively captures the essence of a healthcare system, detailing out various roles, their attributes, the services offered, and the infrastructure. The relationships provide insights into the interactions and dependencies among these entities, helping in understanding the workflow and operations of such an institution.

2.2.2 Family Dental Care Conceptual UML Class Diagram

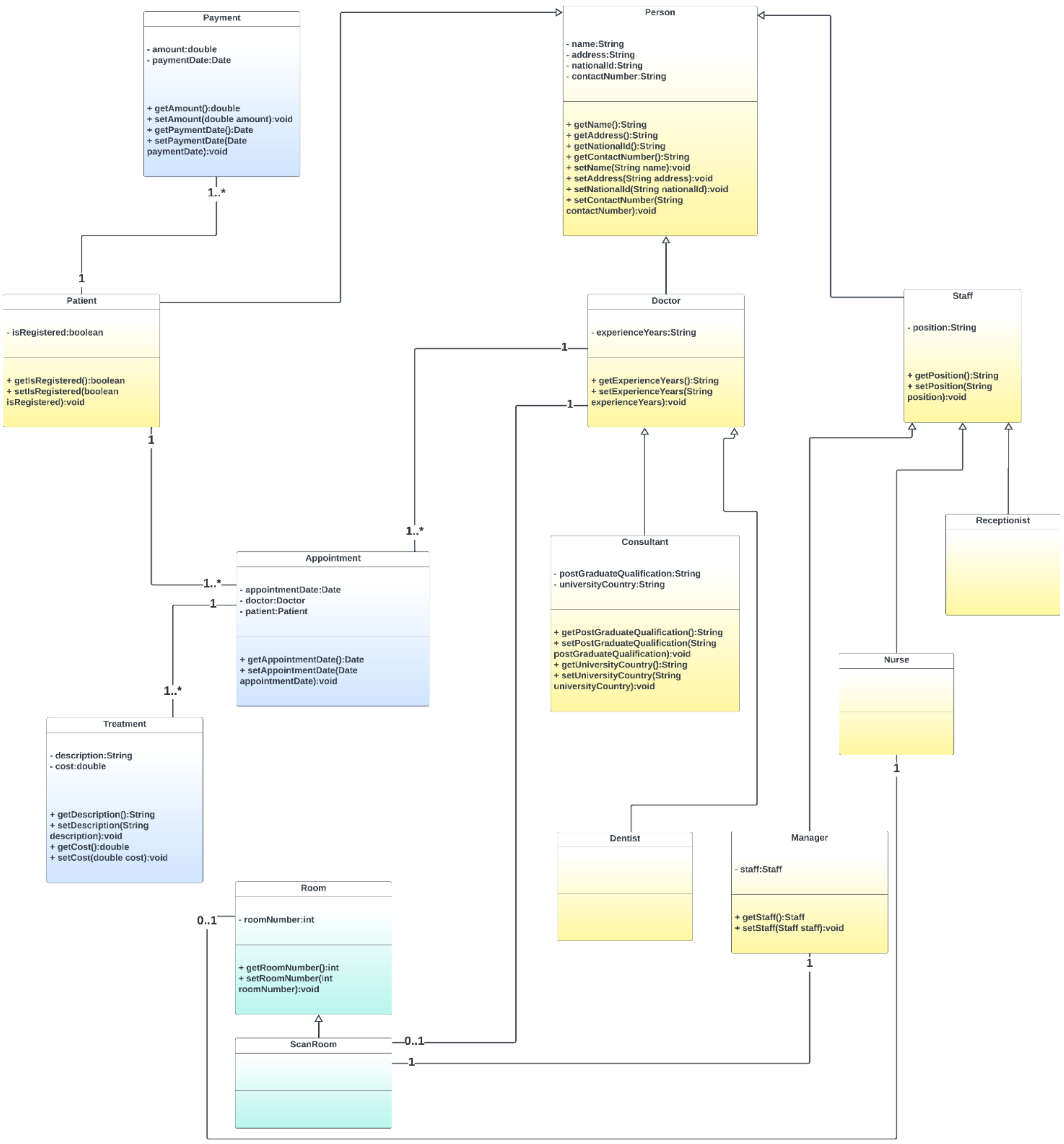

Payment
- amount:double
- paymentDate:Date
+ getAmount():double
+ setAmount(double amount):void
+ getPaymentDate():Date
+ setPaymentDate(Date paymentDate):void
1..*
1
Person
- name:String
- address:String
- nationalId:String
- contactNumber:String
+ getName():String
+ getAddress():String
+ getNationalId():String
+ getContactNumber():String
+ setName(String name):void
+ setAddress(String address):void
+ setNationalId(String nationalId):void
+ setContactNumber(String contactNumber):void
Patient
- isRegistered:boolean
+ getIsRegistered():boolean
+ setIsRegistered(boolean isRegistered):void
Doctor
- experienceYears:String
+ getExperienceYears():String
+ setExperienceYears(String experienceYears):void
Staff
- position:String
+ getPosition():String
+ setPosition(String position):void
Appointment
- appointmentDate:Date
- doctor:Doctor
- patient:Patient
+ getAppointmentDate():Date
+ setAppointmentDate(Date appointmentDate):void
Consultant
- postGraduateQualification:String
- universityCountry:String
+ getPostGraduateQualification():String
+ setPostGraduateQualification(String postGraduateQualification):void
+ getUniversityCountry():String
+ setUniversityCountry(String universityCountry):void
Receptionist
Nurse
Treatment
- description:String
- cost:double
+ getDescription():String
+ setDescription(String description):void
+ getCost():double
+ setCost(double cost):void
Dentist
Manager
- staff:Staff
+ getStaff():Staff
+ setStaff(Staff staff):void
Room
- roomNumber:int
+ getRoomNumber():int
+ setRoomNumber(int roomNumber):void
ScanRoom
1..*
1
1..*
1
1
1..*
1..*
0..1
0..1
1
1
1

Figure 2. 1 Family Dental Care Conceptual UML Class Diagram

2.3 Design the Detailed UML Class Diagram

2.3.1 Family Dental Care Detailed UML Class Diagram

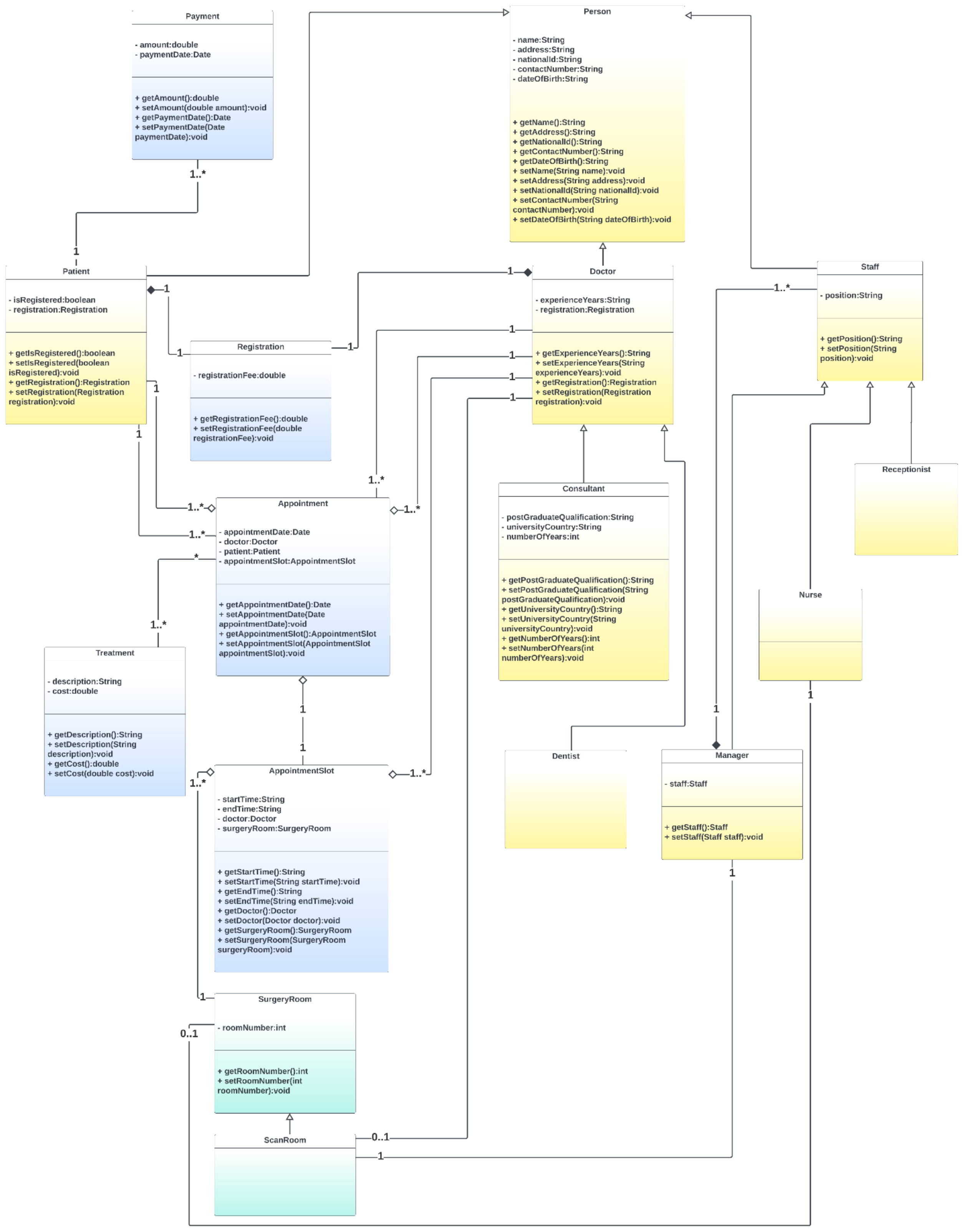

Payment
- amount:double
- paymentDate:Date
+ getAmount():double
+ setAmount(double amount):void
+ getPaymentDate():Date
+ setPaymentDate(Date paymentDate):void
Person
- name:String
- address:String
- nationalId:String
- contactNumber:String
- dateOfBirth:String
+ getName():String
+ getAddress():String
+ getNationalId():String
+ getContactNumber():String
+ getDateOfBirth():String
+ setName(String name):void
+ setAddress(String address):void
+ setNationalId(String nationalId):void
+ setContactNumber(String contactNumber):void
+ setDateOfBirth(String dateOfBirth):void
Patient
- isRegistered:boolean
- registration:Registration
+ getIsRegistered():boolean
+ setIsRegistered(boolean isRegistered):void
+ getRegistration():Registration
+ setRegistration(Registration registration):void
Registration
- registrationFee:double
+ getRegistrationFee():double
+ setRegistrationFee(double registrationFee):void
Doctor
- experienceYears:String
- registration:Registration
+ getExperienceYears():String
+ setExperienceYears(String experienceYears):void
+ getRegistration():Registration
+ setRegistration(Registration registration):void
Staff
- position:String
+ getPosition():String
+ setPosition(String position):void
Receptionist
Appointment
- appointmentDate:Date
- doctor:Doctor
- patient:Patient
- appointmentSlot:AppointmentSlot
+ getAppointmentDate():Date
+ setAppointmentDate(Date appointmentDate):void
+ getAppointmentSlot():AppointmentSlot
+ setAppointmentSlot(AppointmentSlot appointmentSlot):void
Consultant
- postGraduateQualification:String
- universityCountry:String
- numberOfYears:int
+ getPostGraduateQualification():String
+ setPostGraduateQualification(String postGraduateQualification):void
+ getUniversityCountry():String
+ setUniversityCountry(String universityCountry):void
+ getNumberOfYears():int
+ setNumberOfYears(int numberOfYears):void
Nurse
Treatment
- description:String
- cost:double
+ getDescription():String
+ setDescription(String description):void
+ getCost():double
+ setCost(double cost):void
AppointmentSlot
- startTime:String
- endTime:String
- doctor:Doctor
- surgeryRoom:SurgeryRoom
+ getStartTime():String
+ setStartTime(String startTime):void
+ getEndTime():String
+ setEndTime(String endTime):void
+ getDoctor():Doctor
+ setDoctor(Doctor doctor):void
+ getSurgeryRoom():SurgeryRoom
+ setSurgeryRoom(SurgeryRoom surgeryRoom):void
Dentist
Manager
- staff:Staff
+ getStaff():Staff
+ setStaff(Staff staff):void
SurgeryRoom
- roomNumber:int
+ getRoomNumber():int
+ setRoomNumber(int roomNumber):void
ScanRoom
1..*
1
1
1
1
1
1
1
1
1..*
1
1
1..*
1..*
1..*
*
1..*
1
1
1..*
1..*
1
1
1
1
0..1
0..1
1

Figure 2. 2 Family Dental Care Detailed UML Class Diagram

2.3.2 Detailed UML Class Diagram Inter-Class Relationships and their Multiplicities

Inheritance Relationships:

Inheritance, commonly known as the "is-a" relationship, allows classes to inherit attributes and methods from a parent class.

- Person and its Subclasses:
    - `Patient`, `Doctor`, and `Staff` inherit from the `Person` class. This is because they all share common attributes of a person but have additional specialized attributes and behaviors.
- Doctor and its Subclasses:
    - Both `Consultant` and `Dentist` inherit from the `Doctor` class. This indicates that while both consultants and dentists are types of doctors, they may have specialized attributes and behaviors unique to their roles.
- Room and its Subclasses:
    - `ScanRoom` inherits from `SurgeryRoom`. This implies that a `ScanRoom` has functionalities of a `SurgeryRoom` but with some additional specific features or capabilities.

Association Relationships:

Association represents a bi-directional relationship between two classes, where each class is aware of the other.

- Patient and Appointment:
    - One `Patient` can have multiple `Appointments` (1..*). This is because a patient may need to visit the hospital multiple times.
    - Each `Appointment` is tied to one `Patient` (1), ensuring that each appointment is associated with a specific individual.
- Patient and Payment:
    - One `Patient` can make multiple `Payments` (1..*), indicating that a patient might make payments for multiple services or visits.

- Each `Payment` is made by one `Patient` (1), ensuring accountability.
- Doctor and Appointment:
  - A `Doctor` can have multiple `Appointments` (1..*), as doctors typically see more than one patient a day.
  - Each `Appointment` has one `Doctor` (1), ensuring specificity in the medical service.
- Doctor and ScanRoom:
  - A `Doctor` can use a `ScanRoom` based on availability (0..1). Not every doctor may need a scan room, hence the possibility of zero.
  - A `ScanRoom` can be used by one `Doctor` at a time (1), ensuring that the room isn't double-booked.
- Appointment and Treatment:
  - An `Appointment` can have multiple `Treatments` (1..*), as a patient may undergo multiple treatments in a single visit.
  - A `Treatment` can be applied in multiple `Appointments` (*), since a treatment type (like an X-ray) could be given to many patients.
- Manager and ScanRoom:
  - A `Manager` oversees one `ScanRoom` (1), implying each scan room has a specific manager in charge.
  - A `ScanRoom` is overseen by one `Manager` (1), indicating the managerial responsibility.
- Nurse and SurgeryRoom:
  - A `Nurse` can be assigned to a `SurgeryRoom` (0..1), considering not all nurses might be assigned to surgery rooms.
  - A `SurgeryRoom` is attended by one `Nurse` (1) to ensure proper patient care.

Aggregation Relationships:

Aggregation is a form of association that represents a whole-part relationship. It shows that one class is composed of other classes but can exist independently.

- Appointment and Doctor:
    - An `Appointment` involves one `Doctor` (1). This is because each appointment is booked for a particular doctor.
    - A `Doctor` can be involved in multiple `Appointments` (1..*), as a doctor can have many appointments in a day.
- Appointment and Patient:
    - An `Appointment` is made by one `Patient` (1). Each appointment is made by a specific patient.
    - A `Patient` can have multiple `Appointments` (1..*), as a patient may need to book several appointments over a period.
- Appointment and AppointmentSlot:
    - An `Appointment` is allocated one `AppointmentSlot` (1). This represents the time slot for which the appointment has been booked.
    - An `AppointmentSlot` can have one `Appointment` (0..1). Not every slot may be booked, hence there might be some slots without appointments.
- AppointmentSlot and Doctor:
    - An `AppointmentSlot` is assigned to one `Doctor` (1), indicating that this specific time slot is available with this particular doctor.
    - A `Doctor` can have multiple `AppointmentSlots` (1..*), as a doctor's day can be divided into multiple slots.
- AppointmentSlot and SurgeryRoom:
    - An `AppointmentSlot` can be associated with one `SurgeryRoom` (0..1). This represents that a particular time slot may be reserved for a surgery room. Not every slot might be associated with a surgery room, hence the possibility of zero.
    - A `SurgeryRoom` can be associated with multiple `AppointmentSlots` (1..*). This means a surgery room can be booked for different appointments at different times of the day.

Composition Relationships:

Composition is a stricter form of aggregation, implying that the parts cannot exist without the whole.

- Patient and Registration:
    - Each `Patient` has one `Registration` (1), ensuring every patient is registered.
    - Each `Registration` strictly belongs to one `Patient` (1), maintaining a one-to-one record.
- Doctor and Registration:
    - Each `Doctor` has one `Registration` (1), indicating their official registration in the hospital system.
    - Each `Registration` strictly belongs to one `Doctor` (1), ensuring unique identification.
- Manager and Staff:
    - A `Manager` manages multiple `Staff` members (1..*), as managers typically have several staff members reporting to them.
    - Each `Staff` member is strictly managed by one `Manager` (1), signifying a clear hierarchical structure.

2.3.3 Changes between the conceptual and detailed version of the UML class diagram

The modifications between the simplified conceptual version of the UML class diagram and the detailed one for Family Dental Care (FDC) can be seen in the granularity and detailed attributes, methods, and relationships in each class.

Person Class:

- Modified: The `address`, `nationalId`, and `dateOfBirth` attributes are added. Relevant getters and setters for these attributes are also introduced.
- Reason: Adding these attributes ensures a more comprehensive profile of every person. For instance, `nationalId` is crucial for identification, and `dateOfBirth` can be essential for age-related procedures or discounts.

Patient Class:

- Modified: Introduced a `registration` attribute which indicates a composition relationship with the `Registration` class.
- Reason: This helps to establish the fact that a patient has a registration detail at FDC, aligning with the requirement that patients register on their first visit.

Doctor Class:

- Modified: Introduced a `registration` attribute (composition with `Registration` class) and removed direct inheritance of `experienceYears` attribute.
- Reason: This change emphasizes that both doctors and patients have unique registrations. The shift in the `experienceYears` attribute is because it's now specific to the `Dentist` class.

Consultant Class:

- Modified: Added attributes for `universityCountry` and `numberOfYears`.
- Reason: These details provide more information about the consultant's qualification and their experience, crucial for establishing credibility and potentially allocating specific tasks.

Appointment Class:

- Modified: Introduced the attributes `doctor`, `patient`, and `appointmentSlot` which indicate aggregation relationships.
- Reason: This change highlights the association of an appointment with a specific doctor, patient, and a time slot, which is essential for scheduling.

Payment Class:

- Modified: Added `paymentDate` attribute.
- Reason: It's important to know when a payment was made, especially for accounting and record-keeping purposes.

Staff, Manager, Nurse, and Receptionist Classes:

- Modified: Introduced more detailed relationships and attributes. For instance, the manager has a composition relationship with the `Staff` class.
- Reason: Emphasizes the hierarchy and responsibilities within the FDC staff. For example, the manager having a relationship with `ScanRoom` showcases managerial responsibilities in assigning doctors to the scan room.

SurgeryRoom and ScanRoom Classes:

- Modified: Introduced relationships with other classes. The `ScanRoom` class has associations with the `Doctor` and `Manager` classes.

- Reason: This signifies the functionality and usability of these rooms. The `ScanRoom`'s association with `Doctor` and `Manager` emphasizes the managerial task of assigning a free doctor to the scan room.

Registration Class:

- Modified: Remained almost unchanged, but the relationship is emphasized more in the modified diagram.
- Reason: Reinforces the importance of registration in the system.

AppointmentSlot Class:

- Modified: Introduced the attributes `doctor` and `surgeryRoom` which indicate aggregation relationships.
- Reason: These changes are pivotal to scheduling, ensuring that a doctor is available during a particular slot and that there's a room assigned for the procedure.

When discussing about the overall reasons for modifications the modified class diagram provides a more comprehensive view of the system with a higher level of detail, capturing more real-world complexities. And also, the modified diagram is more aligned with the scenario provided, ensuring all requirements are captured. A detailed diagram can be directly translated into code, ensuring developers have a clear understanding of the system's structure, relationships, and functions. Not only that, but also a detailed diagram also makes future modifications and extensions easier because it provides a clearer understanding of the system's architecture.

In essence, while the simplified diagram gives a high-level overview of the system, the modified version delves deep into the intricate details and relationships, offering a more accurate representation of the requirements in the scenario.

2.3.4 Evaluating the UML Class Diagram in Relation to the Family Dental Care (FDC) Scenario

The UML Class Diagram provides a comprehensive visualization of the relationships, hierarchies, and structures necessary for the development of an efficient information system for the Family Dental Care (FDC). By examining each class and its attributes, methods, and relationships, I can deduce how well the proposed system meets the requirements specified in the scenario.

Patient and Doctor Hierarchies:

Both the `Patient` and `Doctor` classes inherit attributes and methods from the base class `Person`, ensuring that common characteristics such as `name`, `address`, `nationalId`, `contactNumber`, and `dateOfBirth` are uniformly implemented. This not only reduces redundancy but also facilitates easier management of data related to all individuals interacting with the FDC.

The scenario emphasizes that the clinic caters to both regular dentists and specialized consultants. This distinction is aptly captured by having the `Consultant` and `Dentist` classes inherit from the `Doctor` class. While the `Dentist` class remains as it is, the `Consultant` class introduces new attributes, including the `postGraduateQualification` and `universityCountry`.

Comprehensive Treatment and Payment Management:

The `Treatment` and `Payment` classes ensure that all financial aspects of the FDC's operations are accounted for. The `Treatment` class, with attributes like `description` and `cost`, enables precise detailing of various treatments available. This aligns with the scenario, where a variety of treatments like extractions and dental implants are offered.

The `Payment` class, with its attributes, captures not just the amount transacted, but also the `paymentDate`, ensuring meticulous tracking of financial interactions over time.

Efficient Appointment System:

The introduction of the `Appointment` and `AppointmentSlot` classes captures the essence of the booking system at FDC. With attributes such as `appointmentDate`, `appointmentSlot`, and associations with both `Doctor` and `Patient`, the system can effectively manage and schedule appointments.

The `AppointmentSlot` class further refines this by marking out specific time periods and associating them with a particular `Doctor` and `SurgeryRoom`. This would streamline the allocation process for the four surgery rooms mentioned in the scenario.

Staff Management and Specialized Roles:

Distinct classes for roles like `Manager`, `Nurse`, and `Receptionist`, inheriting from the base `Staff` class, ensure that staff members are managed effectively. The `Manager` class's association with `ScanRoom` aligns with the scenario where the manager assigns dentists to the dental scan room during available time slots. The `Nurse` class's relationship with `SurgeryRoom` ensures that each surgery room has a dedicated nurse, as specified.

Specialized Surgery and Scan Rooms:

The distinct classes for `SurgeryRoom` and `ScanRoom` (which inherits from `SurgeryRoom`) ensure that FDC's facility nuances are captured. The `ScanRoom`'s relationships with both `Doctor` and `Manager` classes support the idea that dentists are assigned by the manager to operate the dental scan machine.

Registration System:

The `Registration` class ensures that both patients and doctors are duly registered before availing or providing services. By incorporating a registration fee and maintaining this as a composition relationship with both `Patient` and `Doctor`, the system guarantees that the registration process is well-integrated.

In summary, the modified UML Class Diagram meticulously captures the multifaceted operations of the Family Dental Care (FDC). From patient registrations, staff roles, treatment offerings, and appointment systems, to financial transactions, the diagram ensures a systematic and structured approach to design an information system that meets FDC's needs. Drawing upon the object-oriented design approach, this UML diagram provides a robust foundation upon which an efficient, comprehensive, and user-friendly software application for FDC can be developed.

2.4 Derive code for Detailed UML Class Diagram

2.4.1 Person.java

```
// The Person class represents a generic individual with basic personal
details.
public class Person {

    // Attributes
    private String name;
    private String address;
    private String nationalId;
    private String contactNumber;
    private String dateOfBirth;

    // Constructor
    public Person(String name, String address, String nationalId, String
contactNumber, String dateOfBirth) {
        this.name = name;
        this.address = address;
        this.nationalId = nationalId;
        this.contactNumber = contactNumber;
        this.dateOfBirth = dateOfBirth;
    }

    // Getters and Setters

    // Returns the name of the person
    public String getName() {
        return name;
    }

    // Sets the name of the person
    public void setName(String name) {
        this.name = name;
    }

    // Returns the address of the person
    public String getAddress() {
        return address;
    }

    // Sets the address of the person
    public void setAddress(String address) {
        this.address = address;
    }

    // Returns the national ID of the person
    public String getNationalId() {
        return nationalId;
    }

    // Sets the national ID of the person
    public void setNationalId(String nationalId) {
        this.nationalId = nationalId;
    }

    // Returns the contact number of the person
    public String getContactNumber() {
        return contactNumber;
    }

    // Sets the contact number of the person
    public void setContactNumber(String contactNumber) {
        this.contactNumber = contactNumber;
    }

    // Returns the date of birth of the person
    public String getDateOfBirth() {
        return dateOfBirth;
    }

    // Sets the date of birth of the person
    public void setDateOfBirth(String dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }
}
```

2.4.2 Patient.java

```
// The Patient class represents a specific type of person who is either
registered or not in a hospital system.
public class Patient extends Person {

    // Attributes specific to a Patient
    private boolean isRegistered;
    private Registration registration; // Composition relationship

    // Constructor
    public Patient(String name, String address, String nationalId,
String contactNumber, String dateOfBirth,
                   boolean isRegistered, Registration registration) {
        super(name, address, nationalId, contactNumber, dateOfBirth);
        this.isRegistered = isRegistered;
        this.registration = registration;
    }

    // Getters and Setters

    // Returns if the patient is registered
    public boolean getIsRegistered() {
        return isRegistered;
    }

    // Sets the registration status of the patient
    public void setIsRegistered(boolean isRegistered) {
        this.isRegistered = isRegistered;
    }

    // Returns the registration details of the patient
    public Registration getRegistration() {
        return registration;
    }

    // Sets the registration details of the patient
    public void setRegistration(Registration registration) {
        this.registration = registration;
    }
}
```

2.4.3 Doctor.java

```
// The Doctor class represents a medical professional with specific
experience and registration details.
public class Doctor extends Person {

    // Attributes specific to a Doctor
    private String experienceYears;
    private Registration registration; // Composition relationship

    // Constructor
    public Doctor(String name, String address, String nationalId, String
contactNumber, String dateOfBirth,
                  String experienceYears, Registration registration) {
        super(name, address, nationalId, contactNumber, dateOfBirth);
        this.experienceYears = experienceYears;
        this.registration = registration;
    }

    // Getters and Setters

    // Returns the number of years of experience of the doctor
    public String getExperienceYears() {
        return experienceYears;
    }

    // Sets the number of years of experience of the doctor
    public void setExperienceYears(String experienceYears) {
        this.experienceYears = experienceYears;
    }

    // Returns the registration details of the doctor
    public Registration getRegistration() {
        return registration;
    }

    // Sets the registration details of the doctor
    public void setRegistration(Registration registration) {
        this.registration = registration;
    }
}
```

2.4.4 Consultant.java

```
// The Consultant class represents a specialized Doctor with post-
graduate qualifications.
public class Consultant extends Doctor {

    // Attributes specific to a Consultant
    private String postGraduateQualification;
    private String universityCountry;
    private int numberOfYears;

    // Constructor
    public Consultant(String name, String address, String nationalId,
String contactNumber, String dateOfBirth,
                      String experienceYears, Registration registration,
String postGraduateQualification,
                      String universityCountry, int numberOfYears) {
        super(name, address, nationalId, contactNumber, dateOfBirth,
experienceYears, registration);
        this.postGraduateQualification = postGraduateQualification;
        this.universityCountry = universityCountry;
        this.numberOfYears = numberOfYears;
    }

    // Getters and Setters

    // Returns the post-graduate qualification of the consultant
    public String getPostGraduateQualification() {
        return postGraduateQualification;
    }

    // Sets the post-graduate qualification of the consultant
    public void setPostGraduateQualification(String
postGraduateQualification) {
        this.postGraduateQualification = postGraduateQualification;
    }

    // Returns the country of the university where the consultant
studied
    public String getUniversityCountry() {
        return universityCountry;
    }

    // Sets the country of the university where the consultant studied
    public void setUniversityCountry(String universityCountry) {
        this.universityCountry = universityCountry;
    }

    // Returns the number of years since obtaining the post-graduate
qualification
```

```
    public int getNumberOfYears() {
        return numberOfYears;
    }

    // Sets the number of years since obtaining the post-graduate
qualification
    public void setNumberOfYears(int numberOfYears) {
        this.numberOfYears = numberOfYears;
    }
}
```

2.4.5 Dentist.java

```
// The Dentist class represents a medical professional specialized in
dentistry.
public class Dentist extends Doctor {

    // Constructor
    public Dentist(String name, String address, String nationalId,
String contactNumber, String dateOfBirth,
                   String experienceYears, Registration registration) {
        super(name, address, nationalId, contactNumber, dateOfBirth,
experienceYears, registration);
    }

    // As per the UML, there are no additional attributes or methods
specific to Dentist, so it directly inherits from Doctor.
}
```

2.4.6 Appointment.java

```java
import java.util.Date; // I've added the import java.util.Date;
statement at the beginning of the class to utilize the Date class.

// The Appointment class represents a medical appointment that includes
details about the appointment date,
// the doctor involved, the patient, and the appointment slot.
public class Appointment {

    // Attributes of an Appointment
    private Date appointmentDate;
    private Doctor doctor;            // Aggregation relationship
    private Patient patient;          // Aggregation relationship
    private AppointmentSlot appointmentSlot; // Aggregation relationship

    // Constructor
    public Appointment(Date appointmentDate, Doctor doctor, Patient
patient, AppointmentSlot appointmentSlot) {
        this.appointmentDate = appointmentDate;
        this.doctor = doctor;
        this.patient = patient;
        this.appointmentSlot = appointmentSlot;
    }

    // Getters and Setters

    // Returns the date of the appointment
    public Date getAppointmentDate() {
        return appointmentDate;
    }

    // Sets the date of the appointment
    public void setAppointmentDate(Date appointmentDate) {
        this.appointmentDate = appointmentDate;
    }

    // Returns the doctor associated with the appointment
    public Doctor getDoctor() {
        return doctor;
    }

    // Sets the doctor for the appointment
    public void setDoctor(Doctor doctor) {
        this.doctor = doctor;
    }

    // Returns the patient associated with the appointment
    public Patient getPatient() {
        return patient;
    }

    // Sets the patient for the appointment
    public void setPatient(Patient patient) {
        this.patient = patient;
    }
    // Returns the appointment slot details
    public AppointmentSlot getAppointmentSlot() {
        return appointmentSlot;
    }

    // Sets the appointment slot details
    public void setAppointmentSlot(AppointmentSlot appointmentSlot) {
        this.appointmentSlot = appointmentSlot;
    }
}
```

2.4.7 Treatment.java

```
// The Treatment class represents medical treatment details, including a
description and cost.
public class Treatment {

    // Attributes of a Treatment
    private String description;
    private double cost;

    // Constructor
    public Treatment(String description, double cost) {
        this.description = description;
        this.cost = cost;
    }

    // Getters and Setters

    // Returns the description of the treatment
    public String getDescription() {
        return description;
    }

    // Sets the description of the treatment
    public void setDescription(String description) {
        this.description = description;
    }

    // Returns the cost of the treatment
    public double getCost() {
        return cost;
    }

    // Sets the cost of the treatment
    public void setCost(double cost) {
        this.cost = cost;
    }
}
```

2.4.8 Payment.java

```
import java.util.Date; // I've added the import java.util.Date;
statement at the beginning of the class to utilize the Date class.

// The Payment class represents details of a payment made by a patient,
including the amount and date.
public class Payment {

    // Attributes of a Payment
    private double amount;
    private Date paymentDate;

    // Constructor
    public Payment(double amount, Date paymentDate) {
        this.amount = amount;
        this.paymentDate = paymentDate;
    }

    // Getters and Setters

    // Returns the amount of the payment
    public double getAmount() {
        return amount;
    }

    // Sets the amount of the payment
    public void setAmount(double amount) {
        this.amount = amount;
    }

    // Returns the date of the payment
    public Date getPaymentDate() {
        return paymentDate;
    }

    // Sets the date of the payment
    public void setPaymentDate(Date paymentDate) {
        this.paymentDate = paymentDate;
    }
}
```

2.4.9 Staff.java

```
// The Staff class represents a general staff member in the medical
institution.
public class Staff extends Person {

    // Attributes of a Staff member
    private String position;

    // Constructor
    public Staff(String name, String address, String nationalId, String
contactNumber,
                 String dateOfBirth, String position) {
        super(name, address, nationalId, contactNumber, dateOfBirth);
        this.position = position;
    }

    // Getters and Setters

    // Returns the position of the staff member
    public String getPosition() {
        return position;
    }

    // Sets the position of the staff member
    public void setPosition(String position) {
        this.position = position;
    }
}
```

2.4.10 Manager.java

```
// The Manager class represents a manager in the medical institution,
with an added functionality of managing staff.
public class Manager extends Staff {

    // Attributes of a Manager
    private Staff staff;  // Composition relationship: a manager manages
a specific staff member.

    // Constructor
    public Manager(String name, String address, String nationalId,
String contactNumber,
                   String dateOfBirth, String position, Staff staff) {
        super(name, address, nationalId, contactNumber, dateOfBirth,
position);
        this.staff = staff;
    }

    // Getters and Setters

    // Returns the staff member managed by the manager
    public Staff getStaff() {
        return staff;
    }

    // Assigns a staff member for the manager to manage
    public void setStaff(Staff staff) {
        this.staff = staff;
    }
}
```

2.4.11 Nurse.java

```
// The Nurse class represents a nurse in the medical institution.
public class Nurse extends Staff {

    // Constructor
    public Nurse(String name, String address, String nationalId, String
contactNumber,
                 String dateOfBirth, String position) {
        super(name, address, nationalId, contactNumber, dateOfBirth,
position);
    }

    // The Nurse class might have additional methods and attributes
specific to its role, which can be added later.
}
```

2.4.12 Receptionist.java

```
// The Receptionist class represents a receptionist in the medical
institution.
public class Receptionist extends Staff {

    // Constructor
    public Receptionist(String name, String address, String nationalId,
String contactNumber,
                        String dateOfBirth, String position) {
        super(name, address, nationalId, contactNumber, dateOfBirth,
position);
    }

    // The Receptionist class might have additional methods and
attributes specific to its role, which can be added later.
}
```

2.4.13 Registration.java

```
// The Registration class represents the registration details of a
person in the medical institution.
public class Registration {

    // Attributes of a Registration
    private double registrationFee;

    // Constructor
    public Registration(double registrationFee) {
        this.registrationFee = registrationFee;
    }

    // Getters and Setters

    // Returns the registration fee
    public double getRegistrationFee() {
        return registrationFee;
    }

    // Sets the registration fee
    public void setRegistrationFee(double registrationFee) {
        this.registrationFee = registrationFee;
    }
}
```

2.4.14 AppointmentSlot.java

```
// The AppointmentSlot class represents a specific time slot for an
appointment in the medical institution.
public class AppointmentSlot {

    // Attributes of an AppointmentSlot
    private String startTime;
    private String endTime;
    private Doctor doctor;           // Aggregation relationship: each
slot can be associated with a specific doctor.
    private SurgeryRoom surgeryRoom; // Aggregation relationship: each
slot can be associated with a specific surgery room.

    // Constructor
    public AppointmentSlot(String startTime, String endTime, Doctor
doctor, SurgeryRoom surgeryRoom) {
        this.startTime = startTime;
        this.endTime = endTime;
        this.doctor = doctor;
        this.surgeryRoom = surgeryRoom;
    }

    // Getters and Setters

    // Returns the start time of the slot
    public String getStartTime() {
        return startTime;
    }

    // Sets the start time of the slot
    public void setStartTime(String startTime) {
        this.startTime = startTime;
    }

    // Returns the end time of the slot
    public String getEndTime() {
        return endTime;
    }

    // Sets the end time of the slot
    public void setEndTime(String endTime) {
        this.endTime = endTime;
    }

    // Returns the doctor associated with the slot
    public Doctor getDoctor() {
        return doctor;
    }

    // Sets the doctor for the slot
    public void setDoctor(Doctor doctor) {
        this.doctor = doctor;
    }

    // Returns the surgery room associated with the slot
    public SurgeryRoom getSurgeryRoom() {
        return surgeryRoom;
    }

    // Sets the surgery room for the slot
    public void setSurgeryRoom(SurgeryRoom surgeryRoom) {
        this.surgeryRoom = surgeryRoom;
    }
}
```

2.4.15 SurgeryRoom.java

```java
// The SurgeryRoom class represents a room where surgeries or other
medical procedures take place.
public class SurgeryRoom {

    // Attributes of a SurgeryRoom
    private int roomNumber;

    // Constructor
    public SurgeryRoom(int roomNumber) {
        this.roomNumber = roomNumber;
    }

    // Getters and Setters

    // Returns the room number
    public int getRoomNumber() {
        return roomNumber;
    }

    // Sets the room number
    public void setRoomNumber(int roomNumber) {
        this.roomNumber = roomNumber;
    }
}
```

2.4.16 ScanRoom.java

```java
// The ScanRoom class represents a specialized room in the medical
institution where scans are conducted.
// It inherits properties from the SurgeryRoom.
public class ScanRoom extends SurgeryRoom {

    // Constructor
    public ScanRoom(int roomNumber) {
        super(roomNumber); // Calls the constructor of the parent class
(SurgeryRoom)
    }

    // As there are no additional attributes or methods in ScanRoom as
per your UML,
    // the ScanRoom class primarily inherits everything from
SurgeryRoom.
    // If needed, additional methods or attributes specific to ScanRoom
can be added later.
}
```

Task 3

3.1 Design Patterns in Software Development

3.1.1 Introduction to Design Patterns

Design patterns are essentially reusable solutions to common problems that software developers encounter during the software design process. They're not templates for writing code, but rather conceptual frameworks that guide the design of object-oriented systems. By understanding and leveraging design patterns, developers can create software that is scalable, maintainable, and adaptable to change.

The concept of design patterns originated from the architectural designs suggested by Christopher Alexander in the 1970s. This idea was then adapted for software design by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) in their seminal book, "Design Patterns: Elements of Reusable Object-Oriented Software," published in 1995. The book classified 23 design patterns into three categories: Creational, Structural, and Behavioral, each catering to specific design problems.

3.1.2 Importance in Software Development

Solving Recurring Problems: Just as there are recurring problems in architectural design (like how to maximize light in a room), software design has its recurring challenges. Design patterns offer tested and proven solutions to these challenges. When faced with a common problem, developers can lean on the relevant design pattern rather than reinventing the wheel.

Promoting Code Reusability: Design patterns foster the reuse of software components. With a consistent design approach, different parts of a software application can reuse components, making the entire software development process more efficient.

Enhancing Communication: For software development teams, it's crucial to have a shared language. When a developer mentions a specific design pattern, others on the team immediately understand the design approach being referred to. This shorthand boosts team communication, reduces misinterpretation, and makes collaborative efforts more cohesive. When a developer says they're implementing the "Observer pattern," team members are instantly on the same page, aware of the problem being addressed and the general solution being used.

Scalability and Maintainability: Patterns provide a standard approach to solving common problems. By adhering to these standards, the resulting software becomes easier to scale. If a new feature is added or an existing feature needs modification, developers can predict how changes will affect the system due to the predictable structure that patterns offer. This predictability also makes the software more maintainable, as new developers or teams can quickly understand and navigate the system.

Adaptable Software Systems: In the fast-paced world of software development, requirements often change. Design patterns allow for flexibility, ensuring that systems can adapt to new requirements with minimal disruptions. For instance, the Strategy pattern lets algorithms be swapped in and out as needed without altering the code that uses them.

In essence, design patterns are invaluable tools for software developers. They encapsulate best practices derived from real-world experience and collective wisdom. By understanding and employing these patterns, developers can navigate the complexities of software design with greater confidence, efficiency, and foresight. They're not just strategies for writing code; they're foundational elements for building robust, efficient, and scalable software systems.

3.2 Range of Design Patterns

3.2.1 Creational Design Patterns

Creational design patterns focus on mechanisms of object creation in such a way that objects are created in a suitable manner for the specific situation. Instead of creating objects directly using constructors, these patterns offer ways to do this indirectly, adding flexibility and efficiency to the process. Let's delve deeper into each of the primary Creational design patterns.

1. Singleton Pattern:

Intent: Ensures that a particular class has only one instance and provides a global point to access this instance.

Description: Often used where we need to control the resource access like configurations, thread pools, or caching. The Singleton pattern restricts the instantiation of a class to one single instance and provides a method to retrieve that instance. This pattern ensures that the class is instantiated just once and provides a way to access its instance without the need to instantiate it again.

Example: Database connections, Logger classes, Configuration managers.

2. Factory Method Pattern:

Intent: Define an interface for creating a single object, but let subclasses decide which class to instantiate. The Factory Method lets a class defer instantiation to subclasses.

Description: In this pattern, the creation of an object is abstracted and is delegated to its subclasses. Instead of calling the object's constructor directly, a factory method is called to create the object. This pattern is particularly useful when the exact type of the object isn't known until runtime.

Example: A GUI library might use a factory method to create buttons. The exact type of button (like WindowsButton, MacOSButton) would be determined at runtime based on the operating system.

3. Abstract Factory Pattern:

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Description: The Abstract Factory pattern revolves around the creation of object families. Unlike the Factory Method, which allows for the creation of one type of object, the Abstract Factory can create several related types of objects. It typically involves multiple Factory Methods, one for each type of object to be created.

Example: User interfaces that need to remain consistent among devices but might have different visual elements on different platforms (e.g., Windows vs. MacOS).

4. Prototype Pattern:

Intent: Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.

Description: The Prototype pattern is used when creating a duplicate object is easier than creating an instance from scratch. This pattern involves creating a prototype of an object and then cloning it to produce new objects. This is especially useful when the objects have numerous shared configurations and only a few differences.

Example: A game might use the Prototype pattern to clone characters or items. If we have a tree object in a game, instead of creating a new tree from scratch every time, we can simply clone the prototype tree, ensuring consistency and performance.

5. Builder Pattern:

Intent: Separate the construction of a complex object from its representation so that the same construction process can produce different representations.

Description: The Builder pattern is used for constructing complex objects step by step. Instead of using a single constructor with many parameters, the object can be constructed piece by piece, allowing for more control over the final outcome.

Example: Think of constructing a meal at a fast-food restaurant. A typical meal might include a main item, a side item, a drink, and dessert. The Builder pattern allows a customer to customize their meal, choosing each component separately.

Each of these Creational patterns serves a unique purpose in object-oriented software design and adds flexibility and clarity to the object instantiation process.

3.2.2 Structural Design Patterns

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures. The structural design patterns simplify the structure by identifying relationships. These patterns focus on ensuring that the system is composed of entities that interact in a robust and scalable manner. Here's a deeper look into each of the primary Structural design patterns:

1. Adapter Pattern:

Intent: Convert the interface of a class into another interface that clients expect. The Adapter pattern allows classes with incompatible interfaces to work together.

Description: This pattern acts as a bridge between two incompatible interfaces. It involves a class that joins functions of an independent or incompatible interface to another class.

Example: Consider a legacy system with a different data processing structure. Instead of rewriting the entire legacy system, an adapter class could be created to interface with the newer system, translating requests between them.

2. Bridge Pattern:

Intent: Decouple an abstraction from its implementation allowing the two to vary independently.

Description: This pattern involves an interface or abstract class (the 'bridge') that bridges the gap between the abstraction and its implementation. By splitting the logic into separate hierarchical structures, they can evolve independently of each other.

Example: For a graphics system, the `Shape` class can be an abstraction while the `DrawAPI` is the implementation. Shapes like `Circle` or `Rectangle` can use `DrawAPI` without binding directly to its concrete implementation.

3. Composite Pattern:

Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly.

Description: This pattern creates a tree structure of group objects. It treats both part and whole objects uniformly. This allows us to simplify complex tree structures by treating leaf and composite objects the same.

Example: A graphical system might use the Composite pattern to represent complex graphics as a tree structure of simple and compound graphics. Both simple and compound graphics can be rendered in a uniform manner.

4. Decorator Pattern:

Intent: Attach additional responsibilities dynamically to an object. Decorators provide a flexible alternative to subclassing for extending functionality.

Description: This pattern allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.

Example: A coffee shop might offer additives for its beverages. Using the Decorator pattern, we can decorate a simple coffee with extras like milk, sugar, caramel, etc., each adding its unique behavior to the coffee object.

5. Facade Pattern:

Intent: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Description: This pattern hides the complexities of a system and provides an interface to the client from which the subsystem can be accessed. It involves a single class that represents an entire subsystem.

Example: A computer system can be represented by a `ComputerFacade`, hiding the complexities of each component (like CPU, Memory, Hard Drive) but providing a simple method to start the computer.

6. Flyweight Pattern:

Intent: Use sharing to support a large number of fine-grained objects efficiently.

Description: The Flyweight pattern aims to minimize memory use by sharing as much data as possible with similar objects. It is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

Example: A text processor may store each character's glyph as a flyweight, where each character is a shared object, and only its position in the text is unique.

7. Proxy Pattern:

Intent: Provide a surrogate or placeholder for another object to control access to it.

Description: This pattern involves a class that represents the functionality of another class. It can act as an interface to real objects, create expensive objects on demand, or represent objects in remote locations, among other things.

Example: An image viewer might use the Proxy pattern to delay loading the actual image from a file until it is actually displayed, improving performance and reducing system resource usage.

Structural patterns offer various ways to ensure that components in a system are effectively linked, providing solutions that allow entities to work together cohesively while maintaining flexibility and scalability.

3.2.3 Behavioral Design Patterns

Behavioral design patterns are concerned with the assignment of responsibilities between objects and how they communicate. These patterns aim to ensure that entities and objects within the system collaborate and interact in a flexible but consistent manner. Let's explore the main Behavioral design patterns:

1. Chain of Responsibility Pattern:

Intent: Decouple the sender from receiver objects by allowing more than one object to handle a request. The request passes along the chain until an object handles it or it reaches the end of the chain.

Description: It consists of a chain of receiving objects having the reference to a next receiver in the chain. A request is passed through this chain, and each receiver decides either to process it or pass it to the next receiver.

Example: A logging system can have multiple logging levels (Info, Debug, Error). A message can be passed down a chain of loggers until it reaches a logger that can handle that specific level.

2. Command Pattern:

Intent: Encapsulate a request as an object, thereby allowing users to parameterize clients with different requests, queue requests, or log them. It also supports undoable operations.

Description: This pattern involves three classes: invoker, receiver, and command. The invoker issues a request, the command encapsulates that request, and the receiver is responsible for executing the request.

Example: In a home automation system, the Command pattern can represent and encapsulate all details of a lighting on/off operation. The remote control (invoker) can then use different command objects to perform various actions.

3. Interpreter Pattern:

Intent: Provide a way to evaluate language grammar or expressions for particular languages.

Description: Typically used for parsing. It involves evaluating a sequence (often a sentence) based on a set of rules or grammar.

Example: A calculator for simple arithmetic expressions can use the Interpreter pattern to parse and evaluate different user-entered expressions.

4. Iterator Pattern:

Intent: Provide a way to access elements of an aggregate object without exposing its underlying representation.

Description: This pattern provides a mechanism to traverse through a list of items without the client needing to understand the underlying structure.

Example: An old radio tuning dial scanning through frequencies. Each station is an item in a list, and the dial provides a way to iterate through them.

5. Mediator Pattern:

Intent: Define an object that encapsulates how a set of objects interact. The Mediator promotes loose coupling by ensuring that objects don't refer to each other explicitly.

Description: A central hub through which all interaction takes place. Instead of objects communicating directly with each other, they communicate via the mediator.

Example: An air traffic control tower works as a mediator between planes, ensuring they don't come into conflict without communicating with each other directly.

6. Memento Pattern:

Intent: Capture an object's internal state so that it can be restored to that state later.

Description: Used to restore the state of an object to a previous state, which can be useful for "undo" functionalities.

Example: A text editor using the Memento pattern can save states of texts and restore them if the user chooses to undo changes.

7. Observer Pattern:

Intent: Define a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically.

Description: It involves an object (known as the subject) maintaining a list of dependents (observers) and notifying them of state changes, typically by calling one of their methods.

Example: Subscribers and publishers in a magazine subscription model. When a new magazine is published (state change), all subscribers get notified.

8. State Pattern:

Intent: Allow an object to alter its behavior when its internal state changes. The object appears to change its class.

Description: This pattern involves classes that represent various states and a context object whose behavior varies as its state object changes.

Example: A traffic light changing its behavior based on its current state (red, yellow, green).

9. Strategy Pattern:

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. The Strategy pattern lets the algorithm vary independently from clients that use it.

Description: It defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.

Example: A mapping application can have different strategies for calculating routes, such as shortest path, least traffic, or least tolls.

10. Template Method Pattern:

Intent: Define the skeleton of an algorithm in an operation but delay some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Description: It involves an abstract class with a templated method consisting of a series of method calls. Some of these method calls are implemented by the base class and others are deferred to derived subclasses. This structure ensures that the overall algorithm's structure remains the same, but specific steps can be redefined by the subclasses.

Example: Consider a data mining application where the overall process involves loading data, analyzing it, and generating a report. While the overall sequence of steps remains consistent, the specifics of data loading or analysis can vary. The template method can be used here, with the general sequence defined in the base class, and specific data loading or analysis techniques defined in subclasses.

11. Visitor Pattern:

Intent: Represent an operation to be performed on the elements of an object structure without changing the classes on which it operates. This pattern can add new operations to existing class structures without modifying them.

Description: Involves a visitor class and a set of "element" classes. Element classes accept a visitor, and the visitor performs operations on those elements. This decouples operations from the objects on which they operate.

Example: Consider a shopping cart with different types of items (books, electronics, groceries). To calculate shipping costs, instead of adding a shipping calculation method to every item type (which would be intrusive), a visitor class can be created for shipping calculation. Each item then accepts this visitor to compute shipping costs, allowing for flexible and extensible shipping calculations.

Behavioral patterns emphasize the ways in which objects communicate and collaborate, ensuring efficient and flexible interactions. By providing patterns that help define how objects interact in diverse scenarios, behavioral patterns equip developers with tools to build cohesive and decoupled systems.

3.3 Singleton Design Pattern

3.3.1 An Overview

Introduction

The Singleton Design Pattern is one of the creational design patterns that ensures a particular class has only one instance throughout a program's lifecycle and provides a point of access to this instance. It's particularly useful when we need to control access to shared resources, such as configuration settings, connection pools, or logging services.

Core Principle

The primary function of the Singleton pattern is to maintain a single instance of an object across the system. This is achieved by making the constructor of the class private, thus preventing the instantiation of the class from other classes or objects. Instead, a static method or property is often used to access this singular instance, ensuring controlled and consistent access.

Rationale Behind the Pattern

There are scenarios in software design where it's essential to ensure that a class has only one instance. Consider system configurations; if multiple instances of configuration settings existed, there could be inconsistencies in how the system behaves. Another example could be a connection pool which manages database connections. If there were multiple instances, it might lead to connection leaks or inefficiencies. The Singleton pattern elegantly addresses this by encapsulating the single instance logic within the class itself.

Implementation Details

While the core idea behind Singleton is relatively straightforward, its implementation can vary depending on requirements:

- Lazy Initialization: The instance is created only when it's needed. This is beneficial if the instance creation is costly and the instance is not used immediately when the application starts.
- Thread Safety: In multi-threaded environments, care must be taken to ensure that two threads don't create a Singleton instance simultaneously. This is usually achieved using locking mechanisms.
- Static Initialization: This is an alternative to lazy initialization. Here, the instance is created when the class is loaded. This approach is thread-safe but might lead to resource wastage if the instance isn't used.

Advantages and Drawbacks

Advantages:

- Controlled access to the sole instance, ensuring consistent behavior.
- Reduced overhead, as the instance is created only once.

Drawbacks:

- The Singleton pattern can sometimes be misused in scenarios where a single instance isn't strictly necessary, leading to unnecessary constraints.
- Testing can become challenging due to the global state introduced by Singleton.

Conclusion

The Singleton Design Pattern, while simple in its essence, addresses a crucial need in software design, ensuring uniformity and consistency. However, like all tools, it's essential to use it judiciously, understanding its strengths and potential pitfalls.

3.3.2 Understanding its Real-world Applications

The Singleton Design Pattern, one of the creational design patterns, stands out with its unique proposition: ensuring a class has only one instance throughout the runtime of an application and providing a global point of access to this instance. This design pattern is particularly advantageous in scenarios where it's crucial to control access to shared resources or maintain a single state across the system. Let's delve into some classic use cases where the Singleton pattern has proved invaluable:

Database Connections

When dealing with databases, the overhead of establishing a connection is often high, and it's inefficient to create and destroy connections frequently. By employing the Singleton pattern for database connection management, an application ensures that only one connection remains active throughout its lifecycle. This not only minimizes resource overhead but also prevents potential issues from simultaneous connections.

Configuration Management

Many applications require reading configurations from files or external sources. Loading these configurations repeatedly can be resource-intensive and can lead to inconsistencies if configurations change. A Singleton configuration manager can load configurations once and provide a centralized access point, ensuring consistent and efficient configuration usage across the system.

Logging Mechanisms

Logging is a quintessential component of many systems. Whether it's for debugging, auditing, or compliance, having a single instance of a logger ensures that log entries are sequential and consistent. A Singleton logger can manage file access, ensuring entries don't overlap and providing a streamlined mechanism to log data across multiple parts of an application.

Caching Mechanisms

Cache management is a nuanced task where data is temporarily stored to speed up repeated requests. A Singleton cache manager can hold data and ensure that all parts of an application have consistent access to this cache. This not only enhances performance but also ensures that stale or redundant data is minimized.

Hardware Interface Access

In systems where hardware interfaces are involved, like printers or graphical processing units, direct access to the hardware by multiple components can lead to conflicts. A Singleton can serve as a gatekeeper, ensuring synchronized and conflict-free access to the hardware resource.

The Singleton Design Pattern, while straightforward in its premise, caters to a myriad of scenarios in software design. It's not just about ensuring a single instance but more about the consistency, efficiency, and controlled access it offers. However, it's essential to employ it judiciously. Over-reliance can lead to issues like masked dependencies or challenges in parallel testing. Like all design patterns, understanding the problem domain and ensuring the pattern's applicability is paramount.

3.3.3 Basic UML class diagram

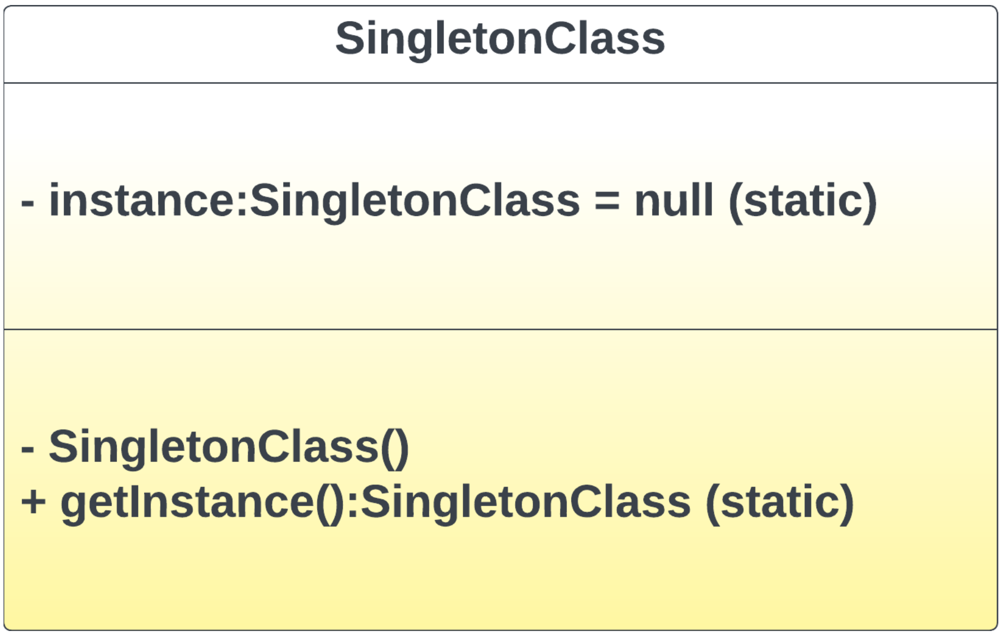

SingletonClass
- instance:SingletonClass = null (static)
- SingletonClass()
+ getInstance():SingletonClass (static)

Figure 3. 1 Singleton Design Pattern Basic UML class diagram

In this design:

- The class name is `SingletonClass`.

- There's a private static attribute `instance` which holds the single instance of the `SingletonClass`. The attribute is initialized as `null` and will be assigned the instance of `SingletonClass` when `getInstance()` is called the first time.

- There's a private constructor to prevent creating an instance of the `SingletonClass` from outside the class.

- The `getInstance` method is public and static. It returns the single instance of `SingletonClass` if it exists or creates one if it doesn't.

3.3.4 Creating UML class diagram for the scenario

Given the scenario, one suitable aspect for the Singleton Design Pattern would be the management of the dental scan room. This is because there's a single dental scan room in the facility, and only one dentist can operate the dental scan machine at any given time. The Singleton pattern ensures that only one instance of the dental scan room management is instantiated and provides a global point of access to the dental scan room's resources.

The below diagram, the Singleton pattern ensures that only one instance of the 'DentalScanRoomManager' class can exist, and the 'getInstance' method provides access to this instance.

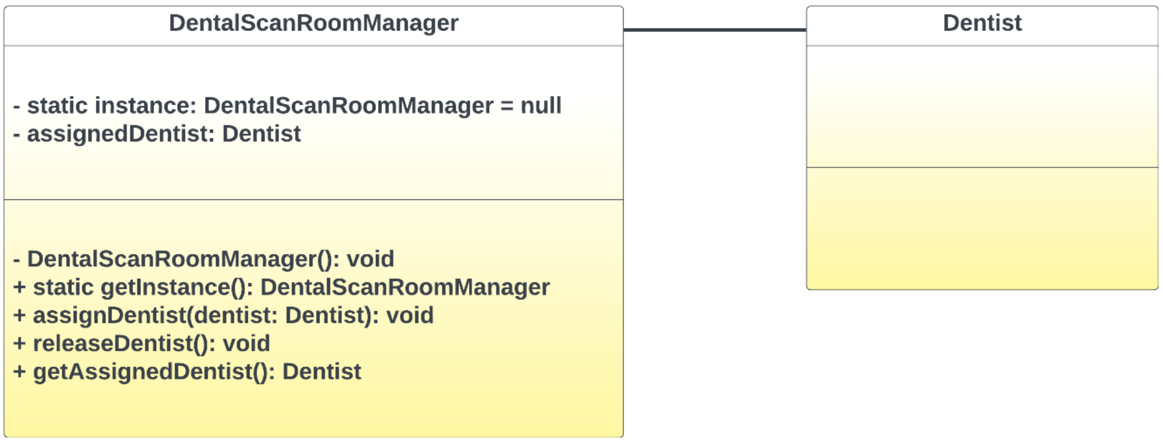
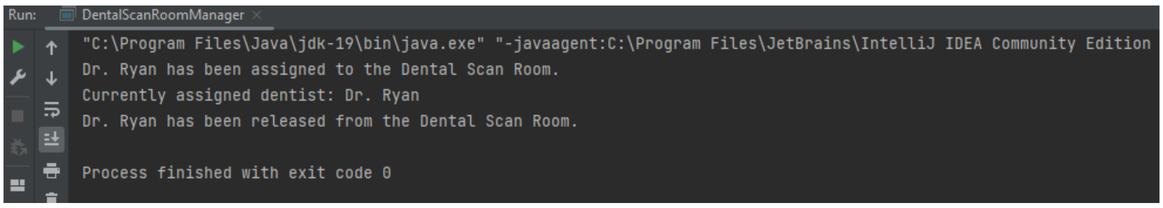

DentalScanRoomManager
- static instance: DentalScanRoomManager = null
- assignedDentist: Dentist
- DentalScanRoomManager(): void
+ static getInstance(): DentalScanRoomManager
+ assignDentist(dentist: Dentist): void
+ releaseDentist(): void
+ getAssignedDentist(): Dentist
Dentist

Figure 3. 2 Singleton Design Pattern Creating UML class diagram for the scenario

3.3.5 Java code with an output for the illustrated UML class diagram of the scenario

```java
// Define the Dentist class which will be associated with the
DentalScanRoomManager
class Dentist {
    private String name;

    // Constructor
    public Dentist(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    // ... other attributes and methods relevant to the Dentist
}

// Singleton class for managing the Dental Scan Room
public class DentalScanRoomManager {

    // Static attribute that holds the single instance of the class
    private static DentalScanRoomManager instance = null;

    // Attribute for storing the dentist currently assigned to the scan
room
    private Dentist assignedDentist = null;

    // Private constructor to prevent instantiation from other classes
    private DentalScanRoomManager() {
        // Nothing to initialize here for now
    }

    // Public method to provide access to the single instance of the
class
    public static synchronized DentalScanRoomManager getInstance() {
        // If instance hasn't been initialized yet, create a new one
        if (instance == null) {
            instance = new DentalScanRoomManager();
        }

        // Return the single instance
        return instance;
    }

    // Assign a dentist to the scan room
    public void assignDentist(Dentist dentist) {
        this.assignedDentist = dentist;
        System.out.println(dentist.getName() + " has been assigned to
the Dental Scan Room.");
    }

    // Release the dentist from the scan room (e.g. after finishing a
scan)
    public void releaseDentist() {
        System.out.println(assignedDentist.getName() + " has been 
released from the Dental Scan Room.");
        this.assignedDentist = null;
    }

    // Get the dentist currently assigned to the scan room
    public Dentist getAssignedDentist() {
        return assignedDentist;
    }

    public static void main(String[] args) {
        // Sample usage
        DentalScanRoomManager manager = 
DentalScanRoomManager.getInstance();
        Dentist drJohn = new Dentist("Dr. Ryan");

        manager.assignDentist(drJohn);
        System.out.println("Currently assigned dentist: " + 
manager.getAssignedDentist().getName());

        manager.releaseDentist();
    }
}
```



```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
Dr. Ryan has been assigned to the Dental Scan Room.
Currently assigned dentist: Dr. Ryan
Dr. Ryan has been released from the Dental Scan Room.

Process finished with exit code 0
```

Figure 3. 3 Singleton Design Pattern Java code output

3.4 Composite Design Pattern

3.4.1 An Overview

Introduction

The Composite Design Pattern falls under the category of behavioral design patterns and offers a solution to the challenge of managing tree-like structures. It's particularly useful when dealing with hierarchies and part-whole relationships, as often seen in graphical systems, file systems, or organizational structures. By enabling we to treat individual objects and compositions of objects uniformly, the Composite pattern simplifies client code and improves its maintainability.

Core Principle

At the heart of the Composite pattern is the idea of treating both single (leaf) elements and their compositions in a consistent manner. This is achieved by having both the leaf and composite nodes implement the same interface or extend the same abstract class. As a result, clients can work with complex tree structures without having to differentiate between individual elements and their groupings, making the code more general and easier to manage.

Rationale Behind the Pattern

In real-world scenarios, many systems consist of hierarchies or tree structures. For instance, in graphical systems, we might have groups of shapes that are composed of simpler shapes. In organizational charts, an executive might have a team of managers reporting to them, each of whom might have their team. To navigate, modify, or extend such systems, one would need a consistent and scalable way to deal with both individual elements and their groupings. The Composite pattern offers this by encapsulating both in a common interface.

Implementation Details

When implementing the Composite pattern, there are a few key elements:

- Component: This is the base component (usually an abstract class or interface) that defines the default behavior and interface for all concrete components, both composite and leaf.
- Leaf: Represents the individual objects that don't have any sub-elements. These are the building blocks of the structure.
- Composite: Represents the composite objects which contain leaf elements or other smaller composites. These objects implement the methods defined in Component and usually delegate the work to their children, iterating over them as needed.

Advantages and Drawbacks

Advantages:

- Provides clarity by ensuring that clients treat single elements and compositions uniformly.
- Facilitates adding new types of components, making the system easily extensible.
- Simplifies code by abstracting away the complexities of the tree-like structures.

Drawbacks:

- If there's a need to restrict certain types of components from having children, the Composite pattern might introduce complications, as the default design allows for general compositions.

Conclusion

The Composite Design Pattern offers a robust mechanism for handling tree structures in software systems. By ensuring a consistent interface for both individual elements and their groupings, it promotes clarity, scalability, and maintainability. While it's particularly useful in scenarios with clear hierarchical structures, like UI designs or file systems, careful consideration is necessary to ensure it's the right fit for the specific problem domain.

Understanding its Real-world Applications

The Composite Design Pattern finds its roots in the desire to treat individual objects and compositions of objects uniformly. This pattern, categorized under structural design patterns, deals with object compositions in a manner where they can be treated similarly to individual objects. Its elegance lies in its ability to form tree-like structures, accommodating both composites and individual nodes in a seamless manner. Let's journey through some scenarios where the Composite pattern has been instrumental:

Graphic Systems and User Interface (UI) Components

One of the most classic applications of the Composite pattern is in graphical systems or UI components. Consider a graphic system where we have primitive shapes like circles, rectangles, and lines. Now, often, we might want to group certain shapes to form a complex figure. With the Composite pattern, we can treat both simple shapes (like circles) and complex figures (groups of shapes) uniformly. This uniformity allows operations like rendering, moving, or resizing to be applied seamlessly, whether it's a single shape or a composition.

Organizational Structures

Many applications require the modeling of organizational hierarchies or structures. For instance, in a company, we might have individual employees and teams or departments. A department could further consist of sub-departments. The Composite pattern is a perfect fit here, as it can represent both individual employees and departments under the same hierarchical structure, enabling operations like calculating total departmental expenses or total employees under a certain hierarchy level.

File and Directory Systems

File systems often consist of files and directories. Here, a directory can further contain files or sub-directories. Leveraging the Composite pattern, we can build a structure where both files and directories can be treated uniformly. This simplifies operations like searching for a particular item, calculating total storage, or implementing access permissions on the filesystem's hierarchy.

Menus and Sub-menus

Many software applications, particularly those with Graphical User Interfaces, have a menu system. Menus can have menu items, and these items might further have sub-menus. The Composite pattern simplifies the creation and management of such nested menus, ensuring uniform operations like displaying, enabling, or disabling menu items, regardless of their level in the menu hierarchy.

The Composite Design Pattern is a testament to the power of structural patterns in handling complex hierarchies and compositions. Its primary strength lies in its ability to provide a unified interface for individual objects and their compositions. This uniformity paves the way for scalability, ease of management, and a consistent approach to operations across hierarchies. Like all design patterns, the key is to discern its relevance to the problem at hand and apply it judiciously to harness its full potential.

3.4.2 Basic UML class diagram

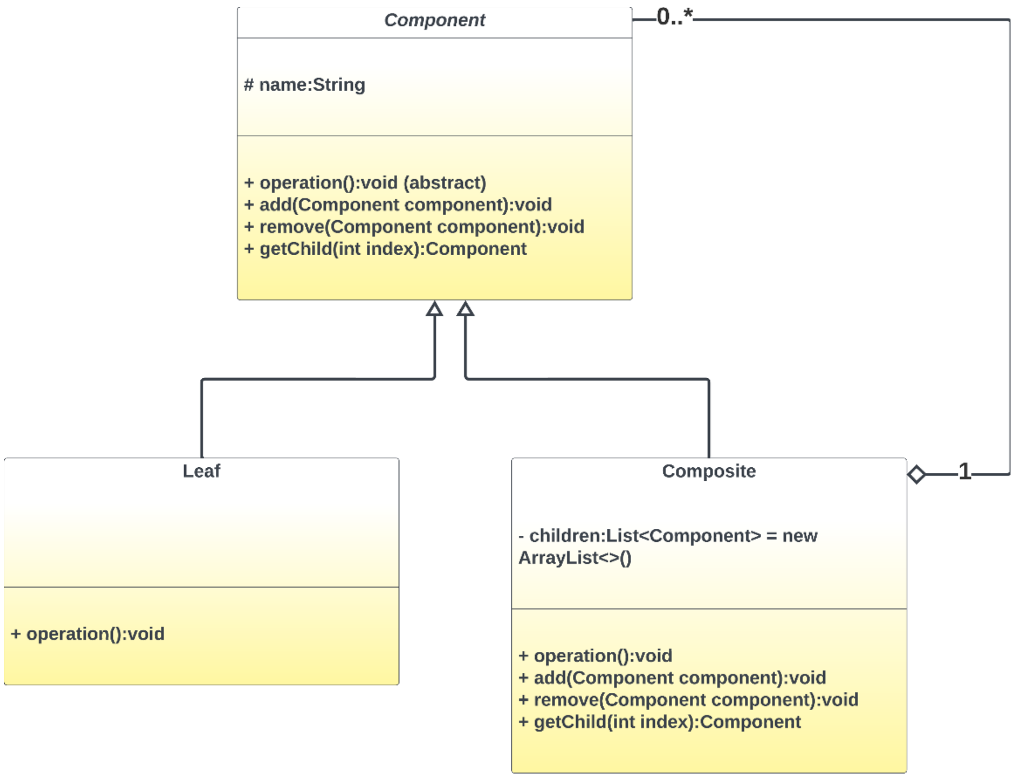

Component
# name:String
+ operation():void (abstract)
+ add(Component component):void
+ remove(Component component):void
+ getChild(int index):Component
0..*
Leaf
+ operation():void
Composite
- children:List<Component> = new ArrayList<>()
+ operation():void
+ add(Component component):void
+ remove(Component component):void
+ getChild(int index):Component
1

Figure 3. 4 Composite Design Pattern Basic UML class diagram

In this design:

- The `Component` abstract class defines an interface for all concrete objects (both composites and leaf nodes).

- The `Leaf` class represents end objects in the composition and inherits from the `Component` class.

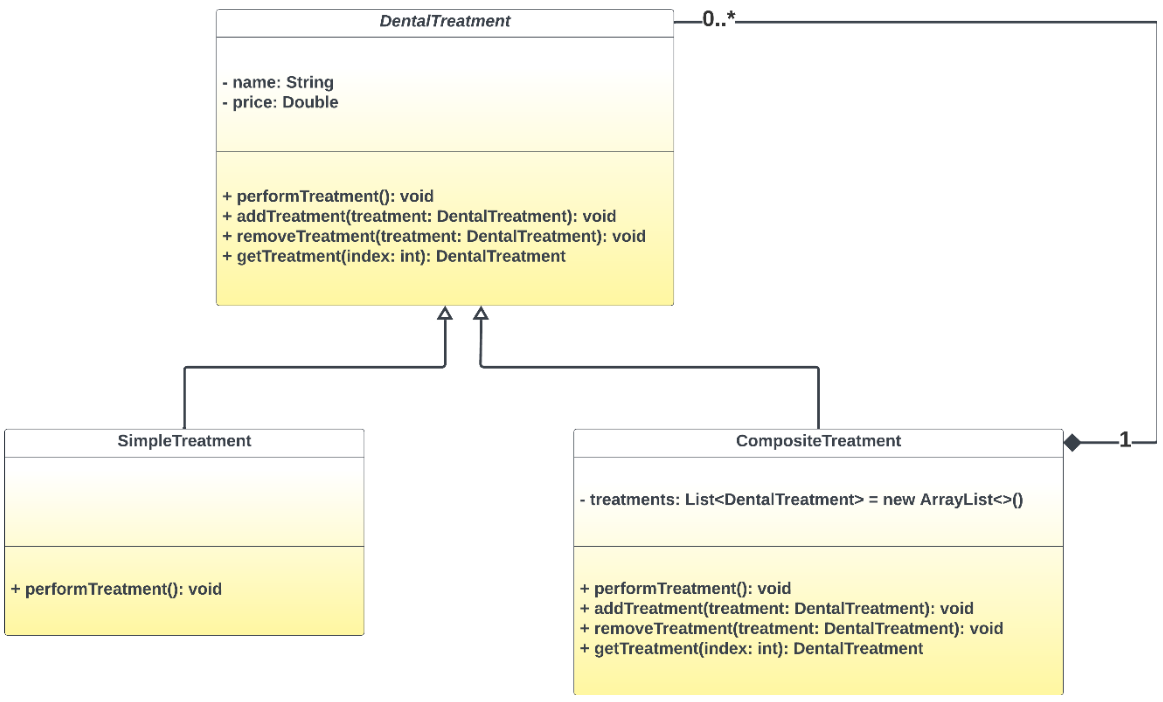
- The `Composite` class has child components, which can be other `Composite` objects or `Leaf` objects, allowing for the creation of tree structures.

The primary idea behind this pattern is that it allows us to treat both simple and complex objects uniformly. So, in scenarios where we need to operate over parts of the objects or the whole hierarchy of objects uniformly, the Composite pattern proves to be highly effective.

3.4.3 Creating UML class diagram for the scenario

Given the scenario, one suitable application of the Composite Design Pattern would be to represent the hierarchy of dental treatments available at the FDC. These treatments can be individual procedures or combinations of procedures. For instance, a dental implant could be considered a composite treatment because it might involve multiple procedures (like extraction, bone grafting, implant placement, etc.), while a simple tooth extraction might be considered a leaf or individual treatment.

The below diagram allows for the creation of complex treatments by combining simpler treatments or other composite treatments, reflecting the range of services offered at FDC.



DentalTreatment
- name: String
- price: Double
+ performTreatment(): void
+ addTreatment(treatment: DentalTreatment): void
+ removeTreatment(treatment: DentalTreatment): void
+ getTreatment(index: int): DentalTreatment
0..*
SimpleTreatment
+ performTreatment(): void
CompositeTreatment
- treatments: List<DentalTreatment> = new ArrayList<>()
+ performTreatment(): void
+ addTreatment(treatment: DentalTreatment): void
+ removeTreatment(treatment: DentalTreatment): void
+ getTreatment(index: int): DentalTreatment
1

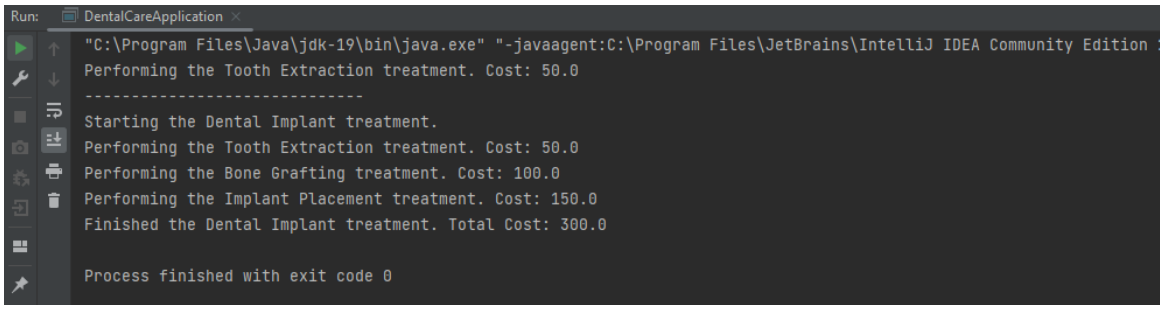
Figure 3. 5 Composite Design Pattern Creating UML class diagram for the scenario

3.4.4 Java code with an output for the illustrated UML class diagram of the scenario

```java
import java.util.ArrayList;
import java.util.List;

// The abstract component class for dental treatments
abstract class DentalTreatment {
    protected String name;
    protected double price;

    public DentalTreatment(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // This method is used to perform the treatment. Since it's
abstract, subclasses have to provide their own implementations
    public abstract void performTreatment();

    // These methods might not be meaningful for simple treatments, so
we provide default implementations
    public void addTreatment(DentalTreatment treatment) {
        throw new UnsupportedOperationException();
    }

    public void removeTreatment(DentalTreatment treatment) {
        throw new UnsupportedOperationException();
    }

    public DentalTreatment getTreatment(int index) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}

// Represents individual treatments that can't have other treatments
nested inside them
class SimpleTreatment extends DentalTreatment {

    public SimpleTreatment(String name, double price) {
        super(name, price);
    }

    @Override
    public void performTreatment() {
        System.out.println("Performing the " + name + " treatment. Cost:
" + price);
    }
}
```

```
// Represents composite treatments that can contain other treatments
class CompositeTreatment extends DentalTreatment {
    private List<DentalTreatment> treatments = new ArrayList<>();

    public CompositeTreatment(String name) {
        super(name, 0); // Initial price is set to 0, it will be
calculated based on the contained treatments
    }

    @Override
    public void performTreatment() {
        System.out.println("Starting the " + name + " treatment.");
        for(DentalTreatment treatment : treatments) {
            treatment.performTreatment();
        }
        System.out.println("Finished the " + name + " treatment. Total
Cost: " + getPrice());
    }

    // Add a treatment to this composite
    @Override
    public void addTreatment(DentalTreatment treatment) {
        treatments.add(treatment);
        price += treatment.getPrice(); // Update the total price when
adding a treatment
    }

    // Remove a treatment from this composite
    @Override
    public void removeTreatment(DentalTreatment treatment) {
        treatments.remove(treatment);
        price -= treatment.getPrice(); // Update the total price when
removing a treatment
    }

    @Override
    public DentalTreatment getTreatment(int index) {
        return treatments.get(index);
    }
}

public class DentalCareApplication {
    public static void main(String[] args) {
        // Creating some basic treatments
        SimpleTreatment toothExtraction = new SimpleTreatment("Tooth
Extraction", 50.0);
        SimpleTreatment boneGrafting = new SimpleTreatment("Bone
Grafting", 100.0);
        SimpleTreatment implantPlacement = new SimpleTreatment("Implant
Placement", 150.0);

        // Creating a composite treatment for dental implants
        CompositeTreatment dentalImplant = new
CompositeTreatment("Dental Implant");
        dentalImplant.addTreatment(toothExtraction);
        dentalImplant.addTreatment(boneGrafting);
        dentalImplant.addTreatment(implantPlacement);
```

```java
        // Performing the treatments
        toothExtraction.performTreatment();
        System.out.println("-----------------------------");
        dentalImplant.performTreatment();
    }
}
```


Run: DentalCareApplication

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
Performing the Tooth Extraction treatment. Cost: 50.0
-----------------------------
Starting the Dental Implant treatment.
Performing the Tooth Extraction treatment. Cost: 50.0
Performing the Bone Grafting treatment. Cost: 100.0
Performing the Implant Placement treatment. Cost: 150.0
Finished the Dental Implant treatment. Total Cost: 300.0

Process finished with exit code 0
```

Figure 3. 6 Composite Design Pattern Java code output

3.5 Observer Design Pattern

3.5.1 An Overview

Introduction

Within the realm of behavioral design patterns, the Observer Design Pattern emerges as an influential pattern, designed primarily to address situations where a change in one object necessitates an automatic update in one or more dependent objects. It's paramount in building systems where components need to maintain consistency, but without tight coupling, ensuring flexibility and scalability.

Core Principle

The quintessence of the Observer pattern is the establishment of a one-to-many dependency between objects. When the state of one object (often termed the "subject" or "observable") changes, all its dependents (the "observers") are notified and updated automatically. This decouples the observables from the observers, ensuring they can operate independently, yet stay in sync when necessary.

Rationale Behind the Pattern

In complex systems, various components might depend on the state or behavior of others. A direct link between these components can make the system rigid, hard to extend, and challenging to maintain. The Observer pattern offers a solution by allowing these components to subscribe (or "observe") other components. When an observed component undergoes a change, it merely notifies its subscribers, ensuring the system remains adaptable and modular.

Implementation Details

Central to the Observer pattern are a few critical roles:

- Observable (or Subject): This is the entity being watched. It maintains a list of its observers and provides methods to add, remove, and notify them.
- Observer: Represents the entities that need to be updated when the Observable changes. They provide an update method which the Observable calls to notify them of any changes.
- ConcreteObservable and ConcreteObserver: These are specific implementations of the Observable and Observer. The ConcreteObservable holds the state that the ConcreteObserver watches and gets updated about.

Advantages and Drawbacks

Advantages:

- Decoupling: The Observable knows only of the Observer's interface, not its concrete class, promoting a loose coupling.
- Dynamic Relationships: Observers can be added or removed at runtime, providing a dynamic relationship.
- Broad Notifications: A single change in the Observable can notify multiple observers, ensuring wide-scale updates when necessary.

Drawbacks:

- Over-notification: If not implemented judiciously, every minor change in the Observable can cascade notifications, potentially impacting performance.
- Unintended Side-effects: Since observers are notified of changes, they might execute operations that have unintended side-effects on other parts of the system.

Conclusion

The Observer Design Pattern is a testament to the power of decoupling in software systems. By allowing components to maintain dependency without tight coupling, it ensures that systems can evolve and adapt without massive overhauls. It's a staple in event-driven systems and forms the foundation for various modern-day frameworks and libraries. However, like all patterns, it's vital to ensure its applicability to the specific problem at hand, balancing between the need for decoupled notifications and the potential overheads it might introduce.

3.5.2 Understanding its Real-world Applications

The Observer Design Pattern, a cornerstone in the realm of behavioral patterns, builds upon the concept of a subject and its observers. The pattern fosters a one-to-many dependency such that when the subject's state alters, all its observers are seamlessly notified and updated. This mechanism promotes loose coupling, ensuring that the subject doesn’t need to know specifics about its observers. Now, let's dive deep into the tapestry of real-world scenarios where the Observer pattern shines brilliantly:

News Publications and Subscribers

Imagine a traditional newspaper publishing house. As soon as a new edition of the newspaper is printed, it's essential that all subscribers receive their copy. In this analogy, the newspaper publisher represents the subject, while the subscribers stand as observers. When a new edition (a change in state) is published, all subscribers (observers) are promptly notified and delivered their copies.

Stock Market Systems

In the dynamic world of stock markets, investors keenly track the price movements of their favored stocks. Here, the stock market system acts as the subject, and the investors are the observers. As soon as a particular stock undergoes price alterations, the system swiftly notifies all the investors tracking that stock, allowing them to make informed decisions.

Real-time Notifications in Digital Platforms

Modern digital platforms, be it social media sites, e-commerce platforms, or online gaming arenas, heavily rely on real-time notifications to enhance user experience. For instance, when a user receives a new message or a product is back in stock, a notification system (subject) sends real-time alerts to the relevant users (observers).

Weather Forecast Systems

Consider a weather forecasting system that multiple agencies rely on. The moment the system detects significant changes in weather patterns, it instantly notifies all registered agencies, enabling them to prepare and respond accordingly. The forecasting system embodies the subject, while the dependent agencies are its observers.

Monitoring and Analytics Tools

Many organizations use sophisticated monitoring tools to keep tabs on their infrastructure's health, performance metrics, or potential security breaches. These tools, acting as subjects, continuously monitor vast arrays of data points. The moment they detect anomalies or breaches, they alert the designated teams or systems (observers), triggering further investigations or corrective actions.

The Observer Design Pattern's beauty lies in its ability to foster decoupled interactions between subjects and observers. Its emphasis on promoting independent evolution of both sides, without compromising on seamless communication, has rendered it invaluable across various domains. As systems grow more interconnected and user expectations for real-time interactions surge, the Observer pattern's relevance only intensifies, standing as a beacon of effective system-to-user and system-to-system communication.

3.5.3 Basic UML class diagram

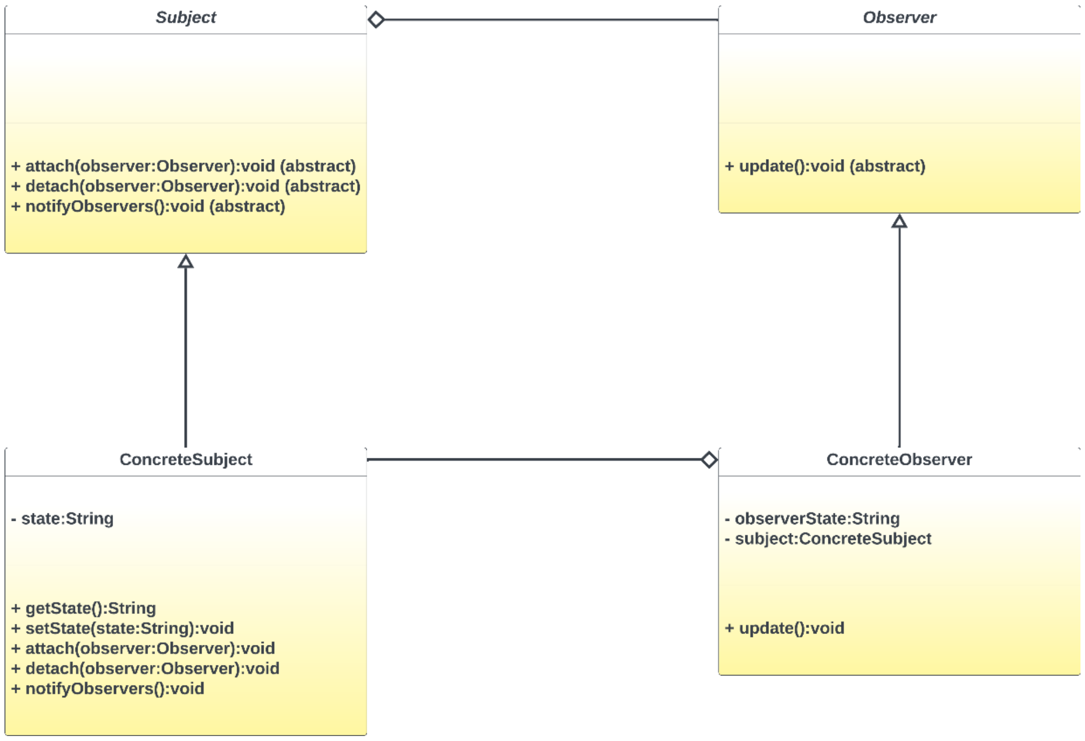

Subject
+ attach(observer:Observer):void (abstract)
+ detach(observer:Observer):void (abstract)
+ notifyObservers():void (abstract)
Observer
+ update():void (abstract)
ConcreteSubject
- state:String
+ getState():String
+ setState(state:String):void
+ attach(observer:Observer):void
+ detach(observer:Observer):void
+ notifyObservers():void
ConcreteObserver
- observerState:String
- subject:ConcreteSubject
+ update():void

Figure 3. 7 Observer Design Pattern Basic UML class diagram

In this design:

- The `Subject` abstract class provides the mechanism to maintain and update observers.

- `ConcreteSubject` is the real class that has the state and notifies the `Observer` objects when a change occurs.

- The `Observer` abstract class provides the updating interface for objects that need to be informed of changes in the `ConcreteSubject`.

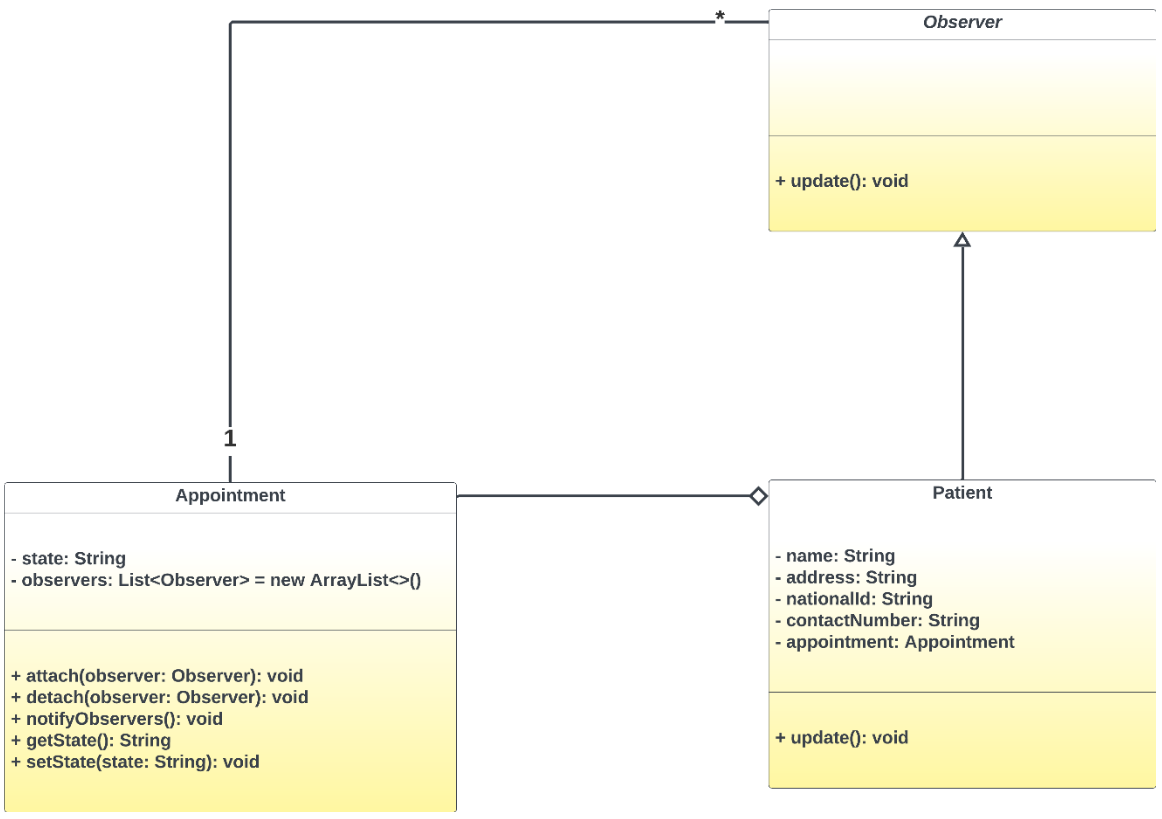
- `ConcreteObserver` keeps a reference to the `ConcreteSubject` object and implements the update mechanism that will get triggered upon state change.

This pattern is particularly useful when there's a need to maintain consistency between related objects without making them tightly coupled. The Observer pattern provides a loosely coupled design that's robust, flexible, and promotes a clear separation of concerns.

3.5.4 Creating UML class diagram for the scenario

Given the scenario, the Observer Design Pattern can be used to notify patients about changes to their appointments, for instance, if a treatment room becomes available earlier than scheduled or if there's a delay.

The below diagram would allow the 'Appointment' class to notify all attached 'Patient' objects about any changes in appointment status. Whenever the appointment's state changes, all observing patients will be updated, which can be particularly useful in cases of rescheduling, delays, or early availabilities.

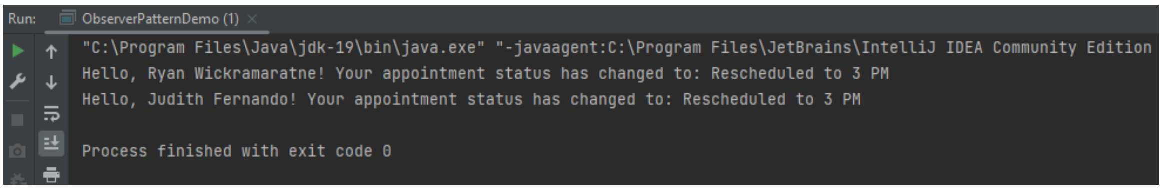


Observer
+ update(): void
*
1
Appointment
- state: String
- observers: List<Observer> = new ArrayList<>()
+ attach(observer: Observer): void
+ detach(observer: Observer): void
+ notifyObservers(): void
+ getState(): String
+ setState(state: String): void
Patient
- name: String
- address: String
- nationalId: String
- contactNumber: String
- appointment: Appointment
+ update(): void

Figure 3. 8 Observer Design Pattern Creating UML class diagram for the scenario

3.5.5 Java code with an output for the illustrated UML class diagram of the scenario

```java
import java.util.ArrayList;
import java.util.List;

// Observer interface that represents entities interested in appointment
state changes
interface Observer {
    void update();
}

// ConcreteSubject class that gets observed for state changes
class Appointment {
    private String state;
    private List<Observer> observers = new ArrayList<>();

    // Attach an observer
    public void attach(Observer observer) {
        observers.add(observer);
    }

    // Detach an observer
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    // Notify all observers about state changes
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }

    // Getter for state
    public String getState() {
        return state;
    }

    // Setter for state that triggers notification
    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }
}

// ConcreteObserver class that gets notified of changes
class Patient implements Observer {
    private String name;
    private String address;
    private String nationalId;
    private String contactNumber;
    private Appointment appointment;

    public Patient(String name, String address, String nationalId,
String contactNumber, Appointment appointment) {
        this.name = name;
        this.address = address;
        this.nationalId = nationalId;
        this.contactNumber = contactNumber;
        this.appointment = appointment;
        this.appointment.attach(this);  // Attach this patient as an
observer to the appointment
    }

    @Override
    public void update() {
        System.out.println("Hello, " + name + "! Your appointment status
has changed to: " + appointment.getState());
    }

    // Additional methods related to patient's details can be added
here...
}

public class ObserverPatternDemo {
    public static void main(String[] args) {
        // Create an appointment
        Appointment appointment1 = new Appointment();

        // Create patients and attach them to the appointment
        Patient RyanWick = new Patient("Ryan Wickramaratne", "84/3
Negombo", "975635465V", "0764170647", appointment1);
        Patient JudithFernando = new Patient("Judith Fernando", "181/124
Lane", "974674563V", "0764170485", appointment1);

        // Simulate changing the state of the appointment, which should
notify all attached patients
        appointment1.setState("Rescheduled to 3 PM");
    }
}
```



```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
Hello, Ryan Wickramaratne! Your appointment status has changed to: Rescheduled to 3 PM
Hello, Judith Fernando! Your appointment status has changed to: Rescheduled to 3 PM

Process finished with exit code 0
```

Figure 3. 9 Observer Design Pattern Java code output

3.6 OOP and Design Patterns Relationship

3.6.1 How Singleton design pattern enhances OOP concepts

The Singleton design pattern, which ensures that a particular class has only one instance and provides a global point of access to that instance, is an interesting example of the interplay between design principles and the foundational concepts of object-oriented programming (OOP). Below shown how the Singleton pattern enhances or leans into core OOP concepts:

Encapsulation:

At the heart of the Singleton design pattern is the principle of encapsulation. The Singleton pattern hides the instantiation logic of the singleton instance and exposes only the necessary mechanisms to access it. By making the constructor private, Singleton ensures that the class cannot be instantiated from outside. This encapsulation guarantees that no external class can create a new instance or change the current instance without going through the designated access method, usually called `getInstance()`. As we observed in the earlier `DentalScanRoomManager` example, the Singleton pattern effectively encapsulates the logic of creating or accessing the single instance of the class.

Abstraction:

Singleton, in its essence, abstracts the idea of single instantiation. While client classes or users interact with the Singleton's `getInstance()` method, they remain abstracted from the implementation details. Users don't need to know whether an instance already exists or if it needs to be created. This level of abstraction ensures that the client classes remain decoupled from the instantiation logic, emphasizing the OOP principle of abstraction wherein users interact with only the necessary functionalities, oblivious to the inner workings.

Polymorphism:

While the Singleton pattern doesn't directly embody polymorphism in the same way as patterns like Strategy or State, it doesn't restrict polymorphism either. If the Singleton class is part of an inheritance hierarchy and employs method overriding, it can still leverage polymorphic behavior. However, the Singleton's emphasis on a single instance can sometimes pose challenges when coupled with inheritance, so care is needed.

Inheritance:

Singleton classes can inherit from other classes or be inherited. However, one must tread with caution. If a Singleton class is subclassed without care, the subclass could inadvertently create a new instance, thereby violating the Singleton principle. To ensure that the Singleton property is maintained, the pattern would need to be carefully tailored when combined with inheritance.

Cohesion and Modularity:

The Singleton pattern promotes high cohesion. The Singleton's responsibility is clear: manage a single instance and provide global access. This clarity and singularity of purpose ensure that the Singleton class remains modular and adheres strictly to its intended role, which is in line with the OOP principle of creating cohesive and modular units of functionality.

Global State Management:

One of the reasons for employing a Singleton, as in the `DentalScanRoomManager` scenario, is to manage global state. While global state is often discouraged in OOP because of potential side effects and maintainability issues, there are legitimate use cases for it. The Singleton pattern offers a structured way to handle global state within the OOP paradigm, ensuring that global state is accessed and modified in a controlled manner.

In conclusion, the Singleton design pattern, while seemingly simple, embodies the spirit of object-oriented programming. It leans heavily into principles like encapsulation and abstraction, ensuring that classes are designed with clear boundaries and responsibilities. While it has its nuances when combined with polymorphism and inheritance, the Singleton pattern, when applied judiciously, reinforces the structured and disciplined approach that OOP advocates for software design.

3.6.2 How Composite design pattern enhances OOP concepts

The Composite design pattern offers a structured approach to treating individual objects and compositions of objects uniformly. In the realm of object-oriented programming (OOP), the Composite pattern stands as a testament to the versatility and power of OOP concepts. By helping developers compose objects into tree structures to represent part-whole hierarchies, this pattern allows clients to treat individual objects and compositions uniformly. Below shown how the Composite pattern enhances and intertwines with the core tenets of OOP:

Encapsulation:

The Composite pattern excels in preserving encapsulation. Each component (whether a leaf or composite) encapsulates its behavior and state. Clients interact with the components using a common interface, remaining oblivious to the intricate details of whether they're dealing with a simple element or a complex composition. In the earlier discussion about dental treatments, we witnessed this principle in action: both simple and composite treatments encapsulate their respective behaviors, yet they can be accessed and utilized uniformly by the client.

Abstraction:

A shining facet of the Composite pattern is its reliance on abstraction. The pattern abstracts the complexity of the tree structure behind a unified interface, simplifying client interaction. When a client wants a treatment performed, it doesn't need to discern between a simple treatment or a composite one; it merely calls the `performTreatment()` method. This level of abstraction ensures that the underlying complexity of managing hierarchies and compositions remains hidden, providing a streamlined and consistent client experience.

Polymorphism:

The Composite pattern thrives on polymorphism. Both the leaf and composite components implement a common interface, allowing them to be treated polymorphically. This ensures that methods defined in the component interface can be executed regardless of whether the object is a simple element or a composite. The pattern leverages this polymorphic behavior to allow seamless interactions with the components, highlighting the object-oriented principle that emphasizes the importance of objects being able to take on many forms.

Inheritance:

The foundation of the Composite pattern rests on inheritance. In the dental treatment example, both the `SimpleTreatment` and `CompositeTreatment` classes inherit from the `DentalTreatment` abstract class. This inheritance ensures a consistent interface while allowing for specialized behaviors to be defined in the subclasses. It showcases the OOP principle of basing an object's definition on another object's definition, reusing common behaviors, and introducing specific ones.

Recursion:

While recursion isn't strictly an OOP concept, the Composite pattern's ability to recursively include components within components exemplifies the power of OOP in modeling real-world scenarios. Composite objects can hold other composites, creating deeply nested structures. This recursive nature allows for the creation of intricate hierarchies while still preserving uniformity in client interactions.

High Cohesion:

The Composite pattern promotes high cohesion, a pivotal OOP concept. Each class in the pattern, whether it's a leaf or a composite, has a clear and distinct responsibility. This ensures that each class is focused on a specific task, leading to a modular and maintainable design. The clarity in roles, be it handling individual tasks or managing compositions, ensures that the design remains coherent and well-organized.

In wrapping up, the Composite design pattern is a masterclass in how OOP concepts can be harnessed to model and manage complex hierarchies and relationships in a systematic and consistent manner. By drawing heavily on the foundational pillars of OOP like encapsulation, abstraction, polymorphism, and inheritance, the Composite pattern showcases the elegance and robustness of object-oriented designs in handling real-world challenges, such as the intricate hierarchies of dental treatments we discussed earlier.

3.6.3 How Observer design pattern enhances OOP concepts

The Observer design pattern is an embodiment of the power and flexibility intrinsic to object-oriented programming (OOP). Commonly used to implement distributed event-handling systems, the Observer pattern defines a dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This pattern is particularly beneficial when a distinction between core and peripheral functionalities is necessary, allowing the core component to focus on its primary responsibility while still notifying other components of changes. Below shown how the Observer pattern magnifies and intertwines with core OOP tenets:

Encapsulation:

At the heart of the Observer pattern is the preservation of encapsulation. Each Observer maintains its state and behavior, responding to notifications as it sees fit. Meanwhile, the Subject (the entity being observed) doesn't need explicit knowledge of its Observers, just an abstract interface. In our earlier discussion about notifying patients of appointment changes, the appointment class (Subject) encapsulates its state, and each patient (Observer) encapsulates its reaction mechanism, thus preserving the principle of data hiding and exposing only necessary functionalities.

Abstraction:

Abstraction is core to the Observer pattern. The pattern typically relies on an abstract Observer interface, allowing for a range of concrete Observer implementations. This abstract layer ensures that the Subject interacts solely with the generalized Observer interface, remaining agnostic to the specific implementations of its Observers. The notion that multiple patient objects (or other Observers) can monitor an appointment without the appointment needing explicit knowledge of them is a testament to the power of abstraction.

Polymorphism:

The Observer pattern is a celebration of polymorphism. Each concrete Observer implements a common interface, which ensures that, despite differences in their specific reactions to notifications, they can be treated uniformly by the Subject. When the Subject's state changes and it's time to notify the Observers, it iteratively invokes the update method on each Observer, allowing each to react in its unique way. This dynamic method dispatch, based on the actual object and not its declared type, is a perfect example of the polymorphic capabilities of OOP.

Inheritance:

Central to the Observer pattern is the use of inheritance. The concrete Observers inherit from a common Observer interface, which ensures a consistent method signature for updates, even as each Observer can override this to provide specialized behavior. By promoting a shared interface while allowing for variations in implementations, the Observer pattern encapsulates the OOP principle of building upon shared blueprints.

Loose Coupling:

While not strictly limited to OOP, the principle of loose coupling is emphasized and realized through the Observer pattern. The Subject and Observers remain independent, interacting only through a shared interface. Changes to the Subject need not impact Observers, and new Observers can be added without modifying the Subject. In the context of the dental appointment system we discussed, this means that new mechanisms (like sending email notifications or using a different communication channel) can be introduced with minimal disruption to the appointment management process.

Modularity & Scalability:

By distinguishing core functionalities (the Subject) from peripheral ones (the Observers), the Observer pattern promotes modularity. Systems can scale seamlessly; adding new Observers or modifying existing ones doesn't necessitate alterations to the core Subject.

For instance, if the dental system were to introduce a new notification mechanism or integrate with a third-party application, it could do so without modifying the core appointment management logic.

In essence, the Observer design pattern stands as a testament to the adaptability and coherence of OOP principles in managing complex, dynamic interactions. It skillfully interweaves encapsulation, abstraction, polymorphism, and inheritance to create systems that are both robust and flexible. Through its judicious application, like in the scenario of the dental appointment system, developers can craft systems that adapt and evolve without compromising on structural integrity or consistency.

3.6.4 How design patterns enhance OOP concepts

Design patterns, often described as time-tested solutions to recurring software design problems, act as templates that can be adapted and applied across diverse contexts. Fundamentally, they are manifestations of the best practices within the realm of object-oriented programming (OOP). By tapping into the foundational principles of OOP, design patterns amplify its strengths, fostering more maintainable, flexible, and scalable software systems. Below shown how design patterns enrich and enhance the key tenets of OOP:

Encapsulation:

Encapsulation, the bedrock of OOP, involves bundling data and methods that operate on that data within a single unit, thereby restricting unauthorized access and modification. Many design patterns, like the Singleton pattern we discussed earlier, strengthen encapsulation. By ensuring that a particular class has only one instance and providing a global point of access to it, the Singleton pattern safeguards the state and behavior of that singular instance, enhancing encapsulation.

Abstraction:

Abstraction permits the extraction of essential features of an object while omitting its intricacies. Design patterns frequently utilize and bolster abstraction. For instance, the Factory pattern, which isn't discussed previously but is a cornerstone pattern, abstracts the process of object creation. By introducing an intermediary object to manage creation, it abstracts away the specifics, allowing the system to be agnostic about which class's objects are being created.

Polymorphism:

Polymorphism enables objects to be treated as instances of their parent class rather than their actual type. The Strategy pattern, for instance, employs polymorphism by defining a family of algorithms, encapsulating each one, and making them interchangeable. By leveraging polymorphism, design patterns like Strategy allow behaviors to be switched on the fly, thereby bestowing the system with dynamic capabilities.

Inheritance:

Inheritance permits the creation of a new class that is based on an existing class, inheriting attributes and behaviors. The Template Method pattern showcases the power of inheritance. By defining the structure of an algorithm and allowing subclasses to implement specific steps, this pattern fosters reusability and consistency. Similarly, as we explored, the Composite and Observer patterns also hinge on inheritance, promoting a shared blueprint while allowing for specialized implementations.

Cohesion & Decoupling:

While not exclusively OOP principles, high cohesion (having related functionalities within one unit) and low coupling (minimizing dependencies between units) are desired in object-oriented designs. Many design patterns, such as Observer and Composite discussed earlier, aim to enhance these qualities. The Observer pattern ensures that core functionalities remain decoupled from peripheral ones, whereas the Composite pattern emphasizes high cohesion by grouping related functionalities.

Modularity & Scalability:

Design patterns inherently endorse modularity. By compartmentalizing specific functionalities or behaviors, patterns like Composite enable independent development, testing, and scalability. Systems imbued with such patterns can expand effortlessly, with new components or behaviors being introduced without disrupting existing functionalities.

In summary, design patterns and OOP are inextricably intertwined, with the former acting as a catalyst to amplify the strengths of the latter. They represent distilled wisdom from countless software projects, encapsulating the optimal ways to address recurring challenges. By adhering to design patterns, developers not only embrace the best practices within OOP but also lay the groundwork for software that's robust, maintainable, and future-proof. As showcased in our discussions on Singleton, Composite, and Observer patterns, these patterns not only employ OOP principles but also elevate them, ensuring that software systems remain agile in the face of evolving requirements and challenges.

3.6.5 Singleton Design Pattern: How the scenario-related code leverages the principles of OOP

```
// Define the Dentist class which will be associated with the
DentalScanRoomManager
class Dentist {
    private String name;

    // Constructor
    public Dentist(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    // ... other attributes and methods relevant to the Dentist
}

// Singleton class for managing the Dental Scan Room
public class DentalScanRoomManager {

    // Static attribute that holds the single instance of the class
    private static DentalScanRoomManager instance = null;

    // Attribute for storing the dentist currently assigned to the scan
room
    private Dentist assignedDentist = null;

    // Private constructor to prevent instantiation from other classes
    private DentalScanRoomManager() {
        // Nothing to initialize here for now
    }

    // Public method to provide access to the single instance of the
class
    public static synchronized DentalScanRoomManager getInstance() {
        // If instance hasn't been initialized yet, create a new one
        if (instance == null) {
            instance = new DentalScanRoomManager();
        }

        // Return the single instance
        return instance;
    }

    // Assign a dentist to the scan room
    public void assignDentist(Dentist dentist) {
        this.assignedDentist = dentist;
        System.out.println(dentist.getName() + " has been assigned to
the Dental Scan Room.");
    }

    // Release the dentist from the scan room (e.g. after finishing a
scan)
    public void releaseDentist() {
        System.out.println(assignedDentist.getName() + " has been
released from the Dental Scan Room.");
        this.assignedDentist = null;
    }

    // Get the dentist currently assigned to the scan room
    public Dentist getAssignedDentist() {
        return assignedDentist;
    }

    public static void main(String[] args) {
        // Sample usage
        DentalScanRoomManager manager =
DentalScanRoomManager.getInstance();
        Dentist drJohn = new Dentist("Dr. Ryan");

        manager.assignDentist(drJohn);
        System.out.println("Currently assigned dentist: " +
manager.getAssignedDentist().getName());

        manager.releaseDentist();
    }
}
```

The given code provides a practical demonstration of the Singleton Design Pattern in the context of managing a Dental Scan Room, which only permits one dentist to be assigned at any given time. By weaving together the principles of object-oriented programming (OOP), this design exhibits both the strength of OOP and the strategic power of the Singleton pattern. Below shown how the code leverages these OOP principles.

Encapsulation:

Encapsulation entails bundling together data (attributes) and methods that manipulate this data into a single cohesive unit, while also restricting direct access to some of the object's components. This principle is vividly evident in the `DentalScanRoomManager` class. The attributes, like the `instance` and `assignedDentist`, are declared private, thereby ensuring they cannot be directly accessed or modified from outside the class. Instead, public methods such as `assignDentist`, `releaseDentist`, and `getAssignedDentist` are provided, allowing controlled access and modifications to these attributes. This encapsulation ensures that the integrity of the room's state is always maintained, particularly that only one dentist is assigned at a time.

Abstraction:

Abstraction focuses on exposing only the essential features of an object, while hiding the inner workings or complexities. The `DentalScanRoomManager` exposes methods like `assignDentist` and `releaseDentist` which allow users to perform key actions without needing to know the inner details of how these actions are implemented. This abstraction simplifies interaction with the manager and allows the underlying implementation details to be changed without affecting the users of the class.

Polymorphism:

While not showcased directly in the provided code, polymorphism plays a subtle role. The system can be expanded to include different types of dental professionals (e.g., Orthodontists, Periodontists, etc.), all inheriting from a potential parent class `DentalProfessional`. The Singleton manager could then manage any subclass type through polymorphism, treating them all as the parent type, and allowing for dynamic assignment.

Inheritance:

In the current implementation, inheritance is implied rather than directly applied. The `Dentist` class has been kept simple, but one could envision a hierarchy of specialized dental professionals, each inheriting from a base class, thereby promoting code reuse and establishing a clear hierarchy of classes.

Cohesion:

The code exhibits high cohesion, as every class is tailored to a specific responsibility. The `Dentist` class revolves around attributes and methods associated with a dentist, while the `DentalScanRoomManager` is solely focused on managing the assignment and release of dentists in the scan room. This clear division of responsibilities ensures that the system remains modular and maintainable.

Singleton Design Principle:

Building on the earlier discussions, the Singleton principle ensures that a class has only one instance and provides a global point to access it. The `DentalScanRoomManager` encapsulates this principle perfectly. The private constructor ensures no other instances can be created externally. The synchronized `getInstance` method ensures thread safety, providing a reliable global access point for the sole instance.

To conclude, the provided code for the Dental Scan Room management using the Singleton Design Pattern seamlessly marries foundational OOP principles with the specialized Singleton principle. This fusion not only ensures a robust, efficient, and maintainable system but also aptly demonstrates how design patterns are instrumental in amplifying the strengths of OOP to address real-world scenarios.

3.6.6 Composite Design Pattern: How the scenario-related code leverages the principles of OOP

```
import java.util.ArrayList;
import java.util.List;

// The abstract component class for dental treatments
abstract class DentalTreatment {
    protected String name;
    protected double price;

    public DentalTreatment(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // This method is used to perform the treatment. Since it's
abstract, subclasses have to provide their own implementations
    public abstract void performTreatment();

    // These methods might not be meaningful for simple treatments, so
we provide default implementations
    public void addTreatment(DentalTreatment treatment) {
        throw new UnsupportedOperationException();
    }

    public void removeTreatment(DentalTreatment treatment) {
        throw new UnsupportedOperationException();
    }

    public DentalTreatment getTreatment(int index) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}

// Represents individual treatments that can't have other treatments
nested inside them
class SimpleTreatment extends DentalTreatment {

    public SimpleTreatment(String name, double price) {
        super(name, price);
    }

    @Override
    public void performTreatment() {
        System.out.println("Performing the " + name + " treatment. Cost:
" + price);
```

```java
    }
}

// Represents composite treatments that can contain other treatments
class CompositeTreatment extends DentalTreatment {
    private List<DentalTreatment> treatments = new ArrayList<>();

    public CompositeTreatment(String name) {
        super(name, 0); // Initial price is set to 0, it will be
calculated based on the contained treatments
    }

    @Override
    public void performTreatment() {
        System.out.println("Starting the " + name + " treatment.");
        for(DentalTreatment treatment : treatments) {
            treatment.performTreatment();
        }
        System.out.println("Finished the " + name + " treatment. Total
Cost: " + getPrice());
    }

    // Add a treatment to this composite
    @Override
    public void addTreatment(DentalTreatment treatment) {
        treatments.add(treatment);
        price += treatment.getPrice(); // Update the total price when
adding a treatment
    }

    // Remove a treatment from this composite
    @Override
    public void removeTreatment(DentalTreatment treatment) {
        treatments.remove(treatment);
        price -= treatment.getPrice(); // Update the total price when
removing a treatment
    }

    @Override
    public DentalTreatment getTreatment(int index) {
        return treatments.get(index);
    }
}

public class DentalCareApplication {
    public static void main(String[] args) {
        // Creating some basic treatments
        SimpleTreatment toothExtraction = new SimpleTreatment("Tooth
Extraction", 50.0);
        SimpleTreatment boneGrafting = new SimpleTreatment("Bone
Grafting", 100.0);
        SimpleTreatment implantPlacement = new SimpleTreatment("Implant
Placement", 150.0);

        // Creating a composite treatment for dental implants
        CompositeTreatment dentalImplant = new
CompositeTreatment("Dental Implant");
        dentalImplant.addTreatment(toothExtraction);
        dentalImplant.addTreatment(boneGrafting);
        dentalImplant.addTreatment(implantPlacement);

        // Performing the treatments
        toothExtraction.performTreatment();
        System.out.println("------------------------------");
        dentalImplant.performTreatment();
    }
}
```

The provided code outlines a dental care scenario that makes excellent use of the Composite Design Pattern. The Composite Pattern allows us to compose objects into tree-like structures to represent whole-part hierarchies, wherein individual objects and compositions of objects are treated uniformly. This design fits naturally within the framework of Object-Oriented Programming (OOP) principles, and here's how the scenario does justice to those principles:

Encapsulation:

The foundation of OOP, encapsulation, is about bundling data (attributes) and methods that operate on the data into single units called classes, while also controlling the accessibility of these attributes. The `DentalTreatment` class, as well as its subclasses, demonstrate this principle. Inside these classes, attributes like `name` and `price` are protected, ensuring controlled access. Each class also provides methods that act upon these attributes, ensuring that the behavior and state of the objects are tightly coupled.

Abstraction:

Abstraction allows us to hide complex implementation details and show only the essential features of an object. The `DentalTreatment` class offers a blueprint for dental treatments without specifying how each method's exact behavior should be. For instance, the `performTreatment` method is abstract, compelling subclasses to provide their own specific implementations. This level of abstraction means that the user of these classes doesn't need to know the intricacies of each treatment; they just need to know that treatments can be performed.

Polymorphism:

A cornerstone of OOP, polymorphism allows objects of different classes to be treated as instances of the same class through inheritance. This principle shines in the provided code. Both `SimpleTreatment` and `CompositeTreatment` are subclasses of `DentalTreatment`, yet they provide distinct implementations for the `performTreatment` method. When we invoke the `performTreatment` method on an object, the correct version of the method (either from `SimpleTreatment` or `CompositeTreatment`) is called, depending on the object's actual type.

Inheritance:

The code utilizes inheritance to build a hierarchy of dental treatments. The `DentalTreatment` class acts as a parent, providing a generalized structure and behavior. The subclasses, namely `SimpleTreatment` and `CompositeTreatment`, inherit attributes and behaviors from this parent class, promoting code reuse. They also extend or override certain behaviors, demonstrating the flexibility offered by inheritance.

Composite Design Principle:

This is the heart of the entire design. The `CompositeTreatment` class can contain multiple `DentalTreatment` objects, forming a tree-like structure. This means we can have complex treatments made up of simpler treatments. The beauty of this design is that both simple and composite treatments are treated uniformly. This uniformity simplifies client code, as seen in the `DentalCareApplication` class, where the `performTreatment` method is called on both simple and composite treatments without any distinction.

Cohesion:

Every class in the design has a clear, singular responsibility. The `SimpleTreatment` handles individual treatments, `CompositeTreatment` deals with the composition of treatments, and the `DentalCareApplication` acts as a client to execute the treatments. Such a high level of cohesion makes the code more maintainable, understandable, and expandable.

To wrap up, the dental care scenario leveraging the Composite Design Pattern is a masterclass in Object-Oriented Programming. The design beautifully intertwines the foundational principles of OOP to create a system that's robust, extensible, and intuitive. Through this pattern, the complexities of real-world scenarios, like dental treatments, can be elegantly represented, showcasing the immense potential and flexibility OOP offers in software design.

3.6.7 Observer Design Pattern: How the scenario-related code leverages the principles of OOP

```
import java.util.ArrayList;
import java.util.List;

// Observer interface that represents entities interested in appointment
state changes
interface Observer {
    void update();
}

// ConcreteSubject class that gets observed for state changes
class Appointment {
    private String state;
    private List<Observer> observers = new ArrayList<>();

    // Attach an observer
    public void attach(Observer observer) {
        observers.add(observer);
    }

    // Detach an observer
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    // Notify all observers about state changes
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }

    // Getter for state
    public String getState() {
        return state;
    }

    // Setter for state that triggers notification
    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }
}

// ConcreteObserver class that gets notified of changes
class Patient implements Observer {
    private String name;
    private String address;
    private String nationalId;
    private String contactNumber;
    private Appointment appointment;

    public Patient(String name, String address, String nationalId,
String contactNumber, Appointment appointment) {
        this.name = name;
        this.address = address;
        this.nationalId = nationalId;
        this.contactNumber = contactNumber;
        this.appointment = appointment;
        this.appointment.attach(this);  // Attach this patient as an
observer to the appointment
    }

    @Override
    public void update() {
        System.out.println("Hello, " + name + "! Your appointment status
has changed to: " + appointment.getState());
    }

    // Additional methods related to patient's details can be added
here...
}

public class ObserverPatternDemo {
    public static void main(String[] args) {
        // Create an appointment
        Appointment appointment1 = new Appointment();

        // Create patients and attach them to the appointment
        Patient RyanWick = new Patient("Ryan Wickramaratne", "84/3
Negombo", "975635465V", "0764170647", appointment1);
        Patient JudithFernando = new Patient("Judith Fernando", "181/124
Lane", "974674563V", "0764170485", appointment1);

        // Simulate changing the state of the appointment, which should
notify all attached patients
        appointment1.setState("Rescheduled to 3 PM");
    }
}
```

The Observer Design Pattern, as demonstrated in the provided dental appointment scenario, is a perfect exemplification of how Object-Oriented Programming (OOP) principles can be intertwined to create a robust and interactive system. The design promotes a decoupled architecture where objects can interact without being explicitly dependent on each other. Below shown how this code captures the essence of OOP principles:

Encapsulation:

Encapsulation is the practice of bundling data and the methods that operate on that data into a single unit, while also restricting direct access to certain parts of the object. The `Appointment` and `Patient` classes encapsulate their internal state and expose only what's necessary through public methods. For instance, the `Appointment` class provides methods like `attach`, `detach`, and `setState` to manipulate its internal list of observers and the state, ensuring that no external entity can alter this state without going through the defined interfaces.

Abstraction:

The essence of abstraction lies in simplifying complex real-world entities into their most basic forms in the software. The code defines an `Observer` interface with a single method, `update`, an abstract representation of entities interested in changes to an `Appointment`. Through this abstraction, various objects (like `Patient` in this case) can implement the `Observer` interface and become interested parties to appointment state changes without any unnecessary complications.

Polymorphism:

Polymorphism promotes the ability of objects to take on multiple forms. The `Observer` interface introduces polymorphism into the design. Any class that implements the `Observer` interface will provide its own version of the `update` method. For example, the `Patient` class has its unique implementation, and if there were other observers, they too could provide their custom implementations. When the `notifyObservers` method is called in `Appointment`, the appropriate `update` method is invoked for each registered observer, even though the actual type of each observer might be different.

Inheritance:

While the code mainly demonstrates interface-based inheritance, it still adheres to the inheritance principle of OOP. The `Patient` class inherits the abstract nature of the `Observer` interface, committing to provide a concrete implementation for the `update` method. This ensures that all classes implementing this interface will have a consistent method to respond to state changes, promoting a structured and predictable design.

Loose Coupling:

The Observer Pattern inherently promotes loose coupling, which, though not a direct OOP principle, is a crucial software design best practice. In this scenario, the `Appointment` (subject) class is decoupled from the `Patient` (observer) class. Changes to the observer's implementation or the introduction of new observer types won't necessitate changes in the subject class. This separation ensures that modifications in one part of the system have minimal ripple effects on others.

Cohesion:

Each class in this scenario is highly cohesive, maintaining a single responsibility. The `Appointment` class is solely responsible for managing its state and notifying observers of any changes. In contrast, the `Patient` class, beyond its role as an observer, encapsulates the properties and behaviors pertinent to a patient. This clear separation ensures that classes remain modular, maintainable, and purpose-driven.

In summary, the provided Observer Design Pattern-based dental appointment scenario is a compelling showcase of Object-Oriented Programming's power and versatility. The architecture allows for flexibility and extensibility while ensuring that each class remains modular and single-purposed. By weaving together the foundational principles of OOP, the design ensures a scalable and adaptable system that can cater to evolving requirements with minimal disruptions.

Task 4

4.1 Scenario 1

4.1.1 Selection of Design Pattern

For the given scenario, the **Singleton Design Pattern** is the most appropriate choice. The Singleton pattern ensures that a particular class has only one instance and provides a global point of access to this instance. Given that FDC will own just one dental scan machine, it makes sense to use this pattern to ensure that only one instance of the machine class is ever created in the system.

4.1.2 Draw the Class Diagram

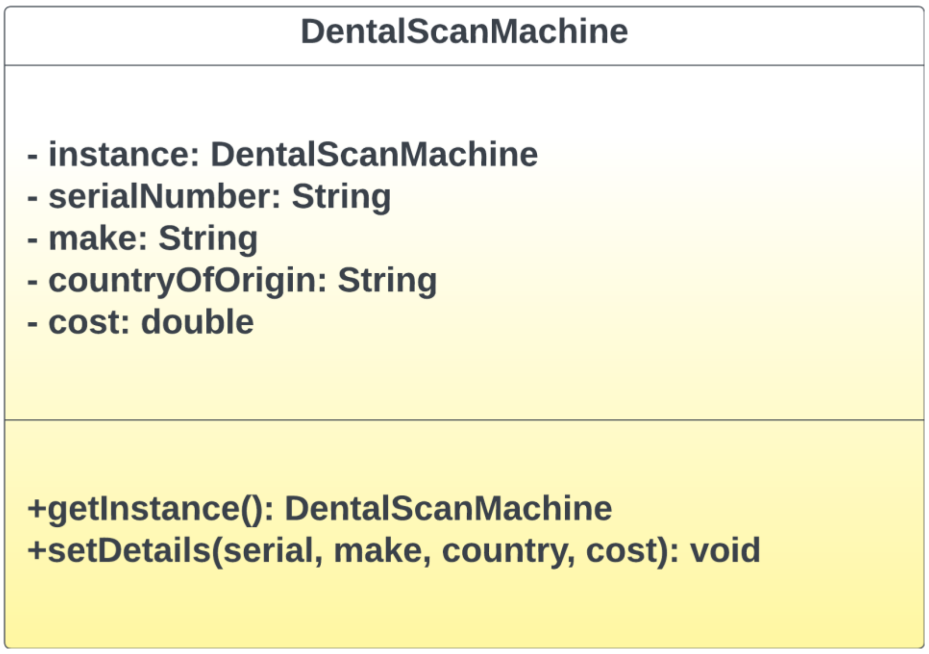

DentalScanMachine
- instance: DentalScanMachine
- serialNumber: String
- make: String
- countryOfOrigin: String
- cost: double
+getInstance(): DentalScanMachine
+setDetails(serial, make, country, cost): void

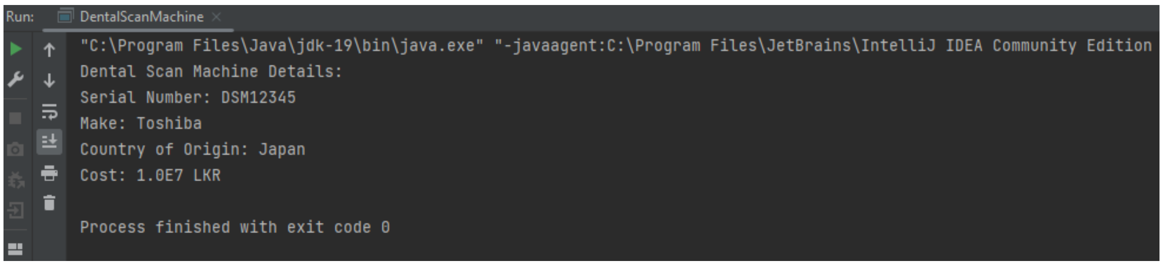
Figure 4. 1 Draw the Class Diagram for the scenario 1

4.1.3 Code Development

Code:

```
public class DentalScanMachine {

    // The sole instance of the class, initialized as null initially
    private static DentalScanMachine instance = null;

    // Attributes of the DentalScanMachine
    private String serialNumber;
    private String make;
    private String countryOfOrigin;
    private double cost;

    // Private constructor to ensure no other instance can be created
    private DentalScanMachine() {}

    // Public method to get the single instance of DentalScanMachine
    public static DentalScanMachine getInstance() {
        // If instance hasn't been created, create it
        if(instance == null) {
            instance = new DentalScanMachine();
        }
        return instance;
    }

    // Method to set the details for the DentalScanMachine
    public void setDetails(String serialNumber, String make, String
countryOfOrigin, double cost) {
        this.serialNumber = serialNumber;
        this.make = make;
        this.countryOfOrigin = countryOfOrigin;
        this.cost = cost;
    }

    // A simple method to display the details of the machine
    public void displayDetails() {
        System.out.println("Dental Scan Machine Details:");
        System.out.println("Serial Number: " + serialNumber);
        System.out.println("Make: " + make);
        System.out.println("Country of Origin: " + countryOfOrigin);
        System.out.println("Cost: " + cost + " LKR");
    }

    public static void main(String[] args) {
        // Get the instance of DentalScanMachine
        DentalScanMachine machine = DentalScanMachine.getInstance();

        // Set the details for the machine
        machine.setDetails("DSM12345", "Toshiba", "Japan", 10000000.0);

        // Display the details
        machine.displayDetails();
        }
}
```

Output:



```
Run:    DentalScanMachine
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
Dental Scan Machine Details:
Serial Number: DSM12345
Make: Toshiba
Country of Origin: Japan
Cost: 1.0E7 LKR

Process finished with exit code 0
```

Figure 4. 2 Java code output of scenario 1

4.1.4 Justification for selected Design Pattern

In the context of FDC's need for a singular, state-of-the-art dental scan machine, the Singleton design pattern emerges as an ideal solution. This design pattern ensures that a class has only one instance throughout a program's lifecycle and provides a global point of access to this instance. By employing the Singleton pattern, we can ensure that no matter how or where we attempt to instantiate the `DentalScanMachine` class, only one unique instance of it will ever exist in the system.

There are several reasons to justify why this design choice aligns perfectly with the scenario. FDC's dental scan machine is not just any ordinary equipment—it's described as a very expensive and superior device when compared to traditional dental X-ray machines. Given the significant investment and the uniqueness of the device, it's pivotal that the system mirrors this real-world scenario by ensuring that only one digital representation of the machine exists. This not only provides clarity but also avoids potential complications or errors that could arise from having multiple instances or representations of such a vital piece of equipment.

Additionally, the real-world context clearly states that FDC will possess only one such machine in the foreseeable future. As software models often aim to be a close representation of real-world systems, the Singleton pattern ensures that the software model remains faithful to this real-world constraint.

Furthermore, by utilizing the Singleton pattern, we're also introducing efficiency. The system won't have to repeatedly create new instances or check for the existence of other instances. Instead, any part of the program requiring access to the dental scan machine's attributes or methods can easily and reliably do so through the singular instance, ensuring consistent and unified data access and modification.

Moreover, in the practical world, the centralized control over the machine's instance can offer other advantages. Should there ever be a need for synchronized access or specific ways to modify the machine's data, having a singular point of control can simplify these processes.

In conclusion, by adopting the Singleton design pattern for FDC's dental scan machine scenario, we're ensuring a robust, efficient, and accurate representation of the real-world context. The pattern inherently mirrors the exclusivity of the dental scan machine, ensuring that the software remains a true reflection of FDC's operational reality.

4.1.5 Comparative Analysis of the Singleton Design Pattern in the Context of FDC's Dental Scan Machine Implementation

In software engineering, the choice of a design pattern often hinges upon the specific requirements of a given scenario. When evaluating the Singleton design pattern in the context of FDC's dental scan machine system, it becomes paramount to juxtapose it against other potential design patterns to elucidate its appropriateness.

Singleton vs. Composite: The Composite pattern is employed to treat individual and composite objects uniformly. It's optimal for representing part-whole hierarchies, allowing us to create tree structures and operate on them uniformly. In the context of the dental scan machine, there is no hierarchy or tree structure to manage. The machine isn't a composite of other entities, nor does it function as part of a more complex system where individual and composite entities need to be handled in a uniform way. The essence of the problem is singular instantiation, not hierarchical organization.

Singleton vs. Adapter: The Adapter pattern primarily serves to bridge the gap between two incompatible interfaces, allowing them to work together. It's like a translator between two systems that don't understand each other. In the scenario, the problem isn't about integrating or making two differing interfaces work together. The challenge is ensuring only one instance of the dental scan machine representation exists. Therefore, the Adapter's interface translation capability doesn't align with the unique instantiation need presented by the scenario.

Singleton vs. Observer: The Observer pattern is designed for scenarios where an object (known as the subject) maintains a list of dependents (observers) and notifies them of any state changes, typically by calling one of their methods. It is used when there's a one-to-many dependency between objects, so when one object changes state, its dependents are notified and updated automatically. The dental scan machine's scenario doesn't present a situation where multiple objects are waiting for state updates from a primary object. The core of the problem is ensuring a single representation, not managing and notifying a group of dependent objects about state changes.

Singleton vs. Prototype: The Prototype pattern, which involves creating a fully initialized instance that can be cloned or copied to produce a similar instance, could theoretically be used to replicate the dental scan machine's attributes. However, given that FDC owns a unique, high-cost machine and there's a clear mandate to have only one such instance, using Prototype would contravene this constraint. The Singleton pattern, on the other hand, rigidly adheres to the requirement by ensuring a sole instance, making it a more fitting choice.

Singleton vs. Factory Method: The Factory Method pattern offers a way to create objects in a superclass, but allows subclasses to alter the type of created objects. In FDC's scenario, there's no indication of varying types of dental scan machines or a need for subclassing. The emphasis is on the exclusivity of a single, state-of-the-art machine. Hence, while Factory Method offers flexibility in instantiation, Singleton directly addresses the uniqueness requirement.

Singleton vs. Builder: The Builder pattern is adept at constructing complex objects step by step, allowing the same construction process to create different types and representations of objects. In our case, the complexity isn't in the construction of various types of dental scan machines, but in maintaining the singularity of the machine's representation. Singleton's emphasis on a single instance aligns more congruently with FDC's needs.

Singleton vs. Object Pool: The Object Pool pattern allows the reuse of objects that are expensive to create. While the dental scan machine is indeed a costly asset, the scenario doesn't necessitate the reuse of multiple instances. FDC's requirement is straightforward: one machine, one representation. Singleton's premise of "one class, one instance" matches this perfectly.

Singleton and its Synergy with Other Patterns: It's worth noting that Singleton doesn't always stand in opposition to other patterns. In different circumstances, it can work in tandem. For instance, while the scenario doesn't demand multiple types of dental machines now, if FDC ever expands its repertoire, a Factory Method pattern producing Singleton instances for each machine type could be an evolution.

In summation, while many design patterns offer sophisticated ways to instantiate and manage objects, the Singleton pattern's simplicity and directness make it an ideal fit for FDC's dental scan machine scenario. It succinctly encapsulates the exclusivity and significance of the machine, ensuring the software system's integrity and mirroring FDC's real-world operational paradigm.

4.2 Scenario 2

4.2.1 Selection of Design Pattern

The scenario described fits the **Composite Design Pattern**. This pattern is particularly useful when dealing with tree-like structures. In this case, the hierarchy of employees in FDC forms a tree structure, where certain employees (like Director and Dentists) have other employees (Dentists or Nurses) reporting directly to them.

4.2.2 Draw the Class Diagram

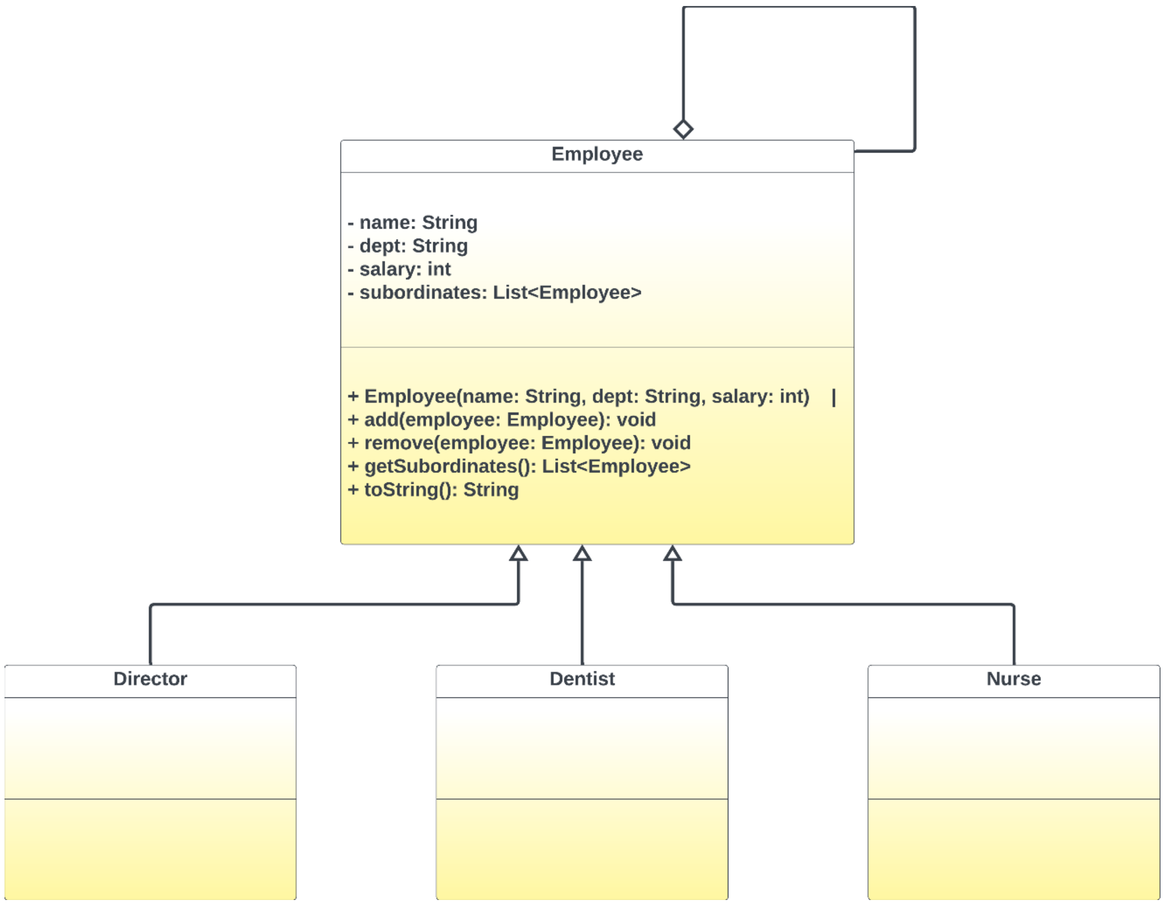

Employee
- name: String
- dept: String
- salary: int
- subordinates: List<Employee>
+ Employee(name: String, dept: String, salary: int)
+ add(employee: Employee): void
+ remove(employee: Employee): void
+ getSubordinates(): List<Employee>
+ toString(): String
Director
Dentist
Nurse

Figure 4. 3 Draw the Class Diagram for the scenario 2

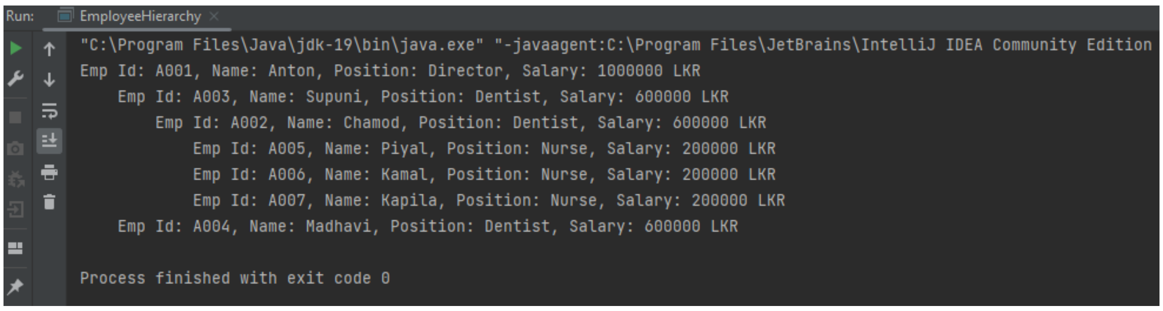
4.2.3 Code Development

Code:

```
import java.util.ArrayList;
import java.util.List;

// The Composite component: Employee class
class Employee {
    private String empId;
    private String name;
    private String position;
    private int salary;
    private List<Employee> subordinates; // list of employees under this
employee

    // Constructor to initialize employee details
    public Employee(String empId, String name, String position, int
salary) {
        this.empId = empId;
        this.name = name;
        this.position = position;
        this.salary = salary;
        this.subordinates = new ArrayList<>();
    }

    // Add an employee to subordinates list
    public void add(Employee employee) {
        subordinates.add(employee);
    }

    // Remove an employee from subordinates list
    public void remove(Employee employee) {
        subordinates.remove(employee);
    }

    // Get subordinates of this employee
    public List<Employee> getSubordinates() {
        return subordinates;
    }

    // Display the employee's details and hierarchy
    public void displayEmployeeDetails(int level) {
        // Print tabs for the current level to show hierarchy
        for (int i = 0; i < level; i++) {
            System.out.print("\t");
        }

        // Display current employee's details
        System.out.println("Emp Id: " + empId + ", Name: " + name + ",
Position: " + position + ", Salary: " + salary + " LKR");

        // Display details of each subordinate recursively
        for (Employee e : subordinates) {
            e.displayEmployeeDetails(level + 1);
```

```java
        }
    }
}

// Main class to test the hierarchy
public class EmployeeHierarchy {
    public static void main(String[] args) {
        // Create employees
        Employee anton = new Employee("A001", "Anton", "Director",
1000000);
        Employee supuni = new Employee("A003", "Supuni", "Dentist",
600000);
        Employee madhavi = new Employee("A004", "Madhavi", "Dentist",
600000);
        Employee chamod = new Employee("A002", "Chamod", "Dentist",
600000);
        Employee piyal = new Employee("A005", "Piyal", "Nurse", 200000);
        Employee kamal = new Employee("A006", "Kamal", "Nurse", 200000);
        Employee kapila = new Employee("A007", "Kapila", "Nurse",
200000);

        // Construct hierarchy as per the given scenario
        anton.add(supuni);
        anton.add(madhavi);

        supuni.add(chamod);

        chamod.add(piyal);
        chamod.add(kamal);
        chamod.add(kapila);

        // Display hierarchy starting from the top level
        anton.displayEmployeeDetails(0);
    }
}
```

Output:



```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
Emp Id: A001, Name: Anton, Position: Director, Salary: 1000000 LKR
    Emp Id: A003, Name: Supuni, Position: Dentist, Salary: 600000 LKR
        Emp Id: A002, Name: Chamod, Position: Dentist, Salary: 600000 LKR
            Emp Id: A005, Name: Piyal, Position: Nurse, Salary: 200000 LKR
            Emp Id: A006, Name: Kamal, Position: Nurse, Salary: 200000 LKR
            Emp Id: A007, Name: Kapila, Position: Nurse, Salary: 200000 LKR
    Emp Id: A004, Name: Madhavi, Position: Dentist, Salary: 600000 LKR

Process finished with exit code 0
```

Figure 4. 4 Java code output of scenario 2

4.2.4 Justification for selected Design Pattern

The Composite design pattern is chosen for this particular scenario because it effectively captures the hierarchical nature of the employee structure in the FDC. By using this design pattern, we aim to treat individual objects (`Employee`) and compositions of objects (like a `Dentist` having multiple `Nurses` or `Director` having subordinates) in a uniform manner. The Composite pattern facilitates the ease of constructing tree-like structures that represent part-whole hierarchies.

In the context of FDC's employee system, there is a clear hierarchy where certain employees, like the Director and Dentists, act as superiors or lead figures and have other employees reporting to them. For instance, Anton as a Director might have multiple Dentists reporting to him, and Dentists, in turn, might have Nurses under them. This hierarchy mirrors a tree-like structure with the Director at the top, followed by Dentists and then Nurses at the leaf nodes. The Composite pattern is particularly effective in such cases where we want to work with the hierarchy in a way that individual employees and groups of employees can be dealt with in a consistent manner.

Furthermore, the operations we might want to perform on employees, whether individual or grouped, are similar. Whether it's an individual Dentist or a group of Nurses, we might want to add, remove, or get details of subordinates. The Composite pattern allows us to apply these operations transparently, be it on a single employee or a group, without having to determine the type of employee or their hierarchical position in the organization.

Lastly, considering future extensibility, if FDC decides to introduce new roles or hierarchies, the Composite design pattern offers a flexible structure that can easily accommodate such changes. By encapsulating the hierarchy and shared operations in this pattern, the design becomes more maintainable, adaptable, and aligned with the principle of object-oriented design that emphasizes on objects over procedures.

In conclusion, the Composite pattern is a natural fit for scenarios like FDC's employee hierarchy due to its ability to abstract individual and composite entities under a unified interface, thereby offering consistency, flexibility, and an intuitive representation of the organization's structural hierarchy.

4.2.5 Comparative Analysis of the Composite Design Pattern in FDC's Employee Hierarchy Implementation

The need to model and effectively represent the intricate hierarchical structure of FDC's employees calls for a design pattern that can not only encapsulate individual entities but also their groupings in a cohesive manner. The Composite design pattern, with its inherent structure and flexibility, appears tailored for such requirements. However, to truly appreciate its appropriateness, we must compare it against other notable design patterns.

Composite vs. Singleton: The Singleton pattern ensures that a class has only one instance and provides a point of global access to that instance. It's commonly used when a single point of control or access is required. The primary requirement in the scenario is to manage multiple employees in a hierarchical manner. Constraining this system to a single instance (like a single employee or single hierarchy) would not make sense, as there are clearly multiple entities (employees) with varying relationships. The Singleton pattern's focus on singular instantiation does not cater to this scenario's multifaceted hierarchical representation needs.

Composite vs. Adapter: The Adapter pattern is designed to make two incompatible interfaces work together, essentially serving as a bridge or translator. The challenge in the scenario was not one of reconciling two mismatched interfaces. Rather, it was about representing and managing a hierarchical structure of employees and their relationships. The Adapter's role as an interface "converter" doesn't align with the need to express and handle hierarchical relationships within the FDC.

Composite vs. Observer: In the Observer pattern, an object (the subject) maintains a list of dependents (observers) and notifies them of any changes in state. It is useful in situations where there's a one-to-many dependency, and state changes in one object must be propagated to others. The scenario revolves around modeling the hierarchy of employees, not dynamically notifying multiple objects of changes in another object. There's no primary object whose state changes need to be communicated to a set of dependent objects. Hence, the Observer's mechanism of state-change propagation doesn't serve the needs of representing and navigating the employee hierarchy.

Composite vs. Chain of Responsibility: The Chain of Responsibility pattern creates a chain of receiver objects that can act on a request. In the context of FDC, while this could be utilized to pass a request through the hierarchy of employees, it doesn’t inherently represent the structural hierarchy as Composite does. The Composite pattern builds a tree structure which directly maps to the described employee hierarchy, making it more suitable for this scenario.

Composite vs. Flyweight: The Flyweight pattern focuses on reusing objects to save memory, particularly when dealing with a large number of almost similar objects. In FDC's scenario, the emphasis is on the relationship and hierarchy among employees, not on object reuse or memory optimization. The Composite pattern, with its capability to represent complex structures, is clearly more aligned with the requirements.

Composite vs. Decorator: The Decorator pattern adds responsibilities to objects dynamically. While it can enhance the functionalities of the `Employee` class, it doesn’t inherently provide a way to represent the hierarchical relationships between employees. Composite, on the other hand, gives a straightforward approach to building and traversing such hierarchical relationships.

Composite vs. Bridge: The Bridge pattern separates an abstraction from its implementation, allowing both to vary independently. While it offers flexibility in terms of implementation, it doesn't specifically cater to the hierarchical needs of the FDC scenario. The Composite pattern’s strength is in simplifying the representation of part-whole hierarchies, making it more fitting.

Composite and its Synergy with Other Patterns: It's vital to note that the strength of the Composite pattern isn't just in its standalone capabilities. For instance, if FDC's application had varied implementations of how employee details are fetched or displayed, the Composite pattern could be effectively combined with the Bridge pattern, ensuring hierarchy representation (via Composite) and flexible implementations (via Bridge).

In summation, while various design patterns present their strengths and areas of application, the Composite design pattern's intrinsic ability to represent and manage hierarchical structures sets it apart for FDC’s specific requirements. The organizational layout of FDC, with its clear tiers of roles and relationships, finds a direct and efficient representation through the Composite pattern. This not only ensures a structured and cohesive design but also offers an intuitive way to manage and traverse the employee hierarchy, highlighting the pattern's aptness for the scenario.

4.3 Scenario 3

4.3.1 Selection of Design Pattern

For the scenario provided, the **Adapter Design Pattern** would be the most appropriate choice. The Adapter pattern allows a system to use classes that have incompatible interfaces by providing a middle-layer class that makes them compatible. By using this pattern, we can extend the functionalities of an existing library or class without altering its structure.

In this scenario, the existing Queue container from the library provides standard operations like `enqueue` and `dequeue`. However, our requirement includes additional operations like `search()` and `showAll()`. The Adapter pattern lets us create an interface that encapsulates the existing Queue and then add our custom functionalities, making the two interfaces work together.

4.3.2 Draw the Class Diagram

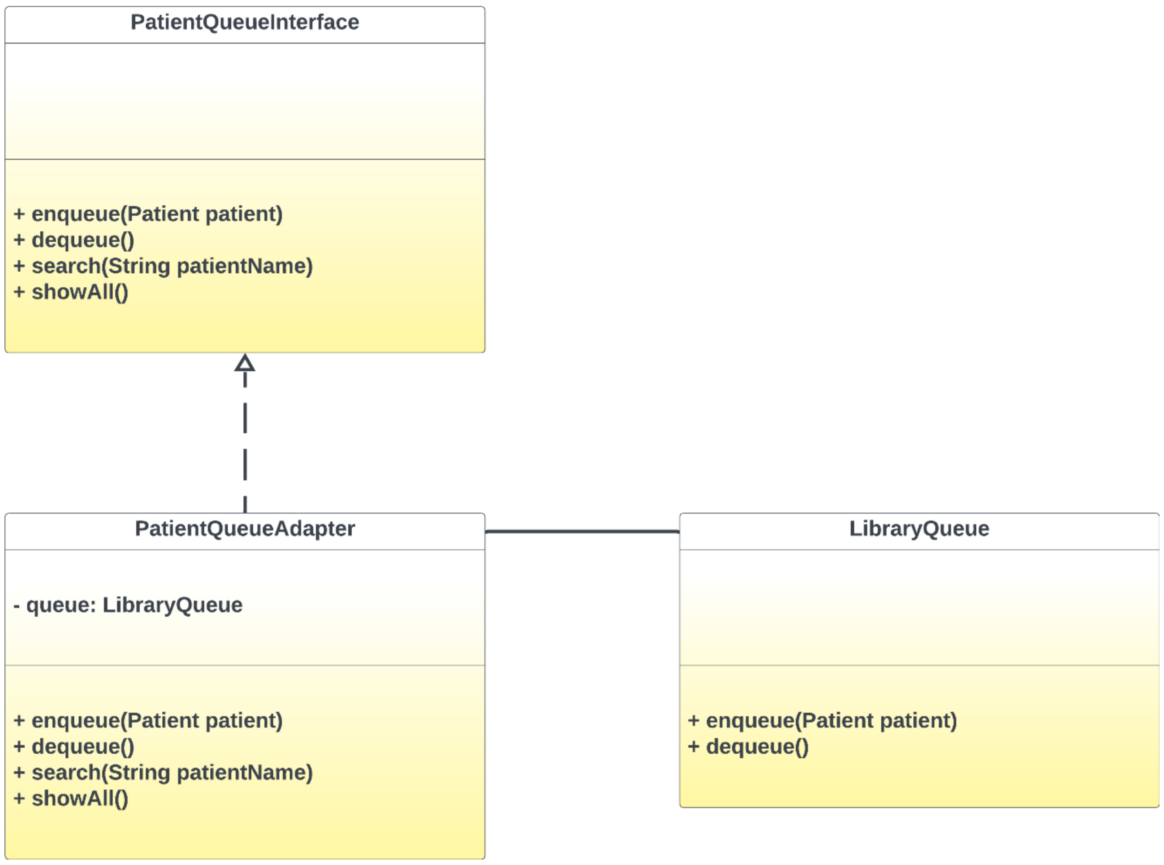

PatientQueueInterface
+ enqueue(Patient patient)
+ dequeue()
+ search(String patientName)
+ showAll()
PatientQueueAdapter
- queue: LibraryQueue
+ enqueue(Patient patient)
+ dequeue()
+ search(String patientName)
+ showAll()
LibraryQueue
+ enqueue(Patient patient)
+ dequeue()

Figure 4. 5 Draw the Class Diagram for the scenario 3

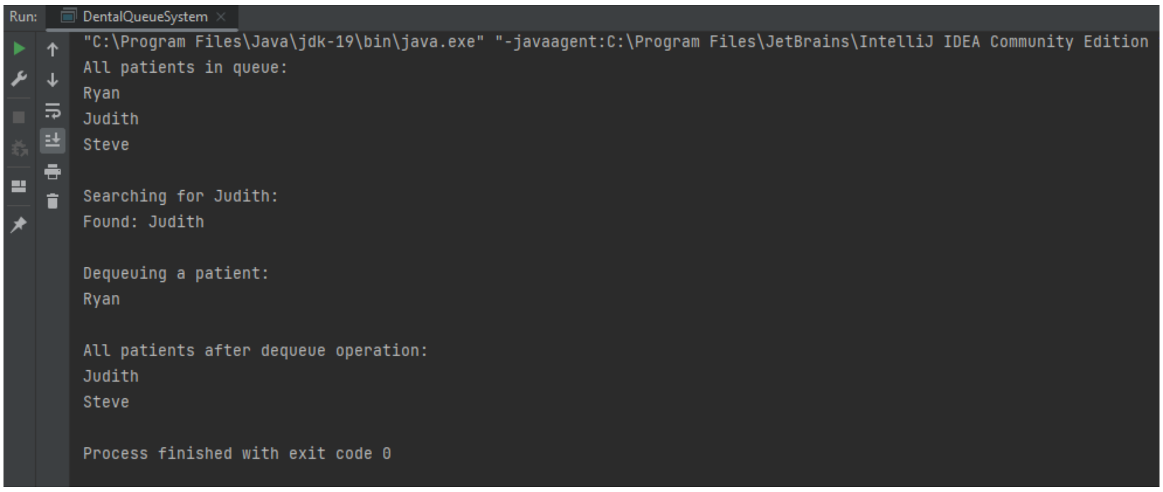
4.3.3 Code Development

Code:

```
import java.util.LinkedList;

// Define the Patient class to hold patient details
class Patient {
    private String name;
    public Patient(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    @Override
    public String toString() {
        return name;
    }
}

// Interface for the Patient Queue operations we expect
interface PatientQueueInterface {
    void enqueue(Patient patient);
    Patient dequeue();
    Patient search(String patientName);
    void showAll();
}

// Basic queue class using library's LinkedList (simulating the library-
provided Queue)
class LibraryQueue {
    LinkedList<Patient> queue = new LinkedList<>();

    public void enqueue(Patient patient) {
        queue.addLast(patient);
    }

    public Patient dequeue() {
        return queue.pollFirst();
    }
}

// Adapter class which makes our LibraryQueue compatible with
PatientQueueInterface
class PatientQueueAdapter implements PatientQueueInterface {
    private LibraryQueue queue;

    public PatientQueueAdapter(LibraryQueue queue) {
        this.queue = queue;
    }

    @Override
    public void enqueue(Patient patient) {
        queue.enqueue(patient);
```

```java
    }

    @Override
    public Patient dequeue() {
        return queue.dequeue();
    }

    // Search patient by name
    @Override
    public Patient search(String patientName) {
        for (Patient patient : queue.queue) {
            if (patient.getName().equals(patientName)) {
                return patient;
            }
        }
        return null;
    }

    // Display all patients
    @Override
    public void showAll() {
        for (Patient patient : queue.queue) {
            System.out.println(patient);
        }
    }
}

public class DentalQueueSystem {
    public static void main(String[] args) {
        LibraryQueue libraryQueue = new LibraryQueue();
        PatientQueueAdapter patientQueue = new
PatientQueueAdapter(libraryQueue);

        // Testing the system
        patientQueue.enqueue(new Patient("Ryan"));
        patientQueue.enqueue(new Patient("Judith"));
        patientQueue.enqueue(new Patient("Steve"));

        System.out.println("All patients in queue:");
        patientQueue.showAll();

        System.out.println("\nSearching for Judith:");
        Patient searchedPatient = patientQueue.search("Judith");
        if (searchedPatient != null) {
            System.out.println("Found: " + searchedPatient);
        } else {
            System.out.println("Not found");
        }

        System.out.println("\nDequeuing a patient:");
        System.out.println(patientQueue.dequeue());

        System.out.println("\nAll patients after dequeue operation:");
        patientQueue.showAll();
    }
}
```

Output:



Run: DentalQueueSystem

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
All patients in queue:
Ryan
Judith
Steve

Searching for Judith:
Found: Judith

Dequeuing a patient:
Ryan

All patients after dequeue operation:
Judith
Steve

Process finished with exit code 0
```

Figure 4. 6 Java code output of scenario 3

<u>4.3.4 Justification for selected Design Pattern</u>

In the given scenario for FDC, where patients awaiting dental scans are queued in a First In First Out (FIFO) order, we chose to employ the Adapter Design Pattern. The justification for this choice is anchored on several fronts, given the constraints and requirements outlined in the task.

To begin, we were presented with a software library that provided a basic Queue container. This container was equipped with the standard operations for a queue namely, enqueue (inserting data) and dequeue (removing data). These operations form the foundational mechanics of a queue system. However, FDC's needs extend beyond just these foundational operations. They required additional functionalities: the `search()` method to locate a specific patient and the `showAll()` method to list every patient in the queue. The existing library, as is often the case with many off-the-shelf software components, did not provide these supplemental methods.

Now, one could argue for directly modifying the library's source code (if it were accessible) to add these functions. However, this approach would come with its own set of complications. Modifying library code can introduce errors, make it difficult to benefit from future updates to the library, and can generally be seen as bad practice unless absolutely necessary.

This is where the Adapter Design Pattern shines. Instead of altering the library itself, we essentially "wrap" the library's queue with our custom class (the adapter). This class not only uses the basic queue operations provided by the library but also seamlessly integrates the additional methods that FDC requires. Thus, we can maintain the integrity of the original library while extending its functionality to meet our specific needs.

The Adapter pattern offers a harmonious solution to the classic software development challenge of making disparate components work together without majorly altering their internal structures. It promotes the Open/Closed principle of software design, which posits that software entities should be open for extension but closed for modification.

In the context of FDC's scenario, the Adapter Design Pattern provides a balanced blend of utilizing existing resources efficiently while tailoring functionality to exact needs. By implementing an adapter, we've essentially given FDC a custom queue system tailored for their operations without the overheads of building a queue system from scratch or meddling with pre-existing library code. It's an elegant, efficient, and maintainable solution to the outlined problem.

4.3.5 Comparative Analysis of the Adapter Design Pattern in FDC's Queue Implementation

The dilemma FDC faced with the existing Queue container and the need for extended operations portrays a common predicament in software design how to make an existing component mesh seamlessly with new requirements without causing a design overhaul. The Adapter Design Pattern was identified as the most fitting solution for this, but to truly discern its appropriateness, a comparison against other potential patterns is instrumental.

Adapter vs. Singleton: The Singleton pattern ensures only one instance of a class exists throughout an application's lifecycle and provides a single point of access to this instance. The scenario at hand is about extending functionalities of a pre-existing library component. It's not about limiting the instantiation of the queue to just one instance. Moreover, it's entirely conceivable that the dental clinic might want multiple queues in the future (e.g., separate queues for general check-ups, dental scans, surgeries, etc.). Enforcing a single instance restriction using the Singleton pattern would be counter-productive to such potential requirements.

Adapter vs. Observer: The Observer pattern deals with a one-to-many dependency between objects. When the state of one object (the subject) changes, all its dependents (observers) are notified and updated automatically. The scenario does not hint at any one-to-many relationships or dependencies that require dynamic notifications. The problem is straightforward: extend the queue functionalities. There's no requirement to notify multiple entities whenever a change occurs in the queue, making the Observer pattern's functionality redundant for this context.

Adapter vs. Decorator: At first glance, the Decorator pattern might seem an apt choice since it dynamically adds responsibilities to objects. For FDC's queue, one could think of "decorating" the queue with extra methods. However, the Decorator's primary aim is to add responsibilities, not interface adjustments. While the Decorator enhances, the Adapter makes different interfaces compatible. In this scenario, the core need was interfacing, making Adapter the better fit.

Adapter vs. Proxy: The Proxy pattern provides a placeholder for another object to control access to it. Although one might conceive using a Proxy to add the required functions, it isn't its intended purpose. The Proxy might control access to the queue, but the Adapter's role is to translate one interface into another, which is the primary need for FDC.

Adapter vs. Bridge: The Bridge pattern separates abstraction from its implementation, so both can vary independently. In FDC's case, the separation between the queue's interface and its operations wasn't the challenge; rather, it was the gap between the existing and the required interface. The Adapter serves this exact purpose by bridging different interfaces, making it more relevant than the Bridge pattern in this context.

Adapter vs. Composite: The Composite pattern, as discussed in a previous scenario, deals with part-whole hierarchies. It's splendid for modeling hierarchical relationships like organizational structures but isn't designed for interface compatibility challenges as presented by FDC's queue scenario.

Adapter and Its Interplay with Other Patterns: While the Adapter pattern stood out for FDC's needs, it’s worth noting that in other contexts, the Adapter can work harmoniously with other patterns. For instance, an Adapter could wrap around a Composite structure if there's a need to make an old compositional structure compatible with a new interface. The strength of design patterns often lies not just in their individual capabilities but also in their potential synergies.

Conclusively, the essence of the problem FDC faced was clear: a mismatch between a desired interface (with `search()` and `showAll()` functions) and the available interface (basic queue operations). This discrepancy is what the Adapter pattern is specifically designed to address. Other patterns, while powerful in their specific domains, didn't align as precisely with the core challenge at hand. By choosing the Adapter, FDC could seamlessly extend the functionality of the existing queue container without disrupting its original design, ensuring an efficient and maintainable solution. It underscores the power of picking the right tool or in this case, the right design pattern for the job, a choice that can vastly streamline development and future adaptations.

Conclusion

Throughout this comprehensive study, we delved deep into the intricate facets of the Object-Oriented Paradigm, specifically within the context of the Family Dental Care (FDC) scenario. Beginning with the foundational concepts of classes and objects, we systematically explored the dynamics of messages, encapsulation, inheritance, and polymorphism. Our emphasis on explaining code, paired with UML diagrams, ensured a practical and visual grasp of these abstract concepts.

Further, the exploration of class relationships underscored the inherent flexibility and expressiveness of object-oriented design. Our analysis then transitioned to a meticulous examination of design patterns. By illuminating patterns such as Singleton, Composite, and Observer, we highlighted the synergy between these design templates and OOP principles. Each pattern's application to the FDC scenario illustrated its real-world relevancy and potential to enhance system design.

Finally, the in-depth scenarios in Task 4 underscored the potency of these patterns. Through detailed class diagrams, code, and comparative analyses, we determined the optimal design strategy for each unique requirement.

In essence, this assignment has not only fortified our understanding of OOP and design patterns but also showcased their pivotal role in architecting robust, maintainable, and efficient software solutions, especially for intricate domains like healthcare.

References

Visual-Paradigm. (no date). ‘UML Class Diagram Tutorial’. Visual-Paradigm. Available at: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

(Accessed: 19 May 2023).

TutorialsPoint. (no date). ‘UML - Class Diagram’. TutorialsPoint. Available at: https://www.tutorialspoint.com/uml/uml_class_diagram.htm

(Accessed: 23 May 2023).

TechTarget. (no date). ‘Object-Oriented Programming (OOP)’. TechTarget. Available at: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP/

(Accessed: 27 May 2023).

JOT. (2004). ‘Object-Oriented Programming’. Available at: https://www.jot.fm/issues/issue_2004_05/column1.pdf

(Accessed: 2 June 2023).

Stackify. (no date). ‘Encapsulation’. Stackify. Available at: https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/

(Accessed: 5 June 2023).

Analytics Vidhya. (2020). ‘Inheritance in Object-Oriented Programming’. Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/10/inheritance-object-oriented-programming/

(Accessed: 8 June 2023).

Sumo Logic. (no date). ‘Polymorphism’. Sumo Logic. Available at: https://www.sumologic.com/glossary/polymorphism/

(Accessed: 14 June 2023).

Educative. (no date). ‘What is Aggregation in Object-Oriented Programming?’. Educative. Available at: https://www.educative.io/answers/what-is-aggregation-in-object-oriented-programming

(Accessed: 18 June 2023).

DigitalOcean. (no date). ‘Composition vs Inheritance’. DigitalOcean. Available at: https://www.digitalocean.com/community/tutorials/composition-vs-inheritance

(Accessed: 22 June 2023).

InfoWorld. (no date). ‘Exploring Association, Aggregation, and Composition in OOP’. InfoWorld. Available at: https://www.infoworld.com/article/3029325/exploring-association-aggregation-and-composition-in-oop.html

(Accessed: 28 June 2023).

UOP. (2008). ‘ASP.NET 3.5 Application Architecture and Design’. Available at: https://www.uop.edu.jo/PDF%20File/petra%20university%20ASP.NET_3.5_Application_Architecture_and_Design_October_2008-20612-Part39.pdf

(Accessed: 3 July 2023).

ScienceDirect. (no date). ‘Dependency Relationship’. ScienceDirect. Available at: https://www.sciencedirect.com/topics/computer-science/dependency-relationship/

(Accessed: 7 July 2023).

Scaler. (no date). ‘Creational Design Pattern’. Scaler. Available at: https://www.scaler.com/topics/design-patterns/creational-design-pattern/

(Accessed: 12 July 2023).

Baeldung. (no date). ‘Core Structural Patterns in Java’. Baeldung. Available at: https://www.baeldung.com/java-core-structural-patterns

(Accessed: 18 July 2023).

Scaler. (no date). ‘Behavioral Design Pattern’. Scaler. Available at: https://www.scaler.com/topics/design-patterns/behavioral-pattern/

(Accessed: 21 July 2023).

SourceMaking. (no date). ‘Design Patterns’. SourceMaking. Available at: https://sourcemaking.com/design_patterns

(Accessed: 25 July 2023).

TutorialsPoint. (no date). ‘Adapter Pattern’. TutorialsPoint. Available at: https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

(Accessed: 29 July 2023).