Java Programming Project

Khattar Daou, M.S., Ph.D. Technical Sciences
Microsoft Certified Trainer (MCT), Microsoft Office Specialist (MOS)

Java Programming – Team Project

Deadline for submission: as per symbol schedule

Student Name	Student ID

Project Initiation – to Create Medical Clinic Java Classes and UML Class Diagram

Suppose you work for a medical clinic that needs to create Medical Clinic System (MCS) to store some basic information about doctors, patients, and other functions. The classes of objects in the problem domain obviously include doctors and patients. You want to store information about each doctor, such as a name, date of birth, and specialty. Similarly, you want to store information about each patient, such as name, date of birth, and insurance company.

1. Start you project by planning, analyzing, designing, and implementing the system.
2. Draw the Use Case Diagram showing the use cases and the actors, making assumption about which actor is the user for each use case.
3. List all the scenarios that might apply to each use case.
4. Find the common attributes and behaviors (if exist) that have these two classes.
5. Create a general class called **Person** which contains the common attributes.
6. Create the **Doctor** and the **Patient** classes.
7. Specify the Abstract class and the inherited classes.
8. Draw the class diagram and show the generalization/specialization hierarchy.
9. The clinic is mainly concerned with providing treatments to patients, so information about treatments should also be included in the system. The problem domain and corresponding class diagram now include one more class, called **Treatment**.
10. Expand your class diagram to provide for the treatment's requirements.
11. Describe the attributes of Treatment required by the system. Note that each treatment is associated with one patient, and each treatment is also associated with one doctor. Naturally, a doctor provides many treatments, and a patient might have many treatments.
12. Include in the class diagram the Treatment class associated with Doctor and Patient classes, using multiplicity notation written on the association relationship line. In addition, the UML diagram may contain additional interfaces, and classes such as Department, Diagnostic Test, Receptionist, Appointment, Billing, Person, Insurance and so on.
13. We may use the class Scanner for inputting data from the standard input device or use Java stream classes, such as FileReader and PrintWriter. You can ask the user to enter a file or directory name to process the data, then save the processed data to disk (normally, later, in a database).
14. Apply your knowledge to add as much functionalities as possible.

Solution – Your solution could be different from this one.

The Doctor and Patient classes have something in common. Both have names and dates of birth as attributes. In fact, all people have names and dates of birth, so we can use a general class called **Person**. Some people are doctors, and some people are patients. For doctors, we want to know their date of employment and their specialty. For patients, on the other hand, we want to know their employer and insurance company. Therefore, we need to define specialized classes for these two types of people because each has different attributes. When a patient makes an appointment to see a doctor, the patient is an object, the doctor is an object, and the appointment itself is an object.

Classes and Objects

A *class* is the general template we use to define and create specific instances, or objects. Every object is associated with a class. For example, all the objects that capture information about patients could fall into a class called Patient, because there are attributes (e.g., name, address, birth date, phone, and insurance carrier) and methods (e.g., make appointment, calculate last visit, change status, and provide medical history) that all patients share.

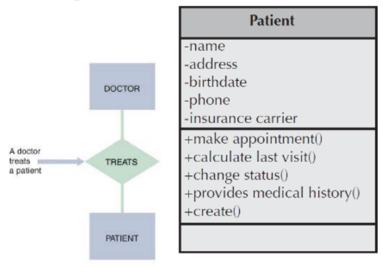


Figure 1: Patient Class

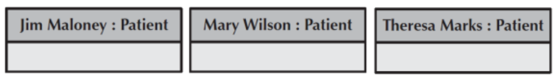


Figure 2: Patient Objects

An *object* is an instantiation of a class. In other words, an object is a person, place, or thing about which we want to capture information. If we were building an appointment system for a doctor's office, classes might include Doctor, Patient, and Appointment. Th e specific patients, such as Jim Maloney, Mary Wilson, and Theresa Marks, are considered *instances*, or objects, of the patient class.

Each object has *attributes* that describe information about the object, such as a patient's name, birth date, address, and phone number. Attributes are also used to represent relationships between objects; for example, there could be a department attribute in an employee object with a value of a department object that captures in which department the employee object works. The *state* of an object is defined by the value of its attributes and its relationships with other objects at a particular point in time. For example, a patient might have a state of new or current or former.

Each object also has *behaviors*. The behaviors specify what the object can do. For example, an appointment object can probably schedule a new appointment, delete an appointment, and locate the next available appointment. In object-oriented programming, behaviors are implemented as methods.

Methods and Messages

Methods implement an object's behavior. A method is nothing more than an action that an object can perform. *Messages* are information sent to objects to trigger methods. A message is essentially a function or procedure call from one object to another object. For example, if a patient is new to the doctor's office, the receptionist sends a create message to the application. The patient class receives the create message and executes its create() method which then creates a new object: aPatient.

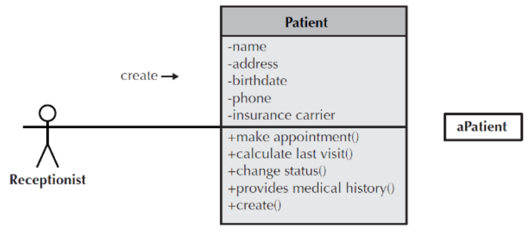


Figure 3: Messages and Methods

Inheritance

Inheritance, as an information systems development characteristic, was proposed in data modeling. The data modeling literature suggests using inheritance to identify higher-level, or more general, classes of objects. Common sets of attributes and methods can be organized into *superclasses.* Typically, classes are arranged in a hierarchy whereby the superclasses, or general classes, are at the top and the *subclasses,* or specific classes, are at the bottom.

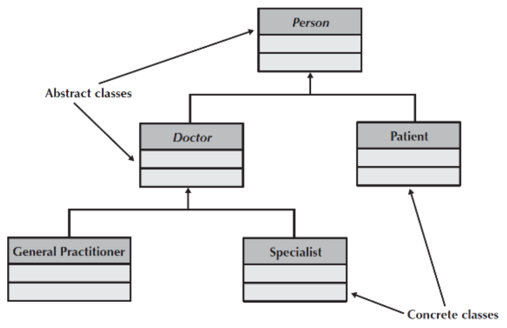
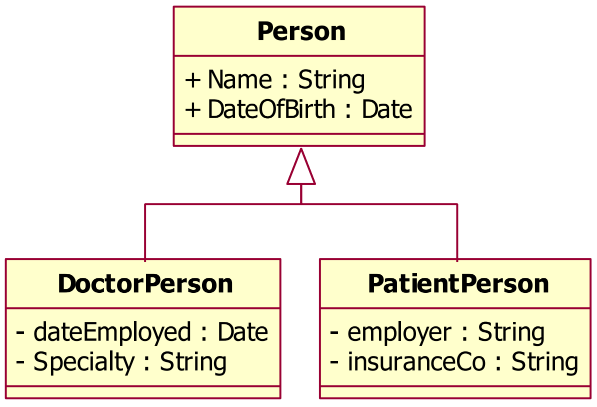


Figure 4: Class Hierarchy with Abstract and Concrete Classes



In Figure 4, Person is a superclass to the classes Doctor and Patient. Doctor, in turn, is a superclass to General Practitioner and Specialist. Notice how a class (e.g., Doctor) can serve as a superclass and subclass concurrently. The relationship between the class and its superclass is known as the *a-kind-of* relationship. For example, in Figure 1-11, a General Practitioner is a-kind-of Doctor, which is a-kind-of Person.

Polymorphism and Dynamic Binding

Polymorphism means that the same message can be interpreted differently by different classes of objects. For example, inserting a patient means something different than inserting an appointment. Therefore, different pieces of information need to be collected and stored. Luckily, we do not have to be concerned with *how* something is done when using objects. We can simply send a message to an object, and that object will be responsible for interpreting the message appropriately.

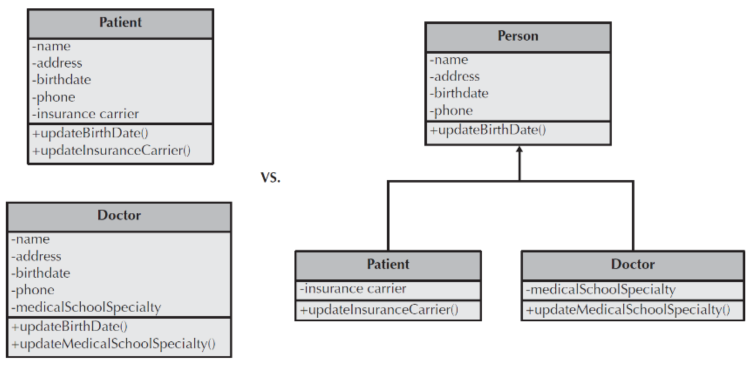


Figure 5: Inheritance Advantage

The two classes are **DoctorPerson** and **PatientPerson**. Person is also shown, as a general class. The generalization/specialization hierarchy is indicated by the triangle symbol on the line that connects Person to DoctorPerson and PatientPerson.

This system does not require that we store information about a person unless they are either a doctor or a patient; therefore, Person is an abstract class that exists only to allow subclasses to inherit from it.

When we store information about a doctor, DoctorPerson will include values for **Name** and **Date of Birth.** Similarly, when we store information about a patient, PatientPerson will include values for Name and Date of Birth. Therefore, both DoctorPerson and PatientPerson "inherit" the attributes Name and Date of Birth from the general class named Person. However, it is important to recognize that neither inherits specific values, unless there are default values.

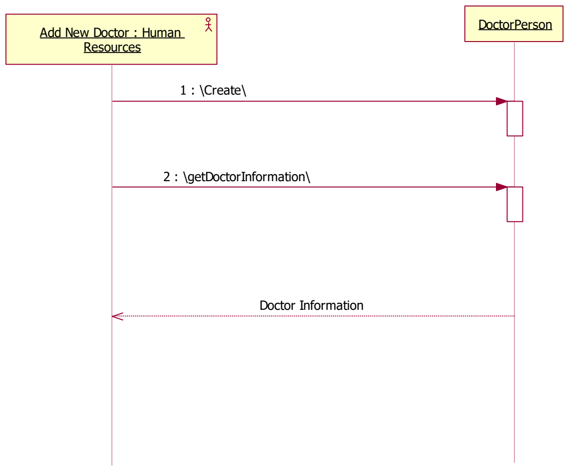
The requirements for the system could be described by scenarios that highlight the user's interaction with the system, organized around events that occur and their corresponding use cases.

1. **Event: A doctor is employed with the clinic.**
   Use Case: Add new doctor, main scenario
   The user sends a message to DoctorPerson asking it to add a new DoctorPerson object.
   DoctorPerson knows it needs the **Name**, **Date of Birth**, **Date Employed**, and **Specialty** to add a DoctorPerson object, so it asks the user for those values.
   The user supplies the requested values.
   DoctorPerson adds the new DoctorPerson object and tells the user the task is complete.

2. **Event: A new patient is added to the clinic.**
   Use Case: Add new patient, main scenario
   The user sends a message to PatientPerson asking it to add a new PatientPerson object.
   PatientPerson knows it needs the **Name**, **Date of Birth**, **Insurance Company**, and **Employer** to add a PatientPerson object, so it asks the user for those values.
   The user supplies the requested values.
   PatientPerson adds the new PatientPerson object and tells the user the task is complete.

Again, notice the user has no reason to add a Person, unless the person is either a DoctorPerson or a PatientPerson. Therefore, there are no objects in this system for the class Person. Also, use cases and scenarios related to other events should be included to clarify the requirements. Obviously, the user wants to look up information about doctors and patients. This query capability is implicit. For example, the user could ask to see all the doctors with a specific specialty. Similarly, the user could ask to see all patients with a specific insurance company. If specific query requirements are important for the user, these can be documented. Otherwise, we can assume these capabilities exist. No custom methods are required to meet the requirements.

Sequence diagrams modeling the interactions in the scenarios described above are shown in figure 2 and figure 3.



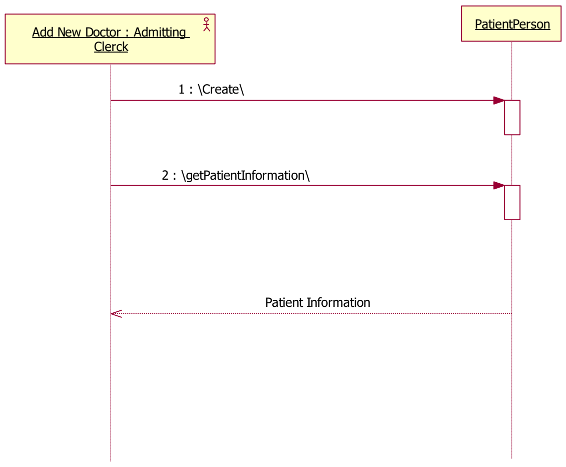

Add New Doctor : Admitting Clerck
PatientPerson
1 : \Create\
2 : \getPatientInformation\
Patient Information

Figure 2 Figure 3

The clinic is mainly concerned with providing treatment to patients. So information about treatments should also be included in the system. Therefore, we will expand the diagram to provide for this requirement. The expanded class diagram is shown in Figure 4.

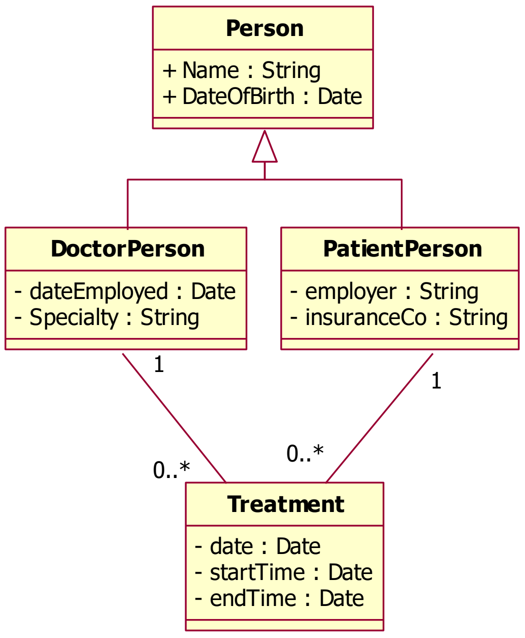


Figure 4

The problem domain and corresponding class diagram now include one more class, called **Treatment.** The attributes of Treatment (required by this system) are the **Date**, **Start Time**, and **End Time.** Each Treatment is associated with one PatientPerson. Each Treatment is also associated with one DoctorPerson. Therefore, we know who was treated and who provided the treatment. Naturally, a DoctorPerson provides many Treatments. Similarly, a PatientPerson might receive many Treatments. These associations are shown on the class diagram, using multiplicity notation.

DoctorPerson and PatientPerson also know how to associate themselves with a Treatment, but the relationship is optional. An optional relationship means that a doctor might not have treated any patients, or a patient might not have received a treatment. These classes of objects also have all of the capabilities listed previously.

1. **Event: A doctor is employed with the clinic.**
   - **Use Case: Add new doctor, main scenario**
     - The interaction is the same as shown previously. Note that there is no requirement that the DoctorPerson object connect to a Treatment, even though this capability is present.
2. **Event: A new patient is added to the clinic.**
   - **Use Case: Add new patient, main scenario**
     - The interaction is the same as shown previously. Note that there is no requirement that the PatientPerson object connect to a Treatment, even though this capability is present.
3. **Event: A patient receives a treatment.**
   - **Use Case: Record a treatment, main scenario**
     - The user sends a message to Treatment asking it to add a new Treatment object.
     - Treatment knows it needs to know the DoctorPerson Name and the PatientPerson Name because it is required to connect to both objects, so it asks the user for these values.
     - The user supplies the requested values.
     - Treatment knows it needs the Date, Start Time, and End Time for the Treatment object, so it asks the user for these values.
     - The user supplies the requested values.
     - The Treatment class adds a new Treatment object, using the Date, Start Time, and End Time, connects to the correct DoctorPerson object, connects to the correct PatientPerson object, and tells the user the task is complete.

The sequence diagram modeling the scenario for recording a treatment is shown in Figure 5.

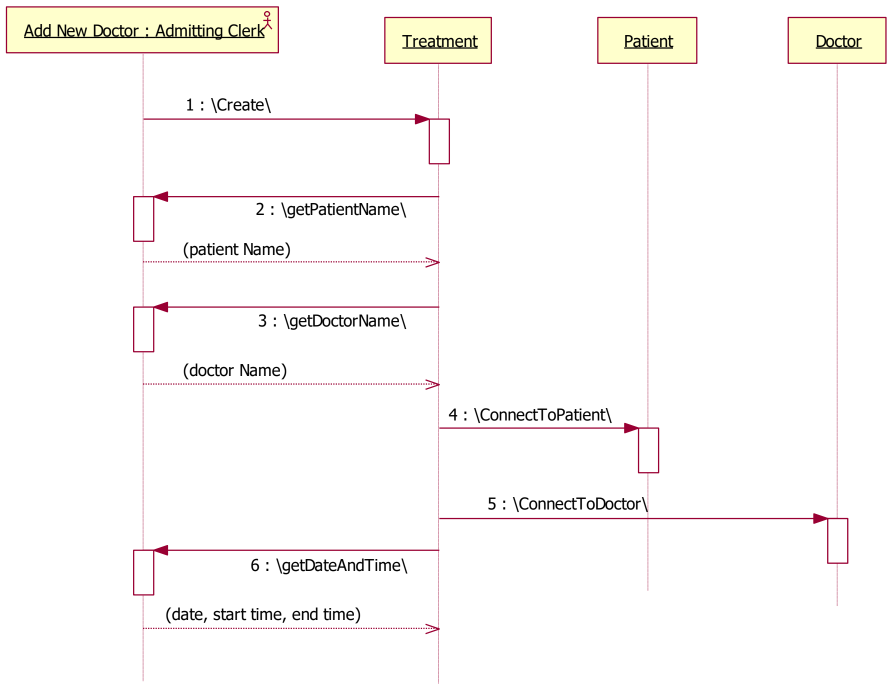


Figure 5

Class-Design Guidelines:

Cohesion

A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff are different entities.

A single entity with many responsibilities can be broken into several classes to separate the responsibilities. The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities. The String class deals with immutable strings, the StringBuilder class is for creating mutable strings, and the StringBuffer class is similar to StringBuilder, except that StringBuffer contains synchronized methods for updating strings.

Consistency

Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor, and place constructors before methods.

Make the names consistent. It is not a good practice to choose different names for similar operations. For example, the length() method returns the size of a String, a StringBuilder, and a StringBuffer. It would be inconsistent if different names were used for this method in these classes.

In general, you should consistently provide a public no-arg constructor for constructing a default instance. If a class does not support a no-arg constructor, document the reason. If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.

If you want to prevent users from creating an object for a class, you can declare a private constructor in the class.

Encapsulation
A class should use the private modifier to hide its data from direct access by clients. This makes the class easy to maintain.

Provide a getter method only if you want the data field to be readable and provide a setter method only if you want the data field to be updateable.

Clarity
Cohesion, consistency, and encapsulation are good guidelines for achieving design clarity. In addition, a class should have a clear contract that is easy to explain and easy to understand.

Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on how or when the user can use it, design the properties in a way that lets the user set them in any order and with any combination of values, and design methods that function independently of their order of occurrence.

For example, the Loan class contains the properties loanAmount, numberOfYears, and annualInterestRate. The values of these properties can be set in any order.

Methods should be defined intuitively without causing confusion.

You should not declare a data field that can be derived from other data fields. For example, the following Person class has two data fields: birthDate and age. Since age can be derived from birthDate, age should not be declared as a data field.

Completeness
Classes are designed for use by many different customers. To be useful in a wide range of applications, a class should provide a variety of ways for customization through properties and methods. For example, the String class contains more than 40 methods that are useful for a variety of applications.

Instance vs. Static
A variable or method that is dependent on a specific instance of the class must be an instance variable or method. A variable that is shared by all the instances of a class should be declared static. For example, the variable numberOfObjects in Circle is shared by all the objects of the Circle class, and therefore is declared static. A method that is not dependent on a specific instance should be defined as a static method. For instance, the getNumberOfObjects() method in Circle is not tied to any specific instance and therefore is defined as a static method.

Always reference static variables and methods from a class name (rather than a reference variable) to improve readability and avoid errors.

Inheritance vs. Aggregation
The difference between inheritance and aggregation is the difference between an is-a and a has-a relationship. For example, an apple is a fruit; thus, you would use inheritance to model the relationship between the classes Apple and Fruit. A person has a name; thus, you would use aggregation to model the relationship between the classes Person and Name.

Interfaces vs. Abstract Classes
Both interfaces and abstract classes can be used to specify common behavior for objects. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent–child relationship should be modeled using classes. For example, since an orange is a fruit, their relationship should be modeled using class inheritance.

A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface.

A circle or a rectangle is a geometric object, so Circle can be designed as a subclass of GeometricObject. Circles are different and comparable based on their radii, so Circle can implement the Comparable interface.

Interfaces are more flexible than abstract classes because a subclass can extend only one superclass but can implement any number of interfaces. However, interfaces cannot contain data fields. In Java 8, interfaces can contain default methods and static methods, which are very useful to simplify class design.

References:

1. Systems Analysis and Design: An Object-Oriented Approach with UML | Alan Dennis, Barbara Haley Wixom, David Tegarden | ISBN: 978-1118804674 © 2015 | Published by Wiley.
2. Introduction to Java Programming and Data Structures, Comprehensive Version - Twelfth Edition, Y. Daniel Liang, ISBN: 978-1-292-22187-8 © Pearson Education Limited 2020
3. Java Programming Style Guidelines https://petroware.no/html/javastyle.html