Critical Thinking Assignment 6: UML Sequence Diagram

Alexander Ricciardi

Colorado State University Global

CSC470: Software Engineering

Dr. Vanessa Cooper

January 26, 2025

Critical Thinking Assignment 6: UML Sequence Diagram

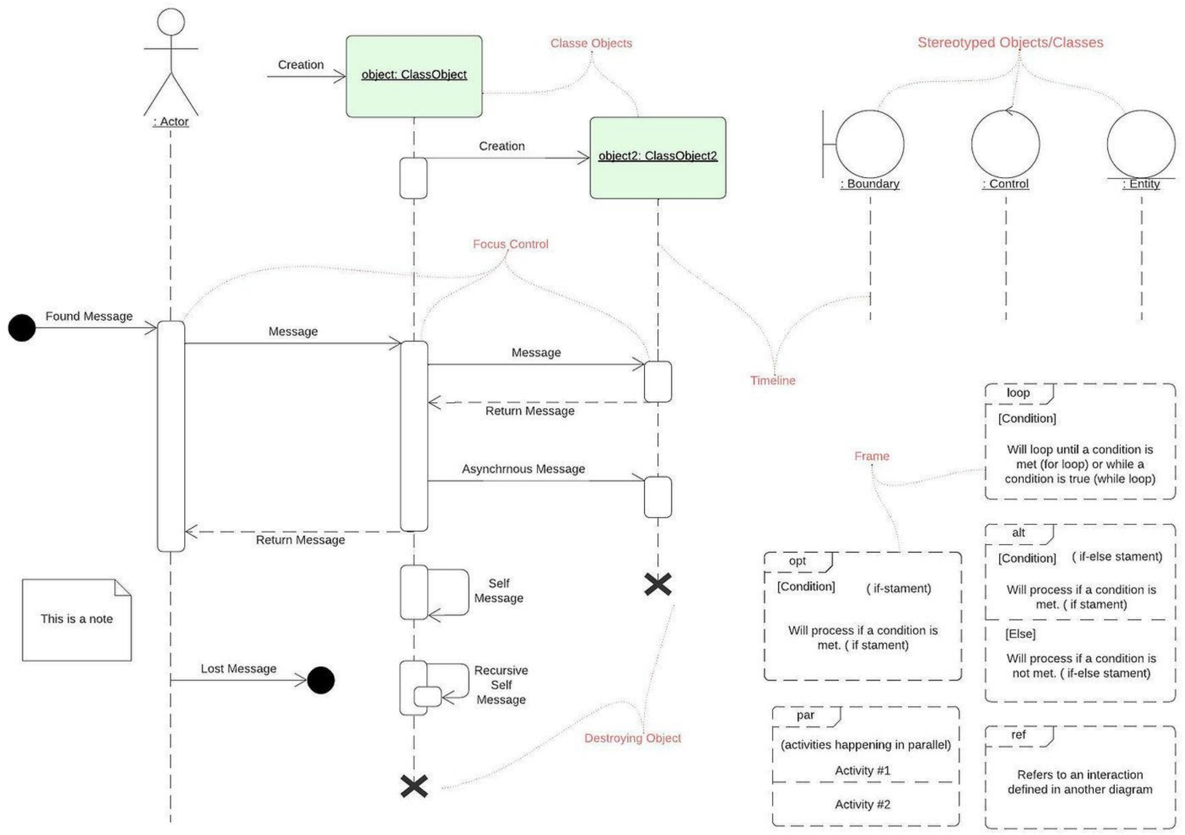
Unified Modeling Language (UML) sequence diagrams are extensively used in Software Engineering (SE) to model the dynamic interactions of objects within systems. They illustrate a sequence of messages between objects within an interaction (IBM, 2021). This essay provides an overview of UML sequence diagrams and a UML sequence diagram, along with its related pseudocode and class diagram, illustrating a doctor-patient examination interactions workflow within a hospital management system.

UML Sequence Diagrams Overview

UML Sequence Diagrams "are usually based on UML class diagrams and are used primarily to show the interactions between objects (classes) in the chronological order in which those interactions occur" (Ricciardi, 2025, p. 1). They provide a dynamic illustration of the interactions between classes (objects) within a given period of time. In SE, they are used, in the problem space, to analyze and model system behavior from the actor's (user's) viewpoint (Unhelkar, 2018). In the solution space, they are used to illustrate interactions within a system in more detail, such as the sequence of the inquiry messages illustrating method calls, the parameter messages within those methods, and the value returned by those methods. The diagram uses a system of notation such as actors (classes), lifelines (timelines), arrows denoting messages or method calls between these lifelines, activation bars indicating method execution, and optional annotations for conditions, loops, or return values. These elements work together to illustrate the flow interaction within a system. The elements are arranged vertically to illustrate the chronological order of interactions and horizontally to reflect the direction of the interactions between objects. The figure below, Figure 1, depicts the main elements of a UML sequence diagram.

Figure 1

UML Sequence Diagram Notation Elements


: Actor
Creation
object: ClassObject
Classe Objects
Stereotyped Objects/Classes
Creation
object2: ClassObject2
: Boundary
: Control
: Entity
Focus Control
Found Message
Message
Message
Timeline
Return Message
loop
[Condition]
Will loop until a condition is met (for loop) or while a condition is true (while loop)
Frame
Asynchrnous Message
Return Message
alt
[Condition] (if-else stament)
Will process if a condition is met. (if stament)
[Else]
Will process if a condition is not met. (if-else stament)
opt
[Condition] (if-stament)
Will process if a condition is met. (if stament)
Self Message
This is a note
Lost Message
Recursive Self Message
Destroying Object
par
(activities happening in parallel)
Activity #1
Activity #2
ref
Refers to an interaction defined in another diagram

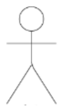
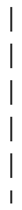
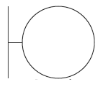
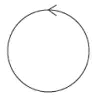
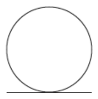
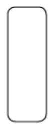
Note: The figure illustrates the elements that can be found in UML sequence diagrams and how they may interact within a diagram. From “A Guide to UML Sequence Diagrams: Notation, Strengths, and Limitations” by Ricciardi (2025).

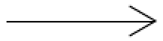
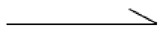
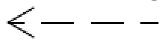
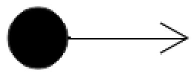
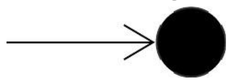
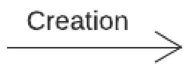
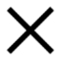
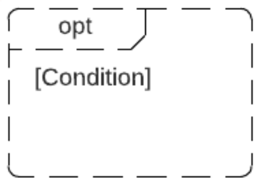
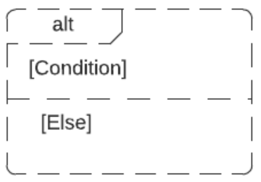
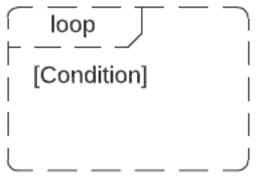
It is important to note that classes’ and objects' names are underlined and they can be illustrated by a stick figure for outside actors and internal actors can be encapsulated in a rectangle or represented by various circular shapes or icons representing specific stereotypes (e.g. boundary, control, and entity). A stereotype in UML “denotes a variation on an existing modeling element with the same form but with a modified intent” (Agile Modeling, 2023, p.1).

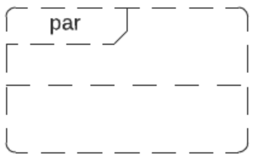
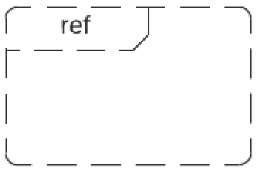
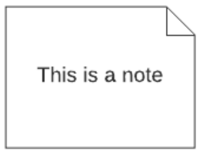
Usually, stereotypes are represented using guillemets (<< >>) encapsulating a keyword; for instance, the specific stereotypes of a sequence diagram, represented by the circular shapes, are <<boundary>>, <<control>>, and <<entity>>. The table below, Table 1, provides detailed descriptions of the notation elements depicted in Figure 1, including the sequence diagram icon stereotypes.

Table 1

UML Sequence Diagram Notation Descriptions

Notation	Description
Actor 	Represents a user (outside actor) or external system.
Lifeline or Timeline 	Represents a participant lifeline in the system
Boundary 	Stereotyped object/class representing a system boundary element, like UI screens or database gateways.
Control 	Stereotyped object/class representing a controlling entity or manager.
Entity 	A stereotyped object/class usually represents system data or a data object.
Focus or Activity Bar 	A thin rectangle on a lifeline that shows the period during which an object is active or performing an action.

Message 	Synchronous messages that represent inquirer messages between objects (e.g. getAttribute()) - The sender waits for confirmation or return data from the receiver to continue processing.
Asynchronous Message 	An asynchronous message, the sender does not wait for confirmation or return data from the receiver to continue processing.
Return Message 	A dashed arrow with an open arrowhead. Indicates the return of control to the message caller.
Found Message 	The message is outside of the scope of the system description; for example, a message that originated outside of the system.
Lost Message 	A message that does not have a receiver or the receiver is unknown; for example, a message sent to an outside unknown system.
Creation Arrow 	Represents the point in a system timeline when a particular object was created and may also indicate who/what created it.
Participant Destruction 	An "X" at the end of a lifeline. Indicates that the participant is destroyed.
Option Frame 	"if-then" logic.
Alternatives Frame 	"if-then-else" logic.
Loop Frame 	Represents the logic of a “for-loop” or “while-loop”

Parrelle Frame 	Represents activities happening in parallel (at the same time).
Reference Frame 	Allows reusing part of one sequence diagram in another. Represented by a frame that refers to another diagram.
Comments 	Comments are used to add explanatory notes to the diagram.

Note: The table describes the notation elements that can be found in UML sequence diagrams. From "A Guide to UML Sequence Diagrams: Notation, Strengths, and Limitations" by Ricciardi (2025).

Sequence diagrams have advantages and disadvantages, they are well-suited for modeling the dynamic behavior of a system by illustrating interactions between different actors or classes over a period of time. They help identify missing elements in class diagrams and they can be used for documentation (Ricciardi, 2025). On the other hand, they struggle to represent parallel processes or complex conditional flows. This is especially true for parallel interactions, as the dynamic and sequential nature of sequence diagrams makes it difficult to depict concurrent events within a system. Furthermore, trying to illustrate multiple concurrent and complex conditional flows using frames (or fragments) in a single diagram can make it cluttered and ultimately it may require the generation of several additional sequence diagrams. See Table 2 for a list of the main advantages and disadvantages of sequence diagrams. To summarize UML sequence diagrams are useful for illustrating dynamic interactions of objects within a system that have their advantages and disadvantages. The next section of this paper explores a hospital management system UML sequence diagram, including its related pseudocode and a UML class diagram.

Table 2

Strengths and Limitations of UML Sequence Diagrams

	Strengths	**Limitations**
Illustration	- Visually represent user-described examples, scenarios, or storyboards, especially useful for mobile app development - Depict a sequence of messages between objects within a specific timeframe, showing preconditions - Provide a "snapshot" of system activity during a specific time period.	- Incomplete, showing only snapshots; unsuitable for depicting entire processes or flows - Cannot easily represent conditional logic (if-then-else) or loops (for-next). - Represent only single threads; multiple threads require separate, unrelated diagrams - Representing the entire sequence in one diagram can lead to confusion; only model appropriate subsets.
Object - Class	- Help identify missing objects that can be added as classes in a class diagram - Help identify missing operations/methods within classes, which can be added to a class diagram - Can display multiple objects of the same class ("instance diagrams").	
Design	- Methods with signatures provide detailed design information for designers and programmers - Mapping between sequence and class diagrams improves class diagrams. - Showing object destruction ensures proper deletion and reduces memory clutter - Can show the same message being passed between objects multiple times, emphasizing dynamic behavior.	- The number of sequence diagrams needed is unclear and relies on the designer's judgment - Misaligned usage between system designers and business analysts can be counterproductive.

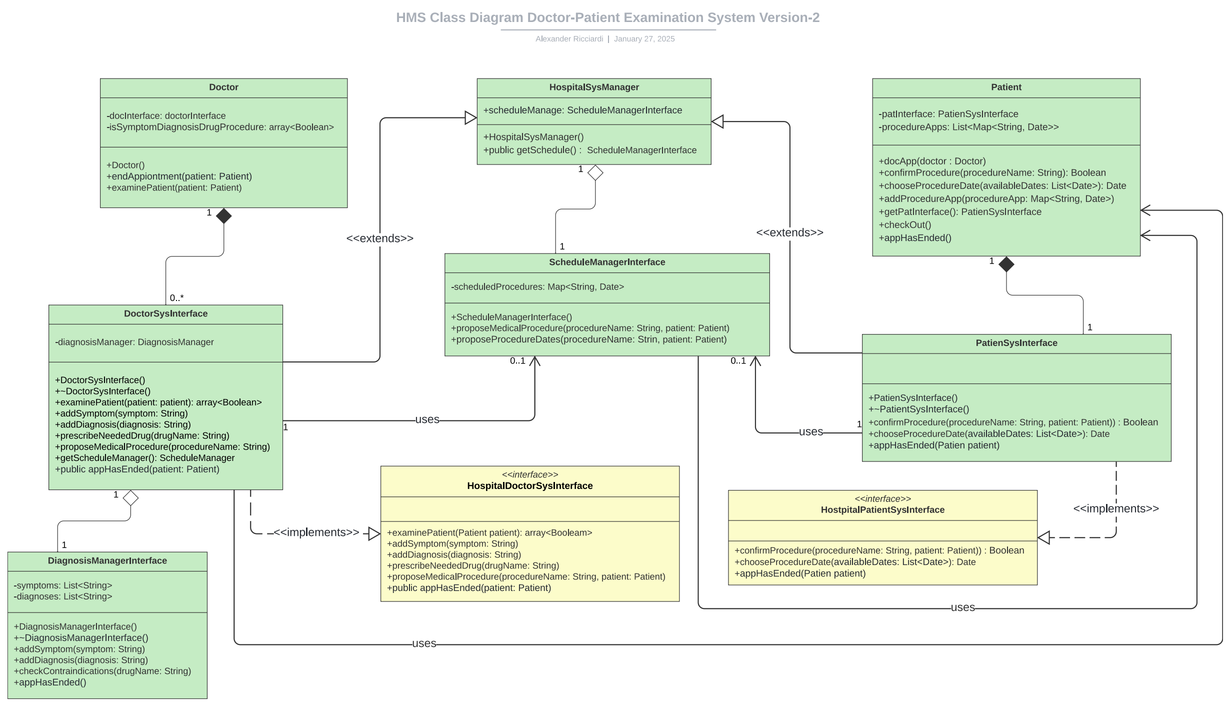
Note: The table describes the main advantages and disadvantages of UML sequence diagrams. From “A Guide to UML Sequence Diagrams: Notation, Strengths, and Limitations” by Ricciardi (2025).

Hospital Management System Sequence Diagram

Hospital Management System (HMS) is a digital application that helps hospitals manage daily operations efficiently (History Medical History, 2024). It is a platform that connects and manages all departments within a hospital, including medical, financial, patient, and administrative departments. As it will be nearly impossible to illustrate every interaction happening within the HSM using a single sequence diagram, this section explores through a sequence diagram how a doctor and a patient may interact with an HMS during an appointment. In other words, the sequence diagram illustrates a doctor-patient examination workflow within an HMS, showing how a doctor and a patient may interact with the HMS interfaces during an appointment.

Figure 2

Doctor-Patient Examination HMS Class Diagram Version-2

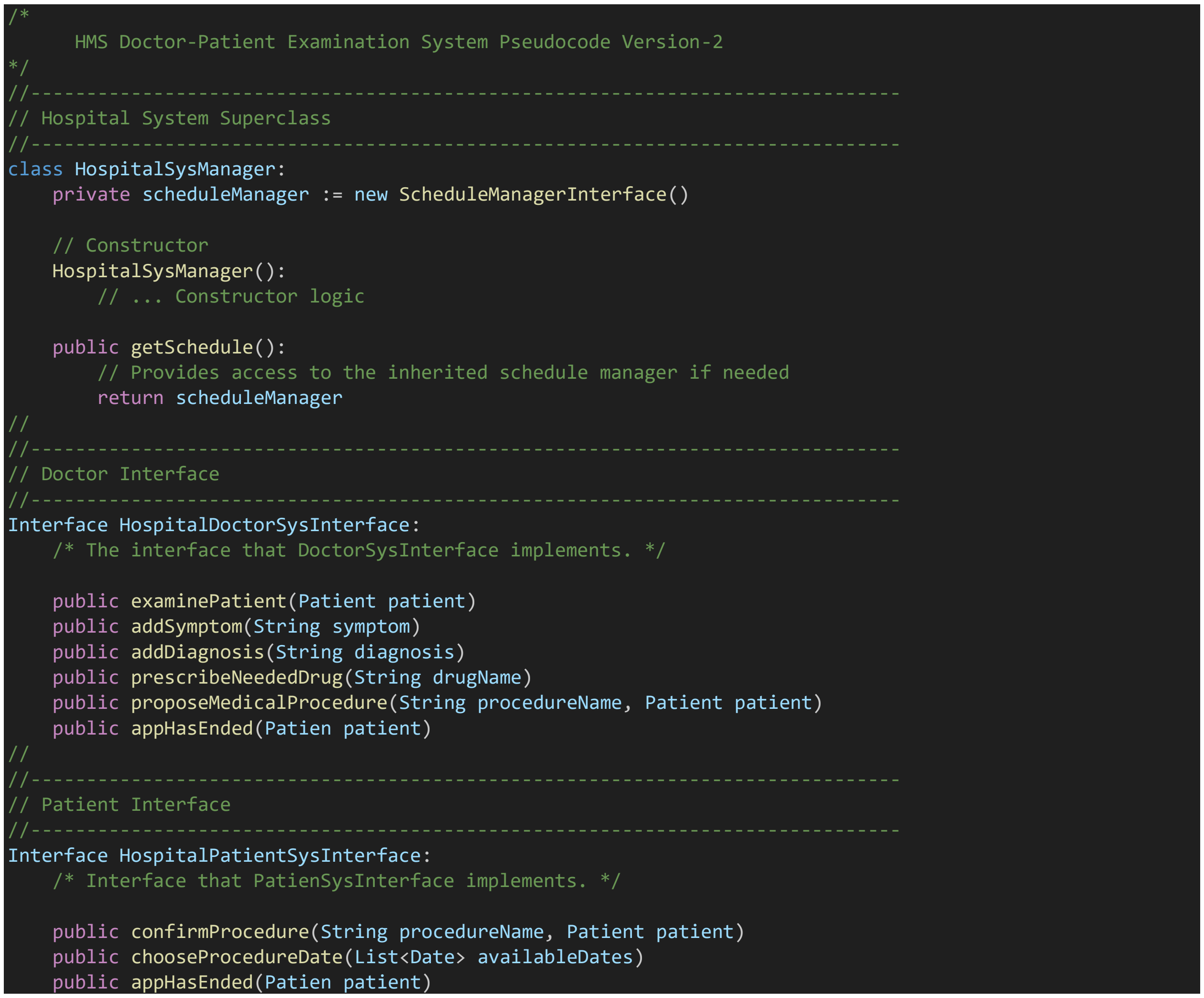

HMS Class Diagram Doctor-Patient Examination System Version-2
Alexander Ricciardi | January 27, 2025
Doctor
-docInterface: doctorInterface
-isSymptomDiagnosisDrugProcedure: array<Boolean>
+Doctor()
+endAppiontment(patient: Patient)
+examinePatient(patient: Patient)
HospitalSysManager
+scheduleManage: ScheduleManagerInterface
+HospitalSysManager()
+public getSchedule() : ScheduleManagerInterface
Patient
-patInterface: PatienSysInterface
-procedureApps: List<Map<String, Date>>
+docApp(doctor : Doctor)
+confirmProcedure(procedureName: String): Boolean
+chooseProcedureDate(availableDates: List<Date>): Date
+addProcedureApp(procedureApp: Map<String, Date>)
+getPatInterface(): PatienSysInterface
+checkOut()
+appHasEnded()
<<extends>>
<<extends>>
ScheduleManagerInterface
-scheduledProcedures: Map<String, Date>
+ScheduleManagerInterface()
+proposeMedicalProcedure(procedureName: String, patient: Patient)
+proposeProcedureDates(procedureName: Strin, patient: Patient)
DoctorSysInterface
-diagnosisManager: DiagnosisManager
+DoctorSysInterface()
+~DoctorSysInterface()
+examinePatient(patient: patient): array<Boolean>
+addSymptom(symptom: String)
+addDiagnosis(diagnosis: String)
+prescribeNeededDrug(drugName: String)
+proposeMedicalProcedure(procedureName: String)
+getScheduleManager(): ScheduleManager
+public appHasEnded(patient: Patient)
PatienSysInterface
+PatienSysInterface()
+~PatientSysInterface()
+confirmProcedure(procedureName: String, patient: Patient)) : Boolean
+chooseProcedureDate(availableDates: List<Date>): Date
+appHasEnded(Patien patient)
uses
uses
<<interface>>
HospitalDoctorSysInterface
+examinePatient(Patient patient): array<Booleam>
+addSymptom(symptom: String)
+addDiagnosis(diagnosis: String)
+prescribeNeededDrug(drugName: String)
+proposeMedicalProcedure(procedureName: String, patient: Patient)
+public appHasEnded(patient: Patient)
<<implements>>
<<interface>>
HostpitalPatientSysInterface
+confirmProcedure(procedureName: String, patient: Patient)) : Boolean
+chooseProcedureDate(availableDates: List<Date>): Date
+appHasEnded(Patien patient)
<<implements>>
DiagnosisManagerInterface
-symptoms: List<String>
-diagnoses: List<String>
+DiagnosisManagerInterface()
+~DiagnosisManagerInterface()
+addSymptom(symptom: String)
+addDiagnosis(diagnosis: String)
+checkContraindications(drugName: String)
+appHasEnded()
uses
uses
1
0..*
1
1
0..1
0..1
1
1
1
1
1
1

Note: This UML class diagram, version 2, illustrates a doctor-patient examination system within a Hospital Management System (HMS). Made with Lucidchart app.

As noted earlier in this paper sequence diagrams are usually based on UML class diagrams. Figure 2 depicts the class diagram that models the static structure of a doctor-patient examination system within a HMS. In Agile-Scrum methodology, this second iteration (version 2) of the class diagram, along with its associated pseudocode and sequence diagram, indicates that the doctor-patient examination feature was refined or modified during the design phase based on usage scenarios captured in the previous sequence diagram version. Below is the pseudocode associated with the feature.

Code Snippet 1

HMS Doctor-Patient Examination System Pseudocode Version-2



```
/*
      HMS Doctor-Patient Examination System Pseudocode Version-2
*/
//--------------------------------------------------------------------------------
// Hospital System Superclass
//--------------------------------------------------------------------------------
class HospitalSysManager:
    private scheduleManager := new ScheduleManagerInterface()

    // Constructor
    HospitalSysManager():
        // ... Constructor logic

    public getSchedule():
        // Provides access to the inherited schedule manager if needed
        return scheduleManager
//
//--------------------------------------------------------------------------------
// Doctor Interface
//--------------------------------------------------------------------------------
Interface HospitalDoctorSysInterface:
    /* The interface that DoctorSysInterface implements. */

    public examinePatient(Patient patient)
    public addSymptom(String symptom)
    public addDiagnosis(String diagnosis)
    public prescribeNeededDrug(String drugName)
    public proposeMedicalProcedure(String procedureName, Patient patient)
    public appHasEnded(Patien patient)
//
//--------------------------------------------------------------------------------
// Patient Interface
//--------------------------------------------------------------------------------
Interface HospitalPatientSysInterface:
    /* Interface that PatienSysInterface implements. */

    public confirmProcedure(String procedureName, Patient patient)
    public chooseProcedureDate(List<Date> availableDates)
    public appHasEnded(Patien patient)
```

```
//
//------------------------------------------------------------------------------
// DoctorSysInterface Class
//------------------------------------------------------------------------------
class DoctorSysInterface extends HospitalSysManager implements HospitalDoctorSysInterface:

    private diagnosesManager: DiagnosisManagerInterface := new DiagnosisManagerInterface()

    // Constructor
    DoctorSysInterface():
        // ... Constructor logic

    // Destructor
    ~DoctorSysInterface():
        // ... Destructor logic

    @override
    public examinePatient(Patient patient):
        private array<Boolean> isSymptomDiagnosisDrugProcedure := [false, false, false, false]
        //  isSymptomDiagnosisDrugProcedure[0] => symptom?
        //  isSymptomDiagnosisDrugProcedure[1] => diagnosis?
        //  isSymptomDiagnosisDrugProcedure[2] => drug needed?
        //  isSymptomDiagnosisDrugProcedure[3] => procedure needed?
        // ... Logic to determine what is needed
        return isSymptomDiagnosisDrugProcedure

    @override
    public addSymptom(String symptom):
        diagnosesManager.addSymptom(symptom)
        return void

    @override
    public addDiagnosis(String diagnosis):
        diagnosesManager.addDiagnosis(diagnosis)
        return void

    @override
    public prescribeNeededDrug(String drugName):
        diagnosesManager.checkContraindications(drugName)
        // ... Logic possibly finalize prescription
        return void

    @override
    public proposeMedicalProcedure(String procedureName, Patient patient):
        // Using inherited scheduleManager from HospitalSysManager
        scheduleManager.proposeMedicalProcedure(procedureName, patient)
        return void

    @override
    public appHasEnded(Patien patient)
        HospitalPatientSysInterface patInterface := patient.getPatInterface()
        // ... logic
        patient.appHasEnded()
        ~DoctorSysInterface()
        return void

    // Optional getter for reusing the same schedule manager
    public getSchedule():
        return scheduleManager
//
//------------------------------------------------------------------------------
// PatienSysInterface Class
//------------------------------------------------------------------------------
class PatienSysInterface extends HospitalSysManager implements HospitalPatientSysInterface:
    // Constructor
    PatienSysInterface():
        // ... Constructor logic

    // Destructor
    ~PatientSysInterface():
    // ... Destructor logic

    @override
    public confirmProcedure(String procedureName, Patient patient):
        private Boolean hasConfirmProcedure := false
        hasConfirmProcedure := patient.confirmProcedure(procedureName)
        return hasConfirmProcedure

    @override
    public chooseProcedureDate(List<Date> availableDates):
        Date desiredDate
        // implement logic for picking one Date from availableDates
        return desiredDate

    @override
    public appHasEnded(Patien patient):
        patient.appHasEnded()
        ~PatientSysInterface()
//
//-------------------------------------------------------------------------------
// DiagnosisManagerInterface Class
//-------------------------------------------------------------------------------
class DiagnosisManagerInterface:
    private List<String> symptoms
    private List<String> diagnoses

    // Constructor
    DiagnosisManagerInterface():
        // ... Constructor logic
        symptoms := []
        diagnoses := []

    // Destructor
    ~DiagnosisManagerInterface():
    // ... Destructor logic

    public addSymptom(String symptom):
        symptoms.add(symptom)
        // ... Possibly analyze symptom, recommend diagnoses
        return void

    public addDiagnosis(String diagnosis):
        diagnoses.add(diagnosis)
        // ... Store/confirm the diagnosis
        return void

    public checkContraindications(String drugName):
        // ... Search for contraindications, return if found
        return void

    public appHasEnded():
        // ... Logic
        ~DiagnosisManagerInterface()
        return void
//
//-------------------------------------------------------------------------------
// ScheduleManagerInterface Class
```

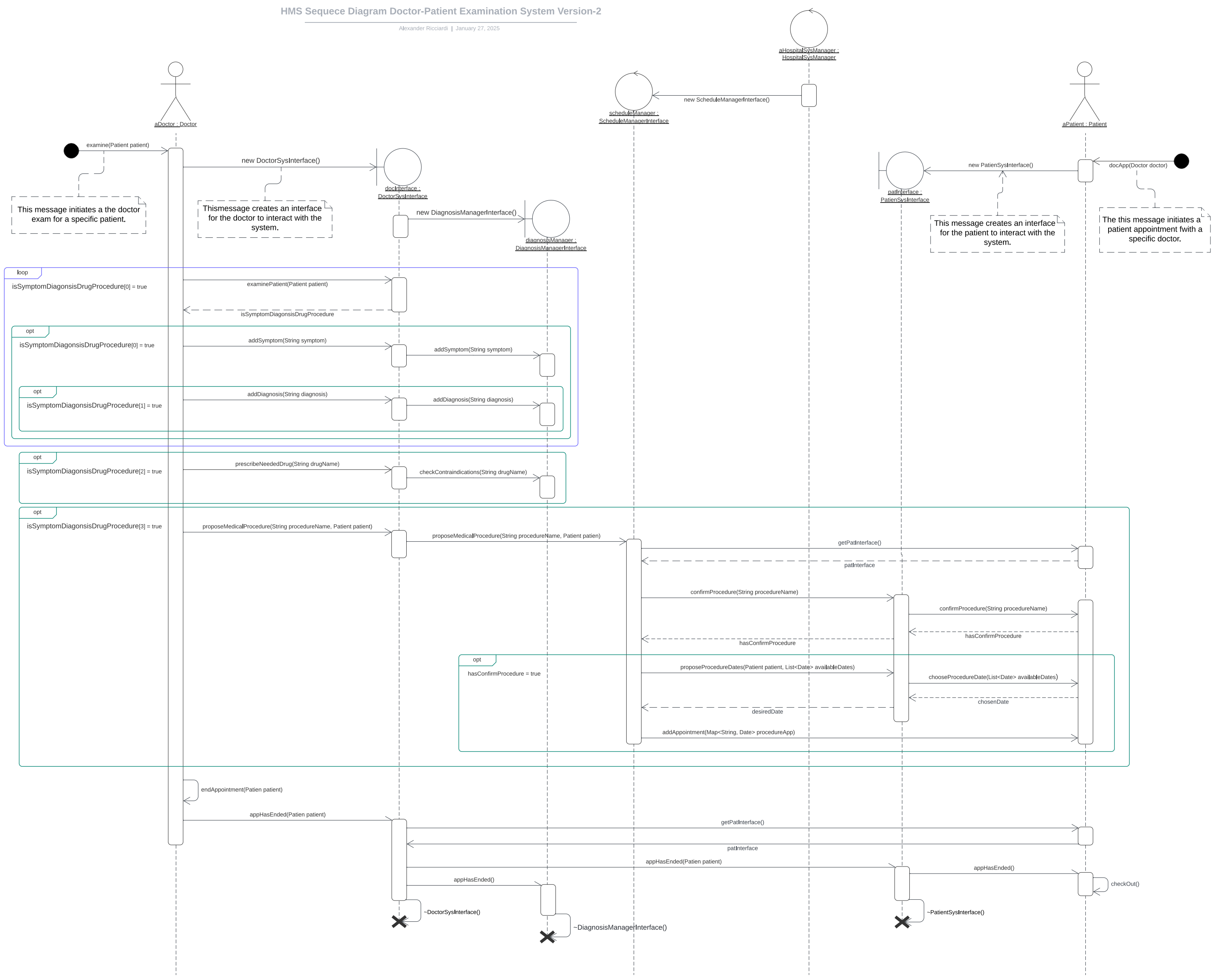
```
//------------------------------------------------------------------------------
class ScheduleManagerInterface:
    /*
       Map procedureName -> [List<Date> availableDates, List<Date> scheduledDates]
    */
    private Map<String, [List<Date>, List<Date>]> proceduresDates

    // Constructor
    ScheduleManagerInterface():
        proceduresDates := Map()
        // ... Constructor logic

    public proposeProcedureDates(String procedureName, Patient patient):
        List<Date> availableDates
        Date desiredDate
        HospitalPatientSysInterface patInterface := patient.getPatInterface()

        // ... Logic to find or generate a list of available dates
        desiredDate := patInterface.chooseProcedureDate(availableDates)
        // ... Add desiredDate to scheduledDates
        patient.addProcedureApp({procedureName : desiredDate})
        return void

    public proposeMedicalProcedure(String procedureName, Patient patient):
        Boolean hasConfirmed := false
        HospitalPatientSysInterface patInterface := patient.getPatInterface()

        hasConfirmed := patInterface.confirmProcedure(procedureName, patient)
        if (hasConfirmed = true)
            proposeProcedureDates(procedureName, patient)

        return void
//
//------------------------------------------------------------------------------
// Doctor Class (external actor who calls the system methods)
//------------------------------------------------------------------------------
class Doctor:
    private docInterface: DoctorSysInterface := new DoctorSysInterface()
    private array<Boolean> isSymptomDiagnosisDrugProcedure := [true, false, false, false]

    // Constructor
    Doctor():
        // ... Constructor logic

    public endAppiontment(Patien patient):
        // ... Logic
        docInterface.appHasEnded(patient)
        return void


    public examine(Patient patient):
        while (isSymptomDiagnosisDrugProcedure[0] == true):
            isSymptomDiagnosisDrugProcedure := docInterface.examinePatient(patient)
            if (isSymptomDiagnosisDrugProcedure[0] == true):
                docInterface.addSymptom("Doctor inputs symptom")
                if (isSymptomDiagnosisDrugProcedure[1] == true):
                    docInterface.addDiagnosis("Doctor inputs diagnosis")

        if (isSymptomDiagnosisDrugProcedure[2] = true):
            docInterface.prescribeNeededDrug("drugName")

        if (isSymptomDiagnosisDrugProcedure[3] = true):
            docInterface.proposeMedicalProcedure("procedureName", patient)
        endAppiontment()
        return void
//
//-------------------------------------------------------------------------------
// Patient Class (external actor who calls the system methods)
//-------------------------------------------------------------------------------
class Patient:
    private patInterface := new PatienSysInterface()
    // List of procedure appointments <procedure name, date>
    private List<Map<String, Date>> procedureApps

    // Constructor
    Patient():
        // ... Constructor logic

    public docApp(Doctor doctor):
        // Implementation not provided
        return void

    public confirmProcedure(String procedureName):
        private Boolean hasConfirmProcedure := false
        // ... Implementation logic, e.g. ask the user for confirmation
        return hasConfirmProcedure

    public chooseProcedureDate(List<Date> availableDates):
        Date chosenDate
        // ... Implementation logic to pick from availableDates
        return chosenDate

    public addProcedureApp(Map<String, Date> procedureApp):
        procedureApps.add(procedureApp)
        return void

    public checkOut():
        // ... Logic
        return void

    public +appHasEnded()
        // ... Logic
        checkOut()
        return void

    public getPatInterface():
        return patInterface
```

Note: The pseudocode provides a code description of the interfaces and classes for the HMS Doctor-Patient Examination System.

The HMS Doctor-patient examination system pseudocode comments label the *Patient Class* and the *Patient Class* as external actors as the objects instantiated by those start their interactions with the system by creating *DoctorSysInterface* and *PatientSysInterface* objects allowing them to interface with the HSM. Note that the *PatientSysInterface Class* and *PatientSysInterface Class* extend the *HospitalSysManager Class.* These relationships are illustrated in the class diagram by the aggregation and inheritance relationships, and it is these relationships that enable the *Doctor* and *Patient* actors to interact with the HMS Doctor-patient examination system as illustrated in the following sequence diagram.

Figure 3

Doctor-Patient Examination HMS Sequence Diagram Version-2


HMS Sequece Diagram Doctor-Patient Examination System Version-2
Alexander Ricciardi | January 27, 2025
aHospitalSysManager : HospitalSysManager
scheduleManager : ScheduleManagerInterface
new ScheduleManagerInterface()
aDoctor : Doctor
aPatient : Patient
examine(Patient patient)
This message initiates a the doctor exam for a specific patient.
new DoctorSysInterface()
Thismessage creates an interface for the doctor to interact with the system.
docInterface : DoctorSysInterface
new DiagnosisManagerInterface()
diagnosisManager : DiagnosisManagerInterface
patInterface : PatienSysInterface
new PatienSysInterface()
docApp(Doctor doctor)
This message creates an interface for the patient to interact with the system.
The this message initiates a patient appointment fwith a specific doctor.
loop
isSymptomDiagonsisDrugProcedure[0] = true
examinePatient(Patient patient)
isSymptomDiagonsisDrugProcedure
opt
isSymptomDiagonsisDrugProcedure[0] = true
addSymptom(String symptom)
addSymptom(String symptom)
opt
isSymptomDiagonsisDrugProcedure[1] = true
addDiagnosis(String diagnosis)
addDiagnosis(String diagnosis)
opt
isSymptomDiagonsisDrugProcedure[2] = true
prescribeNeededDrug(String drugName)
checkContraindications(String drugName)
opt
isSymptomDiagonsisDrugProcedure[3] = true
proposeMedicalProcedure(String procedureName, Patient patient)
proposeMedicalProcedure(String procedureName, Patient patien)
getPatInterface()
patInterface
confirmProcedure(String procedureName)
confirmProcedure(String procedureName)
hasConfirmProcedure
hasConfirmProcedure
opt
hasConfirmProcedure = true
proposeProcedureDates(Patient patient, List<Date> availableDates)
chooseProcedureDate(List<Date> availableDates)
chosenDate
desiredDate
addAppointment(Map<String, Date> procedureApp)
endAppointment(Patien patient)
appHasEnded(Patien patient)
getPatInterface()
patInterface
appHasEnded(Patien patient)
appHasEnded()
appHasEnded()
checkOut()
~DoctorSysInterface()
~DiagnosisManagerInterface()
~PatientSysInterface()

Note: This UML sequence diagram, version 2, illustrates a doctor-patient examination interaction within a Hospital Management System (HMS). Made with Lucidchart app.

The doctor-patient examination system HMS version-2 sequence diagram is a design level (solution space) UML sequence diagram that models the interactions between the external actors, a doctor and a patient, and a doctor-patient examination HMS. The interaction workflow initiates with the message from a found message represented by the method *examine(Patient patient)*, a found message, in this context, is a message generated outside of the system. Similarly, the *docApp(Doctor doctor)* message initiates the workflow from the perspective of the patient. Note that this interaction could be represented as a parallel interaction in the diagram using a parallel (par) frame creating an interaction fragment within the diagram. However, this is not necessary as it will unnecessarily complicate the diagram and add cluttered to it without adding significant value to the understanding and illustration of the interaction. The diagram illustrates very boundaries in the form of the interface *patInterface : PatienSysInterface, docInterface : DoctorSysInterface,* and *diagnosisManager : DiagnosisManagerInterface* showing the layers of abstraction between the external actors themself (the doctor and the patient) and between them and the HMS core system (*aHospitalSysManager : HospitalSysManager*), as well as with the schedule manage (*scheduleManager : ScheduleManagerInterface)*. The external also used those boundaries as interfaces to interact with the system. Additionaly, the diagram illustrates combined fragments; for instance, the loop frame (*isSymptomDiagonsisDrugProcedure[0] = true*) encapsulating the option frame *isSymptomDiagonsisDrugProcedure[0] = true* which encapsulating the option frame *isSymptomDiagonsisDrugProcedure[1] = true*. Furthermore, the sequence diagram illustrates well internal and external actors, object, timeline, focus of control, message, self-message, return message, asynchronous message, object destruction, steps (in a sequence), and notes effectively modeling the dynamic interactions within the doctor-patient examination process of the HMS.

Conclusion

UML sequence diagrams are useful for illustrating dynamic interactions of objects within a system. Sequence diagrams help in identifying missing elements, detailing object interactions, and modeling object destruction. However, they are less effective at representing parallelism, complex conditional logic, or entire complex systems in a single diagram. This essay has also explored the utility of UML sequence diagrams in modeling the dynamic behavior of systems, using a Hospital Management System's doctor-patient examination scenario. It provided a detailed sequence diagram, class diagram, and pseudocode demonstrating how these different tools can be used in the design phase to effectively represent complex interactions, and refine system requirements. As shown by the provided example, in software engineering, they are essential for designing, analyzing, and documenting interactions within a system.

References

Agile Modeling. (2023, November 23). *UML Stereotypes: Diagramming style guidelines*. The Agile Modeling (AM) Method - Effective Strategies for Modeling and Documentation. https://agilemodeling.com/style/stereotype.htm

IBM (2021, March 5). Sequence diagrams. *Rational software modeler.* IBM Documentation. https://www.ibm.com/docs/en/rsm/7.5.0?topic=uml-sequence-diagrams

Ricciardi, A. (2025, January 27). *A Guide to UML Sequence Diagrams: Notation, strengths, and Limitations*. Code Chronicles - Omegapy. https://www.alexomegapy.com/post/a-guide-to-uml-sequence-diagrams-notation-strengths-and-limitations

History Medical History (2024, November 1). *What is a hospital management system and how is HMS good for businesses?* Smart Medical History. https://smartmedhx.ai/trends/what-is-a-hospital-management-system/

Unhelkar, B. (2018). Chapter 12 — Interaction modeling with sequence diagrams. *Software engineering with UML*. CRC Press. ISBN 9781138297432