Object Oriented Design

Object Oriented Design (OOD) is a programming paradigm that focuses on the use of objects to design software systems. OOD is based on the concept of objects, which are instances of classes that encapsulate data and behavior.

In OOD, objects interact with each other through methods, which are functions that define the behavior of the object. Objects can also inherit properties and behavior from other objects through inheritance, which allows for code reuse and modular design.

OOD is widely used in software development because it promotes code reusability, maintainability, and scalability. It also allows for the creation of complex systems that can be broken down into smaller, more manageable parts.

When designing with OOD, it is important to consider the principles of SOLID. SOLID is an acronym for the following principles:

- Single Responsibility Principle (SRP): A class should have only one reason to change.
- Open-Closed Principle (OCP): A class should be open for extension but closed for modification.
- Liskov Substitution Principle (LSP): Subtypes should be substitutable for their base types.
- Interface Segregation Principle (ISP): A class should not be forced to implement interfaces it does not use.
- Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions.

By following these principles, the design of the system becomes more flexible, testable, and maintainable.

Why would we do OO design?

We do object-oriented (OO) design to develop software systems that are scalable, maintainable, and adaptable to changes. OO design helps us to model complex systems in terms of objects, which are entities that have properties (attributes) and behavior (methods). By modeling the system in terms of objects, we can organize the code in a modular and reusable way, making it easier to maintain and extend the system over time.

OO design also allows us to implement the principles of abstraction, encapsulation**封装**, inheritance, and polymorphism**多态性** ,which are key concepts of OO programming. By using these principles, we can create more modular and flexible code that is easier to understand and modify. Additionally, OO design helps us to focus on the problem domain rather than the technical details of the implementation, which can lead to more effective and efficient software solutions.

- Organise ideas
- plan work
- build understanding of the system structure and behaviour
- Communicate with development team
- Help(future) maintenance team to understand

Class diagrams

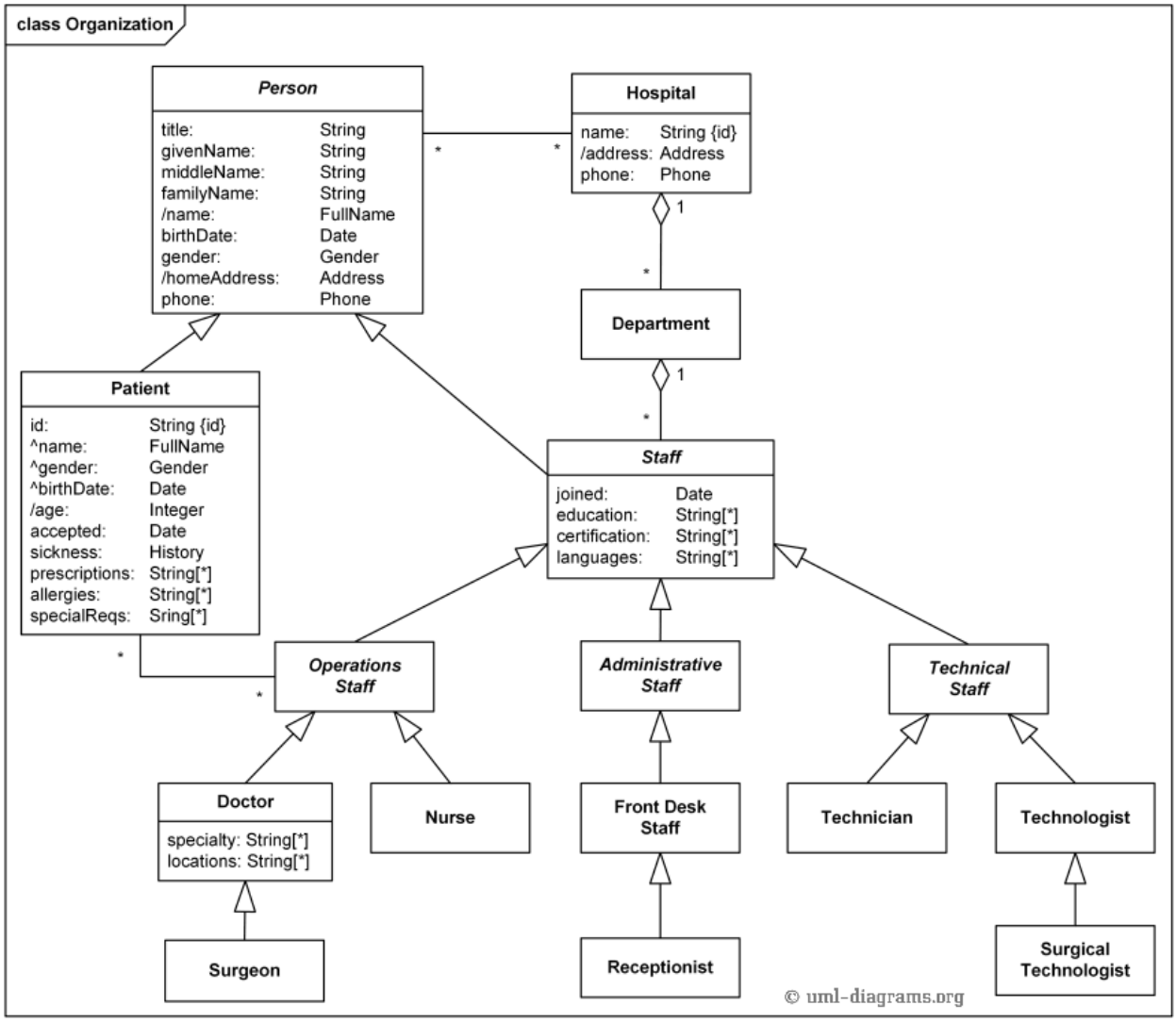
(Static view 静态视图 of a system)

he structure of a system by showing the classes of objects within the system, their attributes, and the relationships between them.

Class diagrams are a fundamental component of UML (Unified Modeling Language) and are used to visualize, document, and construct the software system's architecture.

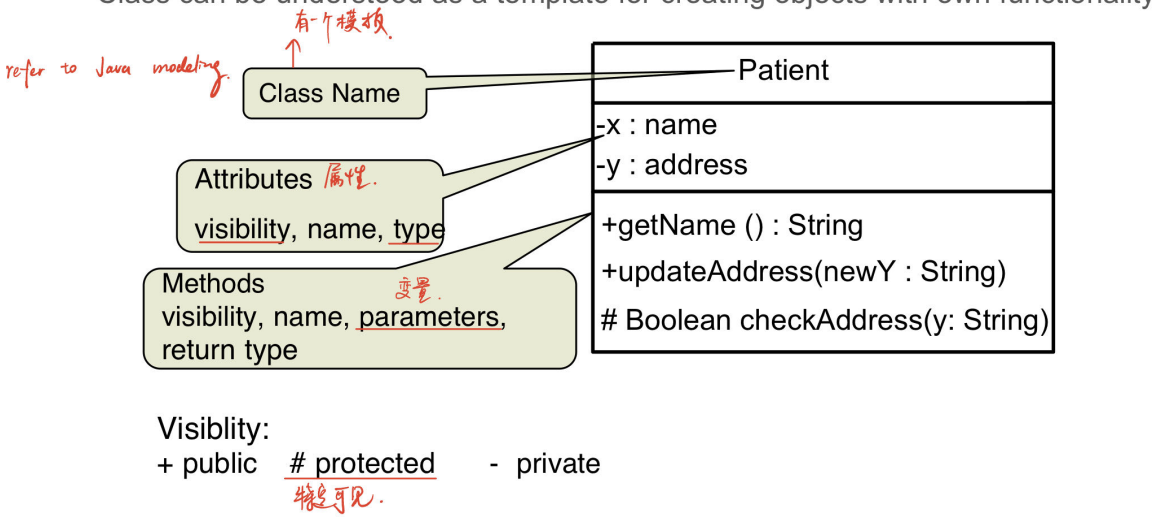
They provide a high-level view of the system's structure and are used to identify the different classes of objects that make up the system, their relationships, and their attributes and operations.

Class diagrams are an essential tool for software developers in the design and implementation of object-oriented systems.


class Organization
Person
title: String
givenName: String
middleName: String
familyName: String
/name: FullName
birthDate: Date
gender: Gender
/homeAddress: Address
phone: Phone
Hospital
name: String {id}
/address: Address
phone: Phone
Department
Patient
id: String {id}
^name: FullName
^gender: Gender
^birthDate: Date
/age: Integer
accepted: Date
sickness: History
prescriptions: String[*]
allergies: String[*]
specialReqs: Sring[*]
Staff
joined: Date
education: String[*]
certification: String[*]
languages: String[*]
Operations Staff
Administrative Staff
Technical Staff
Doctor
specialty: String[*]
locations: String[*]
Nurse
Front Desk Staff
Technician
Technologist
Surgeon
Receptionist
Surgical Technologist
© uml-diagrams.org

Class Diagrams

Class can be understood as a template for creating objects with own functionality



Notation for attributes

In UML class diagrams, attributes are represented as a list of name-value pairs located inside the class box. The general syntax for representing attributes is:

<<visibility>> name : type [= default]

Where:

- <<visibility>> specifies the visibility of the attribute, and can be one of the following:
  - "+" for public visibility
  - "-" for private visibility
  - "#" for protected visibility
  - "~" for package visibility (not common)
- name is the name of the attribute.
- type is the data type of the attribute.
- [= default] is an optional field that specifies the default value of the attribute.

For example, if we have a class called "Person" with two attributes, "name" and "age", the class diagram notation for these attributes could be:

- name : string
- age : int

This indicates that both "name" and "age" are public attributes, "name" is of type string, and "age" is of type int.

Properties: readOnly, union, subsets<property-name>, redefined<property-name?, odered, bag seq ,composite

Static attributes appear underlined

Notation for operations

<visibility> <operation name>(<parameter list>): <return type>

where:

- `visibility` specifies the visibility of the operation and can be one of the following:
  - `+` for public
  - ` ` for private
  - `#` for protected
  - `~` for package (default)
- `operation name` is the name of the method
- `parameter list` is a comma-separated list of input parameters, each with the format `<name>: <type>`
- `return type` is the data type of the return value (if any)

For example, the following operation represents a public method called `getPrice` that takes no parameters and returns a double value:

+ getPrice(): double

Notation for Operations

[visibility] name ([parameter-list]) : [{property}]

- visibility
- method name
- formal parameter list, separated by commas:
  - direction name : type [multiplicity] = value [{property}]
  - static operations are underlined
- Examples:
  - display()
  - - hide()
  - + toString() : String
  - createWindow(location:Coordinates, container: Container): Window

How do we find classes: grammatical parse(找到名词)

Grammatical parsing of the requirements. This involves examining the natural language of the requirements to identify the nouns and verbs used and then grouping them into candidate classes and operations.

For example, consider the following requirement: "The system must allow users to create and edit documents." In this requirement, the nouns are "system," "users," and "documents," while the verbs are "create" and "edit." From this, we can identify two potential classes: "User" and "Document," with the operations "create" and "edit" belonging to the "Document" class.

Grammatical parsing can be a useful starting point for identifying classes, but it is not always sufficient on its own. Other techniques, such as domain analysis and scenario analysis, may also be necessary to fully identify and refine the classes needed for a system.

Classes

- identify nouns from existing text (identify noun)
- narrow down to remove (whether the noun is relevant to your software)

How to define

-duplicates and variations (同义词)

-irrelevant

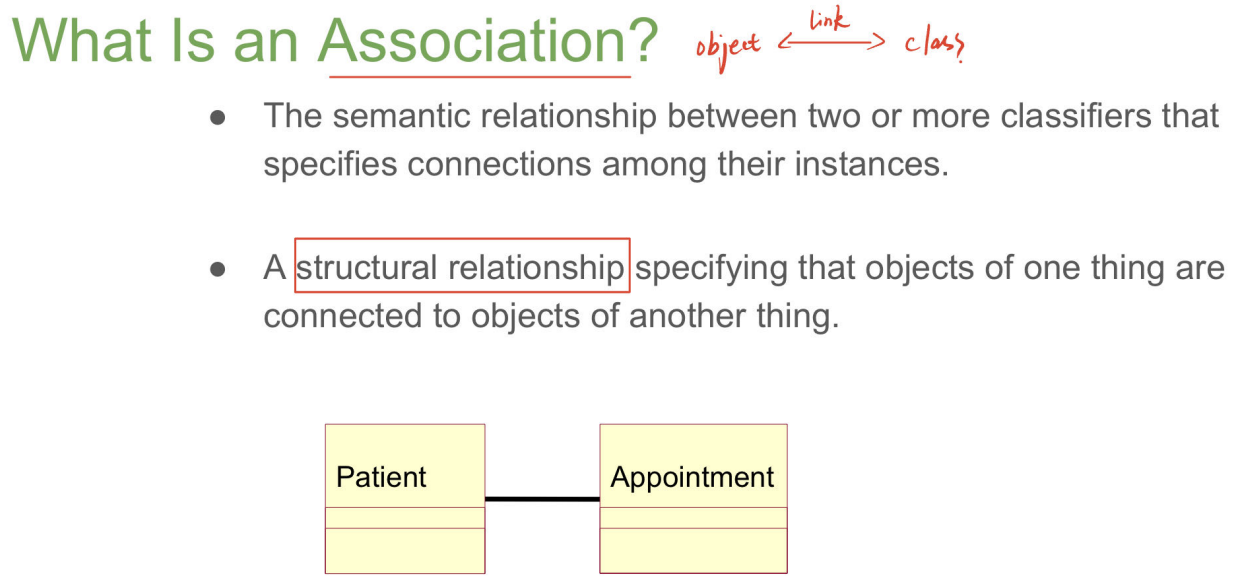
-out of scope

Structural relationships in class diagrams

1. Association**联合**: An association is a structural relationship between two classes that describes a connection or link between them. Associations can be one-to-one, one-to-many, or many-to-many.
2. Aggregation**聚合**: Aggregation is a special type of association that represents a whole-part relationship between two classes. The whole-part relationship implies that one class is composed of one or more instances of another class. In an aggregation relationship, the whole class owns the part class, and the part class can exist independently of the whole class.
3. Composition: Composition is a more specific form of aggregation where the part class is an integral part of the whole class. This means that the part class cannot exist independently of the whole class.
4. Inheritance: Inheritance is a relationship between two classes where one class (the subclass) inherits the attributes and operations of another class (the superclass). The subclass is a specialized version of the superclass, and it can add new attributes and operations or override existing ones.
5. Dependency**从属**: Dependency is a relationship between two classes where one class depends on the other class for some functionality. The dependent class uses the functionality of the other class, but the other class is not dependent on the dependent class.

6. Multiplicity **多样性** :It represents the number of objects that can exist on each end of a relationship between classes. It is often depicted using numbers or ranges placed near the end of the association line. For example, a customer may have one or many orders, while an order may belong to only one customer. In this case, the multiplicity of the relationship between Customer and Order would be "1 to many" or "1..*" (read as "one to many").

Association

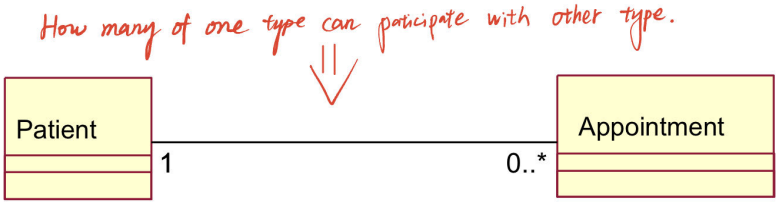


A relationship between two or more classes that represents a connection or interaction between them. It is represented in a class diagram by a line that connects the classes involved and may include an optional name, role, and multiplicity. An association can also have a direction, indicating the flow of communication between the classes. Associations can be uni-directional, bi-directional, or reflexive (self-referencing). They can also be qualified with additional attributes, such as an association class or constraints. Overall, associations help to model the relationships and dependencies between classes in a system.

Multiplicity

What Is Multiplicity?

- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
    - For each instance of Patient, many or no Appointments can be made.
    - For each instance of Appointment, there will be one Patient to see.



Multiplicity is a notation that specifies the number of instances of one class that can be associated with the instances of another class. It represents the cardinality of a relationship between two classes.

Multiplicity is represented using two integers separated by a comma and enclosed in curly braces. The number on the left side of the comma indicates the minimum number of instances, while the number on the right side of the comma indicates the maximum number of instances. The following are some examples of multiplicity notation:

- "1" - specifies that there is exactly one instance
- "0..1" - specifies that there can be zero or one instance
- "1..*" - specifies that there can be one or more instances
- "0..*" - specifies that there can be zero or more instances

Multiplicity can be specified on both ends of an association, indicating the minimum and maximum number of instances that can be associated.

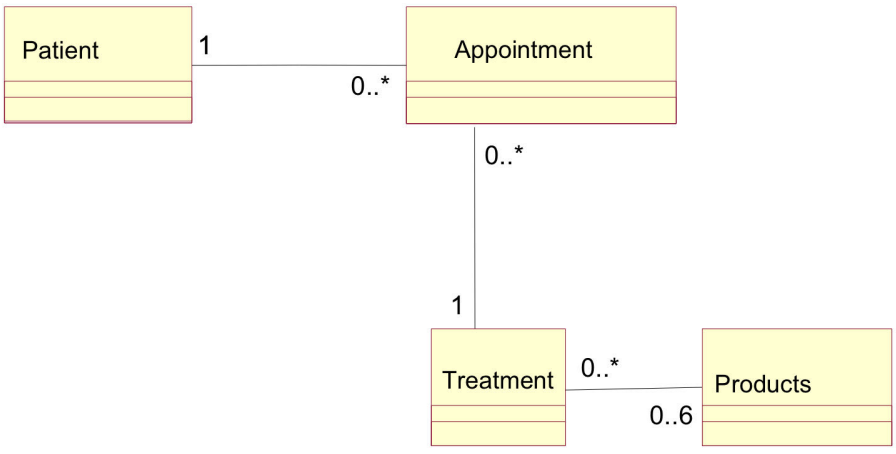
Multiplicity Indicators

how many objects you are allowing to participate in those relationships?

come from — domains 等等领域.

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional value)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

Example: Multiplicity



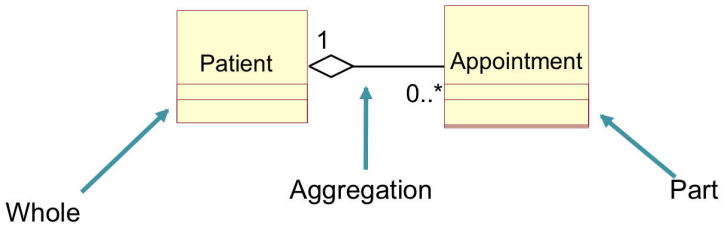
Aggregation

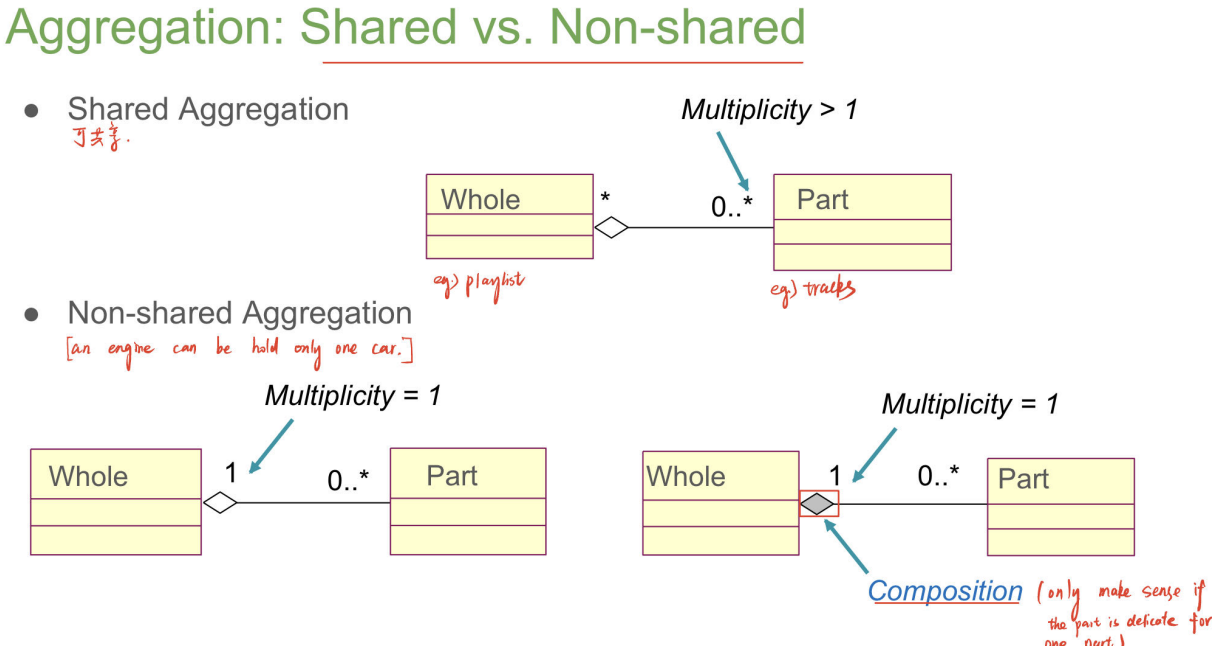
A part-whole or a "has-a" relationship between two classes, where one class is a part of another class. In other words, aggregation is a relationship between two objects where one object contains the other object as a part. Aggregation is represented in a class diagram using a diamond-shaped symbol on the side of the whole class, with a line connecting it to the part class. Aggregation is a weak form of association, which means that the part class can exist without the whole class, and it can be part of another whole class as well.

聚合是两个对象之间的关系，其中一个对象包含另一个对象的一部分。聚合在类图中用一个菱形符号表示，在整个类的一侧，用一条线把它和部分类连接起来。聚合是一种弱的关联形式，这意味着部分类可以在没有整体类的情况下存在，而且它也可以是另一个整体类的一部分。

What Is an Aggregation?

- A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts.
  - An aggregation is an “is a part-of” relationship.
- Multiplicity is represented like other associations.





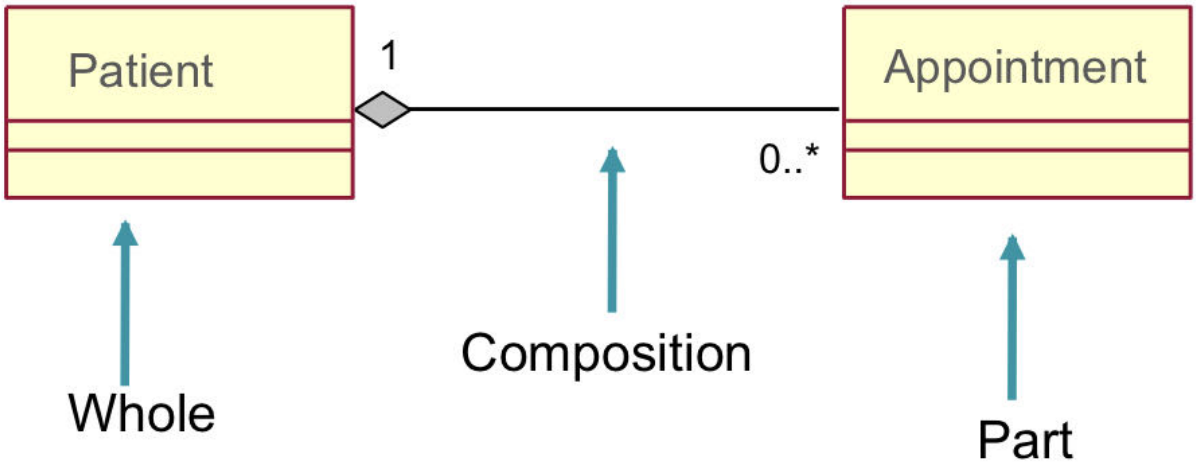
Non-shared aggregation, also known as "composition," indicates that the whole object is responsible for the lifecycle of the part object. In other words, the part object cannot exist independently of the whole object. In the class diagram notation, non-shared aggregation is represented by a filled diamond (实心菱形)at the end of the association line that points to the part class.

Shared aggregation, on the other hand, indicates that the part object can exist independently of the whole object, and can be shared among multiple whole objects. In the class diagram notation, shared aggregation is represented by an unfilled diamond(空心棱形) at the end of the association line that points to the part class.

What is Composition?

Composition is a type of association between two classes where the lifetime of one object is dependent on the lifetime of the other object. In composition, a whole object is made up of its parts, and the parts cannot exist independently outside of the whole object. This means that when the whole object is destroyed, its parts are also destroyed. In a class diagram, composition is represented by a filled diamond shape on the end of the association line that points towards the whole object.

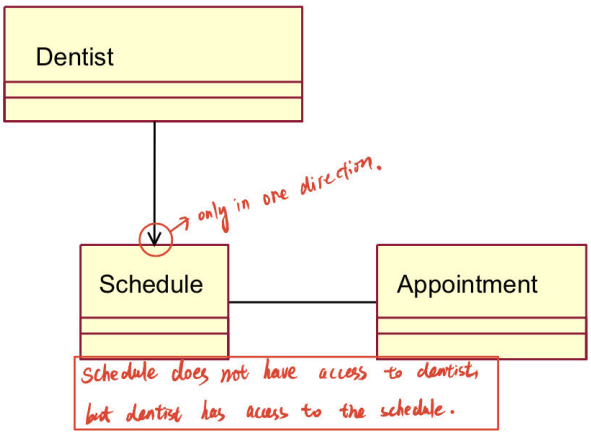
- a form of aggregation with strong ownership and coincident **一致的** lifetimes
- the parts cannot survive the whole/aggregate



What is Navigability?

What Is Navigability?

- Indicates that it is possible to navigate from an associating class to the target class using the association



The ability to traverse an association in a specific direction. In other words, it is the ability to navigate from one class to another through an association in a particular direction. In a class diagram, the navigability of an association is indicated by an arrowhead at one end of the association line. The presence or absence of an arrowhead indicates the direction in which the association can be navigated. If the arrowhead is present at one end of the association line, it means that the association can be navigated in that direction. If the arrowhead is absent, it means that the association can be navigated in both directions.

在一个特定方向上遍历一个关联的能力。换句话说，它是通过一个关联在特定的方向上从一个类导航到另一个类的能力。在类图中，关联的导航性由关联线一端的箭头表示。箭头的存在或不存在表明关联可以被导航的方向。如果箭头出现在关联线的一端，意味着关联可以朝那个方向导航。如果箭头不存在，意味着关联可以在两个方向上导航。

Generalization

- A relationship among classes where one class shares the properties and or behaviour of one or more classes.
- Define a hierarchy of abstractions were a subclass inherits from one or more superclasses.
- is an “is a kind of” relationship.

one class, known as the subclass or child class, is a specialization of the other class, known as the superclass or parent class.

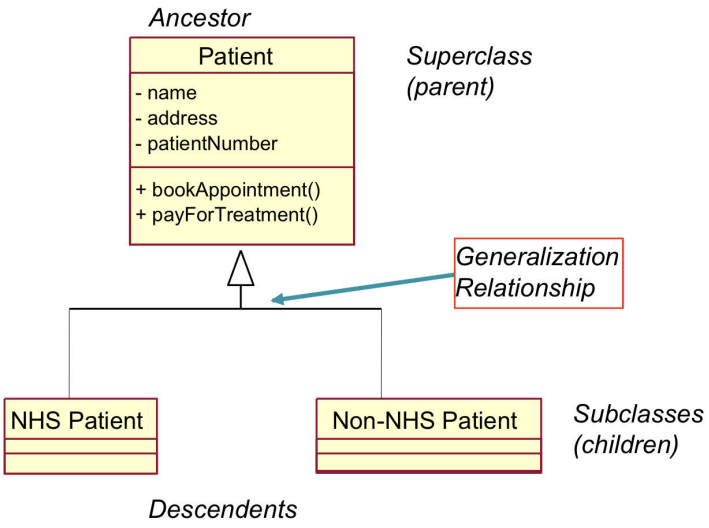
The child class inherits attributes and methods from the parent class, and can also add new attributes and methods or override the behavior of the parent class.

Generalization is a way of creating a hierarchical structure of classes with increasing levels of abstraction and generality. It allows for code reuse, simplifies maintenance, and makes the design more flexible and extensible.

泛化是一种创建类的层次结构的方式，其抽象性和通用性的水平不断提高。它允许代码重用，简化维护，并使设计更加灵活和可扩展。

Example: Inheritance

- One class inherits from another
- Follows the “is a” style of programming
- Class substitutability



Abstract and concrete classes

In object-oriented programming, a class can be either abstract 抽象 or concrete 具体.

An abstract class is a class that cannot be instantiated directly, meaning you cannot create an object of that class. Instead, it provides a template for other classes to inherit from and implement their own specific behavior. Abstract classes often define abstract methods, which are methods that have no implementation and must be implemented by any concrete subclass that inherits from the abstract class. (can not have any objects)

抽象类是一个不能直接实例化的类，这意味着你不能创建该类的一个对象。相反，它为其他类提供了一个模板，使其可以继承并实现自己的特定行为。抽象类经常定义抽象方法，这些方法没有实现，必须由继承自抽象类的任何具体子类来实现。

A concrete class, can be instantiated and used to create objects. It is a class that provides specific implementations for all of its methods and can be used as-is. A concrete class can inherit from another concrete class or from an abstract class. (can have objects)

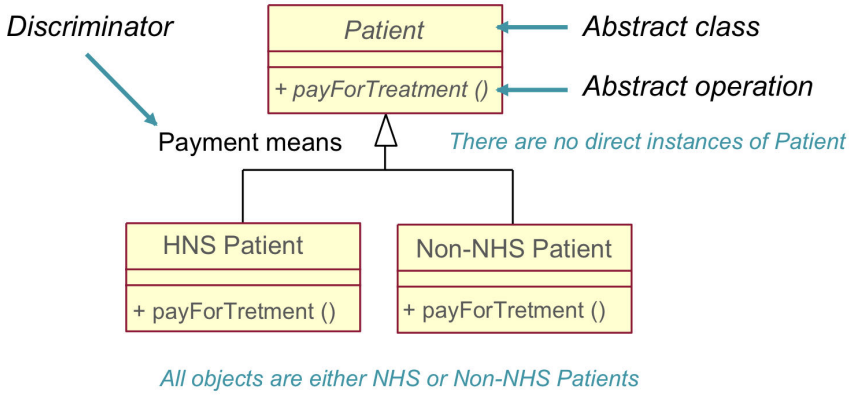
另一方面，具体类可以被实例化并用于创建对象。它是一个为其所有方法提供具体实现的类，可以按原样使用。一个具体的类可以继承自另一个具体的类或一个抽象的类。

In general, abstract classes are used to define a common interface for a group of related classes, while concrete classes are used to provide specific implementations of that interface.

一般来说，抽象类被用来为一组相关的类定义一个共同的接口，而具体类则被用来提供该接口的具体实现。

Abstract and Concrete Classes

- Abstract classes cannot have any objects
- Concrete classes can have objects



"Patient" is shown as an abstract class because it is not a concrete object that can exist on its own in the system. Instead, it is meant to represent a general concept or idea of a patient, with specific details and behaviors defined by its subclasses (e.g., "Outpatient", "Inpatient", etc.).

"病人 "被显示为一个抽象的类，因为它不是一个可以在系统中独立存在的具体对象。相反，它是为了代表一个病人的一般概念或想法，具体细节和行为由其子类定义（例如，"门诊病人"、"住院病人 "等）。

In other words, "Patient" is an abstract class because it is not meant to be instantiated directly, but rather serves as a template or blueprint for creating more specific types of patients. This allows for greater flexibility and modularity in the design of the system, as different types of patients can be defined and customized based on their unique requirements and characteristics.

"病人 "被显示为一个抽象的类，因为它不是一个可以在系统中独立存在的具体对象。相反，它是为了代表一个病人的一般概念或想法，具体细节和行为由其子类定义（例

如，"门诊病人"、"住院病人 "等）。

在本图这种情况中，如果我们想要说设定concrete class;

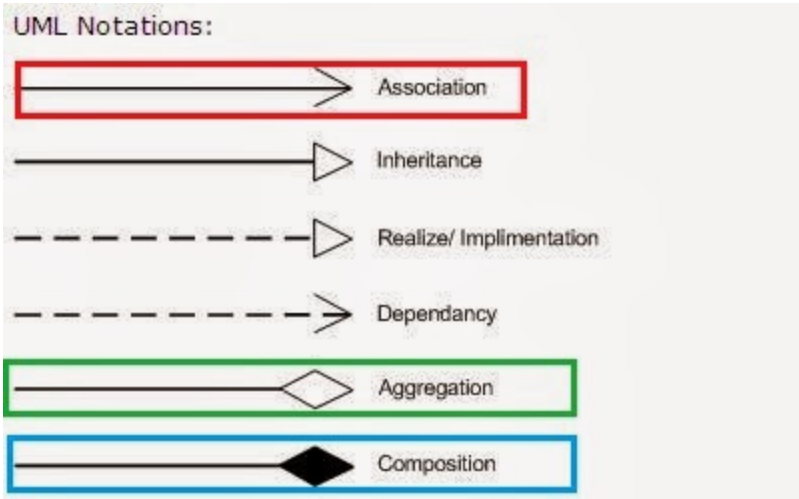
concrete class 就要具体一些， 如：`SurgicalPatient` and `NonSurgicalPatient` can be concrete classes. These classes extend the abstract `Patient` class and provide specific implementation details that are not included in the abstract class.

图中的child class, NHS patients and Non-NHS patients 就是 concrete class.

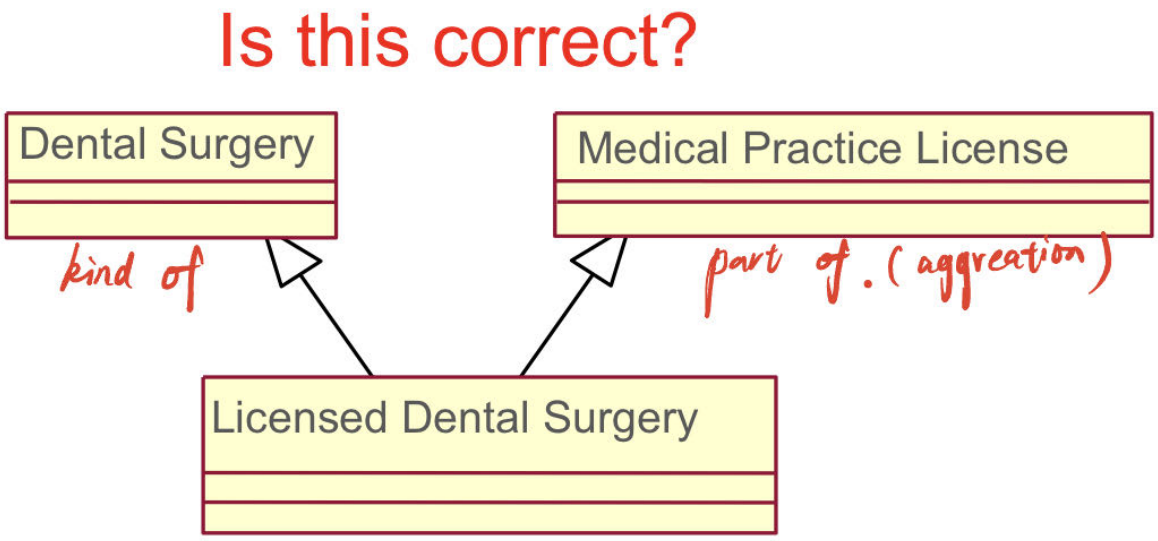
Generalization vs. Aggregation

Generalization represents the relationship between a more general class (the superclass or parent class) and a more specific class (the subclass or child class). The subclass inherits attributes and behaviors from the superclass and can also add its own unique attributes and behaviors. This relationship is shown using an inheritance arrow and an open triangle pointing to the superclass (如下图中的第二条). “is - a” relationship

Aggregation represents the relationship between a whole and its parts. It is a type of association where one class is composed of one or more instances of another class. The parts can exist independently of the whole and can be shared among other wholes. This relationship is shown using a diamond shape(棱形) on the whole class and a line with a diamond at the end pointing to the part class. “has-a” relationship



In class question:



The relationship between `Licensed Dental Surgery` and `Dental Surgery` is ***generalization,*** meaning that `Licensed Dental Surgery` is a more specialized version of `Dental Surgery`.

The relationship between `Licensed Dental Surgery` and `Medical Practice License` is ***aggregation***, meaning that `Licensed Dental Surgery` contains a `Medical Practice License` object as a part of its structure.

Behaviour modelling

The process of representing the behavior of a system or a component of a system as a set of dynamic interactions between actors (users, other systems, hardware devices, etc.) and the system or component being modeled. It involves describing the behavior of a system from the perspective of the external actors that interact with it and the responses of the system to those interactions. Behavior modeling is typically used in software engineering to develop models of software systems that accurately capture their behavior in response to different inputs or stimuli.

Objects Need to Collaborate

- Objects are useless unless they can collaborate to solve a problem.
  - Each object is responsible for its own behavior and status.
  - No one object can carry out every responsibility on its own.

- How do objects interact with each other?
  - They interact through messages.
  - Message shows how one object asks another object to perform some activity.

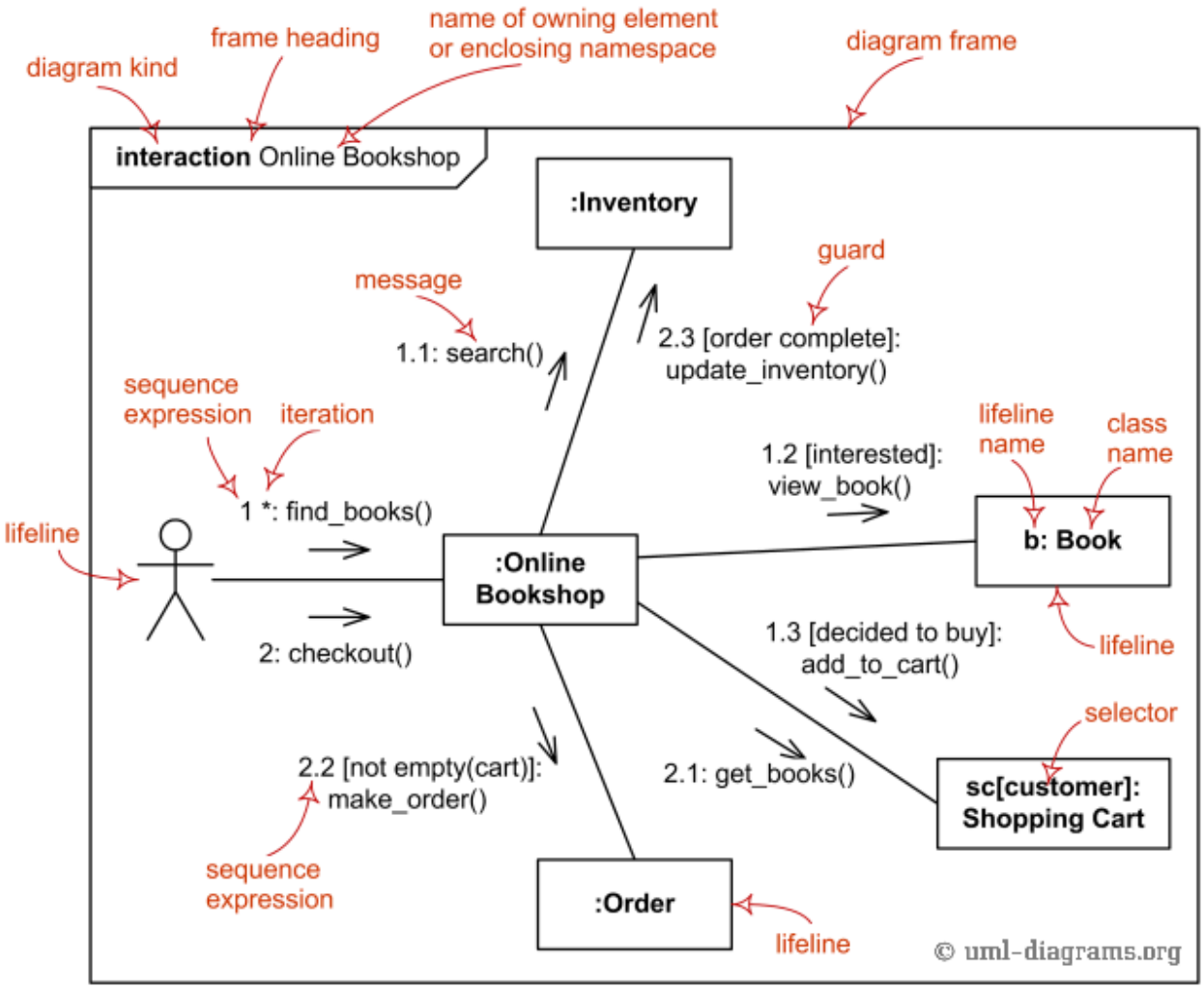
Communication diagrams

Communication diagrams, also known as collaboration diagrams, are a type of interaction diagram that show the interactions between objects or actors in a system to accomplish a specific task or functionality. (emphasize the organisation of the objects that participate in an interaction)

Communication diagrams are used to model the dynamic behavior of a system and show how the objects or actors collaborate with each other to achieve a particular goal or use case. They illustrate the sequence of messages and calls exchanged between objects or actors and help to visualize the flow of data and control between them. Communication diagrams are useful for visualizing complex interactions in a system and identifying potential design issues or improvements.

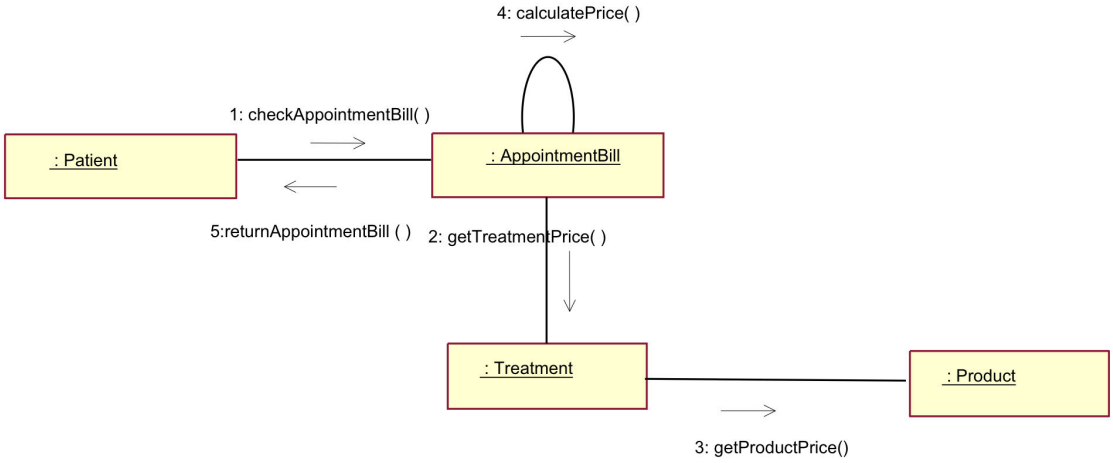
It shows:

- the objects participating in the interaction;
- links between the objects;
- messages passed between the object.



example in class:

Example: Communication Diagram



Sequence diagrams:

A sequence diagram is a type of interaction diagram that illustrates how objects interact with each other in a particular scenario or use case.

It represents the sequence of messages exchanged between the objects or components involved in the scenario, and the order in which they are sent and received.

The objects are depicted as vertical lines or lifelines, and the messages as arrows connecting them. The sequence diagram can be used to model the flow of control or the flow of data in a system, and can be used to identify potential design issues or inefficiencies in the system.

Some rules for creating sequence diagrams:

1. Actors, objects, and lifelines should be clearly defined and labeled.
2. Messages should be properly labeled and ordered chronologically.
3. Activation bars should be drawn to represent the time an object or actor is active in the scenario.
4. Loops and conditions should be clearly marked using combined fragments.

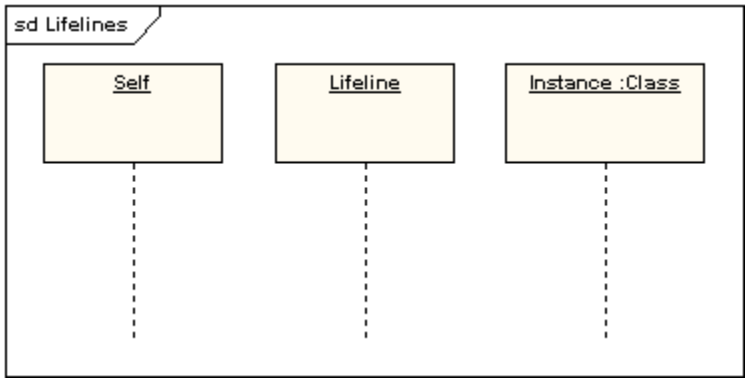
5. Fragments should be properly labeled to indicate their meaning.
6. Synchronous messages should be represented using a solid arrow, while asynchronous messages should be represented using a dashed arrow.
7. The diagram should be organized in a way that is easy to read and follow, with actors and objects listed on the left and messages flowing from top to bottom.
8. The diagram should be consistent with the corresponding use case or scenario.

<tutorial from : https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html>

Lifelines

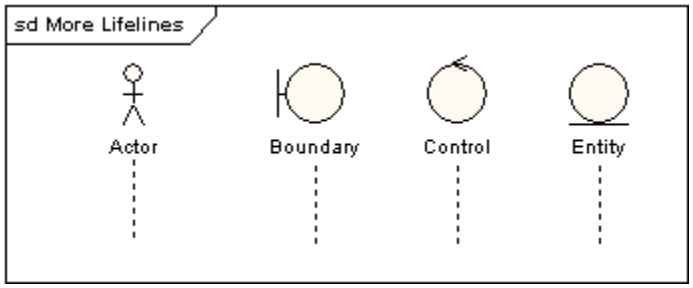
A lifeline represents an individual participant in a sequence diagram. A lifeline will usually have a rectangle containing its object name. If its name is "self", that indicates that the lifeline represents the classifier which owns the sequence diagram.

一条生命线代表了序列图中的一个单独的参与者。一个生命线通常有一个包含其对象名称的矩形。如果它的名字是 "self"，这表明生命线代表拥有该序列图的分类器。



Sometimes a sequence diagram will have a lifeline with an actor element symbol at its head. This will usually be the case if the sequence diagram is owned by a use case. Boundary, control and entity elements from robustness diagrams can also own lifelines.

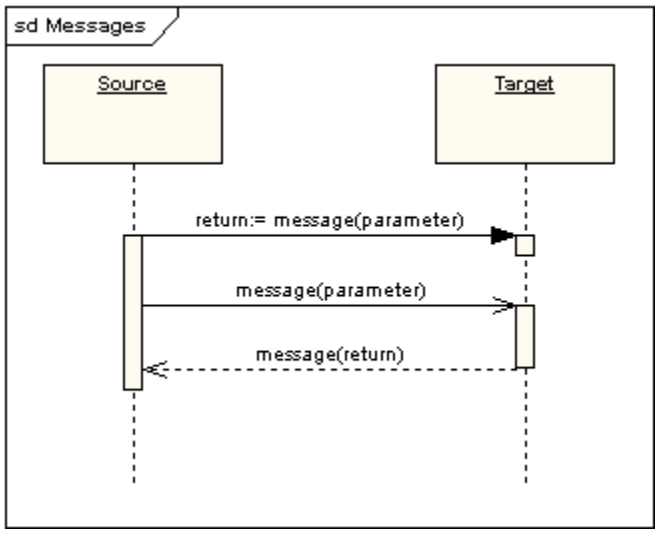
有时，一个序列图将有一条生命线，在它的头部有一个行为者元素符号。如果序列图被一个用例所拥有，通常会出现这种情况。稳健性图中的边界、控制和实体元素也可以拥有生命线。



Messages

Messages are displayed as arrows. Messages can be complete, lost or found; synchronous or asynchronous; call or signal. In the following diagram, the first message is a synchronous message (denoted by the solid arrowhead) complete with an implicit return message; the second message is asynchronous (denoted by line arrowhead), and the third is the asynchronous return message (denoted by the dashed line).

信息显示为箭头。消息可以是完整的、丢失的或找到的；同步的或异步的；调用或信号。在下图中，第一条消息是一条同步消息（由实心箭头表示），完成了一个隐含的返回消息；第二条消息是异步的（由线箭头表示），第三条是异步的返回消息（由虚线表示）。



Execution Occurrence 执行的发生

A thin rectangle running down the lifeline denotes the execution occurrence, or activation of a focus of control. In the previous diagram, there are three execution occurrences. The first is the source object sending two messages and receiving two replies; the second is the target object receiving a synchronous message and returning

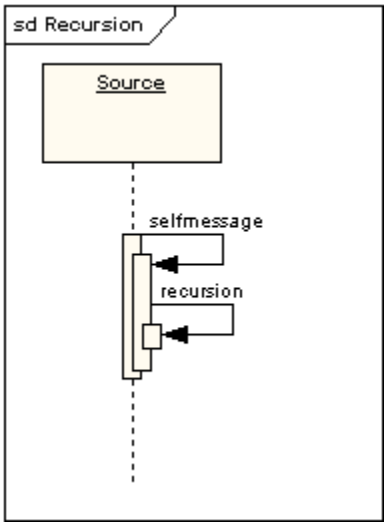
a reply; and the third is the target object receiving an asynchronous message and returning a reply.

沿着生命线的一个细长的矩形表示执行的发生，或控制焦点的激活。在前面的图中，有三个执行事件。第一个是源对象发送两个消息并接收两个回复；第二个是目标对象接收一个同步消息并返回一个回复；第三个是目标对象接收一个异步消息并返回一个回复。

Self Message

A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object. It is shown as creating a nested focus of control in the lifeline's execution occurrence.

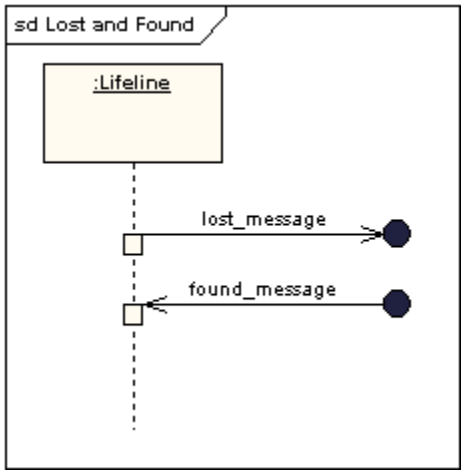
自我信息可以代表一个操作的递归调用，或者一个方法调用属于同一个对象的另一个方法。它被显示为在生命线的执行发生中创建一个嵌套的控制焦点。



Lost and Found Messages

Lost messages are those that are either sent but do not arrive at the intended recipient, or which go to a recipient not shown on the current diagram. Found messages are those that arrive from an unknown sender, or from a sender not shown on the current diagram. They are denoted going to or coming from an endpoint element.

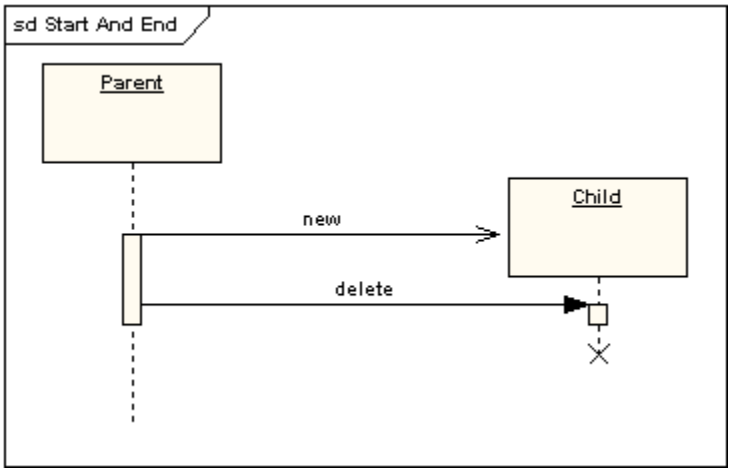
遗失的信息是指那些已经发送但没有到达预定收件人的信息，或者发送到当前图表中没有显示的收件人。找到的信息是那些来自未知发件人的信息，或来自当前图中未显示的发件人的信息。它们被表示为去往或来自一个端点元素。



Lifeline Start and End

A lifeline may be created or destroyed during the timescale represented by a sequence diagram. In the latter case, the lifeline is terminated by a stop symbol, represented as a cross. In the former case, the symbol at the head of the lifeline is shown at a lower level down the page than the symbol of the object that caused the creation. The following diagram shows an object being created and destroyed.

生命线可以在序列图所代表的时间范围内被创建或销毁。在后一种情况下，生命线被一个停止符号所终止，表示为一个十字。在前一种情况下，生命线头部的符号显示在页面下方，比导致创建的对象的符号要低。下图显示了一个正在创建和销毁的对象。

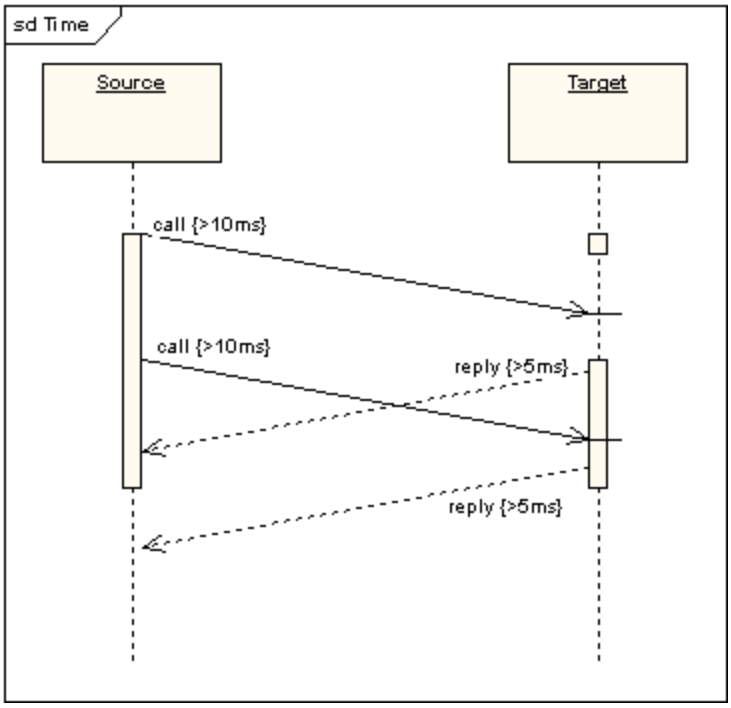


Duration and Time Constraints 期限和时间限制

By default, a message is shown as a horizontal line. Since the lifeline represents the passage of time down the screen, when modelling a real-time system, or even a time-

bound business process, it can be important to consider the length of time it takes to perform actions. By setting a duration constraint for a message, the message will be shown as a sloping line.

默认情况下，一条信息显示为一条水平线。由于生命线代表了时间在屏幕上的流逝，当对一个实时系统，甚至一个有时间限制的业务流程进行建模时，考虑执行动作所需的时间长度可能是很重要的。通过为一条消息设置持续时间约束，该消息将被显示为一条倾斜的线。



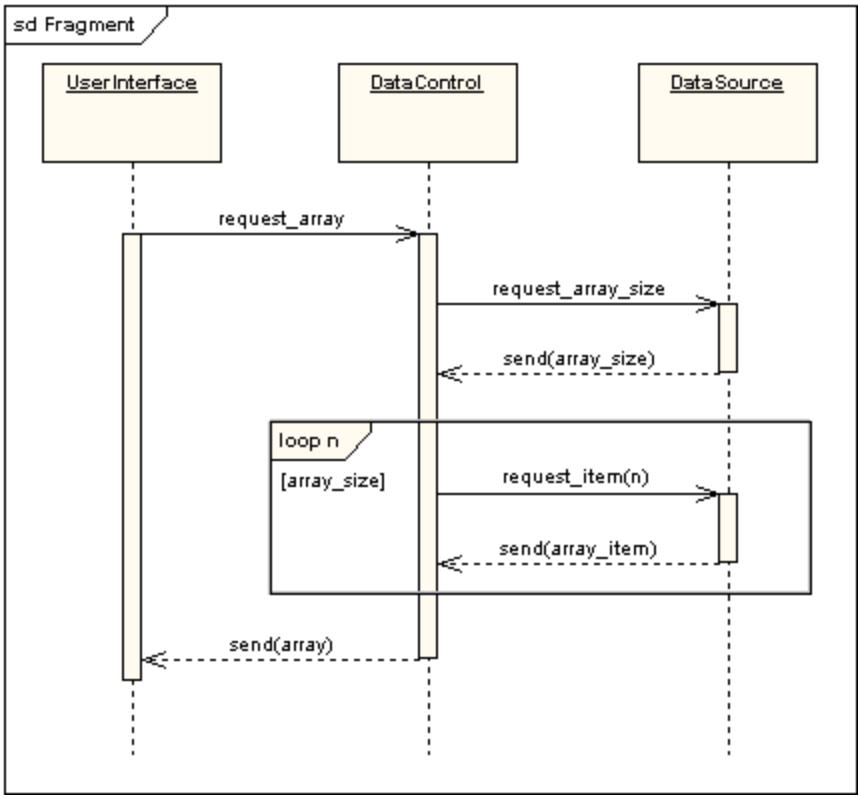
Combined Fragments

It was stated earlier that sequence diagrams are not intended for showing complex procedural logic. While this is the case, there are a number of mechanisms that do allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments. A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances. The fragments available are:

- Alternative fragment (denoted “alt”) models if…then…else constructs.
- Option fragment (denoted “opt”) models switch constructs.

- Break fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.
- Parallel fragment (denoted “par”) models concurrent processing.
- Weak sequencing fragment (denoted “seq”) encloses a number of sequences for which all the messages must be processed in a preceding segment before the following segment can start, but which does not impose any sequencing within a segment on messages that don’t share a lifeline.
- Strict sequencing fragment (denoted “strict”) encloses a series of messages which must be processed in the given order.
- Negative fragment (denoted “neg”) encloses an invalid series of messages.
- Critical fragment encloses a critical section.
- Ignore fragment declares a message or message to be of no interest if it appears in the current context.
- Consider fragment is in effect the opposite of the ignore fragment: any message not included in the consider fragment should be ignored.
- Assertion fragment (denoted “assert”) designates that any sequence not shown as an operand of the assertion is invalid.
- Loop fragment encloses a series of messages which are repeated.

The following diagram shows a loop fragment.

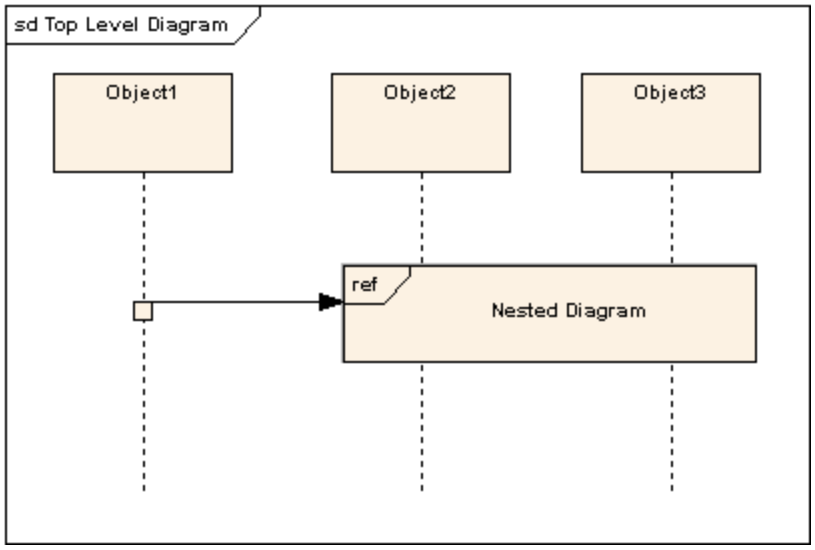


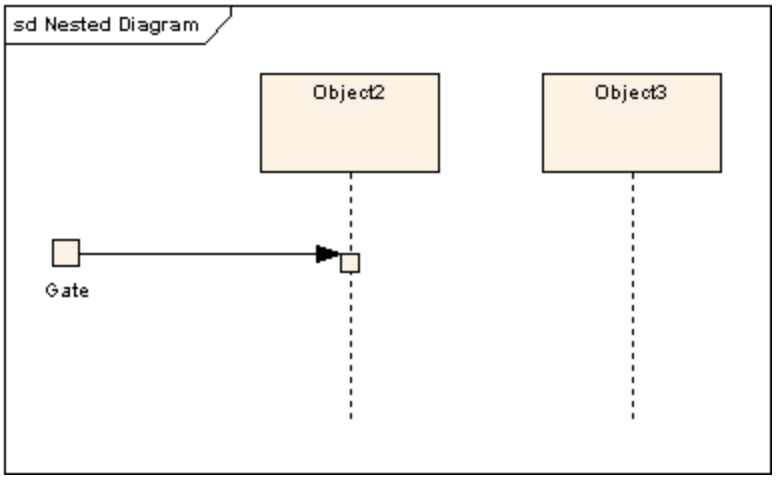
There is also an interaction occurrence, which is similar to a combined fragment. An interaction occurrence is a reference to another diagram which has the word "ref" in the top left corner of the frame, and has the name of the referenced diagram shown in the middle of the frame.

Gate

A gate is a connection point for connecting a message inside a fragment with a message outside a fragment. EA shows a gate as a small square on a fragment frame. Diagram gates act as off-page connectors for sequence diagrams, representing the source of incoming messages or the target of outgoing messages. The following two diagrams show how they might be used in practice. Note that the gate on the top level diagram is the point at which the message arrowhead touches the reference fragment - there is no need to render it as a box shape.

门是一个连接点，用于连接片段内的信息和片段外的信息。EA将门显示为片段框架上的一个小方块。图门作为序列图的页外连接器，代表了传入消息的源头或传出消息的目标。下面两张图显示了它们在实践中的使用情况。请注意，顶层图上的门是消息箭头与参考片段的接触点--没有必要把它渲染成一个盒子形状。

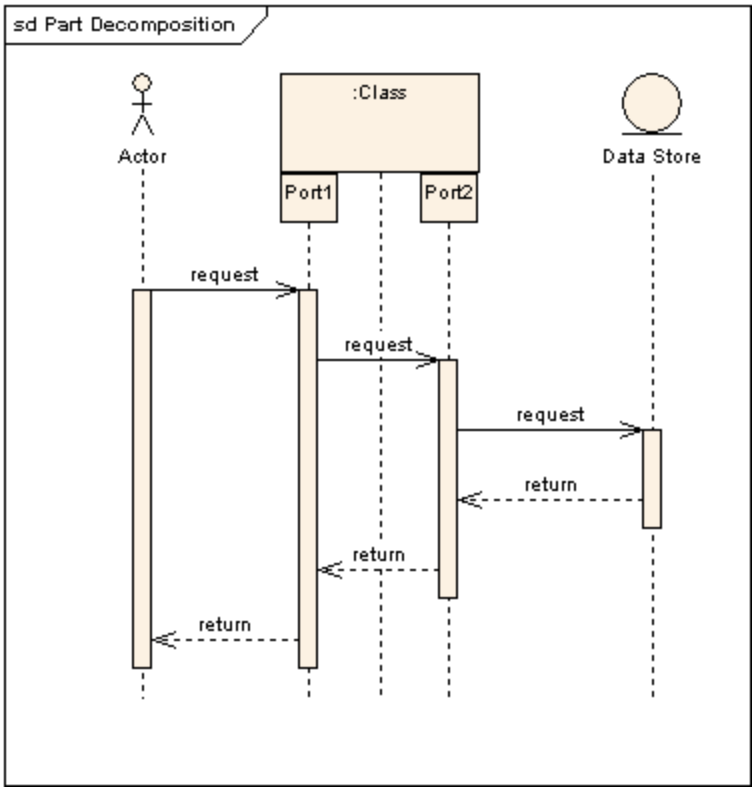




Part Decomposition 部件分解

An object can have more than one lifeline coming from it. This allows for inter- and intra-object messages to be displayed on the same diagram.

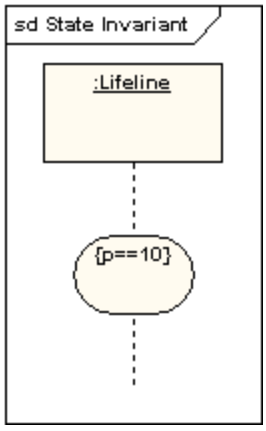
一个对象可以有一个以上的生命线来自它。这使得对象之间和对象内部的信息可以显示在同一个图上。



State Invariant / Continuations 状态不变性/延续性

A state invariant is a constraint placed on a lifeline that must be true at run-time. It is shown as a rectangle with semi-circular ends.

状态不变量是放在生命线上的约束，在运行时必须为真。它显示为一个两端为半圆形的矩形。



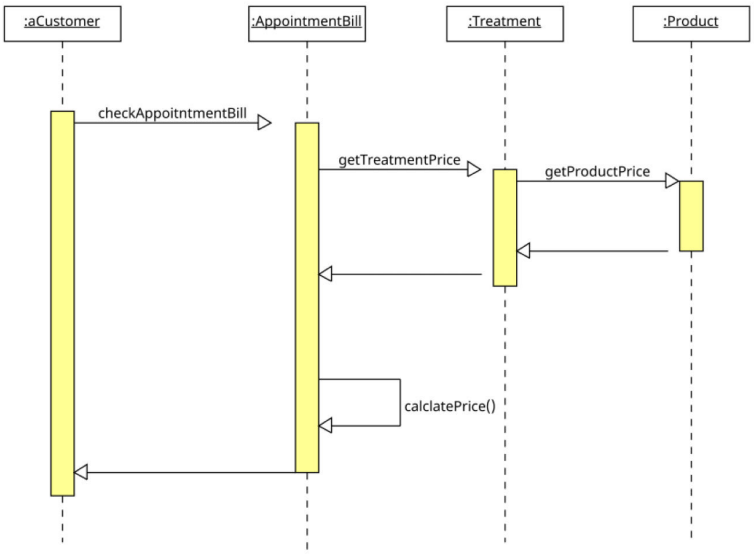
A continuation has the same notation as a state invariant, but is used in combined fragments and can stretch across more than one lifeline.

延续的符号与状态不变的符号相同，但在组合片段中使用，可以跨越一条以上的生命线。

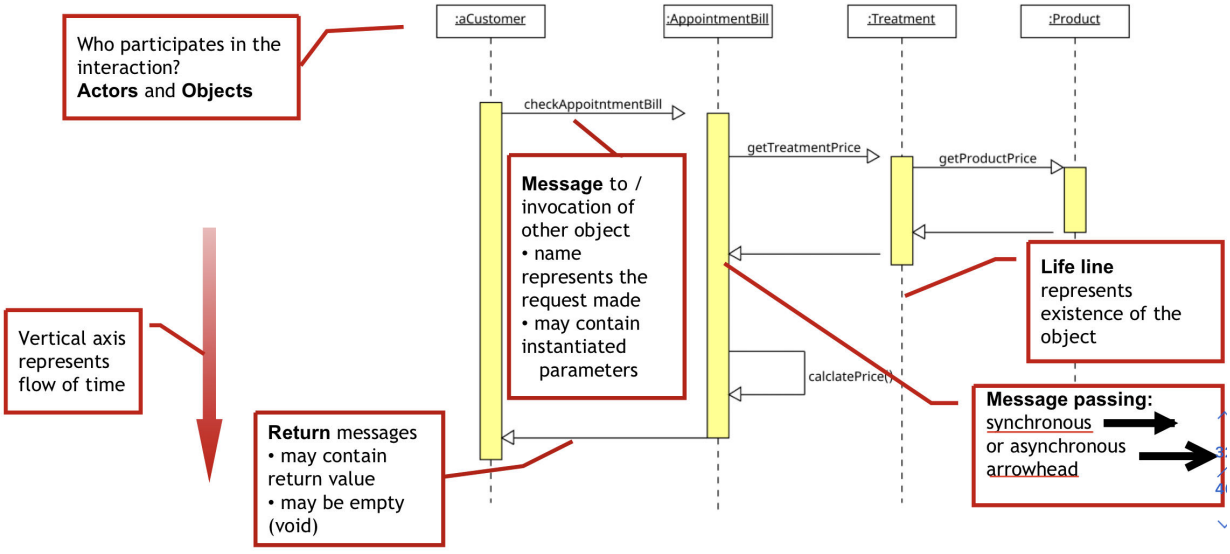
Class note

Sequence Diagrams: Basic Elements

- A set of participants arranged in time sequence
- Good for real-time specifications and complex scenarios



Sequence Diagrams: Basic Elements



Method for analysis sequence diagrams:

1. Identifying objects and actors: Look for the objects (instances of classes) and actors (external entities) involved in the interaction. Identify their roles and responsibilities in the sequence.
2. Identifying messages: Identify the messages sent between the objects and actors, including the type of message (synchronous or asynchronous), the timing of the message, and the content of the message.
3. Identifying lifelines: Identify the lifelines (vertical dotted lines) for each object and actor involved in the sequence, indicating the duration of their involvement in the interaction.
4. Analyzing timing and ordering: Analyze the timing and ordering of the messages to determine the sequence of events and the dependencies between them.
5. Identifying exceptions and error handling: Look for exceptions and error handling mechanisms in the sequence diagram, including the types of exceptions that can occur and how they are handled.
6. Identifying concurrency: Look for concurrency in the sequence diagram, including parallel and concurrent interactions between objects and actors.
7. Validating requirements: Validate the requirements of the system against the sequence diagram to ensure that all requirements are being met and that the interactions between objects and actors are consistent with the requirements.

Method for Analysis Sequence Diagrams

- for each scenario (high-level sequence diagram)
  - decompose to show what happens to objects inside the system
    - objects and messages
  - Which tasks (operation) does the object perform?
    - label of message arrow
  - Who is to trigger the next step?
    - return message or pass on control flow

Sequence diagrams can model simple sequential flow, branching, iteration, recursion and concurrency.

They may specify different scenarios/ runs

-Primary

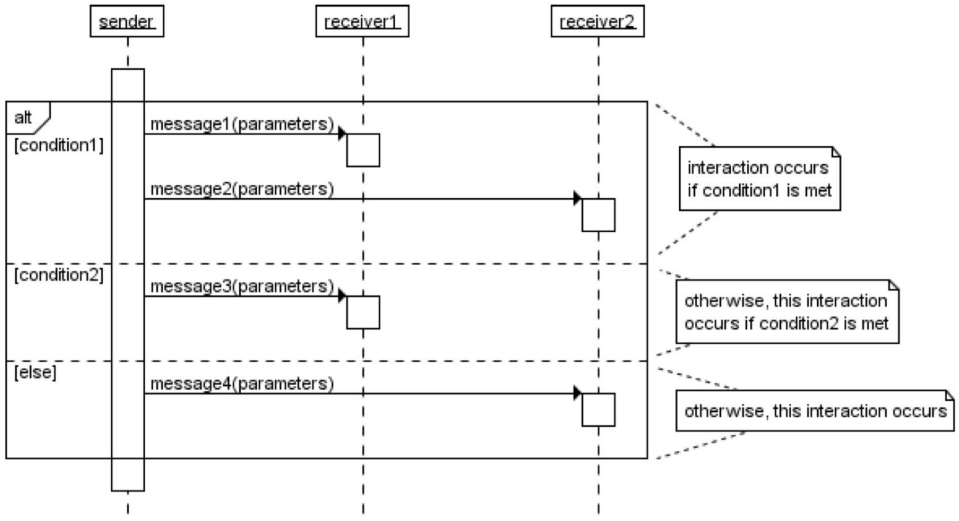
-Variant

-Exceptions

Interaction frames: alt

An interaction frame is a way to show alternative scenarios or branching paths in a sequence diagram or collaboration diagram. An alt frame is used to show a decision point where the behavior of the system may take different paths based on certain conditions.

Interaction frames: alt



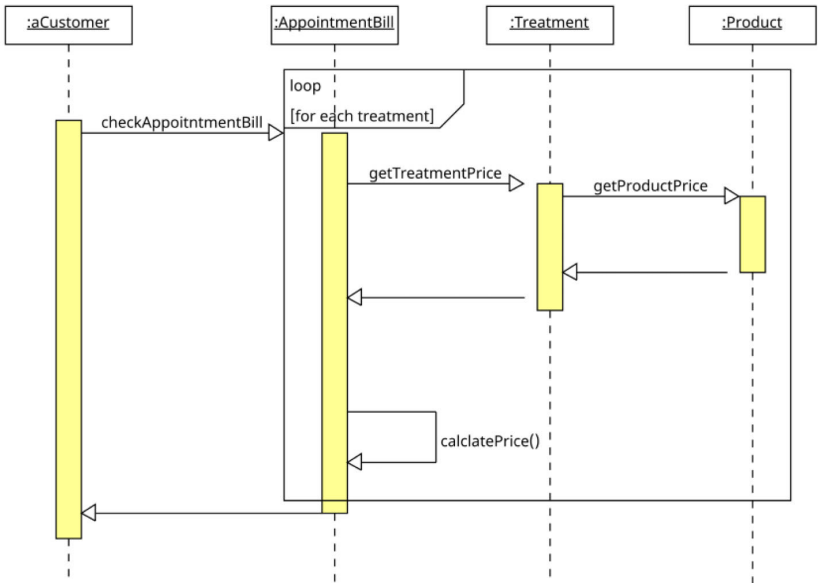
http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/

Interaction frame : loop

An interaction frame with the loop operator in a sequence diagram represents a loop that allows for repeated execution of a sequence of messages. The loop operator is represented by a rectangle with a small rectangle inside and the keyword "loop" written inside the outer rectangle. The inner rectangle contains the condition that is evaluated to determine whether the loop should continue or terminate. The messages within the loop are repeated until the condition is false, at which point the loop terminates and control moves to the next message outside the loop.

The loop operator is useful for representing scenarios where a set of messages must be repeated a fixed number of times or until a particular condition is met. It can also be used to represent scenarios where a set of messages should be repeated until a certain event occurs.

Interaction frames: loop



Comparison: Communication and sequence diagrams

Communication diagrams and sequence diagrams are both used to model the dynamic behavior of a system.

Similarties:

- Semantically equivalent (can covent one diagram to the other without losing any information)
- Model the dynamic aspects of a system
- Model a use-case scenario;

Differences

- Notation: In communication diagrams, objects are represented as boxes with the object name inside, while in sequence diagrams, objects are represented as vertical lines. Messages are represented with arrows in both diagrams, but the direction of the arrows is reversed in communication diagrams compared to sequence diagrams.

- Usage: Communication diagrams are better suited for showing the relationships between objects, especially in complex systems where there are many interactions between objects. Sequence diagrams are better suited for showing the time ordering of messages and the flow of control between objects.

Sequence and Communication Diagram Differences

Sequence diagrams	Communication diagrams
Show the explicit sequence of messages Show execution occurrence Better for visualizing overall flow Better for real-time specifications and for complex scenarios	Show relationships in addition to interactions Better for visualizing patterns of communication Better for visualizing all of the effects on a given object Easier to use for brainstorming sessions

Review

- What does a class diagram represent?
- Define association, aggregation, and generalization.
- How do you find associations?
- What information does multiplicity provide?

- What is the main purpose of a SD?
- What are the main concepts in a SD?
- What are the communication diagrams?
- What is the difference between SD and communication diagrams?

Review:

1️⃣ *what does a class diagram represent ?*

A class diagram represents the static structure of a system by showing the classes, their attributes, methods, and relationships among objects. It helps in visualizing the overall architecture of the system and provides a foundation for designing and implementing the system. The class diagram is a part of the Unified Modeling Language (UML) and is widely used in software engineering and other fields for modeling systems.

2️⃣ *Define association, aggregation and generalization*

1. Association: It represents the relationship between two classes that are not in a hierarchy but are still related to each other. It can be one-to-one, one-to-many, or many-to-many. An association can have an association class that represents additional information about the relationship.
2. Aggregation: It represents a type of association where one class contains another class as a part of its structure. The whole-part relationship is used to describe aggregation, and it is depicted as a diamond shape on the class diagram.
3. Generalization: It represents the inheritance relationship between classes, where one class (subclass or child class) inherits the attributes and behaviors of another class (superclass or parent class). The subclass can add its attributes and behaviors to the inherited ones. It is depicted as an arrow pointing to the superclass.

3️⃣*How do you find associations?*

Associations are found by examining the relationships between classes in a system. They represent the connections and interactions between objects in the system.

To find associations, we can look for verbs or phrases that describe how different classes interact with each other.

For example, if we have a class called "Customer" and a class called "Order", we might identify an association between the two classes based on the fact that a customer can place an order. The association would be represented by a line connecting the two

classes in a class diagram. We can also use multiplicity to indicate how many objects of each class are involved in the association.

4️⃣ *What information does multiplicity provide?*

Multiplicity provides information about how many instances of one class are associated with how many instances of another class. It indicates the minimum and maximum number of objects of a class that can be associated with an object of another class.

For example, in a class diagram representing a library system, the association between the classes "Book" and "Author" could have a multiplicity of "1 to many" which means that each book can have one or many authors, but each author can be associated with only one book. Another example is an association between the classes "Employee" and "Department" with a multiplicity of "0 to many" which means that an employee may or may not be associated with any department, while a department may have many employees.

5️⃣ *What is the main purpose of a SD?*

The main purpose of a sequence diagram (SD) is to illustrate the interactions between objects or components of a system in a time-ordered/ temporal/ chronological sequence. It shows the flow of messages or calls between the objects or components over time, and can help to identify the order of events and potential issues with the system's behavior. SDs are typically used in the analysis and design phases of software development to help developers understand and refine the system's behavior. They can also be used to document and communicate the system's behavior to stakeholders.

6️⃣ *What are the main concepts in a SD?*

1. Object/Actor: The entities involved in the interactions represented by the sequence diagram. Each object or actor is represented by a lifeline, which is a vertical line that extends down the diagram.
2. Message: The communication between the objects or actors. A message is represented by an arrow that points from the sender to the receiver. The arrow can be labeled with the name of the message and any parameters that are passed.
3. Activation: The period of time during which an object or actor is executing a particular message. An activation is represented by a box that appears on the lifeline of the object or actor, with the message name inside the box.

4. Lifeline: The vertical line that represents an object or actor in the sequence diagram.
5. Synchronous Message: A message that requires the sender to wait for a response from the receiver before continuing. It is represented by a solid arrow.
6. Asynchronous Message: A message that does not require the sender to wait for a response from the receiver before continuing. It is represented by a dashed arrow.

7️⃣ *What are the communication diagrams?*

Communication diagrams are a type of UML (Unified Modeling Language) diagram used for modeling object interactions in a system.

They show the interactions between objects or actors in a system in a visual way, using a combination of sequence diagrams and collaboration diagrams. Communication diagrams focus on the flow of messages between objects rather than the sequence of messages, making them useful for capturing the structural relationships between objects as well as the behavior of the system. They can be used to model complex systems with multiple objects and can help identify potential issues in the system design.

8️⃣ *What is the difference between SD and communication diagrams?*

Sequence diagrams (SD) and communication diagrams are both used in object-oriented software design to model the interactions between objects or actors in a system. The main difference between them is the way they represent the interactions.

In a sequence diagram, the interactions between objects are shown in a time-ordered sequence, with the vertical axis representing time and the horizontal axis representing the objects involved. Each object is represented by a lifeline, and the messages between the objects are represented by arrows.

In a communication diagram, the interactions between objects are represented by messages between the objects, with no explicit representation of time. The objects are shown as rectangles, and the messages between the objects are represented by arrows.

So, the main difference is that SDs focus on the time-ordered sequence of interactions, while communication diagrams focus on the relationships between objects and the

messages they exchange, without explicitly showing the sequence of events.

Quizes:

1. **What heading of diagram do Sequence diagrams fall under?**

Answer: Interaction diagram

2. **What is a sequence diagram's purpose?**

Answer: To show objects interactions arranged in time.

3. **A_____ is a connection path between two objects; it indicates some form of navigation and visibility between the objects is possible.**

Answer: link

1.	The scope of problem analysis requires the definition of Please select the best answer.	
	○A.	The operations and attributes for each resource
	○B.	The interactions between actors and use cases
	○C.	The purpose and interface of each resource
	○D.	The events and objects of the problem domain

2.	The order of events in a sequence diagram is determined Please select the best answer.	
	○A.	By the sequence numbers of the events
	○B.	By the relative position of the events on the timeline
	○C.	By following the order of the events in the use case description
	○D.	By placing numbers in comments along the left-hand margin of the diagram

3.	How much logic should a sequence diagram include? Please select the best answer.	
	○A.	One use case
	○B.	One user interaction
	○C.	One scenario
	○D.	One actor's use of the system

4.	The first of the two steps in building the sequence diagram is to Please select the best answer.	
	○A.	Define the purpose and interfaces of the resource objects
	○B.	Define how the actor uses the system resources
	○C.	Define how the resource objects interact with the system
	○D.	Define how the actor interacts with the system in a use case

5.	The second of the two steps in building the sequence diagram is to Please select the best answer.	
	○A.	Define how the system uses the problem domain resources
	○B.	Define how the actor uses the system resources
	○C.	Define how the resource objects interact with the system
	○D.	Define how the actor interacts with the system in a use case

Correct answers:

1. C

2. B

3. C

4. D

5. A