



UML (Unified Modeling Language)

EECS 3311

Ilir Dema

demailir@eecs.yorku.ca

Acknowledge

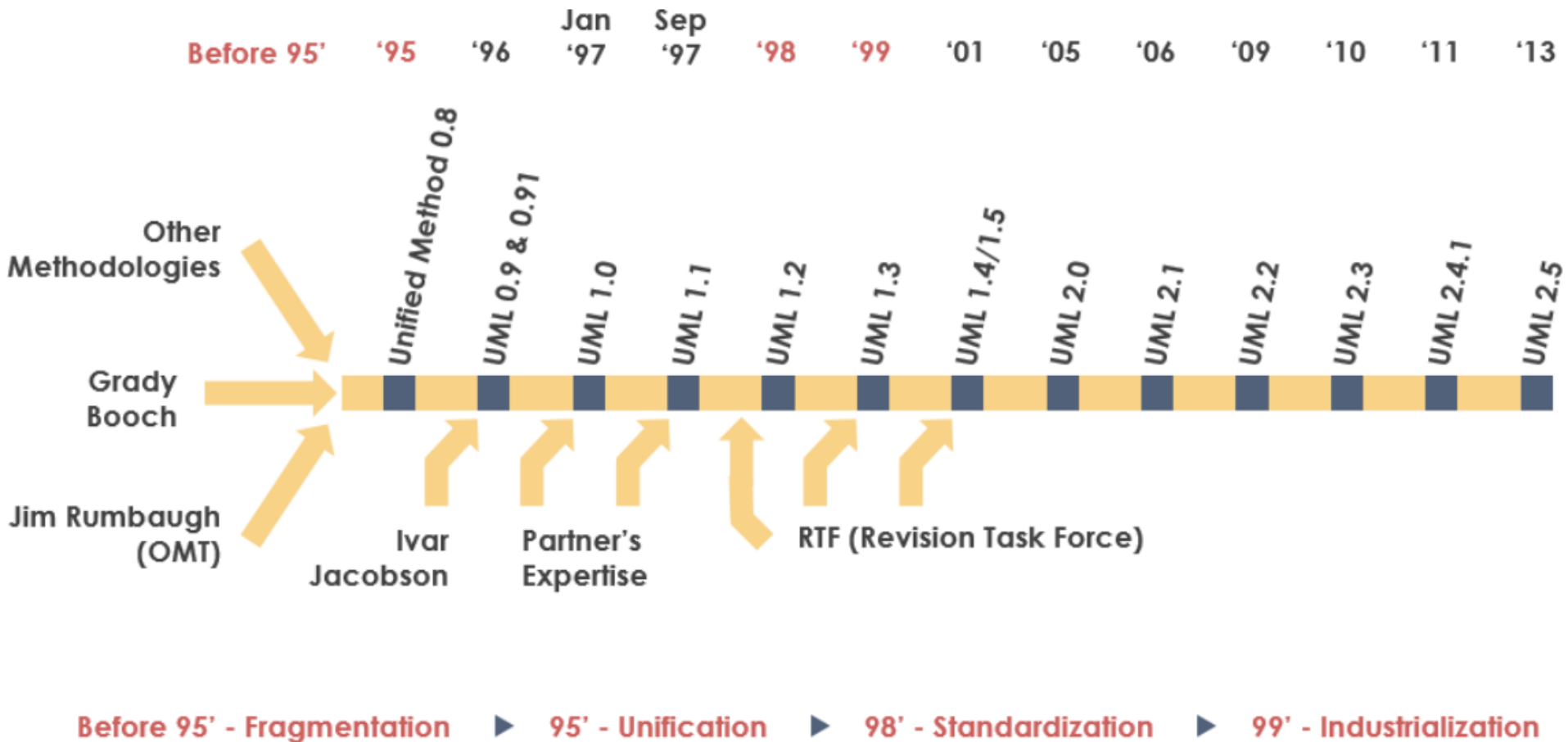
- Some of the covered materials are based on SE463 at UW, SENG321 at UVic, 4010-420 at RIT and previous EECS3311 offerings:
 - Jo Atlee, Dan Berry, Daniela Damian, Nancy Day, Mike Godfrey, Ahmed E. Hassan, Jonathan S. Ostroff, Emad Shihab, Davor Svetinovic, Jack Jiang

What is UML?

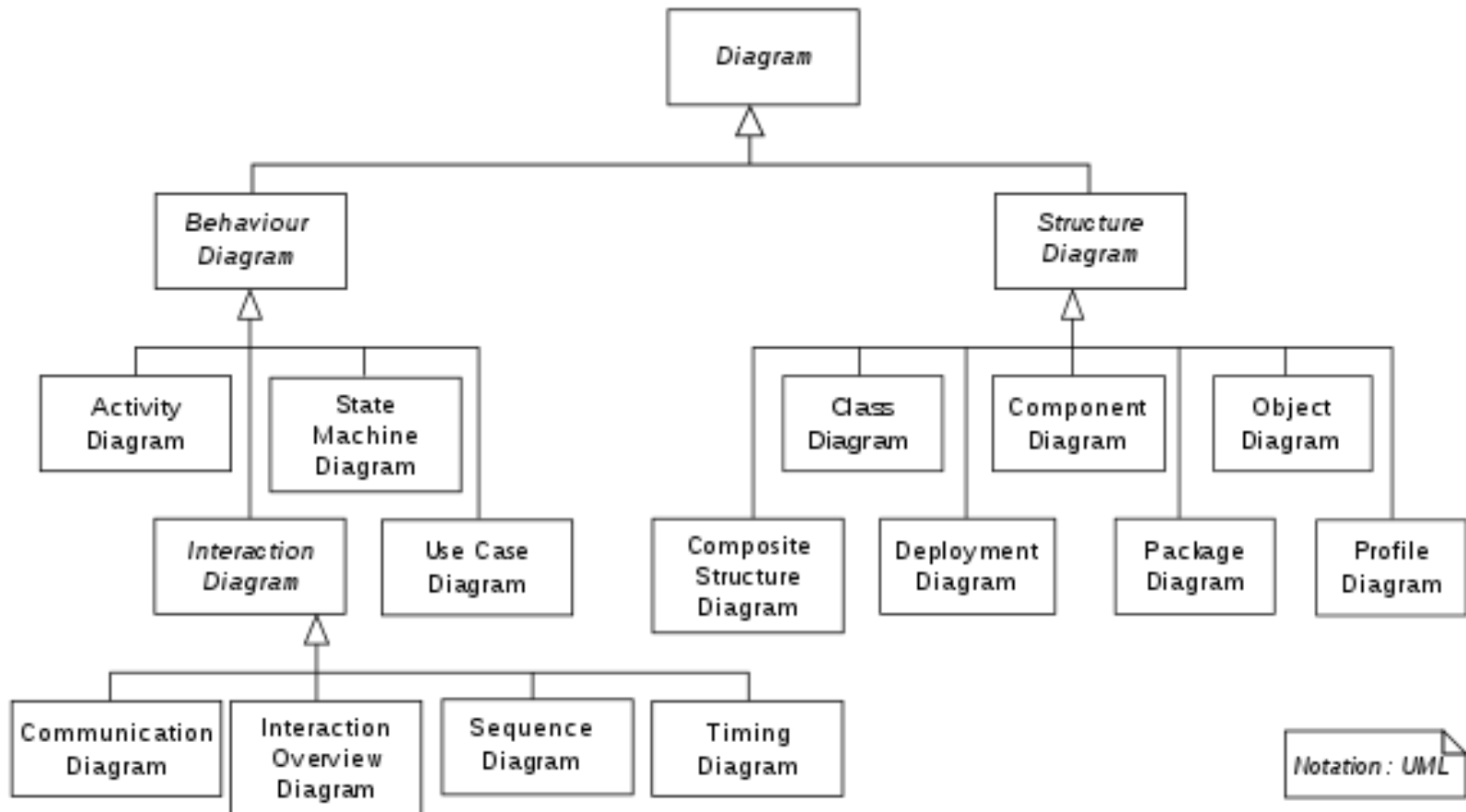
- A standardized, graphical “modeling language” for communicating software req/design.
- Allows implementation-independent specification of:
 - user/system interactions (required behaviors)
 - partitioning of responsibility (OO)
 - integration with larger or existing systems
 - data flow and dependency
 - operation orderings (algorithms)
 - concurrent operations
- UML is not “process”!
 - it doesn’t tell you how to do things, only what you should do

Motivations for UML

- Why do we want UML?
- UML allows us to share high-level design ideas
- UML gives us a standard notation so that we both express the same design the same way (....maybe....)
- UML is unambiguous (...for some things...)
- Design is much more apparent from a UML diagram than source code
 - design patterns are easy to see in UML
- UML is independent of the implementation language
 - a UML design could be realized in Java, C++, Perl, etc....



Types of UML diagrams



Types of UML diagrams

- There are different types of UML diagram, each with slightly different syntax rules:
 - use cases;
 - activity diagrams;
 - sequence diagrams;
 - **class diagrams;**
 - package diagrams;
 - state diagrams;
 - deployment diagrams;
 - ...

UML Tools

- Anything you can find to use
 - Visual Paradigm, ArgoUML, MagicDraw, Rational, Microsoft Visio, etc.
 - Different tools produce slightly different diagrams
 - Don't get stuck in the details
 - Make sure the notations in the diagrams are consistent
- <https://app.diagrams.net/>

UML basic syntax

- **Actors:** a UML actor indicates an interface (point of interaction) with the system.
 - We use actors to group and name sets of system interactions.
 - **Actors may be people, or other systems.**
 - An actor is NOT part of the system you are modeling. An actor is something external that your system has to deal with.
- **Boxes:** boxes are used variously throughout UML to indicate discrete elements, groupings and containment.

UML basic syntax

- **Arrows:** arrows indicate all manner of things, depending on which particular type of UML diagram they're in.
 - Usually, arrows indicate flow, dependency, association or generalization.
- **Cardinality:** applied to arrows, cardinalities show **relative numerical relationships between elements** in a model:
 - **1 to 1, 1 to many**, etc.

UML basic syntax

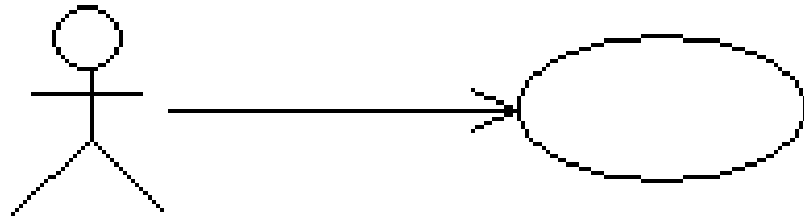
- **Constraints:** allow notation of arbitrary constraints on model elements.
 - to constrain the value of a class attribute
- **Stereotypes:** allow us to extend the semantics of UML with English.
 - A stereotype is usually a word or short phrase that describes what a diagram element does

UML diagrams: use cases

- **A use case encodes a typical user interaction with the system.** In particular, it:
 - captures some user-visible function.
 - achieves some concrete goal for the user.
- **A complete set of use cases defines the requirements for your system:**
 - everything the user can see and would like to do.
- The granularity of your use cases determines the number of them (for your system). A clear design depends on showing the right level of detail.
- A use case maps actors to functions. The actors need not be people.

Use case examples

(**High-level** use case for PowerPoint)



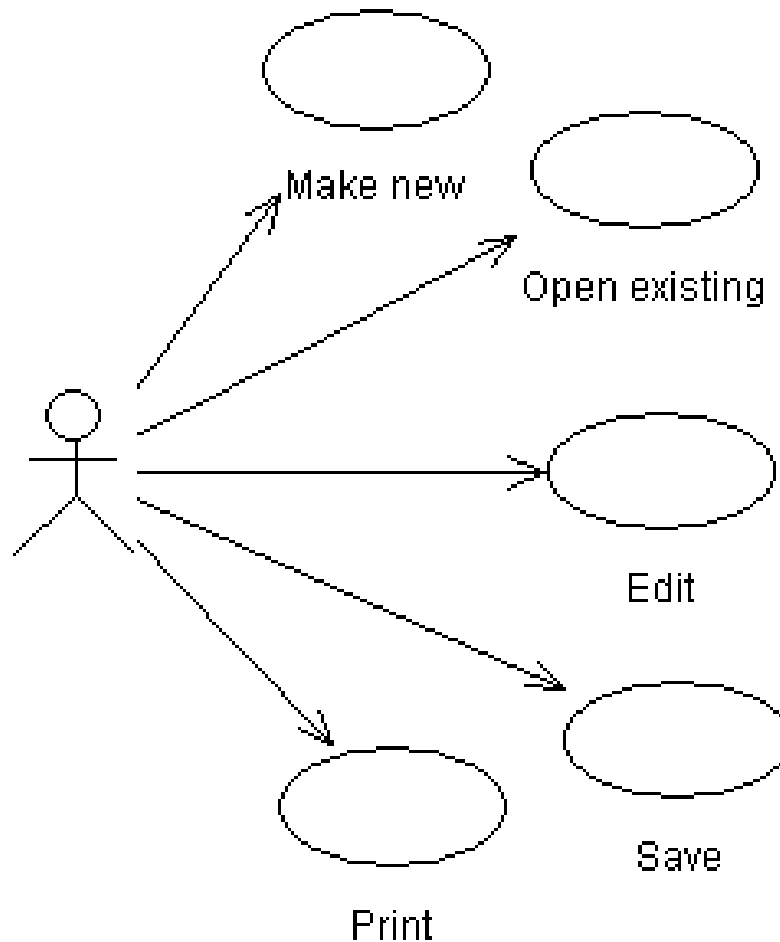
Create slide presentation

About the last example...

- Although this is a valid use case for PowerPoint, and it completely captures user interaction with powerpoint, **it's too vague to be useful.**

Use case examples 2.0

(**Finer-grained** use cases for **PowerPoint**)



About the last example...

- The last example gives a more useful view of PowerPoint (or any similar application).
- The cases are vague, but they focus your attention the key features, and would help in developing a more detailed requirements specification.
- It still doesn't give enough information to characterize PowerPoint, which could be specified with tens or hundreds of use cases (though doing so might not be very useful either).

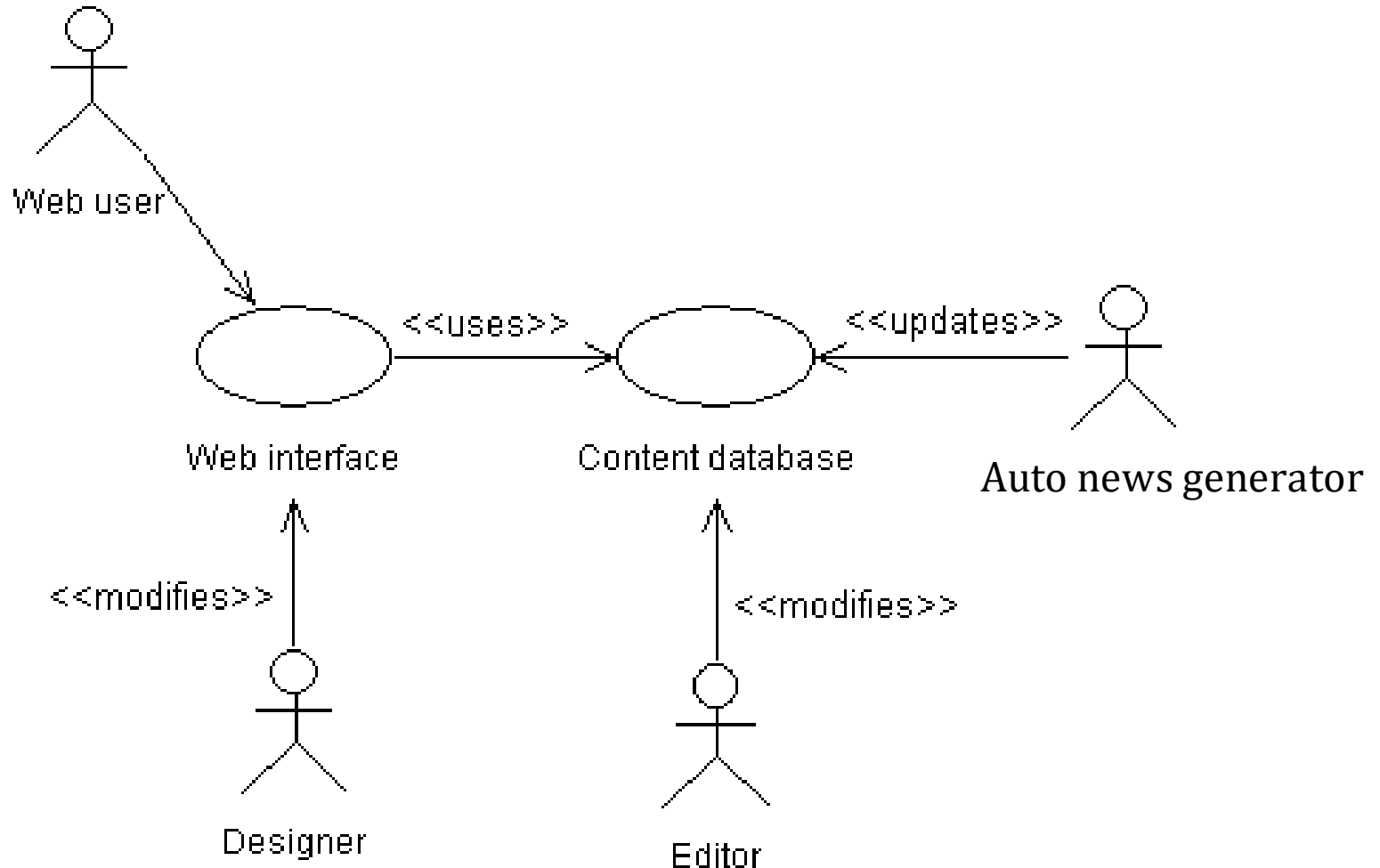
Use case examples 3.0

(**Relationships** use cases for PowerPoint.)

How?

Another use case example

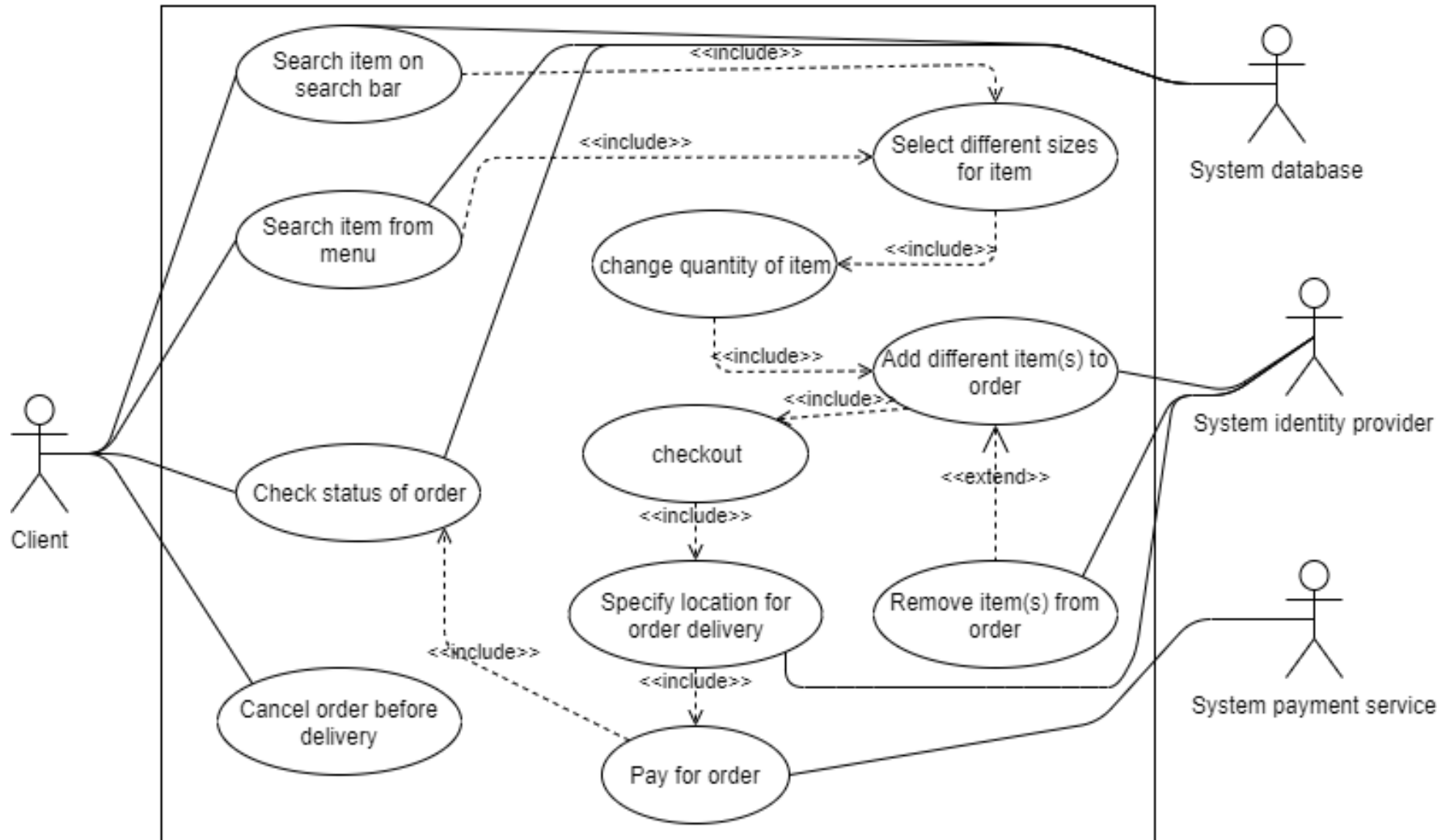
(**Relationships** in a news web site.)



About the last example...

- The last is more complicated and *realistic* use case diagram. It captures several key use cases for the system.
- Note the *multiple actors*. In particular, 'Auto news generator' is an actor, with an important interaction with the system, but is not a person (or even a computer system, necessarily).
- The notes between << >> marks are *stereotypes*: identifiers added to make the diagram more informative. Here they differentiate between different roles (ie, different meanings of an arrow in this diagram).

Freshii online system



Freshii Online System










Activity diagram

A UML activity diagram helps to **visualize a use case at a more detailed level.**

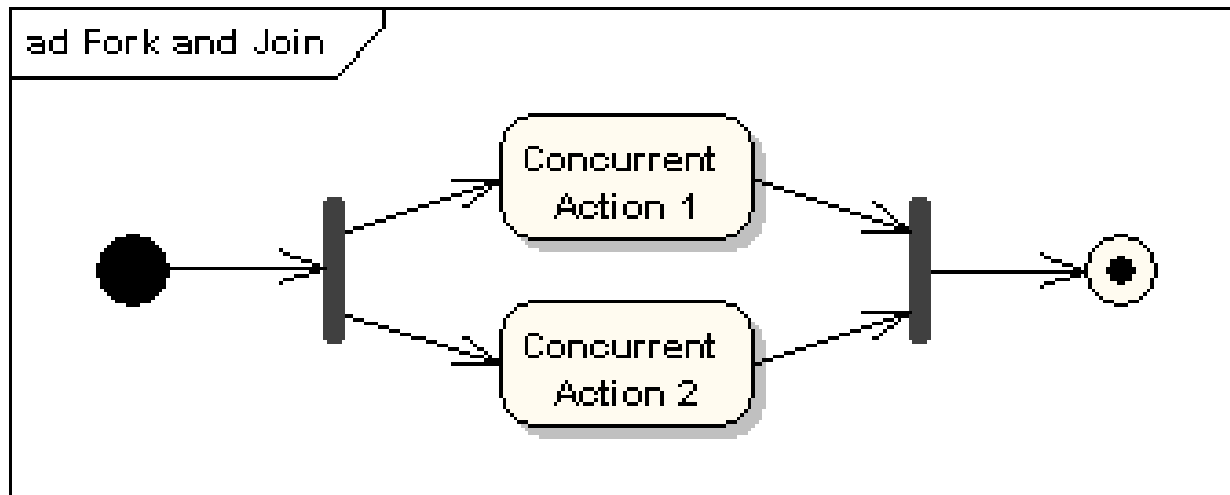
It is a **behavioral diagram** that illustrates the flow of activities through a system.

UML activity diagrams can also be used to depict a flow of events in a business process.

Activity diagram notations

	Start/ Initial Node	Used to represent the starting point or the initial state of an activity
	Activity / Action State	Used to represent the activities of the process
	Action	Used to represent the executable sub-areas of an activity
	Control Flow / Edge	Used to represent the flow of control from one action to the other
	Object Flow / Control Edge	Used to represent the path of objects moving through the activity
	Activity Final Node	Used to mark the end of all control flows within the activity
	Flow Final Node	Used to mark the end of a single control flow
	Decision Node	Used to represent a conditional branch point with one input and multiple outputs
	Merge Node	Used to represent the merging of flows. It has several inputs, but one output.

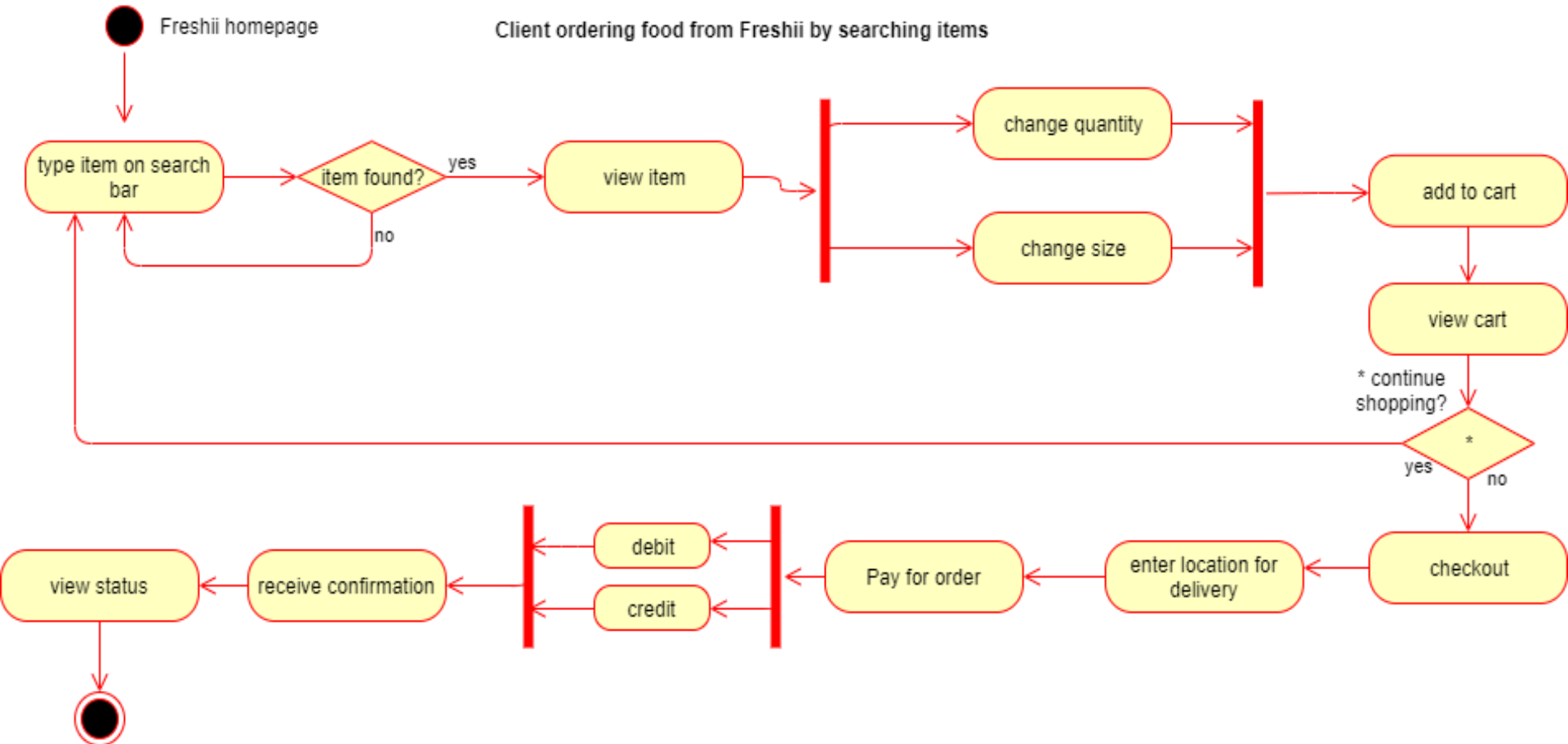
Fork and Join Nodes



How to Draw an Activity Diagram

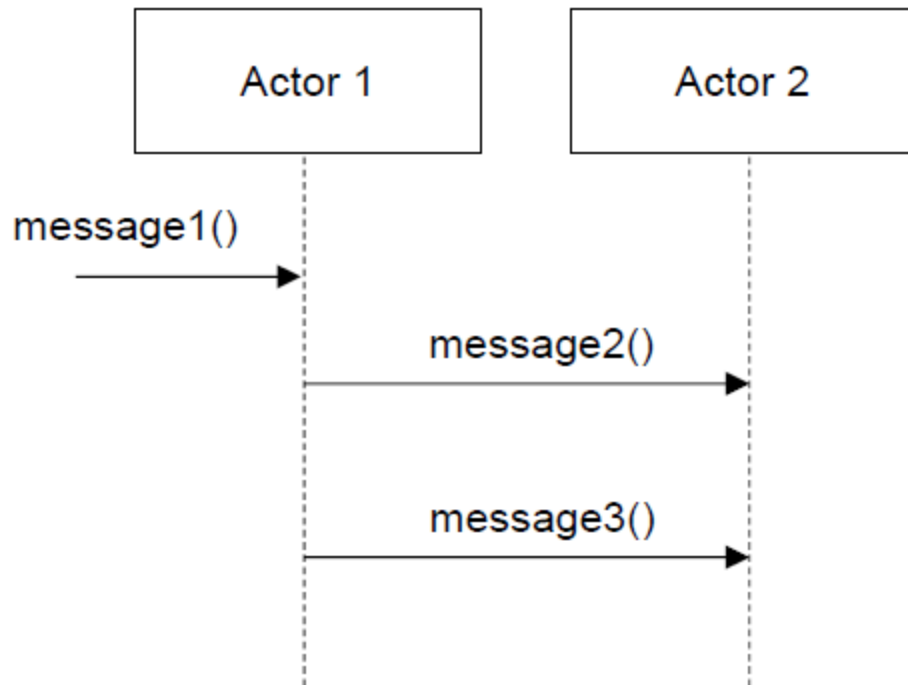
- Step 1: Figure out the action steps from the use case
- Step 2: Identify the actors who are involved
- Step 3: Find a flow among the activities

Clients order food from Freshii



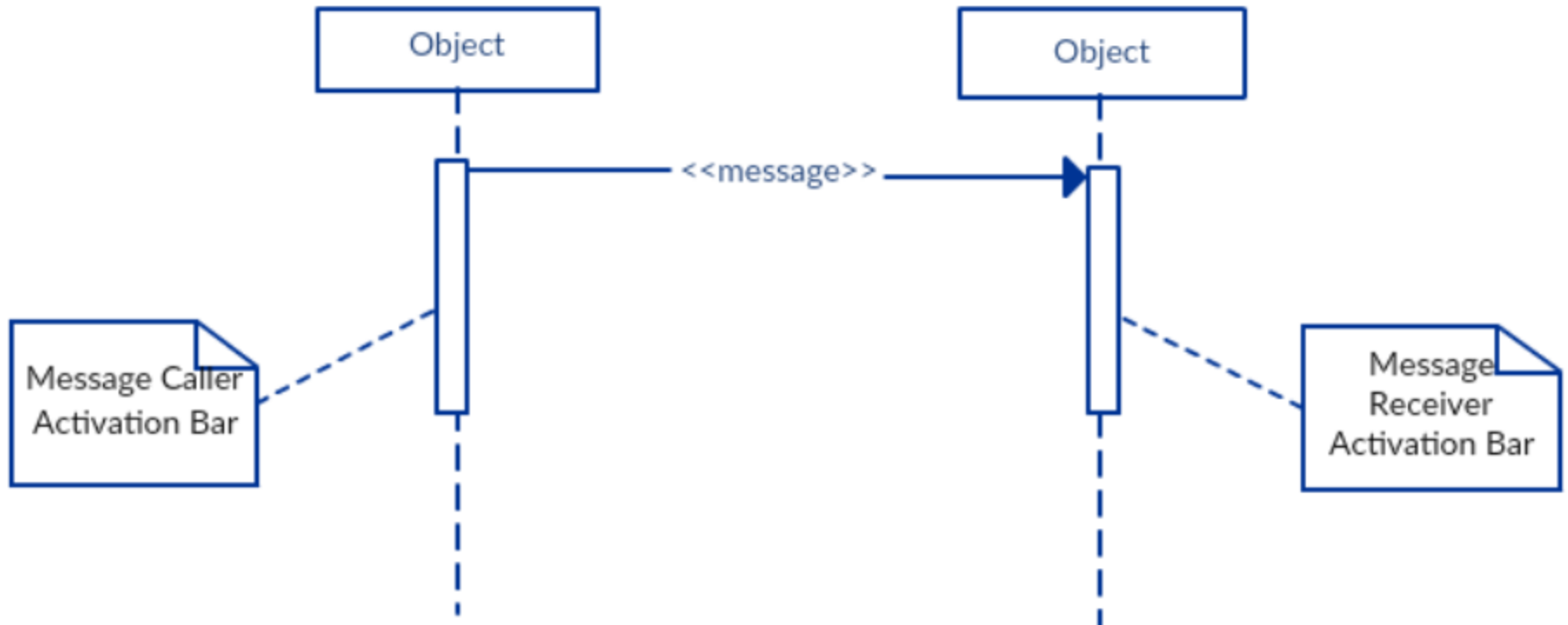
UML diagrams: sequence diagram

- Sequence diagram **describe algorithms, though usually at a high level**: the operations in a useful sequence diagram specify the “message passing” (method invocation) between objects (classes, roles) in the system.
- The notation is based on each object’s life span, with message passing marked in **time-order** between the objects. **Iteration and conditional operations may be specified.**
- May in principle be used at the same three levels as class diagrams, though the specification level will usually be most useful.

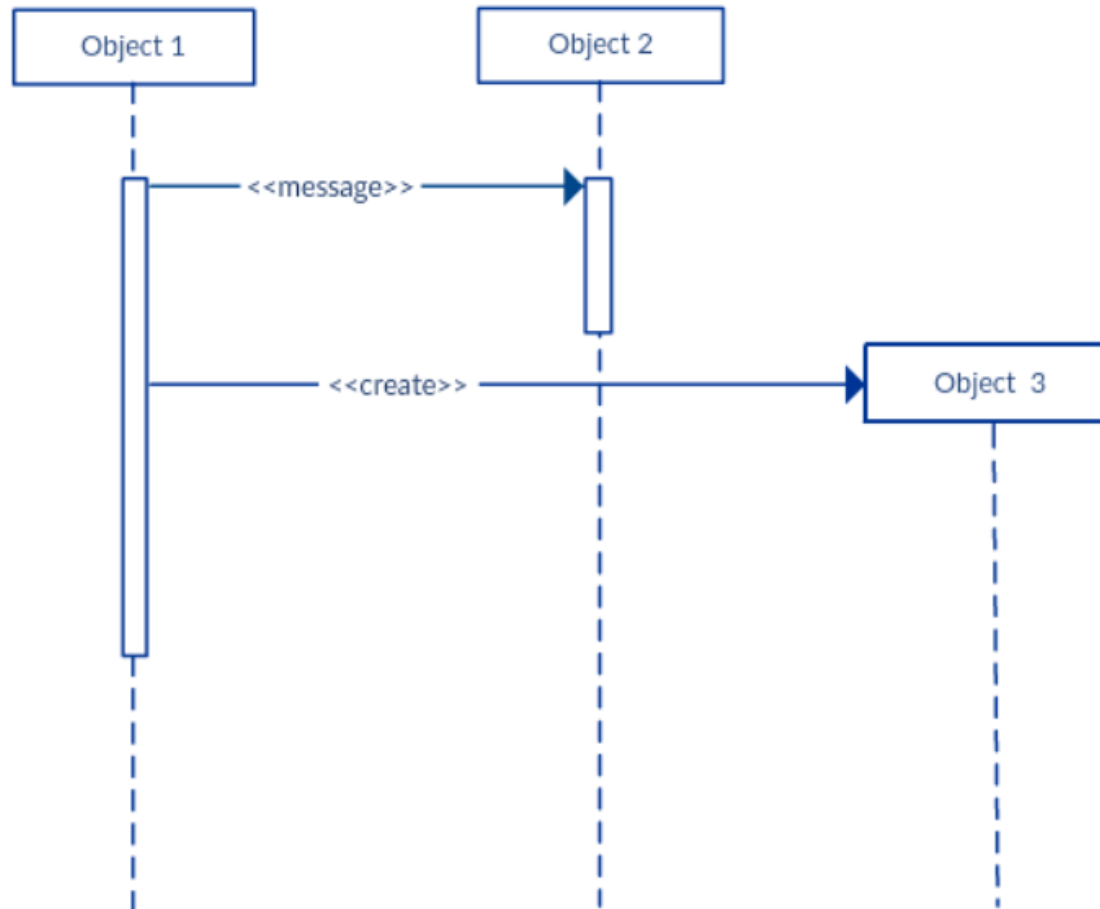


- *Sequence diagrams* illustrate interactions in a kind of fence format.
- Vertical lifelines represent the ordering of messages
- messages are directed from a sender to a receiver
- make it clear “who is doing what” (and when)

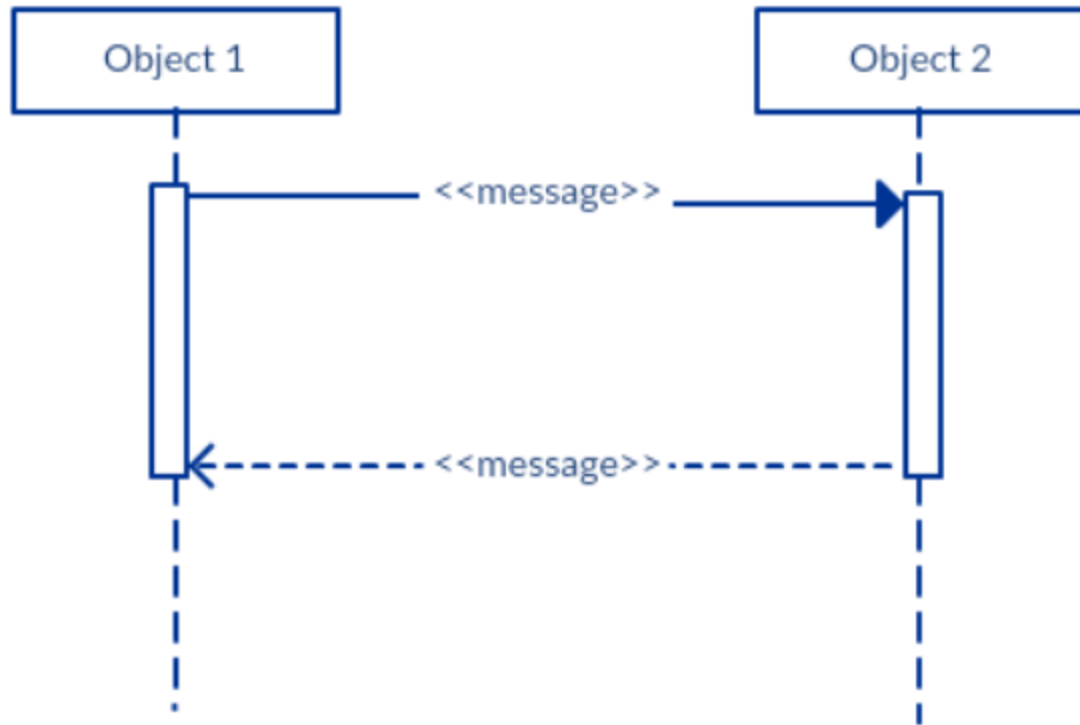
Sequence diagram example



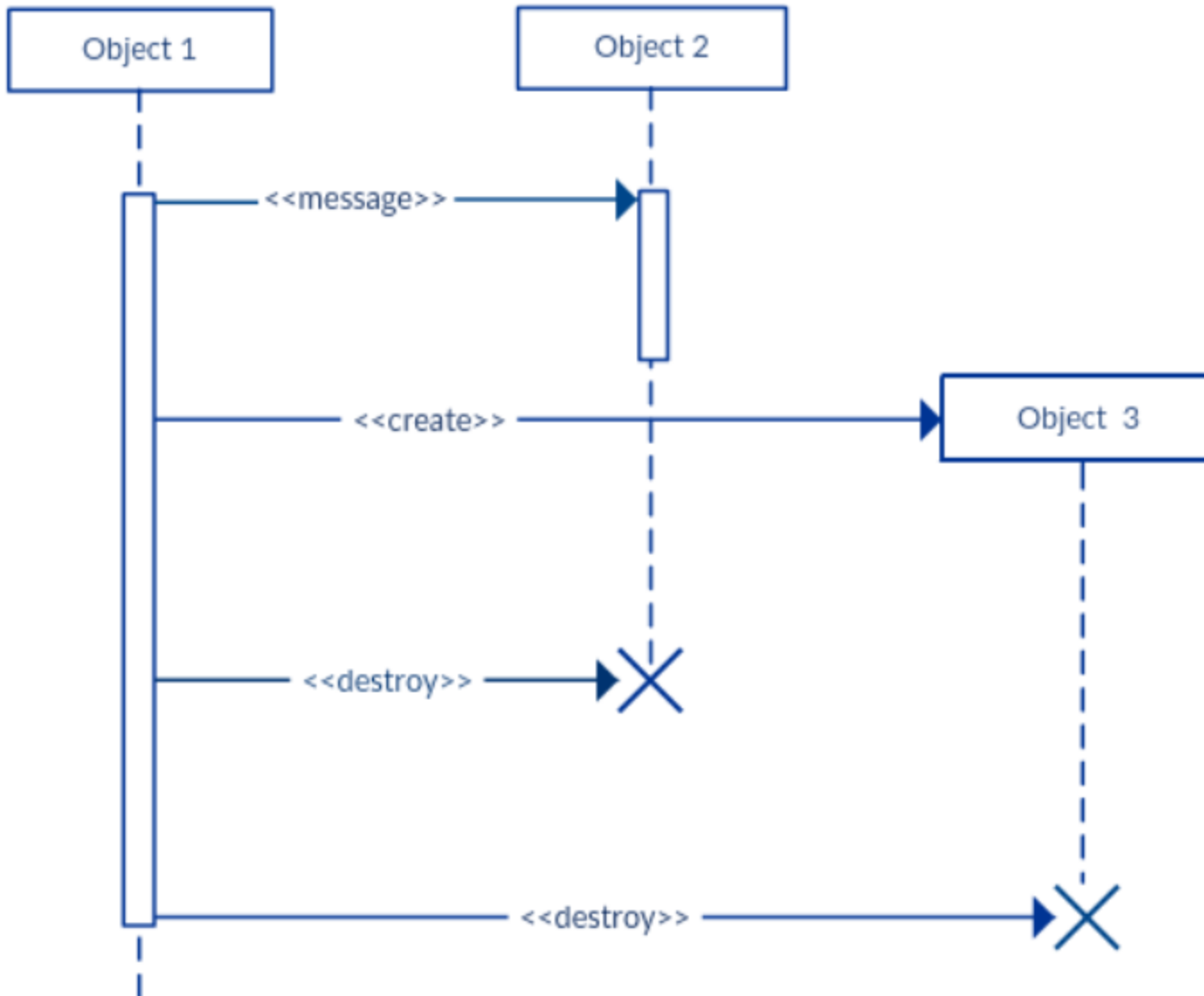
Activation bar



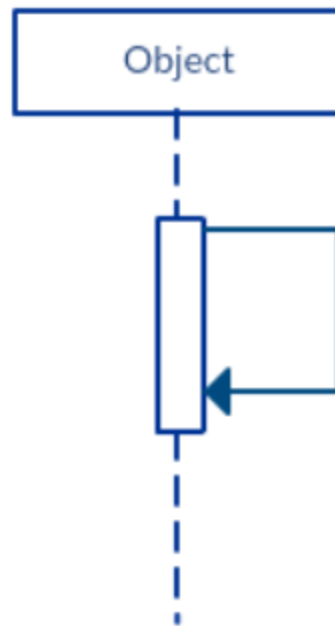
Asynchronous message



Return message

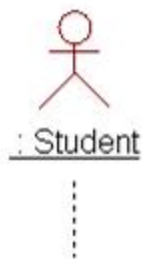


destruction/create message

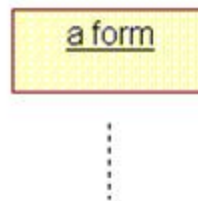


Reflexive message

Actor



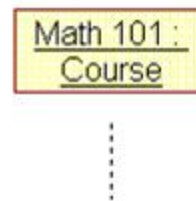
**Object
only**



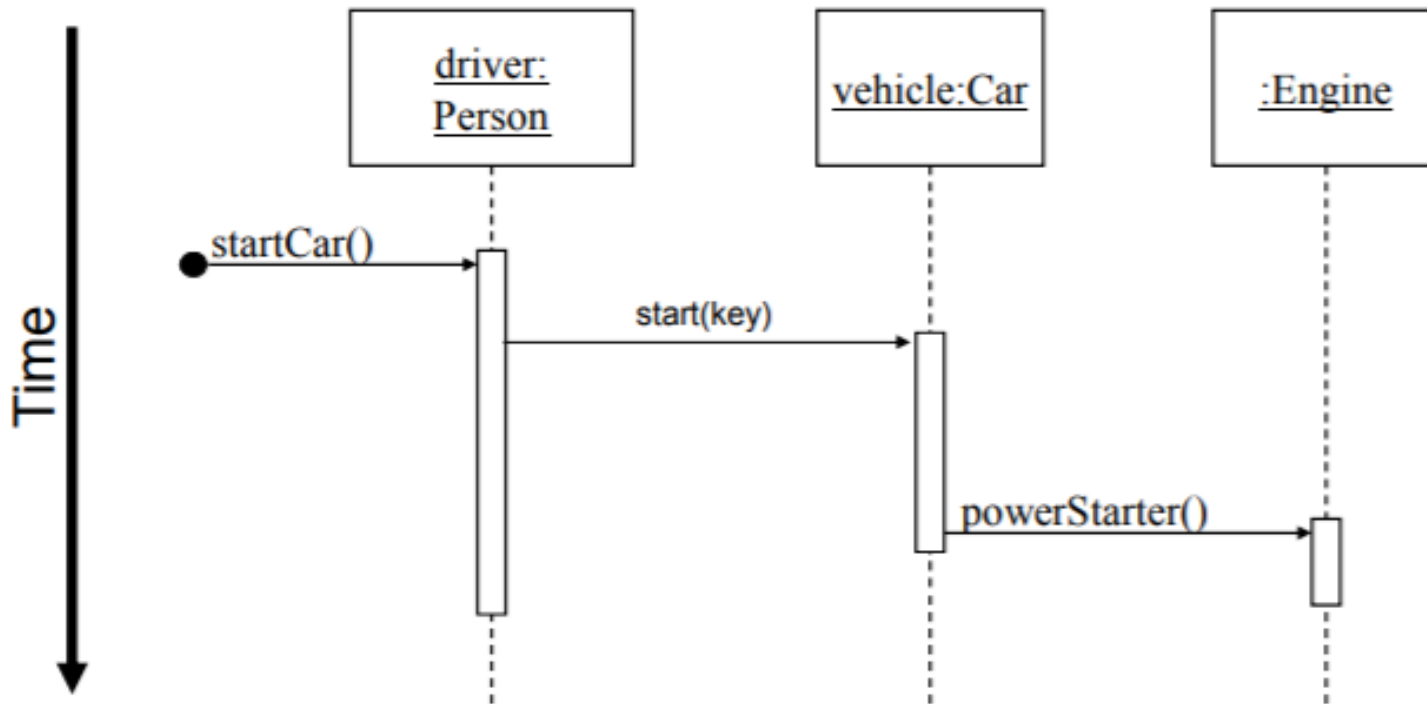
**Class
only**

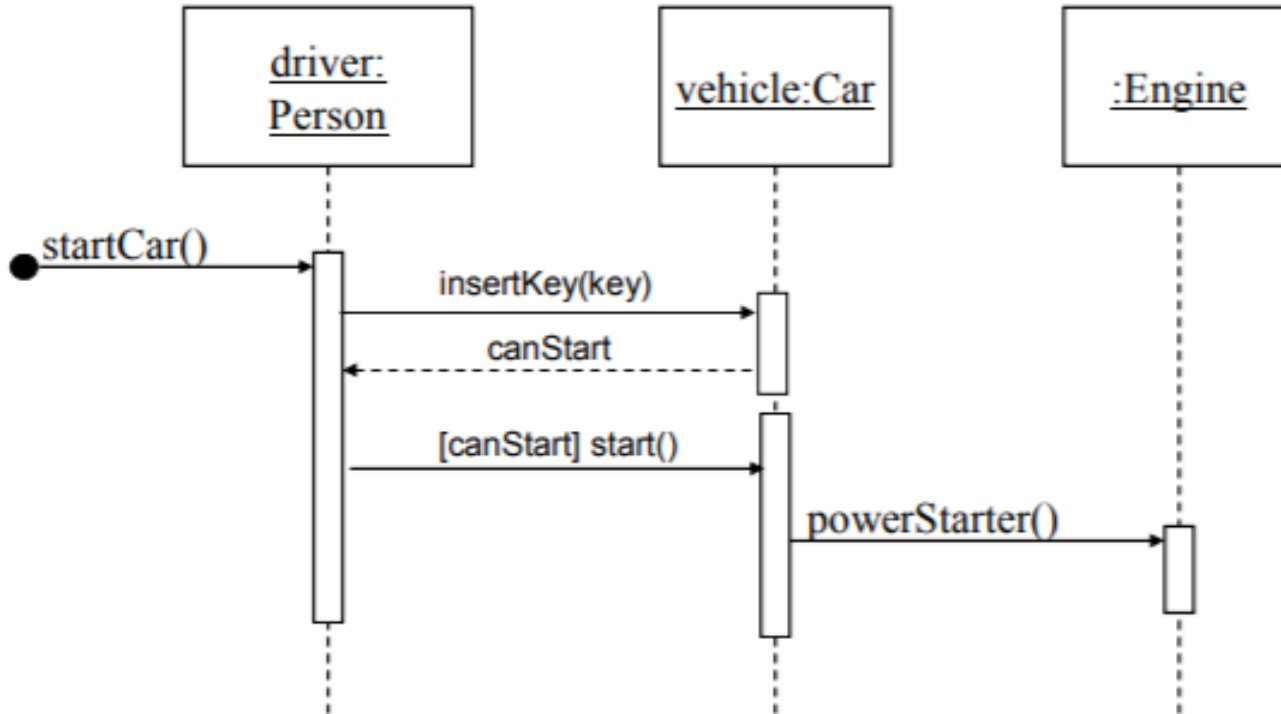


**Object and
Class**

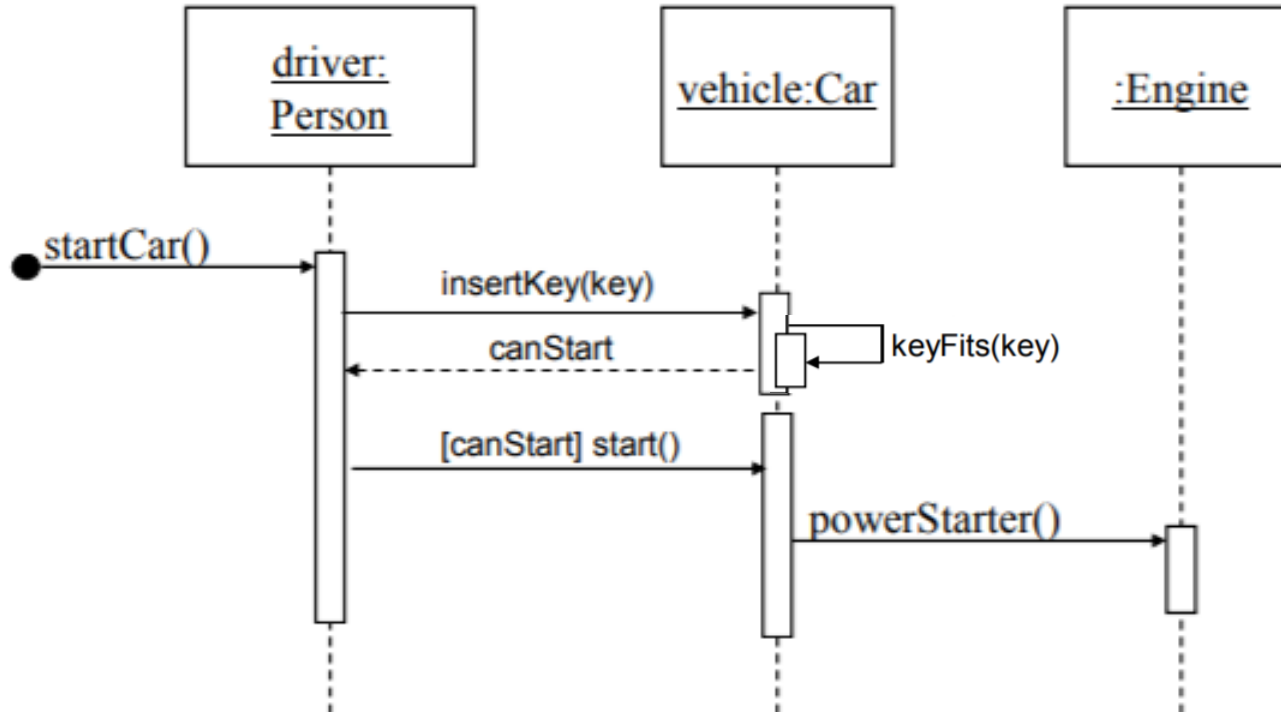


Give more details...

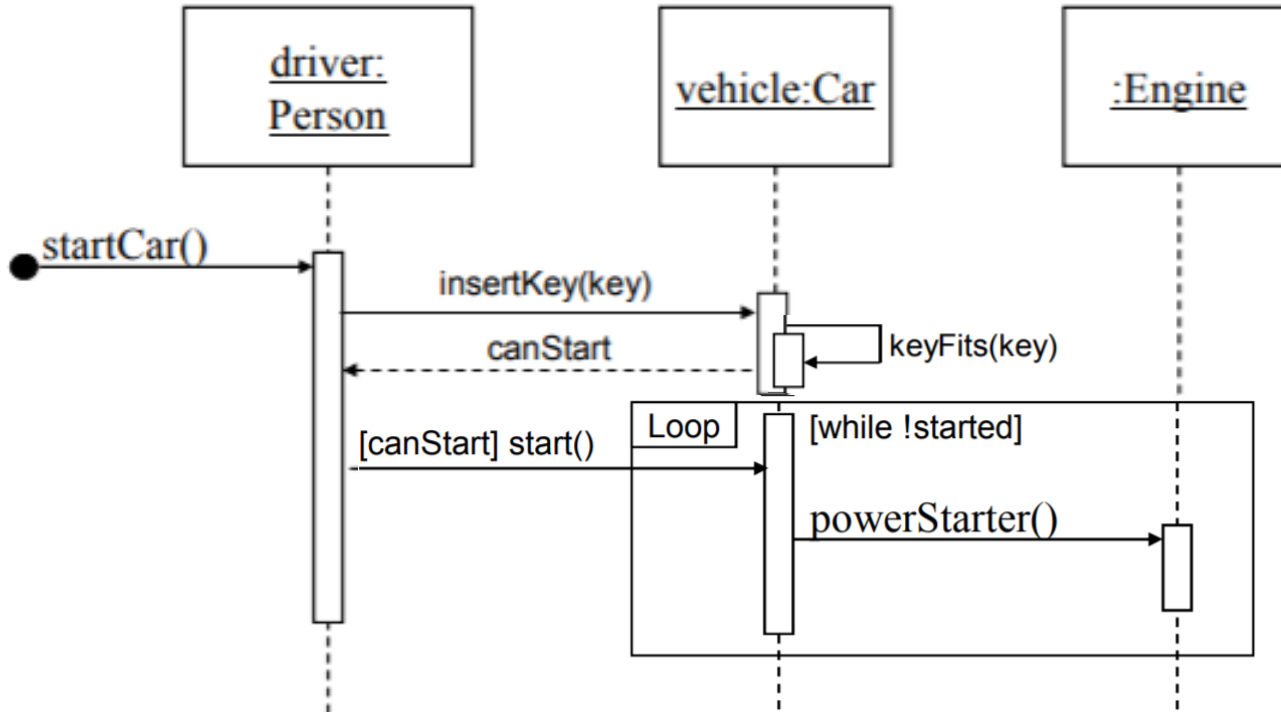




Return Values and Conditions



More Conditions

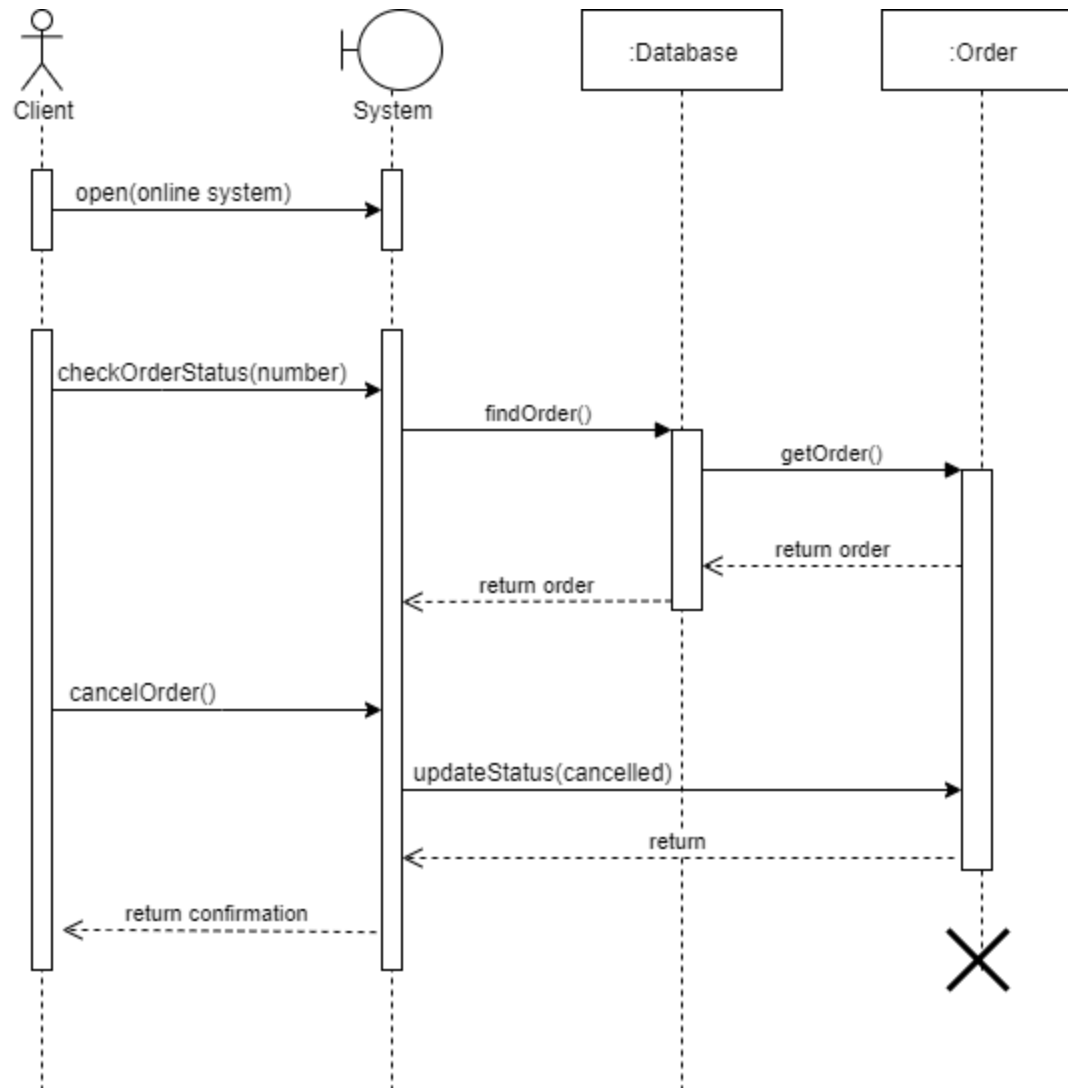


More Conditions

A good sequence diagram can help write precisely code

```
public class Person {  
    private Car vehicle;  
    private Key key;  
    public void startCar(){  
        if(vehicle.insertKey(this.key)){  
            vehicle.start(this.key);  
        }  
    }  
}  
  
public class Car {  
    private Engine engine;  
    private boolean keyInIgnition = false;  
    public boolean insertKey(Key key){  
        keyInIgnition = keyFits(key);  
        return keyInIgnition;  
    }  
    public void start(){this.engine.powerStarter();}  
    private boolean keyFits(Key k){...}  
}
```

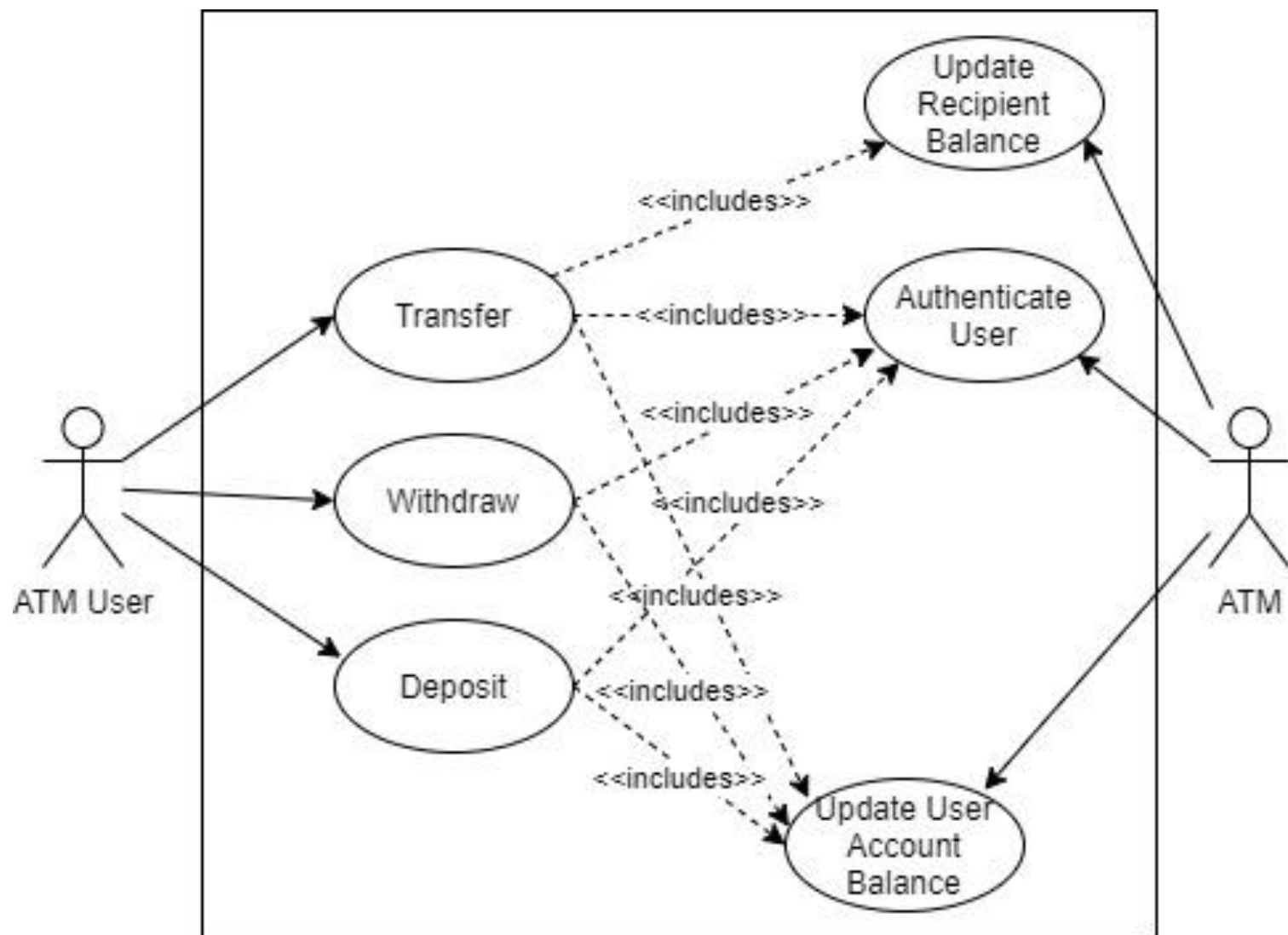
One more example: clients cancel orders on Freshii online system



Exercise

You are expected to design the software for an automated teller machine (ATM). With the ATM, a user can only withdraw, transfer (between users), and deposit cash for this example. Take cash withdrawal as an example, an ATM accepts a debit card, interacts with the user, and verifies the PIN number provided, carries out the transaction, dispenses cash, and prints receipts. ATMs communicate with a central computer, which clears the transactions with the appropriate bank.

Task: Draw a use case diagram for the ATM user, clearly indicating possible use cases and the relationships between them.



UML diagrams: class diagram

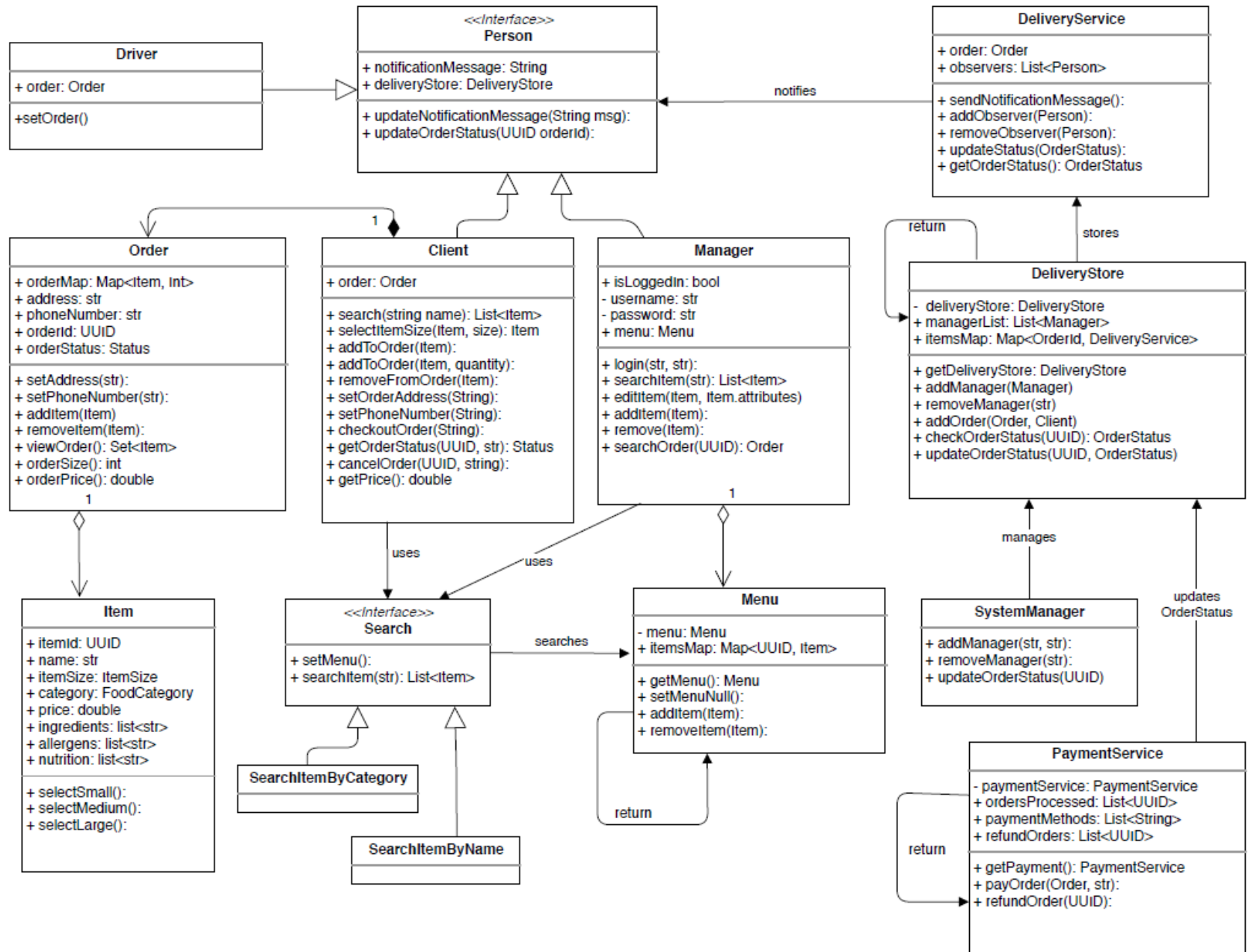
- Motivated by Object-Oriented design and programming (OOD, OOP).
- A class diagram *partitions the system into areas of responsibility* (classes) and shows “associations” (dependencies) between them.
- *Attributes* (data), *operations* (methods), *constraints*, part-of (navigability) and type-of (inheritance) relationships, access, and cardinality (1 to many) may all be noted.

Class diagram “perspective”

- Class diagrams can make sense at three distinct levels, or perspectives:
 - **Conceptual:** the diagram represents the concepts in the project domain. That is, it is a partitioning of the relevant roles and responsibilities in the domain.
 - **Specification:** shows interfaces between components in the software. Interfaces are independent of implementation.
 - **Implementation:** shows classes that correspond directly to computer code (often Java or C++ classes). Serves as a blueprint for an actual realization of the software in code.

Class diagram examples

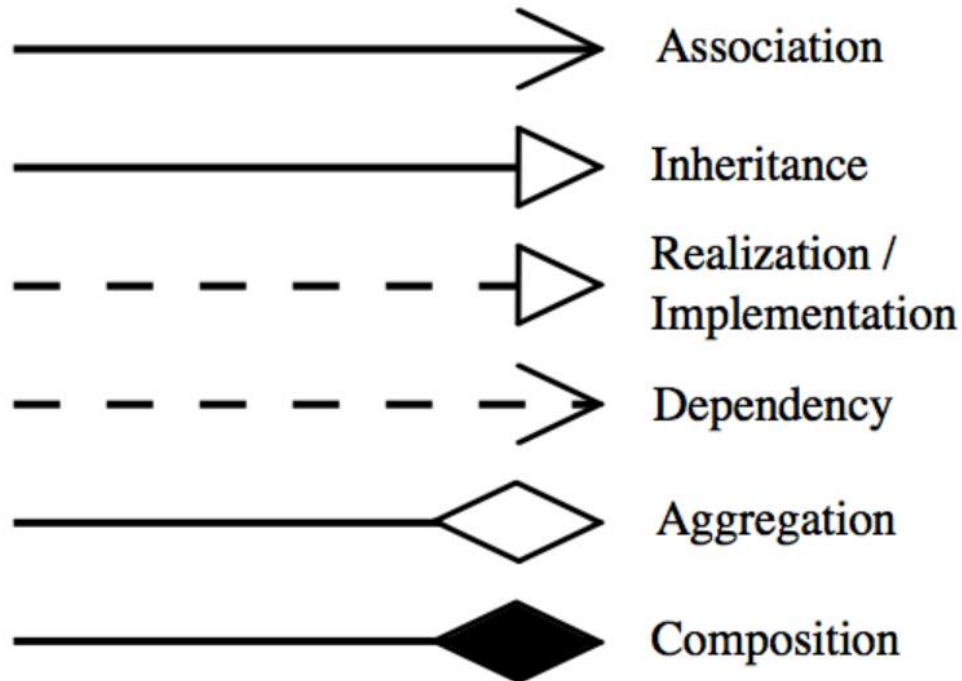
(Freshii online manage system)



Relationships

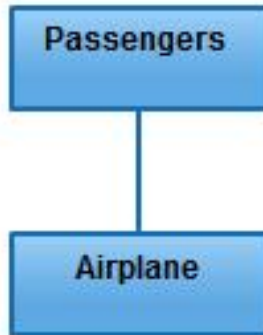
- Class diagrams may contain the following relationships:
 - Association, aggregation, dependency, realize, and inheritance

- Notation:

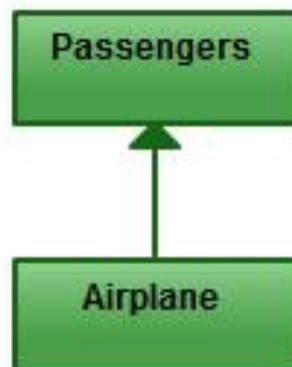


Relationships

Association is a broad term that reflects **any logical connection or relationship between classes**.

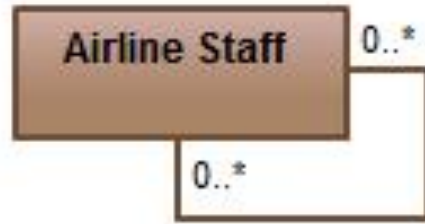


Directed Association refers to a directional relationship represented by a line with an arrowhead.

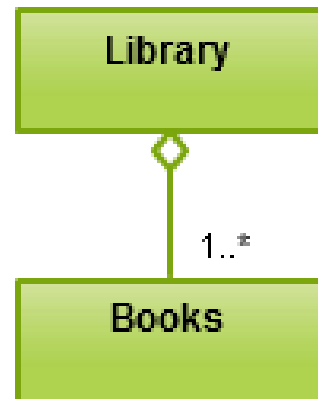


Relationships

Reflexive Association reflects any logical connection or relationship between classes.

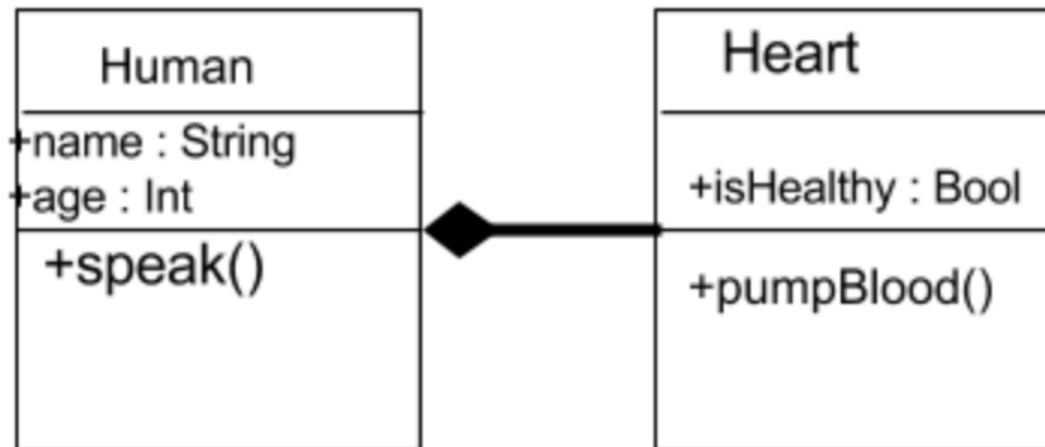


Aggregation refers to the formation of a particular class as a result of one class being aggregated or built as a collection. In aggregation, the contained classes are not strongly dependent on the lifecycle of the container.



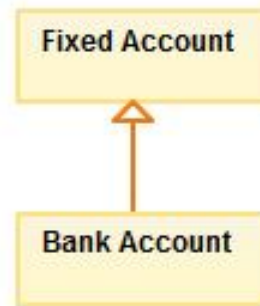
Relationships

Composition is very similar to the aggregation relationship. with the only difference being its key purpose of emphasizing the dependence of the contained class to the life cycle of the container class. That is, **the contained class will be destroyed when the container class is destroyed.**

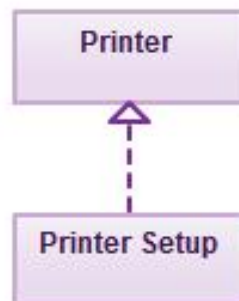


Relationships

Inheritance / Generalization refers to a type of relationship wherein **one associated class is a child of another** by virtue of assuming the same functionalities of the parent class.

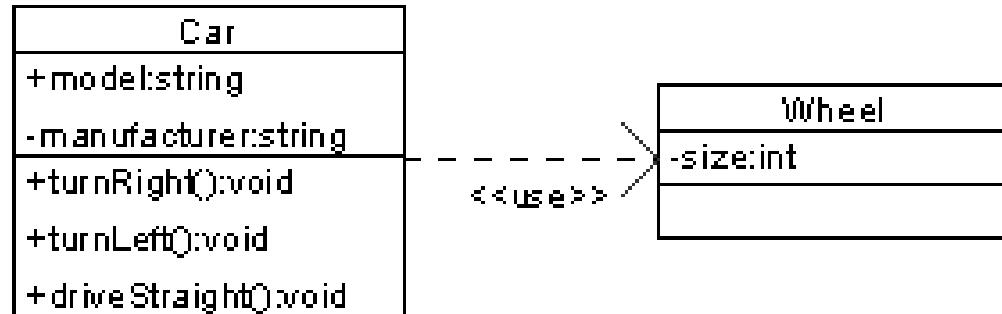


Realization denotes the implementation of the functionality defined in one class by another class.








Relationships

Dependency exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).



Multiplicity Indicators

- Each end of an association or aggregation contains a multiplicity indicator
 - Indicates the number of objects participating in the relationship

 1	Exactly one
 0..*	Zero or more
 1..*	One or more
 0..1	Zero or one
 2..7	Specified range

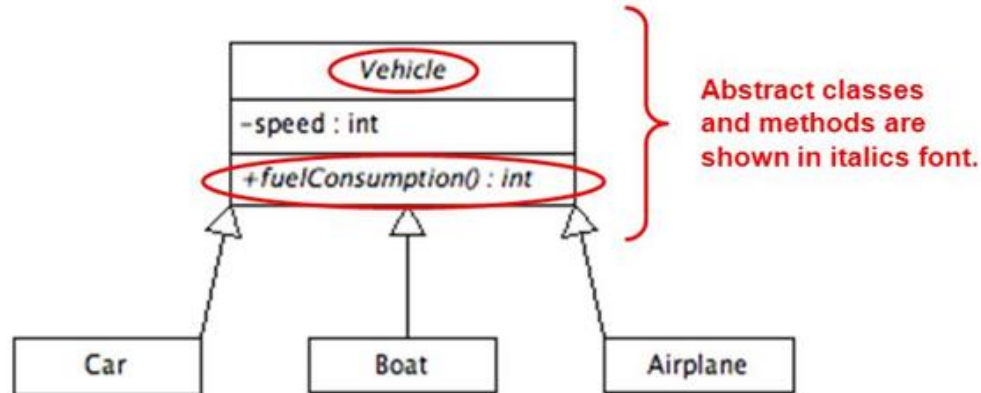
Visibility of Class attributes and Operations

- + denotes **public** attributes or operations
- denotes **private** attributes or operations
- # denotes **protected** attributes or operations
- ~ denotes **default/package** attributes or operations

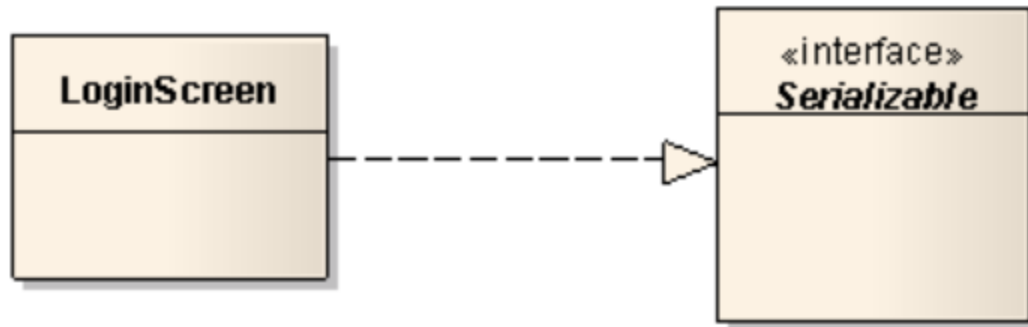
MyClass
+attribute1 : int -attribute2 : float #attribute3 : Circle
+op1(in p1 : bool, in p2) : String -op2(input p3 : int) : float #op3(out p6) : Class6*

Abstract Class vs Interface

- All abstract classes or methods should *be in italic front*



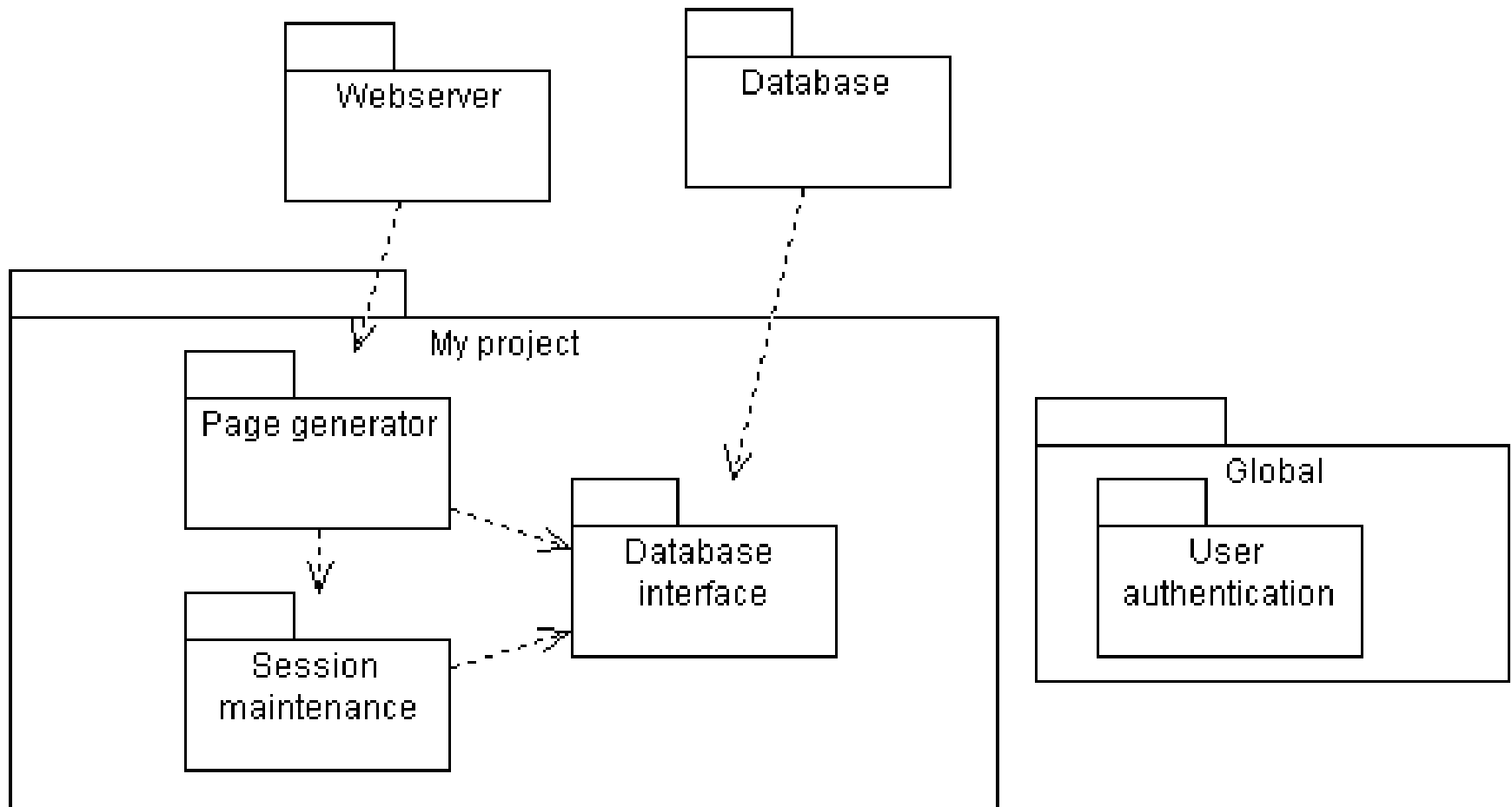
- The name of an interface is *in italic front* and *«interface»* is placed above the interface name.



UML diagrams: Package diagram

- A type of class diagram, package diagrams show dependencies between high-level system component.
- A “package” is usually a collection of related classes, and will usually be specified by it’s own class diagram.
- The software in two distinct packages is separate; packages only interact through well-defined interfaces, there is no direct sharing of data or code.
- Not all packages in a system’s package diagram are new software; many packages (components) in a complex system are often already available as existing or off-the-shelf software.

Package diagram example



UML pitfalls

- UML is a language, with a (reasonably) rigorous syntax and accepted semantics; that is, the diagrams have a meaning. Thus *you have to be careful that the meaning of your diagram is what you intended.*
- However, the semantics of UML *are less well-defined than a programming language* (where the semantics are defined by the compiler). Thus there is some leeway to use UML your own way: but you must be consistent in what you mean by the things you draw.

UML pitfalls

- **Arrow happiness:** people tend to draw arrows (associations) everywhere in their diagrams, *inconsistently* without much regard for the UML meaning of a given arrow.
- **Diagram fever:** *it's easy to do too many diagrams*. The trick is to get the correct granularity. Eg, the requirements document should leave implementation detail to the architecture.
- **General loopiness:** be careful about slapping together UML diagrams, or doing a diagram without thoroughly understanding your system. You should always be able to give a clear and concise explanation of your diagram, and why you did it that way.

Exercise

Draw *class and package diagrams* for classes from a package in Apache-Math

<https://commons.apache.org/proper/commons-math/javadocs/api-3.6/index.html>

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[Prev Package](#) [Next Package](#) [Frames](#) [No Frames](#)

Package org.apache.commons.math3.analysis.polynomials

Univariate real polynomials implementations, seen as differentiable univariate real functions.

See: [Description](#)

Class Summary

Class	Description
PolynomialFunction	Immutable representation of a real polynomial function with real coefficients.
PolynomialFunction.Parametric	Dedicated parametric polynomial class.
PolynomialFunctionLagrangeForm	Implements the representation of a real polynomial function in Lagrange Form .
PolynomialFunctionNewtonForm	Implements the representation of a real polynomial function in Newton Form.
PolynomialSplineFunction	Represents a polynomial spline function.
PolynomialsUtils	A collection of static methods that operate on or return polynomials.

Package org.apache.commons.math3.analysis.polynomials Description

Univariate real polynomials implementations, seen as differentiable univariate real functions.