

# Introduction to UML

Jun Hu

Department of Industrial Design  
Eindhoven University of Technology  
j.hu@tue.nl


<http://id00243.id.tue.nl/ObjectOrientationAndDesignPatterns>

18th February 2005

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	So you have done Java A&B . . . . .	4
1.2	Brainwashing . . . . .	4
1.3	UML: why yet another language? . . . . .	5
1.4	UML History . . . . .	5
1.5	Standardization: OMG . . . . .	6
1.6	General Goals of UML . . . . .	7
1.7	Overview of UML 2.0 . . . . .	8
<b>2</b>	<b>UML 1.x Diagrams</b>	<b>10</b>
2.1	♥ Use Case Diagrams . . . . .	10
2.1.1	Overview . . . . .	10
2.1.2	What is a use case? . . . . .	13
2.1.3	Actors In Use Cases . . . . .	13
2.1.4	Example of a use case . . . . .	13
2.1.5	Simplifying Complex Use Cases . . . . .	14
2.1.6	Use Case Best Practices . . . . .	15
2.1.7	Common Use Case Pitfalls . . . . .	15

2.2	♡ Class Diagrams . . . . .	15
2.2.1	Classes . . . . .	16
2.2.2	Attributes . . . . .	16
2.2.3	Operations . . . . .	17
2.2.4	Visibility . . . . .	17
2.2.5	Class & Instance Scope . . . . .	18
2.2.6	Bi-directional Associations . . . . .	18
2.2.7	Association names & role defaults . . . . .	19
2.2.8	Multiplicity & Collections . . . . .	19
2.2.9	Aggregation & Composition . . . . .	20
2.2.10	Generalization . . . . .	21
2.2.11	Overriding Operations . . . . .	21
2.2.12	Interface & Realization . . . . .	21
2.2.13	Abstract Classes & Abstract Operations . . . . .	22
2.2.14	More on Generalization . . . . .	23
2.2.15	Dependencies . . . . .	23
2.2.16	Qualified Associations . . . . .	24
2.2.17	Association Classes . . . . .	24
2.2.18	Associations, Visibility & Scope . . . . .	24
2.2.19	Information Hiding . . . . .	25
2.2.20	Exercise . . . . .	26
2.3	Object Diagrams & Filmstrips . . . . .	26
2.3.1	Instances of Class Diagrams . . . . .	27
2.3.2	Object State . . . . .	27
2.3.3	Filmstrips . . . . .	29
2.4	♡ Sequence Diagrams . . . . .	29
2.4.1	Why sequence diagrams . . . . .	30
2.4.2	Messages & Timelines . . . . .	30
2.4.3	Object Creation & Destruction . . . . .	30
2.4.4	Collections and Iterations . . . . .	31
2.4.5	Conditional Messages . . . . .	31
2.4.6	Class Operations . . . . .	32
2.4.7	Recursion . . . . .	32
2.5	Collaboration Diagrams . . . . .	32
2.6	♡ Activity Diagrams . . . . .	33
2.6.1	Process Flow . . . . .	33
2.6.2	Concurrency . . . . .	35
2.6.3	Swim lanes . . . . .	36

2.6.4	Signals and Exceptions	36
2.7	State Diagrams	37
2.7.1	State transitions	38
2.7.2	Transitions and Actions	39
2.7.3	Sub States & History States	40
2.8	Component Diagrams	40
2.8.1	Components and Dependencies	41
2.8.2	Components Can Contain Components	41
2.9	Package Diagrams	42
2.10	Deployment Diagrams	43
<b>3</b>	<b>UML Views</b>	<b>43</b>
3.1	Use Case View	44
3.2	Logical View	44
3.3	Component View	45
3.4	Concurrency View	45
3.5	Deployment View	45
<b>4</b>	<b>Where to start?</b>	<b>46</b>
<b>5</b>	<b>Tools</b>	<b>46</b>
	<ul style="list-style-type: none"> <li>• Introduction</li> <li>• UML 1.x Diagrams</li> <li>• UML Views</li> <li>• Where to start?</li> <li>• Tools</li> </ul>	
	► Introduction	
	<ul style="list-style-type: none"> <li>• So you have done Java A&amp;B</li> <li>• Brainwashing</li> <li>• UML: why yet another language?</li> <li>• UML History</li> <li>• Standardization: OMG</li> <li>• General Goals of UML</li> <li>• Overview of UML 2.0</li> </ul>	

📖 ▶ Introduction ▶ So you have done Java A&B

*Owning a hammer doesn't make one an architect.*

- You learnt Object-oriented *Programming*.
- You have encountered/mastered the following concepts:
  - Encapsulation
  - Class/Object
  - Composition/Aggregation
  - Generalization/Inheritance
  - Polymorphism
  - Messages
- Useful for *decomposing/modeling/understanding* the complexity
- From a craftsman to a designer:
  - One level up:** Forget about Java, think about object oriented *Analysis/Design*
  - Even higher:** pattern oriented *Anaysis/Design*

📖 ▶ Introduction ▶ Brainwashing

Please forget about the following first:

- `int i = 0;`
- `if {...} else {...}`
- `System.out.print("hello, hell");`
- `while(true){`  
     `if(Sensor.A.readValue()==1) Sound.playTune(extremely, happy);`  
   `}`

Lets pick up the terms that you prefer as a designer:

- product • system • things • scenario • user, consumer, people • parents, children • relationship • communication • competencies • ...

📖 ▶ Introduction ▶ UML: why yet another language?

- Object oriented analysis and design models are to
  - Communicate
  - Specify
  - Define Software architecture
  - Manage complexity
  - Facilitate reuse
- All these tasks require a concise and unambiguous modeling language.
  - In 1994, more than 50 OO methods:  
*Fusion, Shlaer-Mellor, ROOM, Class-Relation, Wirfs-Brock, Coad-Yourdon, MOSES, Syntropy, BOOM, OOSD, OSA, BON, Catalysis, COMMA, HOOD, Ooram, DOORS*
  - Graphical notations differ
  - The process differs or remains vague
  - But: Industry needs standards!

📖 ▶ Introduction ▶ UML History

- To stop the OO method wars, UML was invented by “3 Amigos”:



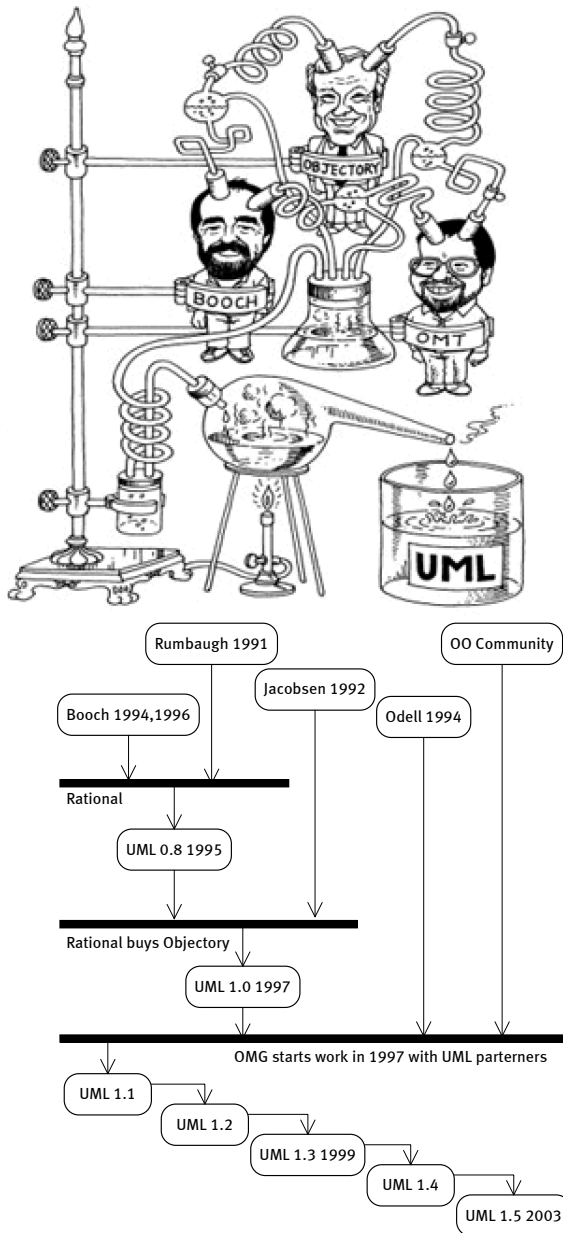
**Grady Booch:** The Booch Method (*Conception, Architecture*)

**Ivar Jacobson:** Object Object-Oriented Software Engineering (OOSE) (*Use Cases*)

**James Rumbaugh:** Object Modeling Technique (OMT) (*Analysis*)

- UML standardization
  - Started in 1994 by putting aside their own methods and notations
  - Version 1.0 published in 1997
  - Version 2.0 is coming soon
  - It has become the formal and *de facto* standard

Introduction ► UML History (2)



Introduction ► Standardization: OMG

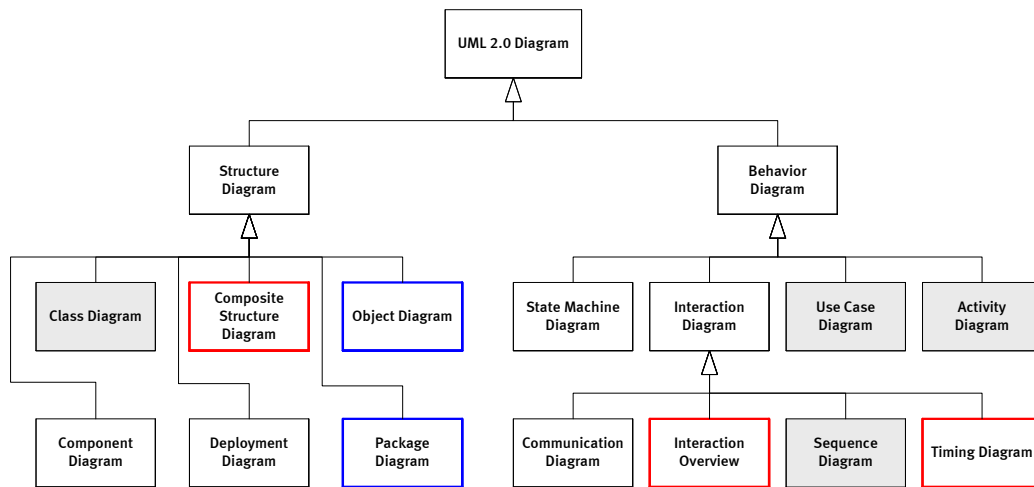
- OMG = Object Management Group

- Non-profit organization founded in 1989
  - Over 700 linked companies
  - Usually known for CORBA (IDL, IIOP ...)
- June 1996 : Task Force Object Analysis & Design
  - Definition of a standard model form
  - Definition of a “Meta-model” standard
  - Definition of a graphical notation (optional)

 ► Introduction ► General Goals of UML

- Model systems using OO concepts
  - express object-oriented designs visually
  - programming language independent
  - provide high-level documentation
  - communicate, evaluate, and reuse designs
  - can think about design, before coding
- Establish an explicit coupling to conceptual as well as executable artifacts
- To create a modeling language usable by both humans and machines
- Models different types of systems  
*information systems, technical systems, embedded systems, realtime systems, distributed systems, system software, business systems, UML itself, ...*

 ► Introduction ► Overview of UML 2.0



There are three classifications of UML diagrams:

**Behavior diagrams** A type of diagram that depicts behavioral features of a system or business process. This includes activity, state machine, and use case diagrams as well as the four interaction diagrams.

**Interaction diagrams** A subset of behavior diagrams which emphasize object interactions. This includes communication, interaction overview, sequence, and timing diagrams.

**Structure diagrams** A type of diagram that depicts the elements of a specification that are irrespective of time. This includes class, composite structure, component, deployment, object, and package diagrams.

The following summarizes the thirteen, up from nine in UML 1.x, diagram types of UML 2.x:

**Activity Diagram** Depicts high-level business processes, including data flow, or to model the logic of complex logic within a system.

**Class Diagram** Shows a collection of static model elements such as classes and types, their contents, and their relationships.

**Communication Diagram** Shows instances of classes, their interrelationships, and the message flow between them. Communication diagrams typically focus on the structural organization of objects that send and receive messages. Formerly called a Collaboration Diagram.



**Component Diagram** Depicts the components that compose an application, system, or enterprise. The components, their interrelationships, interactions, and their public interfaces are depicted.

**Composite Structure Diagram** Depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system.

**Deployment Diagram** Shows the execution architecture of systems. This includes nodes, either hardware or software execution environments, as well as the middleware connecting them.

**Interaction Overview Diagram** A variant of an activity diagram which overviews the control flow within a system or business process. Each node/activity within the diagram can represent another interaction diagram.

**Object Diagram** Depicts objects and their relationships at a point in time, typically a special case of either a class diagram or a communication diagram.

**Package Diagram** Shows how model elements are organized into packages as well as the dependencies between packages.

**Sequence Diagram** Models the sequential logic, in effect the time ordering of messages between classifiers.

**State Machine Diagram** Describes the states an object or interaction may be in, as well as the transitions between states. Formerly referred to as a state diagram, state chart diagram, or a state-transition diagram.

**Timing Diagram** Depicts the change in state or condition of a classifier instance or role over time. Typically used to show the change in state of an object over time in response to external events.

**Use Case Diagram** Shows use cases, actors, and their interrelationships.

## UML 1.x Diagrams

- Use Case Diagrams
- Class Diagrams
- Object Diagrams & Filmstrips
- Sequence Diagrams
- Collaboration Diagrams
- Activity Diagrams
- State Diagrams
- Component Diagrams
- Package Diagrams
- Deployment Diagrams

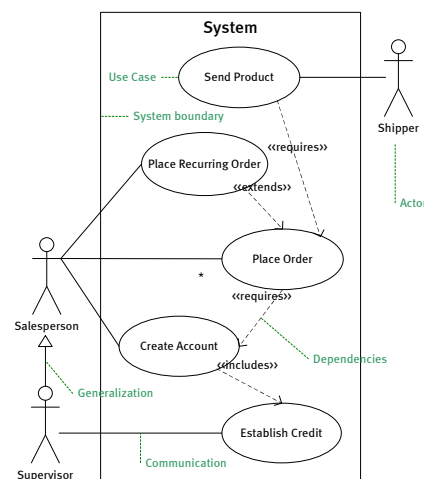
## UML 1.x Diagrams Use Case Diagrams

- Overview
- What is a use case?
- Actors In Use Cases
- Example of a use case
- Simplifying Complex Use Cases
- Use Case Best Practices
- Common Use Case Pitfalls

## UML 1.x Diagrams Use Case Diagrams Overview

Specifies participants in a use case and the relationships between use cases.

- Use cases
- Actors (or roles)
- Communication associations between actors and use cases
- Specialization/Generalization between actors/roles
- Use case dependencies:
  - «equivalent»
  - «extends»
  - «includes»
  - «requires»
  - «follows»
  - «resembles»



The most important thing to know about any use case for a system is its functional goal. Many projects lose sight of the real purpose of use cases, choosing to focus on the scenarios and often irrelevant details.

The functional goals of a system mean *the things users can do with the system*. This is a very important distinction – use cases are not features of the system. They are things users can do using the features of the system.

In this sense, use cases indirectly describe features of the system. If you are unsure as to the functional goals of your system's use cases, you must urge the requirements analyst and the customer to put more thought into why the features they're asking for must exist in the first place.

All too often, analysts invest too much time on the details of the use cases and not on the reasons behind them. Most importantly, you will need to know exactly what the functional goals of the system are in order to validate the usefulness of anything you plan to build.

Use case diagrams are deliberately simple and made up of only a handful of easily-remembered notational elements:

**Use Case:** An oval shape with the name of the use case inside. In UML, a use case generalizes a set of use case instances (or scenarios) in much the same way that classes generalize a set of similar objects. In the running system, we see instances of use cases. People often confuse use cases with use case scenarios, and it is vital to object-oriented development that one understands the subtle distinction as use case scenarios are important in choosing system test cases which will drive the whole development effort.

**Actor:** A stick man that represents a role played by one or more users. Actors are external roles played by people or other systems that can trigger (instigate) use case instances. Poor choices of actor name would be "Jason" or "Managing Director" because these may well not describe the role the actor plays in respect of the use case. Just as many of us wear different "hats" in our professional and personal life (consultant, mentor, boss, father, son and so on), individual users can play many roles and therefore be represented by more than one actor. It's not uncommon, for example, for a system administrator to also be a general user of the system.

**Communication Relationship:** We show that an actor takes part in a use case by drawing a line between them. This tells us that the actor and the system communicate in the course of an instance of a use case (a scenario),

and so is called a communicates relationship. Communication relationships are especially useful for deciding which roles require permission to take part in which use cases – in other words, it can help to show functional entitlement. Notice that this relationship is bi-directional – that is, the communication can go from actor to use case instance, or from use case instance to actor (in UML, when we don't see an arrowhead at one end of a relationship we often assume that the relationship works both ways). So, our withdraw cash use case could communicate with the bank's own internal systems to check the account has sufficient funds.

**System Boundary:** Optionally, it can be useful sometimes to show that certain use cases form part of a specific system. By drawing a system boundary around the withdraw cash use case, we're saying that this use case belongs to the ATM system. In other words, if a card holder wants to withdraw cash, they can do so using the ATM system. In practice, system boundaries are really only useful when there might be more than one system's use cases depicted on the same diagram. Arguably it is redundant otherwise. Still, it is often overlooked in analysis where the system begins and ends, so it can be helpful to make the boundary explicit. In object-oriented development, we are especially concerned in knowing that in order for actors to take part in use case scenarios, there must be some protocol through which the system and the actors communicate – eg, a web browser, a windows UI, a mobile phone, RPC client and so forth.

Often you will find that two or more use cases are in some way related. In UML, we can show three different kinds of relationship between use cases.

**equivalent** Equivalent use cases have identical activities and identical flow, but end users think of them as different. (“Deposit” and “Withdrawal” might have identical activities, though the objects involved might be different.)

**extends** When extension extends base, all the activities of the base use case are also performed in the extension use case, but the extension use case adds additional activities to – or slightly modifies existing activities of – the base use case. (To place a recurring order, you must perform all the activities of placing an order plus set up the recurrence.) If a set of activities occur in several use cases, it's reasonable to "normalize" these common activities out into a base use case, and then extend it as necessary.

**includes** A subcase. If case includes subcase, then the activities of subcase are performed one or more times in the course of performing case. (An "Authenticate" subcase may be included in several larger use cases, for example.)

**requires, follows** If follower requires leader, then leader must be completed before you can execute the follower use case. (You must create an account before you can place an order.)

**resembles** Two use cases are very similar, but do have different activities.

 ► [UML 1.x Diagrams](#) ► [Use Case Diagrams](#) ► **What is a use case?**

Describes a functional requirement of the system as a whole from an external perspective:

- Library Use Case: *Borrow book*
- VCR Use Case: *Set Timer*
- Top1Toy's Use Case: *Buy cheap plastic toy*
- IT Help Desk Use Case: *Log issue*

 ► [UML 1.x Diagrams](#) ► [Use Case Diagrams](#) ► **Actors In Use Cases**

- Actors are external roles
- Actors initiate (and respond to) use cases
  - *Sales rep* logs call
  - *Driver* starts car
  - Alarm system alerts *duty officer*
  - *Timer* triggers email

 ► [UML 1.x Diagrams](#) ► [Use Case Diagrams](#) ► **Example of a use case**

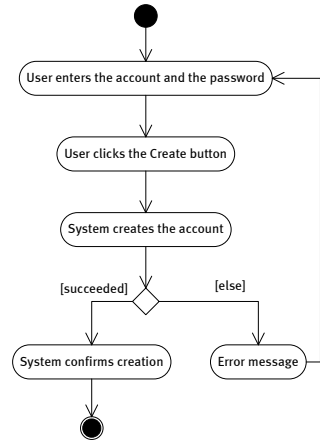
A use case can be specified textually:

Use case: **Create account**  
 Involved actor(s): Salesperson

1. Salesperson enters the account name and password
2. Salesperson clicks the Create button
3. The system creates an account
4. The system confirms the created account

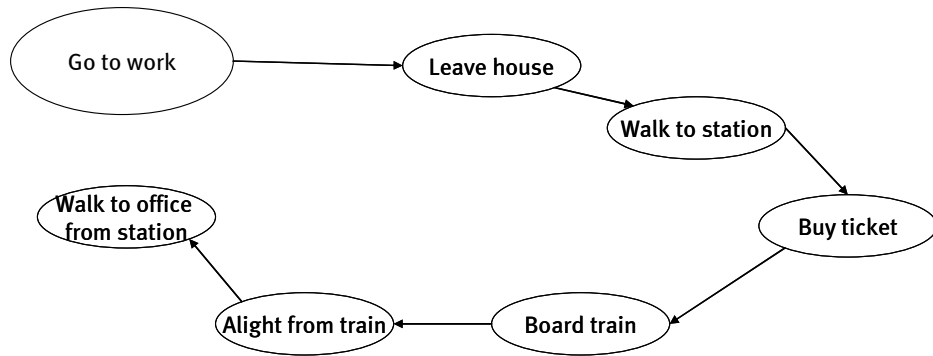
*Alternative: Creation Failed*  
 If the system fails at 3, then the system shows a message about the failure and redirects the user back to the step 1

Or using activity diagrams:



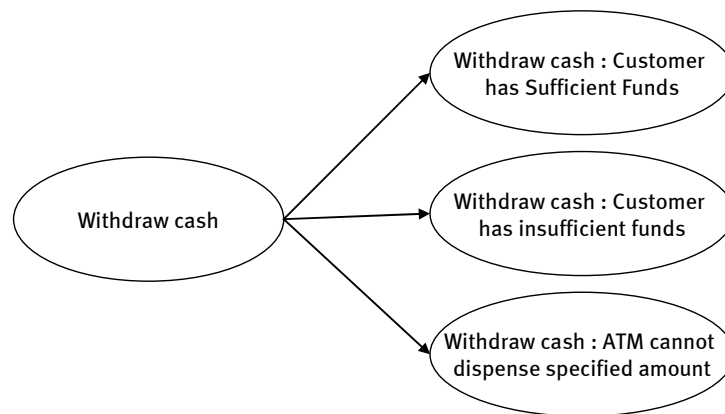
UML 1.x Diagrams > Use Case Diagrams > Simplifying Complex Use Cases

Strategy #1 : Break large/complex use cases down into smaller and more manageable use cases



UML 1.x Diagrams > Use Case Diagrams > Simplifying Complex Use Cases

Strategy #2 : Break large/complex use cases down into multiple scenarios (or test cases)



► UML 1.x Diagrams ► Use Case Diagrams ► Use Case Best Practices

- Keep them simple & succinct
- Don't write all the use cases up front - develop them incrementally
- Revisit all use cases regularly
- Prioritise your use cases
- Ensure they have a single tangible & testable goal
- Write them from the user's perspective, and write them in the language of the business
- Set a clear system boundary and do not include any detail from behind that boundary
- Look carefully for alternative & exceptional flows

► UML 1.x Diagrams ► Use Case Diagrams ► Common Use Case Pitfalls

- The system boundary is undefined or inconstant
- The use cases are written from the system's (not the actors') point of view
- The actor names are inconsistent
- There are too many use cases
- The use-case specifications are too long
- The customer doesn't understand the use cases
- The use cases are never finished

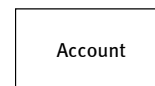
► UML 1.x Diagrams ► Class Diagrams

- Classes
- Attributes
- Operations

- Visibility
- Class & Instance Scope
- Bi-directional Associations
- Association names & role defaults
- Multiplicity & Collections
- Aggregation & Composition
- Generalization
- Overriding Operations
- Interface & Realization
- Abstract Classes & Abstract Operations
- More on Generalization
- Dependencies
- Qualified Associations
- Association Classes
- Associations, Visibility & Scope
- Information Hiding
- Exercise

 ► UML 1.x Diagrams ►  Class Diagrams ► **Classes**

```
class Account {  
}
```



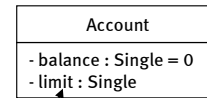
In most cases, we need to know three things about a class in UML. What is its name? What attributes does it have? What responsibilities (operations) does it have? We can model this using the class notation in UML.

A class has at least one compartment, telling us the name of the class. Optionally, it also has a compartment telling us what attributes the class has, and another telling us what operations it has. Conventionally, the name compartment is first, followed by attributes and then operations. A class can also have other compartments that give us more information. For example, sometimes we might want to show that a class must follow a set of rules, so we might add a constraints compartment. Generally, it's better not to overcomplicate a class diagram because it makes them harder to read.

 ► UML 1.x Diagrams ►  Class Diagrams ► **Attributes**



```
class Account {
    private float balance = 0;
    private float limit;
}
```



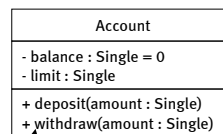
[visibility] [/] attribute\_name[multiplicity] [: type [= default\_value]]

### UML 1.x Diagrams ▶ Class Diagrams ▶ Operations

```
class Account{
    private float balance = 0;
    private float limit;

    public void deposit(float amount){
        balance = balance + amount;
    }

    public void withdraw(float amount){
        balance = balance - amount;
    }
}
```



[visibility] op\_name([[in/out] parameter : type[, more params]])[: return\_type]

### UML 1.x Diagrams ▶ Class Diagrams ▶ Visibility

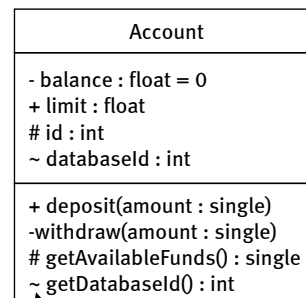
```
class Account{
    private float balance = 0;
    public float limit;
    protected int id;
    int databaseld;

    public void deposit(float amount){
        balance = balance + amount;
    }

    private void withdraw(float amount){
        balance = balance - amount;
    }

    protected int getId(){
        return id;
    }

    int getDatabaseld(){
        return databaseld;
    }
}
```



+ = public  
- = private  
# = protected  
~ = package

In UML, we can denote the visibility of an attribute or operation by placing the appropriate access modifier in front of the member name.

Private members are only accessible from inside the same class.

Public members can be accessed from anywhere in the model.

Protected members are only accessible in the same class or from any of its subclasses (more about subclasses later).

Package visibility means that only classes in the same UML package can access that member (more on packages later).

99.9999% of the time, visibility is entirely an issue for technical design and should not be included in analysis models as it just confuses non-technical folk.

### 📖 ▶ UML 1.x Diagrams ▶ ♥ Class Diagrams ▶ Class & Instance Scope

```
class Person {
    private static int numberOfPeople = 0;
    private String name;

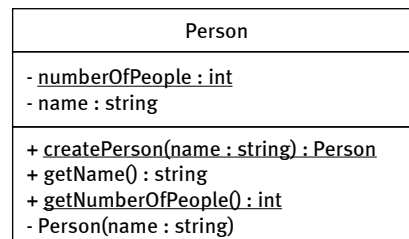
    private Person(string name){
        this.name = name;
        numberOfPeople++;
    }

    public static Person createPerson(string name){
        return new Person(name);
    }

    public string getName(){
        return this.name;
    }

    public static int getNumberOfPeople(){
        return numberOfPeople;
    }
}

/*-----*/
int numOfPeople = Person.getNumberOfPeople();
Person p = Person.createPerson("Jun_Hu");
```

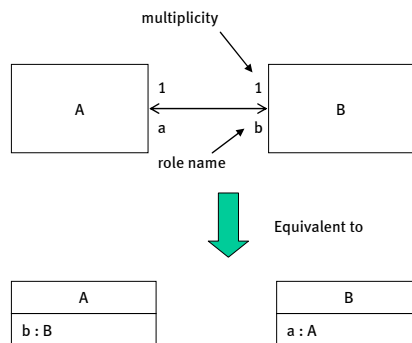


### 📖 ▶ UML 1.x Diagrams ▶ ♥ Class Diagrams ▶ Bi-directional Associations

```
class A
{
    public B b;
    public A(){
        b = new B(this);
    }
}

/*-----*/
class B{
    public A a;
    public B(A a)
    {
        this.a = a;
    }
}

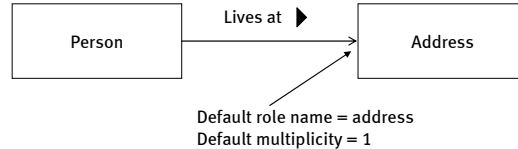
/*-----*/
A a = new A();
B b = a.b;
A a1 = b.a;
assert a == a1;
```



### UML 1.x Diagrams ▶ Class Diagrams ▶ Association names & role defaults

```
class Person
{
    // association: Lives at
    public Address address;

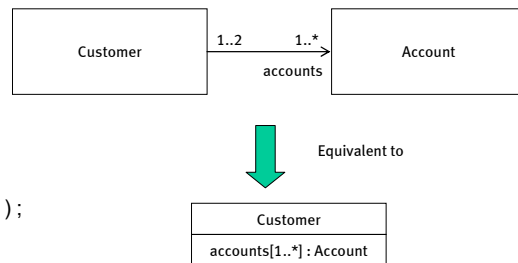
    public Person(Address address)
    {
        this.address = address;
    }
}
```



### UML 1.x Diagrams ▶ Class Diagrams ▶ Multiplicity & Collections

```
class Customer
{
    // accounts[1..*] : Account
    ArrayList accounts = new ArrayList();

    public Customer()
    {
        Account defaultAccount = new Account();
        accounts.add(defaultAccount);
    }
}
```



Just as we can work with collections of objects in Java, we can also model collections in UML. Anything that has a multiplicity with an upper limit of more than 1 is effectively a collection of objects of that type. Just as a private variable in a class could be declared as an array (or similar collection type) of a certain type of object (eg, `string[] middleNames`), so too can an attribute be declared as a collection of objects of that type.

Multiplicity is denoted by a range of two or more numbers separated by “..” or “,”, where “..” means “to” and “,” means “or”:

- 0..\* = zero to many
- 1..4 = one to four
- 2, 4 = two or four
- 1 = just one
- \* = many (implies zero to many)

### UML 1.x Diagrams ▶ Class Diagrams ▶ Aggregation & Composition

```

/***** Aggregation *****/
public class ClassA {
    private Class b;

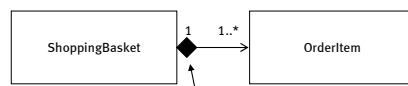
    public classA(ClassB b) {
        this.b = b;
    }
}

/***** Composition *****/
public class ClassA {
    private ClassA b = new ClassB ();
}

```



Aggregation – is made up of objects that can be shared or exchanged



Composition – is composed of objects that cannot be shared or exchanged and live only as long as the composite object

Aggregation is for clearer communication its sometimes necessary to show that one object is made up of one or more other objects. For example, a personal computer is made up of parts like a motherboard, hard drive and a graphics card. Similarly, a motherboard is made up of smaller parts which includes one or more processors.

Aggregation relationships like these are shown by adding an un-shaded diamond at the aggregate end of the relationship (the container end, if you like).

In programming terms, aggregation has no real meaning. Since the objects that make up an aggregate can exist without it and can be shared in relationships with other objects there is no practical way of implementing an aggregation that would make it any different from a straightforward association.

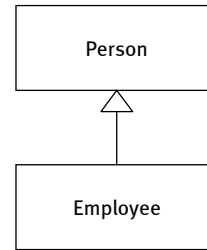
A stronger kind of aggregation is composition. Unlike their aggregated counterparts, composing objects (like the line items of an invoice or elements of an HTML document) can only exist while the composite parent object is around. In strict terms, the lifetime of objects that compose another are limited to the lifetime of the composite object. This has very practical implications

for managing the lifetimes of objects engaged in a composition relationship. They cannot be shared in relationships with other objects, and must be destroyed when the parent is destroyed.

### UML 1.x Diagrams ▶ Class Diagrams ▶ Generalization

```
class Person{
}

class Employee extends Person{
}
```



What gives object oriented programming its real power is the ability to abstract. When we work with abstractions we gain the ability to apply the same logic to a family of related classes that share characteristics – attributes and operations – but that also specialize beyond those characteristics to offer something new.

### UML 1.x Diagrams ▶ Class Diagrams ▶ Overriding Operations

```
class Account
{
    protected float balance = 0;
    protected float limit = 0;

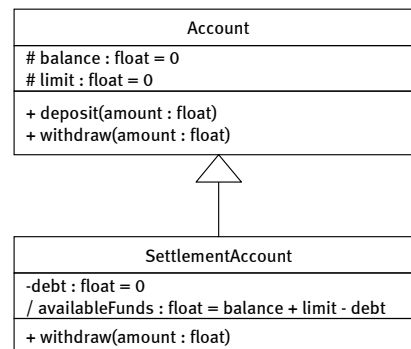
    public void deposit(float amount){
        balance = balance + amount;
    }

    public void withdraw(float amount){
        balance = balance - amount;
    }
}

class SettlementAccount extends Account{
    private float debt = 0;

    float availableFunds(){
        return (balance + limit - debt);
    }

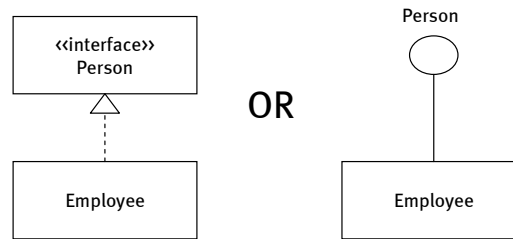
    public void withdraw(float amount){
        if (amount > this.availableFunds()){
            throw new InsufficientFundsException ();
        }
        base.withdraw (amount);
    }
}
```



### UML 1.x Diagrams ▶ Class Diagrams ▶ Interface & Realization

```
interface Person{
}

class Employee implements Person{
}
```



It can often be preferable to implement an interface instead of using inheritance, for example if you knew that different kinds of people eat in different ways then there would no need to have an implementation of the eat() operation on the Person class. A class that has no attributes and implements none of its operations (all of its operations are abstract) is essentially just an interface. In programming languages like C# and Java, classes cannot inherit from more than one class (though in other languages like C++ multiple inheritance is allowed – though not recommended) but they can implement more than one interface. This makes it possible for the same class to wear many hats depending on who is using it.

An interface is entirely a design concept, because in analysis interfaces have no meaning at all.

#### ► UML 1.x Diagrams ► Class Diagrams ► Abstract Classes & Abstract Operations

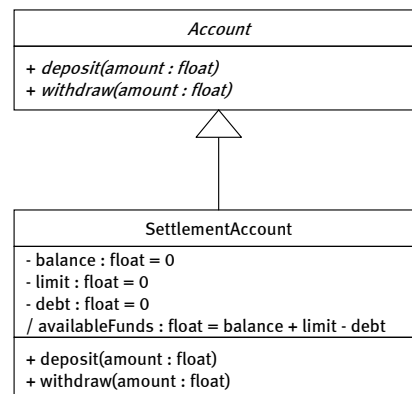
```
abstract class Account{
    public abstract void deposit(float amount);
    public abstract void withdraw(float amount);
}

class SettlementAccount extends Account{
    private float balance = 0;
    private float limit = 0;
    private float debt = 0;

    float availableFunds() {
        return (balance + limit - debt);
    }

    public void deposit(float amount){
        balance = balance + amount;
    }

    public void withdraw(float amount){
        if (amount > this.availableFunds()){
            throw new InsufficientFundsException();
        }
        balance = balance - amount;
    }
}
```

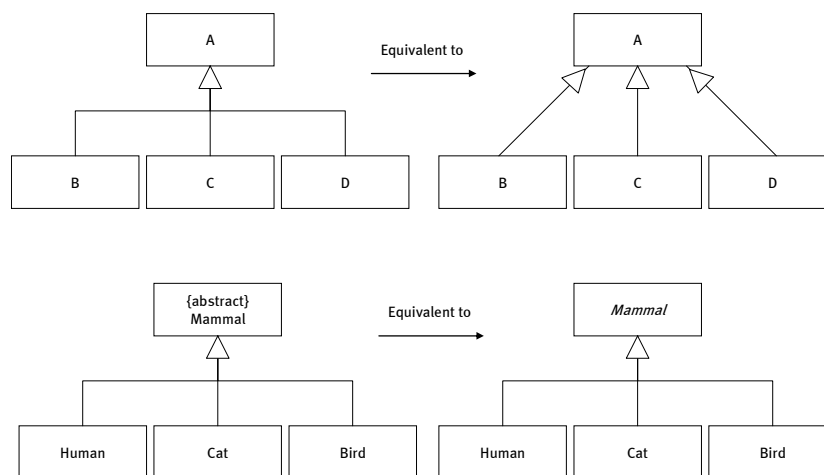


Sometimes we want to define a class that implements shared characteristics of a family of similar classes, but that does not define a set of objects in

their own right. Abstract classes are classes that can have some implementation, but that can never be directly instantiated themselves. They are designed purely to be extended (or to offer class-scope operations only).

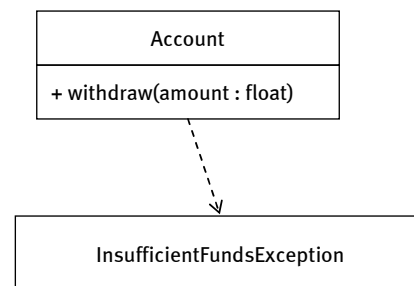
Similarly, some operations are intended only to be implemented by subclasses. Abstract operations define the signature of an operation only – ie, its name, visibility, arguments and return type. Concrete subclasses must override these abstract operations.

📖 ▶ UML 1.x Diagrams ▶ ♥ Class Diagrams ▶ More on Generalization



📖 ▶ UML 1.x Diagrams ▶ ♥ Class Diagrams ▶ Dependencies

```
public class Account {
    public void withdraw(float amount)
        throws InsufficientFundsException {
    }
}
```



The four kinds of relationship we've seen so far can be thought of as structural relationships. An association, aggregation or composition relationship between class A and class B can be thought of as A having an attribute of type B (or a collection of objects of type B). If A inherits from B then A has the same interface and implementation as B.

But sometimes we just want to show that A is somehow dependant on B when it doesn't inherit from B or have an attribute of type B. Sometimes we just want to show that a change to the external characteristics of B (eg, the signature of an operation) would require a change in the implementation of A. For example, both Person and Scuba Diver are dependant on the type Food because they both have operations that take an object of type Food as a parameter.

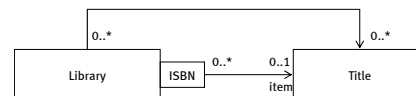
Of course, it would not make sense to draw every single dependency on our class diagrams because they are likely to be many and the diagram would soon get cluttered and be unreadable. But sometimes a dependency is important and needs to be communicated as part of the design.

#### UML 1.x Diagrams ▶ Class Diagrams ▶ Qualified Associations

hash-table, associative array, “dictionary” ...

```
class Library{
  private HashMap titles = new HashMap();

  public Title item(String isbn){
    return (Title)titles.get(isbn);
  }
}
```



#### UML 1.x Diagrams ▶ Class Diagrams ▶ Association Classes

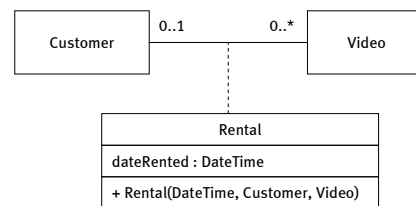
```
class Customer{
  ArrayList rentals = new ArrayList();
}

class Video{
  Rental rental;
}

class Rental{
  Customer customer;
  Video video;

  DateTime dateRented;

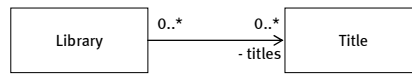
  public Rental(DateTime dateRented, Customer customer, Video video){
    this.dateRented = dateRented;
    video.rental = this;
    customer.rentals.add(this);
    this.customer = customer;
    this.video = video;
  }
}
```



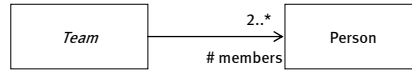
#### UML 1.x Diagrams ▶ Class Diagrams ▶ Associations, Visibility & Scope



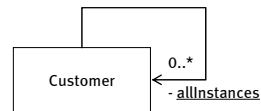
```
class Library{
    private Title [] titles;
}
```



```
class Team{
    protected Person [] members;
}
```

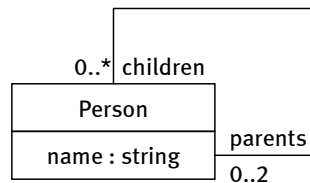


```
class Customer{
    private static Customer [] allInstances;
}
```



UML 1.x Diagrams > Class Diagrams > Information Hiding

```
class Person
{
    public String name;
    public Parent [] parents = new Parent [2];
    public ArrayList children = new ArrayList ();
}
```



```
Person mary = new Person ();
Person ken = new Person ();
Person jason = new Person ();
jason.parents [0] = mary;
jason.parents [1] = ken;
mary.children.add (jason);
ken.children.add (jason);
jason.name = "Jason";
```

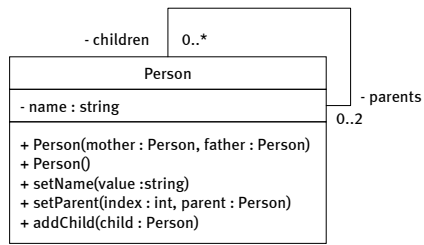
UML 1.x Diagrams > Class Diagrams > Information Hiding (2)

```
class Person{
    private String name;
    private Parent [] parents = new Parent [2];
    private ArrayList children = new ArrayList ();

    public Person(Person mother, Person father){
        this.setParent(0, mother);
        this.setParent(1, father);
    }
    public void setName(String value){
        this.name = value;
    }
    public void setParent(int index, Person parent){
        parents[index] = parent;
        parent.addChild(this);
    }
    public void addChild(Person child){
        this.children.add(child);
    }
    public Person()
{
}
}
```

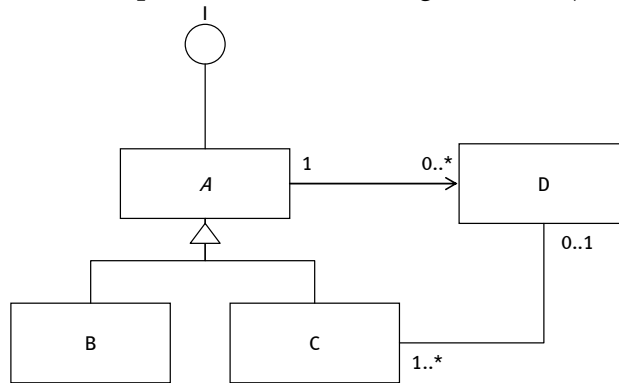
```

{
}
Person mary = new Person ();
Person ken = new Person ();
Person jason = new Person(mary, ken);
jason.setName("Jason");
```



UML 1.x Diagrams ▶ Class Diagrams ▶ Exercise

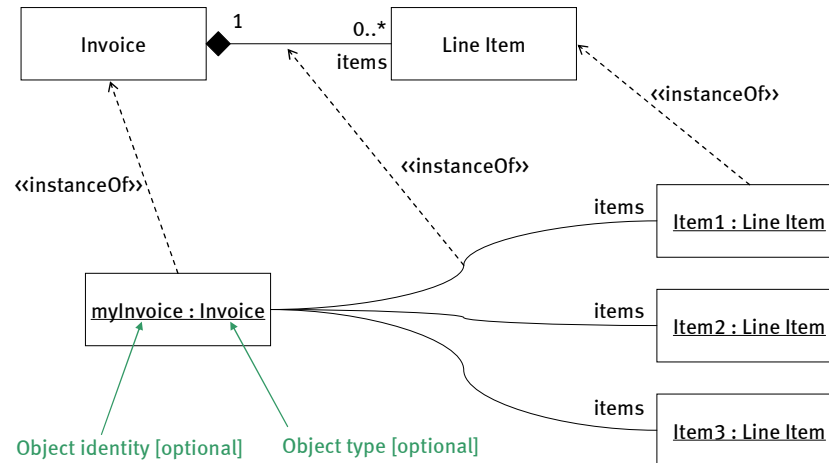
Write the Java code to implement the class diagram exactly as below:



UML 1.x Diagrams ▶ Object Diagrams & Filmstrips

- Instances of Class Diagrams
- Object State
- Filmstrips

UML 1.x Diagrams ▶ Object Diagrams & Filmstrips ▶ Instances of Class Diagrams



In OO programming, objects are instances of classes. In UML, objects are drawn in a way similar to classes, but we optionally can give objects a unique identity (eg, `myInvoice`) and optionally specify an object's type.

When we omit the identifier then the object is an anonymous instance – often the identity is not relevant to what we're trying to communicate. In analysis we often don't want to commit ourselves to a class diagram when we model specific snapshots of the system or business scenario, so we can omit the object type and just pick a suitable identifier.

The identifier and object type are underlined, which helps us easily distinguish between objects and classes.

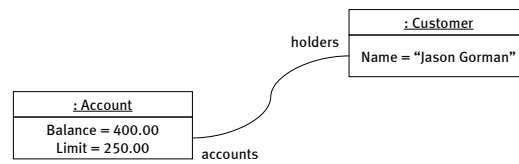
The dependency `<<instanceOf>>` tells us that an object or link is an instance of a specific class or association (including aggregations and compositions).

Object diagrams must conform to the rules defined in the class model. In this example, we have 3 line items inserted into the role `items`. Since `items` has a multiplicity of 0 or more than this conforms to the class model. At the other end, every line item must be linked to exactly one invoice because the role of `Invoice` in this association has a multiplicity of exactly 1.

UML 1.x Diagrams ▶ Object Diagrams & Filmstrips ▶ Object State

- Object diagrams are *snapshots* of class diagrams in execution – that is, if we could execute a model (but that's another story...)
- We may use snapshots to model

the *before* and *after* of tests cases to see what has changed and then assign responsibility for those changes in high-level design.



- We may also use snapshots to *debug* high-level designs in much the same way we use breakpoints to debug our code.

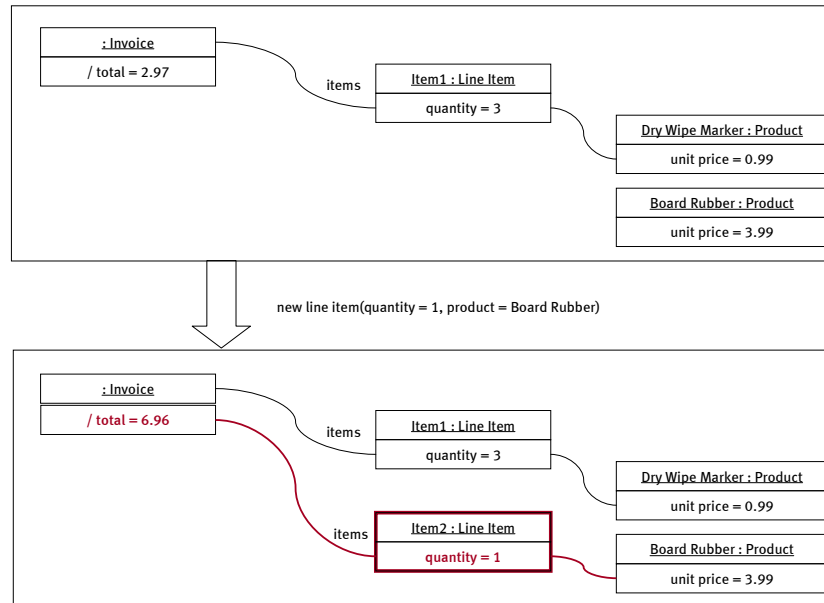
The best way to think of the difference between class diagrams and object diagrams (or “snapshots”) is to relate it to the difference between the source code of an OO program and the objects in memory when you pause execution at a specific point.

Most debugging tools let you set breakpoints in code so you can stop it at that point and inspect the values of variables, the type and identity of object references and so forth to see if things are as you’d expect them to be.

In UML, we use object diagrams to illustrate the state of the system (or part of a system) at some “breakpoint” in execution – that is, if we could execute a model (but that’s another story...)

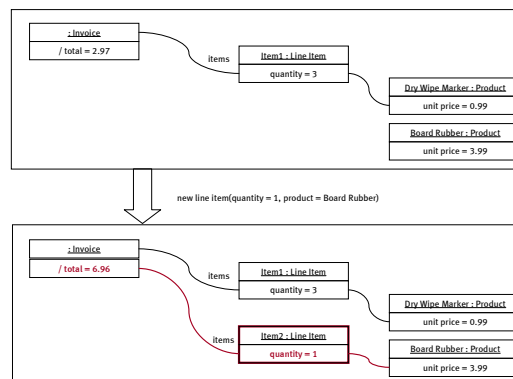
Object diagrams are probably the most useful and the most under-utilized part of UML. In object-oriented development, we use snapshots to model the “before” and “after” of tests cases to see what has changed and then assign responsibility for those changes in high-level design (using sequence diagrams). We also use snapshots to “debug” high-level designs in much the same way we use breakpoints to debug our code.

UML 1.x Diagrams ▶ Object Diagrams & Filmstrips ▶ Filmstrips



UML 1.x Diagrams ▶ Object Diagrams & Filmstrips ▶ Filmstrips (2)

- We can animate the effect of an operation using a pair of snapshots
- Change are highlighted so they are easy to spot.
- There are several effects we need to note in a filmstrip:
  - Changes to attribute values
  - Object creation
  - Link creation
  - Link destruction



UML 1.x Diagrams ▶ Sequence Diagrams

- Why sequence diagrams
- Messages & Timelines

- Object Creation & Destruction
- Collections and Iterations
- Conditional Messages
- Class Operations
- Recursion

📖 ▶ UML 1.x Diagrams ▶ ♥ Sequence Diagrams ▶ Why sequence diagrams

- In OO analysis & design, the real objective is to dream up ways in which groups of objects can collaborate together to complete some useful task.
- In OO terms, we say that objects interact with each other by sending messages (in the form of method calls or events).

📖 ▶ UML 1.x Diagrams ▶ ♥ Sequence Diagrams ▶ Messages & Timelines

```

public class ClassA{
    private ClassB b = new ClassB ();

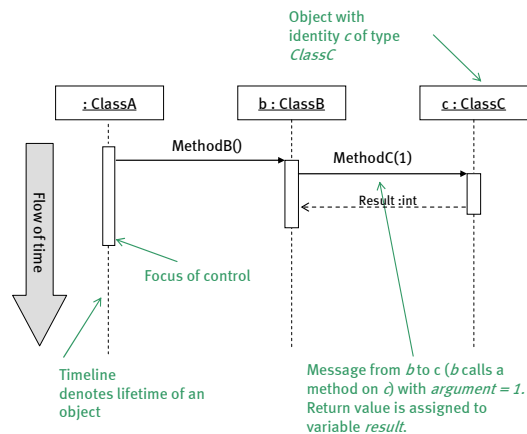
    public void methodA() {
        b.methodB ();
    }
}

public class ClassB{
    private ClassC c = new ClassC ();

    public void methodB(){
        int result = c.methodC(1);
    }
}

public class ClassC{
    public int methodC(int argument) {
        return argument * 2;
    }
}

```



📖 ▶ UML 1.x Diagrams ▶ ♥ Sequence Diagrams ▶ Object Creation & Destruction

```

public class ClassA{
    public void methodA() {
        ClassB b = new ClassB ();
        b.methodB ();
    }
}

```

```

public class ClassB{
    private ClassC c = new ClassC (2);

    public void methodB(){
        int result = c.methodC(1);
    }
}

```

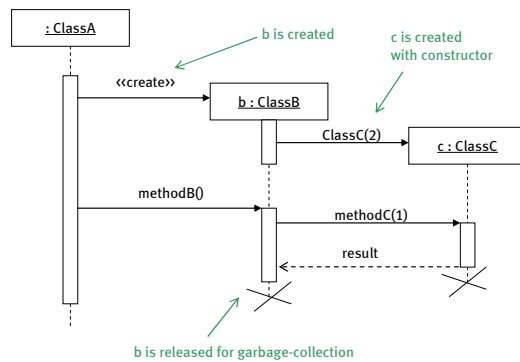
```

        this.factor = factor;

public class ClassC{
    private int factor = 0;
    public ClassC(int factor){

    }

    public int methodC(int argument){
        return argument * factor;
    }
}
    
```

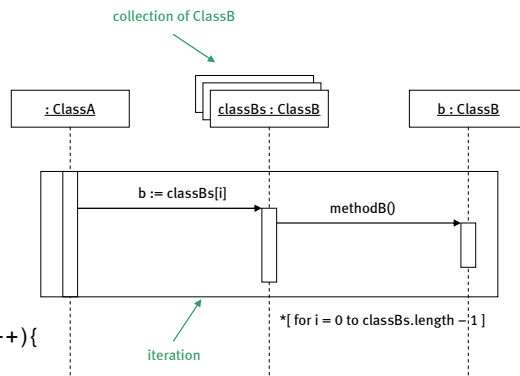


UML 1.x Diagrams > Sequence Diagrams > Collections and Iterations

```

public class ClassA
{
    //a collection of ClassB objects
    private ClassB[] classBs
    = new ClassB []
    { new ClassB (),
      new ClassB (),
      new ClassB ()
    };

    public void methodA(){
        //iteration
        for(int i = 0; i < classBs.length; i++){
            ClassB b = classBs[i];
            b.methodB ();
        }
    }
}
    
```

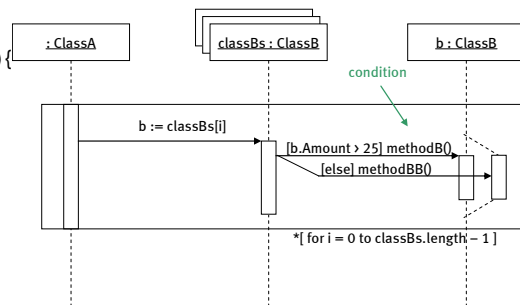


UML 1.x Diagrams > Sequence Diagrams > Conditional Messages

```

public void methodA(){
    for(int i = 0; i < classBs.length; i++){
        ClassB b = classBs[i];

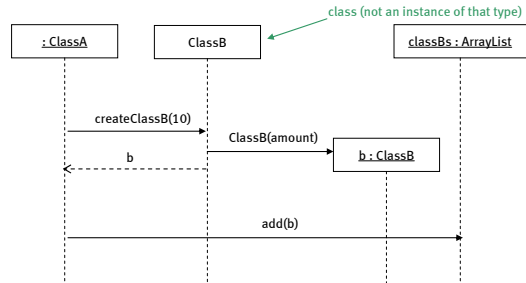
        if(b.Amount > 25){
            b.methodB ();
        }
        else{
            b.methodBB ();
        }
    }
}
    
```



UML 1.x Diagrams ▶ Sequence Diagrams ▶ Class Operations

```
public class ClassA{
    private ArrayList classBs
        = new ArrayList ();

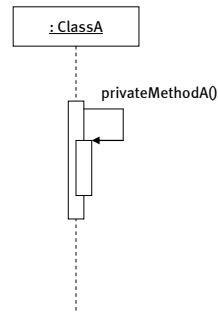
    public void methodA(){
        ClassB b = ClassB.createClassB (10);
        classBs.add(b);
    }
}
```



UML 1.x Diagrams ▶ Sequence Diagrams ▶ Recursion

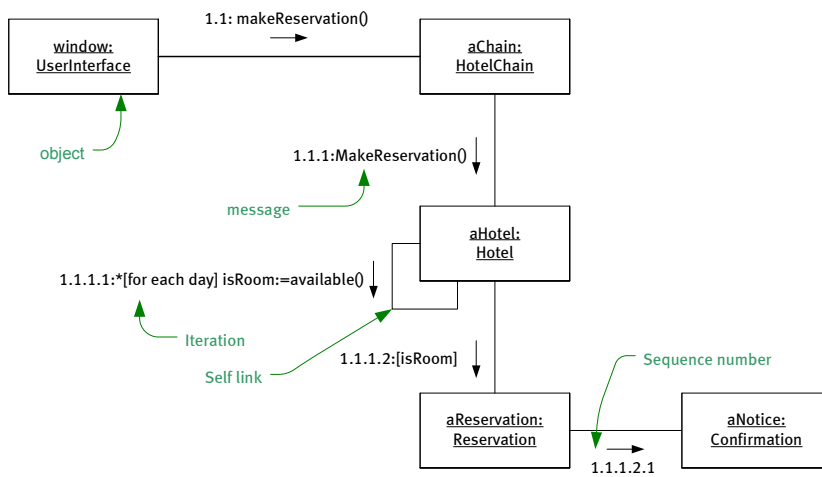
```
public class ClassA
{
    public void methodA() {
        this.privateMethodA ();
    }

    private void privateMethodA () {
    }
}
```



UML 1.x Diagrams ▶ Collaboration Diagrams

= Communication Diagrams in UML 2.x





Collaboration Diagrams are an alternative presentation of a sequence diagram. They tend to be more compact, but harder to read, than the equivalent sequence diagrams.

The boxes are objects. Lines connecting two boxes indicates that the objects collaborate with (send messages to) one another. Use a multiplicity indicator in the box (such as \*) to indicate that all elements of an aggregation receive a message.

The object name typically goes inside the box, but can go outside the box when different collaborators refer to it by different names.

Usually, the instance name (or reference through which the instance is accessed) is the same as the role the instance plays in the collaboration. When the name and role aren't identical, use instance/role:Class. E.g.: given tutor/teacher:Person and lecturer/teacher:Person, an object of class Person, used in the role of teacher, is called tutor in some portion of the code and lecturer elsewhere in the code.

Messages that flow from one object to another are drawn next to the line, with an arrow indicating direction. Arrowhead types have the same meaning as in sequence diagrams. The message sequence is shown via a numbering scheme. Message 1 is sent first. Messages 1.1, 1.2, etc., are sent by whatever method handles message 1. Messages 1.1.1, 1.1.2, etc., are set by the method that handles message 1.1, and so forth.

Guards are specified using the "Object Constraint Language", a pseudo-code that's part of the UML specification. Syntactically, it's more like Pascal and Ada than Java and C++, but it's readable enough. (The operators that will trip you up are assignment [=] equality [=] and not-equals [<>]). As in a sequence diagram, an asterisk indicates iteration.

 ► [UML 1.x Diagrams](#) ►  [Activity Diagrams](#)

- [Process Flow](#)
- [Concurrency](#)
- [Swim lanes](#)
- [Signals and Exceptions](#)

 ► [UML 1.x Diagrams](#) ►  [Activity Diagrams](#) ► [Process Flow](#)

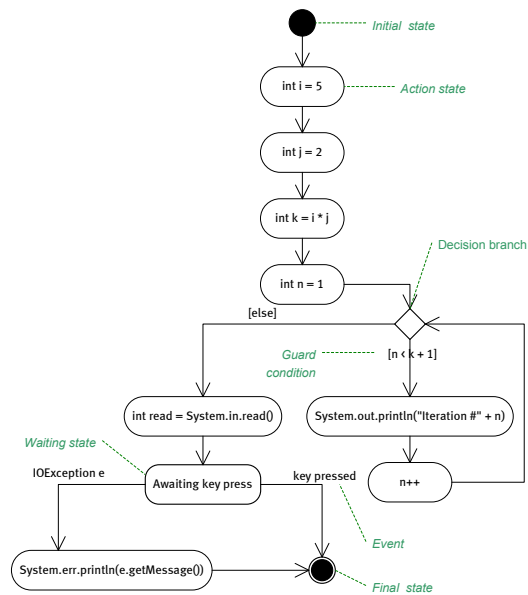
```
int i = 5;
int j = 2;
int k = i * j;
```

```
for(int n = 1; n < k + 1; n++) {
    System.out.println("Iteration_#" + n);
}
```

```

System.err.println(e.getMessage());
try {
    int read = System.in.read();
} catch(IOException e) {
}
}

```



In UML activity diagrams can be used to describe the logical flow of some process or procedure. In coding terms, this would equate to program flow – a concept we are all very familiar with IC so many aspects of activity diagrams may seem familiar.

Some of the key elements of the notation are:

**Actions** discrete steps in the process, like the steps in the execution of a piece of code. Actions are actually a special kind of state that is assumed to happen instantaneously and, when completed, the process moves directly on to the next state automatically.

**Transitions** when a process moves from one state to another, it is called a transition. In activity diagrams, the process transitions automatically from one action to the next.

**Guard conditions** often in program flow we need to specify that the process moves to one of a number of possible next steps depending on whether some condition is true. We call this branching because the flow of execution follows a specific branch of the code. In activity diagrams, we can show branching by having two or more possible transitions from the same action, specifying a guard condition on each transition that tells us when that specific branch is followed.

**Waiting states** in many applications, processes must wait for some kind of input or system event before they can continue execution. For example, our little program here waits for the user to press any key before it can finish. In activity diagrams, we draw waiting states to show that execution of a process is paused until some event occurs. The process can only transition to another action or waiting state in response to an event, which is written next to the specific transition that event will trigger.

**Start state & end states** every process must have a beginning and end (unless it is designed to go on forever, of course!) The start state on an activity diagram explicitly shows where the process begins, and can be used to describe the system state at that point. A process can end in more than one possible way, and therefore there may be several possible end states on an activity diagram.

#### UML 1.x Diagrams ▶ Activity Diagrams ▶ Concurrency

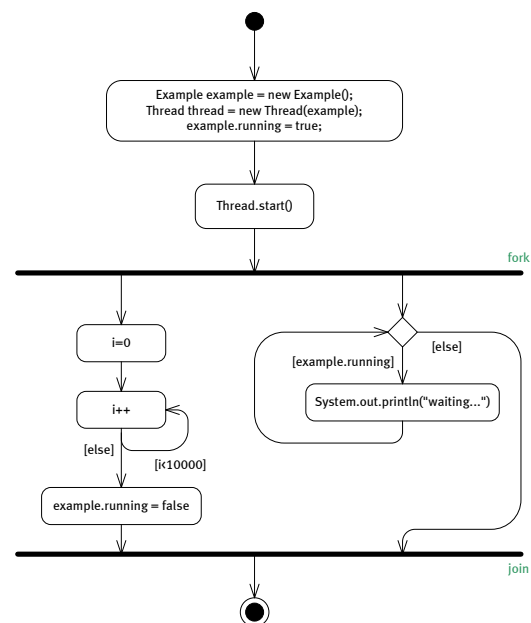
```
public class Example
    implements Runnable {

    private boolean running;

    public static void main(String[] args) {

        Example example = new Example();
        Thread thread = new Thread(example);
        example.running = true;
        thread.start();
        while (example.running) {
            System.out.println("waiting...");
        }
    }

    public void run() {
        for (int i = 1; i < 10000; i++) {}
        running = false;
    }
}
```



In an activity diagram we can easily show how one thread of execution splits into two or more concurrent threads, and then synchronizes and converges back into the original thread. A black synchronization bar is used to

show how a process *forks* into two or more concurrent processes, which then *join* at another bar back into the original thread.

📖 ▶ UML 1.x Diagrams ▶ ♥ Activity Diagrams ▶ Swim lanes

```
public class ClassA{
    private ClassB b = new ClassB ();

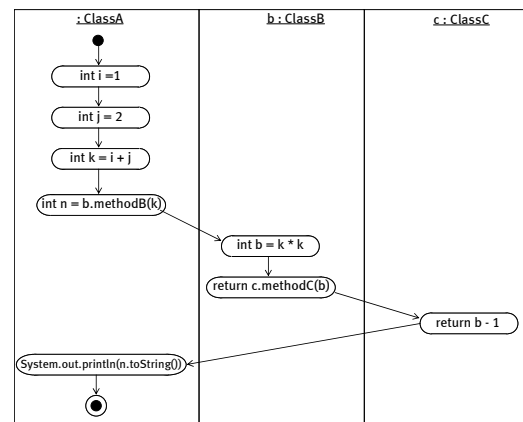
    public void methodA(){
        int i = 1;
        int j = 2;
        int k = i + j;

        int n = b.methodB(k);
        System.out.println(n.toString());
    }
}

public class ClassB{
    private ClassC c = new ClassC ();

    public int methodB(int k){
        int b = k * k;
        return c.methodC(b);
    }
}

public class ClassC{
    public int methodC(int b){
        return b - 1;
    }
}
```



Activities are arranged into vertical or horizontal zones delimited with lines. Each zone represents a broad area of responsibility, typically implemented by a set of classes or objects.

It is important to note here that the naming of variables should be local to the scope of the owning object and method. So, the variable *b* declared by the instance of *ClassC* is not the same object *b* associated to the instance of *ClassA*. Also note that I have chosen argument names to match the names of the variables assigned to those arguments – which makes the logic of the process easier to follow. I could have declared `methodB(int z)`, but without the knowledge of that method's signature `return c.methodC(z)` wouldn't have made a lot of sense – where did the variable *z* come from?

📖 ▶ UML 1.x Diagrams ▶ ♥ Activity Diagrams ▶ Signals and Exceptions

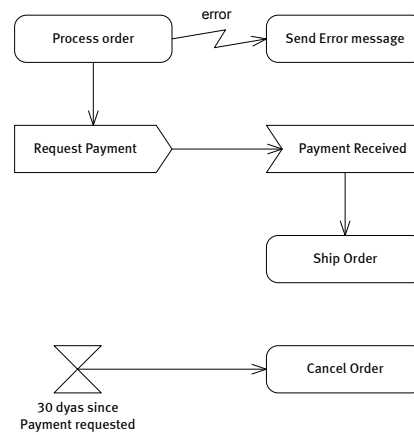
- Signals

**Generating signals:** sent to outside process (Request Payment at left).

**Accepting signals:** received from outside process (Payment Received at left).

**Timer signals:** received when time elapses or a set time arrives (30 days ... at left).

- Exceptions. Extraordinary errors that you typically don't detect with explicit tests are indicated with a "lightning bolt."

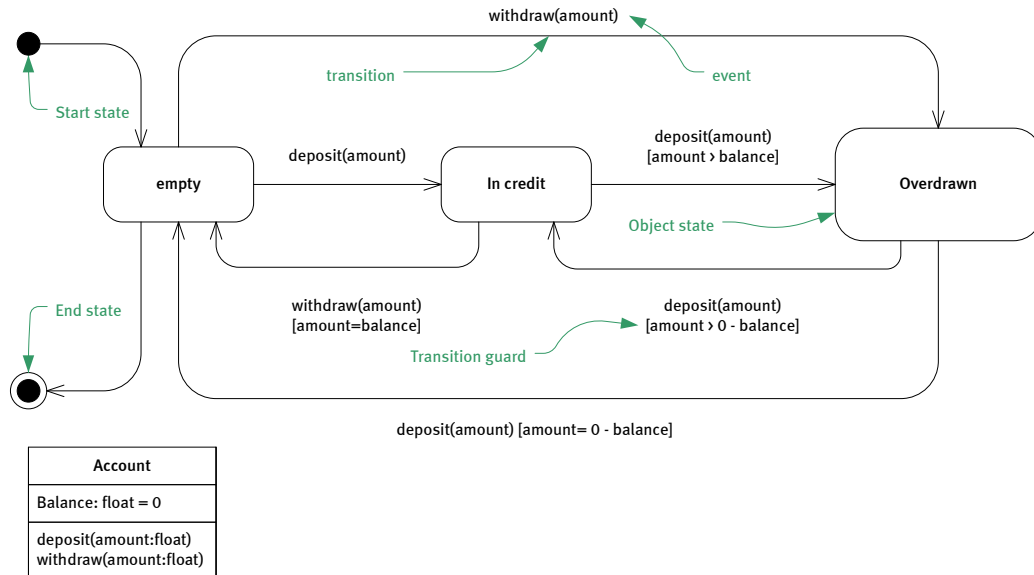


► UML 1.x Diagrams ► State Diagrams

= State Machine Diagrams in UML 2.0

- State transitions
- Transitions and Actions
- Sub States & History States

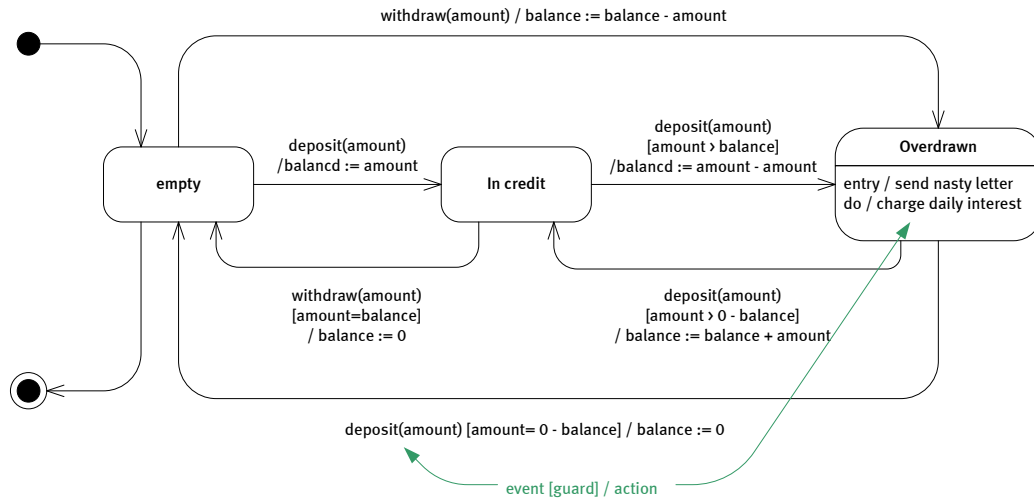
UML 1.x Diagrams ▶ State Diagrams ▶ State transitions



In many examples, the behavior of an object changes dramatically when its attribute values fall within a certain range. For example, what happens the way an account behaves might differ dramatically when it is overdrawn.

In these cases, we can model an object's behavior by considering what it does in these discrete states, how an object can transition from one state to another state and what actions are triggered when a state transition occurs.

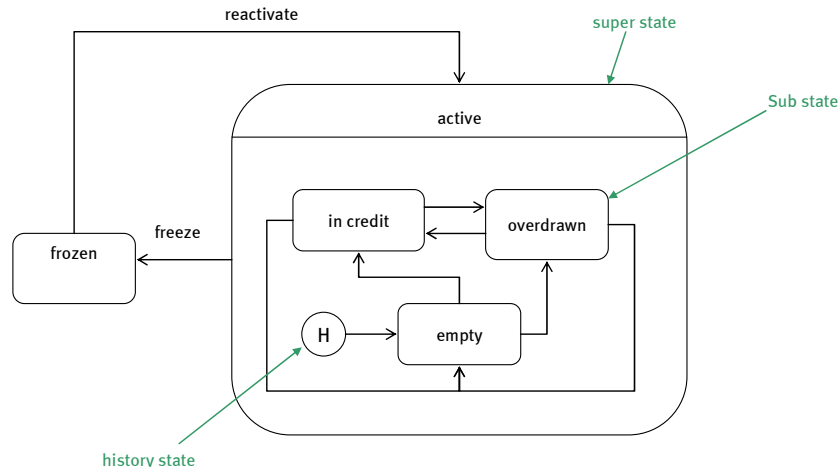
UML 1.x Diagrams ▶ State Diagrams ▶ Transitions and Actions



The most useful aspect of state transition diagrams (or state charts) is that we can describe how certain events trigger certain actions. For example, when an account becomes overdrawn we can show that a nasty letter must be sent to the customer, and while the account remains overdrawn we must charge daily interest.

Without going over the top, state transition diagrams can be a very powerful way of describing object, subsystems or system behavior. Like object diagrams, they are seldom utilized though. In UI-driven design we use a variation of state transition diagrams to model the logic of the user interface and to build and validate a clear understanding of how the user and system interact.

UML 1.x Diagrams ▶ State Diagrams ▶ Sub States & History States



It's very common for objects to find themselves in more than one discrete state at a time. For example, I can be *tired* and *awake* at the same time. I can also be *rested* and *awake* at the same time. In these cases, *tired* and *rested* are sub states of *awake*. If you can be neither if you're not conscious!

In UML, we can draw state transition diagrams inside states to indicate more interesting behavior going on whilst in that state.

If you draw a start state inside another state, that tells us that whenever the **super state** is entered the object starts in that specific state. Having said that, it's not always desirable to start in exactly the same state every time an object enters the super state.

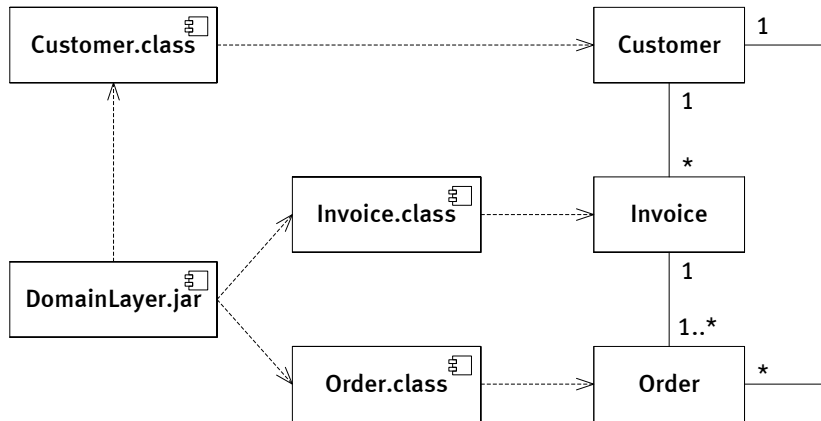
You might want to show that the object “remembers” what state it was in when it last exited the super state. We can use a **history state** to show that when an object re-enters a **composite state** it returns to the state it was in the moment it last exited that state. For example, if an account is overdrawn and the account is frozen, when it's reactivated you want to remember that the account was overdrawn before it was frozen.

UML 1.x Diagrams ▶ Component Diagrams

- Components and Dependencies
- Components Can Contain Components



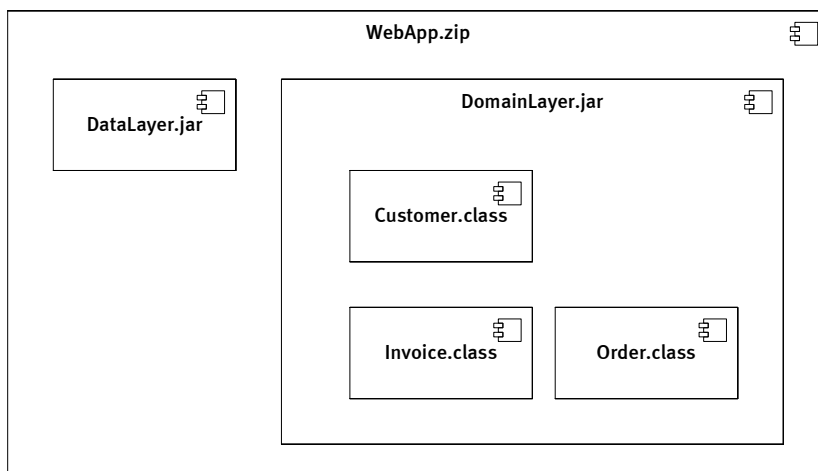
UML 1.x Diagrams ► Component Diagrams ► Components and Dependencies



In UML, we can use component diagrams to describe the physical implementation of a system. In practical terms, a component is thought of as a physical file like a `.java` source file, or a JAR, or an XML file and so on. We can use dependencies to show how different components might be related to each other. For example, the `DomainLayer.jar` requires the files `Customer.class`, `Invoice.class` and `Order.class` to be packaged in a successful build.

We can also show how physical files are related to the logical model. In this example, we're describing how the source (`.class`) files implement classes in the logical model.

UML 1.x Diagrams ► Component Diagrams ► Components Can Contain Components



In the Catalysis style of modeling, we can also describe how one component may be made up of other components. In this example, we're saying that the DomainLayer JAR file contains the bytecode files `Customer.class`, `Invoice.class` and `Order.class`.

Another example of this might be a ZIP file that contains compressed files, or a compound document that contains OLE objects (like a chart in a spreadsheet).

### UML 1.x Diagrams ► Package Diagrams

```

/*=====*/
package nl.tue.id;

class ClassA{
}

/*=====*/
package nl.tue.id.examples;

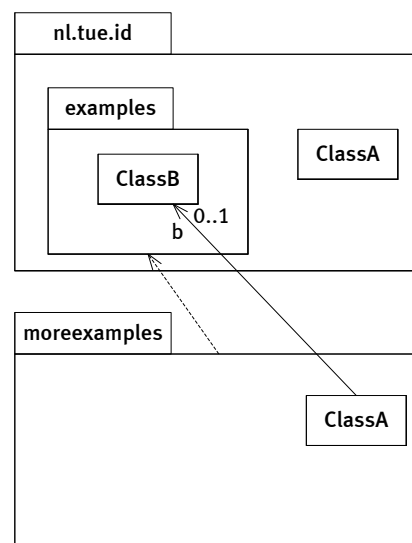
class ClassB{
}

/*=====*/
package moreexamples;

import nl.tue.id.examples.*;

class ClassA{
    private ClassB b;
}

```



Just as we can organize our code into logical packages, we can group model elements in packages. In our example above, this allows us to distinguish between `nl.tue.id.ClassA` and `moreexamples.ClassA`.

UML packages function in pretty much the same way as Java packages, except that we can assign properties to packages that tell us more about them. For example, one package might represent a subsystem.

Just as we can in Java, we can qualify the path to model elements using this notation:

```
nl.tue.id::examples::ClassB
```

We can use dependencies to show how one package is dependant on another. In this example, the dependency between `moreexamples` and

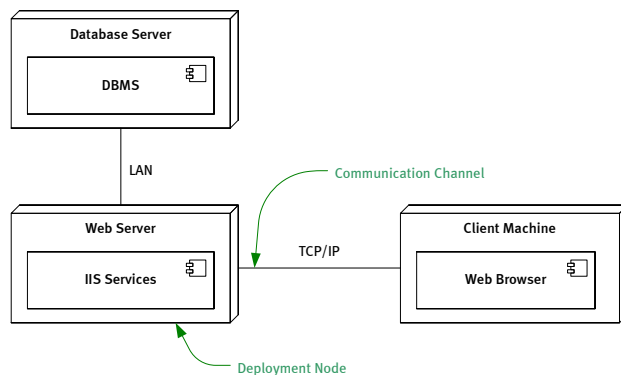
```
nl.tue.id::examples
```

tells us the more `examples` imports `nl.tue.id::examples`.

Packages can be very useful for spotting circular or cyclic package dependencies, which we strive to avoid. They can also be very useful for giving an overview of the system in terms of its different packages and the relationships between them.

### ► UML 1.x Diagrams ► Deployment Diagrams

We can use deployment diagrams to show how in the physical system components are deployed on different machines (called “nodes” in UML)

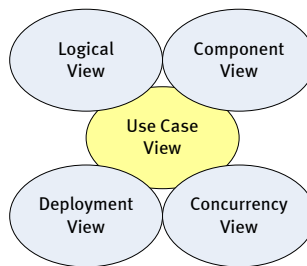


### ► UML Views

- Use Case View
- Logical View
- Component View
- Concurrency View
- Deployment View

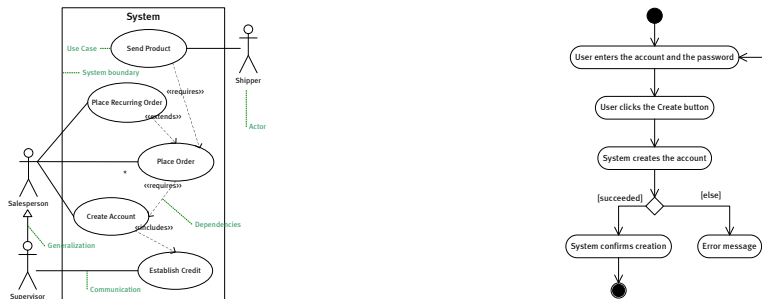
### ► UML Views (2)

- Each view is a projection of the complete system
- Each view highlights particular aspects of the system
- Views are described by a number of diagrams
- No strict separation, so a diagram can be part of more than one view



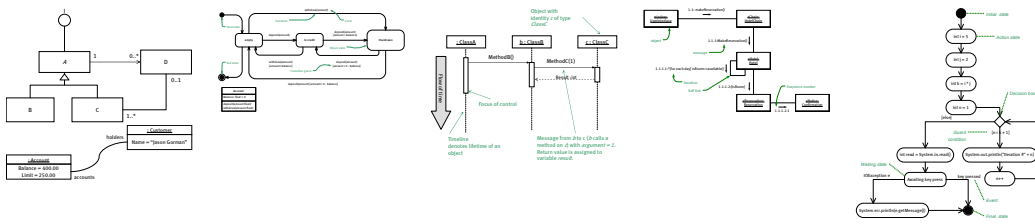
### UML Views ► Use Case View

- Shows the functionality of the system as perceived by external actors.
- Actors can be users or other systems.
- Described by use case and activity diagrams.
- The central view which drives the development of other views.
- Used by customers, designers, developers, testers.



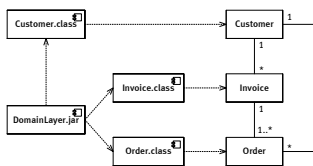
### UML Views ► Logical View

- Shows how the functionality of the system is designed / provided.
- Uses class and object diagrams to represent the static structure.
- Uses state, sequence, collaboration, and activity diagrams for dynamic behavior.
- Used by designers and developers.



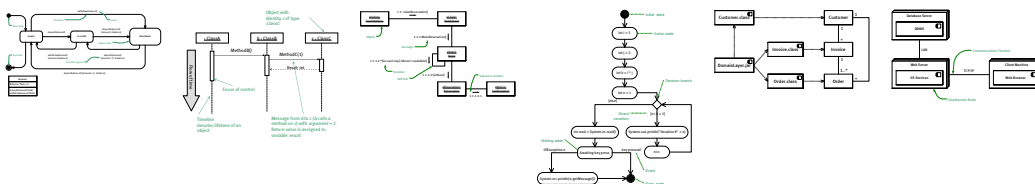
► UML Views ► Component View

- Shows the organization of the code components and their dependencies.
- Described by component diagrams.
- Used by developers.



► UML Views ► Concurrency View

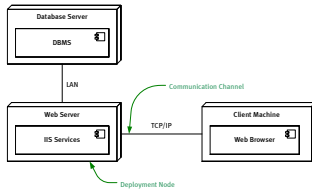
- Addresses the problems with communication and synchronization for a concurrent system.
- Described by state, sequence, collaboration, activity, deployment, and component diagrams.
- Used by developers and system integrators.



► UML Views ► Deployment View

- Shows the deployment of the system into the physical architecture with computers and devices.

- Represented by the deployment diagram.
- Used by developers, system integrators, and testers.



► Where to start?

Must:

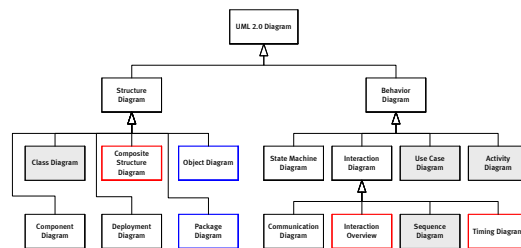
- User case diagrams
- Class diagrams
- Sequence diagrams
- Activity diagrams

Optional:

- Object diagrams
- State diagrams
- Collaboration diagrams

Gadgets:

- Component diagrams
- Deployment diagrams



► Tools

- **Microsoft® Visio®** (with UML stencils), available as campus software.
- ArgoUML: open source software.
- IBM® Rational Rose®
- Borland® Together®
- Sparx Systems® Enterprise Architect®
- ⋮
- ⋮

- The last, the cheapest, the fastest, the most convenient tools are ...  
Pen & Paper!