Introduction to UML

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

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

21st February 2005

- Introduction
- UML 1.x Diagrams
- UML Views
- Where to start?
- Tools

▸ Introduction

- So you have done Java A&B
- Brainwashing
- UML: why yet another language?
- UML History
- Standardization: OMG
- General Goals of UML
- Overview of UML 2.0

▶ 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);
  }

 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

▶ 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.

▶ 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

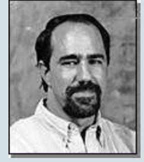
▶ Introduction ▶ UML: why yet another language?

- 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

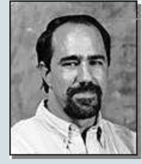
Ivar Jacobson

James Rumbaugh

Grady Booch: The Booch Method (*Conception, Architecture*)
Ivar Jacobson: Object Object-Oriented Software Engineering (OOSE) (*Use Cases*)
James Rumbaugh: Object Modeling Technique (OMT) (*Analysis*)

▶ Introduction ▶ UML History

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



Grady Booch

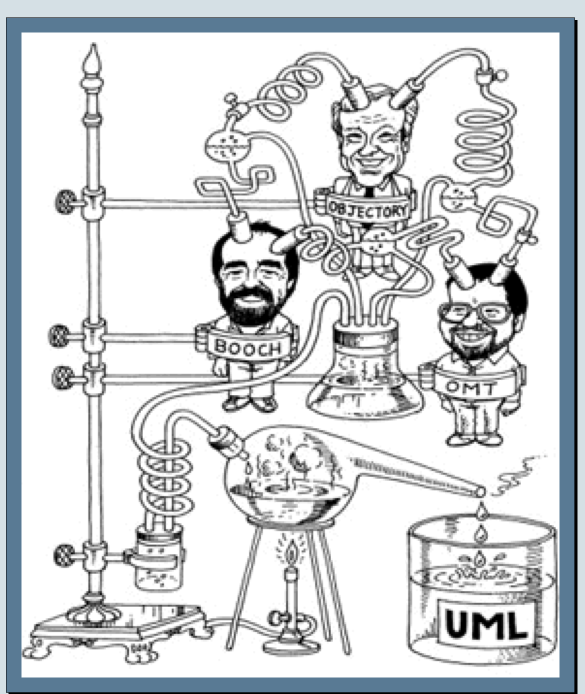
Ivar Jacobson

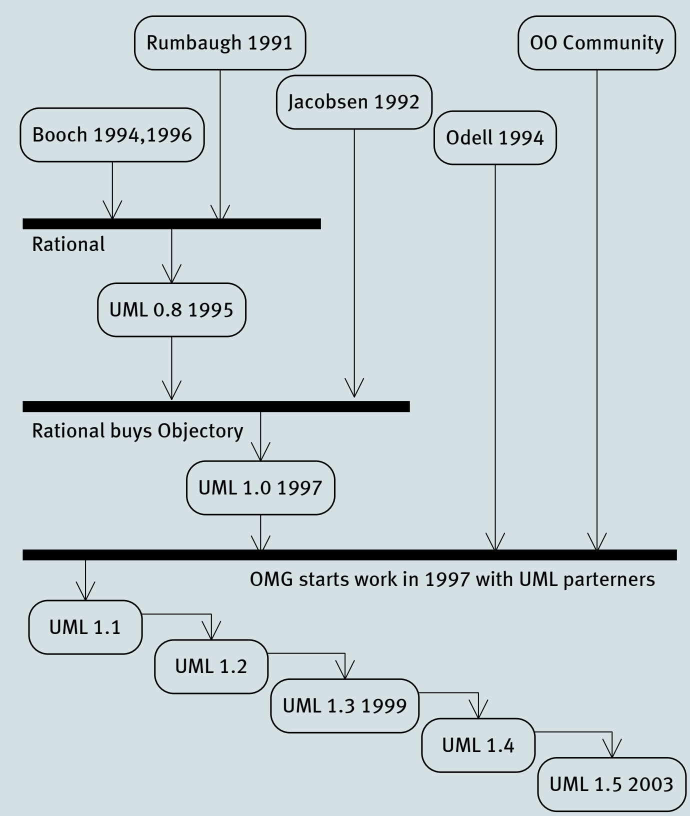
James Rumbaugh

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 ···)

 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)

 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

▸ 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

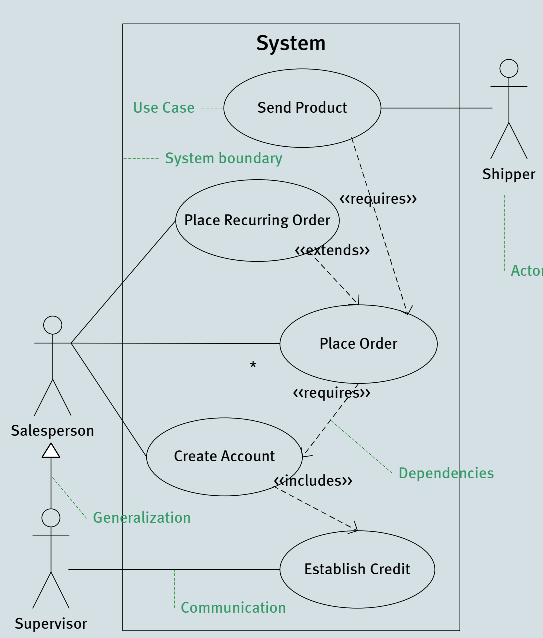
♡ Use Case Diagrams

- 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»



▸ 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*

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

 Example of a use case

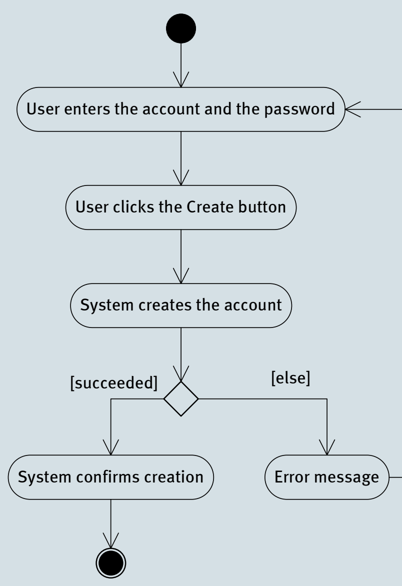
A use case can be specified textually:

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

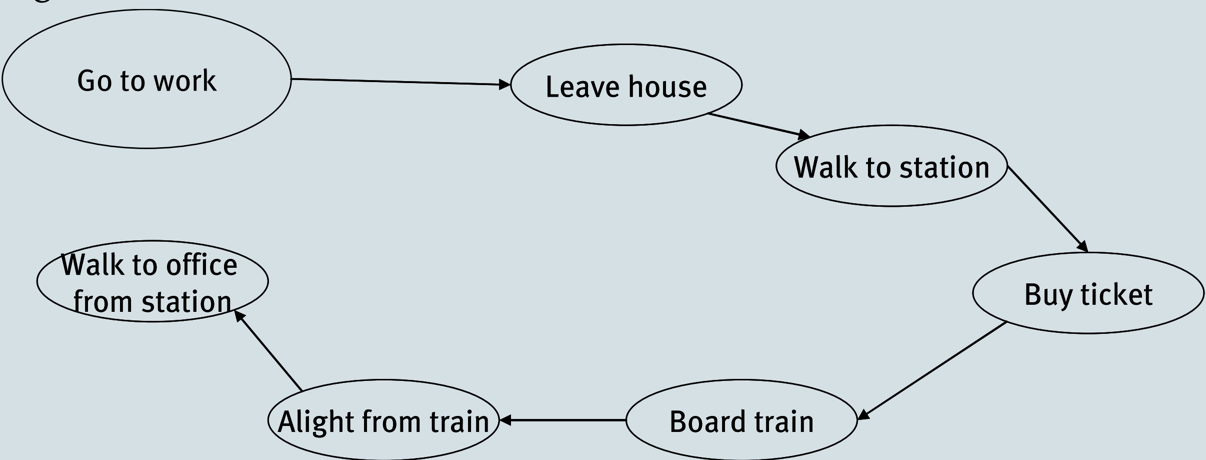
Alternative: Creation Failed
If the system fails at 3, then the system shows a message about the failure and redirects the user back to the step 1

Or using activity diagrams:



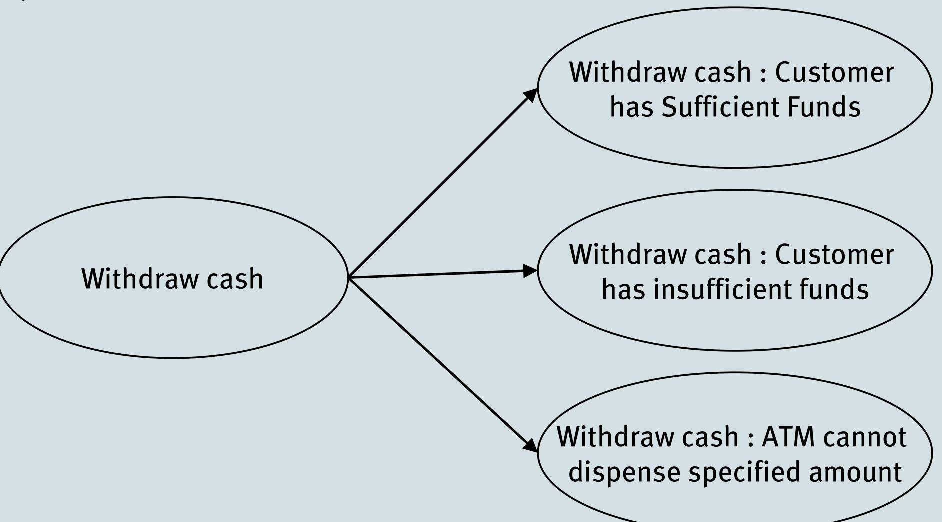
▶ UML 1.x Diagrams ▶ ♡ Use Case Diagrams ▶ Simplifying Complex Use Cases

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



 Simplifying Complex Use Cases

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



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

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

 ♡ Class Diagrams

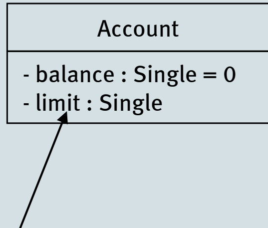
- Classes
- Attributes
- Operations
- Visibility
- Class & Instance Scope
- Bi-directional Associations
- Association names & role defaults
- Multiplicity & Collections
- Aggregation & Composition
- Generalization
- Overriding Operations
- Interface & Realization
- Abstract Classes & Abstract Operations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

▶ UML 1.x Diagrams ▶ ♡ Class Diagrams ▶ Classes

```
class Account {
}
```

▶ UML 1.x Diagrams ▶ ♡ Class Diagrams ▶ **Attributes**

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



[visibility] [/] attribute_name[multiplicity] [: type [= default_value]]

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

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

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

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

  protected int getId(){
    return id;
  }

  int getDatabaseId(){
    return databaseId;
  }
}
```

Account
- balance : float = 0 + limit : float # id : int ~ databaseId : int
+ deposit(amount : single) -withdraw(amount : single) # getAvailableFunds() : single ~ getDatabaseId() : int

+ = public
- = private
= protected
~ = package

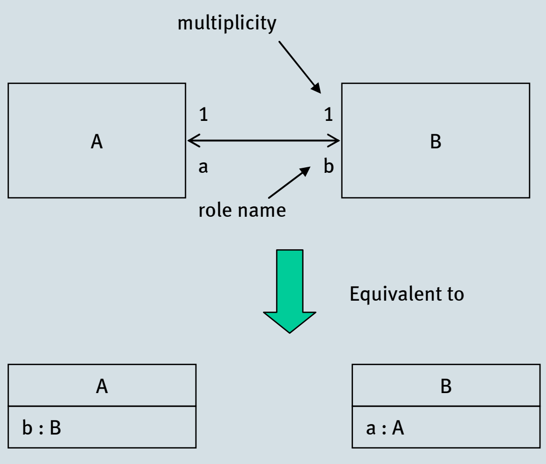
Class & Instance Scope

```
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");
```

Person
- numberOfPeople : int - name : string
+ createPerson(name : string) : Person + getName() : string + getNumberOfPeople() : int - Person(name : string)

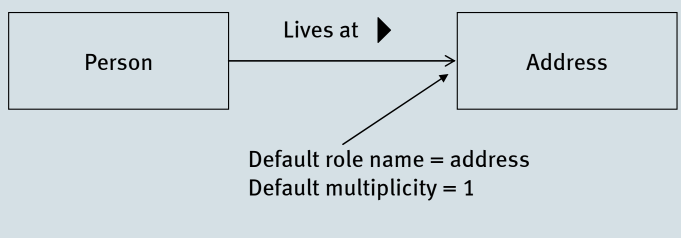
Bi-directional Associations

```
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;
```



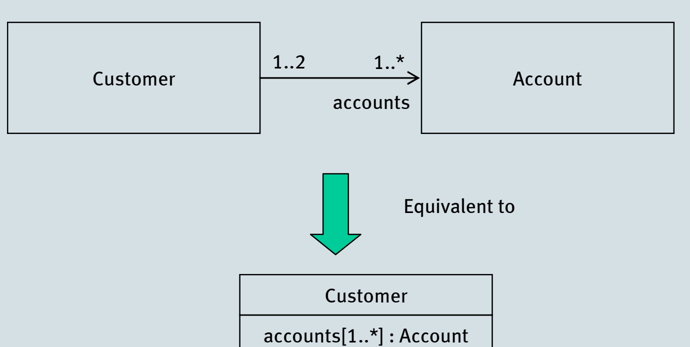
Association names & role defaults

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

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



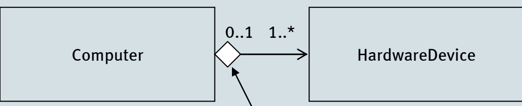
▶ UML 1.x Diagrams ▶ ♡ Class Diagrams ▶ Multiplicity & Collections

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

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

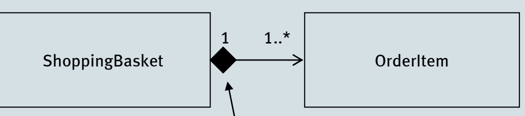


▶ UML 1.x Diagrams ▶ ♡ Class Diagrams ▶ Aggregation & Composition

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

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

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



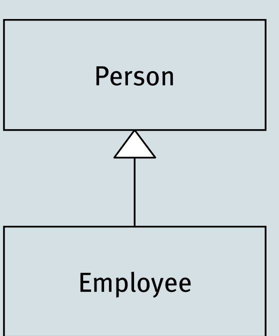
Aggregation – is made up of objects that can be shared or exchanged



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

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

```
class Person{
}

class Employee extends Person{
}
```

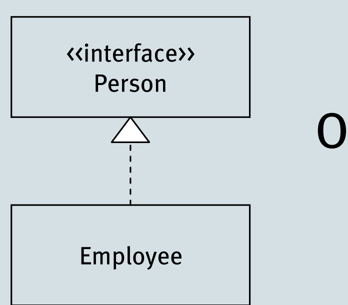


Overriding Operations

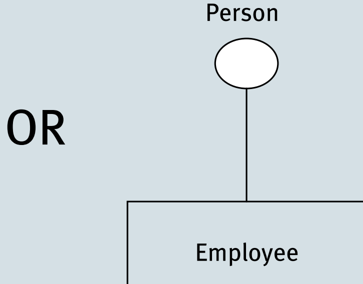
```
class Account
{
  protected float balance = 0;
  protected float limit = 0;

  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{
}
```

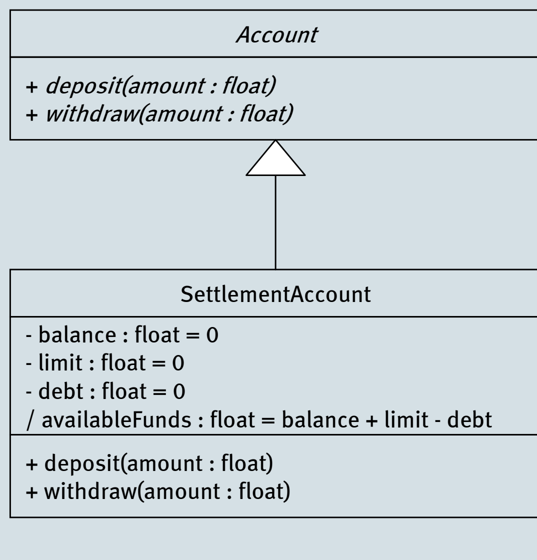


OR

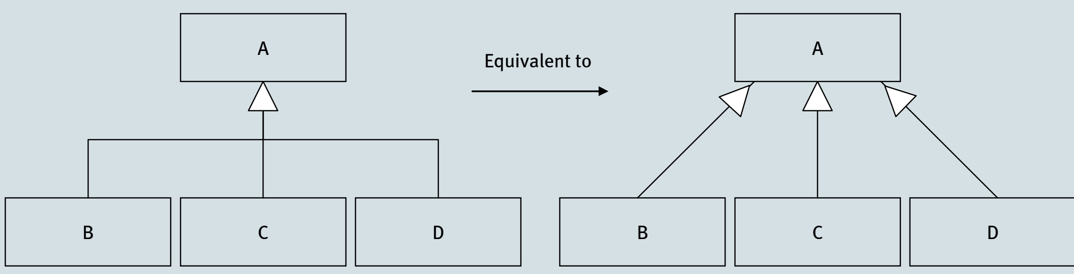


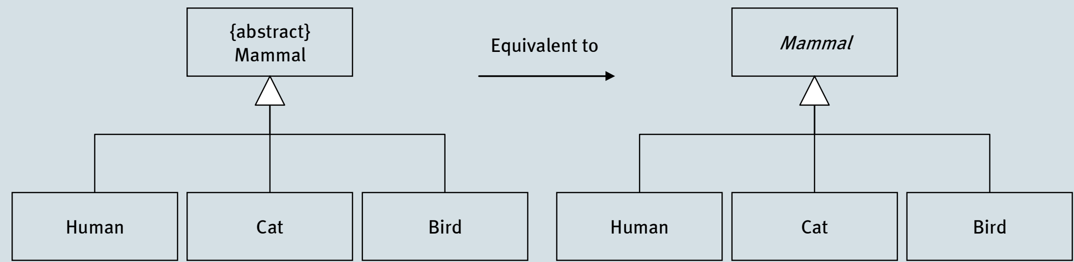
Abstract Classes & Abstract Operations

```
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;
  }
}
```



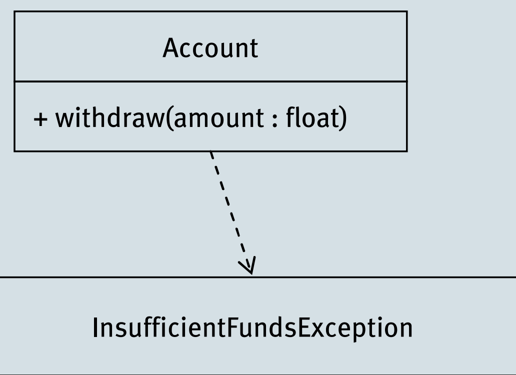
 More on Generalization





▶ UML 1.x Diagrams ▶ ♡ Class Diagrams ▶ Dependencies

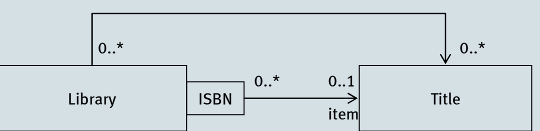
```
public class Account {

  public void withdraw(float amount)
    throws InsufficientFundsException{
  }

}
```



Qualified Associations

hash-table, associative array, "dictionary" …

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

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

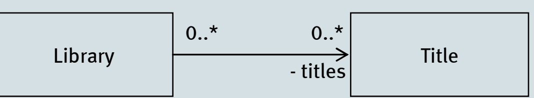


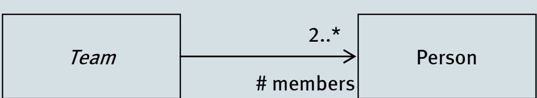
▶ UML 1.x Diagrams ▶ ♡ Class Diagrams ▶ Association Classes

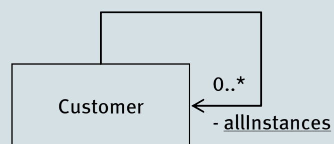
```
class Customer{
    ArrayList rentals = new ArrayList();
}

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;
}
```

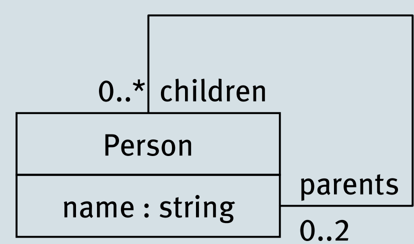






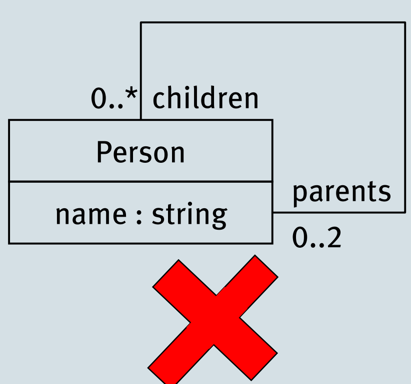
Information Hiding

```
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";
```



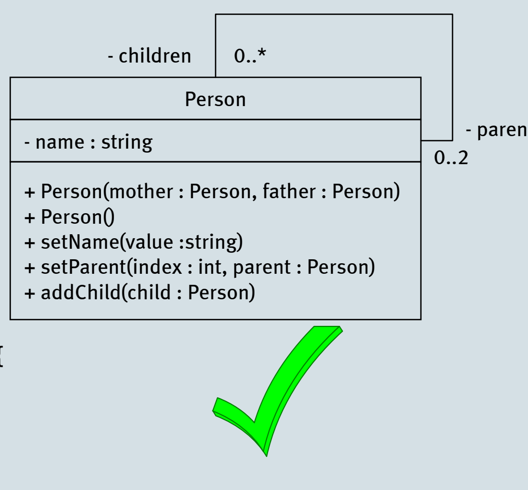
Information Hiding

```
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";
```



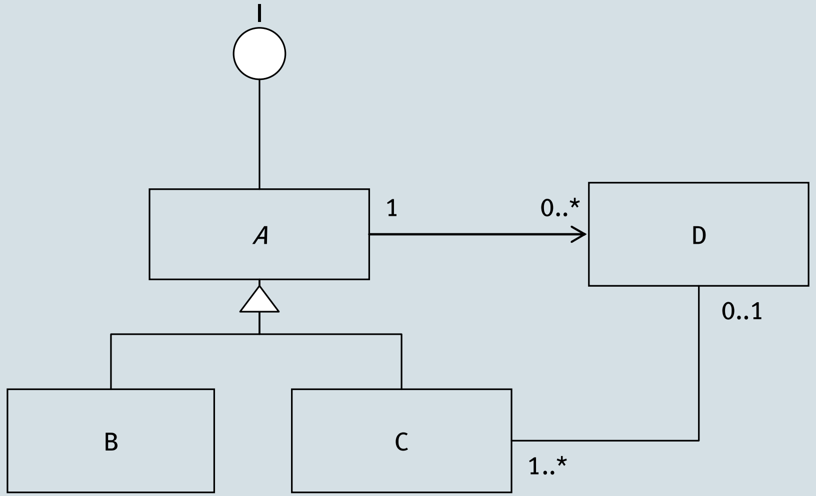
▶ UML 1.x Diagrams ▶ ♡ Class Diagrams ▶ Information Hiding (2)

```
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");
```



 Exercise

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



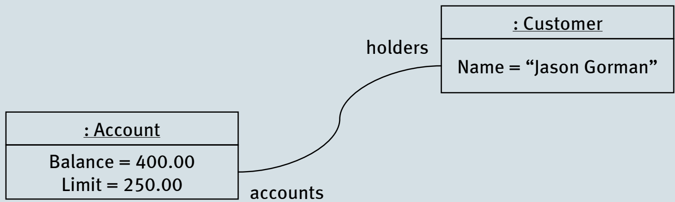
▶ UML 1.x Diagrams ▶ Object Diagrams & Filmstrips

- Instances of Class Diagrams
- Object State
- Filmstrips

 Instances of Class Diagrams

 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.



 Filmstrips

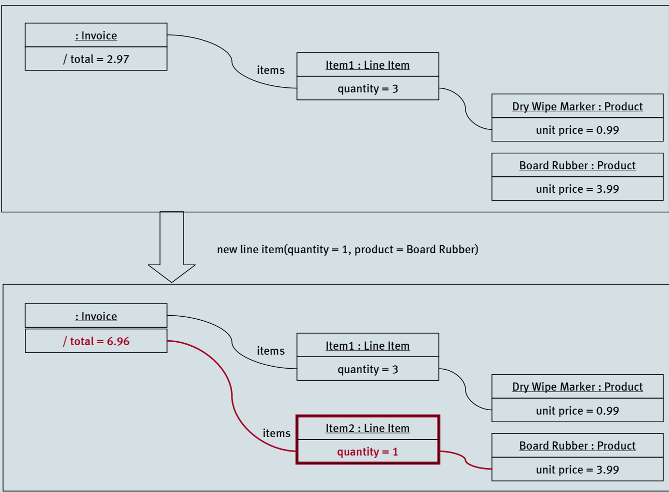
: Invoice
/ total = 2.97
items
Item1 : Line Item
quantity = 3
Dry Wipe Marker : Product
unit price = 0.99
Board Rubber : Product
unit price = 3.99

new line item(quantity = 1, product = Board Rubber)

: Invoice
/ total = 6.96
items
Item1 : Line Item
quantity = 3
Dry Wipe Marker : Product
unit price = 0.99
items
Item2 : Line Item
quantity = 1
Board Rubber : Product
unit price = 3.99

▶ 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



 ♡ Sequence Diagrams

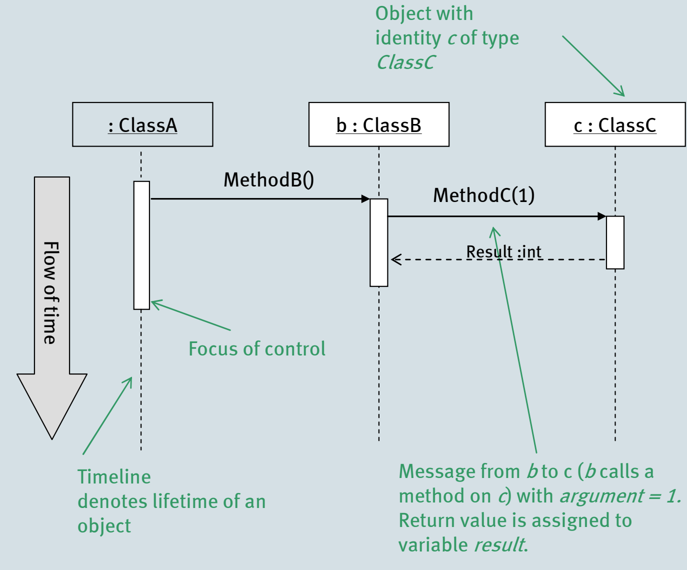
- Why sequence diagrams
- Messages & Timelines
- Object Creation & Destruction
- Collections and Iterations
- Conditional Messages
- Class Operations
- Recursion

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).

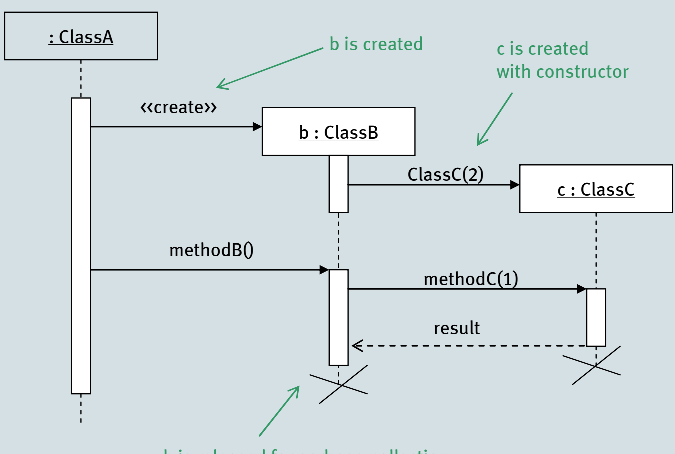
▶ UML 1.x Diagrams ▶ ♡ Sequence Diagrams ▶ Messages & Timelines

```
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);
    }
}

public class ClassC{
    private int factor = 0;
    public ClassC(int factor){
        this.factor = factor;
    }
    public int methodC(int argument){
        return argument * factor;
    }
}
```



▶ UML 1.x Diagrams ▶ ♡ Sequence Diagrams ▶

Collections and Iterations

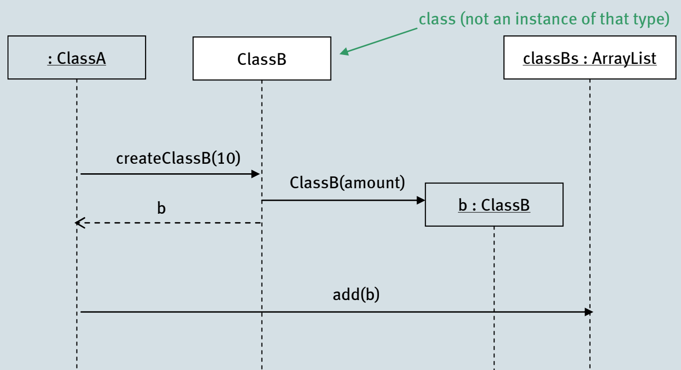
```
public class ClassA
{
  //a collection of ClassB objects
  private ClassB[] classBs
    = new ClassB[]
      { new ClassB(),
        new ClassB(),
        new ClassB()
      };

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

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();
    }
  }
}
```

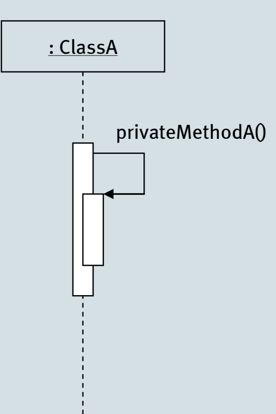
 Class Operations

```
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()  {
  }
}
```



 Collaboration Diagrams

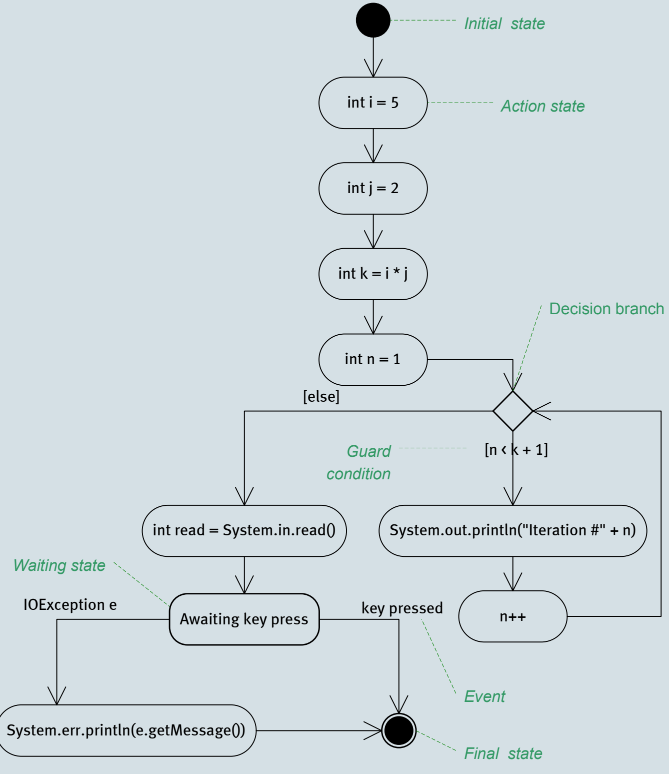
= Communication Diagrams in UML 2.x

♡ Activity Diagrams

- Process Flow
- Concurrency
- Swim lanes
- Signals and Exceptions

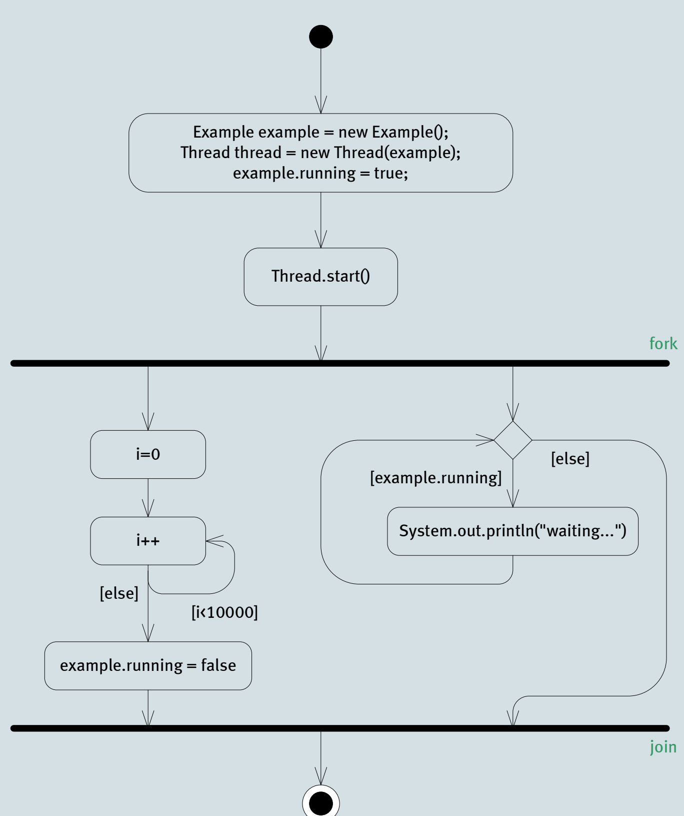
 Process Flow

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

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

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



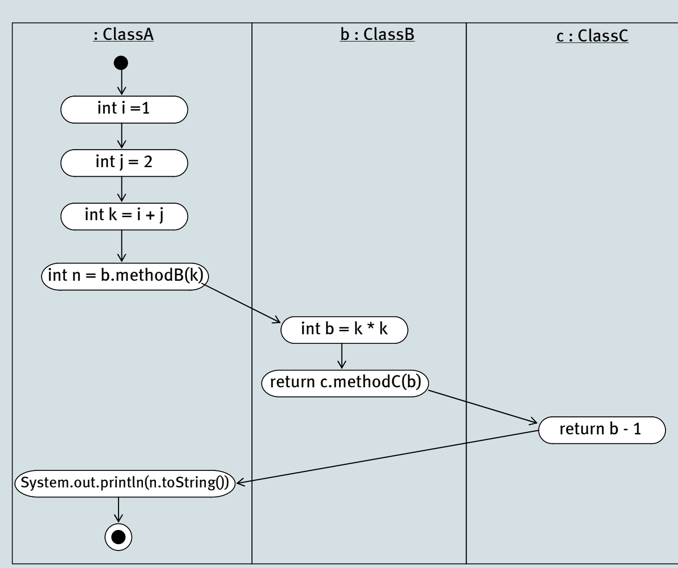
▶ UML 1.x Diagrams ▶ ♡ Activity Diagrams ▶ Concurrency

```
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;
  }
}
```



 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;
  }
}
```

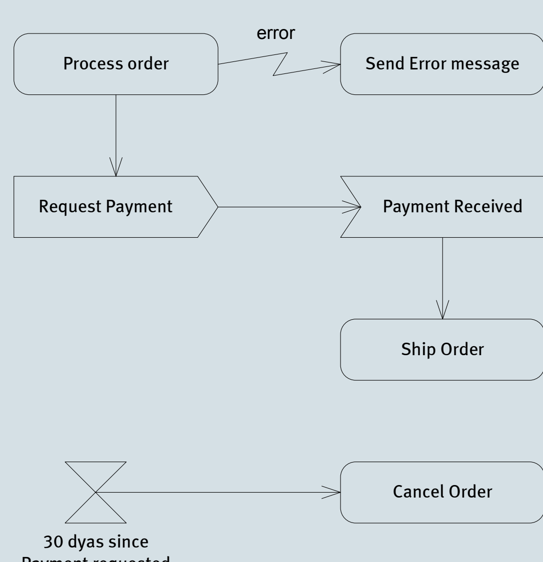


 Signals and Exceptions

- 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."



State Diagrams

= State Machine Diagrams in UML 2.0

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

State transitions

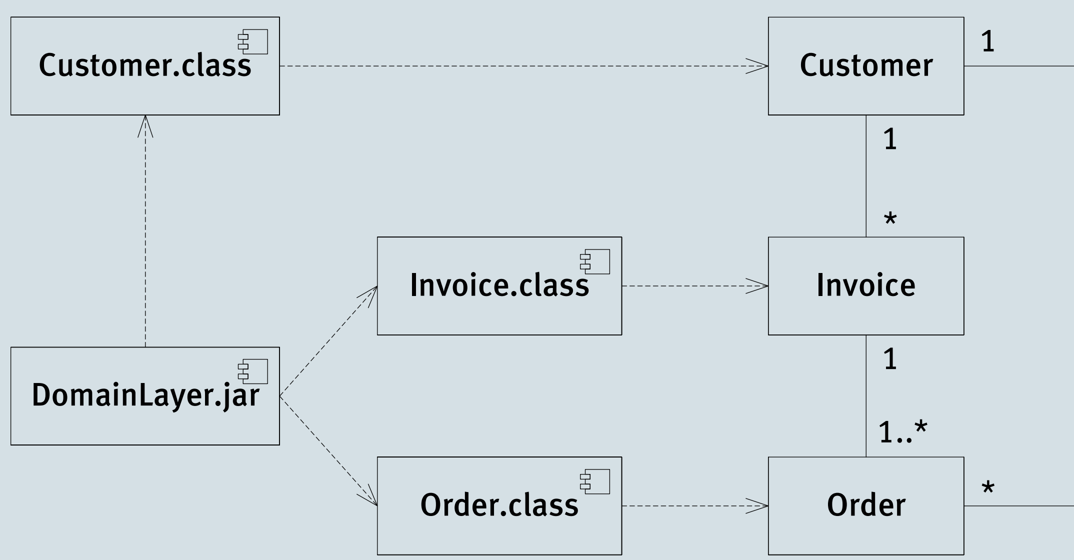
 Transitions and Actions

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

 Component Diagrams

- Components and Dependencies
- Components Can Contain Components

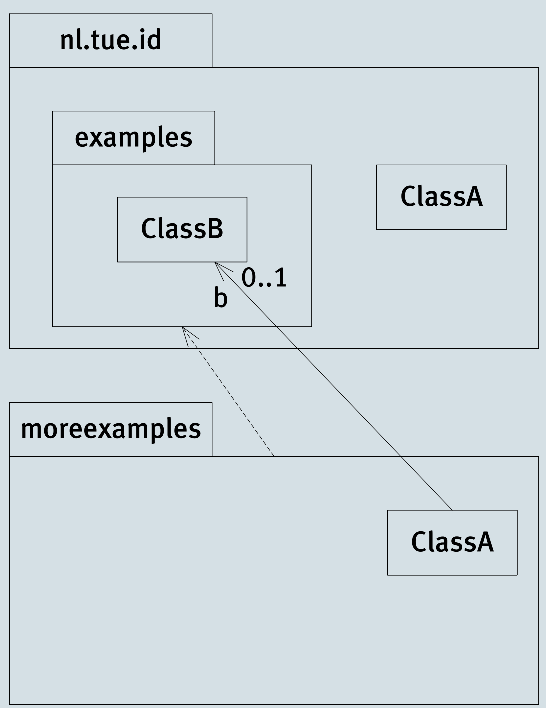
 Components and Dependencies



 Components Can Contain Components

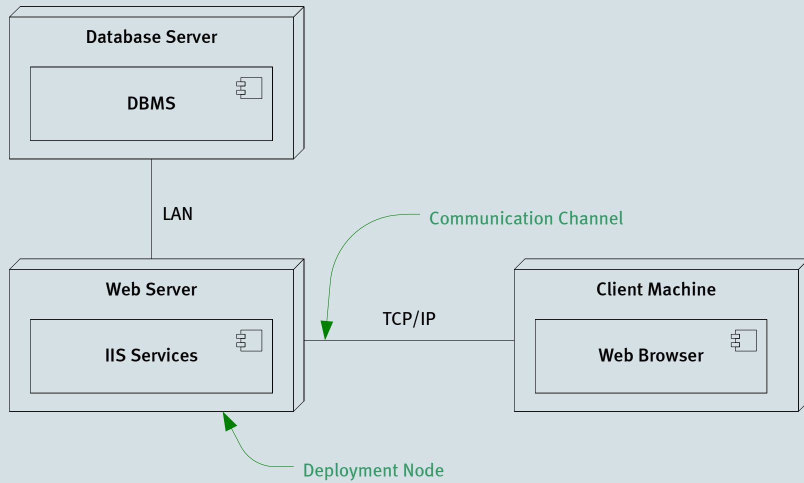
▶ UML 1.x Diagrams ▶ Package Diagrams

```
/*===================================*/
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;
}
```



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

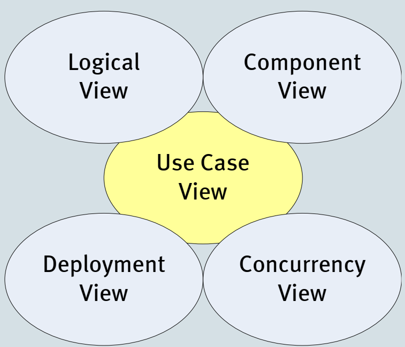
- Use Case View
- Logical View
- Component View
- Concurrency View
- Deployment View

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

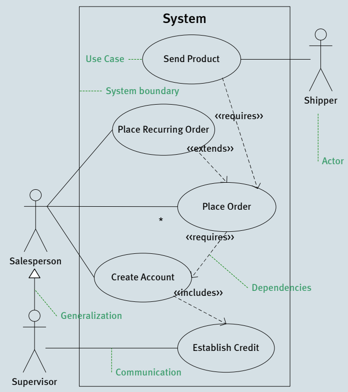
▸ UML Views (2)

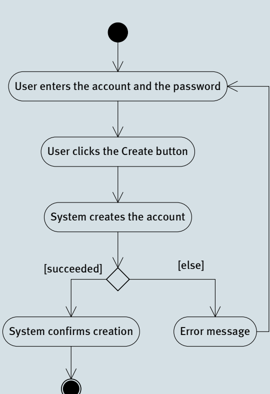
- Each view is a projection of the complete system
- Each view highlights particular aspects of the system
- Views are described by a number of diagrams
- No strict separation, so a diagram can be part of more than one view



▶ UML Views ▶ Use Case View

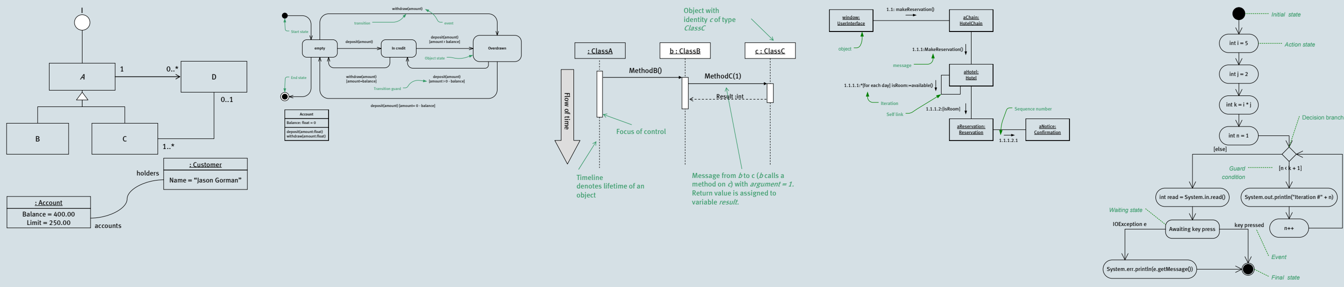
- Shows the functionality of the system as perceived by external actors.
- Actors can be users or other systems.
- Described by use case and activity diagrams.
- The central view which drives the development of other views.
- Used by customers, designers, developers, testers.





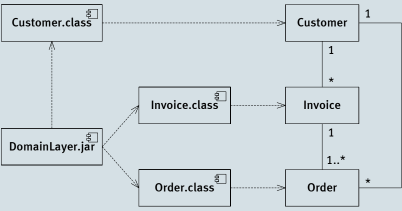
▶ UML Views ▶ **Logical View**

- 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.



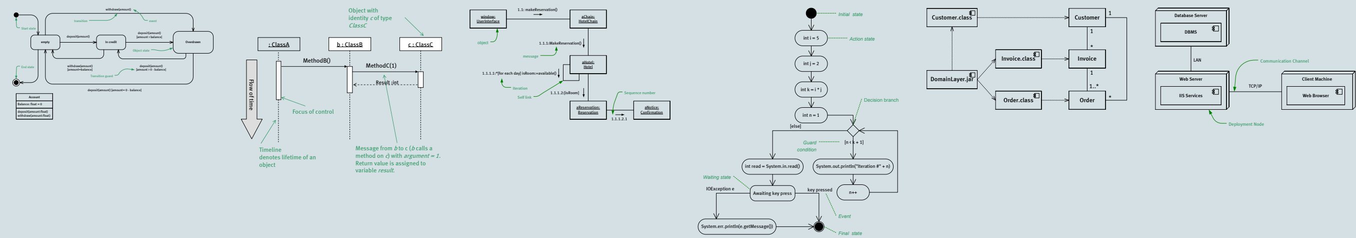
▶ UML Views ▶ Component View

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



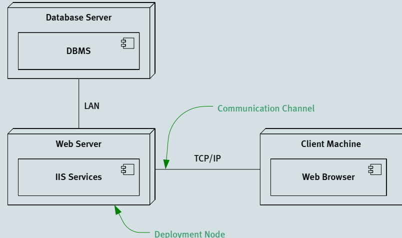
▶ UML Views ▶ Concurrency View

- 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.



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:

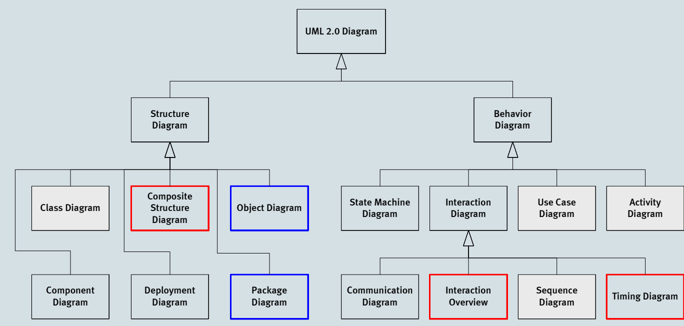
- User case diagrams
- Class diagrams
- Sequence diagrams
- Activity diagrams

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®

  ⋮

  ⋮

▸ 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 …

▸ 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!