Michael Weintraub
And
Frank Tip

SYSTEM AND SOFTWARE DESIGN USING THE UNIFIED MODELING LANGUAGE (UML)

Northeastern University
College of Computer and Information Science
440 Huntington Avenue . 202 West Village H . Boston, MA 02115 . T 617.373.2462 . ccis.northeastern.edu

UML Part 2

DEFINING STATIC MODELS

THE STRUCTURAL PERSPECTIVE

The internal organization of the system

Entities

and

Relationships between Entities

ENTITIES (AKA OBJECTS AKA CLASSES)

- A thing with distinct and independent existence

  (Oxford Dictionaries · © Oxford University Press)

  - *Origin: Late 15th century (denoting a thing's existence): from French entité or medieval Latin entitas, from late Latin ens, ent- being (from esse be).*

ENTITIES (AKA OBJECTS AKA CLASSES)

- A thing with distinct and independent existence

  (Oxford Dictionaries · © Oxford University Press)

  - *Origin: Late 15th century (denoting a thing's existence): from French entité or medieval Latin entitas, from late Latin ens, ent- being (from esse be).*

- In UML (and CS in general), an entity/object/class is a combination of:
  1. *identity (unique name)*
  2. *state (data)*
  3. *behavior (services)*

WHAT'S AN OBJECT?

An object offers

a collection of services (methods) that work on a common state.

There is usually a correspondence between

objects and nouns in the task

("Bug", "Field", "Marker")

methods and verbs in the task

("move", "sit down", "delete")

THREE PERSPECTIVES ON CLASSES

1. Conceptual (domain analysis)
   - Shows concepts of the domain
   - Implementation-independent

2. Specification (design)
   - General structure of the system
   - Interfaces (types, not classes)
   - Used in high-level design

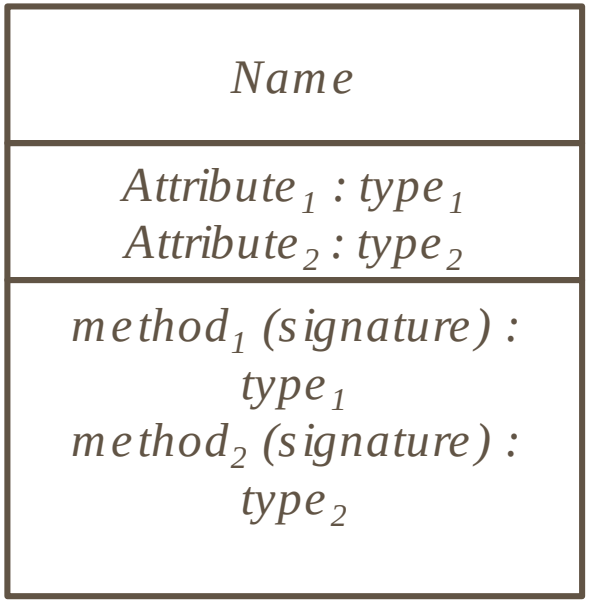
3. Implementation (programming)
   - Structure of the implementation
   - Most often used

Always try to draw from a single perspective!

CLASSES IN UML

- Classes are drawn as a three-part box containing:
  1. class name
  2. list of attributes with names and types
  3. list of methods with argument list

  - Name is required
  - Attributes and methods are not



(there is a fourth, optional box, the *extra compartment,* for capturing responsibilities, signals, or events)

DESCRIBING CLASSES

Classes must have a name.

- Class names should be chosen carefully. The names should reflect the abstractions of the problem domain to maximize clarity:
  1. *use a term accepted in the business domain*
  2. *use a name that is most descriptive (perhaps Customer instead of Shopper)*

Names typically formed using a singular noun or an adjective plus a noun (Customer, UsedCar)

Every class must be defined with a description.

- The description can be a single sentence or a short paragraph explaining *why this class is important to the business or system*.
- An class should have a relationship with a least one other class.
- The list of classes and descriptions, along with attributes, forms a data dictionary.

TIPS FOR IDENTIFYING CLASSES

- During requirements elicitation, listen for when stakeholders discuss:
  1. tangible objects (printer, customer)
  2. conceptual entities (time, job)
  3. roles (student, instructor, voter)
  4. specifications (recipe, plan)
  5. incidents (accident, delivery)
  6. organizations (department, team)

Best Practice

- When you think you found an class, ask the stakeholder to:
  - name at least two occurrences of the class
  - if they can't, then it's not an entity, it is an instance
  - Example: *FedEx* is not an entity, it is an instance of the entity *Shipper*

ATTRIBUTES

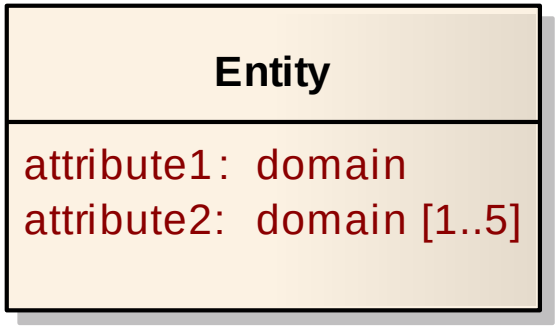
A quality or feature regarded as a characteristic or inherent part of someone

Oxford Dictionaries · © Oxford University Press

A Person usually has the following attributes:

- *name*
- *age*
- *height*
- *weight*

CLASS ATTRIBUTES IN UML



attribute1 is simple.

attribute2 is multivalued (there can be up to five values stored on attribute2)

domain is UML-speak for TYPE

Attributes may be:

1. single, non-decomposable value (simple)
2. more than a single value (multivalued)

   Think list or array of values
3. value that can be calculated from other attributes (derived)

   E.g. ***age*** *is derived from* ***birthdate*** *and* ***today's date***
4. value that is itself an entity with attributes (composite)

   These usually end up becoming entities themselves.

Relationships to other objects are not listed among attributes.

Don't get confused because languages implement relationships as attributes.

COMMON UML DOMAINS (TYPES)

Logical Data Type	Business Meaning
NUMBER	*Any number: real or integer.*
TEXT	*A string of characters, inclusive of numbers. (aka `string`)*
MEMO	*Same as TEXT, but of large, indeterminable size. (aka `blob`)*
DATE	*Any date value in any format.*
TIME	*Any time value in any format.*
BOOLEAN	*A yes/no or true/false value.*
VALUE SET	*A finite set of categorical values with an associated coding scheme. (aka `enumeration`)*
IMAGE	*Any picture of image.*

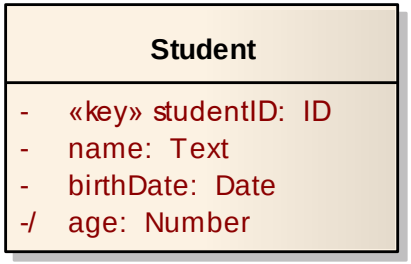
EXAMPLE DATA DICTIONARY

Name	Description	Type	Required?	Default Value	Format	Range	Unique?
Customer	*A person purchasing items in the store.*	*Entity*					
.Name	*Full name*	*TEXT*	*Yes*	*N/A*	*Last Name, First Name*	*1..32*	*No*
.SSN	*Social Security Number*	*TEXT*	*Yes*	*N/A*	*XXX-XX-XXXX*	*11*	*Yes*

UNIQUE IDENTIFIERS

- A *unique identifier* is an attribute or set of attributes that uniquely identifies an class instance.
- Identifier attributes have the stereotype «key»:

- Defining unique identifiers is important for searching.



SECTION II:
BUILDING OBJECT MODELS

MODELING THE RELATIONSHIPS BETWEEN THE ENTITIES

- Two relationship types

  1. Association
  2. Generalization

- Some analysts also model *aggregation,* but often *associations* are sufficient
  - aggregation is typically used for whole-part relationships, but those can also be modeled using association

ASSOCIATION RELATIONSHIP

- An association is a simple semantic relationship between two objects that indicates a link or dependency between them.

- Examples:
  1. a ***portfolio*** is associated with an ***investor***
  2. every ***sale*** is associated with the ***sales representatives*** that worked on the sale
  3. every ***student*** is associated with a ***transcript***

- Associations can be directed, meaning there is a relationship from one object to another, or bi-directional, meaning the relationship works both ways.

- Relationships may be annotated with descriptions.

HEURISTICS FOR IDENTIFYING ASSOCIATIONS DURING REQUIREMENTS GATHERING

- Look through the use cases and identify multiple nouns that appear in the same sentence.
- Listen for connecting verbs during requirements elicitation.

CARDINALITY (AKA MULTIPLICITY)

- Cardinality: the number of elements in a set
- The relationship between two entities has an associated *cardinality* or *multiplicity*
  - multiplicity is expressed with specific numbers or ranges, e.g. 1:1..2 or 1:1..N
- Examples:
  1. A student is associated with a transcript

     One student, one transcript.

  2. Every course is taught by a professor, but a professor must teach at least one course

     One course, one professor. One professor, one or more courses.

  3. An address may have a zip code

     One address, zero or one zip code

DEFINING CARDINALITY IN UML

Any given instructor teaches 1 course.
Any given course is associated with one instructor.

Any given instructor teaches at least 1 and up to 10 courses.
Any given course is associated with one instructor.

Any given instructor teaches 1 or more courses.
Any given course is associated with one instructor.

No cardinality defaults to 1.

OPTIONAL RELATIONSHIPS

- Participation in a relationship may be mandatory or optional.
- A lower bound of 0 indicates an optional relationship.

Any given instructor teaches 0 or 1 course.
Any given course is associated with one instructor.

An instructor may not be teaching a course,
but every course has an instructor.

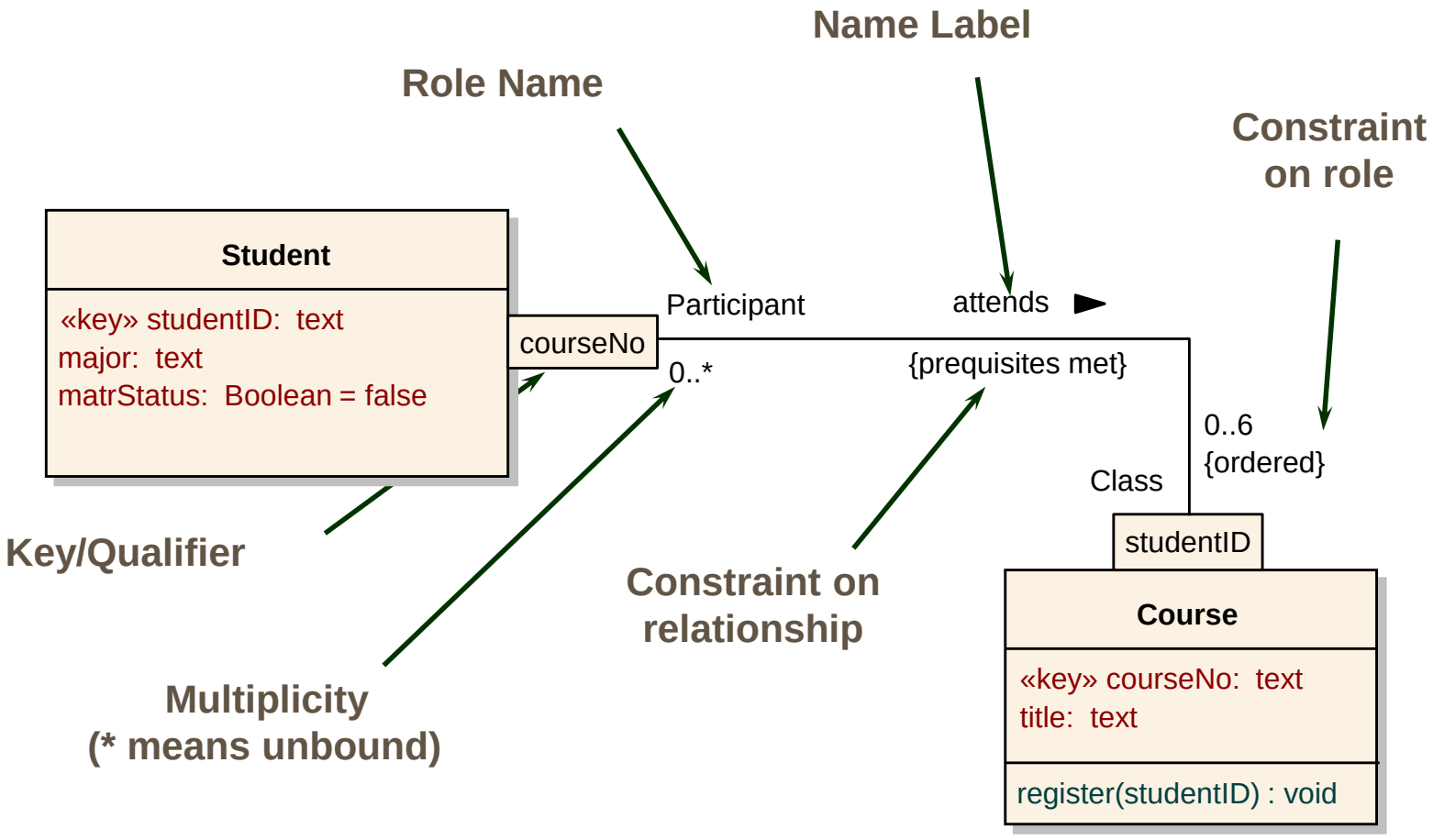
Any given instructor teaches 0 or 1 course.
Any given course is associated with 0 or one instructor.

An instructor may not be teaching a course.
A course may not have an instructor.

Any given instructor teaches any number of courses.
Any given course is associated with 0 or 1 instructor.

An instructor may teach any number of courses.
If a course has an instructor, it has only one.

FULL ASSOCIATION SPECIFICATION

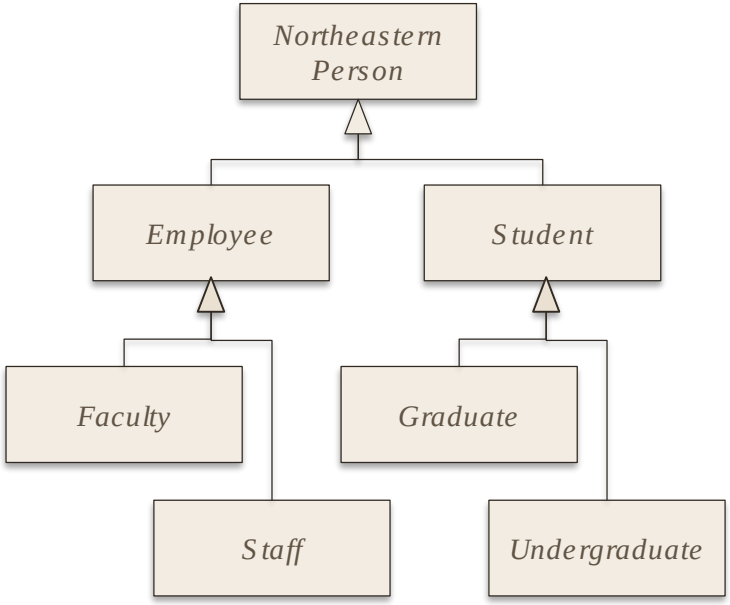


GENERALIZATION HIERARCHIES

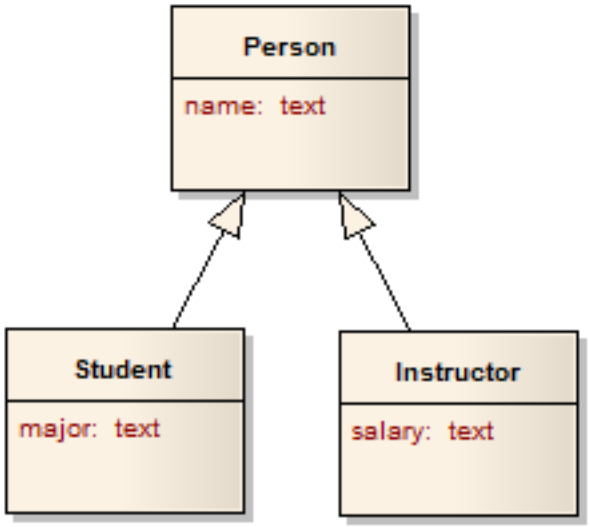
- Generalization is a grouping of entities based on common attributes.
  - Is a taxonomy that identifies similarities and differences between entities.
  - Defines an *is-a* relationship between entities.
- Called inheritance in object-oriented programming.

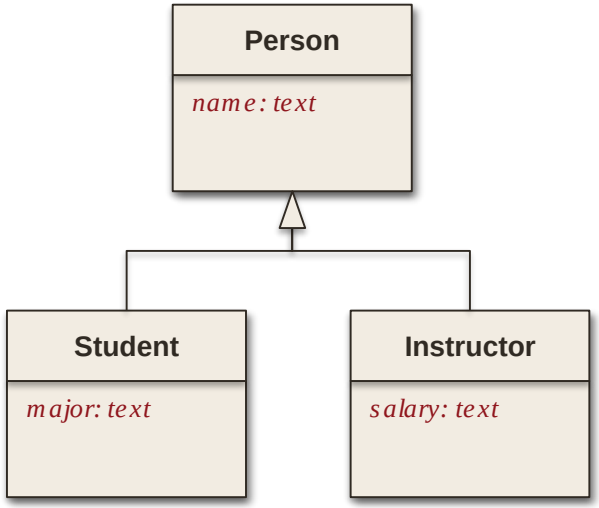
GENERALIZATION

- More generic as you move up
- More specific as you move down
- More specific inherits attributes and operations from the more general
  - May specialize attributes and operations



GENERALIZATION IN UML





These are the same

WHEN TO USE GENERALIZATION

- Generalization is a modeling technique in which attributes common to several entities are grouped into a separate ***supertype***.

- Any attributes specific to one entity remain in the ***subtype*** entity.

- Example:

  Student has attributes: *NU ID, birthday, first day on campus,* and *transcript.*
  Instructor has attributes: *NU ID, birthday, first day on campus,* and *salary.*

  NUPerson generalizes ***Student*** and ***Instructor***
  with attributes: *NU ID, birthday,* and *first day on campus.*

SUBTYPE JUSTIFICATION

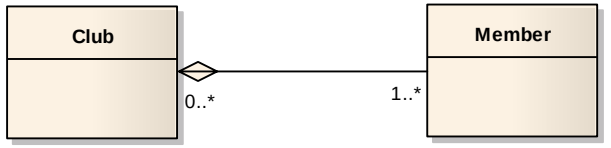
- Each subtype entity must have one or more of the following properties:
  - at least one non-key attribute
  - a relationship with another entity that is logically correct for it and none other

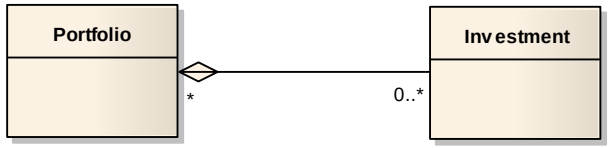
- If this is not true, then you have found a *transient role*, not a subtype.

- Transient role means the object's role changes over time. However, the object isn't changing.
  - For example, a ***Student*** may become a *Freshman, Sophomore, Junior,* or *Senior* over one's career as a student.

AGGREGATION

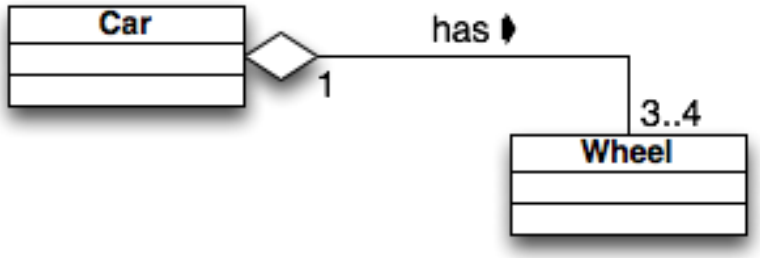
- Aggregation is an association that means a "**whole/part**" or "**containment"** relationship.
  - Usually is a hierarchical structure of objects.
- The aggregate contains attributes which are derived from its parts.
  - The aggregate/composite must in some way depend functionally or structurally on its components.
  - The relationship is anti-symmetric and transitive.
- Aggregation is a somewhat vague concept that has not been defined well in the object-oriented paradigm.
  - In general, aggregation defines an association where one class is part of another class.
- If in doubt, make the relationship a simple association.





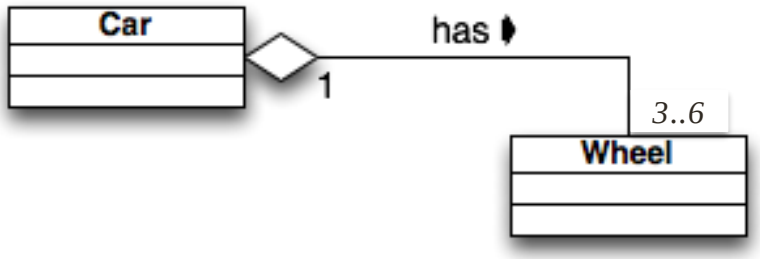
AGGREGATION EXAMPLE

A car has 3–4 wheels



AGGREGATION EXAMPLE

A car has ~~3–4 wheels~~ 3-6 wheels



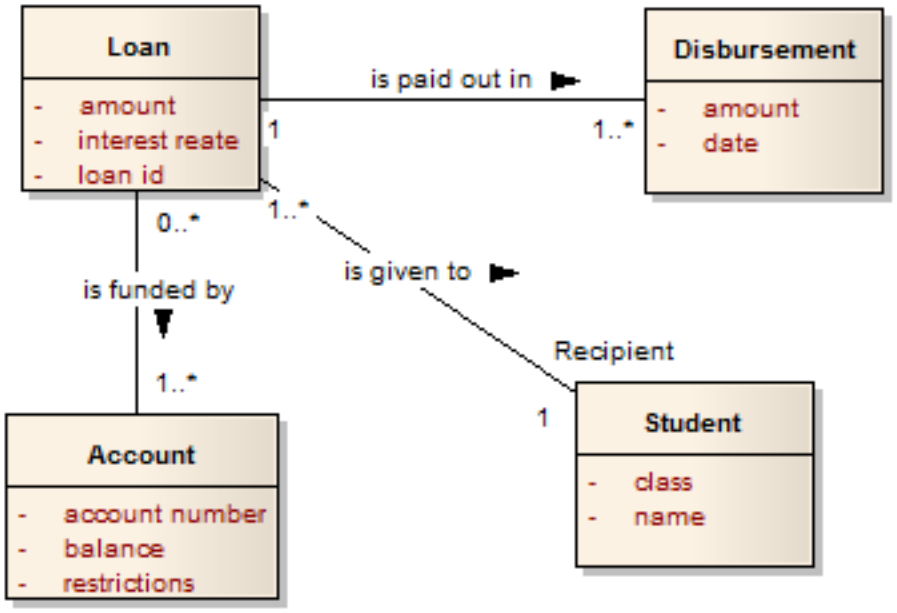
Sometimes models have to be adjusted in the face of new realities…

VALIDATING DATA MODELS

- The requirements and rules captured in a data model must be validated.

- Validation is generally accomplished through stakeholder reviews.
  1. Reviews should be ideally conducted as meetings.
  2. During reviews, the diagrams should be “translated into narratives.”

TRANSLATING DIAGRAMS TO NARRATIVES



Partial Translation:

We have discovered that a loan can be paid out in multiple disbursements. There does not appear to be any limit to the number of disbursements. In addition, each loan is given to a single student. Apparently, students cannot share loans.

- Translate each relationship to a fact, *i.e.*, a statement that can be confirmed or disputed.

MOVING FROM MODELS TO CODE

USING JAVA

FROM MODEL TO PROGRAM

- How does one transform a design into code?
- Classes and hierarchies can be taken directly from the class diagram.
- For each method a complete signature has to be provided.

FROM MODEL TO PROGRAM

- Associations between classes are implemented using attributes (fields)
  - n:1 and 1:1 associations from P to Q are implemented as
    an attribute q of type Q in P.
  - 1:n and n:m associations from P to Q are implemented as
    a set qs of type set(Q). (e.g. array, list...)

FROM MODEL TO PROGRAM

- Methods belonging to associations have to be implemented in extra (helper) classes.
- For each class an invariant has to be formulated and documented; for each method a pre- and postcondition.
- The method bodies have to be implemented using techniques from traditional programming.

FROM MODEL TO PROGRAM

- Verification of the state chart:

  are the only legal call sequences exactly those documented in the dynamic model?

- Illegal calls have to be intercepted using exceptions/errors!
  - For that it is often useful to dynamically check the precondition.

- Testing is done using conventional methods.

MODEL-DRIVEN ENGINEERING

- Modern programming environments automatically create code templates from a model:
  1. You design a system with all its classes and attributes.
  2. The programming environment creates the corresponding code templates.
  3. Now you "only" have to add implementations in the method bodies.

PACKAGES

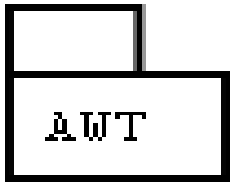
- packages are a general-purpose mechanism for organizing elements into groups.
  - not necessarily restricted to classes
  - hierarchical model: subpackages
  - anonymous “root” package contains all top-level packages

- a package provides a namespace
  - names of package elements are qualified using “::” (C++ style)

PACKAGES (2)

- When to use packages?
  - use packages for high-level design or architecture documents to describe a system's overall structure
  - use packages when class diagrams become too large or cluttered
  - also convenient units for testing

- Notation: tabbed folder



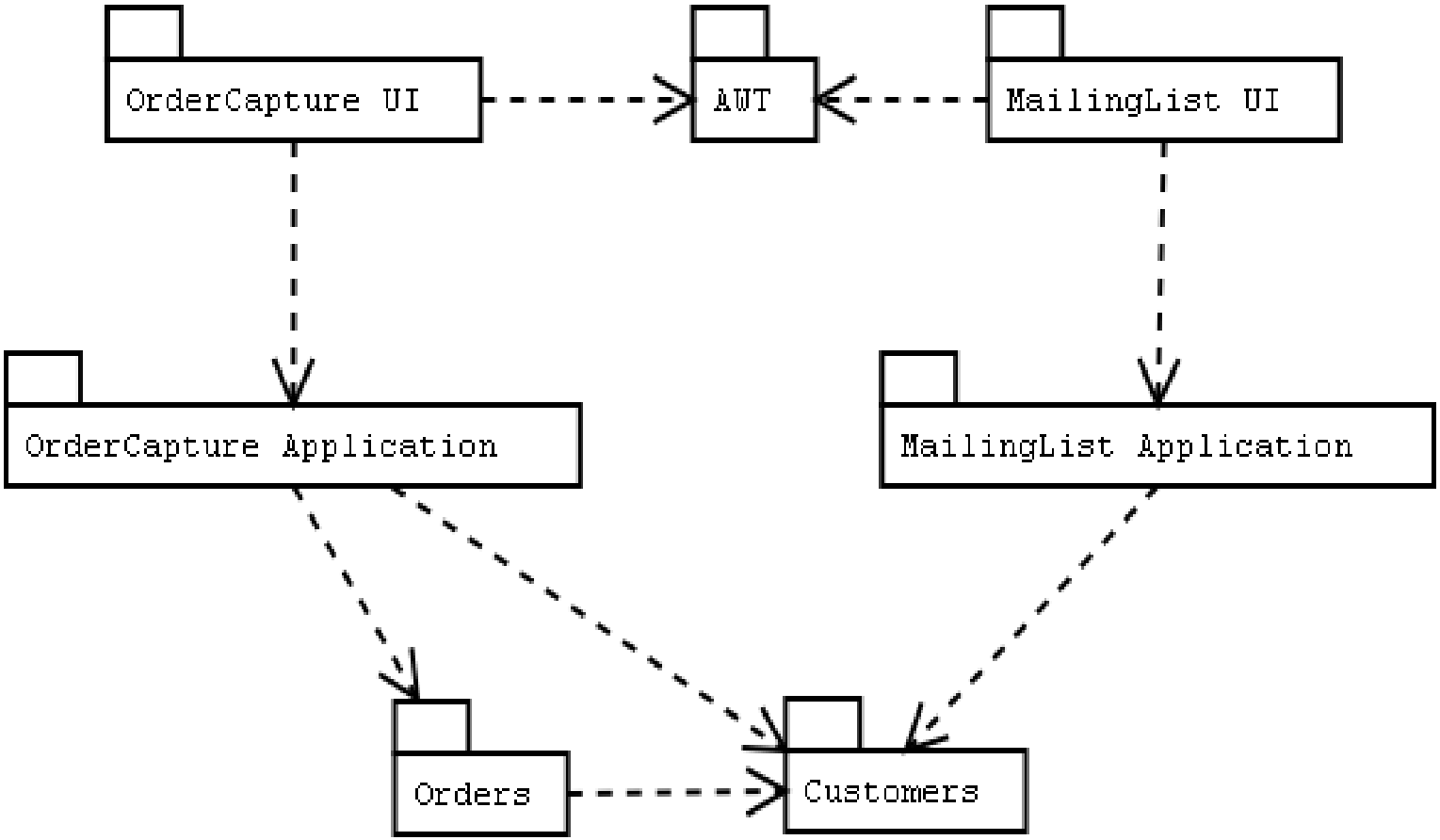
PACKAGE DIAGRAMS

- dependencies between packages:
  - if changes to one package may cause changes to the other
  - reflect dependencies between classes in the packages
    - *class in package A calls method in package B*
    - *class in package A has field of type in package B*
    - *method in package A has parameter/return type of package B*
  - import relationships (also: access relationships)
  - use dashed arrows for dependencies, optionally annotated with <<import>> stereotype

- design consideration: minimize dependencies between packages, especially cycles

- note: dependency between packages is not transitive

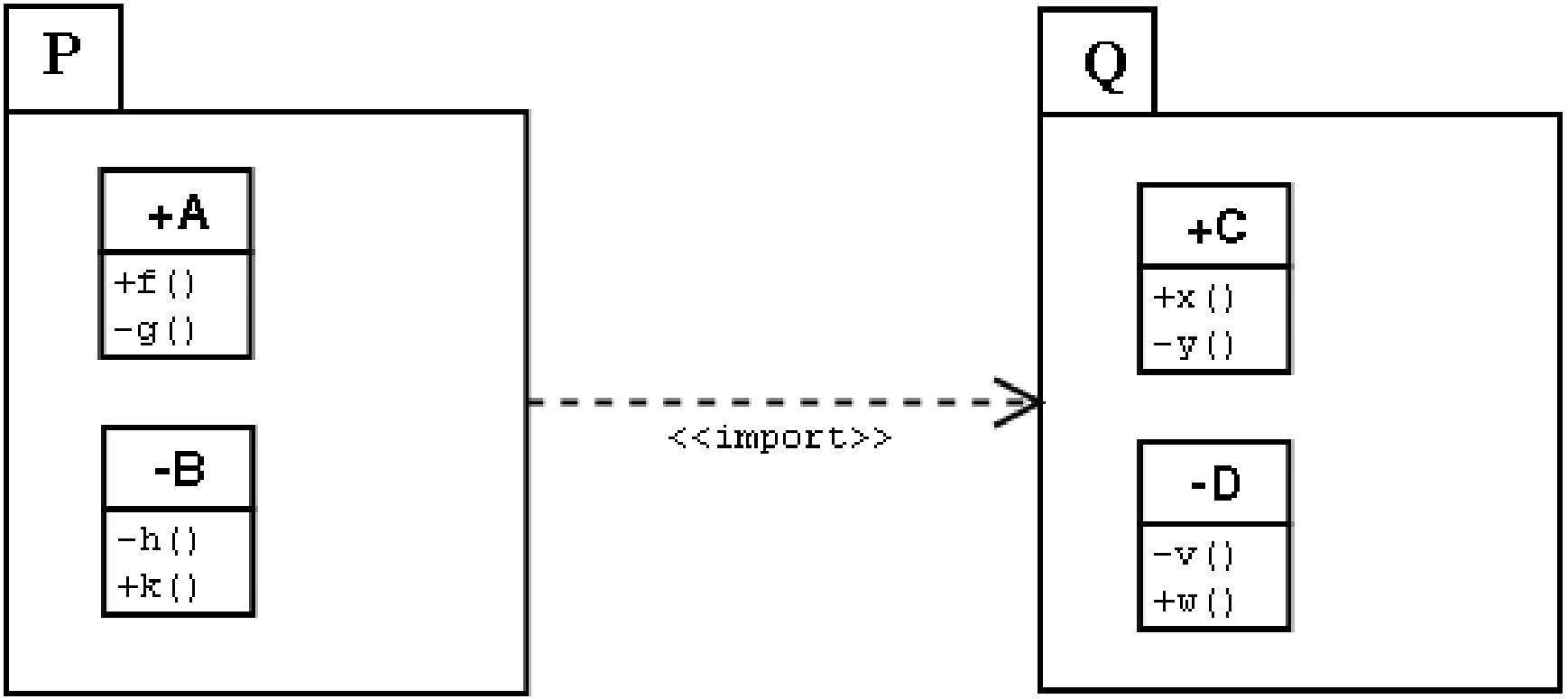
PACKAGE DIAGRAM: EXAMPLE



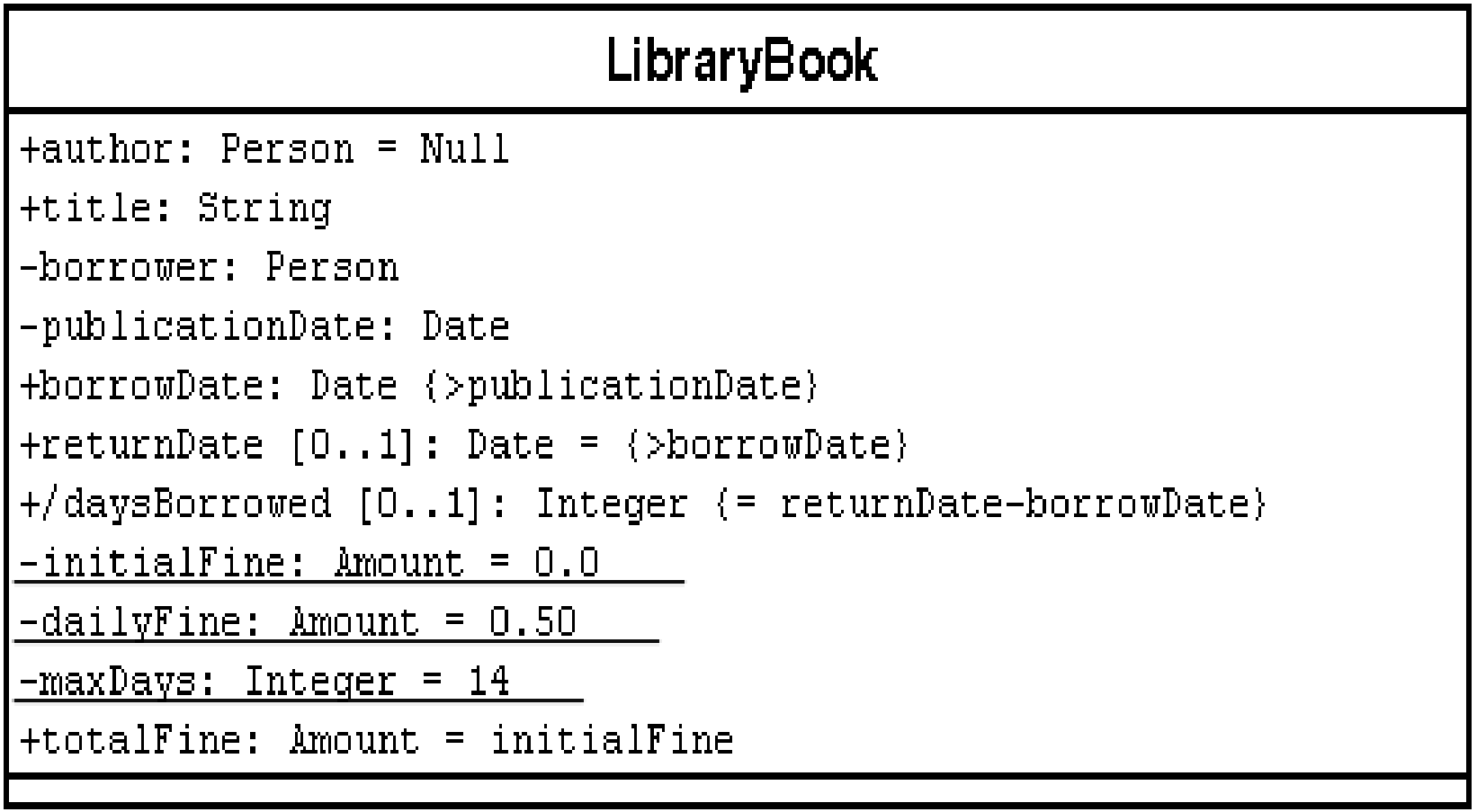
Example taken from “UML Distilled” by Fowler& Scott

PACKAGES: VISIBILITY CONTROLS

- visibility controls at the package level
  - + public, - private
- the public parts of a package are called its exports



FIRST EXAMPLE:
A LIBRARY BOOK AS THE OBJECT WITH ATTRIBUTES



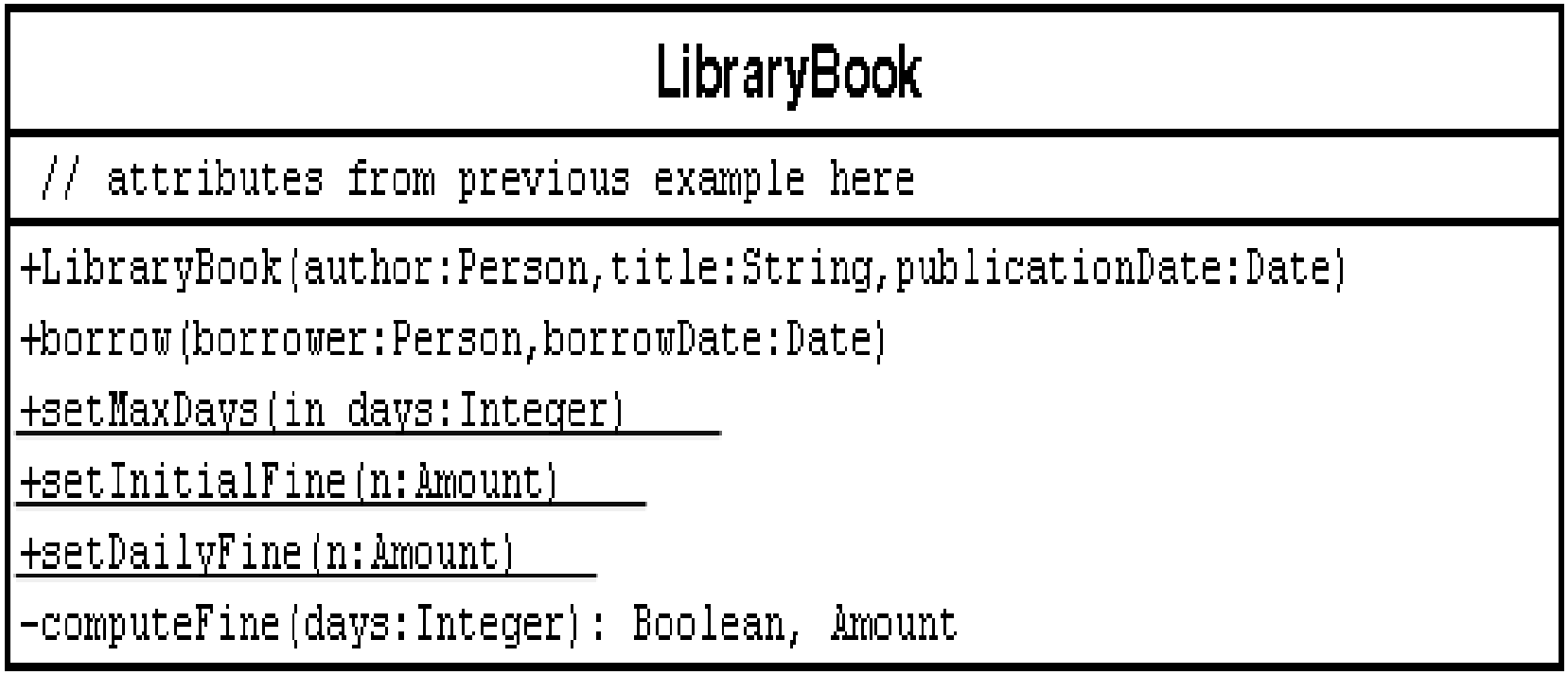
CORRESPONDING JAVA CODE

```
public class LibraryBook {

  // implement attributes
  public Person author = null;
  public String title;
  private Person _borrower;
  private Date _publicationDate;
  private Date _borrowDate;
  private Date _returnDate;

  // note: daysBorrowed not implemented as a field,
  // because it is derived information, namely
  // computed from (returnDate – borrowDate)

  private static double initialFine = 0.0;
  private static double dailyFine = 0.50;
  private static int _maxDays = 14;
  private double totalFine = initialFine;
```

IMPLIED GET/SET METHODS

```
// get/set methods for private fields

  public Person getBorrower(){ return _borrower; }
  public void setBorrower(Person p){ _borrower = p; }

  public Date getPublicationDate(){
    return _publicationDate;
  }
  public void setPublicationDate(Date d){
    _publicationDate = d;
  }

  public Date getBorrowDate(){ return _borrowDate; }
  public void setBorrowDate(Date d){ _borrowDate = d; }

  public Date getReturnDate(){ return _returnDate; }
  public void setReturnDate(Date d){ _returnDate = d; }
```

NOW LOOKING AT LIBRARY BOOK METHODS



JAVA CODE FOR OPERATIONS

```
// constructor method
 LibraryBook(Person author,
             String title,
             Date publicationDate){
   this.author = author;
   this.title = title;
   setPublicationDate(publicationDate);
 }

 public void borrow(Person borrower,
                    Date borrowDate){
   setBorrower(borrower);
   setBorrowDate(borrowDate);
 }

 public static void setMaxDays(int days){
   _maxDays = days;
 }
```

JAVA CODE FOR OPERATIONS (2)

```
// implementation of derived attribute
 public int daysBorrowed(){
   long borrowTimeInMillis =
     getBorrowDate().getTime();
   long returnTimeInMillis =
     getReturnDate().getTime();
   long difference =
     returnTimeInMillis - borrowTimeInMillis;
   return (int) (difference/(1000*60*60*24));
 }

 // use two methods to model pair return type
 public boolean fineApplicable(){
   return (daysBorrowed() > _maxDays);
 }
 public double computeFine(){
   return (_maxDays-daysBorrowed())*dailyFine;
 }
```

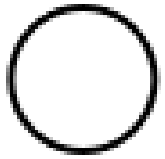
JAVA CODE FOR OPERATIONS (3)

```
// a simple test driver to test the daysBorrowed()
// method
  public static void main(String[] args){
    GregorianCalendar gc1 =
      new GregorianCalendar(1999, 5, 1, 0, 0, 0);
    Person fowler = new Person("Martin Fowler");
    LibraryBook book =
      new LibraryBook(fowler, "UML Distilled",
    gc1.getTime());
    GregorianCalendar borrowDate =
      new GregorianCalendar(2002, 5, 18, 0, 0, 0);
    book.setBorrowDate(borrowDate.getTime());
    GregorianCalendar returnDate =
      new GregorianCalendar(2002, 6, 1, 0, 0, 0);
    book.setReturnDate(returnDate.getTime());
    System.out.println("days = " +
                       book.daysBorrowed());
  }
}
```

INTERFACES, REALIZATION, ROLES

- An interface is a collection of operations that are used to specify a service of a class or component
- Visualize, specify, construct document “seams” of the system
- Provides separation between outside view and inside view
  - Allows for distribution of impl. Responsibilities
  - Replace implementation without affecting clients
- Similar to abstract classes
- In UML, we say that a class realizes an interface, and that a class may play the role of an interface

INTERFACES: NOTATION



java.util.Observable

<<interface>>
java.util.Observable

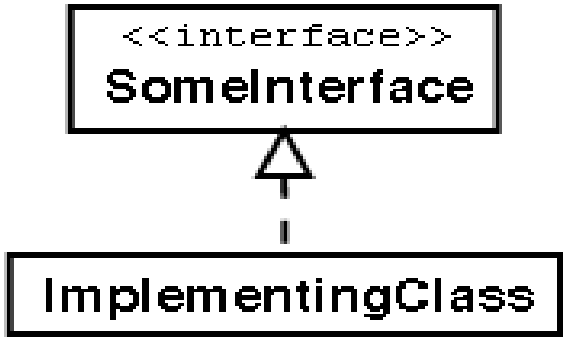
+addObserver(Observer)
+deleteObserver(Observer)
+deleteObservers()
+hasChanged(): boolean
+notifyObservers()
+notifyObservers(Object)
#clearChanged()
+countObservers(): int
#setChanged()

elided form

canonical form

REALIZATION: NOTATION

- two ways to depict how a class realizes an interface
  - lollipop notation (solid line with circle)
  - realization arrow (dashed edge with open head)



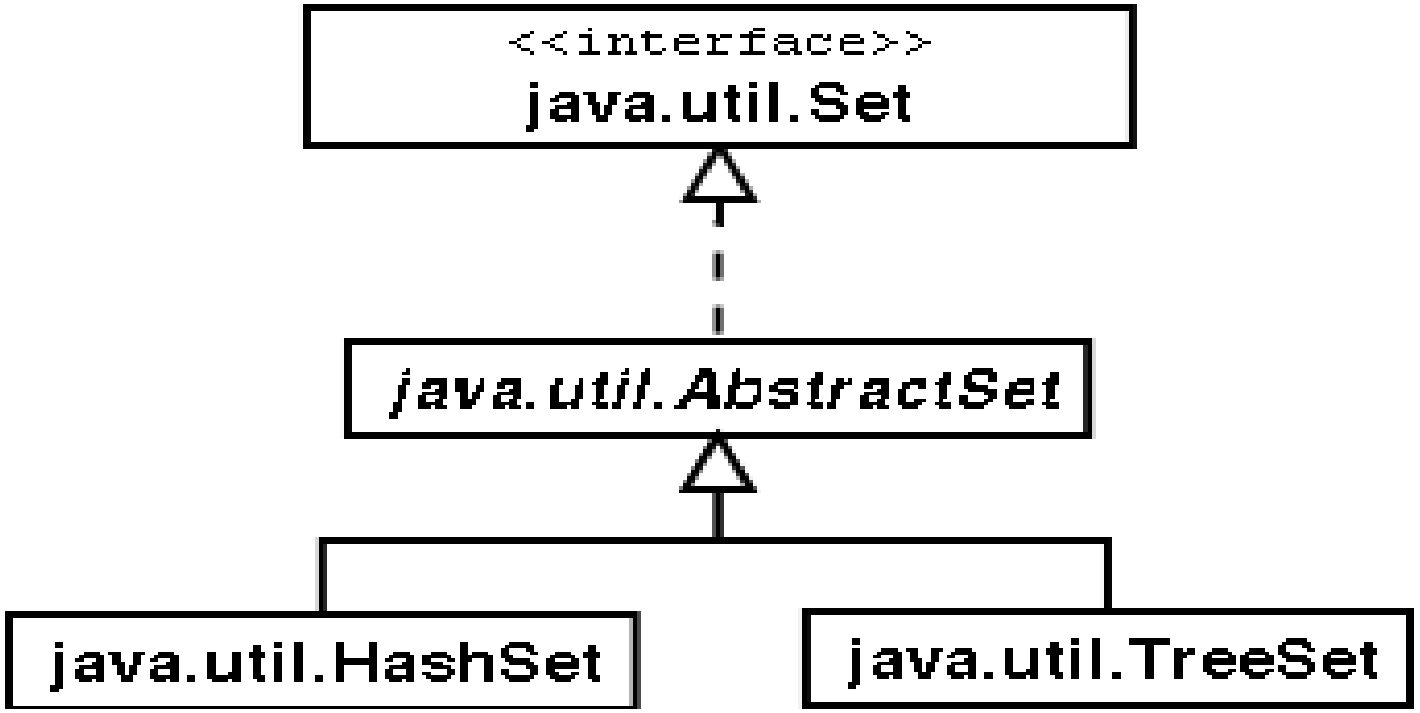
EXAMPLE: HASHSET CLIENT

```
class University {
  University(){
    students = new HashSet();
  }
  private HashSet students;
  private HashSet professors;
  private HashSet courses;

  public void enroll(Person p){
    if (!students.contains(p)){
      students.add(p);
    } else {
      System.out.println(p.toString() +
                         " is already enrolled");
    }
  }
}
```

EXAMPLE: HASHSET

```
C:\>javap java.util.HashSet

This utility can be used to reverse assemble code. Many program
license agreements do not permit reverse assembly. If you are not
the copyright owner of the code which you want to reverse assemble,
please check the license agreement under which you acquired such
code to confirm whether you are permitted to perform such reverse
assembly.

Compiled from HashSet.java
public class java.util.HashSet extends java.util.AbstractSet
implements java.util.Set, java.lang.Cloneable, java.io.Serializable {
    public java.util.HashSet();
    public java.util.HashSet(int);
    public java.util.HashSet(int,float);
    public java.util.HashSet(java.util.Collection);
    public boolean add(java.lang.Object);
    public void clear();
    public boolean contains(java.lang.Object);
    public boolean isEmpty();
    public java.util.Iterator iterator();
    public boolean remove(java.lang.Object);
    public int size();
    ...
```

PART OF THE JAVA COLLECTIONS



INTERFACE JAVA.UTIL.SET

```
public interface java.util.Set extends java.util.Collection {
  public abstract boolean add(java.lang.Object);
  public abstract void clear();
  public abstract boolean contains(java.lang.Object);
  public abstract boolean isEmpty();
  public abstract java.util.Iterator iterator();
  public abstract boolean remove(java.lang.Object);
  public abstract int size();
  ...
}
```

EXAMPLE: REVISED CLIENT

```
class University {
  University(){
    students = createSet();
  }
  private Set students;
  private Set professors;
  private Set courses;

  public void enroll(Person p){
    if (!students.contains(p)){
      students.add(p);
    } else {
      System.out.println(p.toString() +
                         " is already enrolled");
    }
  }
  private static Set createSet(){
    return new HashSet();
  }
}
```