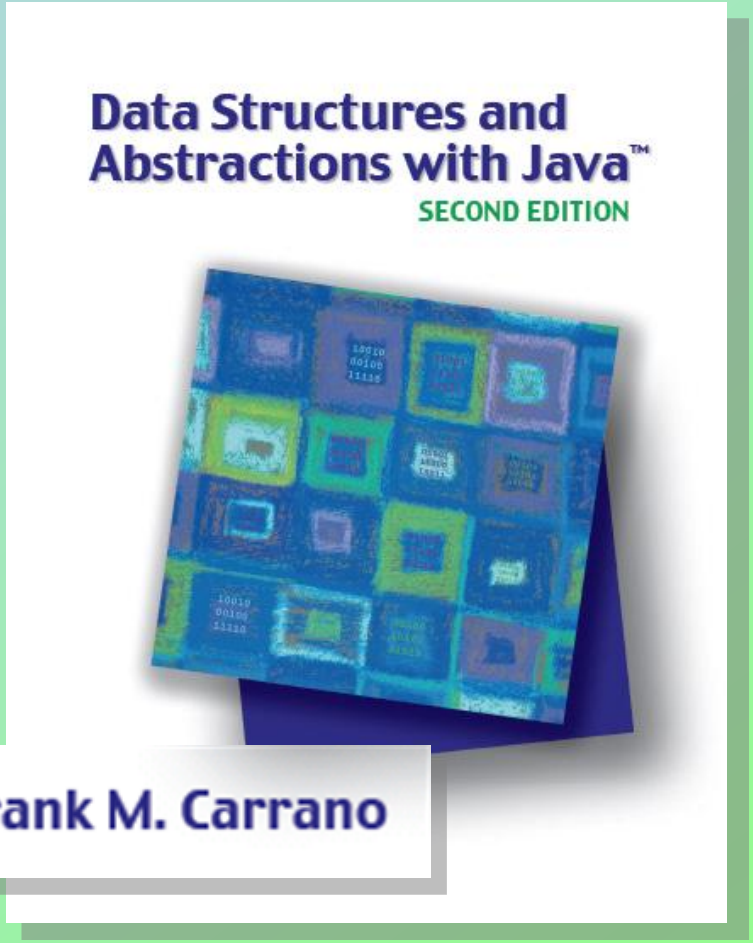
Designing Classes

Appendix D



Slides by Steve Armstrong
LeTourneau University
Longview, TX

Chapter Contents

- Encapsulation
- Specifying Methods
- Java Interfaces
  - Writing an Interface
  - Implementing an Interface
  - An Interface as a Data Type
  - Generic Types Within an Interface
  - Type Casts Within an Interface Implementation
  - Extending an Interface
  - Interfaces Versus Abstract Classes
  - Named Constants Within an Interface

Chapter Contents

- Choosing Classes
  - Identifying Classes
  - CRC Cards
  - The Unified Modeling Language
- Reusing Classes

Encapsulation

- Hides the fine detail of the inner workings of the class
  - The implementation is hidden
  - Often called "information hiding"
- Part of the class is visible
  - The necessary controls for the class are left visible
  - The class interface is made visible
  - The programmer is given only enough information to use the class

Encapsulation

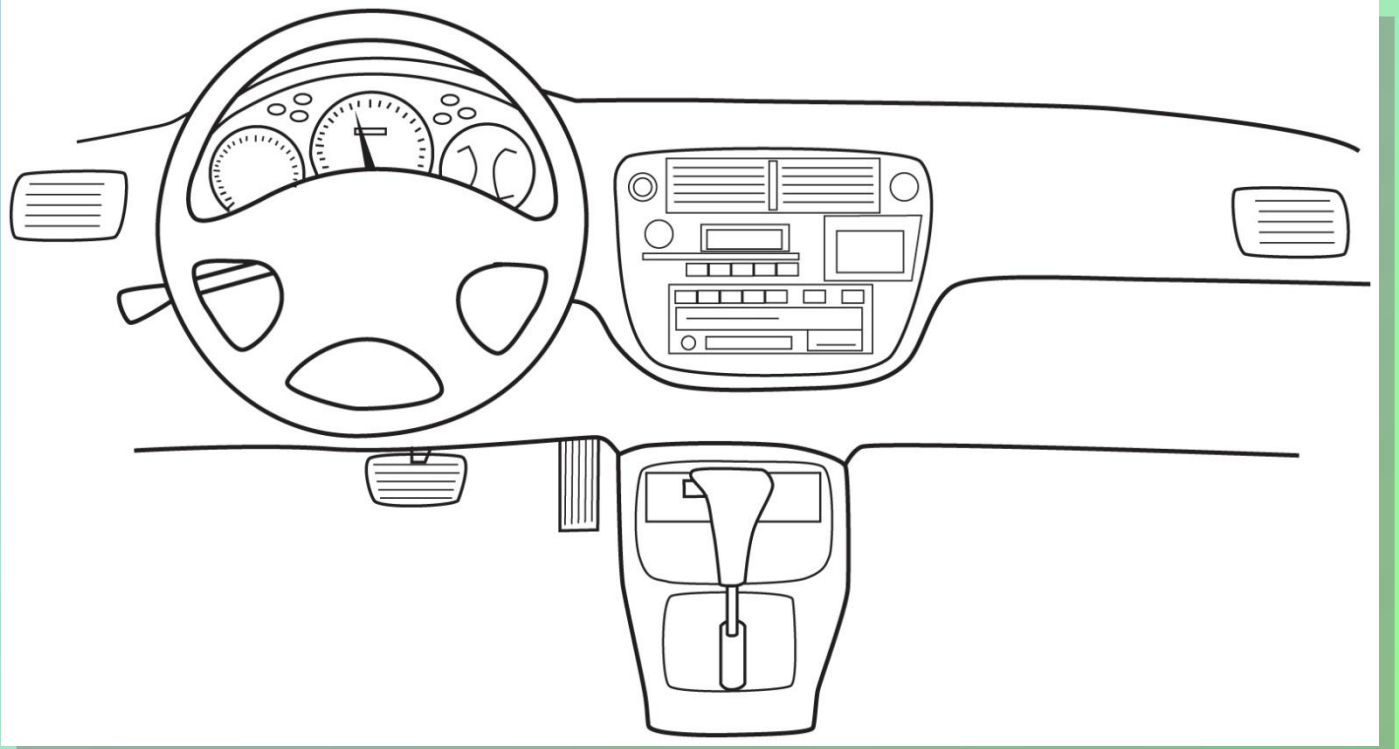


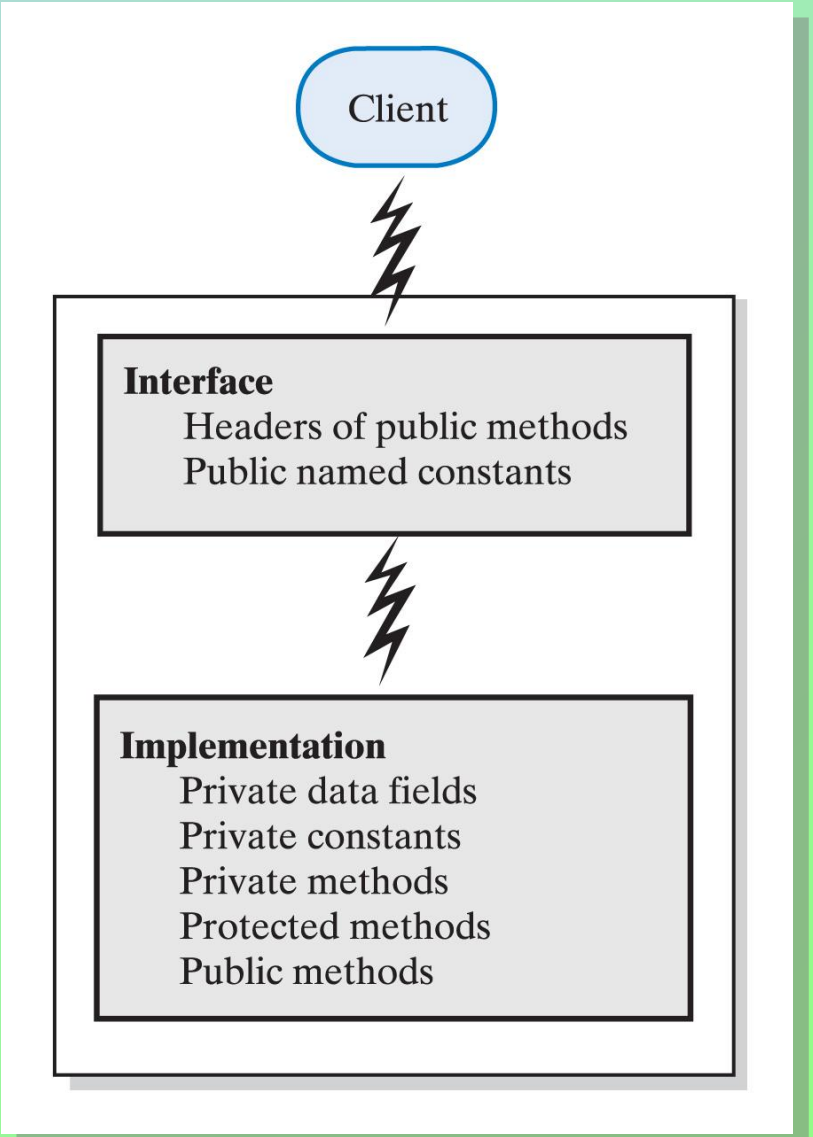
Fig. 3-1 An automobile's controls are visible to the driver, but its inner workings are hidden.

Abstraction

- A process that has the designer ask what instead of why
  - What is it you want to do *with* the data
  - What will be done *to* the data
- The designer does not consider how the class's methods will accomplish their goals
- The client interface is the what
- The implementation is the how

Abstraction

Fig. 3-2 An interface provides well-regulated communication between a hidden implementation and a client.



- **Question 1 How does a client interface differ from a class implementation?**

- **Question 2 Think of an example, other than an automobile, that illustrates encapsulation.** What part of your example corresponds to a client interface and what part to an implementation?

1. **A client interface describes how to use the class. It contains the headers for the class's public methods, the comments** that tell you how to use these methods, and any publicly defined constants of the class. The implementation consists of all data fields and the definitions of all methods, including those that are public, private, and protected.
2. **A television is one example. The remote control and the controls on the TV form the client interface. The implementation** is inside the TV itself.

Specifying Methods

- Specify what each method does
- Precondition
  - Defines responsibility of client code
- Postcondition
  - Specifies what will happen if the preconditions are met

Specifying Methods

- Responsibility
  - Precondition implies responsibility for guarantee of meeting certain conditions
- Where to place responsibility
  - Client? or …
  - Method?
- Best to comment this clearly before method's header
  - Also good to have method check during debugging

- Question 3 Assume that the class Square has a data field side and the method setSide to set the value of side. What header and comments can you write for this method? Keep in mind a precondition and postcondition as you do this.

3. Here are three possibilities:

```
/** Sets the side of the square to a new value.
    @param newSide a real number >= 0 */
public void setSide(double newSide)

/** Sets the side of the square to a new value.
    @param newSide a real number
    @return true if the side is set, or
    false if newSide is < 0 */

public boolean setSide(double newSide)
/** Sets the side of the square to a new value.
    @param newSide a real number
    @throws IllegalArgumentException if newSide is < 0 */
public void setSide(double newSide)
```

Assertions

- Assertion is statement of truth about aspect of a program's logic
  - Like a boolean statement that should be true at a certain point
- Assertions can be written as comments to identify design logic

```
// Assertion: intVal >= 0
```

Question 4 Suppose that you have an array of positive integers. The following statements find the largest integer in the array. What assertion can you write as a comment after the for loop?

```
int max = 0;
for (int index = 0; index < array.length; index++)
{
   if (array[index] > max)
         max = array[index];
} // Assertion:
```

4. // Assertion: max is the largest of array[0],..., array[index]

Assert Statement

- **`assert someVal < 0;`**
- if true, program does nothing
- If false, program execution terminates

  Exception in thread “main” java.lang.AssertionError
- **assert sum > 0 : sum;**

  adds the value of sum to the error message in case sum ≤ 0.

Or

  - **assert sum > 0 : "sum greater than zero";**

- By default, assert statements are disabled at execution time.

Java Interface

- A program component that contains
  - Public constants
  - Signatures for public methods
  - Comments that describe them
- Begins like a class definition
  - Use the word **`interface`** instead of **`class`**

```
public interface someClass
{
   public int someMethod();
}
```

Writing an Interface

- Consider geometric figures which have both a perimeter and an area
- We want classes of such objects to have such methods
  - And we want standardization signatures
- View example of the interface **`Measurable`**

Java Interface Example

- Recall class **`Name`** from previous chapter
- Consider an interface for the class **`Name`**
  - View example listing
- Note
  - Comments of method purpose, parameters, pre- and post-conditions
  - Any data fields should be public, final, and static
  - Interface methods cannot be final

Implementing an Interface

- A class that implements an interface must state so at start of definition with **`implements`** clause

```
public class myClass implements someInterface
```

- The class must implement every method declared in the interface
- Multiple classes can implement the same interface
- A class can implement more than one interface

```
public class C extends B
    implements Measurable, AnotherInterface
```

Implementing an Interface

The interface

```
public interface NameInterface
{
   . . .




}
```

NameInterface.java

The class

```
public class Name implements
                    NameInterface
{
   . . .




}
```

Name.java

The client

```
public class Client
{
   . . .

   NameInterface joe;
   . . .

   joe = new Name();

}
```

Client.java

Fig. 3-3 The files for an interface, a class that implements the interface, and the client.

Question 5 Write a Java interface that specifies and declares methods for a class of students.

Question 6 Begin the definition of a class that implements the interface that you wrote in answer to the previous questions. Include data fields, a constructor, and at least one method definition.

```
5. public interface StudentInterface
{
public void setStudent(Name studentName,
                                        String studentId);
public void setName(Name studentName);
public Name getName();
public void setId(String studentId);
public String getId();
public String toString();
} // end StudentInterface
```

6.

```
public class Student implements StudentInterface
{
private Name fullName;
private String id; // identification number

public Student()
{
    fullName = new Name();
    id = "";
} // end default constructor
public Student(Name studentName, String studentId)
{
    fullName = studentName;
    id =studentId;
} // end constructor
public void setStudent(Name studentName, String studentId)
{
    setName(studentName); // or fullName = studentName;
    setId(studentId); // or id = studentId;
} // end setStudent
```

An Interface as a Data Type

- An interface can be used as a data type

```
public void someMethod (someInterface x)
// argument passed must be an object
// of class that implements someInterface
```

or ...

```
NameInterface myName;
// the variable can invoke only a certain
// set of methods
```

- **Question 7 What revision(s) should you make to both the interface you wrote for** Question 5 and the class Student that implements it to make use of NameInterface?

7. **In the interface and in the class, replace Name with NameInterface in the methods setStudent, setName, and getName.** Additionally in the class, replace Name with NameInterface in the declaration of the data field fullName and in the parameterized constructor.

Generic Types Within an Interface

- Recall class **OrderedPair**
  - Note the **setPair** method
- Consider an interface **Pairable** that declares this method

```
public interface Pairable < S >
{
    public void setPair (S firstItem, S secondItem);
} // end Pairable
```

A class that implements this interface could begin with the statement

```
public class OrderedPair < T > implements Pairable < T >
```

The Interface Comparable

- Method **`compareTo`** compares two objects, say x and y
  - Returns signed integer
  - Returns negative if **`x < y`**
  - Returns zero if **`x == y`**
  - Returns positive if **`x > y`**

```
package java.lang;
public interface Comparable<T>
{
public int compareTo(T other);
} // end Comparable
```

The Interface Comparable

- Consider a class **`Circle`**
  - Methods **`equals, compareTo`**
  - Methods from **`Measurable`**
- View source code
- Note
  - Implements **`Measurable`** interface
  - Shown with two alternative versions of **`compareTo`**

Question 8 Define a class Name that implements the interface NameInterface, as given in Listing D-2, and the interface Comparable.

```
8. /**
A class that represents a person's name.
@author Frank M. Carrano
*/
public class Name implements
    NameInterface, Comparable<Name>
{
        private String first; // first name
        private String last; // last name
public Name()
{
            first = "";
            last = "";
} // end default constructor
public Name(String firstName, String
                                lastName)
{
            first = firstName;
            last = lastName;
} // end constructor
public void setName(String firstName,
                    String lastName)
{
            setFirst(firstName);
            setLast(lastName);
} // end setName
public String getName()
{
            return toString();
} // end getName
public void setFirst(String firstName)
{
            first = firstName;
} // end setFirst
public String getFirst()
{
            return first;
} // end getFirst
public void setLast(String lastName)
{
            last = lastName;
} // end setLast
```

```
public String getLast()
{
        return last;
} // end getLast
public void giveLastNameTo(NameInterface
aName)
{
        aName.setLast(last);
} // end giveLastNameTo
public String toString()
{
        return first + " " + last;
} // end toString
public int compareTo(Name other)
{
    int result = last.compareTo(other.last);
    // if last names are equal, check first names
    if (result == 0)
        result = first.compareTo(other.first);
    return result;
} // end compareTo
} // end class Name
```

Extending an Interface

- Use inheritance to derive an interface from another
- When an interface extends another
  - It has all the methods of the inherited interface
  - Also include some new methods
- Also possible to combine several interfaces into a new interface
  - Not possible with classes

Question 9 Imagine a class Pet that contains the method setName, yet does not implement the interface Nameable of Segment D.24. Could you pass an instance of Pet as the argument of the method with the following header?

void enterShow(Nameable petName)

9. No. The class Pet must state that it implements Nameable in an implements clause.

Interfaces Versus Abstract Classes

- Purpose of interface similar to purpose of abstract class
- But … an interface is not a base class
  - It is not a class of any kind
- Use an abstract base class when
  - You need a method or private data field that classes will have in common
- Otherwise use an interface

Named Constants Within an Interface

- An interface can contain named constants
  - Public data fields initialized and declared as **`final`**
- Consider an interface with a collection of named constants
  - Then derive variety of interfaces that can make use of these constants

Choosing Classes

- Look at a prospective system from a functional point of view
- Ask
  - What or who will use the system
  - What can each actor do with the system
  - Which scenarios involve common goals
- Use a case diagram

Choosing Classes

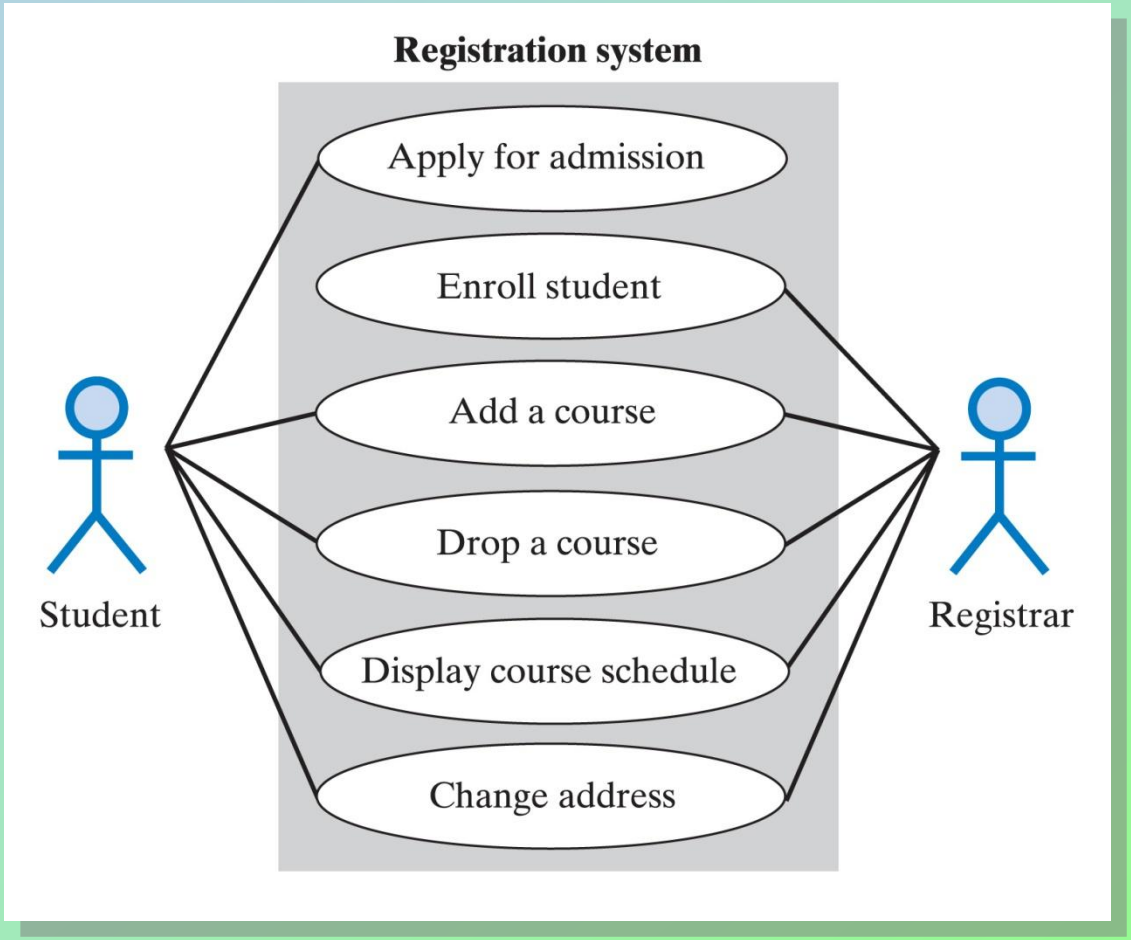


Fig. 3-4 A use case diagram for a registration system

Identifying Classes

- Describe the system
  - Identify nouns and verbs
- Nouns suggest classes
  - Students
  - Transactions
  - Customers
- Verbs suggest appropriate methods
  - Print an object
  - Post a transaction
  - Bill the customer

Identifying Classes

System: Registration
Use case: Add a course
Actor: Student
Steps:

1. Student enters identifying data.
2. System confirms eligibility to register.
   a. If ineligible to register, ask student to enter identification data again.
3. Student chooses a particular section of a course from a list of course offerings.
4. System confirms availability of the course.
   a. If course is closed, allow student to return to Step 3 or quit.
5. System adds course to student's schedule.
6. System displays student's revised schedule of courses.

Fig. 3-5 A description of a use case for adding a course

CRC Cards

- Index cards – each card represents one class
- Write a descriptive name for class at top
- List the class's responsibilities
  - The methods
- Indicate interactions
  - The collaborations
- These are CRC cards "Class-Responsibility-Collaboration"

CRC Cards

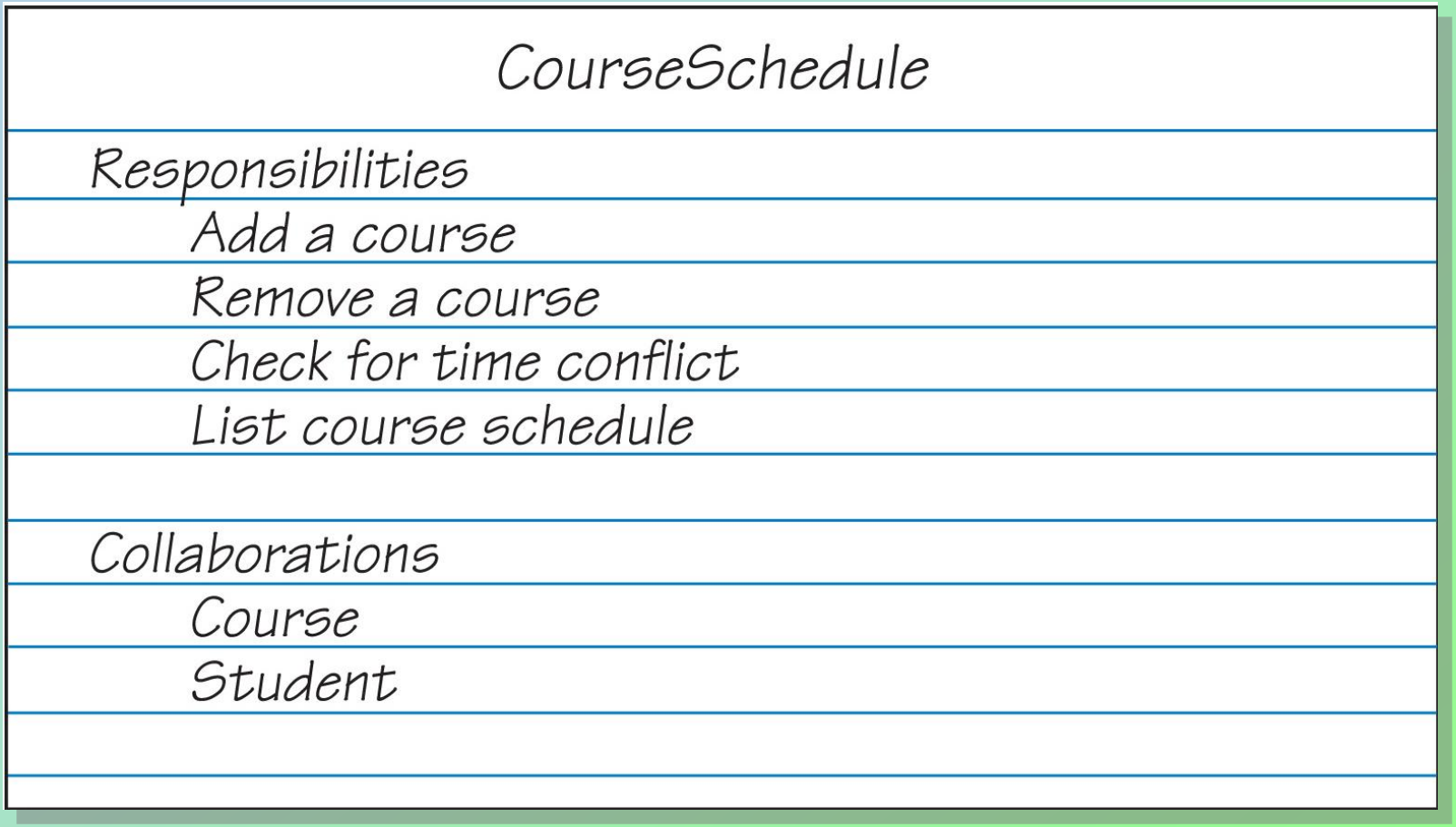


Fig. 3-6 A class-responsibility-collaboration card

Question 10 Write a CRC card for the class Student given in Appendix C.

10.

Student
Responsibilities
Set name and ID
Set name
Set ID
Get name
Get ID
Get a string that represents a student
Collaborations
String
Name

Unified Modeling Language

- Used to illustrate a system's classes and relationships
- Provides a class diagram
  - Class name
  - Attributes
  - Operations

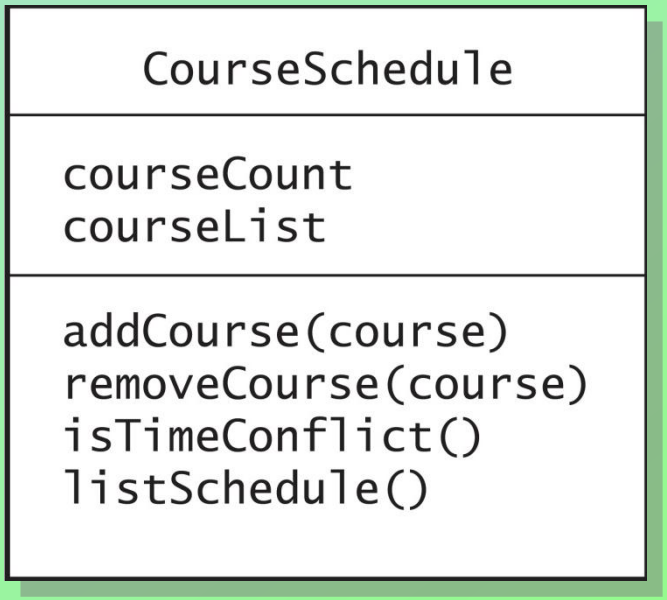


Fig. 3-7 A class representation that can be part of a class diagram.

Unified Modeling Language

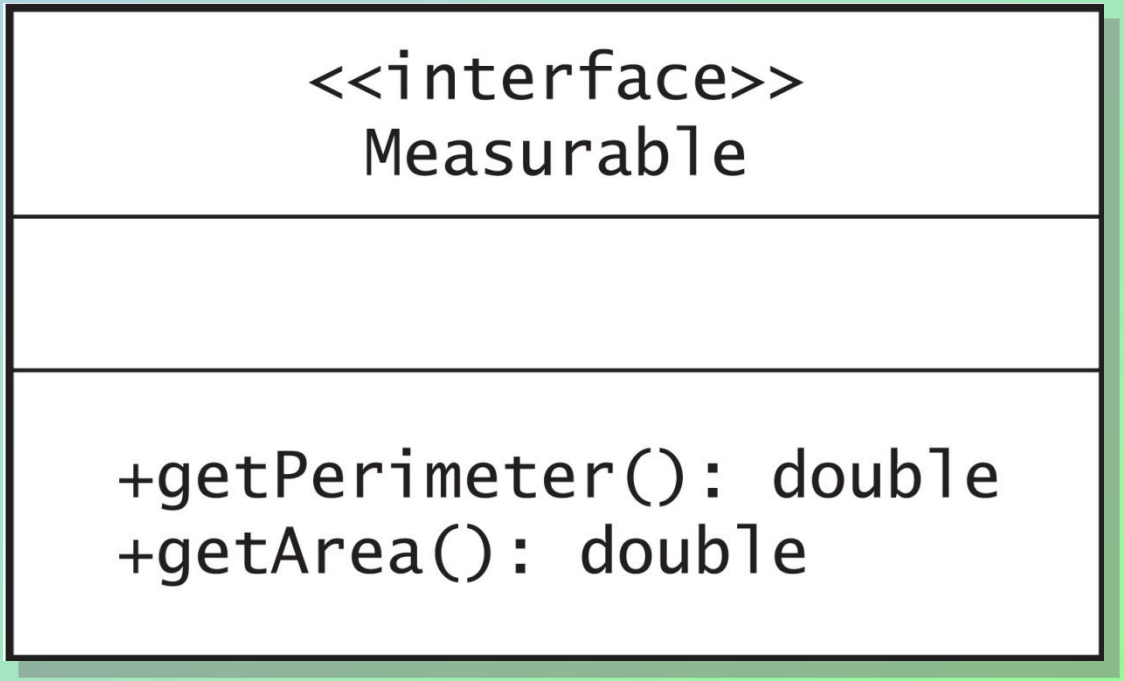
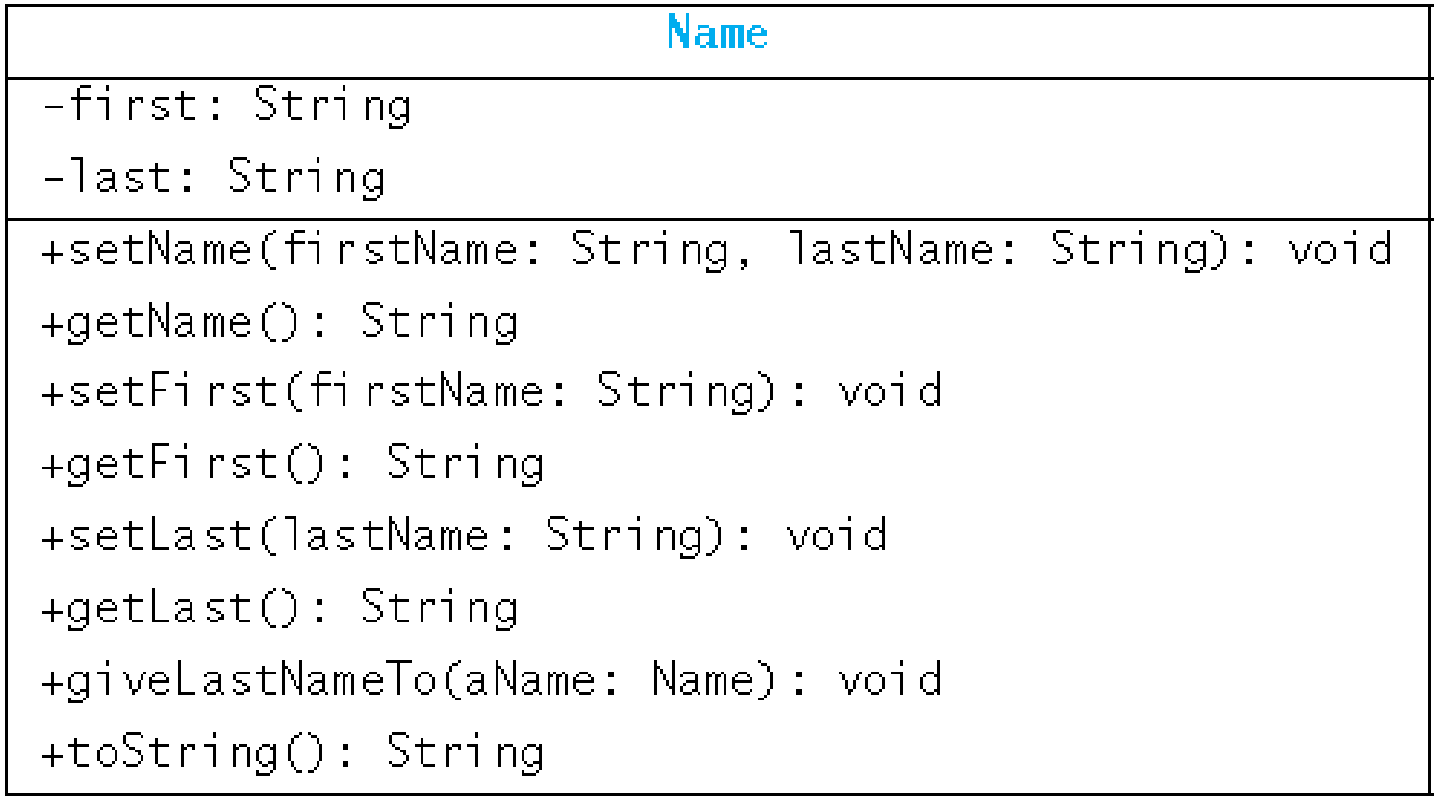


Fig. 3-8 UML notation for a base class **`Student`** and two derived classes

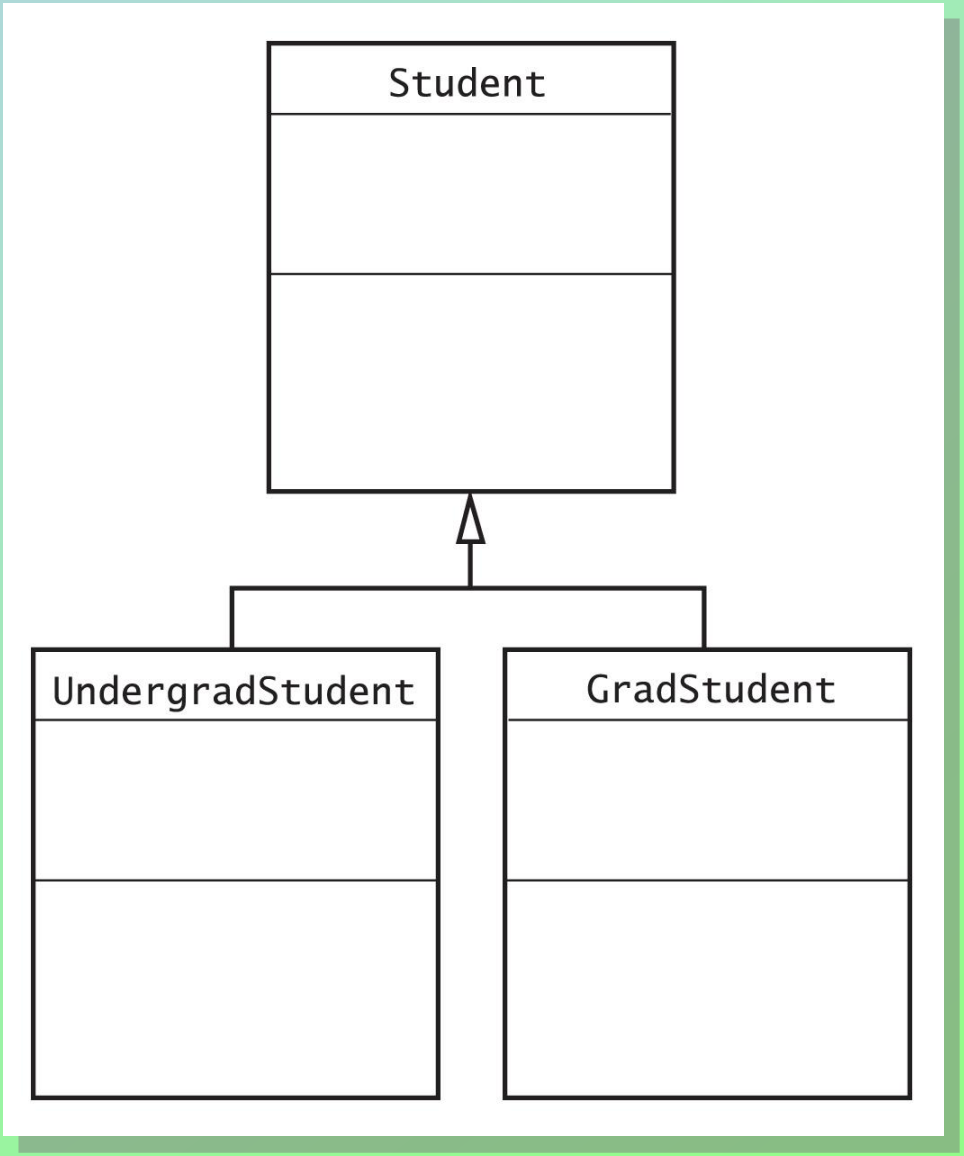
- **Question 11 How would the class Name, given in Appendix B, appear in a class diagram of t**he UML?

11.



Unified Modeling Language

Fig. 3-9 A class diagram showing the base class **`Student`** and two derived classes



Unified Modeling Language

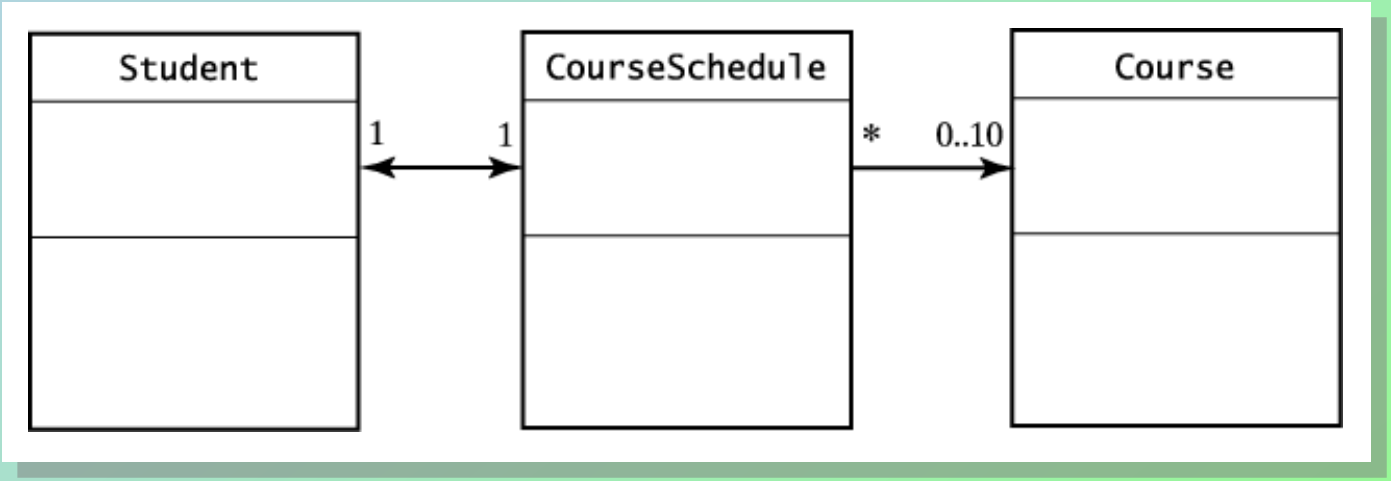


Fig. 3-10 Part of a UML class diagram with associations.

- **Question 12 Combine previous 2 slides into one class diagram. Then add a class** AllCourses that represents all courses offered this semester. What new association(s) do you need to add?

12. Add a unidirectional association (arrow) from AllCourses to Course with a cardinality of 1 on its tail and * on its head.

Reusing Classes

- Much software combines:
  - Existing components
  - New components
- When designing new classes
  - Plan for reusability in the future
  - Make objects as general as possible
  - Avoid dependencies that restrict later use by another programmer