**CPS122 Lecture: Detailed Design and Implementation**

Last revised February 21, 2022

*Objectives:*

1. To introduce the use of a complete UML class box to document the name, attributes, and methods of a class
2. To show how information flows from an interaction diagram to a class design
3. To review javadoc class documentation

*Materials :*

1. Projectables
2. Javadoc documentation for UMLImplementation labs classes
3. Javadoc documentation for java.io.File and java.awt.BorderLayout (online)

## I. Introduction

A. Preliminary note: we will weave the quick-check questions on chapter 10 into presentation instead of going through them all at the outset.

B. So far in the course we have been focussing our attention on two tasks that are part of the process of developing software: analysis and (overall) design. To do this, we have looked at several tools.

1. Class diagrams - a tool to show the various classes needed for a system, and to identify relationships between these classes - a tool to help us document the static structure of a system.

2. CRC cards - a tool to help us identify the responsibilities of each class.

3. Interaction diagrams - a tool to help us document what we discovered by using CRC cards, by showing how each use case is realized by the interaction of cooperating objects - one of several tools to help us capture the dynamic behavior of a system.

4. State Diagrams - a tool to help capture the dynamic behavior of individual objects (where appropriate).

C. We have noted that, in developing CRC cards and interaction diagrams, we often discover the need for additional classes beyond those we initially discovered when we were analyzing the domain.

1. These include classes for boundary objects and controller objects. In fact, a use case will typically be started by some boundary object, and may make use of additional boundary objects to acquire the information it needs. It will have generally have some controller object be responsible for carrying it out.

2. One writer has estimated that the total number of classes in an application will typically be about 5 times the number initially discovered during analysis.

D. We now turn to implementation phase.

1. Here, we will focus on building the individual classes, using the CRC Cards and class diagram to identify the classes that need to be built, and the interaction and state diagrams (and CRC cards) to help us build each class.

2. There are actually three kinds of activity that are part of this:

a) Detailed design of the individual classes

(1) In overall design, we are concerned with *identifying* the classes and discovering their *relationships*. One of the end results of overall design is a class diagram, showing the various classes and how they relate to one another.

(2)In detailed design, we focus on each individual class.

(a)Quick check question (a)

(b)In detailed design, we develop:

i) A class's interface - what "face" it presents to the rest of the system

ii) Its implementation - how we will actually realize the behavior prescribed by the interface.

(c)To document this, we may draw a more detailed UML representation for the class: a rectangle with three compartments:

i) Class name

ii) Attributes (instance variables)

iii)Operations (methods)

(d)A note on notational conventions - UML uses a somewhat different notation than Java does for specifying attributes and operations

i) Quick check question (e) - format for attribute (instance variable) signature

*Visibility Name* : *Type*

where visibility is + for public, # for protected, and - for private.

ii) Quick check question (f) - format for operation (method) signature

*Visibility Name*(*Parameter* : *Type* ...) : *Return Type*

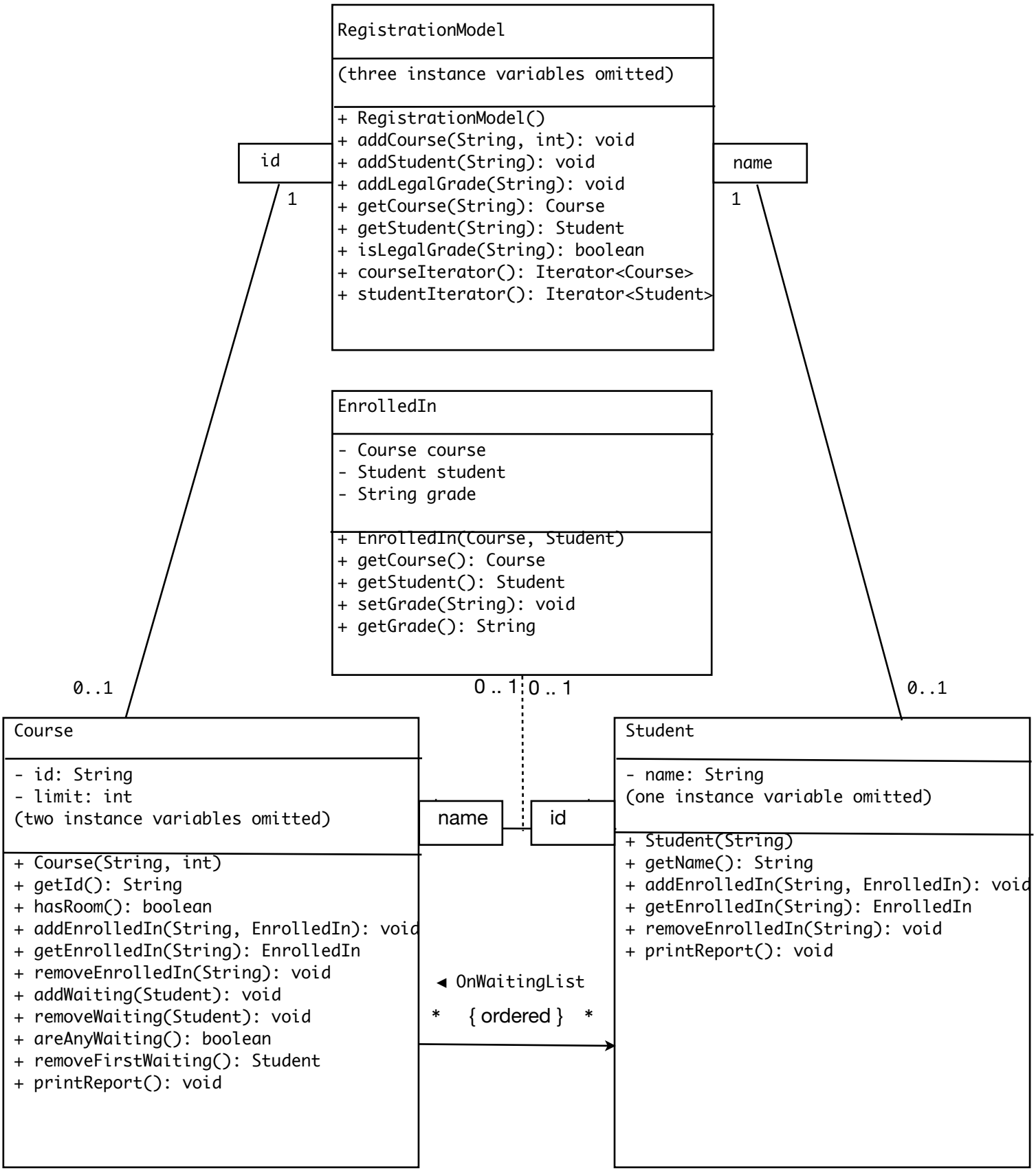
where again visibility is + for public, # for protected, and - for private.

b) Writing the code for the various methods of the class

c) Unit testing each method to be sure it does what it is supposed to do.

E. For our examples we will use a class from the labs you have been doing in the last few weeks - a system that manages student registrations in courses - the class `EnrolledIn.` For the labs, you were given the code for this class. We will now talk about how the design was developed.

1. Project overall class structure


RegistrationModel
(three instance variables omitted)
+ RegistrationModel()
+ addCourse(String, int): void
+ addStudent(String): void
+ addLegalGrade(String): void
+ getCourse(String): Course
+ getStudent(String): Student
+ isLegalGrade(String): boolean
+ courseIterator(): Iterator<Course>
+ studentIterator(): Iterator<Student>
id
name
1
1
EnrolledIn
- Course course
- Student student
- String grade
+ EnrolledIn(Course, Student)
+ getCourse(): Course
+ getStudent(): Student
+ setGrade(String): void
+ getGrade(): String
0..1
0 .. 1
0 .. 1
0..1
Course
- id: String
- limit: int
(two instance variables omitted)
+ Course(String, int)
+ getId(): String
+ hasRoom(): boolean
+ addEnrolledIn(String, EnrolledIn): void
+ getEnrolledIn(String): EnrolledIn
+ removeEnrolledIn(String): void
+ addWaiting(Student): void
+ removeWaiting(Student): void
+ areAnyWaiting(): boolean
+ removeFirstWaiting(): Student
+ printReport(): void
name
id
◄ OnWaitingList
* { ordered } *
Student
- name: String
(one instance variable omitted)
+ Student(String)
+ getName(): String
+ addEnrolledIn(String, EnrolledIn): void
+ getEnrolledIn(String): EnrolledIn
+ removeEnrolledIn(String): void
+ printReport(): void

2. Note that the class `EnrolledIn` is an association class because it has to hold a grade which is specific to the enrollment of a specific student in a specific course.

3. The responsibilities of this class might be given by the following CRC card.

   PROJECT CRC Card for `EnrolledIn`

   Class EnrolledIn

| Responsibilities | Collaborators |
| --- | --- |
| Maintain record of a student's registration in a course | Student<br>Course |
| Record the student's grade in the course | |

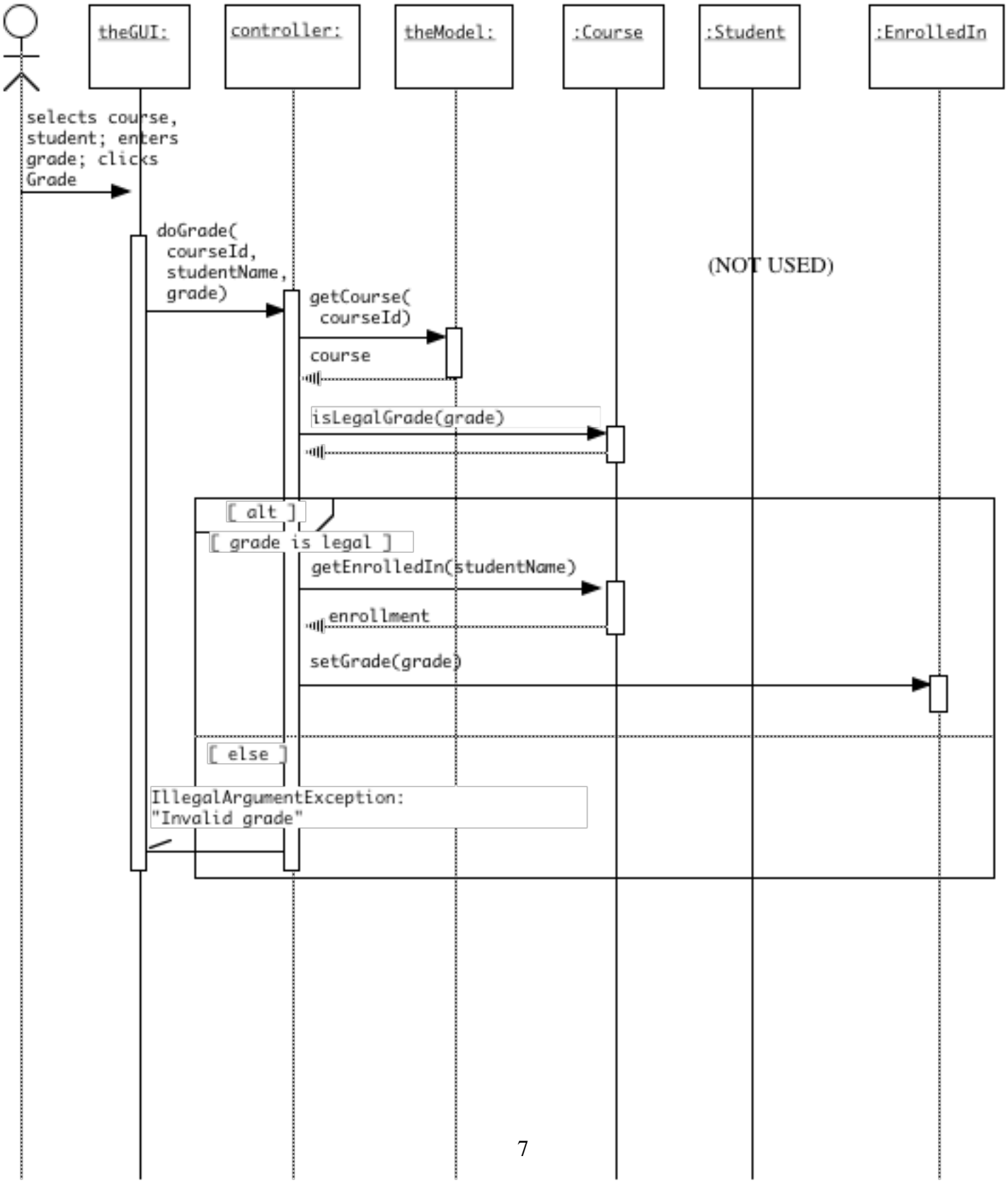
4. It actually participates in several use cases that give rise to several sequence diagrams:

   a) Grade Student

   b) Course Report

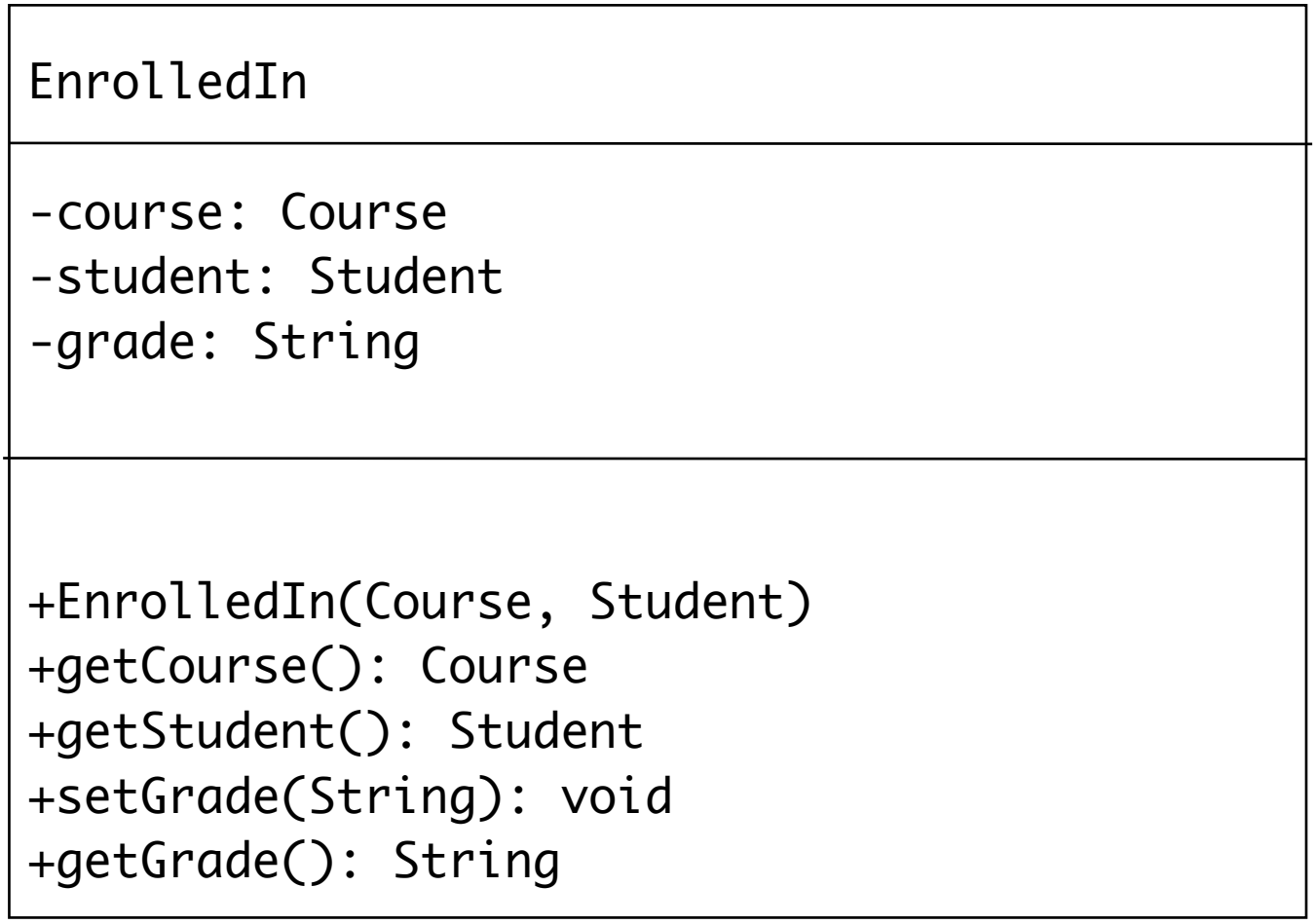
   c) Student Report

   PROJECT sequence diagrams for Grade Student as an example.


theGUI:
controller:
theModel:
:Course
:Student
:EnrolledIn
selects course, student; enters grade; clicks Grade
doGrade( courseId, studentName, grade)
(NOT USED)
getCourse( courseId)
course
isLegalGrade(grade)
[ alt ]
[ grade is legal ]
getEnrolledIn(studentName)
enrollment
setGrade(grade)
[ else ]
IllegalArgumentException: "Invalid grade"

## II. Deciding on the Instance Variables of a Class

A. In detailed design we represent each class as a three-compartment box in which the middle compartment represents the attributes of a class - its instance variables. As an example, here is the detailed design for class EnrolledIn. (PROJECT)




B. How do we decide what instance variables a class needs?

1. Basically, the instance variables hold information about <u>who</u> (what other objects) objects of the class know and <u>what</u> objects of the class know.

2. The "who" question can be answered by looking at the associations between this class and other classes in the class diagram.

   Example: Note that `EnrolledIn` is associated with Course and Student in class diagram and so has instance variables for Course and Student

3. The "what" question can be answered by looking at the CRC cards - what does an object need to know in order to be able to fulfill its responsibilities?

   Example: In the CRC card for `EnrolledIn` it is responsible to maintain a record of a particular student's enrollment in a particular course, it needs student and course instance variables.

   Since it must keep track of a student's grade, it needs a grade instance variable

   Note instance variable grade in detailed design

C. We covered material relevant to three of the quick-check questions for this chapter in conjunction with our discussion of implementing associations (not directly tied to a book chapter), so these questions are a sort of review, but let's do them now

   Quick-check questions (b), (c), (d)

## III. Identifying the Methods of a Class

A. A key question in designing a class is “what methods does this class need”? Here, our interaction diagrams are our primary resource. Every message that an object of our class is shown as receiving in an interaction diagram must be realized by a corresponding method in our class’s interface.

1. As an example of this, consider again the interaction diagram for Grade Student in which `EnrolledIn` is involved.

PROJECT each and then show method setGrade(() in detailed design

Each of the methods in the design actually shows up a message sent *to* an `EnrolledIn` object in some interaction. (Must look at every interaction where `EnrolledIn` appears to find them all. (The others would appear in the report sequences.)

a) No other messages are sent to an `EnrolledIn` object in any interaction, and no other ordinary operations (i.e. other than the constructor) show up in the detailed design as a result.

2. Notice that we are only interested here in the messages a given class of object *receives*; not in the messages it *sends* (which are part of its implementation).

## IV. More About Designing the Interface of a class

A. The interface of a class is the "face" that it presents to other classes - i.e. its public features.

1. In a UML class diagram, public features are denoted by a "+" symbol. In Java, of course, these features will actually be declared as public in the code.

2. The interface of a class needs to be designed carefully. Other classes will depend *only* on the public interface of a given class. We are free to change the implementation without forcing other classes to change; but if we change the interface, then any class that depends on it may also have to change. Thus, we want our interface design to be stable and complete.

B. An important starting point for designing a class is to write out a brief statement of what its basic role is - what does it represent and/or do in the overall context of the system.

1. If the class is properly cohesive, this will be a single statement.

2. If we cannot come up with such a statement, it may be that we don't have a properly cohesive class!

3. We have been documenting our classes using javadoc. One component of the javadoc documentation for the class is a *class comment* - which spells out the purpose of the class. (We will review other javadoc features at the appropriate point later on.)

   *EXAMPLE:* Show online documentation for UML Implementation Labs classes

C. Languages like Java allow the interface of a class to include both attributes (fields) and behaviors (methods). It is almost always the case that fields should be private or protected (some writers would argue always, not just almost always), so that the interface consists only of:

1. Methods

2. Constants (public static final ...)

3. Note that, while good design dictates that methods and constants *may* be part of the public interface of a given class, good design does not *require* that *all* methods and constants be part of the public interface. If we have some methods and/or constants that are needed for the implementation of the class, but are not used by the "outside world", they belong to the private implementation .

4. In general, we should use javadoc to document each feature that is part of the public interface of a class - including any protected features that, while not publicly accessible, are yet needed by subclasses. Using javadoc for private features may be helpful to a maintainer; but the javadoc program, by default, does not include private features in the documentation it generates.

D. An important principle of good design is that our methods should be cohesive - i.e. each method should perform a single, well-defined task.

a) A way to check for cohesion is to see if it is possible to write a simple statement that describes what the method does.

b) In fact, this statement will later become part of the documentation for the method - so writing it now will save time later.

*EXAMPLE:* Look at documentation for class `java.io.File`. Note descriptions of each method.

c) The method name should clearly reflect the description of what the method does. Often, the name will be a single verb, or a verb and an object. The name may be an imperative verb - if the basic task of the method is to *do* something; or it may be an interrogative verb - if the basic task of the method is to *answer a question*.

*EXAMPLE:* Note examples of each in methods of `File`.

d) Something to watch out for - both in method descriptions and in method names - is the need to use conjunctions like "and". This is often a symptom of a method that is not cohesive.

2. Another important consideration in designing a method is the *parameters* needed by the method.

a) Parameters are typically used to pass information *into* the method. Thus, in designing a parameter list, a key question to ask is "what does the sender of the message know that this method needs to know?" Each such piece of information will need to be a parameter.

b) There is a principle of narrow interfaces which suggests that we should try to find the *minimal* set of parameters necessary to allow the method to do its job.

3. A third important consideration is the *return value* of the method.

4. Just as we use a javadoc class comment to document each class, we use a javadoc method comment to document each method. As we have discussed in the past, then, the documentation for a method includes:

   a) A statement of the purpose of the method. (Which should, again, be a single statement if the method is cohesive).

   b) A description of the parameters of the method.

   c) A description of the return value - if any.

   d) An IDE like Netbeans can help to generate these

      DEMO - Create an application and use main class created

      (1)Type method prologue
      (2)Type /** on line before method
      (3)Press enter key (on keypad - not return; or return on a laptop)
      (4)But note that - but note that it is vital to complete the comment by explaining the purpose of the method, the purpose of each parameter., and the meaning of any return value.

E. Sometimes, another issue to consider in determining the methods of an object is the "common object interface" - methods declared in class `Object` (which is the ultimate base class of all classes) that can be overridden where appropriate. Most of the time, you will not need to worry about any of these. The ones you are most likely to need to override are:

1. The `boolean equals(Object)` method used for comparisons for equality of value.

2. The `String toString()` method used to create a printable representation of the object - sometimes useful when debugging.

   *EXAMPLE:* Show overrides in class `SimpleDate` for project in excepts from `SimpleDate.java`

```
    /** Convert this object to a nicely printable
     *  string
     *
     *  @return formatted string representing
     *   this object in MM/DD/YY format
     */
    public String toString()
    {
        return DateFormat.getDateInstance
                                    (DateFormat.SHORT).format
                            (value.getTime());
    }

       ...

    /** Test to see if the date represented by
     *  this object is the same as that
     *  represented by some other object
     *
     *  @param other the date to be compared to
     *  @return true if other object is a
     *          SimpleDate and date represented
     *          by this object is the same as
     *          that represented by other object
     *
     */
    public boolean equals(Object other)
    {
        return (other instanceof SimpleDate) &&
            this.value.equals(
                                ((SimpleDate) other).value);
    }

       ...
}
```

3.

F. While the bulk of a class's interface will typically be methods, it is also sometimes useful to define symbolic constants that can serve as parameters to these methods

1. *EXAMPLE:* `java.awt.BorderLayout`
2. In Java, constants are declared as final static. A convention in Java is to give constants names consisting of all upper-case letters, separated by underscores if need be.

   Public constants should also be documented via javadoc

   *SHOW* Documentation for constants of class `java.awt.BorderLayout`

   *PROJECT:* source code showing javadoc comments.

```
/**
 * The north layout constraint (top of container).
 */
public static final String NORTH  = "North";

/**
 * The south layout constraint (bottom of container).
 */
public static final String SOUTH  = "South";

/**
 * The east layout constraint (right side of container).
 */
public static final String EAST   = "East";

/**
 * The west layout constraint (left side of container).
 */
public static final String WEST   = "West";

/**
 * The center layout constraint (middle of container).
 */
public static final String CENTER = "Center";
```

## V. Some Final Thoughts on Detailed Design/Implementation

A. In the case of class hierarchies, we need to think about what *level* in the hierarchy each attribute belongs on.

   Recall the "Pet Kennel" lab

B. Sometimes, in implementing methods, we discover that it would be useful to introduce one or more *private* methods that facilitate the tasks of the public methods by performing well-defined subtasks.

C. A final consideration is the *physical arrangement* of the source code for a class. A reasonable way to order the various methods and variables of a class is as follows:

   1. Immediately precede the class declaration with a class comment that states the purpose of the class.

   2. Put public members (which are part of the interface) first - then private members. That way a reader of the class who is interested in its interface can stop reading when he/she gets to the implementation details in the private part.

   3. Organize the public interface members in the following order:

      a) Class constants (if any)

      b) Constructor(s)

      c) Mutators

      d) Accessors

   4. In the private section, put method first, then variables.

      But: sometimes it is desirable to put the instance variables at the very start of the class

   5. If the class contains any test driver code, put this last.