CS211 Lecture: Detailed Design and Implementation

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 introduce javadoc class documentation
4. To introduce design by contract: method preconditions, postconditions; class invariants.

Materials:

1. Online display of various ATM Example pages
2. Javadoc documentation for Registration system labs classes and transparency of source code for class RegistrationModel (class comment+selected methods only).
3. Javadoc documentation for java.awt.TextField and transparency of source code for constructors
4. Transparency of source code for equals() method in java.lang.Integer
5. Javadoc documentation for java.awt.BorderLayout and transparency of source code for constants
6. Gries’ coffee can problem - demo plus handout of code

I. Introduction

A. We have seen that, regardless of what software development model is followed, certain tasks will need to be done as part of the development process per se - whether all at once, iteratively, or incrementally.

1. Analysis. The goal of this task is to understand the problem.

2. Design. The goal of this task is to develop the overall structure of a solution to the problem in terms of individual, buildable components and their relationships to one another.

3. Implementation. The goal of this task is to actually build the system as designed.

4. Quality Assurance. The goal of this task is to ensure that the individual components and the system as a whole do what they are supposed to do (which involves identifying their shortcomings and fixing them.)
5. Deployment and Maintenance - using the software, and making changes as necessary.

B. We have been focussing our attention on the second of these tasks: overall design. To do this, we have looked at several tools:

1. CRC cards - a tool to help us discover the classes needed for solving a giving problem, and to identify the responsibilities of each class.

2. 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.

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. (We will see later how this tool also helps us to develop each individual class).

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

C. The relationship between these tools (and the use cases we developed in the analysis phase) can be summarized as follows:
Note that, from the analysis side, we focus on use cases; from the design side, we end up focusing on classes. These are two distinct ways of structuring the system - though not totally unrelated, since each use case will be made the responsibility of some class. (But there will be many classes that play a subordinate role, perhaps to many use cases; and it is possible that one class may have responsibility for several use cases.)

D. We now turn to implementation phase. Here, we will focus on implementing the individual classes, using the CRC Cards and class diagram to identify the classes that need to be built, and the interaction and statechart diagrams (and CRC cards) to help us build each class.

E. There is a design component to this phase as well - sometimes known as detailed design. We can contrast this sort of design with the overall design done in the second phase, as follows.

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.
   
a) In order to keep the drawing manageable, at this stage we usually represent each class by a rectangle containing only its name.

b) In fact, if the number of classes is large, we may group classes into packages and focus on these larger groupings.

2. In detailed design, we focus on each individual class.
   
a) We must develop:

   (1) Its interface - what “face” it presents to the rest of the system

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

b) In the process of doing this, we will identify the class’s:

   (1) Attributes

   (2) Methods

   c) To document this, we may draw a more detailed UML representation for the class: a rectangle with three compartments:
(1) Class name
(2) Attributes
(3) Methods

d) EXAMPLE: SHOW detailed design for class CashDispenser

(1) Attributes:

(a) One represents an association the CashDispenser is part of

i) Which one?

ASK

ii) SHOW: Class diagram - note that a CashDispenser is part of two associations. Why does just one show up as an attribute in the class?

ASK

Both associations happen to be unidirectional. The CashDispenser does need to know about the Log, but not about the ATM.

(Of course, in general, a class can have any number of attributes representing associations.)

(b) Some represent things CashDispenser must know to do its job

Note that, in contrast with attributes representing associations, there is no one place where we can go to find out what attributes a class needs to keep track of its internal status.

i) In this case, the need for the cashOnHand attribute is implicit in the responsibilities given to the class.

SHOW CRC Card

ii) In other cases, a needed attribute may explicitly appear in a sequence diagram. However, not every variable appearing in a sequence diagram should be an attribute!
EXAMPLE: SHOW interaction diagram for class Session. What variables appear in this diagram?

ASK - list on board

Now show detailed design for class Session. Note that only some show up as attributes. Why?

ASK

Only those pieces of information that are part of the permanent state of the object (and which are typically accessed by more than one method) show up as attributes - the rest can be instances variables of a particular method. (Sometimes this will be apparent when doing the design; sometimes, the need to make some variable an instance variable rather than a simple local variable will only be discovered while writing the code. A design can be altered as needed.)

(2) Methods - each of the methods in the design actually shows up a message sent to a CashDispenser object in some interaction. (Must look at every interaction where CashDispenser appears to find them all)

(a) setInitialCash() appears as a message sent to the CashDispenser from the ATM in the System Startup interaction.

SHOW

(b) checkCashOnHand() and dispenseCash() appear as messages send to the CashDispenser from a WithdrawalTransaction in the Cash Withdrawal interaction.

SHOW

(c) No other messages are sent to a CashDispenser object in any interaction.

(3) Note, then, that much of the design for a class can be developed by studying the places it appears in the Class Diagram and the various interaction diagrams.
e) A note on notational conventions - UML uses the notations variable: type for attributes, parameter: type and method(...): type for method signatures, and the symbols + for public, # for protected, and - for private.

3. In implementation, we actually build and test the code for each class, which means translating the UML design into code in Java or whatever programming language we are using.

   a) The translation of the attributes into Java code is trivial.

   b) In the case of the methods, the signatures given in the UML design become the method interfaces. Of course, we still need to write the method bodies. Here, our interaction diagrams come into play.

II. 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. As we did in CS112. we will document 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 discuss other javadoc features at the appropriate point later on.)
EXAMPLE:

a) Show online documentation for Registration Labs classes

b) TRANSPARENCY: javadoc class comment in the source code for class RegistrationModel.

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 (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 will use javadoc to document each feature that is part of the public interface of a class.

D. A key question in designing an interface, then, 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. We have already seen an example of how this plays out in the design of class CashDispenser. Each interaction diagram in which a CashDispenser object appears potentially contributes one or more methods to the interface of a CashDispenser object.

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).

3. Recall that, in general, methods fall into four categories:
a) *Constructor* methods are used to initially construct an object. A constructor is responsible for ensuring that the object constructed is put into a consistent initial state.

(1) Often, a class will have more than one constructor - each with a different signature (number and types of parameters)

*EXAMPLE:* Show documentation for java.awt.TextField - note four different constructors, one with 0 parameters, two with 1 parameter (but of different types) and one with 2 parameters.

(2) The most important constructor is the “working constructor”. The working constructor allows the user to specify through its parameters all the features of the object that a user should be allowed to initially specify.

(3) Other constructors leave some features unspecified by the user, and use standard “default” values for them.

   (a) Of course, the ultimate case of this is what is called a default constructor that has *no* parameters - it uses default values for all user-specifiable parameters.

   (b) These constructors use the this() syntax to invoke the working constructor - passing along whatever values the user specifies, plus the default values for the remaining parameters.

*TRANSPARENCY:* Source code of constructors for java.awt.TextField

b) *Mutator* methods are used to change the state of an object - e.g. to change the values of one or more attributes, or to connect or disconnect an association to some other object. A mutator, by definition, changes the state of the object in some way.

c) *Accessor* methods are used to look at some piece of information about the object, without altering it. An accessor, by definition, does not change the state of the object in any way.

d) A *destructor* method is used to “clean up” an object’s associations when it is destroyed. This typically is needed if the object has acquired “ownership” of some resource that needs to be released when the object is no longer needed - e.g. it may entail:
(1) Freeing up windowing system resources (the dispose() method of class Window).

(2) Closing a file.

(3) Getting rid of other objects that are no longer needed (e.g. deleting an order would delete all of its line items as well).

Note that, once an object has been destroyed, that particular object must never be used again.

4. Programming languages vary in the support they provide for these types of methods, and the way these methods are named:

a) In both Java and C++, a constructor method always has a name that is the same as the name of its class. E.g. if you have a class called MyClass, then any method in the class with then name MyClass() is a constructor for an object of that class.

b) Java has a convention that accessors that allow access to a single field have the name “getFieldName”, and mutators that allow changing a single field have the name “setFieldName”.

(1) This convention has dangers - it can lead to designing a class that falls into what one author calls the “data warehouse trap”, where a class serves only to store data and the logic of using the data is elsewhere.

(2) However, this convention is needed to support component based development (Java beans).

(3) A good way to avoid the data warehouse trap is to let the interaction diagrams dictate the methods.

c) C++ (but not Java) has a mechanism whereby one can specify whether a method is an accessor or a mutator, by ending the method signature with the reserved word “const”. A const method cannot change the state of the object to which it is applied - hence it must be an accessor. Only mutators should be declared without the const specifier.

d) In C++, a destructor method has a name that is the same as the name of the class, but with a “~” added to it. E.g. the destructor for an object of class MyClass would be called ~MyClass().
e) Java does not have destructors in the strict sense of the word. It does allow you to specify a method with the signature void finalize(), which can be used to free up resources the object holds.

(1) The finalize() method is called after there are no remaining references to the object (it is garbage) and before its storage is reclaimed.

(2) However, there is no guarantee *when* or even *whether* the finalize() method will be called, since it is not called until the Java memory management system discovers that there are no outstanding references to the object and needs more storage.

(3) In particular, when the program terminates, the finalize() method of objects that are still in existence (even if they are garbage) will *not* be called.

(4) I have never yet had occasion to write a finalize() method for any class I have created. However, some of the standard library classes - e.g. those involved with file input output - do have finalize() methods.

5. One other 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.

   a) finalize() is actually an overridable method of Object - we have already discussed this.

   b) The Object clone() method used to create a copy of an Object

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

   d) The int hashCode() method used when the object is stored in a HashSet or is the **key** in a HashMap.

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

6. 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:
equals() - if you need to compare objects of your class with one another, and decide equality on the basis of equal value (however you define it) rather than identity.

toString() - if you need to print your object out

**EXAMPLE:** Show override for toString() in class Money (source code for class) - online.

**EXAMPLE:** Show override for equal() in class java.lang.Integer (transparency)

(Note: Object has several other methods that should never be overridden. These are declared as final in class Object, so the compiler will not allow you to do so even if you try.)

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

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

2. 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.

3. 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.

4. 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.

F. One important consideration in designing a method is the **parameters** needed by the method.

1. 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.

2. Meyer’s principle of narrow interfaces suggests that we should try to find the minimal set of parameters necessary to allow the method to do its job.

   EXAMPLE: Discuss parameter lists for each message in the Session Interaction

G. Another important consideration is the return value of the method.

   1. A question to ask: does the sender of this message need to learn anything new as a result of sending this message?

   2. Typically, information is returned by a message through a return value.

   EXAMPLE: Show examples in Session interaction

   3. Sometimes, a parameter must be used - an object which the method is allowed to alter, and the caller of the method sees the changes.

   EXAMPLE:

   The balances parameter to the sendToBank() method of the various types of transaction - SHOW in Withdrawal interaction. Note that this method has to return two pieces of information to its caller:

   a) A status

   b) If successful, current balances of the account

   SHOW Code for class banking.Balances

H. Just as we use a javadoc class comment to document each class, we use a javadoc method comment to document each method. The documentation for a method includes:

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

   2. A description of the parameters of the method.

   3. A description of the return value - if any.
SHOW: Documentation for course-related methods of class RegistrationModel for Registration labs.

TRANSPARENCY: java source code for these methods, showing javadoc comment.

I. 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.

3. Public constants should also be documented via javadoc

SHOW Documentation for constants of class java.awt.BorderLayout

TRANSPARENCY: source code showing javadoc comments.

III. Preconditions, Postconditions, Invariants, and Design by Contract

A. As part of designing the interface for a class, it is useful to think about the preconditions and postconditions for the various methods, and about class invariants.

1. A precondition for a method is a statement of what must be true in order for the method to be validly called.

EXAMPLE:

As you may have discovered in lab, the remove(int) method of a List collection can be used to remove a specific element of a List. However, the method has a precondition that the specified element must exist - e.g. you can’t remove the element at position 5 from a list that contains 3 elements, nor can you remove the element at position 0 (the first position) from an empty list.

What happens if you fail to observe this precondition?

ASK
2. A postcondition for a method is a statement of what the method will guarantee to be true - provided it is called with its precondition satisfied.

*EXAMPLE:* The postcondition for the remove(int) method of a List collection is that the specified element is removed and all higher numbered elements (if any) are shifted down - e.g. if you remove element 2 from a List, then element 3 (if there is one) becomes the new element 2, element 4 (if there is one) becomes the new element 3, etc.

Note that a method is not required to guarantee its postcondition if it is called with its precondition not satisfied. (In fact, it’s not required to guarantee anything!)

3. A class invariant is a statement that is true of any object at any time it is visible to other classes. An invariant satisfies two properties:

   a) It is satisfied by any newly-constructed instance of the class.

      Therefore, a primary responsibility of each constructor is to make sure that any newly-constructed object satisfies the class invariant.

   b) Calling a public method of the class with the invariant true and the preconditions of the method satisfied results in the invariant remaining true (though it may temporarily become false during the execution of the method)

      (1) Therefore, a primary responsibility of any **public** method is to preserve the invariant.

      (2) Technically, **private** methods are not required to preserve the invariant - so long as public methods call them in such a way as to restore the invariant before the public method completes.

   c) That is, the class invariant must be satisfied only in states which are visible to other classes. It may temporarily become false while a public method is being executed.

B. An example of method preconditions and postconditions plus class invariants: David Gries’ Coffee Can problem

1. Explain the problem

2. *DEMO*
3. HANDOUT: CoffeeCan.java

a) Note preconditions and postconditions of the various methods

b) Note class invariant

c) It turns out that the question “what is the relationship between the initial conditions and the color of the final bean?” can be answered by discovering an additional component of the invariant.

ASK CLASS TO THINK ABOUT:

(1) Relationship between initial contents of can and final bean color.

(2) A clause that could legitimately be added to the invariant which makes this behavior obvious.

C. Method preconditions and postconditions and the class invariant become the basis of a design methodology called design by contract. In this methodology, the specifications of a class constitute a contract that other classes can rely upon.

1. The class promises that if its methods are always called with their preconditions satisfied, it will guarantee the postconditions of the methods.

2. The class promises that newly constructed objects will satisfy the class invariant, and so long as methods are applied in accordance with their preconditions, the invariant will remain true.

3. This becomes the basis of an approach to assigning responsibility for software errors:

a) If some code calls a method whose preconditions are not satisfied, the calling code is at fault. The called method has no obligations - it can do anything at all or nothing.

b) If a method is called legally but the postconditions are not satisfied or the class invariant becomes false, then the called method is at fault.

D. Design by contract stands in contrast to other methodologies.

1. Many writers advocate that a called method should check the legitimacy of its parameters to be sure it is called legally, and should
take some suitable corrective action if the call is not legal. Possibilities include:

a) Ignore the problem - obviously bad, but what often happens in practice because the author of a method forgets (or is too lazy) to check a parameter.

b) Print an error message and terminate the program.

c) An approach called **defensive programming** - the called method should attempt to “fix” the error by using some reasonable default value - e.g. if a method requires a nonnegative parameter, and the user calls it with a negative value, then quietly replace the negative value by something like 0 or 1.

d) Throw an exception. This either shifts the burden of responsibility back to the caller (if it catches the exception) or winds up being equivalent to printing an error message and terminating the program.

2. In design by contract, a called method **may assume** that its stated preconditions are satisfied. It does not need to (and indeed should not) test its parameters - if they are illegal then whatever the method happens to do is OK - it’s not the called method’s fault.

3. This requires that each method in a program accept certain responsibilities:

a) Any method that calls another method must ensure that the called methods preconditions are satisfied. If the calling method gets input from a user that could fail to satisfy the preconditions of a method it calls, it needs to check the values and take appropriate action if they are wrong before calling the method whose preconditions would be violated.

b) Any method that is called legally must ensure that its postconditions are satisfied and the class invariant is preserved.

c) Note that throwing an exception under certain circumstances may be made part of the contract - in which case a caller may rely on a method to throw a certain exception under certain circumstances, and may handle that in a suitable way.
IV. Designing the Implementation of a class

A. Once we have designed the interface for a class, including its invariant and the preconditions and postconditions of its methods, it is time to design the implementation. This involves two major tasks:

1. Identifying the attributes

2. Implementing the methods

B. Identifying attributes

1. One task we must perform is listing the attributes each object of a given class must have. To do this, we can ask two basic questions:

   a) What must each object of the class know about itself? (What must it know?)

   b) What objects must each object of the class relate to? (Who must it know?)

2. The first question (what must it know) involves thinking about the responsibilities of the class and what it needs to know to fulfill them.

   a) EXAMPLE: We have seen how the responsibilities of class CardReader (represented in the CRC Cards and interaction diagrams) translate into attributes. This is typically the way we find attributes for controller objects - we look at the interactions for in which it plays a role, and see what it needs to know to do its job.

   b) In the case of entity objects, we may need think about the kind of information the entity represents.

      EXAMPLE: In the video store system, what must a Customer object know? (Note: we’re asking about what the object must know, not about what the human being must know!)

      ASK

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

      (1) EXAMPLE: What must any Transaction object know? (Information common to all transactions, not just one type)
ASK

SHOW : design for Transaction class

(2) EXAMPLE: What additional information must a Withdrawal object know?

ASK

SHOW design for Withdrawal class. Note that, in addition to the attributes just listed, a Withdrawal also inherits all the attributes of a Transaction.

(3) An important consideration in class design when generalization is involved is that attributes need to be put at the appropriate level in the class hierarchy. Basically, any attribute that is common to all the subclasses belongs in the base class; any attribute that is unique to a single subclass, or a subset of subclasses, belongs in the individual subclasses. (However, if there are is an identifiable subset that has several attributes in common not shared by the other subclasses, we may need a new level in the hierarchy. This needs to be considered with caution, however. E.g. we might add a level for transactions that have an amount (everything but Inquiry), but this probably introduces more complexity than it removes!

3. The second question (who must it know) can be answered directly from the associations in the class diagram.

a) An object needs an attribute for each relationship it knows about.

EXAMPLE:

```
  A  --  B  --->  C
```

Each “A” object needs an attribute to record the B object(s) to which it is related. Each “B” object needs an attribute to record the “A” object(s) to which it is related and another attribute to record the “C” objects to which it is related. A “C” object does not need an attribute to record the “B” objects to which it is related, because the navigability on this association is from B to C only. (A “C” object does not know about its “B” object(s)).
b) Normally, this attribute will be either a *reference* or a *collection*.

(1) It will be a reference if the multiplicity at the other end of the association is either 1 or 0..1 - i.e. it must know at most one other object in this particular association. (The reference will be null if the multiplicity is 0..1 and there is currently no object with which it is associated.)

(2) It will be a collection of some sort if the range of the multiplicity at the other end of the association is greater than 1. In this case, we must choose what kind of collection is most appropriate.

(a) Often, we will use one of the standard Java collections, based on how we will be accessing the elements:

   i) A HashSet, if there is no inherent order to the collection.

   ii) A TreeSet, if there is some natural basis for organizing the associated objects in “alphabetical” order. (Note that a TreeSet constructor takes as a parameter a Comparator object that knows how to figure out this order.)

   iii) An ArrayList or LinkedList if we need to control the sequence of the associated objects (e.g. “first-come first-served”).

   iv) A HashMap or TreeMap if we need to be able to access the associated object by some key - e.g. a name or an id number.

(b) If the number of objects at the other end is fixed, or has a small fixed upper bound, an array or even distinct variables may be appropriate.

*EXAMPLE:* Consider developing a “dungeon” sort of game in which we have Rooms linked together. Suppose each Room is allowed to potentially have neighbors in each of the four directions (north, east, south, west). We could implement this in a number of different ways:

*ASK*

i) We could use four variables called northNeighbor, eastNeighbor, southNeighbor, and westNeighbor.
ii) We could use an ordinary array, with element 0 being, say the north neighbor (null if none), element 1 being the east neighbor, etc.

iii) We could use a list, with the list elements being stored in the order north, east, south, west (and a null being stored in a position if there is no neighbor this way.) (Note that this is quite similar to the array representation, since we will use indices 0, 1 etc. to access the neighbor in a specific direction.)

iv) We could use a map, with the strings “North”, “East”, “South” and “West” serving as keys and the corresponding rooms being the values.

v) We could not use a set. Why?

ASK

A set is inherently unordered. In this case, it is vital to know whether a given room is the north neighbor or the south neighbor.

vi) In this case, the simple array may be the best choice, assuming a run time variable (the direction in which the player wants to move) will be used to select a Room. The various collections introduce additional overhead that doesn’t really do anything for us here.

(c) What do we do if we have an association that requires an association class?

i) The association class object will have a simple reference to each of the objects it is associating.

ii) Each of the objects participating in the association will have some sort of collection of association class objects.

EXAMPLE: The EnrolledIn association from the Registration system labs
This association involves attributes in three classes: Course, Student, and EnrolledIn:

// In Course: map keyed on student name; value is
// an EnrolledIn object
private TreeMap enrollments = new TreeMap();

// In Student: map keyed on course id; value is
// an EnrolledIn object
private TreeMap enrollments = new TreeMap();

// In EnrolledIn. Note that each EnrolledIn object
// relates to one Course and one Student:
Course course;
Student student;

(d) Note that this part of the design is driven by the class
diagram - if the class diagram is done well, then identifying
association variables is straightforward. The only tricky part
may be deciding what type of collection to use.

4. Ordinarily, attributes should be declared as private (“-” in UML).
However, if a class is the base of a class hierarchy, and subclasses have
legitimate need to access the attribute, then it may need to be declared
as protected (“#” in UML).

C. Once we have designed the implementation of a class, of course, we then
need to implement its methods.

1. If the class has been designed correctly, and each method has been
specified via preconditions and postconditions, this is usually
straightforward. (Title of talk at OOPSLA Educator’s symposium in
1999 - “Teach design - everything else is SMOP (a simple matter of
programming)”).
2. 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.

D. 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.

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