CPS122 Lecture: Detailed Design and Implementation

Last revised March 3, 2017

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 test-first development
4. To review javadoc class documentation
5. To introduce method preconditions, postconditions; class invariants.

Materials :

1. Diagrams for UMLImplementation labs and class `EnrolledIn`:
   a. Overall class structure
   b. CRC Card for `EnrolledIn`
   c. Sequence Diagrams for Grade, Course Report, and Student Report use cases
   d. Detailed design for `EnrolledIn`
   e. Netbeans project with skeleton of class `EnrolledIn` (methods not implemented) and no JUnit tests.
   f. Netbeans project with completed tests.
   g. Netbeans project with completed tests and code.
2. Javadoc documentation for UMLImplementation labs classes and projectable of source code for class `RegistrationModel` (class comment+selected methods only).
3. Projectable of source code for team project SimpleDate class
4. Javadoc documentation for java.awt.BorderLayout (online) and projectable of source code for constants
5. Gries' coffee can problem - demo plus handout of code

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 next labs you will be doing - a system that manages student registrations in courses - the class `EnrolledIn`

1. Project overall class structure

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`

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

   PROJECT sequence diagrams for Grade Student, Course Report, and Student Report use cases.

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.

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: Show that `EnrolledIn` is associated with Course and Student in class diagram

   Show course and student in detailed design

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: Show CRC card for `EnrolledIn` again. Since it must keep track of a student's grade, it needs a grade instance variable

PROJECT detailed design and show in grade

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 diagrams in which `EnrolledIn` is involved.

   PROJECT each and then show methods in detailed design

   Observe that 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)

   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. **Unit Testing / Test-First Development**

A. As you may recall, we saw earlier that coding comprises about 1/6 of the total effort on a project. How much effort do you think testing comprises (or at least should comprise)?

   ASK

   About 50%!

   (The fact that there's so much buggy software out there suggests that what should be done and what actually is done are not the same thing!)

B. We will talk more about testing later in the course, but for now note that there are actually several different kinds of testing that need to be done.

   1. Unit testing tests the individual pieces of the system - e.g,, in an OO system, the individual methods. Any errors discovered by this process are fixed before development proceeds. We will look at an approach to doing this shortly.

      This should be done during implementation.

   2. Integration puts several (already tested) units together to test how they work together. Any errors discovered at this time are likely the result of a misunderstanding about the interface of one of the methods.

      EXAMPLE:

Suppose a certain method is required to compare two objects and return true or false based on their relative order in a sorted list. Assume this method is used by another method that actually sorts the objects.

Suppose the author of the method understands the expectation to be that the method returns true if the *first* object belongs *before the second*, but the author of a sorting method that uses it assumes that it returns true if the *second* object belongs *before the first*.

Both methods would test successful during unit testing (based on their author's understanding of the interface between them.) However, the error would show up in integration testing with the output being backwards!

This, too, should be done during implementation.

3. System testing tests the overall functioning of the system relative to the specifications (the use cases). By its very nature, this can't be done until implementation is at least nearly complete.

4. Regression testing is the repetition of tests that have already been passed after a fix/change has been made to be sure that the change has not broken another part of the system.

   a) This sort of testing is done as needed during implementation

   b) It is also done during maintenance.

   c) To facilitate this, tests are automated where possible. (We will shortly see an example of this with unit tests).

5. The "50% of effort" rule of thumb includes all kinds of testing that occur during implementation, of course.

C. Today, we are going to focus on a type of testing done as a class is being implemented: unit testing.

1. Many software development organizations actually use an approach to this known as test-first development.

   a) The idea is this. Before writing the code for a method, one first writes the specification (perhaps in the form of comments) and write code to test the method before the method itself is actually written.

   b) This may seem counter-intuitive - but the idea is that writing a specification and a test helps clarify what is to be done before actually doing it.

      Experience has shown that this actually significantly improves code quality and effort.

   c) Last semester those of you who were in CPS121 saw an approach like this using pydoc and pytest. We have already seen that there is a similar facility for documenting Java code known as javadoc, and we will shortly see that there is a similar facility for unit testing Java cide known as JUnit.

   d) You will use this approach during labs 9-11, and will also be expected to use it on your team project. In fact, neither the TA nor I will give you any help with code unless we first see your specification in the form of prologue comments and - in the case of model classes - your JUnit tests.

2. As an example of this, consider the class `EnrolledIn`.

   a) Open NetBeans project that contains just a skeleton of this class.

      PROJECT skeleton of class (comments and method prototypes).

b) Before we actually begin writing code for these methods we can develop mechanisms for testing them.

c) Demonstrate creating JUnit tests using NetBeans - then show contents of created file.

d) Open project with completed tests and show tests.

Note that it was necessary to "stub out" methods that return values to allow code to compile

e) Demo implementation of a couple of methods and show results in testing

f) Open project with completed tests and code and demo testing.

g) Demo some errors and show how JUnit catches:

(1) Omit assignment of grade in setGrade()

(2) Omit initialization of student in Constructor

(3) Omit initialization of grade in Constructor

V. 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:

      a) Show online documentation for UML Implementation Labs classes

      b) *PROJECT*: 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 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.

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

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

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

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

      EXAMPLE: Show examples in Session interaction

   c) 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:

(1) A status

(2) If successful, current balances of the account

SHOW Code for class banking.`Balances`

4. 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:

   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.

      SHOW: Documentation for course-related methods of class `RegistrationModel` for UMLImplementation labs.

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

   d) Netbeans can help to generate these

      (1) Type method prologue

      (2) Type /** on line before method

      (3) Press enter key (on keypad - not return)

      DEMO

(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.

F. 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

G. 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`

   PROJECT: source code showing javadoc comments.

VI.Preconditions, Postconditions, and Invariants,

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 discover 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 new element 3, etc.

   Note that a method is <u>not</u> 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 <u>public</u> method is to preserve the invariant.

      (2) Technically, <u>private</u> 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.

VII. Some Final Thoughts on Detailed Design/Implementation

A. If a 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)").

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.

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