CS211 Lecture: Design Quality; Cohesion and Coupling; Packages

Last revised September 25, 2003

Objectives:

1. To introduce the notion of design quality, tradeoffs, and some principles of quality design
2. To overview the different types of design needed for a system
3. To introduce designing for high cohesion and low coupling.
4. To introduce UML package diagrams
5. To show how UML packages are similar to and different from Java packages

Materials:

1. Transparency of Table 9.1 in the text
2. Worksheet with cohesion and coupling examples
3. Transparency of Table 9.2 in the text
4. Source code for ATM Example system class Transaction (online)
5. Source code for ATM Example system class AccountInformation (online)
6. ATM Example System Withdrawal interaction diagram (online)
7. ATM Example System Package diagram - two forms: with package contents listed, and without (online, draw on board)
8. ATM Example System Class diagram (already handed out)

I. Introduction

A. To wrap up our discussion of Design, we will talk a bit about the topic of quality design.

1. The issues we will discuss are ones that need to be considered *throughout* the design process. We are covering them now for pedagogical reasons - if one didn't think about these issues until after having done the design, it would be like building a bridge out of corrugated cardboard and then realizing that this wasn't a good idea!

2. At an OOPSLA Educators' Symposium I attended several years ago, the title of the keynote address was "Teach Design - Everything Else is Just SMOP" (where SMOP was defined in a footnote as "a simple matter of programming.".

B. It is important to recognize that the design process involves making <u>design decisions</u> - i.e. choosing among various possible alternatives the alternative that is best for a particular situation.

1. That is to say, instead of simply grabbing the first approach to solving a problem that comes to mind, one needs to identify a variety of different approaches and then choose between them based on the requirements for the particular situation.

2. In many cases, there is no such thing as "the one best design" - rather, there are "best designs" for different sets of requirements.

   a) Often times, a key issue in choosing among design alternatives is <u>tradeoffs</u> - it is possible to do better in one area in exchange for doing less well in another.

   b) The book discussed the idea of using project priorities to choose among design alternatives for a given project. We won't pursue this further - the discussion in the text is pretty self-explanatory.

   c) Even with a clear sense of project priorities, there may be no clear "winner" in terms of designs for a system - but at least we can choose a front-runner rather than a definite loser!

C. It is important to recognize that, in any given system, there are actually several different <u>types</u> of design needed:

1. Architectural design - the overall structure of a system.

   a) Often, sophisticated systems are built as *distributed systems* - with different portions residing on different computers, perhaps at different sites. We will discuss this topic later in the course.

   b) Even if all of a system resides on a single computer, we need to think about the overall architecture of the system. (This is also true of each of the subsystems of a distributed system, of course.)

   c) The topic of software architecture is a huge one, and not one we can delve into in depth now. However, it is useful to note that there are certain *architectural patterns* that are worth considering for a given system. The book discussed several possibilities.

   ASK

   (1) Multi-layer architecture

   (2) Transaction-processing architecture

(3) Pipe-and-filter architecture

(4) Various distributed architectures (Client-Server, Broker)

(5) One architectural pattern we have used extensively is the <u>Model-View-Controller</u> (MVC) architecture.

2. Class design - the design of individual classes.

a) We have looked at some aspects of this already

(1) The relationship of a class to other classes - associations, generalization, implementation, and dependency (represented by various kinds of connections between classes in a class diagram.)

(2) The dynamic behavior of a class - represented using tools such as interaction diagrams and statecharts.

b) We will consider the detailed design of classes in more depth in a subsequent lecture.

3. User interface design - we talked about this a couple of weeks ago.

4. Database design - we will spend some time on this toward the end of the course; it's also a topic in the Database Management Systems elective (CS352)

5. Algorithm design - largely a topic in the Computational Structures and Algorithms Course (CS212)

6. Protocol design - the design of protocols systems use to communicate with one another (e.g. over a network). We will say something about this in this course - but ultimately it's a topic for the Computer Networks elective (CS372).

(If you get the idea that design is something that pervades the computer science curriculum, you've gotten the right idea!)

D. The book identified 11 principles of good design.

1. I will not attempt to rehash them here - the discussion in the book is quite clear, and you should study this discussion carefully.

2. We will spend some time, though, on two: cohesion and coupling. We should strive for designs that exhibit high cohesion and low coupling.

II. Cohesion

A. We say that an entity is *cohesive* if it performs a single, well-defined task, and everything about it is essential to the performance of that task.

(Note that we have defined cohesion in terms of an entity. The cohesion test can be applied to classes (as we will discuss here); but it can also be applied on a lower level to the individual methods of a class (each of which should be cohesive) or, on a higher level, to a package of related classes or an overall system or subsystem.)

1. An important goal in design is to try to ensure that each entity we design (class, method, system) exhibits the highest possible level of cohesion.

2. A good test for cohesion is “can I describe the purpose of this entity (class, method, etc.) in a short statement without using words like ‘and’?” (This is one of the benefits of writing a prologue comment for a class or method before you write the code - it helps you to think about whether what you are about to produce is truly cohesive.)

B. Over the years, software engineers have identified various sorts of possible cohesion, which can be ranked from least cohesive (bad) to most cohesive (good).

1. Unfortunately, different writers have different lists of types of cohesion. Of course, the ultimate task is not to determine what kind of cohesion a given entity exhibits, but rather to produce the most cohesive entity possible.

2. The book discussed a number of possibilities

TRANSPARENCY - Table 9.1

3. Some other writers use the term “Informational cohesion” for what our author called “Communicational coheison” (and define “Communicational cohesion” a bit differently). Also, some argue that this is every bit as strong a form of cohesion as “Functional cohesion” - i.e. it belongs tied for best. As noted in the book, well designed classes will exhibit such cohesion.

4. Some writers define a couple of other weak categories - representing designs we should avoid creating, but helpful to recognize when we create systems that exhibit them.

   a) Logical Cohesion - a module performs a set of related actions, one of which is selected by the caller in each case

   b) "Coincidental cohesion" - used to describe a module that performs multiple, unrelated actions.

C. Distribute, discuss handout with examples of different types of cohesion. (Just the first page on cohesion)

1. My answers:

   a) F - the only thing these operations have in common is the fact that they're done at the same time.

   b) A - this is a good example of a module that performs a single, clearly-defined task

   c) E - the cohesion arises from the need to perform steps in a certain order, and in each case the output of one step is input to the next.)

   d) B - this is a good example of a cohesive class

   e) I - the lack of cohesion here is about as bad as it can get!

   f) H - Probably it would be better to have separate drawCircle() etc. methods - which would also make it easier to extend the functionality.

2. There are a couple of kinds of cohesion that cannot really be exemplified in a handout like this, because they are usually found only at the level of packages or (sub) systems.

   a) Utility cohesion - a good example: the `java.util` package.

      (SHOW JDK documentation for package)

   b) Layer cohesion - a good example: the structure of the `java.io` package

Recall that we saw last year that the major classes in this package are organized into three layers

Classes that deal with data at higher levels - e.g. DataInputStream, DataOuputStream, PrintStream, BufferedReader, PrintWriter
Classes that deal with data at the level of individual bytes or characters - e.g. InputStream, OutputStream, Reader, Writer
Classes that represent physical representations for data - e.g. File, Socket

D. Sometimes, we discover that a class lacks cohesion, and this leads us to partition the class into two or more new classes.

1. A tip-off that a class might benefit from being split is the fact that the attributes fall into two or more distinct groups, with some methods using attributes from one group but not the other(s), and some methods using attributes from the second group but not the first.

2. *EXAMPLE:* In the ATM example, we have identified separate subclasses of a common base class Transaction for each type of transaction: Withdrawal, Deposit, Transfer, and Inquiry. Suppose, instead, we chose to define just a single Transaction class, and include a type field as an attribute.

   The class might have attributes like the following:

```
int type;
int fromAccount; // Used only for withdrawal, transfer, inquiry
int toAccount; // Used only for deposit, transfer
Money amount;  // Used only for withdrawal,deposit,transfer
```

   Note that, for any given case, only a subset of the attributes is actually relevant. This would be a bad design choice, because the resulting class would be less cohesive

E. Sometimes, on the other hand, we discover that closely related responsibilities are scattered among several different classes, and we are better off creating a new class to pull them all together. This is related to a principle that Bertrand Meyer calls the “single choice” principle:

whenever a software system must support a list of alternatives, one and only one module should know their exhaustive list.

1. The rationale for this is that, if we need to add a new alternative, we need only need to make changes in one place - rather than throughout the system.

2. *EXAMPLE:* The ATM Example system allows a user to choose one of four alternative types of Transaction (Withdrawal, Deposit, etc.). Clearly, it could be possible to add another option to the system at a later date. This argues for putting the knowledge about the different alternatives in one place.

   In the ATM Example, this is handled by the class Transaction, as follows:

   SHOW source code for class Transaction - note list of options (static final field TRANSACTION_TYPES_MENU) and makeTransaction() method. These are the only things that would need to be changed to add a new type of Transaction - besides, of course, creating a new subclass of Transaction to actually handle it.

   NOTE: The partitioning of Transaction into Withdrawal, Deposit etc. we discussed above does not violate the single choice principle - since each new subclass knows about only one alternative, not the whole list.

3. *EXAMPLE:* The various transactions in the ATM Example System allow a customer to choose which account to withdraw money from, deposit money to, etc. This means that there must somewhere exist a list of types of account (checking, savings, etc.) The possibility of adding a new type of account to the list argues for keeping knowledge about the alternatives in one place.

   In the ATM Example, this is handled by a class AccountInformation, which was added *after* the CRC cards etc. were done.

   SHOW source code for class AccountInformation, with list of account names and abbreviations (for receipts). These are the only things that would need to be changed in the ATM system itself to add a new type of account - though obviously the bank software would also undergo some changes.

III. Coupling

A. Coupling is a measure of the extent to which a class depends on other classes. A class depends on another class if it:

1. Has an association with that class (with navigability toward that class if unidirectional)..

2. Generalizes or realizes that class

3. Has a dependency on that class

B. Of course, there are various *ways* in which a class can depend upon another class.

1. It may use services provided by the other class - i.e. it may depend only on the other class's *interface*.
2. It may use internals of the other class (i.e. may access its variables directly) - i.e. it may depend on the other class's *implementation.*

C. We want to minimize coupling, because:

1. If a class B is coupled to a class A, and we want to build a system that reuses class B, then we must also include class A in the system, whether or not it would otherwise be needed. (This is an issue whenever there is coupling)

2. If a class B is coupled to a class A, and class A is modified, class B ma need to change as well.

   a) If B depends only on A's interface, then only a change to A's interface can potentially affect B.
   b) However, if B depends on A's implementation, any change to A has a potential impact on B. This is another reason for <u>not</u> depending on the implementation of another class. (This is almost always avoidable - and should be avoided - except, perhaps, in the case that B inherits from A and uses protected features of A.)

3. Of course, we can't eliminate coupling altogether, nor would we want to. An object-oriented system is a society of cooperating objects, and cooperation implies some coupling. Our goal is to The goal, however, is to avoid *unnecessary* coupling.

D. Bertrand Meyer gives two rules of thumb related to coupling:

1. The “Few Interfaces” rule: every class should communicate with as few others as possible.

2. The “Small Interfaces” rule: if two classes do communicate, they should exchange as little information as possible. (The greater the amount of information exchanged, the higher the likelihood that a change to one class will require a change to the other.)

   EXAMPLE: The ATM Example uses a Message class to encapsulate details of a Message from a Transaction to the bank. This is used as a parameter to the sendMessage() method of NetworkToBank - in place of a long list of specific parameters. This means that only the sending Transaction and receiving bank need to be aware of the details of the structure of a message - not the intermediate class(es) comprising the network.

3. The “explicit interfaces” rule: whenever two modules A and B communicate, this must be obvious from the text of A and B - we want to avoid hidden side effects.

E. As was the case with cohesion, software engineers have developed some categories of coupling.

TRANSPARENCY: Table 9.2 in the text. Note: the first five entries in the table are the ones most commonly discussed.

F. Once again, we will use a worksheet to see some examples of these

1. Handout - page 2

2. My answers:

   a) D (The java equivalent of the global variable found in some programming languages is a static variable of some class)

   b) B (because we pass in the whole person object, not just the one piece of information we need - hence the calculateAge method needs to know about the interface of Person - e.g. that it has a method to access the individual’s birth date (`getBirthDate()` or the like. This coupling would not occur if the parameter to the method were the birth date - then it could be applied to any birth date, not just the birth date contained in a Person object.)

c) E (It's not actually possible to construct something this bad in Java!)

d) A - there is only one parameter, and it's a simple data item

e) C - one of the parameters is a flag that controls whether the method prints today's date.

IV. Packages

A. We have already noted that one key principle in design is "divide and conquer" - break a large system into smaller (and more manageable) pieces. When a system gets large, it is generally advantageous to break it up into subsystems. The UML facility for doing this is known as a *package*.

B. A UML package is represented by a "file folder" icon. It has a name (shown on the tab), and can contain many kinds of things, including:

1. Classes

2. Interfaces

3. Use Cases

4. Interactions

5. Diagrams

6. Other packages

C. A UML Package diagram shows the various packages and the dependencies between them. Basically, a package B depends on a package A just when some entity in package B depends on some entity in package A. A package diagram can explicitly list the various items that are in each package, or it can just show the packages and their dependencies.

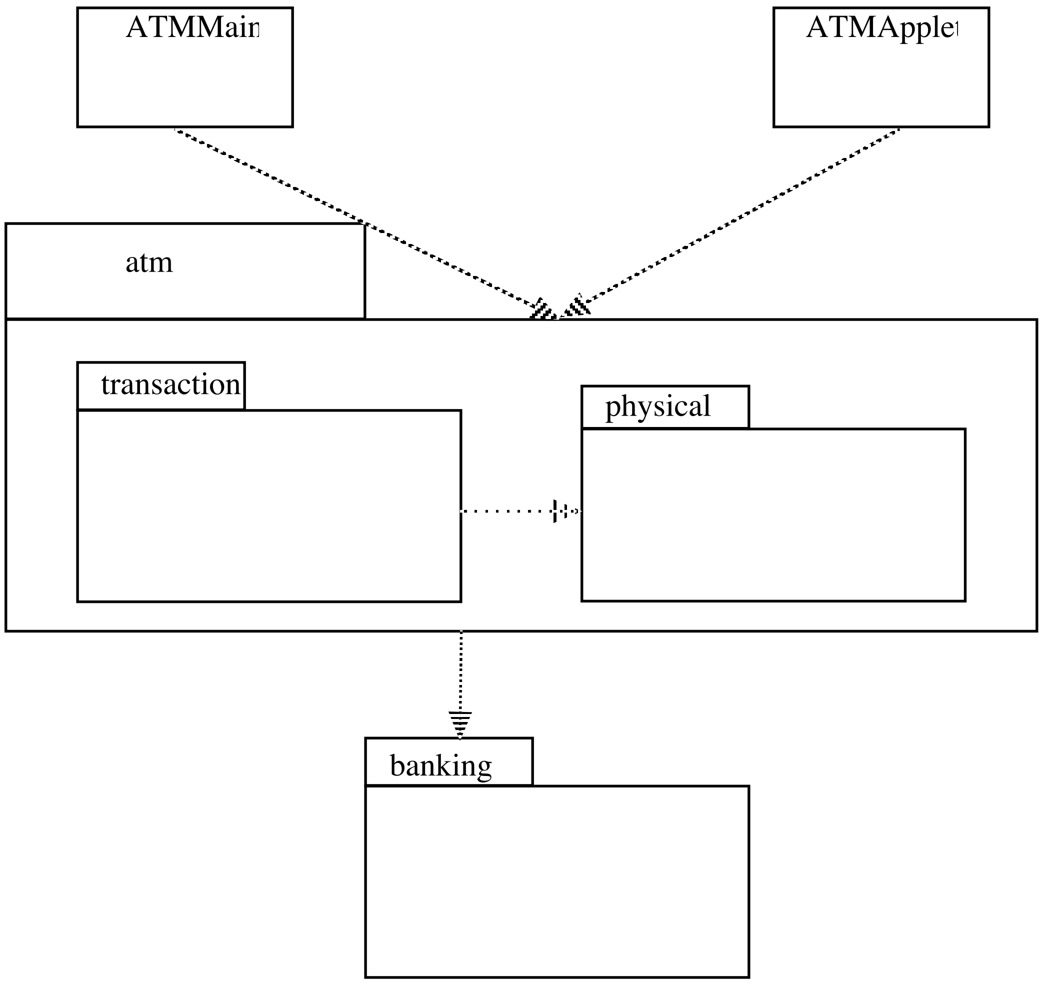
1. *EXAMPLE:* ATM Example System Package diagram.

a) Notice the convention that was used in developing this structure:

• Classes that pertain to banking in general (not just to the ATM system) go into the banking package. (These are classes that arise from domain analysis, and may be reusable in other systems in this domain)

• Classes and packages that are specific to this application go into the atm package. (These are classes that arise from application analysis.)

2. *EXAMPLE:* Redrawing of the above to show the relationships, without explicit listing of contents:



D. Packages can be used at many stages in the development process.

1. If a system has many use cases, it might be desirable to organize them into packages. The interactions that realize the various use cases *might* go in the same package as the use cases, or in a parallel package structure.

2. Alternately, if various interactions involve distinct sets of objects, then the interactions that involve a given set of objects might all be put into a package together.

3. If a class diagram can be partitioned in such a way as to create groups of classes that have relatively high coupling to each other but relatively low coupling to the rest of the system, then those groups of classes might be turned into packages.

   EXAMPLE: Compare class diagram and Package diagram (with contents listed) for the ATM Example system

E. A package introduces a namespace. That is, the full name an entity within a package is packagename::entityname. (The use of :: is a convention adopted from C++)

1. *EXAMPLE:* Given the above diagram, the full names of the various entities in the banking page look like:

   banking::AccountInformation
   etc.

2. Of course, if a package is contained within another package, then the full name would be

   OuterPackageName::InnerPackageName::EntityName

   (This can go on to any depth of nesting).

   EXAMPLE:

   atm::physical:CashDispenser
   etc.

3. The advantage of including the package name in the full name is that it reduces the danger of name conflicts. It is important that every component of a system have a unique name. In a large system, with

many people working on it, this can become a significant problem. However, if packages are used to break up the name space, two entities in *different* packages can have the *same* name, because their full names are different.

F. Note that UML packages are similar in many ways to Java packages.

1. In both cases, the full name of an entity is the name of its package followed by its own name. (Java uses ".", whereas UML uses "::" as a separator).

2. However, Java packages can contain only classes, interfaces, and other packages. UML packages can contain other entities as well. Thus, a UML package is really a more general notion.

3. Typically, when UML packages are used to group classes, their implementations will reside in Java packages having a similar structure. Note, though that while it is conventional for UML package names to begin with upper-case letters and use mixed case, Java package names are typically all lower case. (I have used the Java naming convention for the ATM package diagram, though.)

   EXAMPLE: Java package structure of ATM Example

```
class ATMApplication
class ATMApplet
package atm;
        class ATM
        class Session
        package atm.physical
                class CardReader
                class CashDispenser
                etc.
        package atm.transaction
                abstract class Transaction
                class Withdrawal
                etc.
package banking
        class AccountInformation
        class Balances
        etc.
```