Case Study – PA4 Design and Solution

**The first step in solving a problem is to understand the problem.**

This problem is assigned as [PA4](../PA4.html). For this problem, the main task is to read a file (the name comes in as a command line argument), validate that the data in the file is correct, and then display a transcript based on the file data. There is a header record which identifies the student and 0 or more detail records. There can be 0 records because the problem requires that there be a header, but does not require that there be any detail. For any detail, we must validate that the course records contain valid grades, but the rest of the validation is based on the field values themselves. One requirement in the assignment is that we use an abstract DetailRecord class which implies that we will need to use inheritance to create one or more individual detail record class(es).

**The next step in solving a problem is to design a solution.**

1. One solution (ugly) would have us create a big hunk-o-code in main. This problem could be solved in this way, except for the DetailRecord requirement. Solving it in main, though, would violate all kinds of good programming practices, such as breaking a problem down into its subcomponents and using reusable code components, among others. Such code would be hard to create, hard to test and debug and hard for the instructor or others to read. See Figure 1.



Figure

1. So, I begin to break down the problem. There are two primary tasks:
* Reading and validating the file.
* Producing the transcript.

So another possible design is to have an IO class with two methods, checkFile() and printTranscript(). Then a Transcript class would store the transcript information and do the tasks related to actually calculating hours and GPA and producing the transcript.

This design has a couple of advantages for this problem. One, it isolates the validation steps and the production steps, and gives us a Transcript class which can be used to extend this application (say to build multiple Transcripts). Transcript needs a constructor of some kind and a method to get the transcript detail. See Figure 2.



 Figure

1. So we are beginning to build a design that isolates the two primary sections of the code. Somehow, we still need to be able to do the validation and to build the transcript itself. At this point, I focus on the Transcript class and push that out to deal with the different records, forgetting for now how I will deal with the file input.
2. The Transcript is composed of a Header and Details. We also know that the Transcript will need to interact with the Grade class (or children will). So as we refine, we get this overall structure…again not worrying about the details.



Figure

In this version of the design, we are not concerned yet about the details. We know that they will be some kind of collection that supports what we need to do, namely calculate GPA information and display transcript details. For now, we will have a HeaderLine class, a DetailLine class and the GradeInfo class.

1. Let’s look further at the DetailLines. We know from the statement of the problem that we must use DetailRecord and that it must be abstract. We also know from the problem statement that there will be three different types of detail records. So it is logical to think of three detail classes inheriting from the DetailRecord class. For now, lets name them D1, D2, and D3. We also know that the details “have” grades and so we connect the Grade to the DetailRecords. We also may use grade information in Transcript so for now, we will also say that Transcript uses the Grade. We also insert into the DetailRecord class all of the required methods. See Figure 4.



Figure

1. Now it is time to work through the details of how this will all work (Transcript portion only). We’ll worry about the file shortly.
2. Looking at the detail records, we want to decide on attributes for our DetailRecord class and individual Dn classes. The attributes in common include: year, term, course, hours. These go into DetailRecord.



Figure

1. Then we have individual attributes that vary by detail type. D1: score and testType, D2:school, D3: courseDescription, grade.



Figure

1. We need to look at the methods required in light of what we need to accomplish and what we have been given to start with. So what can be the same, and what needs to be different. I decide that since the hours are the same (it is in our abstract class), that we can define the getCreditHours() method in DetailRecord. But the rest of the abstract methods really need to be in the children, since they “know themselves” and can determine their own graded hours and quality points, etc. So, we will define those methods as methods of the individual classes. That also means that the DetailRecord does not need access to Grade…this will be used by individual detail classes.
2. Now turning to Grade. Grade needs the grade representation, quality points, and whether or not the grade is a graded grade (contributes to GPA) or a successful grade (contributes to earned hours). And we need methods to get the various values that we need to use in the details. See figure 7.



Figure

1. Now I look again at the methods in DetailRecord and the individual classes. To me, there seems to be a lot of duplication of effort. If DetailRecord can get the grade from the children, then it can figure out for any of them what the quality points, earned hours, etc are. I decide to refine and only include a getGrade and getDescription methods in the children and make the rest of the methods concrete.
2. I am pretty happy with the current design of the detail. See Figure 8. The methods are small, there is limited duplication, and this should be manageable during implementation.



Figure

1. Next to the Header: This one is simple…we have three attributes, name, major, and graduation date. We may need get methods, but for this task, if we can simply produce these in the right format for the transcript, we could simply create a getDescription method. The downside is that it is specific to the task of generating the transcript, but that is all we need to do at this point. Perhaps calling it getTranscriptHeader gives us a better way to use this into the future. This is what I do. See Figure 9.



Figure

1. Okay, now we have the Transcript itself pretty well defined. It is time to turn our attention to the IO task.
2. In IO (abbreviation for the actual class name of TranscriptIO) we have three main jobs to do.
	1. Check the file for validity
	2. Create a “Transcript” from the file
	3. Print the transcript.
3. Starting with step a, Check the file for validity, we will need the alleged name of the file. If the Driver program passes the command line arguments to IO during creation, then we could have IO do all of the checking. Alternatively, we could check for the arguments in the driver program which initially sees them and which might know that to create a TranscriptIO object we would need a String. Considering both alternatives, I decide that IO will do all of the validity checking and that the Driver will simply pass the command line arguments to TrascriptIO’s constructor. If there are none, IO will deal with that situation.
4. So, the constructor takes in the command line arguments and will store it in an attribute for later use. The check file for validity then will have several steps. If any of the steps fail, we return null to the driver so that it knows that there will be no transcript. IO will print each of the messages. The operations in checkFile will be as follows in this order:
	1. Check to see that there are command line arguments
	2. Check to see if the first argument is a file that exists
	3. Check to see if the first argument is a file that can be opened
	4. Check to see if the first row is an H record
		1. H must have 2 or 3 tokens
	5. At this point we need to shift gears and check each Detail row for validity.
		1. Row must begin with one of D1, D2,D3
	6. Must check the row for the correct number of tokens
		1. D1 must have 6 tokens
		2. D2 must have 5 tokens
		3. D3 must have 6 tokens
	7. Numeric data must be numeric in the row
	8. Hours must be <0 or >12
	9. Grades must be from the grade list
5. Some of these are relatively simple: Steps a – d can easily be done within a single method. Any failure will result in a return of a null Transcript value and stoppage of the program.
6. We also must consider that we need to produce a Transcript. Do we want to preprocess the file, then build the Transcript or process the file and build the Transcript as a second step. Some possible design alternatives:
	1. checkFile simply returns a Boolean value. This value will indicate whether the file is valid and then we have another method to actually make the records.
	2. checkFile can return a Transcript. We check the detail and build the transcript itself as a single task. It is a big task however.
	3. checkFile could call a preprocessor as a separate method, then create the transcript if the preprocessor returns a good value. The creation of the transcript would also be a separate method. The method will return a Transcript object or null depending on what the preprocessor found. The preprocessor should return the good file to the checkFile method and this should be passed to createTranscript.
7. Considering the three, I kind of like the last alternative. The others are valid and can work, but the third breaks the problem down into its two components and simplifies the process. I make that change in the IO class. Figure 10 shows that class so far.



Figure

1. preprocessFile then will have all the validity checking. It will be a series of steps with the final step being a loop. Considering each of the tasks, I decide that the method will directly check all of the “global” file issues, but that as we check each line we will call a method specific to the type of row we are dealing with. So that would be three checking methods added to IO.
2. As I begin to put them into IO, I consider the kind of checking that needs to be done. Grades, hours, number of tokens, etc. That checking will also need to be done in the individual detail classes too (since they must protect themselves from bad data). Perhaps instead of checking each detail record in IO itself, it can call a method in the Dn classes. This is what I decide to implement…each Dn record will have a method to check a String to see if it is of the correct form and format. Then I can use it in IO as well as when we construct a D1 record itself.
3. And, I also consider the fact that when we make a transcript, we will also need detail lines. Perhaps the individual details should “make themselves” from a String. So now we have two methods to add to each Dn class, a checker and a builder. They now look like this (see Figure 11):
4. Figure
5. Creating the transcript is also a job we need to do in IO. So the Transcript itself has the Header and details. I decide that the Transcript should consist of one HeaderLine and 0 or more detail lines. The design of the attributes should be an ArrayList for the detail. A constructor should take in a Header. So we have to also have a checker and creator in the HeaderLine class as well. After we have the basic transcript, details will be added via a method that takes in a DetailRecord type.
6. So now, the basic design is done. It is detailed enough that I can begin implementation anywhere, but most likely I will begin with the Grade and detail lines and work up from there. The final design looks like what you find in Figure 12.



Figure

**The next step is to test the design.**

I run through the entire design again, from main through the end, checking parameter passage and the other aspects of the interaction of the methods. It appears that all is okay at least to begin the coding process. I notice that we have Transcript as an attribute of IO as well as a return value for the checkFile method. It might be cleaner if IO does operations on Transcripts but does not actually contain a Transcript. I think that this is the way we should go and make that minor change.

Likewise, if we pass in the command line arguments to the check method, we don’t really need to store it, since it is only used there. So I remove the attribute and the constructor, since we only need a default no-arg constructor.

Since the IO class does not contain data, it could technically be a class with only static methods. For now, I will keep the methods non-static and create an IO object in main.

**The next step is to implement the design**

Looking at the design, I decide to start on the lowest level classes. First the Grade class, then, since they are all related, the DetailRecord classes. These are used by other classes.

I also build a Tester that I can use to test each of the classes. I test Grade by printing its values.

In looking at the design, I see that DetailRecord has no constructor. But I will need one if I am to load the values for its children. I build one and make it protected since no one else can build a DetailRecord.