CS193J Handout #7
Winter 2001-02

Design strategies for hw1

(This handout was created by Julie Zelenski) In grading the projects this quarter in CS193J, we are going to de-emphasize design and focus on functionality. Although some part of the grading will look at your design, the majority will be dependent on functional testing. This is not in any way an indication that we don't think that good design is important — on the contrary, thoughtful up-front planning and careful object design are probably the best things you can do to ensure your success on the functional requirements. However, this is not a design class and we have much Java material to cover and thus will only be able to touch on issues of design as time permits. It seems unfair to extensively evaluate you on topics that we won't focus on. Design can be such a personal endeavor and so many variations that are equally valid, it is a complex and subjective process to attempt assigning scores in this area and can introduce inconsistencies into the grading system.

I made this decision based on feedback from previous 193J students and TAs. There was a general sense that our design critique was valuable and welcomed, but having it prominently included in the grading created a sense of unease about the fairness and reasonableness of our expectation about students' design skills. So our arrangement this quarter is that we will focus on functionality and endeavor to be very precise about the requirements so that everyone feels there is a clear and fair expectation. We will try to give some feedback about design on a fairly coarse scale as well but much less weight will be given to this area.

But if you were hoping for more comprehensive input from us about design, we don't want you to feel short-changed! Feel free to ask questions in office hours or e-mail about various approaches and tradeoffs. If you'd like, bring your program to any of our office hours (especially the less-traveled ones right after an assignment was due and before the next one is pressing) and we can take time to talk through what we see as the strengths and weaknesses of your work. Discuss thorny issues with more than one staff member— you'll find that there is no universal sense of what is the "correct" answer in many complex situations and being exposed to a variety of perspectives can broaden your own experience.

I think this arrangement makes the right tradeoffs for this class and I hope that you find that it works for you. As always, if you have some feedback, we're open to suggestions.

HW1

The rest of this handout is a road map with suggestions about how to work through the first homework project. It sketches out some strategies for how to break down the task at hand, the classes to consider building, and some milestones to work toward in your design. You can take our ideas and implement them as we have suggested or choose a different path of your own.

Designing classes

An object-oriented program is decomposed into a number of cooperating classes. The handy thing about OO programs is that the data structure can often cleanly model the real-world ideas being represented. As the designer, your job is to determine what objects you are trying to model and what sort of operations these objects perform. To define a class, you need to answer two fundamental questions:

- What data is required to represent this object?
- What behavior will I need from this object?

The answers to these questions tell you what instance variables you need and what methods the class must implement. Behavior that is associated with an object should be provided as a message you can send to the object to ask it to perform some action, rather than having the client reach into the object and muck around with its data. For example, a client asks a Time object to find out whether it is before another Time object. If you want an Event printed, you send a message to that Event object asking it to print itself.

To introduce you gently to this process, only a few simple classes are needed for this program. These objects include the Time, Event, DailySchedule, and Datebook classes, along with a few helper-utility classes and routines. Each of the main classes has a pretty clear real-world analog to help guide your design and the relationships between the classes are not too complex. Let's give you an overview of each of the classes so you'll know how to proceed.

The Time class

First consider the Time class we used in lecture as our first simple object. It has the straightforward job of representing a particular time in the day. In the starting project, we give you a primitive version of this class, but you'll need to further develop the class to be fully useful.

The Time class is your first opportunity to work through designing a useful and robust object with a sensible and complete interface. Carefully think through your decisions and make choices of which you can be proud. It's not a very complex class, so it is a good one to tackle early, so you have a simple introduction to designing and manipulating data in the object-oriented paradigm.

First, evaluate the current interface of the Time class. Is it missing important features? Does it have unnecessary functionality you want to remove? Think through the rest of the program and what operations you will need to manipulate times and plan to include them as methods for the Time class. Feel free to remove any current methods that are redundant or unnecessary in your design. Be sure to only make those things public which make sense as part of the Time's external interface and which you are committed to supporting forever.

This Time object has the hour/minute/am representation that is intuitive and familiar, but it can be awkward to manipulate in that form. What other representations might work? What are the tradeoffs in terms of space and convenience of these other choices? Consider what operations are easier for a given representation and which ones are more difficult. Think through these before committing on your representation and then go forth and complete the implementation.

Instead of moving on, write some simple test code that exercises the Time class and allows you to find and correct any problems now. Can you reliably create Times, message them, print them, compare them, shift them, etc. using your operations and get the correct results? It's much easier to isolate and fix bugs when you're just dealing with one class at a time than trying to sort out everything at once when you have lumped everything together into a bunch of untested classes.

One convenient place to put your test code is in a static `main` method on the class itself. You can then compile and execute that class itself. You don't even need to remove the testing code before submitting unless it really interferes with the overall readability.

The Event class

Now consider the Event class. An Event object encapsulates all of the details for a particular datebook entry. An Event needs to track its name, color, and url. All three of these fields can easily be represented using Strings. An Event also needs to track its time interval and perhaps its date or days and whether it is regular or one-time event. In truth, the Event class is not that much more sophisticated than a C struct, but it serves to encapsulate the data into a clean abstract unit. A good milestone to aim for before moving on from Event is that you are able to read the events from the file, printing each out as read, to verify that the Event class is doing its job.

The TimeInterval class

Managing the time interval for an Event is little more complex than the other simple String fields. You may want to create a helper class, TimeInterval, just to encapsulate this concept. I choose to do so and I found it a helpful abstraction, but your mileage may vary. It is certainly possible to just directly track the start time and duration in the Event object as an alternative.

Internally, a TimeInterval could store its data as an array of two Times, one start, one stop, or perhaps a starting Time object and duration. Or perhaps something else entirely. Wrestling with these sorts of decisions is the most interesting part of designing a class. As a client of the TimeInterval, you don't care how it is internally represented, all you care about is that you can get the right results when you ask it to print and whether it starts before another TimeInterval and so on. But as the implementor, how you choose to represent it can make quite a difference in terms of ease of writing the code, the resulting efficiency, the size of the object, how difficult it is to modify later, etc. In general, we encourage clean design and clearly written code that may be less efficient over the terse, complicated alternatives that are produced in the name of efficiency.

The DailySchedule class

To manage all the events scheduled on a particular day, you will create the DailySchedule class. Because the number of events is not known in advance or constrained to any fixed size, an array is not appropriate, you will need use some sort of resizable collection to store the events. The java.util.Vector class has been around since 1.0 days and handles a simple, linearly indexed collection of objects. Java2 adds an entire family of new collections with more expansive and full-featured alternatives. The new collections are covered extensively in your text and we will briefly tour them in class, but since they rely on inheritance and interfaces, two concepts we won't cover until later, we think you'll find it simplest if you stick with the simple Vector class for this assignment. However, if you want to read ahead and do some independent investigation, by all means, you're welcome to use the newer facilities.

In order to facilitate orderly printing of the daily schedule, it is convenient to keep the vector of events sorted by start time. The DailySchedule class will include functionality to add events and print out the schedule in various formats. Writing this class is a good way to become familiar with the basic workings of the using collection classes, Vector and its ilk are heavily used in Java programming.

The Datebook class

And finally, you will create a Datebook class that tracks the schedules for all the various days. You will need to manage DailySchedules both for the regular events, as well as those odd one-time events scattered throughout the datebook. For the regular events, you might want an array of DailySchedules, one per day of the week. Those dates with one-time events will require their own separate DailySchedules. It is expected that most dates won't have any one-time events scheduled and thus, you should avoid wasteful storage. Rather than keep a large array with many null references or empty DailySchedules, you should create schedules only for those dates that have one-time events. Using the Date objects as keys and the associated DailySchedule object as the value will facilitate quick lookup of daily schedule by Date, and when no events have been scheduled, the lookup will return null to show there are no one-time events scheduled for that day. To associate a Date object with its DailySchedule, you will want to use one of the built-in key-value "map" objects. The long-standing Hashtable class is the easiest to make use of, so that is what we recommend. The new collections offers some fancier alternatives that would require a little more effort on your part to pick up, but are also fine choices to consider.

When you are printing out a week’s schedule, you will find that you need to display both the regular events with any one-time events that occur during that week. There are several ways to accomplish this task, some easier than others. You may want to think about it for a bit before making your decision about how to handle it.

A good milestone at this point would be producing the list output, which is quite a bit simpler than the table. Constructing the table will probably be the last task you complete for the program.

The built-in utility classes

In lecture and next week's section, we may briefly touch on the built-in utility classes, such as String, StringTokenizer, Vector, Date, Hashtable and foreshadow a bit of the new Collections, but we will not cover these classes in detail. Between your textbook and the on-line class specifications (see link from "Other materials" on our web site), you have access to pretty solid documentation. In general, in this course, we will expect you to be fairly resourceful in researching features and usage of the built-in classes (i.e. we are not going to use lecture time to painstakingly go through the details of the standard Java classes). This assignment is a great first step toward becoming familiar with how to explore the Java packages on your own. Feel free to ask questions if you find the documentation unclear.

A note about **deprecated**: Since the Java libraries are still evolving, you will notice at times the compiler or documentation will indicate a method or class is "deprecated". In moving from Java 1.0 to 1.1 to 2, some methods and classes were replaced with different and improved means of handling that functionality. The old classes/methods are still there but have been deprecated to show that their use is discouraged and they will eventually be removed from the libraries. As an example, the Date class changed quite a bit and much of the original Date class has been deprecated and replaced with facilities in the Calendar class. (There is more info about this in Chapter 16 of your text). We expect you to avoid using deprecated API and instead use its replacement. Feel free to ask if you need help sorting it out.

Some class design suggestions

A working program is definitely a good thing, but a truly worthwhile accomplishment is one that also excels in design and readability: is the program sensibly divided into classes? Are the classes themselves complete and clean? Do classes take care to maintain consistency and encapsulation of the object state? Are the identifier names well chosen? Are complicated operations broken up in helper methods to manage complexity? Would you want to take over a project that had this as its starting code base? Would you find it easy to extend the program to new functionality? …

Most of you are experts on the usual CS106/CS107 style, decomposition, and commenting standards. If you haven't taken those courses here, check the "other materials" part of the web site where I posted some style handouts from those courses that you might want to peruse.

Here's some specific issues to consider and basic rules of thumb that we believe in:

- All instance variables should be `private`. An object should tightly encapsulate its data and not allow outside access.
- If a client will need access to another object's instance variable, the class can provide a public accessor method (a "getter" such as `length()` or `getLength()`). However, be wary about handing out references into your internal data—e.g.. do you see why a stack object shouldn't hand out a reference to its Vector /array and have the client add elements to it? Instead the Stack can have a `push()` method which takes the client's element and adds it to the internal list itself.
- When needed, you can also provide a "setter" function for an instance variable. Be careful about this, just because you have a private instance variable `length` doesn't mean you have to have a method `setLength()`. For that example, most likely the length is changed as needed when elements are added or removed and shouldn't be externally settable. Only include the setter if the client's use will require it and there is no better way to provide that functionality. Take precautions in the setter function so that it cannot be used maliciously to corrupt the internal consistency of your object— i.e. don't allow a

client to change a count to a negative number or something that would cause your object to get confused or misbehave.

- Most methods will be `public`, since they are usually intended for public use. However, any helper methods only for use of the class implementor should be `private`.
- Give responsibility for behavior to the object itself rather than manipulating it using setters/getters from the outside. For example, you want to include methods in the class which can construct a new object given its starting state or print the data out nicely rather than having some other object stuff the data in field by field or extract the data to print it.
- You'll note that object decomposition leads to a different sort of code structuring than you're used to. For example, in C, you would likely group all the printing functions into one unit. In Java, each object takes responsibility for printing its own data, which will have the effect of distributing the code for printing around various classes. This can be a little disconcerting. Although object decomposition is an effective tool for managing complex projects, this sort of consequence is one of the downsides to it.
- At times, you will encounter some code that doesn't fit into the object-oriented paradigm, for example, consider the "main" code that handles the interaction with the user. Don't let this get to you, sometimes the paradigm just doesn't quite fit the task. It's okay to add static methods to a utility class to handle this. Just write the necessary methods in a good readable style and organize them sensibly.