Project 3
A Bit of Design, Lots of Standard Library!
Due: Monday, Oct. 29, 2012, 11:59 PM

Notice:

Corrections and clarifications posted on the course web site become part of the specifications for this project. You should check this page for the project frequently while you are working on it. Check also the FAQs for the project. At a minimum, check the project web pages at the start of every work session.

Introduction

Purpose

The purpose of this project is to provide review or first experience with the following:

- Using the Standard Library strings and containers instead of your own classes.
- Using the Standard Library algorithms, inserters, iterators, and C++11 `bind` and lambdas to work with the containers.
- Using function objects with the containers and algorithms.
- Narrow scoping of auxiliary class, function, or constant declarations in header files.
- Using object-oriented programming principles to improve the design of a set of classes, and to design new functionality.

Problem Domain

The functionality of this program is almost identical to that on Project 2. Unless stated otherwise, your Project 3 solution is supposed to behave exactly like Project 1 and 2 (as amended by the Corrections and Clarifications). Except for a few additional and different output strings in the `strings.txt` file, there are no supplied "starter" or skeleton files for this project - you will be using your Project 2 solution as your "starter".

There is a major new piece of functionality, which is that the program will keep track of each person's commitments - what meetings they are participants in. Any changes to the schedule that conflict with a person's commitments can then be detected. Thus no longer will it be possible for a person to be scheduled in two meetings at the same time! In addition, there will be a new command that will print out a brief summary of a person's commitments.

Overview of this Document

There are two main sections: The *Program Specifications* covers only how this project is different from Project 2. The second section presents the *Programming Requirements* - this how you have to program the project in terms of the structure and organization of the code, and which specific techniques you must apply - remember the goal is to learn and apply some concepts and techniques for programming, not just hack together some code that behaves right. This section is in two Steps, described below. At the end is a section of advice on *How to Build this Project Easily*.

This project will be specified in two steps. Step 1 involves switching over from the home-grown Ordered_list and String classes in Project 2 to using the Standard Library string class, containers, algorithms, iterators, inserters, `bind`, and lambda. This involves a couple of small design changes, and rewriting various bits of your code, such as using a Standard Library container instead of Ordered_list. To make sure you practice using a variety of containers and algorithms, the specifications will require that you use a variety of containers and algorithms for parts of your functionality. These are not necessarily the best choices in all places, and in a few places it might seem like more work than it is worth, but the idea is become familiar with a variety of Standard Library facilities. Then when they are worthwhile in your future work, you will be familiar with the ideas. You are expected to make thorough use of Standard Library facilities, not just the grudging minimum. Since Step 1 is basically changing implementation and not behavior, I will be examining and scoring your code for both quality and how completely and how well you followed the specifications to use the Standard Library facilities.

Step 2 involves design work, in two parts: fixing the design problems noted in Project 2, and implementing the commitment-tracking functionality. It is strongly recommended that you complete Step 1 before embarking on Step 2. I will test the old functionality separately from the new. So if you first get your Step 1 work completely done, you can check this with an early submission to the grading system, and be assured that you have successfully altered the program without damaging its functionality. Step 2 can then focus on adding the commitment-tracking functionality. You will have to modify some aspects of the design in your Step 1 solution, but it will be easier to modify something that works correctly and is written properly rather than try to do both Steps at the same time.

Program Specifications

What Stays the Same

Unless stated otherwise, your Project 3 solution is supposed to behave exactly like Project 1 and 2 (as amended by the Corrections and Clarifications).

Changes to the Commands

For convenience, here in one place is a description of the command changes for both steps.

> **pa** - print allocations; output the information on how many people, meetings, and rooms currently exist - this is only a change in the kind of information produced and its labels. See the sample outputs and the strings.txt file.

There is one new command:

> **pc <lastname>** - print commitments for an individual - prints a short description of each meeting that the person is a participant in. If the person is not scheduled in any meetings, a "no commitments" message appears. The meeting description consists of room number, meeting time, and meeting topic; the descriptions appear ordered first by room, then by time. Error behavior is the same as the **pi** command. See the sample outputs and the strings.txt file.

Other commands involve additional checks for errors. In general, any command that changes what Meetings a Person is committed to, or would be affected by a Person's commitments, must be modified for Step 2. For example, as in Project 1 and 2, deleting a room deletes all of the meetings in it, which frees the participants from all the meetings originally scheduled there. There are some modifications of the effects of specific commands:

> **ap** - after collecting and validating the room number, meeting time, and whether the person is already in the meeting, then checks to make sure the person is not already scheduled for another meeting at the same time, and outputs an error message if so. See the sample outputs and the strings.txt file.
>
> **rm** - Let the first room number and time be called the *old* room number and time, and the second room number and time, the *new* room number and time, and note that the old and new room number might be the same or different, and the old and new time might be the same or different. The program now does something sensible for all four combinations of the new room being the same or different from the old room, and the new time being the same or different from the old time.
>
> Do the error checks as follows: First, check that the old room number can be read, is in range, and that there is a room of that number, and second, that the old meeting time can be read, is in range, and that there is a meeting at that time in the old room. Third, check that the new room number can be read, is in range, and that there is a room with that number. Fourth, check that the new meeting time can be read and is in range. Each of these checks, if failed, results in an error message as in Project 2.
>
> At this point, if the new room and time is the same as the old room and time, output a message that no change was made to the schedule and do nothing further - note this is not an error, simply an information message. Otherwise, check that the new time is available for a meeting in the new room, and output an error message if not. Finally, check for whether a change in meeting time causes a commitment conflict for a participant; if so, output an error message, and make no change. Note that changing the room and not the time should not cause a commitment conflict - all of the participants are able to meet at the same time in a different room!

See the sample outputs for demos, and the strings.txt file for the new message strings.

Notice that except for the additional "already committed" errors and the **pc** command, the effects of the Step 2 commitment functionality are invisible - which is why correctly written code can pass Step 1 tests that don't involve commitment problems without implementing the Step 2 requirements.

Programming Requirements

For all projects in this course, you will be supplied with specific requirements for how the code is to be written and structured. The purpose of these specifications is to get you to learn certain things and to make the projects uniform enough for meaningful and reliable grading. If the project specifications state that a certain thing is to be used and done, then your code must use it and do it that way. If you don't understand the specification, or believe it is ambiguous or incorrect, please ask for clarification, but if you ask whether you really have to do it the specified way, the answer will always be "yes." If something is not specified, you are free to to it any way you choose, within the constraints of good programming practice and any general requirements.

Design: (somewhat) free at last! In this project, you are allowed to include additional declarations in the header files, and make modifications to the public interfaces of the classes. Consequently, I will not be testing your modules separately. However:

- You must follow anything this project document specifically requires or inherits from the Project 2 specification - like using the Error exception class. If not specifically exempted in this document, the Project 2 specifications must be followed.
- Any modifications to the classes must follow the guidelines for good design presented in the course.
- In general, although you can change the public interfaces, responsibilities, and collaborations of the classes, the basic program organization and classes should be recognizable as being based on those in Project 2. For example, you should not entertain solutions in which there are no Person objects anymore, or your Meeting class has radically different responsibilities such as creating and destroying Rooms. A fundamental redesign should not be necessary. If you are thinking otherwise, ask for help or clarification well ahead of time.

Step 1. Using the Standard Library Facilities

Program Modules - Changes from Project 2

The project consists of the following modules, described below in terms of the involved header and source files. Your header files can be modified from those you used in Project 2. But the Project 2 files for String and Ordered_list will not be part of your final project. There is no longer any need for p2_globals or any globals. Due to the similarity of the remaining modules with Project 2's, no skeleton header files are needed. Just make the changes specified below to your Project 2 header files.

All modules. Everywhere you used a String, use a `std::string` instead. Remove all references to any of the Project 2 globals, and remove the p2_globals files from your project; *no global variables are allowed, not even those that are file-scoped*.

While you might find it useful to phase out Ordered_list gradually, before you submit the project, remove all references and #includes for String and Ordered_list, and make sure you do not accidentally submit their files with your project.

Utility.h, .cpp. These files have the same specifications as in Project 2, but remove the `swapem` function template. There is a similar one in the Standard Library, named `swap` which (if necessary) you must use instead.

Person, Meeting, Room.h, .cpp. Remove all the static member variables and functions that are associated with allocation counting of how many Meetings are in existence. This information will be gathered for the **pa** command by looking only at container sizes..

> ***Important:*** Once the allocation counters have been removed, delete the declarations and definitions of any constructors, assignment operators, destructors etc. that are not necessary for you to define. That is, if the compiler will create one that will work, go with it instead of cluttering your code unnecessarily. This requires understanding what the compiler-created functions will do (review the course materials). Defining these members unnecessarily is error-prone and thus bad for code quality.

You can add to the Person, Meeting, or Room header files some declarations of functions or function object classes to be used by client code for ordering the containers of the corresponding objects. Put only the declaration in a header file and the definition in the .cpp file, if it makes it possible to `#include` fewer other headers in the header file. You must put these declarations in the narrowest possible scope: prefer making them private to the Person, Meeting, or Room class to making them public to a class; only if absolutely essential should they be scoped outside a class. Try to design these so that they work in a narrow scope.

> **However:** You must follow the principles of the Header File Guidelines. Do not clutter header files for Person, Meeting, or Room with publicly accessible function object classes, functions, or declarations that are not part of or required for the public interface of these classes. For example, if a function object class will only be used in the main module or the Room class's .cpp file, it is far better to put its declaration in the main module or the Room.cpp file *just before the functions that use them*. Check carefully that your #includes and header files follow the course guidelines.

p3_main.cpp. The main function and its subfunctions must be in a file named `p3_main.cpp`. The file should contain function prototypes for the subfunctions, then the main function, followed by the subfunctions in a reasonable human-readable order.

All input text strings should be read from `cin` or a file directly into a `std::string` variable using its input operator. Single characters must still be read into single character variables.

Top Level and Error Handling

The top level of your program is basically like that of Project 2: it should consist of a loop that prompts for and reads a command into two simple `char` variables, and then calls a function appropriate for the command. There should be a `try`-block wrapped around this followed by a `catch(Error& x)` which outputs the message in the `Error` object, skips the rest of the input line, and then allows the loop to prompt for a new command. All error messages must be packed up in `Error` objects and then thrown, caught, and the messages printed out from a single location - the catch at the base of the command loop. The possible errors, and error-handling behavior, for Step 1 are identical to those of Project 2, amended as described above. See strings.txt and the samples.

Fun with a map of function pointers! Do the following to map between commands and functions that do the commands: Concatenate the two characters of the command into a single `std::string`, and then use this as the key value for a `map` container

that gives you a pointer to the command function. Before processing any commands, the program loads the container with the function pointers for each possible command. Unrecognized command strings should not be allowed to fill up the container, either by not putting them in, or immediately erasing them. Because of its special status, the quit command should be tested for directly; it should not be called using the `map` container because it is awkward to return from main from a function.

> ***Note:*** In C++, calling `exit()` should not be done except in an emergency because it bails out to the OS immediately, bypassing any destructors that would be called in a return from `main()`. Thus your program should always terminate with a return from `main()`.

All of the functions called this way must have identical parameter lists and return types, meaning that they will all have to be called with the Room and Person containers as modifiable arguments, even for functions that don't modify them. The simple, consistent, and easily expandable top-level structure is worth the trade-off in const-correctness in the function parameters. If you wish, you can use an old trick to simplify the parameter lists: Declare a simple `struct` type and make these variables the members of the struct type. Declare a variable of the structure type in `main()`, and then you can simply use this one variable, passed in by reference, as a short-hand to move in all variables by reference at once.

Basic Programming Restrictions

This project is to be programmed in pure and idiomatic Standard C++ only. Everything must be done with typesafe C++ I/O, `new`/`delete`, Standard Library facilities and the classes and any templates that you write. Standard C Library facilities are not needed anywhere and must not be used in this project. You do not need, and must not try to use, any declared or dynamically allocated built-in arrays anywhere in this program. This prohibition does not apply to the good practice of defining `constant char * const` variables pointing to C-string literals for output messages. Your code is expected to follow the C++ Coding Standards document for the course.

Container Requirements

You will choose which kind of Standard Library container to use for each collection of objects or pointers. However, you must have some variety in your containers. The `map<>` container is already being used for the command map, but in addition, you must use each of {`map<>`, `set<>`, `vector<>`, `list<>`} *at least once each*. The `map<>` container is thus used at least *twice* in the whole project. At least one of the uses of `vector<>` must be used with `binary_search` and/or `lower_bound` to search for items and the locations of where to insert them. As in the previous project, the People objects must be referred to with pointers in the containers. You can use either pointers or objects in the containers that hold Meetings and Rooms; your choice.

> ***Important:*** As before, the containers that hold the current Meetings, Rooms, and People must persist from command to command - for example, you can't generate the contents of the people list "on demand" for the **pg** command.

These container requirements apply to the final version of the project that incorporates both Step 1 and Step 2, so you don't have to meet them with only the Step 1 version of your code. In fact, you may want to change your Step 1 choices in light of what you discover when you do Step 2. If your code is well organized (e.g. well-designed "helpers" in the main module) and you use some good typedefs, changing containers can be easy. There are choices that make the project code very simple and smooth.

Note: The other Standard Library containers may not be used in this project; they are all relatively specialized variations on the required four.

Algorithm Requirements

You may NOT use explicit for, range for, while, or do-while loops in this project; you must use a Standard Library algorithm instead. Use Standard Library algorithms like `find`, `copy`, and `for_each` to operate on the containers instead of writing out explicit loops like

```
for(it = people.begin(); it != people.end(), ++it) {crunch crunch}
```

This will require using some of the STL iterators, inserters , `bind`, lambda, and writing some additional simple functions and/or function objects. The result might be unnecessarily fancy, but it is good to get the practice so you will know how to do it in more complex cases where the technique can save tons of messy coding. In short, now is the time to climb the learning curve on getting acquainted with the Standard Library facilities. If you get stuck on a particular situation, ask for help.

- There are *exactly four* exceptions to this restriction:
  1. Your top-level command loop that reads in the two command letters and dispatches the command *must* be an explicit `while` or `do-while`.
  2. Your file restore code for implementing the **ld** command in the main module, Room, and Meeting classes, must use explicit loops to control the creation of the objects and reading the data. Note that the save file contents and format are unchanged from Project 2 - all the required information is logically present already.
  3. You should have a function that skips the rest of the line for error recovery, and this must use either an explicit `while` loop or the `ignore()` function (see the handout).

1. **Exactly once** in the project, you must use a range `for` with a container to get a bit of practice using it. The body of the range `for` should be plain code (as opposed to using a function object, `bind`, or a lambda). Consult the lecture notes and example code about range `for`.

- *No credit will be given for trying to use an algorithm in the first three exceptions*, and in fact, trying to do so will result in convoluted code, detracting from the code quality - that's why the exceptions are specified.

Requirements for function objects, iterators, `bind`, and lambda with algorithms.

- You must use a custom function object class that has member variables with an algorithm and a container at least once in the project. *Hint:* A good place is in implementing one part of the **pa** command where the container sizes are not directly accessible from the main module.
- You must use the `copy` algorithm with an output stream iterator at least once (check Stroustrup, lecture notes, and example code). Remember you can define additional `operator<<` overloads to make this easy (see the Handout)
- You must use a lambda with a captured variable with an algorithm at least once, and `std::bind` with bound values with an algorithm at least once. In other cases, you should choose whichever approach produces the simplest, cleanest result, or your choice if they are equally simple.
- In all other places in your code where you use a Standard Library algorithm, you must not use a helper function or custom function object class if `bind`, `mem_fn`, lambda, a stream iterator, or some other Standard Library facility will work just as well.

Step 2. Design Problem

There are two design problems - one is to improve the Project 2 design, the other is to design and implement the commitment functionality. It is useful to consider them together at first because they might interact. Keep clear that working with the design means that you should expect to modify the Project 2 classes and their public interfaces. So not only are you allowed to do this, but you should expect to do it: a design solution that preserves the Project 2 interfaces at the expense of design quality is a poor solution.

Design improvement. You should redesign the classes to fix the major design problems in Project 2. These were described in the Project 2 Document. The primary problem was that although Room was responsible for keeping track of its Meetings, it could not protect its Meetings container from getting corrupted (disordered) because it was possible to change the time of a Meeting while it was in a Room's container. This means Room could not be responsible for keeping its Meetings in order. The second problem was a fuzziness of responsibilities associated with whether Rooms dealt with participants or only Meetings. Your design should alleviate these problems. The changes are expected to be relatively small modifications of the Project 2 classes. Check with me well ahead of time if you think you need to do a major overhaul.

Commitment implementation. It is up to you to figure out how to implement the commitment-tracking and the **pc** command, but with the following restriction: You must keep track of a Person's commitments as the **ap**, **dp**, and **rm** commands are executed - you can't wait for the **pc** command, **ap** command, etc. to figure it out. That is, your code may not simply scan the whole schedule looking for the person in every meeting! This means you need *some kind of container(s) somewhere* to retain the information about which meetings a person is in. You will access and update the container(s) when a person is added or removed from a meeting, or a meeting is rescheduled or deleted. The save file will have exactly the same information in it as before, so the container(s) will have to be updated during the **ld** command processing as well.

You are allowed to modify the Person, Meeting, or Room classes, including their public interfaces, to arrive at a good solution. A good solution will have a simple and clear code structure, be compact and economical (such as not duplicating data with no compensating advantage), and retain a good class design. Remember that the key to a good design is a clear concept of what responsibilities belong to each class and the main module; each component must be responsible for the data that it is in the best position to manage. A component should not be doing things for other components that they could do for themselves. Only a very few additional member functions or variables should be involved; the interface and members from Step 1 should not be affected very much. If you discover a need to make drastic changes, you might be pursuing a poor solution - get some sleep or discuss it with me.

You can use any of the containers and algorithms in the C++ Standard Library to implement this functionality; in fact, if your solution fails to use them where they would work well, it is not a good solution.

Project Grading

I will announce when the autograder is ready to accept project submissions. There will be two sets of autograder tests. The first set will be similar to those of Project 2, and will not involve any commitment conflicts, or use of the **pc** command. Thus your program should be able to pass these tests without any of the commitment-tracking capability at all. The second set will involve the commitment-tracking capability.

Since you are allowed to modify the public interfaces of the classes, I will not be testing individual components of your code by mixing them with my own, so only input/output behavior will be tested by the autograder.

This project will be both autograder tested and hand-graded, so the Syllabus rules apply, so study the Syllabus information on how the autograder and code quality scores will be determined and weighted. Pay attention to commenting, organization, clarity, and efficiency consistent with good organization and clarity. Study and apply the C++ Coding Standards handout. Review the feedback you got on your Project 1 solution. Do not expect to do well on code quality by slapping the project together at the last minute.

The evaluation will include:

- Quality issues similar to the Project 1 evaluation (in their C++ version). Be sure you modified your code to follow C++ idioms instead of C idioms (e.g. using `nullptr` instead of `NULL`, `bool` instead of `int` where appropriate). The C++ Coding Standards handout covers many of these issues. Be sure to assess your code against the C++ Coding Standards before your final submission.
- Whether your code met the above specifications for using the Standard Library facilities and how well you used them. Check the project specifications carefully.
- How good your design solution in Step 2 is, both in how well it follows the course concepts, and whether it results in a simple and clear code structure with good division of responsibilities and collaborations.

How to Build this Project Easily

Step 1.

If you were thinking of getting a copy of Josuttis, now is the time. Otherwise, keep your Stroustrup handy and open to the containers and algorithms discussion, along with the handouts, lecture notes, and posted code examples to see how to use the Standard Library facilities. Remember: don't going dumpster-diving on the web; use the EECS 381 resources first, and then ask for help.

Follow the Steps by first converting your Project 2 over to use Standard Library facilities. Changing over to `std::string` instead of String is trivial, so get that out of the way first. Most IDE's have a multiple-file search and replace that makes this a minute's work. Change the top-level command dispatching. Next, choose your containers; choose wisely! You can change them one at a time and verify your program still behaves correctly.

Gentle introduction to algorithms and gizmos. To get gently introduced to using the algorithms and gizmos like lambda, bind, function objects, stream iterators, etc, I suggest keeping your explicit loops from Project 2 (after modifying them as needed to suit the container), and then change them one at a time to use the Standard Library algorithms and gizmos. Recompile and check the program after you change each one. This will mean you have to deal with crazy template error messages on only one thing at a time. It will get easier as you learn more.

Change the container type if it helps. As you work through the project, you may discover that a choice you made for the container was not a good one - while it worked well in one place, it was too awkward in another. Don't hesitate to change the container type for a better overall result. If your code is well organized, and you used typedefs, changing the container will be very easy - for example, almost all of the code using STL algorithms will stay the same! If this isn't true, your code is poorly written - fix it!

Key concept. The STL algorithms like `for_each` all run an iterator over a container and do something, like call a function, with the dereferenced iterator. The key to using them easily is to keep in mind what kind of a thing the dereferenced iterator is. Your function/function object/bind object/lambda object must accept a parameter of this type. This is especially important for the `std::map<>` container, which holds `std::pairs`; if you dereference a `std::map<>::iterator`, you get a `std::pair`. To use an algorithm that traverses the map you usually need a way to pick out the `.second` member. Sometimes you can do this easily, such as writing an `operator<<` that takes a pair and outputs only the `.second` member, or a lambda, helper function, or custom function object that does takes a pair parameter. The `bind` facility is the preferable solution if there is already a function or member function that needs to be called.

Choose return types wisely. Don't get stuck on the approach of returning iterators from helper functions - while often handy, the problem with returning an iterator is that the client usually has to declare the iterator type (now simplified with `auto`) or access the container to interpret the iterator (e.g. to tell whether it is `== .end()`). Elegant and more useful helper functions can be written that return Room, Meeting, or Person objects as references or pointers to those objects. Remember that if something has gone wrong, an error exception will be thrown, so you generally don't have to worry about how to make the returned value mean "not good" nor check the returned value for validity. Your code won't be thinking about a returned value if the function threw an exception!

Keep const promises. A `std::set<>` container has a simpler interface than `std::map<>`, but its items are supposed to be unmodifiable so that the contents cannot be changed in ways that will invalidate the order in the container. If you want to use it for changeable objects, remember that the most preferable approach is to change the object while it is not in the container. If you keep pointers in the container, then the pointers cannot be changed, but you can point to the objects with non-const pointers. But in this case, it is up to you to ensure that the changes you make will not result in disordering the container. Review the lecture notes summary on this issue. Using a `const_cast` is a sign of design failure - if you need to change something, why was it const to start with? - and don't abuse `mutable` (see the Coding Standards for the only acceptable use).

Using a vector? Use binary search! The `binary_search` and `lower_bound` algorithms provide a fast log-time search when used with vector<>, but their behavior is somewhat puzzling. When applied to a sorted sequence container, the `binary_search` algorithm will tell you whether the matching item is present, but not where it is! `lower_bound` also does a binary search and returns an iterator, but it doesn't necessarily tell you whether the item is present! If you need to know both whether the item is present, and where to find it or insert it, you can do it all with `lower_bound`:

- If the matching item is present, `lower_bound` returns an iterator that points to the matching item in the sequence.
- If the matching item is not present, the iterator points to where the sought-for item should be inserted (e.g. with a call to the insert member function that takes an iterator argument).
- So a returned value of `.end()` means either that the item is not present, or it should be inserted at the end (which the insert function would automatically do if given this iterator value). So a `.end()` result is unambiguous: if you are looking for the item, it is not there; if you want to know where to put it as a new item, it's at the end.
- The problem is that a non-`.end()` value doesn't tell you whether or not it is there - it means either "here it is" or "here is where to put it". How do you tell? Test to see if the iterator is pointing to an item that matches what you are looking for (such as a Person * that has the sought-for last name). If it matches, then "here it is." If not, it means "here is where to put it."

Step 2.

A seductively appealing idea for the commitments functionality is to make the main module primarily responsible for maintaining the commitment information. But it is a bad idea because it is likely to result in complex code and involves stripping responsibilities and encapsulation from Rooms, Meetings, and Persons, and so is not much of an object-oriented solution - don't go there. Instead, think about how object-oriented programming problems are normally solved: *class objects cooperate to solve the problem*. The design question then becomes "which objects are in the best position to handle different parts of the problem, and how do they cooperate?"

When you start on Step 2, keep an open mind about whether and how you might modify your Step 1 solution. For example, if you did Step 1 with containers of Room and Meeting objects, you might discover that everything works better if some of these containers hold pointers to the objects instead (or vice versa). If you did a good job in Step 1, you will find it relatively easy to make whatever design changes will help you do Step 2. The point is, don't let your Step 1 solution prevent you from arriving at a really nice Step 2 solution. Again, the key is to be able to spend some prime time thinking about the design for Step 2.

The commitment problem has neat design solutions which you can invent with some thought and then implement very easily, and ugly, awful designs that you can invent with little or no thought, but which will require a lot of work to implement (and may not even work). Moral: Give this some prime thinking time! A good question to start your thinking with:

Just exactly what is a commitment? In everyday life? What does it mean in terms of objects in the domain?

Don't jump in and start coding. Think about the design and sleep on it.

- Design Tip #1: Design the public interface(s) first; don't worry, or even think, about the private implementation. Getting distracted about how you will code stuff before you are even sure what you need is the main enemy of creative design.
- Design Tip #2: Delay writing code for as long as you can stand it. For example, sketch out how the public interfaces will be used in pseudocode. Actual code has a habit of setting your thinking in concrete; you need an open mind.
- Design Tip #3: If your design idea turns out to be a bad one, throw it away and start over. This is almost always faster than trying to get a bad design working properly. One reason is that you now understand the problem better, so the better idea should code faster. Another reason is that good designs code easily because it is clear what needs to be done, and where it will be done.

A final suggestion: Spend some time thinking about the design before you start on Step 1, and whenever you take a break while working on Step 1. Doing this thinking before you actually start on Step 2 will help you follow Tips #1 and #2 above.