Project 6
Extending and Enhancing the Maritime Project

Final Deadline for all deliverables: 5:00 PM, Tues. Apr. 20, 2010.
15% bonus: Submit all materials by 5:00 PM Fri. Apr. 16.
5% bonus: Submit all materials by 5:00 PM Mon. Apr. 19.

Notice:

Corrections and clarifications posted on the course web site become part of the specifications for this project. You should check the 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.

Overview

The purpose of this project is to provide further practice with design using the concepts presented in the course. Your task is to choose some "features" to be added to your Project 5 solution, work out a detailed specification of the features, and then design, code, and test their implementation. You are expected to use the design and programming concepts and approaches presented in this course. The project will not be autograded for output correctness (you control what the correct output is supposed to be). However, you must submit a working version of the program and a demonstration script that allows us to use your new features and verify that they work. In addition, you have to supply some design documents. These "deliverables" are described below.

Each feature is only described at the level of an "idea". You have to work out the details of the program feature and how it should behave, as well as how to design and code it.

You have to choose two features: One is simply an extension to, or elaboration of, the existing Project 5 structure, a simple extension. The second is a major Extension that requires a significant amount of new structure. You have to choose one of each type from the following lists.

Simple Extension: Choose one of the following three features:

Option S1. Another kind of ship. Explore the generality of the Project 4/5 framework by designing an implementing one of the following new classes of ships. Any modifications to the existing framework and classes should be as much as possible in the direction of making similar future additions easier.

- A Torpedo Boat - when told to attack a ship, chases it until it is in range, fires torpedoes until the target is sunk, and then returns to its starting point.
- A Merchant Raider - automatically seeks out and attacks only ships that are not warships.
- An Oiler - When instructed, it will match courses with a ship that is in motion and refuel it.
- A Trader - like Tanker, but automatically seeks out producer and consumer islands, and moves fuel from produces to the consumers

Option S2. Another type of view - "statistics" view. This is another kind of view that shows information about a specified ship, like bridge_view does, but the information shown is like the output produced in the "status" command - it shows the current position, fuel state, amount of cargo, current cruise destination, and so forth, for the ship that it is displaying. Like bridge_view, it can be opened or closed for a ship at any time. Your choice on the order and layout of the information in the statistics view and what it looks like for a sunk ship. However, the information has to be current - if you do a "show" or a "status" command at a certain point, the information they show should be equally up-to-date. This must be done in a way that not only is compatible with the existing views, but will also allow creating new views that have mixtures of the previous location information and the new statistics information.

Option S3. Save/restore. Give the user the ability to save the state of the simulation to a specified file at any time. At any time, the user can restore the simulation from a specified file - the result is a simulation that is in the exact same state as it was at the time of the save. This must be done in such a way that if the program is expanded, e.g. by adding another type of ship, it will be easy to include it in the save/restore system with little or no modification to existing code.

Major Extension: Choose one of the following four major extensions:

M1. Multiplayer maritime game.

Project 5 was in the form of a simulation, in which the user could issue commands to any ship, and had available a single map view of the entire world. This new feature is to change the simulation into a multiplayer simulation game in which any number of players would take turns; the "tick" of the simulated time clock happens only after all players have said "go." We'll assume that players do not try to look at the display while some other player is taking their turn. Each player has ships that only he or she can command, and islands on which they have a base. Each player has their own set of views that can only show objects that are within range of that player's ships or islands (also known as "Fog of War"). There can be any number of players, each of whom has potentially any number of ships or islands to be involved with, and has any number of views that are open only during their turn. Players can leave or join the

game at any time. You can add additional game-like features if you wish, but the ones just described are the highest-priority for this project. As examples of additional game features, you might add other kinds of resources besides oil, or permit players to form an alliance, meaning that their view information is combined so that each can see what the other can see.

Design goal: It must be possible to add additional Ship, View, or Island types with no change to the multiple-player code, and the framework must be in place to add additional capabilities to players, or modify the capabilities, with little or no change to the rest of the code.

M2. Groups of Ships or other Commandable Objects.

Provide ways for the user to put ships into groups and control the groups as a whole, in addition to the existing individual ship capability. Those of you who have played games such as Warcraft will recognize this sort of feature: You can form units into groups at any time and move or command the group as a whole, and any unit can be removed from the group and controlled individually, or groups disbanded, at any time. Any number, kind, or mix of units, including groups, can be included in a group. This feature requires you to make decisions about how ships in a group should move, be displayed, etc. At present, only Ships are commandable, but in the future we might want to command other types of objects in the same way - e.g. shipyards on islands that build different kinds of ships; coastal defense gun batteries that attack ships, etc.

Design Goal. There should not be any limits, either current or future, on which kinds of ships or commandable objects can be formed into groups and controlled as a group. In other words, if a new ship class is added in the future, it must be possible to treat it as a member of a group with no changes to the group control code. So the solution should easily extend to other kinds of "commandable objects" in addition to the current Ship classes.

M3. Multiple Object Families

Provide the capability to switch between different families of Sim_objects, while the program is running (this is expected to require that the previous game/simulation be terminated.) This must be illustrated with at least one more set of Sim_objects in addition to the P4/P5 set. Because we don't want to force the user to learn a whole new command language, this other family should respond meaningfully to the more or less the same set of commands, but otherwise be different in some interesting ways, not just straight "clones" of the current Islands and Ships. You need at least three members of the new family of Sim_objects.

Design goal: The mechanism to switch between families of Sim_objects must allow for easy extension to any number of future families of Sim_objects.

M4. Selectively Responsive Ships

Currently the ships do not respond to each other, or interact with each other except in very simple ways. We want to have Ships that interact with each other in interesting ways. For example, suppose we have two warships, and one attacks the other. Each ship behaves differently depending on the type of its opponent. For example, if a Torpedo boat is attacked by a Battleship, it attempts to run away, but if attacked by a Cruiser, it counter-attacks. Likewise, if a Battleship is attacked by anything, it always counter-attacks, while a Cruiser might run away from both a Torpedo boat and a Battleship. (Feel free to change this - it is just an example). All least three different kinds of Ships are required that can interact with each other in ways that different depending on the which types are interacting. Warships interacting with each other is one possibility, but you can choose another possibility. One of the Ship types can be the one added in Option A. The basic approach must allow for any arbitrary type or pattern of interaction; one based on a classification of ships such Warship vs Non-Warship, or relative firepower, is not acceptable.

Design goal: The approach taken to provide this feature must easily extend to additional types of ships in a straightforward way and must be able to represent any arbitrary interaction,, not. For the above example, if we add a fourth type of warship, then it must be possible to easily implement its interactions with the three existing types of Warships. This needs to be done in a way that gives a good tradeoff in the various issues involved, and corresponds well to the guidelines of good OO design.

What you have to achieve in the Major Extensions

The major criterion for a good design is whether the program can be easily extended, the hallmark of good OO design. This includes not just the clarity and simplicity of the code, but also whether additional classes or subclasses of the various types can be easily added to the extend the new capabilities of the program, and in a way that requires little or no modification of existing code - "adding functionality by adding code, not by modifying code."

Each option contains a concise statement of the design goal, which involves developing a general capability of some sort that will support future additions or modifications along the same lines. Your problem in the project is to (1) achieve this general capability with an extensible design, and (2) demonstrate that it works with some specific example implementations.

Two pitfalls to avoid: (1) Implementing a design that does your example cases in the minimal possible way without providing a clear extension pathway - in other words, failing the design goal. Such a project will get an extremely poor evaluation. (2) Running amok with example implementations, such as a dozen new kinds of objects for M3 - this will be a waste of time and won't make up for a defective design, no matter how clever the examples are.

The general and extensible capability is the primary goal of the project; the specific example implementations are there secondarily to demonstrate the scope and power of your general design. In other words, clever specific implementations in a poorly designed framework will not be worth much, while limited but well-chosen implementations that demonstrate the scope and power of a well-designed framework will be considered an excellent result. Please ask if this is not clear.

Other Rules

What You Can Change

The required elements and components of Project 5 must be present in this project, because, as described for Project 5, this is the second part of a two-part project.

Otherwise, you are free to modify the Project 4 and 5 classes as you choose, but keep in mind that this project is supposed to be based on Projects 4 and 5, so the more of that code you can re-use in this project, the better off you will be. Where applicable, the project should be fairly close in its behavior to Project 5 - this is not a new project, but an extension of the previous ones. All previous features should continue to be present, although details might be different and they might be implemented differently. Thus a complete rewrite is not called for, and is almost certainly not needed, but you can make any changes that will help you achieve a good design in this project.

Teams

Object Oriented Programming can work well with development teams. The team members first come to agreement on the responsibilities, collaborations, and public interfaces of the classes in the design, then divide the classes up between the team members, and then each team member develops the private implementation for his or her classes. If done properly, the separate classes will plug-and-play together, and changes and refinements to the design can be easily worked out and implemented. Therefore, if you wish, you may form a team to perform this project.

However, be aware that (1) additional features are required for a team project, and team coordination consumes time. Thus working on a team will not necessarily be less work than working individually. Also (2), the result will be better than individual work only if the team makes a point of constructive criticism of each other's ideas. Often, teams either go overboard with featuritis (see above), or reinforce each other's tendencies to do poor work. So a key function of team members is to keep the project on track to meet the requirements and meet them well.

A team must supply an additional document on how the design, implementation, and documentation work was handled as a team. It is expected that each member will make substantially equal contributions to the project, and all will write code - no "documentation specialists!"

- An individual person must do one of (S1, S2, S3) and one of (M1, M2, M3, M4).
- A team of two people must do two of (S1, S2, S3) and one of (M1, M2, M3, M4).
- A team of three people must do either: three of (S1, S2, S3) and one of (M1, M2, M3, M4), or one of (S1, S2, S3) and two of (M1, M2, M3, M4).
- Teams of more than three are not permitted.

Choose one member of the team to be the "lead" for submission purposes. The code will be submitted to the autograder by this team member.

Deliverables

Autograder deliverables

1. Your source code and a makefile, submitted in the usual way. The makefile must be named "Makefile" and the command "make" with this file should build an executable named "p6exe". I will be using the autograder simply as a way to send in your code and do a check compile of it. The result will be only 0 or 1 points for a failed compile or successful compile, respectively. These points will not be counted in the project score. The lead team member must submit the code. To avoid confusion, other members should not submit any code. Your code must compile and run without error in gcc 4.x, using the submitted makefile, and must be complete as submitted - I will not supply any files of my own. You should do a check compile and run in the CAEN gcc 4.x environment before finalizing your submission.

2. A set of command script input/output text files must be submitted along with your source code, to help demonstrate your new features. These must be suitable for I/O redirection, along the same lines as the sample files that have been provided in the course. The files should be named "demo1_in.txt", "demo1_out.txt", "demo2_in.txt", "demo2_out.txt", etc. There must be at least one such pair of files, but there can be up to 10 pairs. These files should correspond to the annotated hardcopy console script documents, explained below.

My goal in running your program with (and without) your scripts will be to assess whether your program actually does the things you specified. You will lose credit if there are problems that cause inconvenience or prevent me from compiling and running your

code, such as compile errors due to non-standard code, missing files, or run-time errors that interfere with running the program. I will not attempt to fix your code, nor contact you about missing files.

Hardcopy deliverables

You must submit some hard-copy paper documents in addition to your code. The quality of these documents is more important than the reliability of your code - plan plenty of time to prepare them! They are as follows:

1. Feature description document for each feature (hardcopy). This document identifies the Simple Extension or Major Extension that you chose (that is, S1, S2, M1, M2, etc.) and describes the specific details of how the new feature *behaves* and *how to use it* - it corresponds to what might be in the user manual for the program, and thus it is essentially your specifications for the new features. It does ***not*** describe the design or implementation of the feature! The Project 5 specifications for Cruise_ship are a good example of the level of detail and approach you should write for the description document - notice how they basically describe *behavior*, *not* design, and *not* implementation.

I will assume that everything specified in Project 4 and 5 still applies unless you describe how you have changed it, so that if you have changed how the program behaves for project 4 & 5 features and commands, you need to say so, but you can assume it still holds if not. Length: about 1 page/feature.

2. A hard-copy annotated command script console document(s) similar to the console samples in previous projects, showing the input and output of your program for the demo in/out files that you have supplied. The script should be annotated on the hard copy with explanations of what is happening - the annotations should be written by hand on the hard-copy. The document should have a name written on it that corresponds to which of the "demoN_in/out.txt" redirection text file pairs it corresponds to. The submitted script files, described as Autograder Deliverable #2 above, are the corresponding input/output files that I can use for redirection. By running your program using the script files, and examining its behavior and this hardcopy document, I should be able to see your new program features in action, and understand how they work. You can have one script document per feature, or combine them as convenient. Length: as needed. The annotations should be hand-written on the hard copy.

- Note that I will also "play" with your program some - it should not hang, get confused, or crash if I depart from your scripts.
- It's a good idea to figure out how to capture the console transcript early in your work; details differ depending on the platform. Don't take the risk of it being a last-minute show-stopper.

3. Design document for Simple Extension(s). The behavior of the Simple Extension was described in your feature description document, and should not be repeated in this document. This document simply explains how each Simple Extension fits into the rest of the project design; it should be fairly simple if you chose a good design approach. Length: About 1 page or less per feature.

4. Design documents for each Major Extension. Each Major Extension in this project also requires a set of documents that describe the design. The purpose of these documents that after reading them, I will be able to understand your code much more easily, and understand why you organized it the way you did. In particular, the design document for each Major Extension includes:

- A *UML class diagram* showing how the Project 5 classes relate to each other, along with any new classes in your Project 6 design. The Navigation, Geometry, smart pointer classes, and similar "utility" classes should not be included. Also, you can follow the example of the Project 4 class diagram and show only the key members of each class - those that are important for understanding the design. This diagram can be hand-drawn as long as it is clear. Refer to the UML handout and follow the format. If the Major Extensions can all be included in one UML diagram, then only one needs to be included and the document for the other features can simply refer to it.
- A *UML sequence diagram* that illustrates an informative interaction between objects in the project feature. This can be hand-drawn as long as it is clear. Refer to the UML handout and follow the format. Note that this diagram shows the interaction between *objects, not classes* - make sure that yours does likewise.
- A *design document* that explains the design, referring to the diagrams, and using the terminology from the course materials for any patterns, idioms, or concepts that play a role. OOP programmers use this vocabulary to improve communication, and you should too. Length: about 2-3 pages per Major Extension.
- An *extension document* that explains how the design goal would be met for each Major Extension - for example, in Major Extension M4, explain what would have to be done to add another kind of ship and make it respond differently to the existing ship types, and vice-versa, and how you have made it easy to do. Length: about a page per Major Extension.

Important: Reading the design documents should make it easy for me to understand the structure and organization of your code. If the documents are incomprehensible or incomplete, not only will you lose credit for them, but I will not take the extra time to figure out your design from the code, and so will downrate its design quality as well.

5. Team activity document. If a team, you must supply a document that lists the team members, describes whose Project 5 code was used (or what parts from which team members were used), describes how you arrived at the project design as a team, and which team member was responsible for what work in the project. Length: 1 page.

Submission Rules

Because paper documents must be turned in, the deadline for each day (and early submission bonuses) is at 5:00 PM instead of midnight. To give me a chance to get started on evaluating projects as they are turned in, your project will be considered as completely submitted as of 5:00 PM on the day when you deliver the hard-copy documents; after 5:00 PM on that day, I will assume that I can run and print out your code and start evaluating it.

So don't turn in the hard-copy deliverables until you have submitted your final code.

Because of past traumatic and miserable experiences with handling large numbers of paper documents for many projects, you *must* follow these rules for the hardcopy deliverables:

- Your name or uniqname must appear on each paper document (if a team project, the names of all team members should appear on each document).
- There are three document sets: the feature description(s) pages, the design documents pages, and the script(s) pages. Each one of these sets must be stapled together. That is, we should get from you 3 stapled-together batches of paper: one batch of feature description, one batch of design description, and one batch of scripts. In addition, if you are a team, we should get a 1-page team activity document. Do not expect that we will fuss with a mass of separate sheets of paper when we have dozens of projects to evaluate.
- The paper documents must be enclosed in a 10 X 14 clasp-type envelope. The following must be written clearly on the outside of the envelope:
  - Your name (or names, if a team).
  - The uniqname under which the source code was submitted (this is just you, if you are working alone).
  - The combination of features that you chose (e.g. S1, S3, M1, M2), so that I can easily group the projects together that worked on the same feature without pawing through the documents.
- The hardcopy deliverables in their properly identified envelope must be delivered to the professor *in person* at times and places to be announced. If you do not deliver these documents in person, I will not be responsible for them.

Important: If you do not follow these rules for submitting your deliverables, I will refuse to accept them. Be sure to find and purchase a 10 X 14 envelope ahead of time. This might seem mickey-mouse, but it makes a huge difference in handling the projects. So it is a real requirement. If you show up with loose pages of paper, be prepared to run to the bookstore or go dumpster-diving to come up with an envelope.

Project Evaluation

The score for project 6 will be based only hand-grading, and unlike previous projects, the bonus award applies to the hand-grading score. A portion of the hand-grade score will consist of manual run-testing of your program as described, but if the program runs basically correctly, most of the score will be based on document, design, and code quality.

The autograder is used only as an easy way to send in your soft-copy materials, and because I will be run-testing your code in the autograder environment, the autograder will also do a check compile for you using the makefile you supply. The autograder awards a single point for a successful compile, but that single point is not part of the project grade - it means only "yep, it compiles." The actual project score comes completely from the hand grading.

The score will be based on your specific definition of features, and how well your program meets the design goals, as illustrated by the examples that you implement and as explained by your deliverable documents, and manifested in the structure of the code. To put it positively, a good specification of the features, a good design that is general and extensible, quality code, and clear and informative documents, should result in a high score.

To put it negatively, if your specific features and design are minimal in scope, you will receive a minimal evaluation. An *excellent* way to get a *horrible* score is to write code that will only handle the specific examples that you implemented - you've missed the whole point of the project (see above).

Teams: All members of the team will get the same Project 6 score. The submitted Project 6 code will be evaluated for both Project 5 and Project 6 requirements. Differences in number of features will be adjusted by averaging the scores for each feature together, so the scores for all team sizes will be on the same scale.

Important: Do not implement more than the required number of features. If you do, no "extra credit" will be awarded, and I will base the grading on the *worst* of the features that you supply. The idea is to do a good job on the required number features that you choose to design and implement, not deliver a hodge-podge of hacked-up code.

Suggestions for getting a good evaluation

- *Write quality code*. This project is to be programmed in Standard ANSI C++ and take full and appropriate advantage of the Standard Library, and the usual rules of quality coding apply. Compare the C++ Coding Standards Document against your project code. Since this is the third time the general code quality will be evaluated in this course, any shortcomings in general

code quality will be awarded negative points, rather than zero points. So if you've had general code quality problems in previous projects, you should take extra care with this one. Learning from the previous evaluations is critical. If you want more help with this, bring your previous projects in for a discussion.

- *Look for opportunities to apply the concepts, techniques, and design patterns presented in this course*. Do not re-invent the wheel or adopt a clumsy approach when a better one was presented. This is your chance to put these ideas together and apply them to get a great result. To put it negatively, if your project looks like you never took this course, or skipped the last half of the lectures, then it will get a poor evaluation.
- *Take care with the documents* - see the above warning. I won't bother to puzzle out your code if your documents are not helpful. Thus poor documents will lose credit for both the documents and for most other aspects of the project. In terms of scoring, it will be better to submit buggy code than inadequate documents, so allow time to develop the documents. Drafting the feature description and design documents before you start coding will actually help you work faster and better.
- If you are a team, take time to pass the draft documents around and criticize them severely (better you than me!) and fix them. It is permissible to get help with writing the documents if it involves only the content and presentation of the documents, and not any aspects of the code or its design.