Project 6
Improving the Medieval World: Translating Ideas to Good Design and Good Code

15% bonus: Submit all deliverables by 5:00 PM, Friday, Dec. 07, 2012.
10% bonus: Submit all deliverables by 5:00 PM, Monday, Dec. 10, 2012.
Final Deadline for all deliverables: 5:00 PM, Tuesday, Dec. 11, 2012.

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.

More importantly, you are expected to use the design and programming concepts and approaches presented in this course. The concepts presented in the course can be directly applied to produce elegant results in this project. Since good design is much harder than good coding, you need to start thinking about the design of this project days before you start coding - procrastinate on the coding if you must, but procrastinating on the design will probably be fatal!

The project will not be autograded for output correctness because you control what the correct output is supposed to be. I will also not test your components because you control what the components and their interfaces are. However, you must submit a working version of the program and demonstration scripts that allows me to use your new features and verify that they work. In addition, you have to supply some design documents that specify your additional features and explain your design for them. These *deliverables* are described below.

Each feature is only described at the level of an "idea" - like something the marketing people might come up with. You have to specify the feature - work out the details of the 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 enhancement* that requires a significant amount of new or modified structure. You have to choose one of each type from the following lists. Teams have to choose more features (see below).

Simple Extensions

Option S1. Another kind of agent that does something interesting. Take advantage of the Project 5 framework; any modifications to it should be in the direction of making similar future additions easier. This new agent type must have the following characteristics:

- It may not be simply a clone or near-clone of the existing three classes - it must behave differently in a significant way. For example, a new class that is simply a stronger Soldier will not do.
- It must display “automatic” behavior at least as much as Peasant or Archer do, and definitely more than the current Soldier, which is pretty boring.

Option S2. Another kind of view. Test the extendability of the Project 5 MVC framework by adding another kind of view. This can display a different combination of the information present in the Project 5 views, or it can involve a new kind of information as long as any modifications you make to provide the new kind of information make it easy to add additional views using that information. The Project 5 MVC framework must be intact - you are adding another kind of View, not redesigning the whole system.

Option S3. Save/restore. Give the user the ability to save the state of the program to a specified file at any time. At any time, the user can restore the program state from a specified file - the result is that the program 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 Agent, it will be easy to include it in the save/restore system with little or no modification to existing code.

Option S4. Destroyable structures. Make it possible for warrior-type Agents to attack and destroy Structures, which disappear from the world like killed Agents do. This must be done in a way that makes it simple to add new kinds of structures and have them be destroyable as well.

Major Enhancements

M1. Multiplayer game

Project 5 was in the form of a simulation, in which the user could issue commands to any Agent, and had available a single map view of the entire world and various local and other views. 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." Each player has Agents that only he or she can command, and Structures which belong to individual players. Each player has their own set of views which are open only during their turn (we'll assume that players do not try to look at the display while some

other player is taking their turn). Any number can play and have any number of objects or views. The number of players is specified when a game is started, but a players can leave the game at any time. You have to decide and specify what happens to a player and its stuff when the human player decides to leave the game, and under what conditions a player is required to leave the game (e.g. no more Agents left alive). You can add additional game-like features if you wish, but the ones just described are required for this option. As examples of additional game features, players might have additional kinds of resources such as money, or players might have to have some amount of these money or food in order to build or train new objects.

Design goal: It must be possible to add additional Agent, Structure, or View types with no change to the multiple-player code. Even if you don't add any additional game-like features, the framework must be in place to add such features or other capabilities to players with little or no change to the rest of the code. While actually implementing a computer-controlled player is well beyond the scope of the project, a good solution will have an obvious place-holder for such a capability.

M2. Groups of Commandable Agents

Provide ways for the user to put Agents into groups and control the groups as a whole, in addition to the existing individual Agent 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, or groups of units, can be included in the group. This feature requires you to make decisions about how agents in a group should work, move, attack, be displayed, etc. At present, only Agents are commandable.

Design goal. There should not be any arbitrary limits, either current or future, on what kind of Agents can formed into groups and controlled as a group. If a new Agent 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.

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.) You must demonstrate this with at least one more family of Sim_object classes in addition to the P4/P5 set, and it must include at least three kinds of objects. Because we don't want to force the user to learn a whole new command language, this other family should respond meaningfully to more or less the same set of commands, but otherwise be different in some interesting ways, not just straight "clones" of the current Structures and Agents.

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 and classes within each family, within the constraint that the command language applies to each family.

M4. Selectively Responsive Agents

Currently either the Agents do not respond to each other, or interact with each other in very simple ways. We want to have Agents that interact with each other in more interesting ways. For example, suppose we have two Agents, and one attacks the other. Each Agent behaves differently depending on the type of its opponent. For example, if a Foot-Soldier is attacked by a Archer, it closes in to counter-attack, taking advantage of its superior strength and protection. But, if attacked by a Knight, it runs away to hide. Similarly, a Knight might always counter-attack a single attacker, but run away if there are multiple attackers. (Feel free to create a different set of relationships - this is just an example). All least three different kinds of Agents are required that can interact with each other in ways that depend on which types are interacting. Warriors interacting with each other is an obvious possibility, but you can choose another possibility. It is fine if one of the Agent types is the one added in Option S1. You can certainly modify the behavior of existing Agents to provide a better demonstration.

Design goal: The approach taken to provide this feature must first, be general enough that any arbitrary pattern of interaction could be supported - for example a simple distinction based on one or two attribute such as attack-strength (e.g. weak agents always flee from strong agents, strong ones always counter-attack) just won't do. We want to be able to support any kind of interaction or behavior that can be reasonably programmed. Second, the approach must easily extend to additional types of agents in a straightforward way. For the above example, if we add a fourth type of warrior, then it must be possible to easily implement its interactions with the three existing types of warriors. This needs to be done in a way that gives a good tradeoff in the various issues involved, and follows well the guidelines of good OO design.

M5. Agents that are Aware of What's Happening

Project 4 has agents that are aware of being attacked by other agent, thanks to a direct interaction between the two. Project 5 had an agent, Archer, that was aware of the object locations and so could pick a target or refuge. However, in general, the agents don't know what each other is doing in any more general way, and so can't behave very intelligently. This enhancement is to allow the definition of agents that know more about "what is going on" beyond what we provided for in Projects 4 and 5. For example, suppose that Peasants could automatically tell when a farm has an excess amount of food and start collecting from it. Maybe a Peasant could bring food to a Soldier who announces that it is hungry. Maybe if there is a fight going on in the vicinity, an Archer could move close enough and join in. Again, these are just examples to illustrate the concept, not specifications of what you have to do. The idea is that if a Sim_object announces a change in state, or an activity, then agents can respond as suits their type.

Design goal: The approach taken to provide this feature must easily extend to both additional types of Sim_objects, and additional kinds of state changes, activities, or announcements in a straightforward way.

What You have to Achieve in the Major Enhancements

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 examples in the minimal possible way without providing a clear extension pathway - in other words, failing the design goal - this misses the point of the project. (2) Running amok with example implementations, such as a dozen new kinds of objects for M3 - this will be a waste of time and it will distract from creating a good design. A lot of examples won't make up for a defective design, no matter how clever the examples are.

So keep the priorities clear: The primary goal of the major enhancement is to *provide a general capability for extensions of a certain kind*. The specific implementations are secondary, and are required only to demonstrate the scope and power of your general capability. In other words, many clever specific implementations in a poorly designed framework will be worth very little, while a few 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

Programming requirements

This project is to be programmed in Standard ANSI C++11, and take full and appropriate advantage of the Standard Library. Your program must follow the course C++ Coding Standards and the guidelines and concepts for good software design and coding presented in this course. Using additional features of Standard C++11 beyond those covered this semester is acceptable, but it definitely not essential for a good design; take care not to get distracted.

What You Can Change

The required design 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. To be clear, these are the Model Singleton, pervasive use of smart pointers, the MVC organization, the four kinds of views, and the two kinds of warriors.

As long as these design elements are present, 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. Ask for clarification if you aren't certain about this requirement.

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.

If you want to try this out, you may form a team of up to three to perform this project.

Pitfalls. However, be aware that (1) additional features are required for a team project, and team coordination consumes time. Thus working on a team will probably not 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 work. It has been my experience that the team can often appears to encourage each other to do poor-quality work - you don't want to tell your friend that his idea or code stinks. So a key function of team members is to keep the project on track to meet the requirements and meet them well. Ask each other, "What would Prof. Kieras think of this?" Less common is the other extreme: the team talks themselves into a very ambitious and well-designed project that maxes out the grading scheme but involves more work than necessary. I can't reward the extra work, and it can be so much that it interferes with the team's other courses. Come and meet with me about your ideas if this might be a problem.

Your team has to start with a Project 5 implementation. You can choose a single solution authored by one of the team members, or you can combine pieces from more than one member of the team. You should review and compare your solutions, pick the best or best parts, and fix any problems you identify, because the Project 5 implementation will be human-evaluated as part of Project 6, and all team members will get the same score.

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.

- An individual person must do one of (S1, S2, S3, S4) and one of (M1, M2, M3, M4, M5).
- A team of two people must do two of (S1, S2, S3, S4) and one of (M1, M2, M3, M4, M5).

- A team of three people must do either: three of (S1, S2, S3, S4) and one of (M1, M2, M3, M4, M5), or one of (S1, S2, S3, S4) and two of (M1, M2, M3, M4, M5).
- Teams of more than three are not permitted.

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

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 *must build an executable named "p6exe"*. You 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 4.7, 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 4.7 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 *must 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. Your submission files should be "flat", all in the same directory with no subdirectories. I will run a script that runs your p6exe with each _in.txt file and diffs the result with the corresponding _out.txt file.

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 misnamed files, 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 or misnamed 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. How it works/how to use it (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 Archer and the new Views 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 is 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. 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. Be sure the diagram is correct - for example, check that there are no missing connections between classes.

- 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 conforms.
- 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. For example, if you are using the Abstract Factory pattern, use the name "Abstract Factory". 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 agent and make it respond differently to the existing agent 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).
- Do not expect that I will fuss with a zillion sheets of paper when I have dozens of projects to evaluate. Stapling is mandatory. There are three kinds of documents: the feature description(s) pages, the design documents pages, and the script(s) pages. Each instance of each kind must be stapled together. That is, I should get from you a stapled-together set of pages for each feature description, another for each design document, and another for each script. In addition, if you are a team, I should get a 1-page team activity document.
- The paper documents must be enclosed in a 10 X 13 clasp-type envelope (see below). 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 13 clasp-type envelope ahead of time. See the picture nearby for the type of envelope required; this is cheap and effective; the clasp keeps the contents secure when closed, and can be repeatedly opened and closed. 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 to get an envelope.

Project Evaluation

The score for project 6 will be based only on 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, almost all 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 of 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 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. 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. Rather, this is your chance to put these ideas together and apply them to get a great result.
- *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.