Project 5
Medieval Pointers and Patterns
The first part of a two-part project

Due: Friday, April 3, 2009, 11:59 PM

Note: The Corrections & Clarifications posted on the project web site are officially part of these project specifications. You should check them at least at the beginning of each work session.

Because of this project has a fairly open design, these specifications contain only few specifics of how your code should be written, and so run a risk of being under-detailed. Be sure to start early enough to have an opportunity to ask for a clarification or correction.

Purpose

This project is a extension/elaboration/refinement of Project 4. You will need a working version of Project 4 as a "starter." Unless specified otherwise, Project 5 will *behave* identically to Project 4. This project will provide a foundation for the final project in the course, and so is best considered as the first phase of two.

In this project you will:

- Make use of the Singleton pattern to make Project 4's Model globally accessible in a well-controlled way to allow Sim_objects to find about about each other and indirectly interact with Views.
- Try out smart pointers to automate memory management using the to-be-Standard TR1 smart pointers if possible and convenient, or the supplied intrusive reference-counting `Smart_Pointer` template if not.
- Apply the Model-View-Controller pattern more completely by implementing additional kinds of views.
- Demonstrate the value of an OOP framework by adding a new derived class of Agent, observing how only the code required to support the new class needs to be added while the remaining code is unmodified. As the gurus say, *add features by adding code, not by modifying code*.
- Get additional practice designing and refactoring your own classes.

This project assignment attempts to leave as much of the design under your control as possible. Thus the specifications will be considerably less detailed than before, and will emphasize how the program should behave, and very little about how you should accomplish it. Where necessary to make sure you work with the informative design possibilities, some design constraints are specified - a few things you may or may not do in your design. While the design is under your control, you are expected to make good use of the OOP concepts and code quality guidelines and recommendations presented thus far in the course. While you are free to modify Project 4 as needed, you should not have to make any substantial changes at all to the Sim_object class hierarchy. Rather, you will be adding to this hierarchy, designing a new class hierarchy for View, and modifying Model and Controller to accommodate these. In other words, your Project 5 solution should be clearly based on Project 4's. The final project will be a further extension of this project, meaning that this is actually the first part of a two-part project. Be sure to study the Evaluation section below for information on how the code quality will be assessed.

The specifications are expressed as steps in the order you should do them for maximum benefit and smoothest work in this project.

Step 1. Make Model a Singleton class.

Once you complete this step, your program should run identically as before, but now it will be possible to do the remaining steps, which require program-wide access to information and functions in Model. Be sure you follow one of the complete recipes for the Singleton pattern, and *choose wisely* which services you will have Model provide to support the remaining steps - especially in Step 5, where a new class will need to be able to determine which objects are closest to it, meaning it will need access to information held by Model.

Note: The idiom for using a Singleton is to call its get_instance() function whenever you need to talk to the Singleton - so do not store a pointer or reference to it. So remove any stored pointer or reference to Model that you had in your Project 4.

Step 2. Use smart pointers for all Sim_object family objects, and have dying Agents remove themselves immediately with Model's help.

Step 2.1. Change all containers and pointer variables used to refer to Sim_objects, Structures, Agents, etc., over to smart pointers using the Standard TR1 smart pointer classes (see the Example Code directory for examples of their use with gcc 4). If you are not using gcc 4, you can either download and install the Boost TR1 library for e.g. MSVS, or use the Smart_Pointer template, `Smart_Pointer.h`, in the course Example Code directory. You can use *only one* of these: either the Smart_Pointer.h or the tr1 smart pointers, but not both.

- *Hint:* See if your IDE allows you to simply do a global search/replace throughout all of the project source files of "Agent *" with "Smart_Pointer<Agent>" or "shared_ptr<Agent>", etc. This will save a lot of time, but you will still have to fix a few things.

- *Hint*. If you use shared_ptr, you will need to use the `enable_shared_from_this` facility when e.g. a Soldier calls take_hit on another Agent. Because of how the definition of the TR1 smart pointers try to prevent coding mistakes, you will also have to make a few other changes - like using the reset() function instead of setting a pointer to zero.

Once you have switched over to smart pointers, you should not have any "raw" pointers for Sim_object family objects anywhere in your program, nor should you have any explicit deletes of these objects - the smart pointers should do this automatically, and containers of smart pointers will destroy the contained smart pointers when they get destroyed, eventually automatically deleting all the pointed-to objects. After making this change, your program should still function correctly, although dead agents may get deleted at a different time than before, depending on details in your code.

Step 2.2. Once you have the smart pointers in place, it is now possible to simplify how Agents disappear when they are killed; Project 4 avoided a dangling pointer problem by having the Agents go through a series of states to ensure that any other Agent (e.g. an attacker) had a chance to disconnect its pointers before the dying Agent was deleted. However, with smart pointers used throughout, there is no need for this process. In fact, as soon as an Agent realizes it is going to be dead, it should ask Model to remove it from its containers. This way, instead of Model having to be the grim reaper and monitor and remove dying Agents, Agents know when they are dying and simply ask the world (Model) to forget them immediately.

More specifically, Agents are now either **Alive** or **Dead**. There is no need for the **Dying** and **Disappearing** state; they, and their transitions, should be removed as well, along with the is_disappearing() accessor in Agent. Modify Agent::update() so that if an Agent is in the **Dead** state, it stays in that state. Modify Agent::lose_health from Project 4 so that if the Agent becomes no longer alive, set its state to **Dead**, and then, as the last statement before returning, call Model::remove_agent() with a pointer to itself. Ensure that Model removes and discards all pointers to the Agent from its containers. Subsequently, that Agent should not be updated any more or listed in any status command outputs. The code in Model for checking for and removing dead agents after each update is not needed any more and should be removed. Finally, note that in Controller, it will be impossible to try to command a non-**Alive** Agent because they will have been removed from the container of known agents immediately upon becoming non-**Alive**. Thus the Error test that a commanded Agent needs to be **Alive** is redundant and can be removed or turned into an assertion.

Step 2.3. There is a complication to this immediate removal from Model. If Model was holding the last smart pointer to the Agent, the Agent will get destroyed. However, perhaps a Soldier was holding a pointer to the now-dead Agent. If the now-dead Agent got deleted, following the pointer to ask about its internal state is undefined because the member variable values of a deleted object are undefined. Clearly, arrangements need to be made so that if another Agent is pointing to a now-dead object, the status of the dead object is well-defined. Smart pointers make the solution easy. You will do this differently depending on the kind of smart pointers you are using. (1) Soldier keeps a Smart_Pointer to its target, so the dead Agent is kept in existence. Soldier can test the target before and after hitting it and then discard its pointer if its target is dead. (2) Soldier keeps a tr1::weak_ptr to its target. This kind of smart pointer doesn't keep the target in existence, but can be queried to see if the target object still exists, and a shared_ptr created to it if it does, which again will keep it in existence while Soldier attacks it. Check your code carefully to make sure you are correctly handling this situation - if you eliminate all raw pointers to Sim_objects, you will automatically have to make an arrangement of this sort.

Step 2.4. The describe and update functions for Agent and Soldier need a bit of fixing. Since an Agent asks Model to remove it from the Model containers has soon as it is "killed", then the Agent will not be listed in any subsequent status commands. Thus we do not expect to see an Agent listed as being Dead - it will have already been removed from the list of Agents. However, a live Soldier might still be in Attacking state but with a target that is either dead or deleted. This could happen, for example, if another Soldier killed the target in a later update, but this Soldier has not yet been updated, and so is still in attack mode. In this project, either a smart pointer is keeping the dead target in existence, or a tr1::weak_ptr is used and can be tested to see if the object is still there or has been deleted. In either case, your Soldier::describe() should output that it is "Attacking dead target" rather than attempt to access the name of the target.

The update function has a similar quandary. Modify Soldier::update from Project 4 so that it does the following in this order:

update. First update the Agent state. Then do the following in this order:

1. If this Soldier is not Alive, do nothing further (this should be redundant).
2. If Not Attacking, do nothing further.
3. If Attacking, do the following in this order:
    1. Check to see if the target object is dead or deleted. If so, output "Target is dead", set the state to Not Attacking, discard the target pointer, and do nothing further.
    2. Compute the distance to the target as in start_attacking. If the distance is greater than the range, output the message "Target is now out of range" and set the state to Not Attacking, and do nothing further.
    3. Output the message "Clang!" (Soldiers use a sword.) Call the target's take_hit function with this Agent's attack strength and a pointer to "this" object.
    4. Then check to see if the target is dead at this point (maybe we, or some other Agent, just killed it), and if so, say "I triumph!" (even if somebody else should get the credit) and set the state to Not Attacking, and discard the target pointer.

Step 2.5. A final complication is the possibility of "cycles" - the graph-theory term - in smart pointers, that prevent objects from being deleted when they are supposed to be - even at program termination. For example, Soldiers should be keeping track of their targets through smart pointers. Thus if two Soldiers are attacking each other, they will be referring to each other. If they refer to each other with Smart_Pointers or tr1::shared_ptrs, they will keep each other in existence even if Model has discarded its pointers to both

of them. The same thing might happen if they refer to each other indirectly, such as through another Soldier. A similar problem might keep Structures existing if in a future version Structures refer to each other or to Agents that refer back to the Structure. Similarly, a Peasant will be keeping smart pointers to its source and destination Structures, and it is possible that in some future version, there might be a cycle such that the Peasant or Structures keep each other in existence.

You will have to devise a solution for making sure that the cycles get broken so that all objects will get destroyed at the most appropriate time, so that for example in this project, two Soldiers that are attacking each other get deleted if a "quit" command is issued. Your solution must work in away that allows it to be easily extended to any additional sub-types of Sim_object. The specifics of a good solution will be different depending on whether you are using TR1 smart pointers or the `Smart_Pointer.h` template. Your solution should be tailored to which kind of smart pointer you are using, and provide for easy extension to new kinds of Sim_object. One test for the correctness of your solution will be to start two Soldiers attacking each other and then issue a “quit” command. Without this solution, neither will get deleted; with a correct solution, they both will.

Step 2.6. Once you complete the above sub-steps, your program should run pretty much the same as before, but you will see differences in the status output and the order of destruction of Sim_objects, based on the details of which containers get emptied when, or which smart pointers get discarded at what time. In general, if you are doing this right, a killed Agent will get deleted sometime during an update cycle, not at the end of it - you should be able to set up a fight in which the destructor messages appear before some of the update messages. You will also see some differences in command error messages - for example, you will no longer be able to attempt to command a dead agent because their name will have been removed from Model. Observe the destructor messages to see this interesting "garbage collection" at work, and verify what happens when you add the remaining program features. Because exactly when the object gets deleted depends on the details of your code design, a last step will be to remove the constructor/destructor messages - you won’t be trying to match our code in this respect. But you should leave these message in as long as possible - it is an easy and fun way to watch the smart pointers at work and to be sure you really are releasing the object when you are done with it.

Step 3. Apply Model-View-Controller and extend the View class to different kinds of views.

This step is an example of a common activity during software development. It is discovered that it would be a good idea to have variations on a capability already present. In this case, the View from Project 4 is just fine, but we want additional kinds of View as well, and the ability to have more than one View simultaneously active. In addition to modifying Model to handle any number of view objects, this requires refactoring the View class from Project 4 in some way. We want to retain the basic Model-View-Controller (MVC) logic in Project 4, but simply supply additional kinds of Views, and make them very flexible and under the control of the program's user. The overall structure will be very similar to how GUI code is normally organized, making this project a much better example of the MVC pattern and how it is used. Thus, you will be designing a set of View classes and arranging Model to implement this pattern. This is a significant design project; be sure to give it some time.

Model-View-Controller in Action

It is important to follow the MVC pattern closely; review the lecture notes and the Project 4 discussion. Let’s summarize the responsibilities and collaborations of the involved classes:

- ***Controller*** responds to user commands, and is responsible for creating Views and attaching them to the model, or detaching them and destroying them. Because the user specifies the kind of view he or she wants, Controller cares what the different kinds of Views are, but it is the ***only*** component that does.
- ***Model*** keeps track of the Views currently attached, and will broadcast information to them, but does not need to know and does not care what the different kinds of Views are - so far as it is concerned, there is only one kind of View.
- ***Views***, following the approach in Project 4, do not know anything about different kinds of Sim_objects; they merely display information about names and numeric values in a format that depends on the type of View. Model passes this name and numerical value information to them. Thus, Views are decoupled from Sim_objects in addition to Model and Controller.
- ***Sim_objects*** don’t know anything at all about Views, but will notify Model to pass their data to the Views, taking advantage of Model’s global availability as a Singleton.

The last bullet point above refers to a basic change from Project 4 in which Model “pushed” the location information to the View after each update for objects that *might* move, Instead, Model will rely on the Sim_objects to supply *notifications* of their data when they individually change state, as follows:

- Whenever an individual object changes in a relevant way, it will ask Model to notify all of the attached Views. For example, if an Agent moves, it will tell Model to notify the Views that the object named Zug is now at (20, 30). If a farm updates to have more food, it likewise tells Model to notify the Views that the object named Rivendale now has 56 amount.
- If Controller attaches a new View to the model, then Model asks all objects to tell Model their current information to be broadcast to the Views, so that the new View is able come fully up to date (regardless of what kind of View it is).
- Similarly, if an object is added to Model, Model asks the object to notify Model of all its current information for broadcast to all current views.
- If an object is removed from Model (e.g. a dead Agent), then Model notifies all attached views that the object is gone.

Kinds of Views and New Commands

The Project 4 View, called a ***map view*** in this project, will still be available; the user can still control its origin, scale, and size. You can definitely recycle this code. There are three new kinds of views; see the posted samples to get a quick overview. They are:

A ***local view*** looks like the map view, except that it is smaller and is centered on a Sim_object’s location. If the object is a Structure, then the “window” on the world stays in a fixed location. If the object is a moving Agent, then the “window” moves with the Agent, meaning that its X and Y axis labels change as the Agent moves. The user cannot change the size or scale of a local view - it

corresponds to a map view with a size of 9 and scale of 2, and its origin is adjusted to correspond to the location of the designated object. The local view output includes the object that the view is centered on. Unlike the map view, objects outside the view are not listed. If the object originally associated with a local view is gone, then the object no longer appears in the view, but the view remains centered at the last location until the user closes it.

> *Calculation specifics:* You can use the Project 4 map view calculations by setting the map view origin x and y to the centered object's current location x and y coordinates - (size / 2.) * scale.
>
> *Note:* The behavior of a local view for a removed object if a new object of the same name is created is undefined.

A ***health view*** and ***amount view*** are identical in format. The view lists the object names and either the current health of the object or the amount it is currently carrying, listed in alphabetical order by object name. If an object does not have a health (like a Structure) or an amount (like a Soldier) it simply does not appear in the list.

Initially, no views are open - not even the map view. The user commands Controller to create, destroy, and show Views with the following commands:

- **open <view name>.** If the <view name> is `map`, `health`, or `amounts`, then the corresponding view is created and attached to Model. Otherwise, a local view for the named object is created and attached. If a view of this name already exists, an Error exception is thrown. If the name is not recognized as a view type or one of the current sim_objects, an Error exception is thrown.
- **close <view name>**. The view of that name is detached from model and destroyed. If no view of that name is present, an Error exception is thrown.
- **show**. Controller commands all of the current views to draw themselves. They appear in the order that they were opened.
- **default, size, zoom, pan.** Check first for an open map view, and throw an Error exception if the map view is not open. Then read and process the parameters of the command as in Project 4.

Notice that you can open any mixture of views, in any order, and a separate local view for each object known to model, and close any one of them at any time.

Automate memory management. Use smart pointers to Views to automate disposing of the Views. The end result is that there should be *no raw pointers to objects anywhere* in your Project 4!

> *Possible exception*. Since using a smart pointer for a Singleton doesn't make any sense, you can have Model * pointers for your Singleton, depending on how you implement it.

Design Goals and Constraints

For this to be the best exercise, your design must have a separate class for each kind of view, as opposed to different "modes" of fewer classes. However, you can have more than four classes if it helps achieve a good design. Plan to refactor your Project 4 View class.

The basic goal is to add these additional view capabilities to the program in a way that results in ease of extension - if we add additional different kinds of Views and different kinds of Sim_objects, we should have to modify little or no code to fit them in. Following the Model-View-Controller pattern (see above) is critical to a good design. You need to add the notification mechanism to Sim_objects in a way that meets the extensibility goals.

You also need to arrive at a good class design for the four different kinds of Views. A good way to tell whether you have a good design is to consider whether (1) you can add a new kind of Agent or Structure that has the same kinds of data (location, amount, health) with no change to either Model, Views, or Controller; or (2) you can add a new kind of view of the same kinds of data - e.g. one that showed the health of agents on a map along with their locations - without any change at all to Model or the Sim_objects, and trivial changes to Controller; or (3) we can add a View showing a new kind of data by making only the few and simple additional modifications required by the new type of data to Model and the Sim_objects.

To give you maximum flexibility in your class structure while keeping the autograder setup simple, all of your views declarations and definitions will be in a new pair of files Views.h, .cpp (notice the plural!). You won't be using View.h, .cpp any more - do not attempt to submit it.

Suggestion: Set up the complete MVC pattern with the existing Project 4 map view. Get it working. You should now be able to add another view types- say the health view - with no hassle. If so, then do the design and refactoring work for the remaining View classes, implement them, and you're done!

Step 4. Add an Archer Class.

An Archer is in many respects like a Soldier, but has more autonomy; it is both more aggressive and more cowardly than a Soldier. An Archer will automatically start shooting arrows (with "Twang!") at the closest Agent that in its range, but if attacked, will run away to the closest Structure. It isn't smart enough to know whether that will actually protect it or not - so if it gets attacked in Paduca, it doesn't know it should run away to somewhere else - Paduca is still the closest structure! More specifically, Archer's behavior is just like Soldier except for the following:

- **Initial values.** Archer has an attack strength of 1, a range of 6, and outputs "Twang!" (of a bowstring) when it hits its target (instead of Soldier's "Clang!").
- **update**. Do the following in this order:
  - First update the Agent state and if Attacking, do the same checks and actions as specified for Soldier::update. If our previous target is now dead at this point, we will be in a Not Attacking state.

    - If we are now Not Attacking, we need to look for another target. Find the closest Agent (different from this one). If this Agent is within range (distance is less than or equal to the Archer range), start attacking it, outputting the message "I'm attacking". In case of a tie for closest distance, use the target whose name comes first in alphabetical order (this can be implemented trivially).
- **take_hit.** First call lose_health() and do nothing further if now Dead. Otherwise, if the attacker is still alive, find the closest structure, output "I'm going to run away to" that structure, and self-command move_to that structure. Unlike Soldiers, Archers do not automatically counter-attack their attacker. In case of a tie for closest distance, use the Structure whose name comes first in alphabetical order (this can be implemented trivially).
- **describe.** Identical to Soldier, except "Archer" appears instead of "Soldier."

Modify Model's constructor to initially create an Archer named Iriel at (20, 38), which is a convenient position for Zug to move closer - see what happens!

Design Goals and Constraints

Your design goal is to arrive at a new version of the Agent class hierarchy in which both Soldier and Archer are well represented. This is an exercise in applying the course guidelines for good Object-Oriented Design, and it should be done with care. The point here is to demonstrate how you can add a new derived class with minimum changes to the rest of the code. Note that your new class should immediately work correctly with the new Views from Step 4. Therefore, you are not allowed to modify the public interface of Agent in this Step because this might break the rest of the program. However, as long as Agent and Peasant are unchanged, you are free to refactor Soldier, and/or add additional classes, if that will help achieve a good design of these "warrior" classes. The above description of Archer should not be taken as prescribing a class relationship between Archers and your previous Soldier class. You may want a new version of the Soldier class as part of the design. To allow you maximum flexibility but still keep a simple specification for the autograder, all of your code relevant for Archers and Soldiers must be in a new pair of files, Warriors.h, .cpp; the Soldier.h, .cpp files will not be used and must not be submitted.

Step 5. Remove unnecessary constructor/destructor functions and messages.

You should keep these functions and/or messages in your program until the very end, to make sure that everything is being destroyed when it should be. Just remove them before submitting your project. As mentioned before, the order in which objects get destroyed will be rather different from Project 4, but every object that gets created should eventually be destroyed, at the earliest, when it is no longer needed, and at the latest, when the program is terminated.

Important: To demonstrate that you know when you need to define constructors and destructors versus when the compiler-supplied ones will work correctly, you need to not just delete or comment out the message output statements, but you need to delete or comment out the entire constructor and destructor function declarations and definitions if they are not actually required for the program to work correctly (including deleting all allocated objects at termination).

Files to submit

Submit the same set of files that you did for Project 4 except as follows:

- **p5_main.cpp** instead of p4_main.cpp.
- **Warriors.h, .cpp** instead of Soldier.h, .cpp. All of your code for "attackers" must be in this .h/.cpp pair.
- **Views.h, .cpp** (notice the plural) instead of View.h, .cpp. All of your code for the views must be in this .h/.cpp pair
- **Smart_Pointer.h** (but only if you are using it. It must be the version supplied in the examples directory on the course website.)

General Requirements

To practice the concepts and techniques in the course, your project must be programmed as follows:

1. The program must be coded only in Standard C++.
2. You must follow the recommendations and correctly apply the concepts presented in this course. This means you must be familiar with the relevant material in Stroustrup, the posted C++ Coding Standards, and the lecture notes. At the least you must review your code against these sources before submitting it.
3. Your use of the Standard Library should be straightforward and idiomatic, along the lines presented in the books, lectures, and posted examples.
4. You must use the smart pointers correctly, following the guidelines and concepts for their use. For example, a design problem in the supplied Smart_Pointer.h Reference_Counted_Object class is that the increment and decrement functions are publicly available, but should not be called by anything else but the Smart_Pointer template code. tr1::weak_ptr should be used when it is appropriate, rather than tr1::shared_ptr with additional code to solve a cycle problem.
5. You must not use any switch-on-type logic, type codes, RTTI, or dynamic_casts in the program. Seek advice if you think you must use these or you think they would be a good idea.

Project Evaluation

This will be the last autograded project in the course. Because the design is more open, and you are allowed to modify public interfaces to the limited specified extent, component tests will not be performed. We will test your program's output to see whether your Project 4 classes and the new Archer and views behave according to the specifications.

In Project 6, the second phase, you will be choosing features to implement and designing how to implement them, so you will supply some design documentation and demonstration scripts along with your final source code. Project 6 will thus not be autograded, but human-graded throughout. The course schedule simply does not make it possible to complete a Project 5 design evaluation in time for you to respond to it with Project 6. Thus, you will get an autograder score only on Project 5, and then Project 6 will fully human-graded, but it will be evaluated for both the Project 5 design concepts and the Project 6 design requirements. You will have to build your Project 6 based on your version of Project 5. This arrangement is somewhat awkward, but past experience is that it works. One thing that makes it work is the following nasty warning:

Nasty Warning: We will spot-check that the required components and design concepts appear to be present in Project 5; if not, your autograder score for Project 5 will be *considerably* (!) reduced, possibly even being set to **zero**. In other words, you should design and write Project 5 well so that you can then focus on the design problems in Project 6. If you blow off Project 5, you will be sorry when you do Project 6, and will be doubly sorry if we decimate your Project 5 autograder score.

The code and design quality for both the Project 5 and Project 6 components will be weighed extremely heavily in the final project evaluation. It is absolutely critical not only that you design and write your code carefully, but also that you take the time to review and revise your code and your design, and ensure that your final version is the best that you can do. This care needs to cover both the Project 5 and Project 6 work.