Project 5
Extensibility with Design Patterns and Polymorphism
The first part of a two-part project

Due: Friday, November 22 , 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 an extension 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 a two-phase final project.

In this project you will:

- Explore the alternative to private inheritance for implementation re-use.
- 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 out about each other and indirectly broadcast to Views.
- Simplify memory management using the Standard Library smart pointers.
- Get additional practice designing and refactoring your own classes to support two new features:
  - Apply the Model-View-Controller pattern more completely by implementing additional kinds of views in the form of a polymorphic class hierarchy that you design, along with the changes to Sim_object, Model, and Controller to work with the new view types.
  - Add a new derived class of Agent, observing how only the code required to support the new class needs to be added, or possibly refactored, while the remaining code in the project is unmodified, and existing features, like the new kinds of views, automatically work with the new kind of Agent.

This project assignment attempts to leave as much of the design of the new features 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 must or must 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. You are free to modify your Project 4 code, except where there are a few constraints on your design (more details below). However, you should not have to make big changes to the Sim_object class hierarchy — it does most of what you need. You will have to add to this hierarchy, design a new class hierarchy for View, and modify Model's services and Controller's commands to accommodate these. In other words, your Project 5 solution should be clearly based on Project 4's, although it will change some of Project 4's code. 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. Eliminate private inheritance to re-use implementation.

Now that you have had the thrill of multiple and private inheritance, it is time to follow the expert wisdom which says *If all you want to do is to re-use implementation provided by an existing class, instead of inheriting privately from that class, just declare and use a member variable of that class*. This is called "composition," "aggregation," or a "has-a" relationship.

So modify your Agent class so that it no longer inherits from `Moving_object`, but instead declares a `Moving_object` object as a private member variable of Agent. Now Agent "has-a" `Moving_object` instead of Agent "is-a" `Moving_object` (which is actually bogus – review the substitution principle and how it doesn't work for private inheritance). Fix Agent's member functions to call this variable's member functions instead of directly calling the functions inherited from `Moving_object`. The program behavior should be unchanged.

Step 2. Make Model a Singleton class.

Once you complete this step, your program should run identically as before, but now it will be easier to do the remaining steps, which require program-wide access to information and functions in Model. Be sure you *completely* follow one of the 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. Your problem here is to find a good balance between useful specialization and more extendable generality.

Note: Project 4 used a global Model object accessed through a global pointer as a stepping-stone to a Singleton. 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. You must remove the global pointer variable and any stored pointers or references to Model that you had in your Project 4.

Step 3. Use Smart Pointers and Make Agents Die Immediately and Remove Themselves from the World.

Step 3.1. Change all containers and pointer variables used to refer to Sim_objects, Structures, Agents, etc., over to smart pointers using the C++ Standard Library reference-counted smart pointer classes; refer to the assigned Handout, and see the Example Code directory for examples of their use.

Who uses what kind of smart pointers? The key issue is what kind of smart pointers do objects store as member variables? One way to look at this is in terms of ownership, meaning that the component that is responsible for destroying the object should be the "owner" and everybody else is just an "observer." Without smart pointers, the code can be very hard to get right. But smart pointers don't help much if this ownership concept means very few `shared_ptrs` and lots of `weak_ptrs` throughout the project — working with `weak_ptrs` is very clumsy compared to `shared_ptrs`. More importantly, the main reason for using smart pointers is so that we don't have to complicate our program design by trying to impose a "pure" idea about who "owns" what. By using shared ownership, it is easy to ensure that the object gets deleted only when nobody else is interested in it, which means that there will never be a "dangling pointer" that points to a deleted object. The only place we really need "observers" with `weak_ptrs` is to prevent smart pointer cycles between objects that refer to each other.

So here are the specifications for what kind of smart pointers each component should use:

- The Model object has containers of `shared_ptrs` to all Sim_objects (which includes Structures and Agents) and Views. When Model is destroyed, all of the `shared_ptrs` to Sim_objects and Views should get automatically destroyed when its containers are destroyed. We never use `delete` in the program — we simply let a `shared_ptr` go out of scope, reset it, erase it from a container, or let the whole container get destroyed. Review when member variable destructors are called to make sure you don't waste time writing code that the compiler will create for you.
- In this project, only `shared_ptrs` will be passed as function arguments and returned values — never `weak_ptrs`.
- Agents that point to other Agents should just "observe" them using `weak_ptrs`, which prevents any cycle problems that might appear with Agents that are attacking each other.
- At least in this version of the game, Structures don't refer to Agents or other Structures so there won't be any cycle problems involving Structures. So Agents will keep `shared_ptrs` to Structures. When the program is terminated, the exact time a Structure gets destroyed will depend on when any Agents pointing to it get destroyed. By observing the destructor messages, you can see this happening.

Once you have switched over to smart pointers, you should not have any "raw" pointers for View or 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. Containers of smart pointers will destroy the contained smart pointers when they get destroyed, eventually automatically deleting all the pointed-to objects without your code having to explicitly clear the containers beforehand. 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.

- *Hint:* See if your IDE allows you to simply do a global search/replace throughout all of the project source files of "`Agent*`" with "`shared_ptr<Agent>`", etc. This will save a lot of time, but you may have to fix a few things.
- *Hint*. You will need to use the `enable_shared_from_this` facility when e.g. a Soldier calls `take_hit` on another Agent.

Step 3.2. 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 all Views and from all of Model's 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, make the following modifications:

Model class: Add a function to Model, `remove_agent(shared_ptr<Agent>)` that removes an Agent shared_ptr from all of Model's containers. Remove the code from Model::update() that scans for and removes **Disappearing** Agents. Instead, all update() should do is increment the time and then update each Sim_object, iterating through the Sim_objects container.

Important: Notice that with the changes described below, an Agent will not get killed when it is being updated, but rather when some other Agent hits it during that other Agent's update; so the killed Agent gets immediately removed from the container of Sim_objects while some other Agent is being updated. If your Sim_objects container was well chosen, and the updating loop properly implemented, this immediate removal of the killed Agent will not invalidate the iterator used in the updating loop.

Agent class: Modify Agent's `enum` for the Agent state to remove the **Dying** and **Disappearing** state so that Agents are now either **Alive** or **Dead**. Remove the `is_disappearing()` accessor. In `Agent::update()`, remove the cases for **Dying** and **Disappearing**, and ensure 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 next, it asks Model to remove itself from the Views (move this from `Agent::update()`), and finally, as the last statement before returning, `lose_health()` calls `Model::remove_agent()` with a pointer to itself!

Controller class: 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.

At this point, once an Agent is killed it gets removed from Model, Views, and the simulation right away.

Step 3.3. 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. Soldier keeps a `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.

Step 3.4. 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 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, a `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 needs only a slight modification from Project 4 in the checks for whether the target object is not alive. You can test to see if the target object no longer exists, or if it still exists but is dead. The target is considered "dead" in either case, and the target pointer can then be reset.

Step 3.5. If Agents point to each other with `weak_ptrs`, then the cycle problem will be automatically handled. But check to see that this is really at work. A good demonstration is to have two Soldiers busy attacking each other when you issue a "quit" command. They should both get deleted. If you were using `shared_ptrs` between them, neither will get deleted; with `weak_ptrs`, they both will.

Step 3.6. 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 its name will have been removed from Model — it will be an unknown Agent, not a dead Agent.

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, which might differ from the instructor's solution, a last step will be to remove the constructor/destructor messages — it doesn't make sense to try to match when they appear. 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 4. Fully Apply Model-View-Controller with 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. Model needs a container to hold any number of view objects. (specified in Project 4). The View class from Project 4 will need to be refactored in some way, and additional View classes added. 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 than Project 4. Thus, you will be designing a set of View classes and re-arranging Model to fully implement this pattern. This is a substantial design problem; be sure you can give it some time.

Model-View-Controller: Extensibility 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. Controller also controls which View draws itself at what time or in what order, but again, it is the ***only*** component involved in this.
- ***Model*** keeps track of the Views currently attached, and will broadcast information to all of them, *but it has no other responsibilities for the Views whatsoever, and does not need to know and does not care what the different kinds of Views are* — so far as Model is concerned, there is only one kind of View, and it doesn't care what order they got attached in. Giving Model any responsibilities that involve knowing anything about different kinds of Views, the order in which they were opened, or in any way providing management functions for Views, means Model is doing some of Controller's job — it breaks the pattern!

- ***Views***, following the "push" 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 via `update()` calls. 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 when their state changes, which will then pass the data to the Views at appropriate times, taking advantage of Model's global availability as a Singleton.

The last bullet point above refers to how the Sim_objects supply *notifications* of their data to Model when they individually change state, by calling functions such as `Model::notify_location(const string& name, Point location)`. Model will then broadcast the notification by `update()` calls to all of the currently attached Views. You'll have a Model notification and View update function for each type of information that Views might be interested in.

The main difference between this use of MVC and a typical GUI MVC is that normally in a GUI, when Model updates the Views, each View immediately arranges with the GUI run-time environment to redraw itself, while here, the View just "remembers" what it was told, and then draws itself when the user issues a **show** command to Controller. It works this way in this project (and the previous) because if our dumb text-graphics Views always drew themselves after updating, they would generate megabytes of text output, which would just get in the way.

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.

Note that like map view, a local view works only in terms of object names, not object identities. Thus if an Agent originally associated with a local view is gone, it gets removed from the local view, and so its name no longer appears in the view, but the view remains centered at the last location. The user can close the view using the original opening name even if the object is gone.

- *Calculation specifics:* You can re-use the Project 4 map view calculations by setting the view origin (x, y) to the centered object's current location (x, y) coordinates - (size / 2.0) * scale.
- *Note:* Suppose we have a local view for a removed object that is still open, and the user creates a new object of the same name. The resulting behavior is somewhat subtle, but very simple if the above description is followed, because only object names are involved in the view. Thus such a new object will "automatically" become the focus of the local view, without any special-case checking. However, if there was a local view open for Bug, who was killed and replaced with a new Bug, and the user tries to open a new local view with Bug, this should produce a "view already open" error — the local view is associated with object names, not object identities. So the old view will work fine to track the new Bug. If for some reason the user wanted to get a new local view for Bug, the user has to close the old Bug view, and open a new local view for Bug.

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. They are listed in alphabetical order by object name, with the numerical values shown with two places to the right of the decimal point for both health and amount. An Agent's health is currently specified as an int, but it can be simply (and implicitly) converted to a double when supplied to a view. If an object does not have a health (like a Structure) or an amount (like a Soldier) it does not appear in the list — such objects simply never notify any views about this property, so they never show up in the view. Any time an object's amount or health changes, it does the corresponding notification.

Initially, no views are open. 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.

Multiple view and multiple local views are a feature: 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.

Output order. The order in which views are output by a show command is the order in which they were last opened. (Project 4's pervasive alphabetical order rule does not apply to this.) For example, if the user opens a local view for Zug, then the map view, then a local view for Bug, they will be displayed by the show command in the order: Zug, map, Bug. If the Zug view is closed, then of course it is no longer output. But if the user opens a local view for Zug again, then the order of display will be map, Bug, Zug. Likewise, the health and amount views will be output in the order in which they were opened relative to the other views.

Important: the name strings for the views {`map`, `health`, `amounts`} must be disallowed as the names for new Structures or Agents. Otherwise, because of the command syntax, it will be impossible to open that kind of view!

Also important: The Controller will have a more complex set of possible states than in Project 4, so to get a bit more practice with pointers-to-member functions, your Controller must continue to follow the Project 4 requirement that the member functions of Controller must be const-correct, and a member function is called for a command using one or more map containers of strings to pointer-to-member functions.

Automate memory management. When a view is opened, a View object must be dynamically allocated; when closed, the object is deallocated. Use smart pointers to Views to automate disposing of the Views. The end result is that there should be *no raw pointers to dynamically allocated objects anywhere* in your Project 5!

- *Possible exception*. Since using a smart pointer for a Singleton doesn't make any sense, if you implement the Model Singleton in terms of a static Model* pointer, then this will be the only raw pointer in your code.

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 classes if it helps achieve a good design. You need a base class for the different kinds of Views, so this gives at least five classes — one base, and at least one class for each of the map, local, health, and amounts views.

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 in the future, 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 different kinds of Views. Plan to refactor your Project 4 View class. 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 notification data — e.g. a "combat" view that showed the health of agents on a local map, or a "agriculture" view that shows only objects that have reported amounts (including zero) — without any change at all to Model or the Sim_objects, and requiring only trivial changes to Controller to create the new kinds of view; 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, the View base class, and the Sim_objects. *Hint:* To get this flexibility, do not force the View system to work in terms of fixed "bundles" of data — separate notification/update functions for each kind of data is a more flexible and efficient solution.

To give you maximum flexibility in your class structure while keeping the autograder setup simple, your *base* class of views must be submitted in `View.h, .cpp`, which should contain *only* the base class code, and the declarations and definitions of all of your other view types will be in a new pair of files `Views.h, .cpp` (notice the plural!). This setup violates the normal custom in which each class has its own .h, .cpp file pair; but if I dictated which separate file pairs you had to submit, it would come too close to dictating the design, which would make the exercise less valuable. All four of these files (View.h, View.cpp, Views.h, Views.cpp) must be submitted even if one or more of them are empty. It is expected that View.h, .cpp will not have the same code as they did in Project 4. Review the MVC concepts if you are unsure about which classes should be in which files; this decision is a key part of the design.

Some Guidelines for Good OO Design

Review the notes on OO design. Some key concepts:

- Only leaf classes should be concrete — that is, don't have one concrete class inheriting from another concrete class. If this seems worth doing, it is because there is some shared functionality in the two classes that should be expressed in an intermediate base class they both inherit from.
- Put shared functionality as far up the inheritance tree as it makes sense to do. This means that the stuff done by one or more of the concrete classes should be moved up into a base class, but not so far up the tree that it becomes irrelevant for some of the other derived classes.
- Base class member functions that provide services to only derived classes should be protected. Keep base class member variables private — don't make them protected. If you are tempted to make the member variables protected, or you find you have to provide a full set of protected getter/setter functions for the member variables, then something is almost certainly wrong with the design.
- An intermediate base class should represent a meaningful abstraction in the domain. If a base class has only member variables and all of the real work is being done by functions in the derived classes using these variables, then this base class is not doing any real work and doesn't really represent anything — the nature of the member variables is an implementation detail, not a conceptual abstraction in the domain. In other words, putting just member variables in a base class does not count as "shared functionality being moved up the inheritance tree" or a "meaningful abstraction in the domain." Only member variables + substantial member functions count as shared functionality.

Rethink and Refactor. If you have problems with any of these guidelines, you need to rethink the design: what do the concrete classes actually have in common? Can you refactor the code (reorganize what the functions do) so that all of the shared work is done by the base class member functions, leaving the concrete classes to do only their own specialized bits? Maybe all they need to do is provide certain initial values, function parameters, or override a little virtual function.

Step 5. Add an Archer Class.

The description of this new class is limited to describing just the behavior, so that you can design how it gets implemented, and how it relates to the existing Soldier class. An Archer is in many respects like a Soldier, (and identical in behavior except where described below), but has more autonomy; it is both more aggressive and more cowardly than a Soldier. In overview, an Archer will

automatically start shooting arrows (with "Twang!") at the closest Agent that is in 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, including how it responds to commands, 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**. In general, Archer will check the target state, the target range, hit the target, and check again, and stop the attack if necessary, just like Soldier, but if as a result, the target is either dead or out of range, Archer will then look for a new target. More specifically, do the following in this order:
  - First update the Agent state and then 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 there is no closest Agent, return, doing nothing further. 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). Consider how to get the closest Agent and see if you can code it well with use of STL algorithms.
- **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). As with closest Agent, see if you can f ind the closest Structure with a good design and use of the STL algorithms.
- **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 above list of key guidelines applies to this problem as well.

The point here is to demonstrate how you can add a new derived class with minimum changes to the rest of the code. For example, 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 to 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. As with the Views, this setup violates the normal custom in which each class has its own .h, .cpp file pair; but if I dictated which separate file pairs you had to submit, it would come too close to dictating the design, which would make the exercise less valuable.

Step 6. Remove unnecessary constructor/destructor functions and all their 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: You need to demonstrate that you understand when constructors and destructors actually need to be explicitly defined and when they do not. So, eliminate these messages by deleting the ***entire*** constructor or destructor function if this is possible to do without disrupting correct program functioning. If the constructor or destructor function must be defined for other reasons, then delete the message outputting code from it (don't just comment it out).

In other words, *it is a design and code quality error if your final code has any unnecessary declarations or definitions of destructor functions*.

Files to submit

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

- the supplied **p5_main.cpp** (use as-is) 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.
- **View.h, .cpp** for the single base class of Views that you define.
- **Views.h, .cpp** (notice the plural) for the classes for the specific types ofViews that you implement (including the map view).

General Requirements

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

- The program must be coded in Standard C++17; C Library functions should not be used (with the possible exception of `<cmath>` and `<cctype>`.
- 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. You must review your code against these sources before submitting it.
- Your use of the Standard Library should be straightforward and idiomatic, along the lines presented in the books, lectures, and posted examples. As in Project 4, be sure to use the algorithms, iterators, bind, lambda, etc. to get simple and clear code.
- You must use the smart pointers correctly, following the guidelines and concepts for their use.
- Let virtual functions do the work: you must not use any switch-on-type logic, type codes, RTTI, or `dynamic_cast`s 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. I 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:

Warning: I will spot-check that the required components and design concepts specified in this project appear to be present; if not, your autograder score for Project 5 will be *considerably* (!) reduced, possibly even being set to **zero**. In addition, I will spot-check for certain general code quality issues, such proper header file discipline and following namespace guidelines.

In other words, you should design and write Project 5 well so that you can then focus on the design problems in Project 6.

The code and design quality for both the Project 5 and Project 6 components will be weighed extremely heavily in the final project evaluation. Please study any negative feedback you got on Project 3, asking me for clarification or explanation, and make sure that your code is better this time. It is absolutely critical that your final version of these last two projects is the best that you can do.