Project 5
Improving Existing Code: Smart Pointers and Class Design
The first part of a two-part project.

Due: Friday, March 31, 2017, 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 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

Now that we have a working framework for the maritime simulation, it's time to have some fun with it. So this project is a extension/elaboration/refinement of Project 4. You will need a working version of Project 4 as a "starter." This project will provide a foundation for the final Project 6, and so is actually the first part of a two-part project. In this project you will:

- Try out heterogenous lookup if helpful.
- Use the protected constructor idiom to prevent class instantiation instead of a pure virtual destructor.
- Try "aggregation" instead of private inheritance to re-use functionality.
- Try out the C++11 smart pointers to automate memory management.
- Apply the Singleton pattern to make Project 4's Model globally accessible in a more reliable way.
- Demonstrate the value of an OOP architecture by adding a new derived class of Ship, observing how only the code required to support the new class needs to be added, while the remaining client and other code is unmodified. As the gurus say, "*inherit in order to be reused*"; "*add features by adding code, not by modifying code*".
- Follow up on the Model-View-Controller pattern introduced in Project 4 by designing and implementing a set of classes that provides both the Project 4 View capability and two new kinds of View, one of which can have multiple instantiations at run time.
- Get a bit of further design practice by designing and implementing another kind of Ship similar to Cruiser.

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 guidelines presented thus far in the course. While you are free to modify Project 4 as needed, you should not have to make major changes to the Sim_object class hierarchy. Mostly, you will be adding to this hierarchy, and then 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. 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 sailing in this project.

Step 1. Consider heterogenous lookup.

If you didn't use heterogenous lookup in Project 4, consider how it might simplify your Model containers: A good choice of containers and lookup methods can make this code much simpler, more efficient, and maybe less of a kludge. There are choices of containers that work very well without heterogenous lookup, but there is no excuse for clunky and clumsy choices. Be sure to double-check your container code throughout the project to make sure you are not doing something clumsy and inefficient, such as searching the same container twice for the same thing.

Step 2. Use the protected constructor idiom to prevent instantiation.

Project 4 had you declare some classes with pure virtual destructors in order to make the class abstract, so that only objects from leaf class like Tanker or Cruiser could be created. To be clear, making the destructor pure virtual is only necessary to make the class abstract if there are no other pure virtual functions in the class.

In this project, for classes that do not have any pure virtual functions in their interface, instead of making the destructor pure virtual, we will use the conceptually simpler approach of declaring the constructor for intermediate classes like Ship to be protected, so that client code cannot instantiate them, making them “quasi-abstract.” So find all of the pure virtual destructors in your Project 4 code and make them non-pure, and declare the constructors in those classes to be protected. Remember that if a base class destructor as been declared virtual, then all derived class destructors become virtual as well even if you do not explicitly declare and define them.

This project involves designing some additional classes some of which might be base or intermediate (non-leaf) classes. If you have a new base or intermediate class that has no pure virtual functions in its interface, then make the class “quasi abstract” by using the protected constructor idiom. Thus in your final version of this project, (1) there should be no pure virtual destructors, and (2) it should not be possible for the relevant client code to create an object from any base or intermediate class, and (3) constructors are protected only if the class is not otherwise abstract.

In the last step of this project, you will be required to eliminate all destructor declarations and definitions that are not actually required for correct program builds and behavior, but leave them in for now because the destructor messages will help demonstrate the use of smart pointers in Step 4 in this project.

Step 3. Use aggregation instead 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 "aggregation" or a "has-a" relationship.

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

Step 4. Switch over to smart pointers.

In the real world, we probably would have started with smart pointers from the beginning. Change all containers and pointer variables that refer to Sim_objects, Islands, Ships, and Views, over to smart pointers using the new C++11 smart pointer classes, shared_ptr and weak_ptr. Notice that you can declare one of these without a complete declaration of the pointed-to class. For example, just like for Ship* pointers, your Model.h header file can declare a container of shared_ptr<Ship> with only an incomplete declaration of Ship.

See if your IDE allows you to simply do a global search/replace throughout all of the project source files of "Ship*" with or "shared_ptr<Ship>", etc. This makes the change-over very easy. You will have to do some surgery on code that needs to use a weak_ptr, or calls another function with "this" as a parameter. Note that std::bind and std::mem_fn "understand" shared_ptr, so if for example, you were using for_each over a container of Sim_object* with a bind or mem_fn, the same code will work after you change the container to have shared_ptr<Sim_object> elements. When objects are created, you can use either std::make_shared, or create shared_ptrs with "new" as the constructor argument - your choice.

Once you have switched over to smart pointers, you should not have any "raw" pointers for Sim_object family or View 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, and the compiler-supplied destructor code will destroy those containers, eventually automatically deleting all the pointed-to objects. This really makes cleaning up simple! Review when member variable destructors are called to make sure you don't write code when the compiler will do the work for you. Look especially for unnecessary calls to clear() on a container.

Finally, notice how we get some basic exception safety for free if we use smart pointers idiomatically: Unlike previous projects, when inserting newly allocated object pointers into a container, we no longer need local try/catches around the insertion to clean-up the new object.

Who uses what kind of smart pointers? At this time, there seems to be some debate about the best way to use different kinds of shared_ptrs. Some people have advocated using shared_ptrs only for code which owns the objects, and raw pointers to the same objects elsewhere. But such mixed pointer code can be hard to get right, and as a project develops, you might find yourself having to change some module over from using raw pointers to shared_ptrs when you discover that it needs shared ownership after all. My experience has been that overall the situation is simplest if you follow the advice in the C++11 Smart Pointer handout: either use smart pointers, or raw pointers, for a group of objects; don't mix them!

The next issue then is what role shared versus weak pointers should play in the project. We could analyze this in terms of a strict model of ownership, meaning that the component that is responsible for creating and destroying the object should be the "owner" and everybody else is just an "observer." From this point of view, only Model really owns the Ships and Islands, and only one of the containers is really needed to be the site of this ownership, so only the Sim_object container would contained shared_ptrs, and everything else throughout the project would use weak_ptrs. There are two problems with this analysis. A practical problem is that working with weak_ptrs is very clumsy compared to shared_ptrs, so this decision makes the project code very awkward. The second and more important problem is that this strict model of ownership might turn out to be incorrect in some way as we continue to develop the project. Do we really want to bet that we can devise from the beginning a correct idea about who "really owns" what? A major advantage of smart pointers is that they automatically take care of objects even if the ownership situation is complicated. So if we use shared_ptrs widely, then we get simpler and easier-to-develop code. The only place we *really need* "observers" with weak_ptrs is to prevent smart pointer cycles between objects that point to each other.

This analysis isn't guaranteed to be correct or the best idea; we'll have to see how the C++ community develops its thinking on this issue. But it motivates the following specifications for what kind of smart pointers each component should use in this project:

- Only shared_ptrs will be passed as arguments to functions or returned as values - never raw pointers, and never weak_ptrs.
- The Model object has containers of shared_ptrs to all Sim_objects (which includes Ships and Islands) and Views.

- The Controller object keeps shared_ptrs to the Views that it is managing, and (temporarily) to Ships and Islands during command processing.
- Ships that point to other Ships should just "observe" them using weak_ptrs, which prevents any cycle problems that might appear with Warships that are attacking each other.
- At least in this version of the ~~game~~ simulation, Islands don't refer to Ships or other Islands so there won't be any cycle problems involving Islands. So for simple code, Ships will keep shared_ptrs to Islands. When the program is terminated, exactly when an Island gets destroyed will depend on when any Ships pointing to it get destroyed. By observing the destructor messages, you can see this happing.

Step 5. Simplify sinking.

Once you have the smart pointers in place, it is now possible to simplify how Ships disappear when they are sunk. To review, Project 4 avoided a dangling pointer problem by having the Ships go through a series of sinking states to ensure that any other Ship (e.g. an attacker) has a chance to disconnect its pointers before the sunken Ship was deleted. Model oversaw this process by waiting until a Ship was on the bottom, and then deleting it.

However, with smart pointers used throughout, there is no need for a long and painful sinking process. In fact, as soon as a Ship knows it is headed for the bottom, it will let everybody know, and even will ask Model to forget about it. So rather than Model figuring out when a Ship is gone, a Ship is the expert on when it is sunk, and it just tell Model. Other Ships simply discard their pointer to the sunk Ship when they know it is no longer afloat or in existence. We need only the Sunk state; we will discard the Sinking and On the Bottom state. Notice that some of the update and status messages left over from Project 4 may not ever appear in the output, but are still present as a debugging aid.

To implement this concept, make the following changes to the specifications and code from Project 4.

- Add a remove_ship function to Model:
  ```
  // remove the Ship from the containers.
  void remove_ship(shared_ptr<Ship> ship_ptr);
  ```
- Remove the code from Model::update() that scans for and removes sunk ships. Instead, all update() does is increment the time and then update each Sim_object. Notice that with the changes described below, a Ship will not get sunk when it is being updated, but rather when some other Ship fires at it during that other Ship's update, and the result will be that the sunk ship gets immediately removed from the container of Sim_objects. If your container was well chosen, and the updating loop properly implemented, this immediate removal of the sunk ship will not invalidate the iterator used in the updating loop.
- Change is_afloat() to return false if the state is Sunk; true otherwise.
- Remove the Ship::is_on_the_bottom() function.
- Move the check of resistance from Ship::update to the Ship::receive_hit function, whose specification is now:

  receive_hit. Subtract the amount supplied from the resistance, and output "hit with ... resistance now". If the resistance is now less than 0, output "sunk", set the state to Sunk, set the speed to 0, call Model::notify_gone for this ship, and then call Model::remove_ship for this ship.
- Change Ship::update from Project 4 to remove the check of the resistance if the Ship is afloat. If the state is Sunk, output "sunk" and do nothing further in this update.
- Change the description of Ship::describe from Project 4, replacing item #2 with:

  2. If the Ship is Sunk, output "sunk" and describe nothing further.
- Remove the other sinking states as possible values for the Ship state and any outputs that formerly depended on them.
- Change Cruiser::update() so that if its target is no longer afloat, or no longer exists, it terminates its attack and discards (resets) its weak_ptr to the target.
- Change Cruiser:: receive_hit so that after calling Ship::receive_hit, it does nothing further if no longer afloat. This will eliminate messages about trying to attack whoever sent it to the bottom.
- Because Cruiser won't "forget" its target until it updates, change Cruiser::describe() to output the supplied new "absent target" message in case your Cruiser describes itself while it is still in Attacking state and its target is sunk but not yet deleted, or no longer exists.

Once you do this, your program should run basically the same as before, but a sunk ship will no longer appear in the output for **go** and **status** commands, and you will see differences in the order of destruction of objects, based on the details of which containers get emptied when, or which smart pointers get destroyed. I suggest leaving the destructor messages in as long as possible so that you can observe this interesting "garbage collection" at work, and verify what happens when you add the remaining program features.

Step 6. Make Model a Singleton class.

Refer to the lecture notes for this pattern, and how you apply it. Remove the global Model* pointer and all references to it in your code, and replace them with Model::get_instance() calls. Once you complete this step, your program should run identically as before, but notice how the supplied p5_main.cpp no longer has to create and destroy the Model object. The Model Singleton magically creates itself when it is needed! While you are messing with Model, create another Island in Model's constructor, to be "Treasure_Island" at (50, 5), 100 tons of fuel, production rate of 5.

- Use the Singleton idiomatically to take advantage of the fact that your code does not have to keep track of when the Singleton gets created. So do not mess with clumsy “shortcuts” like storing a pointer or reference to the Singleton, but instead call its get_instance() function every time to you need to access it. If this strikes you as clumsy, you can use a shorter function name.

- Note that if you use the version of Singleton that has a static pointer member variable for the Singleton object, this pointer does not need to be a smart pointer - and it fact it makes no sense at all for it to be a smart pointer. If you don't see why, meditate on it and then ask me about it.

Step 7. Add a Cruise_ship class.

A Cruise_ship has the capability to automatically visit all of the islands in a leisurely fashion. A Cruise_ship can be created using the create command, like the other kinds of ships. Because it inherits all of the behavior of a Ship, it behaves like a normal ship until you tell it to go to an island with the "destination" command. It then announces that a cruise is starting and then it visits the specified destination island, then visits the closest next island, followed by the island closest to that, and so forth, visiting each island only once. It travels at the speed specified in the initial command. When it arrives at an island, it docks, and then it stays for a few updates. On the first one, it attempts to refuel from the island. On the next update it does nothing while the passengers see the local sights. On the third update, it sets course for its next destination (the closest unvisited island; in case of a tie, the first in alphabetical order). When it has visited the last island, it returns to the first island, the one it was originally sent to. When it arrives there, it docks, does not try to refuel, and announces that the cruise is over.

If during a cruise, the Cruise_ship is given any navigational commands (stop, course, position, or destination), it cancels the cruise and then follows the command using the already-existing Ship functions. Canceling a cruise means: output a message that the cruise is canceled; discard the information on where it has been; and do not start or continue on a cruise if it arrives at an Island. The effect of a navigational command after canceling the cruise is defined by the already existing Ship navigational command functions. If the navigational command is incorrect (e.g. an impossible speed), the cruise is still canceled, but the Cruise ship will react to the bad navigational command like any Ship does (e.g. it keeps going on its current course). However, if the command is correct the Ship will respond normally. If the command is correct and is another destination command to go to an Island, it will cancel the current cruise and will set up to start a new cruise once it arrives at the newly specified destination.

A fine point: If the Cruise_ship is already at an Island, and you tell it to go to that Island, then on the first update, the ship will realize that it is at its first destination and proceed with the cruise from there. This should happen if your code handles the other cases correctly; it should not be necessary to make it a special case.

You will need to make one or more small additions to Model in the form of services that the Singleton Model provides to the rest of the program. In particular, Model needs to make Island location information available to anyone who needs it now or in future versions. You need to do this in a way that is likely to be generally useful if other kinds of Ships are added in the future.

For simplicity in this project, do not try to compute the route in advance. Rather, at each Island, simply compute the next Island to visit, go there, and repeat until all Islands have been visited.

During updating, if a Cruise_ship discovers that it cannot move, and a cruise is underway, it cancels the cruise as described above.

New command information. The create command creates a Cruise_ship when given the type string "Cruise_ship".

Specific data: A Cruise_ship has fuel capacity and initial amount 500 tons, maximum speed 15., fuel consumption 2 tons/nm, and resistance 0.

Design constraints: You are not allowed to modify the public or protected interface of Ship to add the Cruise_ship class; you have to work through the available virtual function and protected function interface defined by Ship. You may not use any downcasts, safe or not. The idea is to demonstrate software "plug and play" by taking maximum advantage of the existing Ship family structure and commands, so that Cruise_ship is implemented only by adding the Cruise_ship class and adding a couple of lines to the ship factory code.

Error handling. The design goal is to take maximum advantage of the public and protected interface of Ship, and you are not allowed to change this interface in order to implement Cruise_ship. Since the Ship navigational functions can throw exceptions, your Cruise_ship code should call them in such a way that an error will not leave your Cruise_ship in an invalid state. For example, if a Ship navigational function is called to set the destination position and speed, but the supplied speed is not possible, that Ship function will throw an exception. Thus in the Cruise_ship navigational functions, first cancel the current cruise and output the cancel message, then call the Ship navigational functions, and then set any new Cruise_ship state and output any corresponding messages.

Step 8. 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 two additional kinds of View, and the ability to have more than one View simultaneously active. In Project 4, Model was supposed to have a container of Views, so that more than one View could be attached, and updates would be broadcast to all of the attached Views (if not, review the Project 4 specification and fix this before proceeding). Adding the new kinds of View 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. Your code should follow the presented MVC pattern very carefully.

The main difference between this use of MVC and normal GUI MVC is that normally in a GUI, when Model updates 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. 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 output, which would just get in the way.

The Project 4 View, called a *map view* in this project, will still be available, and the user can still control its origin, scale, and size. You can recycle the code with little modification. A major difference is that the user can open and close this view; it only appears as a result of the **show** command if it is open.

One kind of new view is the *sailing data* view. This is a "statistics" view that shows the current fuel amount, course, and speed for all ships. As the ships move about, they supply notifications whenever one of these quantities changes. Like the map view, there can be only one sailing data view open at a time.

A second new kind of view is a *local view*. See the samples. It is a miniature version of a map view; when it is opened, the name of a ship is specified. Thereafter, the local view is centered on that ship's current location. As the ship moves, Islands and other ships appear in the view and move through it, while the specified ship is always in the center of the view. There can be more than one local view open at a time, but only one per ship.

We carry over from Project 4 the concept that a notification should be made whenever a relevant value changes, regardless of whether it changes as a result of a user command issued between "go" commands, such as setting the course and speed, or happens during the update produced by a "go" command. Thus some changes will be visible in the views "show" command even if a "go" command has not yet been issued.

Sailing data view specifics

See the posted samples and strings.txt. When this view is output, it shows in textual form the current fuel, course, and speed of all of the ships. Any time a ship's amount of fuel, course, or speed is changed, this view gets notified so that the up-to-date information appears on the next show command. When a ship is sunk, it no longer appears in the view.

Details. The display has a title line, a column header line, and then a line for each ship containing the ship name, its current fuel, course, and speed. The ships are listed alphabetically by name. Each data value is assumed to be a double, and for simplicity, is output using our default formatting. Allow 10 characters for each field in the column headers and the ship name and data. If the name is too long, the view output will look ugly, but that's OK.

Things to get right. If the ship becomes dead in the water, or stops, or sinks, its speed must be set to zero and this must show in the sailing data view. However, the course is unchanged because the ship is still pointed in that direction even if it isn't moving. Likewise, the amount of fuel can change when the ship is docked as well as when it is moving. Carefully check that you've located all of the places where any of these three data values can get changed.

Local view specifics

A local view looks like the map view, except that it is smaller and is centered on the current location of the ship named when the local view was opened. A bit of nautical terminology: This ship will be referred to as "ownship" for the display. As the ownship moves, the "window" of the local view moves with it. 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 current location of ownship. The local view includes ownship. Unlike the map view, objects outside the view are not listed, and the x- or y-axes are not labeled. If ownship for a local view is removed (sunk), then that ship no longer appears in the view, but the view remains centered at the last location of ownship until the user closes it. The heading names ownship and gives it current location; if ownship has been removed (sunk), the heading changes to show that ownship is sunk at the last location it had. See the samples.

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.) * scale.

Details about sunk ownship. The behavior of a local view for a sunk ownship is somewhat subtle. The basic concept is that a view only knows about a sunken ship when it receives a update_remove() call. If a local view gets an update_remove() for ownship, it interprets this as meaning that ownship has sunk, and so the local view will no longer change its origin and its heading will changed to the "sunk" version, and the view no longer automatically shows the name of ownship in its center. The view has to remember that its ownship has been sunk. The view remains open until the user closes it.

Now for the tricky part: Suppose the user creates a new ship with the same name as the original ownship, and moves it into the local view's window. The local view remembers that its ownship has been sunk and does not treat the new ship of the same name as it's ownship, but simply plots it in the view as it would any other ship. In other words, the view for a sunk ship stays stuck at the last location, and a new ship of the same name is treated as any other ship.

The view can be closed by naming the sunk ownship, but since that ship is gone, a new local view for that same ship cannot be opened. But if a new ship of the same name is created, a new local view for that ship cannot be opened until the old local view of the same name has been closed. In other words, the open and close commands for a local view are governed by the supplied names, not the identity of the ships. See the command syntax below and the samples.

Multiple local views are a feature: You can have a separate local view for every ship. You create one by opening a local view and providing the ship name. You get rid it of closing the one corresponding to a ship name. You can do this at any time, for any combination of ships. In this project, there is no need to have more than one map view, or more than one sailing data view, but they can also be opened and closed at any time.

New view commands.

Updating and commands. All of the attached Views get updated via Model notifications during command processing. The **show** command outputs all of the open views by calling their draw() functions. The user can **open** a view to cause the desired view to be created and attached to the Model, and then displayed by the **show** command, and then **close** the view to detach it from the Model and get delete it so that it no longer appears. If there are no views open, then **show** produces no output (not even an error message).

This gives the following new commands that the user can issue:

open_map_view - create and open the map view. The Project 4 view commands size, zoom, and pan control this view if it is open. Error: map view is already open. *Note*: This replaces Project 4's **open** command.

close_map_view - close and destroy the map view. Error: no map view is open. *Note*: This replaces Project 4's **close** command

Note: The other map view commands for default, scale, origin, size throw an Error if no map view is open; this check is done first, before reading and checking any parameters.

open_sailing_view - create and open the sailing data view. Error: sailing data view is already open.

close_sailing_view - close and destroy the sailing data view view. Error: no sailing data view is open.

open_local_view <name> - create and open a local view centered on the ship with name <name>. Errors in order of checks: no ship with that name; local view is already open for that name.

close_local_view <name> - close and destroy the view with that name. Error: local view is not open for that name. *Note*: the name is enough to identify the local view. It is not necessary to look up or verify the existence of a ship of that name to close the view.

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 Valdez, then the map view, then a local view for Ajax, they will be displayed by the show command in the order: Valdez, map, Ajax. If the Valdez view is closed, then of course it is no longer output. But if the user opens a local view for Valdez again, then the order of display will be map view, Ajax, Valdez. Likewise, the sailing data view will be output in the order in which it was opened relative to the other views.

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 three classes if it helps achieve a good design. Plan to refactor your Project 4 View class. You need a base class for the different kinds of Views, so this gives at least four classes - one base, and at least one class for each of the map, local, and sailing 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 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. A good way to tell whether you have a good design is to consider whether (1) you can add a new kind of Ship or other Sim_object that has the same kinds of data (e.g. location, speed) 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 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, 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.

Your solution to this step must conform to the following constraints:

1. A pure "push from Model" design should be followed.
2. The MVC pattern requires that Model must have only a container of View base class pointers, and that it absolutely must not know or care that there are different kinds of derived classes of View. The basic capability Model needs to support the new views is, as specified in Project 4, that more than one View can be attached to the Model, and Model broadcasts notification updates to all of the attached Views. Also part of the MVC pattern is that Controller is responsible for creating and destroying View objects in response to user commands, so only Controller knows what kinds of View are possible, and how they need to be set up, accessed, which Ships they are associated with, and in what order they are to be displayed. Drawing the Views is a Controller operation, not a Model operation. Note that the modifications to Model required to support our new sinking regime and to support Cruise_ship are not part of the MVC pattern. See the Project 4 document and lecture notes about MVC.
3. When a view is opened, a View object must be created with new; when closed, the object is destroyed.
4. You should use smart pointers with the Views so that all your memory management is automated - except for possibly in your Singleton implementation, no deletes should be present anywhere in your code.
5. We usually like to divide classes up in separate .h/.cpp file pairs to reflect the class structure and design, but I want to avoid imposing or suggesting a design by specifying the files involved. However, it is easiest to get a reliable autograder and spot-checking process if only specified files are involved. But if I make you put all of the views code into a single module, the result can be a bad design. So here is a compromise: You will submit your code for the views in four files named: `View.h, View.cpp, Views.h, Views.cpp` (note the singular and plural). 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.

Step 9. Add another kind of Ship.

Now that we have elaborated the program, it's time for another bit of design practice in the form of adding another kind of ship. Again, this represents a common development situation: we want to add another class which is similar to, but different from, an existing class. This is a **Torpedo_boat** class that is very similar in its behavior to Cruiser - it is a second kind of *warship*. It is created by using the same create command but with a ship type of Torpedo_boat. In summary: If commanded to attack, unlike Cruiser, the Torpedo_boat closes with its target by changing its course on every update. When it is close enough, it fires at the target and continues until the target is sunk, like Cruiser. However, if a Torpedo_boat is fired upon, instead of counter-attacking like Cruiser, it runs away to a island of refuge.

Why is this the (almost) last step? This is to demonstrate the value of OOP and the design concepts in the project architecture. It should be easy to add this new kind of ship, and it should automatically take advantage of the different views, the new sinking behavior, and so forth. In addition, this gives some additional practice at adding a new class to an existing hierarchy. You may need to do some refactoring of the existing classes, but not very much.

The initial values for a Torpedo_boat are fuel capacity = 800, maximum speed =12, fuel consumption factor = 5, firepower =3, maximum range = 5. It's resistance is 9, not because it armored, but because it is small and hard to hit.

Here are the specifics of the behavior, listed here by the relevant function:

attack, stop_attack - behaves the same way as Cruiser.

describe - behaves the same way as Cruiser except "Torpedo_boat" is output instead of "Cruiser".

update - behaves the same way as Cruiser, except that if the target is not in range, and we can move, we start to go to the target's current position at maximum speed.

receive_hit - like Cruiser's receive_hit behavior, except instead of counter-attacking, we make a decision based on whether we can move: If we can't move, we do nothing further in response to the hit (which means staying in attacking state if we were attacking). But if we can move, we announce that we are taking evasive action, stop attacking if we were attacking, and find an island of refuge and make that our destination at maximum speed. The island of refuge is the island closest to the attacker whose distance from the attacker is greater than or equal to 15 nm; if there is no such island, we choose the island that is furthest from the attacker. In case of a tie in distance, the first Island in alphabetical order should be chosen.

Design constraints

As in the case of Views, to avoid dictating the design, the specific .h, .cpp files for individual classes are not specified. Instead, all of your warship-type class code will be in a single file pair. Do the following:

1. Create a new file pair, called Warships.h, .cpp. (Note the plural!)
2. Move the contents of your Cruiser.h, .cpp files into Warships.h, .cpp.
3. Discard and *do not submit* your Cruiser.h, .cpp files.
4. Put your code for the new Torpedo_boat class and any related classes into Warships.h, .cpp.
5. As needed, modify the Cruiser class code in Warships.h, .cpp

Thus all of your Warship-related classes will be in this one file pair instead of a separate file pair for each class. As with Views, this is not the normal arrangement, but permits me to autograde your code without suggesting or dictating which classes are allowed.

Step 10. Remove the constructor/destructor messages.

You should keep these messages in your program until the very end, to make sure that everything is being destroyed when it should be. 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: Where possible, you should eliminate these messages by deleting the ***entire*** constructor or destructor function (this also demonstrates that you understand when you need to define these functions and when you don't). If the function must be defined for other reasons, then ***delete*** the message outputting code from it (don’t just comment it out).

Files to submit

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

- You must use the supplied **p5_main.cpp** instead of p4_main.cpp.
- **Cruise_ship.h, .cpp**.
- **View.h, .cpp; Views.h, .cpp** (notice the singular and the plural!)
- **Warships.h, .cpp** (notice the plural!). *Do not submit Cruiser.h, .cpp - they will not be included in the autograder builds.*

General Requirements

The Project 4 requirements still apply, except where the requirements of this project specifically supersede them. For example, the top-level error handling specifications for Project 4 apply to this project, and likewise the rules for the names of new Ships still apply. In short, your code must comply with the specifications of Project 4 except as necessary to meet the stated requirements of this project.

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.
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.

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 a 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 getter/setter functions for the member variables so that derived classes have full access to them, then something is probably wrong with the design - e.g. the base class isn't do much work.
- 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.
- Review the lecture notes on "Patterns for Reusing Base Class Functionality in Derived Classes" for specific techniques.

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, or override a little virtual function.

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 and refactor classes to the limited specified extent, component tests will not be performed. The autograder will test your program's output to see whether your Cruise_ship, Torpedo_boat, and different kinds of views behave according to the specifications.

In Project 6, the second part of the two-part project, 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:

Spot Check Warning: I will check that the required components and design concepts appear to be present in Project 5, and some easy-to-check aspects of code quality and good design are present; if not, your autograder score for Project 5 will be substantially reduced. 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 do a poor job on Project 5, you will be sorry when you do 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. 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 of Project 5 is the best that you can do. A specific suggestion based on the code quality evaluations done so far: The Coding Standards document should be taken very seriously. Go through it item-by-item and compare your code against it.