Project 4
Using Polymorphism in a Simulation (uh — a game?)
An Introduction to Object-Oriented Programming
Due: Friday, November 8, 2019, 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 the amount of detail required for this project, you should expect these specifications to be incomplete and unclear. Be sure to start the project early enough to have an opportunity to ask for a clarification or correction.

Introduction

Purpose

This project is the introduction to key object-oriented programming techniques. In it you will practice the following basic OOP techniques:

- Working with a set of classes that have clearly defined roles and responsibilities, such as a simple form of the Model-View-Controller pattern.
- Working with a class hierarchy in which inheritance helps re-use code.
- Using virtual functions to achieve polymorphic behavior.
- Observing the order of construction and destruction in a class hierarchy.
- Using abstract base classes to specify the interface to derived classes.
- Using a "fat interface" to control derived class objects polymorphically.
- Using separate containers of derived-class pointers to distinguish object types.
- Calling base class functions from derived class functions to re-use code within the class hierarchy.
- Using mixin multiple inheritance to take advantage of a pre-existing base class.
- Using private inheritance to demonstrate re-use of implementation rather than interface.
- Making header files as simple as possible to avoid unnecessary coupling.
- Using a simple form of Factory to further reduce coupling.

In addition, you will practice some additional programming techniques:

- Working with objects that have state and state changes.
- Using exceptions for user input errors to simplify the program structure, using `std::exception` as a base exception class.
- Further use of the Standard Library to simplify coding.

Finally, certain aspects of the coding may be new to you, although they are ancient and traditional issues:

- Using floating point numbers and dealing with some of the representational issues. For example, comparisons for equality are usually unreliable.
- Using a 3 dimensional array-type data structure.

Problem Domain

Even though it resembles some popular computer games, this is actually a simple simulation of the socio-political system of the medieval world! Actually, many games are simulations, and it turns out that many simulations can be programmed very well using object-oriented programming concepts and techniques, so a game is a good place to learn OOP. Also, an important part of OOP is organizing the software to match the domain, which means you have to understand the domain. Rather than use some "real" simulation domain, which would take a lot of time to understand, we'll indulge in a familiar game-like domain, allowing us to focus on the OOP techniques for representing the domain. If you haven't played games such as Warcraft yourself, please ask me or your friends for explanations of how they work.

This program is very much like the common type of game in which simulated persons or vehicles located in a two-dimension world move around and behave in various ways. The user enters commands to tell the objects what to do, and they behave in simulated time. Simulated time advances one "tick" or unit at a time; Time is "frozen" while the user enters commands, and then the user commands the program to **go** to advance one tick of time. If this were a game, it would be called a "turn-based" game: the player enters commands to the objects in the game, and then issues a turn-complete command. For simplicity in this version of the project, there is only a single player.

The two-dimensional world in this simulation consists of a limitless and featureless plain that is represented with the real plane. The program can display a grid that represents a map-like view of the two-dimensional world, which can be moved around and zoomed or resized. This is provided by a class, View, that encapsulates all the calculations for the display. Later projects will have additional kinds of Views, so this one lays the architectural foundations by following a pattern, called *Model-View-Controlle*r, which will make it easy to add additional kinds of Views without ripping up existing code (*Add features by adding code, not by changing code*).

There are two kinds of objects, Structures and Agents. Structures are stationary buildings, like a Farm and a Town_Hall, that may change state on their own, but can't be commanded to do anything. In this project they produce or store food. Agents are characters that move around and can be commanded to take actions on Structures and each other. There are two kinds of Agents: Peasants are productive hard-working types who haul food from Farms to Town_Halls (or between two Town_Halls), and Soldiers are good for nothing but violence — they can attack another Agent and reduce its health enough to kill it, whereupon the victim gets removed from the world. The user (you) can command Agents to move around, work, or attack. You can display the status of the objects, or view the map, to help you choose what commands to issue.

Overview

Later projects will have fancier possibilities. This project essentially is building a framework in which much more complicated simulations can be constructed easily in future projects. By using the fundamental OOP design concept, a polymorphic class hierarchy, this project can be easily expanded to handle additional kinds of agents and behaviors, again without ripping up existing code (*Add features by adding code, not by changing code*).

This is a fairly complex project because there are many classes involved, and a relatively large number of separate header and implementation files. However, the code in each class is actually fairly simple. The movement process is also somewhat complex, but the use of a pre-existing library of classes and functions eliminates the need to work out the math and its coding for yourself. The basic design of the program is specified, since learning how to construct such programs works best from examples. Later projects will be more open in design.

Overview of this Document

The Program Overview section introduces the overall architecture of the project in the form of a UML class diagram showing the major classes and their relations. Some key features of the class design are summarized. The next subsection presents some important issues involved in using floating-point numbers, which will be used in this project to represent the locations and movement of objects in the plane. A final overview section briefly discusses how state machines will be used to represent the behavior of the Agents.

The Program Specification section first presents the Classes and Components in terms of their responsibilities and interactions with each other. Then a Detailed Specifications section presents the exact details of what the key functions in each class should do. This detail is necessary in order for the objects to interact with each other properly, but the descriptions are probably longer than the actual code! In other words, a small amount of code is required in each class's functions, but it is necessary to be very exact on what this code will do, which takes a lot of prose. UML sequence diagrams are used to help explain some of the critical interactions.

The final sections describe the project evaluation and provide some suggestions.

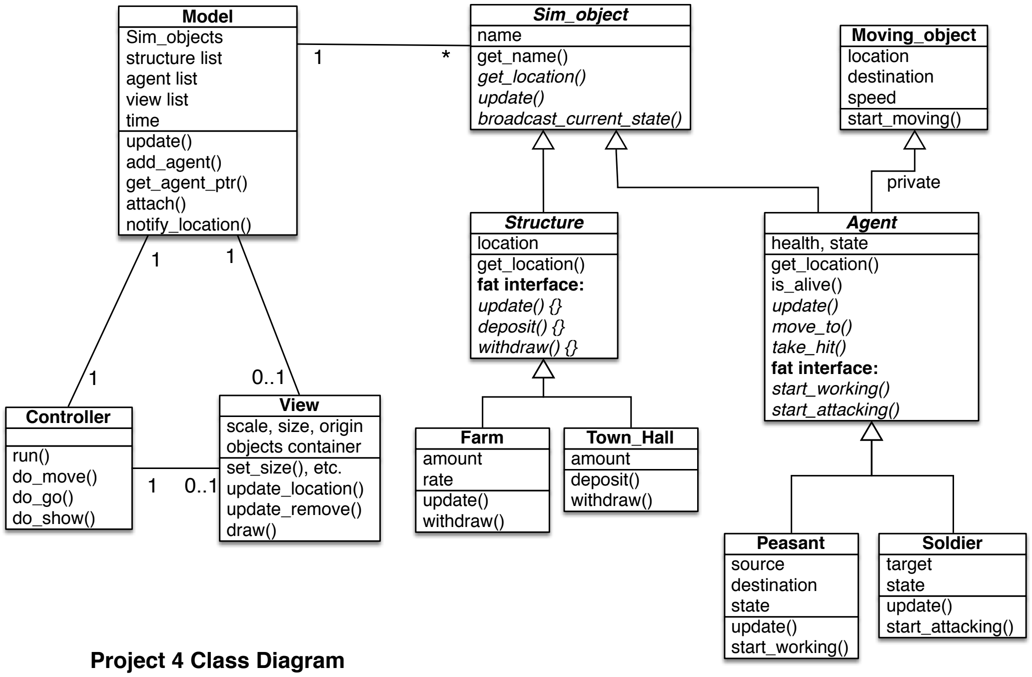
Program Overview

Basic Class Architecture

The architecture of the project is summarized in the UML Class Diagram shown below. The Model-View-Controller (MVC) design pattern shows up here, but in a simple form. It helps organize the top level of the program into a structure where the responsibilities are divided up between three objects: The Controller object interacts with the human user, handling all the input, and controls the rest of the program. The Model object keeps track of the simulated "world" of agents and structures; it is the keeper of which agents and structures currently exist, and instructs them to update themselves or describe themselves, and passes on updated state information to the View as the objects require. The View maintains the information sent to it, and presents a map-like view of the world. Upon command from the user, the Controller creates or destroys the View - it is not always present.

Basic inheritance structure

The class Sim_object is an abstract base class that describes what all of the simulated objects in the world have in common. Each has a name and an accessor for location, and you can tell it to describe itself or update itself. These describe and update functions are pure virtual, meaning that at some point all derived classes will have to supply or inherit a definition of these functions. There are two derived classes which are also abstract: Structures and Agents. Structures receive, provide, and store a commodity represented by a number; this is an amount of foodstuffs in this project. By default, updating a Structure does nothing. However, Farms produce food at a certain rate; the amount of food available on a Farm thus increases on every update.



Project 4 Class Diagram

Showing representative members of each class.
Other member functions and variables are not shown.

Agents are more complex because they have some complicated movement behavior. The mathematical guts of the movement are provided by re-using an implementation provided by a class called Moving_object that will be provided to you; since this is inheriting only for implementation internal to a class, this provides an opportunity to use private and multiple inheritance. Once an Agent is commanded to move, on each update it will update its location and stop once it has arrived at its destination. Derived classes of Agents simply inherit this behavior from Agent, but can modify it to give more interesting patterns. For example, Peasants can move foodstuffs between two Structures, and once commanded, will shuttle back and forth independently. They do this by commanding themselves to move to the other place by calling the same functions that the user command code calls.

Polymorphic behavior through virtual function magic

On each turn (commanded with the **go** command), the Controller will tell Model to tell each Sim_object to update itself. The different types of objects behave differently when told to update themselves. Farms add their new production to the amount of food they have. Town_Halls don't do anything. Both Peasant and Soldiers update their location if they are moving. If working, Peasants decide what to do next depending on whether they are at their source or destination location, or are in-between. Soldiers decide whether to continue or terminate an attack, or respond to being attacked.

Distinguishing between object types

However, it is not possible to treat all objects identically all the time. This project uses two techniques that allow controlling different kinds of Sim_objects differently depending on what type of object they are, described as follows:

The first technique, ***separate containers,*** enables Controller to tell whether a name in a user command refers to an Agent or a Structure. Only Agents can be commanded to do things, and in this version at least, only Agents can be attacked. In addition, only Structures can be designated as food sources and destinations. The program will tell Agents and Structures apart by having separate containers of pointers for Agents and Structures. When an Agent or Structure is created, its pointer is placed in a container of Sim_object pointers for use in commands that apply to all objects, and also in an Agent pointer or Structure pointer container. Thus, if the user commands an Agent by name, the program can look in the Agent* container for the Agent of that name. If the command names what is supposed to be a Structure, the program can look for a match in the Structure* container to verify that there really is a Structure of that name. This way, there is never a case of a Agent being confused for a Structure, even if they were to have the same name. But in this project, no duplicate names for Sim_objects are allowed anyway, because the result would be an extremely confusing View display.

The second technique is the ***fat interface.*** Different kinds of Agents have different possible behaviors: Peasants can work, but have no violent tendencies, while Soldiers can attack but are too lazy to work. How do we command an Agent to do these things unless we know what exact kind of Agent it is? The answer is that the Agent class declares virtual functions for the *union* of the functions we want to call on the derived classes — this is the fat interface. Thus Agent declares virtual functions for both working and attacking, event though the Agent class itself does neither of behaviors. This makes it possible to command an Agent to either work or attack with code that does not need to know whether a Peasant or Soldier is being commanded. That is, Agent has an interface for both commands; Peasant overrides the work command function, while Soldier overrides the attack command function. Agent defines "default" functions that simply throw an Error exception containing the message "I can't work!" or "I can't attack!" Likewise, Structure has a fat interface for the functions used to withdraw food or deposit food. The "default" version of the deposit function does nothing (anything deposited essentially disappears — due to theft, apparently) and the "default" withdraw function returns 0 for any request. Farms override the withdraw function to supply food, while Town_Hall overrides both so that food can be both deposited, and moved elsewhere, such as to another Town_Hall.

Avoiding switch-on-type

The combination of virtual functions, fat interfaces, and the separate lists of object types means that the entire program can operate in a completely type-safe manner without having to commit the sin of switch-on-type logic. Nowhere in the program is there a hunk of code that asks what type the object is and then branches to the function call that is appropriate for the type. Either the virtual function mechanism routes us to the right code automatically, or the object has the right type "by definition" or "by construction."

Decoupling

A key feature of object-oriented design shows up here: the Model, View, and Controller classes are very insulated from the details of the agents and the structures. Basically, the Model and Controller do not know about the different kinds of agents; they only know about the base class Agent, not about Peasants or Soldiers, and similarly for Structure. The View receives only name strings and locations from Model, and plots them; it knows nothing at all about the objects whose names and locations it displays.

New agents and structures are created by two ***Factories*** (not shown in the diagram because they are simple functions), which allow the Controller to create a new Agent on command without having to incorporate any knowledge of the specific agent types. Likewise, new Structures can be created by Controller without any knowledge of the specific structure types. As an example of this decoupling, the Peasant.h header file needs to be #included in *only two* places: Peasant.cpp and Agent_factory.cpp, and not at all in Model, View, or Controller.

Because of this organization, in the future new kinds of Agents or Structures can be added with little or no change to the Model, View, or Controller. For example, if we were to add a "Knight" class that responded to the same commands, all we would have to do would be to write the Knight.h and Knight.cpp files defining the new class, and then modify Agent_factory.cpp to be aware of the new type name and construct Knights on request. This "plug and play" aspect of adding new classes to a polymorphic class hierarchy is a major reason why object-oriented programming is such a powerful approach!

A key requirement of maintaining this flexibility is *minimal header file discipline*: Each .h or .cpp file for a component should only include the headers that are truly essential; this allows changes to the other code without requiring rewrites or recompilation of the component, and helps clarify and enforce the design decisions concerning the responsibilities and collaborations of the classes. If two classes do not collaborate directly, then neither should #include the other!

Floating-point Issues

Computer programmers often get into trouble with floating-point numbers, because unlike integers, floating-point is not an exact representation, but only an approximate one. Throughout this project, double-precision is used, following the normal practice when doing any significant computation, such as trigonometry. The key consequence of the approximate nature of floating-point is that two floating-point values that are supposed to be mathematically equal will almost never be equal at the bit-by-bit level that the == operator tests for. About the only time one can be sure they will be exactly equal is if the specific computations dictates that all the values involved in a computation will always be integral (like exactly 5) and small enough to be within the precision of the representation.

The approximate values of floating-point numbers can be especially problematic if they are converted to quantized values in some way, such as a discrete state (e.g. "alive" versus "dead"), or an integer value (e.g. for plotting on a discrete grid). A difference in the zillionth bit to the right of the "binary point" in the representation can "flip" the quantized result from one value or another, and this zillionth bit can depend on the microscopic details of a computation in combination with the CPU architecture and the compiler code generation policy. The overall effect is that in borderline cases, the result of a quantization can appear almost random to a user.

That said, how will I autograde the output of your program? First, some of the key computational code will be provided, and you should use it as-is (as in the Moving_object class). This will ensure that the instructor's solution and your solution will perform these computations in exactly the same way on the autograder machine. Second, the behavior will be spelled out in detail, so that state changes should happen in exactly the same way. Third, all of the floating-point output written to cout will be rounded to two decimal places. This is done by a couple of lines of code in the p4_main.cpp file that is provided for you to use as-is. The code for View will have to change these settings, and will be responsible for restoring them (see the handout on Output formatting). While rounding is a quantization operation, because of the great many bits carried by double precision, rounding to two decimal places suppresses almost all of the possible glitches. There may be a few surprises, such as a value of "-0.00", which can result when a small negative number is rounded off, but the autograder knows that this should be considered the same as "0.00". Fourth, certain comparisons will be done with a range, rather than testing for equality, and some values will be assigned to suppress any lingering differences. These will be specified where needed. Fifth, I will attempt to use test cases where borderline situations do not appear.

Finally, I will be careful to check that failures to match on numerical grounds will not be penalized if the code was written in the specified way, and will re-evaluate test cases if they appear to hinge on borderline cases where quantization will cause distortions. Past experience suggests that these problems will be rare. They are worth the risk because using floating point makes it possible to do much more interesting and realistic things in the projects! Later projects will involve less emphasis on exact output matching, and more on code design, so the risk of grading problems will be less.

State Machines for Behavior

The Agents in this project behave in complicated ways. A good way to represent complicated behavior is with a state machine. Each object is in an initial state. When commanded to do something, it changes into another state corresponding to the command. On each update, the current state is tested, and the appropriate actions done depending on what the current state is. One of the actions might be to change the state. At most one state change will happen on an update. When the object is next updated, the new current state will control its behavior. Another command, such as **stop**, might change the state as well, whereupon the next update will deal with that new state. An excellent way to code a state machine simply and clearly in C or C++ is to use an enum class variable to represent the state, and then simply switch on that variable in the update function. The default case in the switch can be used to recognize an unknown state due to programming errors. For purposes of explanation, some diagrams will be shown below, but the details of the state transitions are too much to include in the diagrams, and so will be described in text.

In this project, the objects can have multiple states, one for each class they are composed of. The Agent class has a movement state encapsulated in Moving_object, but made available by Agent. They also have a second set of states that represent whether the Agent is Alive, Dying, Dead, or Disappearing. If an Agent is killed or dies, it first becomes Dying, and then when next updated, becomes Dead, and when updated again becomes Disappearing and stays that way.

If an Agent is Disappearing, Model will remove and delete the Agent at the end of the current turn. The reason for these four states is that all of the objects are referred to by pointers, and following a "dangling" pointer to a deleted object produces undefined results. The lingering implemented with these four states give plenty of time for an Agent, upon being updated, to realize that another Agent is no longer Alive, and to cease interacting with it or referring to it before it is deleted. In the next project, smart pointers will be used to simplify this process, but in this project, you have to deal with using built-in "raw" pointers only.

In addition to the Agent state, Peasants and Soldiers have a third set of states and state changes, concerned with working and attacking respectively. The rule is that to update a Peasant or Soldier, we first update their Agent part, and then update their Peasant or Soldier part, which can make use of information about the Agent state — such as whether a Peasant is still alive or is moving.

Program Specifications

Class Responsibilities, Collaborations, and Dependencies

Take special note of the responsibilities and collaborations described for each class and component. These are critical to the design — in a good OO design, each class has well-defined and limited responsibilities and interactions with other classes that make sense in the application domain. Understanding the responsibilities will help you know what the code is supposed to do, and where different information is created, stored, and used. The description of behavior and computations in this section is a general overview; details necessary to actually write the code are in the next section.

The course site contains files for the components listed below, either complete files, or "skeleton" files where you must supply the missing content. Any complete files must be used as-is. You must complete the skeleton files to get the final version of the file. The comments in these files provide additional specification information beyond that in this document.

In all cases, you are not allowed to change the public or protected interface for any class, except where the skeleton header files have comments containing instructions (starting with "***") that ask you to "complete" or modify declarations. In particular, the comments ask you to complete the declarations for virtual functions to give some practice in noticing what needs to be declared virtual and where. Otherwise, you may not add functions to the public or protected interfaces, nor can you remove any that are specified, nor can you change their signatures (names and parameter types), return types, or behaviors.

Except where specified, the private members are for you to choose. All member variables of all classes defined with "class" must be declared private. The only exception is classes defined with "struct" which can be used only where customary (e.g. for simple function objects) or as specified. When a particular function definition is supplied, you must use it in your code in the specified way.

Important: Students new to inheritance often fail to take advantage of it. In this project, derived classes have many opportunities to use functions in a base class to get their work done. If you finding yourself writing code in a derived class that is similar to code you wrote for a base class, *you are doing something wrong*. Stop! Sort it out — ask for clarification or help!

Geometry, Moving_object. These components are supplied on the course file server, and should be used as is, without any modification. Study the header files and their comments to see what it available.

Geometry is responsible for representing points and vectors in the real plane, and providing classes and operators for computing with them. Any computation involving points, distances, etc., can and should be done with the functions in the Geometry module.

Moving_object represents an object that can move in the real plane. It is responsible for maintaining information about the current location of an object and its speed. When told to start moving to a new location, it computes the new position of the object every time its update function is called. The speed is the distance traveled per update. These computations are very simple thanks to Geometry.

Sim_object. This class is an abstract base class that describes the interface, members, and capabilities of all of the simulated objects in our simulated world. Each derived class is required to override the get_location, describe, update, and broadcast_current_state functions to provide their own specific behavior. For this reason, these functions must be declared pure virtual. Because it is basically an *interface class*, its only direct responsibility is maintaining the name of the object, which is a std::string.

Structure. A Structure is a kind of Sim_object, so it inherits from Sim_object. It supplies its location, which can't be modified. It defines an update function that does nothing. It also defines a couple of virtual functions, withdraw and deposit. In this class, withdraw always returns 0.0, while deposit simply does nothing. The "normal" use of these functions can be seen in Farm and Town_Hall, which override these. Structure's describe function outputs the name and location of this object, but no more. Derived classes output information before and after calling this base class function to output the complete description with a minimum of code. This class is made abstract by declaring its destructor to be pure virtual.

Farm. A Farm is a kind of Structure. It has a double member variable, which represents the amount of food present at the Farm. On every update, a certain additional quantity is added to the total, representing how the Farm produces more food over time. It outputs the new total on the update. A Farm's withdraw function allows a Peasant to take some food from the Farm. The Peasant asks for as much as it can carry, and the Farm returns that amount or the amount available, whichever is less, and decrements that amount from its total available. The amount available is not allowed to become negative. When asked to describe itself, it outputs "Farm ", then calls Structure's describe function to output the name and location, and then outputs the quantity of food currently available. Farm does not have a deposit function - it does not accept any food brought to it; the food simply disappears.

Town_Hall. A Town_Hall is also a kind of Structure, and also has a double member variable that represents the amount of food in storage at the Town_Hall. Town_Hall does not have an update function, so nothing happens if a Town_Hall is updated. Town_Hall's deposit function adds the supplied amount to the amount in storage at the Town_Hall. The withdraw function is different from Farm's in that it retains a "tax" on the food it supplies, and won't supply any unless it is at least 1 unit in quantity. Finally, the describe function follows the same pattern as Farm.

Agent. An Agent is a Sim_object. It is made an abstract class by declaring its only constructor to be protected, meaning that client code is not allowed to create an object whose class is simply Agent; only derived class objects can be created by client code. Agent is the most complex class in the program. Some of Agent's capability comes by inheritance from Moving_object. Since Moving_object is used only to implement Agent, Agent privately inherits from Moving_object. That is, none of Moving_object's public interface is relevant to the rest of the program, so it becomes private to Agent. Here is what Agent does:

- Agent has a current location maintained in Moving_object. If requested, Agent can obtain it from Moving_object and supply it to a client. Other aspects of its motion can be supplied similarly.
- When commanded to move to a destination, Agent passes the request to its Moving_object part. When Agent's update function is called, it calls Moving_object's updating function.
- Agent has a health, which is an int value that is initialized to a specified value. If the Agent receives a hit from an attacker, the specified number of "hit points" is subtracted from this Agent's health. If the health is no longer positive, the Agent "dies".
- Agent maintains its state, and supplies it upon request through a public interface. Its state includes whether it is stopped, moving, alive, dead, or is in the process of disappearing from the world.
- Agent describes itself in terms of its name, location, and state. Derived classes will ask the Agent to describe itself as part of their description.

Agent provides a ***fat interface*** for commands to derived classes. This is a set of virtual functions for commands accepted and acted upon only by some of the derived classes. These functions are start_working and start_attacking. The functions are overridden by derived classes as appropriate for the class. In the Agent class, these functions have a "default" implementation that throws an Error exception. Thus if a particular type of Agent is given a command that it cannot execute because it is the wrong type, the default exception-throwing function will execute. Example: A Peasant is told to attack; an Error exception is thrown because Peasant does not define an override for the start_attacking virtual function.

Peasant. Peasants are Agents, and can move around by virtue of their being Agents. However, they can also be told to start working, moving food between two Structures. This means they override the start_working function in Agent. Working requires shuttling back and forth between the two Structures, one of which is normally a food source, the other is a food destination. Note that unlike real peasants, our Peasants are pretty stupid in that they do not know whether their source and destination are Farms or Town_Halls - they are both just Structures. Thus, if necessary, a Peasant will wait at a food source that is empty, until it has some food, either as a result of its producing more, or another Peasant depositing some there. In fact, Peasants will not hesitate to deposit food at a Structure that will not accept a deposit - like a Farm. Such deposits just disappear due to invisible mice or thieves, or possibly both. One other role of Peasants is to be a target for the violent behavior of Soldiers, but if they are moving, they can be surprisingly tough to kill.

Soldiers. Soldiers, another kind of Agent, are the villains of our medieval world. Soldiers are good for nothing except violent behavior. They can be told to move around, like all Agents. A Soldier can be told to attack another Agent (not themselves), and if the other Agent is in range, or alive, the Soldier will start attacking (delivering "hits") with its clanging sword. It will single-mindedly keep this up until it kills its target, gets killed itself, or its target becomes out of range (it can carry on an attack while moving, however). Soldiers do have a rudimentary sense of self-protection. If a Soldier is attacked (through its version of take_hit) it will begin to attack its attacker.

Model. Model has the responsibility for the current time and maintaining the Sim_objects in the simulated world. Whenever a Sim_object is created, it is added to Model's containers. Model provides some services to Controller, such as lookup services to return the pointer corresponding to the name of an Agent. Other services are: On request, Model will update the Sim_objects and the current time, or tell them to describe themselves. Additionally, as described below, Model distributes updates from Sim_objects to the View whenever objects are added, updated, or removed.

Model is thus the "keeper" of the simulated world and collaborates with Controller and View. Model's constructor creates the initial simulated world, and its destructor is responsible for deleting all of the Sim_objects currently in existence. View displays the current state of the world, and Controller allows the user to control the Model and View. A View is "attached" to the Model with the attach() function, and thereafter, when the Sim_objects change (e.g. a new Agent is added) or get updated, they ask Model to inform the View of the current state of the world. Details are below. There is also a detach() function to discard the View pointer, so that when the View is closed and deleted by the user, Model will no longer be trying to inform the View.

View. View has a single responsibility, which is to produce the map-like character-graphics display that shows the locations of the Sim_objects in the world. View has a public interface, called by Controller, to change the parameters of the display, such as the map scale. Each object is represented in the map by the first two letters of its name located in a grid corresponding to the object position. On request by Sim_objects, Model supplies information about the objects to View by calling View’s update_location function, but Model supplies only the names and locations of the objects; View has no knowledge of the kinds of objects it is plotting. View "remembers" the names and locations that it has been told, so that it can generate a display with different plotting parameters at any time. In addition, when an Agent "dies", it will inform Model which calls View's update_remove function, which means that a particular name is no longer to be plotted — this is how a dead Agent will be removed from the display. Specific details are below. In this project, there is only one View, created by Controller when the user enters an open command, and deleted when the user enters a close command, or when Controller terminates or is destroyed. The View can be created or destroyed using these commands whenever the user wants.

View depends only on the Geometry module for its computations.

Controller. Controller has the responsibility of acting as the intermediary between the human user and the Model; as its name suggests, Controller basically "runs the show." Controller is responsible for collecting and checking the input from the user, and then calling the appropriate functions in Model to bring about the user's commands. In response to user commands, Controller also creates the View object and attaches it to Model, and calls the View parameter-setting and drawing functions, and detaches the View and destroys it. Controller collects information about desired new objects from the user, and calls the factories to create the new Structure or Agent, and then adds them to the Model.

Agent_factory, Structure_factory. These are the simplest version of the *factory* concept — a function that when supplied with strings for the object’s name, its type, and its Point location, creates the corresponding object and returns a base-class pointer to it. Thus the factory's .cpp file will need to #include the specific agent or structure headers, but the .h file for the factory does not need to #include any Sim_object headers at all. Thus the factory decouples its Controller client from the specific subclasses of Agent and Structure.

p4_main.cpp. This file is supplied to you as a "starter", and must be submitted by you, and must be used as-is, without any modifications. Consistent with the structure of many OO programs, the main module does very little. It first creates a Model object and then creates a Controller object, and then tells the Controller to start running. All the "action" is thus in the interacting objects.

Detailed Specifications

Additional details appear in the supplied files. The following is a description of what the key functions do in each class, with attention paid especially to how state changes are done. You should examine the supplied starter files for each class as you read these descriptions. *Note*: The output messages are described here to help explain the behavior. The exact strings to use are supplied in the strings.txt starter file; in case of inconsistency, the strings.txt file version should be used.

Constructor and Destructor Messages

It is important to understand the order in which construction and destruction is done in a class hierarchy. For this reason, each of the classes involved in the Sim_object family must print a message in its constructor outputting the name of the object, and another message in its destructor doing the same. Moving_object does not output these messages. In addition, Model,View, and Controller also output constructor and destructor messages so you can see when these objects get created and destroyed in the code. Your code should output these messages as the *last thing* done in the constructor and destructor functions. See the string.txt file for the exact messages, and the samples to see the sequence of messages that you should get automatically. The output code must appear only in the .cpp files — <iostream> should not be included in any of these header files.

Structure behavior

The Structure base class provides almost nothing but the interface to the derived classes. The only member variable is this Structure's location which is returned by an accessor function that overrides Sim_object::get_location.

update. The implementation of this function in Structure is empty; it does nothing.

describe. A Structure outputs its name followed by " at " followed by its location.

withdraw. This function simply returns 0.0.

deposit. This function does nothing — the implementation is empty.

Farm behavior

The Farm behavior is so simple, you would hardly call it behavior. There are no explicit states. A Farm has an amount of food on hand, initially 50. It also has a production rate, the amount of food produced on each update; this has a value of 2.

update. The Farm adds the production rate amount to the amount, and outputs a message about the current amount.

describe. The Farm outputs "Farm " followed by a call to Structure's describe, then outputs the current amount.

withdraw. If the request is greater than the amount on hand, return the amount. Otherwise return the amount of the request. The returned value is subtracted from the amount on hand. No messages are output.

Town_Hall behavior

Town_Hall is even simpler than Farm: it has no explicit states, and does not define an update function. It passively holds an amount of food, initially 0, which is changed only by the deposit and withdraw functions.

describe. The Town_Hall outputs "Town_Hall " followed by a call to Structure's describe, then outputs the amount available.

withdraw. Deduct 10% of the amount on hand to get the amount available (a "tax" by the Town_Hall). If the amount available is less than 1.0, set it to zero (the local baron doesn't want to bother with supplying tiny amounts of food). If the request is greater than the amount available, return the amount available (which could be zero). Otherwise return the amount of the request. The returned value is subtracted from the amount on hand. No messages are output.

deposit. The supplied amount is merely added to the amount of food being stored. Nothing is output.

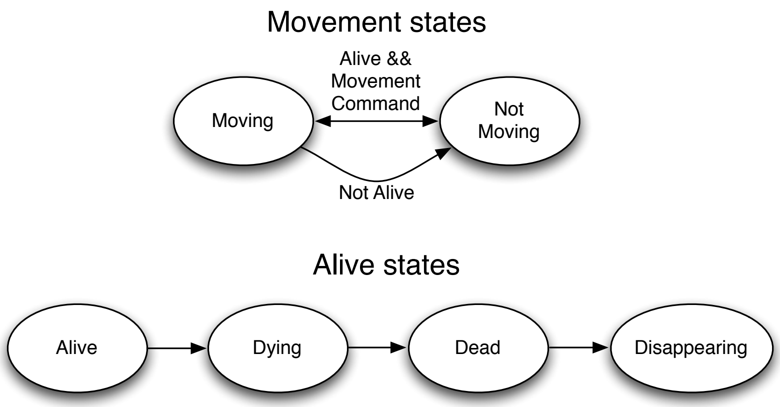
Agent behavior

Agents have two sets of states. The diagram below shows the basic states and their relationships, but the details of the states and their transitions are described in the text below. An Agent's initial state is **Alive** and not moving (as specified by Moving_object). Its speed is 5, and its initial health is 5. Its initial location is supplied in its constructor.

move_to. Call Moving_object's start_moving with the destination. If we are now moving, output "I'm on the way"; otherwise output "I'm already there."

stop. If currently moving (as specified by Moving_object::is_currently_moving()), call Moving_object::stop_moving, output "I'm stopped." If not moving, output nothing at all.

update. The actual calculations for the Agent movement are handled by Moving_object, which is provided. The movement state maintained by Moving_object is either **Moving** or **Not_moving**. There are four states of an Agent that are separate from its movement state: **Alive**, **Dying**, **Dead**, and **Disappearing**. The initial states of the Agent are **Alive** and **Not_moving** (which is the initial state of Moving_object). To update the Agent's state, do the following in this order when Agent::update is called. Each case description entails that we do nothing further in that case.



Agent state transition diagram

1. If this Agent is **Alive**, and is moving, update its location with update_location, then:

    If update_location returns true, the Agent has arrived; output "I'm there!" and ask Model to notify the Views that this object's location has changed.

    If false is returned, the Agent is still moving; output "step..." and ask Model to notify the Views that this object's location has changed.
2. If the Agent is **Dying**, change its state to **Dead**, and ask Model to notify the Views that this object is gone; this means that it should no longer appear in the Views.
3. If the Agent is **Dead**, change its state to **Disappearing**.
4. If the Agent is **Disappearing**, remain in that state.

When Moving_object detects that it has arrived at the destination location, it sets the current location equal to the destination location. This means that if an Agent is told to move to a Structure, you can tell whether it actually arrived at the Structure by determining if is now **Not_moving** and its current location == the location of the Structure.

take_hit. This function simply calls a protected member function, lose_health, which performs the following: Subtract the supplied attack strength from the the current health. If the result is less than or equal to zero, set the state to **Dying**, stop moving, and output "Arrggh!" Otherwise, output "Ouch!" This computation is in a protected member function so that derived classes can call it.

describe. Do the following if Agent::describe() is called:

1. Output the Agent's name and location.

    (Note: A subclass will first output its class name (e.g. "Peasant") and then invoke this function)
2. If Agent is Alive, output the health; if moving, the movement information; if not moving, that it is stopped.
3. If the Agent is **Dying**, **Dead** or **Disappearing**, output the corresponding message.

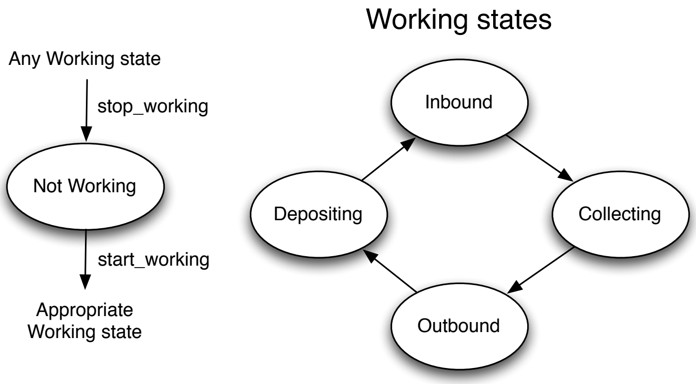
fat interface functions. Throw an Error exception containing the Agent's name plus the message that it can't work or attack.

Peasant behavior

A Peasant has a somewhat complicated state space involved with working. The basic states are shown in the diagram below, but the specifics are described in the following text.

Peasant has a double member variable for the amount of food it is carrying (initially zero), and two member variables that hold a pointer to two Structures: the food source, and the food destination. When working, it moves food from the source to the destination, calling the withdraw function at the source, and the deposit function at the destination. It can carry a specified maximum amount of food. At the source it asks for enough to fill up to its maximum (35) and waits there if nothing is supplied -- it asks again next update. The Peasant states are **Not_working** (the initial state), **Inbound** moving to the source, **Collecting** at the source, **Outbound** moving to the destination, **Depositing** at the destination.

move_to. Because a Peasant can move either as commanded or as part of working, we have to be able to distinguish the two cases. Overriding Agent's move_to, if **Working**, output a message about stopping work, set the Peasant state to **Not_working** (also forgetting source and destination by setting pointers to nullptr) and then call Agent's move_to to conduct the actual move_to activity.



Peasant state transition diagram

start_working. First stop moving with Agent::stop() and then set the state to **Not_working** and reset the source and destination pointers. Then check that the new source and destination are different (*important* — compare their identities, given by their addresses in memory, not their locations[1]), and throw an Error that a Peasant can't move food to and from the same place. Then remember the supplied source and destination pointers. Check and perform the following in this order:

If the amount being carried == 0.0, and

our location is the same as the source, set the state to **Collecting** and do nothing further.

if our location is not the same as the source,

command ourselves (with Agent::move_to) to go to the source and

set the state to **Inbound** and do nothing further.

If the amount being carried is non-zero, and

our location is the same as the destination, set the state to **Depositing** and do nothing further.

if our location is not the same as the destination,

command ourselves (with Agent::move_to) to go to the destination and

set the state to **Outbound** and do nothing further.

Note that any previous state or source and destination information is overwritten and has no effect.

stop. The Peasant stops moving and working. First call Agent's stop(), then if **Working** output a message about stopping work, set the Peasant state to **Not_working**, and forget the source and destinations (set the pointers to nullptr).

update. First update the Agent's state. This will determine whether we are still moving (either as commanded, to the source, or to the destination) or whether we are still alive. Then do the following in this order with regard to the Peasant state:

1. If not **Alive**, or **Not_working**, do nothing further.
2. If **Inbound**, and no longer moving (according to Agent::is_moving()), and we have arrived at the source location, set the state to **Collecting**, and do nothing further.
3. if **Collecting**, compute the request as the difference between the maximum amount we can carry and the amount we currently have. Call the withdraw function with this request; a received amount will be returned by the source; add it to the amount we are carrying. If the amount received is positive (greater than zero), output "Collected" with the amount received, set the state to **Outbound**, and command ourselves to start moving to the food destination by calling Agent::move_to. If the amount received is not greater than zero, output the "waiting" message and stay in the same state. Then do nothing further on this update.
4. If **Outbound** and no longer moving, ,and we have arrived at the destination location, set the state to **Depositing**, and do nothing further.
5. If **Depositing**, call the destination's deposit function with the amount we are carrying, output the "Deposited" message, set our amount to zero, command ourselves to move back to the source, and set our state to **Inbound**.

describe. Output "Peasant" followed by the Agent description, followed by the amount being carried. Then depending on the state, output the appropriate information for **Outbound**, **Inbound**, **Collecting**, and **Depositing**.

Note: Peasant does not override take_hit. When a Peasant becomes not **Alive**, its update function will do nothing thereafter (see update description). However, its working state will remain as whatever it was last doing (e.g. Collecting or Outbound).

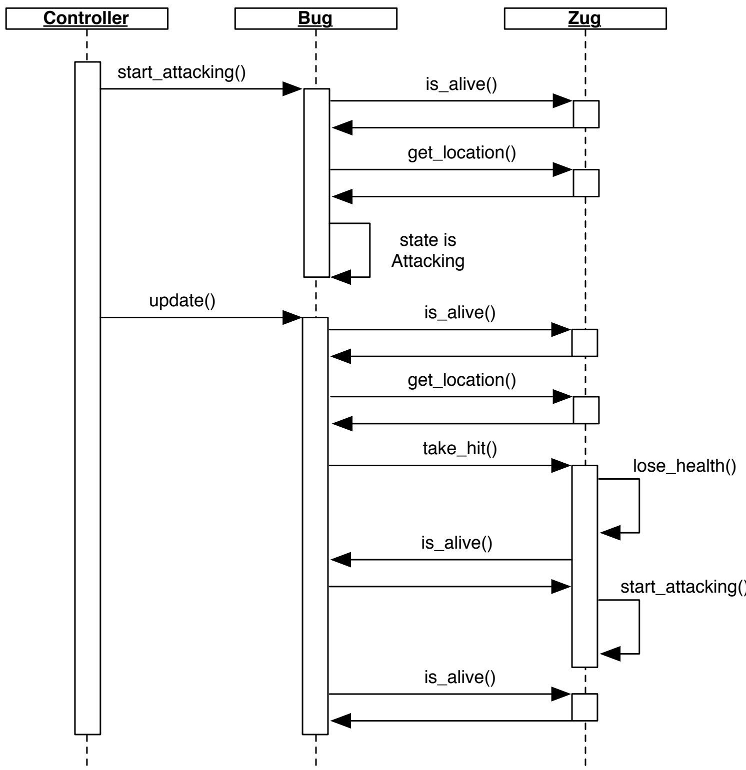
---

[1] For simplicity, we allow two objects be in the same location; let's pretend that medieval metaphysics allowed this (it actually didn't).

Soldier behavior

Since violence is such a simple-minded concept, Soldiers are simple compared to their more enterprising and sophisticated Peasant siblings. It has member variables for its attack strength (an int), its attack range (a double), and a pointer to its current target. A Soldier has an attack strength of 2, and attack range of 2. Its initial state is **Not Attacking**, with no target. Soldiers do not modify the effects of moving or stopping. Hence, if told to attack they will continue to do it until they either kill their target, get killed, or they or their target moves out of range; there is no way to command them to stop attacking directly. The UML sequence diagram below illustrates the sequence of interactions when one Soldier to told to attack another. Compare the diagram with the following specifications:

Project 4 Sequence Diagram
Showing Attack Interaction



The sequence of interactions when the Soldier object Bug is commanded to attack another Soldier object, Zug. Both Soldier objects are currently alive and stay alive during the illustrated sequence. On its next update, Zug will issue a take_hit() on Bug.

start_attacking. A pointer to the target Agent is supplied. Check first that the target is different from "this" Agent and if not throw an Error that this Soldier can't attack itself. Then check if the Target is **Alive**. If it is not in the **Alive** state, throw an Error that the target is not alive. Then, if the distance between this object and the target is greater than the range, throw an Error that the target is out of range. If all is OK, save the pointer to the target, output "I'm attacking", and set the state to **Attacking**.

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

1. If this Soldier is not **Alive**, do nothing further.
2. If **Not Attacking**, do nothing further.
3. If **Attacking**, do the following in this order with the Soldier state:
    1. Check to see if the target object is not **Alive** (maybe somebody else has just killed it). 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**, discard the target pointer, and do nothing further.

3. Output the message "Clang!" (This is not a compiler; Soldiers use a sword.) Call the target's take_hit function with this Agent's attack strength and a pointer to “this” object (See below for the consequences for the target Agent).

4. Then check to see if the target is not **Alive** at this point (maybe we just killed it), and if it is not **Alive**, say "I triumph!" and set the state to **Not Attacking**, and discard the target pointer.

take_hit. Soldiers override this function. First, compute the health with the Agent::lose_health function. If **Attacking**, and now no longer **Alive**, sets the state to **Not Attacking** and forgets the target; but if still **Alive**, no change of state (or target) is made. If this Soldier was **Not Attacking**, and both it and its attacker are **Alive**, it sets its state to **Attacking** and sets its target to its attacker, thus telling itself to attack the attacker, and the "I'm attacking!" message will be output. However, no range check is made and no exceptions will be thrown in this self-command. Thus, if the target is out of range, this will be discovered on the next update.

The Soldier is very stubborn once it starts an attack and will not change its target unless it has first gone out of range or died. Accordingly, if the Soldier is already attacking, when it takes a hit, it does not change the target of its attack in response to the new take_hit.

describe. Output "Soldier " followed by Agent's description, and then the attack information as shown in the examples.

stop. Soldier overrides Agent::stop(), but the only thing stop() does is to output a "don't bother me" message. The Soldier neither stops moving or ceases to attack. These guys are really single-minded once they are told what to do!

Model behavior

The Model must have the following three containers to support its look-up and describe/update functions.

- A container of Sim_object* — all Sim_objects created are added to this container.
- A container of Structure* — all Structure objects created are added to this container and the Sim_objects container.
- A container of Agent* — all Agent objects created are added to this container and the Sim_objects container.

These three containers must be kept in alphabetical order by the name of the object. The container types are up to you. In addition, Model maintains an integer time value that is incremented when the objects are updated, and a container of View* to hold pointers to Views — there is only one in this project, but there will be more in future projects.

When Model is constructed, it creates an initial set of objects with the Structure and Agent factories and adds them to its containers. See the supplied files for the order of creation of the specific objects. Observe the order of constructor messages. When an object is removed, it must be removed from both containers holding its pointer. When Model is destroyed, it must delete any objects remaining in the Sim_object* container, doing so in alphabetical order by name.

attach. When a View is attached to the Model, broadcast_current_state() should be called for each Sim_object.

add_agent, add_structure. When an Agent or Structure is added to Model, Model calls its broadcast_current_state() function.

update. Add one to the time, then update all the Sim_objects; the updates should happen in alphabetical order by name of all of the objects. Then identify all of the Agents that are **Disappearing**, then remove them from the containers, and delete them in alphabetical order. Note that the destructor messages should appear; observe the order of appearance of the messages. The deleted object should no longer appear in any of the program output, and any Agents that were referring to them should have ceased to refer to them while they were **Dying** or **Dead** - your program logic should ensure that there are no dangling pointers to deleted Agents.

View behavior

Controller has the responsibilities of creating the Views and attaching them to the Model. In this first version of the project, on command, Controller will create only one View object, and attach it to the Model, which keeps a pointer to only one View. View generates a map-like display of where the objects are in the plane. See the example output. The display consists of a square matrix of cells, with each cell containing two characters. The size of the matrix is initially 25 X 25 cells, 25 rows and 25 columns. The initial state of the matrix is the "empty" pattern, which consists of a period ('.') in the first (leftmost) character and a space in the second (rightmost) character of each cell. When output, this produces a "grid" effect.

Each cell represents a range of (x, y) coordinates; the size of each cell is the scale. For example, the initial setting is that each cell represents a 2 X 2 square of arbitrary units. In the output, larger values of y are at the top, and larger values of x are to the right. The lower left-hand corner of the display is called the origin; the initial setting is that the origin is at coordinates (-10.0, -10.0). There are member functions that allow the size, scale, and origin of the display to be changed, so that any part of the plane can be covered in whatever detail is desired. View throws Errors if one of the functions receives invalid values. These checks are as follows:

- The map size must be <= 30 and > 6.
- The scale is any double precision floating point value that is greater than zero.
- The origin can be any point given by a pair of double precision floating point values.

The default settings for the matrix are size: 25 X 25, scale = 2.0, origin = (-10.0, -10.0). These settings can be restored with the set_defaults function.

If the map size, scale, or origin is changed, and the draw() function is then called, draw() has to output the map with the new size, scale, or origin with the all of the objects correctly located. This means that View has to "remember" the object names and locations given to it in calls to update_location(). This is easy using an STL container.

To produce the display, each object to be plotted must have been used in at least one call to update_location, which takes the object name and location as arguments. This information is simply remembered. If the update_remove function is called, the name and location is "forgotten" and the object will not be plotted. Then to output the map, call the draw() function. The position of an object is plotted by showing the first two letters of the object's name in a cell, where the cell row and column is produced by translating and scaling the object's position according to the origin and scale of the matrix, and then converting the resulting position to integer subscripts. The function get_subscripts is provided to ensure uniformity in this computation. If two objects are plotted in the same cell, an asterisk ('*') and a space are placed in the cell. You can assume that the characters of an object's name are either letters or digits, and that the name is at least two characters long. This is supposed to be enforced when a new object is created (see below for the **build** and **train** commands). Notice that if the first two characters of two names are the same, the poor user simply has to deal with the possibly ambiguous display.

The draw function outputs the map matrix to produce the display like that shown in the sample output. The size, scale, and origin are printed first. If the subscripts for a particular object lie outside the matrix, a message "<object name> outside the map" is printed next. Multiple objects outside the map are listed separated by a comma and a space. Appearing outside the map is not an error; it simply informs the user that at the current size, origin, and scale, not everything can be seen. Then each row and column for the current size of the map is output.

Note that the grid is plotted like a normal graph: larger x values are to the right, and larger y values are at the top. The x and y axes are labeled with values for every third column and row. You must use the output stream formatting facilities (Output Formatting handout) to save the format settings, set them for neat output of the axis labels on the grid, and then restore them to their original settings. **You must use the stream output formatting functions and manipulators for this purpose. Do not try to "fake" it with your own DIY code fiddling with strings — it will be much harder, and much less useful than learning about the real formatting facilities.** Allow four characters for each numeric value of the axis labels, with no decimal points or places to the right of the decimal point. The axis labels will be out of alignment and rounded off if their values cannot be represented properly in this format. This distortion is acceptable in the name of simplicity for this project.

If you have not worked with a three-dimensional arrays before, I recommend implementing View using a built-in array of char with sizes 30 X 30 X 2. While not the most modern approach, it will be good practice and applicable to C programming as well. If you have worked with multidimensional built-in arrays before, try implementing View using a vector<vector< vector <char> > >, or vector<vector<string> >, which once created, can be used syntactically with subscripts just like a built-in array. Note that vector has a constructor that lets you preset the size of the vector.

Implementation note: Populating and then outputting a 3-dimensional array (or equivalent) is a convenient and intuitive way to generate the output, but it should *not* be a member variable of the View class, because it does not have to retain its contents after the draw() function returns.

Model-View-Controller interaction

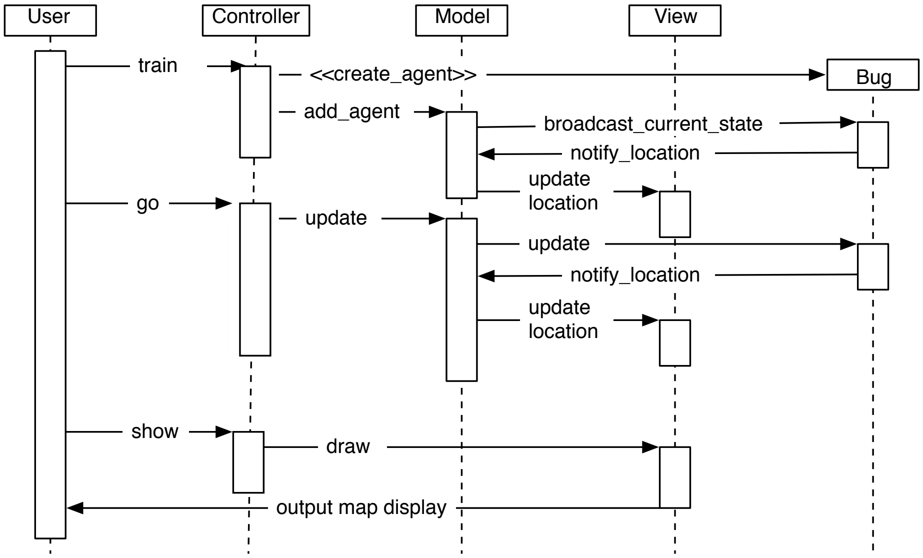
In the Model-View-Controller pattern, Controller normally has the responsibilities of creating the Views and attaching them to Model, and then detaching them and destroying them when they are no longer needed. In this introductory simplified case, there is only one View possible. When Controller gets an open command, it will create a View object with `new` and attach it to the Model, which keeps this single pointer in its container of View pointers. When Controller gets a close command, or terminates, it detaches the View from the model, and then will `delete` it.

The sequence diagram below shows the basic pattern of interaction between the human user, the Model, View, and Controller objects, and a specific Agent object that gets created in response to a user command.

Keep in mind that in this project, there is only one View, and there is only one kind of information shown in this View, namely the objects' current locations. The goal of the Model-View-Controller pattern is to make it as simple as possible to add different kinds of Views and different kinds of view information in the future, with minimum modifications consisting of adding code, rather than changing existing code. We will be doing exactly that in the next projects. So the process will seem roundabout in this Project, but it really works well when there are multiple kinds of Views showing different kinds of information.

This approach recognizes that the Sim_objects are the "experts" about their current state — they know when it changes, and so initiate informing the Views of the change, but we don't want to clutter each object with keeping track of the Views. We want Views to focus on just displaying their information, but with little or no knowledge of the kinds of Sim_objects that are out there. Model will help the process: it knows who the Sim_objects and Views are, and so can be the go-between. Model thus provides a service by which Sim_objects can simply "notify" Model of a state change and then Model will distribute that information to the Views.

Project 4 Sequence Diagram
Showing Basic Model-View-Controller Interaction



The user creates an agent named "Bug". Controller creates the agent (using the factory), then tells Model to add it. Model tells Bug to broadcast its current state. Bug uses Model's notify service to send its initial location to all the current Views (only one in this project). Then, after other commands (not shown), user issues a go command. Controller tells Model to update; Model updates each Sim_object (only Bug is shown). Bug changes location and notifies Model, which forwards the updated location information to View. When the user issues a show command, Controller tells View to draw, producing the map output.

Let's trace this process with Agents. Agents know when their location changes and when they die. So they are the "experts" on their state change. When an Agent changes position, it calls Model::notify_location with its name and new location. Model passes that to all of the currently attached Views by calling the View::update_location function for each View. When an Agent dies, it likewise calls Model::notify_gone, and then Model calls View::update_remove for each View. If an Agent is not moving, it doesn't need to notify Model (or the Views), saving lots of time. Notice how having Model be a global object makes this communication a lot simpler (there is a better solution than a global object, a *Singleton* pattern, which will be used in the next project).

How do we get the View updated if a new Agent is added — which can't be moving right away? If an Agent is added, Model calls its broadcast_current_state() function. This function will call Model::notify_location. Isn't this round-about? In the future, we will want Views that show other information, depending on the type of Agent. This function allows us to customize the calls to notify functions depending on the type of Agent. For example, Peasants might supply how much food they are currently carrying.

What about Structures? In this project, Structures don't move, and don't disappear, and the one kind of View only shows location information. So Structures have a broadcast_current_state() function that calls Model::notify_location when a Structure is added, but would not call Model::notify_location at any other time.

When a View is added with Model::attach(), it needs to be brought up to date. Model will do this by calling broadcast_current_state() for all of the Sim_objects, which will broadcast the current state to all Views for all Sim_objects. This is not very efficient, but it is simple, and keeps Sim_objects insulated from all issues about how many Views are involved. After you implement it this way, you are free to provide a more efficient implementation in Model::attach() so that only the new View is updated, but it must have absolutely no effect on, nor require any changes to, the implementation for Sim_objects or Views.

View saves the update information it receives, namely the name and position of each object currently in the plot. Then when the user asks to see the map, Controller calls the draw() function to output the map. Because View "remembers" which objects it needs to include in the map, it can output the display at any time; the display should always be consistent with the state of the Model, but at the same time, because it only remembers names and locations, it is decoupled from the Model and the Sim_object classes. Also because of its memory for the plotted information, after Controller has told View to change the size, scale, or origin of the map, it can output the display correctly without requiring a fresh update from the model.

Controller: User input syntax and commands

The Controller run function runs a command loop where the user types in commands, and they are carried out. There are three kinds of commands: controlling the program as a whole, controlling the View that shows the map of the objects, and commanding the

Agents to do things. In the Agent commands, the objects are always referred to by name; once the validity of the names are verified, the objects are always referred to by pointer thereafter. The following requirements give some practice with pointers to member functions:

- The top level of Controller should be similar to the top level of Project 3 in that there should be a function for each command that gets the additional input from the user. But unlike Project 3, these command functions must be *member functions* of Controller. This will be more valuable in the next versions of the Project, in which Controller will need more member variables.
- You *must use one or more map containers to map between the command strings and pointer-to-member-functions of Controller to implement the top-level command loop*. A possible approach would be to use different containers for the different kinds of commands (see below). Using a map will both neaten the code and give useful practice with pointers to member functions.

Command words and parameters are strings (with no embedded whitespace) or numbers. Normally, these will be whitespace-delimited in the user's input, but you should read them in as in previous projects, where input whitespace is only required to separate data items that could not otherwise be separated by the standard input operators. Most of the numbers read by this program are to be read into double-precision floating-point variables (type is "double"), and non-integral input values are possible. If you have not had experience reading in floating point numbers, you must read the handout on Stream I/O. If an integer is expected, and one is not read successfully, an Error is thrown. If a double value is expected, and one is not read successfully, an Error is thrown. Notice that redundant decimal points are optional in floating-point input: an integral value like "42" can be read correctly into a double automatically.

The command syntax and processing is as follows (in this order):

1. Read the first word.
2. The first word should be either "quit", the name of an Agent, or a command word.
3. If "quit", output "Done" and return from Controller. Refer to the supplied p4_main.cpp: Controller's destructor should delete the View if it is still open. Then Model's destructor should delete all the current Sim_objects. Observe the order of destructor messages.
4. If it is the name of an Agent, check that the Agent is_alive(), and throw an Error if not. If the Agent is alive, then the next word should be an agent command word. The next items read depend on the command. The information is sent to the Agent named in the first command word.
5. If the first word is not the name of an Agent, then it should be a command word for the program as a whole or for the View, and the next items read depend on the command.
6. If the first word is neither an Agent name nor a valid command word, then throw an Unrecognized command Error.

Error handling. The logic for reading data and handling errors is the same as in previous projects: read and verify input data items one at a time, with a couple of deviations. All error messages produced by your code will be packaged as Error exception objects and thrown; they will be caught at the end of the Controller's command loop, the message output, the rest of the input line skipped, and a new command prompted. Controller should have one catch, for the supplied Error class, which should print the message using the what() function declared in std::exception. The supplied p4_main.cpp is responsible for catching all other kinds of exceptions, so it has two additional catches: one for std::exception that prints the message using the what() function, which covers all exceptions produced by the Standard Library and C++ operators, and another catch-anything in case an unknown exception was thrown.

The View commands are:

- **open** - create the View and attach to the Model. If theView is already open, throw an Error.
- **close** - detach the View from the Model and destroy it. If the View was not open, throw an Error.
- **default** - restore the default settings of the map. If the View is not open, throw an Error.
- **size** - read a single integer for the size of the map (number of both rows and columns). If the View is not open, throw an Error.
- **zoom** - read a double value for the scale of the map (number of units per cell). If the View is not open, throw an Error.
- **pan** - read a pair of double values for the (x, y) origin of the map. If the View is not open, throw an Error.

The whole-program commands are:

- **quit** - see above.
- **status** - have all the objects describe themselves; the output should be in alphabetical order by name of all of the objects.
- **show** - tell the View to draw itself; it should already have been updated as needed.
- **go** - call the Model::update() function.
- **build** - create a new Structure and add it to the Model. Read a string for the object name. Check first and throw an Error that the new object name is invalid if the name is less than two characters in length, and contains other characters besides letters or numbers, or is the same as one of the commands (e.g. "zoom" or "quit"), then ask Model to check whether it is already in use —

that is, it is identical to the name of an existing Structure or Agent. Then read a string for the object type ("Farm" or "Town_Hall"), but do not check it for validity yet. Then read a location as a pair of doubles. If a double cannot be read, throw an Error. Pass the name, type, and location information to the Structure_factory. If the object type is unknown, the factory should throw an Error of trying to create a structure of unknown type; this way the main module does not need to know the names of the different possible Structure types — only the factory does.

- **train** - create a new Agent and add it to the Model. This parallels the **build** command. Read a string for the object name and apply the same validity checks as for the **build** command. Then read a string for the object type ("Peasant" or "Soldier"), and a location as for **build**. Pass the information to the Agent_factory. If the object type is unknown, the factory should throw an Error of trying to create agent of unknown type.

The Agent commands are:

- **move** - read an (x, y) position for the Agent to go to. x, y can have any numerical double values.
- **work** - read a source Structure name and a destination Structure name. After reading each name, see if the corresponding object appears in the Structures container, and if it is not present, throw an Error that the Structure was not found. Note that no check is made for whether the Structure has the appropriate type to function as a source or destination. For example, a Peasant sent to deposit food at a Farm will do so, even though Farms do not accept deposits like Town_Halls do.
- **attack** - read an Agent name for the target of the attack. The commanded Agent will attack this target. Check that the target name is a valid name for an Agent (using Model::get_agent_ptr), which will throw an agent not found Error if it is not.
- **stop** - no additional data required

Utility.h, .cpp

This module is required to contain an Error class, supplied in the skeleton starter file, similar to previous projects, but with the following changes: First, it inherits from `std::exception` and implements the virtual `what()` function, so that the error messages can be printed in the same way throughout. Second, it contains a `std::string` member variable containing the message, and accepts a string as a constructor argument. This makes it easy to build an error message containing the string for the Sim_object name prepended to the text of the message. This is considerably more flexible than our previous Error objects that only contained a pointer to a C-string literal.

Any definitions, declarations, functions, templates, or classes used by *more than one module* and consistent with good header file practice can be placed in this module. As noted previously, your Utility.h should not #include and headers not needed by Utility.h itself. Your Utility module will be used with the component tests that all involve building a complete project executable by combining some set of your modules with the remaining modules coming from the instructor's solution.

Files to be submitted:

p4_main.cpp

.h, and .cpp for:

Model, View, Controller, Sim_object, Structure, Farm, Town_Hall, Agent, Peasant, Soldier, Agent_factory, Structure_factory, Geometry, Moving_object, Utility

General Requirements

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

- Your code must conform to the C++ Coding Standards document and other guideline documents; review and apply it your code during development.
- The program must be coded only in Standard C++; no C I/O functions are allowed.
- Your use of the Standard Library should be straightforward and idiomatic, along the lines presented in the books, lectures, and posted examples. Other than the map container for command dispatching, and unlike Project 3, there are no specific requirements for which containers or algorithms you use for which purpose, but you choices should be good ones. In particular, be sure to use heterogenous lookup, algorithms, iterators, bind, mem_fn, function, and lambda, wherever they can be applied simply and directly to get cleaner code. *Hint:* The Model containers for Agents, Structures, and Sim_objects do not have to be the same type — a good choice here can really simplify the code — think about what the containers are needed for: which ones need to support iteration in alphabetical order, and which need to support lookup by name? Consider heterogenous lookup — it might simplify things further.
- You may *not* use the C++ Standard Library smart pointers (`shared_ptr` or `unique_ptr`), nor may you implement your own version of them. We will be using them in the next project! Raw pointers only in this project!

- Similar to Project 2 and 3, you should ensure basic exception safety in case a container insertion fails due to e.g. memory exhaustion. When you add a new object pointer to a container, you should have a local try/catch-everything structure so that the new object can be deleted before rethrowing the exception. Since you will be adding Sim_objects to two containers in Model, think carefully about how to arrange this local cleanup and the Model destructor cleanup so that (1) the new object definitely gets deleted eventually, and (2) you don't get any double-deletion errors when the destructor runs. This process will be automatic when we use smart pointers in the next project.
- Except for the specified `g_Model_ptr` global variable, and the Standard Library global objects `cout` and `cin`, you may not use any global variables, either file-scope (internal linkage) or program-wide, anywhere for any reason.
- Any member functions of your own choice and design, like those in Controller, should be const-correct – that is, declared as const member functions if they do not alter the state of the object.
- **Minimal header file requirement.** For practice in proper decoupling, in this project, your header files for classes must include the absolute minimum of other header files. Review the C++ Header File Guidelines. With the exception of trivial reader functions, all member functions must be defined in your .cpp files. Incomplete declarations must be used where possible in header files. These steps help minimize the #includes required in a class's header file. In addition, you must avoid any unnecessary #includes in your .cpp files. Review the above descriptions for how the class design allows most components to be decoupled from the leaf classes of Sim_object, and ensure that your header file treatment follows this organization. Finally, no header file is allowed to include <iostream>.

Suggestions

Like all but the most trivial pieces of code, this project is too big and complex to implement all at once, and rewards building a bit at a time. While the specifications are complex, this is due mainly to the need to spell out exactly what update sequences and computations need to be performed so that the code behaves in a known and consistent way; in fact, relatively little code has to be written for each class — the specification takes up more bytes than the actual code! However, plan on spending a lot of time understanding the specifications and getting acquainted with how the project works a bit at a time.

I suggest starting with building the Structure classes and a simple main function to test them. This is a "unit test" approach. You can test the classes separately, as "units", before you try to combine them in complex interactions that are harder to test and debug. First, in your main function, create a couple of Structures, and then tell them to describe themselves, then update them, then have them describe themselves again. The Farms should show the right change. Call the deposit and withdraw functions — they should work as expected. Then try building Agent as a concrete class, create one, and then tell it to move, then update it and call its describe function. Verify that the Agent is moving correctly. Put these calls in a loop, and verify that it stops moving at its destination. Call the take_hit function and verify that the Agent changes state after it is "killed." You could do the same with developing the Soldier and Peasant classes (do the simpler Soldier first), but at some point, these classes will be working well enough that building a first bare-bones version of Controller and Model will make your testing easier. Add the rest of the command functions as you go along.

The major choice point is when you implement View. View is basically simple but nit-picky, especially if you haven't done output formatting before. *Read the handout!* Note that you can unit-test View by itself as well, because View is so well decoupled that it does not need any of the other classes to function properly. View can make it easier to tell what you program is doing, but the textual output of describe, and the blow-by-blow output provided by update, is the most definite way to ensure that the Agents are behaving exactly like they should.

Project Evaluation

I will test your program's output separately with and without View output. This will help isolate any issues having to do with correctness of your View class. If your program fails to match the posted samples due to differences in numeric values (or map positions) please let me know promptly; likewise if it matches the posted samples, but fails to match the corresponding sample tests on the grading machine. I will also mix and match some of my components and your components, so adherence to the public and protected interfaces is critical. Be especially careful to follow the specifications on which messages are emitted by which component.

This project may be evaluated using some spot checks for key aspects of the design as some basic code quality issues. But the next projects, where design and code quality will be evaluated, will be based on this one. Doing quality coding on this project will be rewarded by easier development of the remaining projects.