M3: Differentially Interacting Warriors

**Design Description:**

**Agent:**

The two main patterns used in our design are “Double Dispatch” pattern and “Template Method” pattern.

Since each warrior has to react differentially depending on the type of the attacker, the exact behavior is determined by both the type of the attacker and the type of the target. Thus the function called depends on the type of more than one object, which naturally leads to the Double Dispatch pattern. As indicated by the UML class diagram, the Agent class now has three new functions, **virtual void take\_hit(int, shared\_ptr<Soldier>)**, **virtual void take\_hit(int strength, shared\_ptr<Archer>)**, and **virtual void take\_hit(int strength, shared\_ptr<Witch\_doctor>)**. All these three functions have default implementations in the Agent class (lose\_health() when attacked) so that non-warrior agents (such as Peasant) can simply inherit from the Agent class without providing any implementation. The original function, **virtual void take\_hit(int, shared\_ptr<Agent>)** is deleted because the type of the second argument of take\_hit will always be the precise derived type and never a general Agent (as were in Project 4 and 5). Thus this function will never be called due to the presence of the more specific take\_hit() functions. Each derived class of Warrior overrides these three functions to provide specific behaviors when attacked.

We used the “Combination of virtual functions and overloaded functions” method to solve the double dispatch problem. Since attacking is a single-directional behavior (as opposed to intersecting as presented in the lecture notes, which is a mutual), although all classes are in the same hierarchy, the resulted code looks more like the one for classes in different hierarchies. This works as the follows: suppose we have two warriors W1 and W2, and W1 is attacking W2, then in the update function of W1, the take\_hit() function of W2 will be called, and W1 will pass itself as the attacker pointer. As you will see in the UML sequence diagram, once W2’s take\_hit() function is called, it will do a series of function calls on its own, or on the Model (described below), or on its attacker, to decide what should be done next. W1 will then keep checking if W2 is alive in the next update() call and repeat the procedure above.

Different from the raw **this** pointer, since it is the Agent class that inherits from enable\_shared\_from\_this<> class, if we call shared\_from\_this() from any of its subclasses, we will get a shared\_ptr<Agent> instead of a shread\_ptr of its derived type. For example, even though we call shared\_from\_this() inside an Archer class, we will get a shared\_ptr<Agent> instead of a shared\_ptr<Archer>. But in order to call the correct function we need the exact type of the attacker, and here is where the “Template Method” pattern comes into play. In the Warrior::update() code in Project 5, we called take\_hit(int, shared\_ptr<Agent>) directly. But in Project 6, we wrap this function around using a virtual function called dispatch\_hit(). This function is pure virtual in Warrior class, and is overridden by the derived classed of Warrior. So if this function is called on an Archer object, it will be resolved to the Archer’s version of dispatch\_hit().

In each of the dispatch\_hit() function of each leaf class, the attacker will first print out an attack word (such as “Clang!”), and then call target -> take\_hit(), passing itself as the second argument. Since calling shared\_from\_this() will always return a pointer of type shared\_ptr<Agent>, a static\_pointer\_cast() is required to cast the pointer to the exact type of the attacker. Since we are calling shared\_from\_this() in the leaf class, we know exactly that the returned pointer is of the leaf class type. Thus a static\_cast is always correct. Once we get the exact type, the function overloading mechanism will ensure that the correct take\_hit() function will be called on target’s side, thus the target can react differentially according to what the type of the attacker is.

Therefore, there are two virtual calls under the hood: first is the call on dispatch\_hit(), which will be resolved to calling the function defined in the attacker leaf class. Once we obtain the exact type of the attacker, a second virtual call on take\_hit() will call the corresponding overloaded function defined in the target leaf class. The dispatch problem is thus solved by two virtual function calls.

**Model:**

Some modifications to the Model class are made so that new features can be supported. For example, since an Archer will run away to the farthest structure when attacked by a Soldier, it has to query the Model to know where the farthest structure is. Thus a new function, get\_farthest\_structure\_ptr(), is implemented to get this information. Since it differs from get\_closest\_structure\_ptr() only in exact place, the code is refactored so that code reuse is maximized. To be more specific, the comparator class is now taken out of get\_closest\_structure\_ptr() and declared in the private section of the Model class. This is done so that both get\_closest\_strcture\_ptr() and get\_farthest\_structure\_ptr() can use it. It is not declared as static in the .cpp file because it utilizes the private type alias defined in Model, which makes the code much cleaner.

Since Model is involved in the take\_hit() behaviors, it is included in some of the UML Sequence Diagrams to indicate that a Warrior will query certain information from Model and react according to the return value.