Department of Informatics, Technical University of Munich
Scientific Computing in Computer Science
G. Chourdakis, J.-M. Gallard, F. Simonis

Winter term 2019/20

Advanced Programming Tutorial
Project, Sprint 3: Levelling up and classes

Sprint description

In this third phase, the player character can level up and change his RPG class from the starting Trainee to a Mage or Fighter.

This phase will mostly focus on advanced OOP features (inheritance) as well as using a factory pattern.

User stories

Leveling up

My character has a level that I can see when displaying my stats. After each successful fight, I gain a level and my stats increase a bit.

Player classes

My character starts as a "Trainee". I can see my current class when displaying my stats. Once I'm above a certain level, I can change my class to one of the upgraded classes (see "Fighter" and "Mage" user stories).

Hint: To change the class, you will need to replace the player object managed by the game engine with a new one of the new correct class (and taking over current stats). Using a smart pointer in the engine to hold the player object and a factory pattern to get a new object from the old one facilitates this process.

Fighter class

As a Fighter class, I have a new "critical strike chance" stat representing a percentage. When I attack, I have a chance to perform a critical strike. If I do, I attack for a 1.5 times my attack stat.

When I level up, my stats increases faster than when I was a Trainee.

Hint: A random number generator will be required to see if an attack is a critical strike or not.

Mage class

As a “Mage”, I have 3 new stats: “mana”, “max mana” and “magic”. When I attack, if I have enough mana, I spend some to attack using a magical spell that uses my magic stat instead of my attack stat to inflict damage. Otherwise I attack using my attack stat. Resting restores my mana to full.

When I level up, my magic stat increases faster than my attack stat.

Technical Hints

This section describes some problems that will be encountered during this sprint. It also provides some technical hints you are free to take into consideration. You can also come up with your own solution as long as the user stories are fulfilled.

Inheritance

To avoid duplicating code, inheritance should be used.

The Trainee class can be a child class of the basic character class. The Mage and Fighter classes are children of Trainee and each add its own specificities. Some methods will need to be overloaded and thus have to be declared `virtual` in the base class, for example the resting method will need to be overloaded in the Mage class for it to restore mana too. It is a good practice, even if optional, to use the `override` keyword in child classes when a virtual method is overriden.

When using inheritance, be aware of the access modifiers (`public`, `protected` and `private`) and do not forget to use public inheritance (e.g. `class B : public A`), otherwise private inheritance will be used by default, which would prevent the use of polymorphism we desire.

Polymorphism

Using inheritance allows the engine to use polymorphism and to only work with objects and pointers of the base class, as long as encapsulation is properly made, by upcasting (casting to a parent class) instances of child classes. If your code uses it properly, you should not have to change anything in the engine code to introduce new child classes of your base class.

To be able to find out the true class of an upcasted reference, you can try using `dynamic_cast`, but a simpler method would be to have some field (int or string for example) in the object identifying its true type. You will probably need this kind of ID anyways to be able to properly save and load the player character with its right class.

Be aware that upcasting does not work well with already allocated unique pointers (can be done but not easy/clean) but works with references and when creating the pointer (e.g. `unique_ptr<A> a = make_unique<B>();`). Also when using polymorphism, you should explicitly define the destructor of your classes as virtual to avoid undefined behaviors.

Factory pattern

To be able to keep the engine unaware of the multiple classes, a factory pattern can be used. The factory is responsible for creating an object of the correct class but returns it as a pointer of its base class to be taken ownership by the engine. For example if `B` and `C` are child classes of `A`, the following code is an implementation of a factory:

```cpp
// returns an instance of B if t==1, C if t==2, A otherwise
std::unique_ptr<A> createObject(int t) {
  if(t == 1) {
    return std::make_unique<B>();
  }
  if(t == 2) {
    return std::make_unique<C>();
  }
  return std::make_unique<A>();
}
```

The code calling `createObject(t)` and using the returned pointer needs only to be aware of `A` and uses the methods of `A`. If the object behind the pointer is an instance of `B` or `C` that override some base methods, then the overriden methods will be used. The factory pattern fully exploit the benefits of polymorphism and facilitate the integration of new child classes into the code at a later point in time.

The factory can be used when deserializing or when upgrading a class.