1 Introduction

In games its very common to have multiple classes that are similar. In this assignment we are going to work with classes that represent items a player can use.

We will have four types of items: `Weapon`, `Armor`, `Shield` and `Sword`. Each item in this supposed game will have `damage` (how many hitpoints it will remove from enemies when attacked) and `defense` (how many damage points will be removed when an enemy attack the player). These ratings are retrieved by calling `damage()` and `defense()` respectively on the item.

Here is a short summary of each item type:

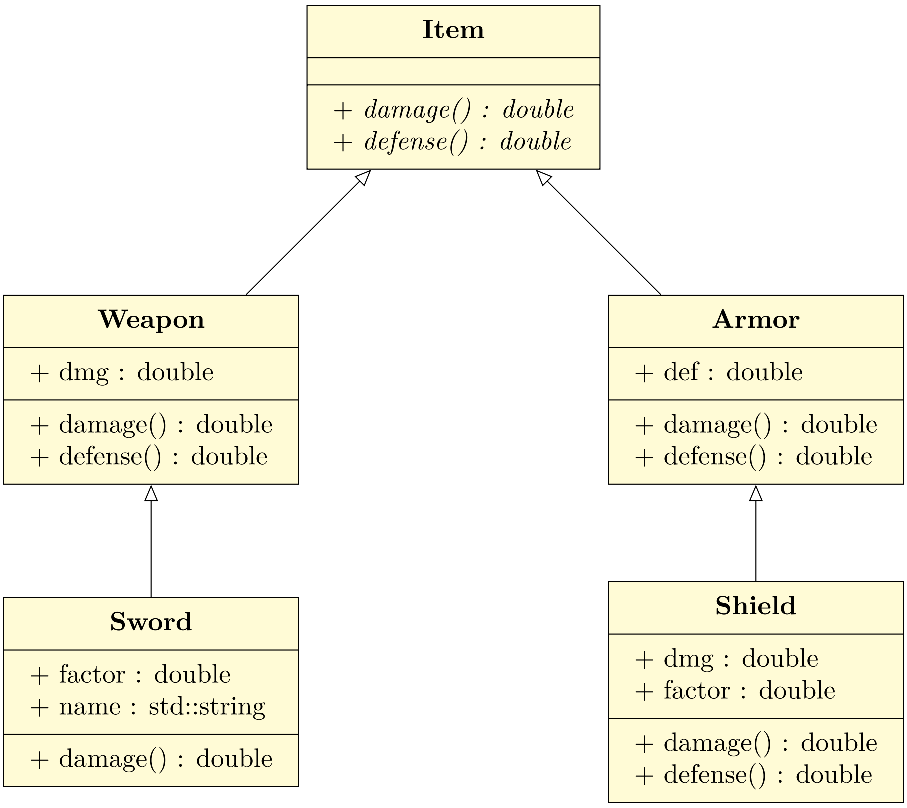
Weapon Has a fixed damage stored in data member `dmg` and 0 defense.

Armor Has a fixed defense stored in data member `def` and 0 damage.

Shield Has a fixed damage stored in data member `dmg` and a fixed defense stored in data member `def`. However it also has a data member `factor` which is multiplied with `def` whenever `defense` is called.

Sword Has a name stored in a string `name`. Has a fixed damage stored in data member `dmg` and a data member called `factor` that is multiplied with `dmg` whenever `damage()` is called. It has 0 defense.

There are multiple ways to implement this. Here is one way:

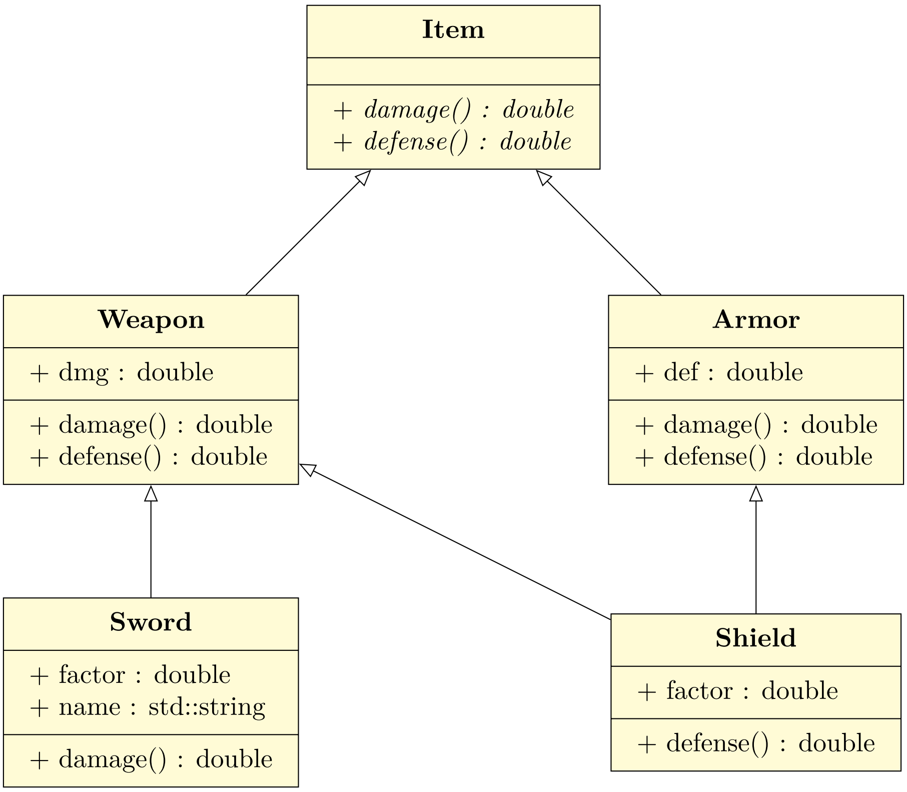


With the following implementations:

```
double Weapon::damage() const
{
  return dmg;
}

double Weapon::defense() const
{
  return 0.0;
}

double Armor::damage() const
{
  return 0.0;
}

double Armor::defense() const
{
  return def;
}

double Sword::damage() const
{
  return factor * Weapon::damage();
}

double Shield::damage() const
{
  return dmg;
}

double Shield::defense() const
{
  return factor * Armor::defense();
}
```

This will work fine but it will quickly get complicated if we add more types of items. Also notice that `Shield::damage` and `Weapon::damage` have exactly the same implementation. Of course, it is fine in this case since they are only one-liners, but imagine we make them more complicated...

One way to solve this code-duplication problem is to use multiple inheritance:



However this introduces *the diamond problem* since `Shield` now inherits twice from `Item`: once through `Weapon` and once through `Armor`. This would force us to introduce virtual inheritance which makes the code a lot slower.

Both of these options work, but one can imagine more complicated situations where the flaws of these two designs gets more troublesome.

2 The exercise

Instead of implementing the design(s) described above, you should solve this problem with *mixins*.

The idea is as follows:

Create a class called `Item_Base` which has two pure-virtual functions `damage()` and `defense()`.

Create a variadic class template called `Item` that takes multiple *components*. `Item` should inherit from `Item_Base` and all the template parameters (i.e. inherit from all components).

Create five components (classes without a base class):

Attack A class with data member `dmg` and a function `damage` that returns `dmg` (you can add whichever parameters you choose).

Defend A class with data member `def` and a function `defense` that returns `def` (you can add whichever parameters you choose).

Damage_Multiplier A class with data member factor and a function damage that takes in the current damage and returns the factor multiplied with the current damage.

Defense_Multiplier A class with data member factor and a function defense that takes in the current defense and returns the factor multiplied with the current defense.

Named A class that contains a public data member name that is a std::string.

These are the components we will add to the Item to create our different versions of the items. I.e.

```
using Weapon = Item<Attack>;
using Armor  = Item<Defend>;
using Shield = Item<Defend, Attack, Defense_Multiplier>;
using Sword  = Item<Named, Attack, Damage_Multiplier>;
```

In order for this to work properly Item must override damage() and defense() with the following implementations:

1. Create a variable total that keeps track of the curret damage or defense points.
2. Go through each component and call corresponding damage (or defense) function. Pass in total as a parameter and store the return value into total.
3. Return total.

Note: Not all components have a damage or defense function, so you should only call them from those components that have them. As a hint, create a function:

```
template <typename T>
double damage_helper(T const* obj, double total)
{
  return obj->damage(total);
}
```

That returns 0.0 if obj doesn't have a damage function (you might have to modify the parameters).

In item.cc there are a few testcases.