


Chapter VI

Exercise 02: This code is unclean. PURIFY IT !

	Exercise 02
This code is unclean. PURIFY IT !	
Turn-in directory : <i>ex02/</i>	
Files to turn in : <i>Squad.hpp</i> , <i>Squad.cpp</i> , <i>TacticalMarine.hpp</i> , <i>TacticalMarine.cpp</i> , <i>AssaultTerminator.hpp</i> , <i>AssaultTerminator.cpp</i> , <i>ISpaceMarine.hpp</i> , <i>ISquad.hpp</i> , <i>main.cpp</i>	
Forbidden functions : <i>None</i>	
Remarks : <i>n/a</i>	

Your mission is to build an army worthy of the Valiant Lion Crusaders. Painted with orange and white stripes. Yeah, yeah, really.

You'll have to implement the elements of your future army, namely a `Squad` and a Tactical Space Marine (`TacticalMarine`)

Let's begin with a `Squad` . Here's the interface you'll have to implement (Include `ISquad.hpp`) :

```
class ISquad
{
    public:
        virtual ~ISquad() {}
        virtual int getCount() const = 0;
        virtual ISpaceMarine* getUnit(int) = 0;
        virtual int push(ISpaceMarine*) = 0;
};
```

You will implement it so that :

- `getCount()` returns the number of units currently in the squad.
- `getUnit(N)` returns a pointer to the Nth unit (Of course, we start at 0. Null pointer in case of out-of-bounds index.)

- `push(XXX)` adds the `XXX` unit to the end of the squad. Returns the number of units in the squad after the operation (Adding a null unit, or an unit already in the squad, make no sense at all, of course...)

In the end, the `Squad` we're asking you to create is a simple container of `SpaceMarines`, which we'll use to correctly structure your army.

Upon copy construction or assignation of a `Squad`, the copy must be deep. Upon assignation, if there was any unit in the `Squad` before, they must be destroyed before being replaced. You can assume every unit will be created with `new`.

When a `Squad` is destroyed, the units inside are destroyed also, in order.

For `TacticalMarine`, here's the interface to implement (Include `ISpaceMarine.hpp`):

```
class ISpaceMarine
{
    public:
        virtual ~ISpaceMarine() {}
        virtual ISpaceMarine* clone() const = 0;
        virtual void battleCry() const = 0;
        virtual void rangedAttack() const = 0;
        virtual void meleeAttack() const = 0;
};
```

Constraints:

- `clone()` returns a copy of the current object
- Upon creation, displays : "Tactical Marine ready for battle"
- `battleCry()` displays "For the holy PLOT !"
- `rangedAttack()` displays "* attacks with bolter *"
- `meleeAttack()` displays "* attacks with chainsword *"
- Upon death, displays : "Aaargh ..."

Much in the same way, implement an `AssaultTerminator`, with the following outputs :

- Birth : "* teleports from space *"
- `battleCry()` : "This code is unclean. PURIFY IT !"
- `rangedAttack` : "* does nothing *"
- `meleeAttack` : "* attacks with chainfists *"

- Death : “I’ll be back ...”

Here’s a bit of test code. As usual, yours should be more thorough.

```
int main()
{
    ISpaceMarine* bob = new TacticalMarine;
    ISpaceMarine* jim = new AssaultTerminator;

    ISquad* vlc = new Squad;
    vlc->push(bob);
    vlc->push(jim);
    for (int i = 0; i < vlc->getCount(); ++i)
    {
        ISpaceMarine* cur = vlc->getUnit(i);
        cur->battleCry();
        cur->rangedAttack();
        cur->meleeAttack();
    }
    delete vlc;

    return 0;
}
```

Output :

```
zaz@blackjack ex02 $ clang++ -W -Wall -Werror *.cpp
zaz@blackjack ex02 $ ./a.out | cat -e
Tactical Marine ready for battle$
* teleports from space *$
For the holy PLOT !$
* attacks with bolter *$
* attacks with chainsword *$
This code is unclean. PURIFY IT !$
* does nothing *$
* attacks with chainfists *$
Aaargh ...$
I'll be back ...$
```

Be thorough when you’re making the `main` function that you will turn in to get your grade...

A `Materia` has an XP total starting at 0, and increasing by 10 upon every call to `use()`. Find a smart way to handle that !

Create the concrete `Materias` `Ice` and `Cure`. Their type will be their name in lowercase ("ice" for Ice, etc ...).

Their `clone()` method will, of course, return a new instance of the real `Materia`'s type.

Regarding the `use(ICharacter&)` method, it'll display:

- Ice : "* shoots an ice bolt at NAME *"
- Cure : "* heals NAME's wounds *"

(Of course, replace NAME by the name of the `Character` given as parameter.)



While assigning a `Materia` to another, copying the type doesn't make sense...

Create the `Character` class, which will implement the following interface :

```
class ICharacter
{
    public:
        virtual ~ICharacter() {}
        virtual std::string const & getName() const = 0;
        virtual void equip(AMateria* m) = 0;
        virtual void unequip(int idx) = 0;
        virtual void use(int idx, ICharacter& target) = 0;
};
```

The `Character` possesses an inventory of 4 `Materia` at most, empty at start. He'll equip the `Materia` in slots 0 to 3, in this order.

In case we try to equip a `Materia` in a full inventory, or use/unequip a nonexistent `Materia`, don't do a thing.

The `unequip` method must NOT delete `Materia` !

The `use(int, ICharacter&)` method will have to use the `Materia` at the `idx` slot, and pass `target` as parameter to the `AMateria::use` method.

As usual, here's a test main that you'll have to improve on :

```
int main()
{
    IMateriaSource* src = new MateriaSource();
    src->learnMateria(new Ice());
    src->learnMateria(new Cure());

    ICharacter* zaz = new Character("`zaz'");

    AMateria* tmp;
    tmp = src->createMateria("`ice'");
    zaz->equip(tmp);
    tmp = src->createMateria("`cure'");
    zaz->equip(tmp);

    ICharacter* bob = new Character("`bob'");

    zaz->use(0, *bob);
    zaz->use(1, *bob);

    delete bob;
    delete zaz;
    delete src;

    return 0;
}
```

Output :

```
zaz@blackjack ex03 $ clang++ -W -Wall -Werror *.cpp
zaz@blackjack ex03 $ ./a.out | cat -e
* shoots an ice bolt at bob *$
* heals bob's wounds *$
```

Don't forget to turn in your main function, because you... well, okay, you know the drill now, don't you ?

Here's the interface to implement for your mining lasers :

```
class IMiningLaser
{
    public:
        virtual ~IMiningLaser() {}
        virtual void mine(IAsteroid*) = 0;
};
```

Implement the two following concrete lasers : `DeepCoreMiner` and `StripMiner` .

Their `mine(IAsteroid*)` method will give the following output :

- `DeepCoreMiner`

```
``* mining deep ... got RESULT ! *''
```

- `StripMiner`

```
``* strip mining ... got RESULT ! *''
```

You'll replace `RESULT` with the return of `beMined` from the target asteroid.

We'll also need some asteroids to pum... er, i mean mine. Here's the corresponding interface :

```
class IAsteroid
{
    public:
        virtual ~IAsteroid() {}
        virtual std::string beMined(...) const = 0;
        [...]
        virtual std::string getName() const = 0;
};
```

The two asteroids to implement are the `AsteroBocal` and the `BocalSteroid` . Their `getName()` method will return their name (You don't say ?), which will be equal to the class name.

Using subtype and parametric polymorphisms (and your brain, hopefully), you will do so that a call to `IMiningLaster::mine` yields a result depending on the type of

asteroid AND the type of laser.

The returns will be as follows :

- StripMiner on BocalSteroid : "Krpite"
- DeepCoreMiner on BocalSteroid : "Zazium"
- StripMiner on AsteroBocal : "Flavium"
- DeepCoreMiner on AsteroBocal : "Thorite"

To that end, you will need to complete the `IAsteroid` interface.



You probably will need two `beMined` methods ... They would take their parameter by non-const pointer, and would both be `const`.



Don't try to deduce the return from the asteroid's `getName()`. You NEED to use TYPES and POLYMORPHISMS. Any other devious way (`typeid`, `dynamic_cast`, `getName`, etc ...) WILL net you a -42. (Yes, even if you think you can get away with it. Because no, you can't.)

Think. It's not that hard.



DD's patcher. (Copyright 2010 "zaz's daily joke")

Now that our toys are finally ready, make yourself a nice barge to go mine with. Implement the following class :

```
class MiningBarge
{
    public:
        void equip(IMiningLaser*);
        void mine(IAsteroid*) const;
};
```

- A barge starts without a laser, and can equip 4 of them, not more. If it already has 4 lasers, `equip(IMiningLaser*)` does nothing. (Hint: We don't copy.)

- The `mine(IAsteroid*)` method calls `IMiningLaser::mine` from all the equipped lasers, in the order they were equipped in.

Good luck.

PS : No, you won't have any test main function. You're big boys now, make your own.

<insert a witty comment about how the students need to turn in their `main` function to get a grade, preferably with some veiled insults on Microsoft developers>