14CS413 / UE14CS413
B.E / BTech 2014-2018
PES Institute of Technology / PES University

Design Patterns Mini Project
December 2017
7th Semester

N-Ton Pattern in a Class Hierarchy

1PI14CS127 - Varun Rao
01FB14ECS131 - Nikhil Prabhu

GitHub: varunnrao/dp-project

Description of the Pattern:

The pattern we have implemented is an N-Ton imposed on a set of classes related through inheritance. Given a value N, any combination of class objects can be created and used as long as the total number of objects add up to N. Objects can be deleted to free up space for more objects to be created. If this is not done, exit handlers delete the objects in reverse order of creation, ensuring a longevity in lifetime. A dependency policy is also imposed. Given a set of objects 'B' who depend on objects of 'D', the objects of the latter cannot be deleted cannot be freed unless no objects of the former exist, or they have been appropriately deleted. This dependency can interfere with the operation of the exit handlers so the user should be careful to handle the deletion of such objects in the appropriate manner.

Implementation Details:

Overview:

All classes inherit from a super class called NTon, that has protected members such as N, the maximum value of objects permitted to exist, and curr_num_of_objects, the number of objects currently in existence. Both these members are static as the information they represent correspond to the class and not to any objects.

It also has a static protected virtual function `del()` that is overridden by its child classes.

It has public static members `set_N(int)` for the purpose of setting the value of `N` as well as `get_curr_num_of_objects()` to get the current number of objects in existence.

In addition it has a function `disp()` that prints the details of all created objects.

Static protected function `exit_handler()` and member `vec_obj` will be explained later in this report with respect to the creation and deletion of objects.

Support for multiple, multi-level, and linear inheritance is provided.

Our implementation however, does not guarantee thread safety and does not safeguard against race conditions in the event of dispatching the `NTon` class on multiple threads.

NTon Interface:

Each class that is a descendant of the `NTon` class is uniform in its design. Each class has the following interface

```
public:
     static ClassName* get();
     static int get_count();
     virtual void del();
     virtual ~ClassName();
private:
     void operator delete( void * );
     static int count;
protected:
     ClassName();
```

Where, in our implementation, the classes are A, B, C, D, and E.

Object Creation:

`get()` is used for the purpose of creating objects. The constructor of each class is kept private so that the client cannot create objects directly.

It works by first checking if there is space for creation of a new object. If this condition is met, it increments the appropriate counts and then creates a new object of the class. In addition to this, it stores a pointer to this newly created object in the vector member of the `NTon` class `vec_obj`.

```
     vec_obj.push_back(new ClassName());
     return dynamic_cast<ClassName*>(vec_obj.back());
```

This member is necessary for the purpose of deletion, both by using `del()` and the automatic deletion by exit handlers. The `atexit(void (*function)(void))` function expects a handler with no argument. Since the exit handler must call `delete (void* ptr)` which expects a pointer to an object as argument, an object reference must be captured as a class data member - done through the `vec_obj` vector.

Using our own static object factory `get()` provides two benefits. Firstly, this implements the "Named Constructor Idiom" ensuring objects of the class are always created via new rather than as local, namespace-scope, global, or static since the default constructor is protected and the only way to obtain an object is through the factory which creates a dynamically allocated object and returns a pointer to it. Secondly, if the constructor were to be explicitly called by the client, the creation of a derived class object would result in constructor calls to all its ancestors causing the count of the current number of objects to be updated incorrectly. Having the factory, thus, circumvents this issue.

Object Deletion:

When `del()` is called, the object is extracted from the vector and then deleted as shown below.

```
void ClassName::del()
{
      count--;
      curr_num_of_objects--;
      auto it = find(vec_obj.begin(), vec_obj.end(), this);
      if(it != vec_obj.end())
      {
            assert((dynamic_cast<ClassName*>(*it)) == this);
            delete( dynamic_cast<ClassName*>(*it) );
            vec_obj.erase(it);
      }
}
```

The `operator delete` of each class is hidden in the private section of the class so that the deletion of the object is only possible using this function.

The reason the `del()` operation is used instead of directly calling the destructor is that a call to the destructor calls the destructors of all parent classes. Because the program requires to keep account of the increase and decrease in counts, this decrement operations cannot be placed in the destructors – they would decrement values of classes for which objects have not been deleted. Instead these operations are placed in the `del()` function inside which the destructor is called.

`NTon` contains a `virtual del()` so as to ensure that the `del()` function is polymorphically dispatched and appropriate `del()` function is called according to the object being deleted.

Ensuring Longevity and `atexit` handlers:

In the case that the client does not delete the objects they have created, the objects will be automatically deleted by the exit handler – in reverse order of creation (Note, that this could fail and give an exception due the nature of the dependencies involved in ensuring longevity for certain class objects).

This vector contains the pointers of the objects that are created by the class constructors. It iterates through them and calls their `del()` functions in order to delete them.

```
void NTon::exit_handler()
{
     if(vec_obj.size() != 0)
     {
          for (
            vector<NTon*>::reverse_iterator i =
          vec_obj.rbegin();                              i !=
          vec_obj.rend(); ++i )
          {
               cout<<"\n\n";
               (*i)->del();
          }
     }
}
```

Once an object has been deleted by the client, the client is free to create more objects as long as the total count of alive objects does not exceed `N`.

Preserving Dependency Amongst classes in the hierarchy:

To demonstrate the existence of dependencies amongst various classes in the hierarchy, we show that objects of class B can only be deleted if all objects of class D are deleted.

```
void D::del()
{
     if ( B::get_count() == 0 )
     {
          count--;
          curr_num_of_objects--;

          auto it = find(vec_obj.begin(), vec_obj.end(), this);
          if(it != vec_obj.end())
          {
               assert((dynamic_cast<D*>(*it)) == this);
               delete( dynamic_cast<D*>(*it) );
               vec_obj.erase(it);
          }
     }
     else
     {
          throw std::runtime_error("Cannot delete objects of D
while objects of B exist");
     }
}
```

Conclusion:

In this mini project we have demonstrated an implementation of the NTon design pattern in a class hierarchy. Scope for future improvements include guaranteeing thread safety and providing guarantees amongst dependencies that exist between different NTon hierarchies.