Research Project

Is using Factory Pattern better than using Normal Inheritance?

Name: S M Ragib Rezwan

ID: 103172423

CONTENT:

Abstract:

A comparison between factory pattern and normal inheritance system, on programs written in C# language using OOP, in terms of design and performance.

Introduction:

According to definition, **design patterns** are "a general repeatable solution to a commonly occurring problem in software design". Basically, it's nothing more than a group of templates used to solve repeatedly occurring similar problems and thus speed up the development process of the software (via effective software design). Among these patterns, one of the most commonly used one is the Factory Pattern.

Factory Pattern (also known as factory method or virtual constructor) is a type of creational design pattern in which we define an interface or abstract class for making an object, but let the subclass decide what type of object will be made. This is beneficial in the certain situations, like not wanting client class to know how to instantiate an object, letting subclass determine type of object to be created, letting client class call another class to make new objects, etc.

But although it has such benefits, some programmers tend to avoid using it and instead favoring normal coding via inheritance as an alternative, seeing as they both give the same output.

Thus, in this report, I am going to compare and contrast between similar pieces of simple codes written both ways to try answer the question "Is using Factory Pattern better than using Inheritance?"

Setup and output for both cases: (Code and UML)

A) Normal/ Inheritance system (without factory pattern):

(I) Codes for setting it up:

1)Program.cs file:

```
using System;

namespace WithoutUsingFactoryPattern
{
  class Program
  {
    static void Main(string[] args)
    {
      //instantiating new objects
      Item gun = new Gun("A45", "revolver", "this is a good revolver");
```

```
            Item sheild = new Sheild("B23", "Electric Sheild", "this is an electric shield") ;

            //outputing details about the object made
            Console.WriteLine(gun.itemDetails());
            Console.WriteLine(sheild.itemDetails());
        }
    }
}
```

2) Item.cs file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WithoutUsingFactoryPattern
{
    public class Item           //Parent class for Gun and  sheild
    {
        private string _id;            //to store id of the item
        private string _name;          //to store name of the item
        private string _description;   //to store description of the item


        //parent class constructor takes in the id, name, desc to make the item
        public Item(string id, string name,
        string desc)
        {
            _id = id;
            _name = name;
            _description = desc;



        }

        //getter property for id so that it can be called in child class
        public string id
        {
            get { return _id; }

        }

        //getter property for name so that it can be called in child class
        public string name
        {
            get { return _name; }

        }

        //getter property for description so that it can be called in child class
        public string desc
        {
            get { return _description; }

        }

        //virtual method that will be overriden by child class
        public virtual string itemDetails()
        {
            string fd = "blah blah";
            return fd;
        }
    }
}
```

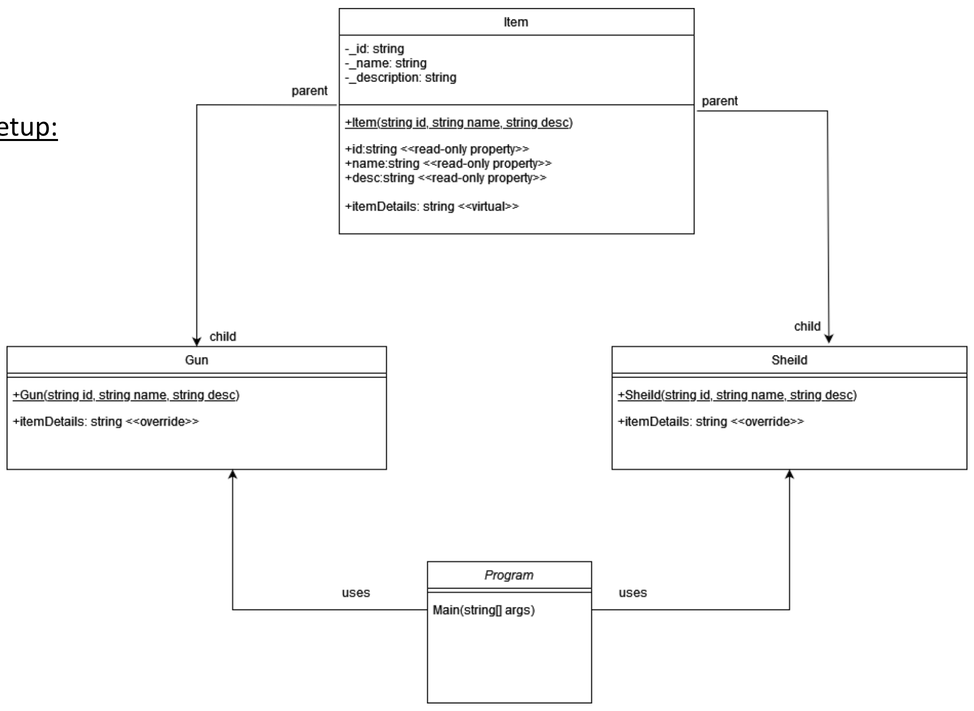
3) Gun.cs file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WithoutUsingFactoryPattern
{
    public class Gun : Item
    {
        //Child class constructor inherits id, name, desc from parent class constructor
        public Gun(string id, string name, string desc):base(id, name, desc)
        {
        }

        //overriding the method provided by parent class to fulfill its own purpose
        public override string itemDetails()
        {
            string fd = "This is a Gun\nit's id is " + base.id + " and its name is " + base.name +
"\nDetails about it: " + base.desc;
            return fd;
        }

    }
}
```

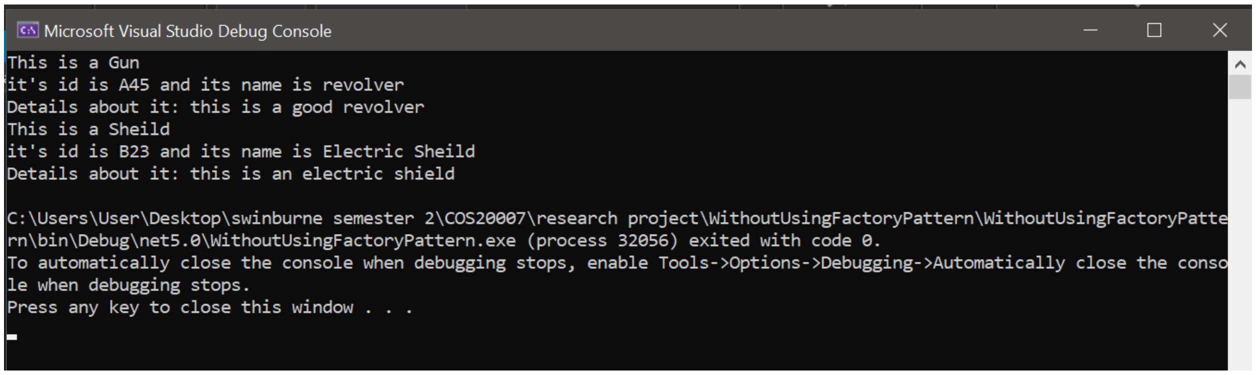
4) Sheild.cs file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WithoutUsingFactoryPattern
{
   public class Sheild : Item
   {
      //Child class constructor inherits id, name, desc from parent class constructor
      public Sheild(string id, string name, string desc) : base(id, name, desc)
      {
      }

      //overriding the method provided by parent class to fulfill its own purpose
      public override string itemDetails()
      {
         string fd = "This is a Sheild\nit's id is " + base.id + " and its name is " + base.name +
"\nDetails about it: " + base.desc;
         return fd;
      }

   }
}
```

(II) UML for the setup:


Item
-_id: string
-_name: string
-_description: string
+Item(string id, string name, string desc)
+id:string <<read-only property>>
+name:string <<read-only property>>
+desc:string <<read-only property>>
+itemDetails: string <<virtual>>
parent
parent
child
child
Gun
+Gun(string id, string name, string desc)
+itemDetails: string <<override>>
Sheild
+Sheild(string id, string name, string desc)
+itemDetails: string <<override>>
uses
uses
Program
Main(string[] args)

(III)Output for the code:



```
This is a Gun
it's id is A45 and its name is revolver
Details about it: this is a good revolver
This is a Sheild
it's id is B23 and its name is Electric Sheild
Details about it: this is an electric shield

C:\Users\User\Desktop\swinburne semester 2\COS20007\research project\WithoutUsingFactoryPattern\WithoutUsingFactoryPatte
rn\bin\Debug\net5.0\WithoutUsingFactoryPattern.exe (process 32056) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

B) Following Factory Pattern

(I)Codes for setting it up:

1) Program.cs

```
using System;

namespace UsingFactoryPattern
{
   class Program
   {
      static void Main(string[] args)
      {
         //instantiating new factory
         ItemFactory itemFactory = new ItemFactory();

         //Letting the factory create the two types of items
         IItem gun = itemFactory.createItem("Gun");
         IItem sheild = itemFactory.createItem("Sheild");

         //outputing details about the object made

         Console.WriteLine(gun.itemDetails());
         Console.WriteLine(sheild.itemDetails());

      }
   }
}
```

2) Sheild.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UsingFactoryPattern
{
    public class Sheild:IItem
    {
        private string _id;           //to store id of the sheild
        private string _name;         //to store name of the sheild
        private string _description;    //to store description of the sheild

        //Sheild takes in id, name, desc to make itself
        public Sheild( string id, string name, string desc)
        {
            _id = id;
            _name = name;
            _description = desc;
        }

        //implimenting the method in the interface
        public string itemDetails()
        {
            string fd = "This is a Sheild\nit's id is " + this._id + " and its name is " + this._name +
"\nDetails about it: " + this._description;
            return fd;
        }

    }
}
```

3) Gun.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UsingFactoryPattern
{
    public class Gun:IItem
    {
        private string _id;           //to store id of the gun
        private string _name;         //to store name of the gun
        private string _description;   //to store description of the gun

        //Gun takes in id, name, desc to make itself
        public Gun( string id, string name, string desc)
        {
            _id = id;
            _name = name;
            _description = desc;
        }

        //implimenting the method in the interface
        public string itemDetails()
        {
            string fd = "This is a Gun\nit's id is " + this._id + " and its name is " + this._name +
"\nDetails about it: " + this._description;
            return fd;
        }

    }
}
```

4) IItem.cs

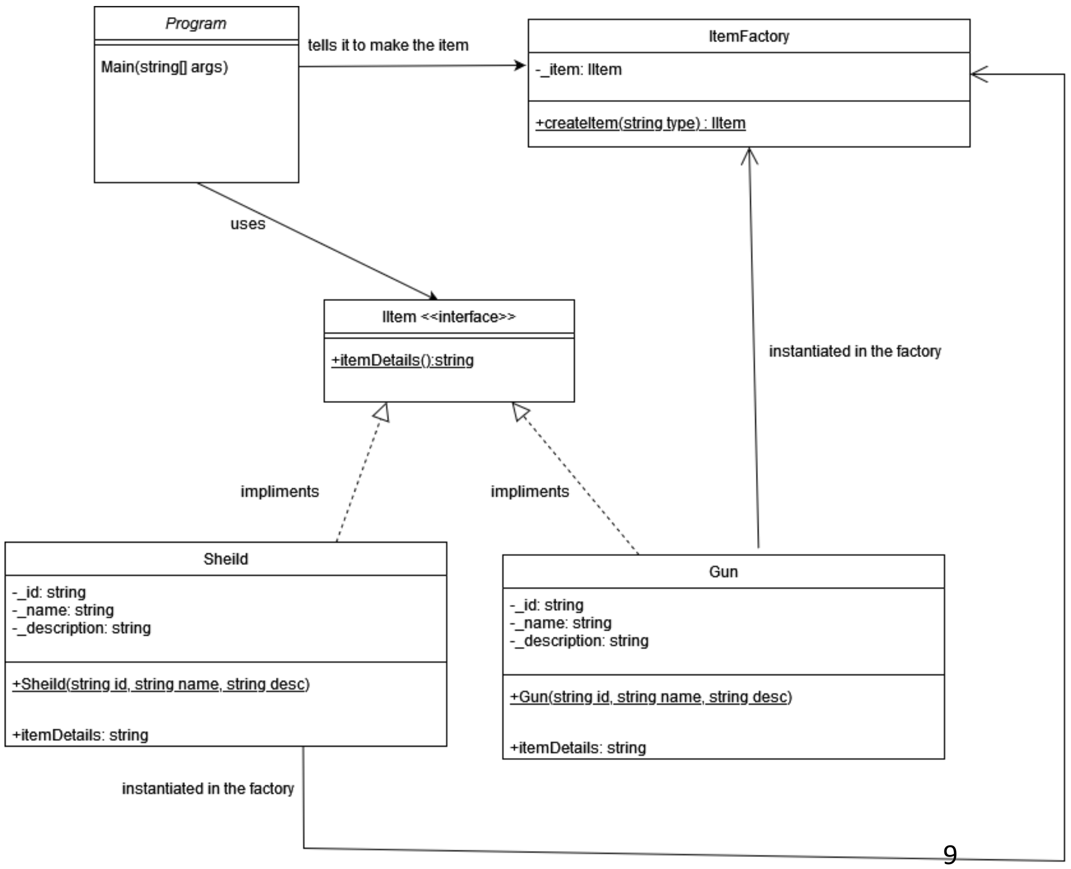
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UsingFactoryPattern
{
    //a common interface for items where all items need to have method itemDetails, with no
parameter, that will return a string type variable
    public interface IItem
    {
        string itemDetails();
    }
}
```

5) ItemFactory.cs

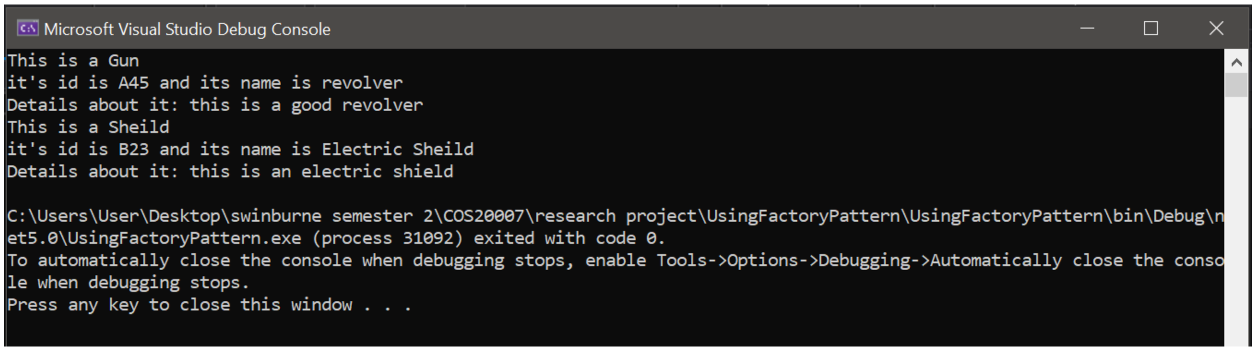
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace UsingFactoryPattern
{
   public class ItemFactory
   {
        private IItem _item; //utilising interface to instanciate the specific items
       public IItem createItem(string type)
       {
          _item = null;

          //if gun type then make a new gun object
          if (type == "Gun") {
              _item = new Gun("A45", "revolver", "this is a good revolver");
         }

          //if sheild type then make a new sheild object
          if (type == "Sheild")
          {
              _item = new Sheild("B23", "Electric Sheild", "this is an electric shield");
          }

          return _item; //return the created object
       }

   }
}
```

<u>(II)UML for the setup:</u>



(III)Output for code:



Microsoft Visual Studio Debug Console

```
This is a Gun
it's id is A45 and its name is revolver
Details about it: this is a good revolver
This is a Sheild
it's id is B23 and its name is Electric Sheild
Details about it: this is an electric shield

C:\Users\User\Desktop\swinburne semester 2\COS20007\research project\UsingFactoryPattern\UsingFactoryPattern\bin\Debug\n
et5.0\UsingFactoryPattern.exe (process 31092) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Since we can see that both codes are giving the exact same output for the setup, we can conclude the setup has been successful and thus start comparing between the two methods

1) Design or qualitative difference:

a) Code setup line + file count:

	Program (ie client)	Gun	Sheild	Item	IItem <<interface>>	ItemFactory	Cumulative Line Count
Normal Inheritance Coding style	18	24	24	56			122
Factory Pattern	23	31	31		14	31	130

So, with respect to line and file count, normal coding style using inheritance has the upper hand with less number of total lines of code and less number of classes (and thus files) needed for setup compared to that produced in factory pattern. Thus it’s faster for programmer to use the Normal coding style rather than factory pattern, at least in the case of small initial setup codes.

b) What Client knows/handles:

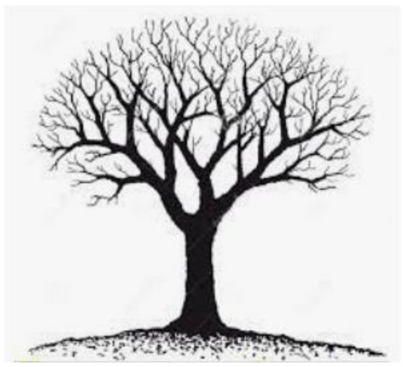
In the normal coding style, the client is the main or the program class. Here it has the responsibility of creating the items “Gun” and “Sheild” using the new keyword. So, it needs to call the constructor of the two classes, pass in the relevant parameters data and types, etc to instantiate them.

But in Factory pattern, the client (ie the main) is instead instantiating a factory class and delegating the task of creating the subclasses to it instead. Thus the client does not need to call the constructor of the subclasses or even keep track of parameters to pass to them. Instead it only needs to tell the factory what sort of item it needs (ie a gun or sheild).

Thus here we can hide the subclass' instantiating logic from the main/ client, promoting loose-coupling (as it is dependent on interface rather than a concrete class) and making it easier to isolate, test and even swap parts of subclasses without affecting the main. Furthermore it also fulfills the Dependency inversion principle as the higher class (ie the main) depends on abstraction of lower class (done by the interface IItem here) instead of depending on the lower class (ie gun or sheild) itself.

So, in this regards, Factory pattern is the better choice for the programmer.

c) Code Extensibility:



In normal code, we had used inheritance method to create a hierarchy for Gun and Sheild. If we wanted to add more items to it, like different brands of gun and shields, we would need to create another new hierarchy between the gun and different gun types and between shields and different shield types. Thus, if we keep adding more niche items (like specific version or model or a certain), we will end up producing a deep level of hierarchical chain, making our code unnecessarily complex.

But if we use Factory pattern, then we can avoid this unnecessary complication. Instead, we can just ensure that all the objects inherit from the same interface and thus directly call the constructor in the Factory to make them. This provides a centralized management of the objects and hence makes it easier to add, modify and remove objects.

2) Performance difference

a) RunTime

In order to compare the run times, I have taken 6 different reading, 1 for the initial, and next 5 for taking average of time it takes to run the code for the other times. The results I found were quite surprising to be honest.

	Initial runtime (ms)	2nd runtime (ms)	3rd runtime (ms)	4th runtime (ms)	5th runtime (ms)	6th runtime (ms)	Average runtime from 2nd to 6th
Normal Inheritance Coding style	376	145	191	193	186	190	181
Factory Pattern	233	201	207	220	235	205	213.6

Thus, for the first time the codes are run, the one following the factory pattern gives its output faster than normal one by 61.4%. But if we run the same code again and again (without making any changes), the normal one starts taking a shorter time now compared to factory pattern by 18%. Furthermore, if we look at the table overall, we can see that the factory pattern has little differences between its initial and the other runtimes, but for normal one, there is a significant different between those times.

Thus, although average runtime (after the initial one) is shorter for normal one, Factory pattern wins overall in this regards as it has more consistent runtimes and also while programming we will be more likely to alter codes and test ,instead of running the same code over and over again

<u>b) Processor's Memory usage and the overall Memory Usage:</u>

In both cases, the maximum memory occupied by the processor while running the code was 7MB.

For finding the overall Memory usage, I have utilized the debug option, gave the following result:

	Object	Heap size
Normal Inheritance Coding style	337	73.33 kb
Factory Pattern	341	75.30 kb

This shows that by running the code written in both ways, we get similar number of object in native memory and .Net and also that the amount of bytes used in .Net and the native heap is similar for both.

But if we look at it critically, we can see that Normal has defeated the Factory pattern in terms of memory usage, albeit by a small margin.

<u>Conclusion:</u>

Thus we can overall conclude that using Factory Pattern is indeed better than using Normal Inheritance coding style. But its necessity actually depends on size of the code the programmer needs to create. So, if the programmer is making a small code and instantiating a few objects, it is not really necessary for him or her to use factory pattern. But if he or she plans on extending the program, creating a big code with multiple levels of inheritance or instantiating multiple objects, it will be better for the person to rely on factory pattern instead. So, when considering whether or not to use Factory pattern, one should keep the words of the Chinese philosopher Confucius in mind,

"Don't use a cannon to kill a mosquito!"

Some Useful Concepts/ terms:

- **Loose-coupling** is when objects interact with each other but has limited knowledge about what the other object can or cannot do.
- **Dependency Inversion Principle (DIP)** states that high level modules should not depend on low level modules; both should depend on abstractions.

- The **Objects** display the number of objects in .NET and native memory when the snapshot was taken.
- The **Heap Size** displays the number of bytes in the .NET and native heaps

Reference Links:

https://sourcemaking.com/design_patterns

https://www.javatpoint.com/factory-method-design-pattern

https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

https://dzone.com/articles/what-is-loose-coupling

https://en.wikipedia.org/wiki/Dependency_inversion_principle

https://docs.microsoft.com/en-us/visualstudio/profiling/memory-usage?view=vs-2019

https://www.quotes.net/quote/1255