

Research Project

Is Factory Pattern with Composition, better than Inheritance?

Name: S M Ragib Rezwana

ID: 103172423

CONTENT:

- Abstract
- Introduction
- Method
- Result
- Discussion
- Conclusion
- Some useful concepts
- Setup codes for both cases
- Reference links

Abstract:

Design Patterns are basically groups of templates used to solve repeatedly occurring problems and speed up the development process of software. Although there are several types of patterns, they all basically focus on different ways to create a class, create an object, or determine interclass interaction and responsibility, depending on our need. And it is for these reasons that they are commonly used in different object oriented languages.

Among these design patterns, one of the most popular is using Factory Pattern with Composition, which is used as an alternative to inheritance in codes. But is it really better to use Factory Pattern with Composition instead of using Inheritance?

Introduction:

According to definition, **design patterns** are “a general repeatable solution to a commonly occurring problem in software design”. But that doesn’t mean that they solve the problem themselves. Instead it’s better to think of them as tools or blueprints that help us solve our problems. One of the most important benefits they have is the fact that they are general and thus code independent. So, developers coding different software in different languages can discuss, use and reuse patterns in different areas of their codes without much issue!



One such pattern is the Factory Pattern with Composition (also known as factory method or virtual constructor). It 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 instead. This is basically a mechanism in which classes are separated into parent and child classes where the parent classes contain fields and methods common to and inherited by the child classes. But the child classes have their own unique methods and fields in their own classes as well. So the objects created from the child class will have both its own unique aspects and also some common aspects from its parent. Thus it gives the benefit of maintaining high reusability of code, which in turn makes the code more reliable.

So, in both cases, we can see that, although both are working in different ways, the final output is the creation of object. Hence comes the obvious question, which way is better: The one performed by Factory Method using Composition? Or the one accomplished by Inheritance?

Method:

Here, I am going to make a similar, small, simple code for both Factory Pattern using Composition and inheritance to see how they compare against each other. I am doing this for two different reasons:

- a) I am going to find the minutest difference between them by using simple codes and thus extrapolate on these data to scale up the difference between them for large scale programs. (assuming that they behave similarly in both cases)
- b) I also want to ensure no other factors affect the codes other than whether it uses factory pattern with composition or inheritance. Thus, by using these simple short codes, I can control the test environment and ensure it stays constant.

To ensure that the outputs for both cases are same, I am making the following 2 programs:

[Note: I am not giving code snippets here. So if you wish to see the main codes, please scroll down to the part between useful concepts and reference]

[Note: in both cases, all fields are made private and methods are made public here and thus property is needed to call them in different classes]

i) **For normal Inheritance case:**

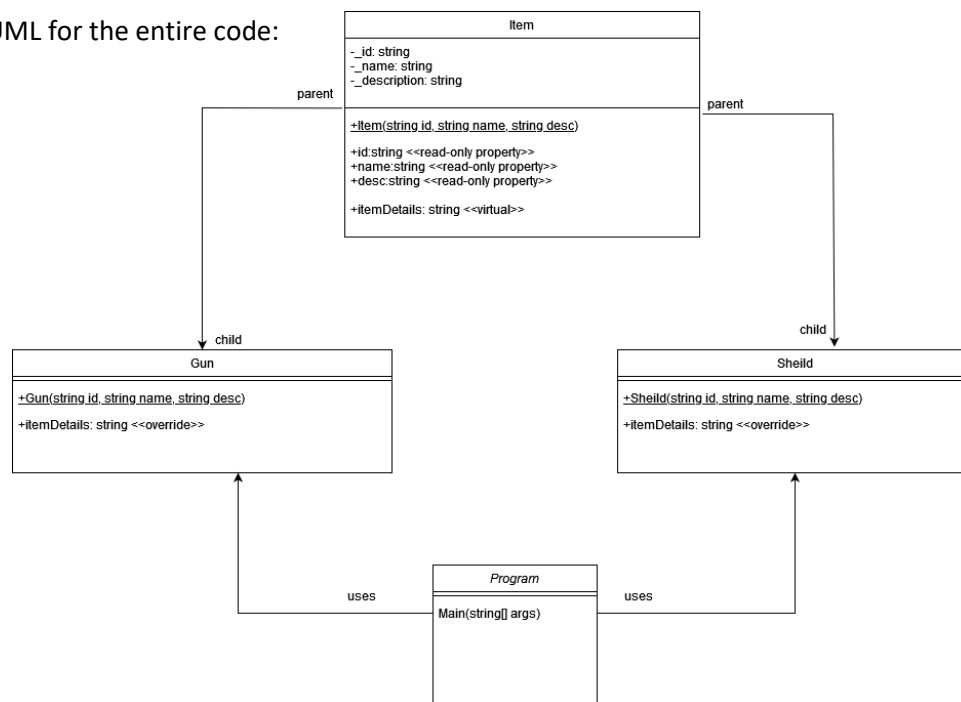
Here I am utilizing 4 files: Program, Item, Gun, Sheild.

In Program class, I am instantiating the new Gun and Sheild objects and passing the result of their methods to console writeline. This will be used to see output of methods on the console.

In Item class, I am using it as a parent class. Thus I am creating the fields for id, name and description alongside a constructor to set these. Furthermore there are properties made for these fields as well to call them in child classes later on. Also, there is virtual method called item details which will be overridden by the child classes.

In Gun and Shield class, I am using them as child classes. They will use their parent's constructor to make themselves, setting their id, name and desc, and call them from their parent class (using the property set before) and then use it to call their own, overridden ItemDetail method.

The UML for the entire code:



ii) For Factory Pattern using Composition

Here I am utilizing 5 files: Program, Item, ItemFactory, Gun, Sheild.

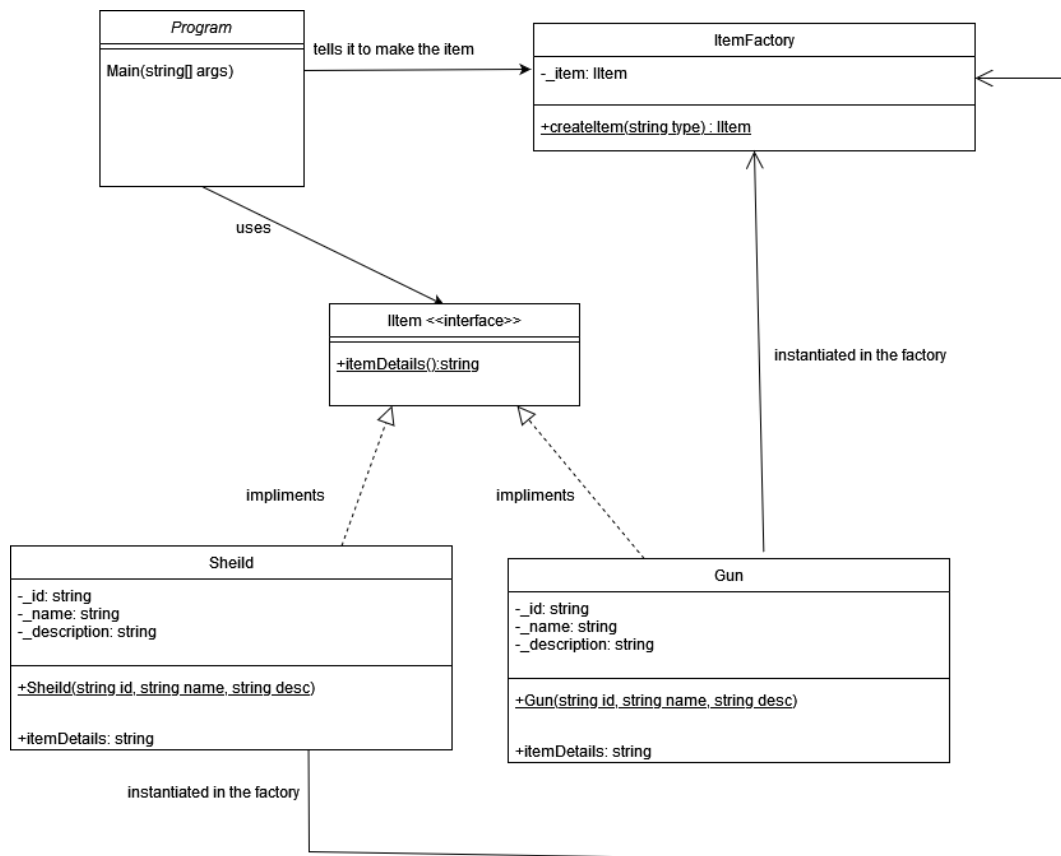
In Program class, I am instantiating a new itemFactory, then making gun and shields as Items by passing strings to itemfactory's create item method. After this, I am passing the result of their own methods to console writeline. This will be used to see output of methods on the console.

In Item class, I am using it as an interface. This will ensure all classes using this interface will have a method called ItemDetails, with no parameters, returning a string output.

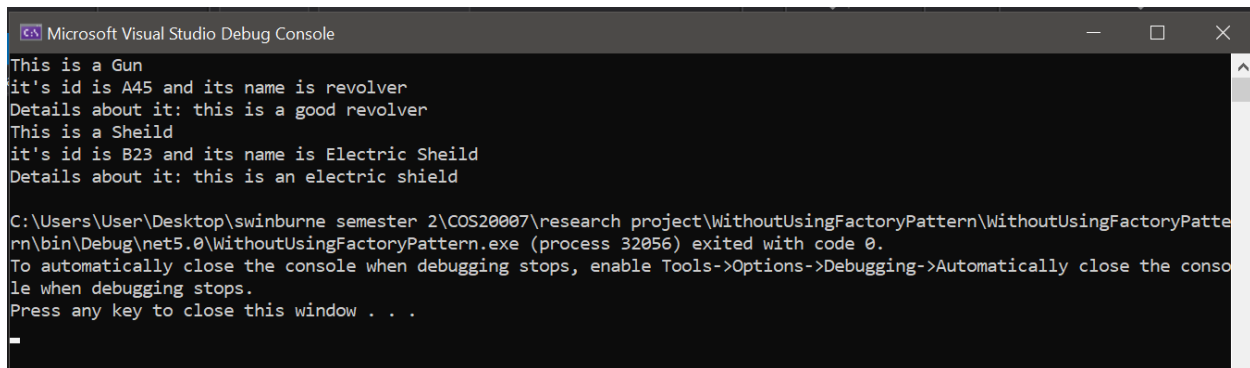
In ItemFactory, I am making a field for Item datatype and making a method that will take a string and use it to decide whether to instantiate a gun or shield, returning that as an Item

In Gun and Shield class, I am creating fields for id, name and description and using them in ItemDetails to output their own personalized strings. They must have ItemDetails as they are using the Item interface.

The UML for the code is:



After coding these, I ensured that the output is same by running both codes setups:

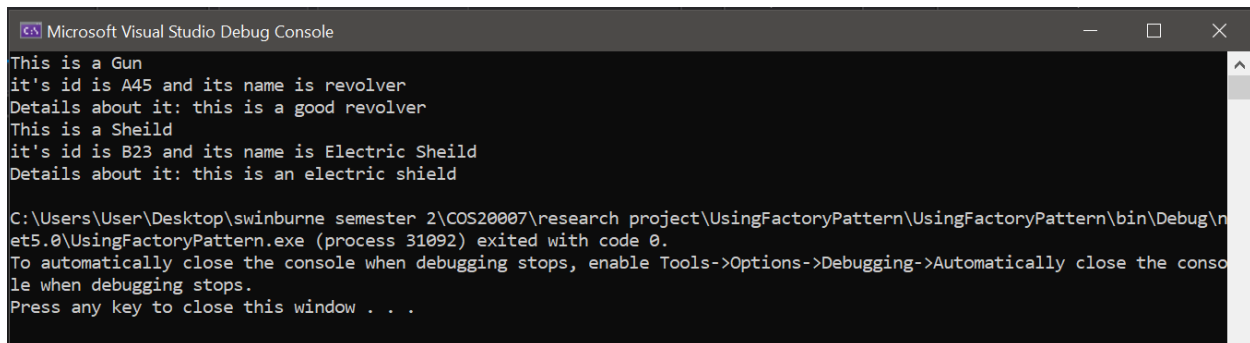


```
Microsoft Visual Studio Debug Console

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\WithoutUsingFactoryPattern\WithoutUsingFactoryPattern\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 console when debugging stops.
Press any key to close this window . . .
```

[output for normal inheritance, without factory pattern]



```
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\net5.0\UsingFactoryPattern.exe (process 31092) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

[output for using factory pattern using composition]

Here I am going to compare the two codes in terms of design and performance matrixes:

- For design matrixes, I will look at setup File and Line Count, information client handles, and code extensibility. As these are qualitative, I will not use any specific methods to measure these. Instead I will just observe the codes written and state what is and is not present.
- For Performance matrixes, I will look at Run Time and Processor usage and overall memory usage. As these are quantitative, I will utilize the debugging tools given in VS Community to insert breakpoints and make the observations.

Result:**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

b) What Client knows/handles:

In the normal coding style, the client is the main or the program class. Here we can see that it needs to know how to create instances of the Gun and Sheild as it is using the keyword new and calling their constructors

```

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") ;
}

```

But in Factory pattern, the client (ie the main) is instead instantiating a factory class and using methods of the factory to make the Gun and Sheild.

```

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");
}

```

c) Code Extensibility:

In the both codes, you can see two instances of the desired object. In Inheritance class, it is the two child class and in factory with composition, it is the two classes it instantiates in its itemFactory. Thus by using these we can notice the followings:

- a) In case of inheritance, for every new item we add, we will need to add it as a child class of the parent, keeping features common with the parent. So, it won't need to create

fields for ids, names and desc. But it must need to use a constructor that uses those parameter and pass to parent and override a method called itemDetails().

- b) In case of Factory with Composition, for every new item we add, we will need to provide all the needed fields and it must have method called itemDetails(). Also, its constructor must have needed codes to make instances of itself.

2) Performance difference

a) Run Time

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

Here the Run times came very weird as at first Factory pattern beat Inheritance. But in concurrent runs, although factory pattern remained same, inheritance code had suddenly sped up, which does not make sense. Thus I repeated it again, removing the external factors that can affect it (like explicitly running garbage cleaner, turning off other softwares, etc), noting down the results I had obtained:

Using Factory Method			
1 st time run (after boot)	295	212	232
2 nd time run	189	193	199
3 rd time run	193	174	178
4 th time run	204	189	194
5 th time run	178	193	185

Inheritance Method			
1 st time run (after boot)	344	333	336
2 nd time run	194	201	199
3 rd time run	203	212	205
4 th time run	189	192	194
5 th time run	195	194	202

But these showed that for both cases, it took a long time just after bootup and then sped up afterwards for some reason, with both having similar data during those times. Thus I ran further trials, noting the times taken to just run the different parts in both codes, which gave the following times:

Using Factory Method			
1 st time run (after boot)	64	29	45
2 nd time run	17	16	18
3 rd time run	19	15	16
4 th time run	18	14	18
5 th time run	16	17	16

Using Inheritance Method			
1 st time run (after boot)	32	40	48
2 nd time run	23	22	22
3 rd time run	25	23	25
4 th time run	21	21	22
5 th time run	22	24	21

[Note: Garbage collector had been added later on to check the runtime accuracy. Thus the data obtained in the last 4 tables in this section is that taken after implementation of that extra line of code. But the all the other parts use the code before the introduction of GC.Collect() command]

b) Processor's Memory usage and the overall Memory Usage:

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

Discussion:

- Comparing the line and file count for the setup of the small code, we can see that the design that uses Inheritance has both smaller number of lines in total and also uses less number of files, compared to factory pattern with composition. This means programmers need to type less lines of code and need to make less number of files to obtain the same output. Thus in cases where one needs to make such small code, it will be slightly faster to do so using Inheritance instead of using Factory pattern with composition.
- In terms of information known by client, we can see that in Inheritance, the client needs to instantiate the classes. This means that the client has to handle the responsibility for creating the objects. While this isn't an issue in small codes, this might cause problems in larger codes in terms of cohesion as the responsibility to create the objects may not be strongly related to the other responsibilities it may have. Furthermore, in such larger codes, the class may not be "lazy enough" when having this extra responsibility.

But in the case of Factory with composition, the client (ie the main) is delegating the task of creating the subclasses to itemFactory instead. Thus the client does not need to call the constructor of the subclasses or even keep track of what parameters to pass to them. Instead it only needs to tell the factory what sort of item it needs (ie a gun or shield) as a string to its method. While this may not have much impact in smaller codes, in larger codes, this separation helps properly distribute logic among the classes and ensures cohesion in the client class. Thus it is better use Factory Pattern with composition in this regard.

- In term of code extensibility, in inheritance, we need to maintain the hierarchy already created. Thus if want to 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 add more niche objects (like specific version or model of a certain gun or sheild), we will end up producing a deep level of hierarchical chain, making our code unnecessarily complex as we scale up. Furthermore, in all cases we would need to ensure that the object created uses id, name and desc and that it doesn't end up limiting/restricting the parent class in anyway.



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. Furthermore, it also promotes loose-coupling (as it is dependent on interface rather than a concrete class), making it easier to isolate, test and even swap parts of those classes (like removing id, name and placing something else instead) without affecting the main. Thus it is better to use Factory pattern with composition in this regard.

- Comparing Run Times (in the first table in that section), we had already noticed the weird data that had come up. After all, when the codes were run the first time, the one following the factory pattern gave its output faster than normal one by 61.4% (according to the data). This had been normal and it had fit with our common perception of factory design pattern being better. But when the same codes were re-run consecutively, we saw that while factory pattern remained consistent throughout, inheritance code kept taking far less times than before! Even if we call off the 2nd runtime as a fluke, the other run times still remains lower than Factory Pattern, beating it consistently.

While it's true that these were measured in milliseconds (ms) and fluctuations can happen, such change seemed extremely weird to me back then. Thus, I reran the experiment (as seen in the

following two tables) removing the other factors that could have been affecting it, like removing all other background processes by turning off laptop and restarting, making sure Garbage cleaner was working properly by adding `GC.Collect()` to call it directly after main code's ending line, using breakpoint to isolate the codes, etc.

But even there we can see that it had taken a long time just after bootup, but then it still sped up drastically for all cases, producing similar data. This inconsistency, made it hard to draw any conclusion from it, unless one considered the boot up run time only, as all other external factors are guaranteed to not be present then (as no other process or file is running at that time and memory is completely fresh)

Even so, it didn't feel right that in some cases factory was slower and in some cases it was faster than inheritance method. This led me to rerun the experiments a third time, focusing on the time taken to run the main different parts in those codes (as seen in the final 2 tables in run time section).

There we can see that for the 1st time it was run after boot time, the codes inside main took comparatively shorter time in inheritance method than factory one. But, for consecutive runs, we can see factory method actually took slightly less time to execute its codes compared to Inheritance method, which fits with our common thoughts. But even so, since data isn't pointing to a single style in all cases and instead is close to one another, it is better to say that "run time is similar to both" and that more accurate testing and isolation condition is needed to check it more precisely and see if there is any other factor affecting these.

After reaching this stage, I had run out of reasons that can cause this issue. So I looked up online to see what other factors could be causing the issue, leading me to come across an article in stack overflow asking a similar question. There Shang Zhou explicitly measured the run time in his code, noticing different results each time. In response to that everyone had stated that variations in runtime are actually normal in any OS.

Seeing this, I looked up related information regarding runtime on the Swin library, finding the paper "Hardware trace reconstruction of runtime compiled code" (by Sharma D. S. et al.,2018) where they noticed that "interrupt-driven debugging can cause unintended latency in execution" and used a new mechanism called FlowJit where they found that "[their] experiments demonstrate that with a very low overhead of 1.1 to 2.8 μ s, runtime code reconstruction using hardware-only tracing is indeed possible".

Since they verified their efficiency on Intel Processor Trace (which is what my laptop runs on), I hypothesize that by using it, the overhead in the run times can be reduced significantly and thus provide a valid comparison between factory pattern with composition and inheritance, regarding small code's run time.

[Note: this overhead is already quite small in modern Intel Processors. So if we use large enough codes for both Inheritance and Factory pattern with composition, the error percentage caused by this overhead in runtime will be so small that it could be considered negligible. Thus the two codes can be accurate compared in those cases.]

- In case of processor memory and overall memory usages, the exact same result has been repeatedly obtained proving its validity. Here we can see that by running both ways we get similar no of objects in native memory and .Net, and also in Heap Size. This is already expected as it's a small code itself and not creating too many items. Thus here we need to compare the data critically to notice the difference. This results in inheritance being better as it uses 4 less objects and its Heap Size is 2.17kb smaller than that used by Factory Pattern with composition. But it has to be kept in mind that this result has been obtained for the given small code and thus result may not be the same for large codes as well.

Conclusion:

Thus overall, if we focus on this small sample code specifically and not think about extending the code in any way for future purposes, then we can see that it is better to use Inheritance, albeit very slightly.

But is this also the case for large programs?

In terms of design aspects: not at all. Instead it would be better to use Factory pattern following composition instead as it provides far more benefits than inheritance in terms of low coupling, high cohesion, and central management, etc.

In terms of performance: not certain. This is because:

- a) Test for run time could not be guaranteed as too much fluctuation and too similar times (as the programs themselves are very small)
- b) Processor memory and overall memory usage showed a slight benefit towards inheritance side, but since there is only one small sample of code, it cannot be guaranteed that similar thing will be seen in case of all other codes written in these two ways.

Thus, it will be better to repeat this experiment on a far larger sample of code (to reduce the impacts of the error caused here in runtime due to overhead) written in both ways, to contrast and observe. It is only then can we find an accurate, valid and clear-cut answer to the question, "Is Factory Pattern with Composition, better than Inheritance?"

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.
- 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

A) Normal/ Inheritance system (without factory pattern):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 overridden 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;
    }

```

B) Following Factory Pattern

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
            Item gun = itemFactory.createItem("Gun");
            Item 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:Item
    {
        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 instantiate 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
    }
}
```

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://stackoverflow.com/questions/54271755/why-the-execution-time-of-the-same-code-on-the-same-computer-could-be-different/54271796>

<https://onlinelibrary-wiley-com.ezproxy.lib.swin.edu.au/doi/full/10.1002/spe.2567>