Factory Pattern

OOPs A PIE

Stands for:

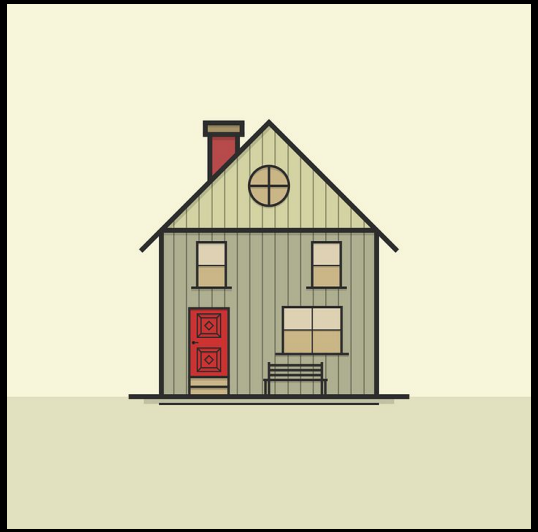
A - abstraction

P - polymorphism

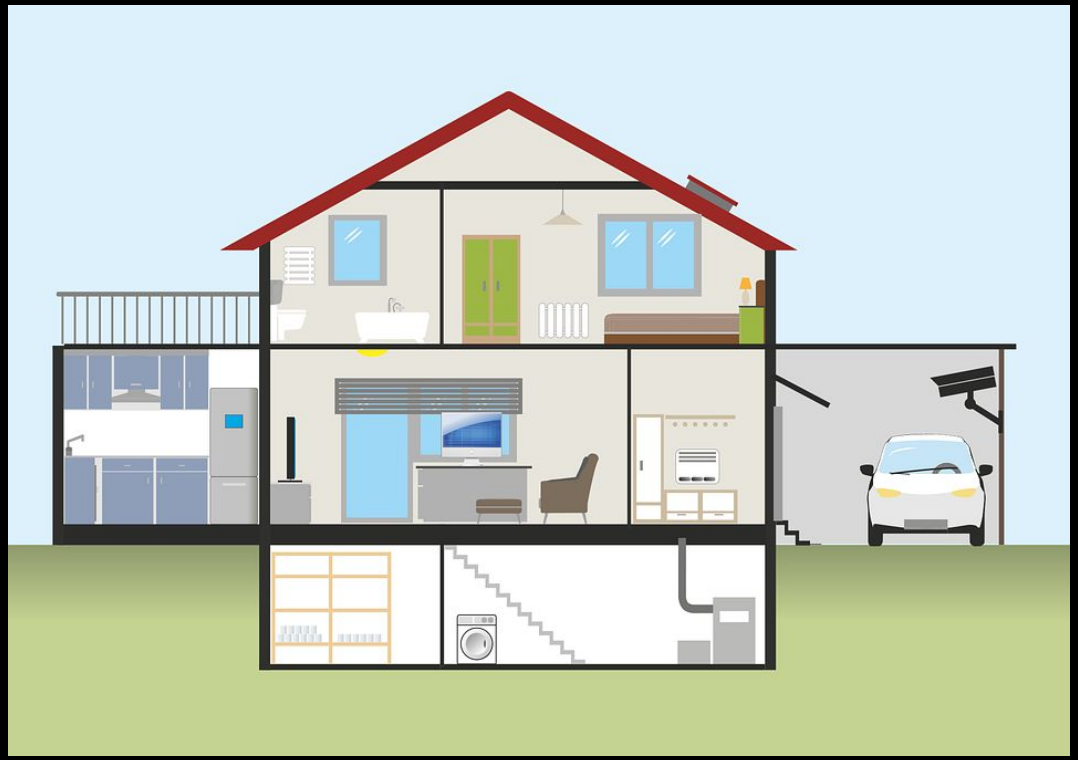
I - inheritance

E - encapsulation

Factory Pattern is a Design Pattern

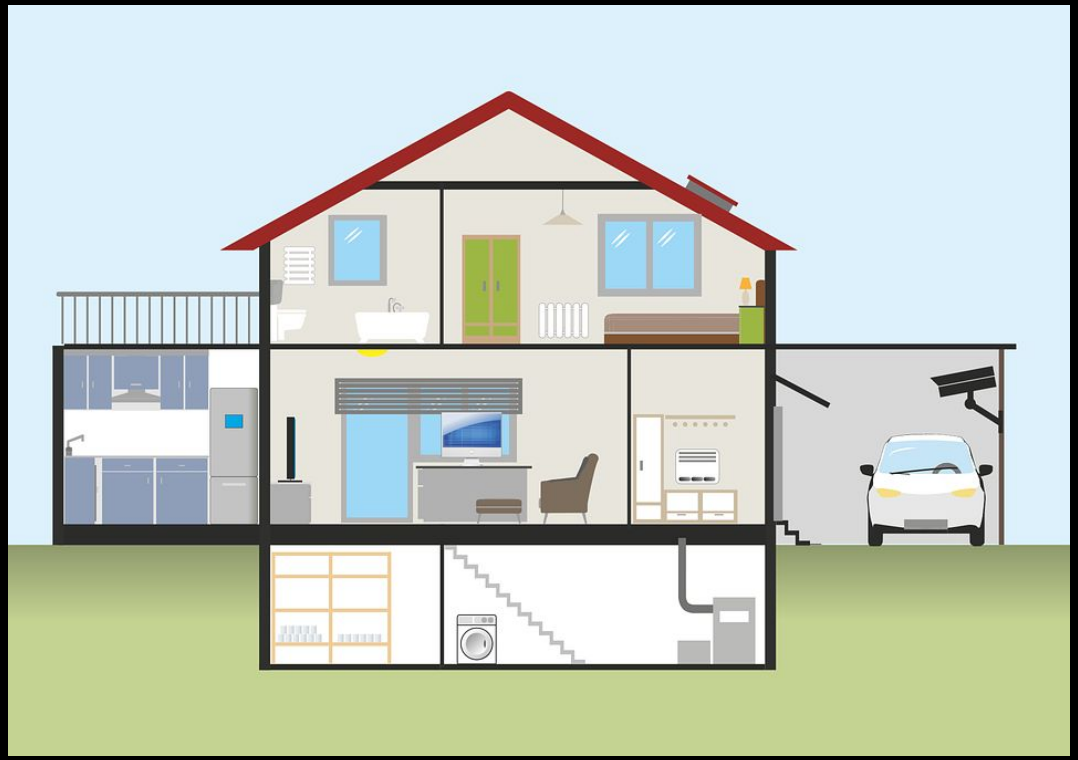


Just like houses have design patterns, programs have patterns too

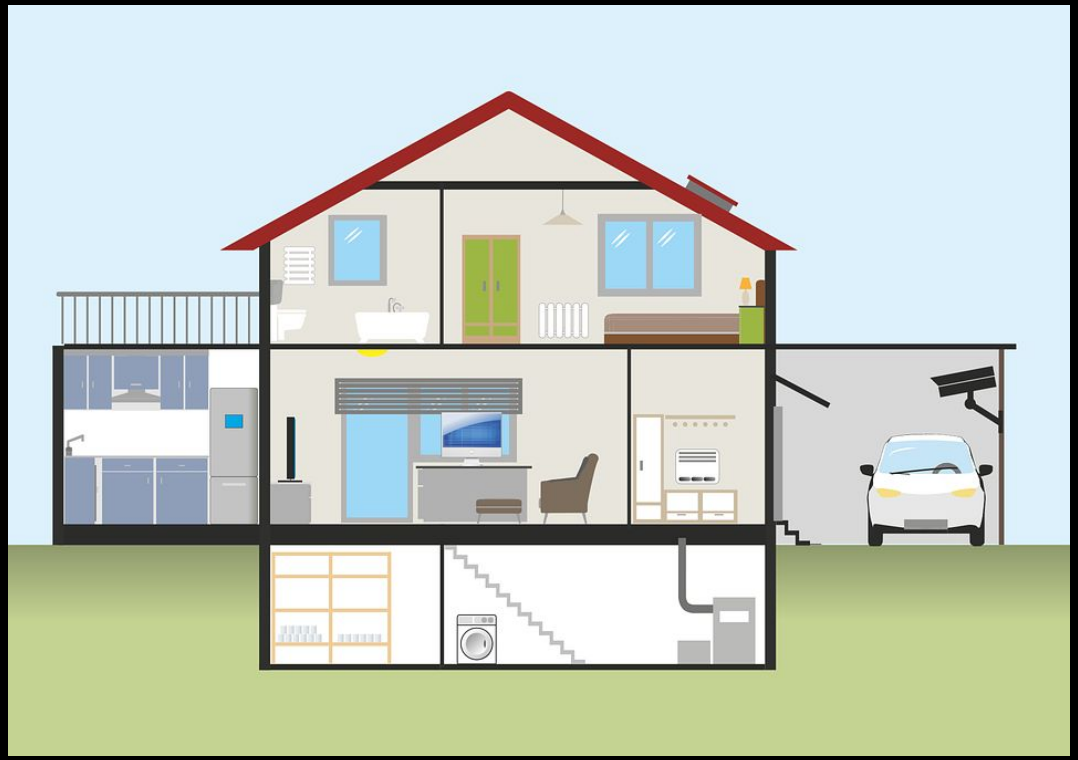


Pick Design Pattern based on your needs

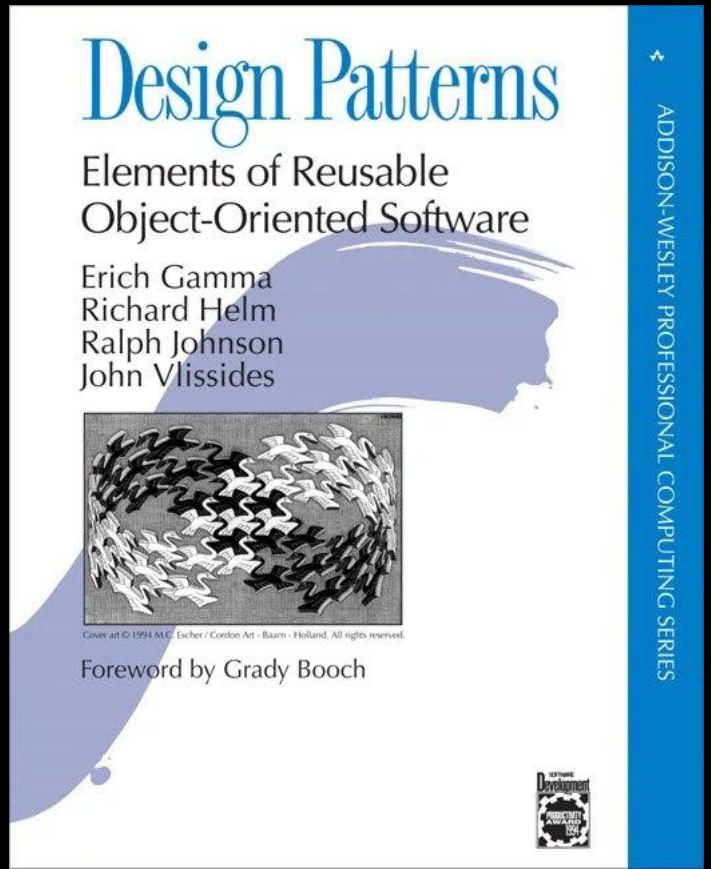
Ex. Small lot, need stairs for more room



Pick a Design for your Coding Needs



Factory Design Pattern is one of the 23 design patterns developed by the Gang of Four.



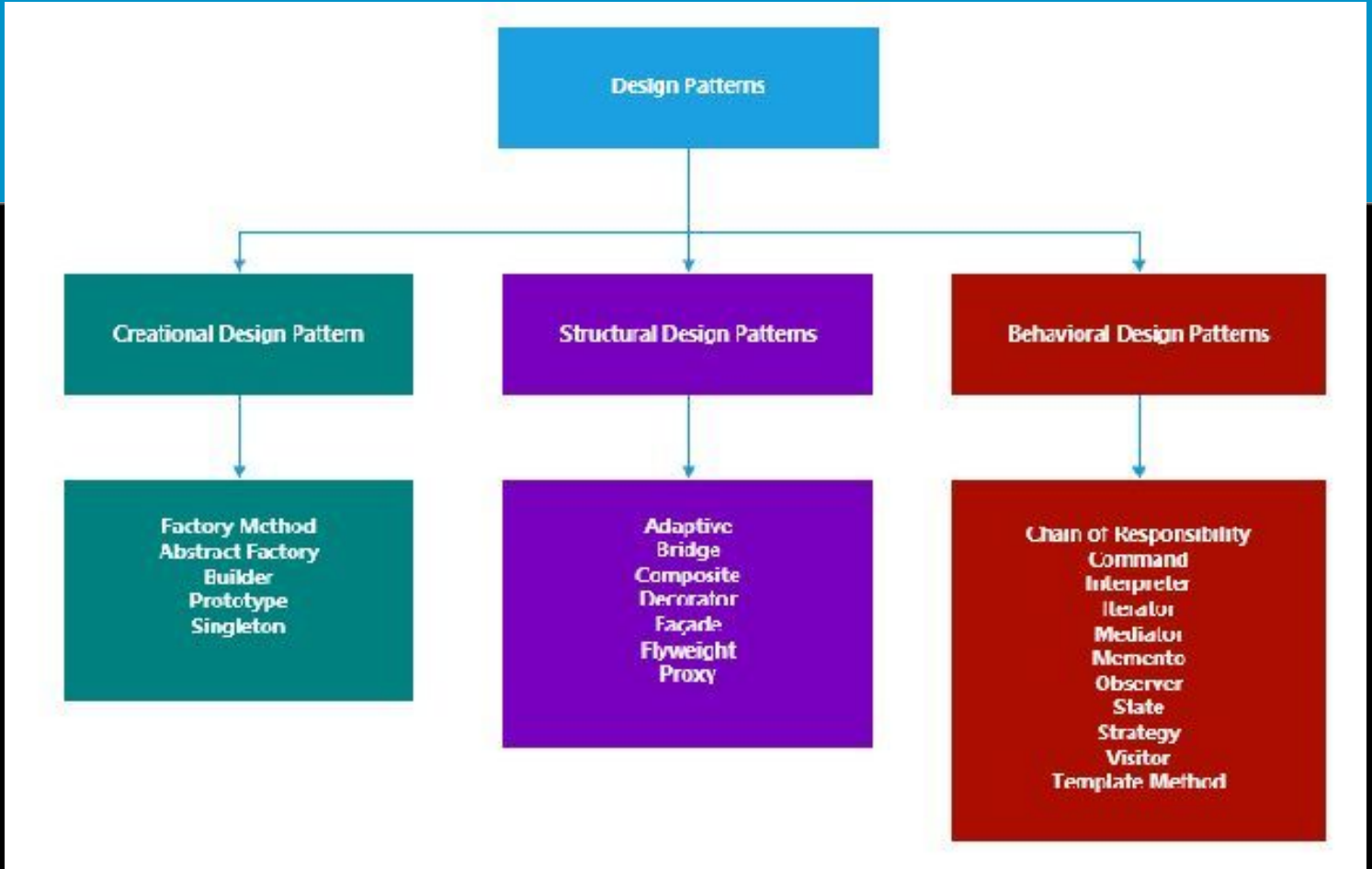
Types of Patterns

1. Creational Design Patterns
2. Structural Design Patterns
3. Behavioral Design Patterns

Types of Patterns

1. Creational Design Patterns
2. Structural Design Patterns
3. Behavioral Design Patterns

Types of Patterns



Factory

Factory

Place where you make something

Factory Pattern is a Pattern to Create Objects

Factory Pattern is a Pattern to Create Instances

Factory Pattern - Why we use it

- Keep your code DRY!

Dry - Don’t Repeat Yourself

Factory Pattern - Why we use it

- When you repeat yourself in code you are creating **brittle code** or “easily broken code”.

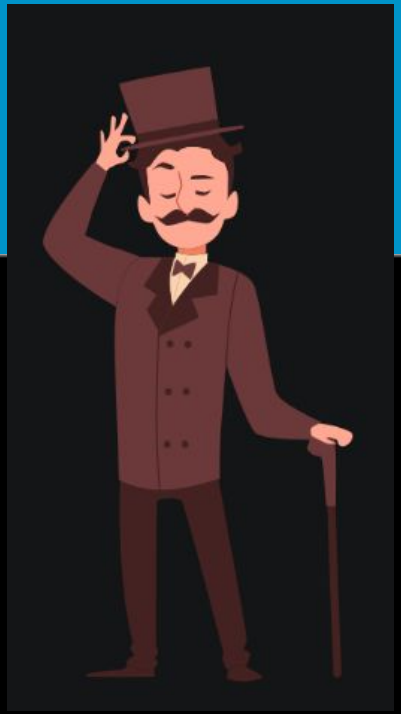
Factory Pattern - Why we use it

- **Centralized Object Creation: Factory Pattern prevents having to make numerous changes across your application**, which is not only a bad practice but will cause many headaches down the line.

Factory Pattern - Why we use it

- **Not repeating yourself also makes your code more readable;** in 6 months or a year you will be able to come right back into the code and see what is happening very easily.

Factory Pattern - Why we use it

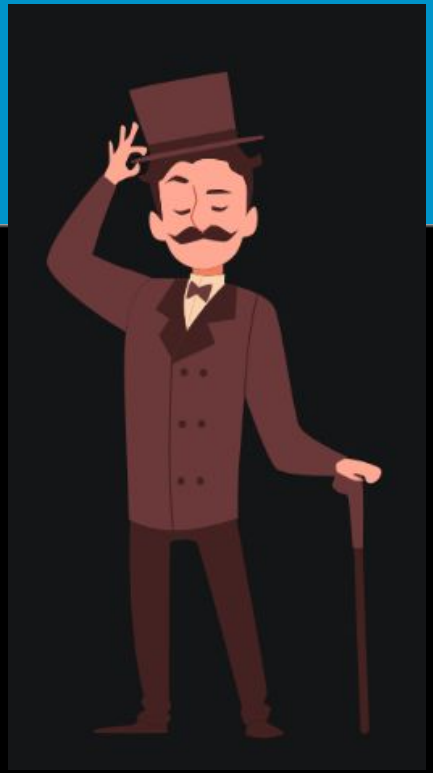


- **Centralized Object Creation**
- **Dynamic Instantiation**
- **Allows our program to be “loosely coupled”**

See link:

oop - What is the difference between loose coupling and tight coupling in the object oriented paradigm? - Stack Overflow

Loosely Coupled



In the image, the hat and the body are connected in a way that allows for easy separation. This represents **"loose coupling," meaning the hat can be removed without affecting the person**. This concept is explained further below.

Consider your skin, which adheres closely to your body, almost like a second glove. Now, imagine wanting to alter your skin color. The process of removing, dyeing, and reattaching the skin illustrates the challenges due to its tight integration with the body. **This is an example of "tight coupling," where modifications are complex and require significant alterations to the underlying structure.**

Factory Pattern - Why we use it

Bottom line: Provides **flexibility, readability,** and it provides easily **maintainable** code

Scenario

You're building an application that needs to connect to multiple types of databases like SQL Server, Oracle, MySQL, or others. The connection for each database differs, but we don’t want to make major changes to the core logic.

If you were to hard-code database connection everywhere in your application, it would become tightly coupled to that specific database.

Solution

Factory Pattern!

- Create a factory!
- At runtime, the correct database connection logic will execute.
- The core logic will remain independent.

Factory Pattern - Concept

The Factory Design Pattern - allows one class to create new objects.

Factory Pattern - Concept

Encapsulate the object creation process → you have one location to make changes to the way objects are instantiated.

Factory Pattern - Concept

- Can accomplish Factory Pattern using:
  - 1. Interface

    OR

  - 2. Abstract Class

Factory Pattern - Concept

- Can accomplish Factory Pattern using:
  - 1. Interface

    OR

  - 2. Abstract Class

We will use an interface!

Factory Pattern - Use Case

Making Phones - User chooses which one they want

Apple Android Samsung “Vintage”

```
Console.WriteLine("What kind of phone do you want to create?");
string userPhone = Console.ReadLine();

if (userPhone.ToLower() == "android")
{
    AndroidPhone android = new AndroidPhone();
    android.Build();
    Console.ReadLine();
}
else if (userPhone.ToLower() == "apple")
{
    ApplePhone apple = new ApplePhone();
    apple.Build();
    Console.ReadLine();
}
else if (userPhone.ToLower() == "iphone")
{
    ApplePhone apple = new ApplePhone();
    apple.Build();
    Console.ReadLine();
}
else if (userPhone.ToLower() == "google")
{
    GooglePhone google = new GooglePhone();
    google.Build();
    Console.ReadLine();
}
else
{
    AndroidPhone android = new AndroidPhone();
    android.Build();
    Console.ReadLine();
}
```

```
Console.WriteLine("What kind of phone do you want to create?");
string userPhone = Console.ReadLine();

if (userPhone.ToLower() == "android")
{
    AndroidPhone android = new AndroidPhone();
    android.Build();
    Console.ReadLine();
}
else if (userPhone.ToLower() == "apple")
{
    ApplePhone apple = new ApplePhone();
    apple.Build();
    Console.ReadLine();
}
else if (userPhone.ToLower() == "iphone")
{
    ApplePhone apple = new ApplePhone();
    apple.Build();
    Console.ReadLine();
}
else if (userPhone.ToLower() == "google")
{
    GooglePhone google = new GooglePhone();
    google.Build();
    Console.ReadLine();
}
else
{
    AndroidPhone android = new AndroidPhone();
    android.Build();
    Console.ReadLine();
}
```

Factory Pattern - Use Case

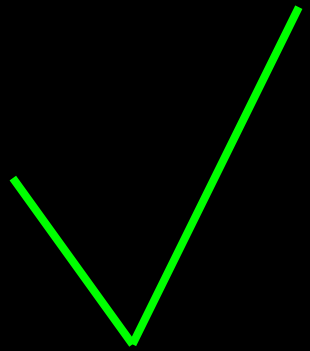
Anytime we need to make a change to the way apple phones are created we will need to make that change multiple times. That is NOT an efficient coding practice!

Apple

Android

Samsung

“Vintage”



```
//Ask the user for the phone they wish to create
Console.WriteLine("What kind of phone do you want to create?");
string userPhone = Console.ReadLine();

//Choose the correct type of phone to create through the factory method that implements the ICallable interface
ICallable phone = PhoneFactory.GetPhone(userPhone);
phone.Build();
Console.ReadLine();
```

All comes down to methods

```
//Ask the user for the phone they wish to create
Console.WriteLine("What kind of phone do you want to create?");
string userPhone = Console.ReadLine();

//Choose the correct type of phone to create through the factory method that implements the ICallable interface
ICallable phone = PhoneFactory.GetPhone(userPhone);
phone.Build();
Console.ReadLine();
```

Factory Pattern - How to get started

1. Create interface

```
5 references
interface ICallable
{
    4 references
    void Build();
}
```

Factory Pattern - How to get started

2. Create classes that conform to interface

```
2 references
class AndroidPhone : ICallable
{
    4 references
    public void Build()
    {
        ConsoleLogging.PhoneBuildDialogue();
        Console.WriteLine("Building an Android phone!");
    }
}
```

Factory Pattern - How to get started

3. Create Factory

```
/// <summary>
/// Creates different types of phone objects
/// </summary>
1 reference
static class PhoneFactory
{
    /// <summary>
    /// Takes the users input and returns a phone type that conforms to the ICallable interface
    /// </summary>
    /// <param name="phoneType"></param>
    /// <returns>
    /// ICallable
    /// </returns>
    1 reference
    public static ICallable GetPhone(string phoneType)
    {
        switch (phoneType.ToLower())
        {
            case "android":
                return new AndroidPhone();
            case "apple":
                return new ApplePhone();
            case "iphone":
                return new ApplePhone();
            case "google":
                return new GooglePhone();
            default:
                return new AndroidPhone();
        }
    }
}
```

Takeaways:

- The Factory Design Pattern is a programming concept that allows one class, separate to the main program or “client”, to create new objects.

- By using the factory pattern to encapsulate object creation, you have one location to make changes to the way objects are instantiated

3 Steps to Start

1. Create interface
2. Create Classes that conform to interface
3. Create Factory