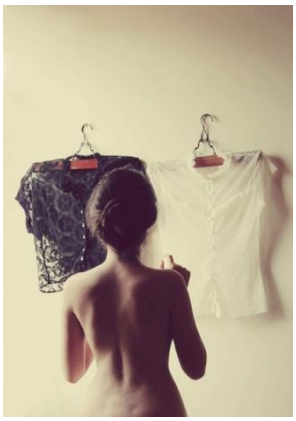
3. Фабричний Метод (Factory Method)



Уявіть, що ваша аплікація є дуже складною, і так склалося, що ви використовуєте два логінг-провайдери: один *Log4Net* та інший — *Enterprise.Logging*. Ваш колега додумався помістити вибір провайдера прямо у конфігураційний файл. Так як всю логіку логування ви абстрагуєте за інтерфейсом *ILogger*, то вам не хотілося б, щоб при потребі логгера вам приходилося по умові перевіряти, що записано у конфізі і тоді створювати необхідний екземпляр. Мабуть, було б добре приховати специфіку створення конкретного провайдера та винести її в окремий клас. Скажімо в *Фабричний Метод*.

Фабричний Метод вирішує, яку реалізацію інстанціювати. Вирішують або нащадки фабричного методу, або він сам, приймаючи якийсь параметр.[15]

Як на мене, то цей дизайн-патерн є одним із найбільш відомих і найпростіших. Я переконаний, що більшість читачів бачили його багато разів. Завдання *Фабричного Методу* полягає в прихованні конкретного класу, що має бути створений та повернений під виглядом спільної абстракції. Якщо в метод передаються параметри, від яких залежить, який клас буде створено, то такий *Фабричний Метод* називають *Параметризованим Фабричним Методом*.

У нашому прикладі класами, що мають бути створені, є *Log4NetLogger* та *EnterpriseLogger*, які імплементують *ILogger*, який широко використовується у нас в аплікації.

Уривок коду 3.1. Інтерфейс *ILogger* та одна із його реалізацій

```
interface ILogger
{
    void LogMessage(string message);
    void LogError(string message);
    void LogVerboseInformation(string message);
}
class Log4NetLogger : ILogger
{
    public void LogMessage(string message)
    {
    Console.WriteLine(string.Format("{0}: {1}", "Log4Net", message));
    }
    // Інші методи не наводимо
```

[15] **Factory Method**. Intent. Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Фабричний метод. Призначення. Визначити інтерфейс для створення об'єкта, але надати підкласам вирішувати який клас інстанціювати. Фабричний метод відділеговує інстанцювання своїм підкласам.

Як можна здогадуватися, може статися так, що в майбутньому нам знадобиться додати ще декілька провайдерів логування (буває і таке). Як ми уже згадували, рішення приймається на основі того, що є у файлі конфігурації. Щоб не показувати, який клас ми створюємо, ми делегуємо цю роботу до *LoggerProviderFactory*. І ось як фабрика справляється із цим:

Уривок коду 3.2. Реалізація Фабричного Методу

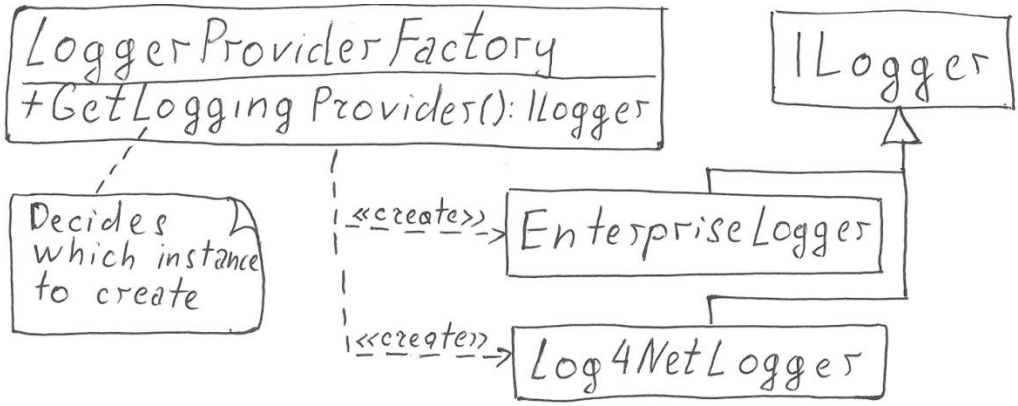
```
class LoggerProviderFactory
{
    public static ILogger GetLoggingProvider(LoggingProviders logProviders)
    {
        switch (logProviders)
        {
            case LoggingProviders.Enterprise:
                return new EnterpriseLogger();
            case LoggingProviders.Log4Net:
                return new Log4NetLogger();
            default:
                return new EnterpriseLogger();
        }
    }
}
```

Те, що ми отримуємо від методу *GetLoggingProvider*, є об'єктом, що реалізує потрібний інтерфейс. *Фабричний Метод* вирішує, який із конкретних класів створювати на основі вхідного параметру. Глянемо на використання:

Уривок коду 3.3. Використання Фабричного Методу

```
public static void Run()
{
    var providerType = GetTypeOfLoggingProviderFromConfigFile();
    ILogger logger = LoggerProviderFactory.GetLoggingProvider(providerType);
    logger.LogMessage("Hello Factory Method Design Pattern.");
    // Вивід: [Log4Net: Hello Factory Method Design Pattern]
}
private static LoggingProviders GetTypeOfLoggingProviderFromConfigFile()
{
    // Це такий собі хадркод, щоб не ускладнювати прикладу
    return LoggingProviders.Log4Net;
}
```

А тепер UML:



UML-діаграма 3. Фабричний Метод