Creational - Abstract Factory Design Pattern

Birden fazla ürün ailesi ile çalışmak durumunda kaldığımızda , ürün ailesi ile istemci tarafını soyutlamak için kullanılır. Factory design pattern'den farkı budur. Soyut fabrika tasarım kalıbının en belirgin özelliği, üretilecek nesnelerin birbirleriyle ilişkili olmasıdır. Oluşturulan bu yapıda üretilen nesnelerin kendisiyle ilgilenmeye gerek yoktur. Diğer bir deyişle üretim sınıfında, üretimin yapılacağı fabrikanın hangi fabrika olduğu veya üretilen nesnelerin hangi tür olduğu ile ilgilenilmez. Bu sayede aynı arayüzü veya soyut sınıfı kullanarak yeni nesneleri kalıba eklemek kolaylaştırılmıştır. Birçok platformda çalışacak biçimde geliştirilen uygulamalar var. Yani, hem web üzerinden erişebildiğiniz; hem de Android, IOS ya da Windows 10 üzerinde native olarak kullanabildiğiniz uygulamalardan bahsediyorum. Böyle bir uygulamada basit bir ekran varsayalım. Basitçe, iki metin giriş kontrolü ve bir butondan oluşan bir kullanıcı giriş ekranı olsun mesela. Buradaki kontrollerin arayüzü ve özellikleri çalışacağı işletim sistemine bağlıdır değil mi? Android'de başka, IOS'da başka, Windows'da başka, Web'de başka bir buton ama hepsinde bir buton sonuç olarak.O zaman şöyle bir uygulama oluşturucusu yapsak ne iyi olurdu… Biz ekran tasarımını yapsak sonra da desek ki, al bunu şu işletim sistemine göre oluştur. Uygulama da tasarımı alıp, belirttiğimiz işletim sistemine göre sıfırdan yaratsa.

Loglama için abstract class'ımız

```
abstract class LoggingBase{
   public abstract void log(String message);
}
```

Loglama için concrete classlarımız

```
class NLogger extends LoggingBase{
   @Override
   public void log(String message) {
      System.out.println("Logged with NLogger");
   }
}
class Log4Net extends LoggingBase{
   @Override
   public void log(String message) {
      System.out.println("Logged with Log4Net");
   }
}
```

Cacheleme için abstract class'ımız

```
abstract class CachingBase{
   public abstract void Cache(String data);
}
```

Cacheleme için concrete classlarımız

```
class RedisCache extends CachingBase{
   @Override
   public void Cache(String data) {
      System.out.println("Cached with Redis Cache");
   }
}
class MemCache extends CachingBase{
   @Override
   public void Cache(String data) {
      System.out.println("Cached with MemCache");
   }
}
```

Factory'lere hizmet edecek interface'imiz

```
interface FactoryBase {
    public LoggingBase createLogger();
    public CachingBase createCacher();
}
```

Factory'lerimiz

```
class FactoryOne implements FactoryBase{
    @Override
    public LoggingBase createLogger() {
        return new Log4Net();
    }
    @Override
    public CachingBase createCacher() {
        return new RedisCache();
    }
}
class FactoryTwo implements FactoryBase{
    @Override
    public LoggingBase createLogger() {
        return new NLogger();
    }
    @Override
    public CachingBase createCacher() {
        return new MemCache();
    }
}
```

Business

```
class ProductManager{
    private FactoryBase factoryBase;
    private LoggingBase loggingBase;
    private CachingBase cachingBase;
    public ProductManager(FactoryBase factoryBase){
        this.factoryBase = factoryBase;
        this.loggingBase = factoryBase.createLogger();
        this.cachingBase = factoryBase.createCacher();
    }
    public void getAll(){
        System.out.println("All product returned");
        loggingBase.log("Logged");
        cachingBase.Cache("Cached");
    }
}
```

Main

```
public static void main(String[] args) {
    ProductManager pm = new ProductManager(new FactoryOne());
    pm.getAll();
}
```