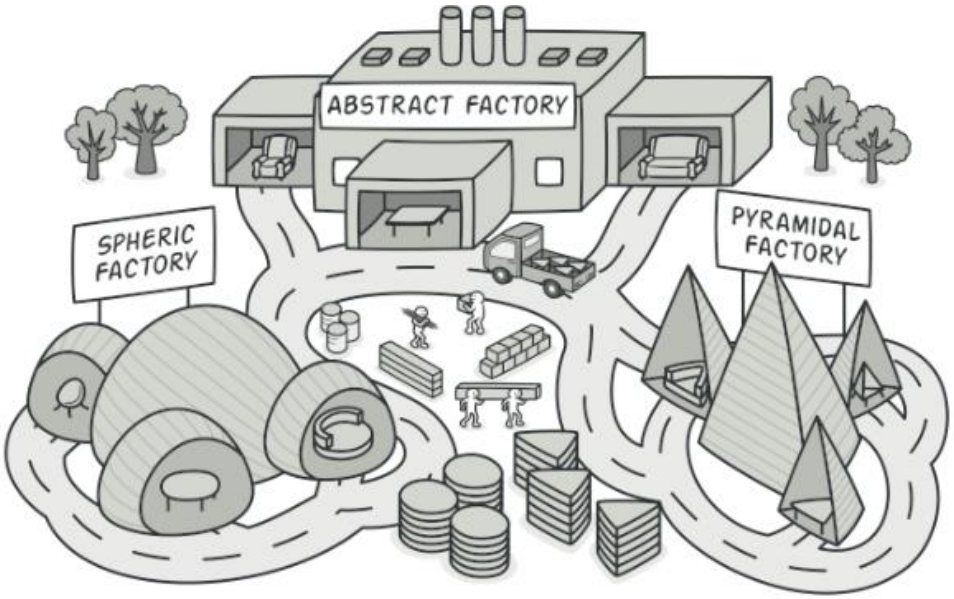
فاکتوری انتزاعی(Abstract Factory)

مفهوم

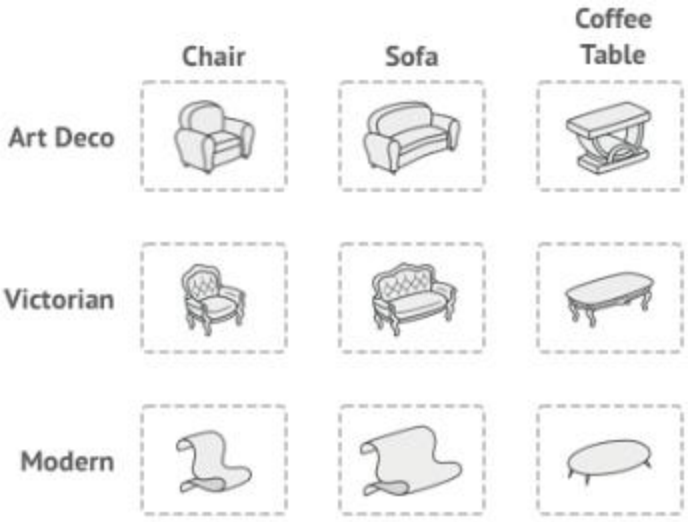
فاکتوری انتزاعی الگوی طراحی خلاقانه ای است که با کمک آن می توان خانواده هایی از اشیاء مرتبط را بدون تعریف کلاس های عینی آن ها تولید کرد.



مشکل

فرض کنید می خواهید شبیه سازی برای یک فروشگاه مبلمان بسازید. کد شما متشکل از کلاس هایی است که بیانگر موارد زیر است:

1. **خانواده ی محصولات مرتبطی مثل Chair + Sofa + CoffeeTable.**
2. **متغیرهای متعدد این خانواده. برای مثال محصولات Chair + Sofa + CoffeeTable در این متغیرها موجود هستند: IKEA, VictorianStyle, ArtDeco.**



خانواده های محصولات و متغیرهای آن ها

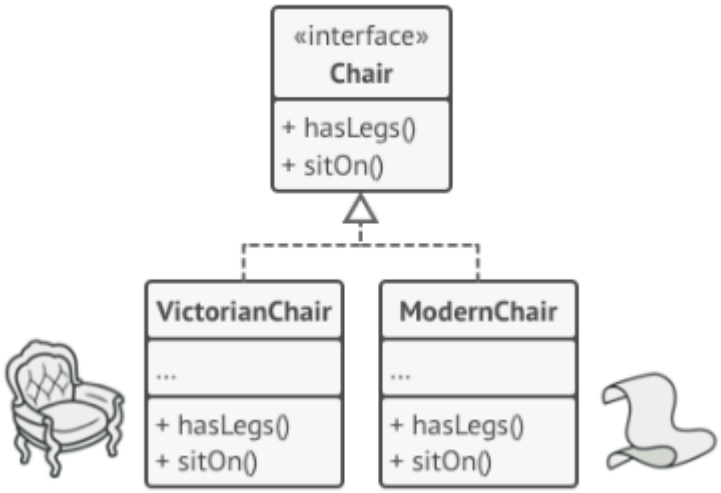
شما به روشی نیاز دارید که بتوانید تک تک اشیاء مبلمان را به گونه ای ایجاد کنید که این اشیاء با اشیاء دیگر همین خانواده مطابقت داشته باشند. مشتریان در صورتی که مبلی را دریافت کنند که سفارش نداده اند، بسیار عصبی می شوند.

یکی از مبل های IKEA با یکی از صندلی های *Victorian-style* مطابقت ندارد.

همچنین نیازی به تغییر کدهای موجود طی اضافه کردن محصولات جدید یا خانواده های محصولات به برنامه نیست. فروشندگان مبلمان کاتالوگ های خود را مرتب به روز می کنند و نیازی نیست که کد اصلی را با هربار به روزرسانی تغییر دهید.

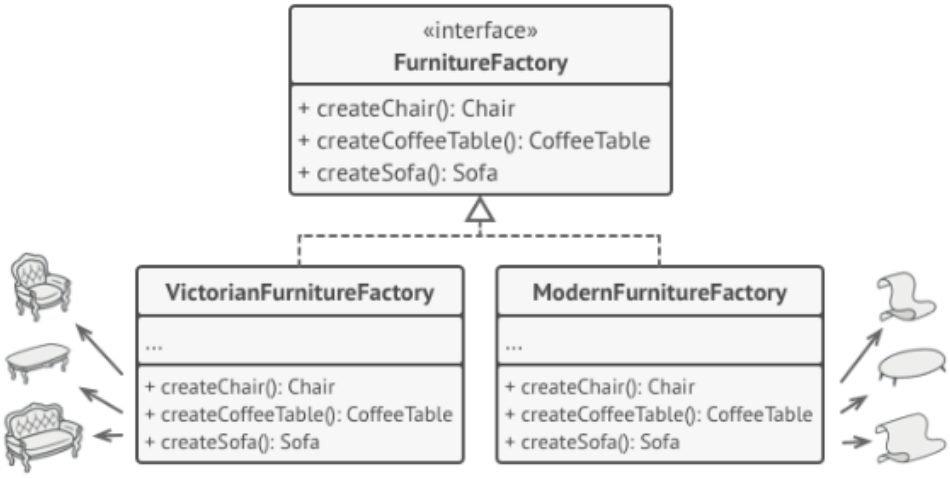
راه حل

اولین امکانی که الگوی فاکتوری انتزاعی فراهم می کند، اعلان صریح رابط ها برای هر یک از محصولات مجزای خانواده ی محصولات است، مثلا صندلی، مبل یا میز قهوه خوری. پس از آن می توانید تمامی متغیرهای محصولات را مجبور به پیروی از این رابط ها کنید. برای مثال تمامی متغیرهای صندلی می توانند رابط Chair و تمامی متغیرهای میز قهوه خوری می توانند رابط CoffeeTable را پیاده سازی کنند.



تمامی متغیرهای یک شیء باید به تنها یک سلسله مراتب از کلاس منتقل شوند.

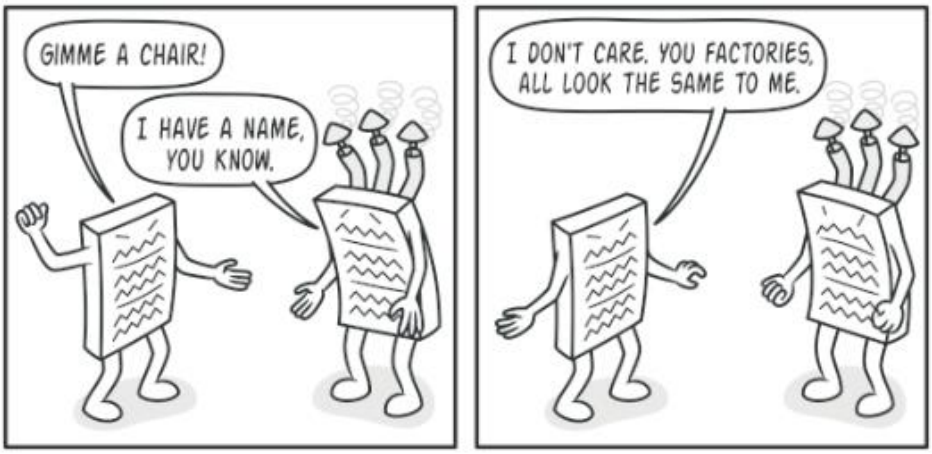
قدم بعدی اعلان رابط AbstractFactory به همراه لیستی از متدهای ایجاد تمامی محصولاتی است که بخشی از خانواده ی محصولات هستند (مثلا createChair، createSofa و createCoffeeTable). این متدها باید انواع انتزاعی محصولات بیان شده توسط رابط هایی که ما قبلا استخراج کرده ایم را برگشت دهند، یعنی Chair، Sofa، CoffeeTable و . . . .



هر یک از concrete factory ها با یک متغیر مشخص محصول ارتباط دارند.

حالا باید متغیرهای محصولات را چکار کنیم؟ برای هر یک از متغیرهای خانواده ی محصولات باید بر اساس رابط AbstractFactory یک کلاس فاکتوری مجزا را ایجاد کنیم. فاکتوری کلاسی است که محصولاتی با نوع مشخص را برگشت می دهد. برای مثال IKEAFactory تنها می تواند اشیاء IKEAChair، IKEASofa و IKEACoffeeTable را ایجاد کند.

کد کلاینت باید بتواند از طریق رابط های انتزاعی مربوطه با هر دو فاکتوری و محصولات آن ها کار کند. از این طریق می توانید نوع کارخانه ای که شما به کد کلاینت می دهید را به همراه نوع محصولی که کد کلاینت دریافت می کند، تغییر دهید؛ بدون این که کد کلاینت واقعی را بشکنید.

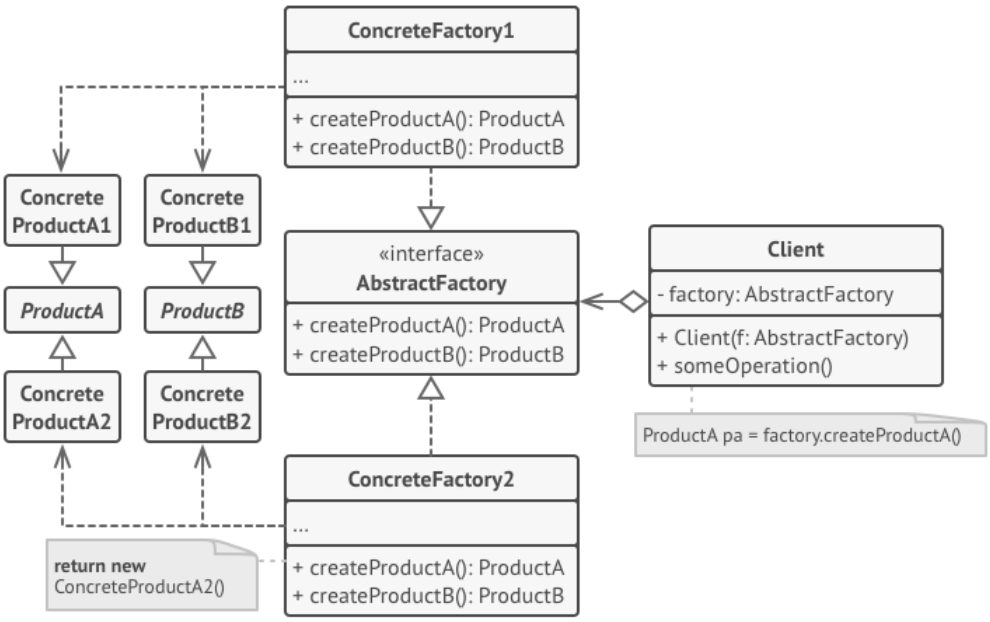


کلاینت اهمیتی به کلاس عینی فاکتوری که با آن کار می کند، نمی دهد.

فرض کنید کلاینت از یک فاکتوری بخواهد یک صندلی را تولید کند. کلاینت نیازی نیست حتما نسبت به کلاس فاکتوری آگاه باشد، همچنین نوع صندلی دریافتی نیز برای او اهمیتی ندارد. این صندلی چه از مدل IKEA مدرن باشد و چه Victorian-style، به هرحال کلاینت باید به یک صورت با تمامی صندلی ها رفتار کند؛ یعنی با استفاده از رابط انتزاعی Chair. از این طریق تنها چیزی که کلاینت درباره ی صندلی می داند، این است که این صندلی به گونه ای متد sit را اجرا می کند. همچنین هر متغیری از صندلی که برگشت داده می شود، همیشه با نوع مبل یا میز ناهارخوری تولید شده توسط یک شیء فاکتوری یکسان مطابقت دارد.

برای روشن شدن مطلب یک نکته ی دیگر باقی مانده است. اگر کلاینت تنها در معرض رابط های انتزاعی قرار داشته باشد، در این صورت چه چیزی اشیاء واقعی فاکتوری را می سازد؟ معمولا برنامه در مرحله ی مقداردهی اولیه یک شیء فاکتوری عینی را می سازد. درست قبل از آن برنامه باید بر اساس پیکربندی یا تنظیمات محیطی نوع فاکتوری را انتخاب کند.

ساختار

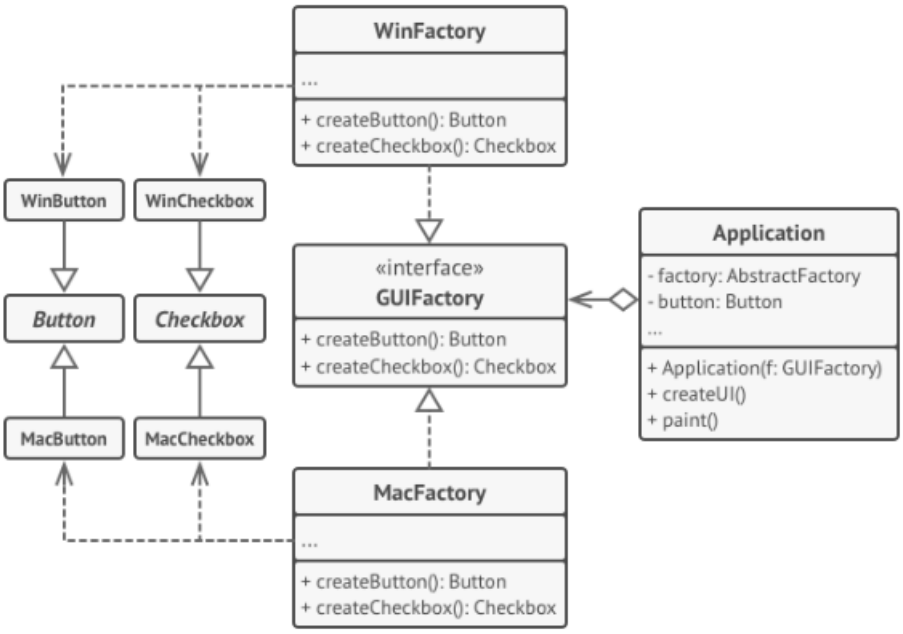


1. محصولات انتزاعی رابط ها را برای مجموعه ای از محصولات مجزا و در عین حال مرتبط اعلان می کنند تا خانواده ی محصولات شکل گیرد.

2. محصولات عینی پیاده سازی های مختلفی از محصولات انتزاعی هستند که توسط متغیرها گروه بندی شده اند. هر یک از محصولات انتزاعی (صندلی و مبل) باید در تمامی متغیرهای معلوم (مدرن و ویکتوریایی) پیاده سازی شوند.
3. رابط فاکتوری انتزاعی مجموعه ای از متدها را برای ایجاد هر یک از محصولات انتزاعی اعلان می کند.
4. فاکتوری های عینی متدهای ایجاد فاکتوری انتزاعی را اجرا می کنند. هر یک از فاکتوری های عینی به متغیر مشخصی از محصولات ارتباط دارند و تنها متغیرهای این محصولات را ایجاد می کنند.
5. گرچه فاکتوری های عینی محصولات عینی را نمونه سازی می کنند، اما امضاهای متدهای ایجاد آن ها باید محصولات انتزاعی متناظر را برگشت دهند. در این صورت کد کلاینتی که از یک فاکتوری استفاده می کند، به متغیر مشخصی از محصول دریافتی از فاکتوری جفت نمی شود. کلاینت می تواند با تمامی متغیرهای محصول و فاکتوری عینی کار کند، به شرط آن که از طریق رابط های انتزاعی با اشیاء دیگر در ارتباط باشد.

شبه کد

در این برنامه چگونگی استفاده از الگوی فاکتوری انتزاعی برای ایجاد عناصر رابط کاربری چندپلتفرمی بدون جفت کردن کد کلاینت به کلاس های عینی رابط کاربری نشان داده شده است. به گونه ای که در عین حال بتوان تمامی عناصر ایجاد شده را منطبق با یک سیستم عامل مشخص نگه داشت.



مثال کلاس های چندپلتفرمی رابط کاربری

انتظار می رود عناصر رابط کاربری یکسان موجود در یک برنامه ی چندپلتفرمی رفتار مشابهی داشته باشند. اما در عین حال در سیستم عامل های مختلف این عناصر رفتار متفاوتی دارند. علاوه بر این تضمین این که عناصر رابط کاربری با سبک سیستم عامل فعلی مطابقت داشته باشند، وظیفه ی شماست. هیچ کس نمی خواهد برنامه اش در ویندوز کنترل های سیستم عامل مک را نمایش دهد.

رابط فاکتوری انتزاعی مجموعه ای از متدهای ایجاد کننده ای را اعلان می کند که کد کلاینت بتواند برای تولید انواع مختلفی از عناصر رابط کاربری از آن ها استفاده کند. فاکتوری های عینی با سیستم عامل های مشخصی مرتبط بوده و عناصری از رابط کاربری را ایجاد می کنند که با این سیستم عامل مشخص مطابقت دارند.

شیوه ی عملکرد به این صورت است که زمانی که برنامه ای اجرا می شود، این برنامه نوع سیستم عامل فعلی را بررسی می کند. برنامه برای ایجاد یک شیء فاکتوری از کلاسی که سیستم عامل را مطابقت می دهد، از این اطلاعات استفاده می کند. بخش های باقی مانده ی کدها برای ایجاد عناصر رابط کاربری از این فاکتوری استفاده می کنند. از این طریق از ایجاد عناصر اشتباه جلوگیری می شود.

با این رویکرد، کد کلاینت به کلاس های عینی فاکتوری ها و عناصر رابط کاربری وابسته نمی شوند، به شرط آن که این کد از طریق رابط های انتزاعی این اشیاء با آن ها کار کند. همچنین از این طریق کد کلاینت از فاکتوری ها و عناصر رابط کاربری دیگری که شما ممکن است در آینده آن ها را اضافه کنید، پشتیبانی می کند.

در نتیجه باید کد کلاینت را هربار که متغیر جدیدی از عناصر رابط کاربری را به برنامه ی خود اضافه می کنید، تغییر دهید. تنها کافی است کلاس فاکتوری جدیدی را ایجاد کنید که این عناصر را ایجاد کند و تا حدی کد مقداردهی اولیه ی برنامه را تغییر دهید تا برنامه بتواند در زمان لازم این کلاس را انتخاب کند.

```
// The abstract factory interface declares a set of methods that
// return different abstract products. These products are called
// a family and are related by a high-level theme or concept.
// Products of one family are usually able to collaborate among
// themselves. A family of products may have several variants,
// but the products of one variant are incompatible with the
// products of another variant.
interface GUIFactory is
  method createButton():Button
  method createCheckbox():Checkbox


// Concrete factories produce a family of products that belong
// to a single variant. The factory guarantees that the
// resulting products are compatible. Signatures of the concrete
// factory's methods return an abstract product, while inside
// the method a concrete product is instantiated.
class WinFactory implements GUIFactory is
  method createButton():Button is
    return new WinButton()
  method createCheckbox():Checkbox is
    return new WinCheckbox()

// Each concrete factory has a corresponding product variant.
class MacFactory implements GUIFactory is
  method createButton():Button is
    return new MacButton()
  method createCheckbox():Checkbox is
    return new MacCheckbox()


// Each distinct product of a product family should have a base
// interface. All variants of the product must implement this
// interface.
interface Button is
  method paint()
```

```
// Concrete products are created by corresponding concrete
// factories.
class WinButton implements Button is
    method paint() is
        // Render a button in Windows style.

class MacButton implements Button is
    method paint() is
        // Render a button in macOS style.

// Here's the base interface of another product. All products
// can interact with each other, but proper interaction is
// possible only between products of the same concrete variant.
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in Windows style.

class MacCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in macOS style.


// The client code works with factories and products only
// through abstract types: GUIFactory, Button and Checkbox. This
// lets you pass any factory or product subclass to the client
// code without breaking it.
class Application is
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()


// The application picks the factory type depending on the
// current configuration or environment settings and creates it
// at runtime (usually at the initialization stage).
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
```

```
else
  throw new Exception("Error! Unknown operating system.")

Application app = new Application(factory)
```

کاربرد

زمانی که نیاز است کد شما با خانواده های مختلفی از محصولات مرتبط کار کند و در عین حال نمی خواهید کدتان به کلاس های عینی این محصولات وابسته باشد، از فاکتوری انتزاعی استفاده کنید. این محصولات ممکن است پیشاپیش مجهول باشند، یا اینکه صرفا شما می خواهید فضایی را برای توسعه ی آینده در نظر بگیرید.

فاکتوری انتزاعی این امکان را به شما می دهد تا برای ایجاد اشیاء از هر یک از کلاس های خانواده ی محصولات از یک رابط استفاده کنید. تا زمانی که کد شما اشیاء را از طریق این رابط ایجاد می کند، نیازی به نگرانی درباره ی ایجاد متغیر اشتباه از محصول و مطابق نبودن آن با محصولات از قبل ایجاد شده توسط برنامه نیست.

- زمانی که کلاسی را با مجموعه ای از متدهای فاکتوری در اختیار دارید، به گونه ای که مسئولیت اصلی آن نامشخص می شود، سعی کنید از فاکتوری انتزاعی استفاده کنید.
- در برنامه ای که به خوبی طراحی شده است، هر یک از کلاس ها تنها مسئول یک کارهستند. زمانی که کلاسی با چندین نوع محصول مواجه می شود، ممکن است استخراج متدهای فاکتوری آن داخل یک کلاس فاکتوری مجزا یا یک پیاده سازی فاکتوری مجزای کامل با ارزش باشد.

چگونگی اجرا

1. ماتریسی از انواع مجزایی از محصولات را در برابر متغیرهای این محصولات بکشید.
2. به ازای تمامی انواع محصولات، رابط های انتزاعی را اعلان کنید، سپس تمامی کلاس های عینی محصولات را وادار به اجرای این رابط ها کنید.
3. رابط فاکتوری انتزاعی را به همراه مجموعه ای از متدهای ایجاد تمامی محصولات انتزاعی ایجاد کنید.
4. مجموعه ای از کلاس های فاکتوری عینی را برای هر یک از متغیرهای محصولات اجرا کنید.

5. در جایی از برنامه کد مقداردهی اولیه ی فاکتوری را ایجاد کنید. از این طریق یکی از کلاس های فاکتوری عینی نمونه سازی می شود که این امر به پیکربندی یا محیط فعلی برنامه بستگی دارد. این شیء فاکتوری را به تمامی کلاس هایی که محصولات را می سازند، عبور دهید.
6. کد را مرور کنید و تمامی فراخوان های مستقیم سازنده های محصولات را پیدا کنید و جای آن ها را با فراخوان های متد ایجاد مناسب شیء فاکتوری عوض کنید.

معایب و مزایا

- ✓ می توانید از بابت سازگار بودن محصولات دریافتی از یکی از فاکتوری ها اطمینان حاصل کنید.
- ✓ از جفت شدن محصولات عینی با کد کلاینت جلوگیری کنید.
- ✓ اصل مسئولیت واحد. شما می توانید کد ایجاد محصول را در یک مکان واحد از برنامه انتقال دهید و کار با کد را آسان تر کنید.
- ✓ اصل باز و بسته. می توانید انواع جدیدی از محصولات را بدون نیاز به شکستن کدهای کلاینت موجود وارد کنید.
- × پیچیدگی کد ممکن است افزایش یابد. چرا که باید جهت پیاده سازی این الگو رابط ها و کلاس های جدید بسیاری را ایجاد کنید.

رابطه با الگوهای دیگر

- بسیاری از طرح ها با استفاده از متد فاکتوری آغاز می شوند (با قابلیت شخصی سازی بیشتر و پیچیدگی کمتر از طریق زیرکلاس ها) و از طریق Abstract Factory ، Prototype یا Builder کامل می شوند (انعطاف پذیری و در عین حال پیچیدگی بیشتر).
- Builder به صورت گام به گام بر ساخت اشیاء پیچیده متمرکز است. فاکتوری انتزاعی در ساخت خانواده های اشیاء مرتبط تخصص دارد. فاکتوری انتزاعی بلافاصله محصول را برگشت می دهد، این درحالی است که Builder این امکان را به شما می دهد تا قبل از گرفتن محصول برخی از مراحل ساخت اضافی را اجرا کند.

- کلاس های فاکتوری انتزاعی اغلب مبتنی بر مجموعه ای از متدهای فاکتوری هستند، اما می توانید برای ساخت متدهای این کلاس ها از Prototype نیز استفاده کنید.
- در مواقعی که تنها می خواهید روش ایجاد اشیاء توسط زیرسیستم را از کد کلاینت مخفی کنید، می توانید به عنوان جایگزین فاکتوری انتزاعی از Facade استفاده کنید.
- می توانید در کنار Bridge از فاکتوری انتزاعی استفاده کنید. این ارتباط زمانی به کار شما می آید که برخی از انتزاعات تعریف شده توسط Bridge بتوانند با پیاده سازی های مشخصی کار کنند. در این حالت فاکتوری انتزاعی می تواند این روابط را بپوشاند و پیچیدگی را از دید کد کلاینت مخفی کند.
- Prototypes، Builders و Abstract Factories را می توان تماما به صورت Singletons پیاده سازی کرد.