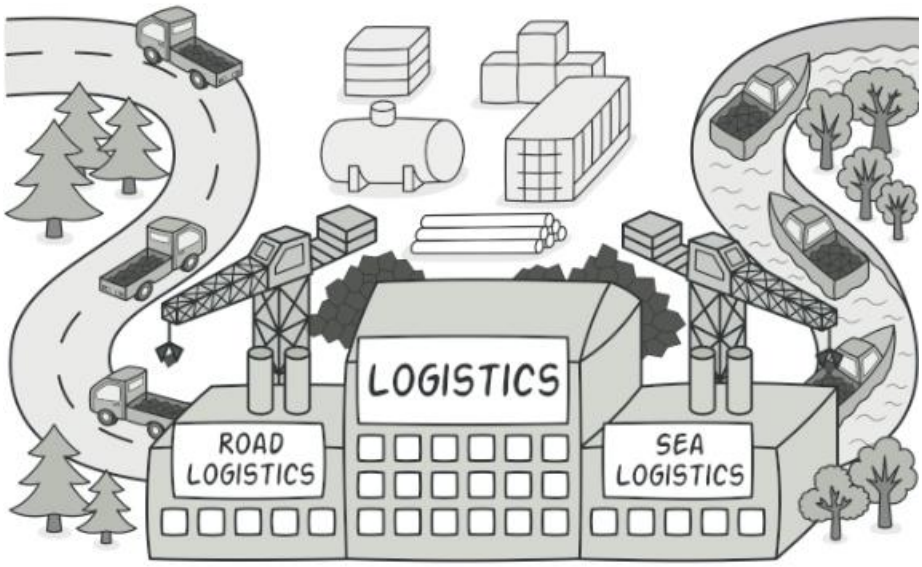
# Factory Method

## مفهوم

متد فاکتوری الگوی طراحی خلاقانه ای است که با کمک آن می توان رابطی را برای ایجاد اشیاء در ابر کلاس ها فراهم کرد. در عین حال این ابر کلاس ها از طریق آن می توانند نوع اشیائی که قرار است ایجاد شود را تغییر دهند.




## مشکل

فرض کنید که می خواهید یک برنامه ی مدیریت لجیستیکی را ایجاد کنید. اولین نسخه از برنامه ی شما می تواند فقط نقل و انتقال کامیون ها را مدیریت کند. بنابراین حجم زیادی از کدها داخل کلاس Truck قرار دارند.

بعد از مدتی برنامه ی شما کاملا محبوب می شود. به صورت روزانه ده ها درخواست از شرکت های حمل و نقل دریایی را دریافت می کنید و از شما خواسته می شود تا لجیستیک های دریایی را داخل برنامه جای دهید.

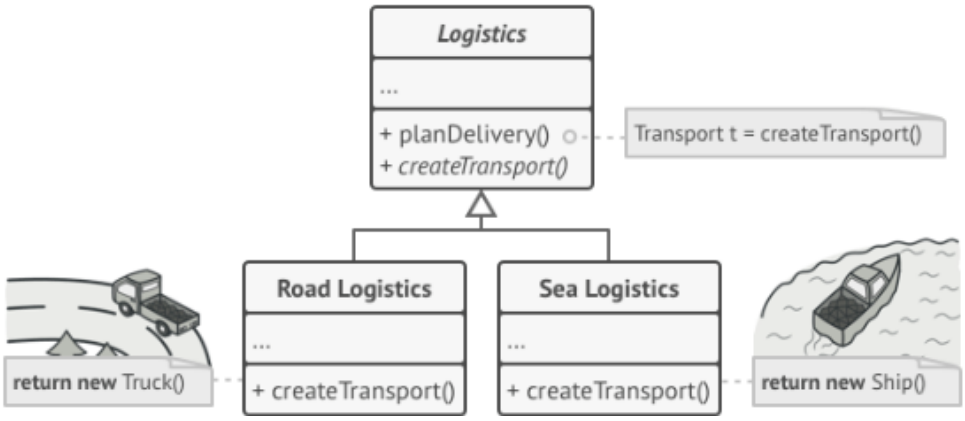

اضافه کردن یک کلاس جدید به برنامه خیلی ساده نیست مخصوصا اگر بخش های باقی مانده ی کد با کلاس های موجود جفت شده باشند.

خبر خوبی است، نه؟ اما کد را باید چه کنیم؟ در حال حاضر اغلب کد شما به کلاس Truck جفت شده است. اضافه کردن Ships به برنامه مستلزم ایجاد تغییراتی در کل پایگاه کد است. علاوه بر این اگر شما بعدها بخواهید نوع دیگری از حمل و نقل را به برنامه اضافه کنید، باید دوباره تمام این تغییرات را اضافه کنید.

در نتیجه کدتان تا حد زیادی به هم ریخته می شود و مملو از شروطی می شود که بر اساس کلاس اشیاء حمل و نقل رفتار برنامه را تغییر می دهد.

## راه حل

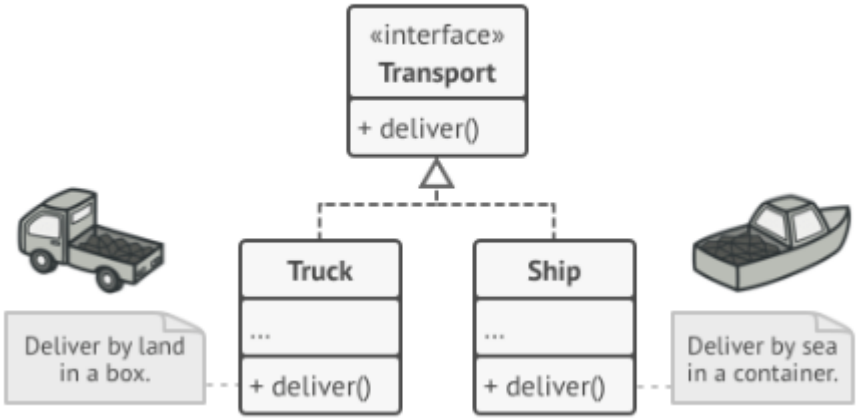
الگوی متد فاکتوری این امکان را به شما می دهد تا فراخوان های مستقیم ساخت شیء را (با استفاده از عملگر new) با فراخوان های متد ویژه ی فاکتوری عوض کنید. جای نگرانی نیست، چرا که اشیاء همچنان از طریق عملگر new ایجاد می شوند اما از داخل این متد فاکتوری فراخوانی می شوند. اشیائی که توسط یک متد فاکتوری برگشت داده می شوند، اغلب محصول نامیده می شوند.




**زیرکلاس ها می توانند کلاس اشیائی که توسط متد فاکتوری برگشت داده می شوند را تغییر دهند.**

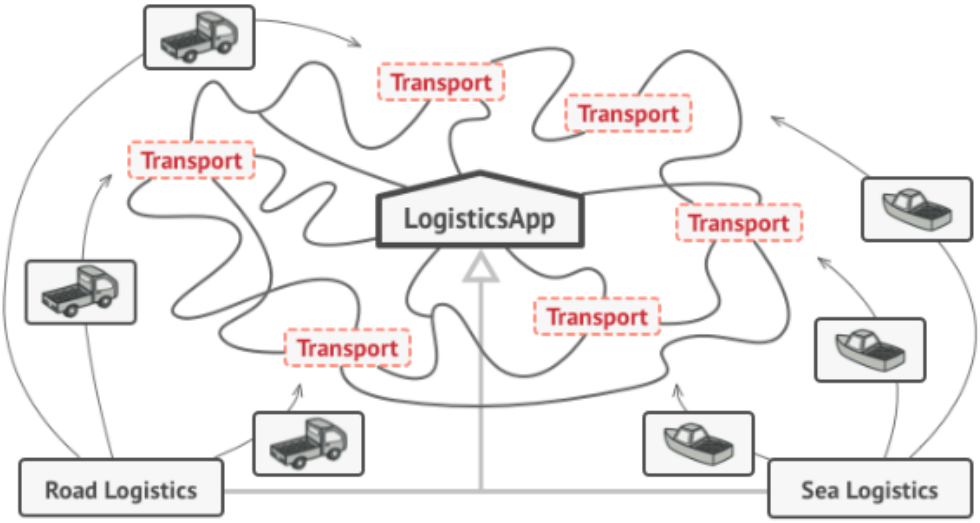
**در نگاه اول این تغییر ممکن است بی فایده به نظر برسد، چرا که ما فقط فراخوان سازنده را از بخشی به بخش دیگری از برنامه انتقال داده ایم. اما این را در نظر داشته باشید که الان شما می توانید متد فاکتوری را در یک زیرکلاس اور راید کنید و کلاس محصولاتی که توسط این متد در حال ایجاد شدن هستند را تغییر دهید.**

**با این حال محدودیت کوچکی نیز پیش روی خود دارید. به این صورت که زیر کلاس ها ممکن است انواع مختلفی از محصولات را برگشت دهند، اما به شرط آن که این محصولات کلاس یا رابط پایه ی مشترکی داشته باشند. همچنین نوع برگشتی متد فاکتوری موجود در کلاس پایه مانند این رابط باید تعریف شود.**




**تمامی محصولات باید از رابط یکسانی پیروی کنند.**

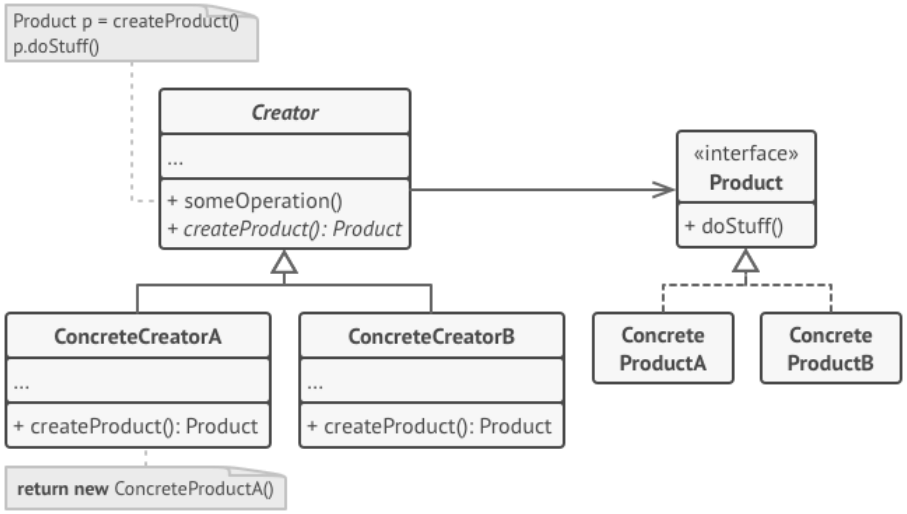
برای مثال هم کلاس Truck و هم کلاس Ship باید رابط Transport را پیاده سازی کنند که از این طریق متدی به نام deliver اعلان می شود. هر یک از کلاس ها به طرق متفاوتی این متد را پیاده سازی می کنند. به این صورت که کامیون ها بار را به صورت زمینی به مقصد می رسانند و کشتی ها از طریق دریا این کار را انجام می دهند. متد فاکتوری کلاس RoadLogistics اشیاء truck را برگشت می دهد. این در حالی است که متد فاکتوری کلاس SeaLogistics اشیاء Ship را برگشت می دهد.




تا زمانی که تمامی کلاس های product رابط مشترکی را پیاده می کنند، شما می توانید اشیاء آن ها را به کد کلاینت بدهید، بدون آن که نیاز به شکستن این کد داشته باشید.

کدی که از متد فاکتوری استفاده می کند (اغلب به آن کد کلاینت گفته می شود) تفاوتی بین محصولات واقعی برگشت داده شده توسط زیرکلاس های مختلف قائل نمی شود. کلاینت تمامی محصولات را به صورت Transport انتزاعی در نظر می گیرد. کلاینت می داند که تمامی اشیاء حمل و نقل باید دارای متد deliver باشند، اما چگونگی کارکرد آن برای کلاینت اهمیتی ندارد.

## ساختار




1. Productرابط را اعلان می کند که این رابط برای تمامی اشیاء مشترک است و می توان این اشیاء را توسط creatorو زیرکلاس های آن تولید کرد.
2. Concrete Product ها پیاده سازی متفاوتی از رابط محصول هستند.
3. کلاس Creator متد فاکتوری را اعلان می کند و در نهایت اشیاء محصولات جدید برگشت داده می شوند. مهم است که نوع برگشتی این متد با رابط محصول مطابقت داشته باشد.

می توانید متد فاکتوری را به صورت انتزاعی تعریف کنید تا تمامی زیرکلاس ها مجبور به پیاده سازی نسخه های متد مخصوص به خود شوند. به عنوان جایگزین متد پایه ی فاکتوری می تواند برخی از انواع پیش فرض محصولات را برگشت دهد.

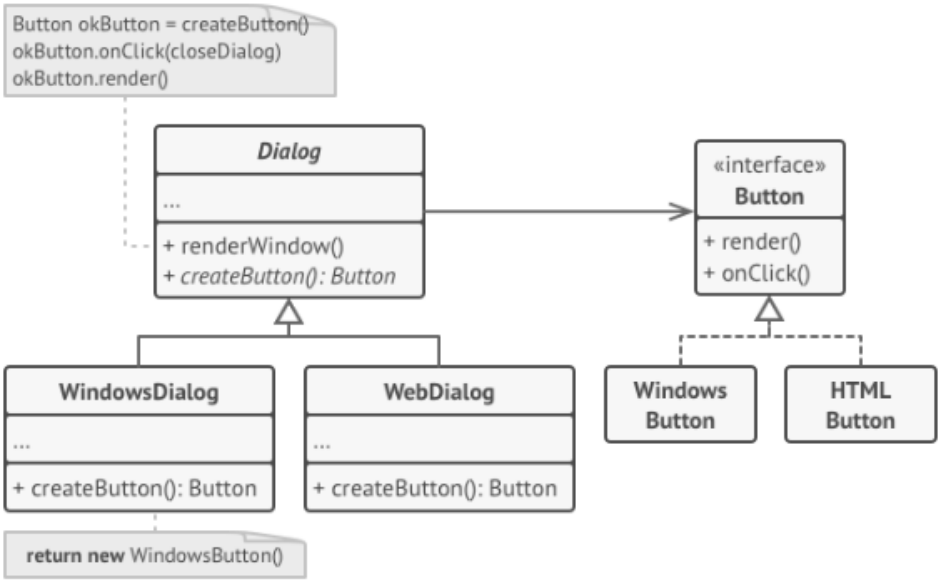
**نکته:** برخلاف اسم Creator ایجاد محصول مسئولیت اصلی آن نیست. معمولا کلاس Creator دارای برخی از منطق های اصلی تجاری مرتبط با محصولات است. با استفاده از متد فاکتوری می توان این منطق را از کلاس های concrete product ها جدا کرد. برای تشبیه، یک شرکت بزرگ توسعه ی نرم افزار می تواند برای آموزش برنامه نویسان خود بخشی را در نظر بگیرد. با این حال وظیفه ی اصلی این شرکت به صورت کلی همچنان کدنویسی و نه تولید برنامه نویس است.

4. Concrete Creator ها متد فاکتوری پایه را اور راید می کنند. به گونه ای که این متد نوع مختلفی از محصول را برگشت دهد.

توجه داشته باشید که متد فاکتوری حتما نباید همیشه نمونه های جدید را ایجاد کند، بلکه می تواند اشیاء موجود را از حافظه ی کش از گروهی از اشیاء یا منبع دیگری برگشت دهد.

## شبه کد

در این مثال چگونگی استفاده از متد فاکتوری برای ایجاد عناصر رابط کاربری چندپلتفرمی بدون نیاز به جفت کردن کد کلاینت به کلاس های عینی رابط کاربری توضیح داده شده است.




نمونه ای از دیالوگ چندپلتفرمی

کلاس دیالوگ پایه برای رندر کردن پنجره ی خود از عناصر مختلفی از رابط کاربری استفاده می کند. در سیستم عامل های مختلف این عناصر اندکی باهم تفاوت دارند، اما ممکن است رفتار آن ها یکی باشد. ماهیت یک دکمه در ویندوز و لینوکس تفاوتی ندارد.

زمانی که متد فاکتوری وارد عمل می شود، نیازی به بازنویسی منطق دیالوگ برای هر یک از سیستم عامل ها وجود ندارد. اگر ما یک متد فاکتوری را اعلان کنیم تا دکمه هایی را داخل کلاس دیالوگ پایه ایجاد کند، می توانیم بعدا زیرکلاس دیالوگی را ایجاد کنیم که دکمه های به سبک ویندوز را از متد فاکتوری برگشت دهند.

سپس این زیرکلاس اغلب کدهای دیالوگ کلاس پایه را به ارث می برد، اما به لطف متد فاکتوری دکمه های شبه ویندوزی را بر روی صفحه نمایش می دهد.

برای آن که این الگو جواب دهد، کلاس دیالوگ پایه باید با دکمه های انتزاعی همکاری کند، به این صورت که کلاس و رابط پایه ای وجود داشته باشد که تمامی دکمه های عینی از آن ها پیروی کنند. در این صورت کد دیالوگ صرف نظر از نوع دکمه هایی که با آن ها کار می کند، همچنان به کار خود ادامه می دهد.

البته می توانید این رویکرد را بر روی عناصر رابط کاربری دیگر نیز پیاده کنید. با این حال با هر متد فاکتوری جدیدی که به این دیالوگ اضافه می کنید، شما به الگوی Abstract Factory نزدیک می شوید. نگران نباشید، در آینده در رابطه با این الگو بیشتر صحبت خواهیم کرد.

```
// The creator class declares the factory method that must
// return an object of a product class. The creator's subclasses
// usually provide the implementation of this method.
class Dialog is
  // The creator may also provide some default implementation
  // of the factory method.
  abstract method createButton()

  // Note that, despite its name, the creator's primary
  // responsibility isn't creating products. It usually
  // contains some core business logic that relies on product
  // objects returned by the factory method. Subclasses can
  // indirectly change that business logic by overriding the
  // factory method and returning a different type of product
  // from it.
  method render() is
    // Call the factory method to create a product object.
    Button okButton = createButton()
    // Now use the product.
    okButton.onClick(closeDialog)
    okButton.render()


// Concrete creators override the factory method to change the
// resulting product's type.
class WindowsDialog extends Dialog is
  method createButton() is
    return new WindowsButton()

class WebDialog extends Dialog is
  method createButton() is
    return new HTMLButton()
```

```
// The product interface declares the operations that all
// concrete products must implement.
interface Button is
    method render()
    method onClick(f)

// Concrete products provide various implementations of the
// product interface.
class WindowsButton implements Button is
    method render(a, b) is
        // Render a button in Windows style.
    method onClick(f) is
        // Bind a native OS click event.

class HTMLButton implements Button is
    method render(a, b) is
        // Return an HTML representation of a button.
    method onClick(f) is
        // Bind a web browser click event.


class Application is
    field dialog: Dialog

    // The application picks a creator's type depending on the
    // current configuration or environment settings.
    method initialize() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating system.")

    // The client code works with an instance of a concrete
    // creator, albeit through its base interface. As long as
    // the client keeps working with the creator via the base
    // interface, you can pass it any creator's subclass.
    method main() is
        this.initialize()
        dialog.render()
```

## کاربرد

زمانی که پیشاپیش نوع و وابستگی های دقیق اشیائی که کد شما قرار است با آن ها کار کند را نمی دانید، از متد فاکتوری استفاده کنید.

متد فاکتوری کد ساخت محصول را از کدی که در حقیقت از این محصول استفاده می کند، جدا می کند، بنابراین توسعه ی کد ساخت محصول به صورت مستقل از کدهای دیگر آسان تر می شود.

برای مثال برای اضافه کردن نوع جدیدی از یک محصول، تنها کافی است یک ایجاد کننده ی زیرکلاس جدید را ایجاد کنید و متد فاکتوری را در آن اور راید کنید.

زمانی که می خواهید کاربران کتابخانه یا فریمورک خود را به گونه ای فراهم کنید که بتوانید اجزای داخلی آن را توسعه دهید، از متد فاکتوری استفاده کنید.

وراثت احتمالا آسان ترین راه برای توسعه ی رفتار پیش فرض یک کتابخانه یا فریمورک است. اما چگونه فریمورک باید بتواند تشخیص دهد که زیرکلاس شما باید به جای جزء استاندارد استفاده شود؟

راه حل در این جا کوتاه کردن کدی است که اجزا را در سراسر فریمورک و داخل یک متد فاکتوری واحد می سازد و در ادامه اور راید کردن این متد علاوه بر توسعه ی خود این جزء است.

بیایید ببینیم این کار چگونه انجام می شود. فرض کنید که با استفاده از یک فریمورک رابط کاربری اپن سورس برنامه ای را نوشته اید. برنامه ی شما باید دکمه های گردی داشته باشد، اما فریمورک فقط دکمه های مربعی شکل را فراهم می کند. شما کلاس استاندارد Button را به همراه زیرکلاس عالی RoundButton توسعه می دهید. اما الان باید به کلاس اصلی UIFramework بگویید که به جای زیرکلاس پیش فرض از زیرکلاس دکمه ی جدید استفاده کند.

برای دستیابی به این هدف باید یک زیرکلاس UIWithRoundButtons را از یکی کلاس فریمورک پایه ایجاد کنید و متد createButton آن را اور راید کنید. زمانی که این متد اشیاء Button موجود در کلاس پایه را برگشت می دهد، شما باید زیرکلاس خود را مجبور به برگشت اشیاء RoundButton کنید. حالا به جای UIFramework از UIWithRoundButtons استفاده کنید. همین!

زمانی که می خواهید منابع سیستم را با استفاده ی مجدد از اشیاء موجود به جای ساختن مجدد و هرباره ی آن ها ذخیره کنید، از متد فاکتوری استفاده کنید.

معمولا زمانی که با اشیاء بزرگ و منبع طلب سروکار داشته باشید، با این مورد مواجه می شوید. اشیائی مثل اتصالات دیتابیس، فایل های سیستمی و منابع شبکه.

بیایید ببینیم برای استفاده ی مجدد از یک شیء موجود چه کارهایی باید انجام شود:

1. ابتدا باید برای نگهداری از رد تمامی اشیاء ایجاد شده، حافظه ای را ایجاد کنید.
2. زمانی که فردی شیئی را درخواست می کند، برنامه باید داخل آن مجموعه به دنبال یک شیء آزاد بگردد.
3. ... و پس از آن، آن را به کد کلاینت برگشت دهد.
4. اگر هیچ شیء آزادی موجود نباشد، در این صورت برنامه باید شیء جدیدی را ایجاد کند و آن را به مجموعه اضافه کند.

این کار کدنویسی زیادی می طلبد و تمامی این کدها را باید در یک مکان واحد قرار داد تا برنامه با کدهای تکراری آلوده نشود.

احتمالا واضح ترین و راحت ترین جایی که این کد را می توان نگهداری کرد، سازنده ی کلاسی است که ما می خواهیم از اشیاء آن مجددا استفاده کنیم. با این حال سازنده همواره باید به صورت تعریفی اشیاء جدید را برگشت دهد و نمی تواند نمونه های حاضر را برگشت دهد.

بنابراین باید متد معینی را داشته باشید که بتواند در کنار استفاده ی مجدد از اشیاء فعلی، اشیاء جدیدی را نیز ایجاد کند. این توصیفات تا حد زیادی تداعی کننده ی متد فاکتوری است.

## چگونگی اجرا

1. تمامی محصولات را مجبور به پیروی از یک رابط کنید. این رابط باید متدهایی را اعلان کند که در تمامی محصولا معنا و مفهوم داشته باشند.
2. داخل کلاس creator یک متد فاکتوری خالی را اضافه کنید. نوع برگشتی این متد باید با رابط مشترک محصول یکی باشد.

3. در کد creator تمامی ارجاعات به سازنده های محصول را پیدا کنید. یکی یکی آن ها را با فراخوان های متد فاکتوری عوض کنید و در عین حال کد ایجاد محصول را داخل متد فاکتوری اکسترکت کنید.

   می توانید پارامتری موقتی را به متد فاکتوری اضافه کنید تا نوع محصول برگشتی را کنترل کنید.

   در این لحظه کد متد فاکتوری ممکن است زشت به نظر برسد. به این صورت که می تواند عملگر بزرگ switch ای داشته باشد که کار آن انتخاب کلاس محصولات جهت نمونه سازی است. اما جای نگرانی نیست، ما به زودی این مشکل را حل خواهیم کرد.

4. حالا مجموعه ای از زیرکلاس های creator را برای هر نوع از محصولات لیست شده در متد فاکتوری ایجاد کنید. متد فاکتوری را داخل زیرکلاس ها اور راید کنید و بیت های مناسب کد ساختاری را از متد پایه استخراج کنید.
5. اگر نوع محصولات زیاد است و ایجاد زیرکلاس برای تمامی آن ها منطقی به نظر نمی رسد، می توانید پارامتر کنترلی کلاس پایه ی موجود در زیرکلاس ها را مجددا استفاده کنید.

   برای نمونه فرض کنید سلسله مراتب کلاس های زیر را داشته باشید: کلاس پایه ی Mail همراه با دو زیرکلاس AirMail و GroundMail؛ به گونه ای که Plane، Truck و Train کلاس های Transport هستند. در حالی که کلاس AirMail تنها از اشیاء Plane استفاده می کند، GroundMail می تواند با هر دو شیء Truck و Train کار کند. می توانید زیرکلاس جدیدی را (مثل TrainMail) ایجاد کنید تا بتوانید هر دو حالت را مدیریت کنید. اما راه دیگری نیز وجود دارد. کد کلاینت می تواند آرگومانی را به متد فاکتوری کلاس GroundMail عبور دهد تا کنترل محصولات دریافتی خود را دردست گیرد.

6. اگر بعد از تمامی این استخراج ها متد فاکتوری پایه خالی شد، می توانید آن را انتزاعی کنید. اگر چیزی باقی ماند، می توانید آن را به رفتار پیش فرض متد تبدیل کنید.

## مزایا و معایب

- ✓ جلوگیری از جفت شدن creator و concrete product ها
- ✓ 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 زیرکلاس های جدید بسیاری را ایجاد کنید. بهترین حالت ممکن زمانی اتفاق می افتد که این الگو را داخل سلسله مراتب موجود در کلاس های creator 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).
- کلاس های Abstract Factory اغلب مبتنی بر مجموعه ای از متدهای فاکتوری هستند. اما همچنان می توانید برای ساختن این متدها در این کلاس ها از Prototype استفاده کنید.
- می توانید برای فراهم کردن امکان برگشت انواع مختلفی از تکرارکننده ها توسط زیرکلاس های مجموعه ای از متد فاکتوری و Iterator استفاده کنید. به گونه ای که این تکرار کننده ها با این زیرکلاس ها همخوانی داشته باشند.
- Prototype مبتنی بر وراثت نیست. بنابراین معایب وراثت را ندارد. از سویی دیگر Prototype نیاز به مقداردهی اولیه ی پیچیده ای برای شیء کپی شده دارد. متد فاکتوری مبتنی بر وراثت است اما به این مقداردهی اولیه نیاز ندارد.
- متد فاکتوری نسخه ی تخصصی متد قالب است. در عین حال متدهای فاکتوری می توانند نقش یک مرحله را در یک متد قالب بزرگ ایفا کنند.