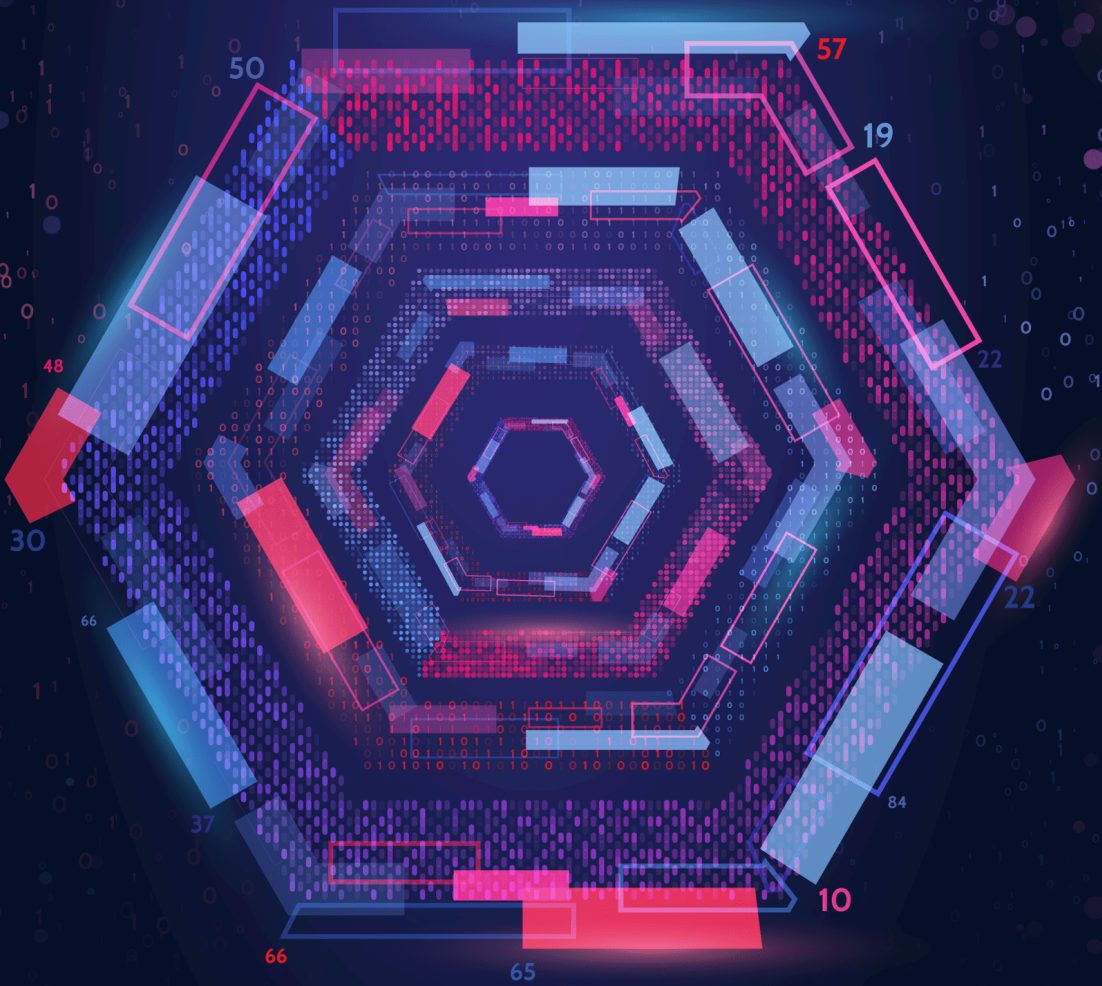
直撃！デザインパターン



Alexander Shvets（アレグザンダー・シュベッツ）

直撃！デザインパターン

v2022-1.0

お試し版

製品版を購入：
https://refactoring.guru/ja/design-patterns/book

著作権について一言

こんにちは。私は、Alexander Shvets（アレグザンダー・シュベッツ）と申します。『直撃！デザインパターン[1]』という本と、『リファクタリング・コース[2]』というオンライン学習コースの著者です。

本書は個人利用に限られます。ご家族以外の方と共用しないようお願いします。本書をお友達や同僚にも使ってほしいと思われましたら、1冊購入して送ってあげてください。チームや会社向けのサイト別ライセンスも購入できます。

本書とオンラインコースの販売から得られる利益は、**Refactoring.Guru**（邦題：リファクタリング グル）の開発に当てられます。一冊の販売から得られる利益は、プロジェクトの促進に大いに役立ち、次の本の出版時期を少しですが早めることができます。

✉ **support@refactoring.guru**

🖼 イラスト：Dmitry Zhart
📖 翻訳：黒坂 輝彦（Kuro Kurosaka）
🖉 校正：スミス さとこ（Satoko Smith）

---

1. 『直撃！デザインパターン』（原題：Dive Into Design Patterns）：**https://refactoring.guru/ja/design-patterns/book**
2. 『リファクタリング・コース』（原題：Dive Into Refactoring）：**https://refactoring.guru/ja/refactoring/course**

イタリアのリッチ家ジョゼッペさん、サラさん、シモーネさん、マリナーさんに本書を捧げます。本書の製作中、私を家族共々暖かく受け入れていただきました。安住の地を提供していただいたことに大変感謝します。おかげで、戦時にもかかわらず、平時のような状況を保つことができました。

Vorrei dedicare questo libro alla grande famiglia italiana dei Ricci, Giuseppe, Sara, Simone e Marina, che hanno condiviso la loro casa con la mia famiglia mentre questo libro prendeva forma. Grazie per averci fornito un appoggio stabile e una parvenza di normalità in tempo di guerra!

目次

本書の読み方

本書は、「Gang of Four」（4 人のギャング。GoF とも略される）によって公式化された 22 種類の伝統的デザインパターンについて説明します。

章ごとに、特定の一つのパターンを見ていきます。ですので、最初から最後まで読むこともできますし、興味のあるパターンの章だけ読むこともできます。

多くのパターンは関連しているので、一つのトピックから別のトピックへ、リンクを通して簡単にジャンプできます。章の最後には、関連パターンへのリンクのリストが収められています。まだ見たこともないパターンの名前に出くわしても、そのまま読み続けてください。後続の章で説明されています。

デザインパターンは、普遍的です。そのため、本書のコード例全部は、特定のプログラミング言語に依存しない、疑似コードで書かれています。

デザインパターン学習前に、**オブジェクト指向プログラミングの主要用語**を読んでおくのもお勧めです。その章では、本書で多用する UML 図の基本も説明してあります。そんなことはもう知っている、と言う方は、**パターンの学習**へ直行してください。

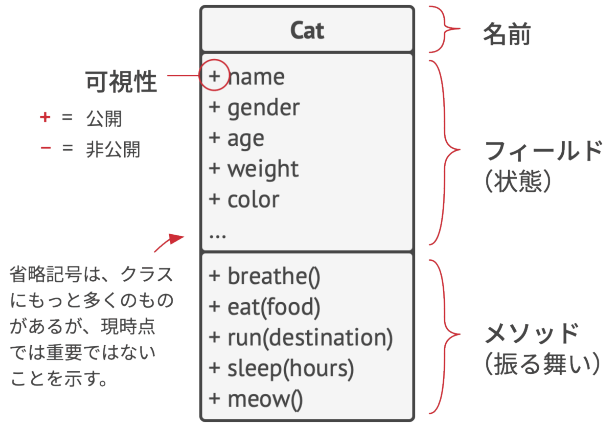
OOP 入門

OOP の基本

***O**bject-**o**riented **p**rogramming*（OOP：オブジェクト指向プログラミング）は、データとそのデータに対する振る舞い（行動）とを包み込み（ラップして）、**オブジェクト**と呼ばれるひとかたまりのモノにする、という考えに基づいた枠組みです。オブジェクトは、プログラマーによって定義された**クラス**と呼ばれる「青写真」で構築されます。

オブジェクトとクラス

猫はお好きですか？そうであってほしいと思っています。というのは、これから OOP の概念を猫を例にとって説明しようと思っているからです。



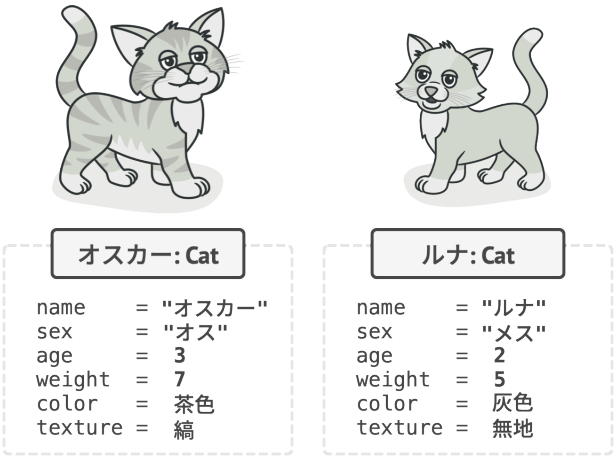
これは UML クラス図です。本書では、こういう図をたくさん見ることになります。

オスカーという名の猫を飼っているとします。オスカーは、`Cat`（猫）クラスのインスタンスです。猫には、名前、年齢、体重、色、好きな食べ物など、たくさんの標準属性があります。これらはクラスのフィールドと呼ばれます。

> この本では、クラス名は、UMLダイアグラムやコードに表示されているのとまったく同じ英語クラス名で参照しますが、わかりやすくするために概念の説明では日本語を使う場合があります。

すべての猫には共通点があります。猫は、息をして、食べて、走り、寝て、ニャーンと鳴きます。これらは、クラスのメソッドとなります。クラスのフィールドとメソッドは合わせてメンバーと呼ばれます。

> オブジェクトのフィールドに保存されたデータは、状態と呼ばれます。オブジェクトのメソッドは振る舞い（行動）を定義します。



オブジェクトは、クラスのインスタンスです。

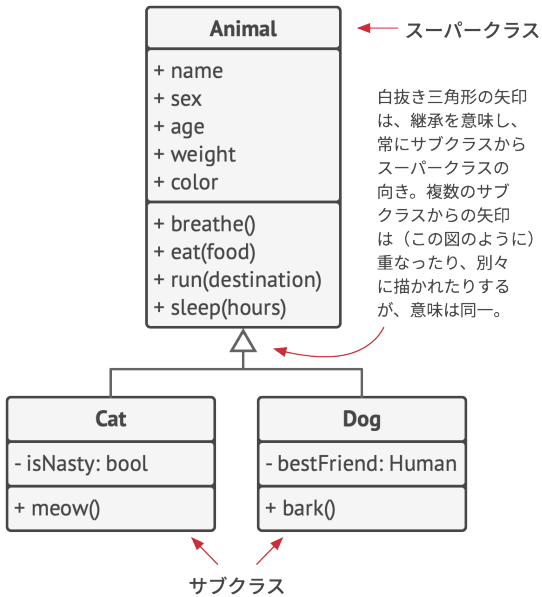
友達の飼っている猫のルナも `Cat` クラスのインスタンスです。オスカーと同じだけの属性があります。しかし属性の値は違います。性別はメス、色は異なり、体重は軽いです。

ということで、クラスは、オブジェクトの構造を定義する青写真のようなものです。オブジェクトは、クラスの具体例、つまりインスタンスです。

クラス階層

一つのクラスについてだけ話す分には、簡単でわかりやすいですが、実際のプログラムにはいくつものクラスがあります。これらのクラスのいくつかは、**クラス階層**の中に位置づけられているかもしれません。どういう意味か説明しましょう。

隣人がフィドという犬を飼っていたとします。よく見ると犬と猫にはいろいろ共通点があります。名前、性別、年齢、色は、犬と猫の両方が持っている属性です。犬は猫と同じように息をして、寝て、走ります。ということは、共通の属性と振る舞いを定義した、`Animal`（動物）という基底クラスを定義することが可能です。

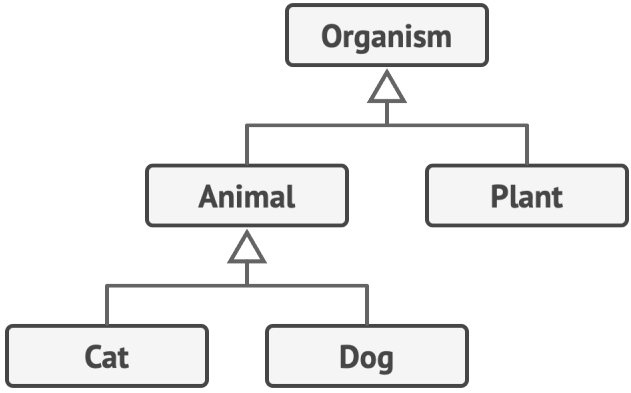


クラス階層の UML 図。この図のすべてのクラスは、`Animal` クラス階層の部分です。

今さっき定義した親クラスは、**スーパークラス**とも呼ばれます。その子は**サブクラス**です。サブクラスは、状態と振る舞いを親クラスから継承し、親と異なる属性と振る舞いだけを定義します。ということで、`Cat` クラスは、`meow`

（ニャーンと鳴く）メソッドを定義し、`Dog` クラスは、`bark`（吠える）メソッドを定義することになるでしょう。

ビジネス上の要求があれば、`Animal` と `Plant`（植物）のスーパークラスとして機能する、すべての生きる `Organism`（有機体）を表現したもっと一般的なクラスを抽出することだってできます。このようなクラスのピラミッド的な関係が、**階層**です。そのような階層では、`Cat` クラスは、`Animal` クラスと `Organism` クラスの両方からすべてを継承します。



UML 図中のクラスは、その中身よりも関係が重要な場合、単純化が可能。

サブクラスは、親クラスから継承したメソッドの振る舞いを上書きすることができます。サブクラスは、デフォルトの振る舞いを完全に置き換えることも、ちょっとした追加で向上させることもできます。

ソフトウェア設計の原則

良い設計の特徴

実際のパターンの話に進む前に、ソフトウェア・アーキテクチャーを設計するプロセスについてお話ししたいと思います。何を目指し、何を避けるかということです。

♺ コードの再利用

開発コストと時間は、あらゆるソフトウェア製品を開発する際の最も価値のある二つの指標です。短時間の開発は、競合他社より早く市場に参入できることを意味します。開発コストを下げることは、より多くの資金をマーケティングと、潜在顧客への訴求に使えることを意味します。

コードの再利用は、開発コストを抑えるための最も一般的な方法の一つです。意図は明らかです。何でも一から開発する代わりに、既存コードを新プロジェクトで使おう、ということです。

このアイデア、机上の論議ではよく見えますが、実際には、既存コードを新しい環境で動作させるには、通常余分な労力がかかります。コンポーネント間の密な結合、インターフェースではなく具象クラスへの依存、ハードコードされた運用などがコードの柔軟性を低減させ、コードの再利用を困難にします。

デザインパターンの適用は、ソフトウェア・コンポーネントの柔軟性を高め、再利用しやすくする一つの方法です。しかし、これには、時としてコンポーネントを複雑化するという代償があります。

デザインパターンの生みの親の一人である、Erich Gamma（エリック・ガンマ）氏[1]による、コード再利用におけるデザインパターンの役割についての賢者の一言をご覧ください：

> 再利用には三つのレベルがあると思うんですよ。
>
> 一番低いレベルは、クラスの再利用ですね。クラスのライブラリー、コンテナ、そうですね、コンテナとかイテレーターと言ったクラスの「チーム」もそうかもしれません。
>
> フレームワークは、最上位に来ます。それらは、あなたの設計上の決定事項を純化しようと試みます。問題を解決するための主要な抽象化事項を特定し、クラスで表現し、その間の関係を定義します。 たとえば JUnit は、 小さなフレームワークですね。フレームワークの「Hello World」のようなものです。 `Test` と `TestCase` と `TestSuite` があり、関係が定義されています。

1. Erich Gamma on Flexibility and Reuse（エリック・ガンマ、柔軟性と再利用について語る）**https://refactoring.guru/ja/gamma-interview**

典型的には、フレームワークはたった一つのクラスに比べると粒度は大きいです。それと、どこかでサブクラスを作り、フレームワークにフックします。「我々に電話かけないでください。こちらから電話します。」という、いわゆるハリウッド原則に従ってですね。フレームワークでは、あなた独自の振る舞いを定義でき、順番が来たらフレームワークがそれを呼び出します。JUnit と同じでしょ？JUnit は、テストを実行したくなったらあなたのテストを呼び出しますが、残りのことはフレームワーク内で起きます。

そして中間のレベルがあります。パターンはここが守備範囲だと思うんですよ。デザインパターンは、フレームワークに比べるとより小さく、より抽象的です。それは、いくつかのクラスに関する記述ですね。それらがお互いにどう関係し、どう関わるかについてです。再利用のレベルは、クラスからパターンへ、そして最後にフレームワークへ移動するにつれ、向上します。

中間層のいいところは、パターンが、フレームワークに比べてリスクが小さい再利用の方法を提供するということです。フレームワークの構築は、リスクが高く、高価な投資になります。パターンを使うと、設計上のアイディアや概念を具体的なコードとは独立して再利用できます。

”

拡張性

変化は、プログラマーの人生で唯一の定めです。

- Windows 用のビデオゲームをリリースしたと思ったら、今度は macOS バージョンが求められる。
- 四角いボタンの GUI フレームワークを作ったら、数か月後には丸ボタンが流行する。
- 素晴らしい E コマースのウェブサイトのアーキテクチャーを設計した 1 か月後には顧客から電話注文機能の要求が来る。

ソフトウェア開発者なら誰でもこの手の話はいくつも経験しています。どうしてこうなるのかには、いくつかの理由があります。

まず、問題を解決し始めると、問題をより深く理解できるようになる、ということがあります。多くの場合、アプリの最初のバージョンが完成する頃になると、問題の多くの点をより深く理解しているので、一から書き直す準備ができています。また、技術者としても進歩しているので、自分の書いたコードがゴミのように見えます。

自分の管理下にない何かが起きることもあります。多くの開発チームが初期のアイディアから何か新規のものに回れ右するのは、このせいです。自分のオンライン・アプリケーションが Flash に頼っていた場合、ブラウザーが次から次と Flash のサポートを止めるにつれ、コードの再開発や移行をした経験があると思います。

三つ目の理由は、目標の移行です。あなたの顧客は、当初アプリケーションの現バージョンをとても喜んでいましたが、元々の計画会議の時口にもしなかった、違うことをするための 11 点の「ちょっとした」変更が、今になって見えてきました。これは、根拠のない変更ではありません。最初の素晴らしいバージョンを使って、もっといろんなことができると気づいたことによる変更要求です。

> 楽観的に眺めると、誰かがアプリの何かの変更を望んでいるということは、アプリのことを気に留めてくれている人がいる、ということです。

熟練した開発者たちが全員、アプリケーション・アーキテクチャーの設計中に、将来起きるであろう変更を容易にできるようにすることを考慮するのは、このためです。

デザインの原則

ソフトウェアのいい設計とは何ですか？どう測定しますか？いい設計を達成するために何に従う必要がありますか？柔軟で安定していて理解しやすいアーキテクチャーをどう作れますか？

これらは素晴らしい質問ですが、残念ながら、作成するアプリケーションの種類によって答えは異なります。ですが、ご自身のプロジェクトのためにこれらの質問に答える助けになるソフトウェア設計のいくつかの普遍的な原則というものがあります。本書記載のデザインパターンのほとんどは、これらの原則に基づいています。

変化するものをカプセル化

> アプリケーション中で変化する側面を特定し、不動のものから分離せよ。

この原則の主目的は、変更の影響を最小にすることです。

自分のプログラムを船に喩え、変更を水面下にただよう魚雷だと思ってください。魚雷にやられると船は沈みます。

これを知っての上で、船体を安全に仕切られた独立した部分に分け、損傷を一部分に限定するようにできます。これで、船が魚雷に接触しても、船全体は浮かんだままでいられます。

同様にして、プログラムの変化する部分を独立したモジュールに分割・隔離し、残りのコードを悪影響から守るようにできます。結果として、変更の実装とテストをしてプログラムを復旧する時間を削減できます。変更に使う時間が減ると、機能の実装に使える時間が増えます。

メソッドのレベルでのカプセル化

E コマースのウェブ・サイトを作っているとしましょう。コードのどこかに、税金を含んだ注文の総計を計算する `getOrderTotal` メソッドがあります。

税関連のコードは将来変更が必要であろうことは予想できます。税率は、顧客の住む国、州、市によっても異なります。新しい法律や規制のため、長い時間の間に、実際の計算式も変更になるかもしれません。結果、`getOrderTotal` を頻繁に変更する必要が出てきます。しかし、メソッド名も示唆する通り、それは、どのように税金を計算するかは気にしません。

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  if (order.country == "US")
    total += total * 0.07 // 米国販売税
  else if (order.country == "EU"):
    total += total * 0.20 // 欧州付加価値税

  return total
```

適用前：税計算コードはメソッドのコード他の部分と混在。

税金計算のロジックを個別のメソッドに抽出し、元のメソッドから隠蔽することができます。

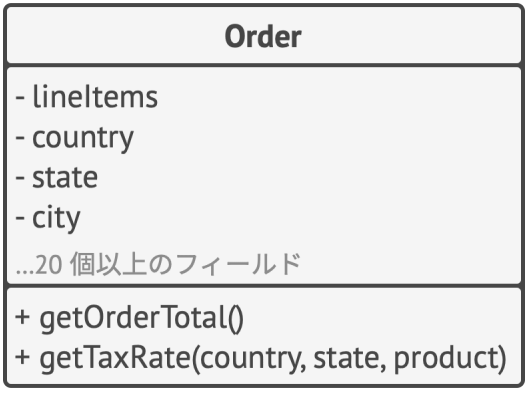
```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  total += total * getTaxRate(order.country)

  return total

method getTaxRate(country) is
  if (country == "US")
    return 0.07 // 米国販売税
  else if (country == "EU")
    return 0.20 // 欧州付加価値税
  else
    return 0
```

適用後：専用メソッドを呼んで税率を取得できる。

税関連の変更は単一のメソッド内に分離されました。さらに、もし税計算のロジックが複雑になりすぎた場合は、別のクラスへ移行することが容易になりました。

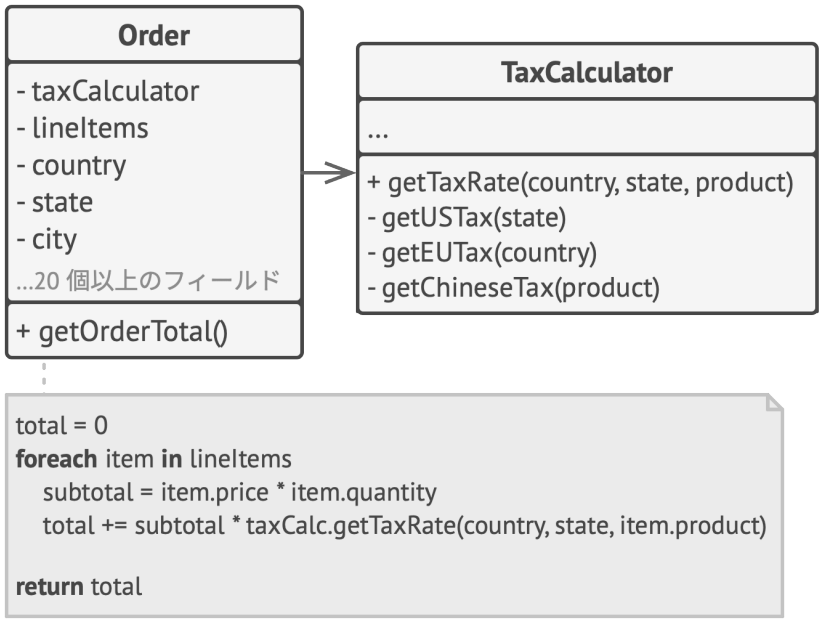
クラスのレベルでのカプセル化

時間が経つにつれて、かつて簡単なことを行っていたメソッドの責任がどんどん増えていくかもしれません。これらの追加された振る舞いの実装には、多くの場合、独自の補助フィールドとメソッドがあり、それを含むクラスの主要な責任があやふやになっていきます。すべてを新しいクラスに抽出すれば、何もかもよりわかりやすく簡単になるかもしれません。



適用前： `Order` クラスの中で税計算。

`Order` クラス中のオブジェクトは、税関連の仕事を、ただそれだけを行う特別なオブジェクトに委任。

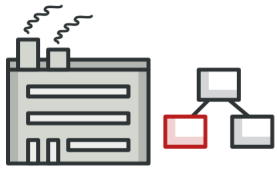


適用後：税計算は、`Order` クラスから隠蔽される。

デザインパターン
カタログ

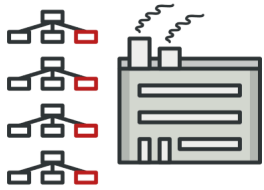
生成に関するデザインパターン

生成に関するデザインパターンは、柔軟性を増し、コードの再利用を促すような、オブジェクト生成の仕組みを提供します。



Factory Method

スーパークラス内でオブジェクトを作成するためのインターフェースを提供しますが、サブクラスでは作成されるオブジェクトの型を変更することができます。

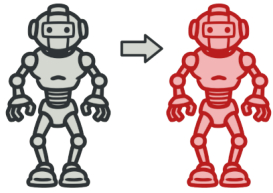


Abstract Factory

関連したオブジェクトの集りを、具象クラスを指定することなく生成することを可能にします。

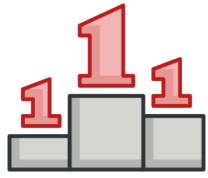
Builder

複雑なオブジェクトを段階的に構築できます。このパターンを使用すると、同じ構築コードを使用して異なる型と表現のオブジェクトを生成することが可能です。



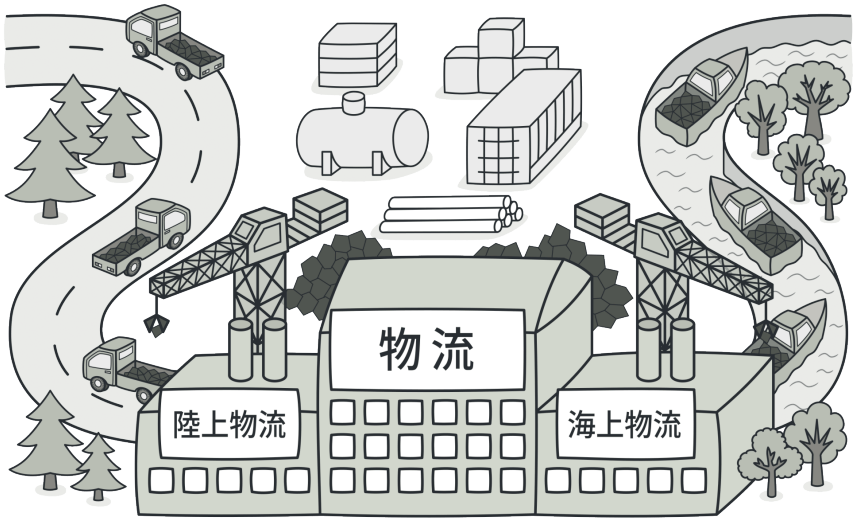
Prototype

既存オブジェクトのコピーをそのクラスに依存することなく可能とします。



Singleton

クラスが一つのインスタンスのみを持つことを保証するとともに、このインスタンスへの大域アクセス・ポイントを提供します。



FACTORY METHOD

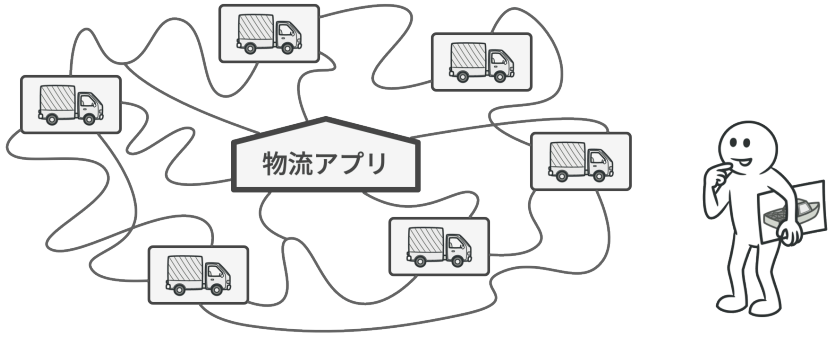
別名：Virtual Constructor、ファクトリー・メソッド、
バーチャル・コンストラクター、仮想コンストラクター

Factory Method（ファクトリー・メソッド）は、生成に関するデザインパターンの一つで、スーパークラスでオブジェクトを作成するためのインターフェースが決まっています。しかし、サブクラスでは作成されるオブジェクトの型を変更することができます。

☹ 問題

物流管理アプリケーションを作成するとします。アプリの最初のバージョンは、トラック輸送のみを処理できます。コードの大部分は `Truck`（トラック）クラス内に存在します。

しばらくして、このアプリは、かなり人気が出ます。海運会社から、アプリに海上物流を組み込んでほしいという要望が毎日数十件寄せられるようになりました。



コードの大部分がすでに既存のクラスと密に結合されていると、新しいクラスのプログラムへの追加はあまり容易ではない。

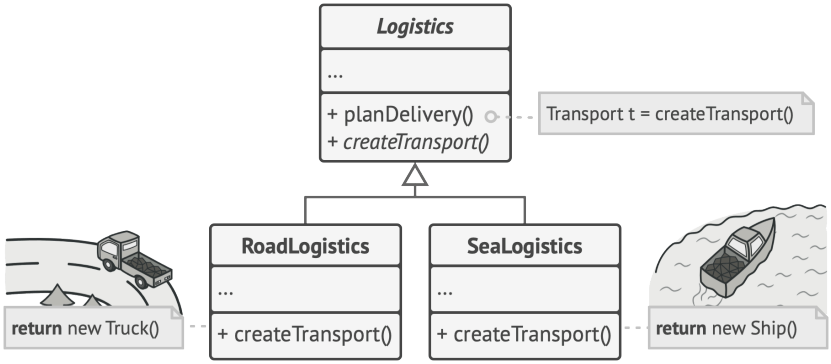
素晴らしいことです！でもコードの方はどうしましょう？現状では、コードのほとんどは、`Truck` クラスに結合されています。アプリに `Ships`（船）を追加するには、コードの大部分に手を入れる必要があります。さらに、後でアプ

リに別の種類の輸送手段を追加することにした場合、おそらくこの変更作業すべてを再度行うことになるでしょう。

結果として、運輸オブジェクトのクラスに応じてアプリの動作を切り替える条件文だらけの、かなり厄介なコードができてしまうでしょう。

😊 解決策

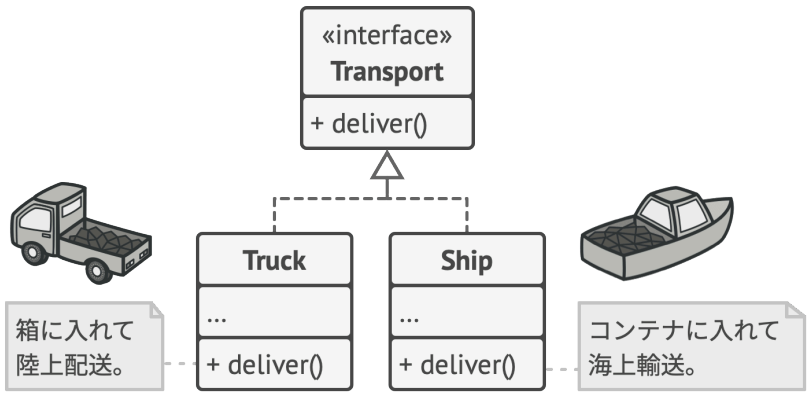
Factory Method パターンに従うと、`new` 演算子を使用した直接のオブジェクト作成呼び出しを、特別なファクトリー・メソッドへの呼び出しで置き換えます。ご心配なく！それでもオブジェクトは、`new` 演算子で作成されます。ただそれはファクトリー・メソッド内で呼び出されます。ファクトリー・メソッドから返されるオブジェクトはよくプロダクトと呼ばれます。



サブクラスは、ファクトリー・メソッドから返されるオブジェクトのクラスを変更できる。

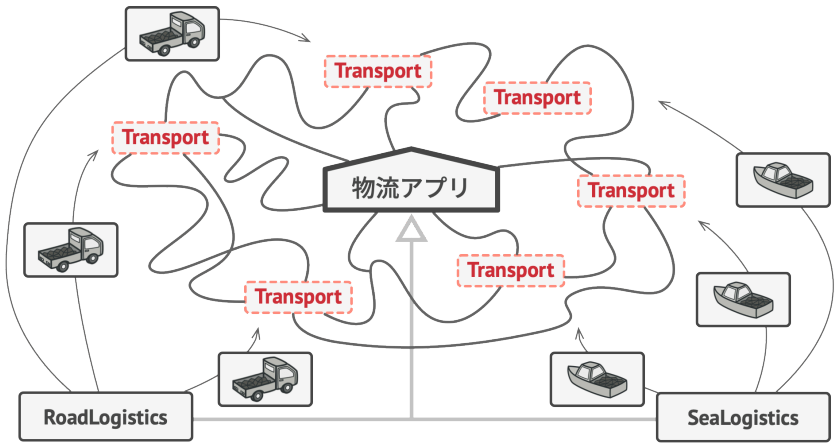
一見すると、この変更は無意味に見えるかもしれません。コンストラクターの呼び出しをプログラムのある部分から別の部分に移動しただけです。しかし、これにより、サブクラスのファクトリー・メソッドを上書きさえすれば作成されるプロダクトのクラスの変更が可能というメリットがあります。

しかし、わずかな制限があります：これらのプロダクトに共通のベースクラスまたはインターフェースがある場合にのみ、サブクラスは、異なる型のプロダクトを返すことができます。また、ベースクラス内のファクトリー・メソッドの戻り値の型は、このインタフェースとして宣言されている必要があります。



すべてのプロダクトは、同じインターフェースに従う必要がある。

たとえば、`deliver`（配送）というメソッドが宣言された`Transport`（運送）というインターフェースを、`Truck`と`Ship`の両クラスが実装します。Truck は陸上で、Ship は海上で貨物を届けるというように、それぞれのクラスでこのメソッドの実装が異なっています。`RoadLogistics`（陸路物流）クラスのファクトリー・メソッドは Truck オブジェクトを返し、`SeaLogistics`（海上物流）クラスのファクトリー・メソッドは Ship を返します。

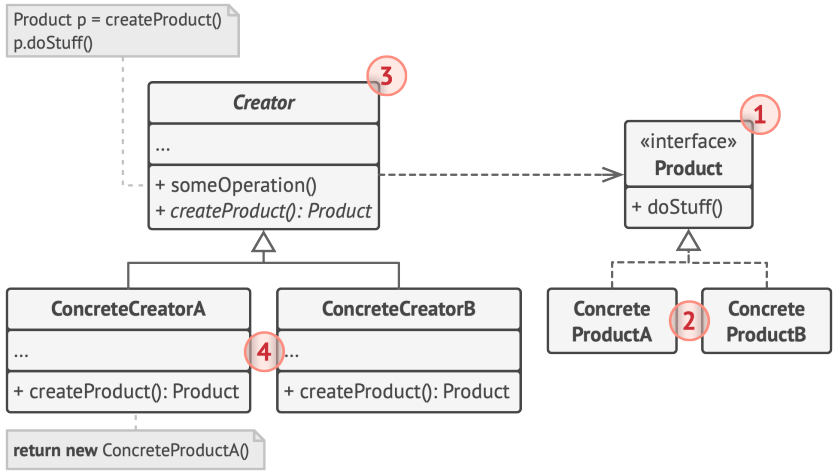


すべてのプロダクト・クラスが共通のインターフェースを実装している限り、どのクラスのオブジェクトでも問題なくクライアント・コードに渡すことができる。

ファクトリー・メソッドを使用するコード（クライアントコードとよく呼ばれる）からは、様々なサブクラスが返す実際のプロダクトの間に違いは見られません。クライアントはすべての製品を抽象的な`Transport`として扱いま

す。クライアントは、すべての `Transport` オブジェクトが `deliver` メソッドを持っていることは知っていますが、それが厳密にどのように振る舞うかは、クライアントにとっては重要なことではありません。

構造



1. **プロダクト**（Product）は、クリエーターとそのサブクラスによって生成されるすべてのオブジェクトに共通なインターフェースを宣言します。

2. **具象プロダクト**（Concrete Product）は、プロダクトのインターフェースの種々の異なる実装です。

3. **クリエーター**（Creator）クラスは、新しいプロダクトのオブジェクトを返すファクトリー・メソッドを宣言します。

このメソッドの戻り値の型がプロダクトのインターフェースと一致していることが要点です。

ファクトリー・メソッドを `abstract` と宣言して、すべてのサブクラスに独自のメソッドの実装を強制することができます。代わりの方法としては、基底クラスのファクトリー・メソッドで、何らかのデフォルトのプロダクト型を返すようにもできます。

その名前にもかかわらず、プロダクトの作成はクリエーターの主要任務では**ありません**。通常、クリエーター・クラスはすでにプロダクトに関連するいくつかの中核となるビジネス・ロジックを持っています。ファクトリー・メソッドは、このロジックを具象クラスから分離するのに役立ちます。比喩を使うとこういうことです：大規模ソフトウェア開発会社にはプログラマーのための研修部門があるが、会社全体の主な機能は、コードを書くことであって、プログラマーを育成することではない。

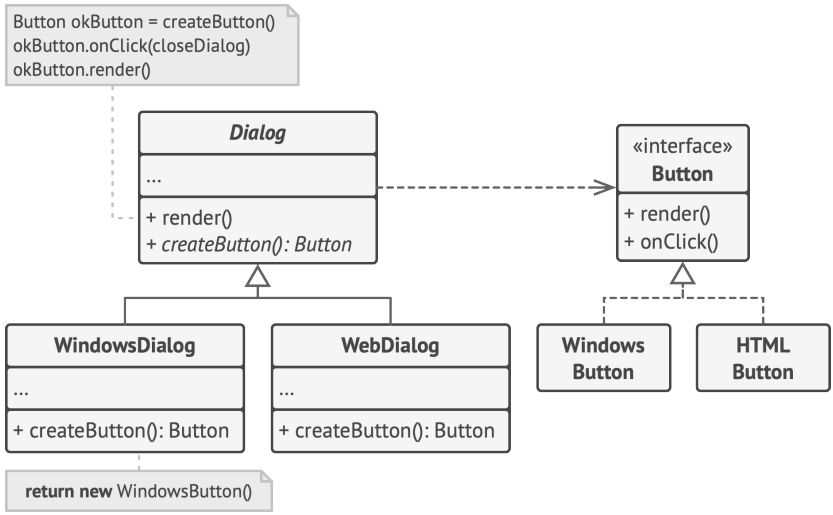
4. **具象クリエーター**（Concrete Creator）は、異なる型のプロダクトを返すように、基底クラスのファクトリー・メソッドを上書きします。

   ファクトリー・メソッドは、常に新しいインスタンスを**作成**する必要はないことに注意してください。キャッシュ、

オブジェクト・プール、その他の方法で既存のオブジェクトを返してもかまいません。

＃ 擬似コード

この例では、クライアント・コードを具体的な UI クラスと結合することなく、プラットフォーム互換な UI 部品を作成するために、**Factory Method** を適用する方法を説明します。



プラットフォーム互換ダイアログの例

基底クラスである `Dialog`（ダイアログ）クラスは、ウィンドウを描画するために異なる UI 部品を使用します。オペレーティングシステムよって、これらの UI 部品は少し異なって見えるかもしれませんが、それでも一貫して動作す

る必要があります。Windows 上でのボタンは、Linux 上でもボタンです。

Factory Method を適用すると、オペレーティングシステムごとにダイアログのロジックを書き直す必要はありません。ダイアログの基底クラス内でボタンを生成するファクトリー・メソッドを宣言しておけば、ファクトリー・メソッドから Windows 形式のボタンを返すようにした、ダイアログのサブクラスを後で作成できます。サブクラスは基底クラスからダイアログのコードの大部分を継承しますが、ファクトリー・メソッドのおかげで、Windows のボタンをスクリーン上に描画できます。

このパターンが機能するためには、ダイアログの基底クラスは抽象的ボタン（すべての具象的ボタンが従う基底クラスかインターフェース）と機能する必要があります。こうしておけば、どのような種類のボタンであっても、ダイアログのコードは機能し続けます。

勿論このやり方は、他の UI 部品にも適用できます。ただし、ダイアログに新しいファクトリ・メソッドを追加するたびに、**Abstract Factory**（抽象ファクトリー）に近いものとなっていきます。でも大丈夫です。このパターンについては後で説明します。

```
// クリエーター・クラスは、ファクトリー・メソッドを宣言。これは、プロダク
// トのクラスのオブジェクトを返さなければならない。クリエーターのサブクラ
// スは通常このメソッドを実装する。
class Dialog is
  // クリエーターは、また、ファクトリー・メソッドの何らかのデフォルトの
  // 実装を提供してもよい。
  abstract method createButton():Button

  // その名前にもかかわらず、プロダクトの作成はクリエーターの主要任務で
  // はない。通常、クリエーター・クラスにはすでにファクトリー・メソッド
  // から返されるプロダクト・オブジェクトに依存した何らかの中核となるビ
  // ジネス・ロジックが存在している。サブクラスは、ファクトリー・メソッ
  // ドを上書きし、異なる種類のプロダクトを返すようにすることで、間接的
  // にビジネス・ロジックを変更可能。
  method render() is
    // プロダクト・オブジェクトを作成するためにファクトリー・メソッド
    // を呼ぶ。
    Button okButton = createButton()
    // そして今、そのプロダクトを使用。
    okButton.onClick(closeDialog)
    okButton.render()


// 具象クリエーターはファクトリー・メソッドを上書きして、結果のプロダクト
// の型を変更。
class WindowsDialog extends Dialog is
  method createButton():Button is
    return new WindowsButton()

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

```
33
34
35  // プロダクト・インターフェースは、全具象プロダクトが実装しなければならな
36  // い操作を宣言。
37  interface Button is
38    method render()
39    method onClick(f)
40
41  // 具象プロダクトは、プロダクト・インターフェースのさまざまな実装を行う。
42  class WindowsButton implements Button is
43    method render(a, b) is
44      // ボタンを Windows 風に描画。
45    method onClick(f) is
46      // OS 本来のクリック・イベントと結びつける。
47
48  class HTMLButton implements Button is
49    method render(a, b) is
50      // ボタンを HTML で表現したものを返す。
51    method onClick(f) is
52      // ウェブブラウザーのクリック・イベントと結びつける。
53
54
55  class Application is
56    field dialog: Dialog
57
58    // アプリケーションは、現在の構成または環境設定に依存して、クリエー
59    // ターの型を決定。
60    method initialize() is
61      config = readApplicationConfigFile()
62
63      if (config.OS == "Windows") then
64        dialog = new WindowsDialog()
```

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

  // クライアント・コードは、基底インターフェースを通してではあるが、具
  // 象クリエーターのインスタンスに対して動作する。クライアントが基底イ
  // ンターフェースを介してクリエーターと作業を続けている限り、いかなる
  // クリエーターのサブクラスでも渡すことが可能。
  method main() is
    this.initialize()
    dialog.render()
```

適応性

コードが機能する対象のオブジェクトの正確な型と依存関係が前もってわからない場合に、Factory Method を使用します。

Factory Method では、プロダクト作成のコードを実際にプロダクトを使用するコードから分離します。したがって、プロダクト作成のコードの拡張が残りのコードから独立して簡単に行えます。

たとえば、アプリに新しいプロダクトの型を追加するには、新しい作成クラスのサブクラスを作成し、その中のファクトリー・メソッドを上書きするだけですみます。

自分の書いたライブラリーやフレームワークのユーザーに内部のコンポーネントを拡張する方法を提供したい場合、Factory Method を使用します。

ライブラリーやフレームワークのデフォルト動作を拡張する最も簡単な方法は、おそらく継承です。しかし、フレームワークは、標準コンポーネントの代わりにサブクラスを使用すべきだということをどう認識するのでしょうか？

解決策は、フレームワーク中に散らばった、コンポーネント構築コードを削減し、単一のファクトリー・メソッドに集めることです。こうすれば、コンポーネント自体を拡張することに加えて、誰でもこのメソッドを上書きできます。

それでは、それがどううまくいくのか見てみましょう。あるオープン・ソースの UI フレームワークを使ってアプリを書くことを想像してみてください。アプリには丸いボタンが必要ですが、フレームワークは四角いボタンしか提供していません。そこで、標準の `Button` クラスを拡張して、輝かしい `RoundButton` サブクラスを作成します。しかし、ここで、デフォルトのボタンの代わりに新しいボタンのサブクラスを使用するように、メインの `UIFramework` クラスに伝える必要が出てきます。

そのためには、フレームワークの基底クラスから `UIWithRoundButtons`（丸いボタンで UI）というサブクラ

スを作り、その `createButton` メソッドを上書きします。このメソッドは基底クラスでは `Button` オブジェクトを返しますが、サブクラスでは `RoundButton` オブジェクトを返すようにします。ここで、`UIFramework` の代わりに `UIWithRoundButtons` クラスを使用してください。それだけです！

毎回再構築する代わりに、既存オブジェクトを再利用してシステム資源を節約したい場合に、Factory Method を使用します。

データベース接続、ファイルシステム、ネットワーク資源等、資源を大量に消費するオブジェクトを扱う場合に、このような状況によく直面します。

オブジェクトの再利用のために何をしなければいけないか考えてみましょう。

1. まず、作成されたすべてのオブジェクトを追跡するための記録場所を作成する必要があります。
2. 誰かがオブジェクトを要求してきたら、プログラムはそのプール（蓄積地）内の空きオブジェクトを探す必要があります。
3. そして、クライアントのコードに対して見つかったオブジェクトを返します。

4. 再使用可能なオブジェクトがない場合、プログラムは新しいオブジェクトを作成し、そしてプールに追加する必要があります。

   これは結構な量のコードになります！重複したコードでプログラムを汚染しないように、このコードは一箇所に納めるべきです。

   おそらく、このコードを置くべき最も明白で便利な場所は、オブジェクトの再利用をしたいクラスのコンストラクター内です。しかし、コンストラクターは定義により常に **新規オブジェクト**を返さなければなりません。既存インスタンスを返すことはできません。

   というわけで、新規オブジェクト作成も既存オブジェクトの再利用もできる通常メソッドが必要となります。それ、ファクトリー・メソッドのように聞こえますね。

実装方法

1. すべてのプロダクトが同じインターフェースに従うようにします。このインターフェースでは、すべてのプロダクトにとって意味のあるメソッドを宣言する必要があります。

2. クリエーター・クラス内に空のファクトリー・メソッドを追加します。メソッドの戻り値の型は、共通のプロダクト・インターフェースと一致する必要があります。

3. クリエーターのコード内で、プロダクトのコンストラクターの参照を全部探し出します。プロダクト作成コードをファクトリー・メソッドに抽出しながら、一つずつ、コンストラクター呼び出しをファクトリー・メソッドへの呼び出しで置き換えていきます。

   ファクトリー・メソッドに、プロダクトの戻り値の型を決めるためのパラメーターを一時的に追加する必要があるかもしれません。

   この時点では、ファクトリー・メソッドのコードはあまりきれいなものではないかもしれません。どのプロダクトのクラスをインスタンス化するかを選択するかを決める巨大な `switch` 文の塊になっているかもしれません。しかし、これはすぐ修正するので、心配は無用です。

4. 次に、ファクトリー・メソッドに並んでいるプロダクトの型ごとに、クリエーターのサブクラスを作成します。基底クラスのファクトリー・メソッドの構築コードの該当部分を抽出して、それでサブクラスのファクトリー・メソッドを上書きします。

5. プロダクトの型が多すぎて、すべての型に応じたサブクラスを作成することがあまり現実的ではない場合は、基底クラスに追加した制御パラメーターを再利用できます。

   たとえば、次のようなクラスの階層があるとします：基底クラスである `Mail`（郵便）クラスとそのサブクラス、`AirMail`（航空便）と `GroundMail`（陸送便）、`Transport`（運輸）クラスとして `Plane`（飛行機）、`Truck`（トラック）、`Train`（鉄道）。`AirMail` クラスは `Plane` オブジェクトのみを使用しますが、`GroundMail` は `Truck` と `Train` 両方のオブジェクトを扱うことができます。両方のケースを扱うために新しいサブクラス、たとえば `TrainMail`（鉄道便）を作ることもできます。が、もう一つ選択肢としては、クライアント・コードが `GroundMail` クラスのファクトリー・メソッドを引数として渡して、どちらのプロダクトを受け取りたいかを伝えるようにすることです。

6. すべての抽出作業の後、基底クラスのファクトリー・メソッドが空になった場合は、そのクラスを抽象クラスとすることができます。何か残った場合は、それをメソッドのデフォルト動作とすることができます。

⚖ 長所と短所

✓ クリエーターと具象プロダクトとの密な結合を回避。

✓ 単一責任の原則。プロダクト作成コードがプログラム中の一箇所にまとめられ、保守が容易。

✓ 開放閉鎖の原則。プロダクトの新しい型をプログラムに導入しても、既存のクライアント・コードの機能に影響しない。

✕ 本パターンの適用では、多数の新規サブクラス導入の必要があり、コードの複雑化の恐れあり。既存クリエーター・クラスの階層にこのパターンを適用する場合、最善の結果が得られる。

⇄ 他のパターンとの関係

- 多くの設計は、まず比較的単純でサブクラスによりカスタマイズ可能な、**Factory Method** から始まり、次第に、もっと柔軟だが複雑な **Abstract Factory** や **Prototype** や **Builder** へと発展していきます。

- **Abstract Factory** クラスは、多くの場合 **Factory Methods** の集まりですが、**Prototype** を使ってメソッドを書くこともできます。

- **Factory Method** を **Iterator** と一緒に使って、コレクションのサブクラスが、コレクションと互換な、異なる型のイテレーターを返すようにできます。

- **Prototype** は継承に基づいていないので、継承の欠点はありません。一方、*Prototype* は、クローンされたオブジェクトの複雑な初期化が必要となります。**Factory Method** は継承に基づいていますが、初期化のステップは必要ありません。

- **Factory Method** は、**Template Method** の特別な場合です。同時に、*Factory Method* は、大きな *Template Method* の一つのステップとして使うこともできます。