デザインパターンの概要

デザインパターンとは

- 過去のソフトウェア設計者が発見し、蓄積してきた設計のノウハウ集
  - オブジェクト指向プログラミングにおいて、よく出会う問題とその解決策がまとめられている

- 一般的にはGang of Four(GoF)と呼ばれる4人の開発者によってまとめられた23種の設計パターン
  - 1995年に「オブジェクト指向における再利用のためのデザインパターン」という書籍を発行

https://www.amazon.co.jp/オブジェクト指向における再利用のためのデザインパターン-エリック-ガンマ/dp/4797311126

デザインパターンの分類

- オブジェクトの生成に関するパターン
  - Abstract Factory、Builder、Factory Method、Prototype、Singleton
- プログラムの構造に関するパターン
  - Adapter、Bridge、Composite、Decorator、Facade、Flyweight、Proxy
- オブジェクトの振る舞いに関するパターン
  - Chain of Responsibility、Command、Interpreter、Iterator、Mediator、Memento、Observer、State、Strategy、Template Method、Visitor

デザインパターンを学ぶ意義

- ✓ よくある問題のベストプラクティスを学ぶことができる
- ✓ 再利用性の高い柔軟な設計ができる
- ✓ 開発者同士の共通言語になる
- ✓ オブジェクト指向をより深く理解できる

デザインパターンを学ぶ上での注意点

× 無理にでもデザインパターンに当てはめようしようとしない

- デザインパターンは規則やルールではなく、あくまでノウハウ集

× 使い方を間違えると逆に複雑な設計になることもある

- 非常に小さく、再利用がされないようなプログラムにとっては過剰な設計

TS

本章の進め方

1. スライドを使った講義
    i)概要
    ii)構成要素
    iii)オブジェクト指向的要素
    iv)メリット・デメリット
    v)使い所
    vi)適用例

2.講義の内容をコーディング
    i)パターンの適用例をコーディング

Template Method

Templateとは

「型板」や「雛形」といった意味を持つ英単語

例）スライドのテンプレート、メールのテンプレート

Templateを使うメリット

✓ 類似したものを簡単に作ることができる

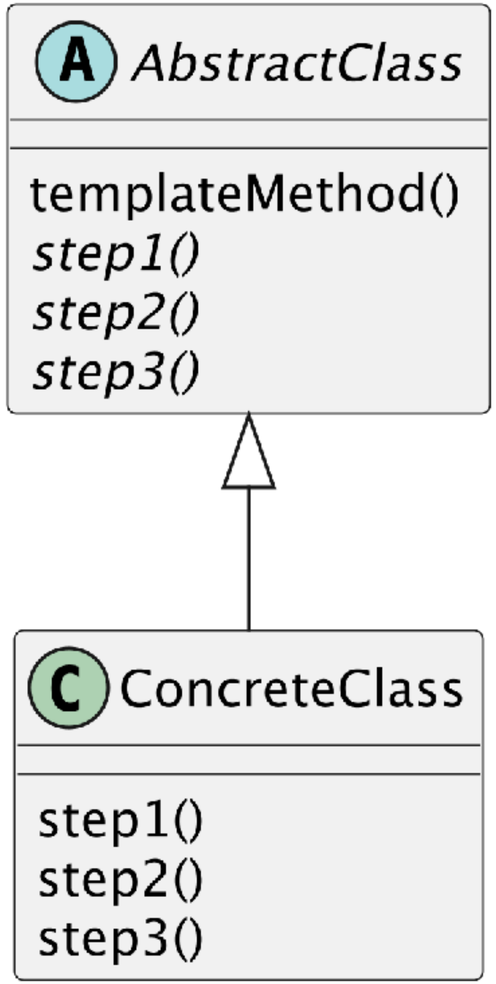
✓ 具体の内容は変更することができる

TS

Template Methodとは

- 親クラスで処理の枠組みを定め、子クラスで枠組みの具体的な内容を定めるようなパターン
  - 処理フローがほとんど同じで、その中の一部の処理内容が異なる場合に、その異なる処理の部分だけを子クラスに実装させる
- 振る舞いに関するデザインパターン

Template Methodの構成要素



AbstractClass

- 抽象クラス
- 処理全体の流れを決定するテンプレートメソッド
- テンプレートメソッドで使用する抽象メソッド

ConcreteClass

- AbstractClassを継承したクラス
- AbstractClassで定義された抽象メソッドを実装

Template Methodのオブジェクト指向的要素

- 「継承」を利用したパターン
  - Concrete ClassがAbstract Classを継承する
  - 抽象クラスを継承することで、抽象メソッドの実装を強制することができる
  - 抽象メソッドの実装を変えることで、子クラスごとに異なる処理フローを実現する

Template Methodのメリット・デメリット

✓ 共通な処理を親クラスにまとめることができる

✓ 処理全体の流れは変えずに、子クラスごとに一部の処理内容を変えることができる

× 処理全体の流れが親クラスに決められるので、子クラスの拡張が制限される

× 子クラスで親クラスのメソッドの振る舞いを変えてしまうとリスコフの置換原則に違反する

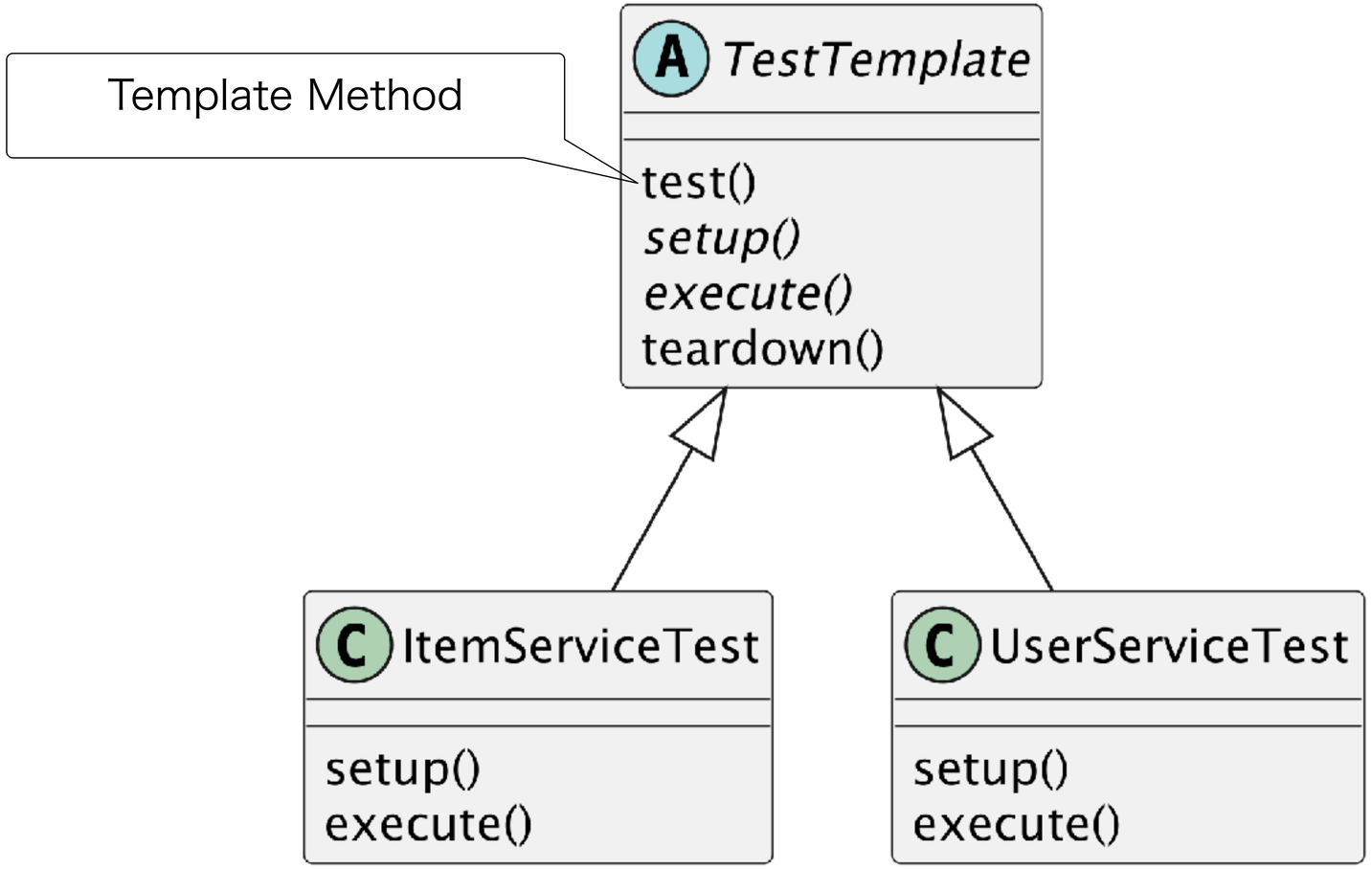
TS

Template Methodの使い所

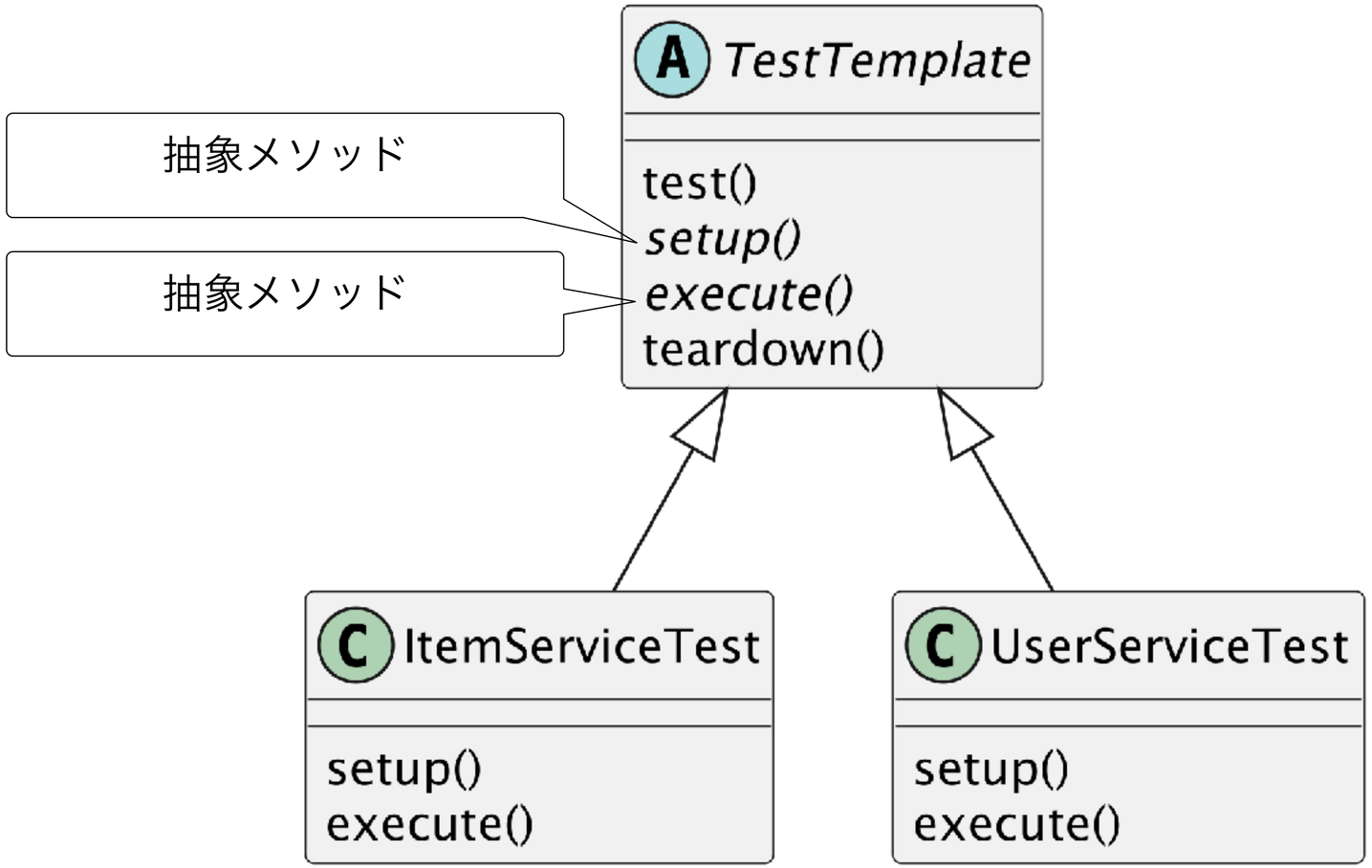
- 処理フローの全体構造は変えることなく、処理の一部のみを変更したい場合
  - 例）テスト
    - setup -> execute -> teardown
- 多少の違いはあるが、ほぼ同一の処理を持つクラスが複数ある場合
  - 同一の部分を親クラスにまとめて重複を排除できる

TS

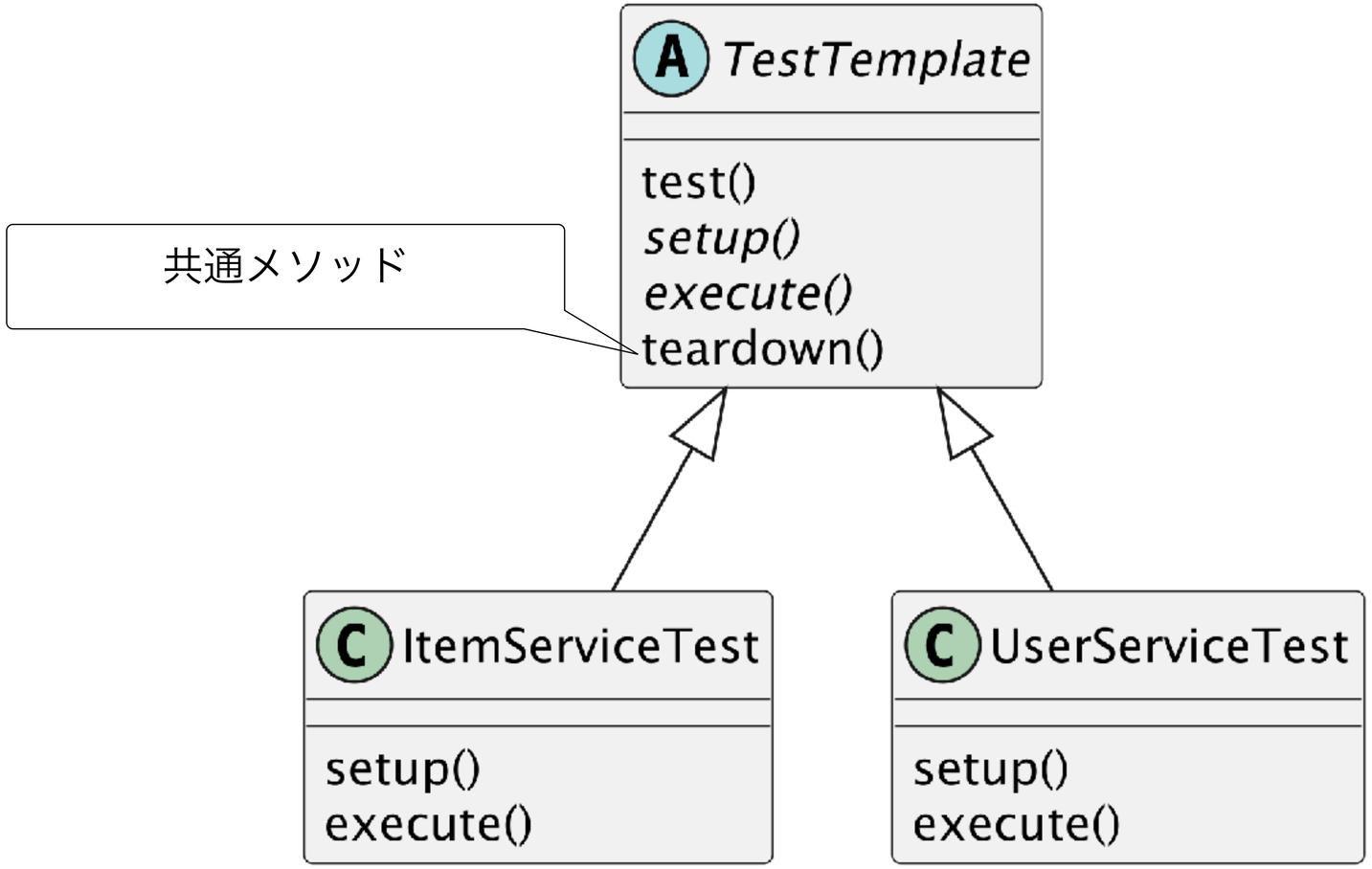
Template Methodの適用例



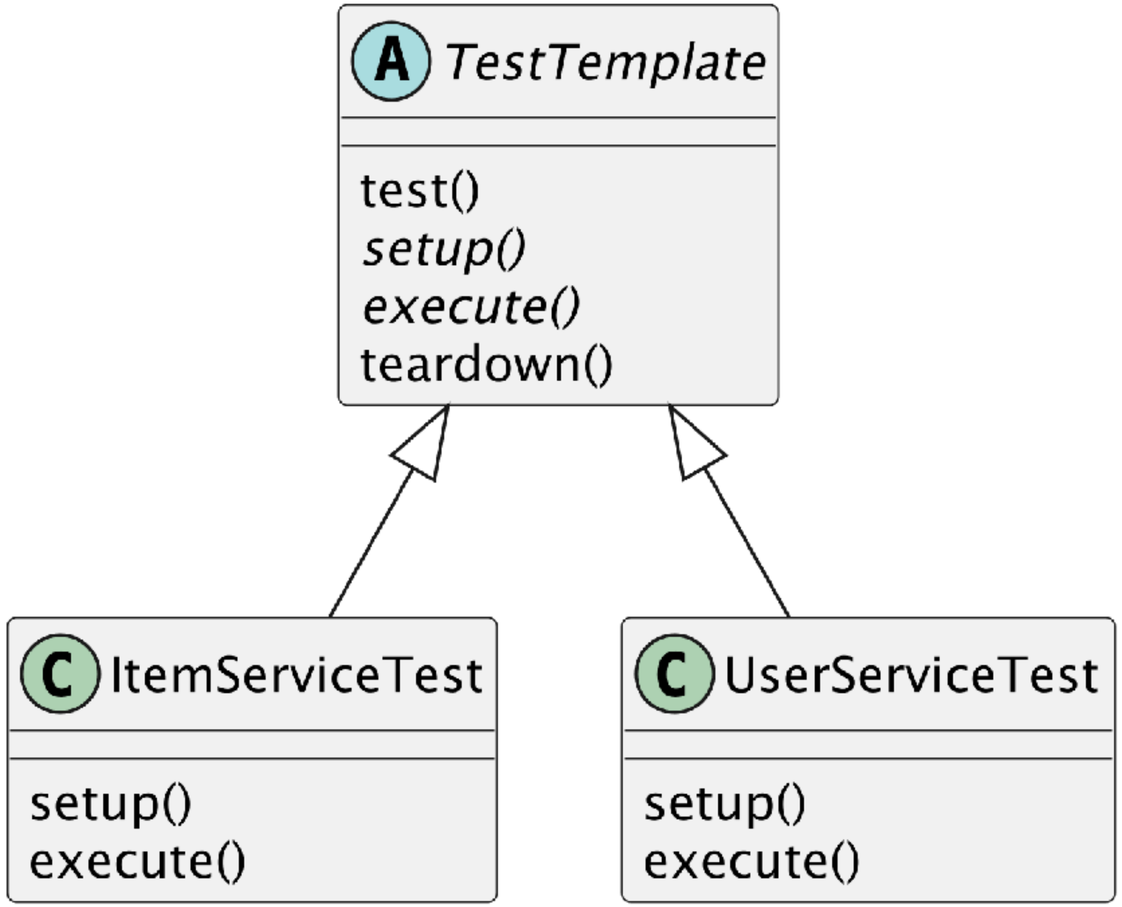
Template Methodの適用例



Template Methodの適用例



Template Methodの適用例



Singleton

Singletonとは

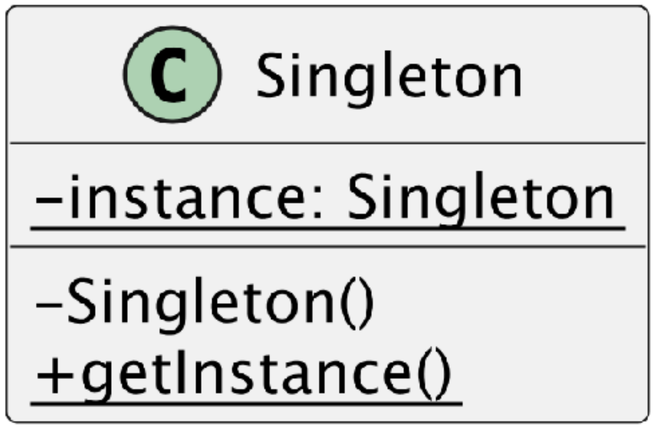
- クラスが一つのインスタンスのみを持つことを保証し、このインスタンスへアクセスするためのグローバルな方法を提供するパターン
  - 開発者は一度しかnewしてはならないといったことを気にしなくて良い
- 生成に関するデザインパターン

※Singletonはアンチパターンと言われることが多い

- GoFの一人、ErichもSingletonをパターンから削除することに賛成
- https://www.informit.com/articles/article.aspx?p=1404056

TS

Singletonの構成要素



Singleton

- privateなコンストラクタ
- 唯一のインスタンスを得るためのstaticメソッド
- いつも同じインスタンスを返却する

Singletonのオブジェクト指向的要素

- 「カプセル化」を利用したパターン
  - 自分自身のインスタンスを内部に保持して管理
  - 他のクラスからのインスタンスへのアクセス方法も提供している
    - この方法でしか外部からはアクセスできない

Singletonのメリット・デメリット

- ✓ クラスが1つのインスタンスしか持たないことを保証できる
- ✓ インスタンスが1つなのでメモリ効率が良い
- ✓ 最初にインスタンス化されたら以降は使い回しなので、生成コストがかからない

- × 依存関係が非常にわかりにくくなる
- × Singletonが状態を保つ場合、密結合になる
- × 単体テストの実行が困難
- × マルチスレッドでのSingletonの扱いが難しい

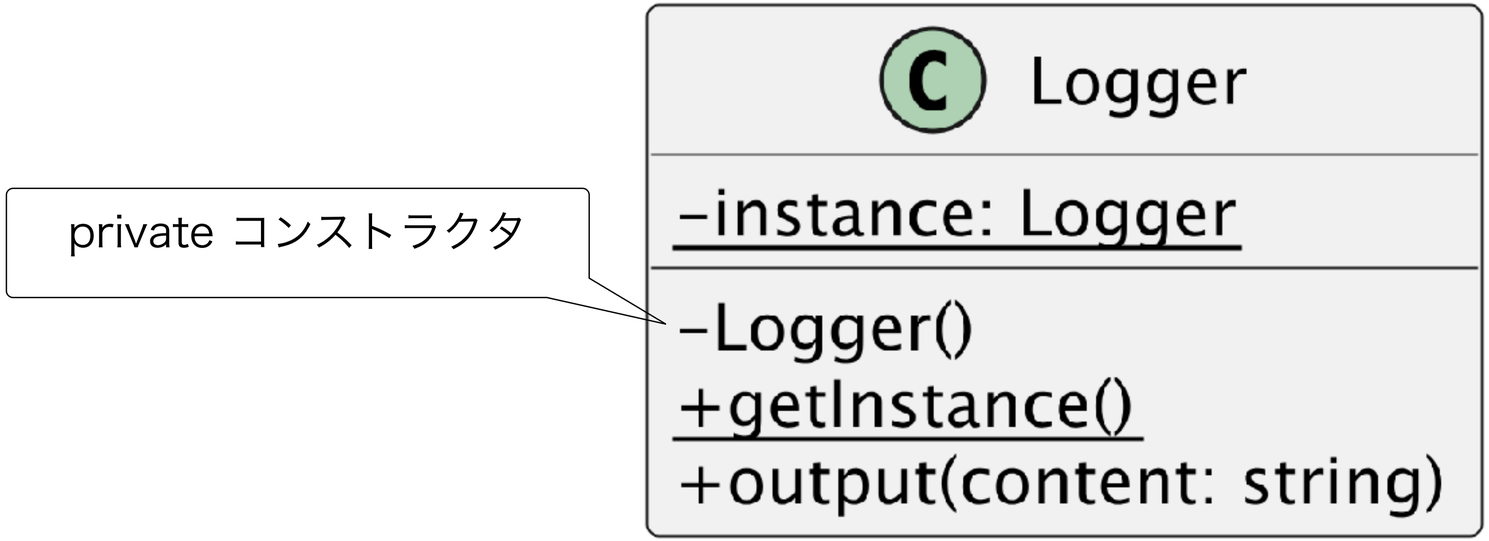
TS

Singletonの使い所

- プログラム内のクラスで、全てのクライアントが使用できるインスタンスを必ず1つだけに制限したい場合
  - ロギング
  - キャッシュ管理
  - コンフィグ
  - データベース接続ドライバ

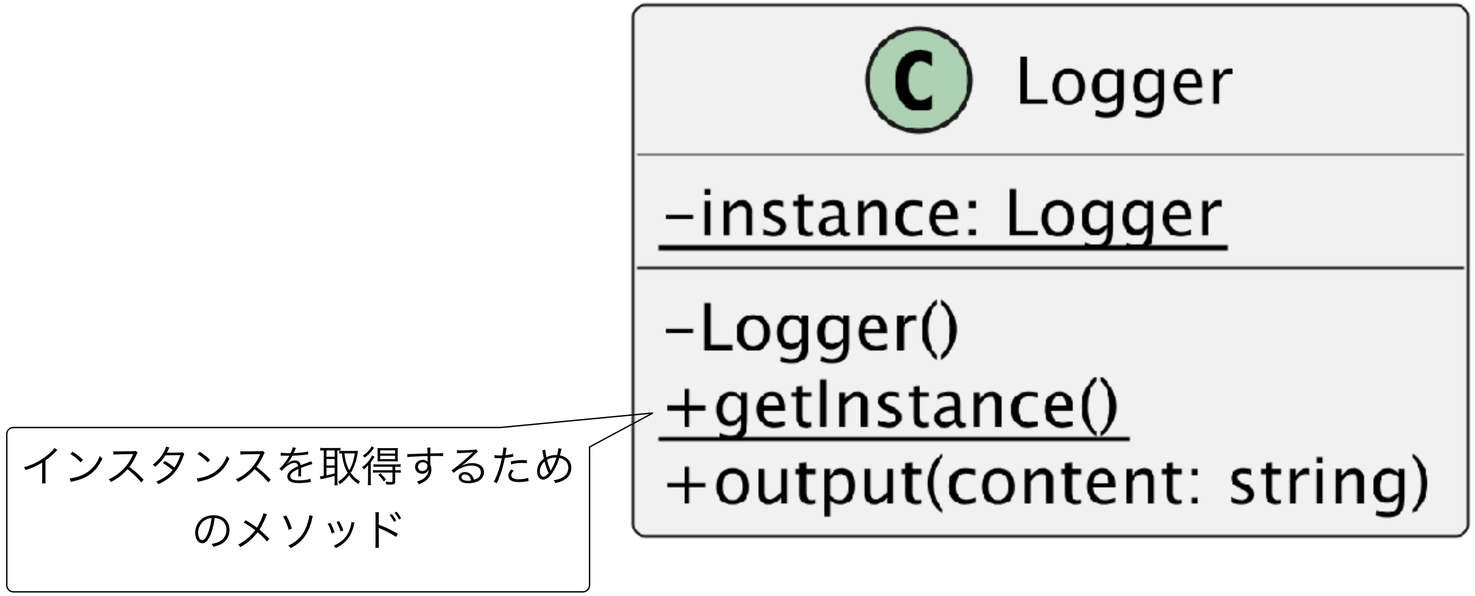
Singletonの適用例

- Loggerクラス



Singletonの適用例

- Loggerクラス



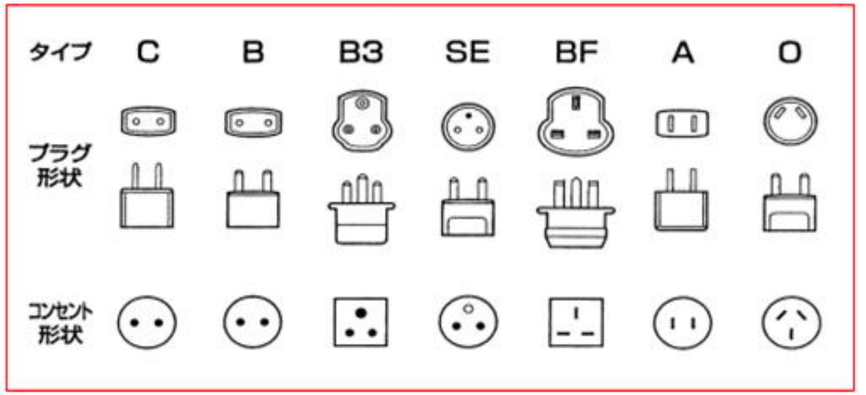
Adapter

Adapterとは

- あるクラスのインターフェースを、そのクラスを利用する側が求める他のインターフェースへ変換するパターン
  - インターフェースに互換性のないクラス同士を組み合わせることができる

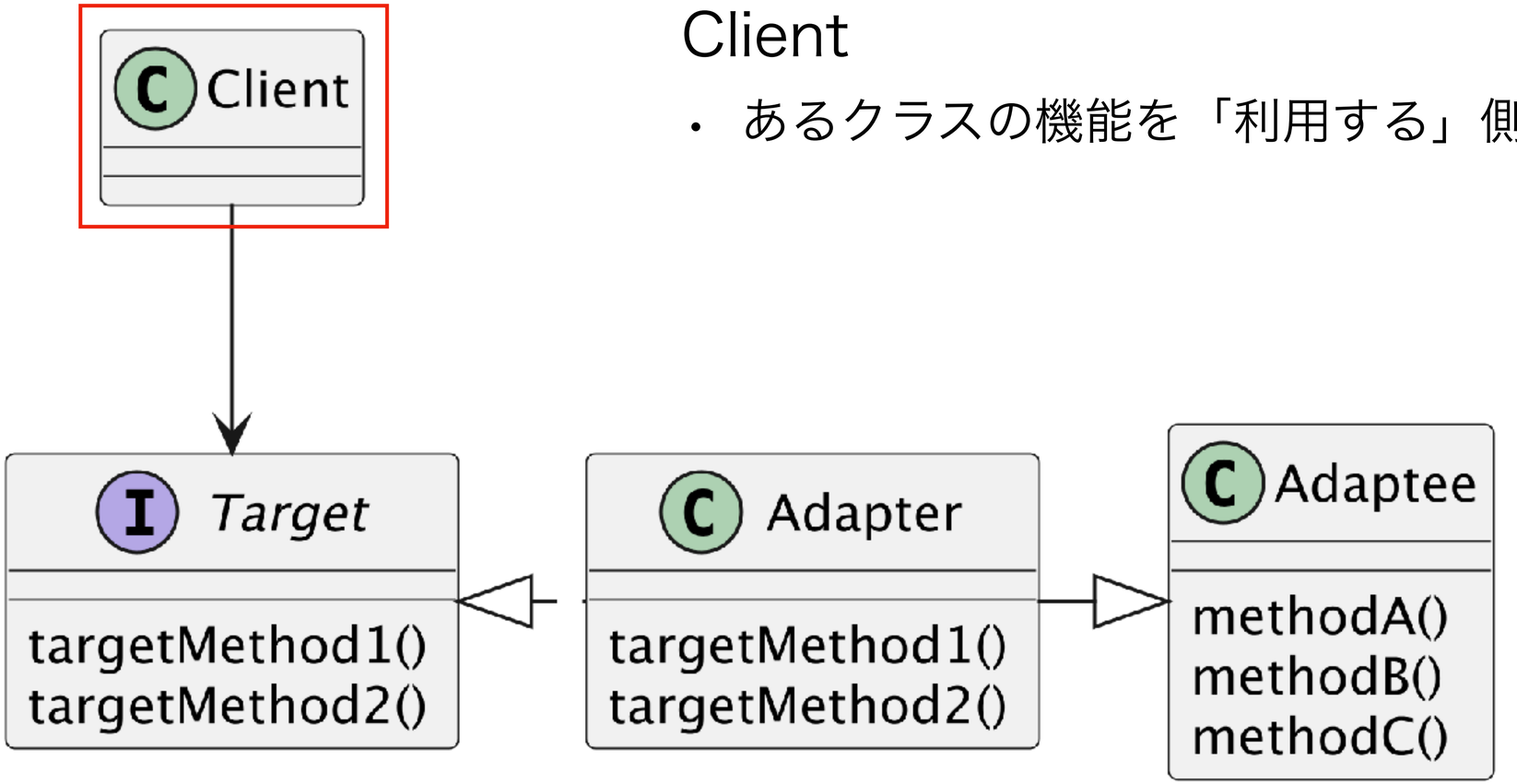
  例）日本と海外のコンセントの変換アダプター

- 構造に関するデザインパターン



http://www.yamagatadenki.com/mame4.html

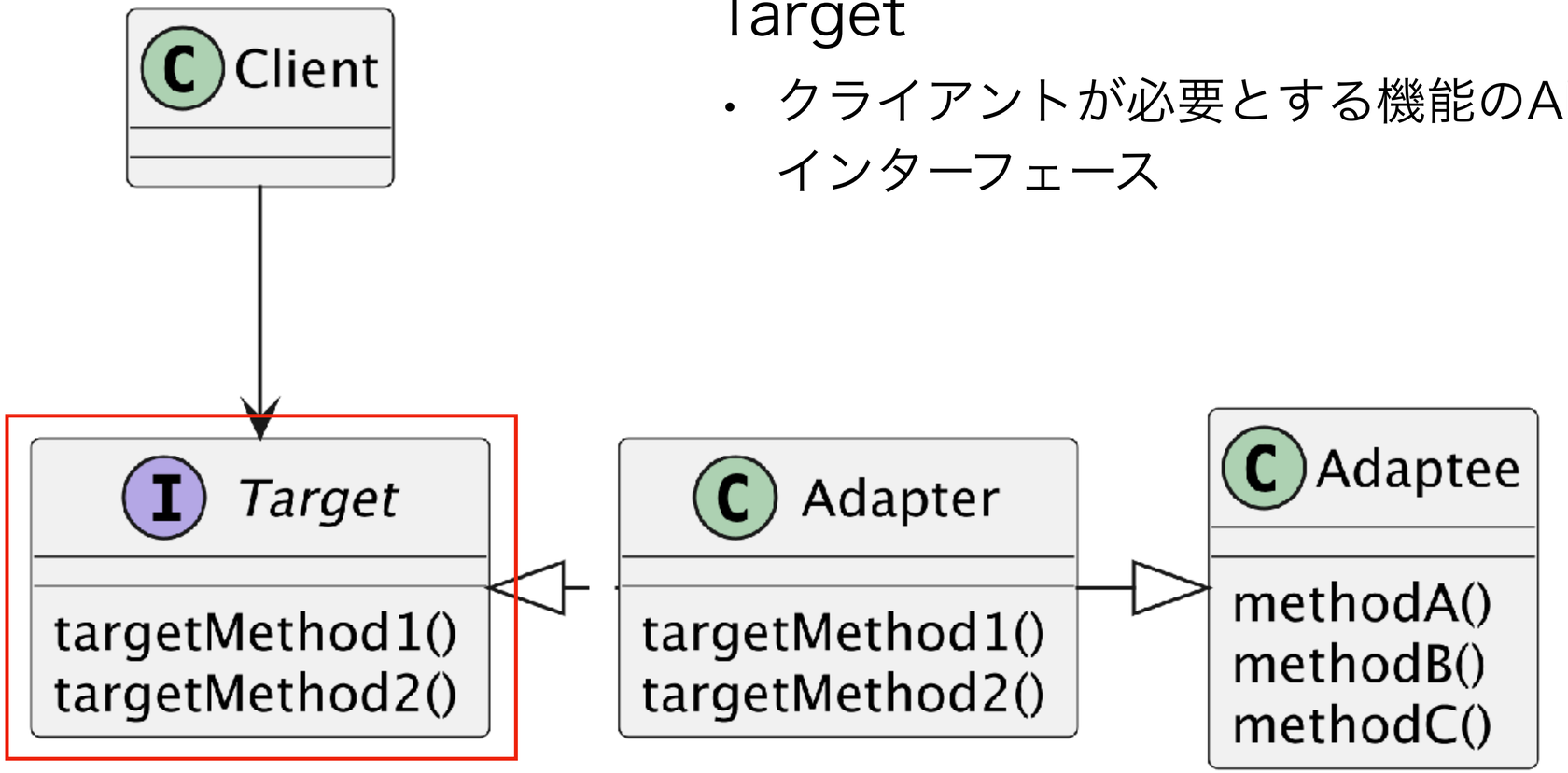
Adapterの構成要素



Client

- あるクラスの機能を「利用する」側のクラス

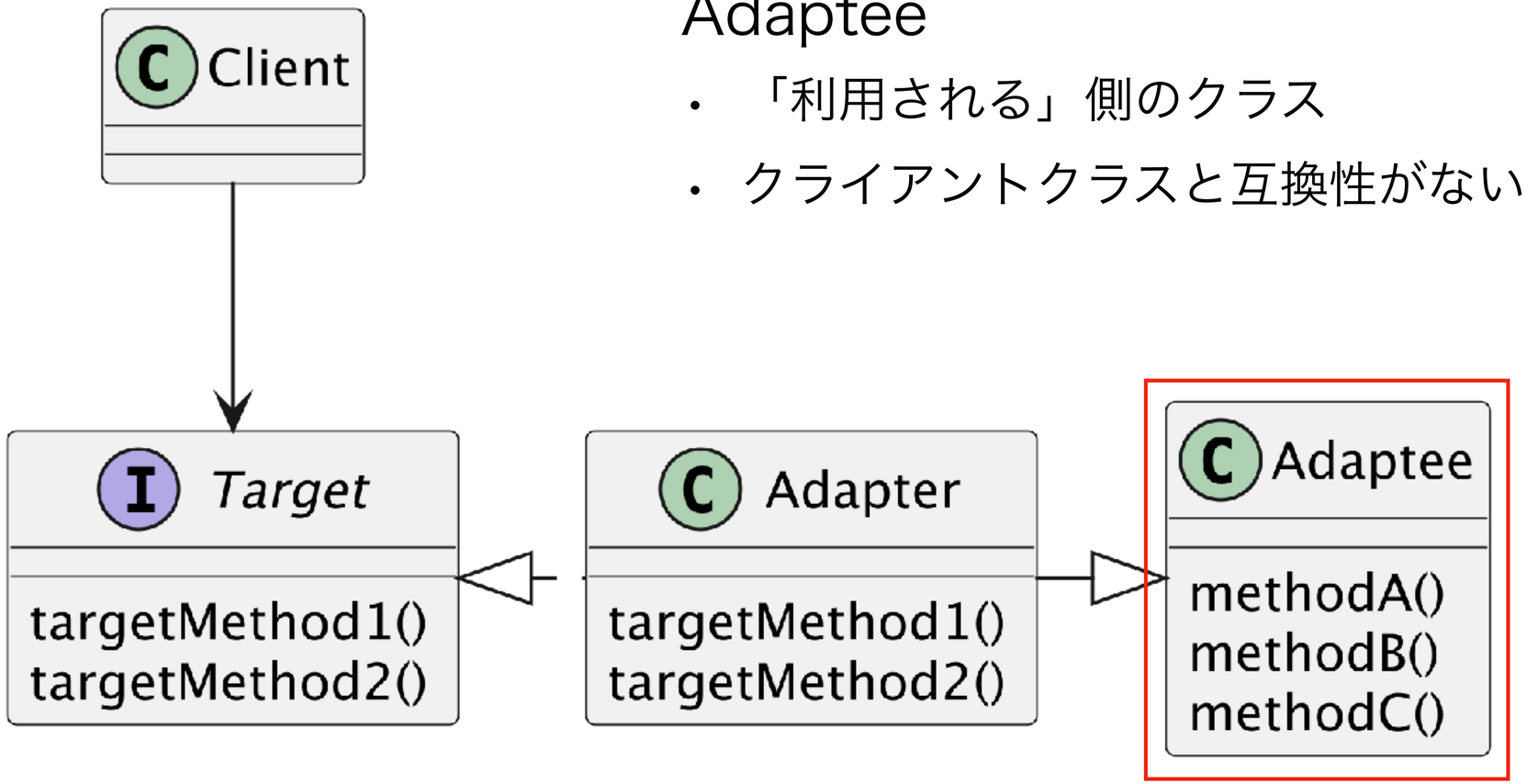
Adapterの構成要素



Target

- クライアントが必要とする機能のAPIを定義するインターフェース

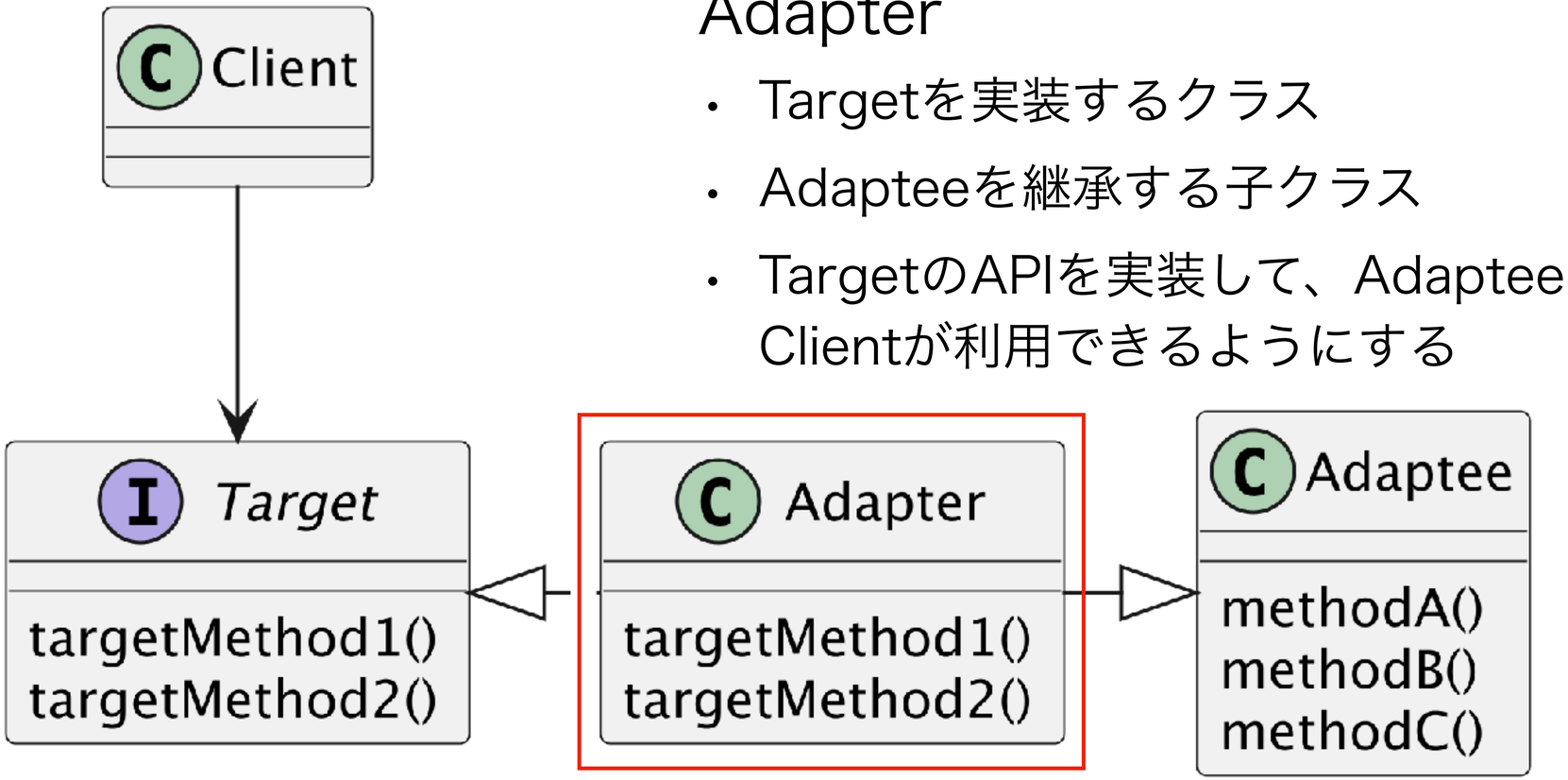
Adapterの構成要素



Adaptee

- 「利用される」側のクラス
- クライアントクラスと互換性がない

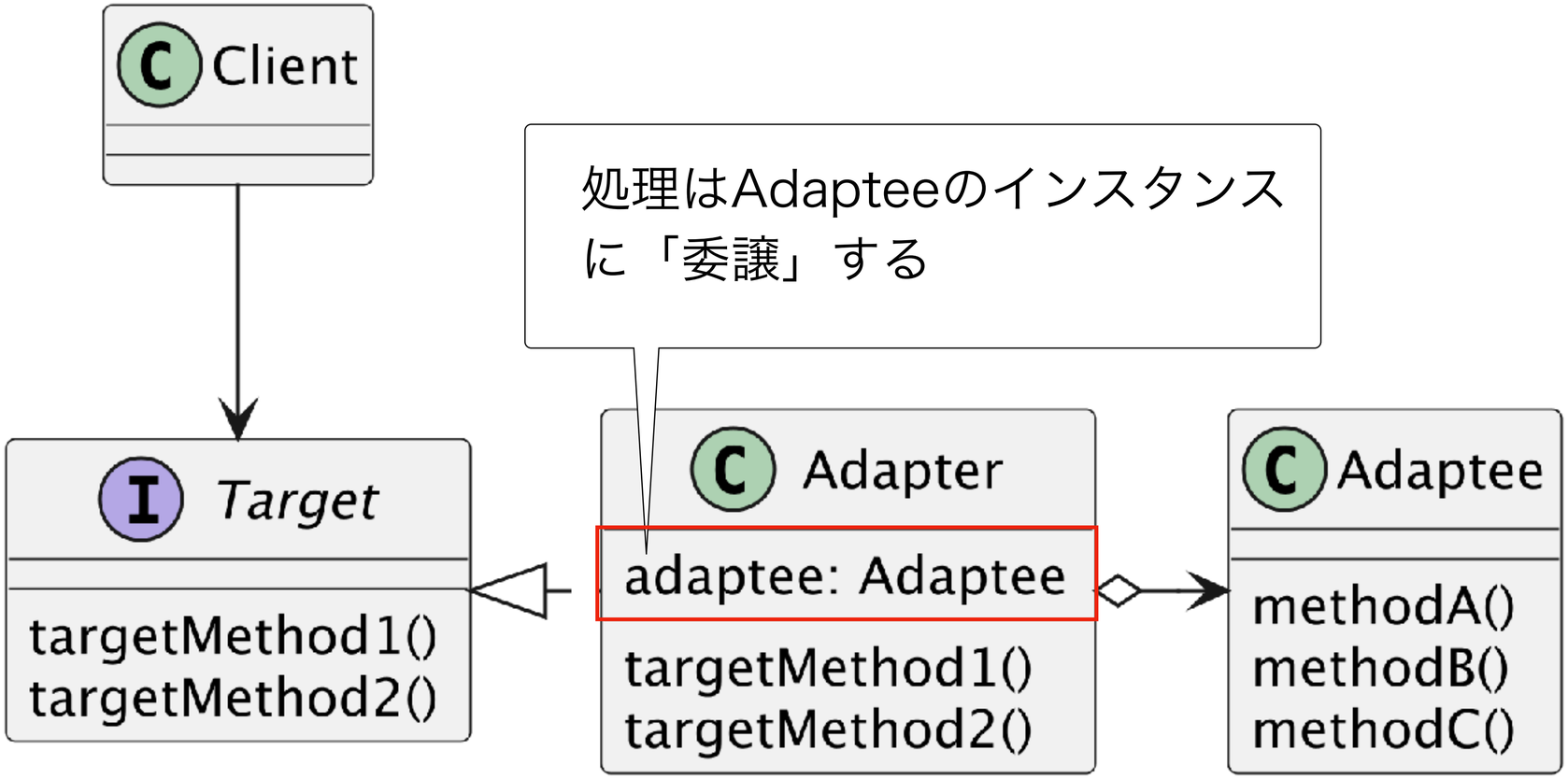
Adapterの構成要素



Adapter

- Targetを実装するクラス
- Adapteeを継承する子クラス
- TargetのAPIを実装して、Adapteeの機能をClientが利用できるようにする

Adapterの構成要素



Adapterのオブジェクト指向的要素

- 「継承」「委譲」「ポリモーフィズム」を利用したパターン
  - 継承を使用した実装では、AdapterがAdapteeクラスを継承して新しいインターフェースを提供する
  - 委譲を使った実装では、Adapteeクラスをフィールドに持つクラスを用意し、そのフィールドに処理を任せて新しいインターフェースを提供する
  - ClientからはTargetインターフェースしか見えなくなり、その先にある処理の実装を気にする必要がなくなる

Adapterのメリット・デメリット

✓ 既存のクラス(Adaptee)を修正しないので再テストが不要になる

✓ 変換のためのコードをプログラムのビジネスロジックと分離できるので単一責任の原則に違反しない

✓ インターフェースを介してアダプタと連携するのでオープンクローズドの原則に違反しない

× インターフェースやクラスが増えるので、小さなシステムなどではAdapteeを直接修正した方が良い場合もある

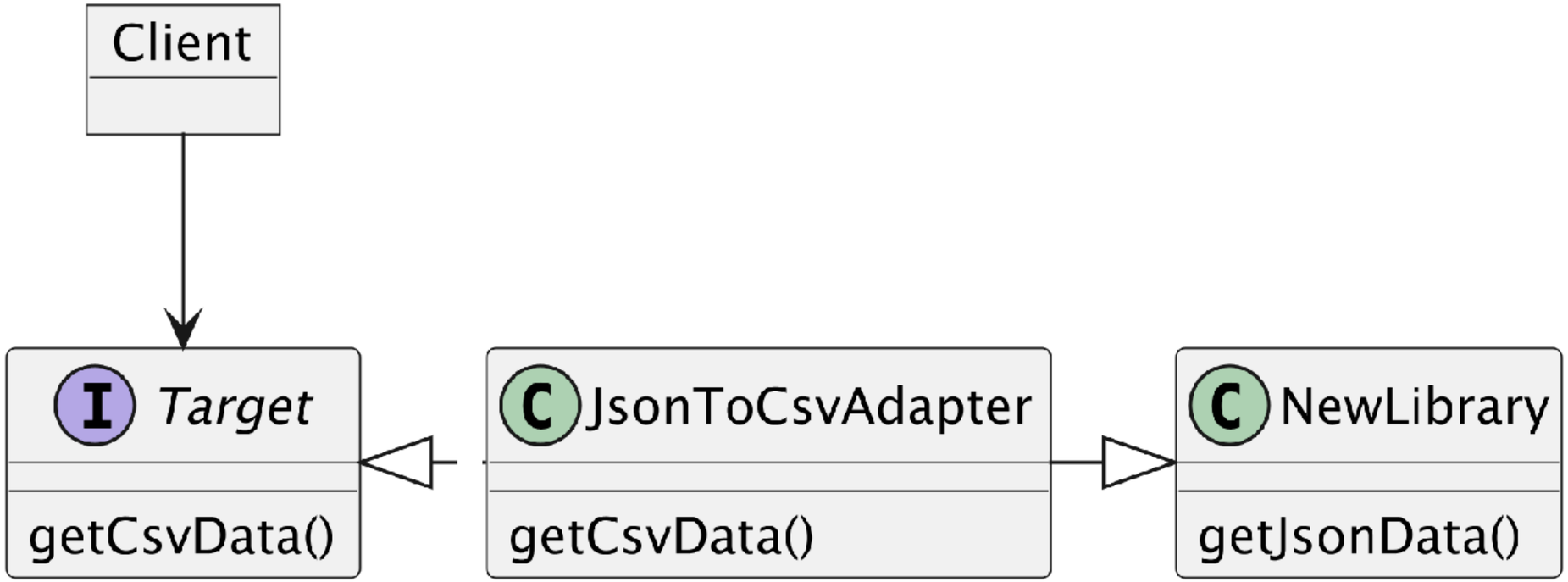
TS

Adapterの使い所

- 既存のクラスを使用したいが、そのインターフェースが利用したい側のコードと互換性がない場合
  - 過去に十分テストされて実績のあるクラスに手を加えず再利用したい場合
  - Adapteeのソースコードが手に入らない場合

Adapterの適用例

JSONデータをCSVに変換する例（継承を使用した実装）



Adapterの適用例

JSONデータをCSVに変換する例（委譲を使用した実装）

Iterator

Iteratorとは

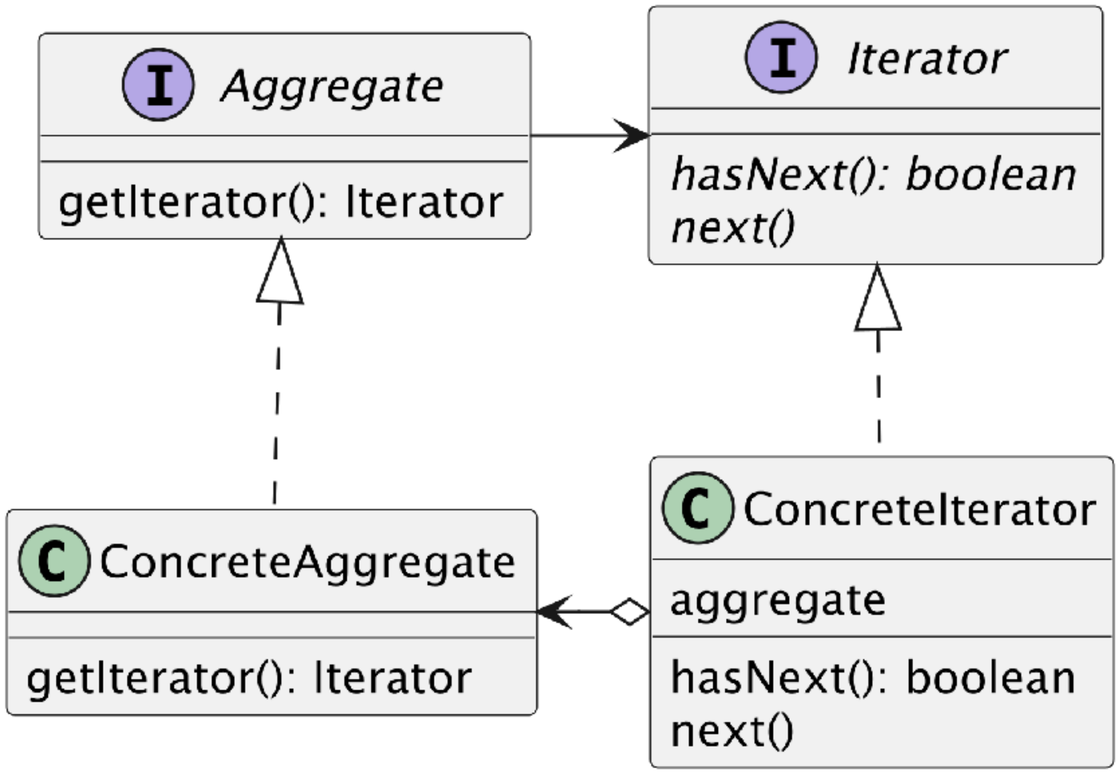
- コレクションの内部構造を利用者に見せずに、その要素に順番にアクセスする方法を提供するパターン

  ※コレクション：配列や連想配列などのデータをまとめて格納するもの

- ループ処理のインデックスiの役割を抽象化し一般化したもの

- 振る舞いに関するデザインパターン

Iteratorの構成要素



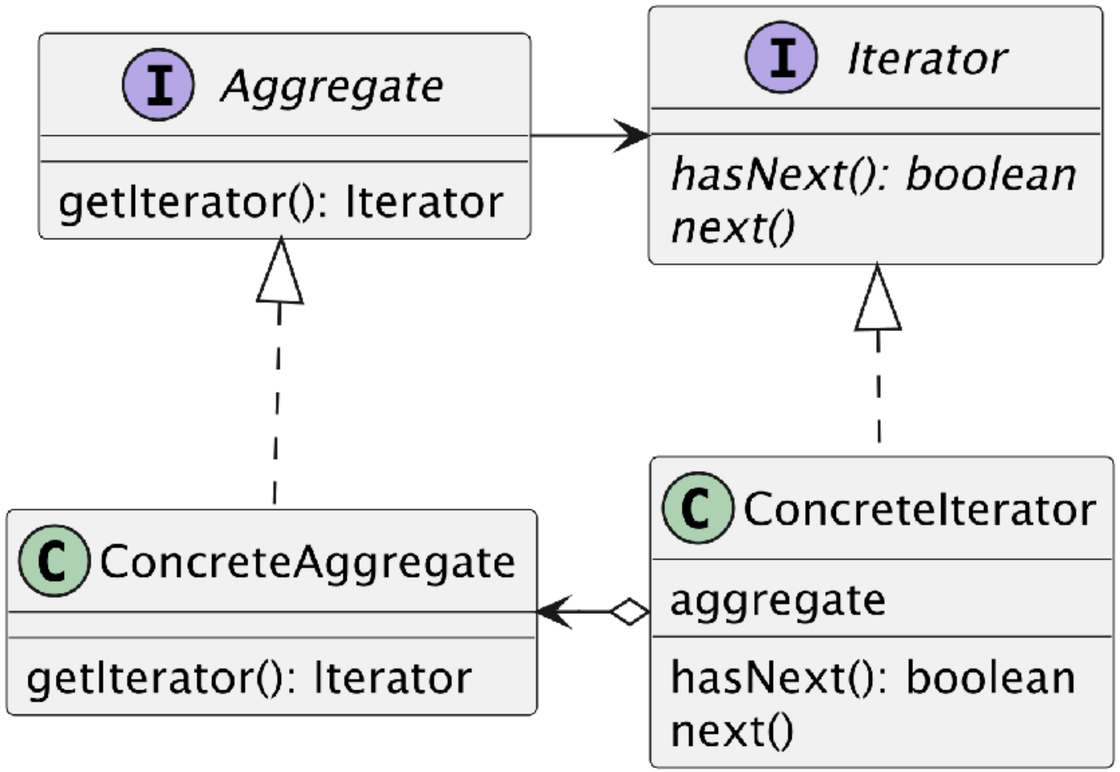
Iterator

- コレクションを探索するために必要な操作を定義するインターフェース

ConcreteIterator

- Iteratorで定義したメソッドを実装するクラス
- ConcreteIteratorの実装によって探索の内容を変更することができる
- 探索を行うコレクションをフィールドに持つ

Iteratorの構成要素



Aggregate

- 探索を行うコレクションを表すインターフェース
- Iteratorを生成するためのメソッドを定義

ConcreteAggregate

- Aggregateで定義したメソッドを実装するクラス
- ConcreteIteratorクラスの新しいインスタンスを返却する

Iteratorのオブジェクト指向的要素

- 「カプセル化の逆」（※本コース内での用語）を利用したパターン
  - データと操作を1つのクラスにまとめるのが「カプセル化」
  - IteratorではAggregateの操作を切り出して別のクラスとして定義
  - 別クラスとして切り出すことで、コレクションクラスがシンプルに保たれ、再利用性やメンテナンス性を高めることができる

Iteratorのメリット・デメリット

✓ 利用者がコレクションの詳細なデータ構造を知る必要がなくなる

✓ コレクションの実装と探索のためのアルゴリズムを分離することができる

✓ 既存のコードに修正を加えることなく、新しい種類のコレクションやイテレーターを追加できる

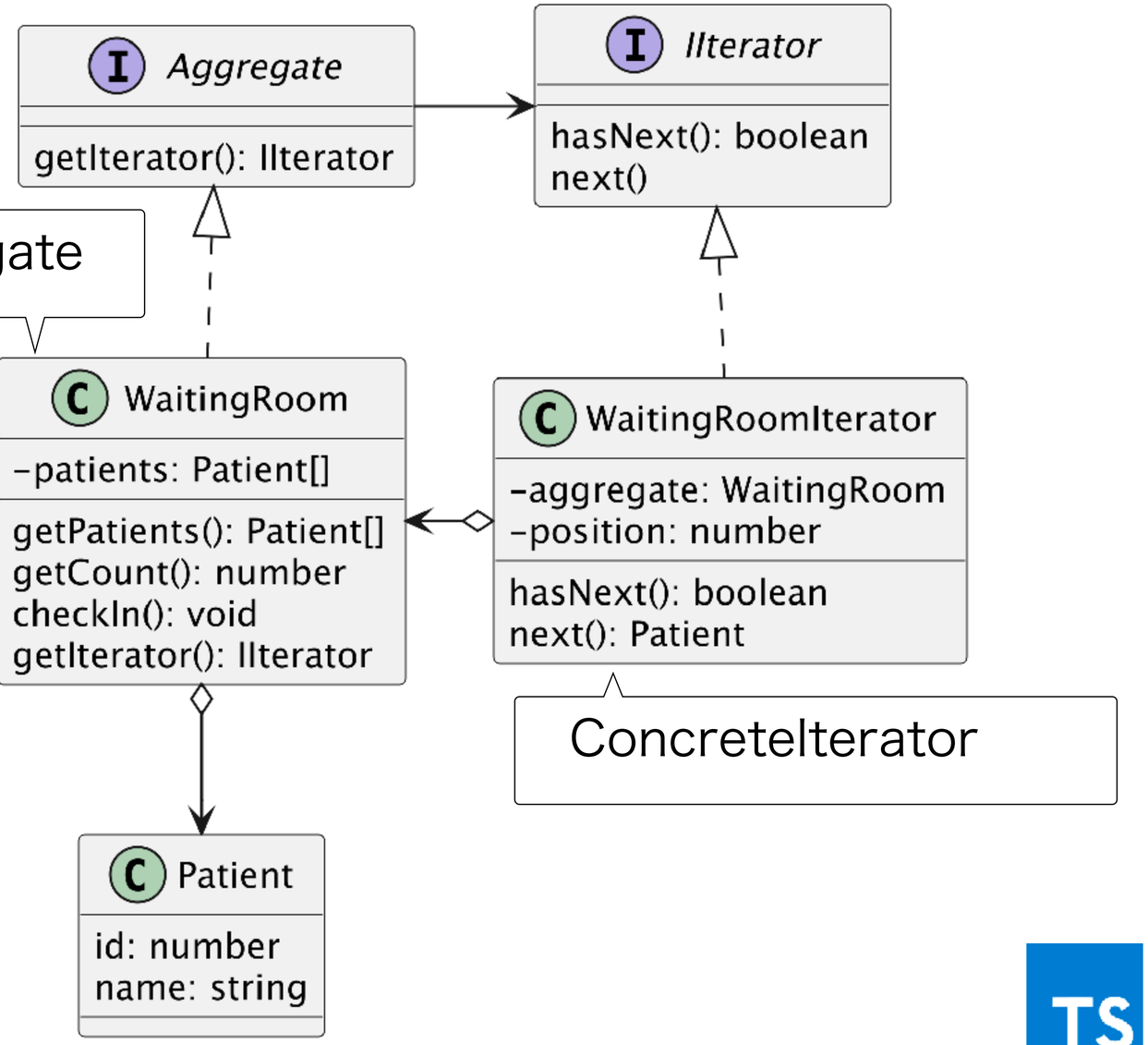
× 単純なコレクションの場合、Iteratorを使用しない方がコードがシンプルになる

Iteratorの使い所

- コレクションが複雑なデータ構造をしており、その複雑さを利用者から隠したい場合
  - Iteratorのメソッドを呼び出すだけでデータを取得可能
- 探索のための方法を複数持たせたい場合
  - オープンクローズドの原則に違反することなく探索のためのロジックを追加可能

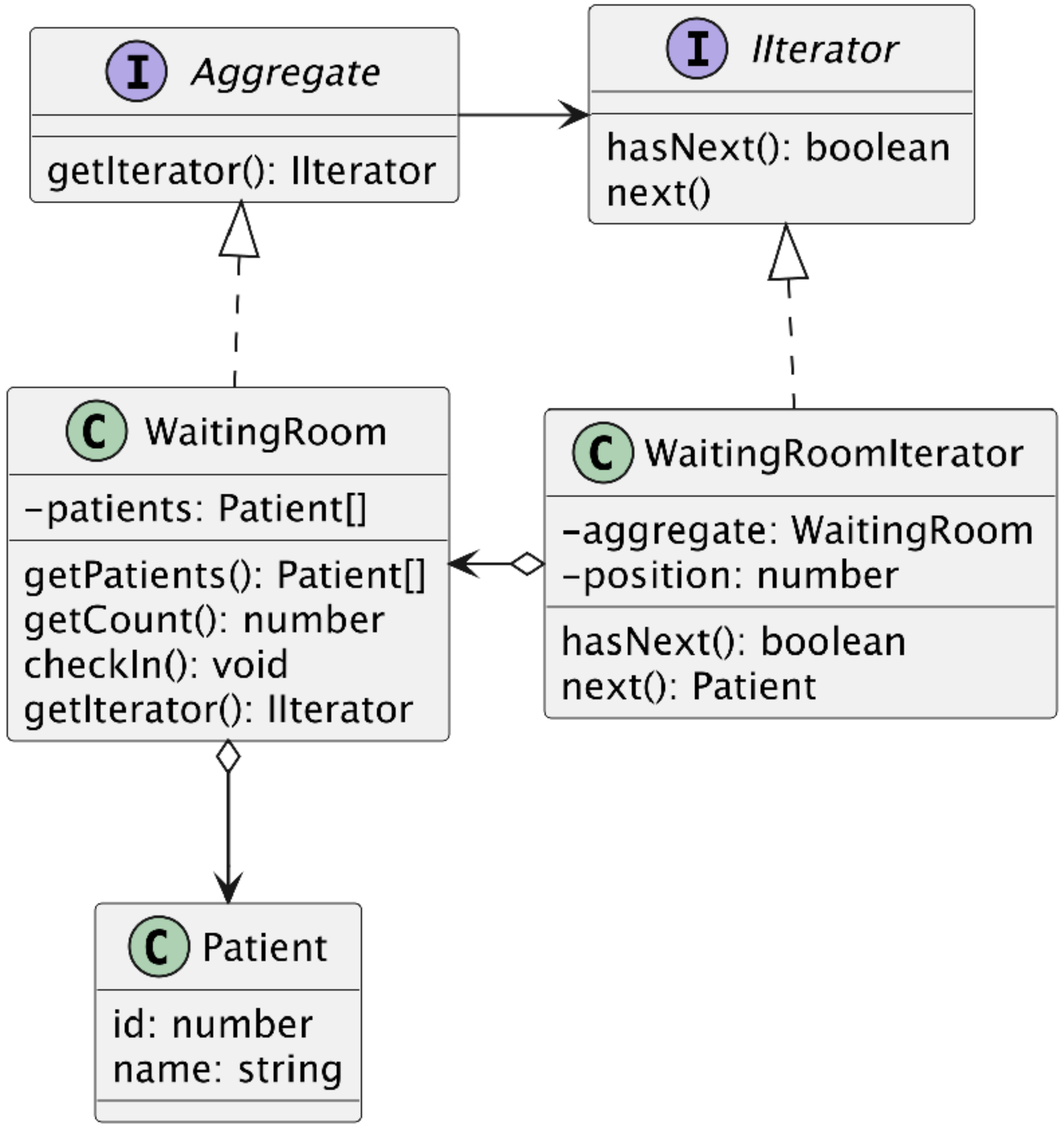
Iteratorの適用例

- 病院の待合室で順番が呼ばれる例



Iteratorの適用例

- 病院の待合室で順番が呼ばれる例



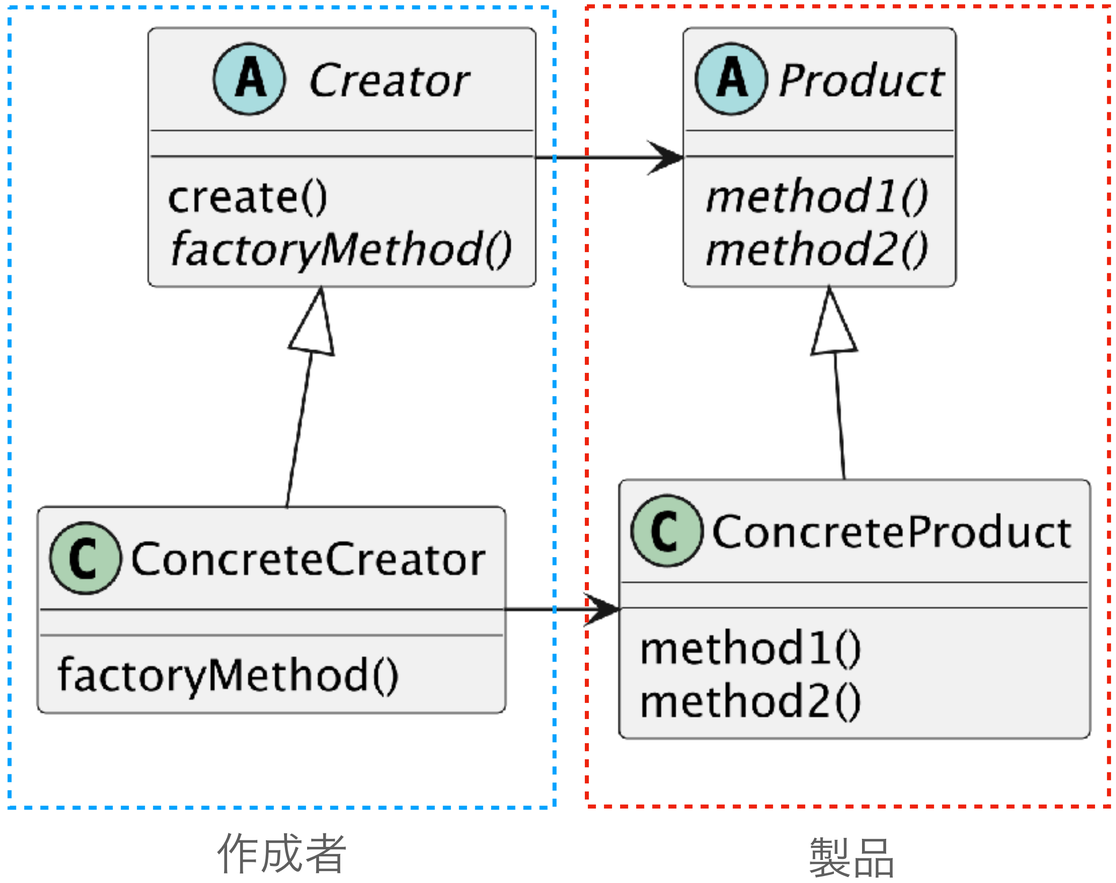
Factory Method

Factory Methodとは

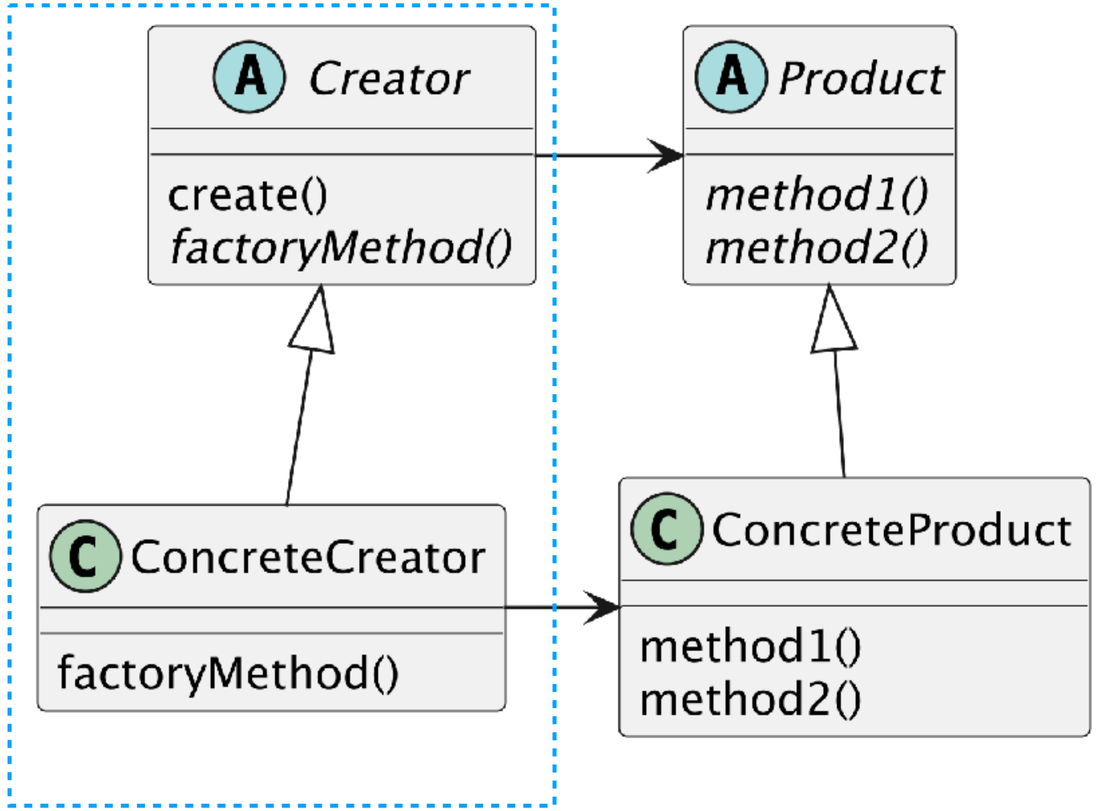
- 親クラスでインスタンスの生成方法を定め、具体的に何をどうやって作るかは子クラスで定めるようなパターン
  - 生成したいオブジェクトのコンストラクタを呼び出してインスタンスを生成するのではなく、親クラスに定義された生成用のメソッドを呼び出してインスタンスの生成を行う
- Template Methodを応用したパターン
- 生成に関するデザインパターン

TS

Factory Methodの構成要素



Factory Methodの構成要素



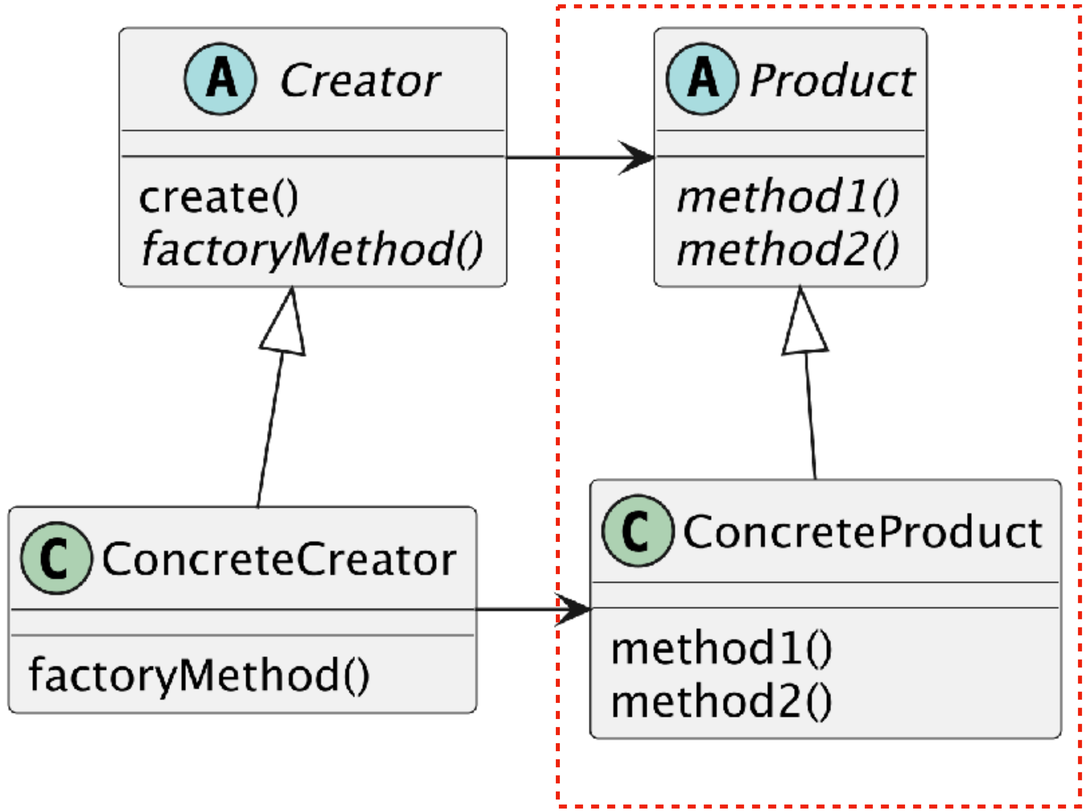
Creator

- Productを生成する抽象クラス
- Productの生成を行うcreateメソッド
- 具体的な生成方法を実装するためのAPIを提供

ConcreteCreator

- Creatorを継承したクラス
- 製品生成のための具体的な方法を実装
- ConcreteProductクラスのインスタンスを返却

Factory Methodの構成要素



Product

- Creatorのオブジェクト生成メソッドで生成される抽象クラス or インターフェース
- 生成される製品が持つべきAPIを定義する

ConcreteProduct

- Productを継承（実装）したクラス
- ProductのAPIに沿った具体的な製品の機能を実装

Factory Methodのオブジェクト指向的要素

- 「継承」を利用したパターン
  - ProductとConcreteProduct、CreatorとConcreteCreatorの間にそれぞれ継承関係がある
  - Template Methodと同様に、処理（生成）の枠組みを親クラスで決定し、子クラスごとに具体的な生成方法を実装

Factory Methodのメリット・デメリット

✓ オープンクローズドの原則に違反することなく新しいProductを追加することができる

✓ オブジェクトの利用側とオブジェクトの結びつきを弱くすることができる

× 簡単な生成処理の場合はFactory Methodを使用しない方がコードがシンプルになる

TS

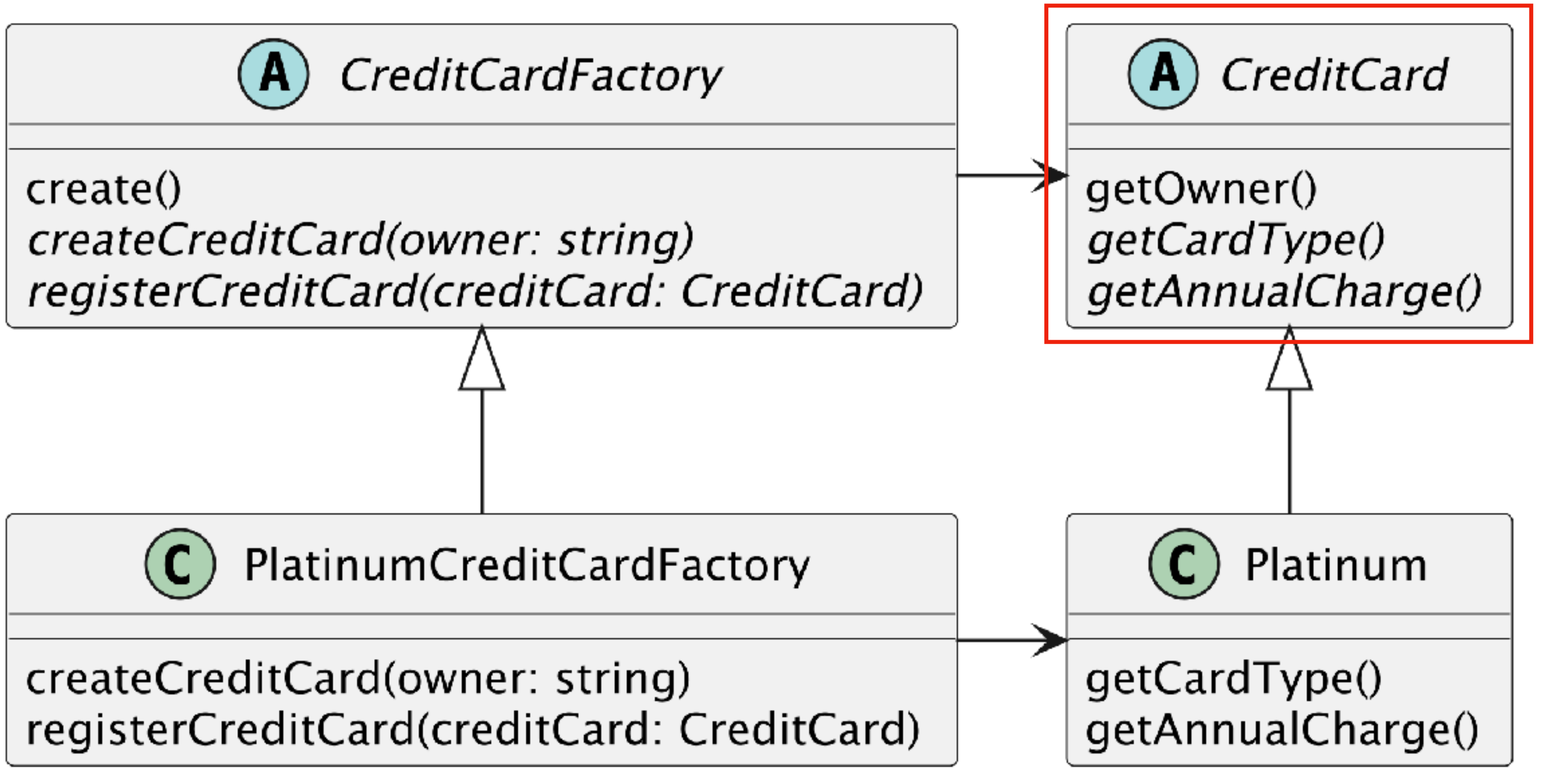
Factory Methodの使い所

- 類似した複数種類のオブジェクトを生成する必要がある場合
  - オープンクローズドの原則に違反することなく別のオブジェクトの追加が可能
- オブジェクトの生成ロジックが複雑な場合
  - createメソッドを呼び出すだけで複雑な生成ロジックを記述せず生成可能
- Productの種類や生成手順が頻繁に変更される可能性がある場合
  - 利用側とProductの結びつきが弱いので、変更に強い設計となる

TS

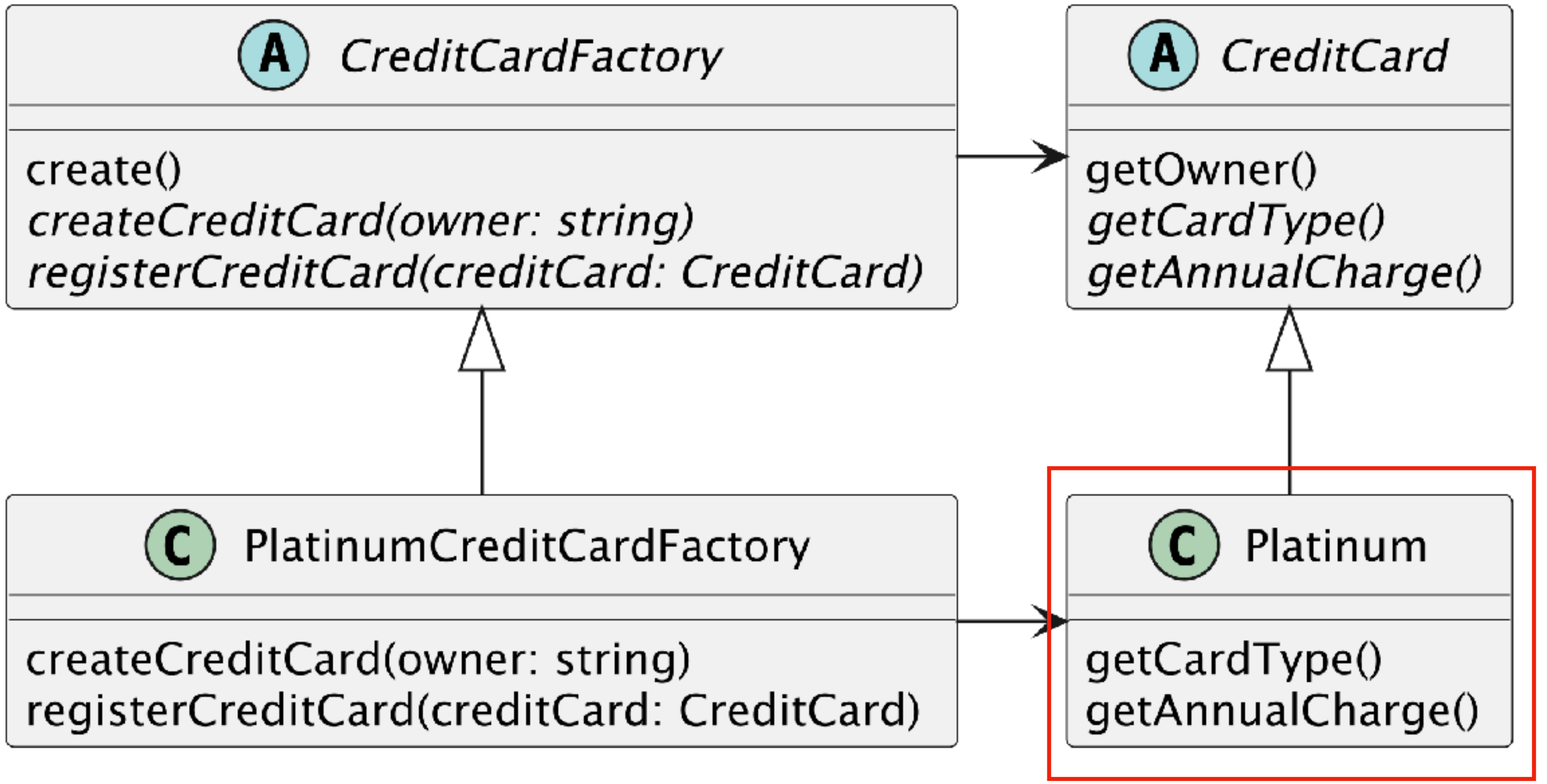
Factory Methodの適用例

- クレジットカードを作成する例



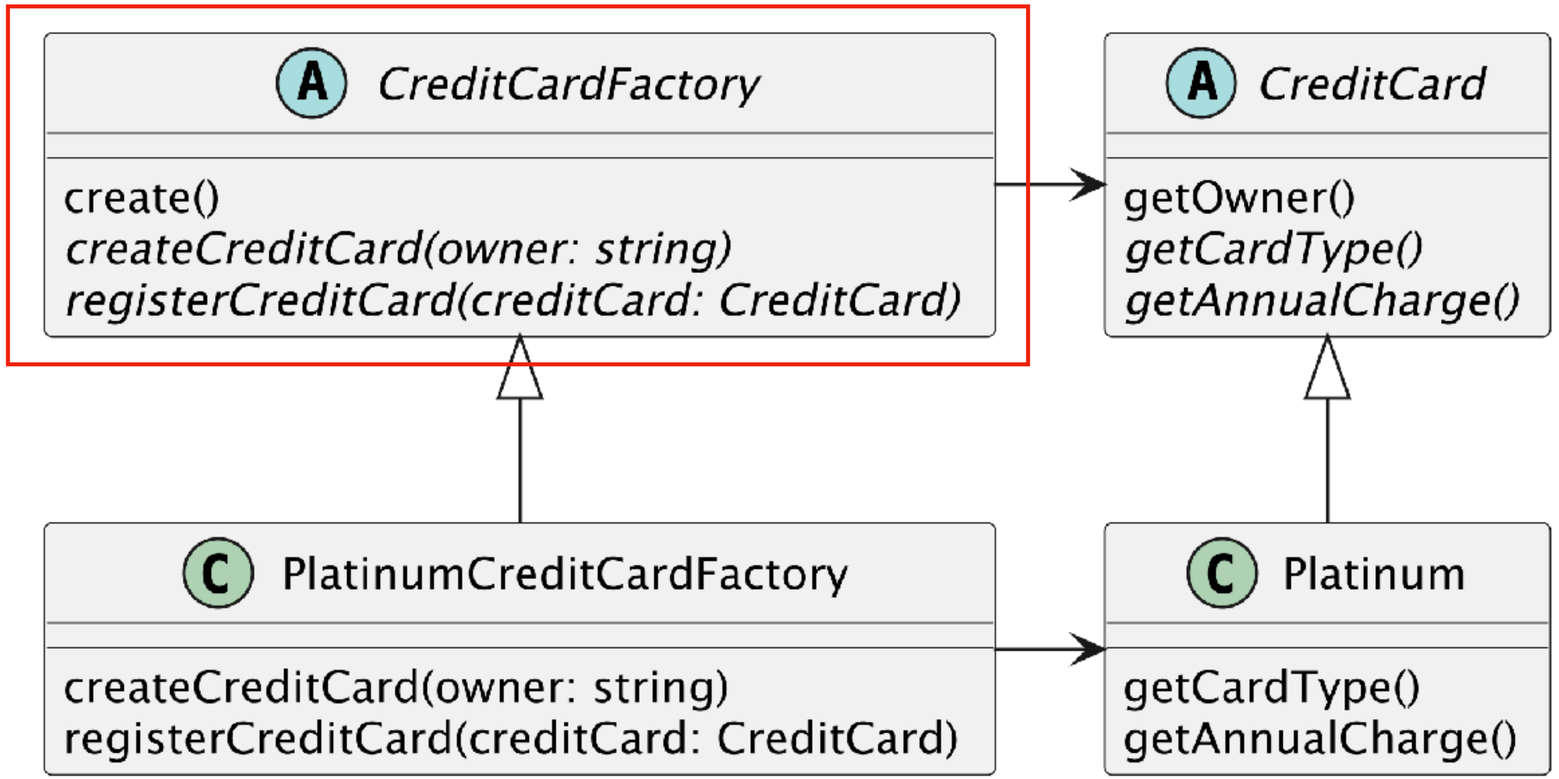
Factory Methodの適用例

- クレジットカードを作成する例



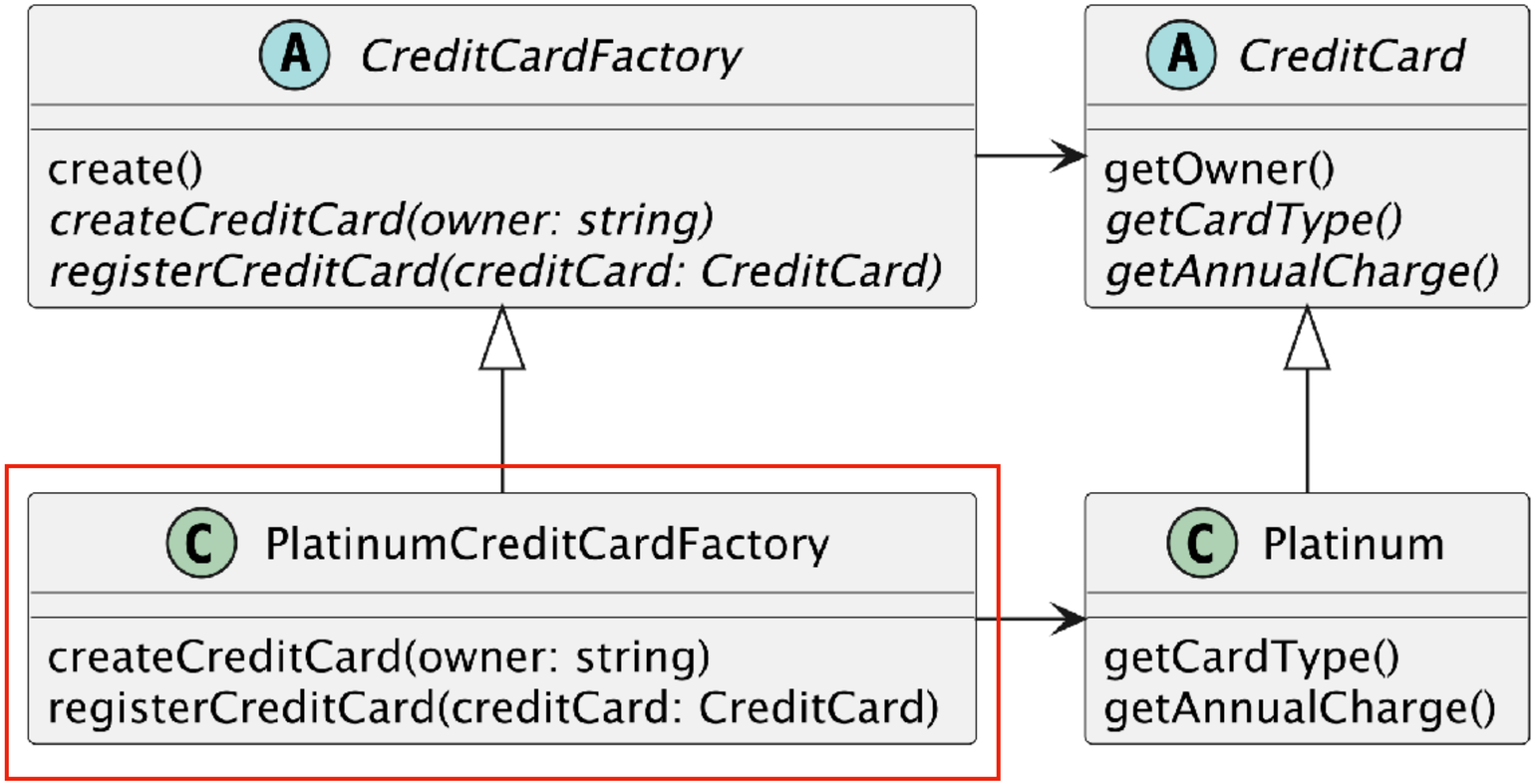
Factory Methodの適用例

- クレジットカードを作成する例



Factory Methodの適用例

- クレジットカードを作成する例

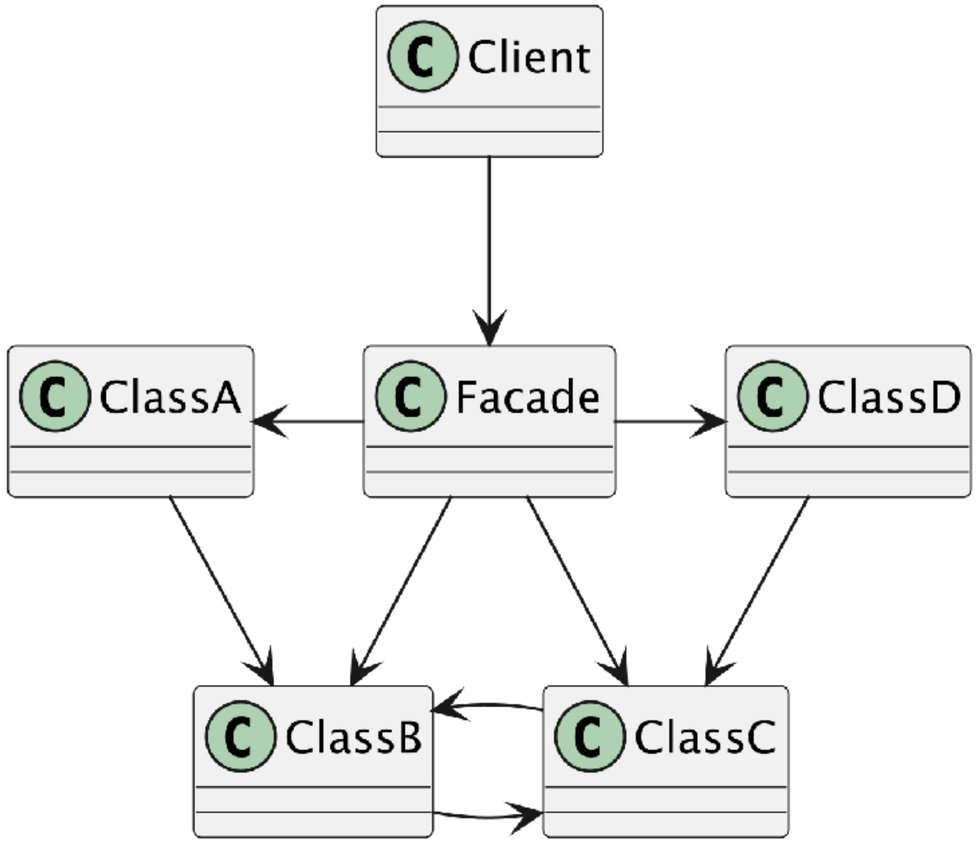


Facade

Facadeとは

- フランス語で「建物の正面」という意味
- 複雑な内部処理をまとめ、システムの外側に簡素化されたインターフェース（API）を提供するパターン
  - システムの内側にある各クラスの役割や依存関係を考えて、正しい順番でクラスを利用できるようにする
- 構造に関するデザインパターン

Facadeの構成要素



Facade

- Clientに高レベルでシンプルなインターフェースを提供する
- システム内部のクラス同士の関係を知っている
- 実際の処理はシステム内部のクラスに委譲する

その他クラス

- それぞれの機能を持っている
- Facadeのことは意識しない
- Facadeを呼び出すことはない

Facadeのオブジェクト指向的要素

- 「カプセル化」を利用したパターン
  - クライアントからはFacadeクラスで提供されたAPIのみが見える
  - システム内部のクラス群を非常に大きなクラスと捉えると、Facadeで提供されたAPIはメソッドに相当する
  - システム内部のクラス群の状態や構造、複雑さを隠蔽している

Facadeのメリット・デメリット

✓ その他クラスの構成要素を隠蔽することができる

✓ その他クラスとクライアントの結びつきを弱くする

× Facadeクラスがその他クラスのすべてに結合されたゴッドクラスになる可能性がある

TS

Facadeの使い所

- 複雑なサブシステムの一部の機能を使用する場合
  - クライアントが内部構造を知らなくてもシンプルなAPIで機能を利用できる
- 複数のクラスの処理を呼び出す一連のコードがいろいろな箇所に書かれている場合

Facadeの適用例

- 商品の注文を行う例