変化に強いプログラミング

～デザインパターン第一章「生成」～

梅林(高田明宏)@わんくま同盟

デザインパターンとは何か(1)

- デザインパターンの定義
  - 「ソフトウェア開発における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編み出した設計ノウハウを蓄積し、名前をつけ、再利用しやすいように特定の規約に従ってカタログ化したもの」(Wikipedia)
- 参考書籍
  - 「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」(Gamma, Helm, Johnson, Vlissides)
  - 「Java言語で学ぶデザインパターン入門」(結城　浩)

デザインパターンとは何か(2)

- デザインパターンの構成要素
  1. パターン名
     - パターンの問題、解法、結果を数語で表したもの
     - 他の開発者との認識の共有のために重要
  2. 問題
     - どのような場合にパターンを適用すべきかを記述したもの
  3. 解法
     - 設計問題を抽象的に記述し、クラスやオブジェクトなどの要素の配置によってどのようにその問題を解決するかを示したもの
  4. 結果
     - パターンを適用することによる利点やトレードオフを示したもの

デザインパターンの分類

生成型	構造型	振る舞い型
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

生成型デザインパターン

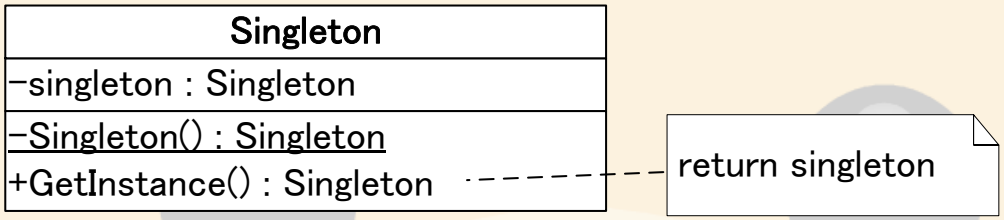
- なぜ重要か
  - 使用される具象クラスに関する情報を隠蔽したい
    - ▲クラス名を明示することで、生成されるクラスとの間に依存関係が発生してしまうため
  - インスタンスが生成され、組み合わされる方法を隠蔽したい
    - ▲オブジェクト生成オペレーションの詳細やクラス同士の関係を隠蔽することで、変更に強い設計にするため
- 関連する技術・方法論
  - リフレクション
  - DI(Dependency Injection)

Singletonパターン(1)

- 目的
  - クラスがただ一つのインスタンスを持つことを保証する
    - ▲アプリケーション全体で使用する設定情報を保持するオブジェクトなど、常に単一のオブジェクトにアクセスする必要がある場合
- 実装
  - クラスをインスタンス化するオペレーションをクラス内部に隠蔽する
    - ▲通常はクラスのコンストラクタを非publicで定義する
  - インスタンスにグローバルにアクセスできるインタフェースを公開する
    - ▲通常はクラスのpublicなメソッド
    - ▲.NET Frameworkではpublicなプロパティも使用される

Singletonパターン(2)

- 構造



Singletonパターン(3)

- 変化に対する設計
  - クラスオペレーションより柔軟
    - ▲クラスメンバに対するオペレーションでは型を隠蔽できない
  - サブクラスのインスタンスを返すよう、アクセスメソッドの実装を変更できる
    - ▲Factory Methodパターンに似た使用方法
  - インスタンスの数を変更することができる
    - ▲オブジェクトプーリングへの変更など

Singletonパターン(4)

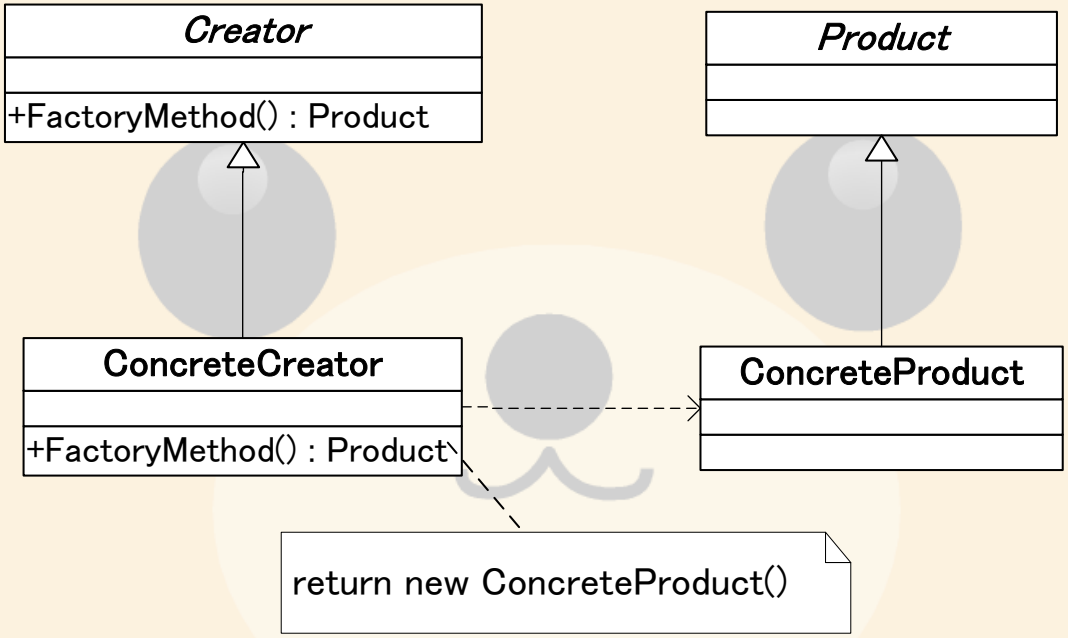
- 注意点
  - Singletonインスタンスの初期化のタイミングに注意する
    - ▲マルチスレッド環境での遅延初期化
    - ▲静的初期化
  - .NET Frameworkではコンストラクタをprivateにしても、別の方法でインスタンスを生成できる
    - ▲Activator.CreateInstance
    - ▲ConstructorInfo.Invoke
- 参考
  - 「C#でのシングルトンの実装」
    http://msdn.microsoft.com/library/ja/default.asp?url=/library/ja/dnpatterns/htm/ImpSingletonInCsharp.asp

Factory Methodパターン(1)

- 目的
  - オブジェクトを生成するインタフェースを別クラスに定義することにより、実際に生成されるオブジェクトの型や生成方法を隠蔽する
- 実装
  - 基本クラスでオブジェクトを生成するオペレーション(ファクトリメソッド)を定義する
  - 上記クラスのサブクラスでファクトリメソッドをオーバーライドして、実際にオブジェクトを生成する処理を実装する

Factory Methodパターン(2)

- 構造



Factory Methodパターン(3)

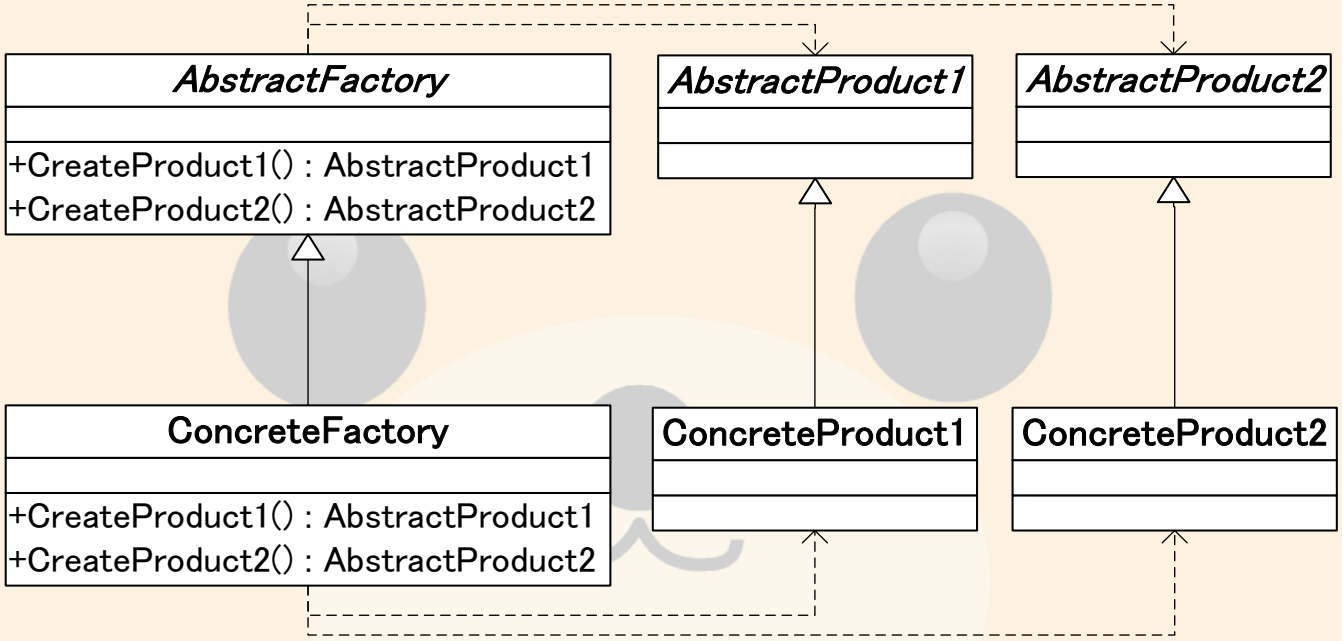
- 変化に対する設計
  - 生成されるオブジェクトの型をコード中に直接記述しないようにすることで、互換性のある別の型への変更が可能になる
  - オブジェクト生成のオペレーションをファクトリメソッド内に隠蔽することで、それらのオペレーションに変更があった場合の修正範囲を局所化することができる

Abstract Factoryパターン(1)

- 目的
  - 関連・依存する複数のクラスについて、インスタンス生成のオペレーションを集約したインタフェースを提供する
- 実装
  - 関連する複数のクラスを生成するインタフェースを定義する(AbstractFactory)
  - AbstractFactoryのメソッドでは、実際に生成されるオブジェクトのインタフェース(AbstractProduct)を戻り値の型とする
  - AbstractFactoryを継承したクラス(ConcreteFactory)で、具象クラスのオブジェクト(ConcreteProduct)を生成する

Abstract Factoryパターン(2)

- 構造



Abstract Factoryパターン(3)

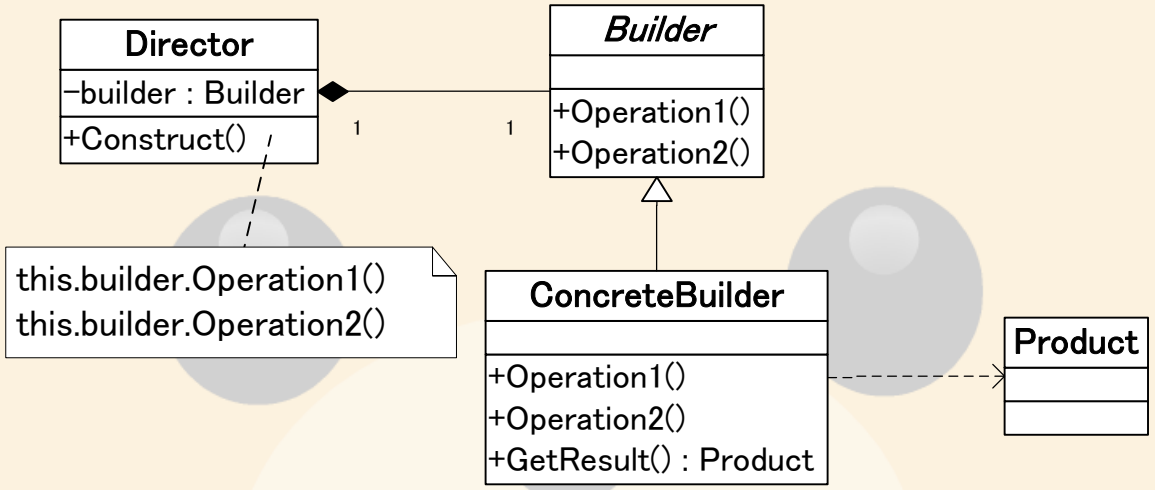
- 変化に対する設計
  - 関連するオブジェクトの生成オペレーションを一ヶ所に集中することで、別の代替可能なクラスの集合への置き換えが容易になる
  - 関連するオブジェクトの集合が安定的な場合は適切な設計といえる
    - ▲生成するオブジェクトの種類が増減したり、生成オペレーションのインタフェースが変更された場合、すべてのConcreteFactoryについて同じ変更を行わなければならない

Builderパターン(1)

- 目的
  - オブジェクトの生成ロジックを、実際に生成されるオブジェクトの実装から切り離し、再利用できるようにする
- 実装
  - オブジェクト生成に必要な複数のオペレーションを定義した基本クラス(Builder)を定義する
  - Builderを継承したクラスで、基本クラスで定義したオペレーションを実装する(ConcreteBuilder)
  - Builderクラスのオペレーションを呼び出し、オブジェクトを生成する(Director)

Builderパターン(2)

- 構造



Builderパターン(3)

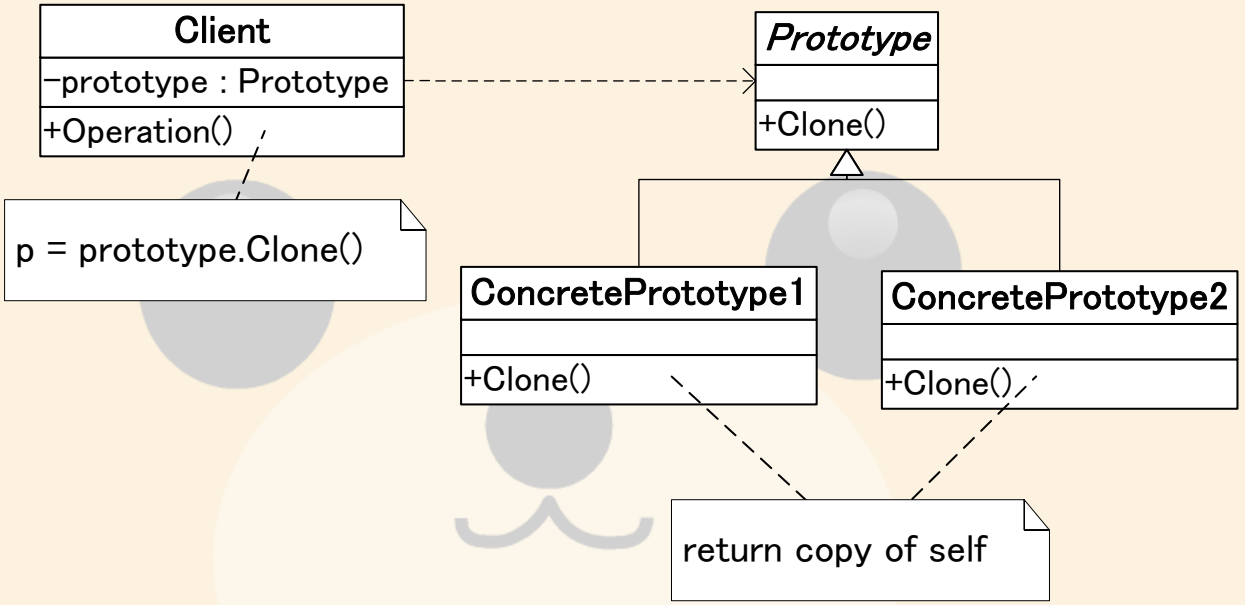
- 変化に対する設計
  - オブジェクトを生成するための作業を抽象化し、実装クラス内部に隠蔽することで、同様の作成ロジックを持つ別種のオブジェクトを作成する際のコードの変更を極小化することができる

Prototypeパターン(1)

- 目的
  - 既存のオブジェクトをコピーすることで新しいオブジェクトを生成する
- 実装
  - オブジェクトを複写するオペレーションをオブジェクトに実装する
    - ▲.NET FrameworkではICloneable.Clone()
    - ▲JavaではCloneable.clone()

Prototypeパターン(2)

- 構造



Prototypeパターン(3)

- 変化に対する設計
  - 型やコンストラクタのシグニチャを指定せずにオブジェクトを生成でき、クラス間の依存関係を減らすことができる
  - 複雑な初期化プロセスをもつオブジェクトの場合、既存オブジェクトの複製というシングルオペレーションに単純化することで、変更箇所を局所化することができる。

Prototypeパターン(4)

- 注意点
  - 「Shallow Copy」と「Deep Copy」
    - ▲オブジェクトの参照のコピー(Shallow Copy)
    - ▲オブジェクトの実体のコピー(Deep Copy)
- 参考
  - 「ICloneableとMemberwiseClone」
    http://blogs.wankuma.com/jeanne/archive/2006/04/06/22272.aspx
  - 「シャローコピーとディープコピー」
    http://blogs.wankuma.com/jeanne/archive/2006/04/07/22287.aspx

リフレクション

- 目的
  - クラスやメソッドの名前をもとに、それらの型情報を実行時に動的に取得するためのAPI
- 変化に対する設計
  - クラス間の静的な依存関係を回避することができる
    - ▲Factory Methodを使う場合でも、最終的にはConcreteCreatorとそれを使用するクラスの間に静的な依存関係が生じてしまう
- 注意点
  - クラス名やメソッドのシグニチャが変更された場合に、リフレクションを使用している箇所で実行時エラーになる可能性がある
    - ▲安全性を犠牲にして柔軟性を実現しているといえる

DI(Dependency Injection)

- 目的
  - クラス間の依存関係をソースコードから排除し、実行時に外部ファイルなどから依存性を注入できるようにする
- 変化に対する設計
  - ソースコードの変更なしに、実行時に生成されるオブジェクトの型を変更することなどが可能になる
- .NET Framework向け実装
  - Spring.NET

    http://www.springframework.net/
  - S2Container.NET

    http://s2container.net.seasar.org/ja/