No.1(Design Patten using GoF)

GoFとはなにか？

プログラミング用語としてのGoF(Gang of Four)とは、オブジェクト指向プログラミングにおける**再利用性の高いコーディングのパターン、デザインパターン**とそれをまとめた4人のプログラマのことを指す。これは以下の三つに分類される23種類の古典的なパターンから構成される。

- 生成的なパターン
- 構造的なパターン
- 振舞的なパターン

生成的パターンについて

生成的パターンには以下のようなものがある。

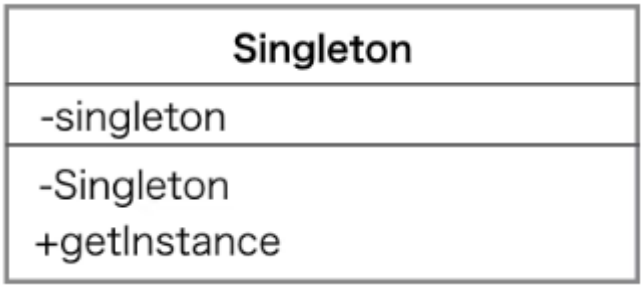
Singleton	クラスをオブジェクトとして実体化（インスタンス化）することに関して、**唯一のインスタンスの生成に限定する**機構を提供するパターン。これによって、システム全体で大域的に情報を管理（共有）させるためのオブジェクトを設計できる。
Factory	一つの**オブジェクトを生成するための手段を提供する**パターンであり、継承（子）クラスによって実体化するクラスを指定する。

Builder	複雑な**オブジェクトの構築と表現の方法を分離する**パターンであり、**様々な表現に対しても同じ構築方法**を提供する。
Abstract Factory	**互いに関連または依存するオブジェクトの集まりを、その具体的な中身を指定する事なく生成するための機構を提供する**パターンであり、**Factory クラスを生成するための上位概念の Factory（工場）とも考えられる。**
Prototype	**原型となるインスタンス（クラスの実体）を用いて、生成するオブジェクトの種類を指定する**パターン。これは、**存在するオブジェクトの「骨格」から新たなオブジェクトを生成する仕組みを提供**し、計算性能やメモリの消費を最小限に抑える。

Singleton

Singleton パターンで設計されたクラスでは、**インスタンス（オブジェクトとしての実体）は一つしか生成できません**。

クラス図👇



クラス変数としてsingletonを定義、インスタンスを得るためのstaticメソッドで同じインスタンスを返す。

以下に、その簡単な実装例を示します。

```
public class SingleObject {
  // create a single instance of SingleObject
  private static final SingleObject instance = new SingleObject();

  // prohibit to access a constructor
  private SingleObject() {}

  // public method for getting an instance
  public static SingleObject newInstance() {
    return instance;
  }

  // omitted below… (以下、省略)
}
```

まず最初に注意すべき点は、コンストラクタ（構築子）が private として宣言されている点です。

これにより、**以下の様なコンストラクタを用いたインスタンス化は不可能**となります。

```
SingleObject instance = new SingleObject();
```

その代わりに、インスタンスを生成する際には以下の静的（static）なメソッドを使います。

```
public static SingleObject newInstance() {
    return instance;
}
```

したがって、**このクラスのインスタンス化は以下の方法で可能**となります。

```
SingleObject instance = SingleObject.newInstance();
```

ここで、newInstance()は static として宣言されているので、その呼び出しはオブジェクトからではなく、クラスの名称SingleObjectを用いてSingleObject.newInstance();となります。

また、フィールドinstanceの値は、このクラスが定義された最初に一度だけ、静的（static）で最終的な（final）変数として生成されている点に注意してください。

ここで、**final 宣言されたフィールドは、最初に値が代入された後には値を更新できない**設定になっています。さらに、private 宣言されているので、他のクラスから値を直接参照（読み取り）できない設定にもなっています。

したがって、`SingleObject.newInstance()`で生成されるインスタンスは、最初に生成されて**フィールド `instance` に代入された常に同じ実体**の値となります。

Singletonパターン設計法のまとめ

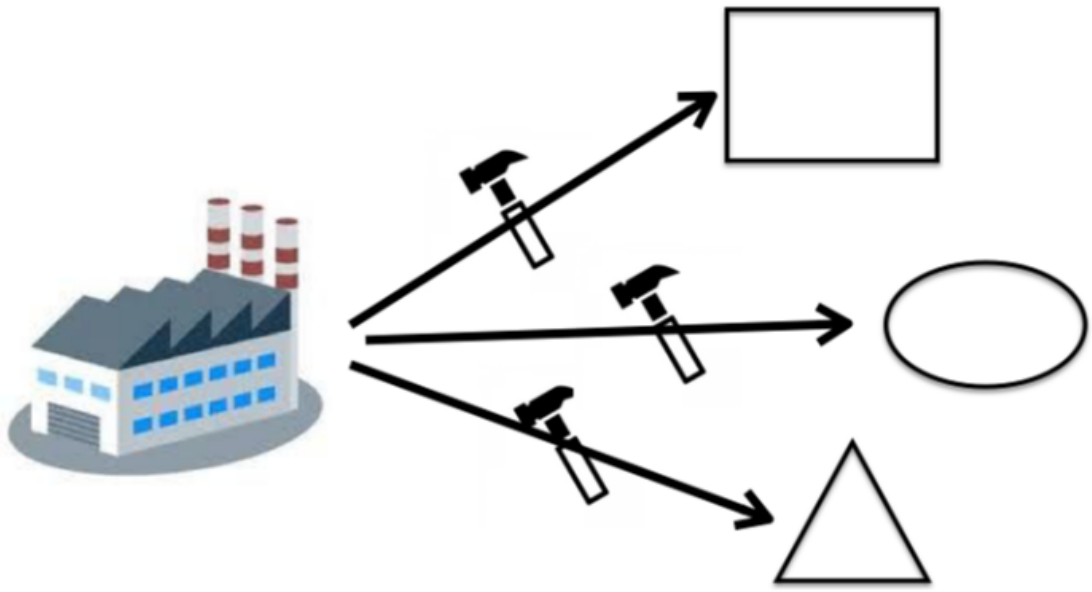
- コンストラクタをprivate宣言して、アクセス不可とする。
- 自信のインスタンスを、静的(static)で最終形(final)でアクセス不可(private)なフィールドに保持する。

- 自信の唯一のインスタンスを返す静的(static)なメソッド(newInstanvce)を設定する。

Factory

これは、オブジェクトの生成論理を明示しなくともそのインスタンスが得られ、共通のインタフェースを用いて新たなオブジェクトを参照するためのものです。

以下にその簡単な構成例として、幾何形状の各種のクラス（Rectangle:長方形、Ellipse:楕円、Triangle:二等辺三角形）を生成する ShapeFactory クラスを示します。



```
public class ShapeFactory {
    // Get object of shapeType
    public Shape create(String shapeType, int w, int h){
        if(shapeType.equals("ELLIPSE"))
            return new Ellipse(w, h);
        else if(shapeType.equals("RECTANGLE"))
            return new Rectangle(w, h);
        else if(shapeType.equals("TRIANGLE"))
            return new Triangle(w, h);
        else
            return null;
   }
}
```

ここで、各種の形状クラスは共通の概念クラス（abstract class）`Shape`を継承して構築されます。

```
public abstract class Shape {
    protected int width, height;
    public Shape (int w, int h) {
        width = w; height = h;
    }
    abstract void draw();
}
```

この概念クラスShapeでは、必ず幅（width）と高さ（height）によって形が決められ、その全てに対して描画用のメソッド（draw）を実装する必要があると宣言しています。

(自分用メモ)

abstractクラスについて

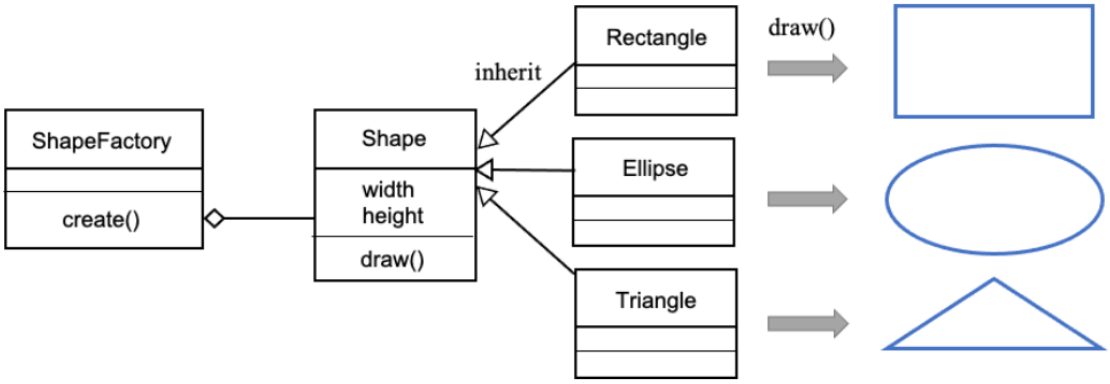
このクラスを継承して、例えば長方形クラスは以下の様に構成されます。

```
public class Rectangle extends Shape {
   public Rectangle(int w, int h) {
      super(w, h);
   }
   @Override
   public void draw() {
      // Draw Rectangle of width X height
   }
}
```

これらの形状クラスのインスタンスは、それらの工場である`ShapeFactory`クラスのオブジュエクトの`create`メソッドを用いて、以下の様に得られます。

```
ShapeFactory shapeFactory = new ShapeFactory();
// get an object of Ellipse
Shape shape = shapeFactory.create(“ELLIPSE”, 20, 30);
shape.draw(); // call draw method of Ellipse
shapeFactory.create("TRIANGLE", 20, 10).draw();
```

ここで、得られたインスタンスから描画関数（`draw`）が呼び出されているので、以下の様な描画結果となることが想定されます。



Factoryパターン設計法のまとめ

- 工場(Factory)が設計するオブジェクト群に共通した特徴を、抽象クラスとして宣言する。
- 製造されるすべてのオブジェクトは、この共通の抽象くらすを継承して作成する。
- 工場(Factory)を担当するクラスに、製造したオブジェクトを返すメソッド(create)を作成する。ただし、引数でオブジェクトの種類を指定できるようにし、戻り値は共通の抽象くらすの型を指定する。

Builder

Builder（建築者）のデザインパターンは、Factory と同様に各種のオブジェクトを生成するための工場的な機能を提供するものですが、Factory よりも複雑な構造のオブジェクトを、その構築過程を**単純なオブジェクトを使いステップを踏みながら組み立てていく**というアプローチを取ります。

メリットは、通常のコンストラクタの初期化処理ではできない、複雑な初期化処理ができます。

コンストラクタを増やせばいいのでは？👇

```
public class ConstractaTest {

  /* 引数1コンストラクタ */
  ConstractaTest(String param){
  }
  /* 引数2コンストラクタ */
  ConstractaTest(String param,int param2){
  }
  /* 引数5コンストラクタ */
  ConstractaTest(String param,int param2,int param3,int param4,int param5){
```

```
  }

}
```

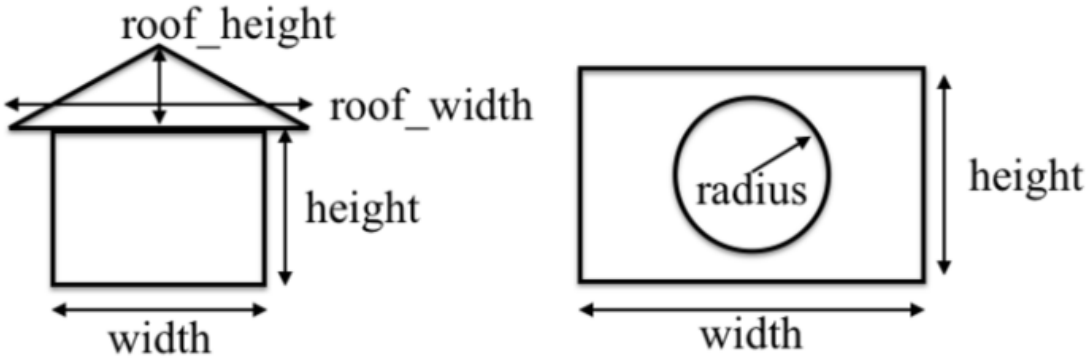
これには以下のようなデメリットがある。

- コンストラクタが多くてだるい
- 開発者が混乱する
- コンストラクタ引数の数が増減した際にメンテナンスで沼る

ということで具体的方法👇

まず最初に、先ほどの幾何形状のクラスを拡張して、基本的な形の組み合わせで構成される House（家屋形状）と、Hinomaru（日の丸形状）のクラスを以下の様に宣言します。

```
public class House extends Shape {
    protected int roof_width, roof_height;
    public House (int w, int h, int rw, int rh) {
        super(w, h);
        roof_width = rw;
        roof_height = rh;
    }
    @Override
    public void draw() {
        // draw the shape of a house
    }
}

public class Hinomaru extends Shape {
    protected int radius;  // radius of inner circle
    public Hinomaru (int w, int h, int rad) {
        super(w, h);
        radius = rad;
    }
    @Override
    public void draw() {
        // draw hinomaru
    }
}
```



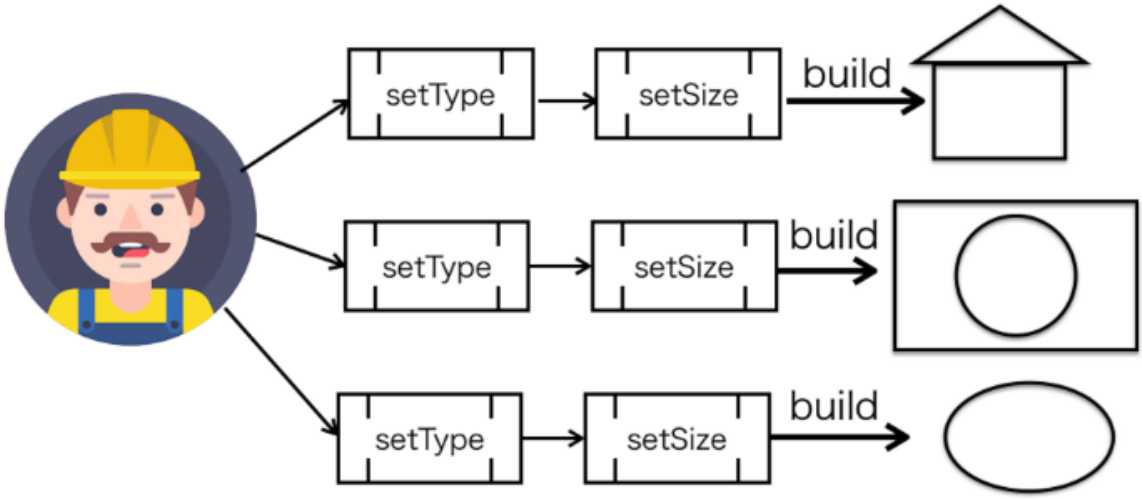
これらの拡張クラスに対して Factory クラスで生成すると、 以下の様な方法が考えられます。

```
ShapeFactory shapeFactory = new ShapeFactory();
// get an object of Ellipse
Shape shape = shapeFactory.create(“ELLIPSE", 20, 30);
shape.draw(); // call draw method of Ellipse
❌ shapeFactory.create("HOUSE", 50, 30, 22, 10).draw();
❌ shapeFactory.create("HINOMARU", 30, 20, 8).draw();
```

しかしながら、コード中に `❌` が記されている命令文は、メソッド `create` の引数の数が異なるのでエラーとなります。

仮に、`ShapeFactory` の中で引数を配列やリストで渡すことが考えられますが、形状ごとに設定される値の意味が異なるので、呼び出しのインタフェースとしては複雑になります。

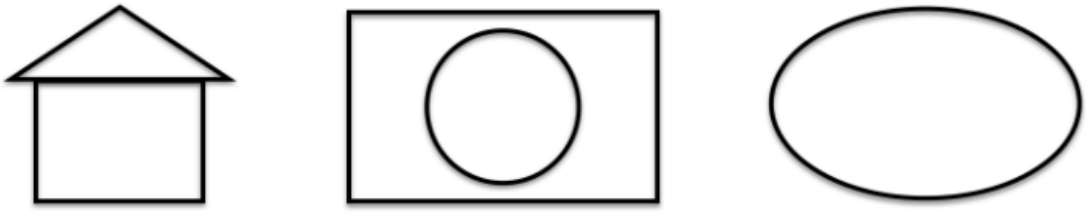
そこで、Builder のパターンは、以下の図の様に形状の型（長方形、楕円等の type）とその大きさ（size）を指定するメソッドを別途用意し、それらを順次呼び出して段階的に形状のクラスを作成します。



これをコードで記述すると、以下の様になります。

```
ShapeBuilder shapeBuilder = new ShapeBuilder();
shapeBuilder.setShapeType("HOUSE");
shapeBuilder.setSize(30, 20);
shapeBuilder.setOptionSize(32, 10);
Shape house = shapeBuilder.build();
house.draw();
shapeBuilder.setShapeType("HINOMARU");
shapeBuilder.setSize(50, 30);
shapeBuilder.setOptionSize(10, 10);
shapeBuilder.build().draw();
shapeBuilder.setShapeType(“ELLIPSE”);
shapeBuilder.build().draw(); // this uses previous setting of size
```

ただし、`setSize`で指定されるサイズは長方形の大きさであり、 屋根の三角形と日の丸の円のサイズは`setOptionSize`で指定されています。



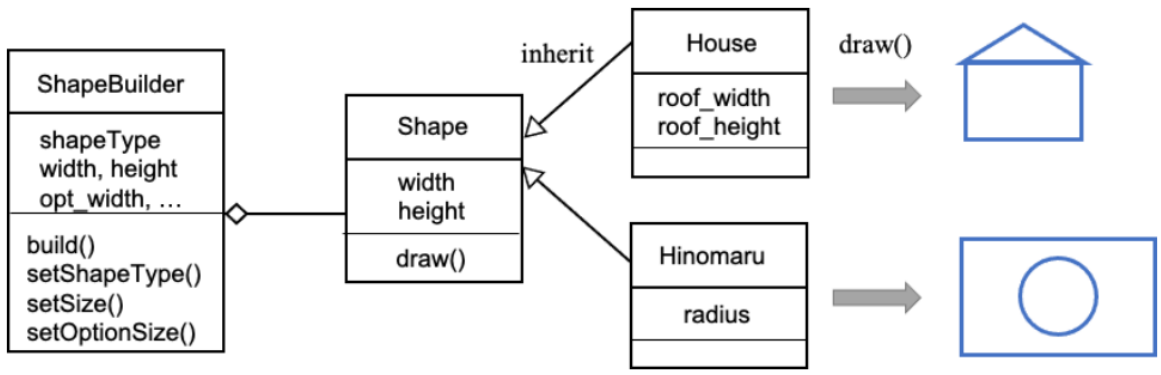
上記コードの実行結果

この様にして使用される`ShapeBuilder`の構成例は、以下の様になります。

```
public class ShapeBuilder {
    private String shapeType = "ELLIPSE";  // default type is Ellipse
```

```
    private int width = 10, height = 10;
    private int opt_width = 10, opt_height = 10; // optional values for size

    public Shape build(){
        if(shapeType.equals("ELLIPSE"))
            return new Ellipse(width, height);
         … // for ELLIPSE, TRIANGLE
        else if(shapeType.equals("HOUSE"))
            return new House(width, height, opt_width, opt_height);
        else if(shapeType.equals("HINOMARU"))
            return new Hinomaru(width, height, opt_width);
        else
            return null;
    }
    public void setShapeType (String type) {
        shapeType = type;
    }
    public void setSize(int w, int h) {
        width = w;
        height = h;
    }
    public void setOptionSize(int w, int h) {
        opt_width = w;
        opt_height = h;
    }
}
```

ここで、インスタンスを生成するメソッド `build()` は引数を持たず、 それらの値は事前に `setShapeType(String), setSize(int, int), setOptionSizeint(int, int)` で指定される点に注意してください。



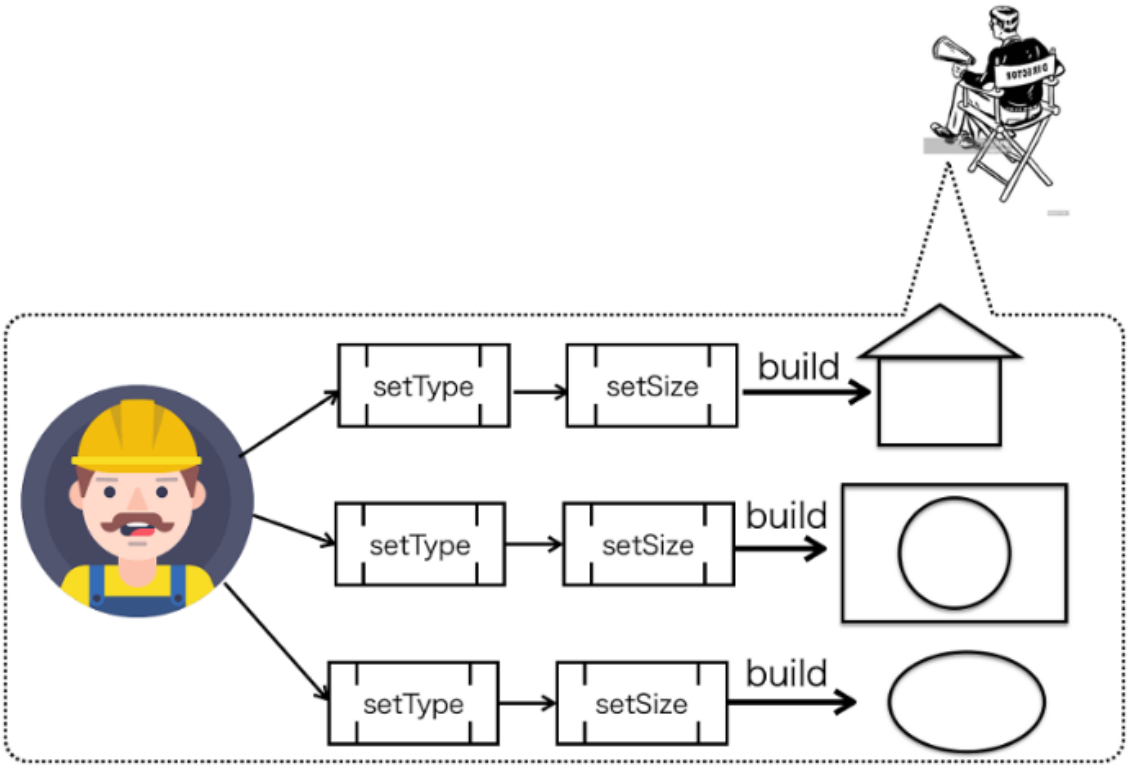
UML を用いた House と Hinomaru の構成図

Directorの導入

Builder パターンでは、クラスごとの生成の手順を指揮するクラスである **Director**を導入するパターンも存在します。ここでは、`ShapeBuilder` に対する構成例を以下に示します。

```
public class ShapeDirector {
    private final ShapeBuilder shapeBuilder;
    public ShapeDirector () {
        shapeBuilder = new ShapeBuilder();
    }
    public Shape construct(String shapeType, int size[]){
        shapeBuilder.setShapeType(shapeType);
        if(shapeType.equals(“ELLIPSE") || shapeType.equals(“RECTANGLE"))
            shapeBuilder.setSize(size[0], size[1]);
        else if (shapeType.equals("HOUSE")) {
            shapeBuilder.setSize(size[0], size[1]);
            shapeBuilder.setOptionSize(size[2], size[3]);
        }
        else if (shapeType.equals("HINOMARU")) {
            shapeBuilder.setSize(size[0], size[1]);
            shapeBuilder.setOptionSize(size[2], 0);
        }
        else
            return null;
        return shapeBuilder.build();
    }
}
```

ここで、インスタンスを生成するメソッド `construct` の引数は、その個数を柔軟に変えられる様に配列で渡されています。



Builder の処理過程を指揮する Director

上記の Director を用いた実行形式は、以下の様に簡素化されます。

```
ShapeDirector shapeDirector = new ShapeDirector();
int size4[] = {20, 30, 22, 10};
shapeDirector.construct("HOUSE", size4).draw();
int size3[] = {50, 30, 10};
shapeDirector.construct("HINOMARU", size3).draw();
```

Director の constructor の中では、Builder パターンではなく Factory パターンを使うことも可能ですが、 Factory はコンストラクタの呼び出しという簡素な形式なのに対して、Builder はオブジェクトが複雑になるに従いメソッドの呼び出しも多く複雑になる傾向があります。

ゆえに、**Director は Builder の複雑な処理過程のカプセル化のためのクラス**として考えられます。

Buildパターン設計法のまとめ

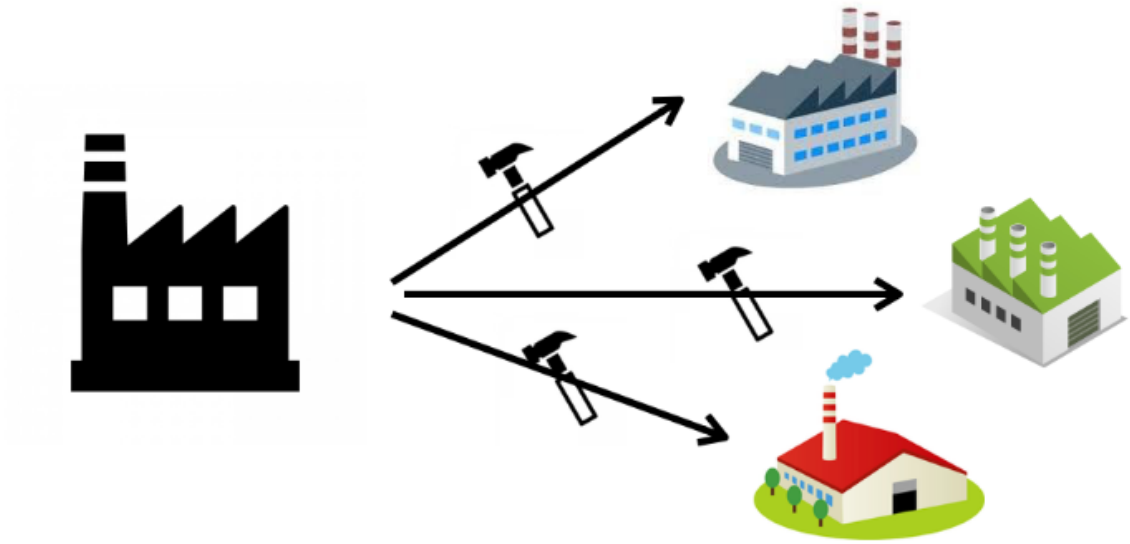
- 製造者（Builder）が製造するオブジェクト群に共通した特徴を、概念クラス（または、インタフェース）として宣言する。
- 製造される全てのオブジェクトは、この共通の概念クラス（または、インタフェース）を継承して作成する。
- 製造者（Builder）のクラスに、製造したオブジェクトを返すメソッド（`build`）を作成する。
- 製造者（Builder）のクラスに、製造するオブジェクトの種類や大きさ、およびその他の諸変数を設定するためのメソッド群を作成し、複雑なオブジェクトの仕様を複数の処理（メソッド呼び出し）を介して指定できる様にする。
- 製造者（Builder）のクラスを用いた複雑なオブジェクトの設計を簡略化してくれる、指揮者（Director）のクラスを用いることも考えられる。

Abstract Factory

Abstract Factory パターンは、直訳すると「概念的な工場」、すなわち工場を生成するための上位概念の工場として捉えられます。

組み合わせて使う多数のサブクラス群を、まとめて交換できるようにするパターンです。

メリットは、実行するメインプログラムを修正することなく、サブクラス群を環境変化に伴って交換することができることです。



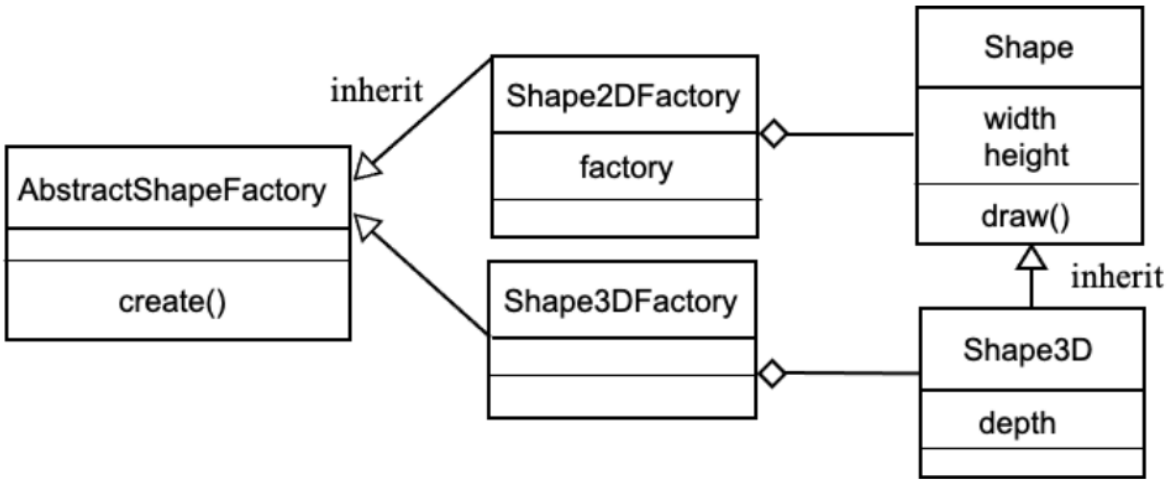
例えば、上記の `Shape` クラスは２次元形状に限られてきましたが、３次元空間での形状を扱う工場を `Shape3DFactory` とし、上記で構成した `ShapaFactory` を `Shape2DFactory` として再定義し、これらが共通な概念クラスである `AbstractShapeFactory` を継承すると設定した場合に、以下の様な構成が考えられます。

```
public abstract class AbstractShapeFactory {
    abstract Shape create(String shapeType, int size[]);
}

public class Shape2DFactory extends AbstractShapeFactory {
    ShapeFactory factory = new ShapeFactory();
    @Override
    Shape create(String type, int size[]) {
        return factory.create(type, size[0], size[1]);
    }
}

public class Shape3DFactory extends AbstractShapeFactory {
    @Override
    Shape create(String type, int size[]) {
        if(type.equalsIgnoreCase("CUBOID"))
            return new Cuboid(size[0], size[1], size[2]);
        else
            return null;
    }
}
```

ただし、３次元形状の Cuboid（直方体）は、形状クラス `Shape` を継承して例えば以下の様に構成されます。

```
public class Cuboid extends Shape {
    int depth;
    public Cuboid(int w, int h, int d) {
        super(w, h);
        depth = d;
    }
   @Override
    public void draw() {
        // draw cuboid 直方体
    }
}
```

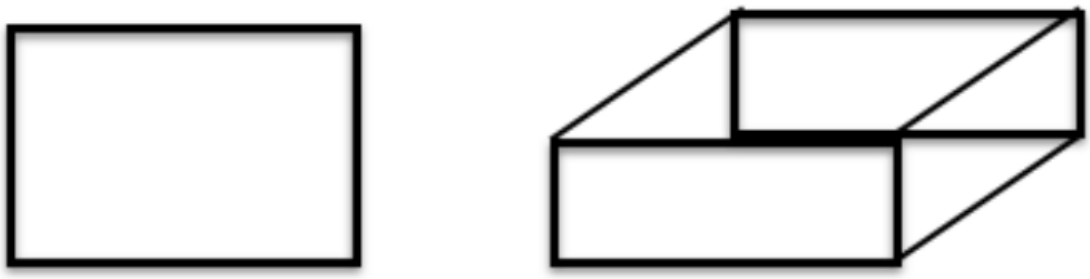
ここで、`depth`は形状の奥行き値を表すフィールドです。



UML を用いた AbstractFactory の構成図

これらを用いて２次元と３次元の形状を生成するコードは、以下の様になります。

```
Shape2DFactory shape2DFactory = new Shape2DFactory();
int size_2d[] = {30, 20};
shape2DFactory.create("RECTANGLE", size_2d).draw();
Shape3DFactory shape3DFactory = new Shape3DFactory();
int size_3d[] = {30, 10, 20};
shape3DFactory.create("CUBOID", size_3d).draw();
```



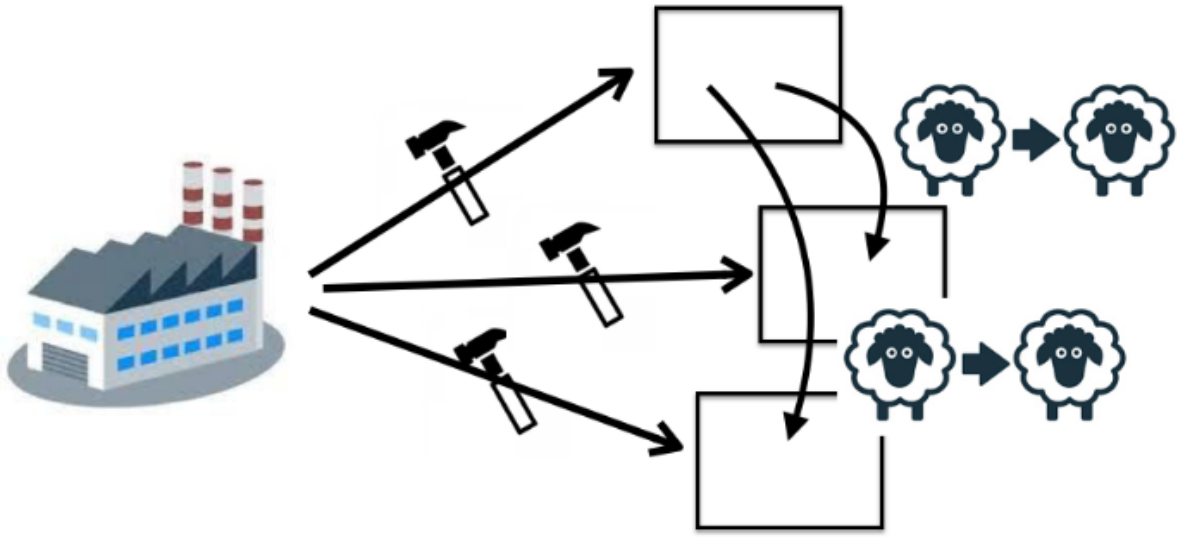
２種類の工場で生成される２／３次元形状

Abstract Factoryパターン設計法のまとめ

- 複数の種類の工場（Factory）に共通する要素（例えば、製造のメソッド `create`）を含む、概念クラス（または、インタフェース）を宣言する。
- 各工場は、この概念クラス（または、インタフェース）を継承して作成する。

Prototype

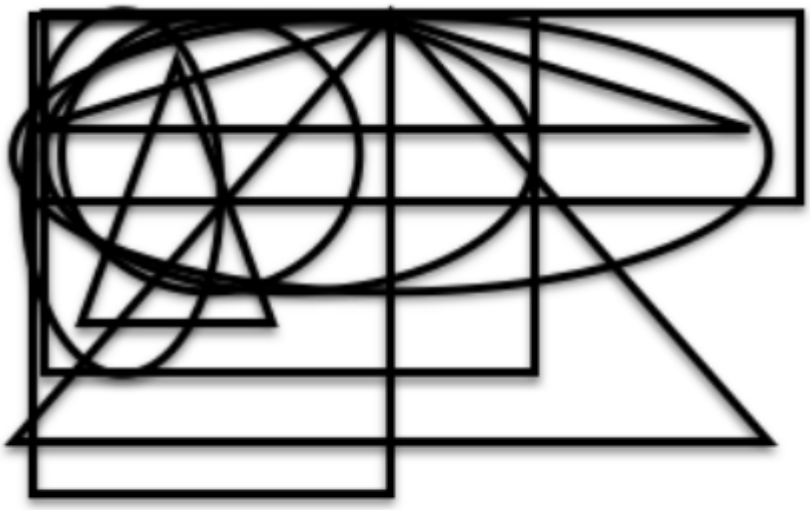
Prototype は、オブジェクトが多数個生成され、各オブジェクトの生成には多くの計算資源やメモリ資源を必要とする場合に、**原型となるオブジェクトのコピー（クローン化）を用いる**
ことによって、オブジェクト生成の計算およびメモリのコストを節約するためのパターンです。



Prototype の概念図

ここで、乱数によって形状の型とサイズ（幅と高さ）が決定される幾何形状オブジェクトを10個生成する、以下のコードを考えます。

```
import java.util.Random;

ShapePrototypeFactory sp_factory = new ShapePrototypeFactory();
Random rnd = new Random();
for (int i=0; i < 10; i++) {
    int type = rnd.nextInt(3);  // type = 0,1,2
    int width = rnd.nextInt(5) + 1; // width = 1,2,3,4,5
    int height = rnd.nextInt(5) + 1; // height = 1,2,3,4,5
    switch (type) {
        case 0: sp_factory.create("RECTANGLE").draw(width, height);
                break;
        case 1: sp_factory.create("ELLIPSE").draw(width, height);
                break;
        case 2: sp_factory.create("TRIANGLE").draw(width, height);
                break;
    }
}
```



上記コードの生成例

sp_factoryは、生成するオブジェクトが同じ形状の場合は、そのサイズが異なっても 原型となるオブジェクト（最初にインスタンス化されたオブジェクト）のクローンを用いて新たなオブジェクトを生成します。これは、コンストラクタを用いたオブジェクトの生成に、多くの計算資源とメモリ資源が必要であるときに有効です。

この Prototype として定義された `ShapePrototypeFactory` は、以下の様に宣言されます。

```
public class ShapePrototypeFactory {
    HashMap<String, ShapeCloneable> protoMap;

    public ShapePrototypeFactory (){
        protoMap = new HashMap<String, ShapeCloneable>();
        protoMap.put ("RECTANGLE", new RectangleCloneable ());
        protoMap.put ("ELLIPSE", new EllipseCloneable ());
        protoMap.put ("TRIANGLE", new TriangleCloneable ());
    }

    public ShapeCloneable create (String type) {
        return protoMap.get (type).clone ();
    }
}
```

ここで、このクラスのコンストラクタ内では、形状クラスの原型となるオブジェクト（`RectangleCloneable, EllipseCloneable, TriangleCloneable`）が生成されて、辞書の様に扱われる`HashMap`クラスのオブジェクト`protoMap`に格納されます。

これらの原型となるオブジェクトに対しては、元の形状オブジェクト（Rectangle, Ellipse, Triangle）に対して**クローン化を可能とする**、以下の様な拡張を施します。

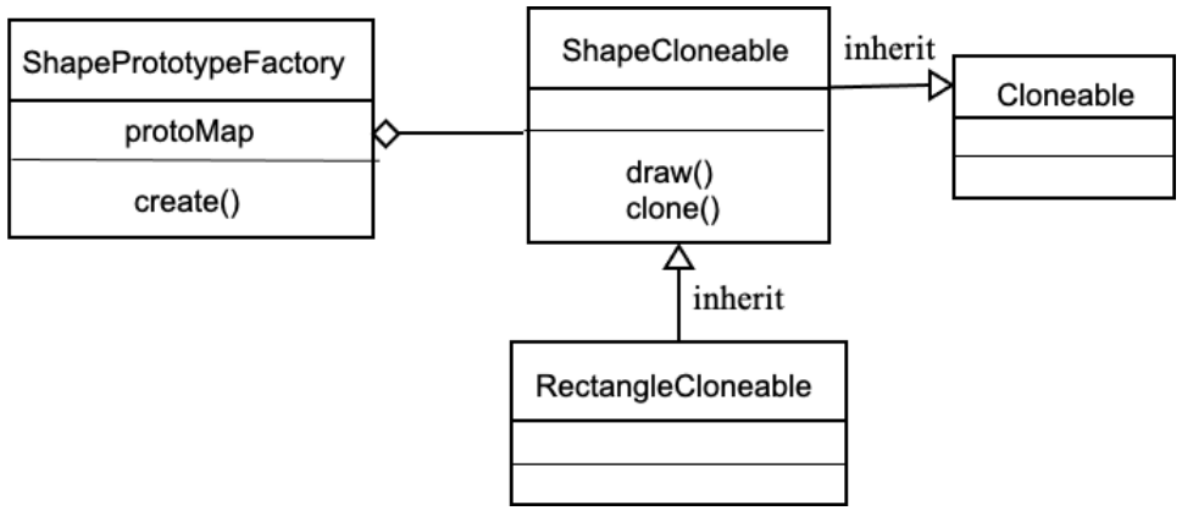
```
public abstract class ShapeCloneable implements Cloneable {
    public ShapeCloneable () {
    }
    abstract void draw(int w, int h);
    @Override
    public ShapeCloneable clone () {
        ShapeCloneable cloned = null;
        try {
            cloned = (ShapeCloneable) super.clone ();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace ();
        }
        return cloned;
    }
}
```

ここで、Cloneableはクローン化を導入するための Java の標準クラスです。

この概念クラスを継承して、例えばRectangleクラスは以下の様に拡張されます。

```
public class RectangleCloneable extends ShapeCloneable {
    public RectangleCloneable () {
        super();
    }
    @Override
    public void draw(int w, int h) {
       // draw rectangle using width == w and height == h
```

```
    }
}
```



UML を用いた Prototype の構成図

Prototypeパターン設計法のまとめ

- オブジェクトを生成するクラスのフィールドに、各種オブジェクトの原型となるインスタンスを登録するためのフィールド（例えば、HashMap）を設定する。
- 原型として登録されるオブジェクトを、クローン化が可能なクラスとして構成する。具体的には、`Cloneable` インタフェースを実装した概念クラスを継承して作成する。
- オブジェクトを生成するメソッド（`create`）が呼ばれた際には、登録された原型オブジェクトのコピーを、そのオブジェクトの `clone` メソッドを用いて返す。