発展プログラミング

第3回：デザインパターンの基礎

宮田章裕 <miyata.akihiro@nihon-u.ac.jp>

前回講義の復習

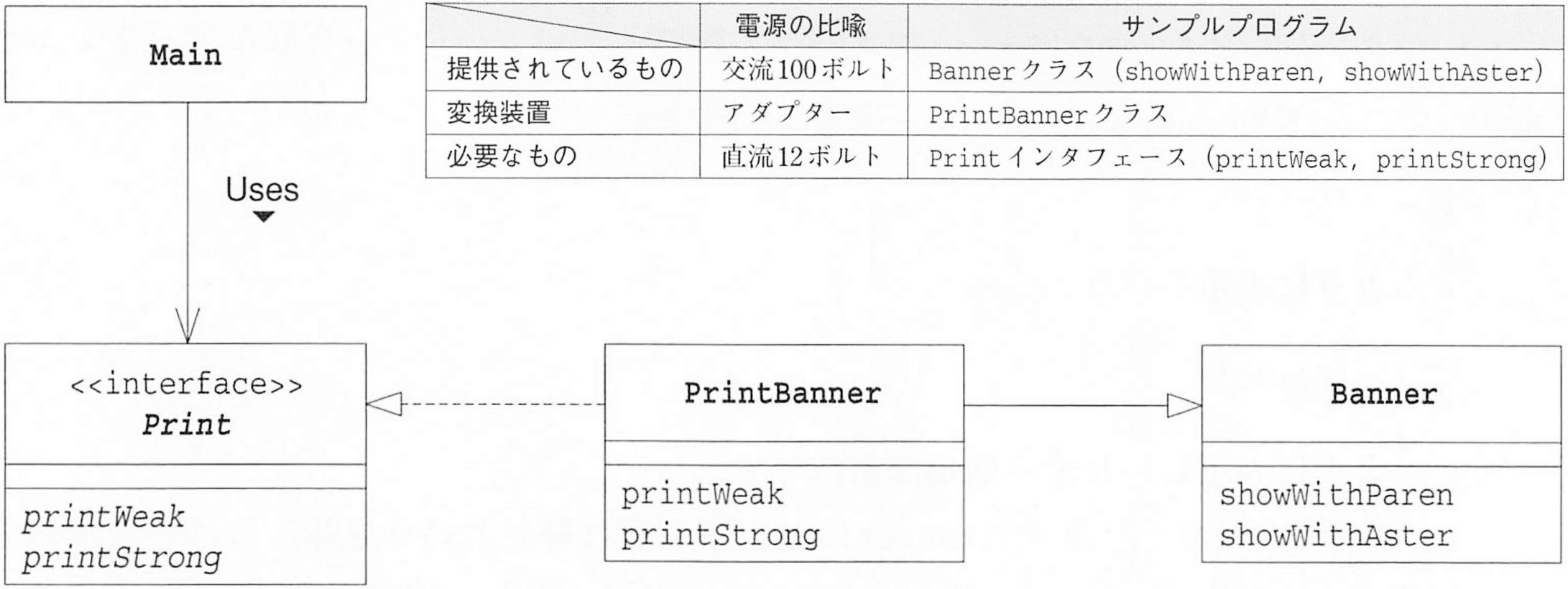
デザインパターン

- the Gang of Four（GoF: E. Gamma, R. Helm, R. Johnson, J. Vlissiders）が整理した，プログラミングにおいて頻出するパターン

- デザインパターンを学ぶ主目的：再利用しやすいコードを作成できるようになる

- あるいは，既存コードを再利用する（せざるをえない）ため，
  という理由の方が学生のみなさんには共感しやすいかもしれない
  - 例1：あるコード群がライブラリとして提供されていて，自分にはそれらの変更権限が無い
  - 例2：試験が済んだ既存コード群があり，そこに手を入れると試験がやり直しになってしまう

継承を用いたAdapterパターン (1/2)

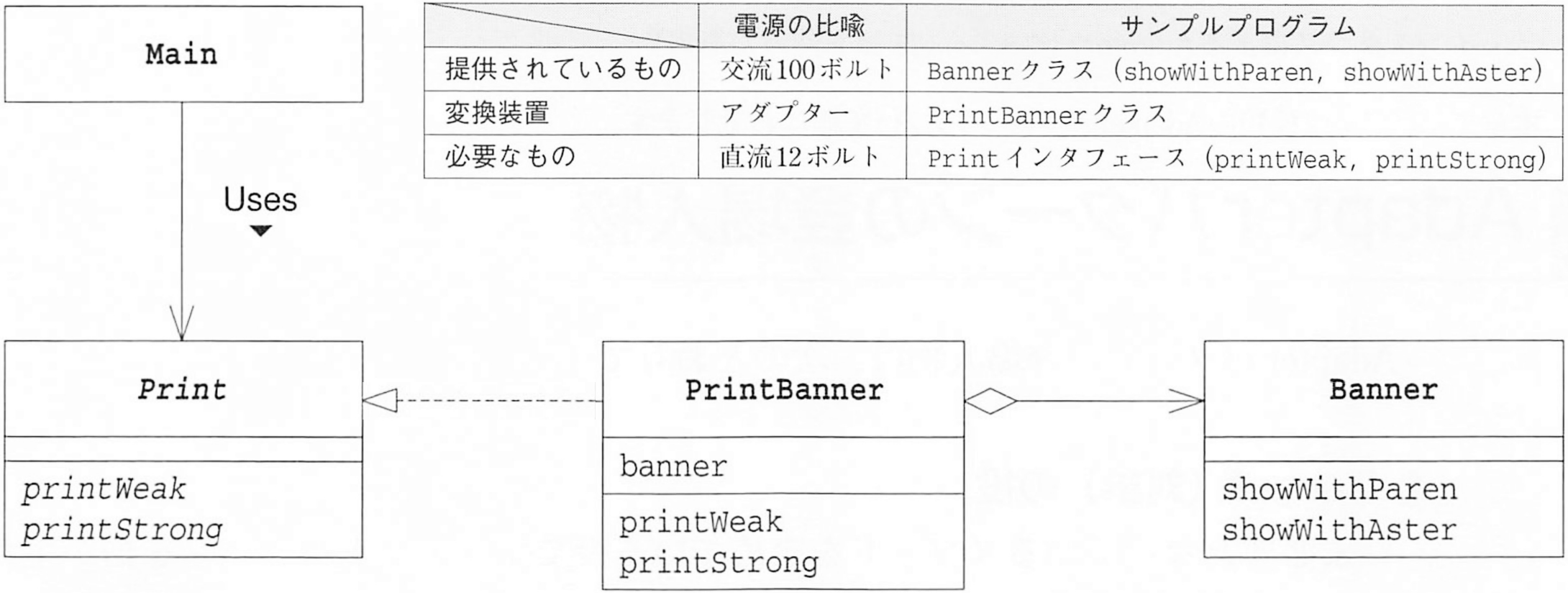


	電源の比喩	サンプルプログラム
提供されているもの	交流100ボルト	Bannerクラス（showWithParen, showWithAster）
変換装置	アダプター	PrintBannerクラス
必要なもの	直流12ボルト	Printインタフェース（printWeak, printStrong）

結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用

移譲を用いたAdapterパターン (1/2)

	電源の比喩	サンプルプログラム
提供されているもの	交流100ボルト	Bannerクラス（showWithParen, showWithAster）
変換装置	アダプター	PrintBannerクラス
必要なもの	直流12ボルト	Printインタフェース（printWeak, printStrong）



結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用（誤りと思われる点を一部修正）

Adapterパターンのまとめと利用シーン (1/2)

- 必要なコードと，提供されているコードとの間に入って，その間を埋めるコードを書くという考え方

- 利用シーン1
  - 既に仕様が決まっており，自分には仕様を変更する権限が無い
  - 仕様と似たことが達成できそうだが，微妙に挙動が異なる外部ライブラリがある

  → 仕様と外部ライブラリの溝を埋めるAdapterプログラムを書けば良い

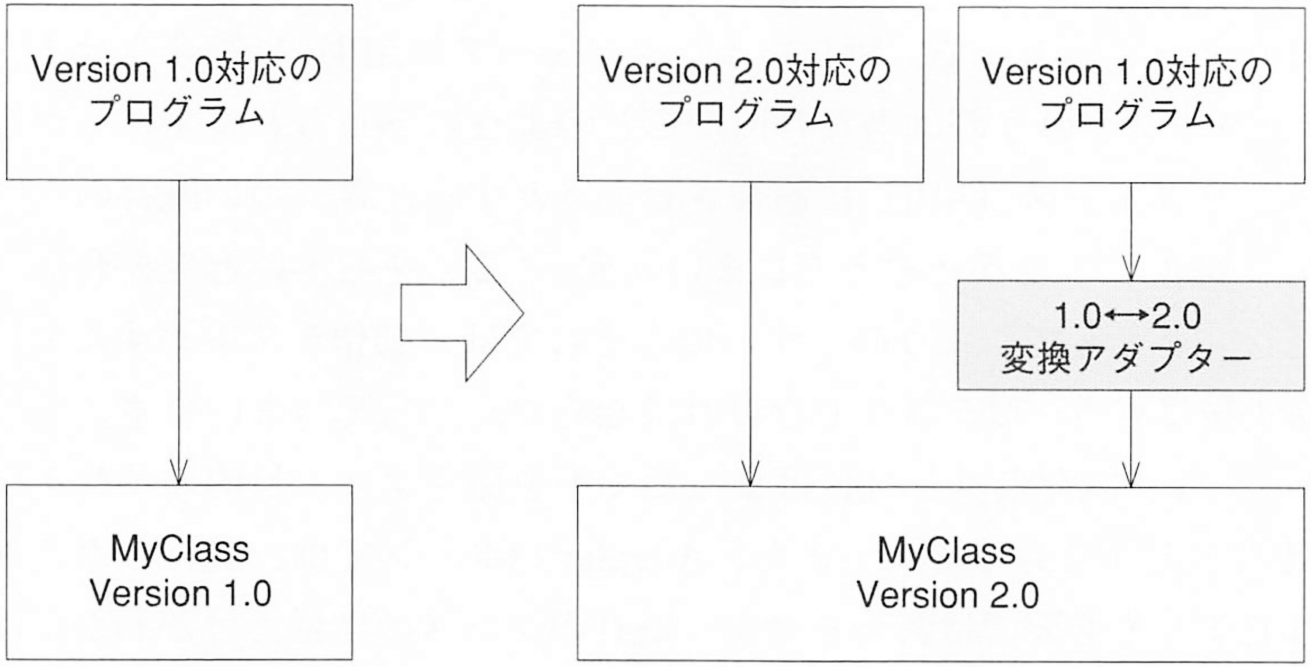
- 利用シーン2
  - 既に徹底的にテストされた巨大な既存プログラムがあり，できればそこに手を入れたくない
  - 既存プログラムに新機能を追加するライブラリがあるが，既存プログラムのインタフェースと整合しておらず，そのままライブラリを導入することはできない

  → 既存プログラムとライブラリの溝を埋めるAdapterプログラムを書けば良い
  （仮にバグが生じても，原因はAdapterかライブラリにあると絞り込める）

Adapterパターンのまとめと利用シーン (2/2)

- 利用シーン3
  - システムのバージョンアップを行いたいが，それを行うと旧バージョンのブラウザをサポートできなくなる
  - 旧バージョンのブラウザのコードを全部書き直すリソース（金・時間）は無い

→ 新システムと旧バージョンブラウザ関連コードの溝を埋めるAdapterプログラムを書けば良い

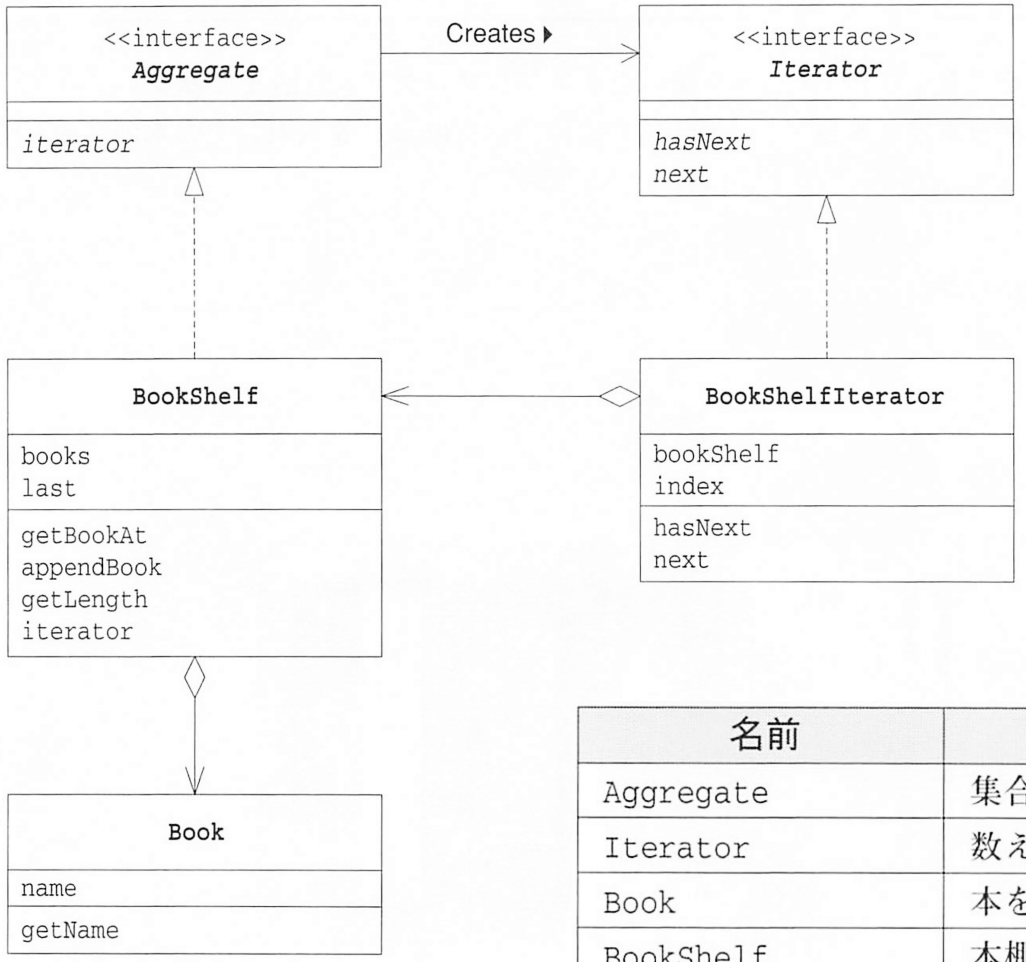


結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用

Iteratorパターン

- 何かがたくさん集まっているとき，各要素を順番に指し示していく処理を方法を統一するための考え方
- iterate：繰り返す
- for／whileで配列を走査する際のループ変数iを抽象化したものと捉えてもよい

Iteratorパターン (1/5)

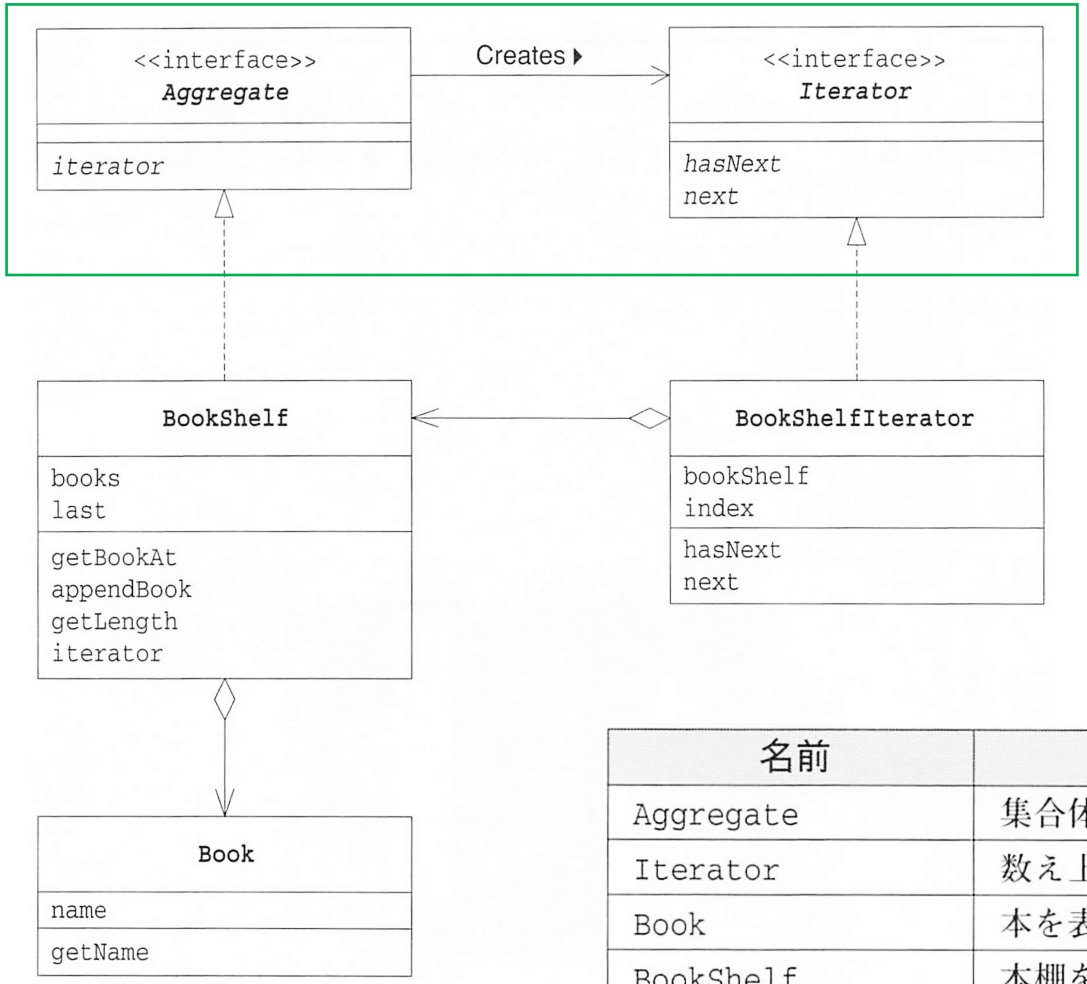


名前	解　説
Aggregate	集合体を表すインタフェース
Iterator	数え上げ、スキャンを行うインタフェース
Book	本を表すクラス
BookShelf	本棚を表すクラス
BookShelfIterator	本棚をスキャンするクラス
Main	動作テスト用のクラス

結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用

Iteratorパターン (2/5)

Aggregate（集合体）は
数え上げの道具を
返却できる。



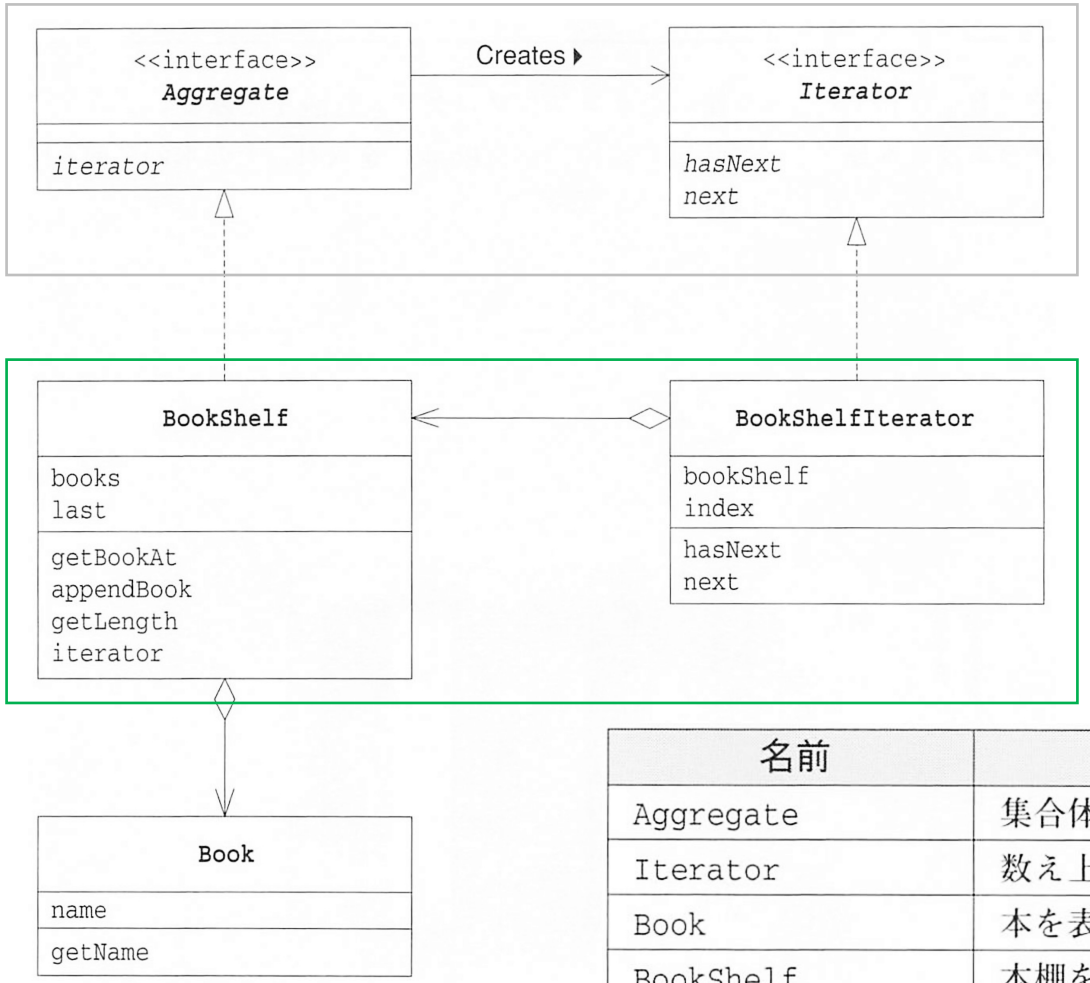
Iterator（数え上げの道具）は
集合体を走査して
次の要素の有無（hasNext）と，
次の要素（next）を返却できる

名前	解　説
Aggregate	集合体を表すインタフェース
Iterator	数え上げ、スキャンを行うインタフェース
Book	本を表すクラス
BookShelf	本棚を表すクラス
BookShelfIterator	本棚をスキャンするクラス
Main	動作テスト用のクラス

結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用

Iteratorパターン (3/5)

Aggregate（集合体）は
数え上げの道具を
返却できる。



Iterator（数え上げの道具）は
集合体を走査して
次の要素の有無（hasNext）と，
次の要素（next）を取得できる

BookShelfクラスは
Aggregateインタフェースを
具体的に実装している

BookShelfIteratorクラスは
Iteratorインタフェースを
具体的に実装している

名前	解　説
Aggregate	集合体を表すインタフェース
Iterator	数え上げ、スキャンを行うインタフェース
Book	本を表すクラス
BookShelf	本棚を表すクラス
BookShelfIterator	本棚をスキャンするクラス
Main	動作テスト用のクラス

結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用

Iteratorパターンのまとめと利用シーン

- 何かがたくさん集まっているとき，各要素を順番に指し示していく処理を方法を統一するための考え方

- 利用シーン
  - 集合体を走査したいが，集合体の表現が変わる可能性がある場合
  - 例えば，配列かもしれないし，ArrayListかもしれないし，未知のデータ構造かもしれない

Singletonパターン

Singletonパターンの必要性

- ❖通し番号が入ったチケットを発券するシーンを考える
- ❖下記コードではTicketMakerクラスのインスタンスを複数作成できてしまうため，通し番号の重複が起きてしまう

TicketMaker.java

```java
public class TicketMaker {
    private int ticket = 1000;

    public int getNextTicketNumber() {
        return ticket++;
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {

        TicketMaker tMaker1 = new TicketMaker();

        for(int i = 0; i < 5; i++) {
            System.out.println("No. " + tMaker1.getNextTicketNumber());
        }

        TicketMaker tMaker2 = new TicketMaker();

        for(int i = 0; i < 3; i++) {
            System.out.println("No. " + tMaker2.getNextTicketNumber());
        }
    }
}
```

TicketMakerクラスのインスタンスを
自由に何個でも作成できてしまう

実行結果

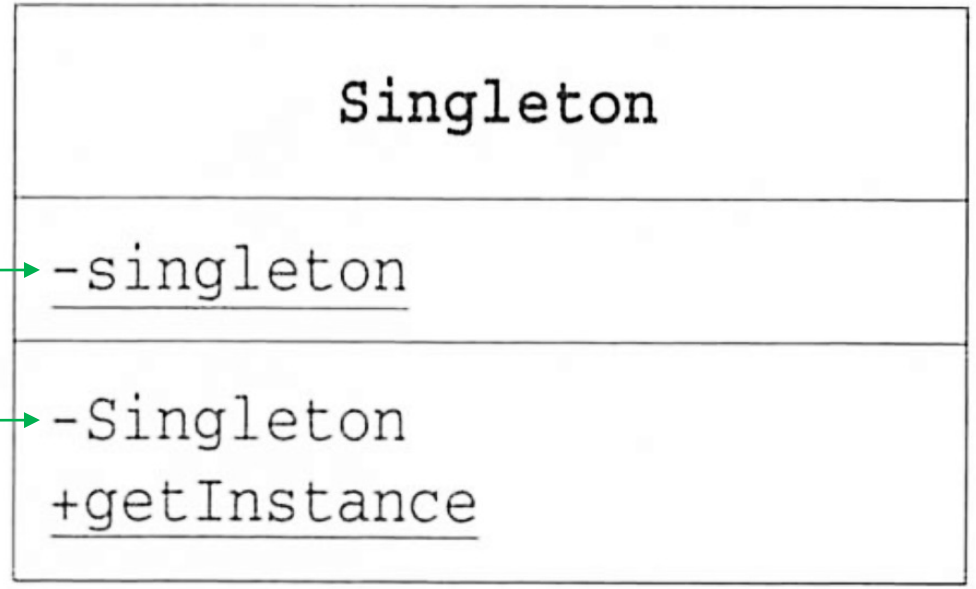
```
No. 1000
No. 1001
No. 1002
No. 1003
No. 1004
No. 1000
No. 1001
No. 1002
```

Singletonパターン (1/2)

- 「－」はprivate，「＋」はpublic，下線はstaticを表す
- コンストラクタがprivateであり，このクラスのインスタンスを取得するためにはgetInstance()を利用しないといけないのがポイント

このクラスのインスタンスが
private（外からアクセス不可）かつ
static（インスタンス変数ではなくクラス変数）

コンストラクタがprivate



結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用

Singletonパターン (2/2)

Singleton.java

```java
public class Singleton {

    private static Singleton singleton = new Singleton();

    private Singleton() {
        System.out.println("The instance of Singleton is created.");
    }

    public static Singleton getInstance() {
        return singleton;
    }
}
```

全体を通して唯一で（static）
外部から直接アクセスできない（private）
Singletonクラスインスタンスを作成

Singletonクラスのインスタンスを
取得するためにはこのメソッドを利用

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();
        if(obj1 == obj2) {
            System.out.println("obj1 equals to obj2.");
        }
    }
}
```

実行結果

```
obj1 equals to obj2.
```

作業準備

- 本日の作業ディレクトリの作成・移動
  - mkdir -p SOMEWHERE/2021_ap/03
  - 以降，SOMEWHERE/2021_ap/03をWORK_DIRとする

- 「Singletonパターン IR03-1」の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 03: Singleton, Template Method & Builder > singleton_prac.zip をWORK_DIRにダウンロード
  - unzip singleton_prac.zip
  - cd singleton_prac

IR03-1

- singleton_prac内のプログラムにSingletonパターンを適用し，チケットが通し番号になることが保証されるように修正せよ。

目標の結果

```
No. 1000
No. 1001
No. 1002
No. 1003
No. 1004
No. 1005
No. 1006
No. 1007
```

IR03-1 解答例

TicketMaker.java

```java
public class TicketMaker {
    private static TicketMaker tMaker = new TicketMaker();
    private int ticket = 1000;

    private TicketMaker() {
        System.out.println(
            "The instance of TicketMaker is created.");
    }

    public static TicketMaker getInstance() {
        return tMaker;
    }

    public int getNextTicketNumber() {
        return ticket++;
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        TicketMaker tMaker1 = TicketMaker.getInstance();
        for(int i = 0; i < 5; i++) {
            System.out.println(
                "No. " + tMaker1.getNextTicketNumber());
        }

        TicketMaker tMaker2 = TicketMaker.getInstance();
        for(int i = 0; i < 3; i++) {
            System.out.println(
                "No. " + tMaker2.getNextTicketNumber());
        }
    }
}
```

Singletonパターンのまとめと利用シーン

- インスタンスが1個しか生成されないことを保証するための考え方

- 利用シーン
  - カウンタ（例：通し番号），状態管理変数（例：システムの状態），リソースへのアクセス（例：NWインタフェースにアクセスするインスタンス）などを，アプリケーション全体を通して唯一のものとして扱いたい場合

Template Methodパターン

現実世界におけるTemplate

- 文字の形に穴が空いている薄いプラスチックの板
  - どの文字になるかを定義している
  - 具体的にどのような筆記具（例：鉛筆，ペン）で書くかは定義していない

デザインパターンにおけるTemplate Method

- 処理の大きな流れをスーパークラスで定義
- 各処理の具体的な内容をサブクラスで定義

Template Methodパターンの説明のための想定シーン

- ❖文字や，文字列を5回出力したい（→ 大きな流れは共通）
- ❖ただし，文字を出力する場合と，文字列を出力する場合は，異なる挙動としたい

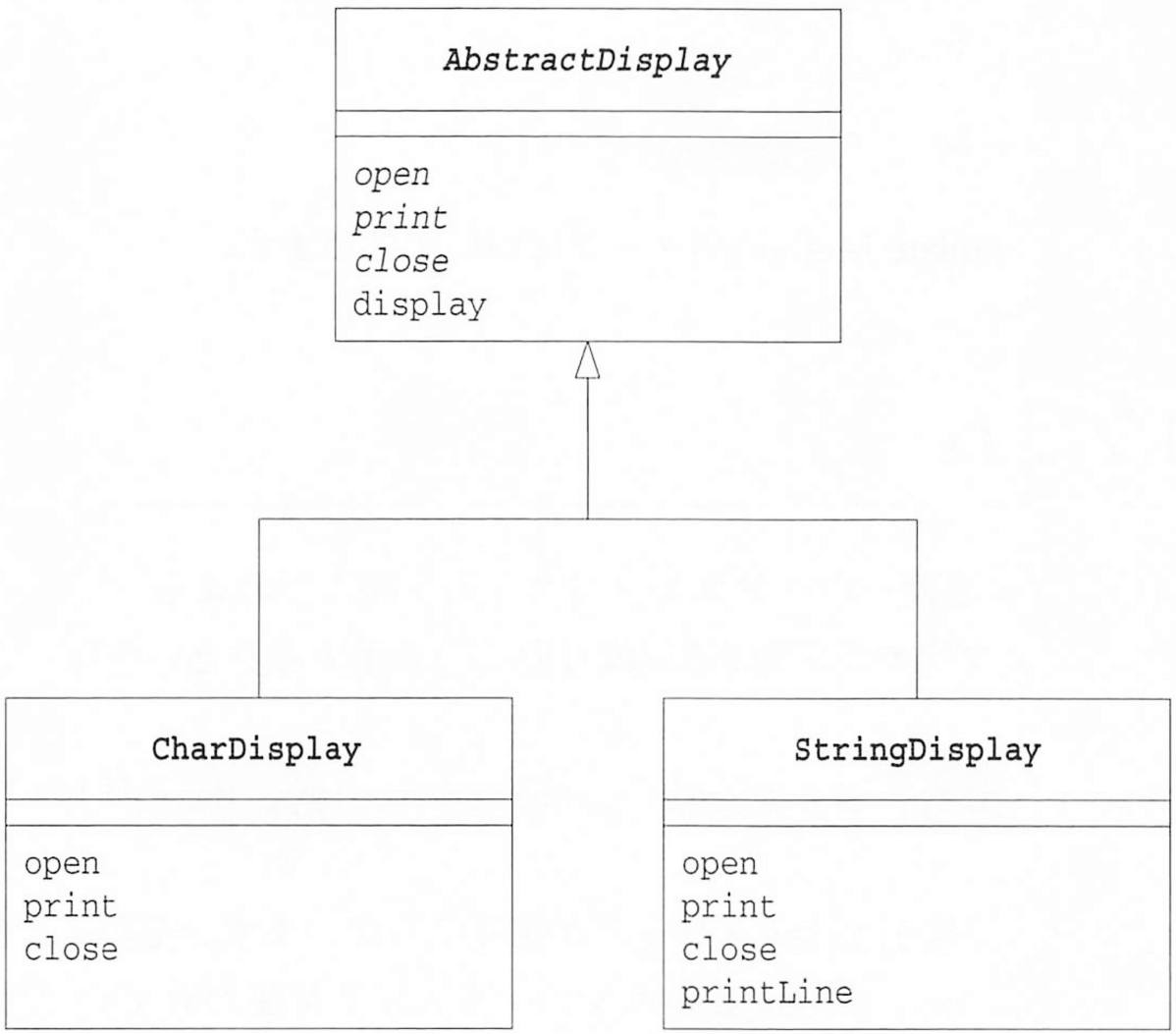
目標の結果（文字の場合）

```
<<NNNNN>>
```

目標の結果（文字列の場合）

```
+-----------+
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
+-----------+
```

Template Methodパターン (1/2)



名　前	解　説
AbstractDisplay	メソッドdisplayのみ実装されている抽象クラス
CharDisplay	メソッドopen，print，closeを実装しているクラス
StringDisplay	メソッドopen，print，closeを実装しているクラス
Main	動作テスト用のクラス

結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用

Template Methodパターン (2/2)

Main.java

```java
public class Main {
    public static void main(String[] args) {
        AbstractDisplay chDisplay = new CharDisplay('N');
        AbstractDisplay stDisplay = new StringDisplay("Nihon Univ.");
        chDisplay.display();
        stDisplay.display();
    }
}
```

AbstractDisplay.java

```java
public abstract class AbstractDisplay {
    public abstract void open();
    public abstract void print();
    public abstract void close();

    public final void display() {
        open();
        for(int i = 0; i < 5; i++) {
            print();
        }
        close();
    }
}
```

処理の大きな流れを
定義している

CharDisplay.java

```java
public class CharDisplay extends AbstractDisplay {
    private char ch;

    public CharDisplay(char ch) {
        this.ch = ch;
    }

    public void open() {
        System.out.print("<<");
    }

    public void print() {
        System.out.print(ch);
    }

    public void close() {
        System.out.println(">>");
    }
}
```

StringDisplay.java

```java
public class StringDisplay extends AbstractDisplay {
    private String str;

    public StringDisplay(String str) {
        this.str = str;
    }

    public void open() {
        printLine();
    }

    public void print() {
        System.out.println("|" + str + "|");
    }

    public void close() {
        printLine();
    }

    private void printLine() {
        System.out.print("+");
        for(int i = 0; i < str.length(); i++) {
            System.out.print("-");
        }
        System.out.println("+");
    }
}
```

各処理の具体的な内容を定義している

作業準備

- 「Template Methodパターン IR03-2」の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 03: Singleton, Template Method & Builder > template_method_prac.zip をWORK_DIRにダウンロード
  - unzip template_method_prac.zip
  - cd template_method_prac

IR03-2

- template_method_prac内にて，必要最低限の新クラス作成と既存クラス修正を行い，目標の結果を得られるプログラムを作成せよ。

目標の結果（最後の1行以外は配布プログラムで実装済み）

```
<<NNNNN>>
+-----------+
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
+-----------+
{ 100 100 100 100 100 }
```

IR03-2 解答例（下記以外のクラスは変更不要）

Main.java

```java
public class Main {
    public static void main(String[] args) {
        AbstractDisplay chDisplay = new CharDisplay('N');
        AbstractDisplay stDisplay = new StringDisplay("Nihon Univ.");
        AbstractDisplay nmDisplay = new NumberDisplay(100);
        chDisplay.display();
        stDisplay.display();
        nmDisplay.display();
    }
}
```

NumberDisplay.java（新規作成）

```java
public class NumberDisplay extends AbstractDisplay {
    private int num;

    public NumberDisplay(int num) {
        this.num = num;
    }

    public void open() {
        System.out.print("{");
    }

    public void print() {
        System.out.print(" " + num);
    }

    public void close() {
        System.out.println(" }");
    }
}
```

実行結果

```
<<NNNNN>>
+-----------+
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
|Nihon Univ.|
+-----------+
{ 100 100 100 100 100 }
```

NumberDisplayクラスでは，
抽象クラスであるスーパークラスAbstractDisplayに
実装を義務付けられているopen()，print()，close()を実装するだけでよい。
大まかな処理の流れはAbstractDisplayクラスが定義してくれている。

Template Methodパターンのまとめと利用シーン

- スーパークラスで処理の枠組みを定め，サブクラスでその具体的内容を定めるという考え方

- 利用シーン
  - 似てはいるが，具体的な処理が異なる複数のクラスを作成したい場合
    → 個々のクラスで大枠の処理のアルゴリズムを考えなくてよい
    → 大まかな処理にバグがあっても，スーパークラスを修正するだけで済む

Builderパターン

Builderパターンの説明のための想定シーン

- 下記の構造の文書を作成したい
  - タイトルを1つ含む
  - 文字列をいくつか含む
  - 箇条書き項目をいくつか含む

  → この文章構造の定義が複雑

- 文書の具体的なフォーマットは
  プレーンテキストとHTML

プレーンテキスト

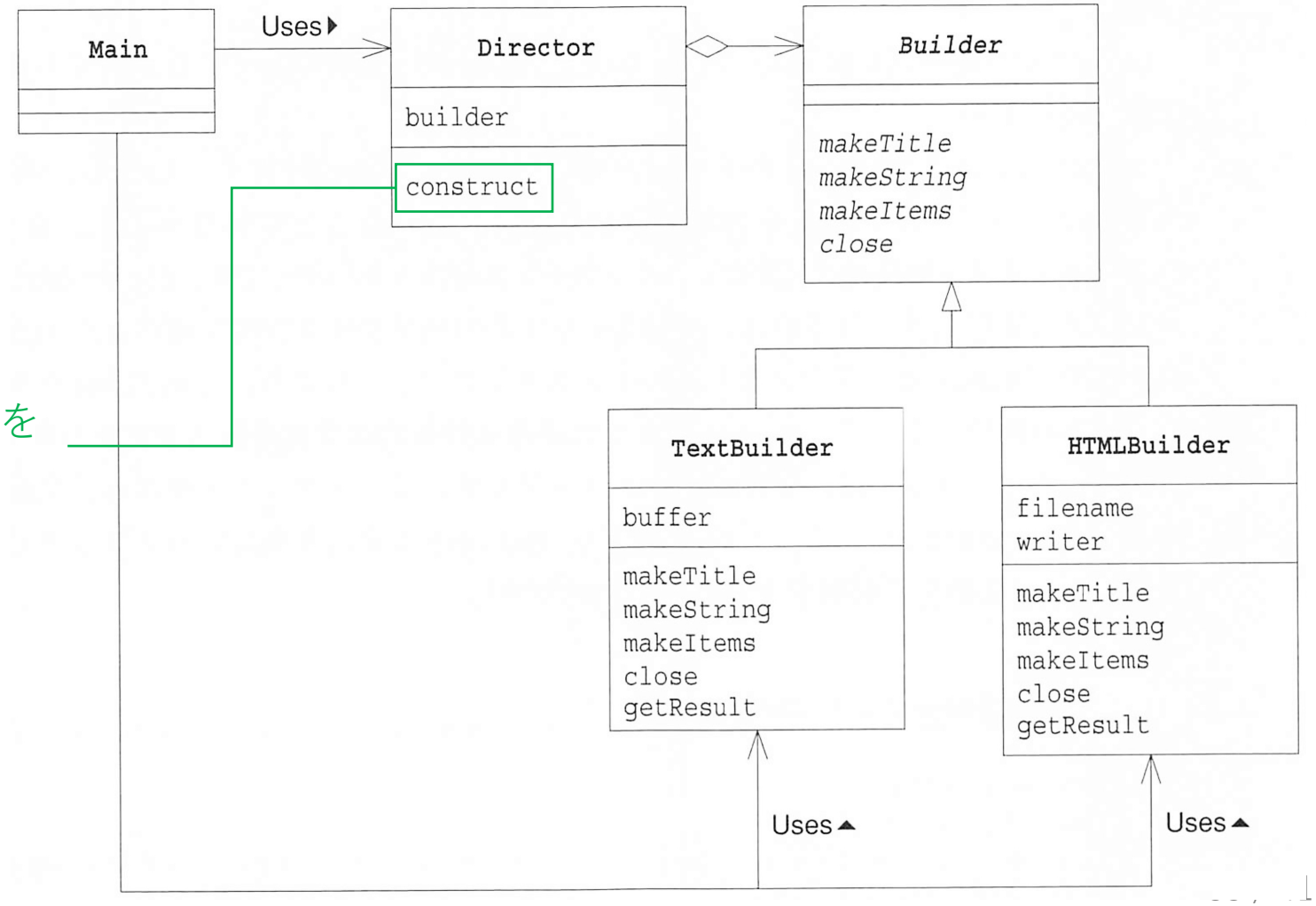
```
==============================
[Greeting]
+ Morning and daytime
 - Good morning
 - Good afternoon

+ Evening
 - Good evening
 - Good night

==============================
```

HTML

```
<html>
<head><title>Greeting</title></head>
<body>
<h1>Morning and daytime</h1>
<ul>
<li>Good morning</li>
<li>Good afternoon</li>
</ul>
<h1>Evening</h1>
<ul>
<li>Good evening</li>
<li>Good night</li>
</ul>
</body>
</html>
```

Builderパターン (1/4)

名　前	解　説
Builder	文書を構成するためのメソッドを定めた抽象クラス
Director	1つの文書を作るクラス
TextBuilder	プレーンテキスト（普通の文字列）を使って文書を作るクラス
HTMLBuilder	HTMLファイルを使って文書を作るクラス
Main	動作テスト用のクラス



タイトル，文字列，箇条書き項目を
適切な順番で必要数記載するという複雑な作業を
construct()内で行う（隠蔽する）ことで
Mainクラスで複雑な処理を行わなくて済む

結城浩：増補改訂版 Java言語で学ぶデザインパターン入門 から引用

Builderパターン (2/4)

Main.java

```java
public class Main {
    public static void main(String[] args) {
        TextBuilder tBuilder = new TextBuilder();
        Director tDirector = new Director(tBuilder);
        tDirector.construct();
        System.out.println(tBuilder.getResult());

        HtmlBuilder hBuilder = new HtmlBuilder();
        Director hDirector = new Director(hBuilder);
        hDirector.construct();
        System.out.println(hBuilder.getResult());
    }
}
```

Director.java

```java
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }

    public void construct() {
        builder.makeTitle("Greeting");
        builder.makeString("Morning and daytime");
        builder.makeItems(new String[] {
            "Good morning", "Good afternoon"});
        builder.makeString("Evening");
        builder.makeItems(new String[] {
            "Good evening", "Good night"});
        builder.close();
    }
}
```

Builder.java

```java
public abstract class Builder {
    public abstract void makeTitle(String title);
    public abstract void makeString(String str);
    public abstract void makeItems(String[] items);
    public abstract void close();
}
```

TextBuilder.java

```java
public class TextBuilder extends Builder {
    private final String LINE
        = "==============================\n";
    private StringBuffer buffer = new StringBuffer();

    public void makeTitle(String title) {
        buffer.append(LINE);
        buffer.append("[" + title + "]\n");
    }

    public void makeString(String str) {
        buffer.append("+ " + str + "\n");
    }

    public void makeItems(String[] items) {
        for(int i = 0; i < items.length; i++) {
            buffer.append(" - " + items[i] + "\n");
        }
        buffer.append("\n");
    }

    public void close() {
        buffer.append(LINE);
    }

    public String getResult() {
        return buffer.toString();
    }
}
```

HtmlBuilder.java

```java
public class HtmlBuilder extends Builder {
    private StringBuffer buffer = new StringBuffer();

    public void makeTitle(String title) {
        buffer.append("<html>\n");
        buffer.append(
            "<head><title>" + title
            + "</title></head>\n");
        buffer.append("<body>\n");
    }

    public void makeString(String str) {
        buffer.append("<h1>" + str + "</h1>\n");
    }

    public void makeItems(String[] items) {
        buffer.append("<ul>\n");
        for(int i = 0; i < items.length; i++) {
            buffer.append(
                "<li>" + items[i] + "</li>\n");
        }
        buffer.append("</ul>\n");
    }

    public void close() {
        buffer.append("</body>\n");
        buffer.append("</html>\n");
    }

    public String getResult() {
        return buffer.toString();
    }
}
```

Builderパターン (3/4)

Main.java

```java
public class Main {
    public static void main(String[] args) {
        TextBuilder tBuilder = new TextBuilder();
        Director tDirector = new Director(tBuilder);
        tDirector.construct();
        System.out.println(tBuilder.getResult());

        HtmlBuilder hBuilder = new HtmlBuilder();
        Director hDirector = new Director(hBuilder);
        hDirector.construct();
        System.out.println(hBuilder.getResult());
    }
}
```

Mainクラスではconstruct()を呼び出すだけで
複雑な作業は行わなくて済む

Director.java

```java
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }

    public void construct() {
        builder.makeTitle("Greeting");
        builder.makeString("Morning and daytime");
        builder.makeItems(new String[] {
            "Good morning", "Good afternoon"});
        builder.makeString("Evening");
        builder.makeItems(new String[] {
            "Good evening", "Good night"});
        builder.close();
    }
}
```

タイトル，文字列，箇条書き項目を並べる
複雑な作業はここに隠蔽する

Builderパターン (4/4)

Builderクラスは
サブクラスに
・makeTitle()
・makeString()
・makeItems()
の実装を義務付ける。

各サブクラスはそれぞれ
プレーンテキスト，HTMLで
タイトル，文字列，箇条書きを
出力する方法を実装する。

Builder.java

```
public abstract class Builder {
    public abstract void makeTitle(String title);
    public abstract void makeString(String str);
    public abstract void makeItems(String[] items);
    public abstract void close();
}
```

TextBuilder.java

```
public class TextBuilder extends Builder {
    private final String LINE
        = "==========================\n";
    private StringBuffer buffer = new StringBuffer();

    public void makeTitle(String title) {
        buffer.append(LINE);
        buffer.append("[" + title + "]\n");
    }

    public void makeString(String str) {
        buffer.append("+ " + str + "\n");
    }

    public void makeItems(String[] items) {
        for(int i = 0; i < items.length; i++) {
            buffer.append(" - " + items[i] + "\n");
        }
        buffer.append("\n");
    }

    public void close() {
        buffer.append(LINE);
    }

    public String getResult() {
        return buffer.toString();
    }
}
```

HtmlBuilder.java

```
public class HtmlBuilder extends Builder {
    private StringBuffer buffer = new StringBuffer();

    public void makeTitle(String title) {
        buffer.append("<html>\n");
        buffer.append(
            "<head><title>" + title
            + "</title></head>\n");
        buffer.append("<body>\n");
    }

    public void makeString(String str) {
        buffer.append("<h1>" + str + "</h1>\n");
    }

    public void makeItems(String[] items) {
        buffer.append("<ul>\n");
        for(int i = 0; i < items.length; i++) {
            buffer.append(
                "<li>" + items[i] + "</li>\n");
        }
        buffer.append("</ul>\n");
    }

    public void close() {
        buffer.append("</body>\n");
        buffer.append("</html>\n");
    }

    public String getResult() {
        return buffer.toString();
    }
}
```

BuilderパターンとTemplate Methodパターンの比較

- Builderパターン：
  Director役がBuilder役をコントロール

Director.java

```java
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }

    public void construct() {
        builder.makeTitle("Greeting");
        builder.makeString("Morning and daytime");
        builder.makeItems(new String[] {
            "Good morning", "Good afternoon"});
        builder.makeString("Evening");
        builder.makeItems(new String[] {
            "Good evening", "Good night"});
        builder.close();
    }
}
```

Builder役を
コントロール

Builder.java

```java
public abstract class Builder {
    public abstract void makeTitle(String title);
    public abstract void makeString(String str);
    public abstract void makeItems(String[] items);
    public abstract void close();
}
```

- Template Methodパターン：
  スーパークラスがサブクラスをコントロール

AbstractDisplay.java

```java
public abstract class AbstractDisplay {
    public abstract void open();
    public abstract void print();
    public abstract void close();

    public final void display() {
        open();
        for(int i = 0; i < 5; i++) {
            print();
        }
        close();
    }
}
```

サブクラスを
コントロール

IR03-3

- 先程のtemplate_method_prac内のプログラムを，Builderパターンを用いるように修正せよ。

IR03-3 解答例（下記以外のクラスは変更不要）

Main.java

```
public class Main {
    public static void main(String[] args) {
        AbstractDisplay chDisplay = new CharDisplay('N');
        Director chDirector = new Director(chDisplay);
        chDirector.display();

        AbstractDisplay stDisplay = new StringDisplay("Nihon Univ.");
        Director stDirector = new Director(stDisplay);
        stDirector.display();
    }
}
```

AbstractDisplay.java

```
public abstract class AbstractDisplay {
    public abstract void open();
    public abstract void print();
    public abstract void close();
}
```

Director.java（新規作成）

```
public class Director {
    private AbstractDisplay display;

    public Director(AbstractDisplay display) {
        this.display = display;
    }

    public void display() {
        display.open();
        for(int i = 0; i < 5; i++) {
            display.print();
        }
        display.close();
    }
}
```

Template Methodパターンと異なり，
スーパークラスAbstractDisplayがサブクラスをコントロールしなくなっている。

Builderパターンのまとめと利用シーン

- 複雑な構造を持つインスタンスを組み上げるための考え方

- 利用シーン
  - 複雑な構造のインスタンスの作成を，Mainクラスなどから隠蔽したい場合
    → 複雑な構造設計処理はDirector役に隠蔽される

本日のまとめ

- 講義内容
  - Singletonパターン
  - Template Methodパターン
  - Builderパターン

- 授業内課題提出
  - 各授業内課題（IR）の解答を記載せよ。
  - 「講義内容のまとめ」の解答欄に
    上記「講義内容」の各項目について文章で説明を記載せよ。