発展プログラミング

第4回：デザインパターンの応用

宮田章裕 <miyata.akihiro@nihon-u.ac.jp>

前回講義の復習

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

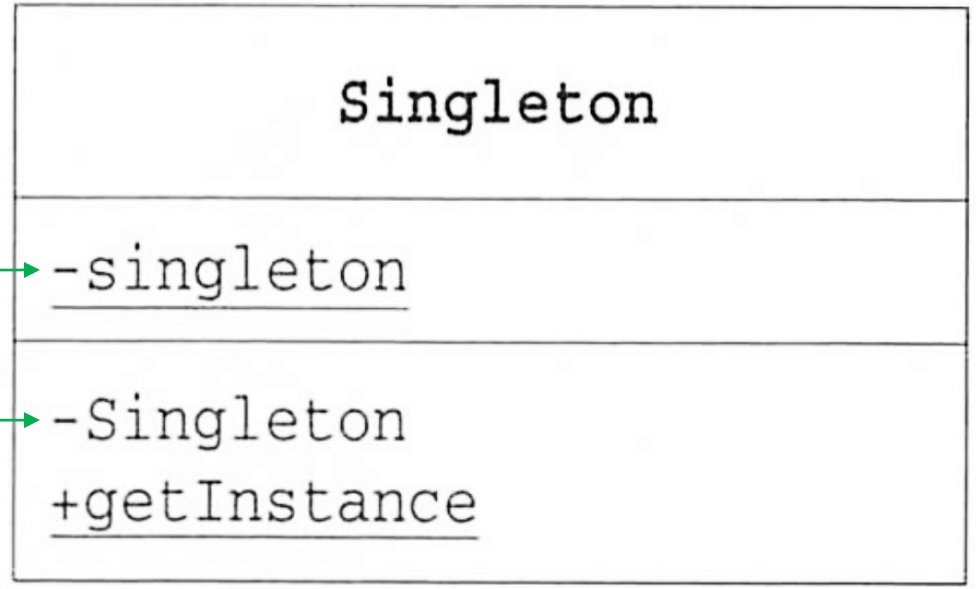
```
No. 1000
No. 1001
No. 1002
No. 1003
No. 1004
No. 1000
No. 1001
No. 1002
```

Singletonパターン (1/2)

- 「ー」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.
```

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

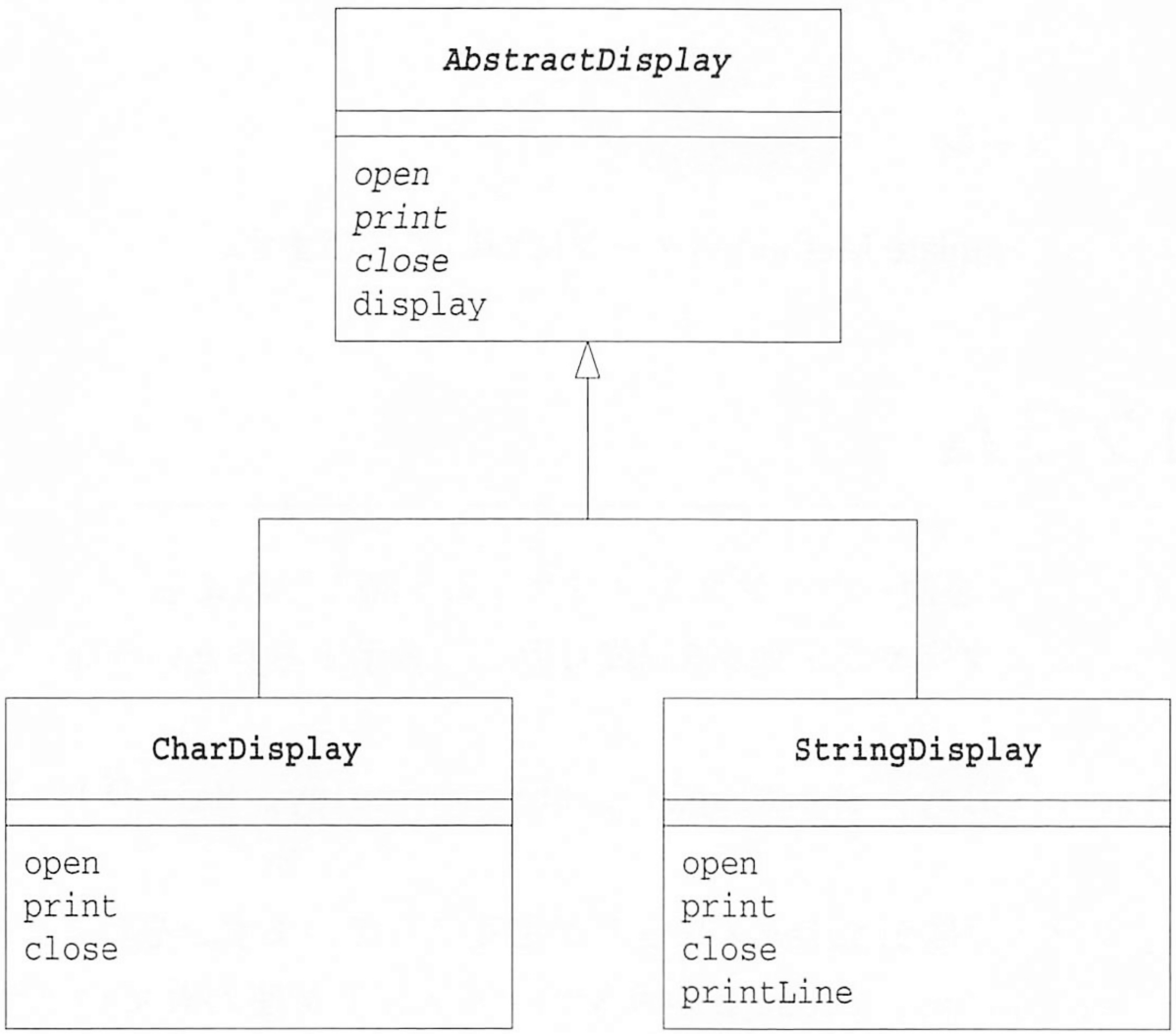
目標の結果（文字の場合）

```
<<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

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

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

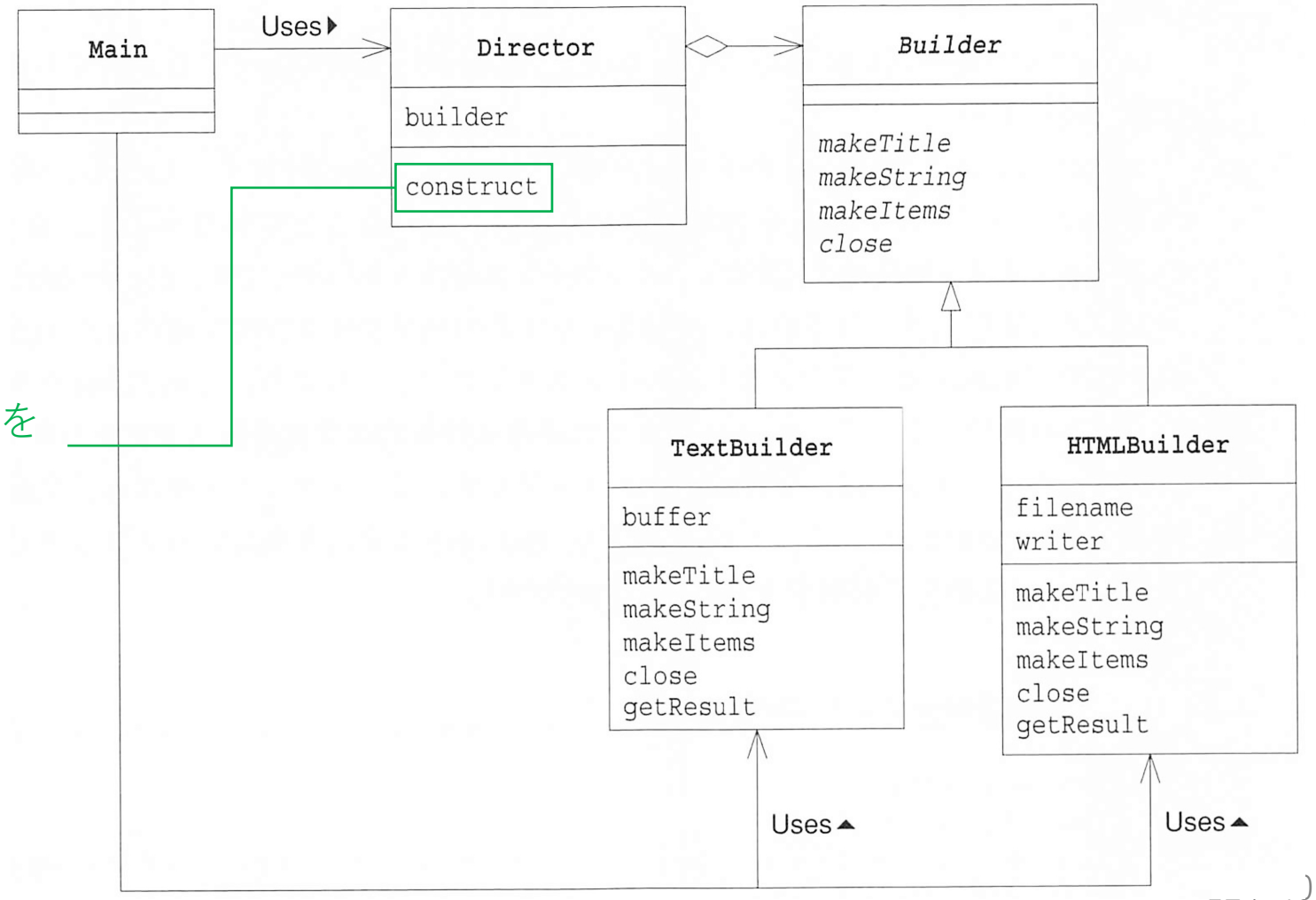
HTML

```
<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

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

Bridgeパターン

Bridgeパターンの必要性

❖ 抽象クラスの継承時に，「機能追加」と「実装」を同時に行うとコードの見通しが悪くなる

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

    public final void display() {
        open();
        print();
        close();
    }
}
```

```java
public class StringDisplay extends Display {
    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("+");
    }

    public void multiDisplay(int times) {
        open();
        for(int i = 0; i < times; i++) {
            print();
        }
        close();
    }
}
```

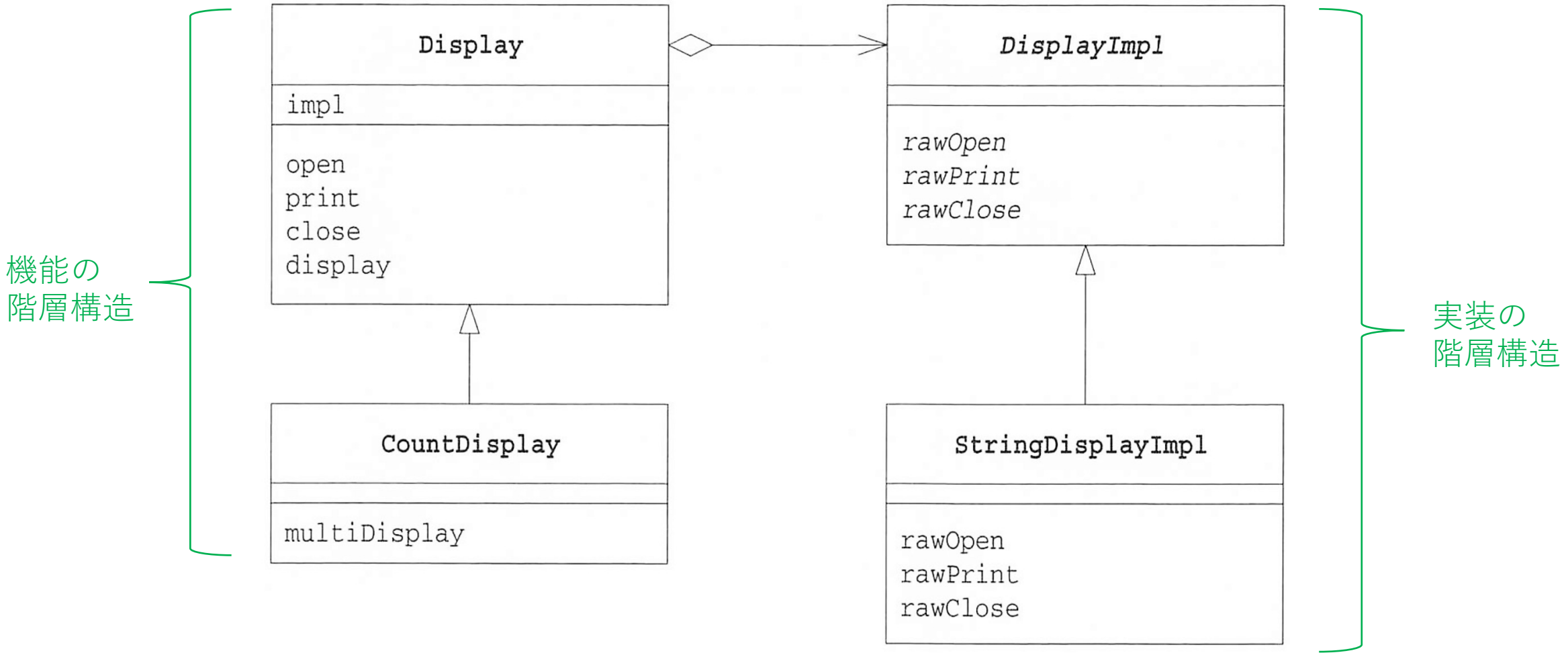
スーパークラスの抽象メソッドを「実装」している

スーパークラスに「機能追加」している

▶ この継承の意味が実装＋機能追加という複雑なものになってしまう

Bridgeパターン (1/5)

橋のどちら側？	名　前	解　説
機能のクラス階層	Display	「表示する」クラス
機能のクラス階層	CountDisplay	「指定回数だけ表示する」という機能を追加したクラス
実装のクラス階層	DisplayImpl	「表示する」クラス
実装のクラス階層	StringDisplayImpl	「文字列を使って表示する」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

Bridgeパターン (2/5)

Display.java

機能の
階層構造

CountDisplay.java

```java
public class Display {
    private DisplayImpl impl;

    public Display(DisplayImpl impl) {
        this.impl = impl;
    }

    public void open() {
        impl.rawOpen();
    }

    public void print() {
        impl.rawPrint();
    }

    public void close() {
        impl.rawClose();
    }

    public final void display() {
        open();
        print();
        close();
    }
}
```

```java
public class CountDisplay extends Display {
    public CountDisplay(DisplayImpl impl) {
        super(impl);
    }

    public void multiDisplay(int times) {
        open();
        for(int i = 0; i < times; i++) {
            print();
        }
        close();
    }
}
```

```java
public abstract class DisplayImpl {
    public abstract void rawOpen();
    public abstract void rawPrint();
    public abstract void rawClose();
}
```

DisplayImpl.java

実装の
階層構造

StringDisplayImpl.java

```java
public class StringDisplayImpl extends DisplayImpl {
    private String str;

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

    public void rawOpen() {
        printLine();
    }

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

    public void rawClose() {
        printLine();
    }

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

Bridgeパターン (3/5)

Display.java

```java
public class Display {
    private DisplayImpl impl;

    public Display(DisplayImpl impl) {
        this.impl = impl;
    }

    public void open() {
        impl.rawOpen();
    }

    public void print() {
        impl.rawPrint();
    }

    public void close() {
        impl.rawClose();
    }

    public final void display() {
        open();
        print();
        close();
    }
}
```

機能の
階層構造

CountDisplay.java

```java
public class CountDisplay extends Display {
    public CountDisplay(DisplayImpl impl) {
        super(impl);
    }

    public void multiDisplay(int times) {
        open();
        for(int i = 0; i < times; i++) {
            print();
        }
        close();
    }
}
```

スーパークラスに「機能追加」しているので
「機能の階層構造」である。

スーパークラスの抽象メソッド（そもそも無いが）を
具体的に実装するという「実装の階層構造」ではない。

Bridgeパターン (4/5)

スーパークラスの抽象メソッドを
具体的に実装するという「実装の階層構造」である。

スーパークラスに新機能を追加するという
「機能の階層構造」ではない。

```java
public abstract class DisplayImpl {
    public abstract void rawOpen();
    public abstract void rawPrint();
    public abstract void rawClose();
}
```

DisplayImpl.java


```java
public class StringDisplayImpl extends DisplayImpl {
    private String str;

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

    public void rawOpen() {
        printLine();
    }

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

    public void rawClose() {
        printLine();
    }

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

StringDisplayImpl.java

Bridgeパターン (5/5)

Main.java

```
public class Main {
    public static void main(String[] args) {
        Display d1 = new Display(new StringDisplayImpl("Hello"));
        d1.display();

        CountDisplay d2 = new CountDisplay(new StringDisplayImpl("Hello, world."));
        d2.display();
        d2.multiDisplay(5);
    }
}
```

実行結果

```
+-----+
|Hello|
+-----+
+-------------+
|Hello, world.|
+-------------+
+-------------+
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
+-------------+
```

d1.display()による出力

d2.display()による出力

d2.multiDisplay(5)による出力

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/04
  - 以降，SOMEWHERE/2021_ap/04をWORK_DIRとする

- IR04-1・IR04-2の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 04: Bridge & Chain of Responsibility > bridge.zip をWORK_DIRにダウンロード
  - unzip bridge.zip
  - cd bridge

IR04-1

❖bridge内のプログラムに，ランダム回数だけ文字列を表示するという新機能を追加せよ（ヒント：機能の階層側に対する修正である）。

実行結果

```
+-----+
|Hello|
+-----+
+-------------+
|Hello, world.|
+-------------+
+-------------+
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
+-------------+
+-------------+
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
+-------------+
```

d1.display()による出力

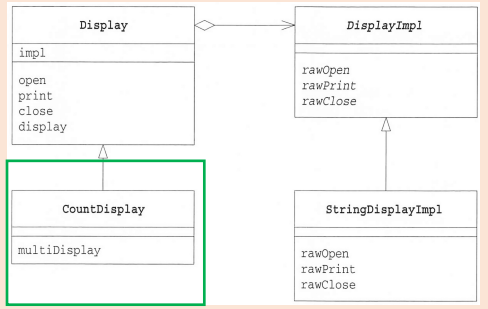
d2.display()による出力

d2.multiDisplay(5)による出力

新機能による出力
（文字列の表示回数はランダム）

■乱数作成方法
import java.util.Random; をした上で
Random rand = new Random();
rand.nextInt(10);
とすると0以上10未満の整数乱数が生成できる。

IR04-1 解答例（下記以外のクラスは変更不要）



CountDisplay.java

```java
import java.util.Random;

public class CountDisplay extends Display {
    private Random rand;

    public CountDisplay(DisplayImpl impl) {
        super(impl);
        rand = new Random();
    }

    public void multiDisplay(int times) {
        open();
        for(int i = 0; i < times; i++) {
            print();
        }
        close();
    }

    public void randomDisplay(int times) {
        open();
        for(int i = 0; i < rand.nextInt(times) + 1; i++) {
            print();
        }
        close();
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Display d1 = new Display(new StringDisplayImpl("Hello"));
        d1.display();

        CountDisplay d2 = new CountDisplay(
            new StringDisplayImpl("Hello, world."));
        d2.display();
        d2.multiDisplay(5);
        d2.randomDisplay(20);
    }
}
```

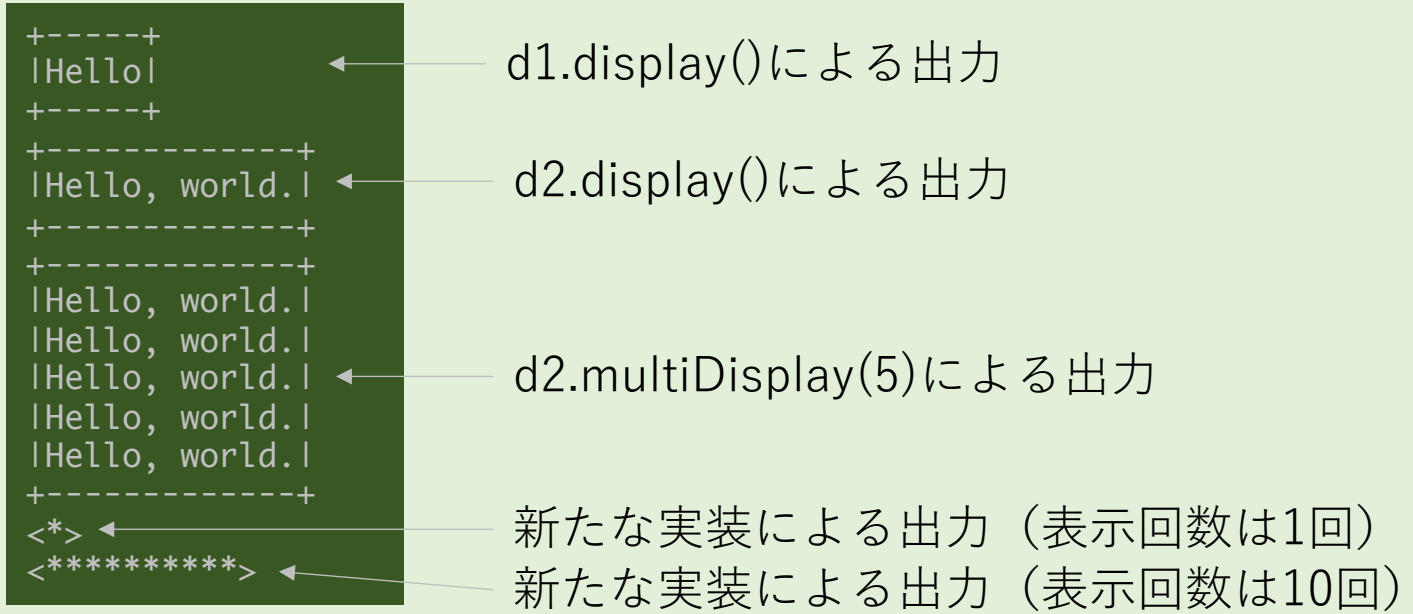
CoundDisplayクラスに追加された新メソッドを呼び出す

新機能追加なので
機能の階層の方に新メソッドを追加

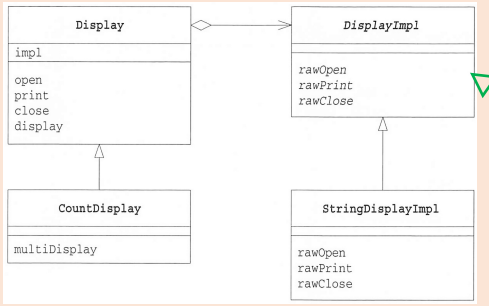
IR04-2

- bridge内のプログラムに，文字を1回／指定回数表示するという新たな実装を追加せよ（ヒント：実装の階層側に対する修正である）。

実行結果



IR04-2 解答例（下記以外のクラスは変更不要）



CharDisplayImpl.java（新クラス）

```java
public class CharDisplayImpl extends DisplayImpl {
    private char c;

    public CharDisplayImpl(char c) {
        this.c = c;
    }

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

    public void rawPrint() {
        System.out.print(c);
    }

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

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Display d1 = new Display(new StringDisplayImpl("Hello"));
        d1.display();

        CountDisplay d2 = new CountDisplay(
            new StringDisplayImpl("Hello, world."));
        d2.display();
        d2.multiDisplay(5);

        CountDisplay d3 = new CountDisplay(new CharDisplayImpl('*'));
        d3.display();
        d3.multiDisplay(10);
    }
}
```

新たなクラスのインスタンスでdisplay(), multiDisplay()を実行

新たな実装の追加なので
実装の階層の方に新クラスを追加

Bridgeパターンのまとめと利用シーン

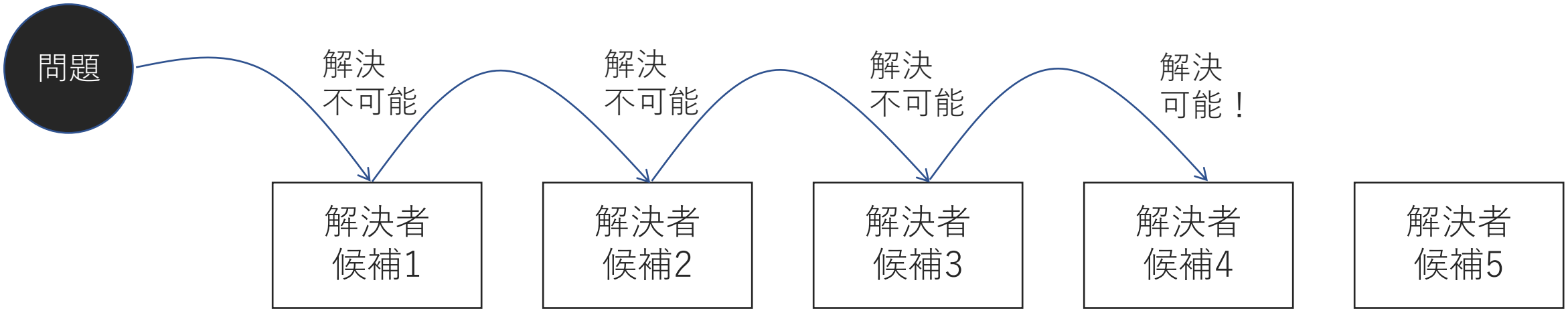
- 「機能の階層」と「実装の階層」を分けるための考え方

- 利用シーン
  - 機能，実装とも今後拡張が予想される場合
    → 機能の拡張と，実装の拡張が，混在せずに済む。
    → 例えば，各OS共通の機能は，「機能の階層」を拡張する。
    → 新OSに対応する場合は，「実装の階層」を拡張する。

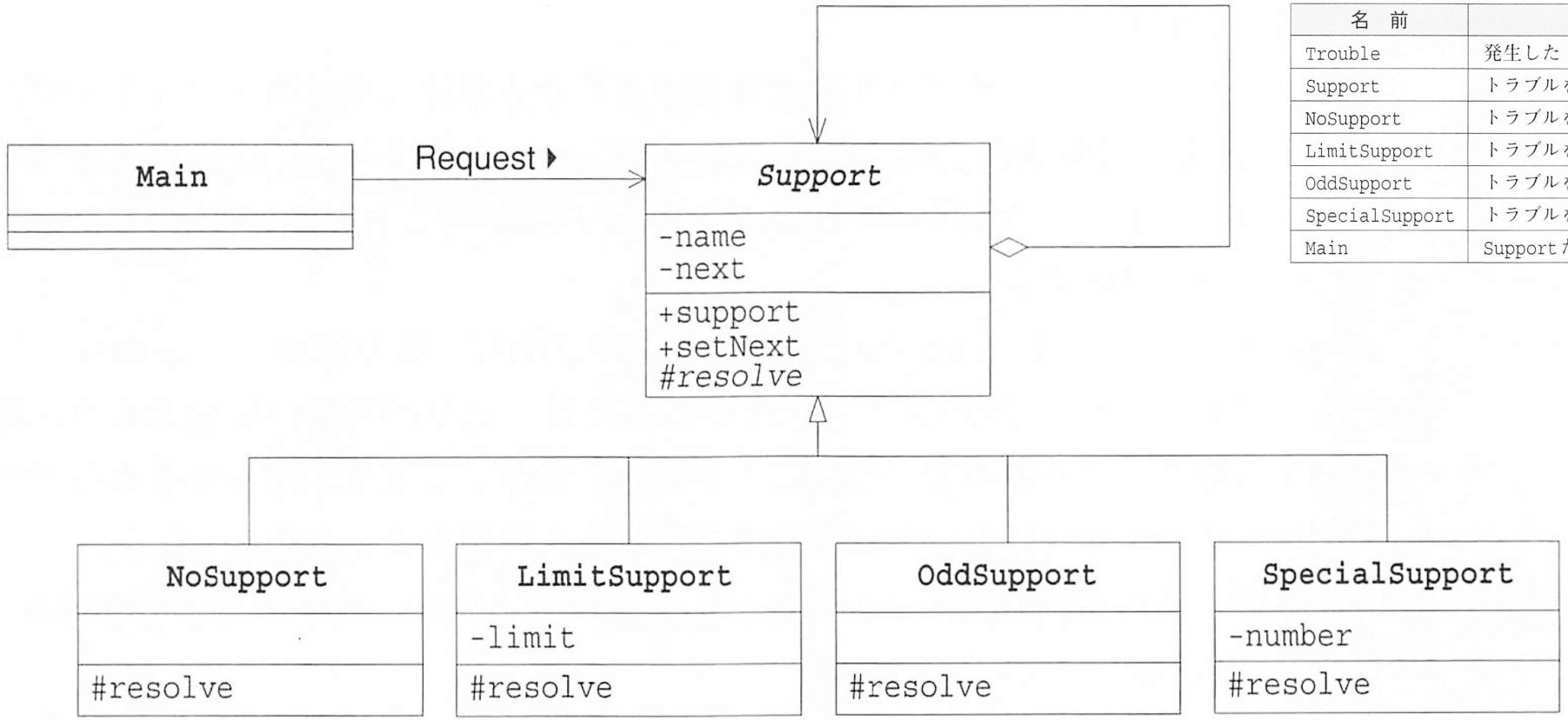
Chain of Responsibilityパターン

Chain of Responsibilityのイメージ

❖責任の連鎖（たらい回し）



Chain of Responsibilityパターン (1/5)



名　前	解　説
Trouble	発生したトラブルを表すクラス。トラブル番号（number）を持つ
Support	トラブルを解決する抽象クラス
NoSupport	トラブルを解決する具象クラス（常に「処理しない」）
LimitSupport	トラブルを解決する具象クラス（指定した番号未満のトラブルを解決）
OddSupport	トラブルを解決する具象クラス（奇数番号のトラブルを解決）
SpecialSupport	トラブルを解決する具象クラス（特定番号のトラブルを解決）
Main	Supportたちの連鎖を作り、トラブルを起こす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

Chain of Responsibilityパターン (2/5)

Support.java

```java
public abstract class Support {
    private String name;
    private Support next;

    public Support(String name) {
        this.name = name;
    }

    public Support setNext(Support next) {
        this.next = next;
        return next;
    }

    public final void support(Trouble trouble) {
        if(resolve(trouble)) {
            done(trouble);
        } else if(next != null) {
            next.support(trouble);
        } else {
            fail(trouble);
        }
    }

    public String toString() {
        return "[" + name + "]";
    }

    protected abstract boolean resolve(Trouble trouble);

    protected void done(Trouble trouble) {
        System.out.println(
            trouble + " is resolved by " + toString() + ".");
    }

    protected void fail(Trouble trouble) {
        System.out.println(trouble + " cannot be resolved.");
    }
}
```

OddSupport.java

```java
public class OddSupport extends Support {
    public OddSupport(String name) {
        super(name);
    }

    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() % 2 == 1) {
            return true;
        } else {
            return false;
        }
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Support alice = new SpecialSupport("Alice", 4);
        Support bobby = new SpecialSupport("Bobby", 8);
        Support chris = new OddSupport("Chris");

        alice.setNext(bobby).setNext(chris);

        for(int i = 1; i <= 20; i++) {
            alice.support(new Trouble(i));
        }
    }
}
```

SpecialSupport.java

```java
public class SpecialSupport extends Support {
    private int number;

    public SpecialSupport(String name, int number) {
        super(name);
        this.number = number;
    }

    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() == number) {
            return true;
        } else {
            return false;
        }
    }
}
```

Trouble.java

```java
public class Trouble {
    private int number;

    public Trouble(int number) {
        this.number = number;
    }

    public int getNumber() {
        return number;
    }

    public String toString() {
        return "Trouble " + number + "";
    }
}
```

Chain of Responsibilityパターン (3/5)

Support.java

```java
public abstract class Support {
    private String name;
    private Support next;

    public Support(String name) {
        this.name = name;
    }

    public Support setNext(Support next) {
        this.next = next;
        return next;
    }

    public final void support(Trouble trouble) {
        if(resolve(trouble)) {
            done(trouble);
        } else if(next != null) {
            next.support(trouble);
        } else {
            fail(trouble);
        }
    }

    public String toString() {
        return "[" + name + "]";
    }

    protected abstract boolean resolve(Trouble trouble);

    protected void done(Trouble trouble) {
        System.out.println(
            trouble + " is resolved by " + toString() + ".");
    }

    protected void fail(Trouble trouble) {
        System.out.println(trouble + " cannot be resolved.");
    }
}
```

自分の次の解決者候補を取得

もし解決できたら（resolve()がtrueを返したら）done()を実行,
次の解決者候補がいるなら（next != null）次の人に依頼,
解決できず次の人もいないならfail()を実行

Chain of Responsibilityパターン (4/5)

OddSupport.java

```java
public class OddSupport extends Support {
    public OddSupport(String name) {
        super(name);
    }

    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() % 2 == 1) {
            return true;
        } else {
            return false;
        }
    }
}
```

トラブル番号が奇数なら
解決する（trueを返す）

SpecialSupport.java

```java
public class SpecialSupport extends Support {
    private int number;

    public SpecialSupport(String name, int number) {
        super(name);
        this.number = number;
    }

    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() == number) {
            return true;
        } else {
            return false;
        }
    }
}
```

トラブル番号が規定の数なら
解決する（trueを返す）

Trouble.java

```java
public class Trouble {
    private int number;

    public Trouble(int number) {
        this.number = number;
    }

    public int getNumber() {
        return number;
    }

    public String toString() {
        return "Trouble " + number + "";
    }
}
```

トラブルには番号（number）がある

Chain of Responsibilityパターン (5/5)

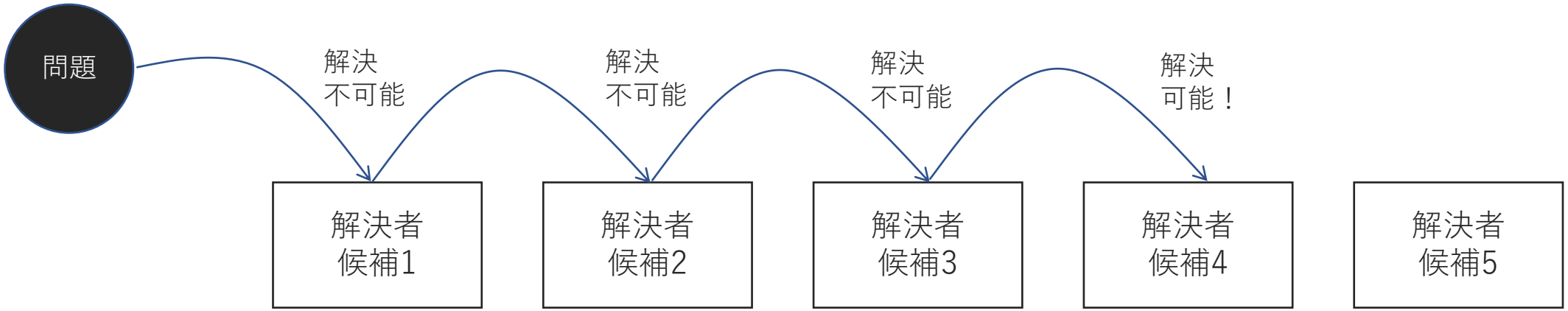
Main.java

```java
public class Main {
    public static void main(String[] args) {
        Support alice = new SpecialSupport("Alice", 4);
        Support bobby = new SpecialSupport("Bobby", 8);
        Support chris = new OddSupport("Chris");

        alice.setNext(bobby).setNext(chris);

        for(int i = 1; i <= 20; i++) {
            alice.support(new Trouble(i));
        }
    }
}
```

Aliceの次はBobby，Bobbyの次はChrisを設定

実行結果

```
Trouble 1 is resolved by [Chris].
Trouble 2 cannot be resolved.
Trouble 3 is resolved by [Chris].
Trouble 4 is resolved by [Alice].
Trouble 5 is resolved by [Chris].
Trouble 6 cannot be resolved.
Trouble 7 is resolved by [Chris].
Trouble 8 is resolved by [Bobby].
Trouble 9 is resolved by [Chris].
Trouble 10 cannot be resolved.
Trouble 11 is resolved by [Chris].
Trouble 12 cannot be resolved.
Trouble 13 is resolved by [Chris].
Trouble 14 cannot be resolved.
Trouble 15 is resolved by [Chris].
Trouble 16 cannot be resolved.
Trouble 17 is resolved by [Chris].
Trouble 18 cannot be resolved.
Trouble 19 is resolved by [Chris].
Trouble 20 cannot be resolved.
```



作業準備

- IR04-3の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 04: Bridge & Chain of Responsibility > cor.zip をWORK_DIRにダウンロード
  - unzip cor.zip
  - cd cor

IR04-3

- cor内のプログラムに，指定範囲内の番号のトラブルを解決できる役割の新クラスを作成せよ。Daisyという名前で，11以上15以下の番号のトラブルを解決できる新クラスのインスタンスを作成し，Alice，Bobby，Daisy，Chrisの順番でトラブル対応するプログラムを完成させよ。

目標の実行結果

```
Trouble 1 is resolved by [Chris].
Trouble 2 cannot be resolved.
Trouble 3 is resolved by [Chris].
Trouble 4 is resolved by [Alice].
Trouble 5 is resolved by [Chris].
Trouble 6 cannot be resolved.
Trouble 7 is resolved by [Chris].
Trouble 8 is resolved by [Bobby].
Trouble 9 is resolved by [Chris].
Trouble 10 cannot be resolved.
Trouble 11 is resolved by [Daisy].
Trouble 12 is resolved by [Daisy].
Trouble 13 is resolved by [Daisy].
Trouble 14 is resolved by [Daisy].
Trouble 15 is resolved by [Daisy].
Trouble 16 cannot be resolved.
Trouble 17 is resolved by [Chris].
Trouble 18 cannot be resolved.
Trouble 19 is resolved by [Chris].
Trouble 20 cannot be resolved.
```

IR04-3 解答例（下記以外のクラスは変更不要）

RangeSupport.java

```java
public class RangeSupport extends Support {
    private int start;
    private int end;

    public RangeSupport(String name, int start, int end) {
        super(name);
        this.start = start;
        this.end = end;
    }

    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() >= start
            && trouble.getNumber() <= end) {
            return true;
        } else {
            return false;
        }
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Support alice = new SpecialSupport("Alice", 4);
        Support bobby = new SpecialSupport("Bobby", 8);
        Support chris = new OddSupport("Chris");
        Support daisy = new RangeSupport("Daisy", 11, 15);

        alice.setNext(bobby).setNext(daisy).setNext(chris);

        for(int i = 1; i <= 20; i++) {
            alice.support(new Trouble(i));
        }
    }
}
```

Chain of Responsibilityパターンのまとめと利用シーン

- 「要求を出す人」と「要求を処理する人」を緩やかにつなぐための考え方
  - このパターンを使わないと，「要求を出す人」が，
    各要求は誰が処理すべきか，中央集権的に管理しないといけない。

- 利用シーン
  - 要求と処理の関係が流動的（今後，変わるかもしれない）場合
    → 新たな処理が発生したら，連鎖中の適切な位置に挿入するだけで良い

- 注意
  - このパターンは，「たらい回し」方式のため処理速度が遅い可能性がある。
    要求・処理の関係が固定的であるなら，単にifブロックで処理を決定した方が処理は速い。

本日のまとめ

- 講義内容
  - Bridgeパターン
  - Chain of Responsibility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。