発展プログラミング

第5回：デザインパターンの実践

宮田章裕 <miyata.akihiro@nihon-u.ac.jp>

前回講義の復習

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();
    }
}
```

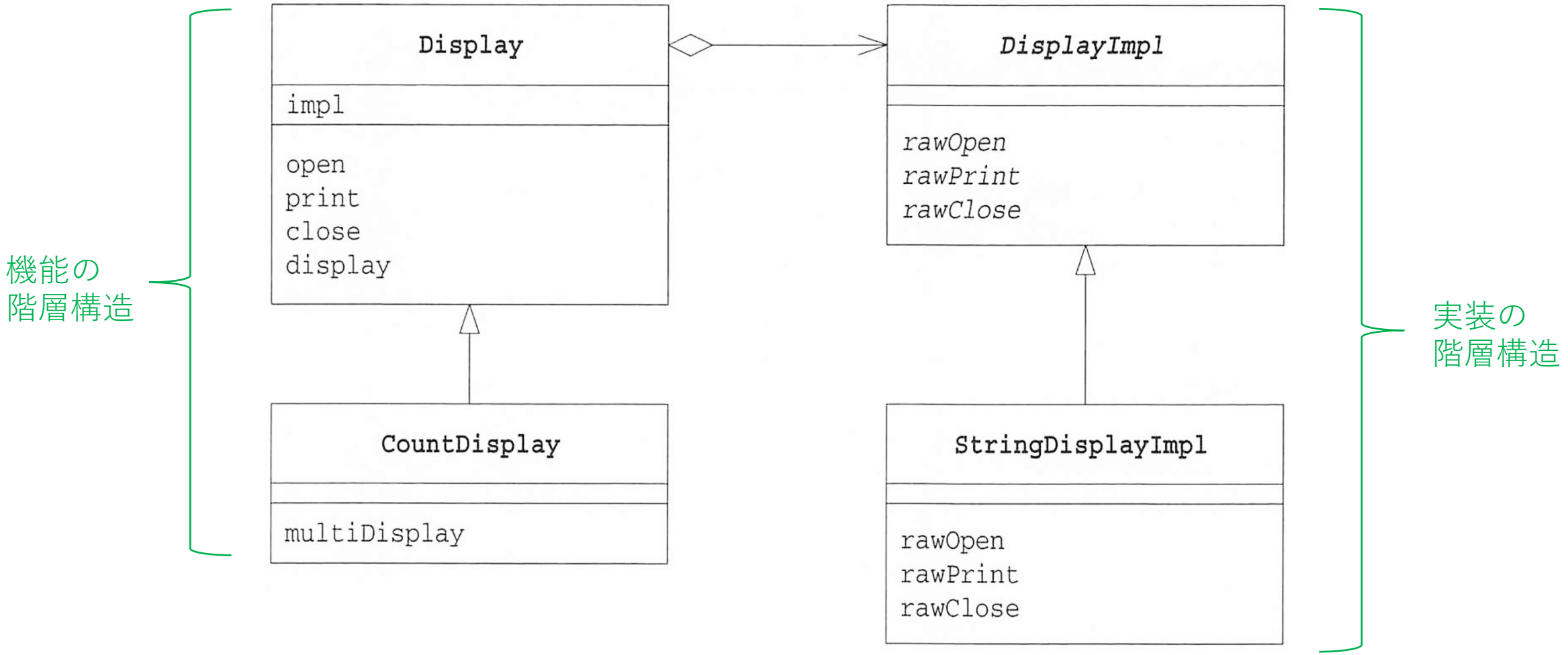
スーパークラスの抽象メソッドを「実装」している

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

```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();
    }
}
```

```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("+");
    }
}
```

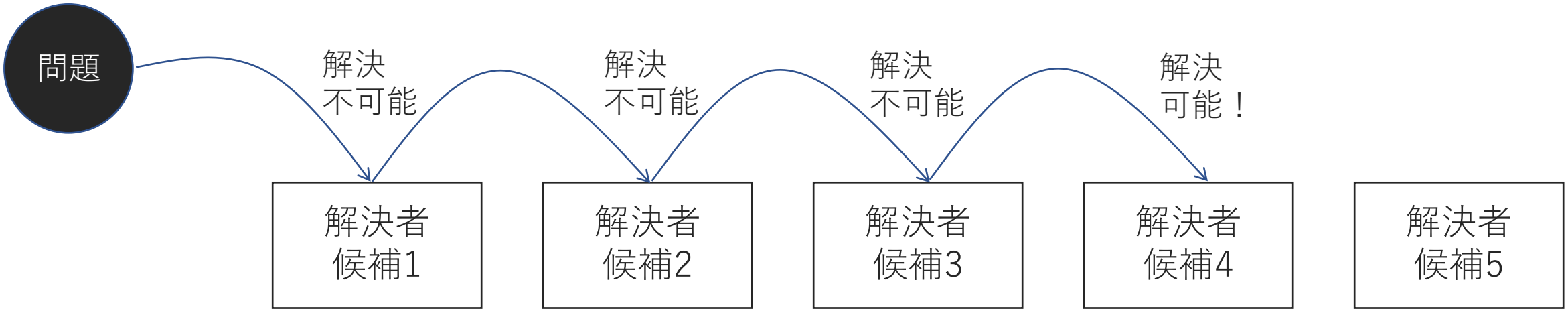
Bridgeパターンのまとめと利用シーン

- 「機能の階層」と「実装の階層」を分けるための考え方

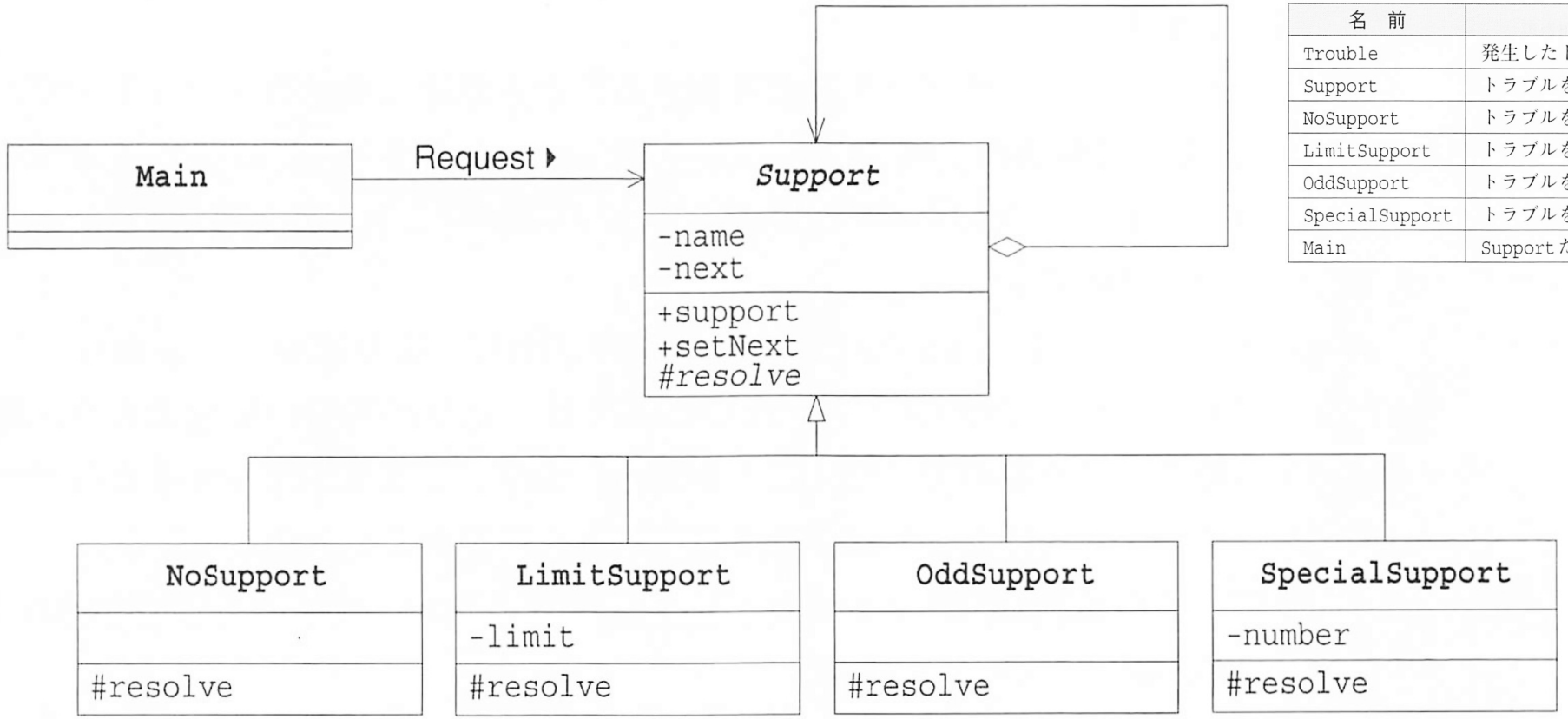
- 利用シーン
  - 機能，実装とも今後拡張が予想される場合
    → 機能の拡張と，実装の拡張が，混在せずに済む。
    → 例えば，各OS共通の機能は，「機能の階層」を拡張する。
    → 新OSに対応する場合は，「実装の階層」を拡張する。

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)

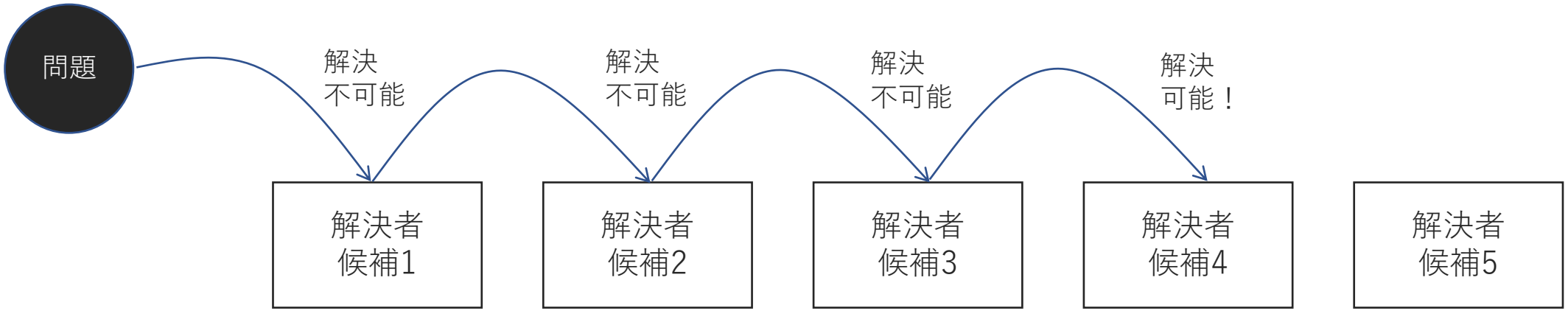
Main.java

```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.
```



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。

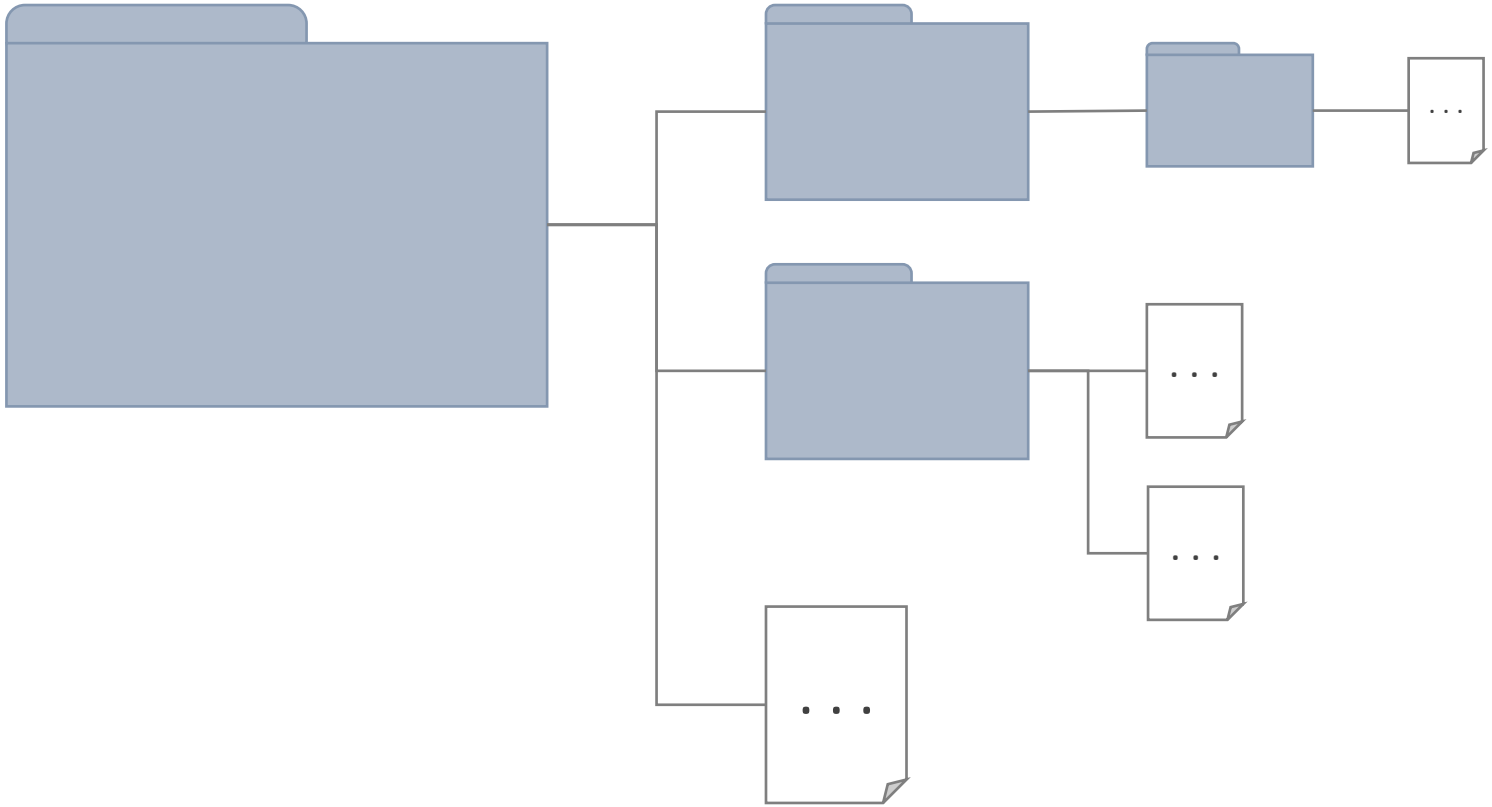
- 利用シーン
  - 要求と処理の関係が流動的（今後，変わるかもしれない）場合
    → 新たな処理が発生したら，連鎖中の適切な位置に挿入するだけで良い

- 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。

Compositeパターン

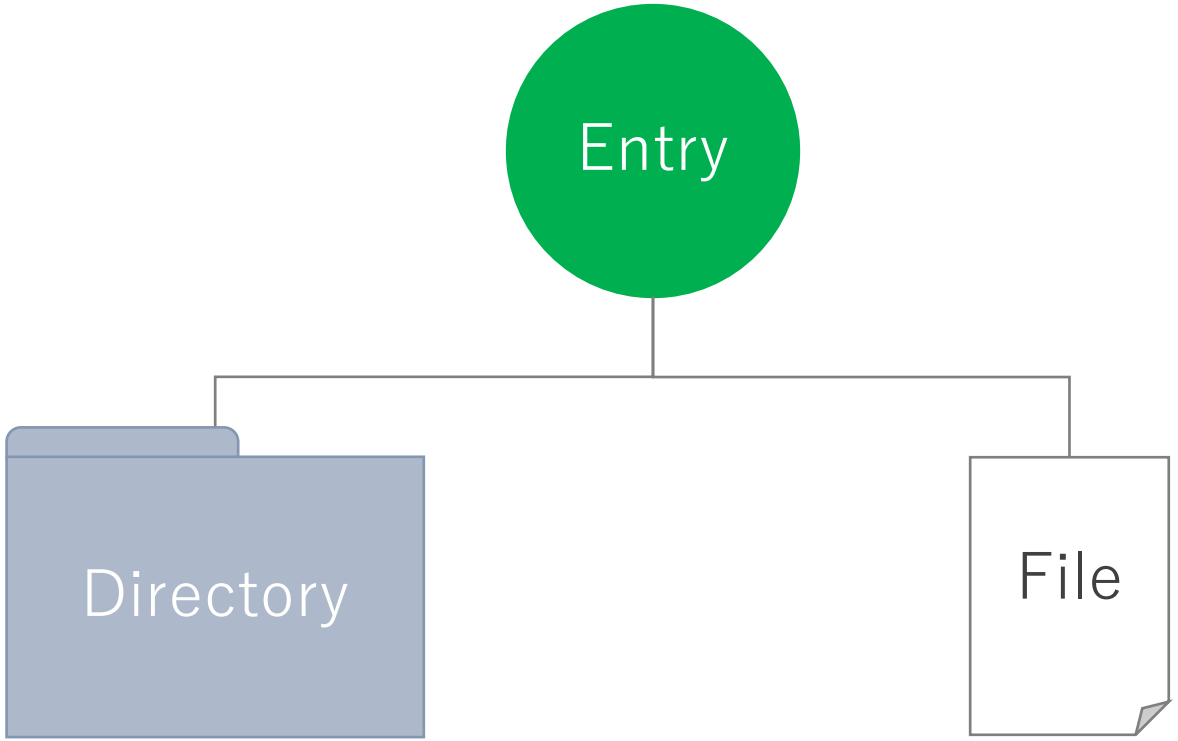
Compositeパターンの必要性

- 容器と中身を同一視したい（同じように扱いたい）ことがある
  - 例えば，ディレクトリ（容器）とファイル（中身）に対して，
    同じ操作（名称・サイズの取得）をしたいことがある



Compositeパターンのアプローチ

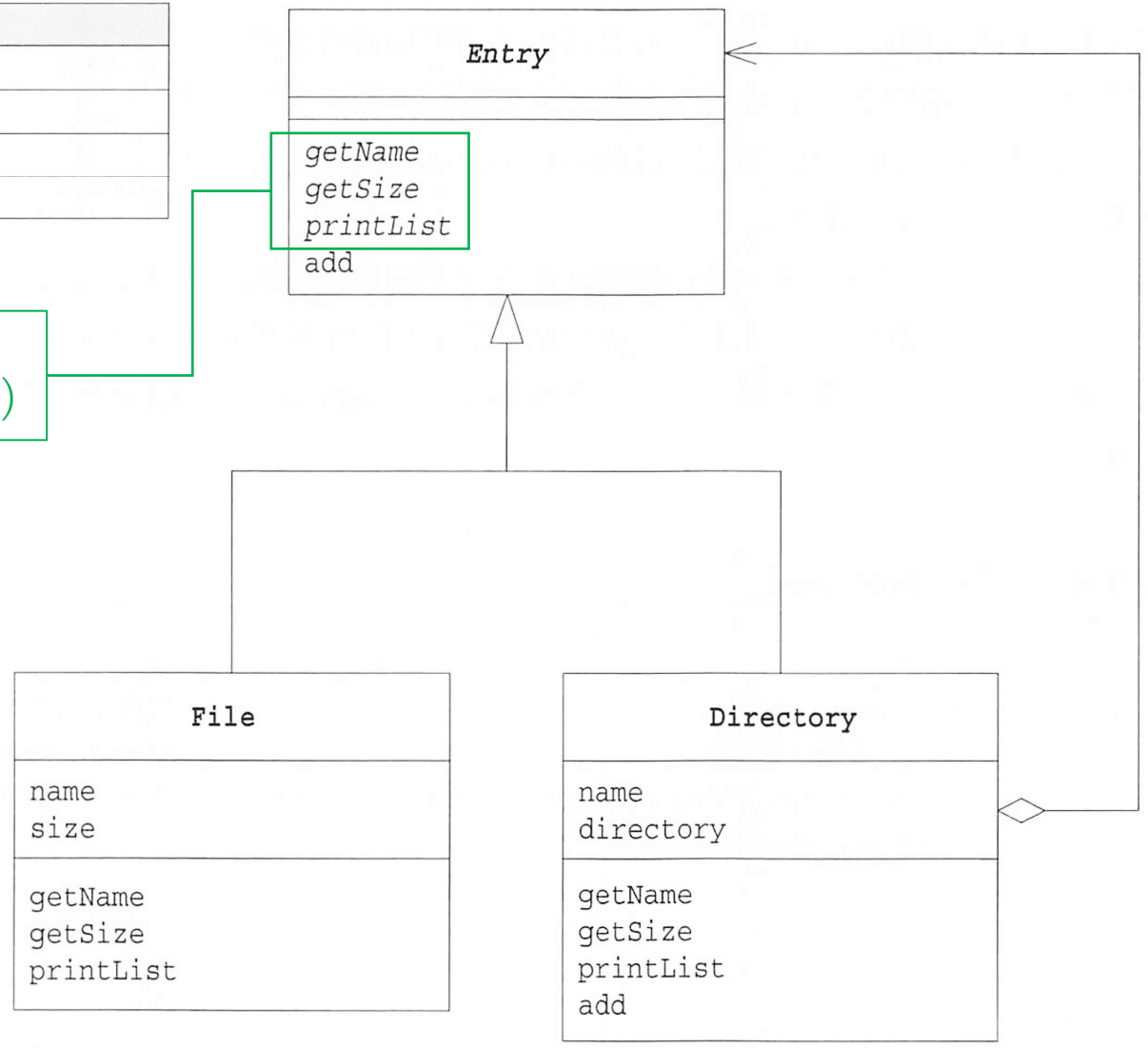
- 容器と中身を同一視するためのスーパークラスを作成する
  - 容器・中身に共通する機能はスーパークラスで定義する
  - ただし，その機能の具体的な挙動は容器・中身で個別に実装する



Compositeパターン (1/6)

名　前	解　説
Entry	FileとDirectoryを同一視する抽象クラス
File	ファイルを表すクラス
Directory	ディレクトリを表すクラス
Main	動作テスト用のクラス

FileとDirectoryの共通機能
(FileとDirectoryの両方で同じように利用できる)



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

Compositeパターン (2/6)

Entry.java

```java
public abstract class Entry {
    public abstract String getName();

    public abstract int getSize();

    public void add(Entry entry) {
        System.out.println("Error");
    }

    public void printList() {
        printList("");
    }

    protected abstract void printList(String prefix);

    public String toString() {
        return getName() + " (" + getSize() + ")";
    }
}
```

File.java

```java
public class File extends Entry {
    private String name;
    private int size;

    public File(String name, int size) {
        this.name = name;
        this.size = size;
    }

    public String getName() {
        return name;
    }

    public int getSize() {
        return size;
    }

    protected void printList(String prefix) {
        System.out.println(prefix + "/" + toString());
    }
}
```

Directory.java

```java
import java.util.Iterator;
import java.util.ArrayList;

public class Directory extends Entry {
    private String name;
    private ArrayList<Entry> entries
        = new ArrayList<Entry>();

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

    public String getName() {
        return name;
    }

    public int getSize() {
        int size = 0;
        // Do something.
        return size;
    }

    public void add(Entry entry) {
        entries.add(entry);
    }

    protected void printList(String prefix) {
        System.out.println(prefix + "/" + toString());
        Iterator it = entries.iterator();
        while(it.hasNext()) {
            Entry entry = (Entry)it.next();
            entry.printList(prefix + "/" + name);
        }
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Directory rootDir = new Directory("root");

        Directory homeDir = new Directory("home");
        rootDir.add(homeDir);

        Directory aliceDir = new Directory("alice");
        Directory bobbyDir = new Directory("bobby");
        Directory chrisDir = new Directory("chris");
        homeDir.add(aliceDir);
        homeDir.add(bobbyDir);
        homeDir.add(chrisDir);

        aliceDir.add(new File("Main.java", 1000));
        aliceDir.add(new File("Analyze.java", 3200));
        aliceDir.add(new File("Output.java", 2050));

        chrisDir.add(new File("index.html", 500));
        chrisDir.add(new File("main.js", 2500));

        rootDir.printList();
    }
}
```

Compositeパターン (3/6)

Entry.java

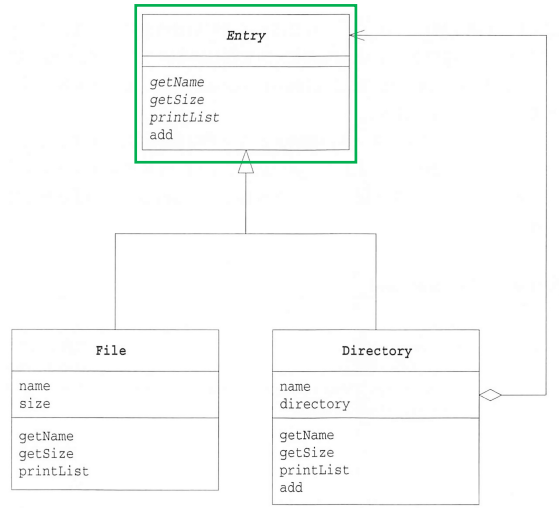
```java
public abstract class Entry {
    public abstract String getName();

    public abstract int getSize();

    public void add(Entry entry) {
        System.out.println("Error");
    }

    public void printList() {
        printList("");
    }

    protected abstract void printList(String prefix);

    public String toString() {
        return getName() + " (" + getSize() + ")";
    }
}
```

サブクラスに実装を義務付ける
(Subclass responsibility)

中身を追加するメソッド。
Fileクラスのために，ひとまずエラーするようにしておく。

引数無しでprintList()が呼ばれたら
空文字を引数にしてprintList(String)を呼ぶ。
(オーバーロード)

サブクラスの挙動を定義する。
(Template Method)



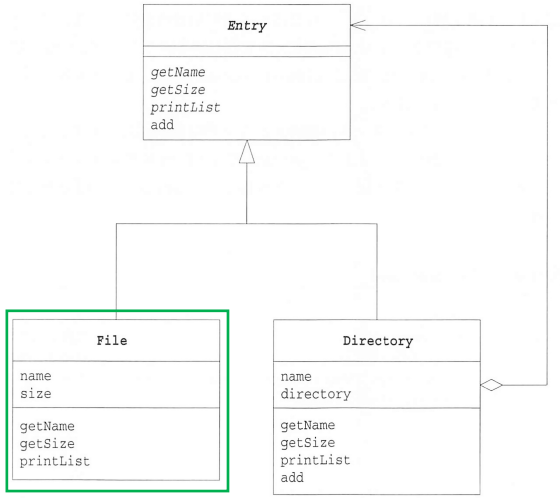
Compositeパターン (4/6)

Entry.java

```java
public abstract class Entry {
    public abstract String getName();

    public abstract int getSize();

    public void add(Entry entry) {
        System.out.println("Error");
    }

    public void printList() {
        printList("");
    }

    protected abstract void printList(String prefix);

    public String toString() {
        return getName() + " (" + getSize() + ")";
    }
}
```

File.java

```java
public class File extends Entry {
    private String name;
    private int size;

    public File(String name, int size) {
        this.name = name;
        this.size = size;
    }

    public String getName() {
        return name;
    }

    public int getSize() {
        return size;
    }

    protected void printList(String prefix) {
        System.out.println(prefix + "/" + toString());
    }
}
```



スーパークラスに実装を義務付けられていた
メソッドを具体的に実装

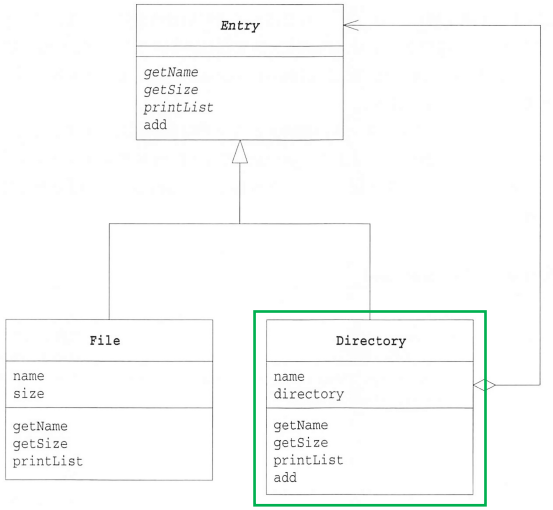
Compositeパターン (5/6)

Entry.java

```java
public abstract class Entry {
    public abstract String getName();

    public abstract int getSize();

    public void add(Entry entry) {
        System.out.println("Error");
    }

    public void printList() {
        printList("");
    }

    protected abstract void printList(String prefix);

    public String toString() {
        return getName() + " (" + getSize() + ")";
    }
}
```

Directory.java

```java
import java.util.Iterator;
import java.util.ArrayList;

public class Directory extends Entry {
    private String name;
    private ArrayList<Entry> entries = new ArrayList<Entry>();

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

    public String getName() {
        return name;
    }

    public int getSize() {
        int size = 0;
        // Do something.
        return size;
    }

    public void add(Entry entry) {
        entries.add(entry);
    }

    protected void printList(String prefix) {
        System.out.println(prefix + "/" + toString());
        Iterator it = entries.iterator();
        while(it.hasNext()) {
            Entry entry = (Entry)it.next();
            entry.printList(prefix + "/" + name);
        }
    }
}
```



スーパークラスに
実装を義務付けられていた
メソッドを具体的に実装。

スーパークラスのadd()を
上書きして，自分の中に
Entry(File or Directory)を
追加できるようにする。
(オーバーライド)

対象がFileかDirectoryか意識せず
Entryとして同一視して
printList()を呼び出している。

Compositeパターン (6/6)

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Directory rootDir = new Directory("root");

        Directory homeDir = new Directory("home");
        rootDir.add(homeDir);

        Directory aliceDir = new Directory("alice");
        Directory bobbyDir = new Directory("bobby");
        Directory chrisDir = new Directory("chris");
        homeDir.add(aliceDir);
        homeDir.add(bobbyDir);
        homeDir.add(chrisDir);

        aliceDir.add(new File("Main.java", 1000));
        aliceDir.add(new File("Analyze.java", 3200));
        aliceDir.add(new File("Output.java", 2050));

        chrisDir.add(new File("index.html", 500));
        chrisDir.add(new File("main.js", 2500));

        rootDir.printList();
    }
}
```

実行結果 (Directoryのサイズは未実装)

```
/root (0)
/root/home (0)
/root/home/alice (0)
/root/home/alice/Main.java (1000)
/root/home/alice/Analyze.java (3200)
/root/home/alice/Output.java (2050)
/root/home/bobby (0)
/root/home/chris (0)
/root/home/chris/index.html (500)
/root/home/chris/main.js (2500)
```

DirectorのprintList()が再帰的に
中身のEntry（File or Directory）に対して
printList()を実行する

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/05
  - 以降，SOMEWHERE/2021_ap/05をWORK_DIRとする

- IR05-1の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 05: Composite & Decorator > composite.zip をWORK_DIRにダウンロード
  - unzip composite.zip
  - cd composite

IR05-1

- Directory.javaのgetSize()を完全に実装し，目標の実行結果のようにDirectoryのサイズとして，当該Directory内の全ファイルのサイズの和を表示できるようにせよ。

Directory.java

```java
import java.util.Iterator;
import java.util.ArrayList;

public class Directory extends Entry {
    private String name;
    private ArrayList<Entry> entries = new ArrayList<Entry>();

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

    public String getName() {
        return name;
    }

    public int getSize() {
        int size = 0;
        // Do something.
        return size;
    }

    public void add(Entry entry) {
        entries.add(entry);
    }

    protected void printList(String prefix) {
        System.out.println(prefix + "/" + toString());
        Iterator it = entries.iterator();
        while(it.hasNext()) {
            Entry entry = (Entry)it.next();
            entry.printList(prefix + "/" + name);
        }
    }
}
```

目標の実行結果

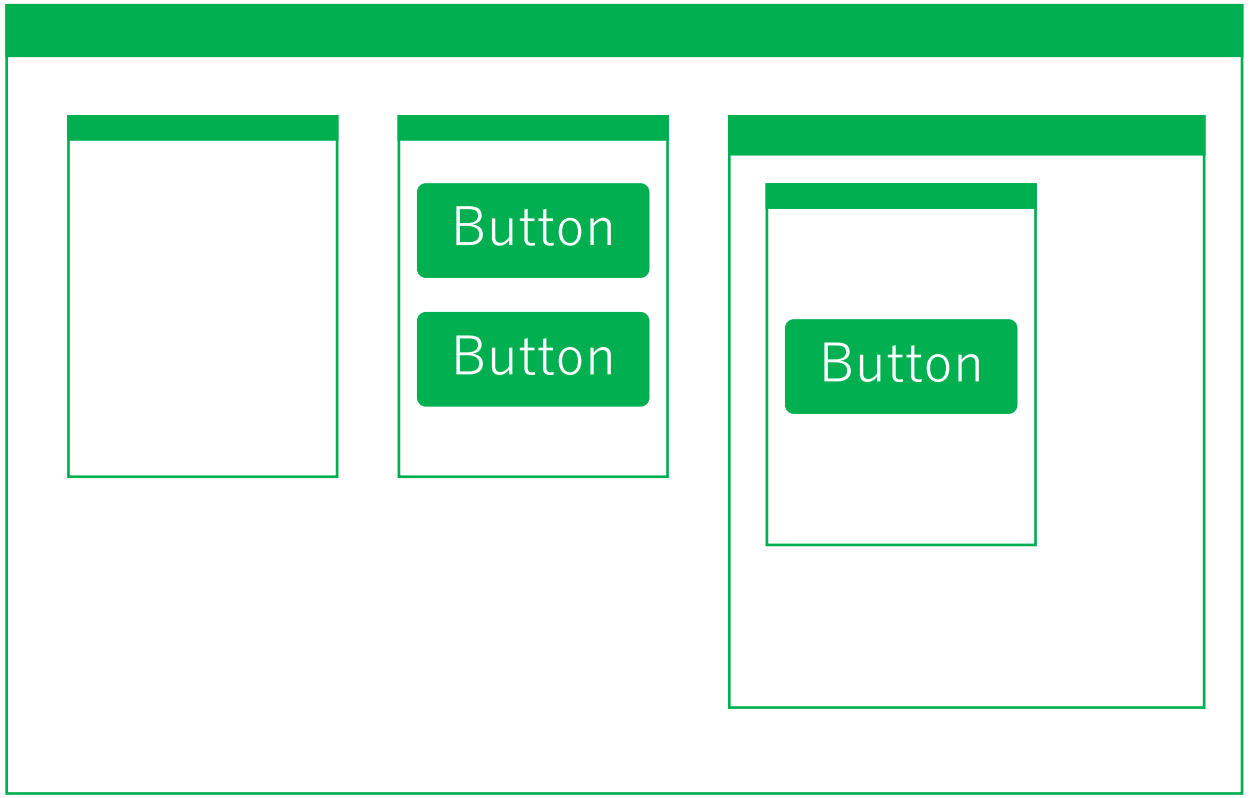
```
/root (9250)
/root/home (9250)
/root/home/alice (6250)
/root/home/alice/Main.java (1000)
/root/home/alice/Analyze.java (3200)
/root/home/alice/Output.java (2050)
/root/home/bobby (0)
/root/home/chris (3000)
/root/home/chris/index.html (500)
/root/home/chris/main.js (2500)
```

ヒント：ここと同じように
対象がFileかDirectoryか意識せず
EntryのgetSize()を呼び出して合計値を求める。

Compositeパターンのまとめと利用シーン

- 容器と中身を同一視し，再帰的な構造を作るための考え方
  - 容器と中身を区別せず，共通する機能を持たせることができる。
  - 必要に応じて，容器と中身に固有な機能を持たせることもできる。

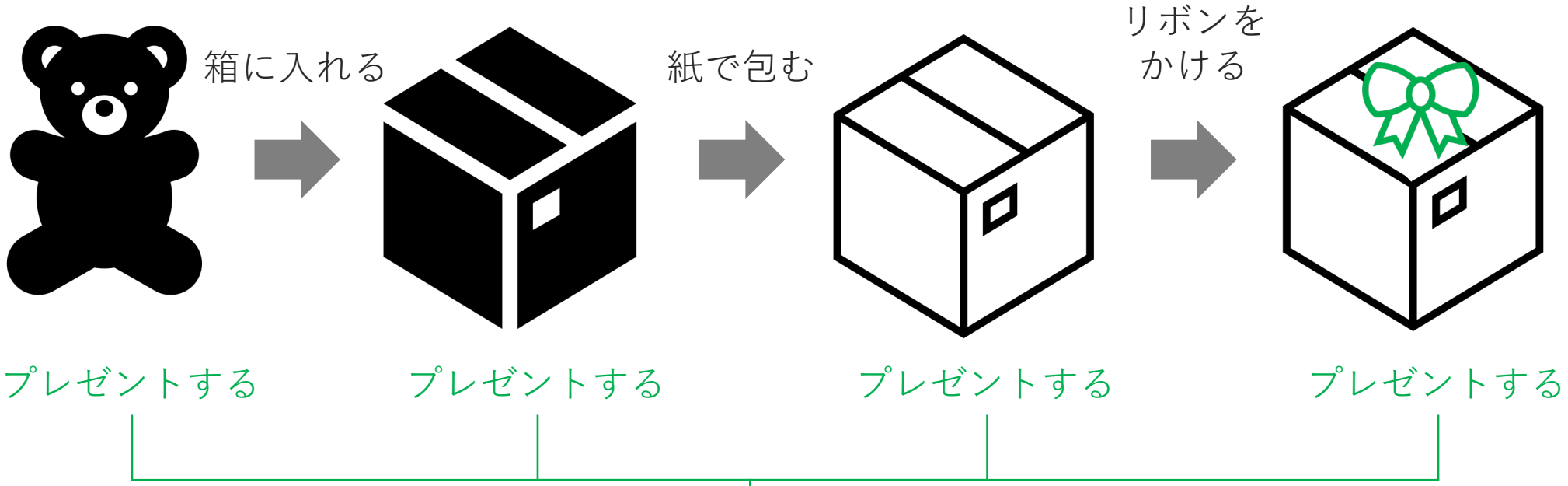
- 利用シーン
  - 再帰的な構造が生じる場合
  - 例1： ファイルシステム
  - 例2： ウィンドウシステム



Decoratorパターン

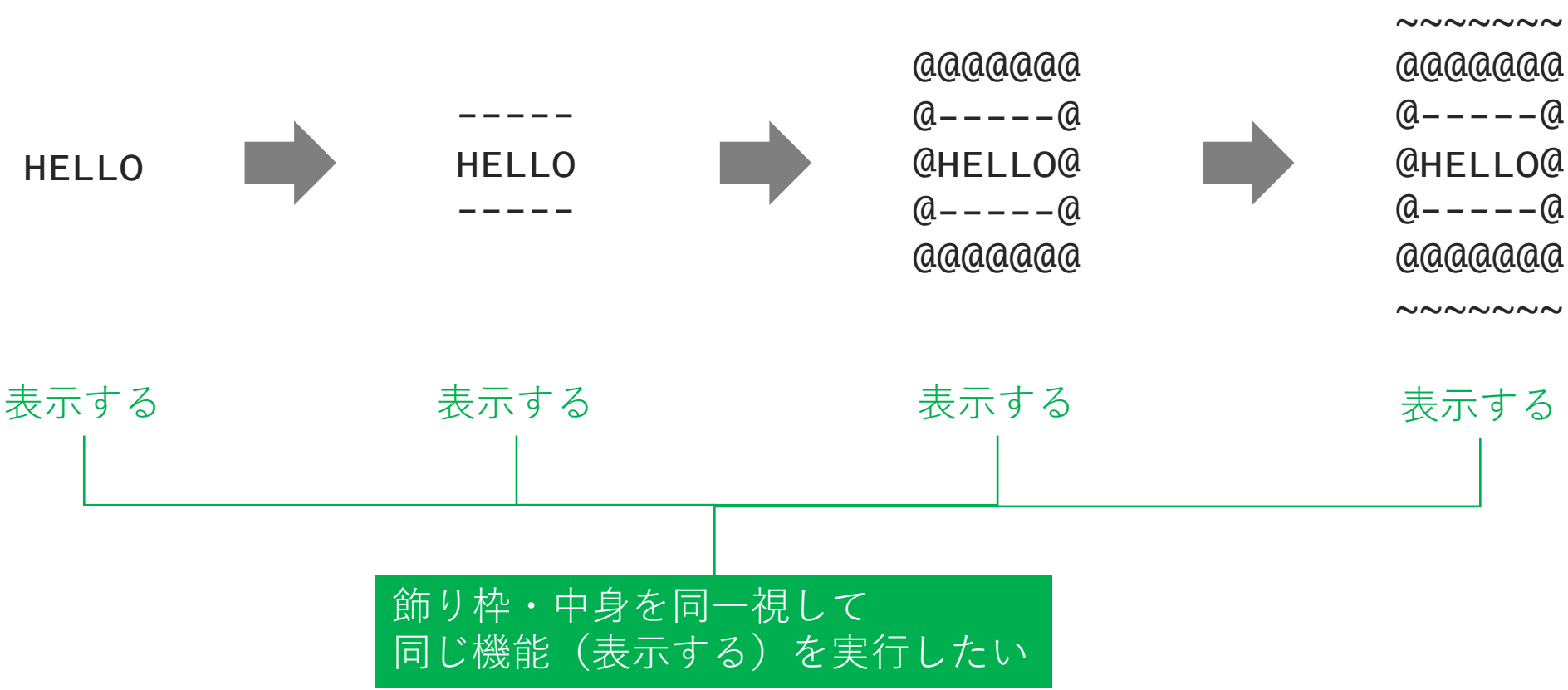
Decoratorパターンの必要性 (1/2)

- ❖飾り枠と中身を同一視したい（同じように扱いたい）ことがある
- ❖現実空間では，商品を何重にも包装するシーンが該当する



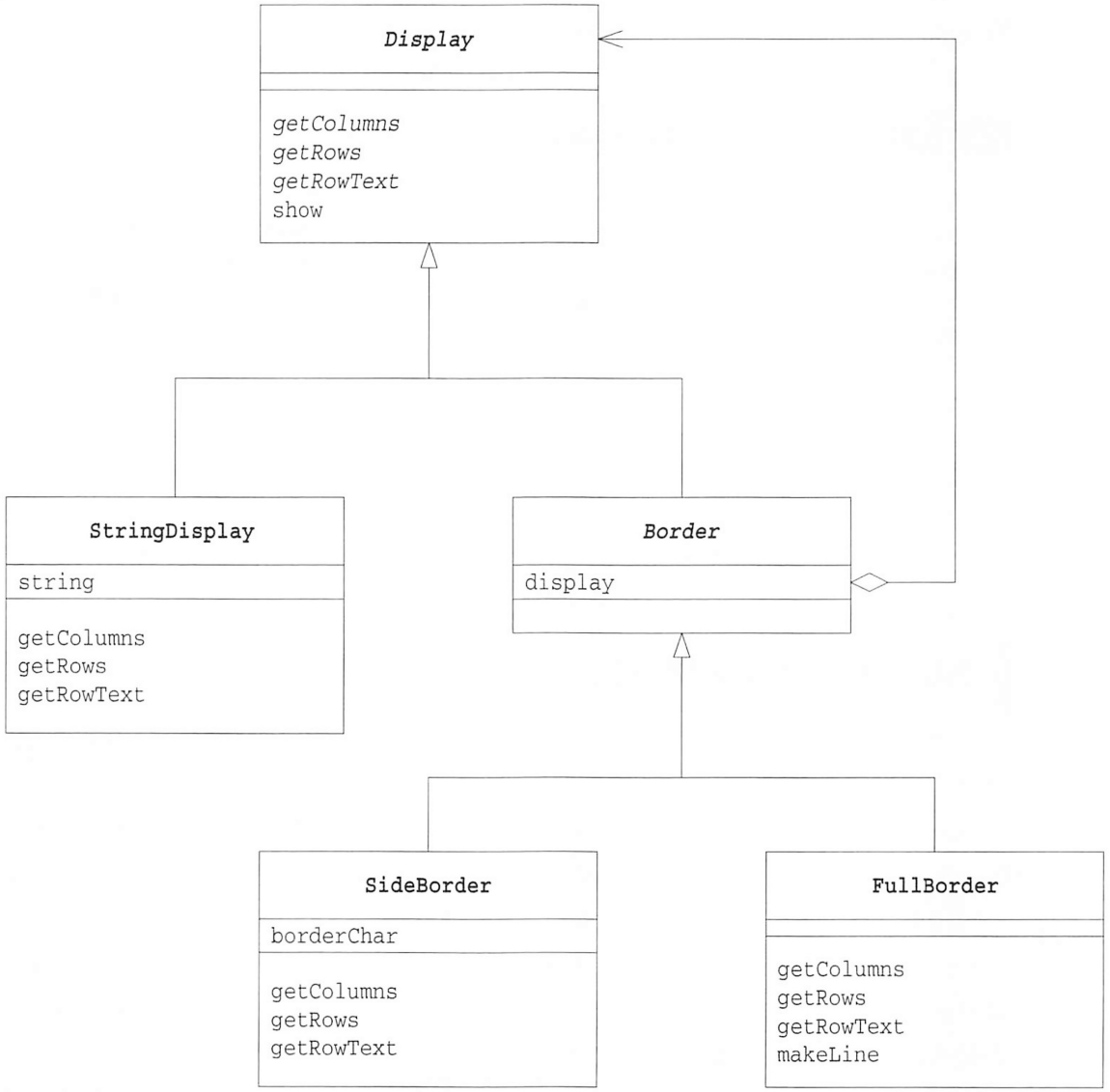
Decoratorパターンの必要性 (2/2)

❖飾り枠と中身を同一視したい（同じように扱いたい）ことがある
❖プログラミングでは，テキストを装飾するシーンが該当する



Decoratorパターン (1/6)

名　前	解　説
Display	文字列表示用の抽象クラス
StringDisplay	1行だけからなる文字列表示用のクラス
Border	「飾り枠」を表す抽象クラス
SideBorder	左右にのみ飾り枠をつけるクラス
FullBorder	上下左右に飾り枠をつけるクラス
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

Decoratorパターン (2/6)

```java
public abstract class Display {
    public abstract int getColumns();
    public abstract int getRows();
    public abstract String getRowText(int row);
    public final void show() {
        for(int i = 0; i < getRows(); i++) {
            System.out.println(getRowText(i));
        }
        System.out.println();
    }
}
```

```java
public abstract class Border extends Display {
    protected Display inside;

    protected Border(Display inside) {
        this.inside = inside;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Display org = new StringDisplay("Hello, world.");
        Display side1 = new SideBorder(org);
        Display side2 = new SideBorder(side1);
        Display full1 = new FullBorder(side2);
        Display full2 = new FullBorder(full1);
        Display side3 = new SideBorder(full2);

        org.show();
        side1.show();
        side2.show();
        full1.show();
        full2.show();
        side3.show();
    }
}
```

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

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

    public int getColumns() {
        return str.length();
    }

    public int getRows() {
        return 1;
    }

    public String getRowText(int row) {
        if(row == 0) {
            return str;
        } else {
            return null;
        }
    }
}
```

```java
public class SideBorder extends Border {
    private final char SIDE_MARK = '*';

    public SideBorder(Display inside) {
        super(inside);
    }

    public int getColumns() {
        return inside.getColumns() + 2;
    }

    public int getRows() {
        return inside.getRows();
    }

    public String getRowText(int row) {
        return SIDE_MARK
            + inside.getRowText(row) + SIDE_MARK;
    }
}
```

```java
public class FullBorder extends Border {
    private final char CORNER_MARK = '+';
    private final char HORIZONTAL_MARK = '-';
    private final char VERTICAL_MARK = '|';

    public FullBorder(Display inside) {
        super(inside);
    }

    public int getColumns() {
        return inside.getColumns() + 2;
    }

    public int getRows() {
        return inside.getRows() + 2;
    }

    public String getRowText(int row) {
        if(row == 0 || row == inside.getRows() + 1) {
            return CORNER_MARK + makeLine(inside.getColumns()) + CORNER_MARK;
        } else {
            return VERTICAL_MARK + inside.getRowText(row - 1) + VERTICAL_MARK;
        }
    }

    private String makeLine(int count) {
        StringBuffer buf = new StringBuffer();
        for(int i = 0; i < count; i++) {
            buf.append(HORIZONTAL_MARK);
        }
        return buf.toString();
    }
}
```

Decoratorパターン (3/6)

Display.java

```
public abstract class Display {
    public abstract int getColumns(); // 列数を得る

    public abstract int getRows();    // 行数を得る

    public abstract String getRowText(int row); // row行目の文字列を得る

    public final void show() {
        for(int i = 0; i < getRows(); i++) {
            System.out.println(getRowText(i));
        }
        System.out.println();
    }
}
```

サブクラスの挙動を定義する。
(Template Method)

Decoratorパターン (4/6)

Display型の何か (飾り枠 or 文字列)を
包めることだけを定義。
(飾り枠と中身の同一視)

Display.java

```java
public abstract class Display {
    public abstract int getColumns();
    public abstract int getRows();
    public abstract String getRowText(int row);
    public final void show() {
        for(int i = 0; i < getRows(); i++) {
            System.out.println(getRowText(i));
        }
        System.out.println();
    }
}
```

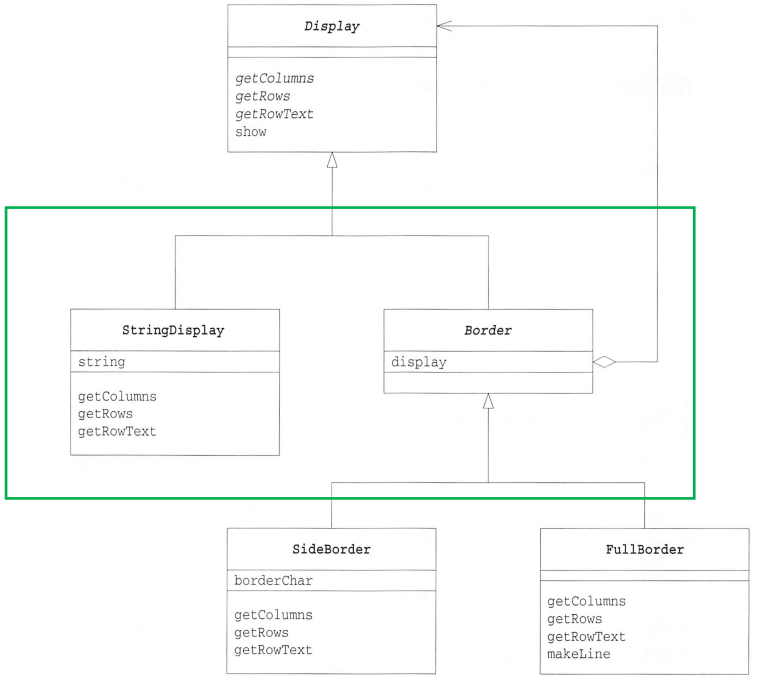
スーパークラスに
実装を義務付けられていた
メソッドを具体的に実装。

StringDisplay.java（文字列）

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

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

    public int getColumns() {
        return str.length();
    }

    public int getRows() {
        return 1;
    }

    public String getRowText(int row) {
        if(row == 0) {
            return str;
        } else {
            return null;
        }
    }
}
```

Border.java（飾り枠の抽象クラス）

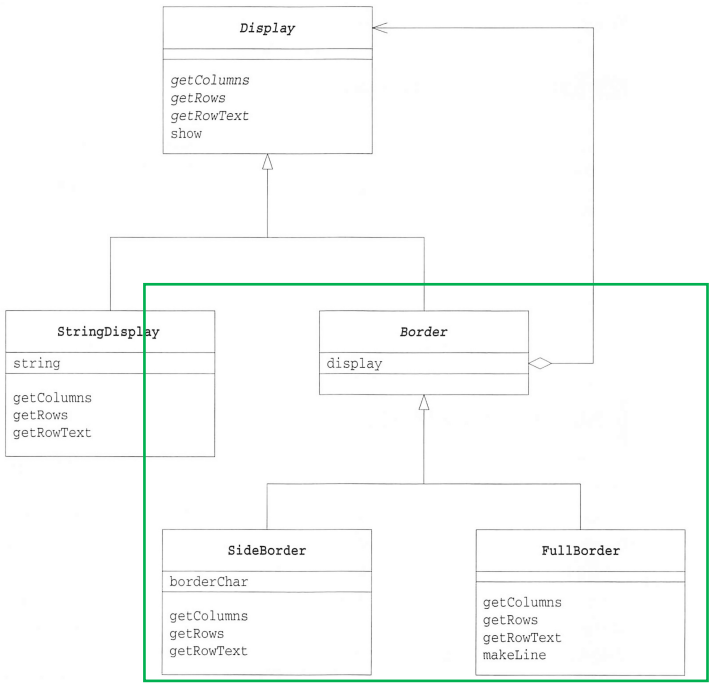
```java
public abstract class Border extends Display {
    protected Display inside;

    protected Border(Display inside) {
        this.inside = inside;
    }
}
```



Decoratorパターン (5/6)

Border.java

```java
public abstract class Border extends Display {
    protected Display inside;

    protected Border(Display inside) {
        this.inside = inside;
    }
}
```



SideBorder.java

```java
public class SideBorder extends Border {
    private final char SIDE_MARK = '*';

    public SideBorder(Display inside) {
        super(inside);
    }

    public int getColumns() {
        return inside.getColumns() + 2;
    }

    public int getRows() {
        return inside.getRows();
    }

    public String getRowText(int row) {
        return SIDE_MARK
            + inside.getRowText(row)
            + SIDE_MARK;
    }
}
```

スーパークラスに
実装を義務付けられていた
メソッドを具体的に実装。

FullBorder.java

```java
public class FullBorder extends Border {
    private final char CORNER_MARK = '+';
    private final char HORIZONTAL_MARK = '-';
    private final char VERTICAL_MARK = '|';

    public FullBorder(Display inside) {
        super(inside);
    }

    public int getColumns() {
        return inside.getColumns() + 2;
    }

    public int getRows() {
        return inside.getRows() + 2;
    }

    public String getRowText(int row) {
        if(row == 0 || row == inside.getRows() + 1) {
            return CORNER_MARK
                + makeLine(inside.getColumns()) + CORNER_MARK;
        } else {
            return VERTICAL_MARK
                + inside.getRowText(row - 1) + VERTICAL_MARK;
        }
    }

    private String makeLine(int count) {
        StringBuffer buf = new StringBuffer();
        for(int i = 0; i < count; i++) {
            buf.append(HORIZONTAL_MARK);
        }
        return buf.toString();
    }
}
```

Decoratorパターン (6/6)

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Display org = new StringDisplay("Hello, world.");
        Display side1 = new SideBorder(org);
        Display side2 = new SideBorder(side1);
        Display full1 = new FullBorder(side2);
        Display full2 = new FullBorder(full1);
        Display side3 = new SideBorder(full2);

        org.show();
        side1.show();
        side2.show();
        full1.show();
        full2.show();
        side3.show();
    }
}
```

実行結果

```
Hello, world.

*Hello, world.*

**Hello, world.**

+-----------------+
|**Hello, world.**|
+-----------------+

+-------------------+
|+-----------------+|
||**Hello, world.**||
|+-----------------+|
+-------------------+

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

作業準備

- IR05-2の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 05: Composite & Decorator > decorator.zip をWORK_DIRにダウンロード
  - unzip decorator.zip
  - cd decorator

IR05-2

- decorator内のコードを参考にして，対象の上下に飾りを付与するUpBottomBorderクラスを実装せよ。

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Display org = new StringDisplay("Hello, world.");
        Display side1 = new SideBorder(org);
        Display side2 = new SideBorder(side1);
        Display full1 = new FullBorder(side2);
        Display full2 = new FullBorder(full1);
        Display side3 = new SideBorder(full2);
        Display ub1 = new UpBottomBorder(side3);

        org.show();
        side1.show();
        side2.show();
        full1.show();
        full2.show();
        side3.show();
        ub1.show();
    }
}
```

目標の実行結果

```
Hello, world.

*Hello, world.*

**Hello, world.**

+-----------------+
|**Hello, world.**|
+-----------------+

+-------------------+
|+-----------------+|
||**Hello, world.**||
|+-----------------+|
+-------------------+

*+-------------------+*
*|+-----------------+|*
*||**Hello, world.**||*
*|+-----------------+|*
*+-------------------+*

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

Decoratorパターンのまとめと利用シーン

- 飾り枠と中身を同一視し，再帰的な構造を作るための考え方
  - Compositeパターンと似ているが，
    Decoratorパターンは次々と機能追加できることに主眼を置いている。

- 利用シーン
  - 透過的なインタフェースを保ったまま，次々と臨機応変に機能追加したい場合。
  - IR05-2では，新たな機能（上下に飾りをつける）を追加したにも関わらず，
    依然としてshow()という機能が呼び出せた（＝透過的なインタフェース）。

本日のまとめ

- 講義内容
  - Compositeパターン
  - Decorator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。

授業外レポート OR05

- 問1. デザインパターンの利点・効果・必要性などについて，自分の言葉で考察・説明せよ。
- 問2. 任意のデザインパターンを1件選択し，そのパターンを用いる前の独自のコードと，そのパターンを用いた後のコードの両方を作成して示し，そのパターンを用いたことの効果を，自分の言葉で説明せよ。

- 採点基準
  - オリジナリティ (75%)（注：授業中のコードの変数名を変えただけのコードは独自のものとはみなさない）
  - 内容の妥当性 (25%)

- 提出期限・提出方法
  - 11/12(金) 23:59
  - Bbの指定ページにPDF形式（54xxxx_lastname_firstname.pdf）で提出する。
  - コードも体裁を整えてレポート内に記述すること（javaファイルは受理しない）。
  - 提出期限を過ぎると提出場所が消える。

- 諸注意
  - これは「大学3年生向けの3週間の猶予があるレポート」である。稚拙なメモのような，レポートの体裁をなしていないものは大幅な減点，あるいは，採点対象外とする。
  - 他者に伝わるような，適切な文章で記述すること。これには入念な下書き・推敲が要るはずである。
  - 他者のコードの盗用が発覚した場合は，厳重に処罰する。学生間で類似レポートが発見された場合は，見せた方・見た方を問わず，双方を採点対象外・処罰の対象とする。