11 ソフトウェア工学
Software Engineering

デザインパターン

DESIGN PATTERNS

デザインパターンとは？

デザインパターン：　過去のソフトウェア設計者が生み出したオブジェクト指向設計の
ノウハウを蓄積し、名前をつけ、再利用しやすいようにカタログ化したもの

■各デザインパターンの主な内容

- そのデザインパターンの目的と効果
- どのような役割の部品（クラス，インタフェース）が必要か
- それらの部品をどのように組み立てるか
- 個々の部品がどのように関連して大きな機能を果たすのか

■下記文献の著者たちが23個のデザインパターンを書籍にまとめたのが始まり：

E. Gamma, R. Helm, R. Johnson, J. Vlissides:
Design Patterns: Elements of Reusable Object-Oriented Software,
Addison-Wesley (1995)

■この授業では下記の書籍を参考

結城　浩：Java言語で学ぶデザインパターン入門（増補改訂版），
SBクリエイティブ(2004)

デザインパターンの例

■生成に関するパターン

Factory Method	スーパークラスでインスタンスの作り方を抽象的に定め，具体的な作成はサブクラスにまかせる
Singleton	クラスのインスタンスが一つしかないことを保証する

■構造に関するパターン

Adapter	異なるインタフェースをもつ2つのクラスを接続するクラスを作る
Facade	複数のサブシステムの窓口となる共通のインタフェースを提供してシステムをシンプルにする

■振る舞いに関するパターン

Iterator	複数の要素をもつ集合体の要素を1つ1つ順番にアクセスする方法を提供する
Template Method	必要な処理の一部をテンプレートとして抽象的に記述し，その具体（カスタマイズ）をサブクラスにまかせる
Observer	状態が変化するクラスと，その変化を通知してもらうクラスを分けて設計する

今回の授業では，Template Methodパターン と Adapterパターン を具体的に学び，Factory Methodパターンの調査を演習課題とする

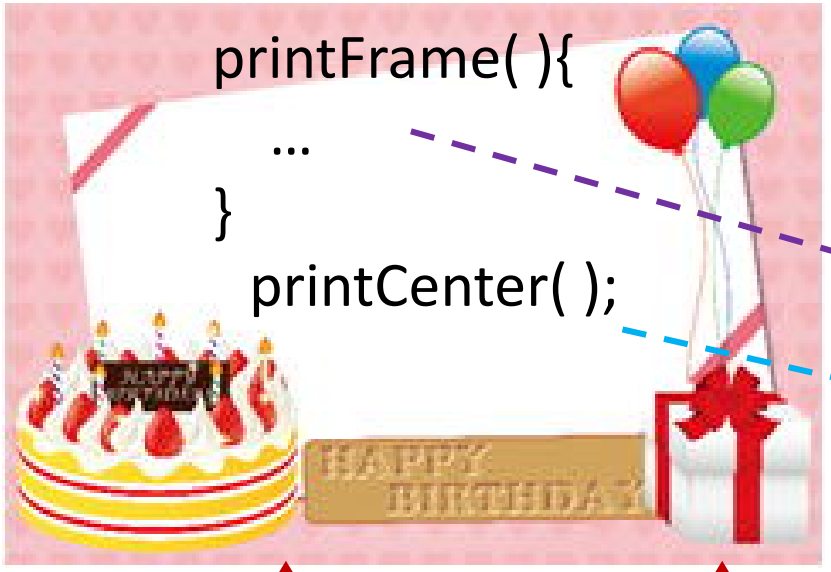
Template Method パターン

必要な処理の一部をテンプレートとして抽象的に記述し，その具体（カスタマイズ）をサブクラスにまかせる.

いろいろな人がカスタマイズを考案し，それを簡単に実装できる.

テンプレートの概念

抽象カード



テンプレート
処理の枠組み(framework)を定め，
細部は利用者がカスタマイズできる

枠組みはテンプレートで実装済み

抽象メソッドは未実装
（カスタマイズ可能な部分）

実装1

```
printCenter() {
   画像をプリント
}
```

具体カード

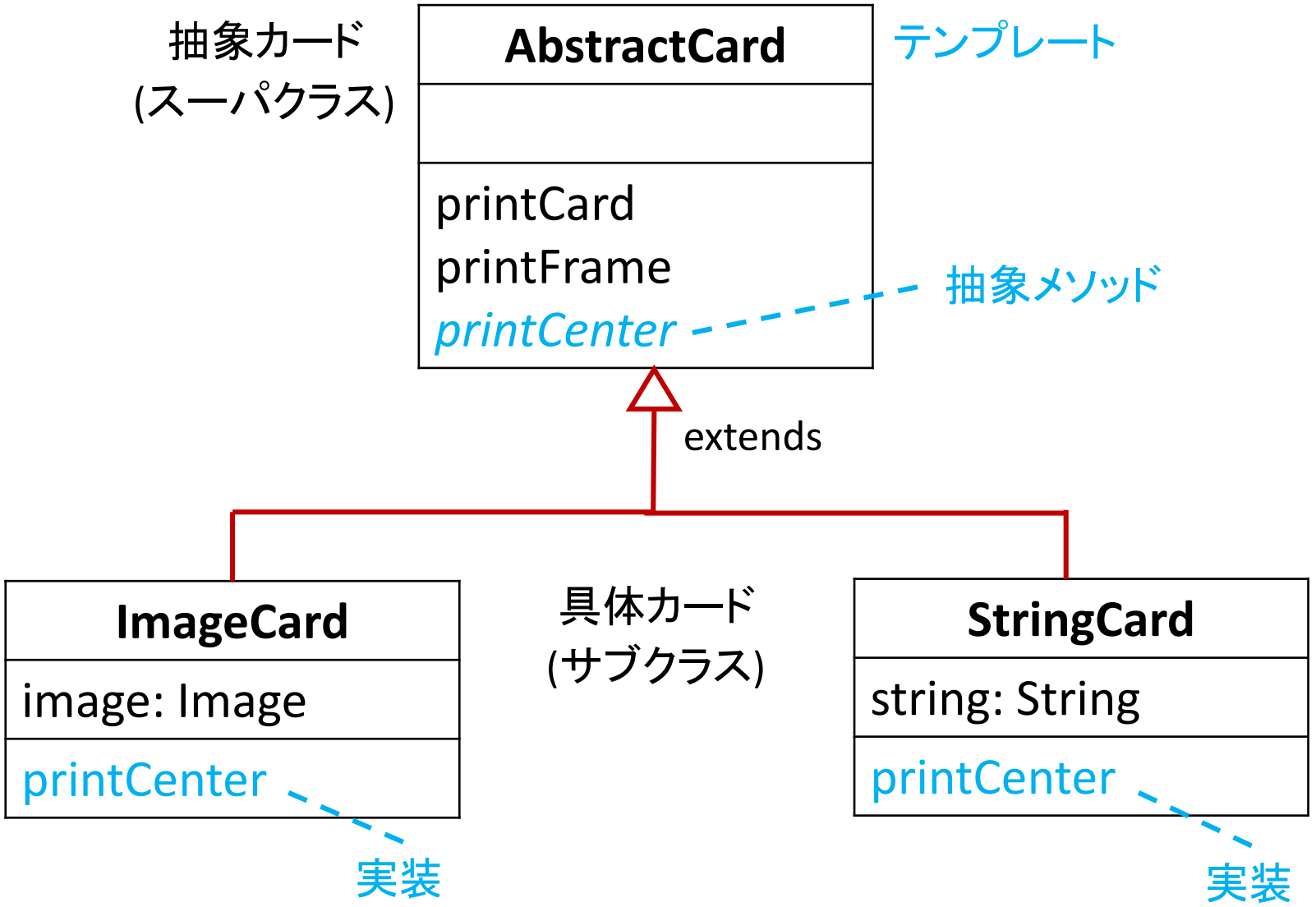
実装2

```
printCenter() {
   文字列をプリント
}
```

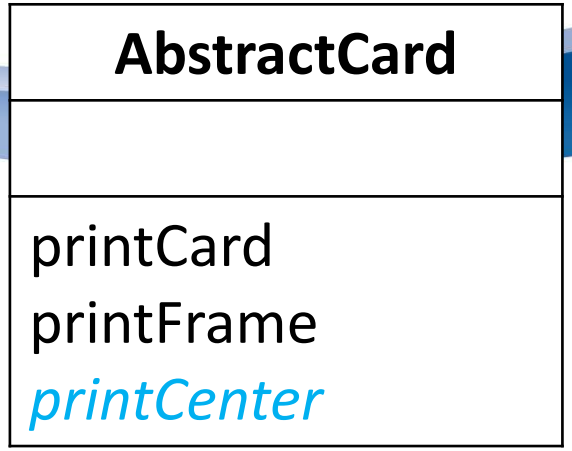
利用者1

利用者2

テンプレートのデザイン



AbstractCard クラス



【Javaによる記述】

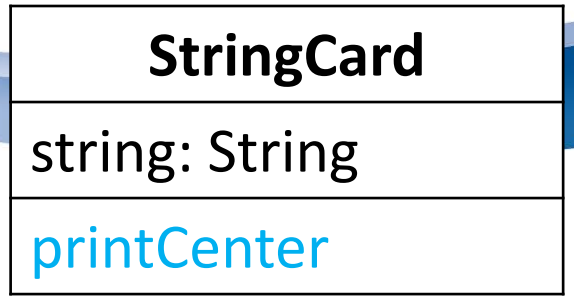
抽象クラス

```
public abstract class AbstractCard {
  public void printCard(Graphics g) {
    printFrame(g);
    printCenter(g);
  }

  public void printFrame(Graphics g) {
    〈省略〉 //グラフィクス g にきれいな枠を表示する高品質の長いプログラム;
  }

  public abstract void printCenter(Graphics g);
}
```

テンプレートメソッド

抽象メソッド

StringCardクラス



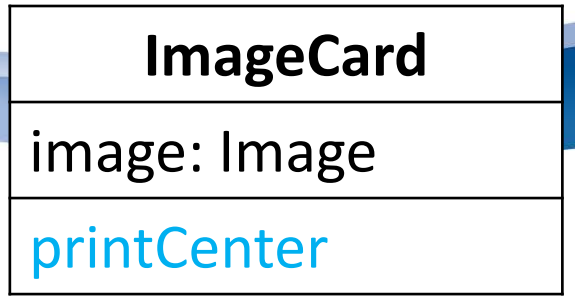
サブクラスの定義

```
public class StringCard extends AbstractCard{
  private String string;

  public StringCard(String string) {
    this.string = string;
  }

  public void printCenter(Graphics g) {
    g.drawString(string, 140, 70);
}
```

コンストラクタ

(140, 70) は表示する位置の x-y座標

ImageCardクラス



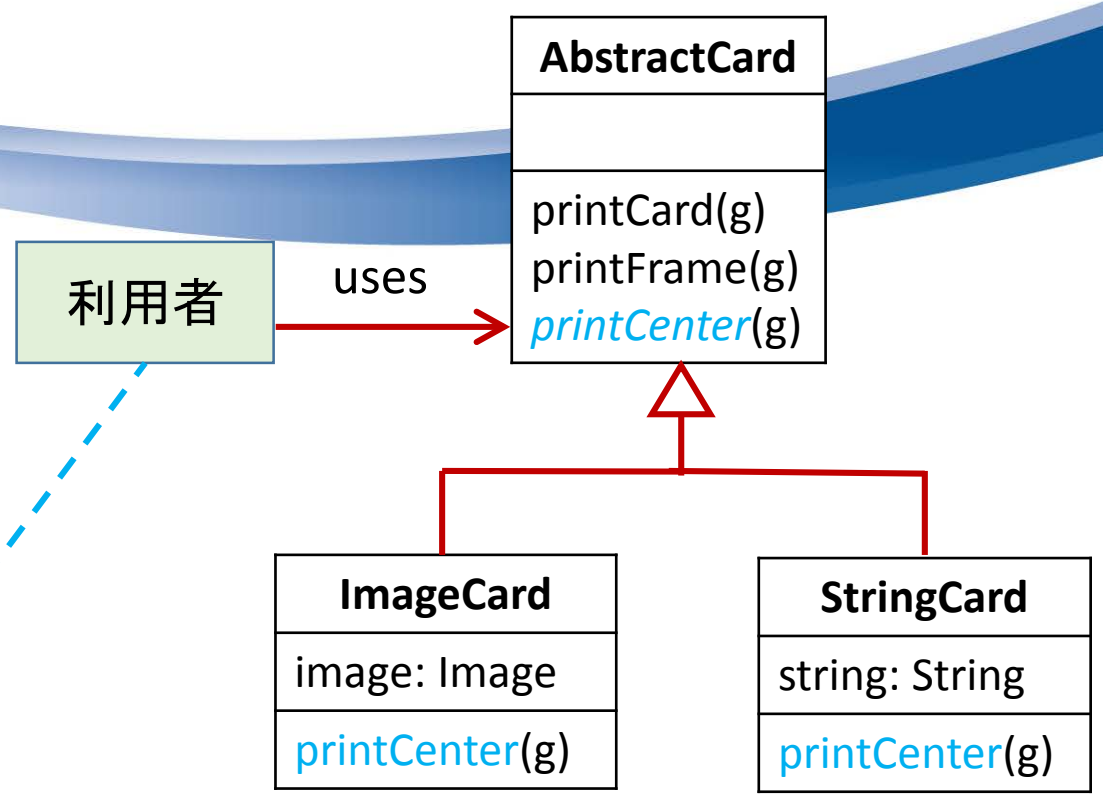
サブクラスの定義

```
public class ImageCard extends AbstractCard{
  private Image image;

  public ImageCard(Image image) {
    this.image = image;
  }

  public void printCenter(Graphics g) {
    g.drawImage(image, 140, 70);
}
```

コンストラクタ

（実際にはもう少し複雑な実装となる）

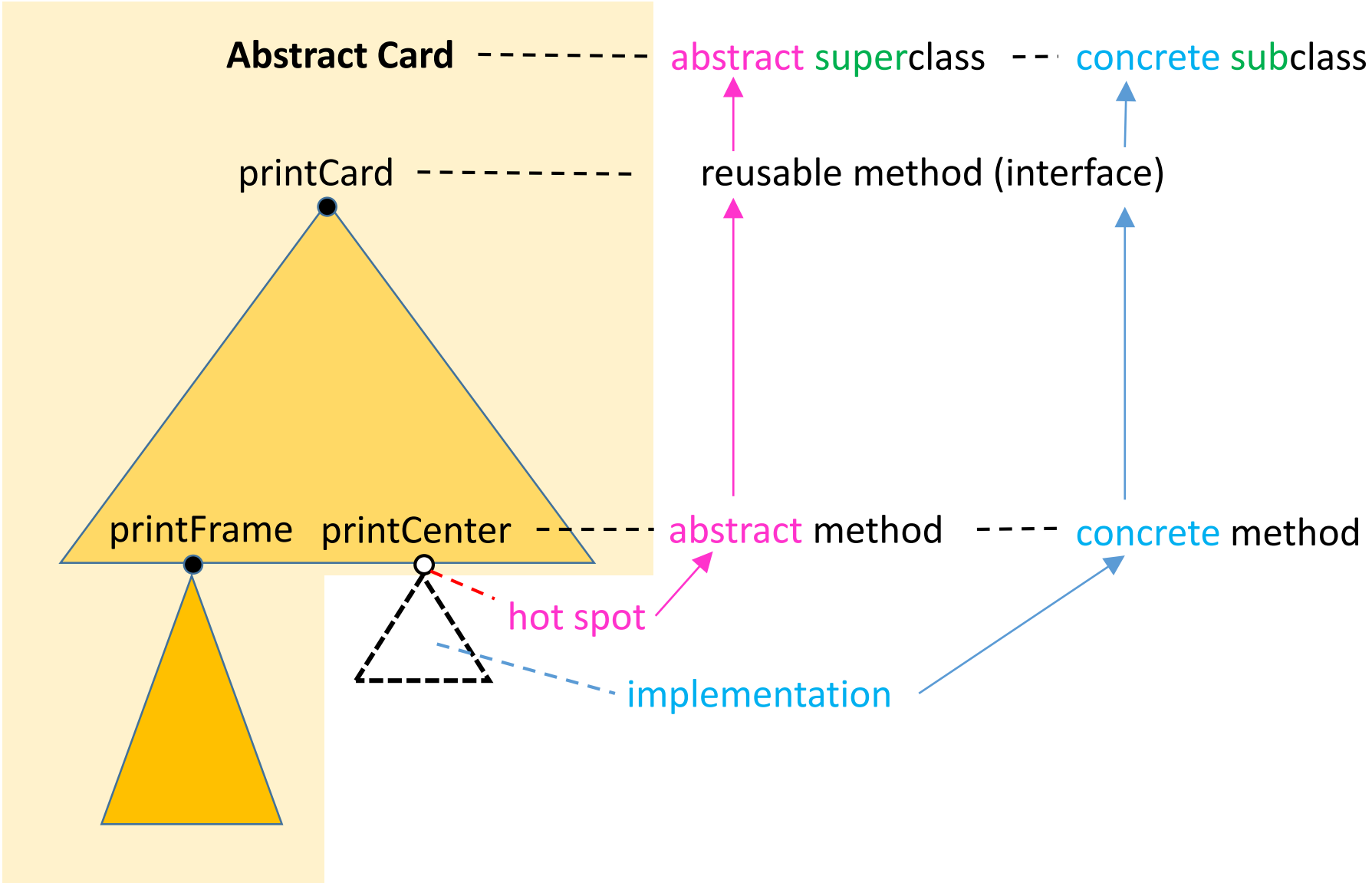
利用者のメソッド




```
public void printTwoCards(Graphics g,
                          Image image,
                          String string) {
  AbstractCard cardholder[2];
  cardholder[0] = new ImageCard(image);
  cardholder[1] = new StringCard(string);
  for(int i=0; i<2; i++) {
    cardholder[i].printCard(g);
  }
}
```

AbstractCard が規定する
統一的なインタフェース

ホットスポット


Abstract Card
abstract superclass
concrete subclass
printCard
reusable method (interface)
printFrame
printCenter
abstract method
concrete method
hot spot
implementation

Template Method パターン

一般化すると，つぎのデザインパターンが得られる

【Template Method パターン】

参加者(participants)

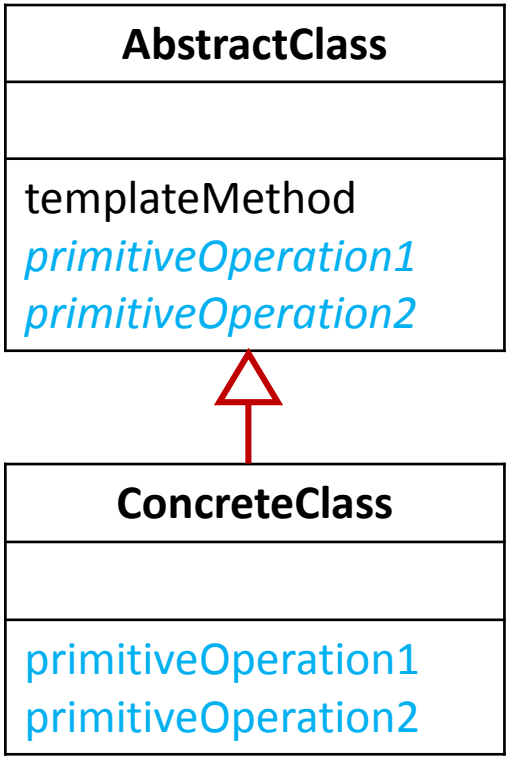
●AbstractClass (Application)

― 抽象メソッド *primitiveOperations* を定義

― *primitiveOperations* を使い，アルゴリズムの枠組みを定義する templateMethod を実装

●ConcreteClass (MyApplication)

― この具体クラス特有の処理を実行するように primitiveOperations を実装



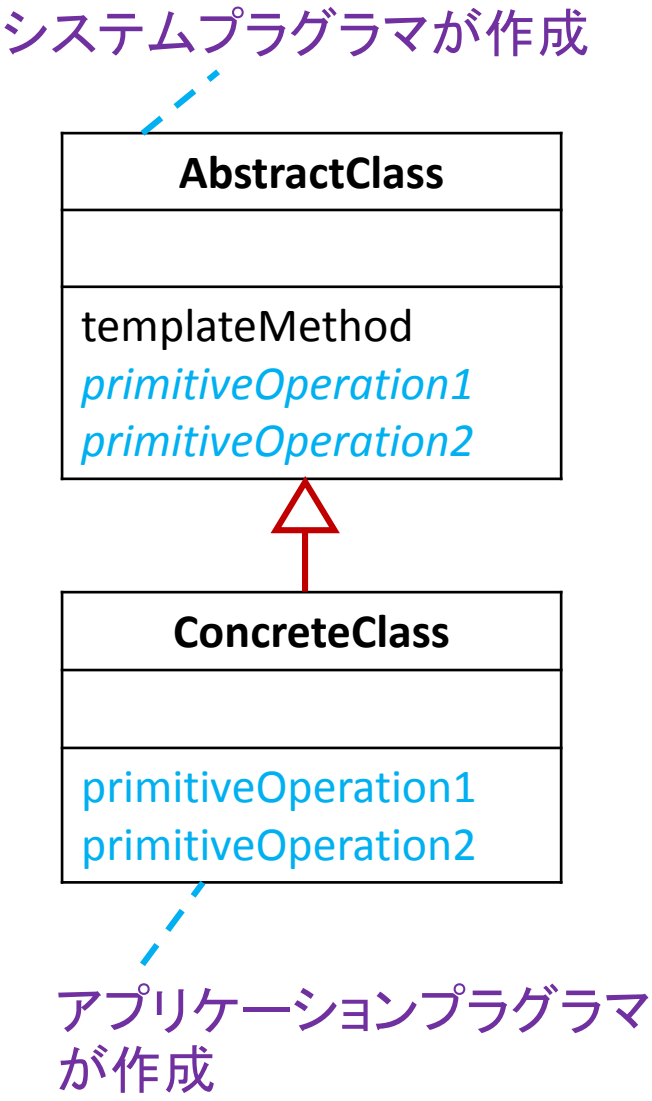
Template Methodパターンのありがたみ

【システムプラグラマの立場からのありがたみ】

- AbstractClassの中にはConcreteClass名を書かないので，すべてのConcreteClassに影響を与えずにtemplateMethodの実装の修正（進化）が可能
- アプリケーションプログラマにAbstractClassのソースコードを公開しなくてもよい

【アプリケーションプラグラマの立場からのありがたみ】

- AbstractClassのtemplateMethodで複雑なアルゴリズムが記述されているとき，そのコードを修正することなく，ConcreteClassでそれをカスタマイズして再利用可能
- 複数のConcreteClass を実装したとき，ConcreteClassの具体的な種類を気にせずに，それらのインスタンスをいずれもAbstractClass型とみなし，AbstractClassが定める統一的なインタフェース(API)（templateMethodの呼び出し）でプログラムを記述可能

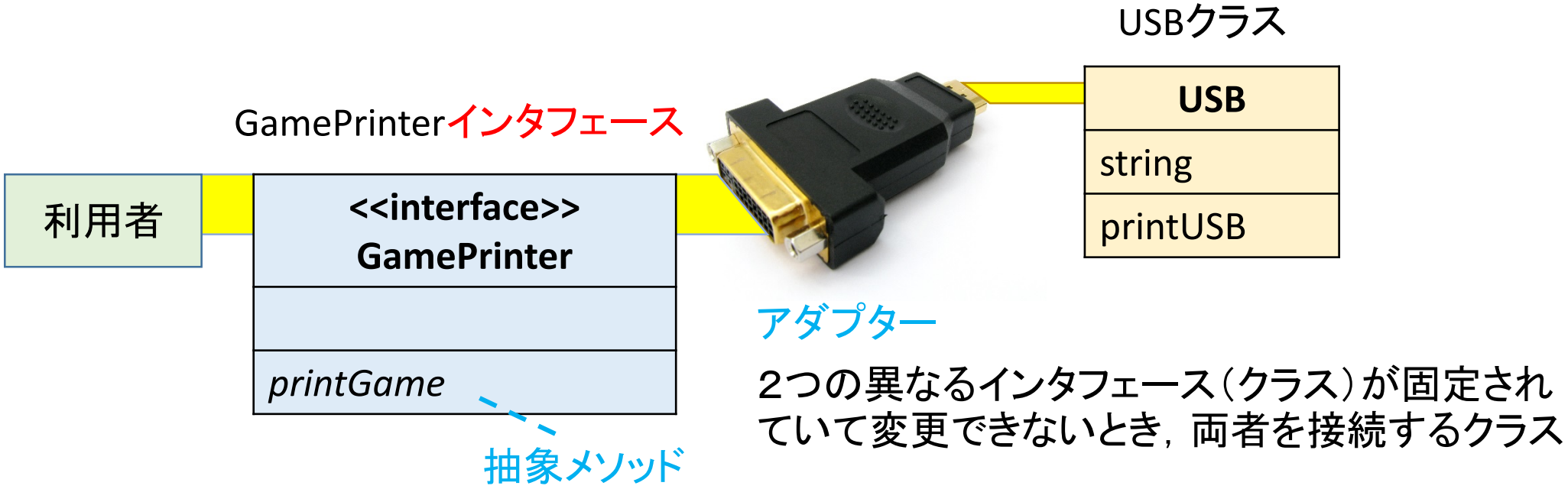


Adapter パターン

異なるインタフェースをもつ2つのクラスを接続する.

開発するシステムのインタフェースを変えずに，外部のいろいろなインタフェースに接続できる.

アダプターの概念

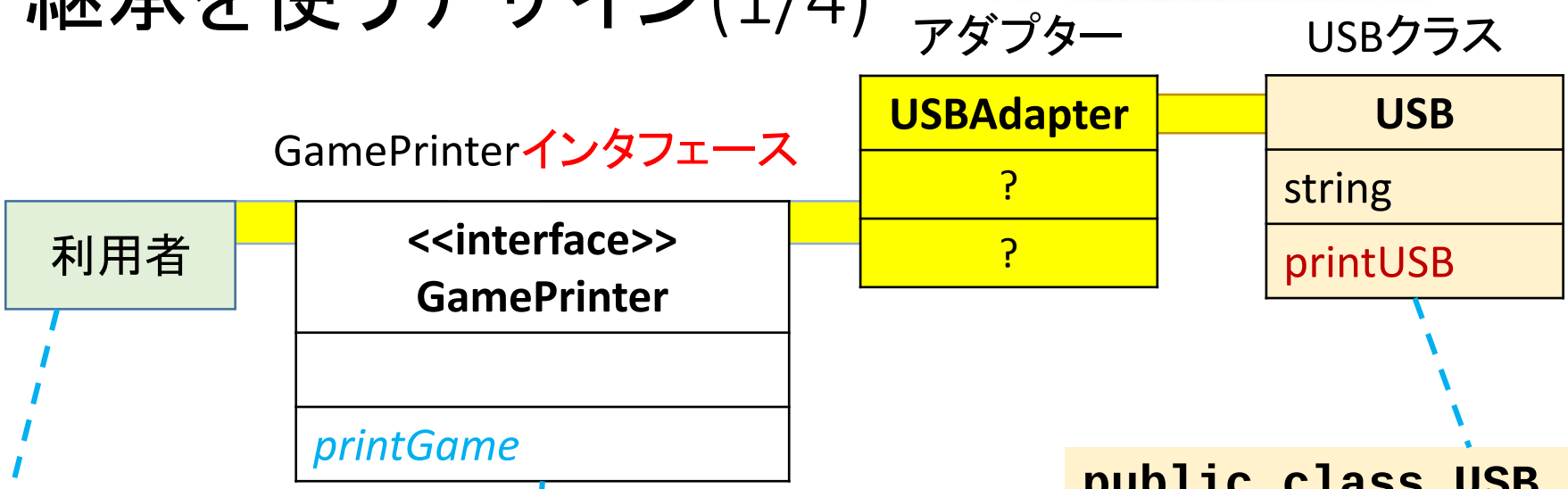


2つの異なるインタフェース（クラス）が固定されていて変更できないとき，両者を接続するクラス

【提供されているもの】
USBクラスがプリンタメーカーから提供済．usb.printUSB( ) により string を印字可．

【欲しいもの】
ゲームメーカーが規定するGamePrinterインタフェースを実装したクラス（アダプター）．
利用者は，gameprinter.printGame( ) によって string を印字したい．

継承を使うデザイン(1/4)




```
public interface GamePrinter {
  public abstract void printGame( );
}
```

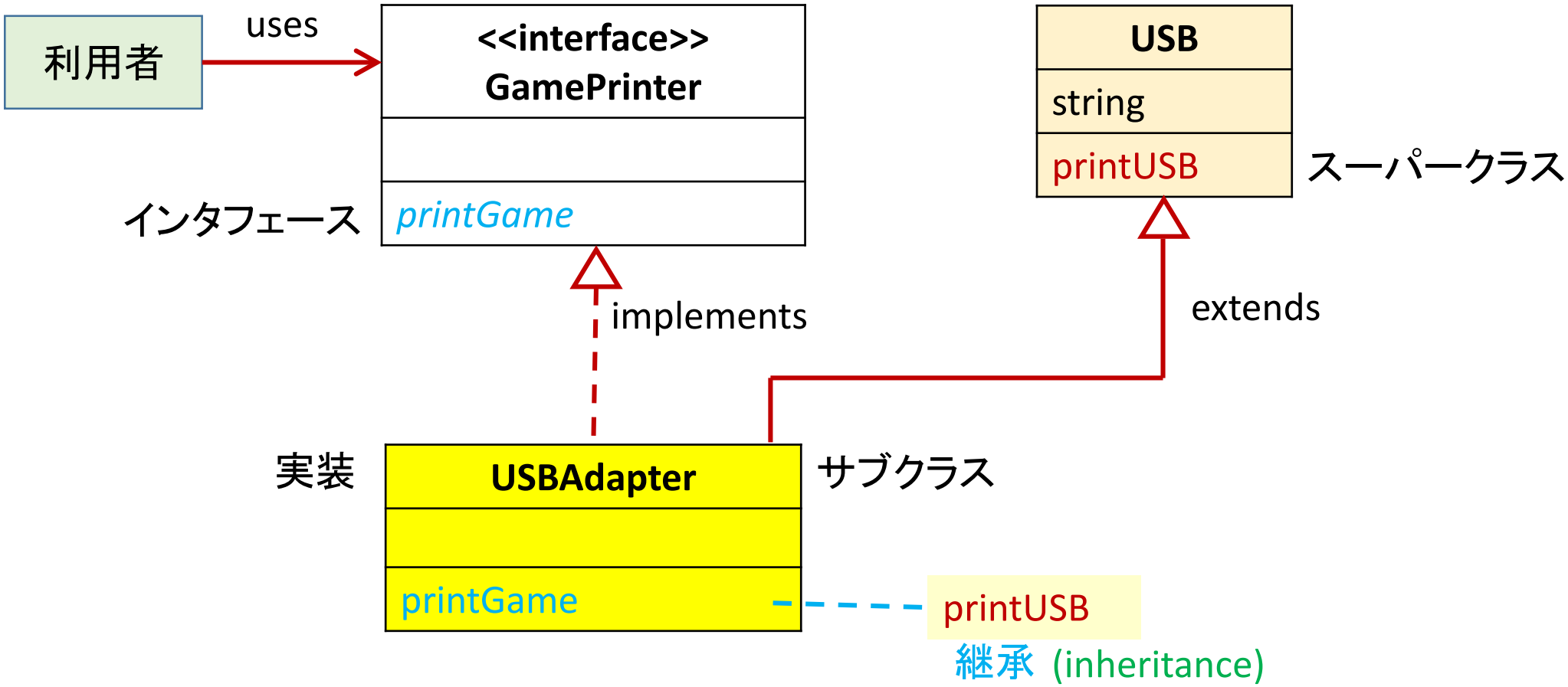
```
public class USB {
  private String string;
  public USB(String string)
   {..}
  public void printUSB()
   {..}
}
```

実際にはクラスファイルのみ提供

```
public static void main(String[ ] args) {
  GamePrinter gp = new USBAdapter(”You win!”);
  gp.printGame( );
}
```

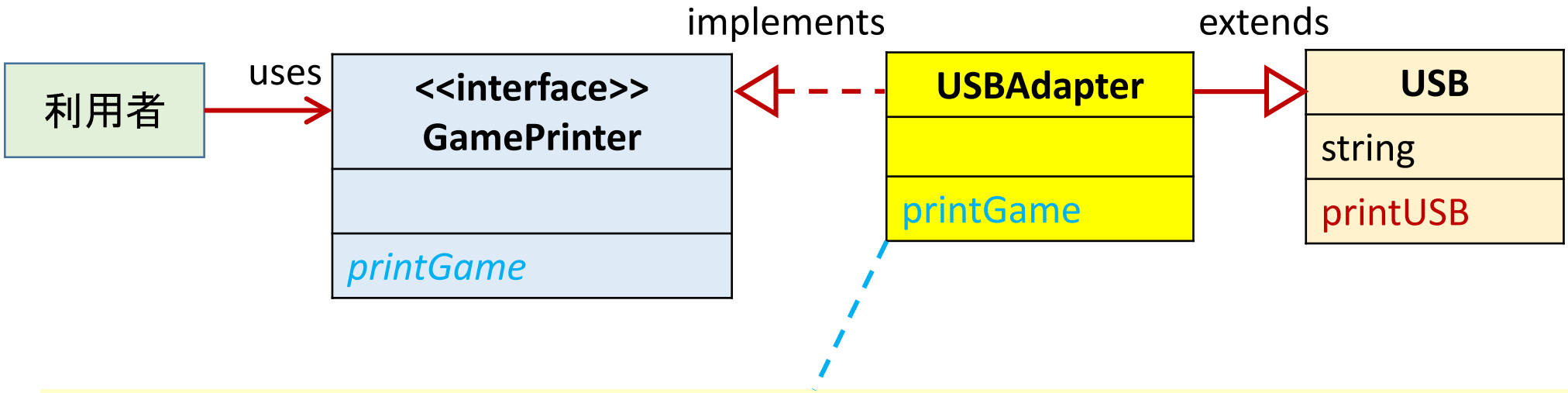
利用者に対して USB が隠されている

継承を使うデザイン(2/4)



※ USBAdapter は，GamePrinter型でもある

継承を使うデザイン(3/4)




```
public class USBAdapter extends USB implements GamePrinter {
  public USBAdapter(String string) {
    super(string);
  }
  public void printGame() {
    printUSB();
  }
}
```

コンストラクタ

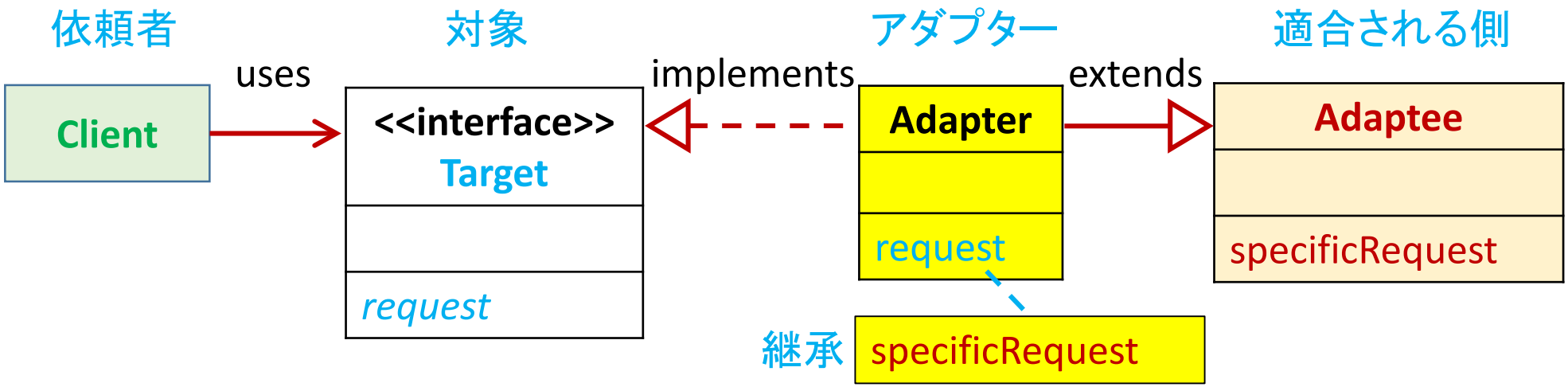
継承を使うデザイン(4/4)

一般化すると，つぎのデザインパターンが得られる

【Adaperパターン：継承版】

参加者(participants)

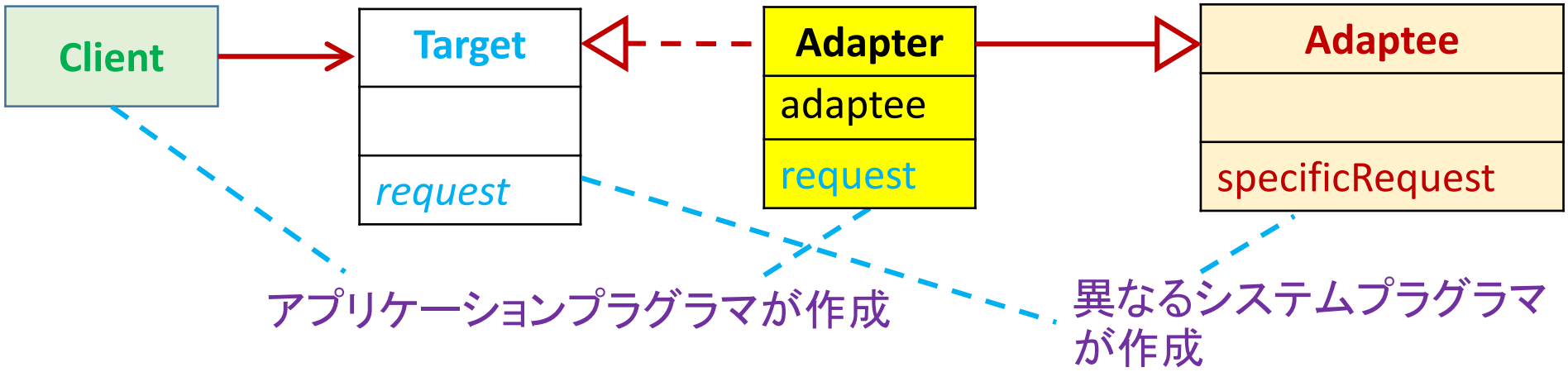
- ●Client:　Targetインターフェースをもつオブジェクトを利用する
- ●Target：　Client が使用できる特定のインタフェースを規定する
- ●Adaptee:　適合させたい既存のインタフェースをもつ実装
- ●**Adapter**:　AdapteeのインタフェースをTargetインタフェースに適合させる



Adapterパターンのありがたみ(1/2)

【アプリケーションプラグラマの立場からのありがたみ】

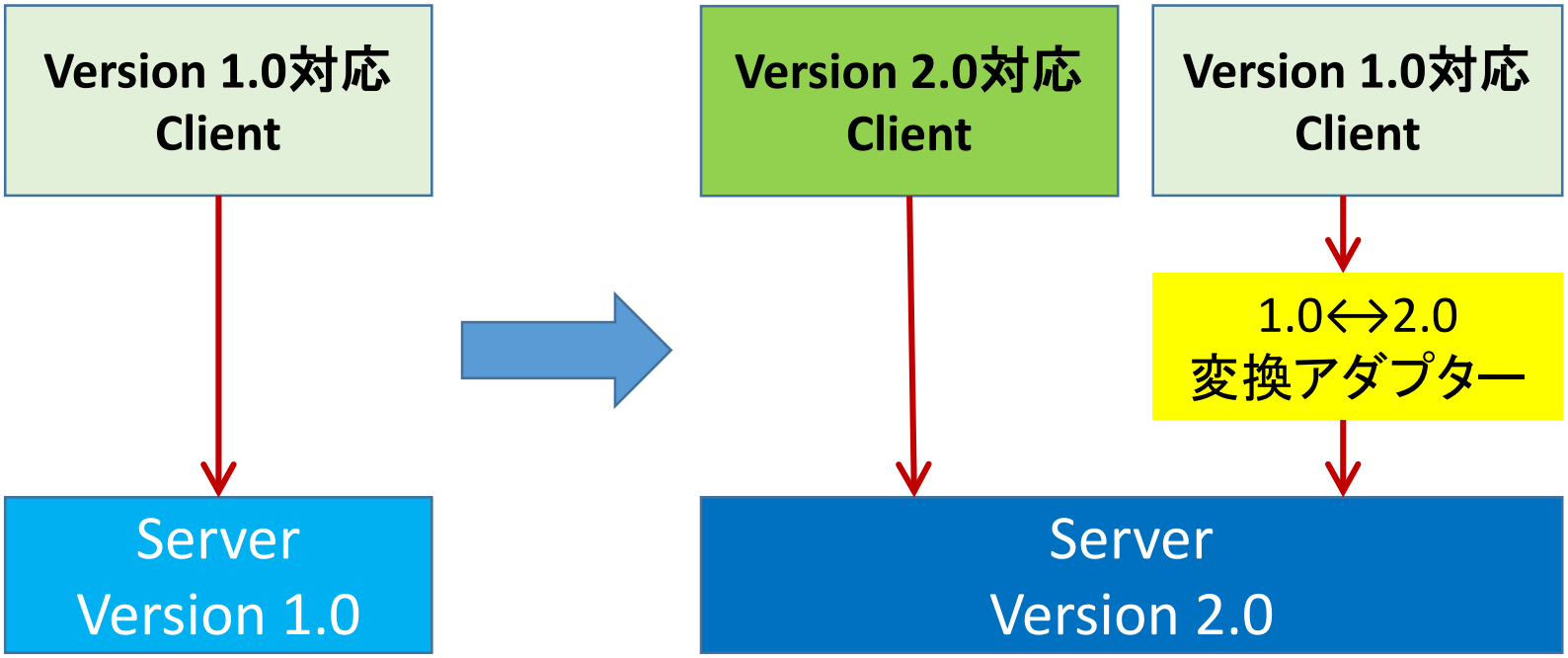
- AdapteeのspecificRequestで複雑なアルゴリズムが記述されているとき，Adapterがそのコードを修正することなくカスタマイズして，Targetインタフェースを変えずに再利用可能
- 複数のAdapterを実装したとき，Adapterの具体的な種類を気にせずに，それらのインスタンスをいずれもTarget型とみなし，Targetが定める統一的なインタフェース(API)（request）でプログラムを記述可能



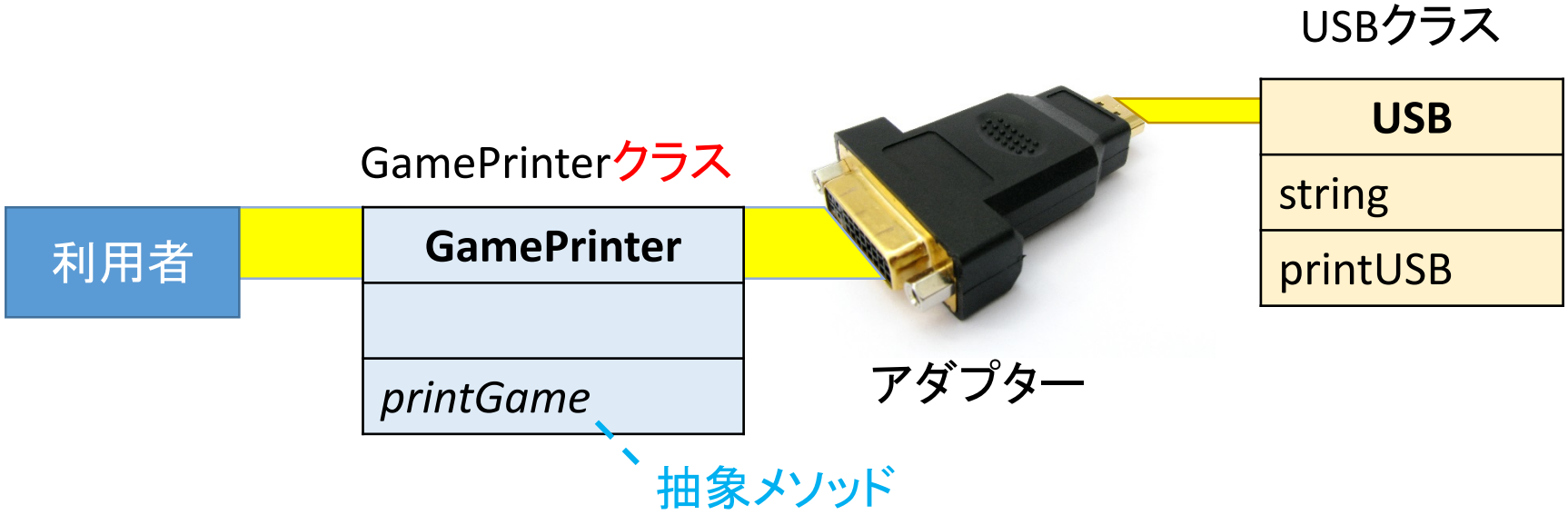
Adapterパターンのありがたみ(2/2)

【システムプラグラマの立場からのありがたみ】

- ClientとTargetに影響を与えずに Adaptee (specificRequest)の実装を変更可能
- アプリケーションプログラマにTargetとAdapteeのソースコードを公開しなくてもよい
- Adapterを提供することにより，システムの古い版と新しい版を互換性を保って共存させられる（下図）



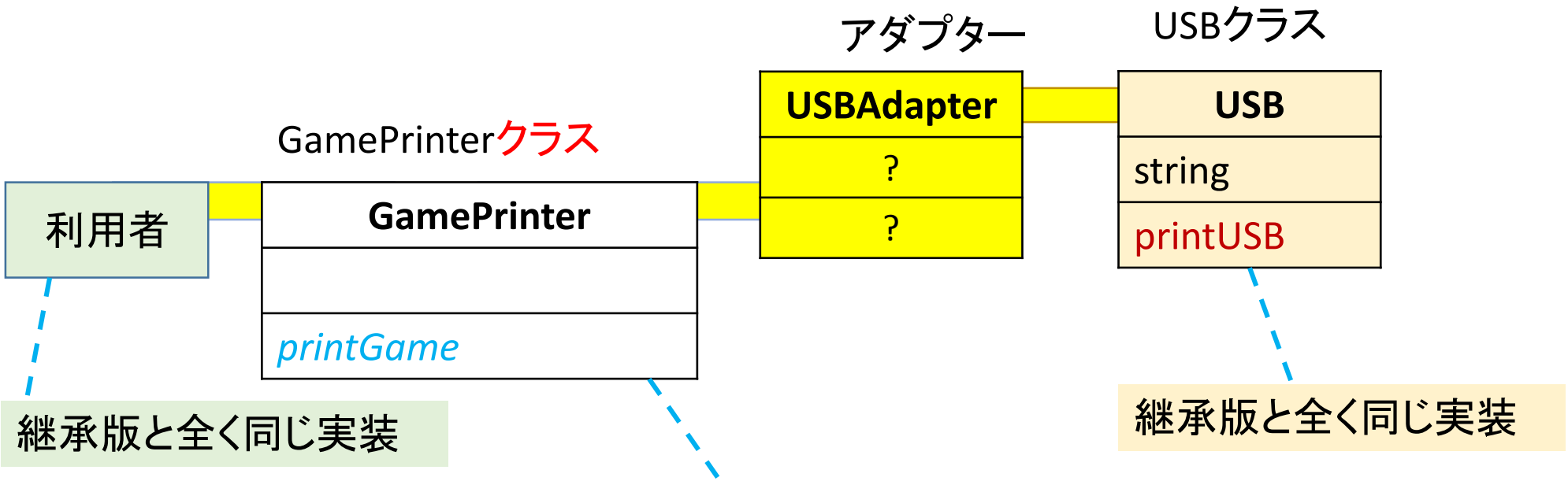
アダプターの概念（別バージョン）



GamePrinterクラスだけを継承したい.
USBは継承しない.

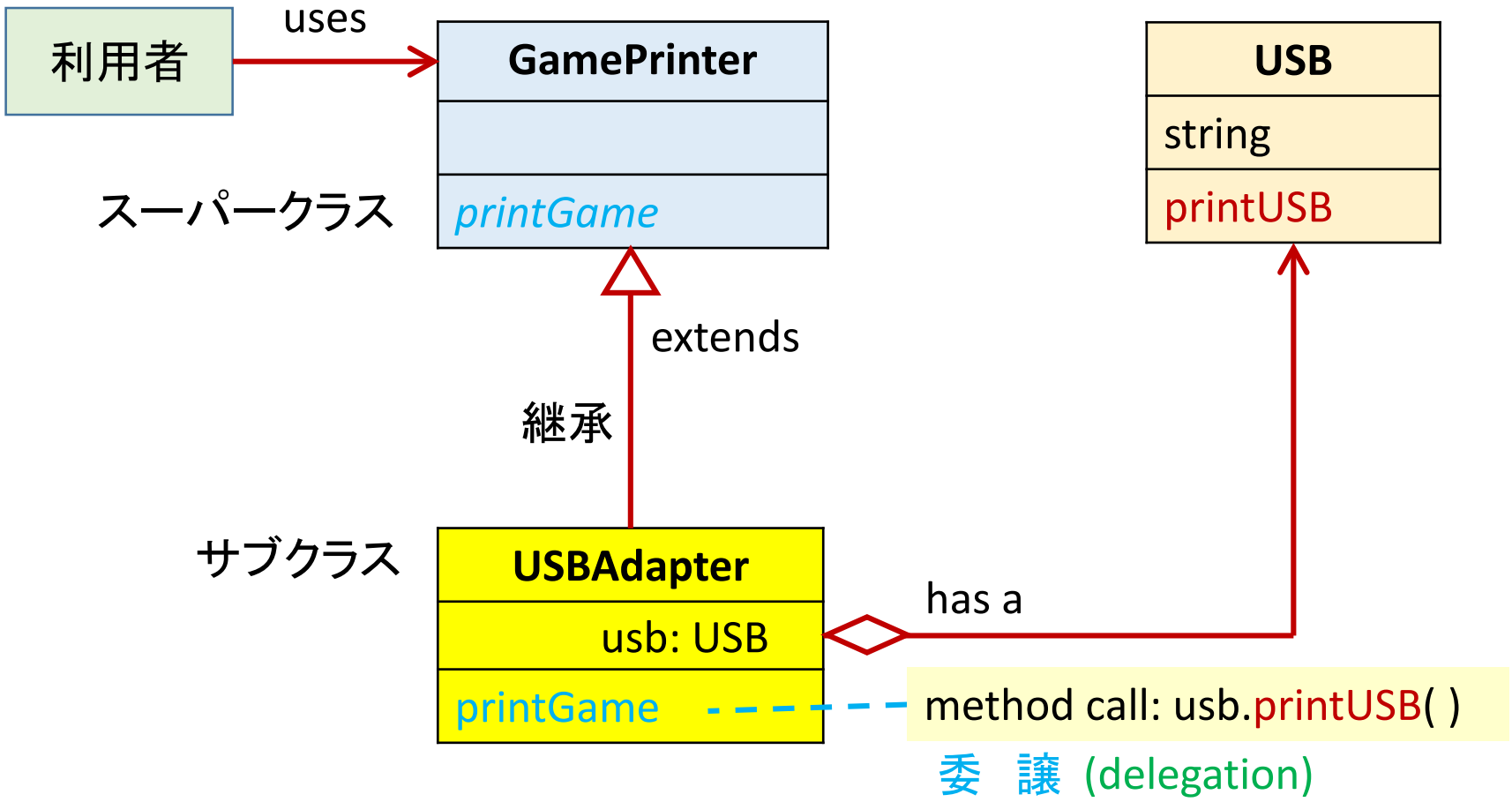
※ Javaは多重継承の機能を許していない

委譲を使うデザイン(1/4)

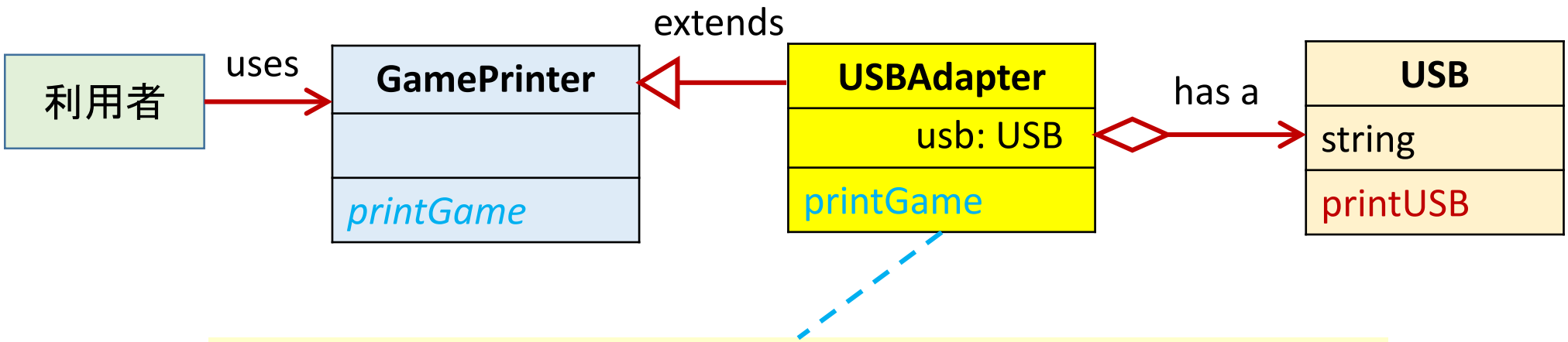



```
public abstract class GamePrinter {
  public abstract void printGame( );
}
```

委譲を使うデザイン(2/4)



委譲を使うデザイン(3/4)

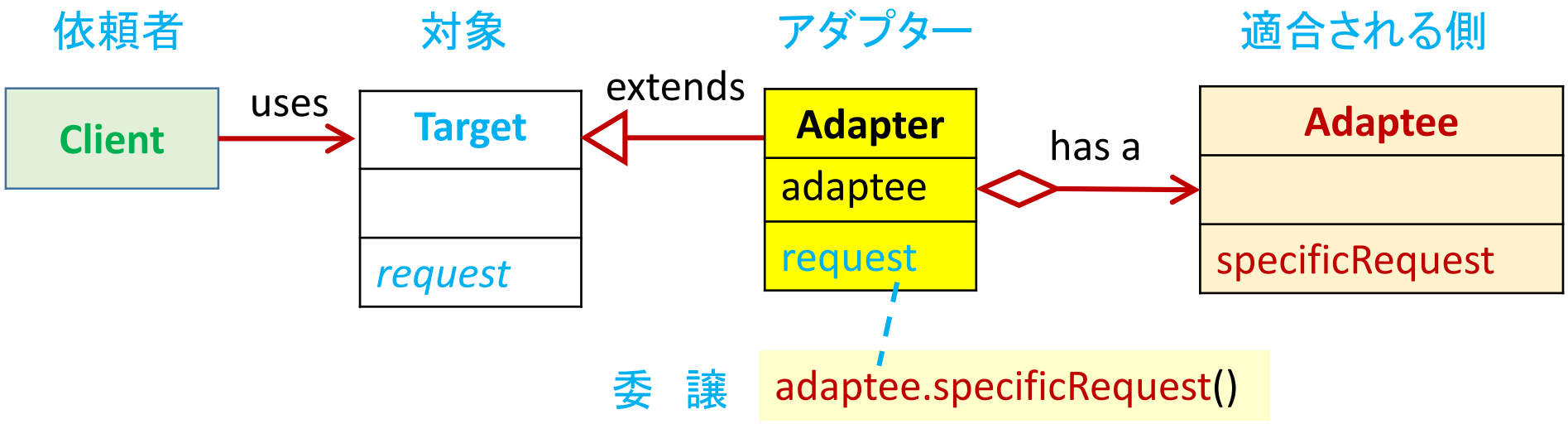



```
public class USBAdapter extends GamePrinter {
  private USB usb;
  public USBAdapter(String string) {
    usb = new USB(string);
  }
  public void printGame() {
    usb.printUSB();
  }
}
```

委譲を使うデザイン(4/4)

一般化すると，つぎのデザインパターンが得られる

【Adaperパターン：委譲版】



演習問題 11

Factory Method パターンについて調べ，概略を説明しなさい．
特に，つぎの点について示すこと．
(1) Factory Method パターンの目的
(2) パターンの要素（参加者）とその役割
(3) パターンの構造（UMLで図示すること）