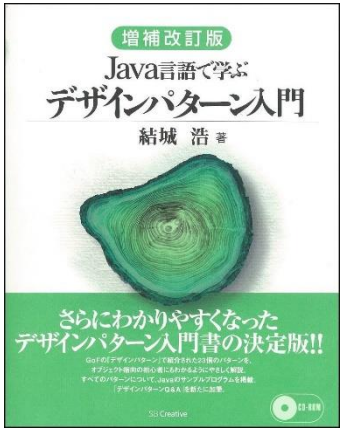
教科書輪講

Javaで学ぶデザインパターン入門

秋山研　Ｂ４　林　孝紀



目次

1. デザインパターンとは
2. UMLについて
3. Iteratorパターン
4. Adapterパターン
5. 実装演習
6. 解説

デザインパターンとは

1. デザインパターンとは
2. UMLについて
3. Iteratorパターン
4. Adapterパターン
5. 実装演習
6. 解説

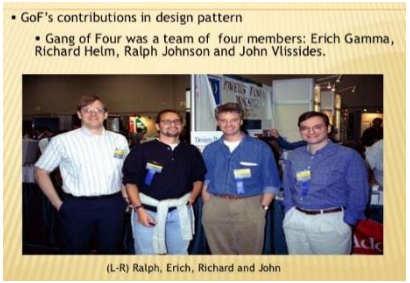
デザインパターン

デザインパターン：

　プログラムを開発するときに同じような書き方をする場合が多い。これをパターン化してGoFがまとめたもの。

　そのため経験豊富なプログラマーは当たり前に使ってることが多い。

デザインパターンを（うまく）使用すれば、（楽して）保守性が高くきれいなコードが書けるようになる（かも）



クラスの責務

・クラスはそのクラスのオブジェクトが責任を持って
行わなければならない振る舞いのことを責務と呼ぶ。

・責務には以下の二種類がある

- 振る舞いに対する責務 (Responsibility for Behavior)
- 知識に対する責務 (Responsibility for Knowledge)

これらの責務を意識すること（肥大化を防ぐ）により、
デザインパターンの思想がよりわかりやすくなる（かも）

UMLについて

1. デザインパターンとは
2. UMLについて
3. Iteratorパターン
4. Adapterパターン
5. 実装演習
6. 解説

UML

統一モデリング言語（Unified Modeling Language）の略で
主にオブジェクト指向分析や設計のための、
記法の統一がはかられたモデリング言語。

要約すると…

クラス図書いたりするやつ

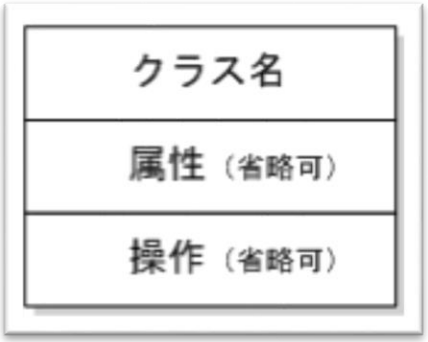
表3: UMLダイアグラムの種類

ダイアグラム	役割	開発フェーズ
ユースケース図	システムの境界，使用機能を定義	分析
アクティビティ図	システムの動作の流れの表現	分析，設計
状態図	オブジェクトの取りうる状態，遷移を表現	分析，設計
クラス図	概念や静的なクラス間相互関係を表現	分析，設計
パッケージ図	各モデル要素の階層的グルーピング	分析，設計
相互作用図		
シーケンス図	オブジェクト間のメッセージ交換の時系列表現	分析，設計
コラボレーション図	オブジェクトの集団の協調動作の表現	分析，設計
オブジェクト図	実行時のオブジェクト状態のスナップショット	分析，設計
コンポーネント図	システムを構成する実行可能モジュールやソースコードの物理的構造を表現	設計
配置図	システムを構成するマシンや装置の繋りを表現	設計

＊UMLのツールがない人はおそらく
入れておいておいた方がいいかも？
今回はeclipseのプラグインのAmaterasUML
を使用している。

クラスの書き方

正式には以下のとおりに書かれる。



1. クラス名
   <<種別>> パッケージ名 : クラス名
2. 属性
   可視性 名前 : 型 = 初期値 {制約条件}
3. 操作
   可視性 名前 (引数の名前 : 型) : 戻り値の型

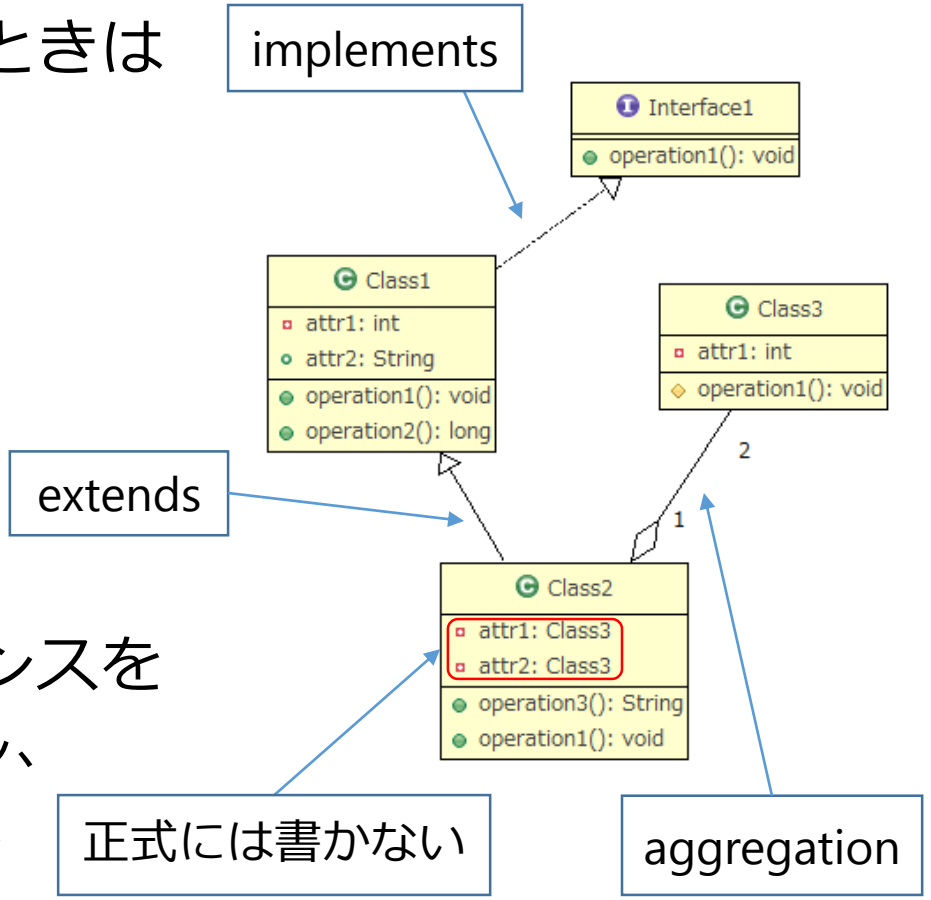
＊今回はAmaterasUML
を利用しているため
この記法とは少し違った
記法で記述している。

可視性	意味
+	public ： 全てにおいて参照可能
-	private ： 自クラスでのみ参照可能
#	protected ： 自クラス及びその派生クラスにおいて参照可能
~	package ： 同パッケージ内で参照可能

関係の書き方

1. 継承
   Class2 が Class1 を継承するときは
   実線の矢印で表す
2. 実装
   Class1 が Interface1 を実装
   (implements) するときは
   点線の矢印で表す
3. 集約
   Class2 が Class3 のインスタンスを
   持つときひし形の矢印で表し、
   始点と終点に数の対応を書く

そのほかの関連は線で示す。



Iteratorパターン

1. デザインパターンとは
2. UMLについて
3. Iteratorパターン
4. Adapterパターン
5. 実装演習
6. 解説

Iteratorとは

イテレータは日本語では反復子と呼ばれる。
集合を順番に指し示していくもの。
これは集合の実装によらず、すべての集合に共通している
性質。

集合を表すクラスをイテレータにすることにより、
集合の実装によらず、集合を走査することができる

JavaではIteratorインターフェイスが存在し、hasNext()とnext()
という二つのメソッドが宣言されている。
またイテレータにすることができることを示す
Iteratableインターフェイスがあり、iterator()が宣言されている。
基本的にはこれらを用いてイテレータを作成する。
また拡張for構文の実装にも用いられている。

Iteratorを作成するクラスの実装

主に自作コレクションクラスを作成するときに使用する。

例えば、Bookクラスを格納するBookShelfクラスを作成する。
このクラスは以下のように実装する。

BookShelf.java

```
public class BookShelf implements Aggregate {
    private Book[] books;
    private int last = 0;
    public BookShelf(int maxsize) {
        this.books = new Book[maxsize];
    }
    public Book getBookAt(int index) {
        return books[index];
    }
    public void appendBook(Book book) {
        this.books[last] = book;
        last++;
    }
    public int getLength() {
        return last;
    }
    @Override
    public Iterator iterator() {
        return new BookShelfIterator(this);
    }
}
```

Iteratorの作成

普通はIteratable<Book>だが、
総称性を使いたくないのか
自作クラスを使用してる

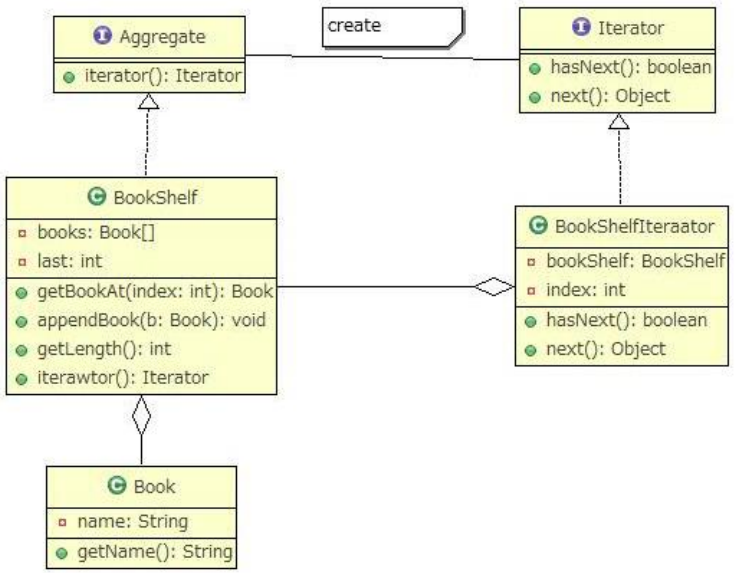
Iterator本体の実装

BookShelf.java

```
public class BookShelfIterator implements Iterator {
    private BookShelf bookShelf;
    private int index;
    public BookShelfIterator(BookShelf bookShelf) {
        this.bookShelf = bookShelf;
        this.index = 0;
    }
    public boolean hasNext() {
        if (index < bookShelf.getLength()) {
            return true;
        } else {
            return false;
        }
    }
    public Object next() {
        Book book = bookShelf.getBookAt(index);
        index++;
        return book;
    }
}
```

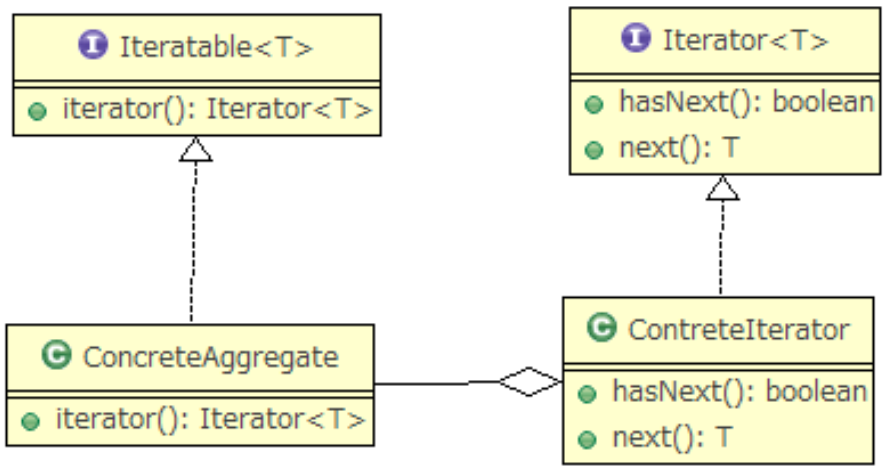
ここも普通Iterator<Book>になる

クラス図



Iteratorパターンのまとめ

すでにJavaに実装されているインターフェイスである
Iteratableインターフェイスと Iteratorインターフェイスを
使用したクラス図を以下に示す。



Adapterパターン

1. デザインパターンとは
2. UMLについて
3. Iteratorパターン
4. Adapterパターン
5. 実装演習
6. 解説

Adapterの役割

古い実装と新しい実装で使用するインターフェイスが異なるとする。このとき古いメソッドを用いて、新しいインターフェイスのメソッドを実装したくなることがある。このインターフェイス間の仕様の差を埋めるのがAdapterパターンである。
これには継承を用いたものと委譲を用いたものがある。

例えば、文字列を以下の二種類でプリントすることを考える。
現在これは以下のメソッドで実装されている。

- *ではさんで表示 : Banner#showWithAster() *String*
- ()ではさんで表示 : Banner#showWithParen() (String)

継承を利用したAdapterパターン

ここで新しいインターフェイスPrintにはprintWeakとprintStrongというメソッド名で実装したい。

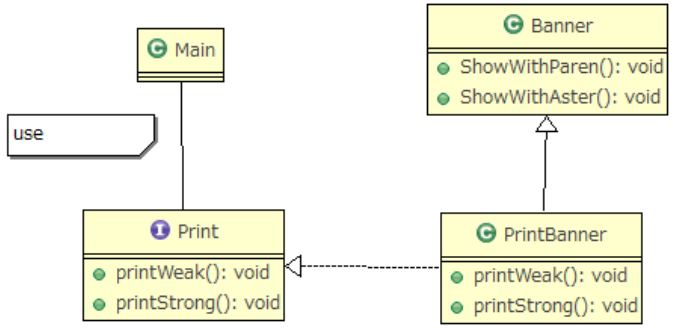
そのためBannerクラスを継承し、Printインターフェイスを実装したPrintBannerクラスを作成する。

PrintBanner.java

Bannerクラスを継承することでそのメソッドを呼び出す

```
public class PrintBanner extends Banner implements Print {
    public PrintBanner(String string) {
        super(string);
    }
    public void printWeak() {
        showWithParen();
    }
    public void printStrong() {
        showWithAster();
    }
}
```

クラス図



委譲を利用したAdapterパターン

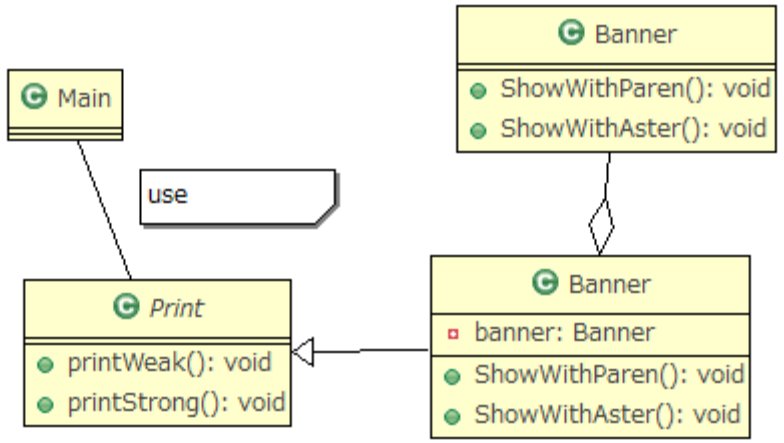
また、別の解決策としてBannerクラスへの委譲をすることも考えられる。

Bannerクラスのインスタンスを持って、そのメソッドを呼び出す

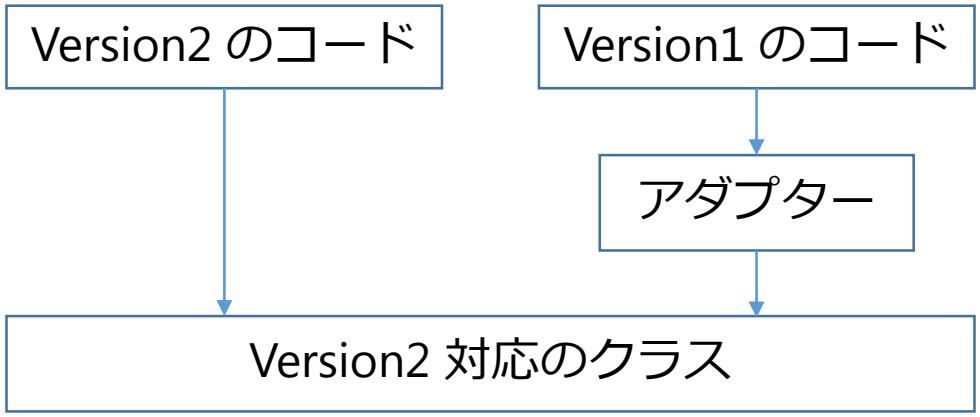
PrintBanner.java

```
public class PrintBanner extends Print {
    private Banner banner;
    public PrintBanner(String string) {
        this.banner = new Banner(string);
    }
    public void printWeak() {
        banner.showWithParen();
    }
    public void printStrong() {
        banner.showWithAster();
    }
}
```

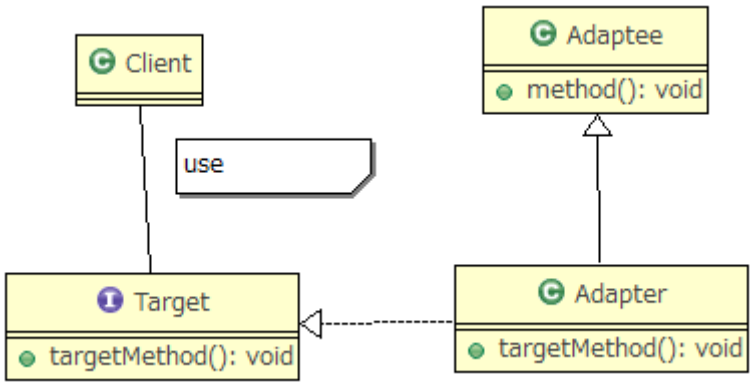
クラス図



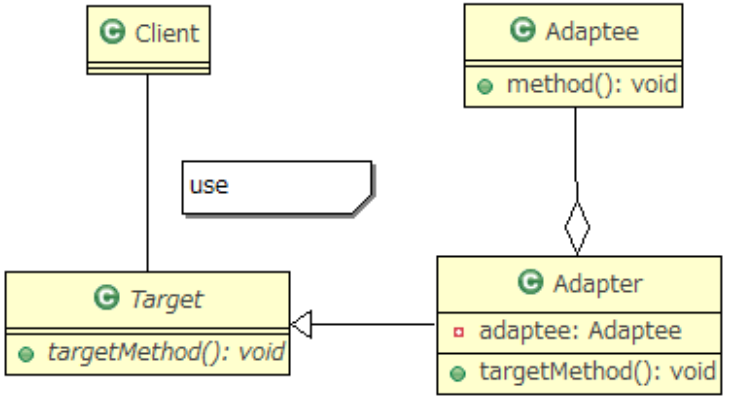
Adapterパターンまとめ



継承を用いたAdapter パターン



委譲を用いたAdapter パターン



実装演習

1. デザインパターンとは
2. UMLについて
3. Iteratorパターン
4. Adapterパターン
5. 実装演習
6. 解説

実装演習

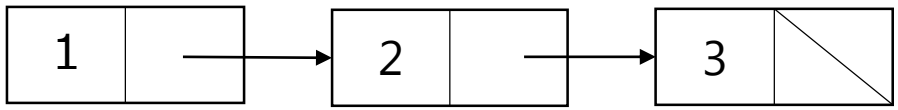
Entryクラスを継承したContentクラスがある。
EntryクラスはgetContent()メソッドを持つ。
Entryクラスの集合をこれまでリストで管理していたが、
リストの入れ子構造も扱いたくなり、そこでConsセルを
使用して実装することになった。
Consの仕様は以下のとおりである。

1. ConsはConsまたはEntryクラスを継承したクラスのペア
   として表すことができる。
2. Consはcar(), cdr() によりそれぞれのメンバを返す
3. 空のConsとはnullのペアをあらわす
4. 簡単のため外部からnullの入力がくることは
   考えなくてよい（実装したい人は各自）

実装演習

5. Consにaddするとは以下の操作を表すこととする。
   - car部がnullのConsにaddするとは、car部にオブジェクトを代入すること
   - cdr部がnullのConsにaddするとは、cdr部にcar部がそのオブジェクトとなっているConsを代入すること
   - それ以外の場合はfalseを返して失敗とする

Consセルを用いたリストの実装



これは[1,[2,[3,()]]]
とあらわすものとする

このメソッドを実装しておくことにより、リストとの互換性を保つ。
ConsがAbstractCollectionなどを継承してるとさらにいいかも。
その際size()メソッドは個人の自由で実装してください。

実装演習

6. 今回は簡単のためEntryクラスに変更を
   与えてもいいものとする。（普段は継承、委譲すべき？）

7. ConsのgetContentは"[(carのgetContent),(cdrのgetContent)]"を
   返すものとする。（フォーマットは個人で変えても可）

ConsのgetContentの実行例

```
Cons c0 = new Cons();                                              ⟶  [(),()]
c0.add(new Content("1"));                                          ⟶  [1,()]
c0.add(new Content("2"));                                          ⟶  [1,[2,()]]
Cons c1 = new Cons(new Content("1"),new Content("2"));             ⟶  [1,2]
c1.add();                                                          ⟶  false
Cons c2 = new Cons();
c2.add(new Content("1"));
c2.add(c0);                                                        ⟶  [1,[[1,[2,()]],()]]
Cons c3 = new Cons();
c3.add(new Content("1"));
c3.add(c1);                                                        ⟶  [1,[1,2]]
```

実装演習

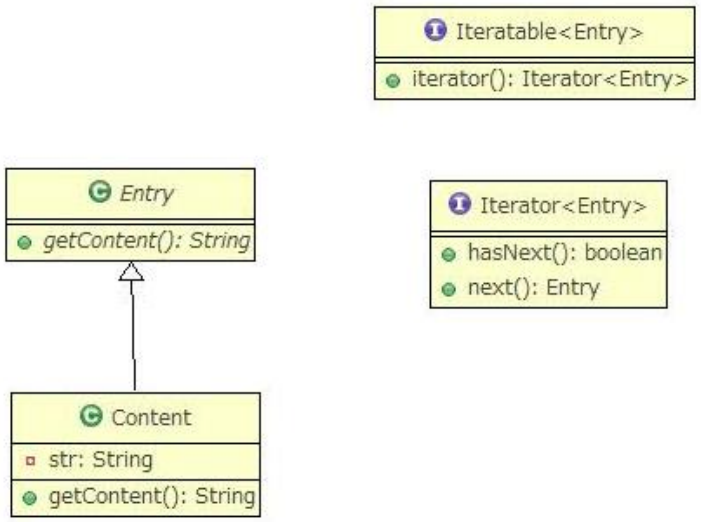
現在exerciseフォルダにあるコードが与えられています。

1. 上で説明したConsクラスを実装してください。
2. 自分が作成したConsクラスをイテレータにすることを考え、iteratorメソッドを実装してください。

注意：例えば[1,[2,3]]で表されるConsセルをイテレータにした場合最初のnextメソッドでは1、次は[1,2]を返すようします。
最初1を返し、その次に2、その次に3を返すものではありません。
（要はConsセルを返していいということ）
[1,[2,[3,()]]]では1,2,3の順に返します。
[1,[[2,[3,()]],()]]では1を返し、その後[2,[3,()]]を返します。

exercise フォルダの中身

Exercise フォルダには以下のようなクラスが実装されている。



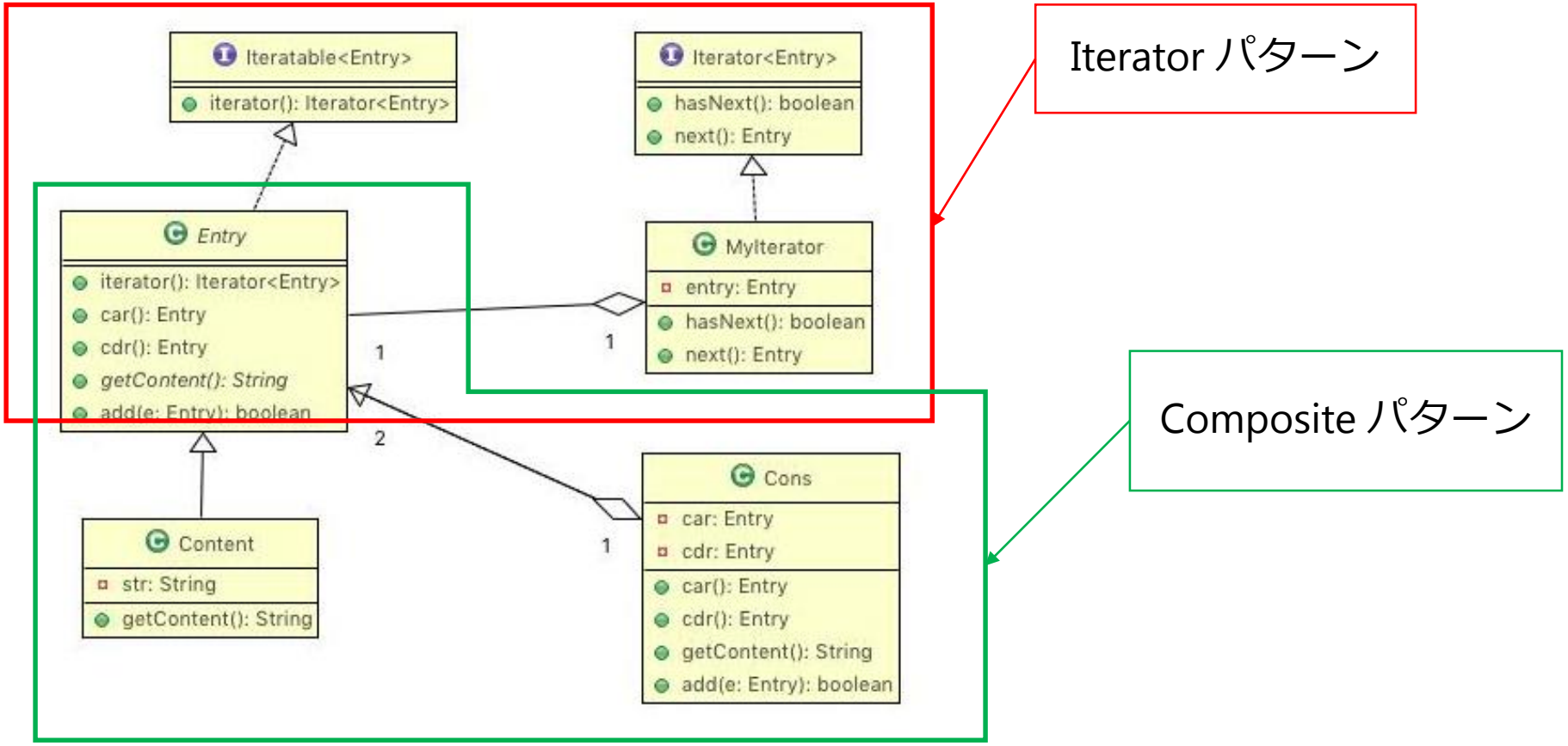
クラス図

クラス図

実装演習

1. デザインパターンとは
2. UMLについて
3. Iteratorパターン
4. Adapterパターン
5. 実装演習
6. 解説

実装演習（答え）


Iteratable<Entry>
iterator(): Iterator<Entry>
Iterator<Entry>
hasNext(): boolean
next(): Entry
Entry
iterator(): Iterator<Entry>
car(): Entry
cdr(): Entry
getContent(): String
add(e: Entry): boolean
MyIterator
entry: Entry
hasNext(): boolean
next(): Entry
1
1
2
Content
str: String
getContent(): String
1
Cons
car: Entry
cdr: Entry
car(): Entry
cdr(): Entry
getContent(): String
add(e: Entry): boolean
Iterator パターン
Composite パターン

実装演習（答え）

Entry.java

```
public abstract class Entry extends AbstractCollection<Entry>
  implements Iterable<Entry> {

  public abstract String getContent();

  public int size(){
    return 1;
  }

  public Entry car(){
    return this;
  }

  public Entry cdr(){
    return null;
  }

  public boolean add(Entry e){
    return false;
  }

  @Override
  public Iterator<Entry> iterator() {
    return new MyIterator(this);
  }

}
```

MyIterator.java

```
public class MyIterator implements Iterator<Entry> {

  private Entry entry;

  public MyIterator(Entry entry){
    this.entry = entry;
  }

  @Override
  public boolean hasNext() {
    if(entry == null) return false;
    return true;
  }

  @Override
  public Entry next() {
    Entry e = entry.car();
    this.entry = entry.cdr();
    return e;
  }

}
```

実装演習（答え）

Cons.java

```
public class Cons extends Entry {
  private Entry car;
  private Entry cdr;
  public Cons(){
    car = null;
    cdr = null;
  }

  public Cons(Entry car, Entry cdr){
    if(car == null) throw new NullPointerException();
    this.car = car;
    this.cdr = cdr;
  }

  public boolean add(Entry e){
    if(car == null){
    //this is cons cell whose size is 0
      car = e;
      return true;
    }else if(cdr == null){
      cdr = new Cons(e,null);
      return true;
    }else{
      return cdr.add(e);
    }
  }
```

```
 @Override
  public Entry car() {
    return car;
  }

  @Override
  public Entry cdr() {
    return cdr;
  }

  @Override
  public int size(){
    if(car == null){
      return 0;
    }else if(cdr == null){
      return 1;
    }else{
      return 1 + cdr.size();
    }
  }

  @Override
  public String getContent() {
    if((car == null) && (cdr == null)) return "[(),()]";
    if(cdr == null) return "[" + car.getContent() + ",()" + "]";
    return "[" + car.getContent() + "," + cdr.getContent() + "]";
  }
}
```