Design Patterns

Object-Oriented Programming Lecture 10

https://www.tutorialspoint.com/design_pattern/index.htm

https://www.journaldev.com/1827/java-design-patterns-example-tutorial

April 20, 2021

Radboud University

DESIGN PATTERNS

the point of design patterns

proven solutions to common design problems (in a standard format)

answering the questions:

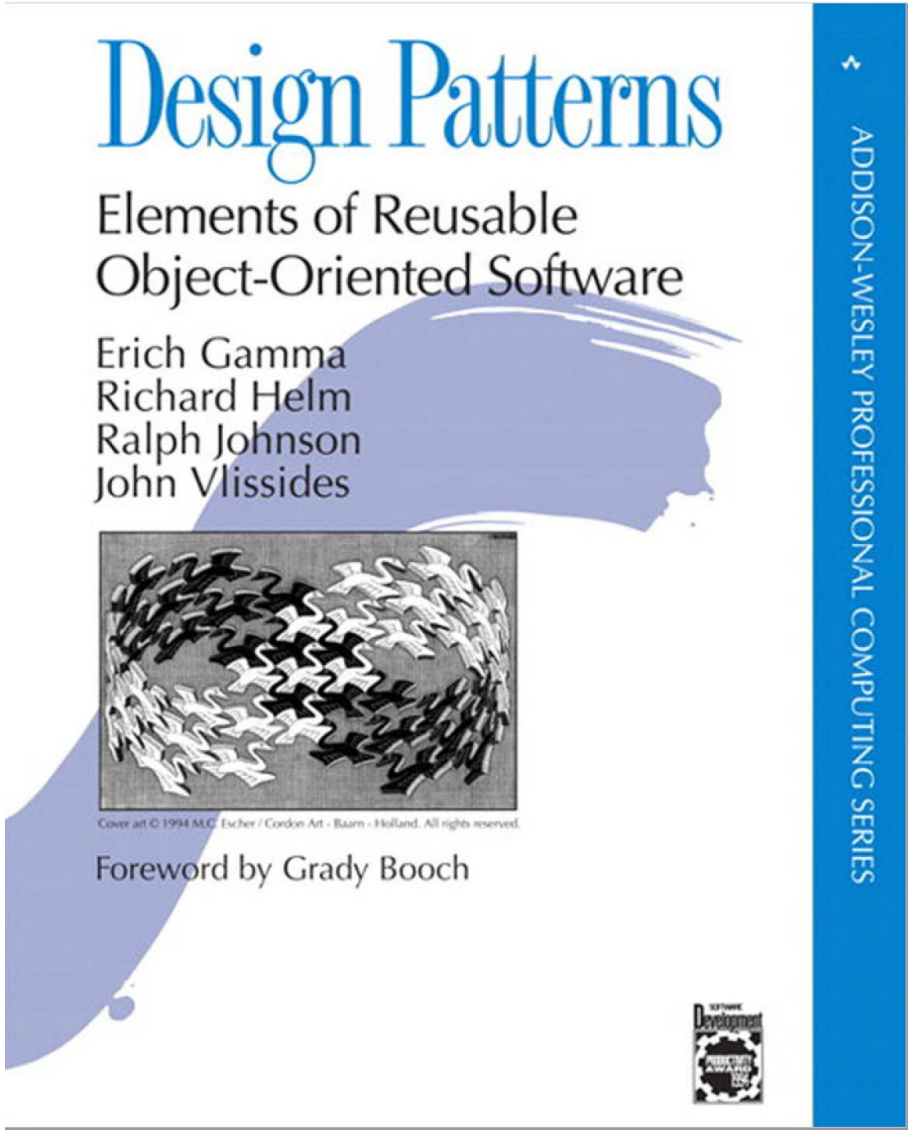
1) What are common features in good designs that are not in poor designs?

2) What are common issues in poor designs that are not in good designs?

first coined in building architecture by Christopher Alexander in 1979

applied to object oriented programming by the “gang of four”

Design patterns in OOP



pattern description

Each pattern has:

1) a short *name*

2) a brief description of the *context*
   - problem description, restrictions

3) a lengthy description of the *problem*

4) a prescription for the *solution*
   - often not programming language specific,
     same pattern works for Java, C#, C++, JavaScript, Python, ..

patterns in programming

patterns in programming are techniques to solve
frequently encountered design problems and programming tasks

solution by

- general class or interface
  - iterator, option, observer, ...
- scheme to define classes and relationships
  - Model-View-Controller, Strategy, ...
- seems vague, but is actually quite clear
  e.g. iterator is a pattern to traverse a container and access its elements

we have seen some patterns

you should be able to:
1) apply the shown patterns
2) recognize when to apply a pattern

we want only one object of a class

SINGLETON PATTERN

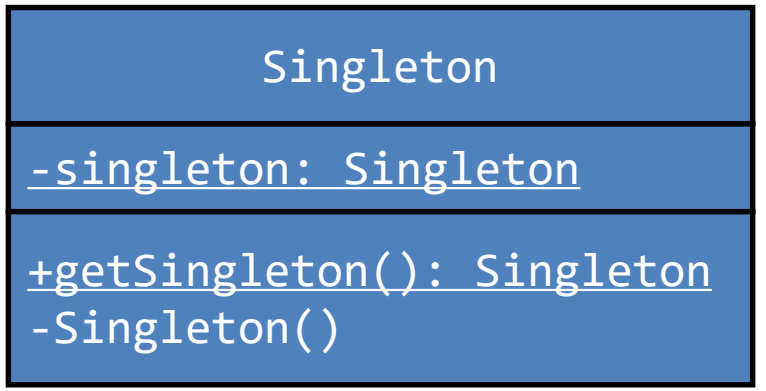
singleton pattern

context

- we want a maximum of one object of a class
  - e.g. JavaFX Application object, database, audio player, ..

solution

- make the constructor private
- add a static attribute to hold the (unique) object



use this as:

```
Singleton obj = Singleton.getSingleton();
```

since the constructor is private we cannot write:

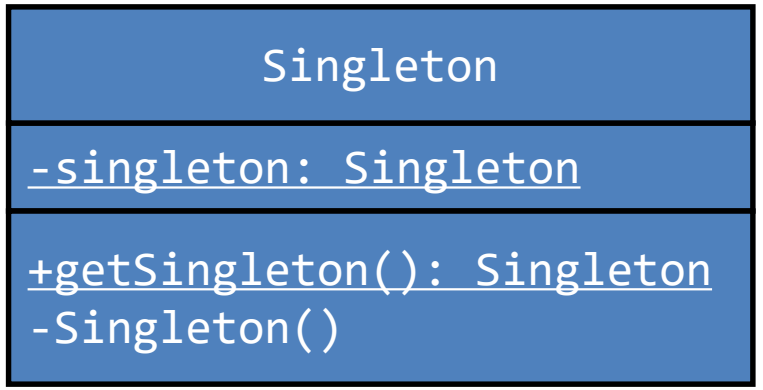
```
Singleton obj = new Singleton();
```

NOT POSSIBLE

eager Singleton

the object is made when the class is loaded in the JVM

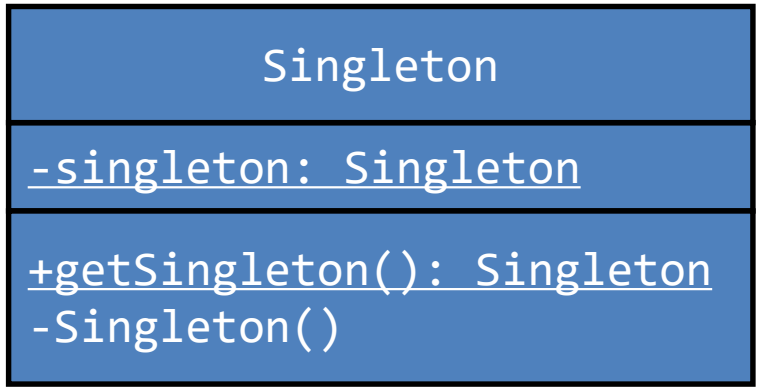
```
public class SingletonEager {
  private static final SingletonEager INSTANCE = new SingletonEager();

  private SingletonEager() {
    System.out.println("Object created eagerly");
  }

  public static SingletonEager getSingleton() {
    return INSTANCE;
  }
}
```



lazy Singleton

object is made when program gets first object from class

```
public class SingletonLazy {
    private static SingletonLazy instance = null;

    private SingletonLazy() {
        System.out.println("Object created lazy");
    }

    public static SingletonLazy getSingleton() {
        if (instance == null) {
            instance = new SingletonLazy();
        }
        return instance;
    }
}
```



using these singletons

```
public void run() {
    SingletonEager se1 = SingletonEager.getSingleton();
    SingletonEager se2 = SingletonEager.getSingleton();
    SingletonLazy  sl1 = SingletonLazy.getSingleton();
    SingletonLazy  sl2 = SingletonLazy.getSingleton();

    if (se1 == se2) {
      System.out.println ("Strict objects are equal.");
    }
    if (sl1 == sl2) {
      System.out.println ("Lazy objects are equal.");
    }
  }
```

```
Object created eagerly
Object created lazy
Strict objects are equal.
Lazy objects are equal.
```

(dynamic) change of behaviour

- for many different behaviours
- for dynamic change of behaviour

STRATEGY PATTERN

strategy pattern

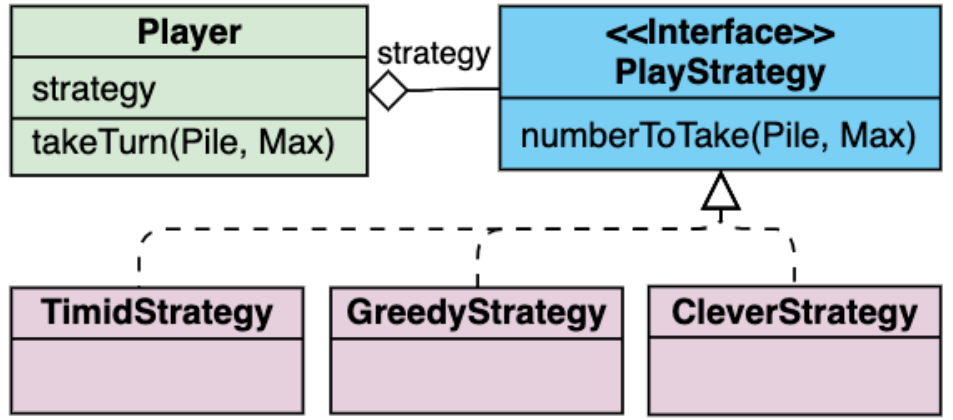
context

- choice of algorithm/functionality depends on client making the request
- clients can replace standard algorithms with custom version

solution

- define an interface (the strategy) that is an abstraction of the algorithm
- actual (concrete) strategy classes realize this interface type
- clients provide the instantiated concrete strategy class to the context class
- whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object
- e.g. `Collections.sort(Collection<T>, Comparator<T>)`

from lecture 3: player strategy




```
public class Player {
  private PlayStrategy strategy;
  public void takeTurn(Pile pile, int max){
    int sticks = strategy.numberToTake(pile, max);
    pile.remove(sticks);
  }

  public Player (PlayStrategy play) {
    strategy = play;
  }
}
```

```
interface PlayStrategy {
  int numberToTake(Pile pile, int max);
}
```

```
class TimidStrategy implements PlayStrategy {
  public int numberToTake(Pile p, int m) {
    return 1;
  }
}
```

(dynamic) addition of behaviour

- multiple additions to same object

DECORATOR PATTERN

decorator pattern

context

- to enhance the behaviour of a *component classes*
- a decorated component can be used the same way as the undecorated one
- there are too many variations for separate subclasses

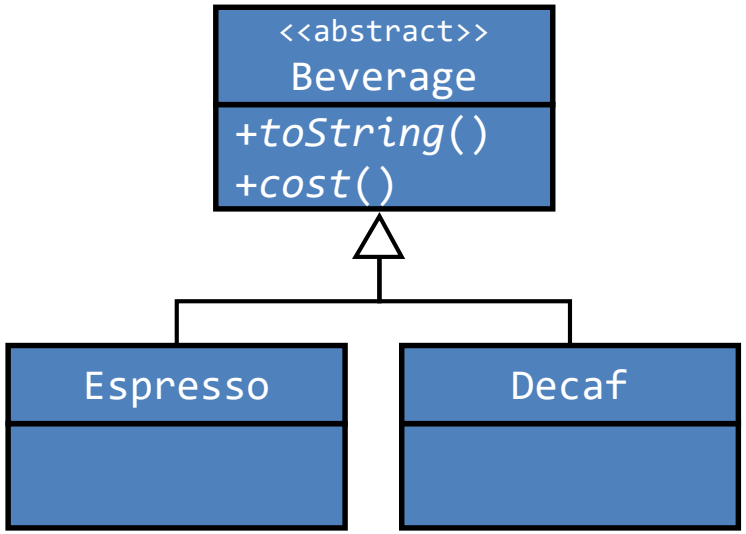
solution

- define an interface type (or an abstract class) that is an abstraction of the component class
- actual component classes implement this interface type
- decorator classes also implement this interface type
- decorator object manages the component object it decorates

beverage decoration

classes for a coffee ordering system

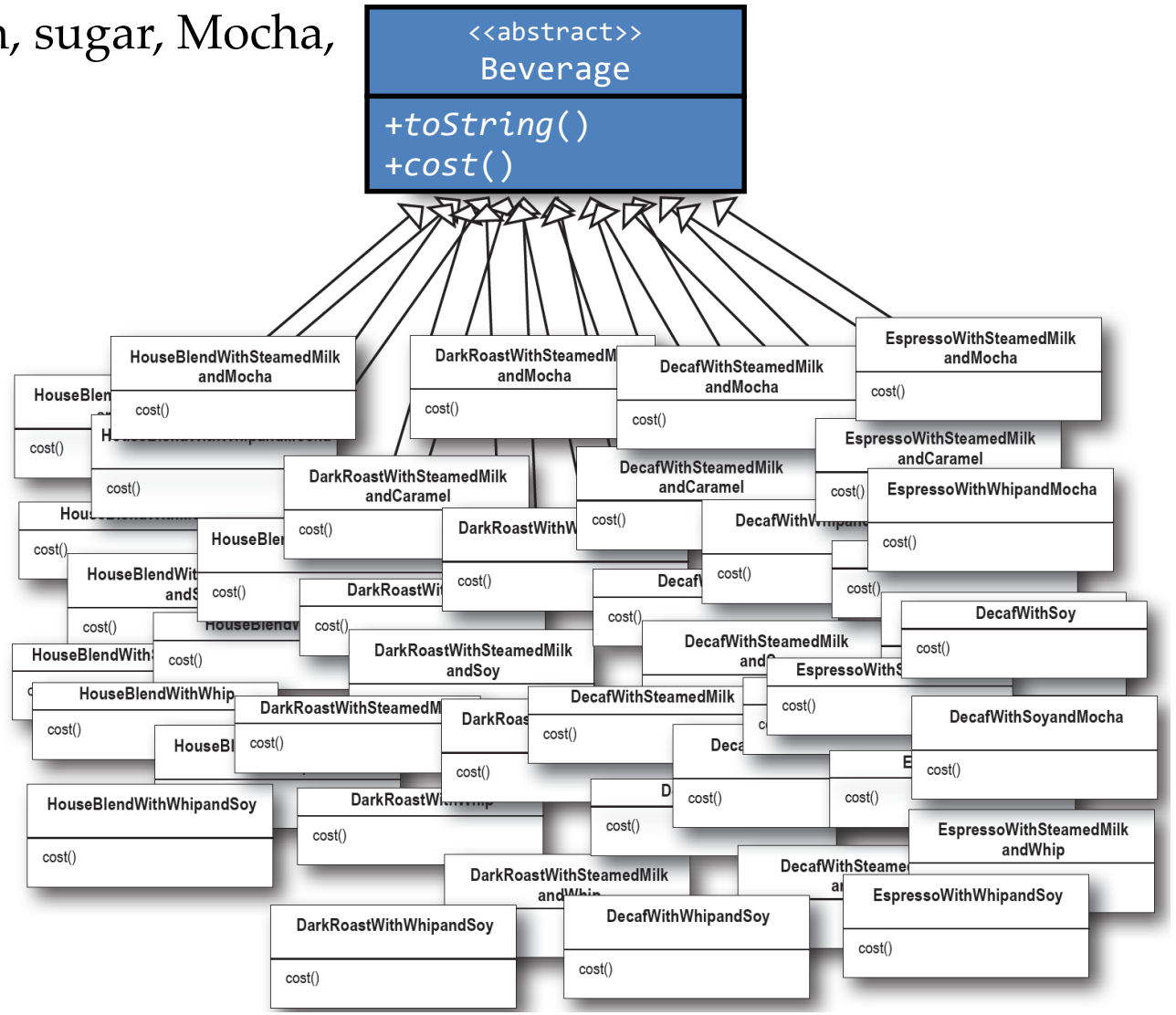
- beverages: Espresso, Decaf, DarkRoast, …



class explosion

~~we~~ some people need additions to those beverages

- milk, whipped cream, sugar, Mocha,



using attributes

making subclass for all variants is too much work and completely unmaintainable

we can add boolean attributes for each component

- `hasMilk`, `hasWhip`, `hasSugar`, `hasMocha`, ..
- requires many setters and getters

we can also add a single attribute

- `List<Addition> additions = ..`
- `Addition` is a tailor made enumeration type:
  `enum Addition {Milk, Sugar, Whip, Mocha;}`
- we have to scan this list over and over again; not class based

both approaches require a change of existing code for each new beverage addition

- we prefer to leave existing code untouched and add new classes

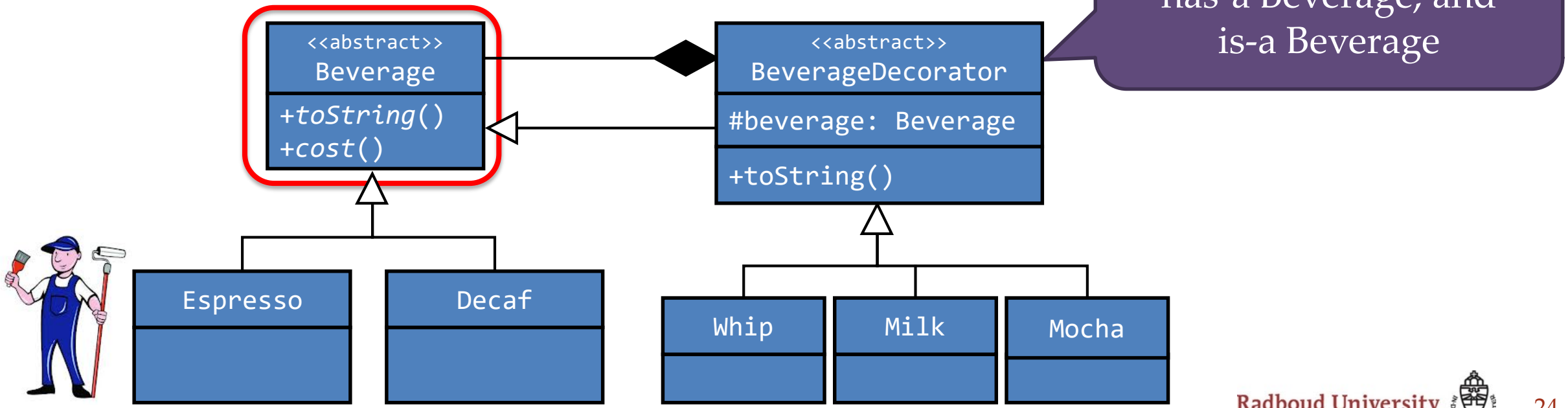
the decorator pattern offers a class based solution

decorator pattern for beverage: base class

```
public abstract class Beverage {
  @Override
  abstract String toString();
  abstract double cost();
}
```

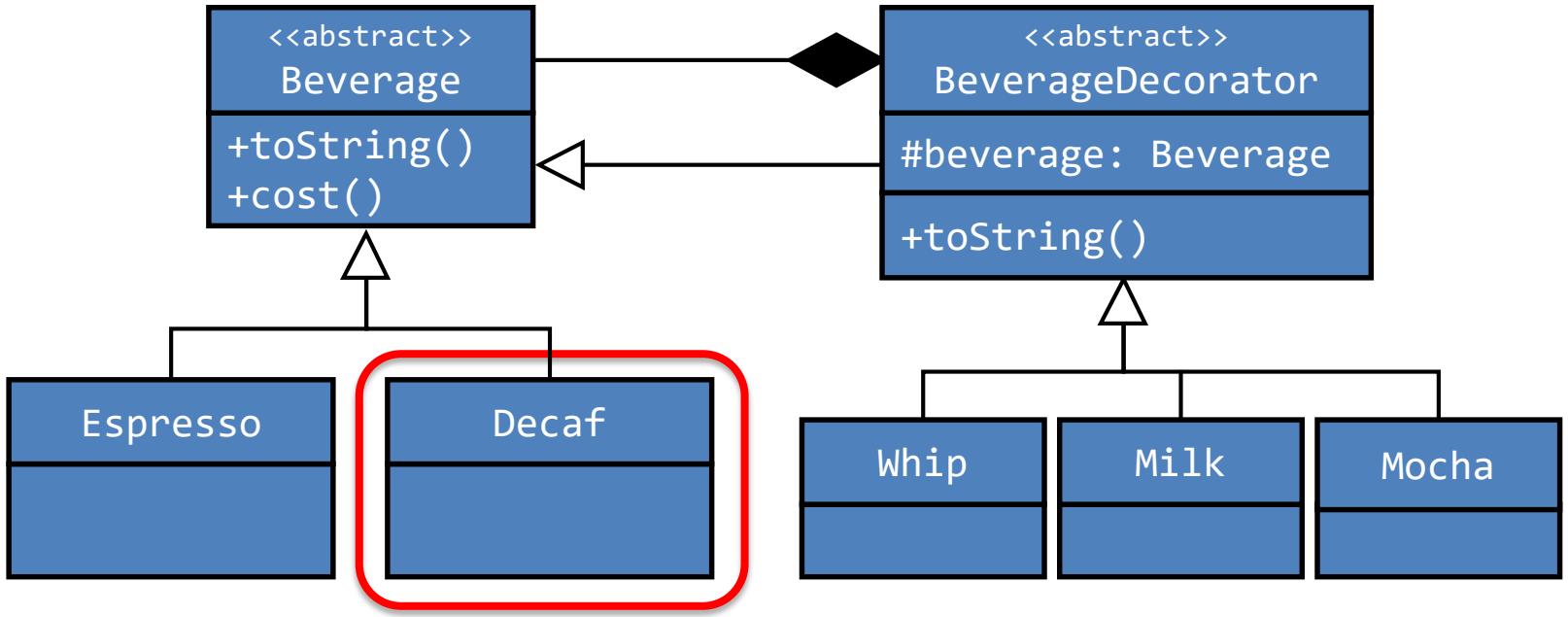
require a toString implementation by making method abstract

the decorator class has-a Beverage, and is-a Beverage



beverage: subclass for Decaf

```
public class Decaf extends Beverage {
  @Override
  public String toString() { return "Decaf"; }

  @Override
  public double cost() { return 1.50; }
}
```



beverage decorator

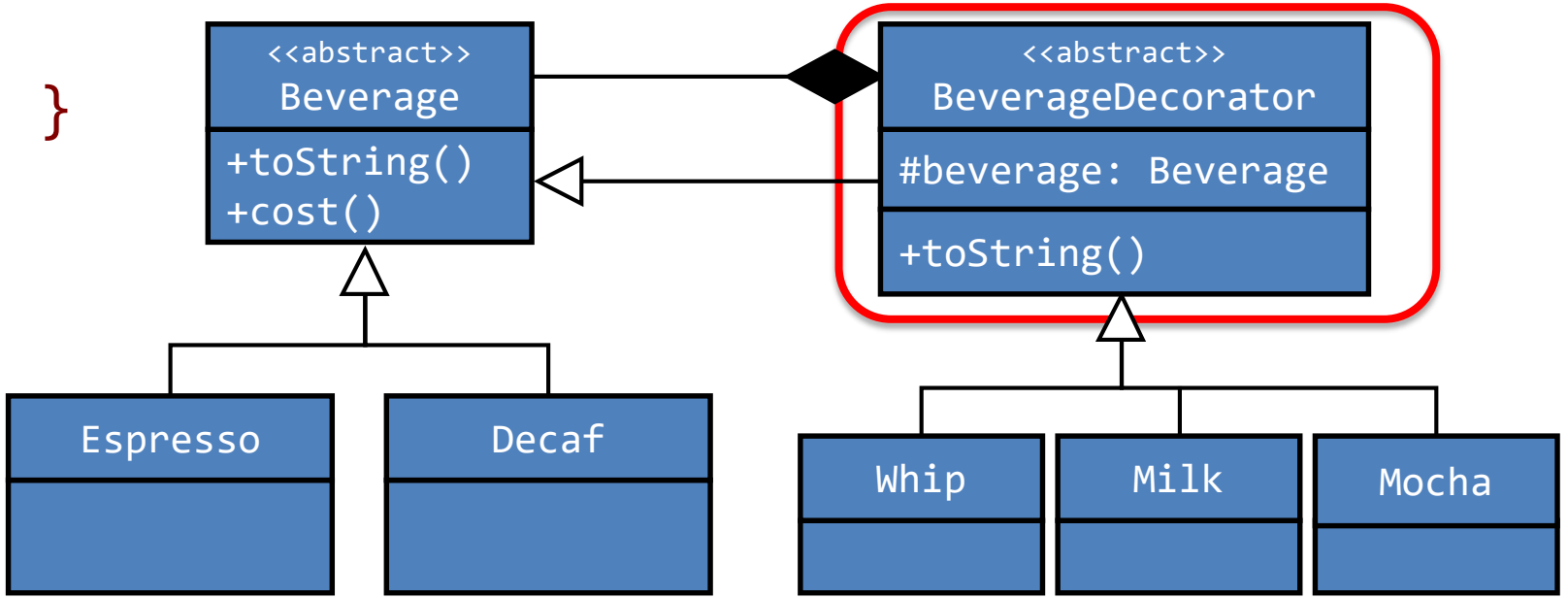
```
public abstract class BeverageDecorator extends Beverage{
   protected final Beverage beverage;
   public BeverageDecorator(Beverage beverage) {
       this.beverage = beverage;
   }
   @Override
   public double cost() { return beverage.cost(); }
}
```

is-a: I am a Beverage

has-a: decorated object

delegate behaviour

abstract since toString not implemented



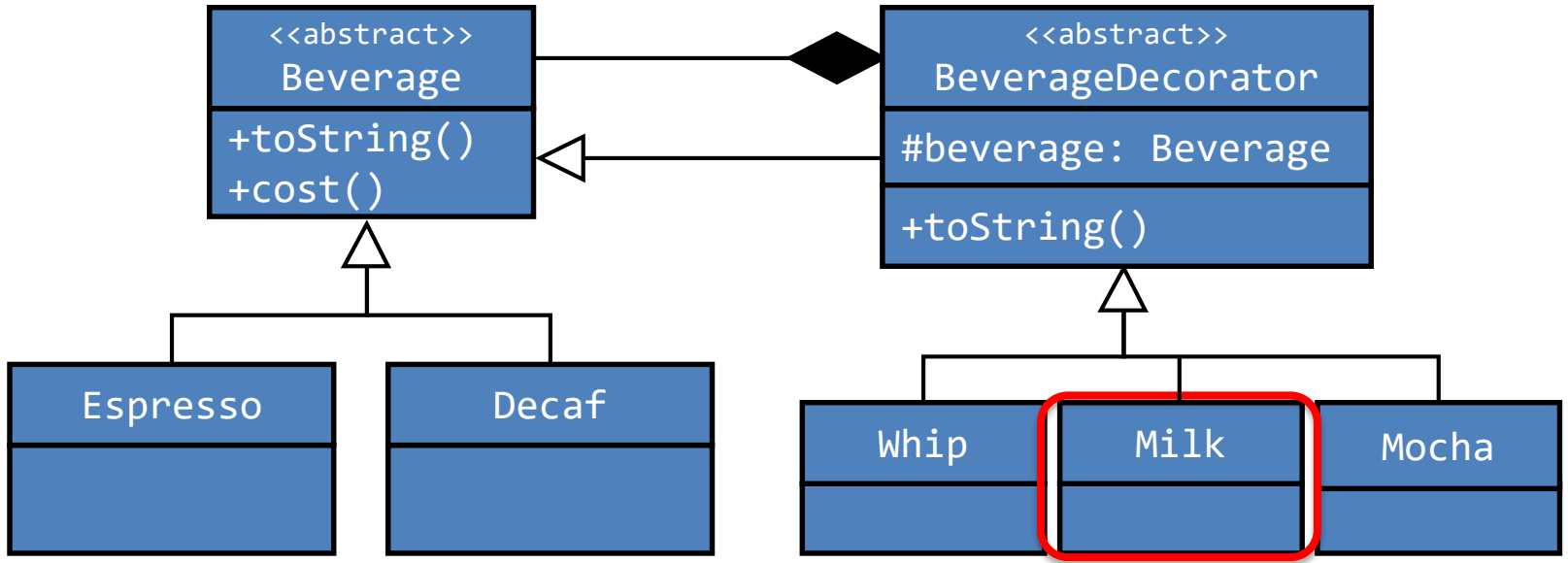
Milk, a beverage decorator

```
public class Milk extends BeverageDecorator {
   public Milk( Beverage beverage ) {
      super( beverage );
   }
   @Override
   public String toString() {
      return beverage.toString() + ", milk";
   }
   @Override
   public double cost() {
      return super.cost() + 0.15;
   }
}
```

decorated object

extend behaviour

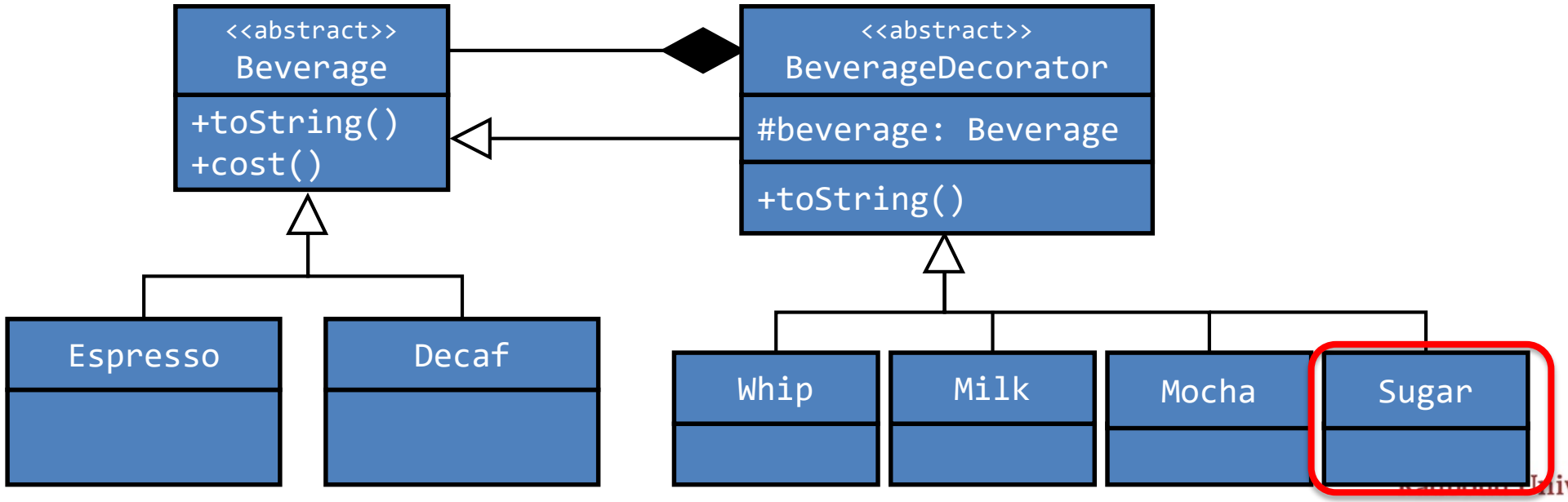
extend behaviour



Sugar, another beverage decorator

```
public class Sugar extends BeverageDecorator {
  public Sugar( Beverage beverage ) {
    super( beverage );
  }
  @Override
  public String toString() {
    return beverage.toString() + ", sugar";
  }
}
```

no new cost method

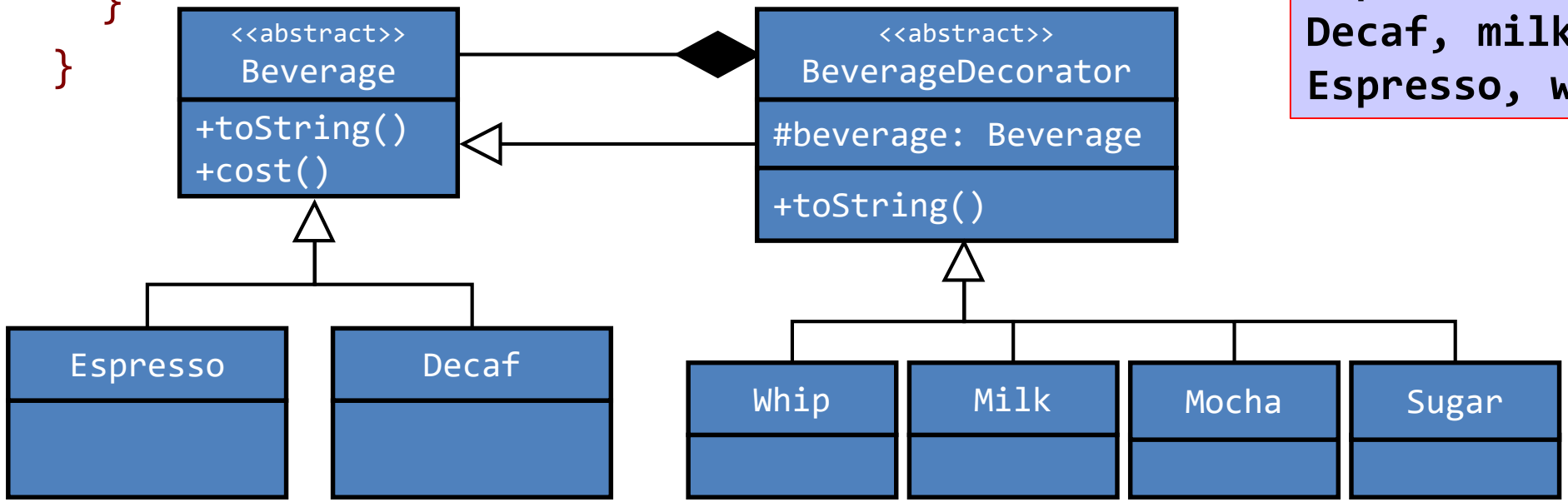


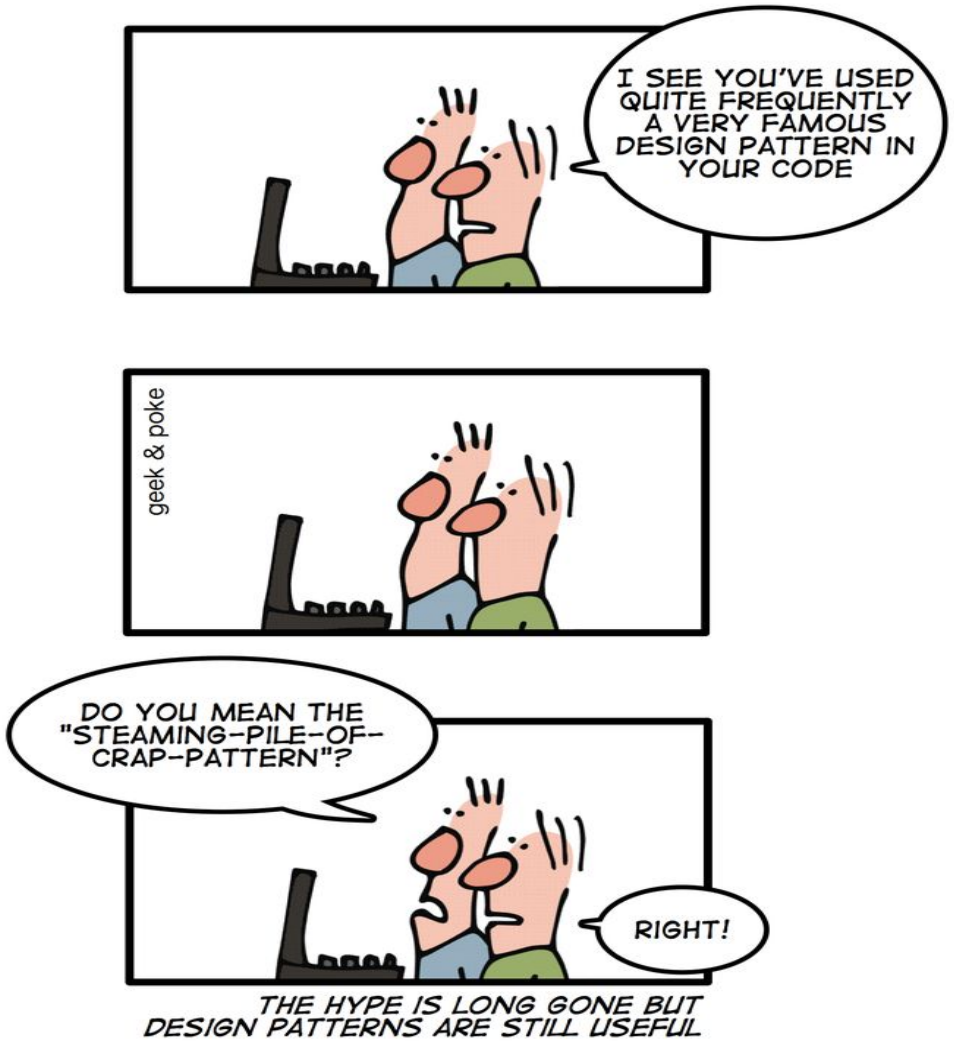
using decorators

```
public void run() {
   List<Beverage> beverages = new LinkedList<>();
   beverages.add(new Espresso());
   beverages.add(new Sugar( new Milk( new Decaf())));
   beverages.add(new Whip( new Whip( new Espresso())));
   for ( Beverage b: beverages ) {
      System.out.println( b + " $" + b.cost());
   }
}
```

output:

```
Espresso $1.99
Decaf, milk, sugar $1.65
Espresso, whip, whip $2.49
```




I SEE YOU'VE USED QUITE FREQUENTLY A VERY FAMOUS DESIGN PATTERN IN YOUR CODE
geek & poke
DO YOU MEAN THE "STEAMING-PILE-OF-CRAP-PATTERN"?
RIGHT!
THE HYPE IS LONG GONE BUT DESIGN PATTERNS ARE STILL USEFUL

https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

Null pointers were invented by Tony Hoare, inventor of QuickSort.
In 2009 he called it

THE BILLION DOLLAR MISTAKE

the problem

```
public static <E extends Comparable<E>> E max0(Collection<E> c) {
  E result = null;
  for (E e : c)
    if (result == null || e.compareTo(result) > 0)
      result = e;
  return result;
}
private List<String> list0 = Arrays.asList();
private List<String> list1 = Arrays.asList("Allice", "Carol", "Bob");
public void run() {
  String m00 = max0(list0);
  String m01 = max0(list1);
  System.out.println("m00 = " + m00 + " m01 = " + m01);
}
```

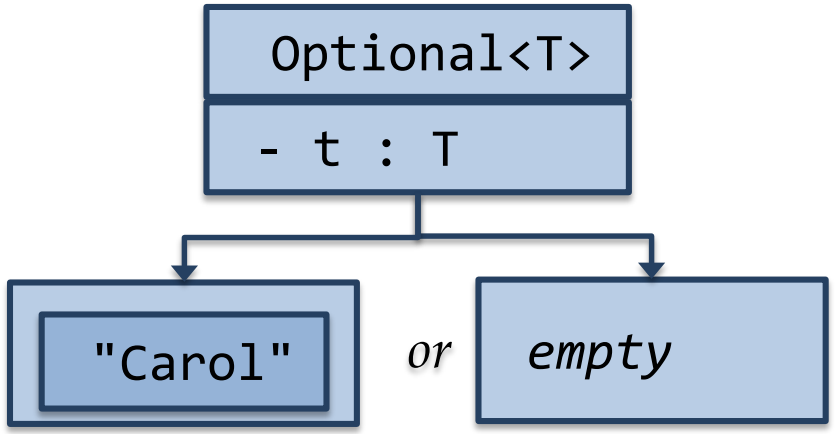
returns null for empty collection

```
m00 = null m01 = Carol
```

using exceptions + try & catch

```
public static <E extends Comparable<E>> E max1(Collection<E> c) {
  if (c.isEmpty())
    throw new IllegalArgumentException("max of empty collection");
  E result = null;
  for (E e : c)
    if (result == null || e.compareTo(result) > 0)
      result = e;
  return result;
}
public static void run() {
  String m10;
  try { m10 = max1(list0);
  } catch (IllegalArgumentException e) { m10 = "UNKNOWN"; }
  String m11;
  try { m11 = max1(list1);
  } catch (IllegalArgumentException e) { m11 = "UNKNOWN"; }
  System.out.println("m10 = " + m10 + " m11 = " + m11);
```

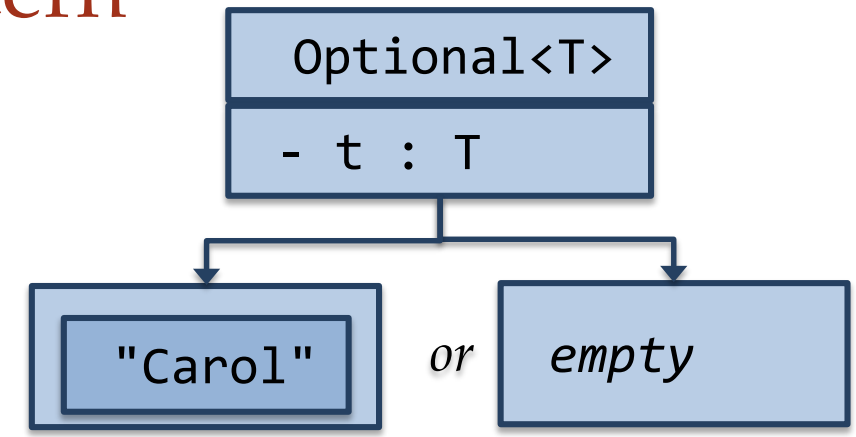
```
m10 = UNKNOWN m11 = Carol
```



Fixing the Billion Dollar Mistake

OPTIONAL PATTERN

the optional pattern



context

- `null` as a result is the billion dollar mistake

solution

- wrap value `T` in an `Optional` object (of type `Optional<T>`); the `Optional` object is always there, even when the `T` value is `null`

`Optional.empty()`	returns an empty Optional object
`Optional.of(T o)`	returns an Optional object
`Optional.ofNullable(T o)`	the Optional object or an empty Optional
`boolean isPresent()`	
`T get ()`	returns object if available (otherwise throws `NoSuchElementException`)
`T orElse(T other)`	

using Optional

```
public static <E extends Comparable<E>> Optional<E>
                                        max2(Collection<E> c) {
  if (c.isEmpty())
    return Optional.empty();
  E result = null;
  for (E e : c)
    if (result == null || e.compareTo(result) > 0)
      result = e;
  return Optional.of(result);
}
public static void run() {
  String m20 = max2(list0).orElse("UNKNOWN");
  Optional<String> o21 = max2(list1);
  String m21 = o21.isPresent() ? o21.get() : "UNKNOWN";
  System.out.println("m20 = " + m20 + " m21 = " + m21);
```

```
m20 = UNKNOWN m21 = Carol
```

the compiler complains if you forget this; Optional is not a String

more concise

the verbose way

using Optional take 2

```
public static <E extends Comparable<E>> Optional<E>
                                        max3(Collection<E> c) {
  E result = null;
  for (E e : c)
    if (result == null || e.compareTo(result) > 0)
      result = e;
  return Optional.ofNullable(result);
}
public static void run() {
  String m30 = max3(list0).orElse("UNKNOWN");
  String m31 = max3(list1).orElse("UNKNOWN");
  System.out.println("m30 = " + m30 + " m31 = " + m31);
```

the concise way

```
m30 = UNKNOWN m31 = Carol
```

alternative actions for `Optional`

instead of an alternative value we can specify an action or exception

```
T orElseGet(Supplier<T> other)
```

example

```
System.out.println("orElseGet: " + max3(list0).orElseGet(() -> {
  System.out.println("supplier called");
  return "default";
}));
```

```
supplier called
orElseGet: default
```

only invoked when the result is needed

problems with `null`

1. the value `null` is no value
2. subverts types
3. is sloppy
4. makes poor APIs
5. exacerbates poor language decisions
6. difficult to avoid null pointer exceptions
7. is non-composable

`Optional` is a partial solution (we still have `null` and exceptions in Java)

a better language design replaces `null`-based semantics by `Optional`

- it is too late for Java, it would require a massive update of all existing Java code
- done in Haskell, ML, F#, Scala, Kotlin, Rust, ...

double call-back

VISITOR PATTERN

Visitor patterns

context

- apply distinct, unrelated operations to a collection of objects of different types
- we don't want to change the classes of the objects on which new operations operate
- we don't want to have to query the type of each object and cast the pointer to the correct type before performing the desired operation

solution

- define a **visitor** that can perform the operation to objects
  - one visitor class for each operation
- inspected classes implement one method to apply/accept the visitor
  - dynamic binding is necessary to select correct method for accepting the visitor,

naive OO solution

following the OO paradigm naively:

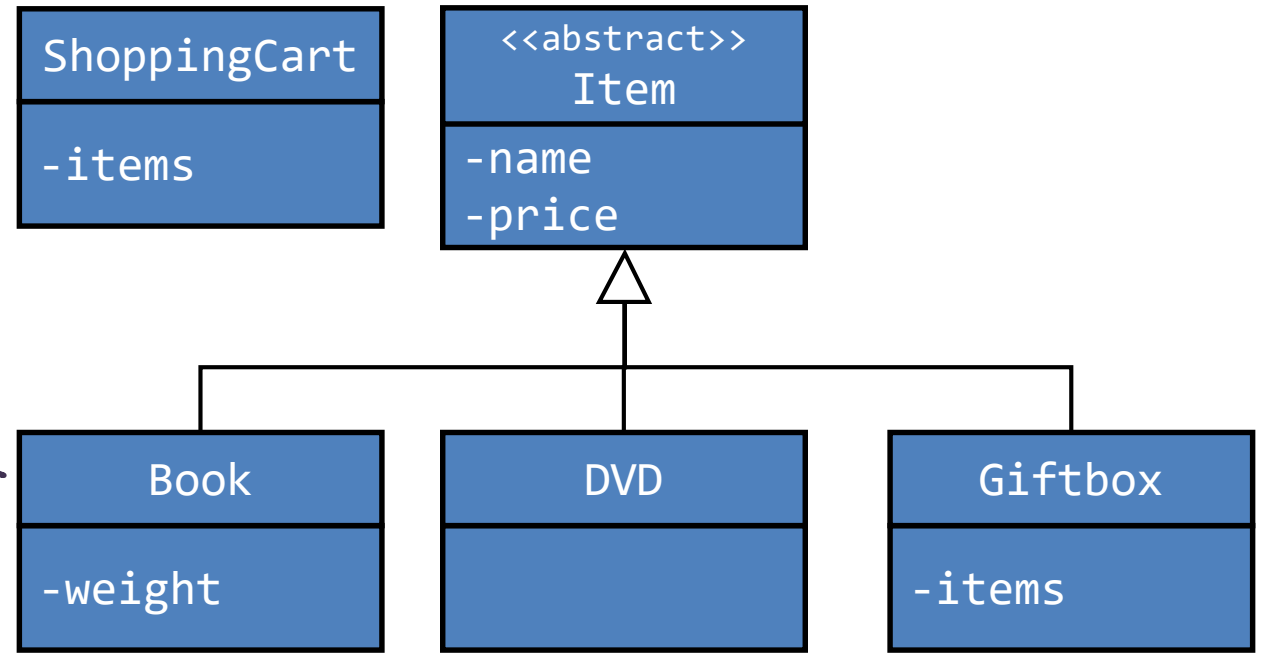
- add one method for each behaviour to all classes in the collection
  - often methods for atypical behaviour of the object
  - the methods reflecting a specific operation are distributed over the classes involved

solution

- add one method that allows ‘any visitor’ to inspect/handle the object
- define/implement visitor for each specific behaviour
  - we can have as many visitors as needed

shop selling Books, DVDs and Gift-boxes using a shopping-cart

we omit some attributes and methods for brevity



class Item, Book, DVD, Giftbox

```
public abstract class Item {
  private final String name;
  private final double price;
  public Item(String name, double price) {
    this.name  = name;
    this.price = price;
  }
  // and other methods
}
```

```
public class DVD extends Item {
   private final double duration;
   private final int discs;

   public DVD(double d, int c, String n, double p) {
      super(n, p);
      this.duration = d; this.discs = c;
   }
   // and other methods
}
```

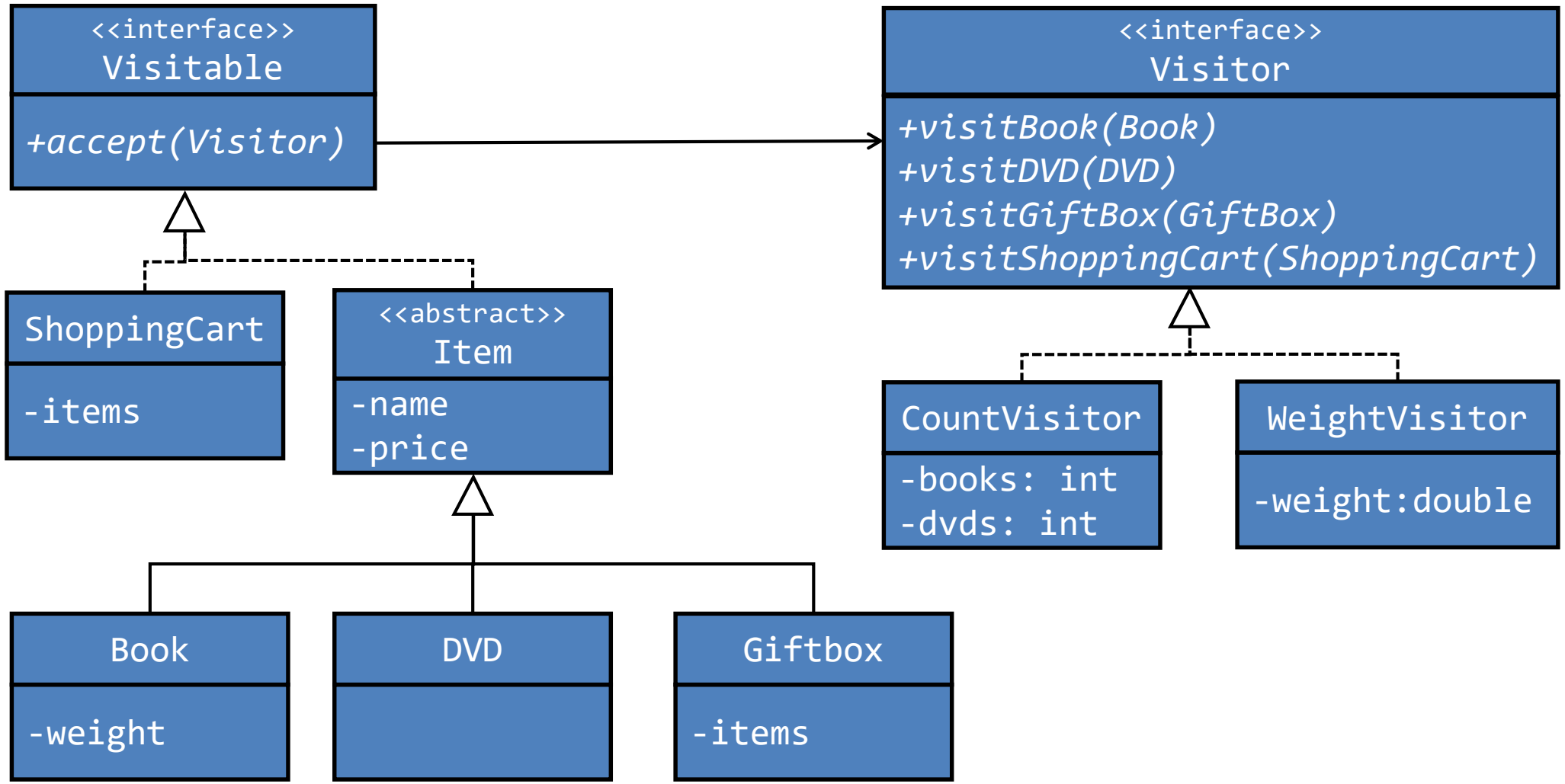
```
public class Book extends Item {
  private final double weight;
  private final String author;

  public Book(double w, String a, String t, double p) {
    super(t, p);
    this.weight = w; this.author = a;
  }
  // and other methods
}
```

```
public class GiftBox extends Item {
   private final List<Item> items;

   public GiftBox(String name, Item ... items) {
      super(name, getTotalPrice(items));
      this.items = Arrays.asList(items);
   }
   // and other methods
}
```

class ShoppingCart

```
public class ShoppingCart {
  private List<Item> items;
  public ShoppingCart() {
    items = new LinkedList<>();
  }

  public void add(Item item) { items.add(item); }
  public List<Item> getItems() { return items; }
  @Override
  public String toString() {
    StringBuilder out = new StringBuilder();
    items.forEach(item -> out.append(item).append("\n"));
    return out.toString();
  }
}
```

making classes visitable



interface Visitable and Visitor

```
public interface Visitable {
    void accept(Visitor visitor);
}

public interface Visitor {
  void visitBook( Book book );
  void visitDVD( DVD dvd );
  void visitGiftBox( GiftBox box );
  void visitShoppingCart( ShoppingCart cart );
}
```

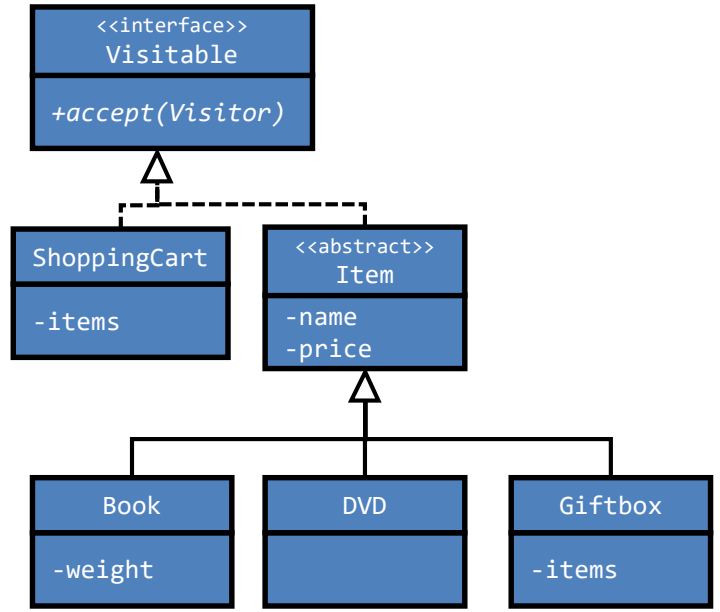
to admit a visitor

making Book, DVD, Giftbox and ShoppingCart visitable

```
public abstract class Item implements Visitable {
  // Attributes, Constructor and other methods
}
```

```
public class Book extends Item {
  // Attributes
  // Constructor and other methods
  @Override
  public void accept(Visitor v) {
     v.visitBook( this );
  }
}
```

```
public class DVD extends Item {
  // Attributes
  // Constructor and other methods
  @Override
  public void accept(Visitor v) {
     v.visitDVD( this );
  }
}
```




```
public class GiftBox extends Item {
  // Attributes
  // Constructor and other methods
  @Override
  public void accept(Visitor v) {
     v.visitGiftBox( this );
  }
}
```

```
public class ShoppingCart implements Visitable {
  // Attributes
  // Constructor and other methods
  @Override
  public void accept(Visitor v) {
     v.visitShoppingCart( this );
  }
}
```

count visitor: counting number of books and DVDs

```
public class CountVisitor implements Visitor {
  private int books, dvds;

  @Override
  public void visitBook(Book book) {
    books += 1;
  }

  @Override
  public void visitDVD(DVD dvd) {
    dvds += 1;
  }

  @Override
  public void visitGiftBox(GiftBox box) {
    for (Item item : box.getItems()) {
      item.accept(this);
    };
  }

  @Override
  public void visitShoppingCart(ShoppingCart cart) {
    for (Item item : cart.getItems()) {
      item.accept(this);
    };
   }
```

```
  public int getBooks() {
    return books;
  }

  public int getDVDs() {
    return dvds;
  }

  public String toString() {
    return "books = " + books +
           ", dvds = " + dvds;
  }

}
```

application

```
private static void run() {
    ShoppingCart cart = new ShoppingCart();
    cart.add(new Book(0.8, "Douglas Hofstadter", "Godel, Escher, Bach", 17.99));
    cart.add(new Book(0.1, "J. K. Rowling", "Sorcerer's Stone", 12.35));
    cart.add(new DVD(2.28, 1, "Spectre", 14.99));
    cart.add(new GiftBox("Hitchhiker's Guide",
            new Book(0.5, "Adams", "The Hitchhiker's Guide to the Galaxy", 14.99),
            new Book(0.5, "Adams", "The Restaurant at the End of the Universe", 14.99)));

    System.out.println(cart);
    CountVisitor count = new CountVisitor();
    cart.accept(count);
    System.out.println(count);
}
```

```
Book: Douglas Hofstadter Godel, Escher, Bach 17,99 euro
Book: J. K. Rowling Sorcerer's Stone 12,35 euro
DVD: Spectre 14,99 euro, 1 discs
Hitchhiker's Guide 29,98 euro

books = 4, dvds = 1
```

counting without a visitor: a single method

```
private static void countBooksAndDVDs(ShoppingCart cart) {
  int books = 0;
  int dvds  = 0;
  for (Item item: cart.getItems()) {
    if (item.getClass() == Book.class) {
      books += 1;
    } else if (item.getClass() == DVD.class)
      dvds += 1;
    } else if (item.getClass() == GiftBox.class) {
      GiftBox box = (GiftBox) item;
      for (Item subItem: box.getItems()) {
        if (subItem.getClass() == Book.class) {
          books += 1;
        } else if (subItem.getClass() == DVD.class) {
          dvds += 1;
        } else if (subItem.getClass() == GiftBox.class) {
          throw new IllegalArgumentException("No gift box inside another gift box please");
        }
      }
    }
  }
  System.out.println("Found " + books + " books, and " + dvds + " DVDs.");
}
```

`instanceof` or `getClass` (slide from Tutorial 6)

Beware of **`instanceof`** operator or **`getClass`** method.

Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself [Scott Meyers]

```
public abstract class Animal {

}

public class Cat extends Animal {
    public String meow() {
        return "meow, meow";
    }
}

public class Dog extends Animal {
    public String bark() {
        return "woof, woof";
    }
}
```


```
public class BadInstanceOf {
    public static void makeSound( Animal animal ) {
        if (animal instanceof Cat) {
            Cat cat = (Cat) animal;
            System.out.println( cat.meow() );
        } else if (animal instanceof Dog) {
            Dog dog = (Dog) animal;
            System.out.println( dog.bark() );
        }
    }
}
```

`instanceof` or `getClass` (slide from Tutorial 6)

Use *polymorphism*

- *Only possible if you can adjust the classes*

```
public class Animal {
    public String makeSound () {
        return "<silence>";
    }
}

public class Cat extends Animal {
    @Override
    public String makeSound() {
        return "meow, meow";
    }
}

public class Dog extends Animal {
    @Override
    public String makeSound() {
        return "woof, woof";
    }
}
```

```
public class GoodPolymorphism {
    public static void makeSound( Animal animal ) {
        System.out.println( animal.makeSound() );
    }
}
```

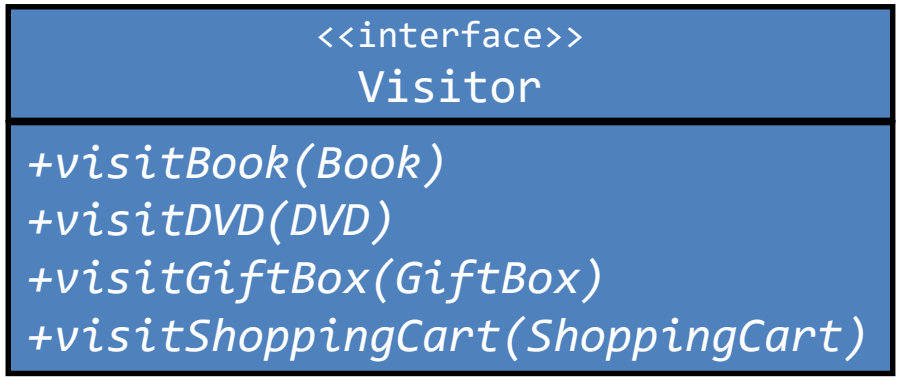
overloaded visitor

Overloading: **different** methods having the same name (but different signatures)

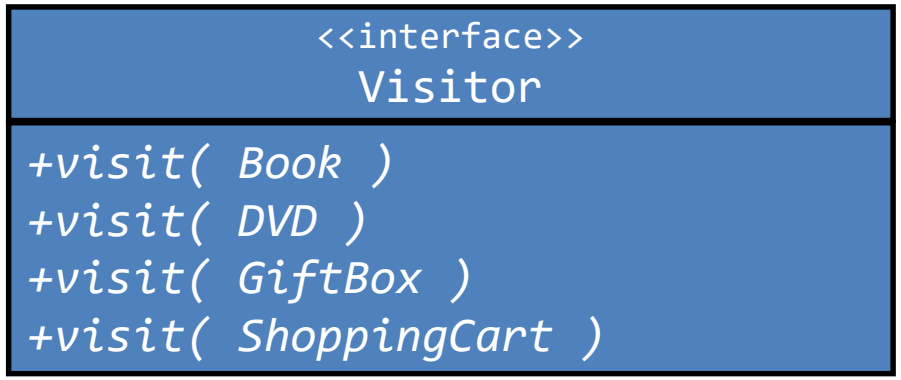
- related to compile-time (or static) polymorphism.

Overloading allows us to use one name for all methods in the `Visitor` interface.

Instead of:



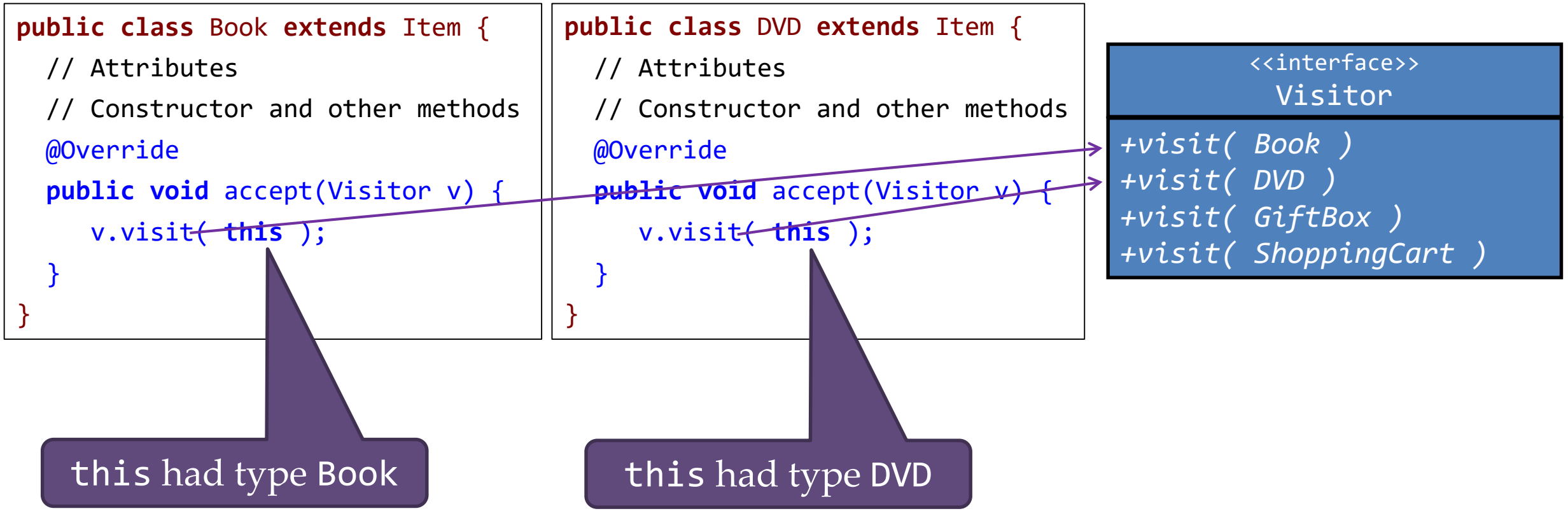
we define




```
public interface Visitor {
  void visit( Book book );
  void visit( DVD dvd );
  void visit( GiftBox box );
  void visit( ShoppingCart cart );
}
```

overloaded visitor (II)

adjustments to the visitables are small. E.g. for Book and DVD:



double call back

```
public class Book extends Item {
  protected final double weight;
  protected final String author;
  public void accept(Visitor v) {
    v.visit(this);
  }
```

the static type of this: correct binding

```
public class CountVisitor implements Visitor {
  private int books = 0;
  private int dvds  = 0;
  public void visit(Book book) {
    books += 1;
  }
```

Java selects this method at compiletime

dynamic binding, even for Item

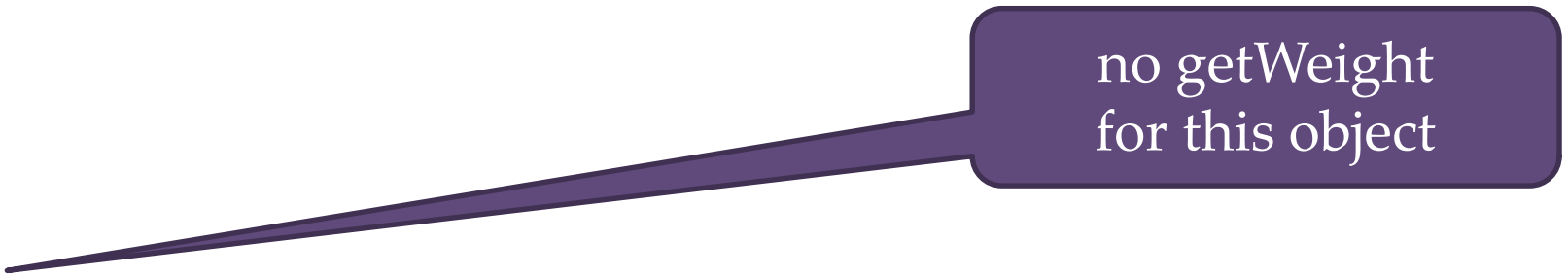
```
  Item book = new Book (..);
  book.accept(count);
```

fails (at compile time

instead of `count.visit(book)`

Weight Visitor, some items have `getWeight()`

```
public class WeightVisitor implements Visitor {
  private double totalWeight = 0;
  @Override
  public void visit(Book book) {
    totalWeight += book.getWeight();
  }
  @Override
  public void visit(DVD dvd) {
    totalWeight += 0.05 + dvd.getDiscs() * 0.026;
  }
  @Override
  public void visit(GiftBox box) {
    totalWeight += 0.1;
    box.getItems().forEach( item -> item.accept(this));
  }
  @Override
  public void visit(ShoppingCart cart) {
    cart.getItems().forEach( item -> item.accept(this));
  }
  public String toString() {
    return "Total weight = " + totalWeight + "Kg";
  }
}
```



application

```
Book: D. Hofstadter, Godel, Escher, Bach 17.99 euro
Book: J.K. Rowling, Sorcerer's Stone 12.35 euro
DVD: Spectre, 1 discs, 14.99 euro
Total weight = 0.976Kg
Price: 45.33 euro
```

```
private static void run() {
  ShoppingCart cart = new ShoppingCart();
  cart.add(new Book(0.8, "D. Hofstadter", "Godel, Escher, Bach", 17.99));
  cart.add(new Book(0.1,"J.K. Rowling", "Sorcerer's Stone", 12.35));
  cart.add(new DVD(2.28, 1, "Spectre", 14.99));
  System.out.println(cart);
  WeightVisitor weight = new WeightVisitor();
  cart.accept(weight);
  System.out.println(weight);
  PriceVisitor price = new PriceVisitor();
  cart.accept(price);
  System.out.println("Price: " + price.getPrice() + " euro");
}
```

patterns

software **patterns**:
descriptions of recurring problems with tried and tested solutions

- very useful
- there are very many patterns, study them and how to use them to be a better programmer
- in this course we stick to the patterns introduced (including previous lectures!)

do not overuse patterns

- KISS: use a simple solution if it works,
  do not introduce patterns just because you can

Lecture 11: Streams