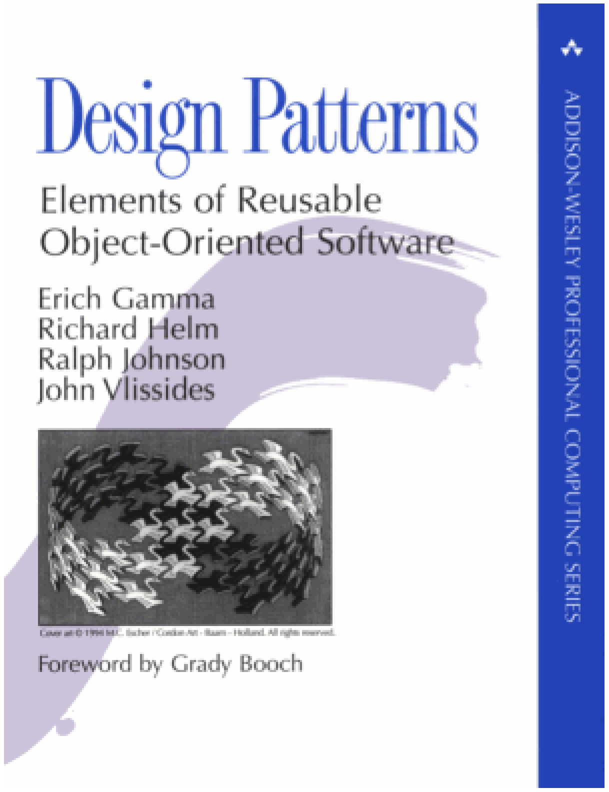
Software Development *Design patterns*

Koen Pelsmaekers

Unit Informatie (GT 03.14.05)

email: koen.pelsmaekers@kuleuven.be

The bible of Design Patterns [GoF]


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns, Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1993.

Design Pattern Concept

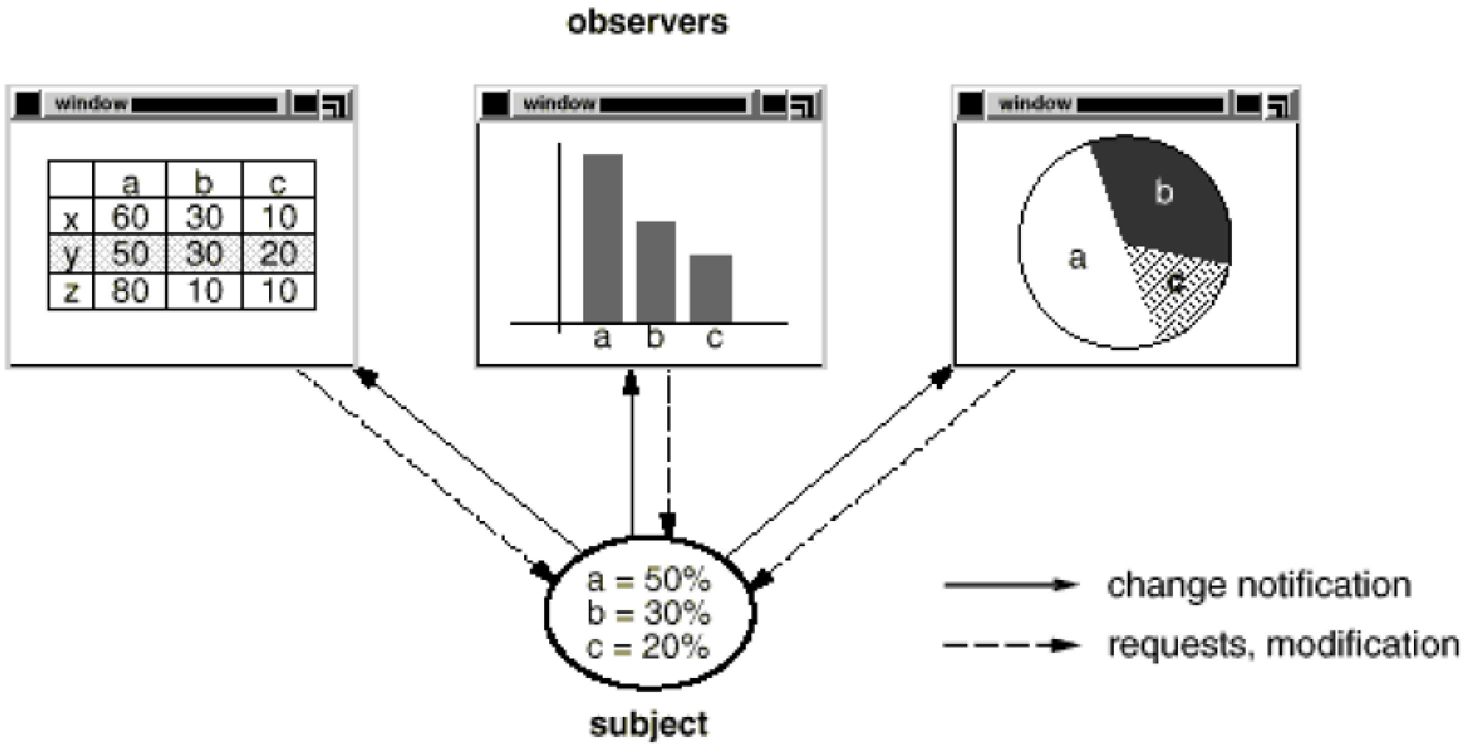
- Design Pattern
  - Architectural patterns (Christopher Alexander)
  - A general, reusable solution to a commonly occurring problem within a given context (in software design)
  - Definition of relationships and interactions between classes or objects, to solve a certain kind of recurring problem
  - "documented common sense", "encapsulate what changes", "isolate what varies"
- Each pattern has
  - short name
  - brief description of the context
  - lengthy description of the problem
  - prescription for the solution

Design patterns categories

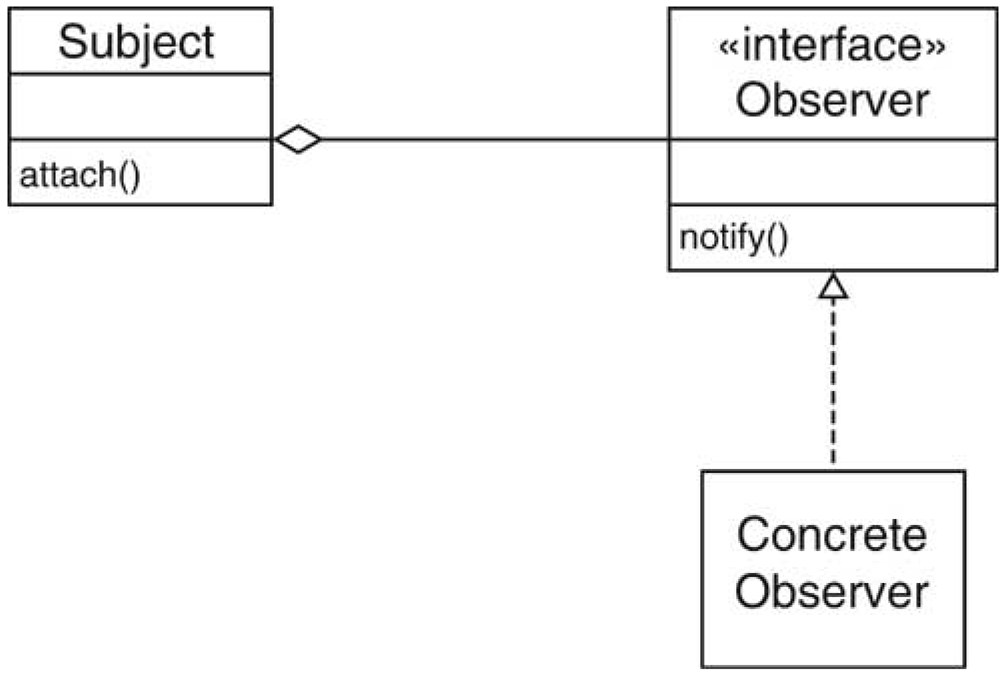
- based on purpose
  - Creational
    - defer object creation to some other class/object
    - examples: Singleton, Abstract Factory, Factory Method
  - Structural
    - composing classes/objects
    - examples: Adapter, Composite, Decorator
  - Behavioral
    - algorithms, flow of control, objects working together
    - examples: Observer, Strategy, Template Method
- implementation
    - interface
      - Composite, Decorator, Observer, Strategy, ...
    - inheritance
      - Template method, ...

Observer pattern

Model/View/Controller



Observer pattern



- Subject = source of events
- Observer = consumer of events
  - Uses "callback" method(s)

Observer pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Applicability
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Observer Examples

- Java API
  - ActionListener, ItemChangeListener
  - java.util.Observer (interface) and java.util.Observable (class) (deprecated)
  - callback mechanism
  - can be implemented as a lambda expression (@FunctionalInterface)

Singleton pattern

Singleton Pattern

- Only one instance of a class exists in the java virtual machine
- Use cases
  - logging, drivers, caching, pools, ...
- Java API examples
  - Desktop (java.awt), Runtime (java.lang)
- Implementation
  - private constructor, private static field + public static getter

  or

  - enum (implicit private constructor) with one constant

Singleton implementation(s)

```java
public class SingletonExample {
    private static SingletonExample theInstance = new SingletonExample();

    public static SingletonExample getInstance() {
        return theInstance;
    }

    private SingletonExample() {
    }
}
```

```java
public enum BetterSingleton {
    THE_INSTANCE;
}
```

```java
public enum BetterSingleton {
    THE_INSTANCE;

    private BetterSingleton() {
        System.out.println("constructor called");;
    }

    public void doSomeWork() {
        System.out.println("working...");
    }
}
```

Marker interface

Marker (interface)

- To "mark" a class
- Implementations:
  - Interface
    - Empty interface
    - f.i. Cloneable, Serializable: to "identify the semantics of being cloneable or serializable"
    - Check with "instanceof" or Class class
  - Marker annotation
    - Annotation without elements
    - Special kind of Interface
    - Since Java 5
    - Check with Class class

Marker: example

```
public interface Shippable {
}
```

```
public class Product implements Shippable { //"marked"
    …
}
```

```
private static void ship(Object p) {
    if (p instanceof Shippable) {
        System.out.println("shippable: " + p);
    } else {
        System.out.println("not shippable: " + p);
    }
}
```

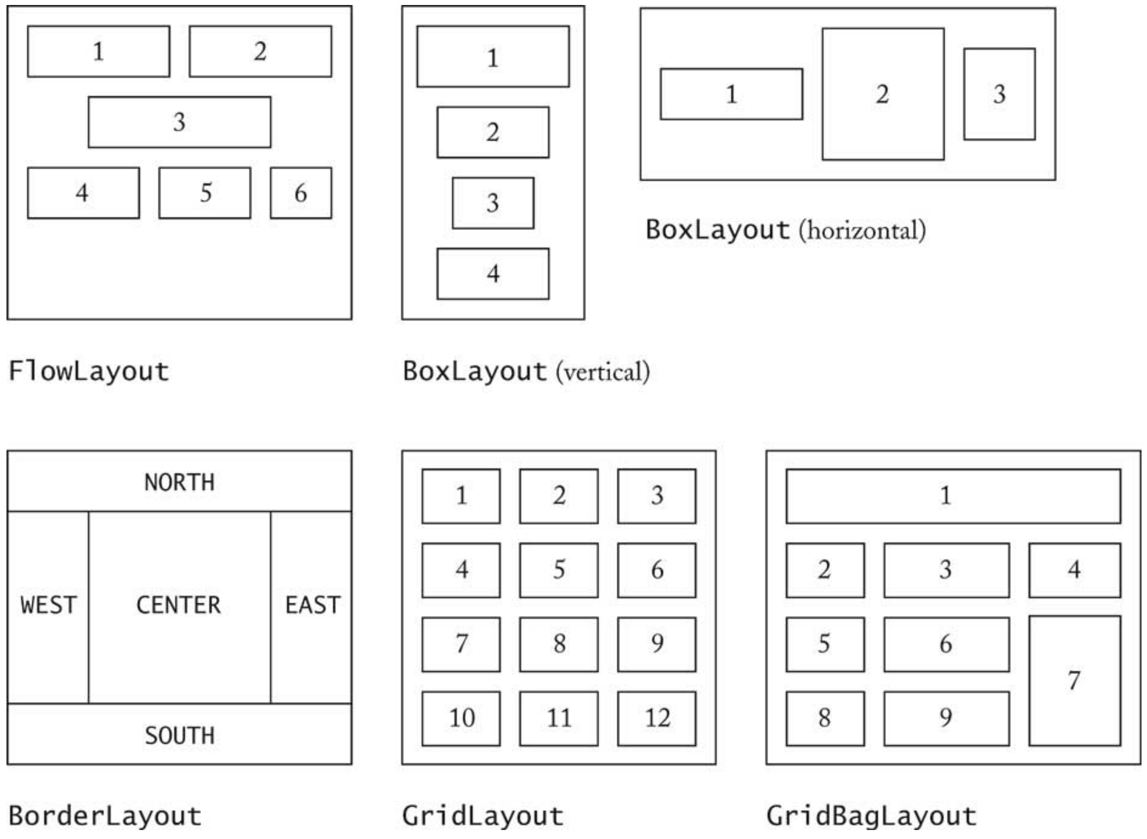
```
@Retention(value=RUNTIME)
public @interface Shippable {
}
```

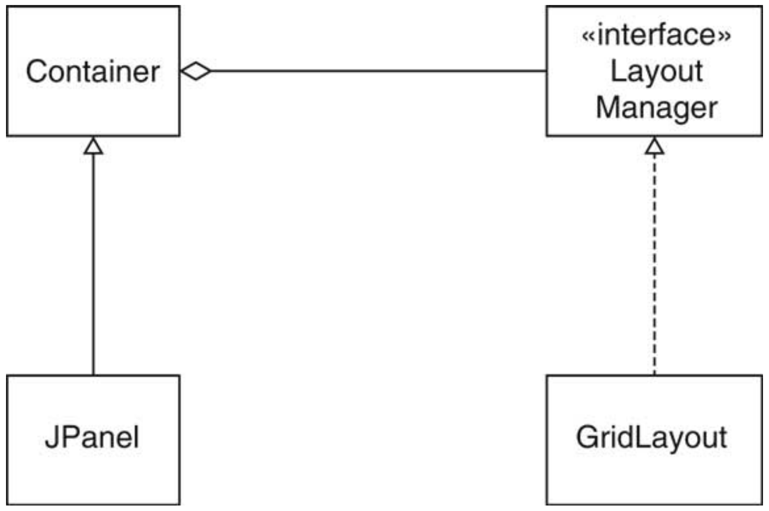
```
@Shippable //"marked"
public class Product  {
    ...
}
```

```
private static void ship(Object p) {
    if (p.getClass().isAnnotationPresent(Shippable.class)) {
        System.out.println("shippable: " + p);
    } else {
        System.out.println("not shippable: " + p);
    }
}
```

Strategy pattern

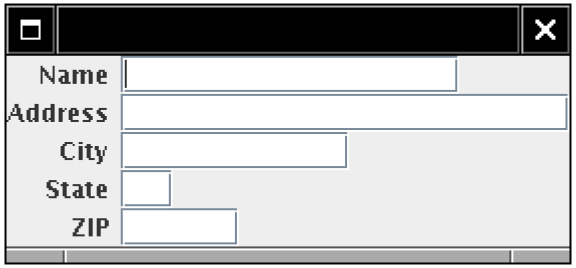
Java awt layout managers





FormLayout?

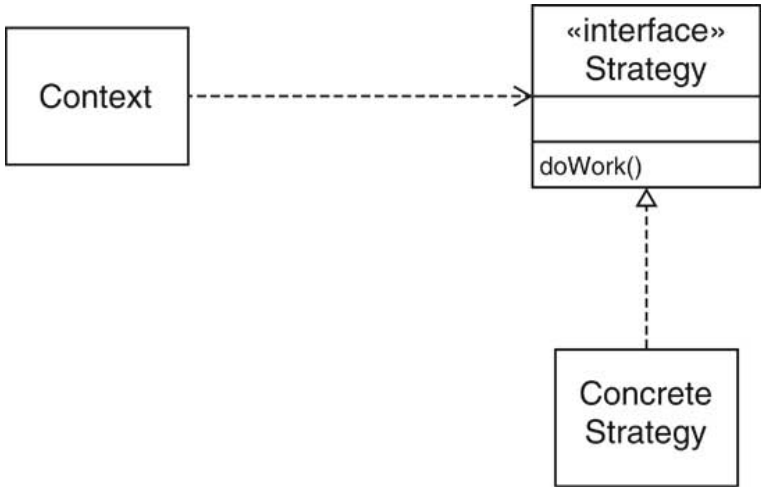
- New layout manager?
  - Form layout
  - Odd-numbered components right aligned (= label)
  - Even-numbered components left aligned (= field)
  - Implement LayoutManager interface type



```
public interface LayoutManager
{
   void layoutContainer(Container parent);
   Dimension minimumLayoutSize(Container parent);
   Dimension preferredLayoutSize(Container parent);
   void addLayoutComponent(String name, Component comp);
   void removeLayoutComponent(Component comp);
}
```

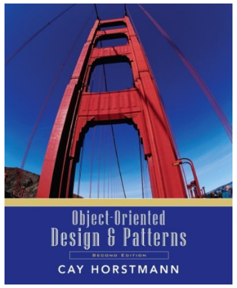
Strategy pattern

- Intent
  - A class can benefit from different variants for an algorithm
  - Clients sometimes want to replace standard algorithms with custom versions
- Example
  - sorting



Design patterns & refactoring example

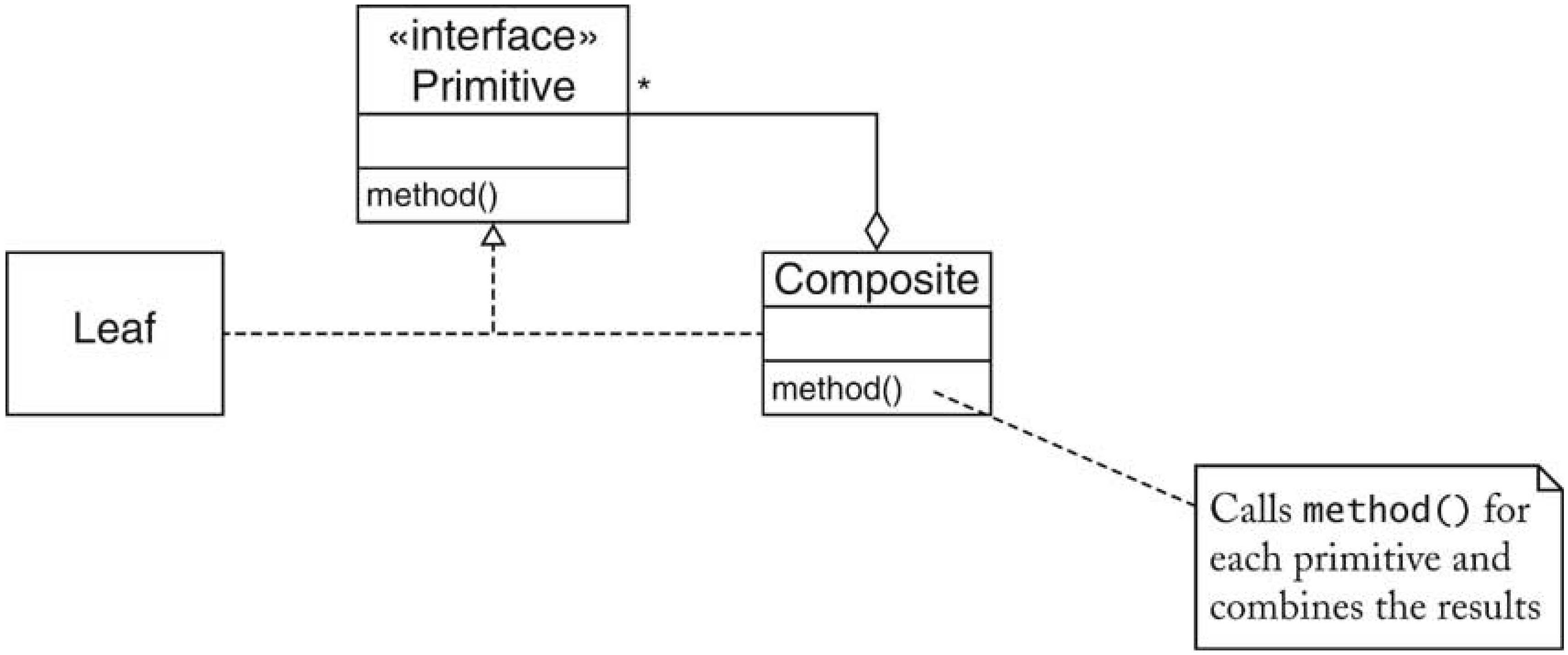
- Object-Oriented Design & Patterns, Chapter 5
- Invoice
  - Product (description & price)
  - Bundle of products (Composite design pattern)
  - Discounted products (Decorator design pattern)
  - Format invoice (Strategy design pattern)
  - Update invoice (Observer design pattern)



http://horstmann.com/design_and_patterns.html

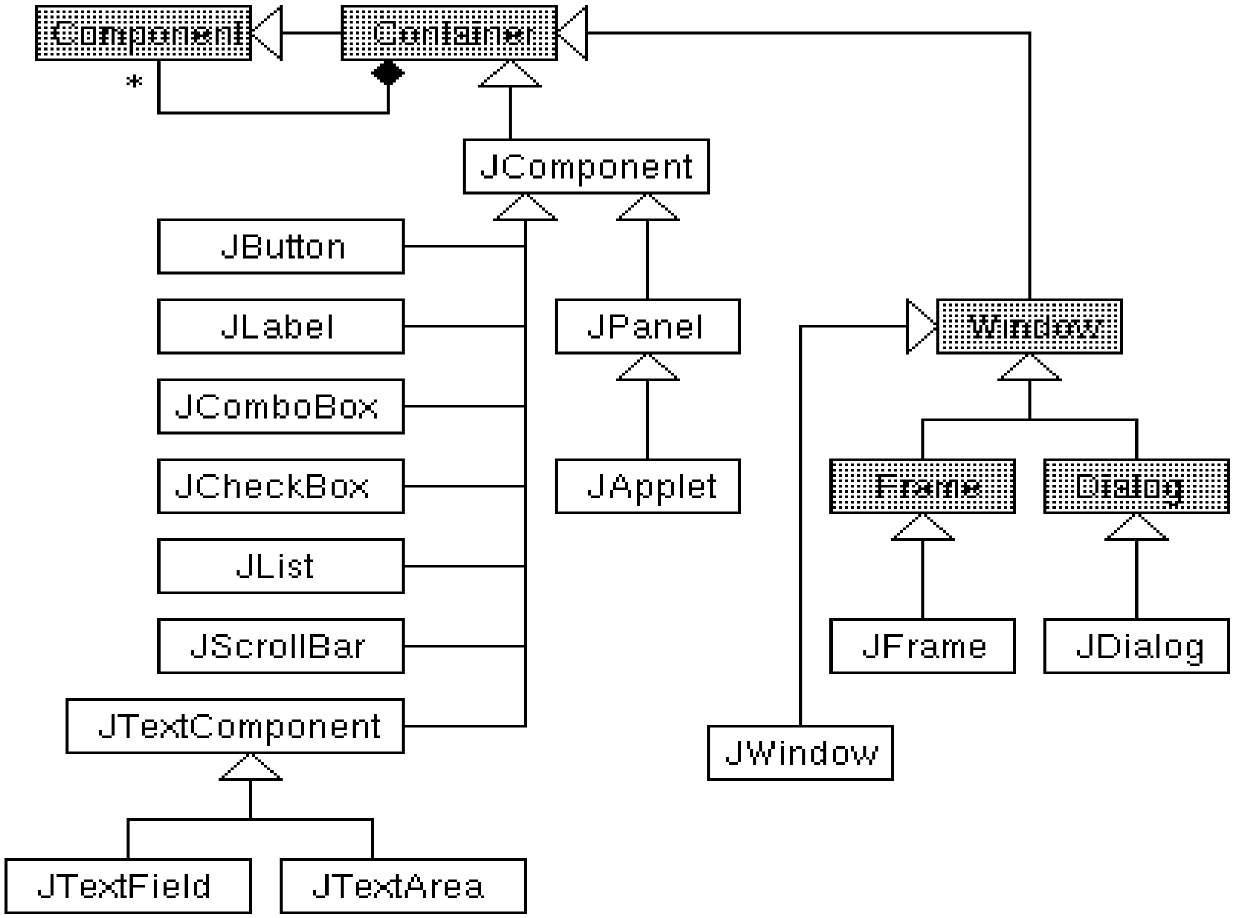
Composite pattern

- Intent
  - Primitive objects can be combined to composite objects
  - Clients treat a composite object as a primitive object

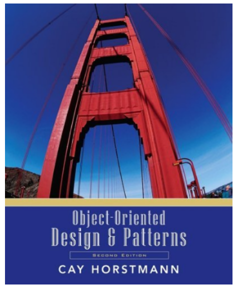


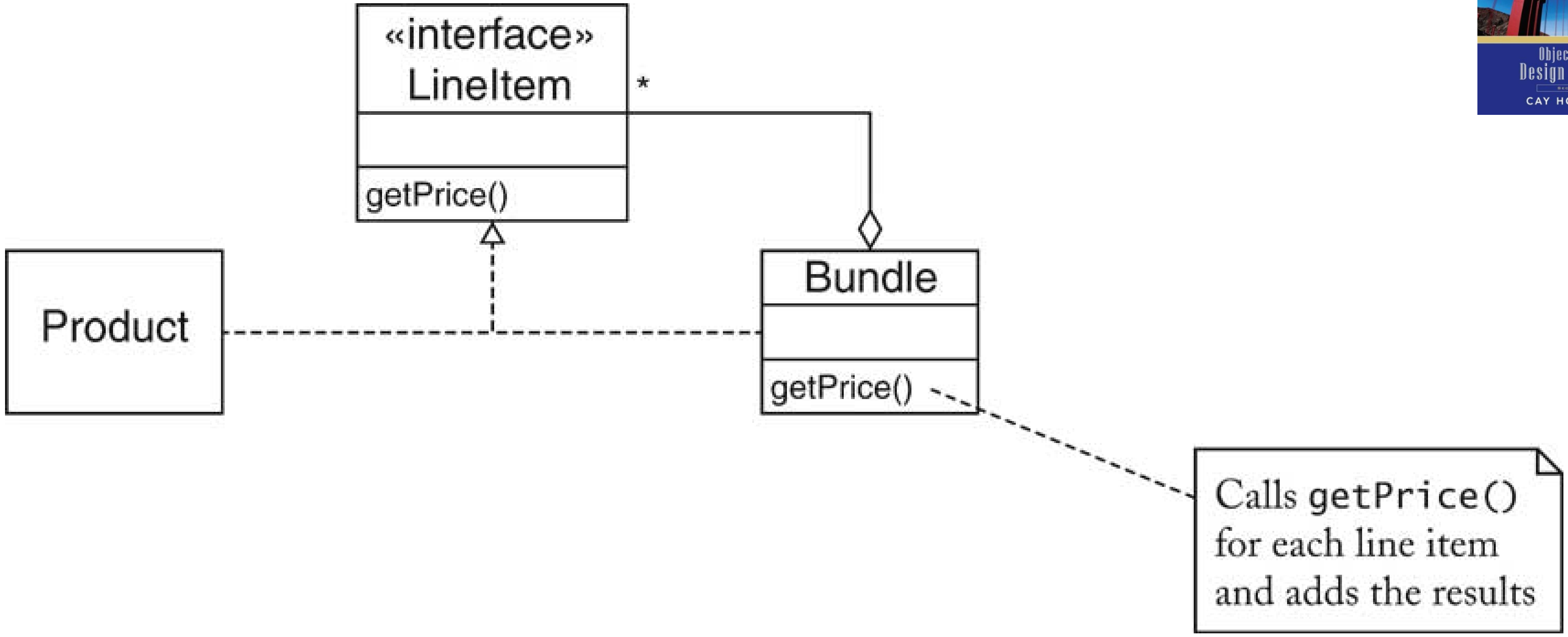
Composite pattern

- Example



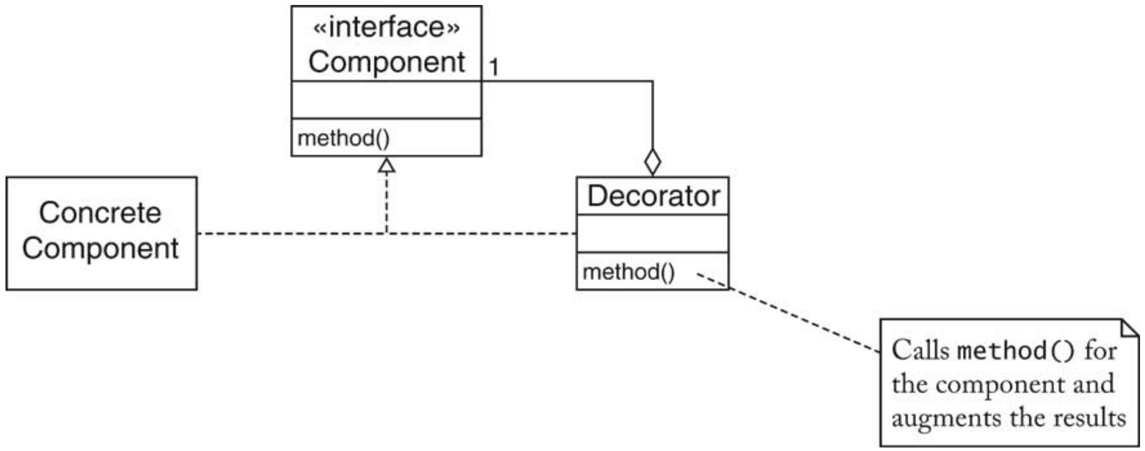
Bundle (Composite pattern)





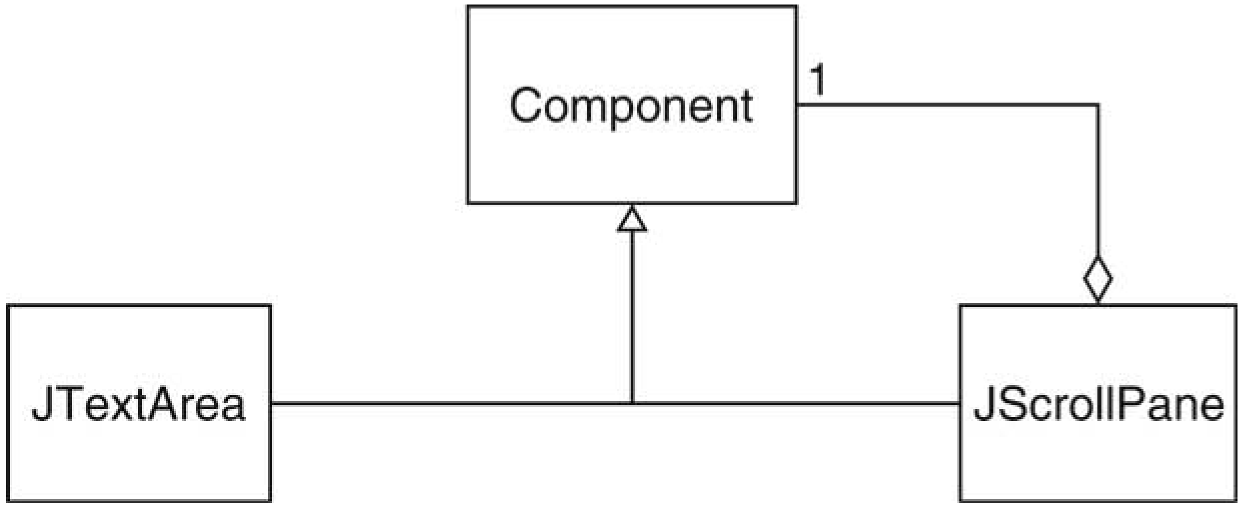
Decorator pattern

- Intent
  - Component objects can be decorated (visually or behaviorally enhanced)
  - The decorated object can be used in the same way as the undecorated object
  - The component class does not want to take on the responsibility of the decoration
  - There may be an open-ended set of possible decorations

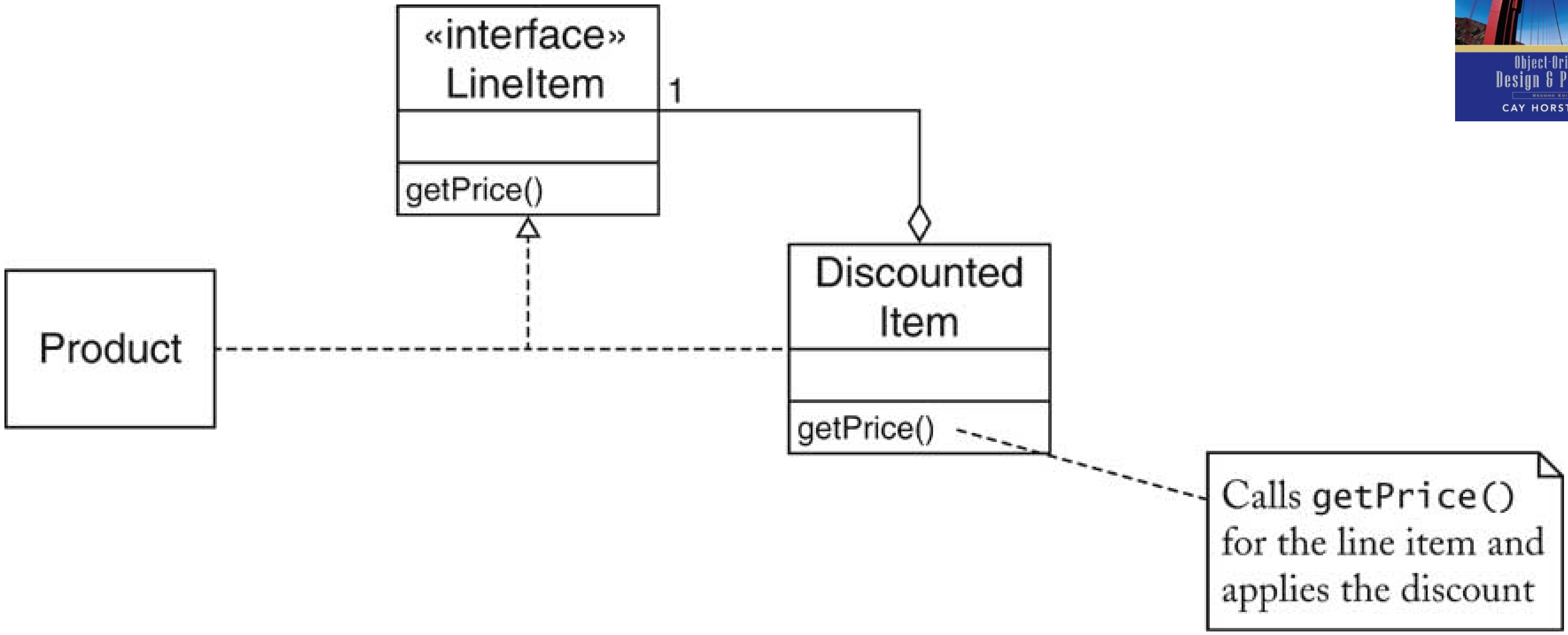


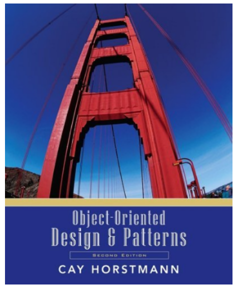
Decorator pattern

- Example

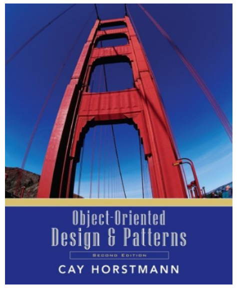


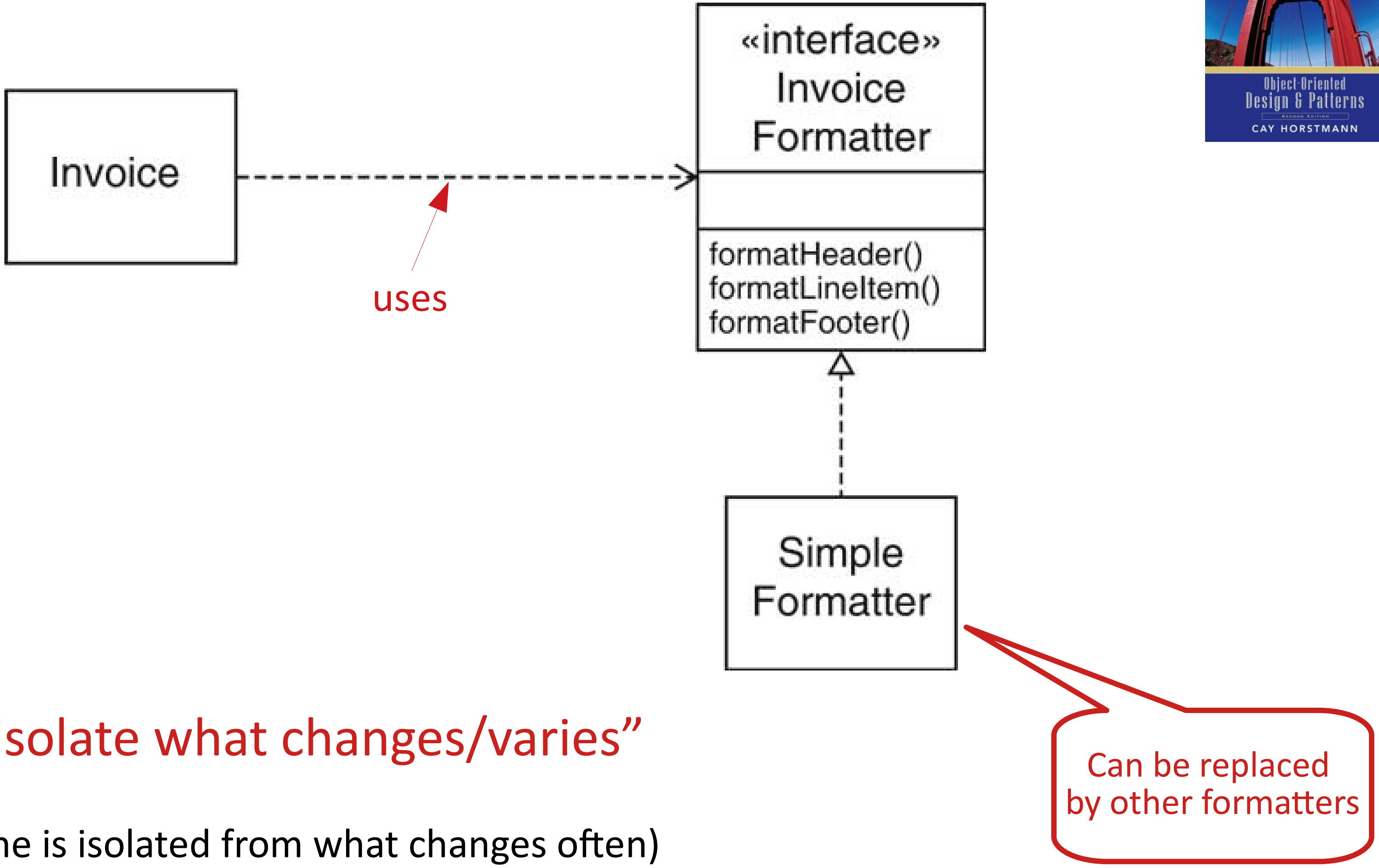
Discounted Item (Decorator pattern)





Invoice formatting (Strategy pattern)

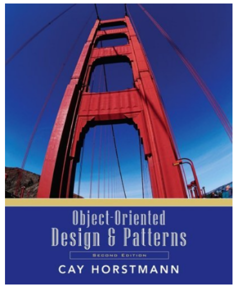


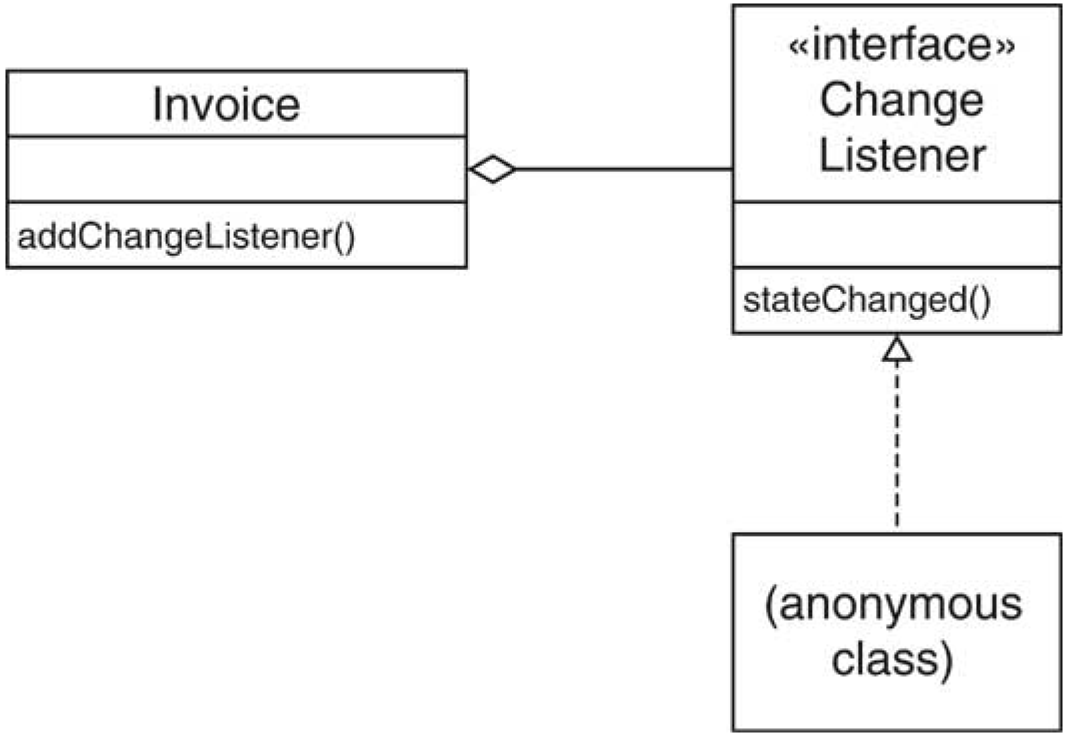


"Encapsulate/isolate what changes/varies"

(what stays the same is isolated from what changes often)

Invoice changes (Observer pattern)





Template method pattern

Not part of this year's (2020-2021) course content

(but I kept the 4 slides)

Template Method pattern

- Intent
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
  - Base class declares algorithm 'placeholders' (aka *frozen* or *invariant spot*), and derived classes implement the placeholders (aka *hot* or *variant spot*).
- Applicability
  - Used in frameworks: framework has the template method; your application implements the specific parts
  - "Don't call us, we'll call you" (Hollywood principle)
  - To avoid code duplication
- Example
  - CrossCompiler for Iphone and Android

Template Method pattern: example (1)

```java
public abstract class CrossCompiler {

    public abstract void collectSource();
    public abstract void compileToTarget();

    public void convertToIntermediate() {
        System.out.println("convert to intermediate");
    }

    // template method
    public final void crossCompile() {
        collectSource();
        convertToIntermediate();
        compileToTarget();
    }
}
```

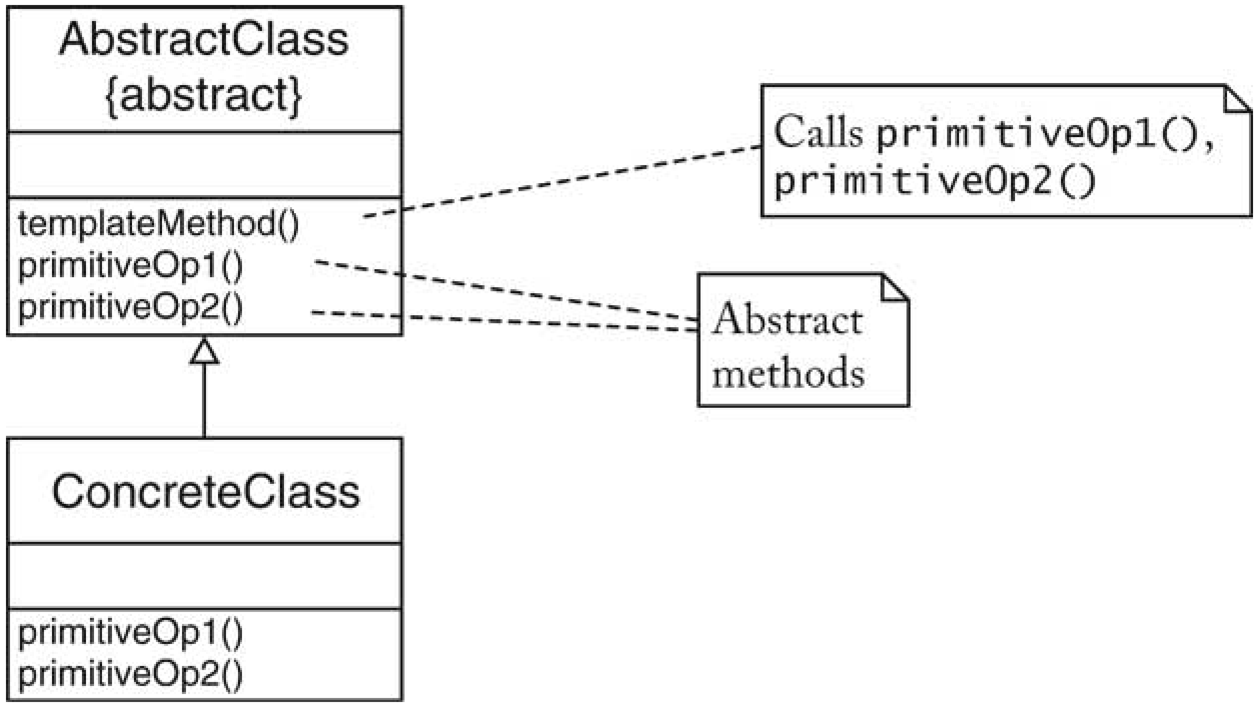
Template Method pattern: example (2)

```java
public class IphoneCompiler extends CrossCompiler {
    @Override
    public void collectSource() {
        System.out.println("collect IPhone source");
    }

    @Override
    public void compileToTarget() {
        System.out.println("compile to IPhone");
    }
}
```

```java
// ...
CrossCompiler iphone = new IphoneCompiler();
iphone.crossCompile();
```

Template Method pattern



Template method pattern

Until here not part of this year's (2020-2021) course content