Design Patterns

http://d3s.mff.cuni.cz

Dominik Škoda
<skoda@d3s.mff.cuni.cz>

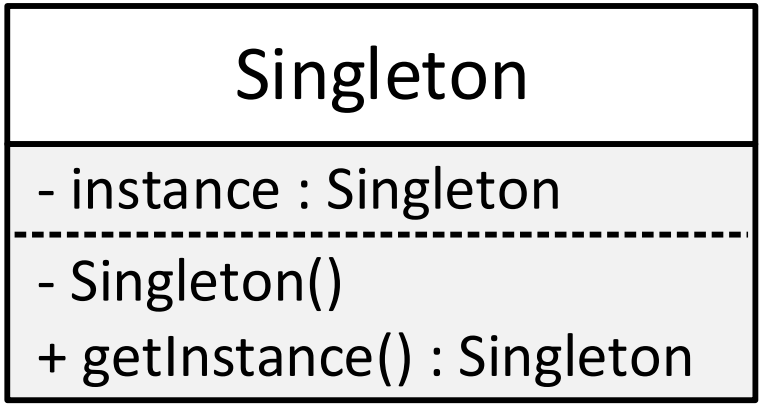
CHARLES UNIVERSITY
Faculty of Mathematics and Physics

Design Patterns

- “Standard” solution of common problems
- It is not complete pieces of code
- Steps, how to design a solution
- Using it may significantly speed up and clarify the development
- Many programmers recognize these patterns
  - They will understand the code faster
- https://en.wikipedia.org/wiki/Design_Patterns

Singleton

- A class that can have only one instance



Singleton

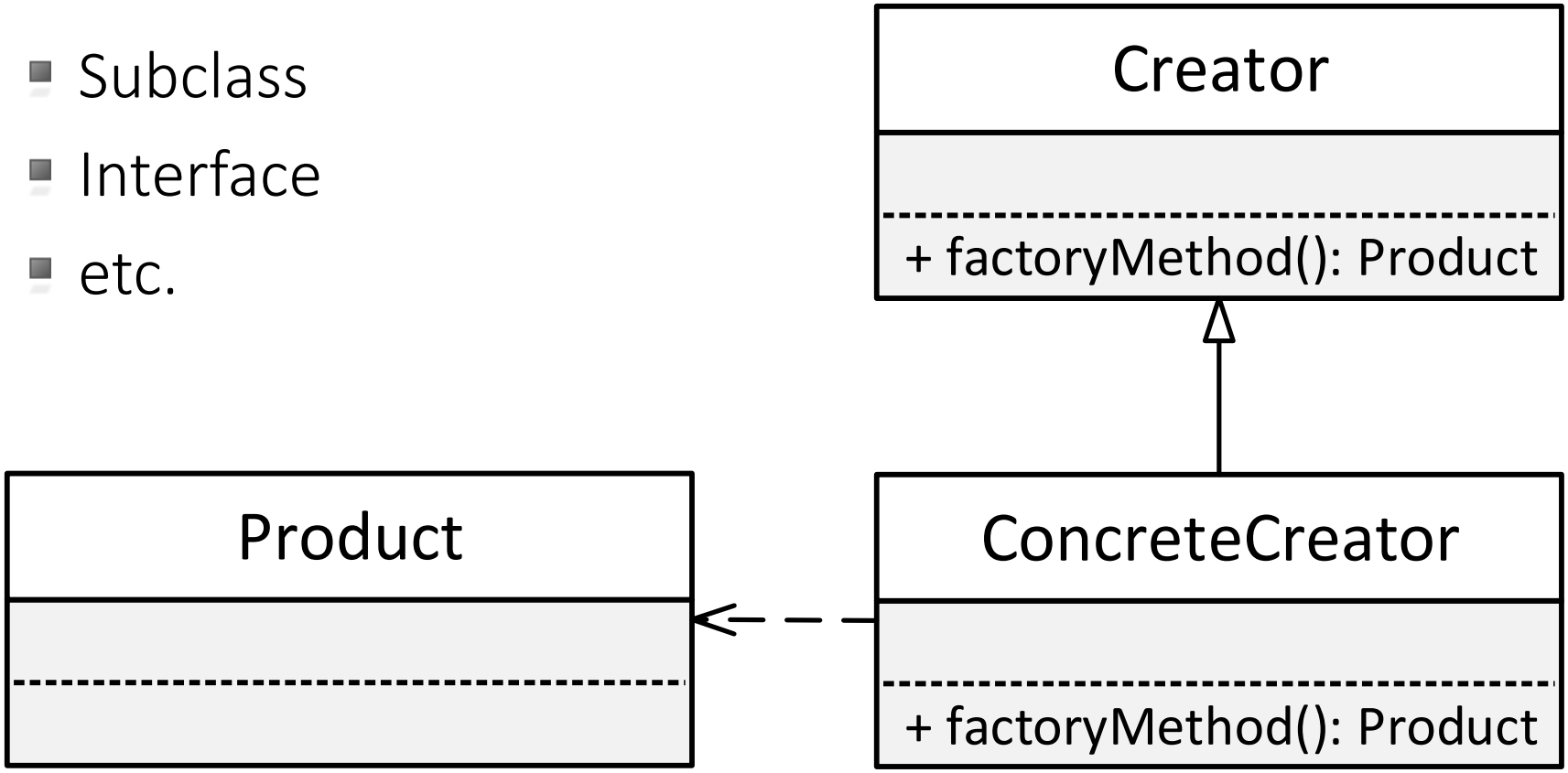
```
public class Singleton {

  private static final Singleton INSTANCE = new Singleton();

  // Private constructor prevents instantiation
  // from other classes
  private Singleton() {
  }

  public static Singleton getInstance() {
    return INSTANCE;
  }
}
```

Singleton

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

  private World() {
  }

  public static World getInstance() {
    return INSTANCE;
  }

  public void create(int inDays) {
    waitDays(inDays);
  }

  public void armageddon() {
    System.out.println("Bruce didn't make it. Sorry...")
  }
}

public class God {
  public static void main(String[] args) {
    World.getInstance().create(7);
    // Do something
    World.getInstance().armageddon();
  }
}
```

Factory Method

- Creating an object without exact knowledge of the class
  - Subclass
  - Interface
  - etc.



Factory Method

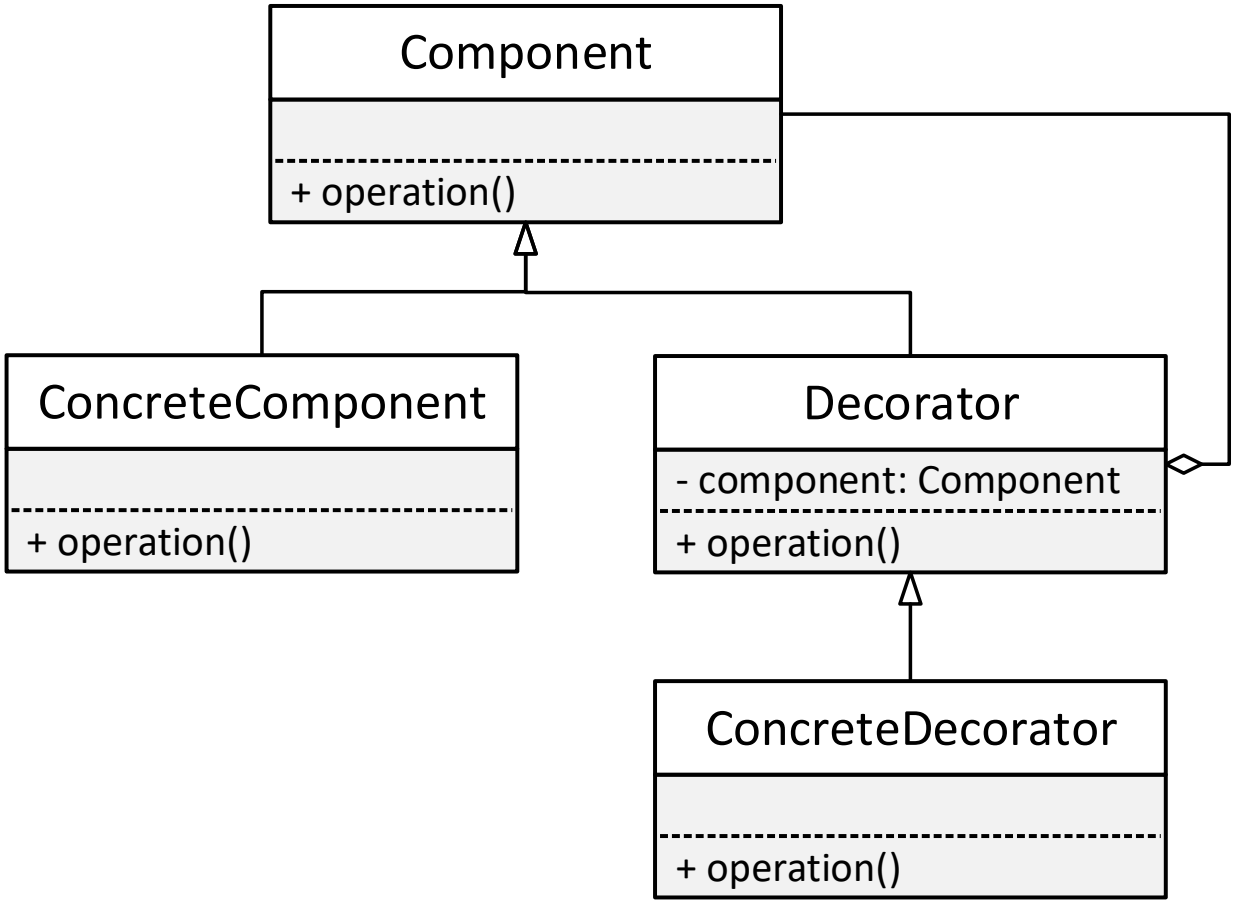
```
class Complex {

  public static Complex fromCartesian(double real,
                                      double imaginary) {
    return new Complex(real, imaginary);
  }

  public static Complex fromPolar(double modulus, double angle) {
    return new Complex(modulus * cos(angle), modulus * sin(angle));
  }

  private Complex(double a, double b) {
    //...
  }
}

Complex c = Complex.fromPolar(1, pi);
```

Factory Method

```
public interface ImageReader {
  public DecodedImage getDecodedImage();
}
public class GifReader implements ImageReader {
  public DecodedImage getDecodedImage() {
    // ...
    return decodedImage;
  }
}
public class JpegReader implements ImageReader {
  public DecodedImage getDecodedImage(){
    // ...
    return decodedImage;
  }
}
public class ImageReaderFactory {
  public static ImageReader getImageReader(InputStream is) {
    int imageType = determineImageType(is);

    switch(imageType){
      case ImageReaderFactory.GIF:
        return new GifReader(is);
      case ImageReaderFactory.JPEG:
        return new JpegReader(is);
      // etc.
    }
  }
}
```

Decorator

- Allows to extend functionality of a class
- Without changing the instance of an original class



Decorator

```
// The Window interface
interface Window {
  // draws the Window
  public void draw();

  // returns a description of the Window
  public String getDescription();
}
// implementation of a simple Window without any scrollbars

class SimpleWindow implements Window {
  public void draw() {
    // draw window
  }

  public String getDescription() {
    return "simple window";
  }
}
```

Decorator

```
// Abstract decorator class - note that it
// implements Window
abstract class WindowDecorator implements Window {
  // the Window being decorated
  protected Window decoratedWindow;

  public WindowDecorator(Window decoratedWindow) {
    this.decoratedWindow = decoratedWindow;
  }

  public void draw() {
    decoratedWindow.draw();
  }
}
```

Decorator

```
// the first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {

  public VerticalScrollBarDecorator(Window decoratedWindow) {
    super(decoratedWindow);
  }

  public void draw() {
    decoratedWindow.draw();
    drawVerticalScrollBar();
  }

  private void drawVerticalScrollBar() {
    // draw the vertical scrollbar
  }

  public String getDescription() {
    return decoratedWindow.getDescription() +
           ", including vertical scrollbars";
  }
}
```

Decorator

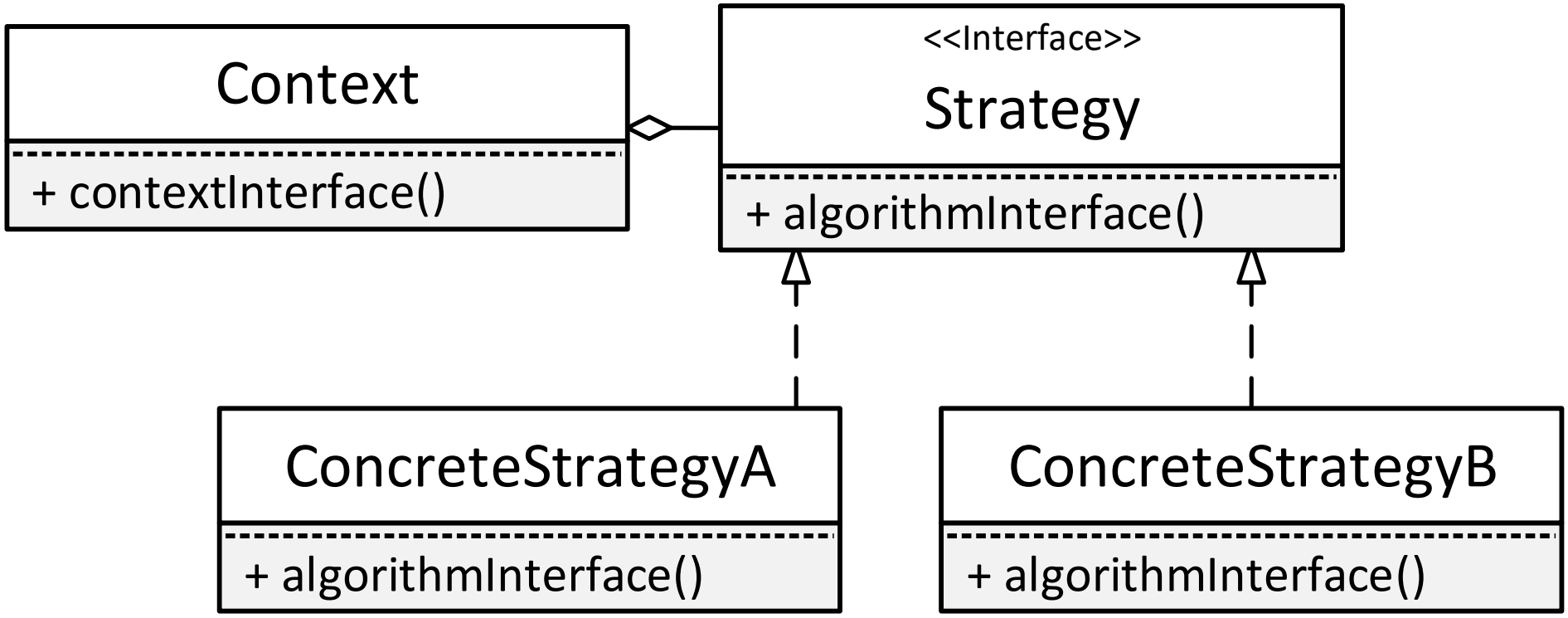
```
// the second concrete decorator which adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {

  public HorizontalScrollBarDecorator(Window decoratedWindow) {
    super(decoratedWindow);
  }

  public void draw() {
    decoratedWindow.draw();
    drawHorizontalScrollBar();
  }

  private void drawHorizontalScrollBar() {
    // draw the horizontal scrollbar
  }

  public String getDescription() {
    return decoratedWindow.getDescription() +
           ", including horizontal scrollbars";
  }
}
```

Decorator

```
public class DecoratedWindowTest {

  public static void main(String[] args) {
    // create a decorated Window with horizontal
    // and vertical scrollbars
    Window decoratedWindow = new HorizontalScrollBarDecorator(
                                 new VerticalScrollBarDecorator(
                                   new SimpleWindow()));

    // print the Window's description
    System.out.println(decoratedWindow.getDescription());
  }
}
```

Strategy

- Allows to select concrete algorithm at runtime
- Encapsulation and interchange of algorithms



Strategy

```
interface Strategy {
  int execute(int a, int b);
}

// Implements the algorithm using the strategy interface
class ConcreteStrategyAdd implements Strategy {

  public int execute(int a, int b) {
    System.out.println("Called ConcreteStrategyAdd's execute()");

    // Do an addition with a and b
    return a + b;
  }
}

class ConcreteStrategySubtract implements Strategy {
  public int execute(int a, int b) { ...  }
}

class ConcreteStrategyMultiply implements Strategy {
  public int execute(int a, int b) { ...  }
}
```

Strategy

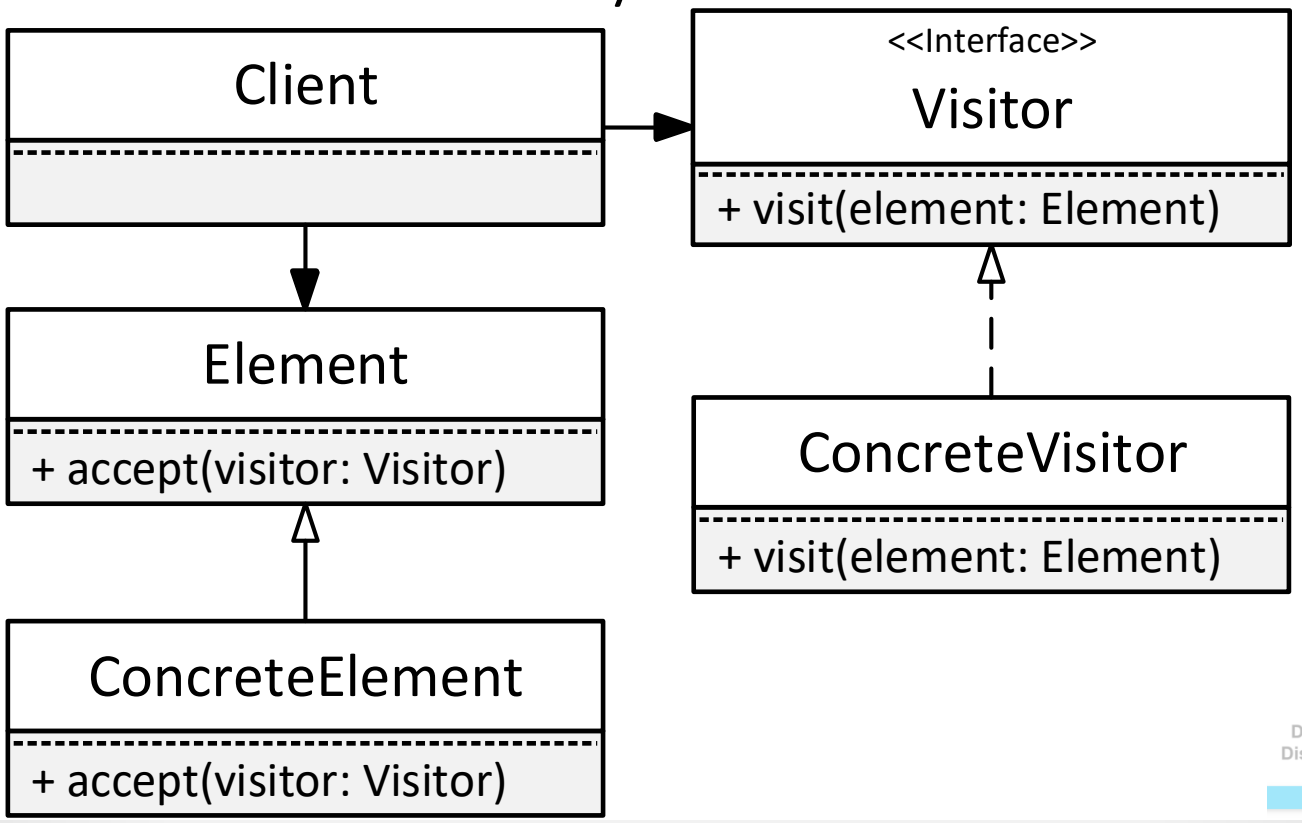
```
// Configured with a ConcreteStrategy object and
// maintains a reference to a Strategy object
class Context {
  private Strategy strategy;

  // Constructor
  public Context(Strategy strategy) {
    this.strategy = strategy;
  }

  public int executeStrategy(int a, int b) {
    return strategy.execute(a, b);
  }
}
```

Strategy

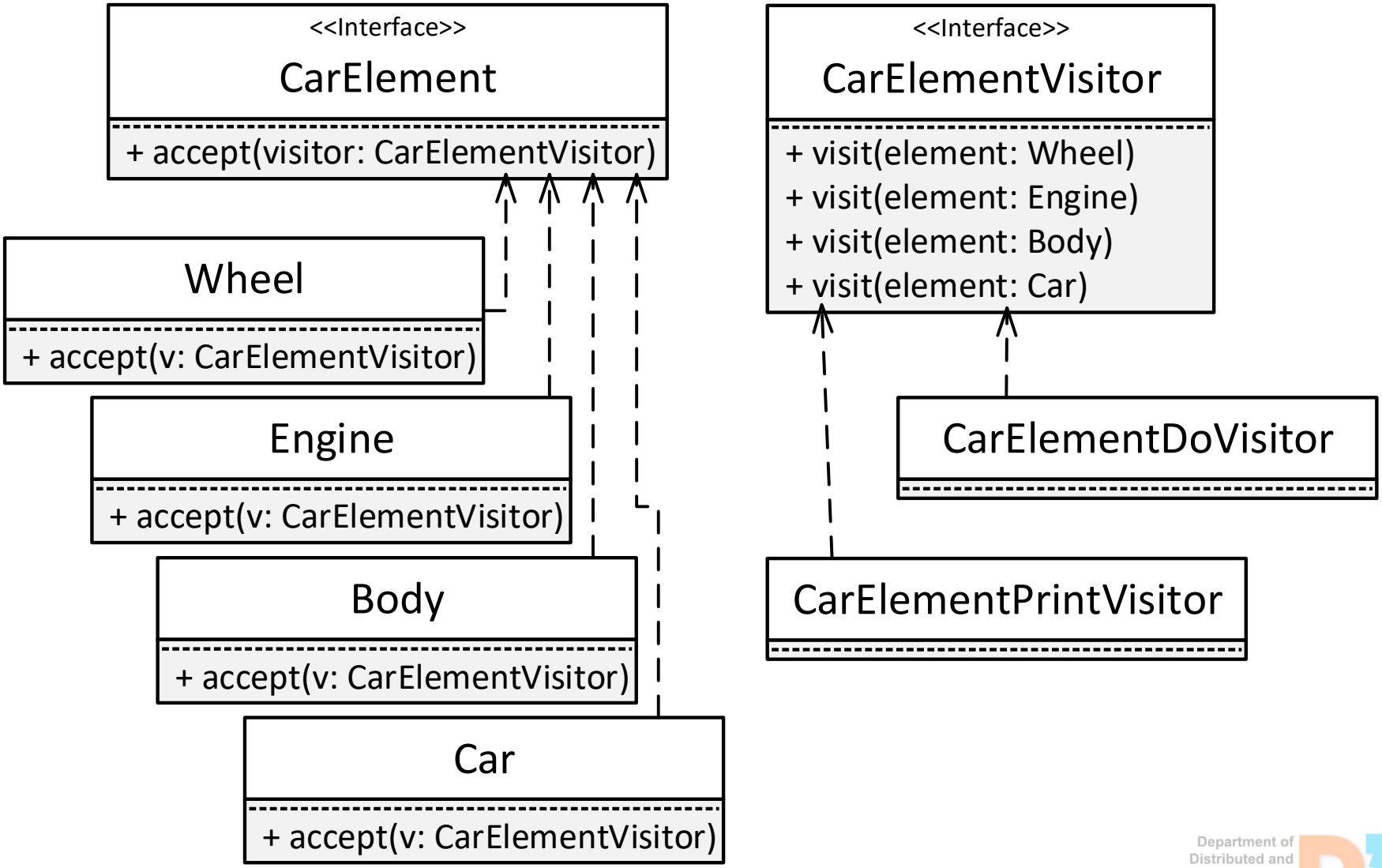
```
class StrategyExample {

  public static void main(String[] args) {
    Context context;

    // Three contexts following different strategies
    context = new Context(new ConcreteStrategyAdd());
    int resultA = context.executeStrategy(3,4);

    context = new Context(new ConcreteStrategySubtract());
    int resultB = context.executeStrategy(3,4);

    context = new Context(new ConcreteStrategyMultiply());
    int resultC = context.executeStrategy(3,4);
  }
}
```

Visitor

- Separation of data and algorithms
- Allows to add operations into existing classes
- Assumes fixed hierarchy of classes



Visitor



Visitor

```
interface CarElement {
  void accept(CarElementVisitor visitor);
}

class Wheel implements CarElement {
  private String name;

  public Wheel(String name) { this.name = name; }
  public String getName() { return this.name; }

  public void accept(CarElementVisitor visitor) {
    visitor.visit(this);
  }
}

class Engine implements CarElement {
  public void accept(CarElementVisitor visitor) {
    visitor.visit(this);
  }
}

class Body implements CarElement {
  public void accept(CarElementVisitor visitor) {
    visitor.visit(this);
  }
}
```

Visitor

```
interface CarElementVisitor {
  void visit(Wheel wheel);
  void visit(Engine engine);
  void visit(Body body);
  void visit(Car car);
}

class CarElementPrintVisitor implements CarElementVisitor {
  public void visit(Wheel wheel) {
    System.out.println("Visiting "+ wheel.getName() + " wheel");
  }
  public void visit(Engine engine) { System.out.println("Visiting engine"); }
  public void visit(Body body) { System.out.println("Visiting body"); }
  public void visit(Car car) { System.out.println("Visiting car"); }
}

class CarElementDoVisitor implements CarElementVisitor {
  public void visit(Wheel wheel) {
    System.out.println("Kicking my "+ wheel.getName() + " wheel");
  }
  public void visit(Engine engine) {
    System.out.println("Starting my engine");
  }
  public void visit(Body body) { System.out.println("Moving my body"); }
  public void visit(Car car) { System.out.println("Starting my car"); }
}
```

Visitor

```
class Car implements CarElement {
  CarElement[] elements;

  public CarElement[] getElements() {
    return elements.clone();
  }

  public Car() {
    this.elements = new CarElement[] {
      new Wheel("front left"), new Wheel("front right"),
      new Wheel("back left") , new Wheel("back right"),
      new Body(), new Engine() };
  }

  public void accept(CarElementVisitor visitor) {
    for(CarElement element : this.getElements()) {
      element.accept(visitor);
    }
    visitor.visit(this);
  }
}

public class VisitorDemo {
  static public void main(String[] args){
    Car car = new Car();
    car.accept(new CarElementPrintVisitor());
    car.accept(new CarElementDoVisitor());
  }
}
```