INFORMATION

- **No more lectures**
- Specific questions regarding course: kiko.fernandez@it.uu.se
- **Going for grade 5**: Talk to me before 7th December
- **Don’t forget to send material at least 24h before deadline**

PREVIOUSLY ON ASD

- Architectural patterns
    - Think about non-functional requirements!
- Design patterns I
    - Create flexible design

ADVANCED SOFTWARE DESIGN
LECTURES 8
DESIGN PATTERNS

Kiko Fernandez

DESIGN PATTERNS

ESSENTIALS OF A DESIGN PATTERN

A **pattern name** by which we can call it
– let's us talk about design at a higher level of abstraction.

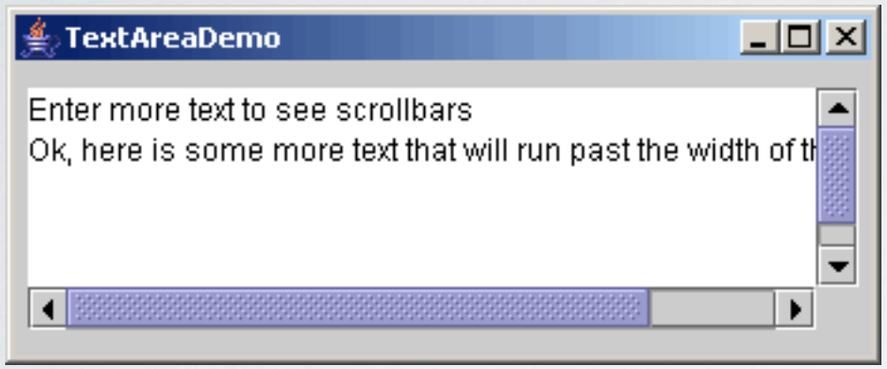
The **problem** to which the pattern applies.

The **solution**, consisting of elements which make up the design, their relationships, responsibilities and collaborations.

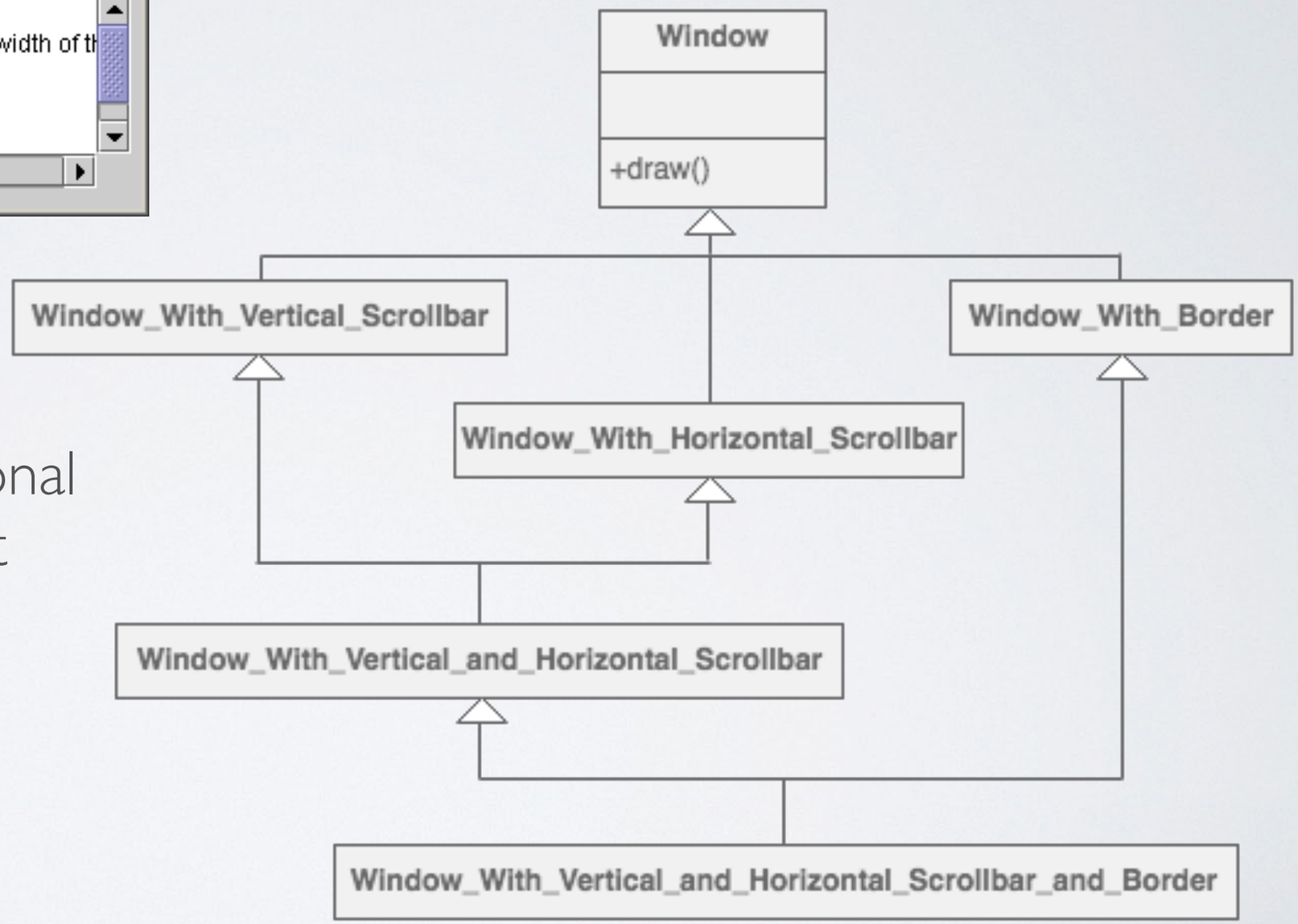
The **consequences**, namely the results and trade-offs of applying the pattern.

DESIGN PROBLEMS

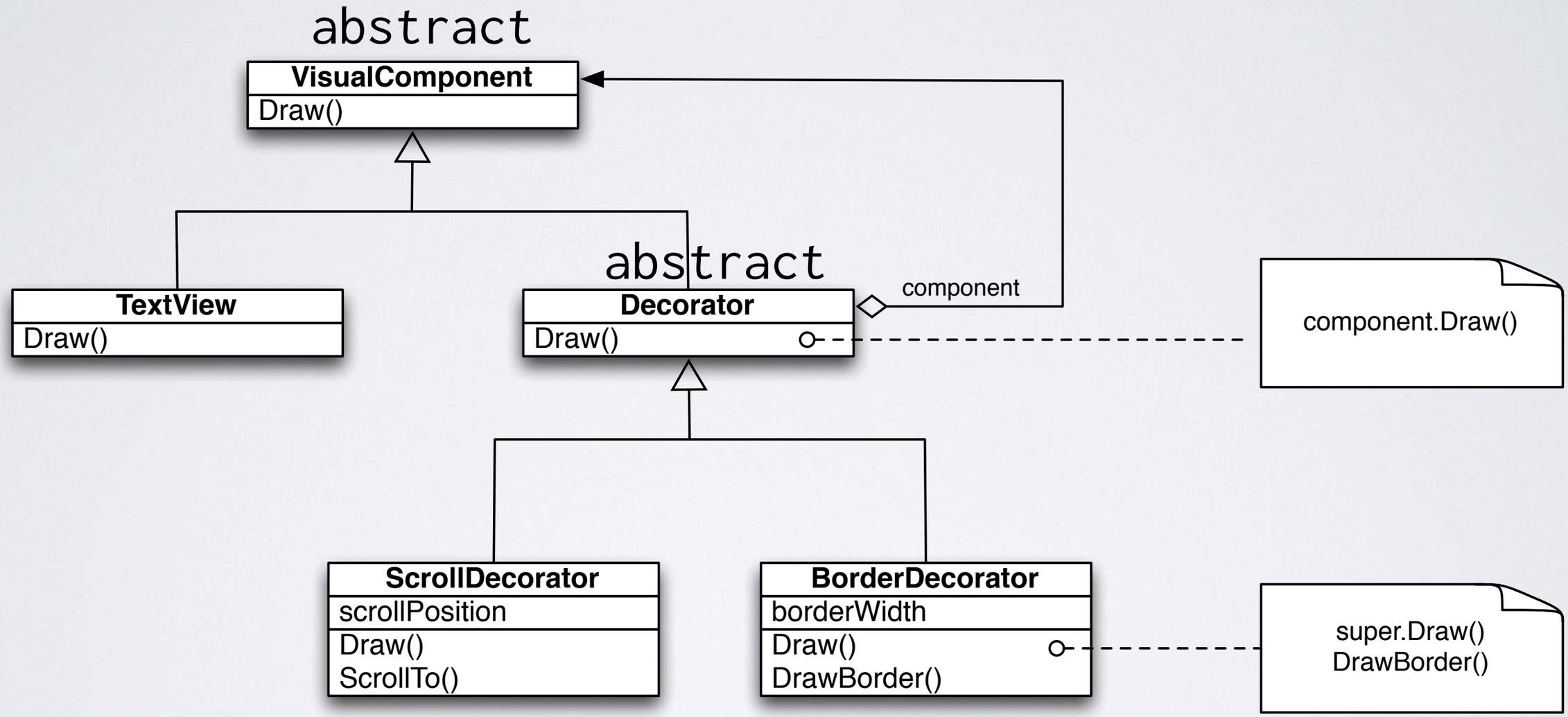
DECORATOR



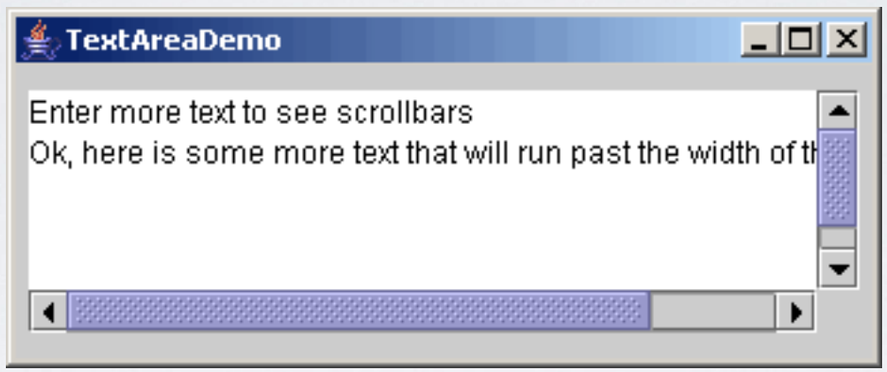
Decorator – Attach additional responsibilities to an object dynamically.



DECORATOR


abstract
VisualComponent
Draw()
TextView
Draw()
abstract
Decorator
Draw()
component
component.Draw()
ScrollDecorator
scrollPosition
Draw()
ScrollTo()
BorderDecorator
borderWidth
Draw()
DrawBorder()
super.Draw()
DrawBorder()

DECORATOR EXAMPLE



CODE

```
abstract class VisualComponent {
  void draw() { .. }
  void resize() { .. }
}
```

Base class of all visual components

```
abstract class Decorator extends VisualComponent {
  private VisualComponent _component;

  Decorator(VisualComponent component) {
    _component = component;
  }

  void draw() {
    _component.draw();
  }

}
```

Abstract **base class** of all decorators

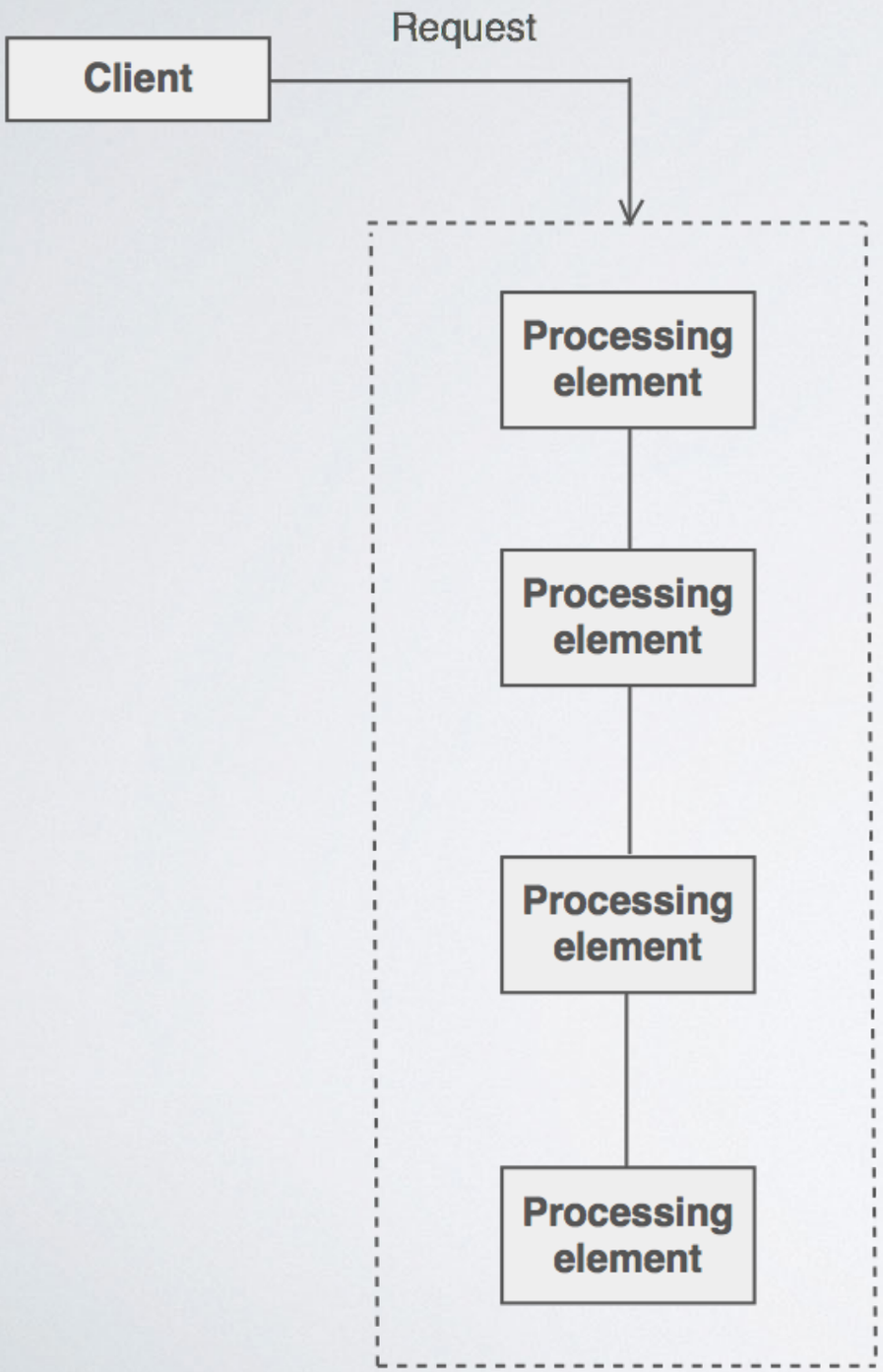
CODE – CONCRETE DECORATOR

```
public class BorderDecorator extends Decorator {
  private int _width;

  BorderDecorator(VisualComponent component, int borderWidth) {
    super(component);
    _witdh = borderWidth;
  }

  void drawBorder(int width) {
    ...
  }

  void draw() {
    super.draw();
    drawBorder(_width);
  }
}
```

PROBLEM

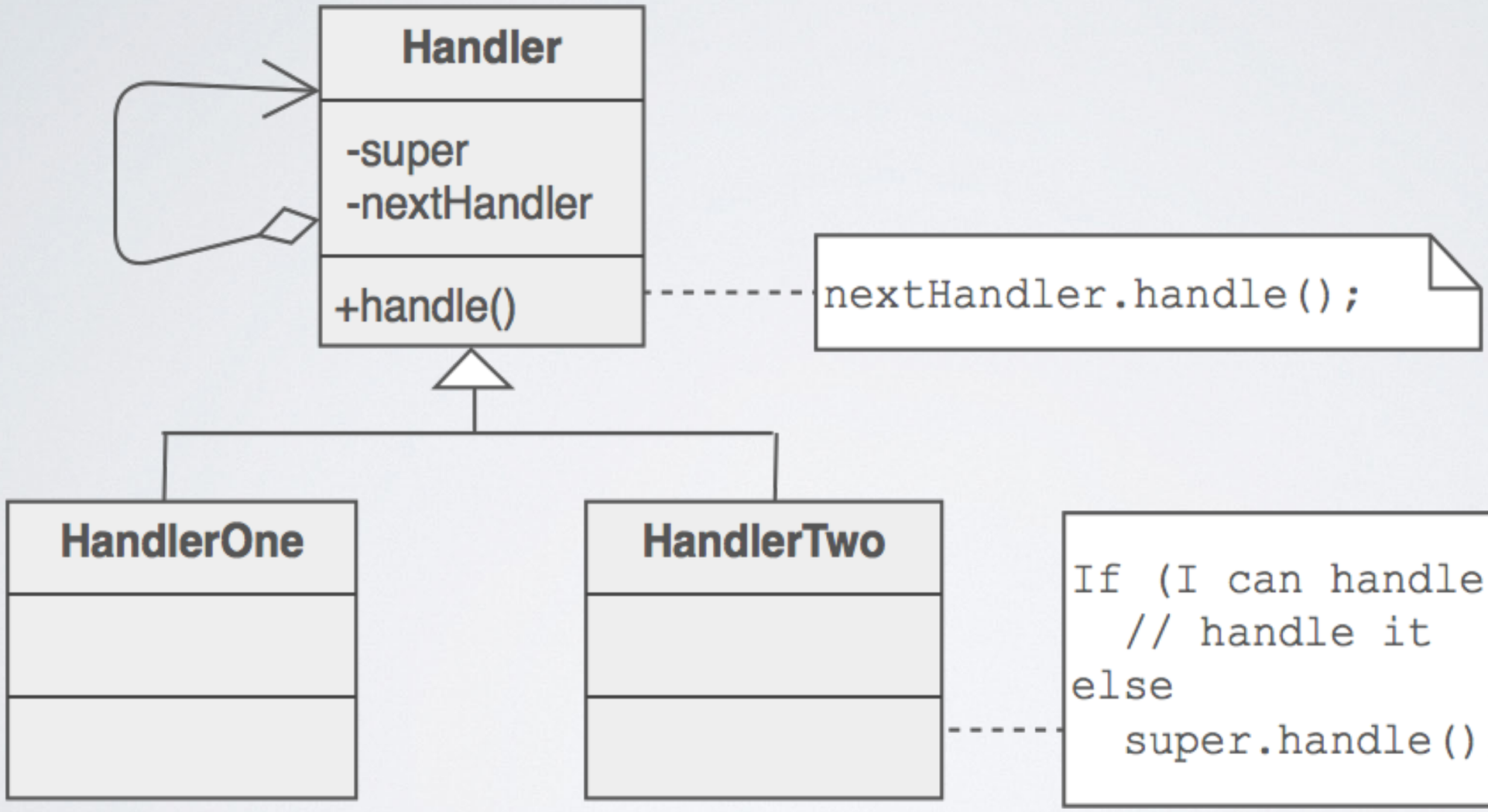
Chain of Responsibility – Avoid coupling the sender of a request to its receiver

PROBLEM



Chain of Responsibility – Avoid coupling the sender of a request to its receiver

CHAIN OF RESPONSIBILITY



```
atm = new ATM50(new ATM20(new ATM10()))
atm.handle(1200 SEK)
```

```
atm = new ATM50(new ATM20(new ATM10()))
atm.handle(1200 SEK)
```

```
class ATM500 {
  public void handle(int money) {
    while(money - 500 > 0){
      money -= 500
      issue500bill()
    }
    super.handle(money)
  }
}
```

```
atm = new ATM50(new ATM20(new ATM10()))
atm.handle(1200 SEK)
```

```
class ATM500 {
  public void handle(int money) {
    while(money - 500 > 0){
      money -= 500
      issue500bill()
    }
    super.handle(money)
  }
}
```

```
class ATM200 {
  public void handle(int money) {
    while(money - 200 > 0){
      money -= 200
      issue20bill()
    }
    super.handle(money)
  }
}
```

```
atm = new ATM50(new ATM20(new ATM10()))
atm.handle(1200 SEK)
```

```
class ATM500 {
  public void handle(int money) {
    while(money - 500 > 0){
      money -= 500
      issue500bill()
    }
    super.handle(money)
  }
}
```

```
class ATM50 {
  public void handle(int money) {
    while(money - 50 > 0){
      money -= 50
      issue50bill()
    }
    super.handle(money)
  }
}
```

```
class ATM200 {
  public void handle(int money) {
    while(money - 200 > 0){
      money -= 200
      issue20bill()
    }
    super.handle(money)
  }
}
```

```
atm = new ATM50(new ATM20(new ATM10()))
atm.handle(1200 SEK)
```

```
class ATM500 {
  public void handle(int money) {
    while(money - 500 > 0){
      money -= 500
      issue500bill()
    }
    super.handle(money)
  }
}
```

```
class ATM50 {
  public void handle(int money) {
    while(money - 50 > 0){
      money -= 50
      issue50bill()
    }
    super.handle(money)
  }
}
```

```
class ATM200 {
  public void handle(int money) {
    while(money - 200 > 0){
      money -= 200
      issue20bill()
    }
    super.handle(money)
  }
}
```

```
class ATMNull {
  public void handle(int money) {
    return
  }
}
```

PROBLEM

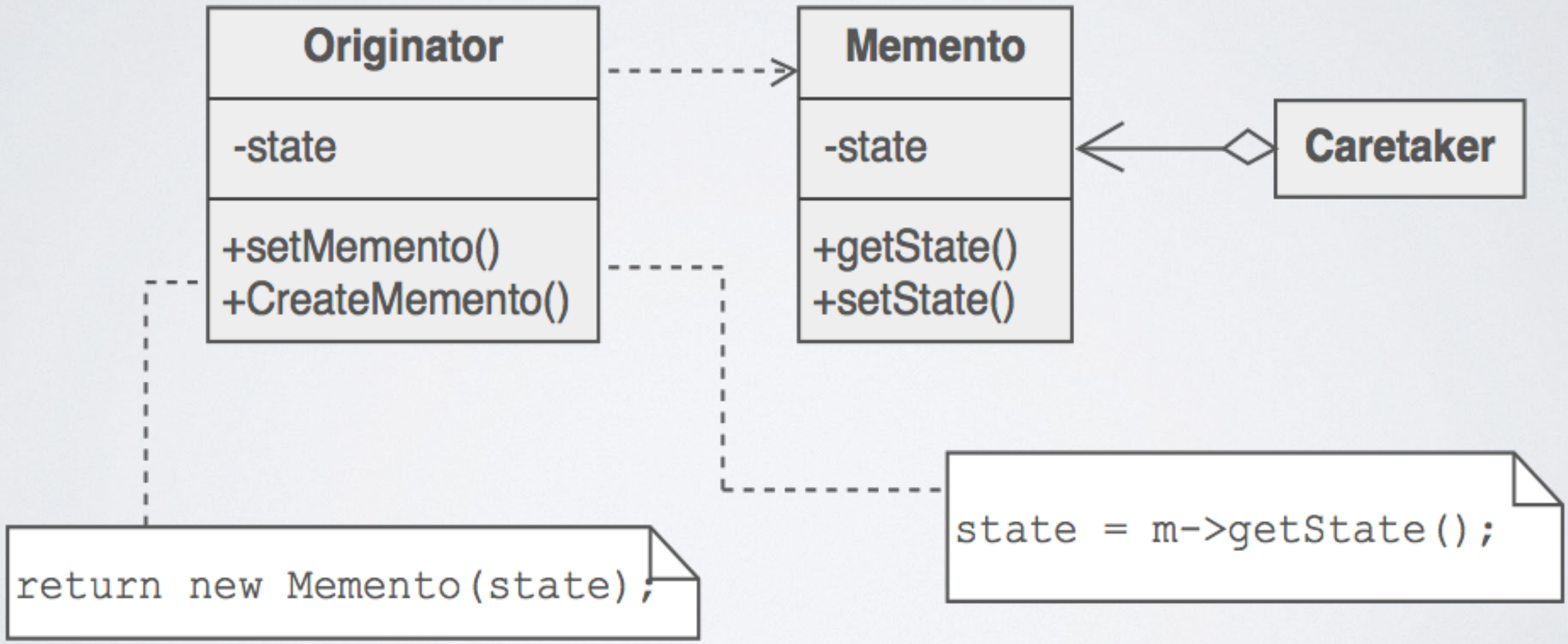
- Need to save the state of an object in order to restore it in the future

PROBLEM

- Need to save the state of an object in order to restore it in the future

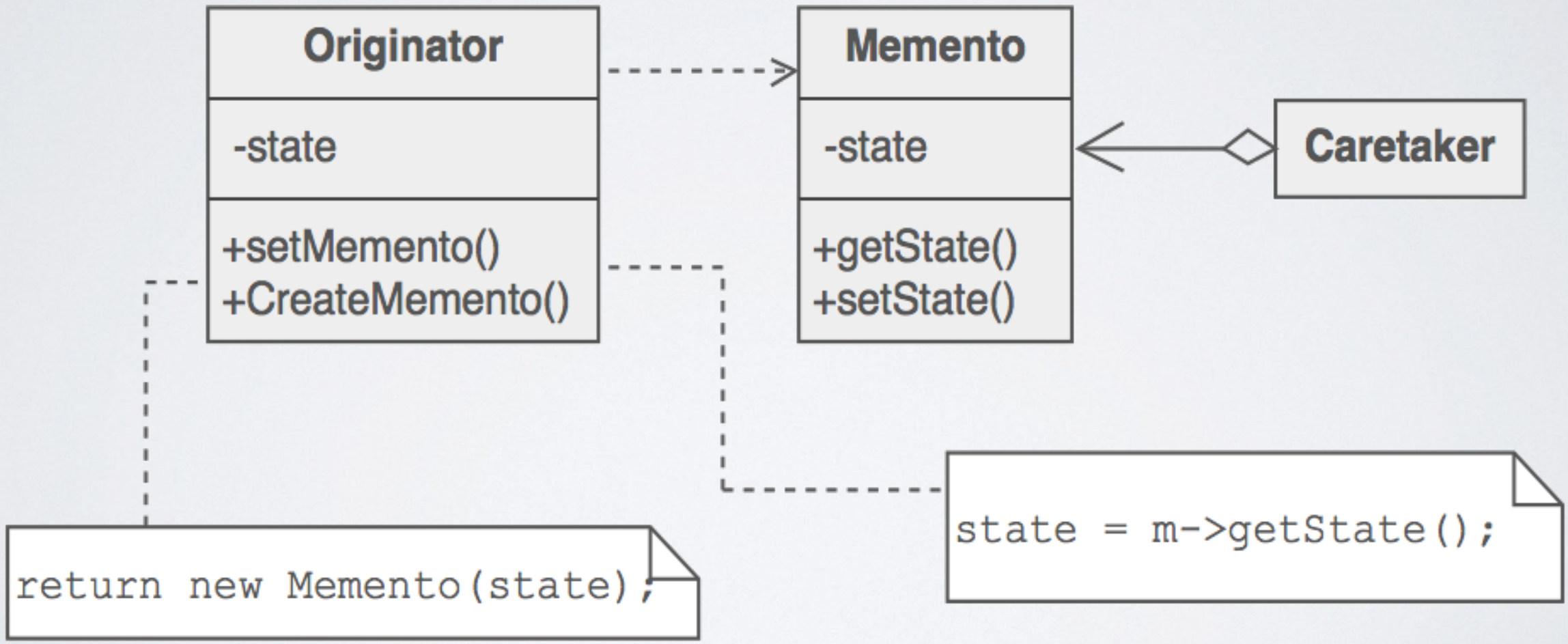
e.g. How to allow easy **undo** actions on Text Editor?

MEMENTO


Originator
-state
+setMemento()
+CreateMemento()
Memento
-state
+getState()
+setState()
Caretaker
return new Memento(state);
state = m->getState();

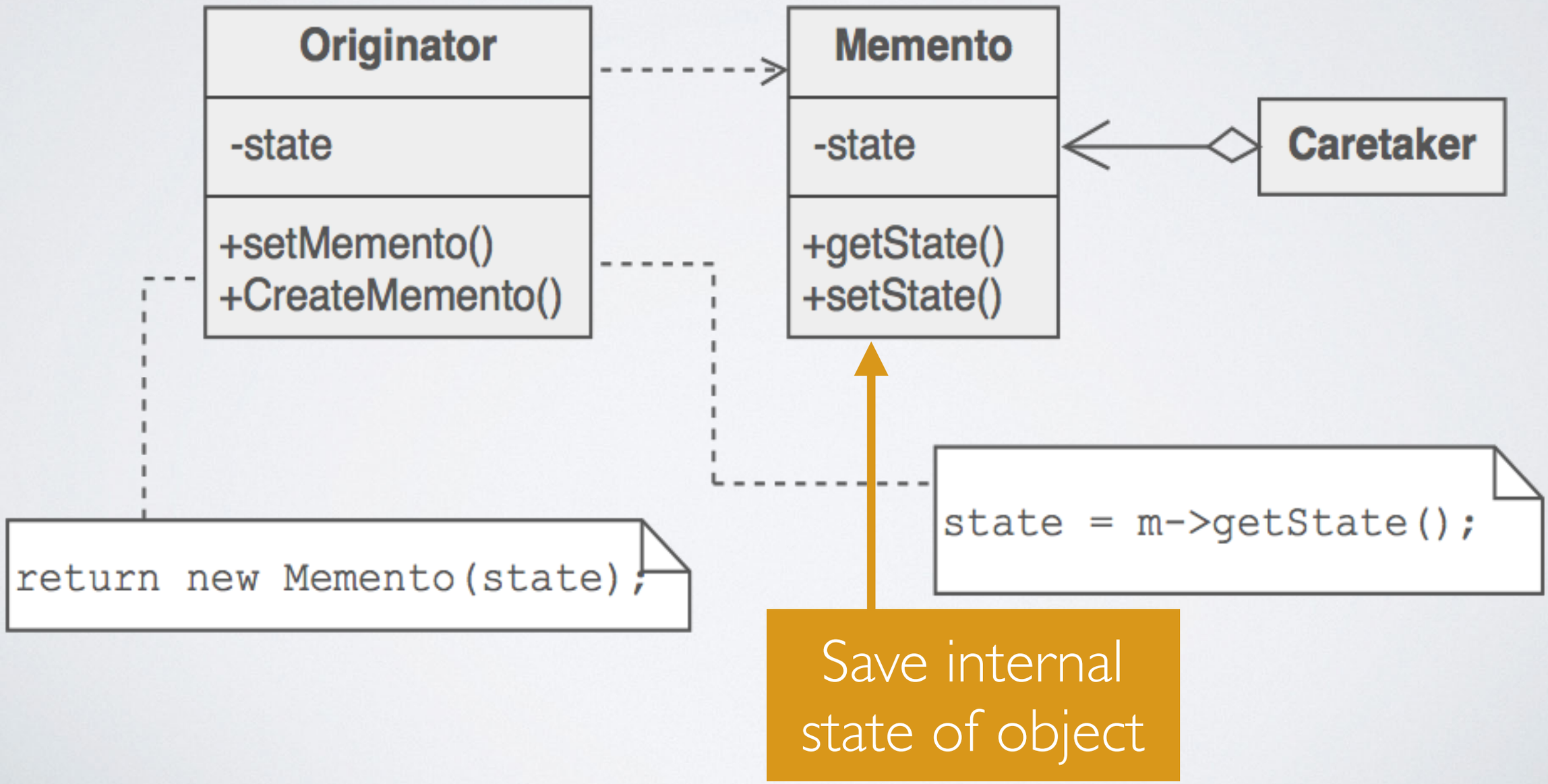
MEMENTO

class that knows how to save and restore its state



MEMENTO

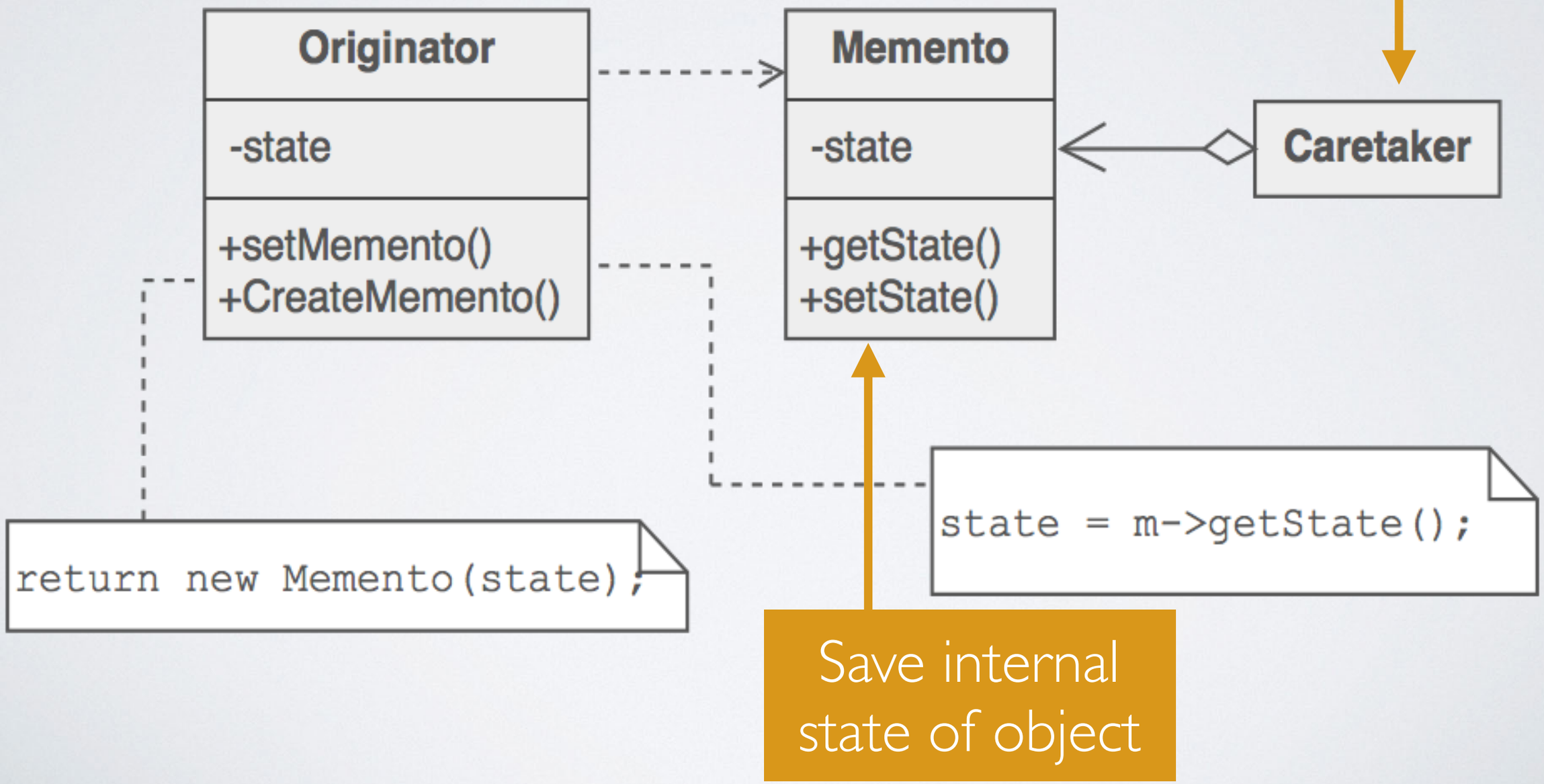
class that knows how to save and restore its state



MEMENTO

class that knows how to save and restore its state

Manages mementos



MEMENTO

```
class Originator {
    private String state;

    public void set(String state) {
        this.state = state;
    }

    public Memento saveToMemento() {
        return new Memento(state);
    }
    public void restoreFromMemento(Memento m)
        state = m.getSavedState();
    }
}
```

```
class Memento {
    private String state;

    public Memento(String stateToSave) { state = stateToSave; }
    public String getSavedState() { return state; }
}
```

```
class Caretaker {
   private ArrayList<Memento> savedStates = new ArrayList<Memento>();
   public void addMemento(Memento m) { savedStates.add(m); }
   public Memento getMemento(int index) { return savedStates.get(index); }
}
```

```
class MementoExample {
   public static void main(String[] args) {
       Caretaker caretaker = new Caretaker();

       Originator originator = new Originator();
       originator.set("State1");
       originator.set("State2");
       caretaker.addMemento( originator.saveToMemento() );
       originator.set("State3");
       caretaker.addMemento( originator.saveToMemento() );
       originator.set("State4");

       originator.restoreFromMemento( caretaker.getMemento(1) );
   }
}
```

PROBLEM

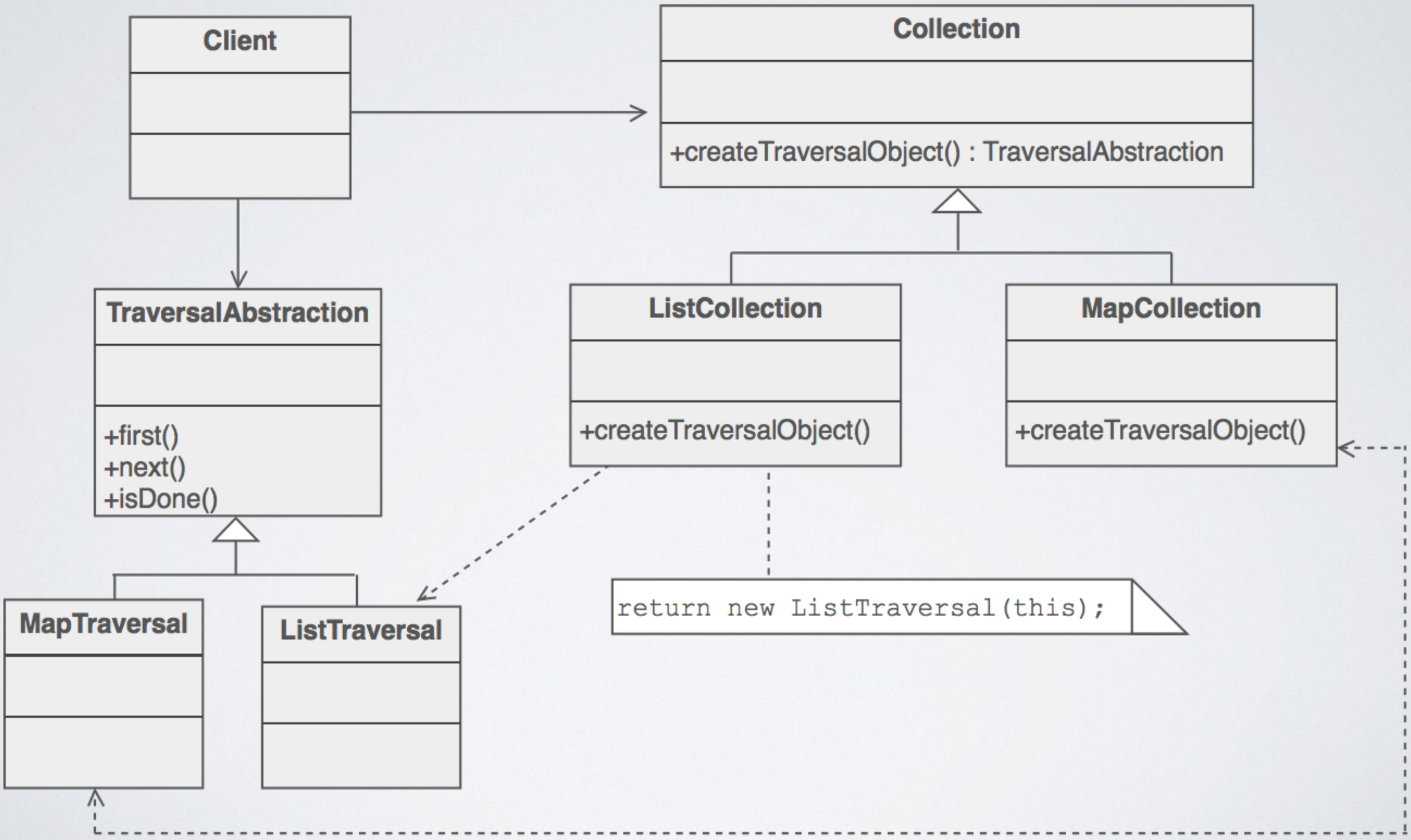
Traverse through a collection without exposing the internals.

PROBLEM

Traverse through a collection without exposing the internals.

Iterator – Providing a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

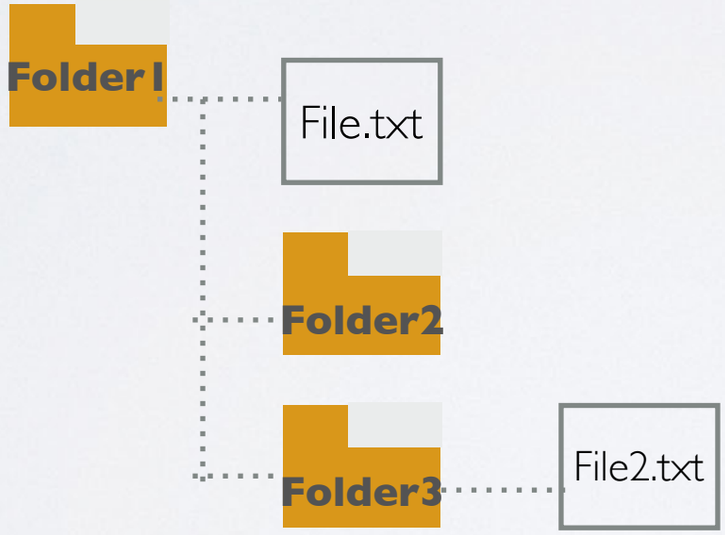
ITERATOR


Client
Collection
+createTraversalObject() : TraversalAbstraction
TraversalAbstraction
+first()
+next()
+isDone()
ListCollection
+createTraversalObject()
MapCollection
+createTraversalObject()
MapTraversal
ListTraversal
return new ListTraversal(this);

PROBLEM

How to represent part-whole hierarchies?

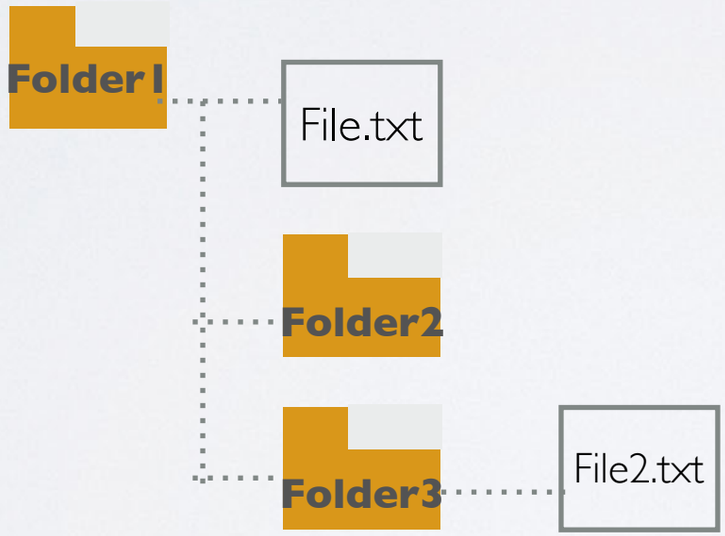
e.g. Folder - File relation, AST, etc?



PROBLEM

How to represent part-whole hierarchies?

e.g. Folder - File relation, AST, etc?

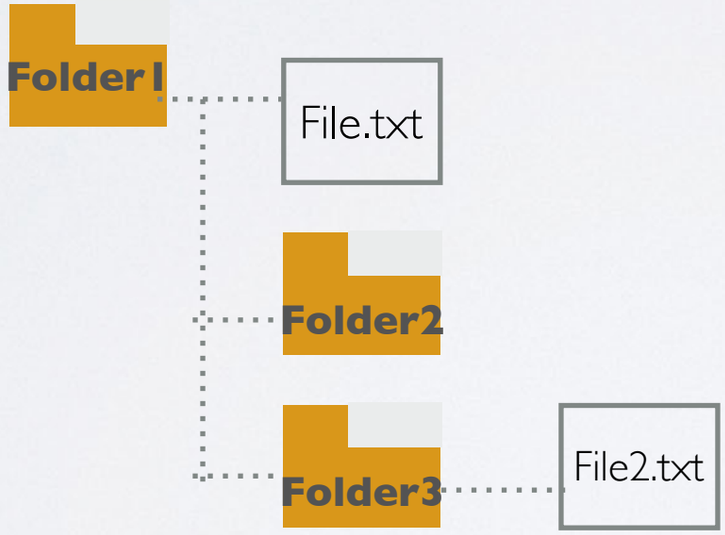


Composite – Compose objects into tree structures to represent part-whole hierarchies — e.g., abstract syntax tree.

PROBLEM

How to represent part-whole hierarchies?

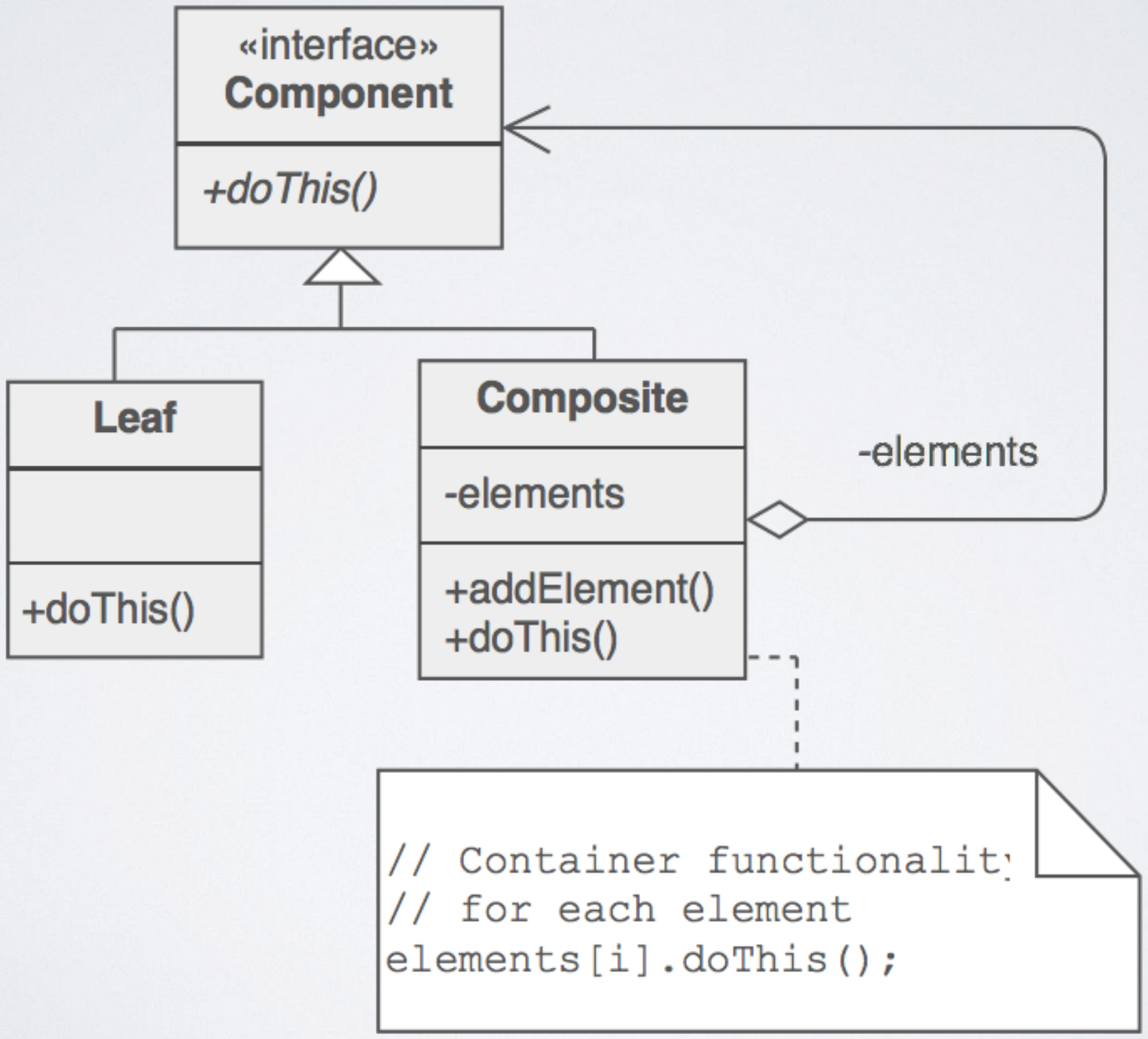
e.g. Folder - File relation, AST, etc?



Processing of a primitive object is handled one way, and processing of a composite object is handled differently

Composite – Compose objects into tree structures to represent part-whole hierarchies — e.g., abstract syntax tree.

COMPOSITE


«interface»
Component
+doThis()
Leaf
+doThis()
Composite
-elements
+addElement()
+doThis()
-elements
// Container functionalit
// for each element
elements[i].doThis();

PROBLEM

PROBLEM

```
$ python
>>> x = 4;|
```

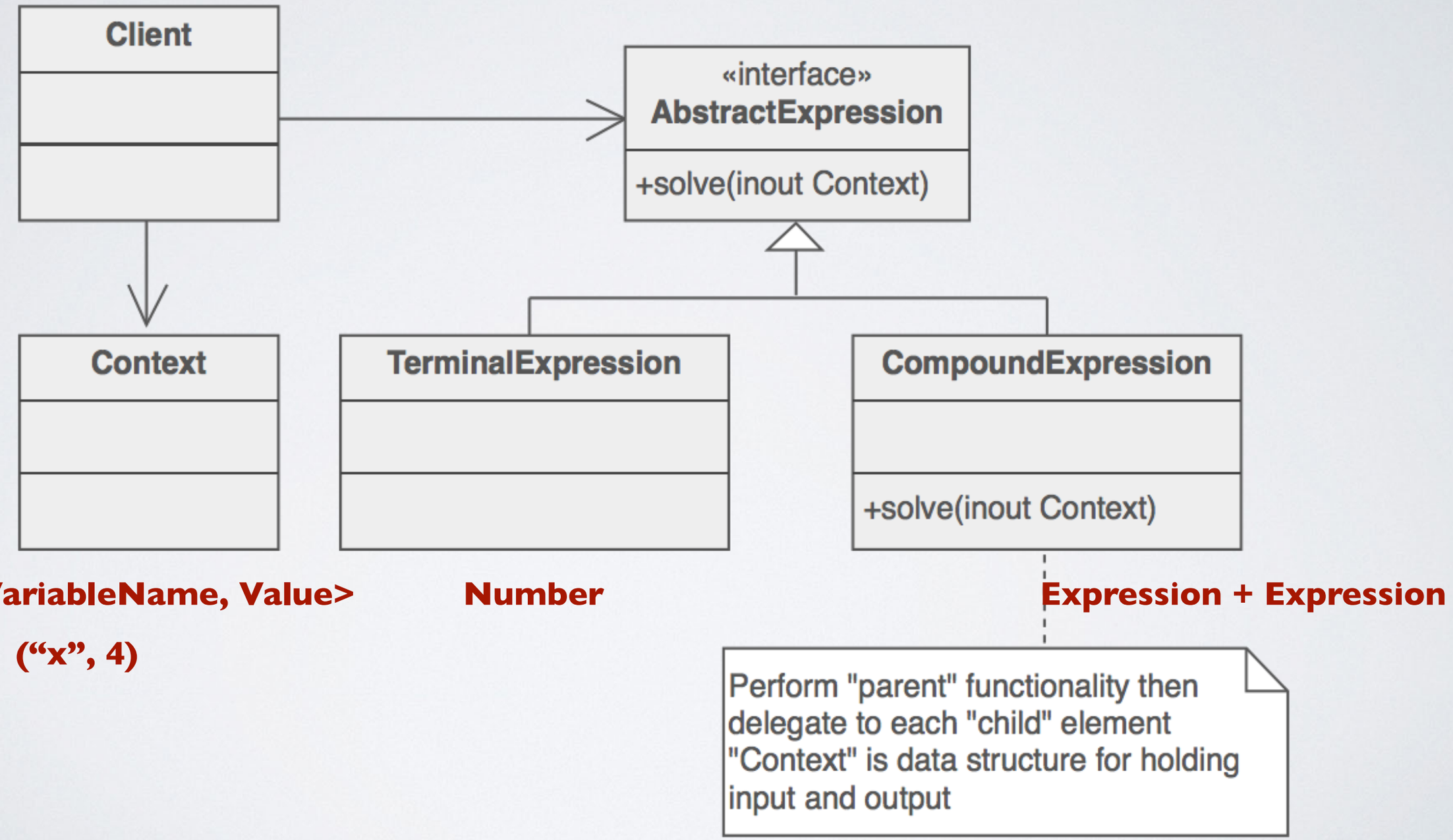
PROBLEM

```
$ python
>>> x = 4;
>>> 3 + x
```

INTERPRETER

Interpreter – Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences from the language.

INTERPRETER



INTERPRETER

```
interface Expression {
    public int interpret(Map<String,Expression> variables);
}
```

```
class Plus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Plus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(Map<String,Expression> variables)  {
        return leftOperand.interpret(variables) + rightOperand.interpret(variables);
    }
}
```

INTERPRETER

```
interface Expression {
    public int interpret(Map<String,Expression> variables);
}


class Number implements Expression {
    private int number;
    public Number(int number)       { this.number = number; }
    public int interpret(Map<String,Expression> variables)  { return number; }
}


class Variable implements Expression {
  private String name;
  public Variable(String name)       { this.name = name; }
  public int interpret(Map<String,Expression> variables)  {
   return !variables.get(name) ? 0 : variables.get(name).interpret(variables);
  }
}
```

PROBLEM

Flexibly allow various (future) operations on your data structures.

VISITOR

Visitor – Represents an operation to be performed on the elements of an object structure.

Visitor lets you define a new operation without changing the classes of the elements on which it operations.

VISITOR

I have a new car that displays which parts are currently working in the system.

```
Engine is on!
```

VISITOR

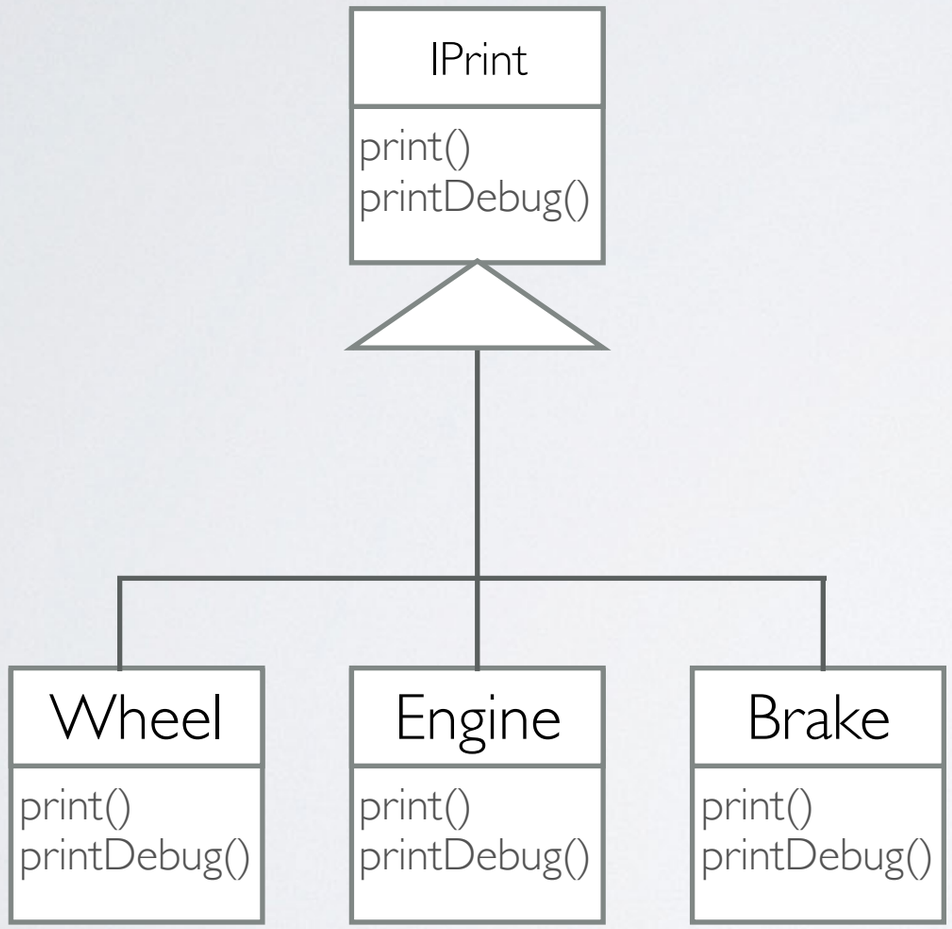
I have a new car that displays which parts are currently working in the system.

```
Engine is on!
```

Go to the garage for the annual revision:

```
Engine life 58%
```

VISITOR



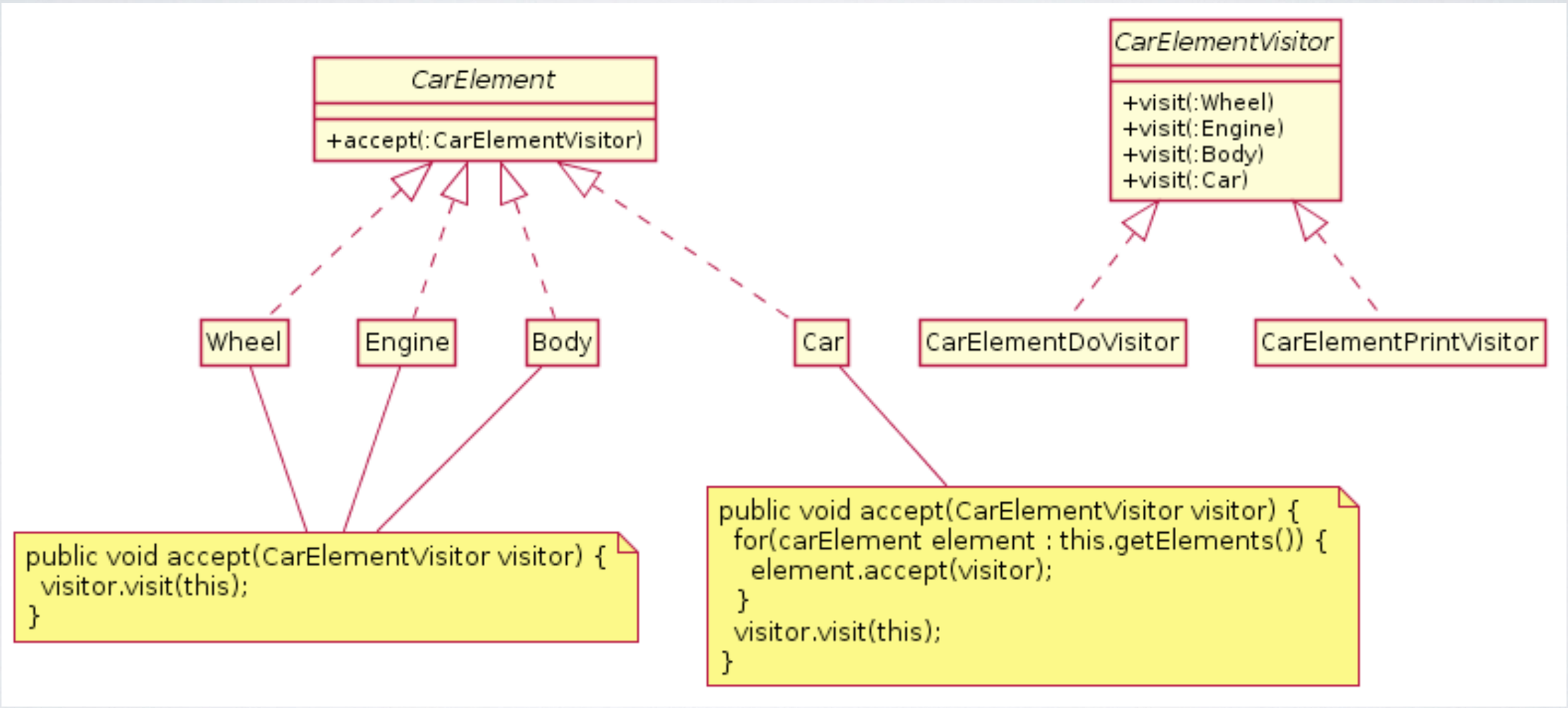
I have a new car that displays which parts are currently working in the system.

```
Engine is on!
```

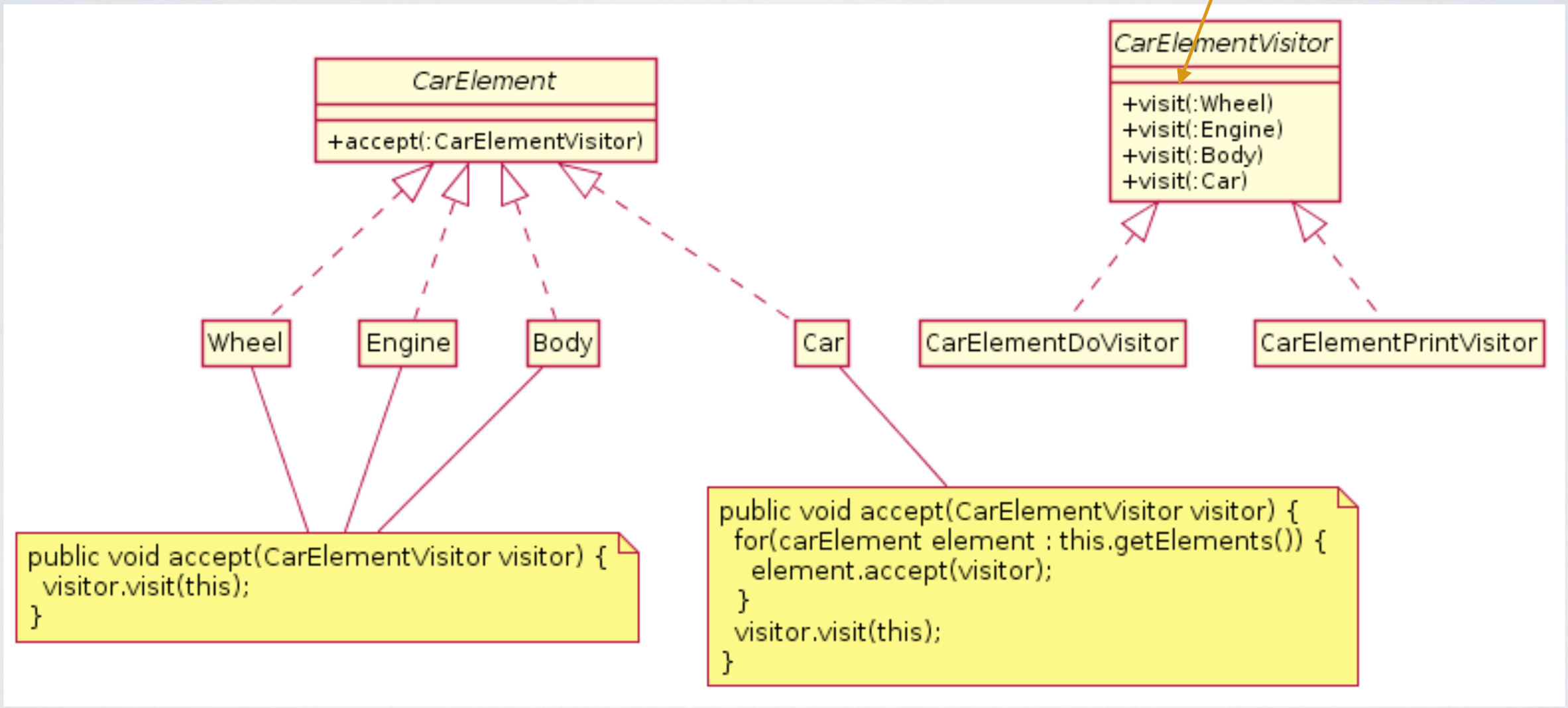
Go to the garage for the annual revision:

```
Engine life 58%
```

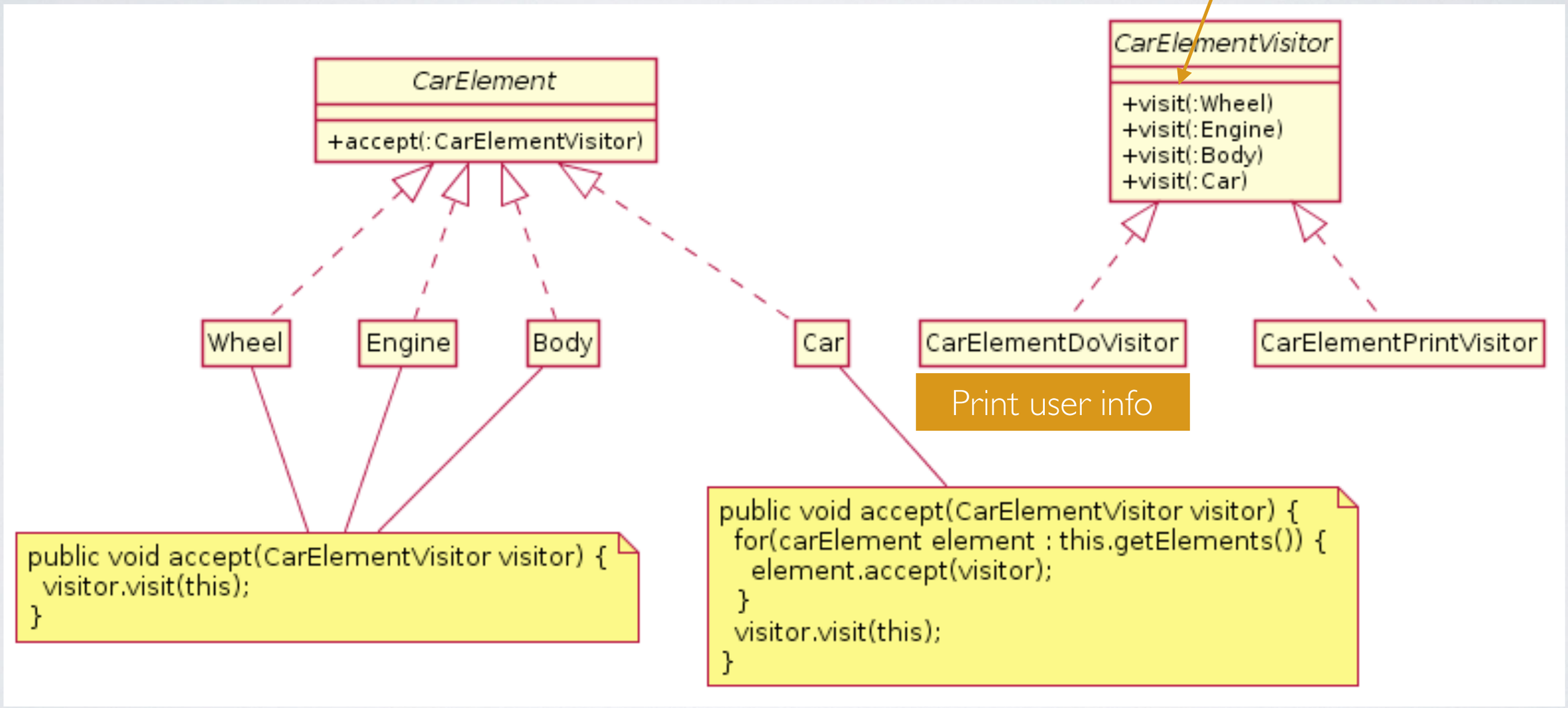
VISITOR


CarElement
+accept(:CarElementVisitor)
CarElementVisitor
+visit(:Wheel)
+visit(:Engine)
+visit(:Body)
+visit(:Car)
Wheel
Engine
Body
Car
CarElementDoVisitor
CarElementPrintVisitor
public void accept(CarElementVisitor visitor) {
visitor.visit(this);
}
public void accept(CarElementVisitor visitor) {
for(carElement element : this.getElements()) {
element.accept(visitor);
}
visitor.visit(this);
}

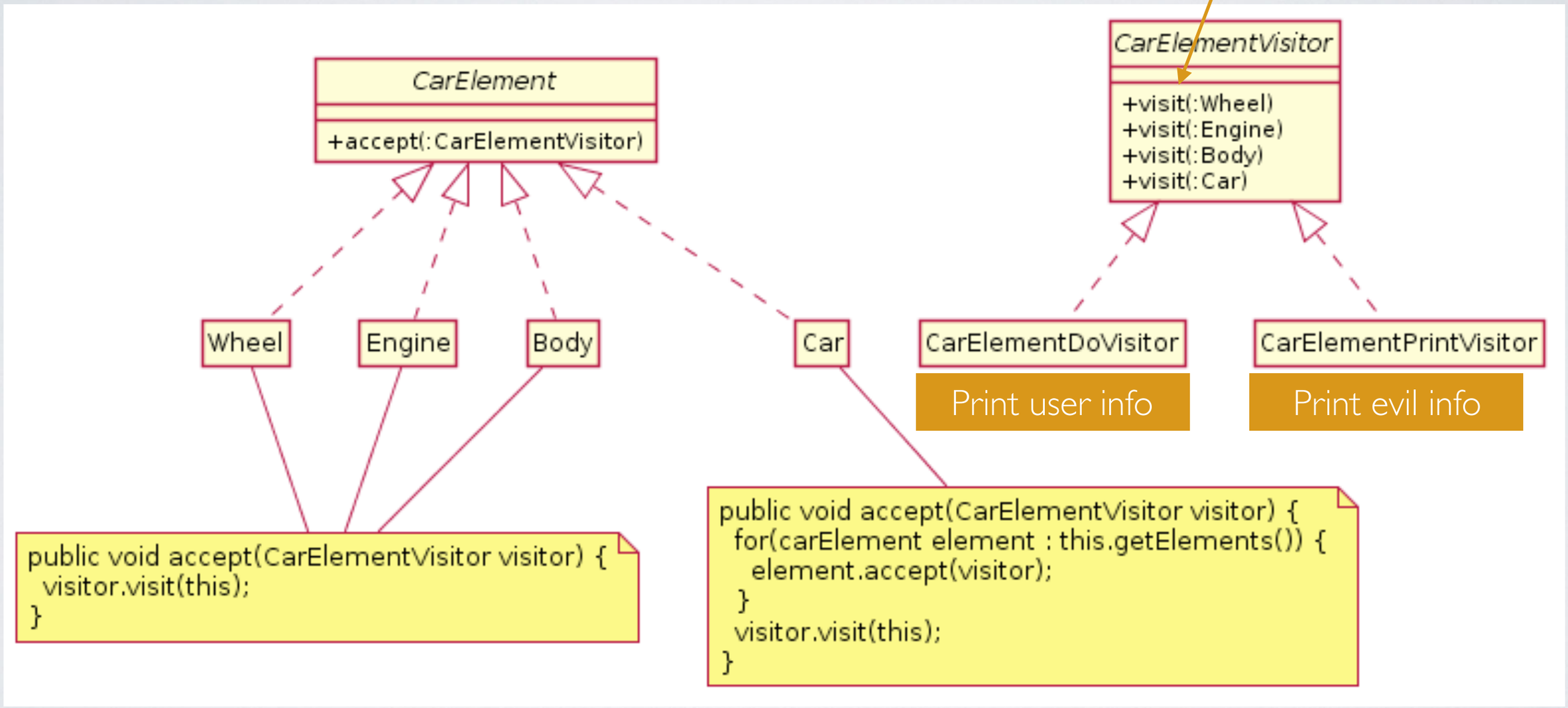
VISITOR


my printing operation
CarElement
+accept(:CarElementVisitor)
CarElementVisitor
+visit(:Wheel)
+visit(:Engine)
+visit(:Body)
+visit(:Car)
Wheel
Engine
Body
Car
CarElementDoVisitor
CarElementPrintVisitor
public void accept(CarElementVisitor visitor) {
visitor.visit(this);
}
public void accept(CarElementVisitor visitor) {
for(carElement element : this.getElements()) {
element.accept(visitor);
}
visitor.visit(this);
}

VISITOR


my printing operation
CarElement
+accept(:CarElementVisitor)
CarElementVisitor
+visit(:Wheel)
+visit(:Engine)
+visit(:Body)
+visit(:Car)
Wheel
Engine
Body
Car
CarElementDoVisitor
CarElementPrintVisitor
Print user info
public void accept(CarElementVisitor visitor) {
visitor.visit(this);
}
public void accept(CarElementVisitor visitor) {
for(carElement element : this.getElements()) {
element.accept(visitor);
}
visitor.visit(this);
}

VISITOR


my printing operation
CarElement
+accept(:CarElementVisitor)
CarElementVisitor
+visit(:Wheel)
+visit(:Engine)
+visit(:Body)
+visit(:Car)
Wheel
Engine
Body
Car
CarElementDoVisitor
CarElementPrintVisitor
Print user info
Print evil info
public void accept(CarElementVisitor visitor) {
visitor.visit(this);
}
public void accept(CarElementVisitor visitor) {
for(carElement element : this.getElements()) {
element.accept(visitor);
}
visitor.visit(this);
}

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

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

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

interface CarElement {
    void accept(CarElementV
}
```

```
class Wheel implements CarElement {
    private String name;

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

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

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

```
class Wheel implements CarElement {
    private String name;

    public Wheel(String name) {
        this.name = name;
```

```
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");
    }
}
```

```
sitor visitor) {
```

```
sitor visitor) {
```

Dynamic operation that
I want to perform

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

```
class Wheel implements CarElement {
    private String name;
```

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

    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 elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}
```

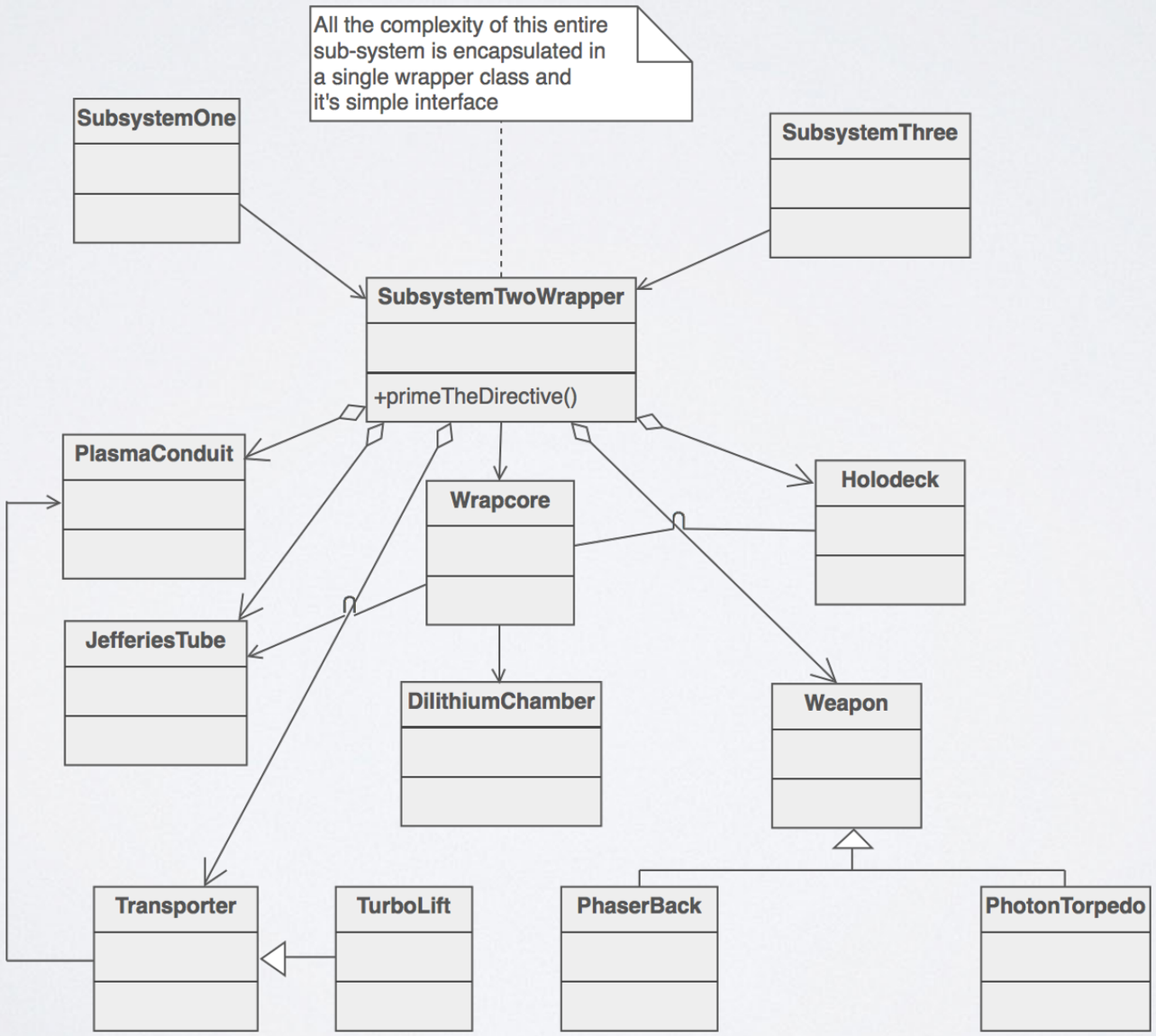
```
class CarElementPrintVisitor impl
    public void visit(Wheel wheel
        System.out.println("Visit
    }

    public void visit(Engine engi
        System.out.println("Visit
    }

    public void visit(Body body)
        System.out.println("Visit
    }

    public void visit(Car car) {
        System.out.println("Visit
    }
}
```

PROBLEM

Provide uniform interface to subsystem — avoid exposing subsystem implementation

FACADE

Facade – Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



SUMMARY

In this lecture and the previous one we learn a little about a lot of design patterns.

We glossed over a lot of details – these can be found in one of many patterns books or online.

More examples in sourcemaking.com