Software Design & Programming Techniques

Design Patterns

Prof. Dr-Ing. Klaus Ostermann

Based on slides by Prof. Dr. Mira Mezini

2.1 Introduction to Design Patterns

Life is filled with patterns – many of which we overlook due to the business of our days – however once you get an eye for spotting them (and it takes being intentional and some practice) you'll be amazed by what you see and you'll wonder why you didn't incorporate them into your photography before.

Read more: http://digital-photography-school.com/using-repetition-and-patterns-in-photography#ixzz0V2dVjxjO

What is the Pattern in the Patterns Here?

While repetition in the humdrum of daily life can at times be a boring – capturing it in your photography can create an image with real impact. ... you can either emphasize it or break it.

Read more: http://digital-photography-school.com/using-repetition-and-patterns-in-photography#ixzz0V316Sfan

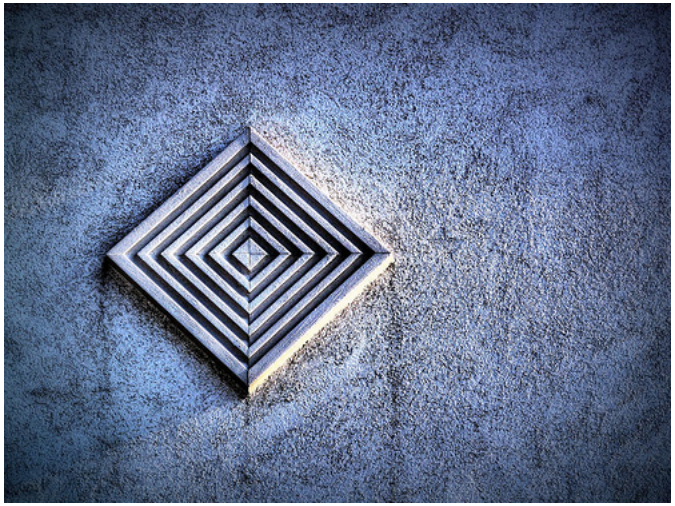
What's the Pattern Here?

Shallow Background

The depth of field that you select when taking an image will drastically impact the composition of an image. It can isolate a subject from its background and foreground (when using a shallow depth of field) or it can put the same subject in context by revealing it's surrounds with a larger depth of field.

Read more: http://digital-photography-school.com/5-elements-of-composition-in-photography#ixzz0V2ysR0Tg

What's the Pattern Here?



With clever use of 'texture' images become almost three dimensional.
Texture particularly comes into play when light hits objects at interesting angles.

Read more: http://digital-photography-school.com/5-elements-of-composition-in-photography#ixzz0V2yBLekA

What's the Pattern Here?

Lines can add dynamic impact to a photograph in terms of mood as well as how they lead an image's viewer into a photo.

Read more: http://digital-photography-school.com/working-the-lines-in-your-photography#ixzz0V2wyPgSC

What is a Pattern?

A design pattern describes:

- A **problem that occurs over and over again** in our environment.
- The **core of the solution to that problem**, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander, professor of architecture.

What is a Pattern?

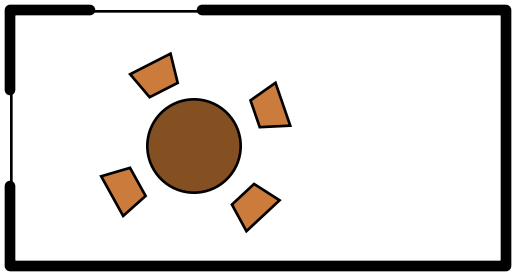
Aggressive disregard for originality.

Rule of three:

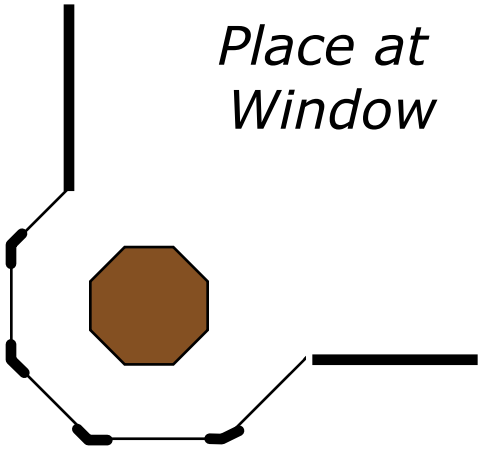
- Once is an event.
- Twice is an incident.
- Thrice it’s a pattern.

Patterns in Architecture

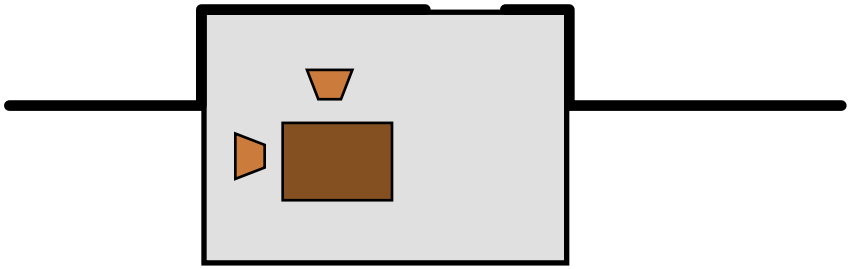
Light from two sides



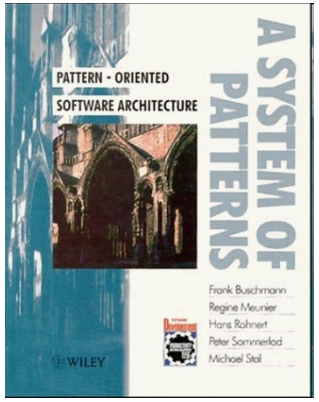
Place at Window

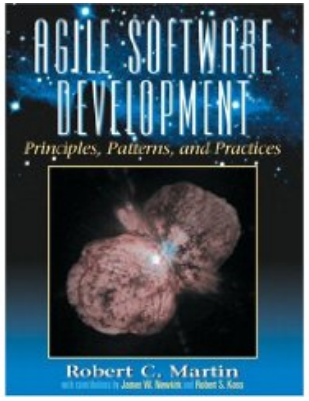


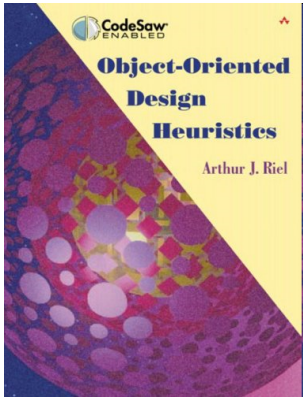
Deep terrace

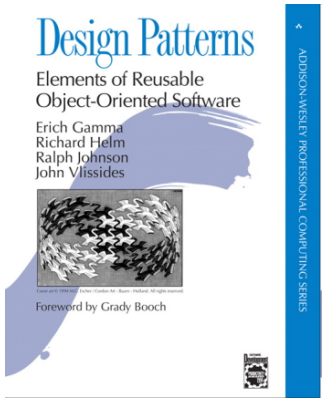


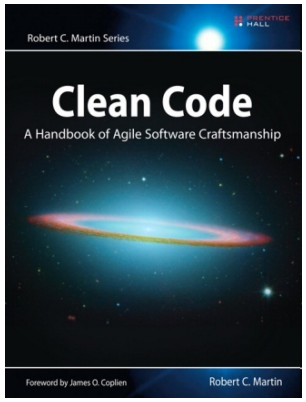
Sofware Patterns

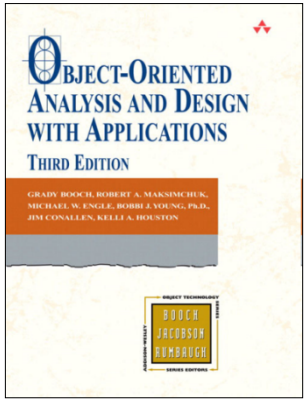


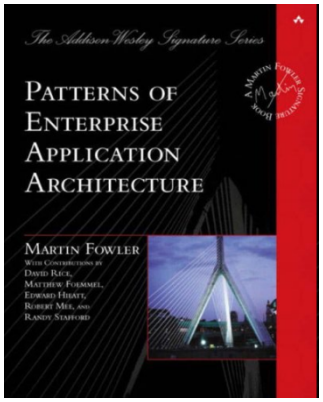










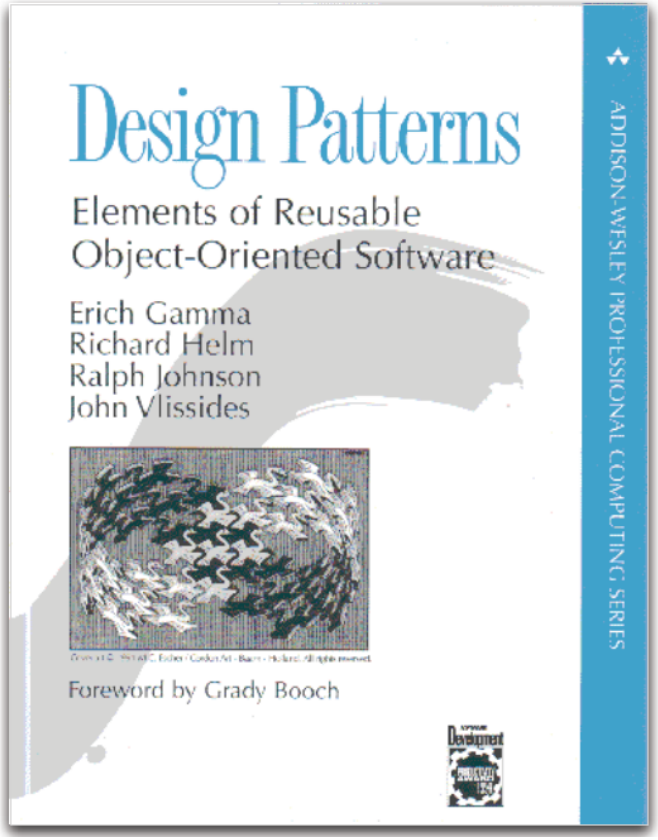


Motivation for Software Design Patterns

- **Designing reusable software is hard**!
  (Originality is Overrated)
  - Novices are overwhelmed.
  - Experts draw from experience.

- **Some design solutions reoccur**.
  Understanding their core is beneficial.
  - Know when to apply.
  - Know how to establish them in a generic way.
  - Know the consequence (trade-offs).

- **Systematic software-development**
  - Documenting expert knowledge.
  - Use of generic solutions.
  - Use of shared vocabulary.
  - Raising the abstraction level.



Design Patterns and Change

- **Most patterns address issues of software change.**
- Most patterns allow some part of the system to vary independent of the other parts.
- We often try to identify what varies in a system and encapsulate it.

R. Martin's Chess Analogy

When people begin to play chess they learn the rules and physical requirements of the game. They learn the names of the pieces, the way they move and capture, the board geometry and orientation.
At this point, people can play chess, although they will probably be not very good players.

As they progress, they learn the principles. They learn the value of protecting the pieces, and their relative value. They learn the strategic value of the center squares and the power of a threat…

At this point, they can play a good game. They know how to reason through the game and can recognize "stupid" mistakes.

However, **to become a master** of chess, one must **study games of other masters**. Buried in those games are patterns that must be understood, memorized, and applied repeatedly until they become second nature.

There are thousands upon thousands of these patterns. Opening patterns are so numerous that there are books dedicated to their variations. Midgame patterns and ending patterns are also prevalent, and the master must be familiar with them all.

R. Martin's Chess Analogy

So it is with software. **First one learns** the rules. The **algorithms, data structures and languages of software**.
At this point, one can write programs, albeit not very good ones.

Later, **one learns the principles of software design**. One learns the importance of cohesion and coupling, of information hiding and dependency management.

But **to truly master software design, one must study the designs of other masters.** Deep within those designs are patterns that can be used in other designs. Those patterns must be understood, memorized, and applied repeatedly until they become second nature.

Elements of Design Patterns

- **Pattern Name**
  A short mnemonic to increase your design vocabulary.

- **Intent**
  Description when to apply the pattern (conditions that have to be met before it makes sense to apply the pattern).

- **Solution**
  The elements that make up the design, their relationships, responsibilities and collaborations.

- **Consequences**
  Costs and benefits of applying the pattern. Language and implementation issues as well as impact on system flexibility, extensibility, or portability.
  The goal is to help understand and evaluate a pattern.

GOF Design Patterns Overview

Abstract Factory	Composite	Interpreter
Factory method	**Decorator**	Iterator
Builder	Facade	Mediator
Prototype	Flyweight	Memento
Singleton	Proxy	Null Object
Adapter	Chain of Responsibility	Observer
Bridge	**Command**	State
Strategy	**Template method**	**Visitor**

Taught here

Design Patterns

- 2.1 Introduction to Design Patterns
- 2.2 Quick Warm Up with Template Method
- 2.3 The Strategy Pattern
- 2.4 Decorator
- 2.5 Decorator vs. Strategy
- 2.6 Bridge
- 2.7 Visitor
- 2.8 Adapter
- 2.9 Builder
- 2.10 Command

2.2 Quick Warm Up with Template Method

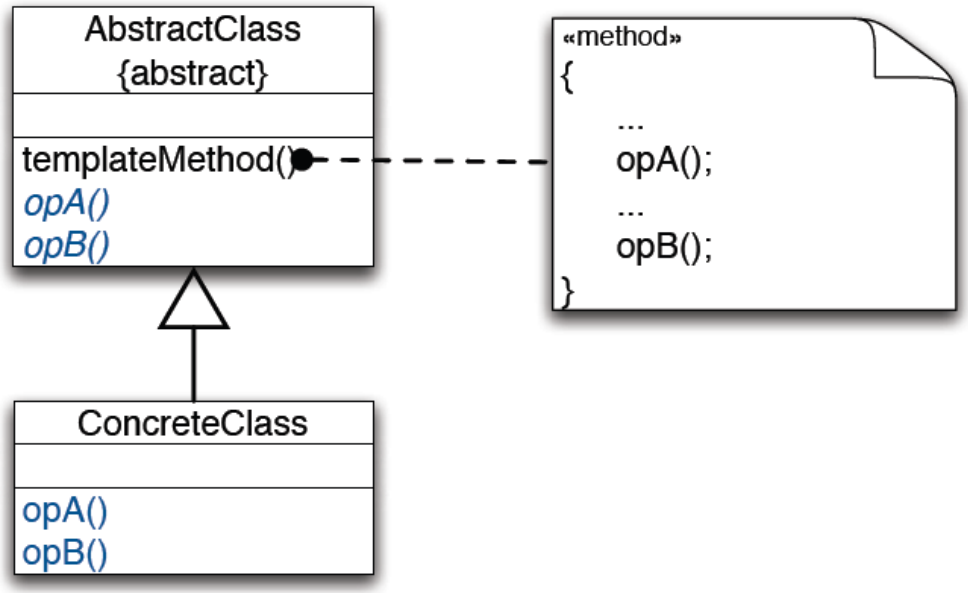
The Template Pattern in a Nutshell

Intent:

- Separate policies from detailed mechanisms.
- Separate invariant and variant parts.

Solution Idea

- Use abstract classes to
  - Define interfaces to detailed mechanisms and variant parts.
  - Implement high-level policies and invariant parts on top of these interfaces.
  - Control sub-class extensions.
  - Avoid code duplication.



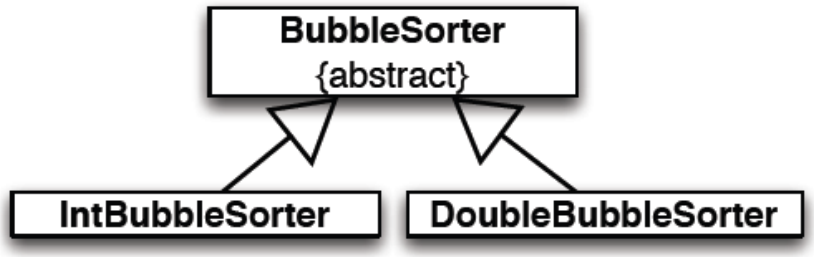
The Template Method Pattern plays a key role in the design of object-oriented frameworks.

Example Application of Template Method

- Need a family of sorting algorithms.
- Clients should be oblivious of (reusable with) the variety of specific algorithms.

- Need to separate the high-level policy of „sorting“ from low-level mechanisms
  - Deciding when an element is out of order
  - Swapping elements.

Separating the Policy of Sorting



Implement the policy in a template method.
Hide mechanisms needed for implementing the policy behind abstract methods which are called by the template method.

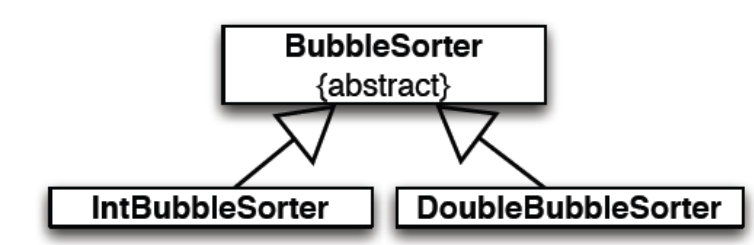
```
public abstract class BubbleSorter {

    protected int length = 0;

    protected void sort() {
        if (length <= 1) return;
        for (int nextToLast = length - 2; nextToLast >= 0; nextToLast--)
            for (int index = 0; index <= nextToLast; index++)
                if (outOfOrder(index))
                    swap(index);
    }

    protected abstract void swap(int index);
    protected abstract boolean outOfOrder(int index);
}
```

Policy

Mechanisms

Filling the Template for Specific Sorting Algorithms

```
public class IntBubbleSorter extends BubbleSorter {
  private int[] array = null;

  public void sort(int[] theArray) {
        array = theArray;
        length = array.length;
        super.sort();
  }
  protected void swap(int index) {
        int temp = array[index];
        array[index] = array[index + 1];
        array[index + 1] = temp;
  }
  protected boolean outOfOrder(int index) {
        return (array[index] > array[index + 1]);
  }
}
```



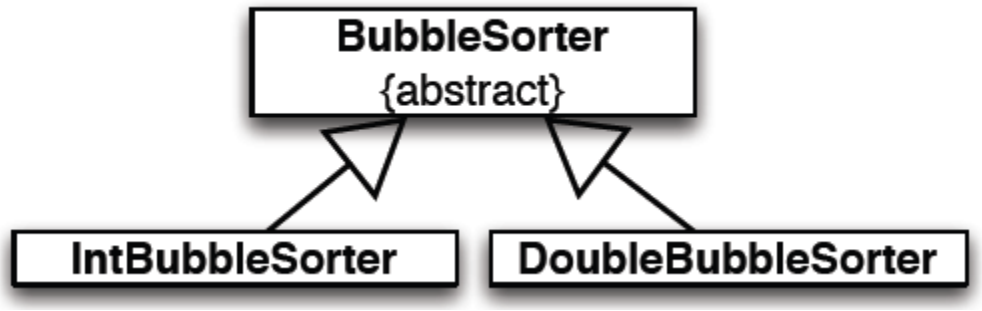
Functional Counterpart of Template

- One can look at the template method pattern as a style for emulating higher-order functions available in functional programming languages.
- Higher-order function: A function parameterized by other functions.
- First-order functions abstract over variations in data.
- Higher-order functions abstract over variations in sub-computations.
- First-class functions are values that can be passed as parameters and returned as results.
- Advantage of functional style: Template parameters can be varied per call, possibly dynamically. In OO style the number of variations is statically fixed.
  - FP style actually corresponds more to strategy pattern
- Advantage of OO style: Easier to provide default implementations of low-level methods

Advantages and Deficiencies of Template

- … basically those of inheritance …

Variations Cannot Be Reused



- **Template method forces detailed implementations to extend the template class.**

- The implementation of low-level mechanisms depends on the template.
- Cannot re-use low-level mechanisms functionality. `swap` and `out-of-order` implemented in `IntBubbleSorter` may be useful in other contexts as well, e.g., for quick sort.
- Problem addressed by mixin-based inheritance or traits (Squeak, Scala)

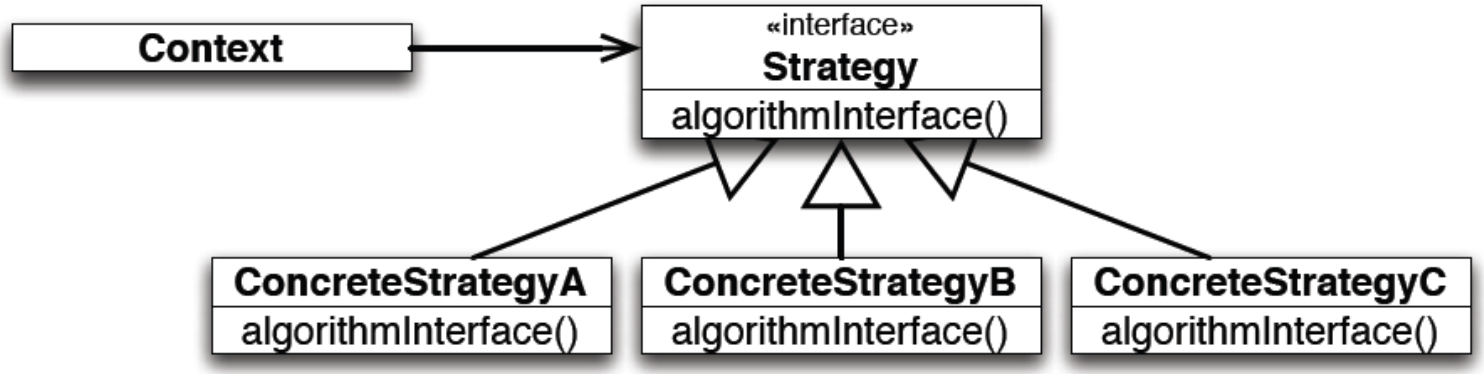
Design Patterns

2.3 The Strategy Pattern

2.3.1 The Strategy Pattern in a Nutshell

- **Intent:** Define a family of algorithms, encapsulate each one, and make them interchangeable.

- Strategy lets the algorithm vary independently from clients that use it.



When to Use the Strategy Pattern

- **Many related classes differ only in their behavior rather than implementing different related abstractions (types).**
  Strategies allow to configure a class with one of many behaviors.

- **You need different variants of an algorithm.**
  Strategies can be used when variants of algorithms are implemented as a class hierarchy.

- **A class defines many behaviors that appear as multiple conditional statements in its operations.**
  Move related conditional branches into a strategy.

Strategy Illustrated

```
class Table extends Widget {
  TableCB cb;
  ...
  void setClipboard(TableCB clipboard) { cb = clipboard; }
  void keyPressed(KeyEvent e) {
    super.keyPressed(e);
    cb.keyPressed(e);
  }
  void copy() { cb.copyToClipboard(); }
  ...
}

abstract class TableCB {
   Table table;
   void keyPressed(KeyEvent e) {
      if (e.getKeyCode() == VK_COPY) { copyToClipboard(); }
      ...
   }
   abstract void copyToClipboard();
   ...
}
```

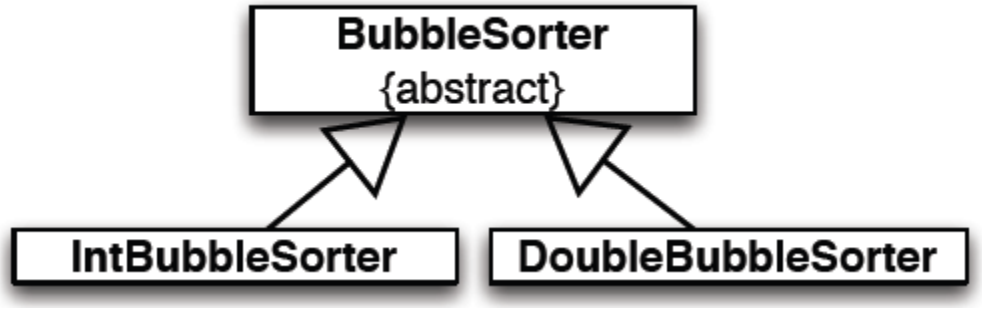
Strategy Illustrated

```
class TableAppCB extends TableCB {
   void copyToClipboard() {
     AppClipboard.setText(table.getCellText(currRow, currCol));
   }
   ...
}

class TableSystemCB extends TableCB {
   void copyToClipboard() {
       /* copy to the system clipboard */ }
   ...
}
```

Strategy as an Alternative to Inheritance

- The strategy pattern represents an alternative to modeling different algorithms (sub-behaviors) as subclasses of Context.
- Inheritance mixes algorithm‘s implementation with that of Context. Context may become harder to understand, maintain, extend.
- Inheritance results in many related classes.
  Only differ in the algorithm or behavior they employ.
- **When using subclassing we cannot vary the algorithm dynamically**.
- Encapsulating the algorithm in Strategy:
  - Lets you vary the algorithm independently of its context.
  - Makes it easier to switch, understand, and extend the algorithm.

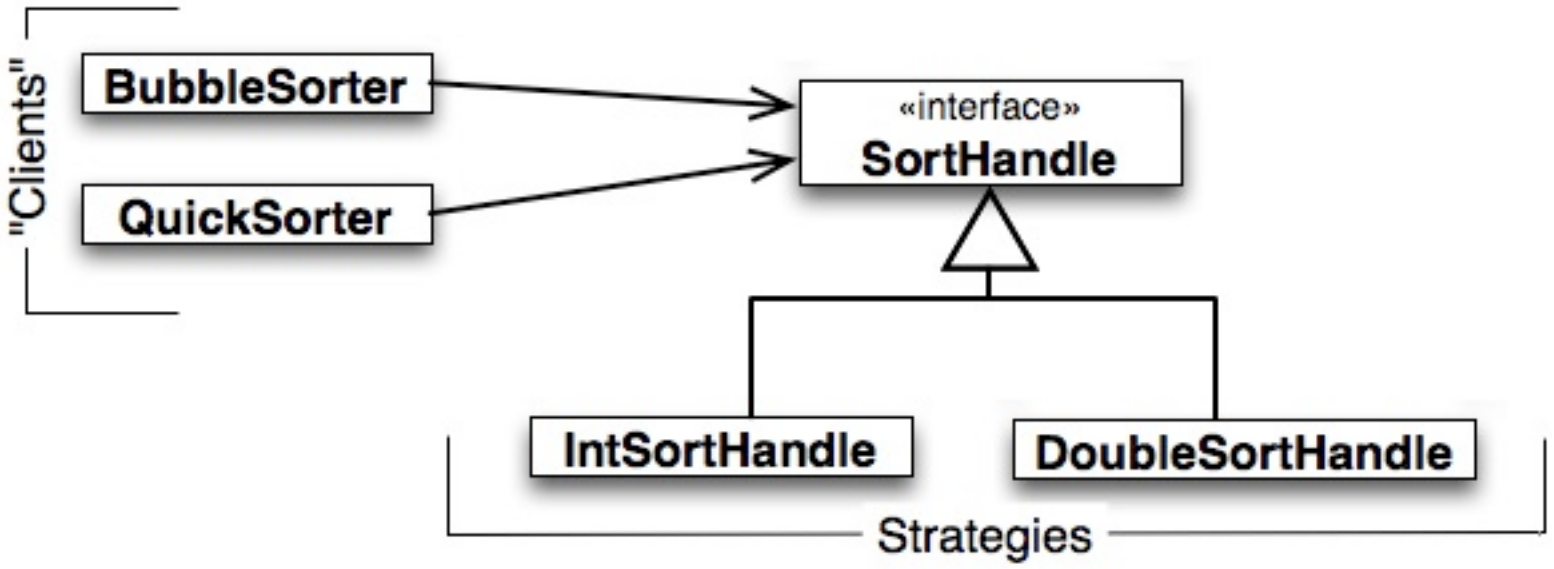
Sorting Example Revisited



Can we solve the reusability problems with Strategy?

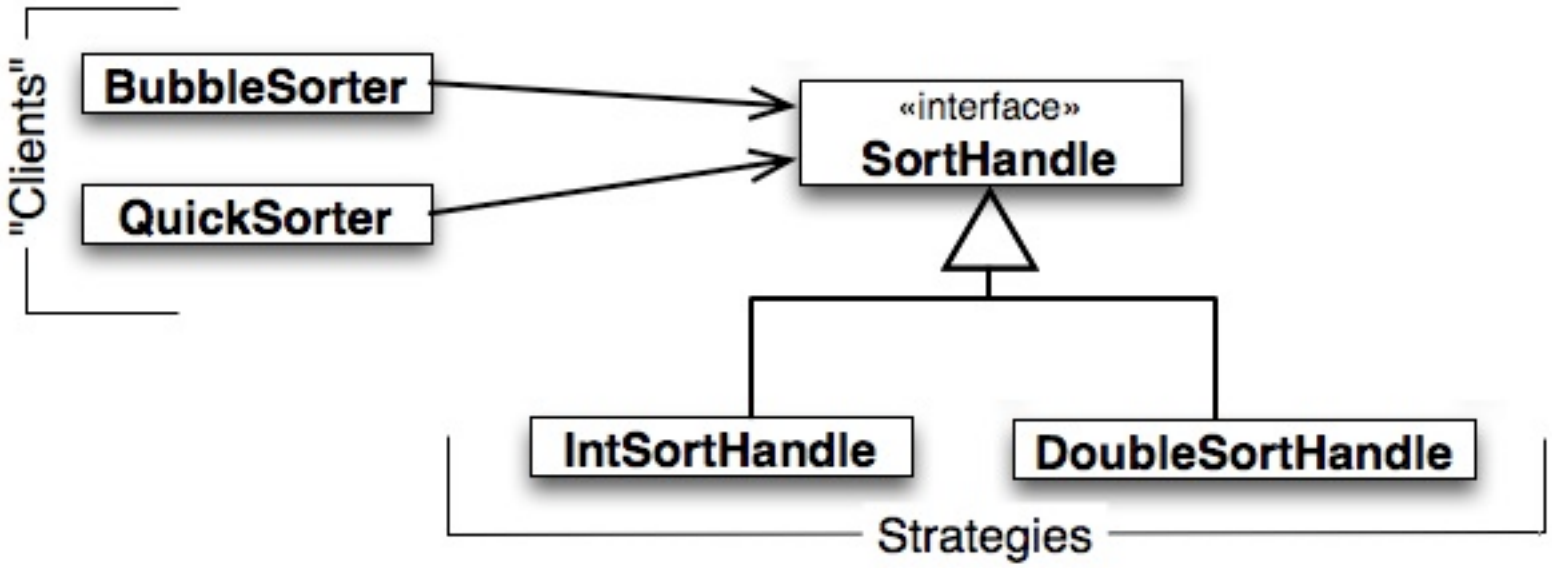
- **Template method forces detailed implementations to extend the template class.**
- Detailed implementation depend on the template.
- Cannot re-use detailed implementations‘ functionality.
  `swap` and `out-of-order` implemented in `IntBubbleSorter` may be useful in other contexts as well, e.g., for quick sort.

Sorting Example Revisited



- `BubbleSorter` and `QuickSorter` embodies different policies for sorting the elements of a list.
- They use `SortHandle` to decide about the concrete element ordering and swapping mechanisms.

Sorting Example Revisited



- Not only are sorting policies reusable with different ordering and swapping mechanisms; The latter become reusable with different sorting policies.

- **DIP**: High-level policies should not depend on low-level mechanisms. Both should depend on abstractions.

Functional Counterpart of Strategies

- Every function passed as an argument to a higher-order function can be considered a strategy
- More sophisticated application of the idea: type classes in Haskell
  - A dictionary of functions is passed implicitly as an argument to a type-class polymorphic function
  - E.g. `avg :: Fractional a => a -> a > a`
    `avg x y = (x + y) / 2`

Clients Need to be Aware of Variations

- Clients must be aware of different strategies and how they differ, in order to select the appropriate one.
- Clients might be exposed to implementation issues.
- Use Strategy only when the behavior variation is relevant to clients.

- Optional Strategy objects.
- Context checks if it has a Strategy before accessing it.
  - If yes, Context uses it normally.
  - If no, Context carries out default behavior.
- Benefit: clients don't have to deal with Strategy objects; unless they don't like the default behavior.

Footprint of Variations in Base Functionality

```
class Table extends Widget {
  TableCB cb;
  ...
  void setClipboard(TableCB clipboard) { cb = clipboard; }
  void keyPressed(KeyEvent e) {
    super.keyPressed(e);
    cb.keyPressed(e);
  }
  void copy() { cb.copyToClipboard(); }
  ...
}
```

- The clipboard functionality leaves its footprint in the interface and the implementation of the class Table:
  - Methods to manage strategy objects: setClipboard
  - Facade methods forwarding functionality to strategy: copy

- There may be clients not interested in clipboard functionality.
- Violations of SRP and ISP

An Example „One Size Fits All“ Strategy Interface

- When we outsource the implementation of a variable feature into a strategy object, we need to design a fixed interface to the strategy that fits all possible variations of the outsourced feature.

- This can lead to bloated interfaces which might be too complicated for some clients not interested in sophisticated variations of a feature.

An Example „One Size Fits All"-Interface

Consider the list selection feature of Swing's `JTable` outsourced to class `ListSelectionModel`

```
interface ListSelectionModel {
  int SINGLE_SELECTION = 0;
  int SINGLE_INTERVAL_SELECTION = 1;
  int MULTIPLE_INTERVAL_SELECTION = 2;

  /** ...
   * In {@code SINGLE_SELECTION} selection mode,
   * this is equivalent to calling {@code setSelectionInterval},
   * and only the second index is used.
   * In {@code SINGLE_INTERVAL_SELECTION} selection mode,
   * this method behaves like {@code setSelectionInterval},
   * unless the given interval is immediately
   * adjacent to or overlaps the existing selection,
   * and can therefore be used to grow the selection.
   * ...
   */
   void addSelectionInterval(int index0, int index1);
   ...
}
```

An Example „One Size Fits All"-Interface

- The interface of `ListSelectionModel` is designed to satisfy the needs of the most flexible selection model (multiple interval selection).
- As a result, the interface is too complicated for clients of simpler selection models.
- See the comments of the methods in the interface.
- Yet, the design is not flexible enough, e.g., to handle arbitrary cell range selection…

Use Strategy for features, whose variations do not affect the interface.

Communication Overhead

- A Strategy interface is shared by all concrete Strategy classes whether the algorithms they implement are trivial or complex.

- Some concrete strategies won't use all the information passed to them (Simple concrete strategies may use none of it.)
  (Context creates/initializes parameters that never get used.)

  If this is an issue use a tighter coupling between Strategy and Context; let Strategy know about Context.

Giving Strategy Visibility for the Context

- Two possible approaches:
  - **Pass the needed information as a parameter.**
    - Context and Strategy decoupled.
    - Communication overhead.
    - Algorithm can’t be adapted to specific needs of context.
  - **Context passes itself as a parameter or Strategy has a reference to its Context.**
    - Reduced communication overhead.
    - Context must define a more elaborate interface to its data.
    - Closer coupling of Strategy and Context.

„One Size Fits All“ Strategy Interface

Use Strategy for features whose variations do not affect the interface.

Increased Number of Objects

- Potentially many strategy objects need to be instantiated!
- The number of strategy objects can sometimes be reduced by stateless strategies that several Contexts can share.
- Any state is maintained by Context.
- Context passes it in each request to the Strategy object.
  (No / less coupling between Strategy implementations and Context.)
- Shared strategies should not maintain state across invocations.
  (Services)

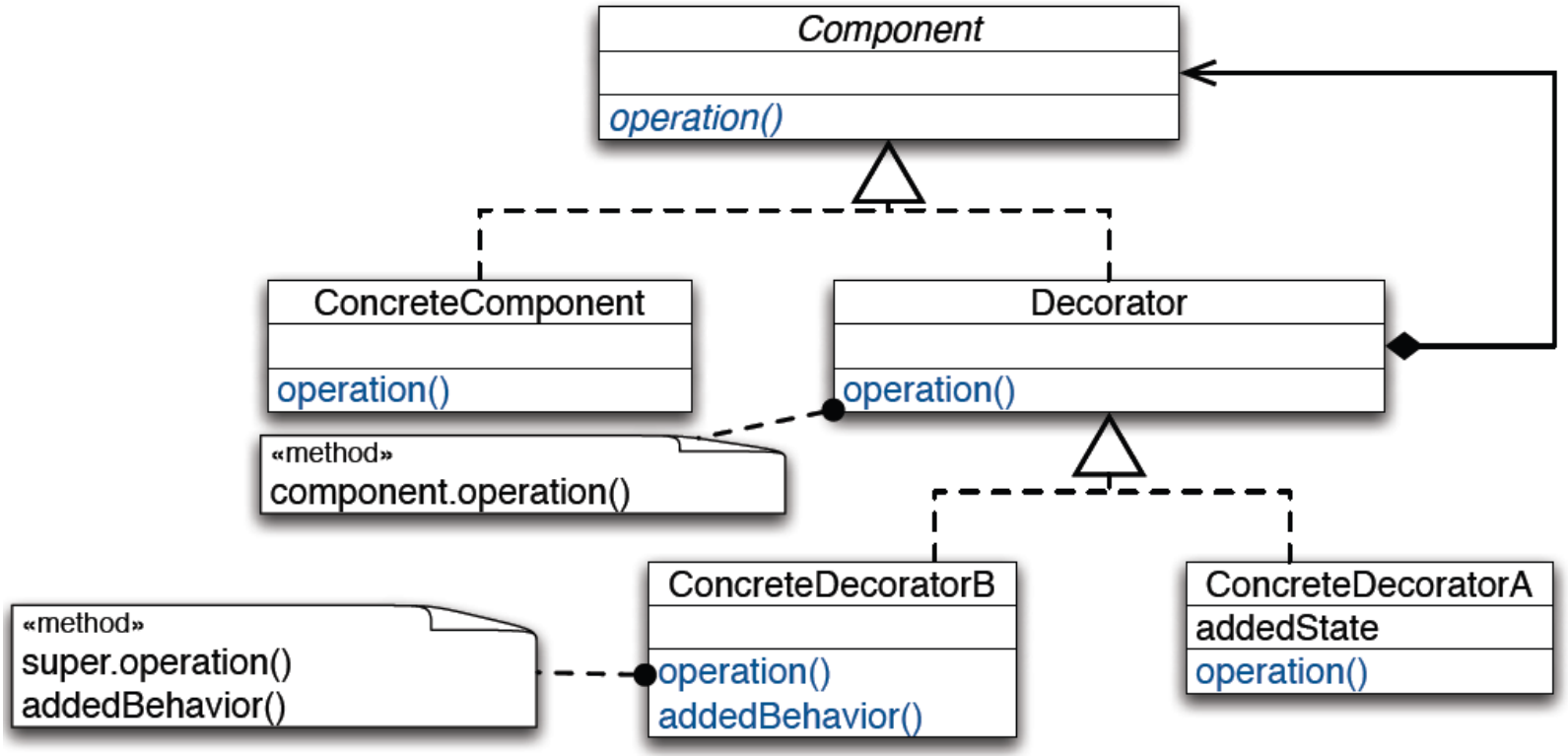
2.4 Decorator

2.4.1 The Intent of Decorator

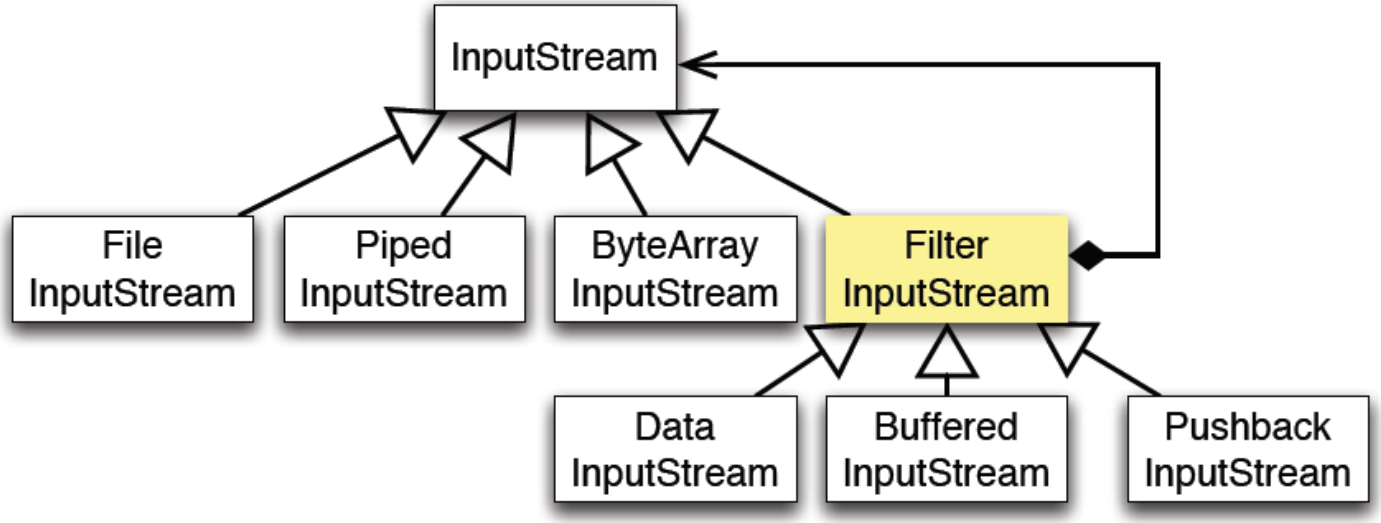
- **Intent:** We need to add functionality to existing objects dynamically and transparently, without affecting other objects.
  - Dynamically meaning during runtime.
  - Transparently meaning without having to implement conditional logic to use the new functionality.

- Usual way off adding new functionality to a existing design: Inheritance

2.4.2 Decorator Structure



2.4.3 Example



- `java.io` abstracts various data sources and destinations, as well as processing algorithms:
  - Programs operate on stream objects.
  - Independently of ultimate data source / destination / shape of data.

- Code example:

```
new DataInputStream(new FileInputStream(file)).readUnsignedByte()
```

2.4.4 Advantages

- Decorator enables more flexibility than inheritance:
  - Functionality can be added / removed at run-time.
  - Different Decorator classes for a specific Component class enable to mix and match responsibilities.

- Easy to add a functionality twice.
  E.g., for a double border, attach two `BorderDecorators`.

- Helps to design software that supports OCP.

Decorator and cohesion

- **Decorator avoids incoherent classes:**
  - Feature-laden classes high up in the hierarchy.
    (This also breaks encapsulation)
  - Pay-as-you-go approach do not bloat, but extend using fine-grained Decorator classes
    - Functionality can be composed from simple pieces.
    - An application does not need to pay for features it does not use.
  - A fine-grained Decorator hierarchy is easy to extend.

2.4.5 Problems

- **Lots of little objects**
  - A design that uses Decorator often results in systems composed of lots of little objects that all look alike.
  - Objects differ only in the way they are interconnected, not in their class or in the value of their variables.
  - Such systems are easy to customize by those who understand them, but can be hard to learn and debug.

Object identity

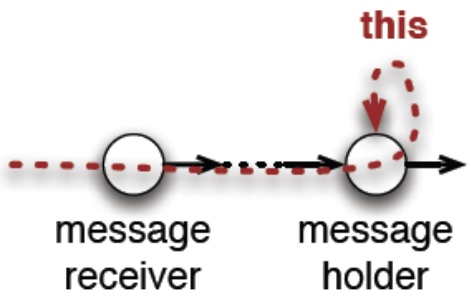
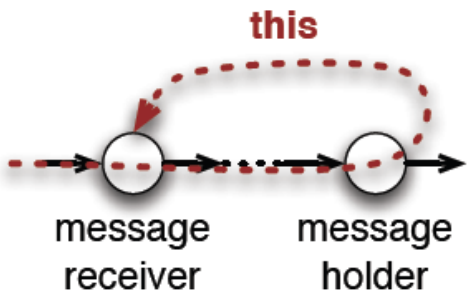
- A decorator and its component are not identical!
  - From an object identity point of view, a decorated component is not identical to the component itself.
  - You should not rely on object identity when you use decorators.

- Example:

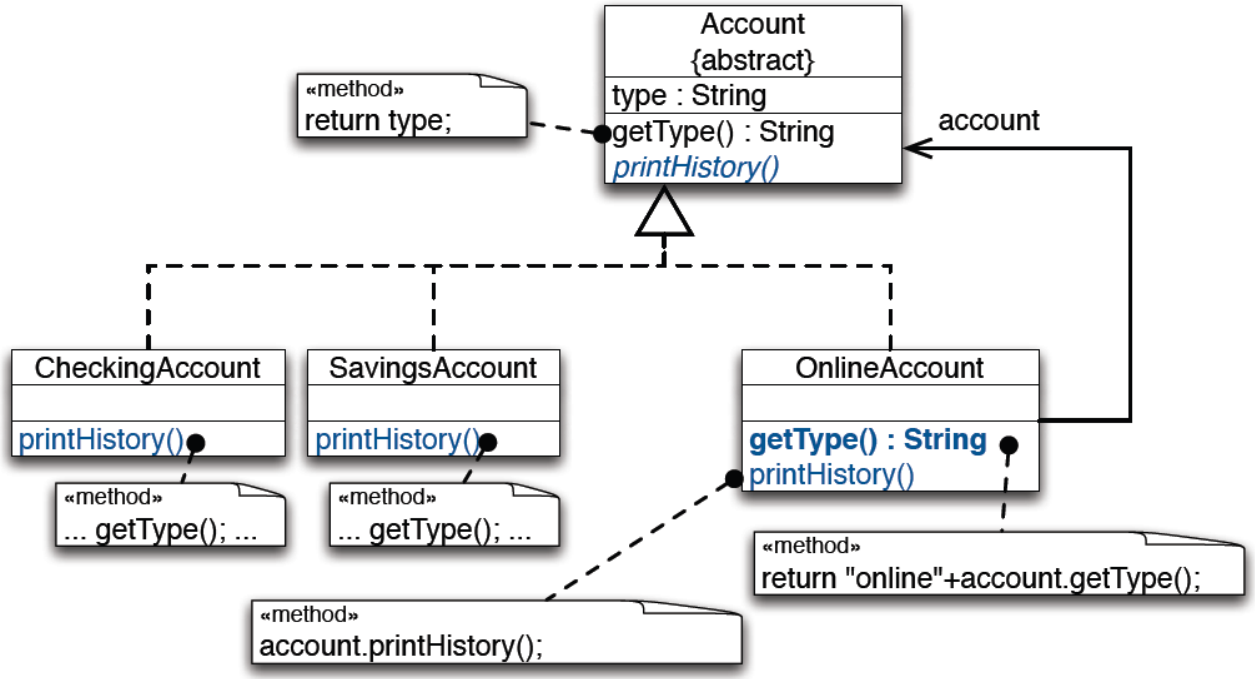
```
FileInputStream fin = new FileInputStream("a.txt");
BufferedInputStream din = new BufferedInputStream(fin);
fin.read();
```

No late binding

- Delegation vs. forward semantics.

Forwarding	Delegation
	
Forwarding with binding of `this` to method holder; "ask" an **object to do something on its own**.	Binding of `this` to message receiver: “ask” an object to **do something on behalf of the message receiver**.

Example

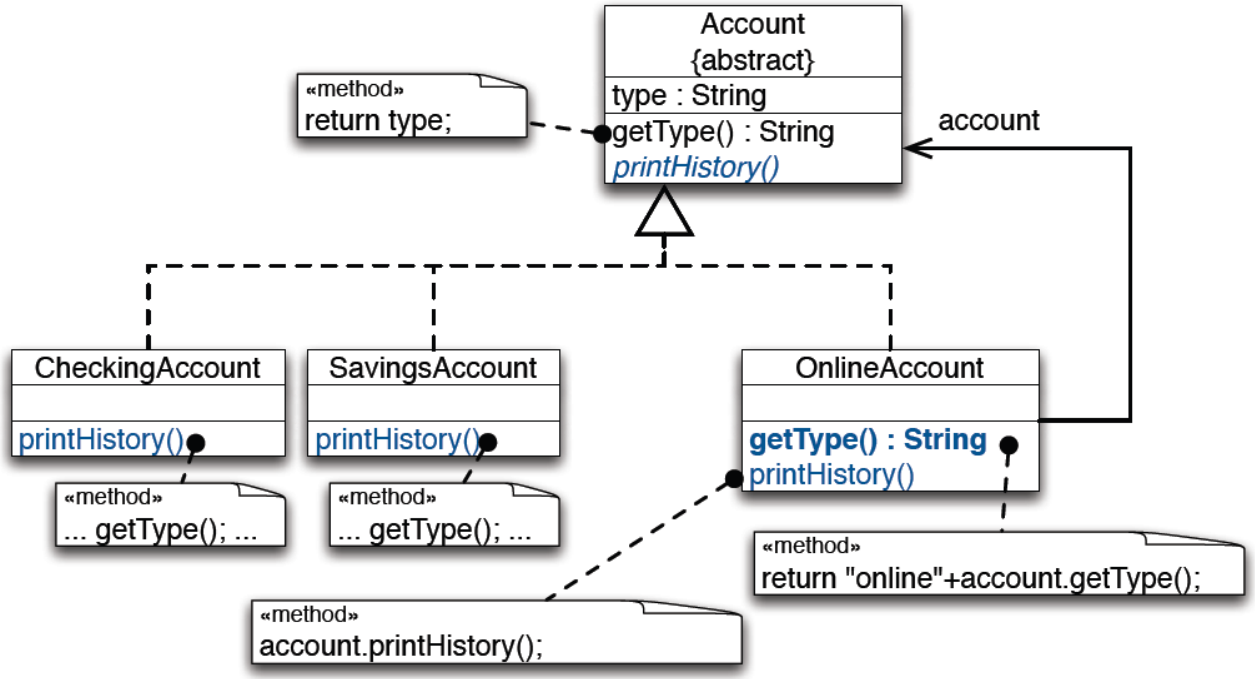


1) A checking account, `checkingAcc`, is created.
2) An online decorator, `onlineDec`, is created with `checkingAcc` as its attribute.
3) Call to `onlineDec.printHistory()`.

   a) Call to `checkingAcc.printHistory()` as the result of the forwarding by the call to `account.printHistory()` in the implementation of `OnlineDecorator.printHistrory()`.

   b) Execution of `CheckingAccount.printHistory()`.
   Call to `getType()` inherited from `Account`, not `OnlineAccount`!

Example



- `OnlineDecorator` decorates both `printHistory()` and `getType()`.
- Yet, since `CheckingAccount.printHistory()` calls `getType()` via this, the execution escapes the decoration of `getType()` in `OnlineDecorator`.

2.4.6 Implementing Decorators

- **Keep the common class (`Component`) lightweight!**
  - It should focus on defining an interface.
  - Defer defining data representation to subclasses.
    Otherwise the complexity of Component might make the decorators too heavyweight to use in quantity.

- Putting a lot of functionality into Component makes it likely that subclasses will pay for features they do not need.

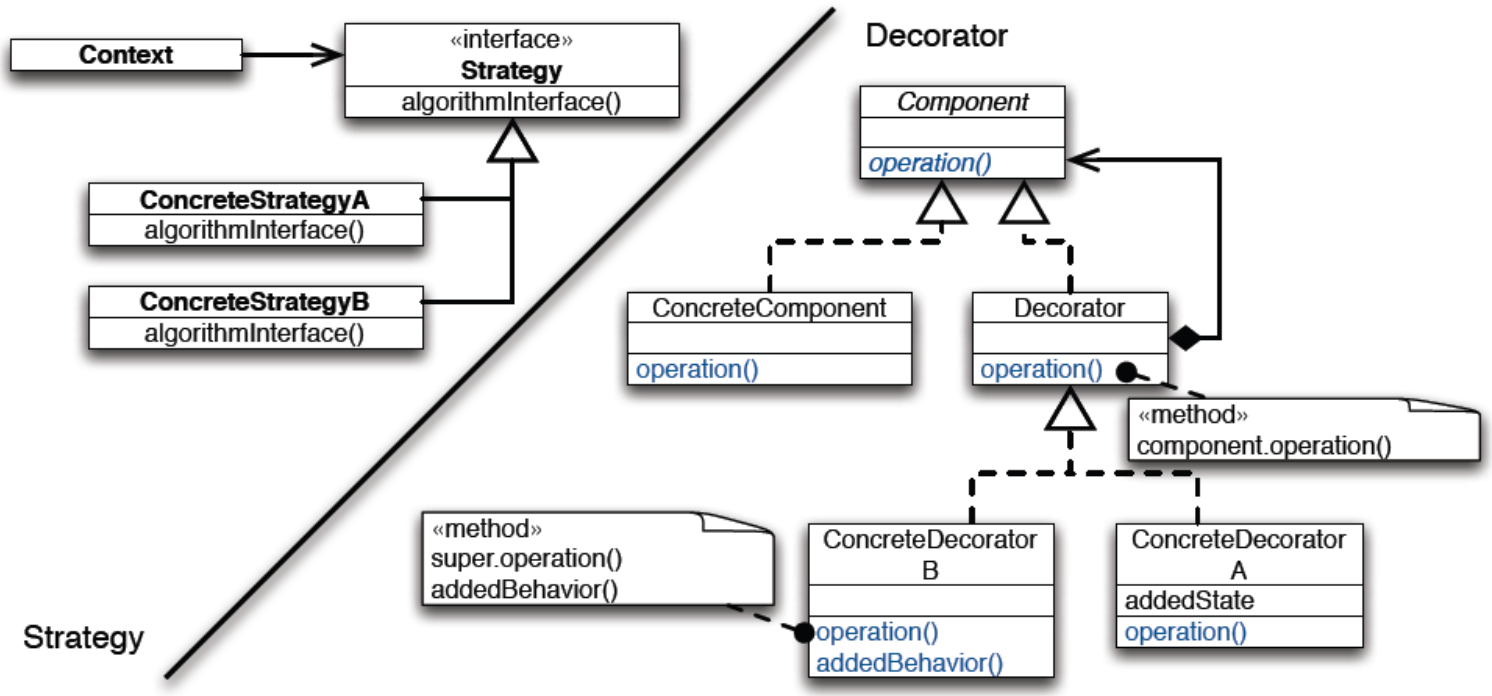
- These issues require pre-planning.
  Difficult to apply decorator pattern to 3rd-party component class.

Implementing Decorators

- A decorator's interface must conform to the interface of the component it decorates.
  Concrete Decorator classes must therefore:
  - Inherit from a common class (C++) or
  - Implement a common interface (Java).

- There is no need to define an abstract Decorator class when you only need to add one responsibility.
  - That's often the case when you're dealing with an existing class hierarchy rather than designing a new one.
  - Can merge Decorator's responsibility for forwarding requests to the component into the concrete Decorator.

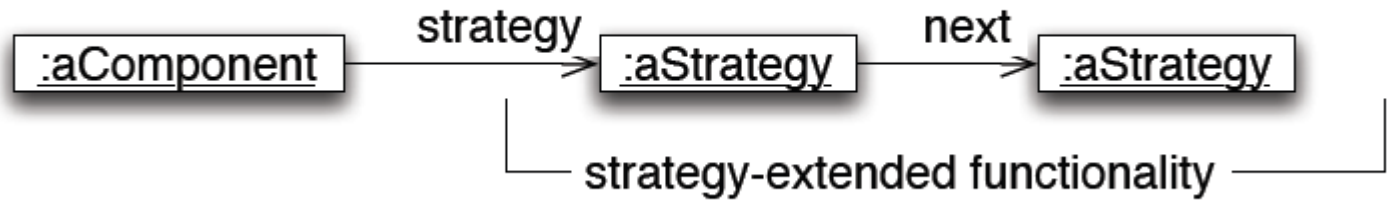
2.5 Decorator vs. Strategy



- Decorator and strategy share the goal of supporting dynamic behavior adaptation.
  - Can be used to simulate the effect of each other.
  - So, when to use them?

Discussion

- By extending the number of strategies from just one to an open-ended list, we achieve the same effect as nesting decorators recursively.



- Using `Strategy` to simulate border decoration of text fields:
  Different border styles by having the component forward border-drawing to a `Border` object.
  (`Border` is a `Strategy` that encapsulates a border-drawing strategy.)

Discussion

- Decorator changes a component from the outside.
  - The component does not know about its decorators.
  - The decorators are transparent to the component.

- Component knows about Strategy-based extensions:

Skin and guts

- Changing an object's skin versus changing its guts:
  - Decorator can be viewed as a skin over an object that changes its behavior.
  - Strategy can be viewed as guts inside an object that changes its behavior.

Interim Take Away

- Strategy is better when Component is intrinsically heavyweight, because:
  - Decorator is too costly to apply
  - Single Responsibility Principle

- A Decorator's interface must conform to Component's interface.
- A Strategy can have its own specialized interface.
  E.g., a strategy for rendering a border need only define the interface for rendering a border (`drawBorder()`, `getWidth()`, …).

- Strategy can be lightweight even if the Component class is heavyweight.

2.5.1 Solution to Fragile Base-Class Problem?

- Decorator is suggested in many „OO with Style"-Books (e.g., Effective Java, by Bloch) as a solution to the fragile base class problem.

- Instead of inheriting from a class C consider:
  - Declare the interface of C, IC
  - Let C implement IC
  - Implement the extension of C in a class CD that implements IC and at the same time has a ic reference to an object of type IC
  - CD reimplements methods in IC affected by the extension and forwads the rest to ic.

An Alternative InstrumentedHashSet

```
import java.util.*;
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;

    public ForwardingSet(Set<E> s) { this.s = s; }
    public void clear() { s.clear();}
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty(){ return s.isEmpty();}
    public int size(){ return s.size();}
    public Iterator<E> iterator(){ return s.iterator();}
    public boolean add(E e){ return s.add(e);}
    public boolean remove(Object o) { return s.remove(o);}
    public boolean containsAll(Collection<?> c) { ... }
    public boolean addAll(Collection<? extends E> c) { ... }
    public boolean removeAll(Collection<?> c) {...}
    ...
}
```

An Alternative InstrumentedHashSet

```
import java.util.*;
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;
    public InstrumentedSet(Set<E> s) { super(s); }
    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c){
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
    public static void main(String[] args) {
        InstrumentedSet<String> s =
            new InstrumentedSet<String>(new HashSet<String>());
        s.addAll(Arrays.asList("aaa", "bbb", "ccc"));
        System.out.println(s.getAddCount());
    }
}
```

An Alternative InstrumentedHashSet

- Bloch mentions that disadvantages of the decorator-based solutions are few.
  - Self Problem
  - Tedious to write forewarding methods, „but you do it only once“.
  - Efficiency impact of forwarding and memory footprint -> but „neither turns out to have much impact in practice“
- What do you think?

An Alternative InstrumentedHashSet

```
import java.util.*;
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;

    public ForwardingSet(Set<E> s) { this.s = s; }
    public void clear() { s.clear();}
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty(){ return s.isEmpty();}
    public int size(){ return s.size();}
    public Iterator<E> iterator(){ return s.iterator();}
    public boolean add(E e){ return s.add(e);}
    public boolean remove(Object o) { return s.remove(o);}
    public boolean containsAll(Collection<?> c) { ... }
    public boolean addAll(Collection<? extends E> c) { ... }
    public boolean removeAll(Collection<?> c) {...}
    ...
}
```

What happens if I add a new method to the interface?
Doesn't the same problem as with inheritance reappear?

An Alternative InstrumentedHashSet

```
import java.util.*;
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;
    public InstrumentedSet(Set<E> s) { super(s); }
    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c){
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
```

This makes assumptions about how addAll in the underlying set object works.

Will this work, if addAll only adds elements that are not already in the set?

Also, wouldn't one have to carefully reimplement all methods that may or may not call state changing methods internally?

```
}
```

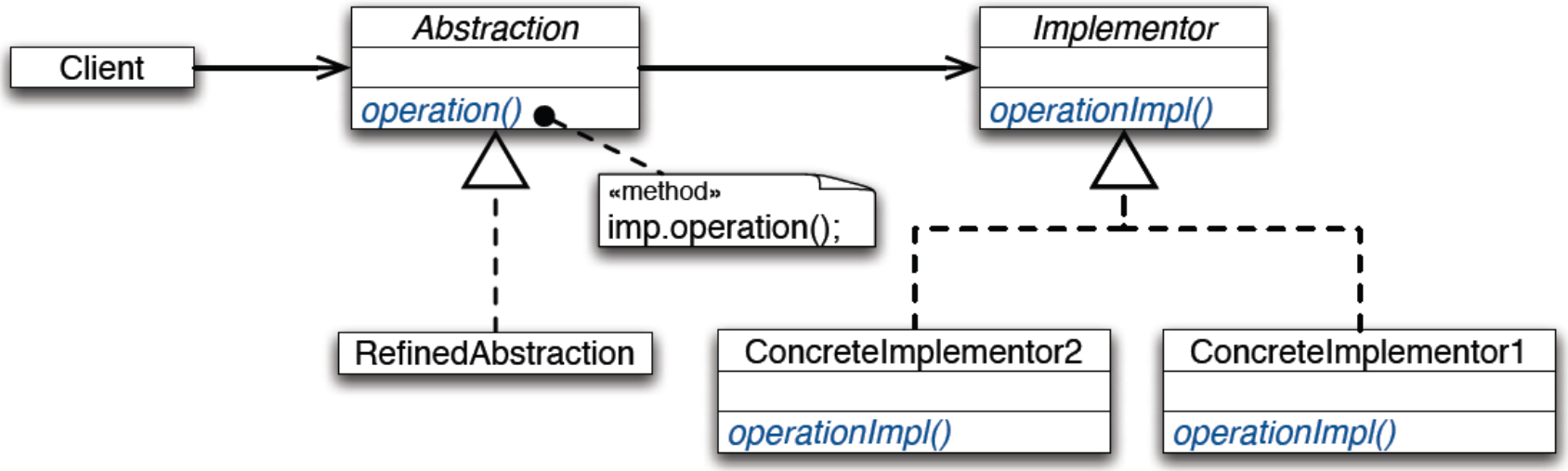
Design Patterns

2.6 Bridge

- 2.6.1 Example
- 2.6.2 Advantages
- 2.6.3 Disadvantages

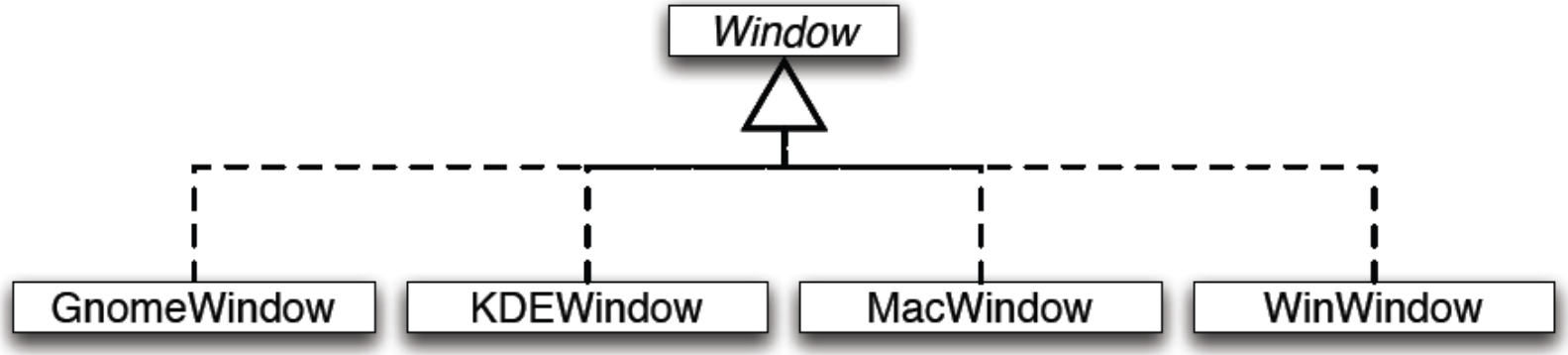
2.6 Bridge

- **Intent:** Decouple an abstraction from its implementation so that the two can vary independently.

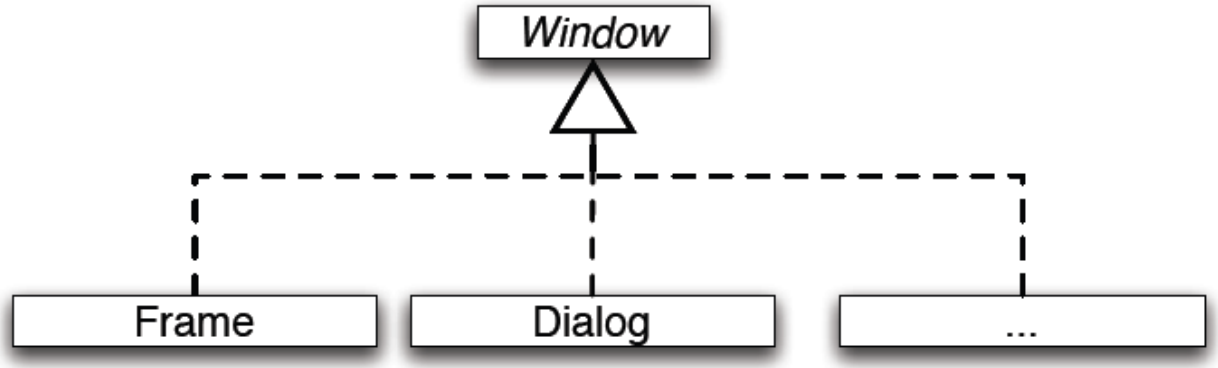


2.6.1 Example

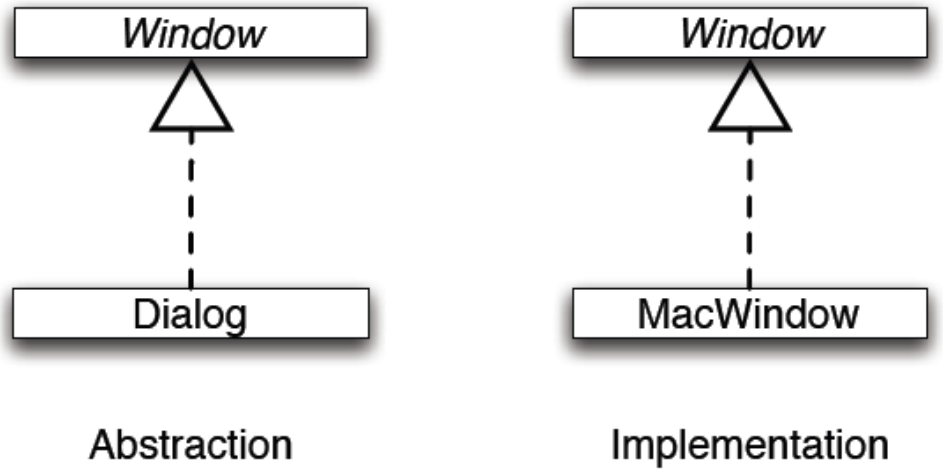
We want to support multiple operating systems...



We want to provide different types of windows...



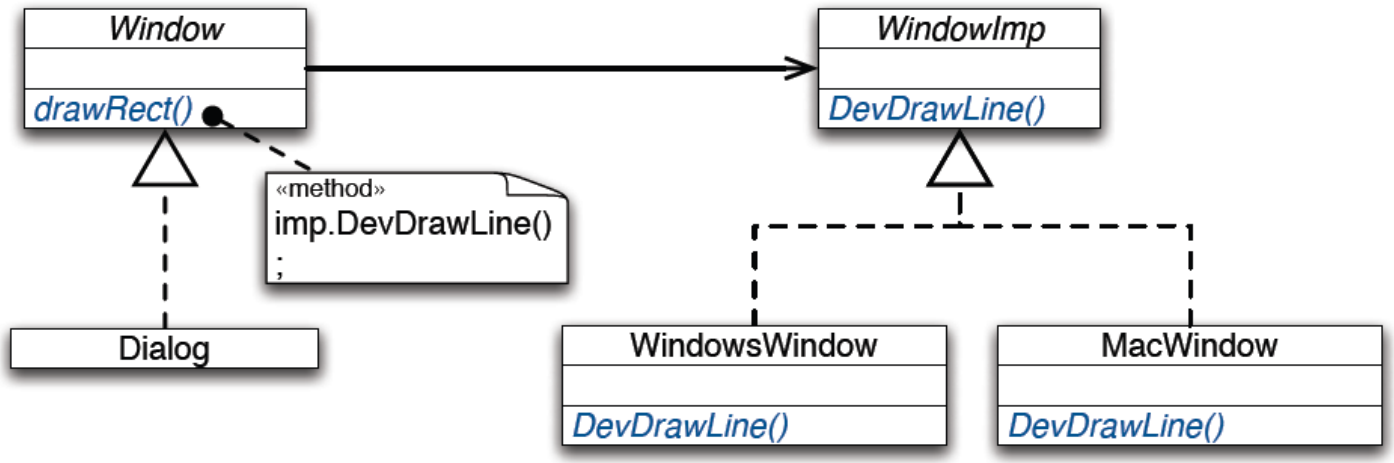
Implementation and abstraction



- Deriving `Dialog` is an abstraction.
  It further specializes `Window`.

- Deriving `MacWindow` is an implementation.
  It realizes `Window` for an operating system.

Solution using Bridge



- By encapsulating the concept that varies we can avoid problems with inheritance conflicts.
  This is very similar to the technique used in the Strategy pattern.

- **Inheritance allows adding of new features.**
- **Composition demands a fixed interface, features cannot vary.**

2.6.2 Advantages

- **Decoupling interface and implementation:**
  - Implementation can be configured at run-time.
  - Implementation being used is hidden inside abstraction.

- **Improved extensibility**
  - Abstraction and Implementor hierarchies can be extended independently.

2.6.3 Disadvantages

- Problems of parallel inheritance hierarchies
  - Type safety problems
  - Object identity problems
  - Maintaining wrapper identity

Design Patterns

2.7 Visitor

2.7 Visitor

- 2.7.1 The Intent of Visitor
- 2.7.2 Structure
- 2.7.3 Example
- 2.7.4 Intersecting shapes with Visitor
- 2.7.5 Adding new elements
- 2.7.6 Partial visiting
- 2.7.7 Conclusion

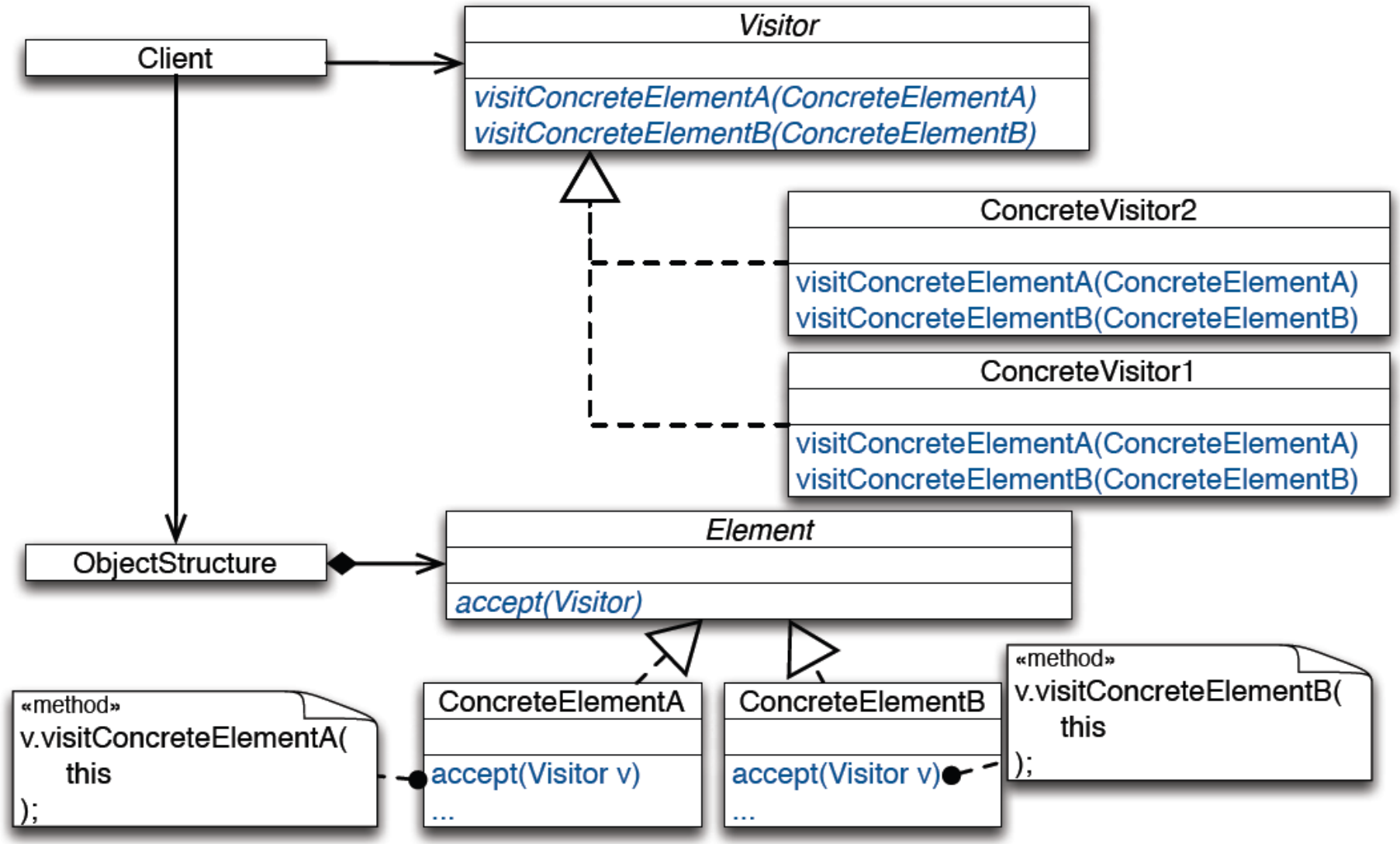
2.7.1 The Intent of Visitor

- **Intent:** To represent 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 operates.

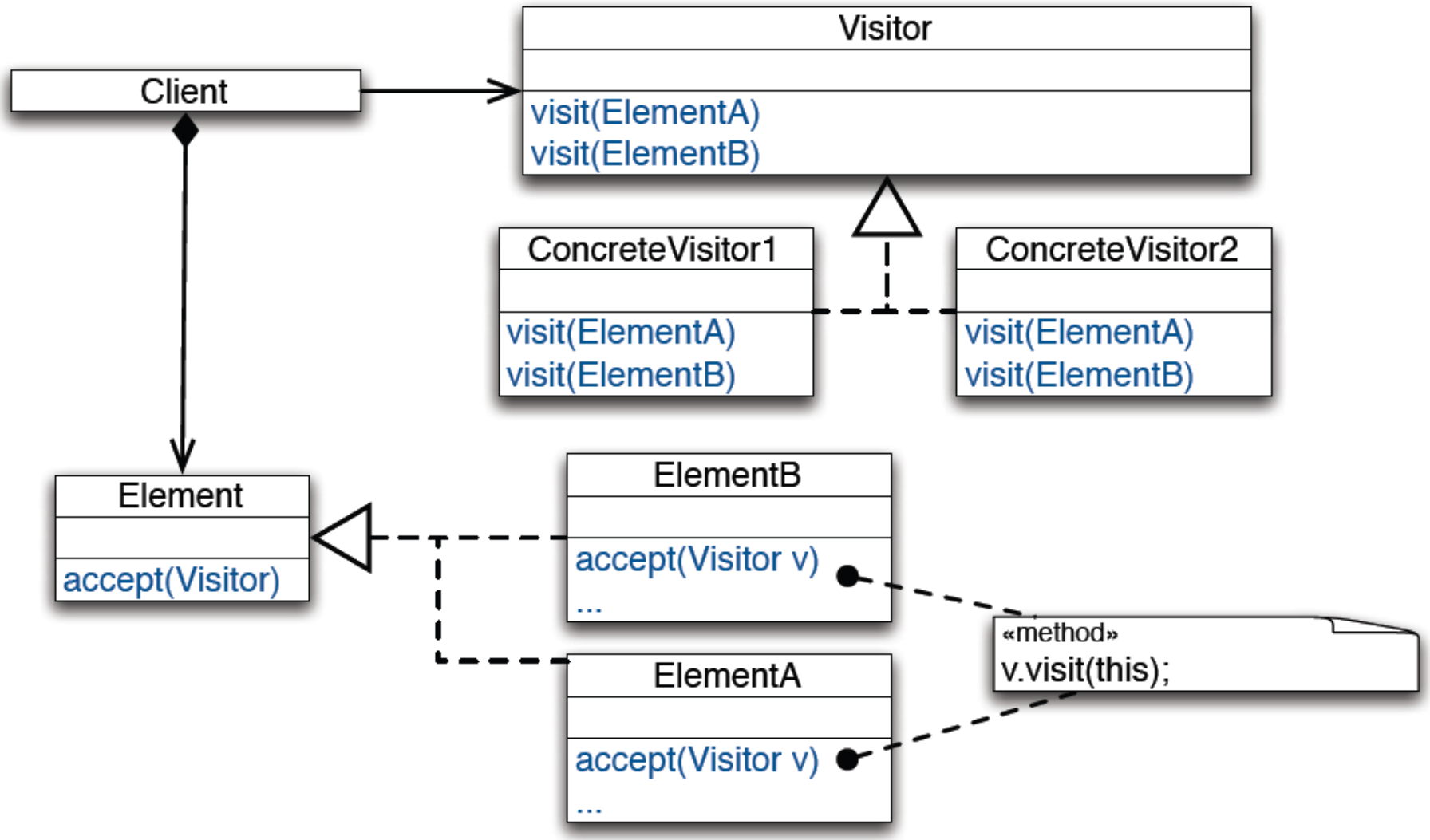
- The visitor pattern enables to add new behavior to existing classes in a fixed class hierarchy without changing this hierarchy.
- Implementing a concrete visitor is like adding a new method to the elements of the class hierarchy.

- A visitor interface describes how to “treat” the element types.

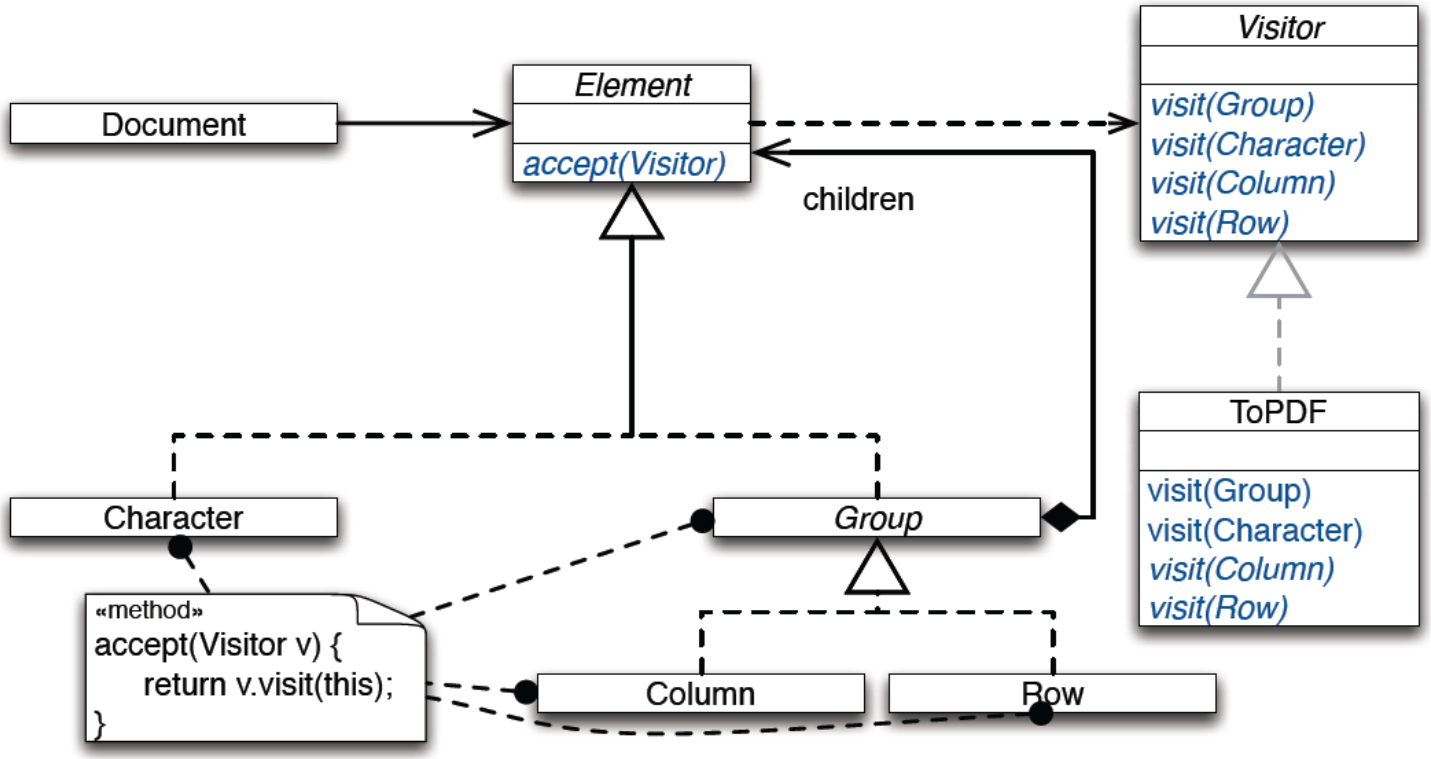
2.7.2 Structure



Structure if method overloading is supported



2.7.3 Example

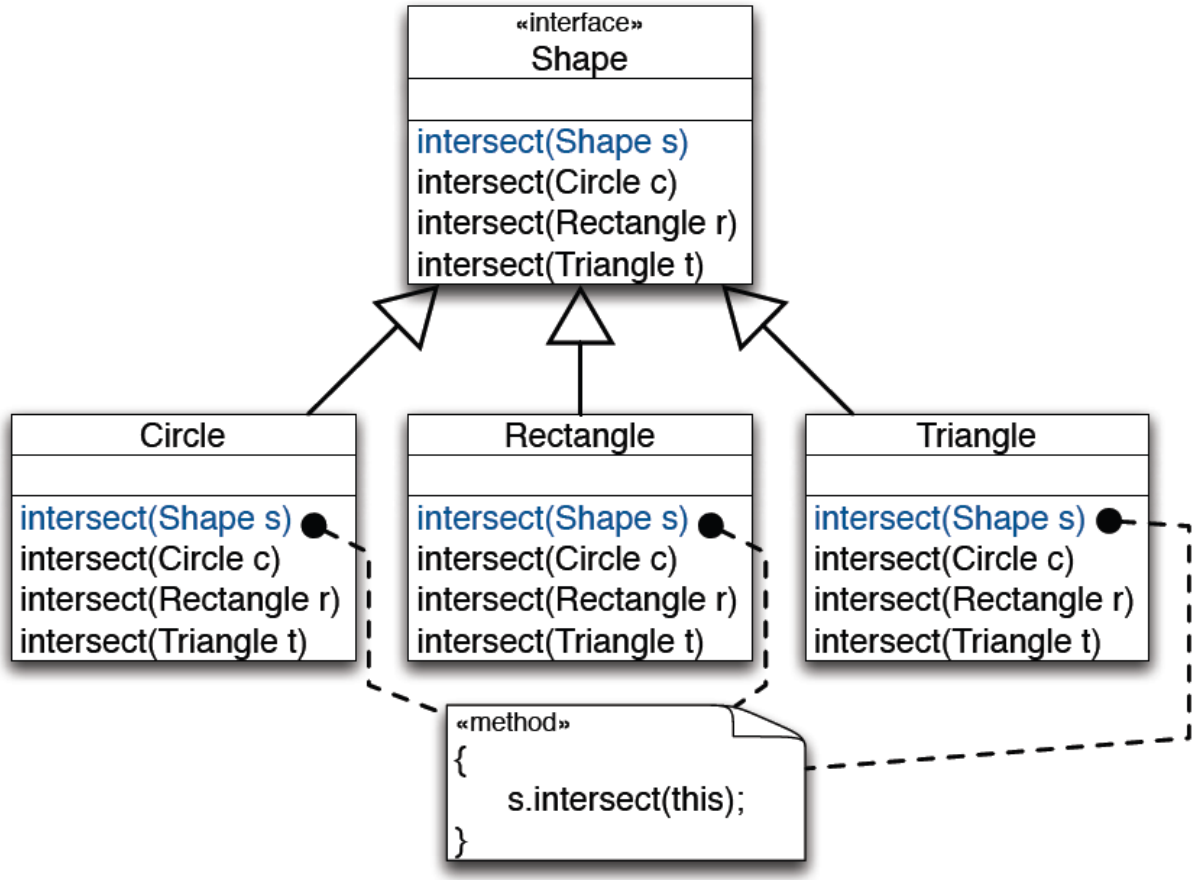


- Visitor visits all elements of a document.
- `ToPDF` converts concrete objects to PDF.
- Various other visitors may be implemented:
  Spell checking, Grammar checking, Text analysis, Speaking Text Service, ...

Advantages

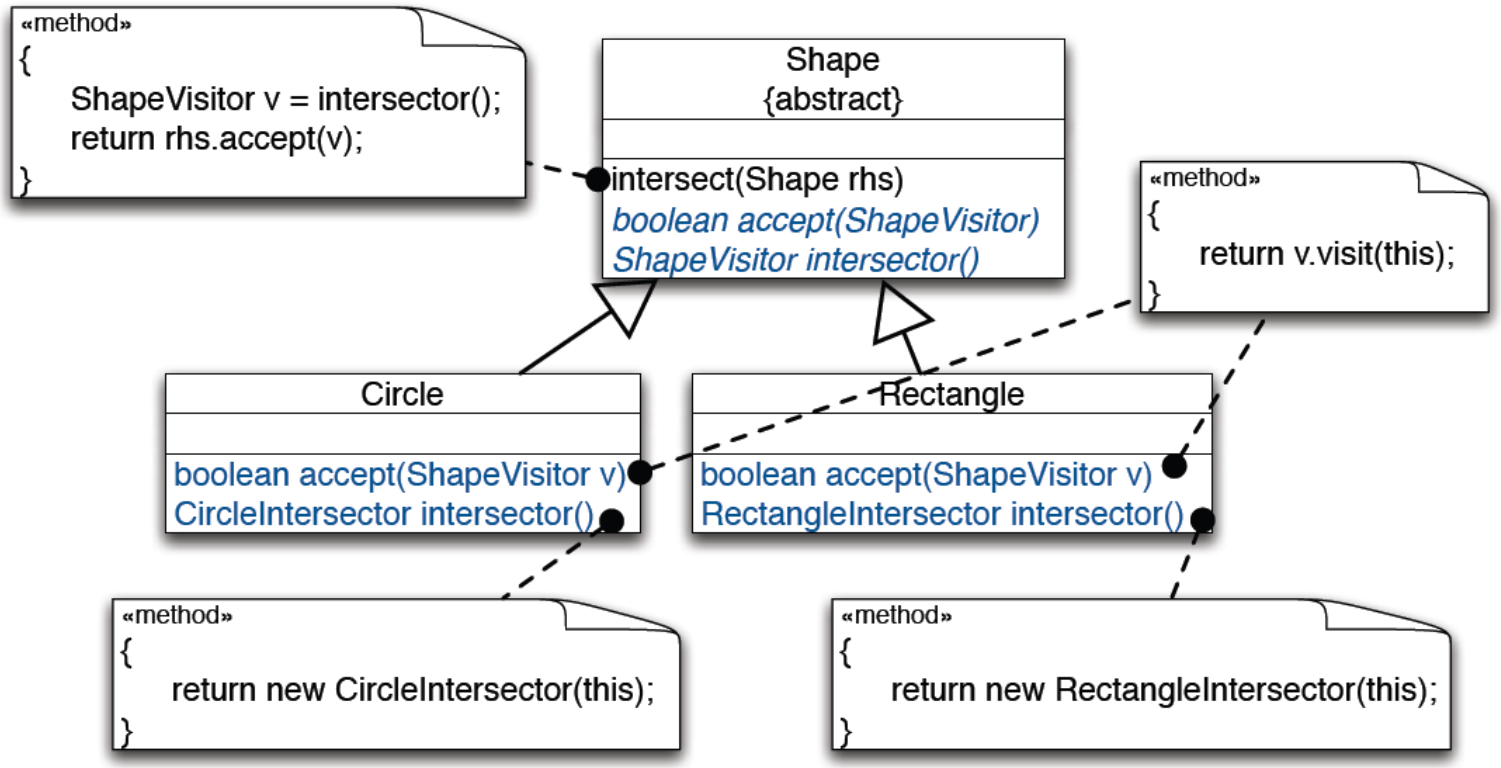
- **New operations easy to add** without changing Element classes (add a new concrete visitor).
  Different concrete elements don not have to implement their part of a particular algorithm.
- **Related behavior focused** in a single concrete visitor.
- **Visiting across hierarchies**: Visited classes are not forced to share a common base class.
- **Accumulating state**: Visitors can accumulate state as they visit each element, thus encapsulating the algorithm and all its data.

2.7.4 Intersecting shapes with Visitor



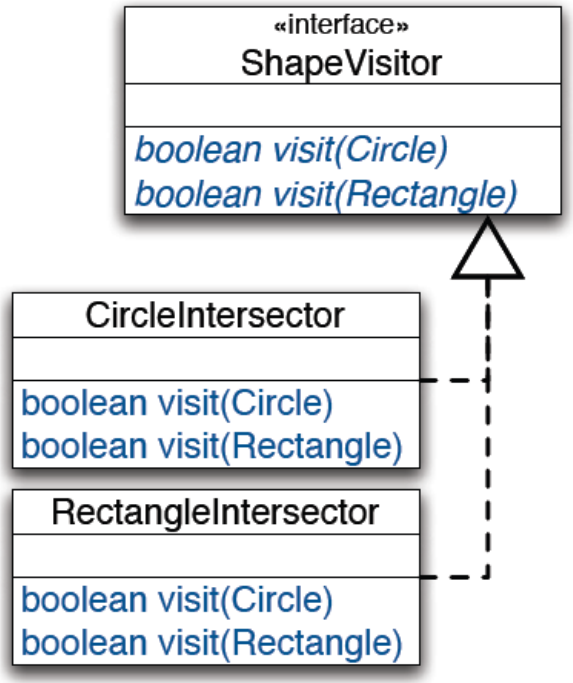
- Given design forces every shape class to implement its intersection with every other shape.
  Adding new shapes means implementing new methods in every other shape.

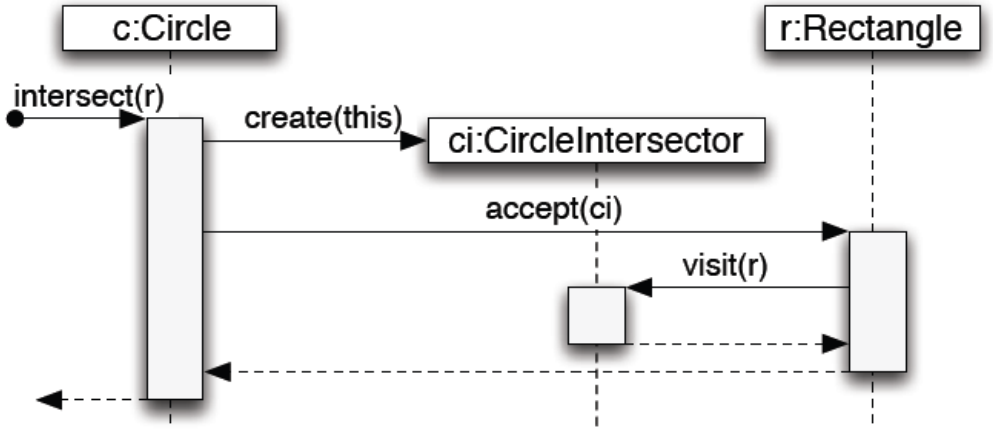
Solution using Visitor (1)



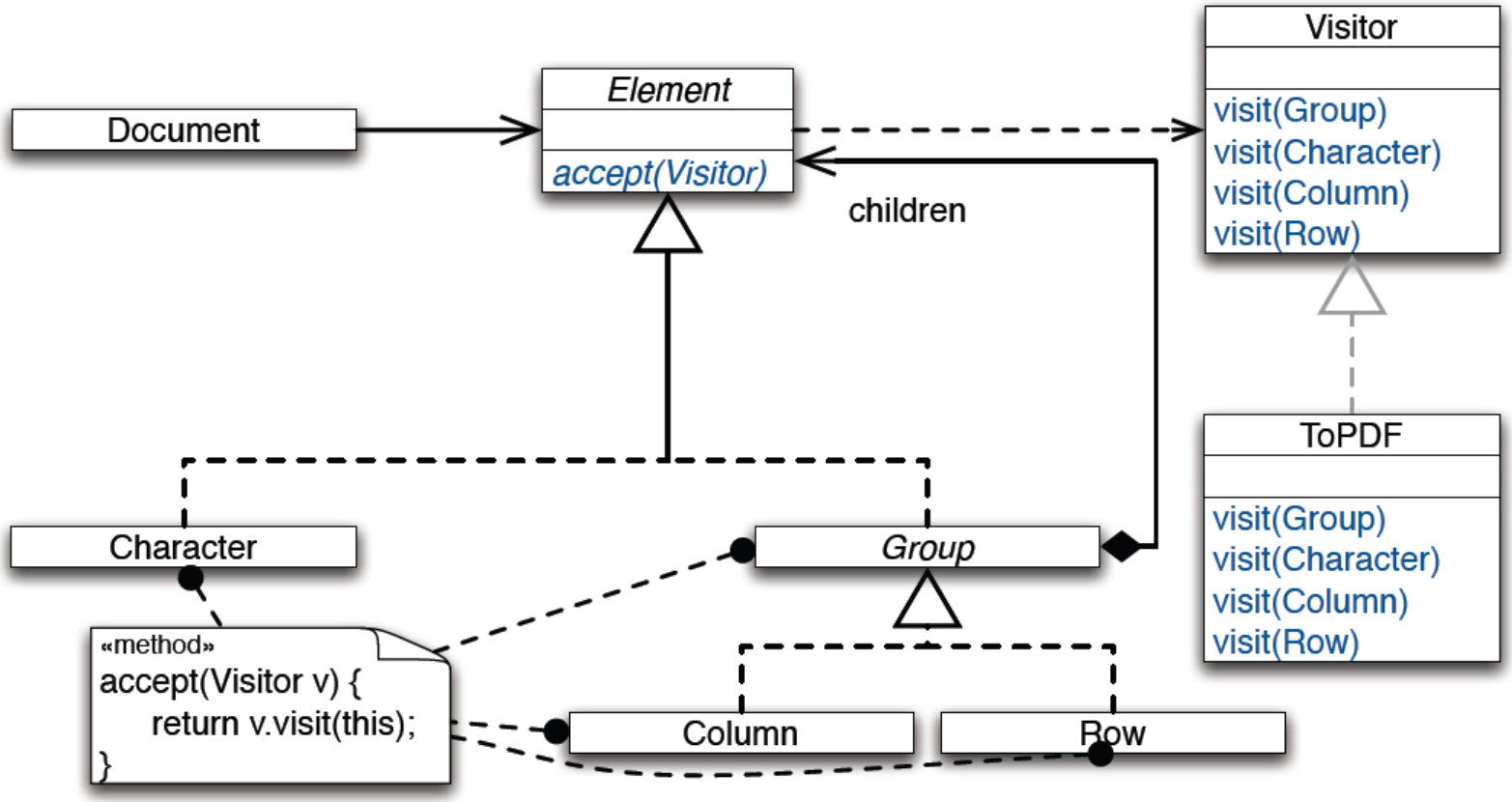
Solution using Visitor (1)

```
Shape c = new Circle(…);
Shape r = new Rectangle(…);
if (c.intersect(r)) {…}
```



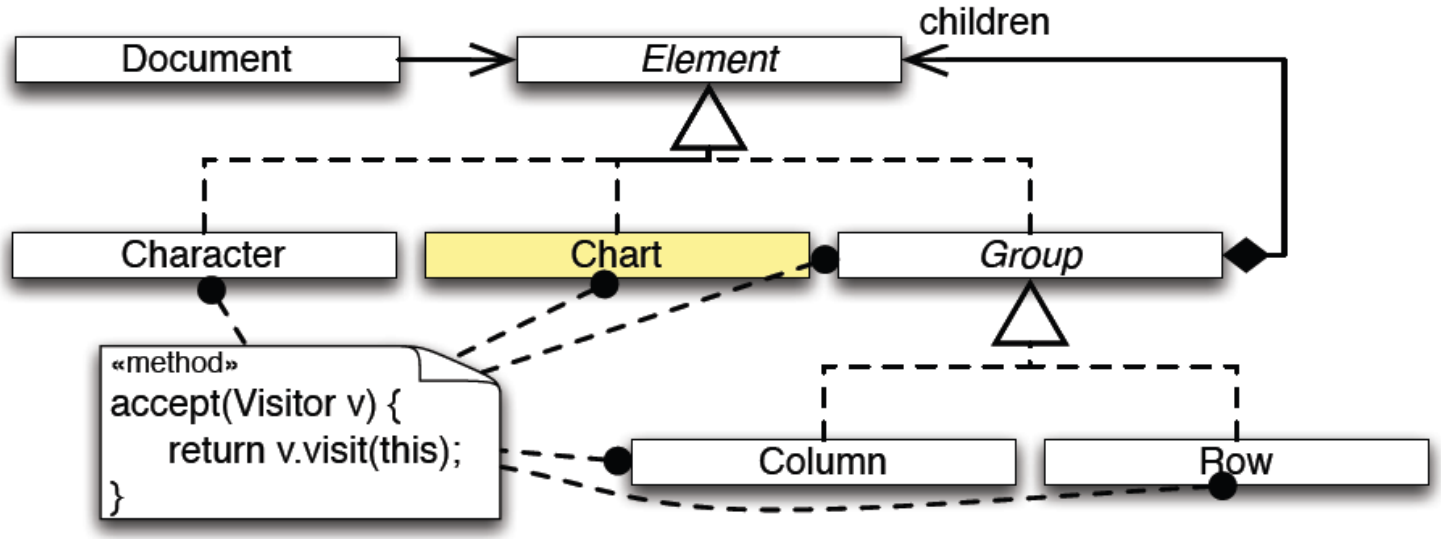


2.7.5 Adding new elements



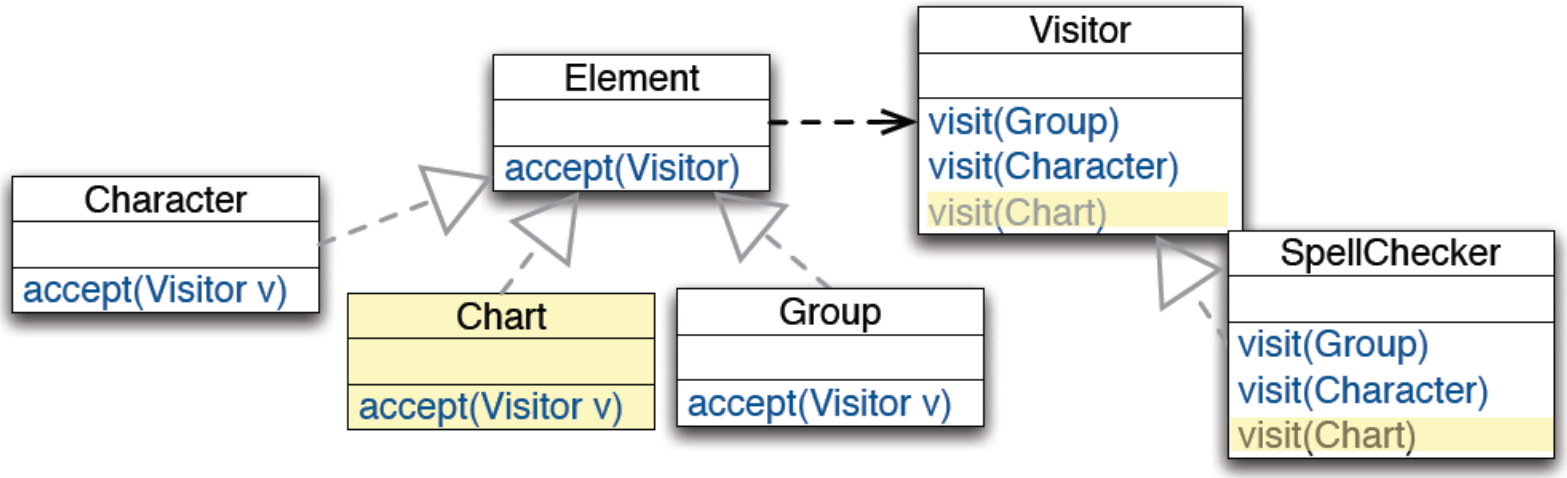
- As we have seen, it is easy to add new operations to this design.
- But what happens if we want to add a new element?

Adding `Chart`



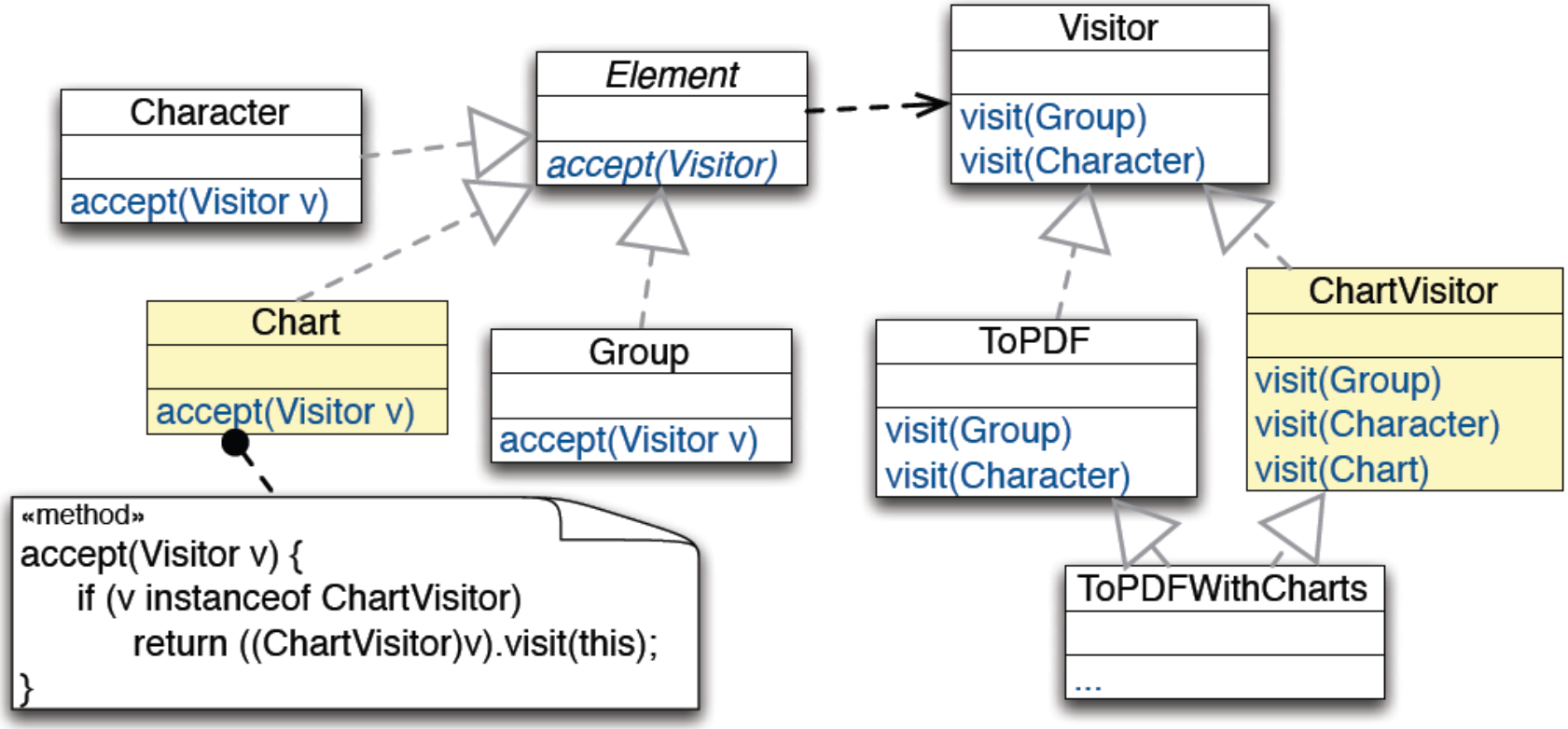
- Problem: Since `Visitor` has no method for `Chart`, it wont be processed by any visitor!
- Our design is not closed against this kind of change.

Solution: Changing visitors



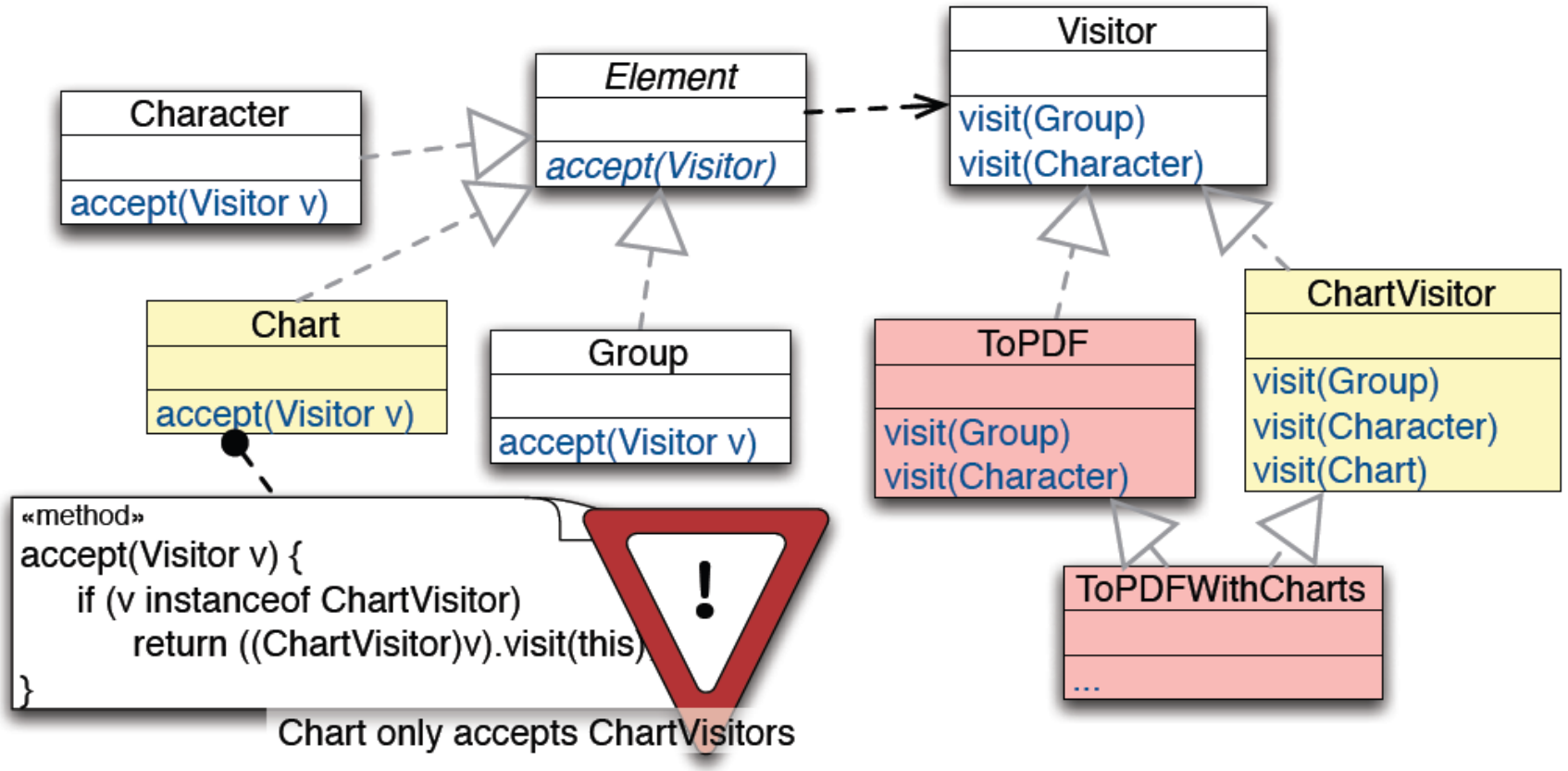
- Problems:
  - We have to change all visitors for every new element.
  - Many visitors will have empty methods to comply to the interface. E.g. `SpellChecker.visit(Chart)`

Solution: Keeping visitors unchanged (1)



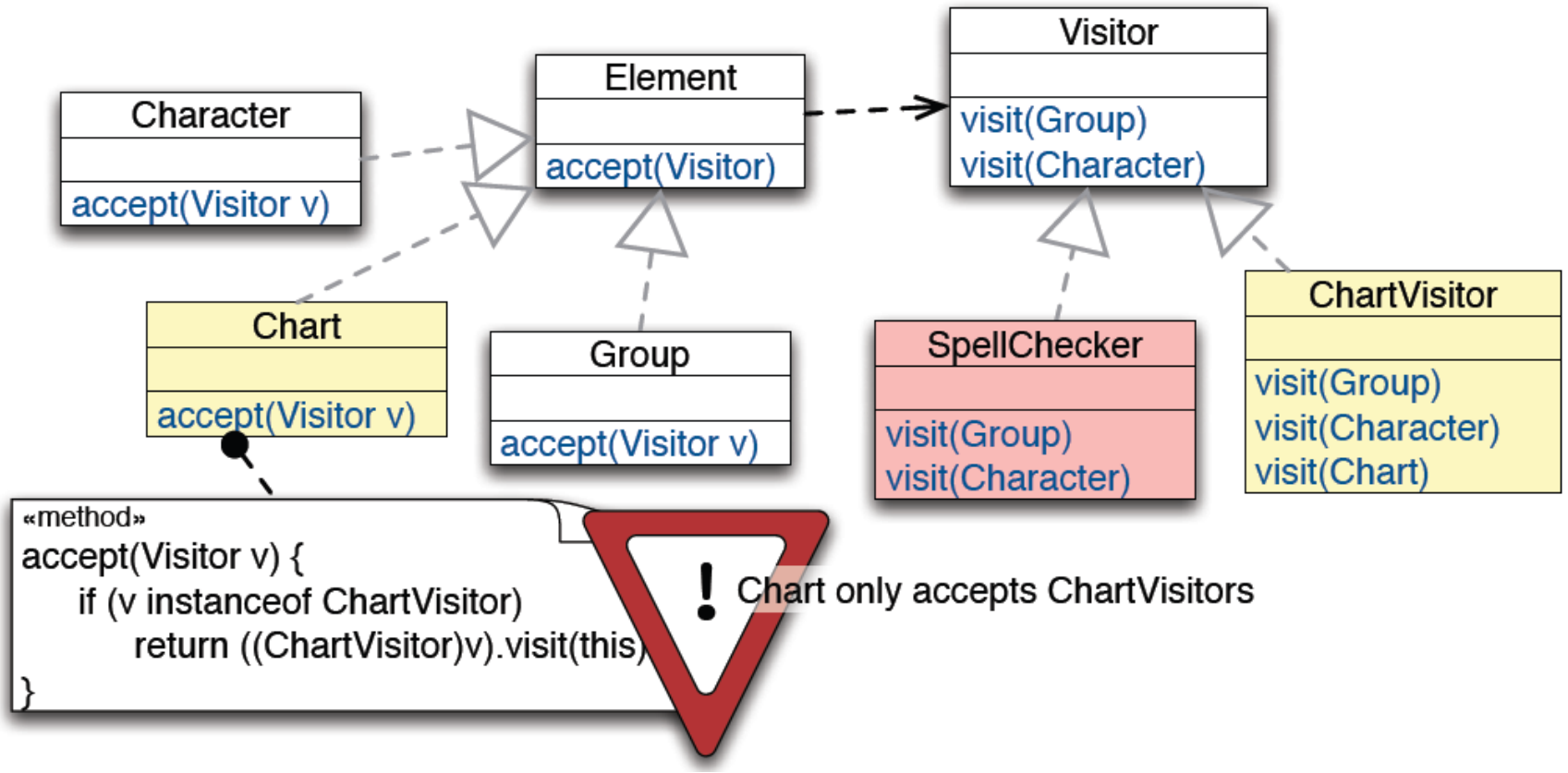
- Sometimes data structures are extended, but it‘s optional to process extensions.
  E.g., it doesn’t make sense to spell check charts.

Solution: Keeping visitors unchanged (2)



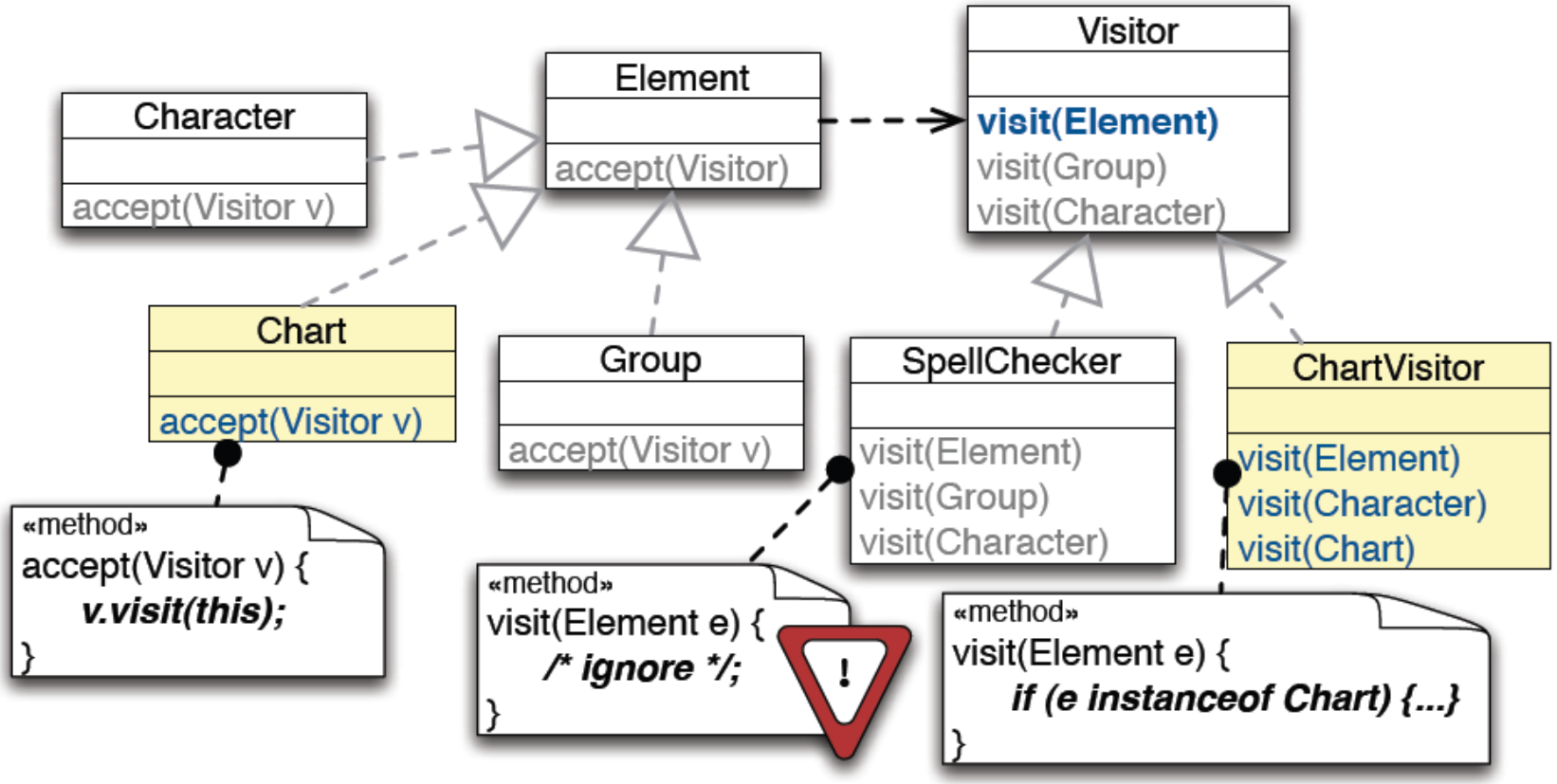
- **Try to avoid such visitors as these implementations are extremely fragile.**
  No static type safety

Solution: Keeping visitors unchanged (3)



- **Try to avoid such visitors as these implementations are extremely fragile.**
  No static type safety

Solution: Designing the visitors open



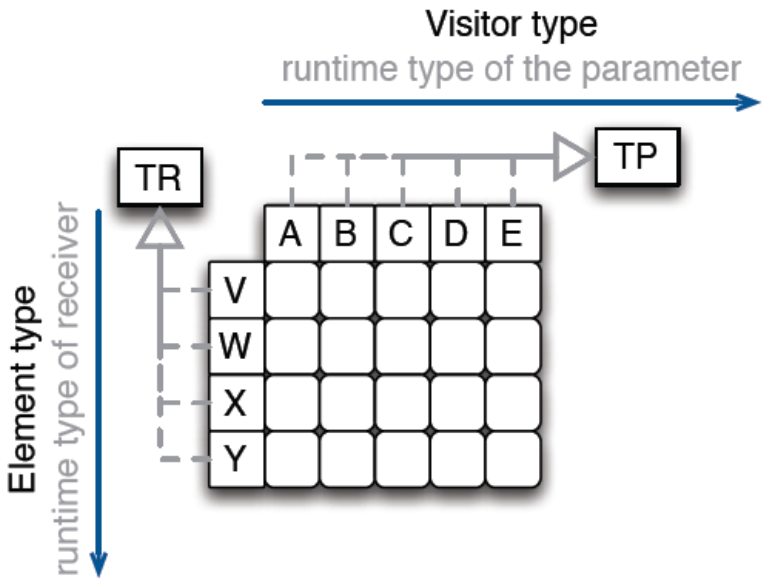
- The operation `visit(Element)` provides a trap door through which to visit unforeseen element subclasses.

- Attention: Put the type check / type cast in the visit method of `ChartVisitor` and not in the accept method of `Chart`.

2.7.6 Partial visiting

- Partial visiting is not supported!
  To provide a common abstract Visitor interface to Element, **every derivative of Element need to be addressed by every derivative of** Visitor; even if this might not make sense or is not needed.
  We have seen this for `SpellChecker.visit(Chart)`

- Visitor is like a matrix (cross product of all Visitor and Element classes):



2.7.7 Conclusion

- **Use Visitor for stable element hierarchies!**
  Visitor works well in data hierarchies where new elements are never or at least not very often added.
- Do not use it if new elements are a likely change.

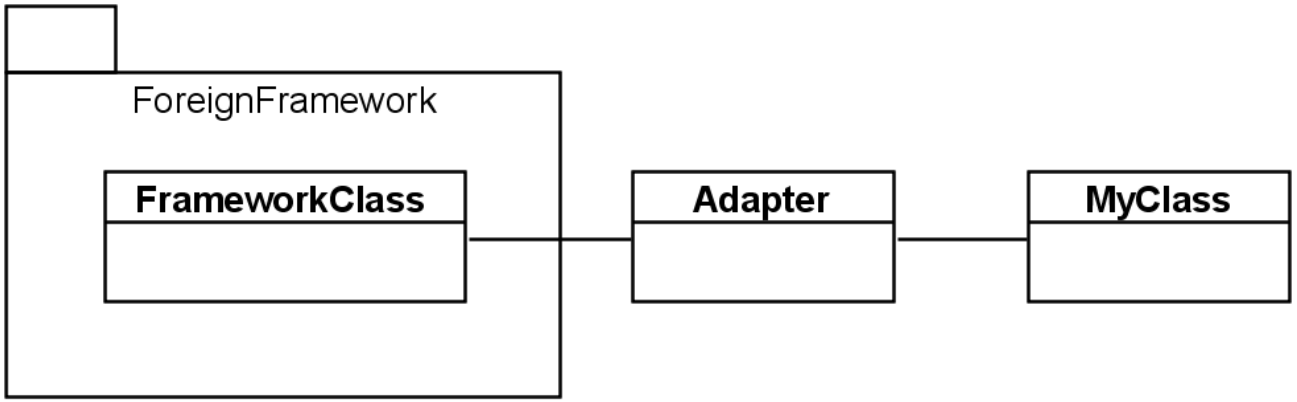
- **Visitor only makes sense if we have to add new operations often!**
  In this case Visitor closes our design against these changes.

2.8 Adapter

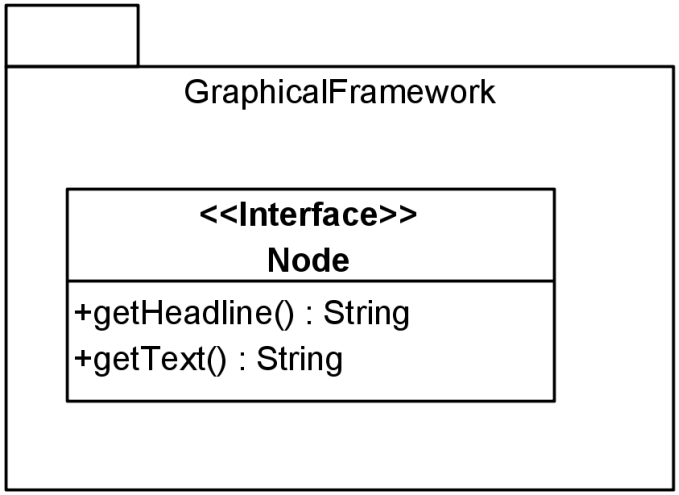
- 2.8.1 Example
- 2.8.2 Object Adapter
- 2.8.3 Class Adapter

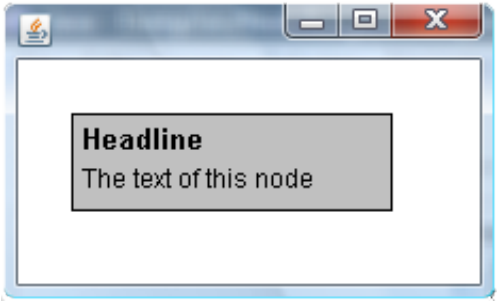
2.8 Adapter

- **Intent:** Fit foreign components into an existing design.
  - We want to reuse existing frameworks or libraries in our software, even if they do not match with our design.
  - We do not want to change our design to adhere to the structure of the reused components.



2.8.1 Example

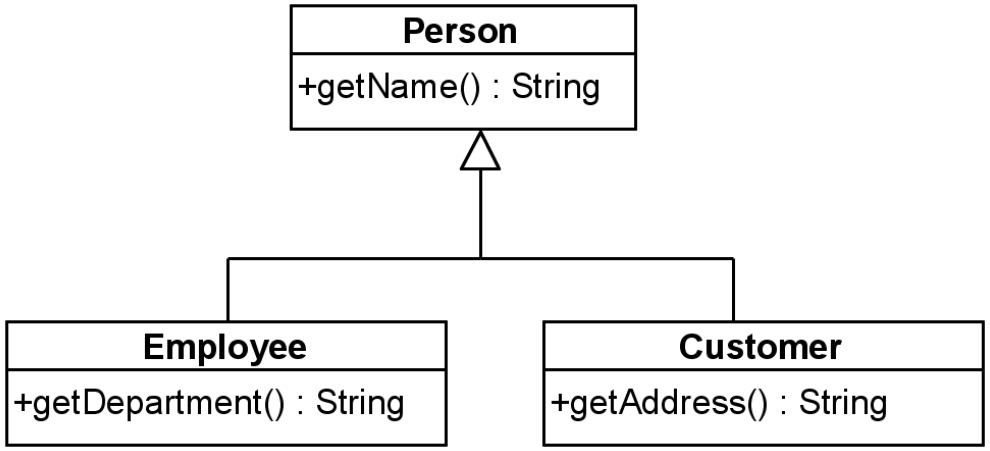




- We have acquired the framework `GraphicalFramework`.
- `GraphicalFramework` provides the interface `Node` to draw rectangles with a headline and text to the screen.

- Drawing is done by the framework, we just need to provide the data via the interface `Node`.

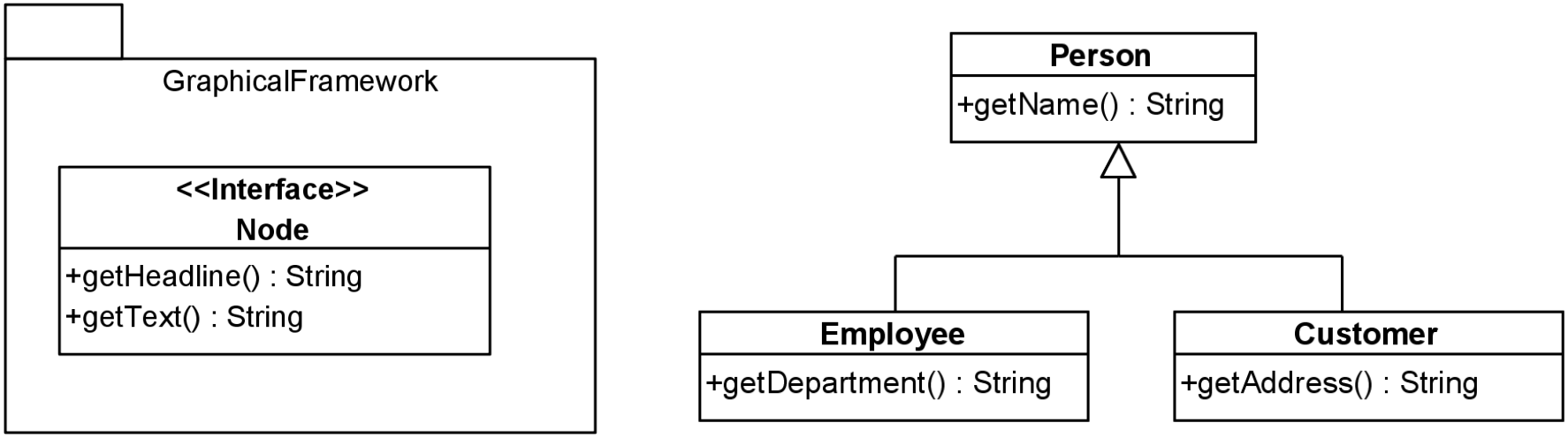
Desired usage of framework



- Our own design represents different kinds of persons.

- We want to draw our data to the screen:
  - Name and department of `Employee`.
  - Name and address of `Customer`.

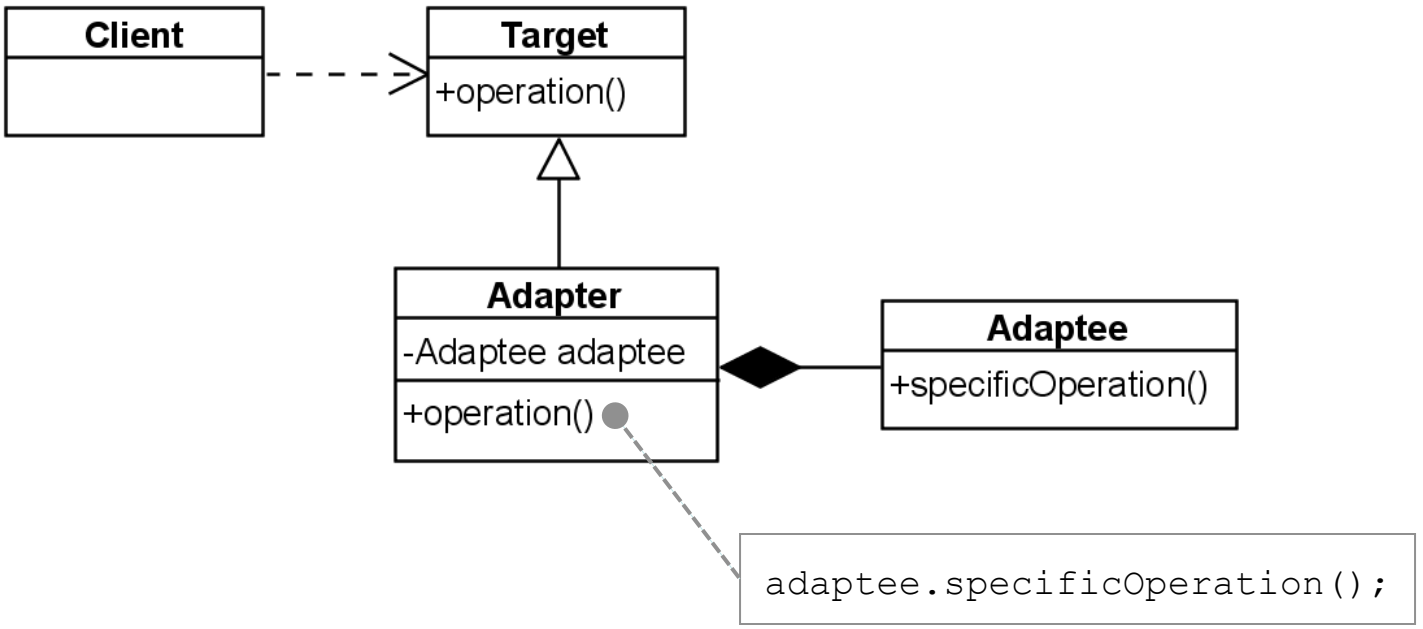
Adapting the framework



- We will create adapters to use the functionality of `GraphicalFramework` for our classes.
  We have to wrap `Employee` and `Customer` to fit with `Node`.

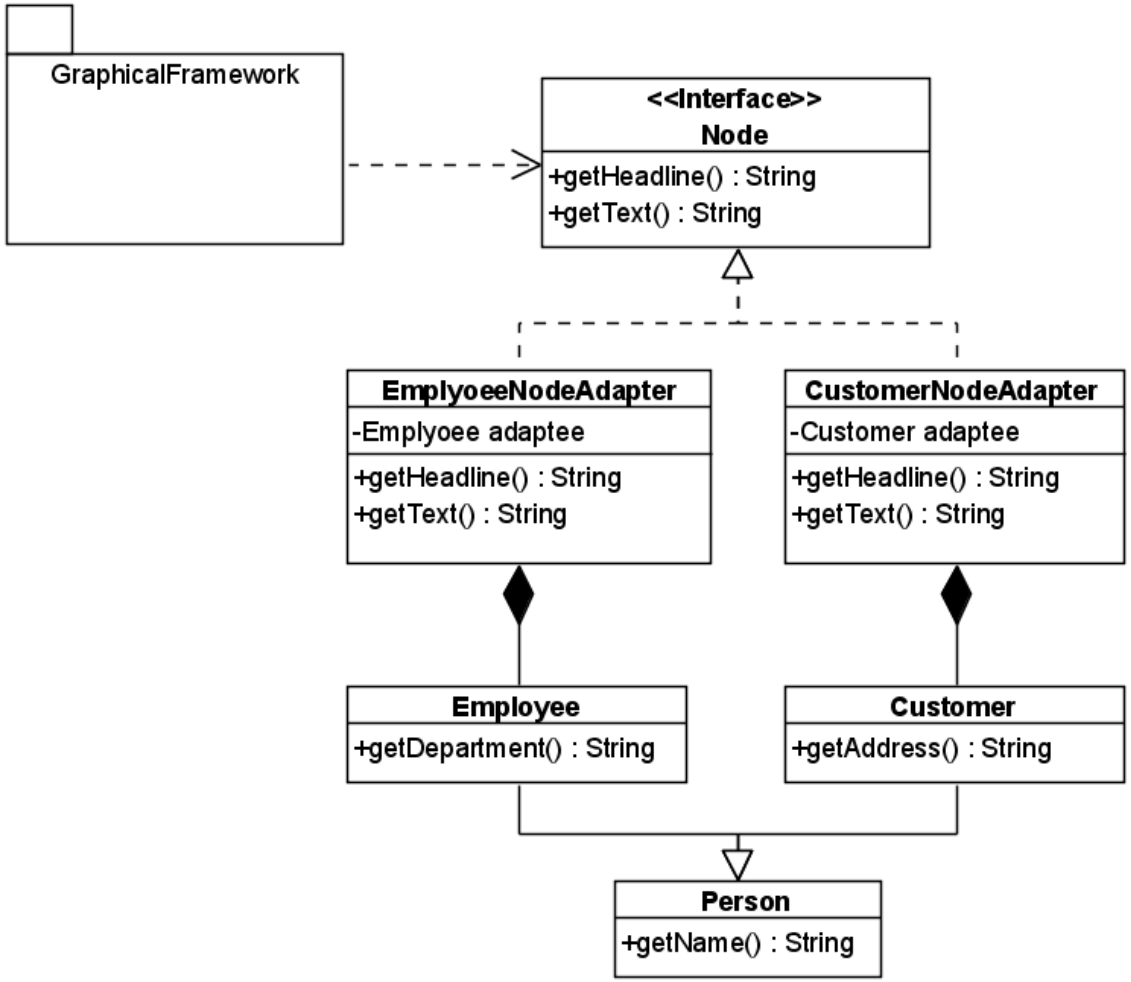
- There are two different kind of adapters:
  - Object Adapter
  - Class Adapter

2.8.2 Object Adapter

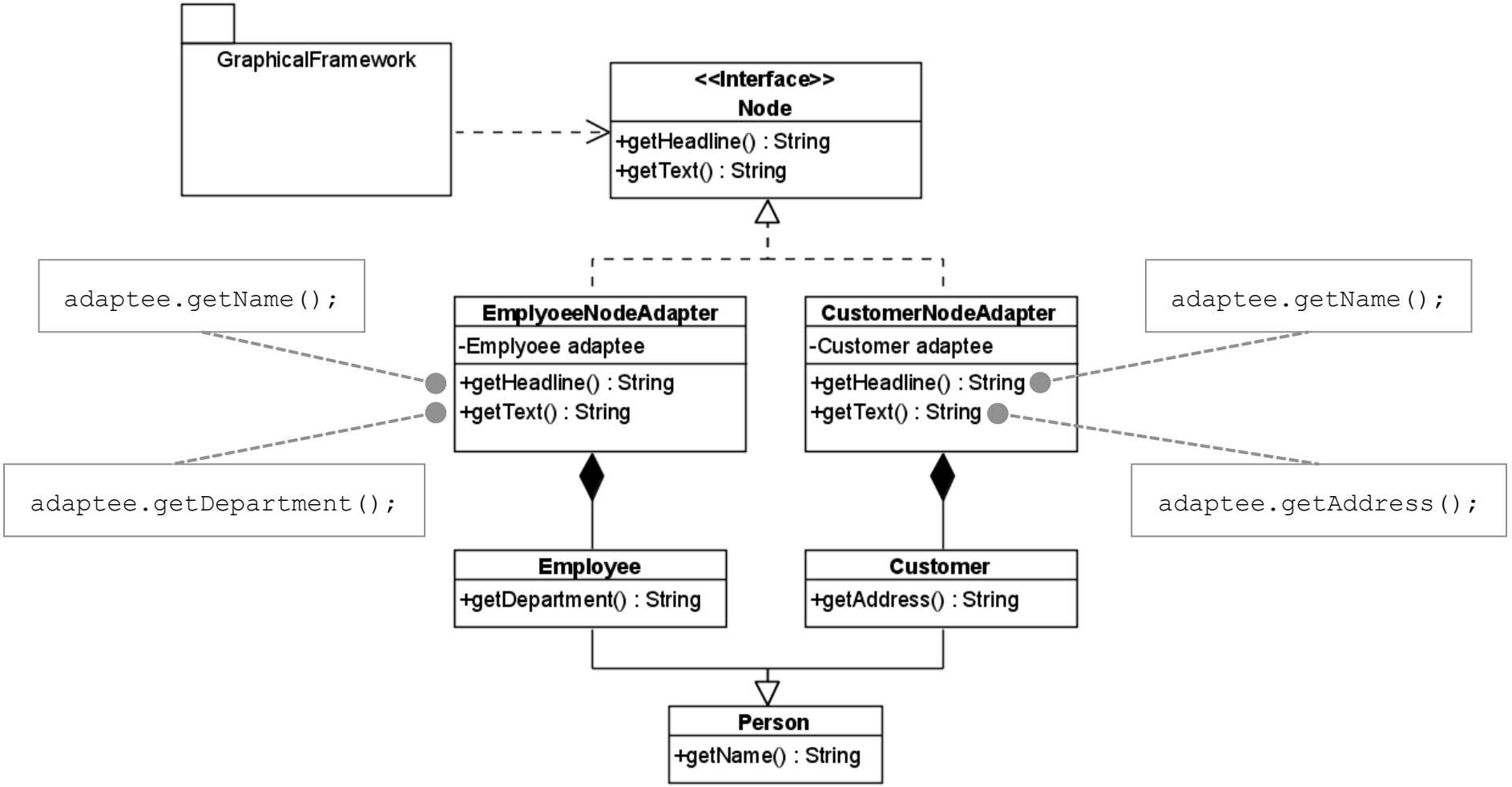


- `Adaptee` is wrapped by `Adapter` to fit in the interface of `Target`.
- `Adapter` forwards calls of `Client` to `operation()` to the specific methods of `Adaptee`.

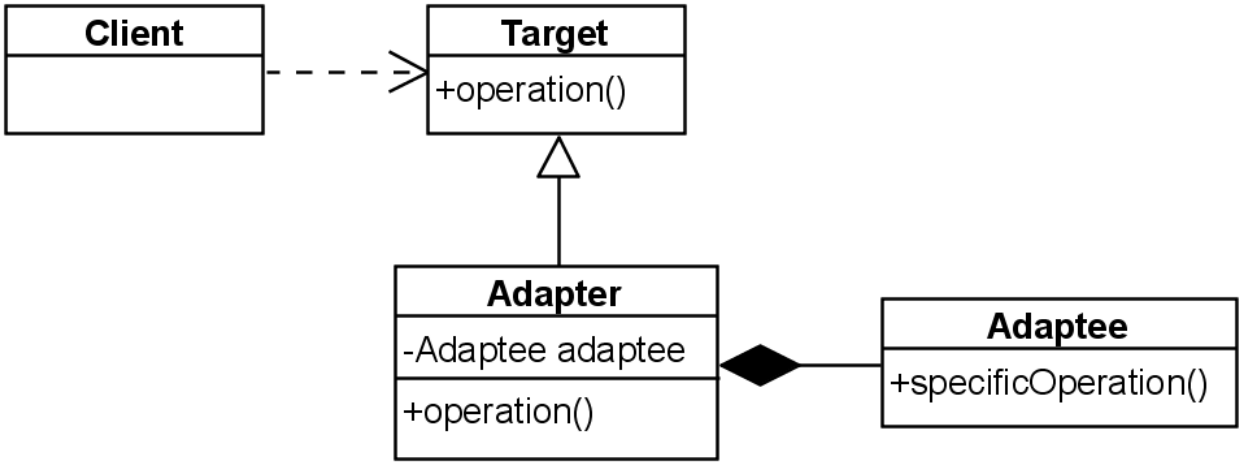
Using Object Adapter



Using Object Adapter



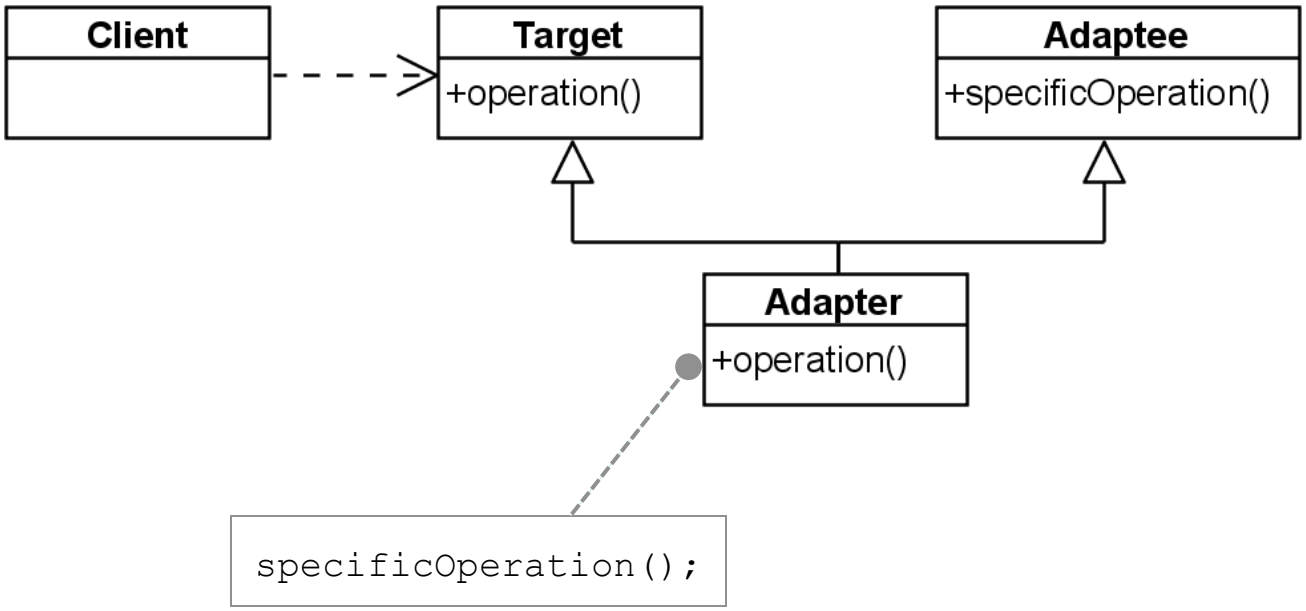
Discussion



- Advantages:
  - `Adapter` works with `Adaptee` and any subclass of it.
  - `Adapter` can add functionality to `Adaptee` and its subclasses.

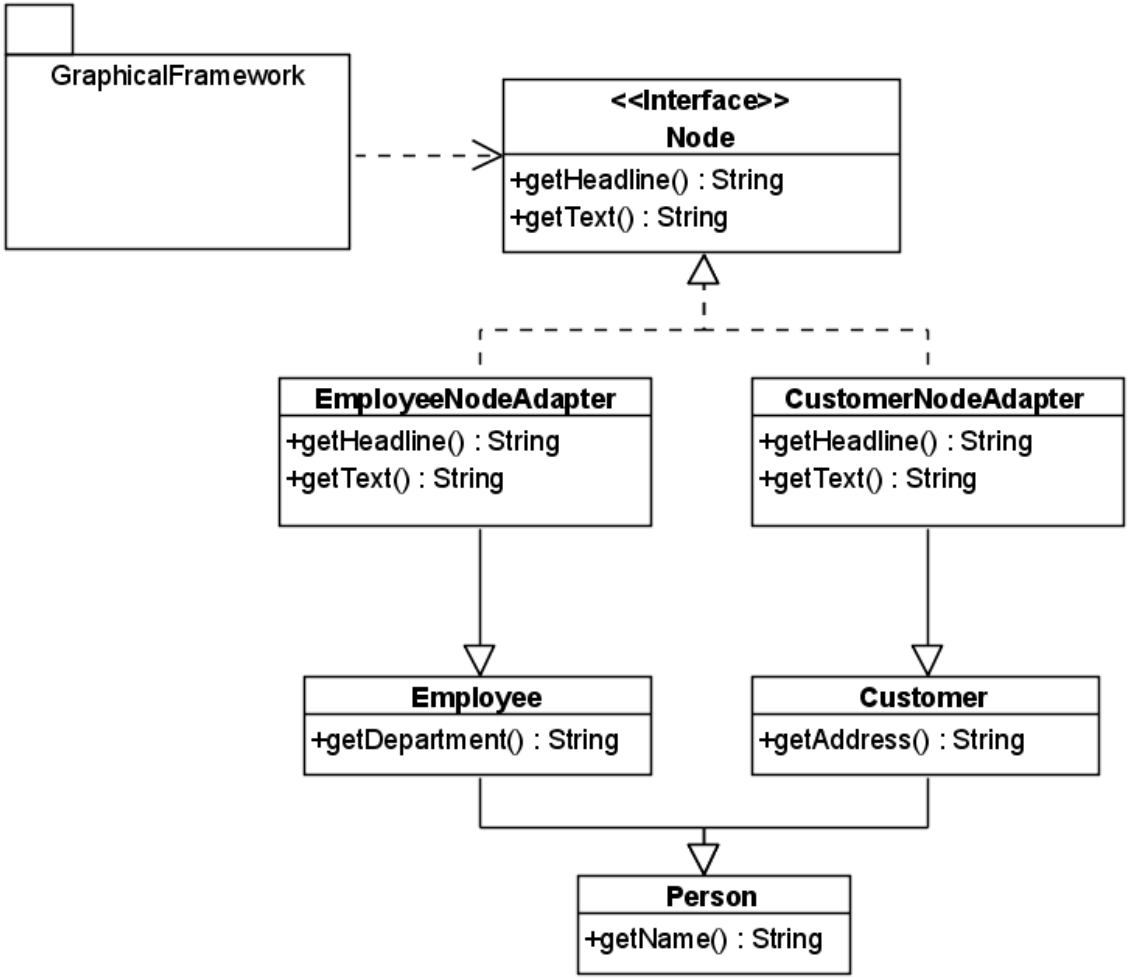
- Disadvantages:
  - Cannot override `Adaptee`.
  - Cannot reuse `Adapter` with subclasses of `Target`.
  - `Adapter` and `Adaptee` are different objects.
    (Need to maintain relation between adaptee and his adapter)

2.8.3 Class Adapter

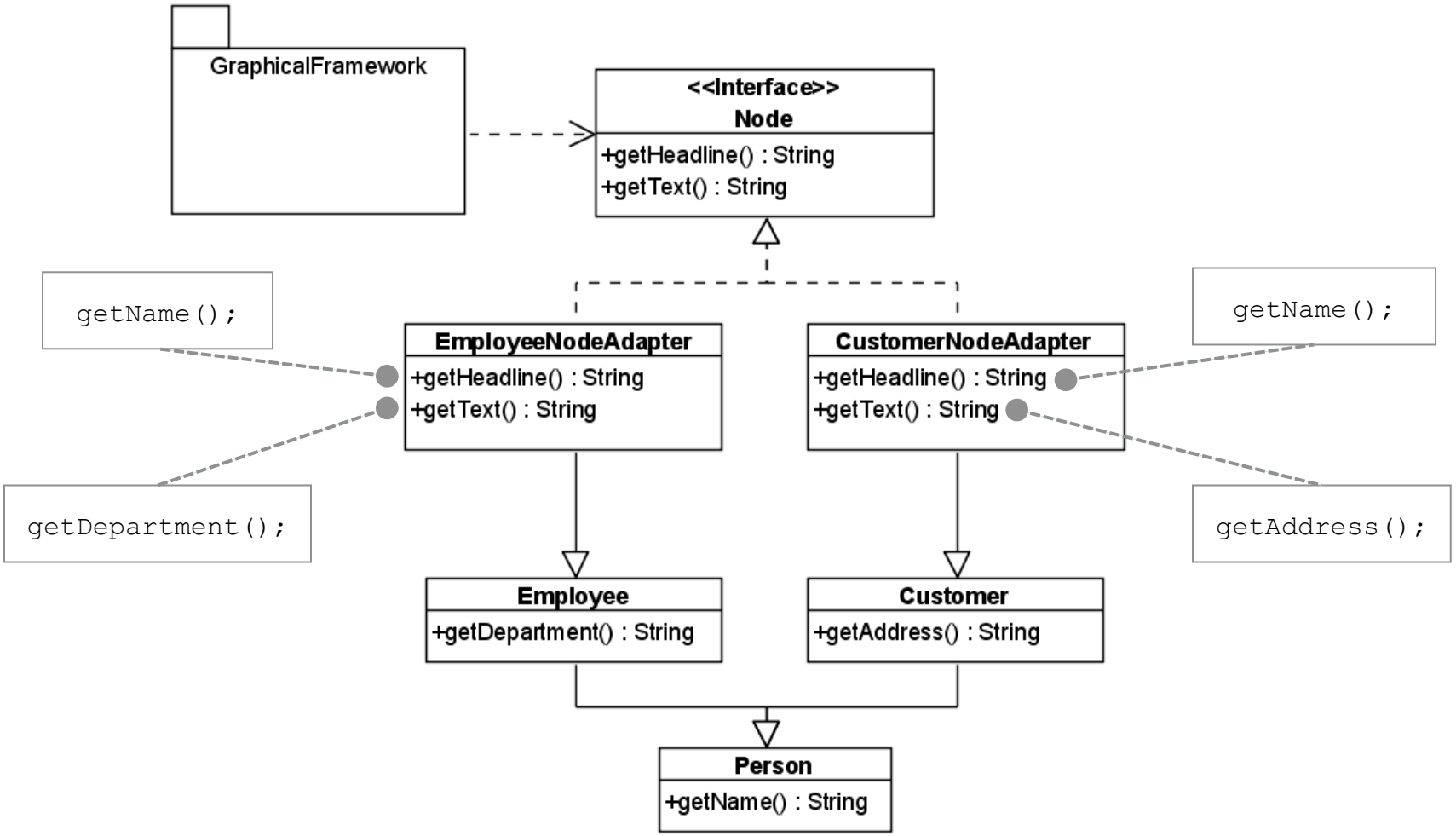


- Instead of having `Adaptee` as an attribute, `Adapter` inherits from `Adaptee` and forwards calls of `Client` to `operation()` to itself.

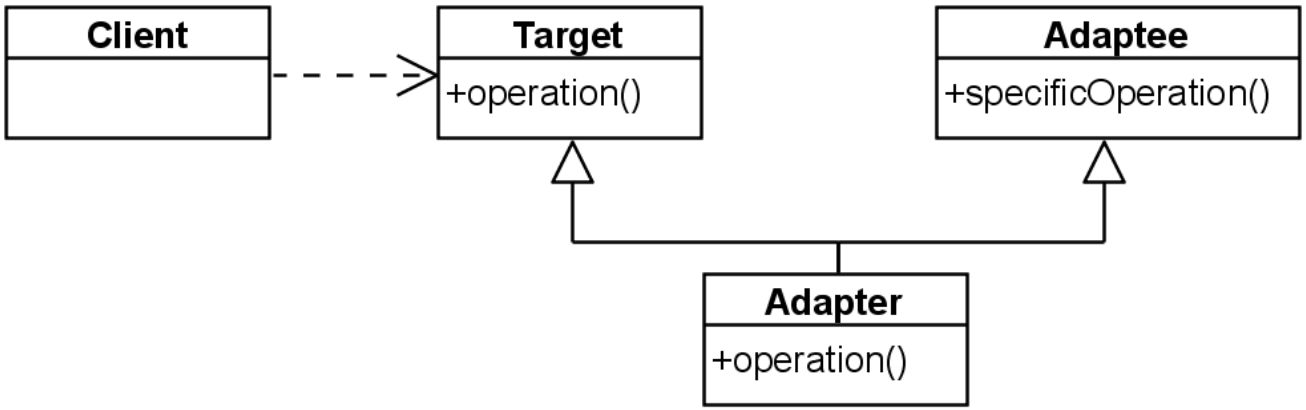
Using Class Adapter



Using Class Adapter



Discussion



- Advantages:
  - Behavior of `Adaptee` can be overridden.
  - `Adapter` and `Adaptee` are the same object, no forwarding.

- Disadvantages:
  - `Adapter` cannot be used with subclasses of `Adaptee` nor `Target`.
  - Cannot adapt existing objects/must change constructor calls
  - No independent extensibility with multiple adapters
  - Multiple inheritance must be possible.
    E.g. Java: At least one of `Target` and `Adaptee` must be an `Interface`.

2.9 Builder

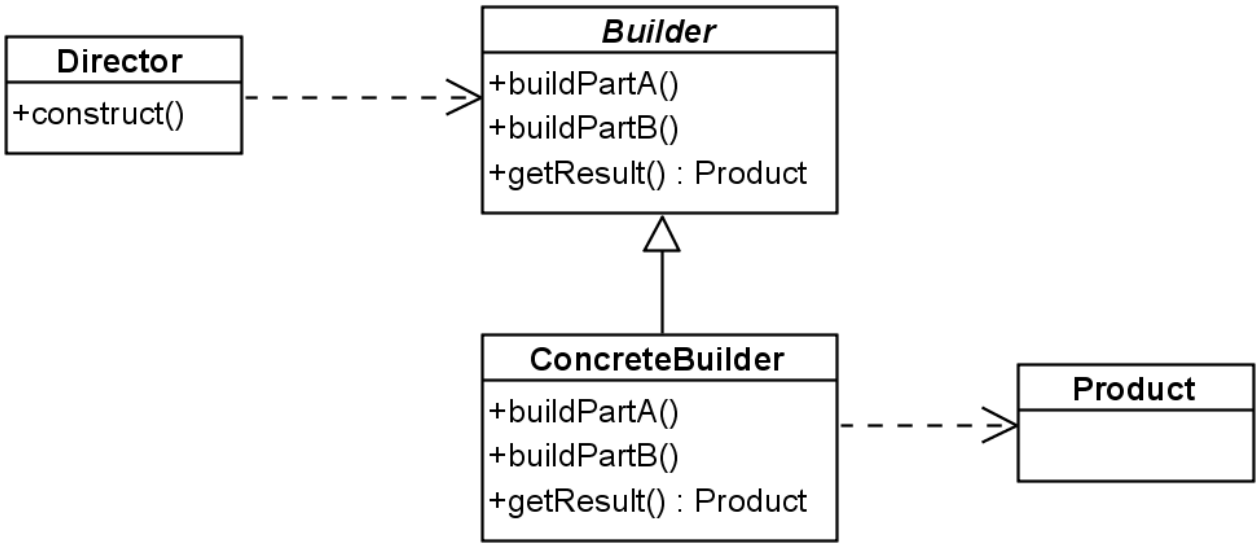
- 2.9.1 Structure
- 2.9.2 Example
- 2.9.3 Discussion

2.9 Builder

- **Intent:** Divide the construction of objects so that different implementations of these steps can construct different representations of objects.

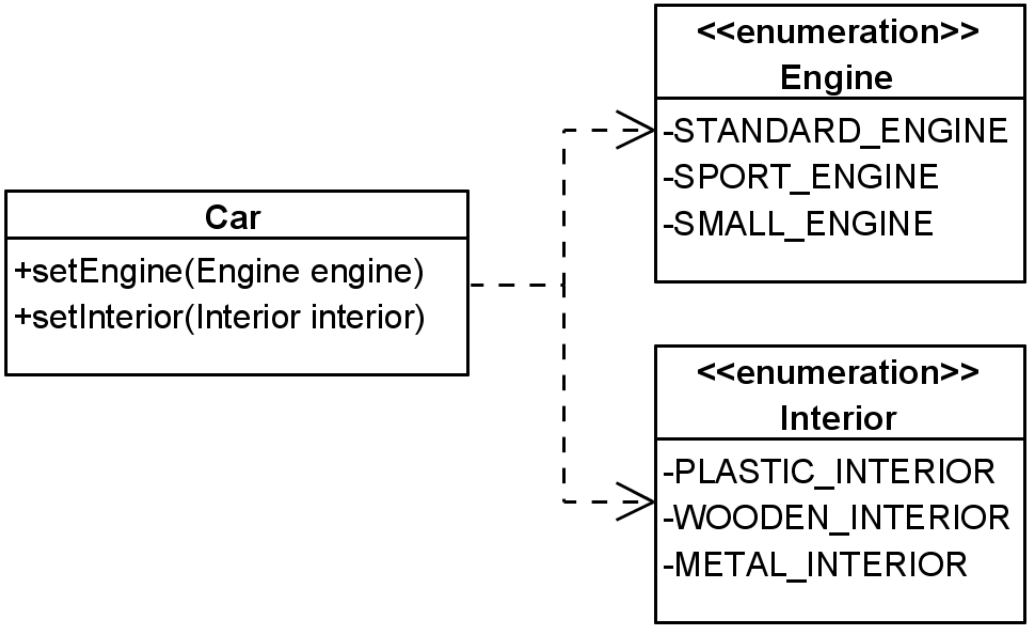
- Often there are objects that are constructed from multiple parts. Builder divides the construction of such objects into different steps.
- Builder also allows to abstract these steps so objects can be constructed from different parts.

2.9.1 Structure



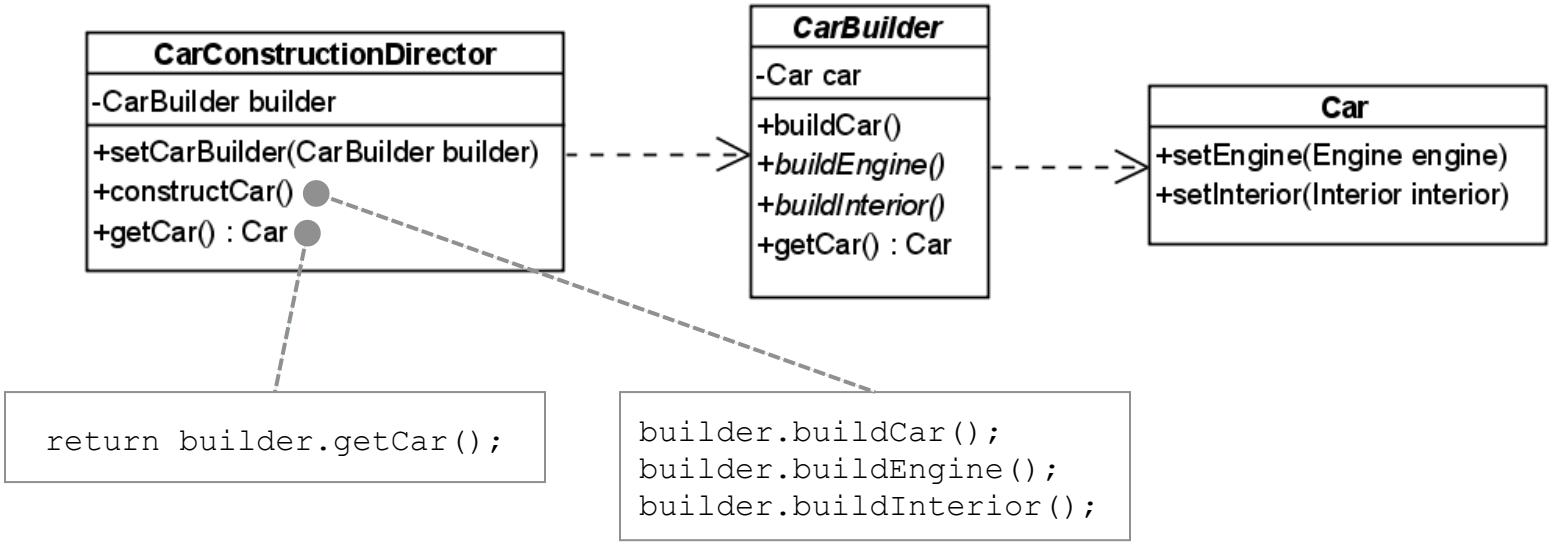
- `Builder` defines the individual steps of the construction of `Product`.
- `Director` knows in which order to construct `Product`.
- `ConcreteBuilder` implements the steps of construction.

2.9.2 Example



- We want to construct different types of cars.
  - In this example, cars have an engine and an interior.

A car builder

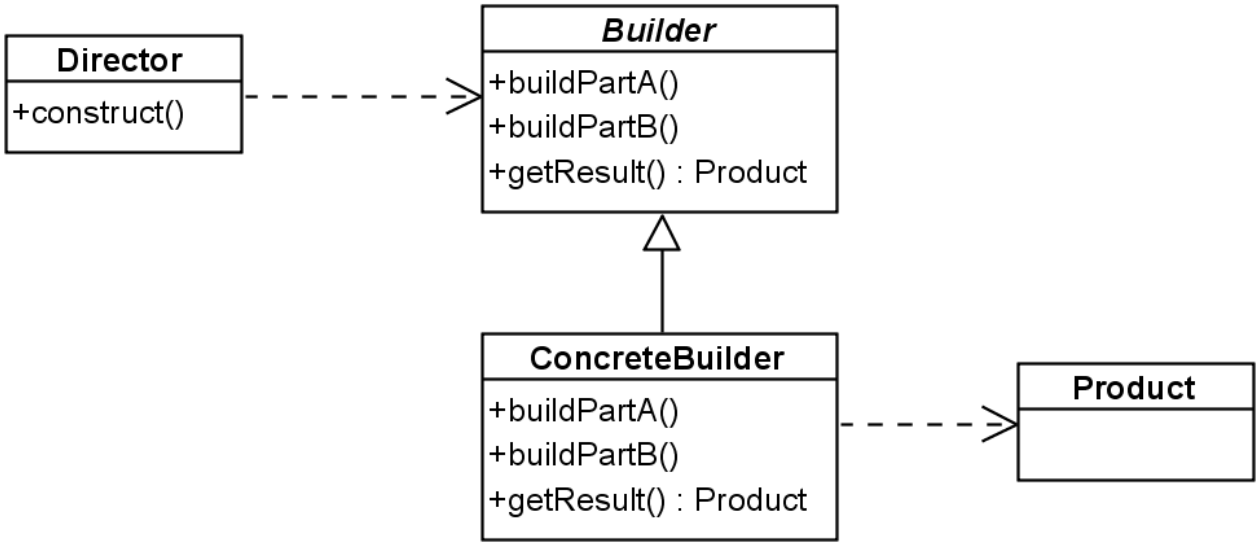


- `CarBuilder` defines the two methods to construct cars.
  Concrete builders must implement these methods.
  For convenience, the instantiation of cars (`buildCar()`) is implemented in `CarBuilder`.
- `CarConstructionDirector` is configured with a `CarBuilder` and calls the construction methods in the correct order.

Two possible car builders

```
class CheapCarBuilder extends CarBuilder {

        void buildEngine() {
                car.setEngine(Engine.SMALL_ENGINE);
        }

        void buildInterior() {
                car.setInterior(Interior.PLASTIC_INTERIOR);
        }
}

class LuxuryCarBuilder extends CarBuilder {

        void buildEngine() {
                car.setEngine(Engine.SPORT_ENGINE);
        }

        void buildInterior() {
                car.setInterior(Interior.WOODEN_INTERIOR);
        }
}
```

2.9.3 Discussion



- Advantages:
  - Creation of objects can be configured on runtime.
  - Concrete builders can use complex logic.
    E.g. a car builder creating cars depending on available parts in storage.
  - Good way to create composite structures.

- Disadvantages:
  - May yield many classes.
  - Only works if all objects can be constructed using the same order.

Builder and Abstract Factory

- Abstract Factory focuses on creating multiple objects of a common family.
  - Abstract Factory knows what object to create.
  - Configuration is fixed after deployment of the software.

- Builder focuses on creating complex objects step by step.
  - The director knows how to construct the object.
  - Configuration is chosen at runtime via the concrete builder.

- Use Abstract Factory for creating objects depending on finite numbers of factors you know in advance.
  E.g. if there are only three kind of cars.
- Use Builder for creating complex objects depending on unbound number of factors that are decided at runtime.
  E.g. if cars can be configured with multiple different parts.

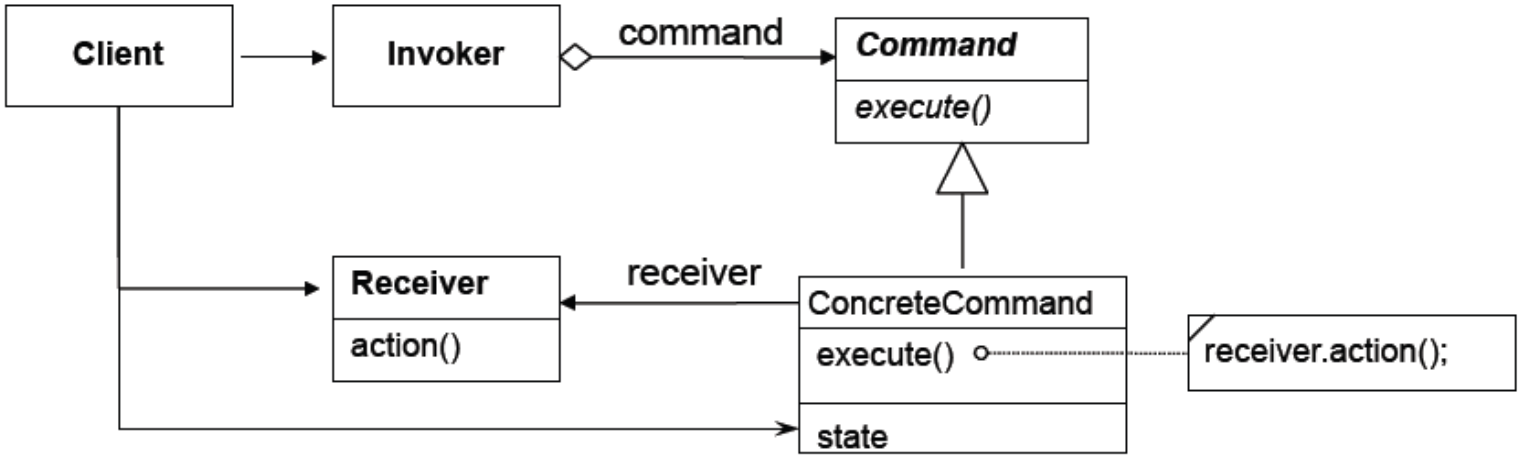
2.10 Command

- 2.10.1 Structure
- 2.10.2 Example
- 2.10.3 Advantages
- 2.10.4 Implementing a command history
- 2.10.5 Implementing Macro Commands

2.10 Command

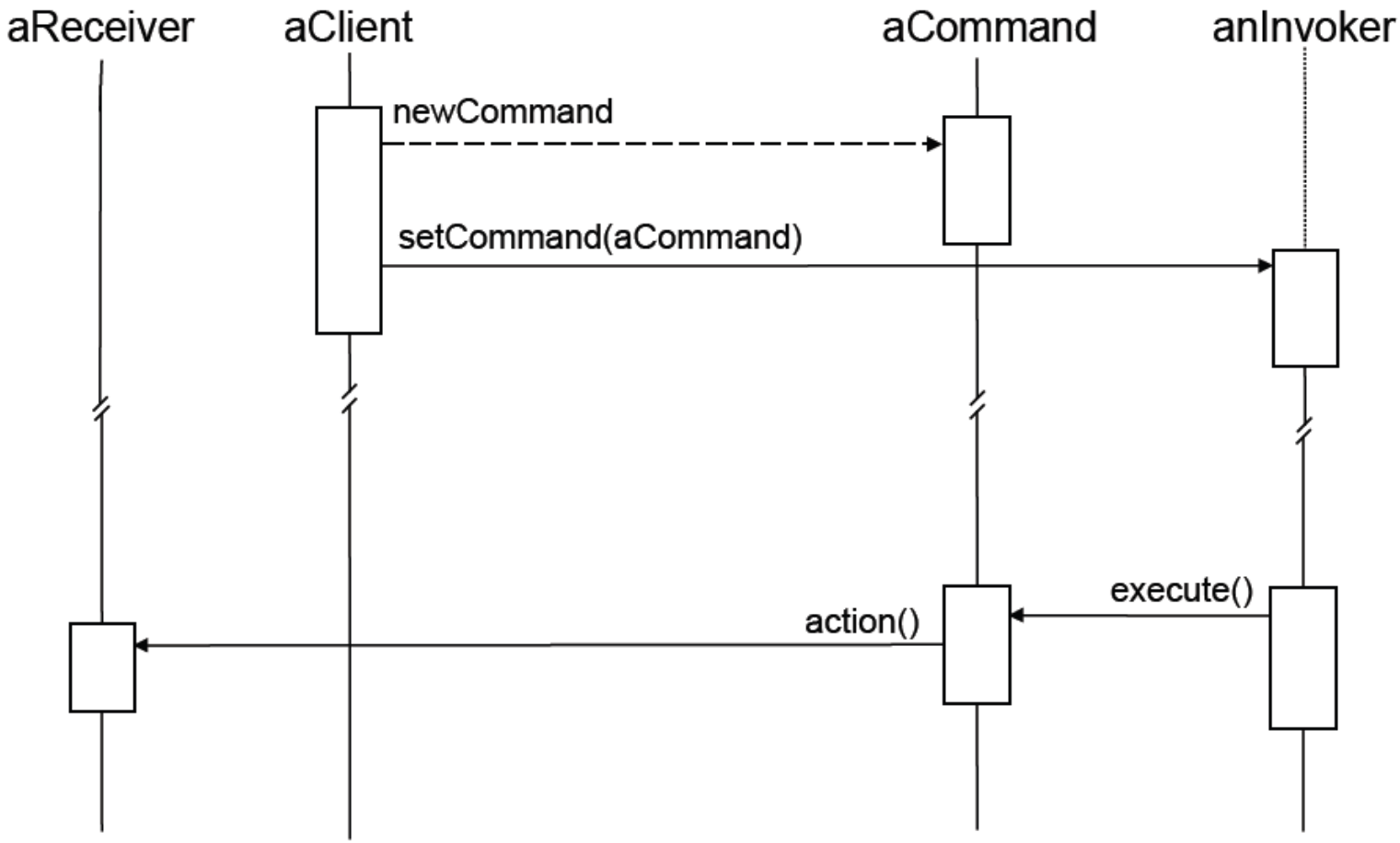
- **Intent:** Encapsulate a request to an object, thereby allowing to:
  - Issue requests without knowing the receiver or the operation being requested.
  - Parameterize clients with different requests.
  - Queue or log requests and support undoable requests.

2.10.1 Structure



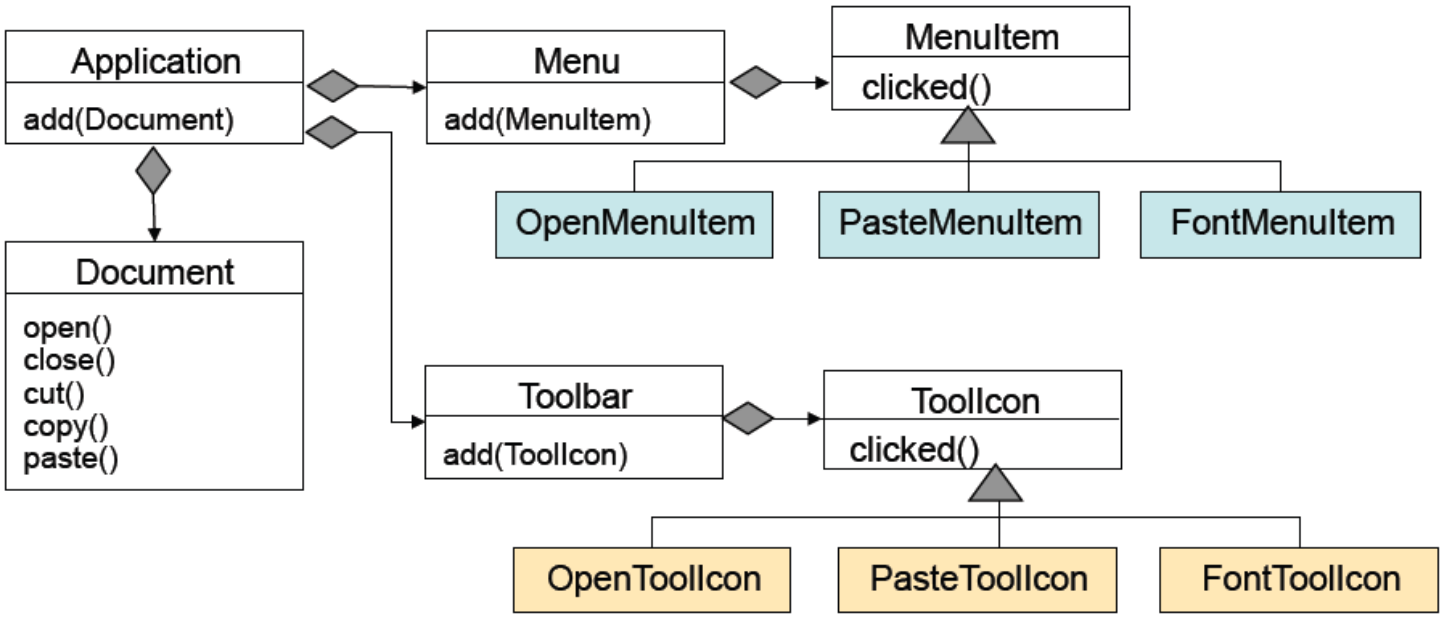
- `Command` declares the interface for executing an operation.
- `ConcreteCommand` defines a receiver-action binding by implementing `execute()`.
- `Client` creates a `ConcreteCommand` object and sets its `Receiver` and configures the command of the `Invoker`.
- `Invoker` asks its command to carry out the request.
- `Receiver` knows how to perform the operations associated with carrying out a request.

Collaboration



2.10.2 Example

- Given user operations:
  - Creating a document, opening, saving, printing a document.
  - Cutting selected text and pasting it back in, etc.



- Want a mechanism for accessing operations from more than one place in UI (a menu and a toolbar).
  We want to **decouple invoker from receiver.**

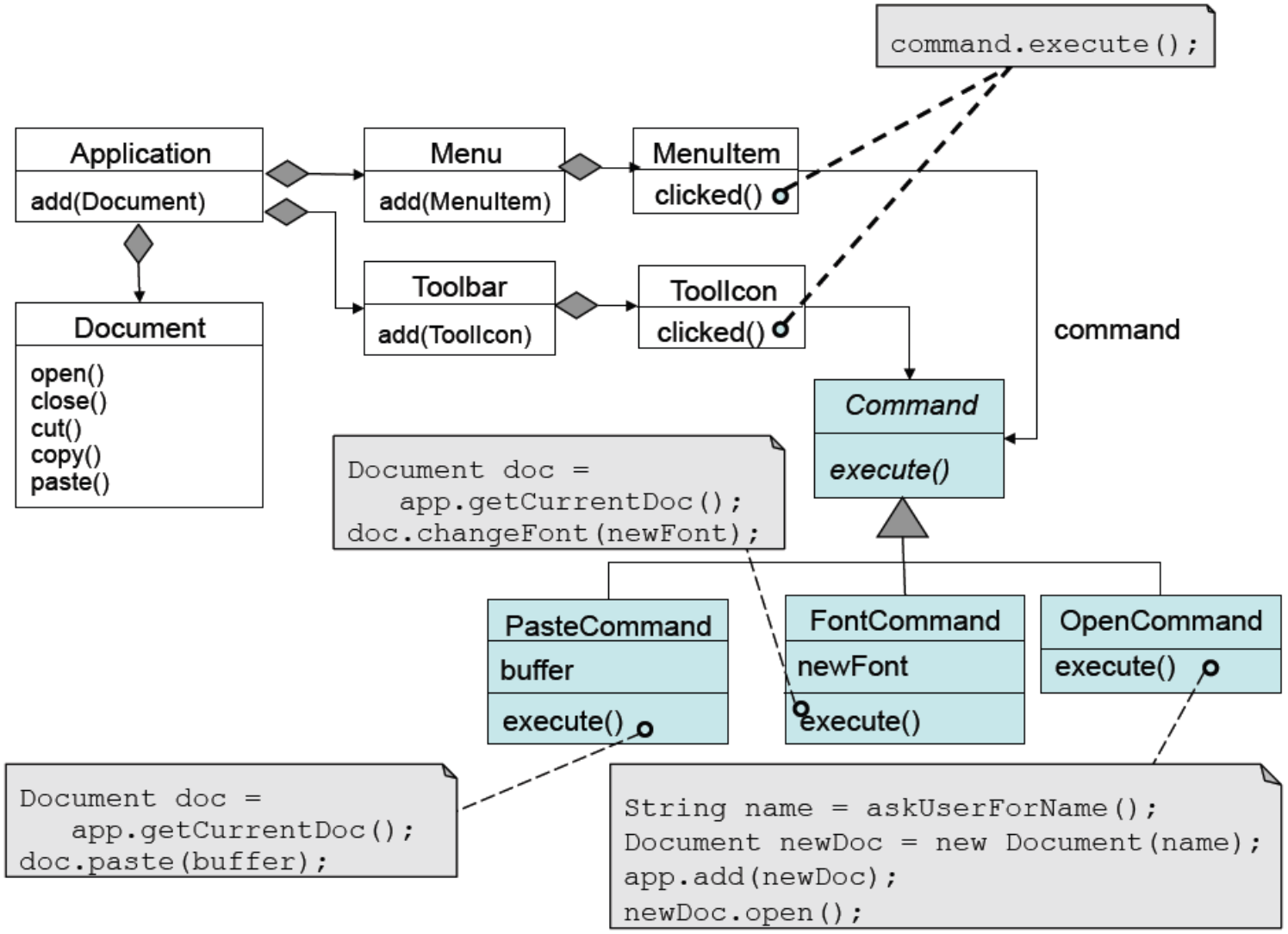
Problems

- The implementation of each of the `MenuItem` subclasses are the same as the implementation of one of the `ToolIcon` subclasses.
  Multiple copies of the same functionality present a maintenance problem.

- Need a mechanism for `MenuItem` and `ToolIcon` to share implementations.
  Need to separate the user interface control from it's implementation so that implementations can be shared.

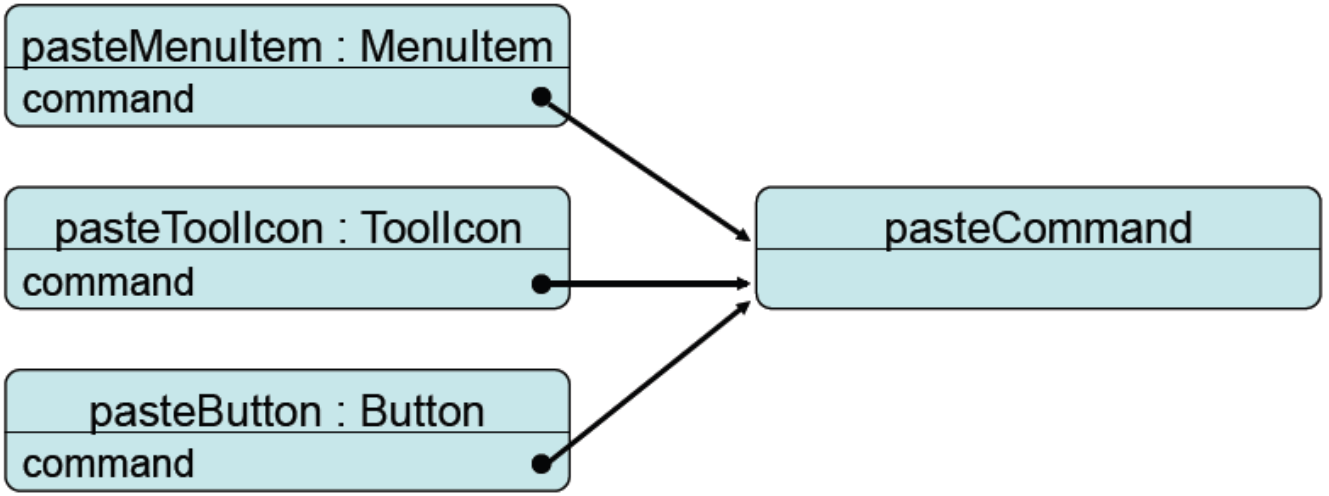
- Want to also support a general undo capability so that the user can reverse previous operations.

Solution with Command



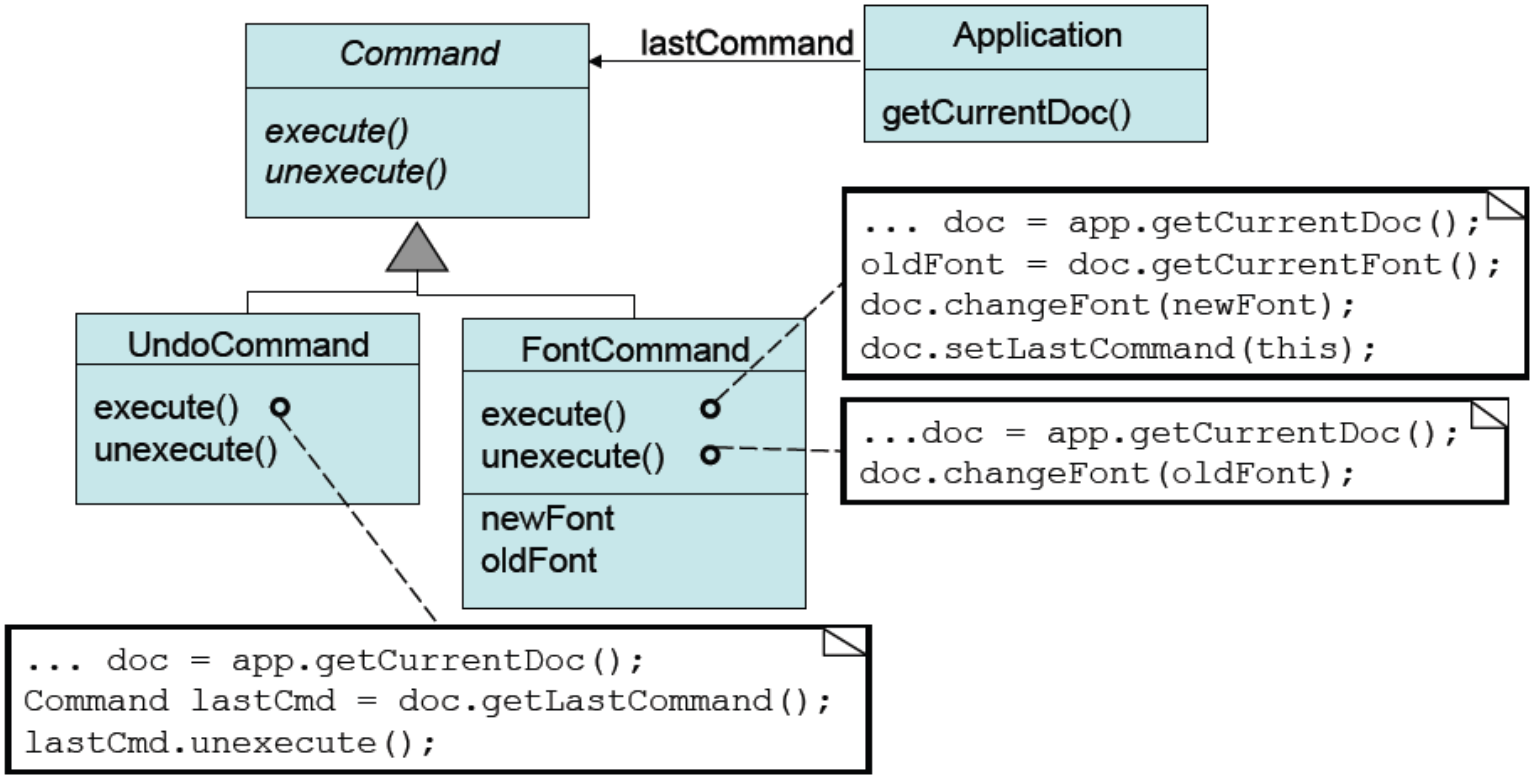
2.10.3 Advantages

- **Implementation Sharing**
  - A command centralizes an operation to a single location so that multiple copies of the code are not necessary.
  - Different user interface controls can share the same implementation (e.g., a button, tool icon, and menu item can all perform the same operation).
  - Decouples the user interface from the operation being performed.



Supporting Undoable Operations

- Commands store enough information to undo the performed operation.
- Each command subclass implements its `unexecute()` function; when `unexecute()` is called the command reverses its action.

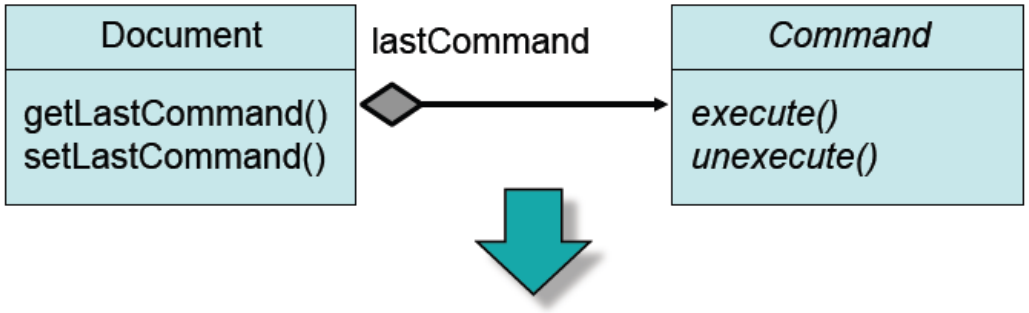


Multiple Levels of Undo

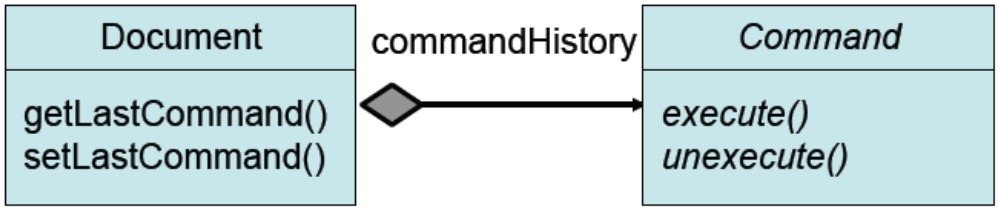
- Undoing more than just the last command allows the user to back up farther and farther each time undo is selected from the menu.

- Adding a redo feature: it would also be nice for a user to be able to redo an undone operation.
  Redo should have multiple levels corresponding to the number of undo's issued by the user.

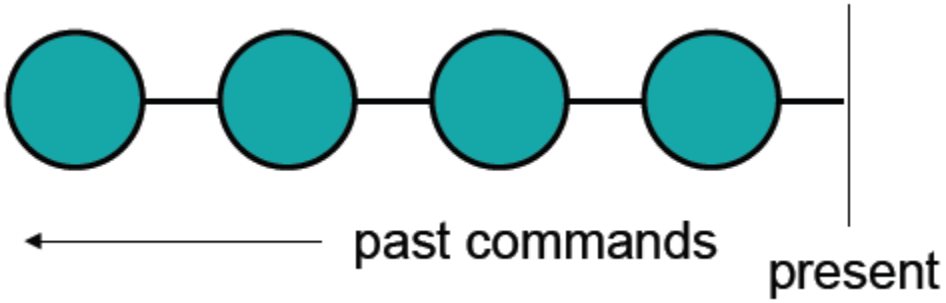
Single Level of Undo



Multiple Levels of Undo

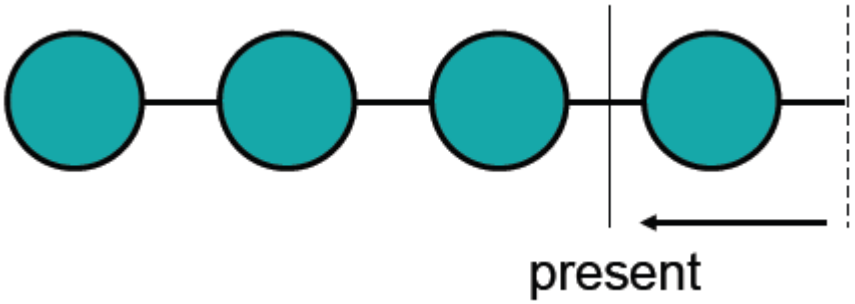


2.10.4 Implementing a command history



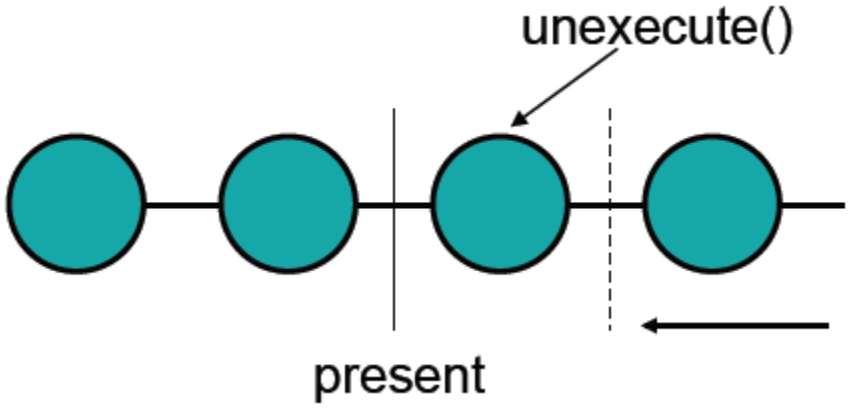
- The command history can be seen as a list of past commands.
- As new commands execute they are added to the front of the history.

Undoing



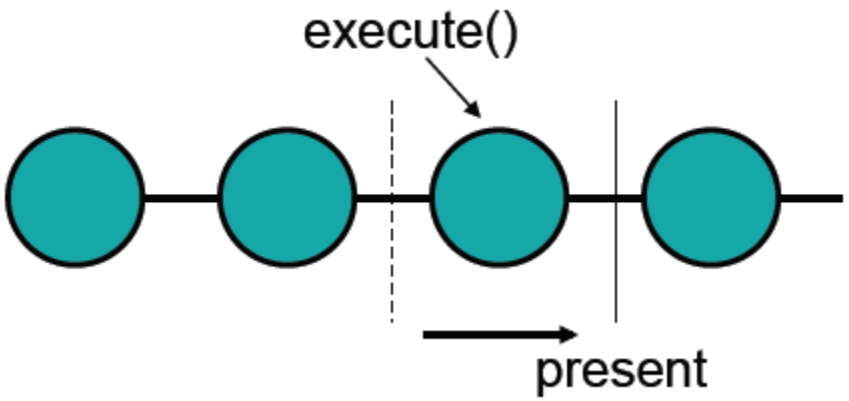
- To undo a command, `unexecute()` is called on the command at the front of the list.
- The present pointer is moved past that command.

Undoing further



- To undo the command before that, unexecute() is called on the next command in the history.
- The present pointer is moved to point before that command.

Redoing



- To redo the command that was just undone, `execute()` is called on that command.
- The present pointer is moved up past that command.

2.10.5 Implementing Macro Commands