2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Dependency Inversion Principle (DIP)
- Callbacks Toolkit
- Strategy combined with Composite = Observer
- Adapter design pattern
- Decorator design pattern
- Inheritance versus Composition

SOLID Object-Oriented Design Principles

- **S**ingle Responsibility Principle (SRP, see Lecture 2)
- **O**pen Closed Principle (OCP, treated later)
- **L**iskov Substitution Principle (LSP, see Lecture 4)
- **I**nterface Segregation Principle (ISP, treated later)
- **D**ependency Inversion Principle (DIP, treated in this lecture)

Dependency Inversion Principle (DIP)

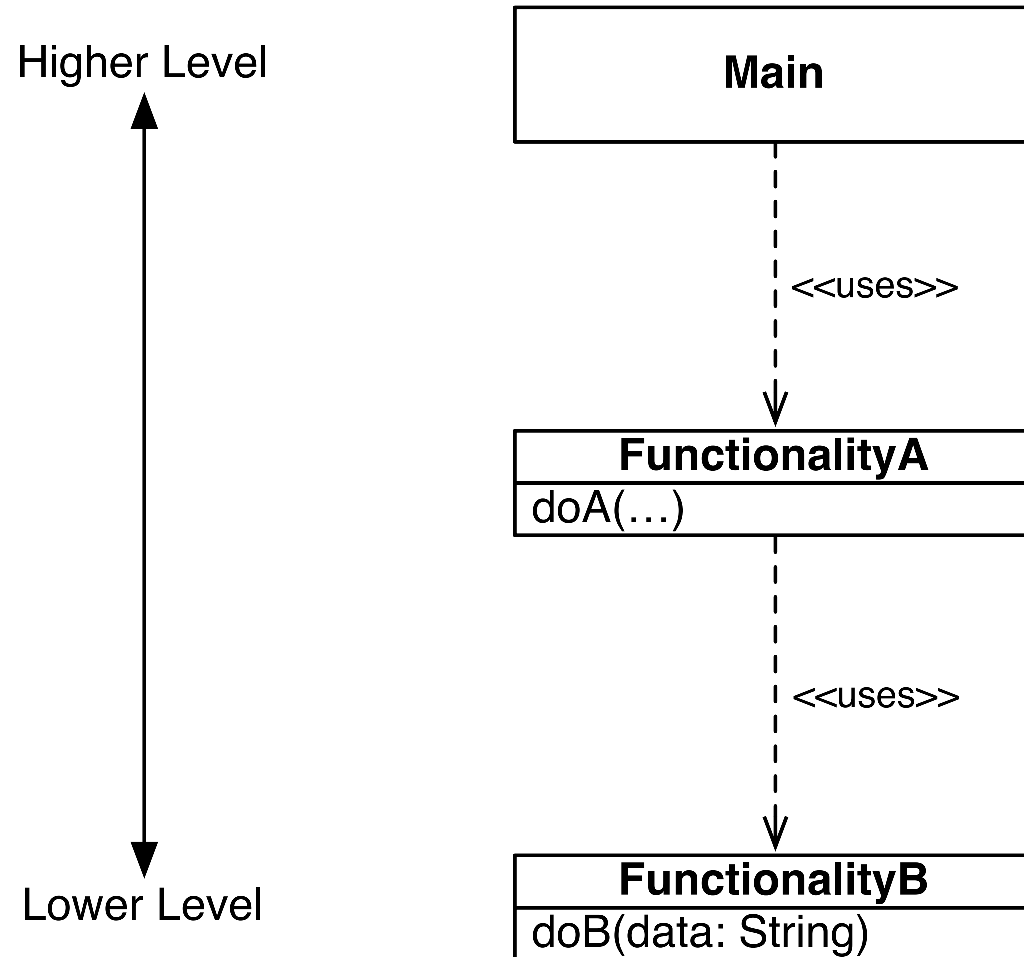
- High-level modules should not depend upon low-level modules.

  Both should depend upon *abstractions*.

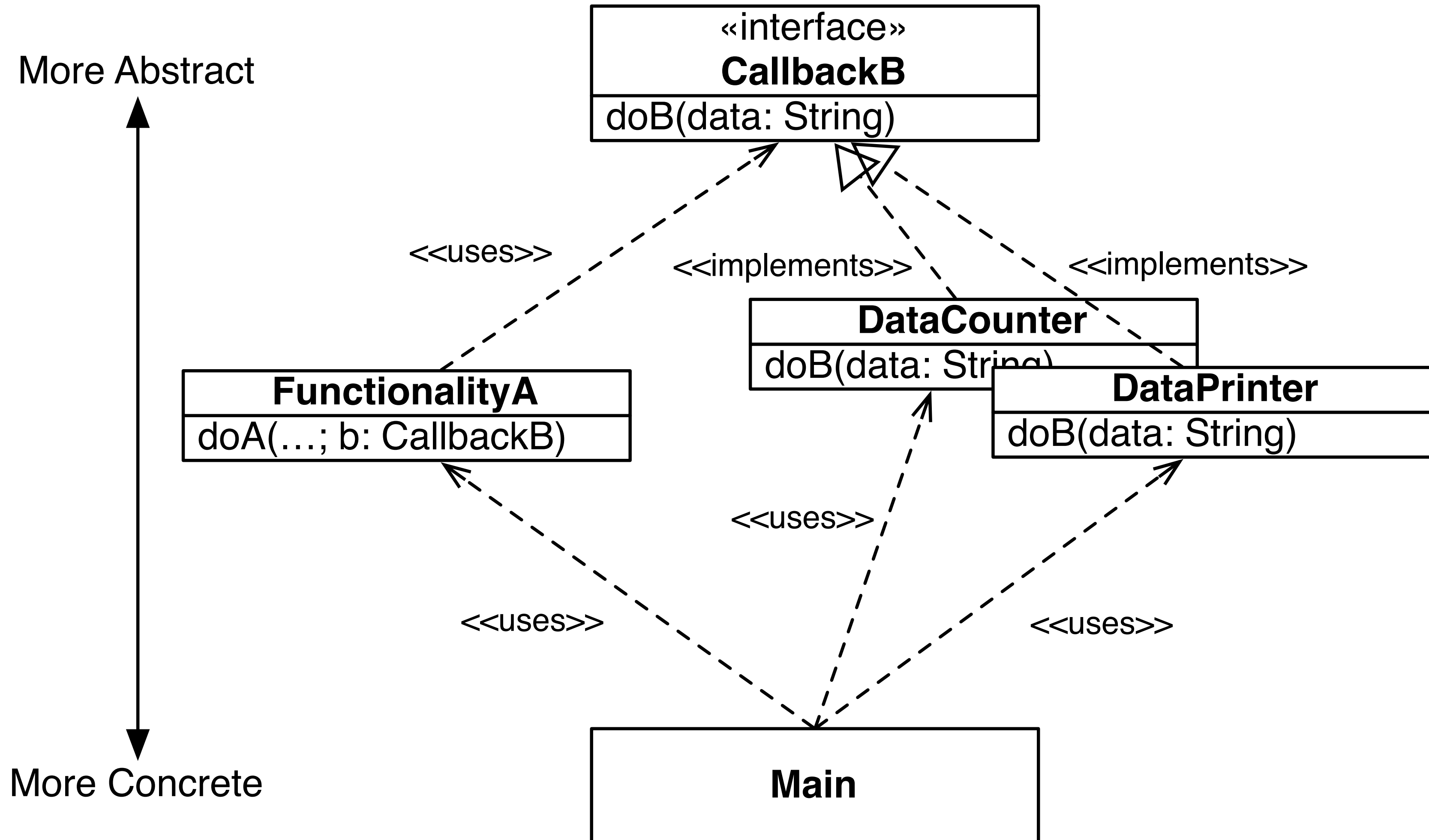
- Abstractions should not depend upon details.

  Details should depend upon abstractions.

Without Callback: Violates DIP



Strategy Design Pattern: Adheres to DIP



Callback Interface

```java
1 public interface CallbackB {
2
3     /**
4      * Processes data.
5      *
6      * @param data  the data to process
7      */
8     void doB(String data);
9
10 }
```

Using the Callback Interface

```
1  public class FunctionalityA {
2
3      /**
4       * Produces a number of data items and processes them.
5       */
6      public static void doA(int n, CallbackB b) {
7          for (int i = 0; i < n; ++ i) {
8              final String data; // data item produced
9              data = i + " produced by A"; // "complex" computation
10             b.doB(data);
11         }
12     }
13
14 }
```

Implementinng the Callback Interface

```
1  public class DataPrinter implements CallbackB {
2
3      /** Name to print in front of each data item. */
4      private final String name;
5
6      /**
7       * Constructs a data printer with given name.
8       */
9      public DataPrinter(String name) {
10         this.name = name;
11     }

12     /**
13      * Prints data.

14     public void doB(String data) {
15         System.out.println(name + " processed " + data);
16     }
17 }
```

Combining Usage and Implementation

```
1  public class Main {
2
3      /**
4       * Invokes functionality A with various processors for functionality B.
5       */
6      public static void main(String[] args) {
7          final int n = Integer.parseInt(args[0]);
8          final CallbackB printer = new DataPrinter(args[1]);
9          FunctionalityA.doA(n, printer);
10     }
11
12 }
```

What If We Have a Class with Another Interface?

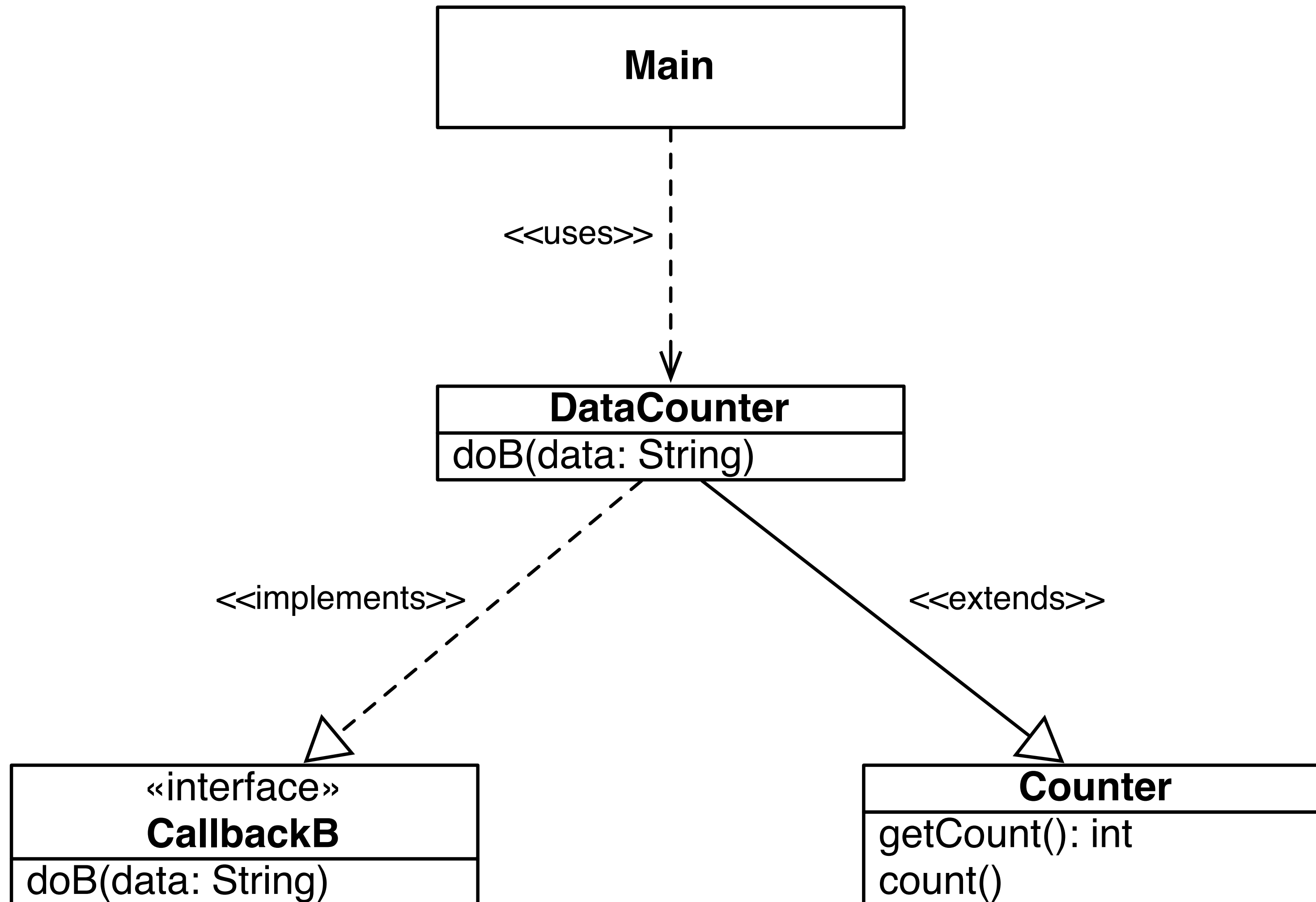
```java
1 public class Counter {
2     /** Current count */
3     private long count;
4     /**
5      * Gets current count.
6      */
7     public long getCount() {
8         return count;
9     }
10    /**
11     * Counts one up.
12     */
13    public void count() {
14        ++ count;
15    }
16 }
```

Wrapping a Class to Provide a Callback, via Inheritance

```
1  /**
2   * Wraps a {@code Counter} to obtain a {@code CallbackB},
3   * to count how many data items were processed.
4   */
5  class DataCounter extends Counter implements CallbackB {
6
7      /**
8       * Counts a data item.  The content of the item is ignored.
9      public void doB(String data) {
10         count();
11     }
12 }
```

```
1          final DataCounter counter = new DataCounter();
2          FunctionalityA.doA(n, counter);
3          System.out.println(counter.getCount()
4                  + " items received from A");
```

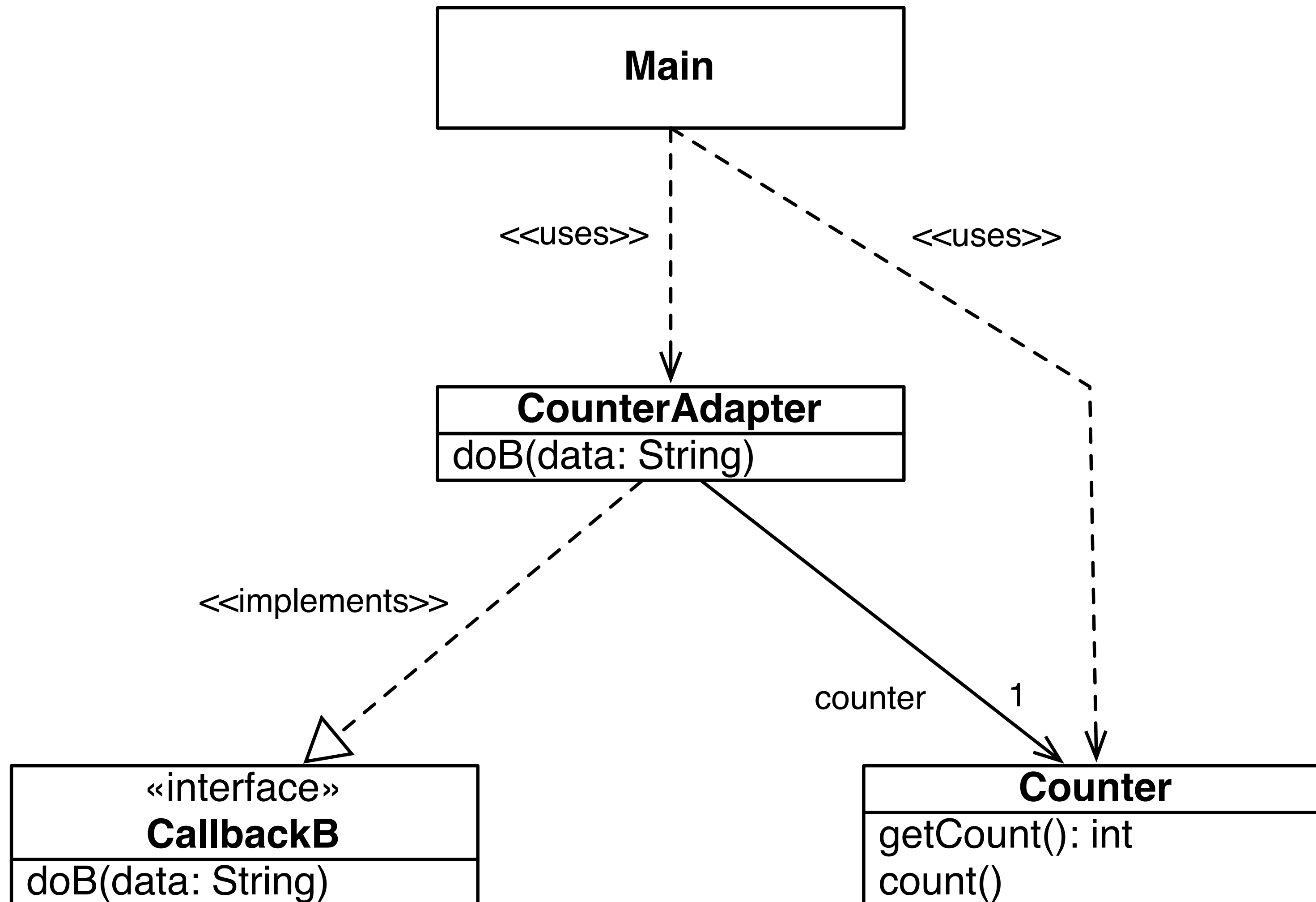
Wrapping via Inheritance



Adapting a Class to Provide a Callback, via Composition

```
1  class CounterAdapter implements CallbackB {
2
3      /** Counter being adapted. */
4      private final Counter counter;
5
6      /**
7       * Constructs a new adapter for a given counter
8       */
9      public CounterAdapter(Counter counter) {
10         this.counter = counter;
11     }
12
13     /**
14      * Counts a data item.  The content of the item is ignored.
15     public void doB(String data) {
16         counter.count();
17     }
```

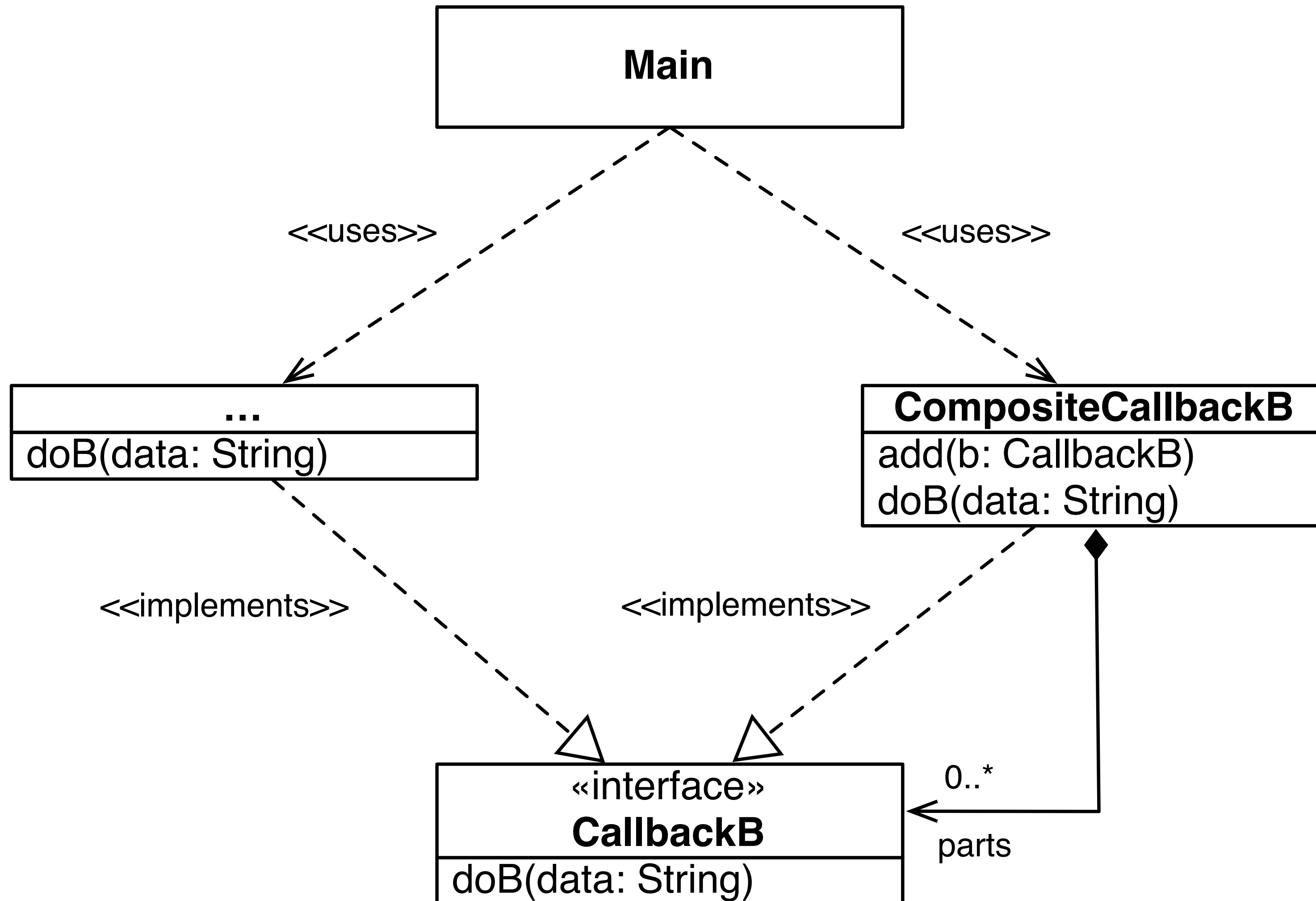
Adapting via Composition



Distributing a Callback to Multiple Callback Handlers

```
1 /**
2  * A composite callback that distributes data
3  * to all registered callbacks for functionality B.
4  */
5 public class CompositeCallbackB implements CallbackB {
6     private final List<CallbackB> parts;
7     public CompositeCallbackB() {
8         parts = new ArrayList<CallbackB>();
9     }
10    public void add(CallbackB b) {
11        parts.add(b);
12    }
13    public void doB(String data) {
14        for (CallbackB b : parts) {
15            b.doB(data);
16        }
17    }
18 }
```

Distributing a Callback = Application of Composite Pattern



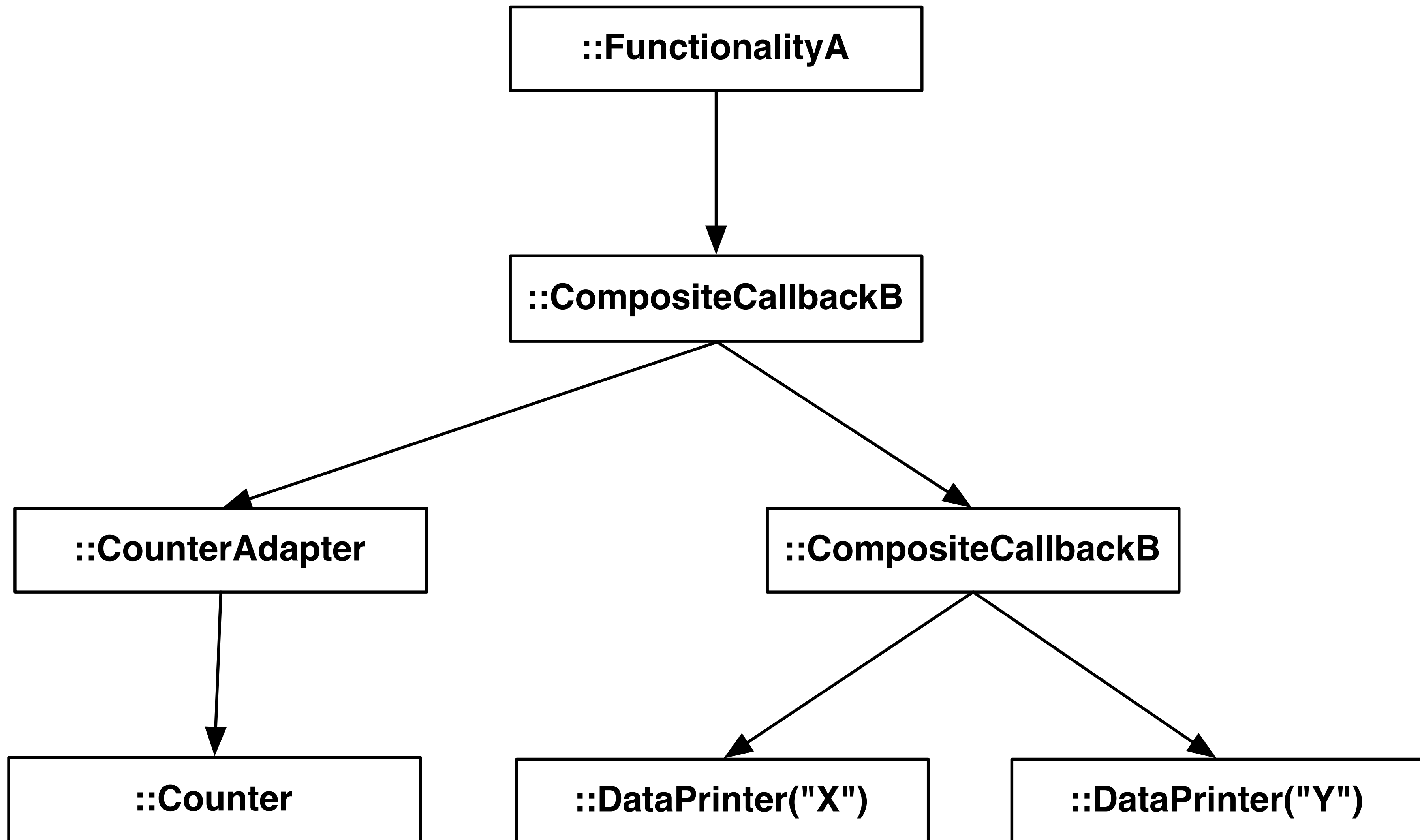
Using a Callback Distributor (Observer Pattern)

```
1         final CallbackB printer = new DataPrinter(args[1]);
2         final DataCounter counter = new DataCounter();
3         final CompositeCallbackB printer_counter
4                 = new CompositeCallbackB();
5         printer_counter.add(printer);
6         printer_counter.add(counter);
7         FunctionalityA.doA(n, printer_counter);
8         System.out.println(counter.getCount()
9                 + " items received from A");
```

`FunctionalityA` + `CompositeCallbackB` = Observable Subject

Cf. Observer pattern

Hierarchical Distribution of a Callback



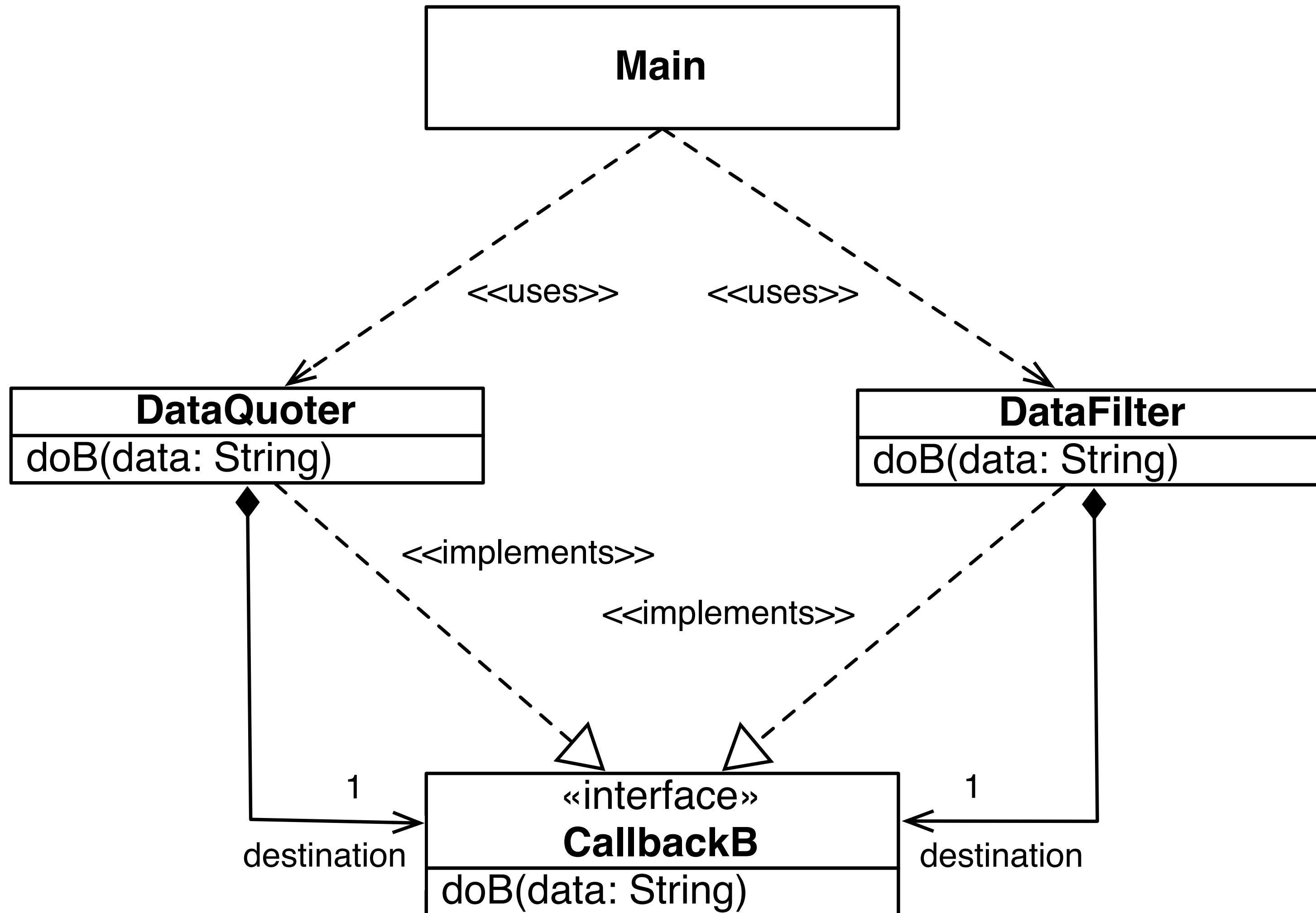
Modifying a Callback's Behavior

```
1 /**
2  * Put quotes around data and pass on to another {@code CallbackB}.
3  */
4 class DataQuoter implements CallbackB {
5     private CallbackB destination;
6     public DataQuoter(CallbackB destination) {
7         this.destination = destination;
8     }
9     public void doB(String data) {
10        destination.doB('"' + data + '"');
11    }
12 }
```

Filtering a Callback

```
1 /**
2  * Filter to selectively pass data to another {@code CallbackB}.
3  */
4 class DataFilter implements CallbackB {
5     private CallbackB destination;
6     private String pattern;
7     public DataFilter(CallbackB destination, String pattern) {
8         this.destination = destination;
9         this.pattern = pattern;
10    }
11    public void doB(String data) {
12        if (data.contains(pattern)) {
13            destination.doB(data);
14        }
15    }
16 }
```

Decorating via Composition

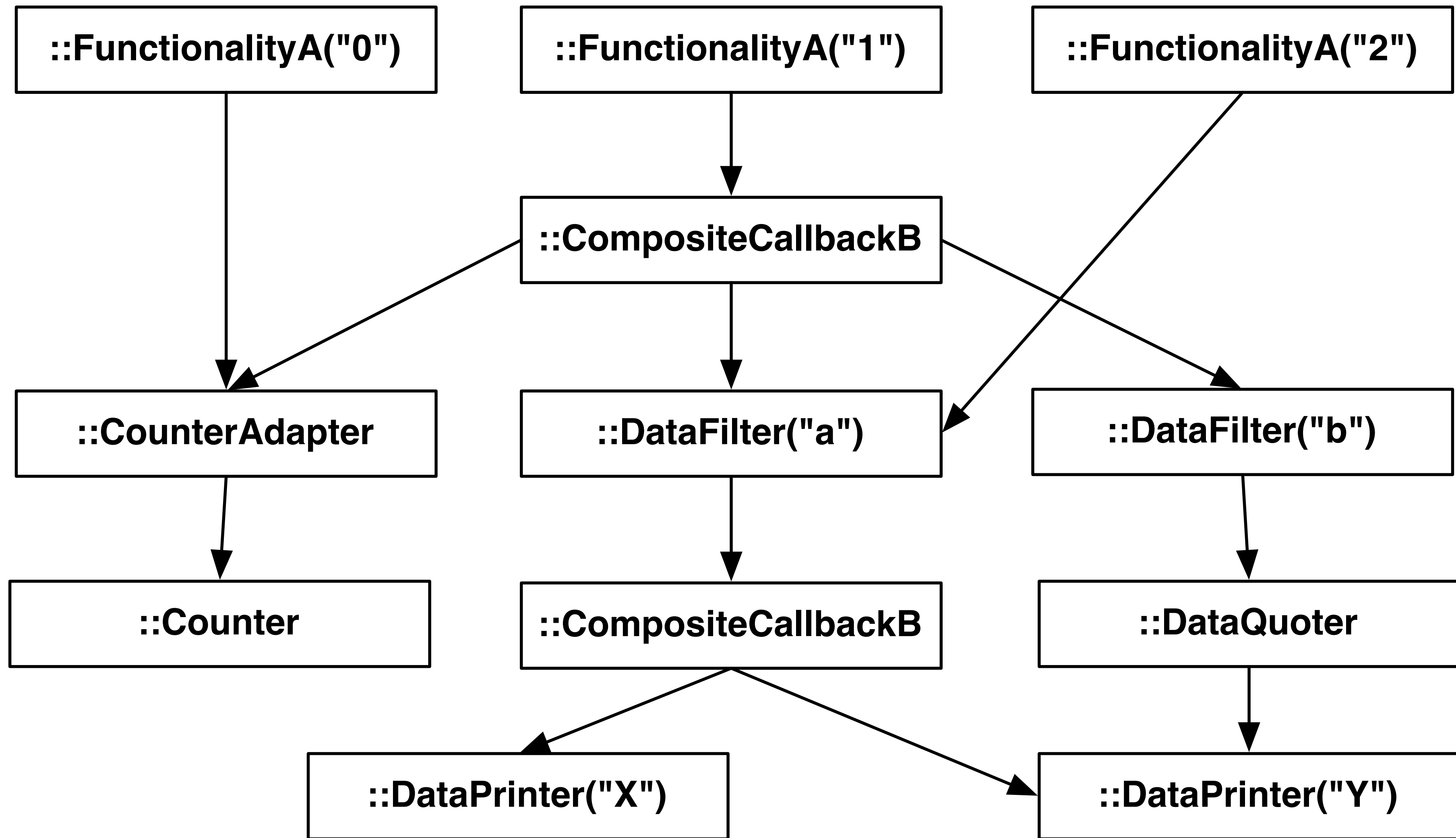


Combining Decorators

```
new DataFilter(new DataQuoter(new DataPrinter("PQF")), "1")
```

Prints quoted versions that contain a `1`

Flexible Callback Toolkit



Flexible Callback Toolkit: Evaluation

Advantages:

- Small number of components: adapters, distributors, decorators
- Can be combined in unlimited ways

Disadvantages:

- Performance penalty because of all indirections and data transfers
- Configuration needs to be programmed and is created at run time

Later: Domain Specific Language to hide these details

Variations

Decision on *what data* to transfer:

- Push to Observers: Subject Decides
- Notify Observer + Pull from Subject: Observer Decides
- Combination

Direction of data transfer:

- Update the Observers: Subject to Observer
- Query the Observer: Observer to Subject
- Combination

Adapter Design Pattern (adapted from Eddie Burris)

Intent

- The Adapter design pattern is useful in situations where an existing class provides a needed service but there is a mismatch between
  - the interface offered and
  - the interface that clients expect.
- The Adapter pattern shows how to convert the interface of the existing class into the interface that clients expect.

Adapter: Antipattern 1

- Modify (hack) the existing class to offer the expected interface.
- Why not a good idea?

Adapter: Antipattern 2

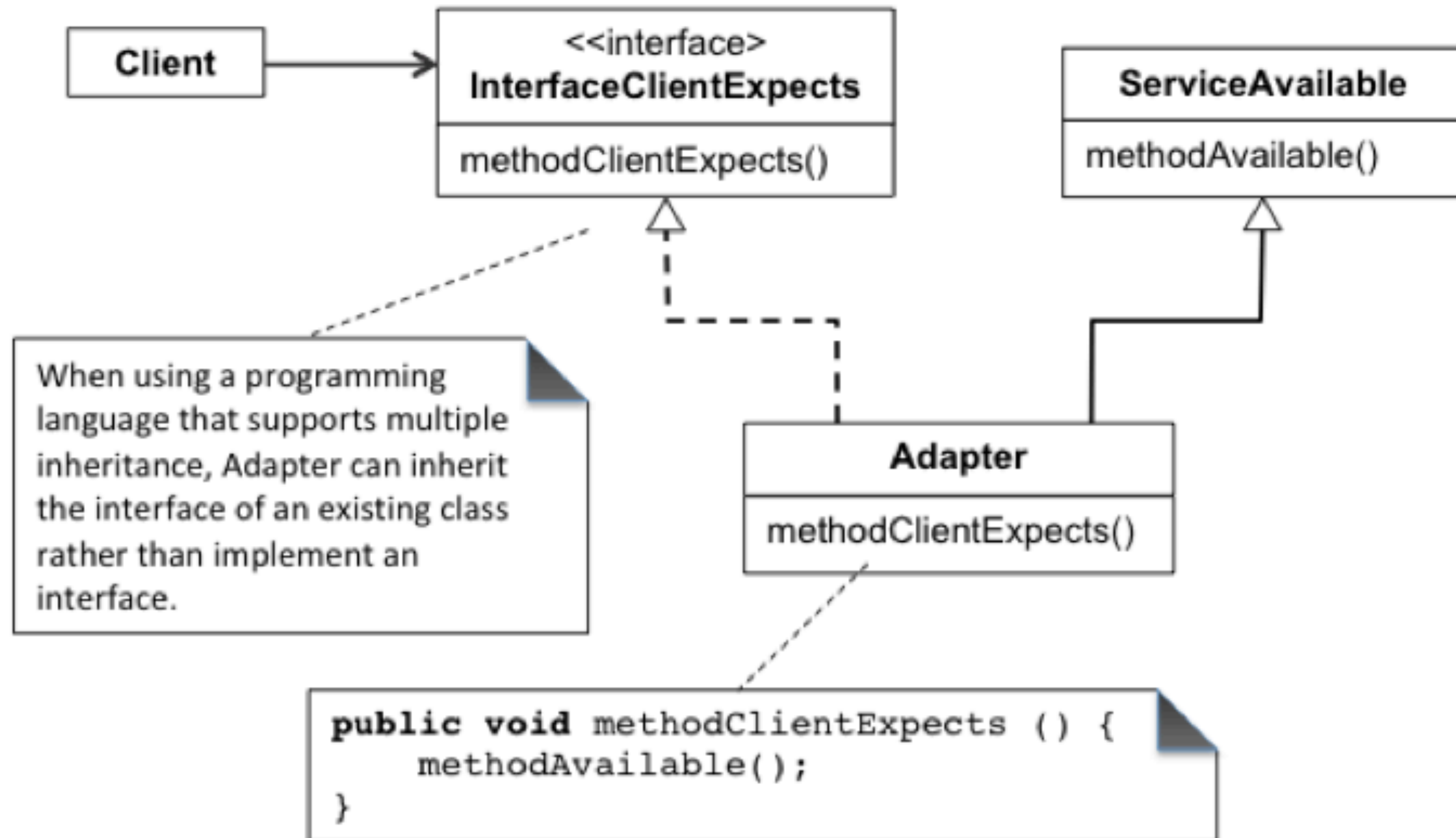
Why not modify existing class? Because

- It is easy to introduce defects in the internals of working class.
- You loose the original.

Alternative:

- Modify a copy of the existing class to offer the expected interface.
- Why also not a good idea?

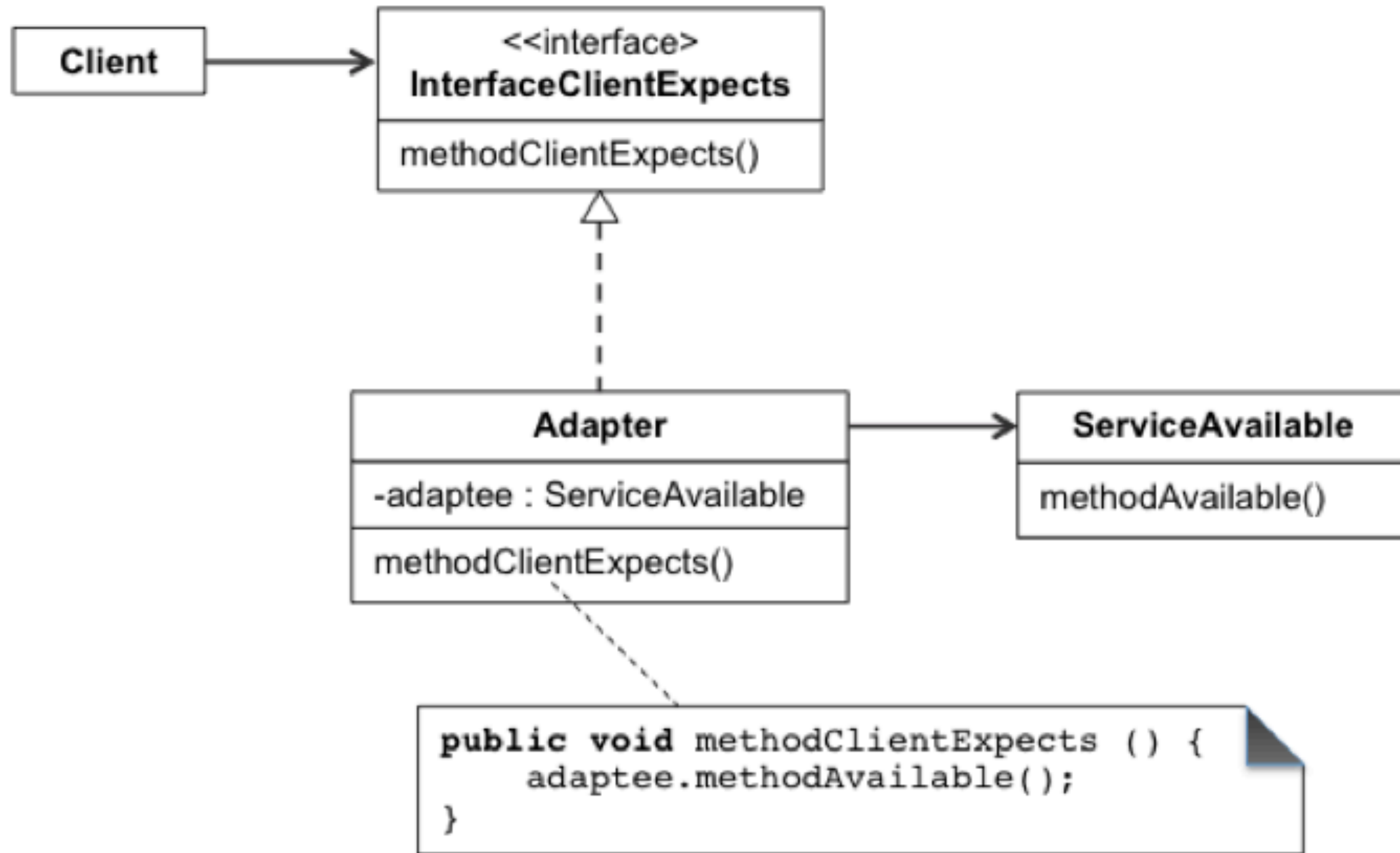
Adapter Pattern using Inheritance (from Burris)



Adapter via Inheritance

- Reuses existing class without modifying or copying it.

Adapter Pattern using Composition (from Burris)



Adapter via Composition

- Reuses existing class without modifying or copying it.

Inheritance versus Composition

- Inheritance creates tighter coupling, to a specific implementation

  Couples to a *class*; gives compile-time dependency

  Cannot easily vary the underlying implementation of adapted class

- Inherited **`protected`** internals are exposed

- Inheritance yields class with larger interface

  Violates *Interface Separation Principle* (ISP)

- Composition offers looser coupling: couples to an *object*

  Can vary the underlying implementation of adapted class.

  Extra cost: indirection via stored reference to adapted class

Decorator Design Pattern (adapted from Eddie Burris)

Intent

- The decorator design pattern provides a way of attaching additional responsibilities to an object dynamically.
- It uses object composition rather than class inheritance for a lightweight flexible approach to adding responsibilities to objects at runtime.

Decorator: Antipatterns

- Modify (a copy of) an existing class to offer the expected service.
- Even worse than with the Adapter!

  Why?

Decorator: Antipatterns Explained

- Decorators obtained by modification cannot be combined

  Combinatorial explosion when needing multiple decorations

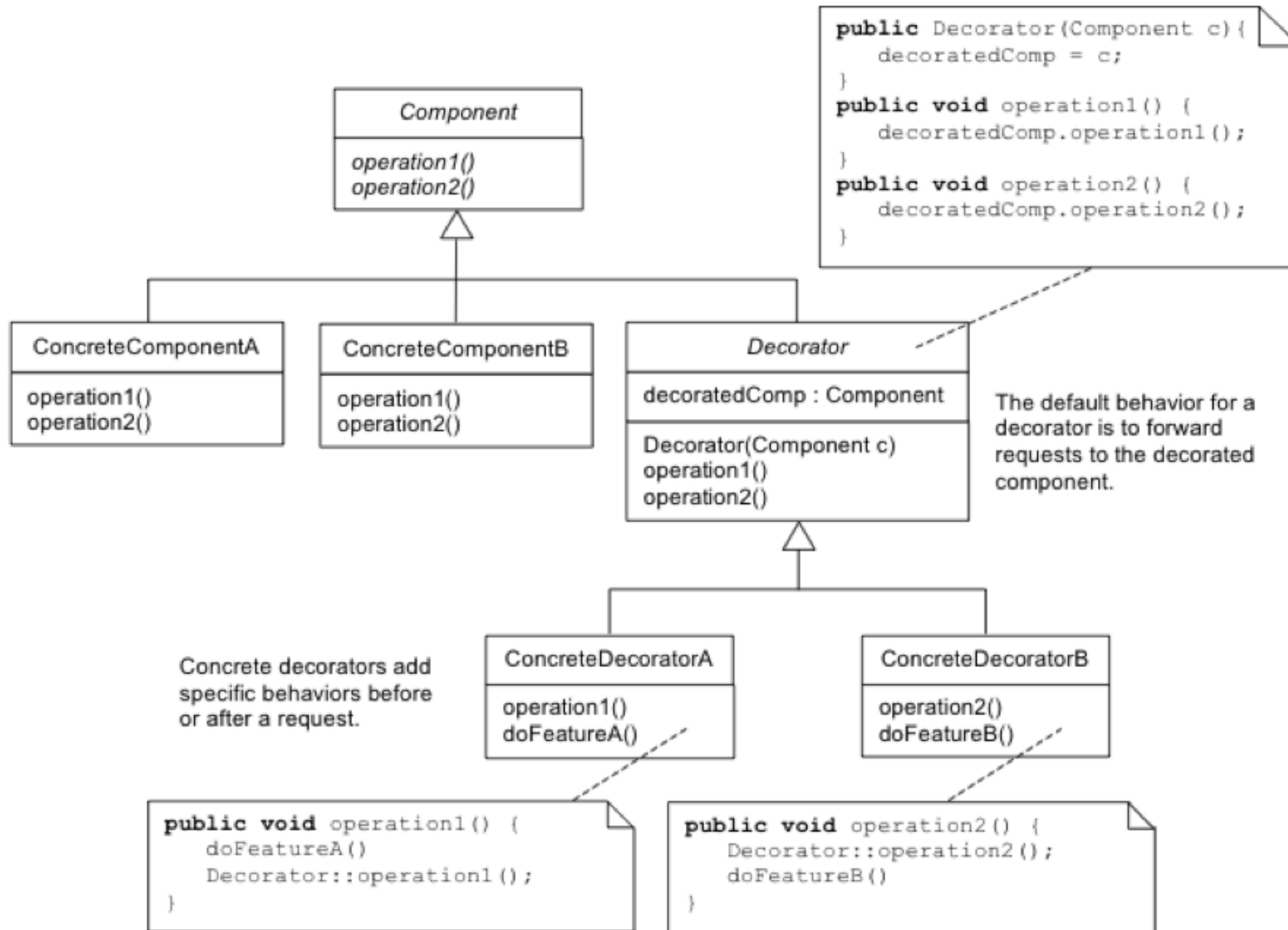
  Example: quoting, trimming, reversing, capitalizing a string

- Decorators obtained by composition can be mixed and matched

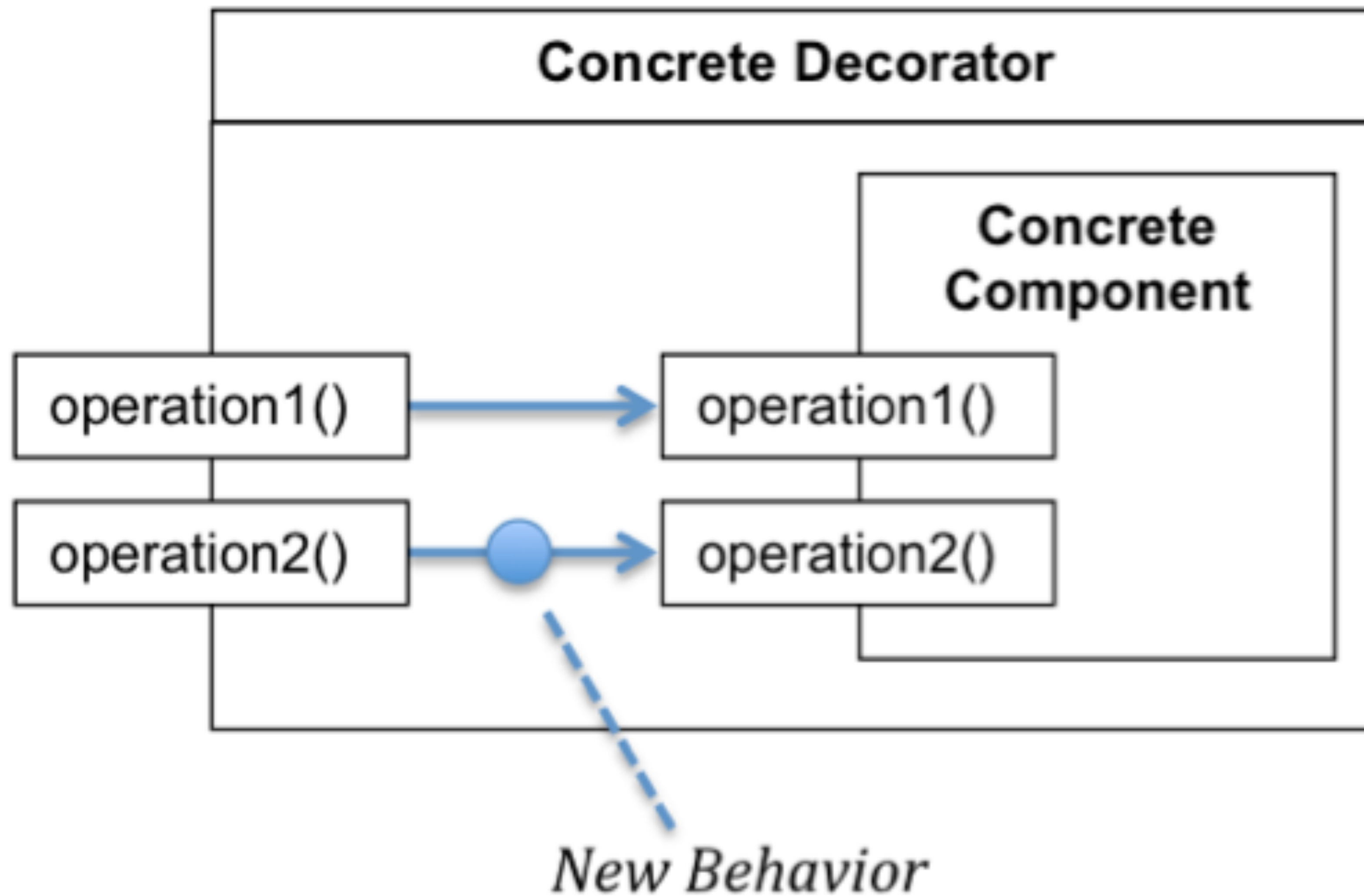
  Small set of decorators can be combined at will

  Similar to functional composition

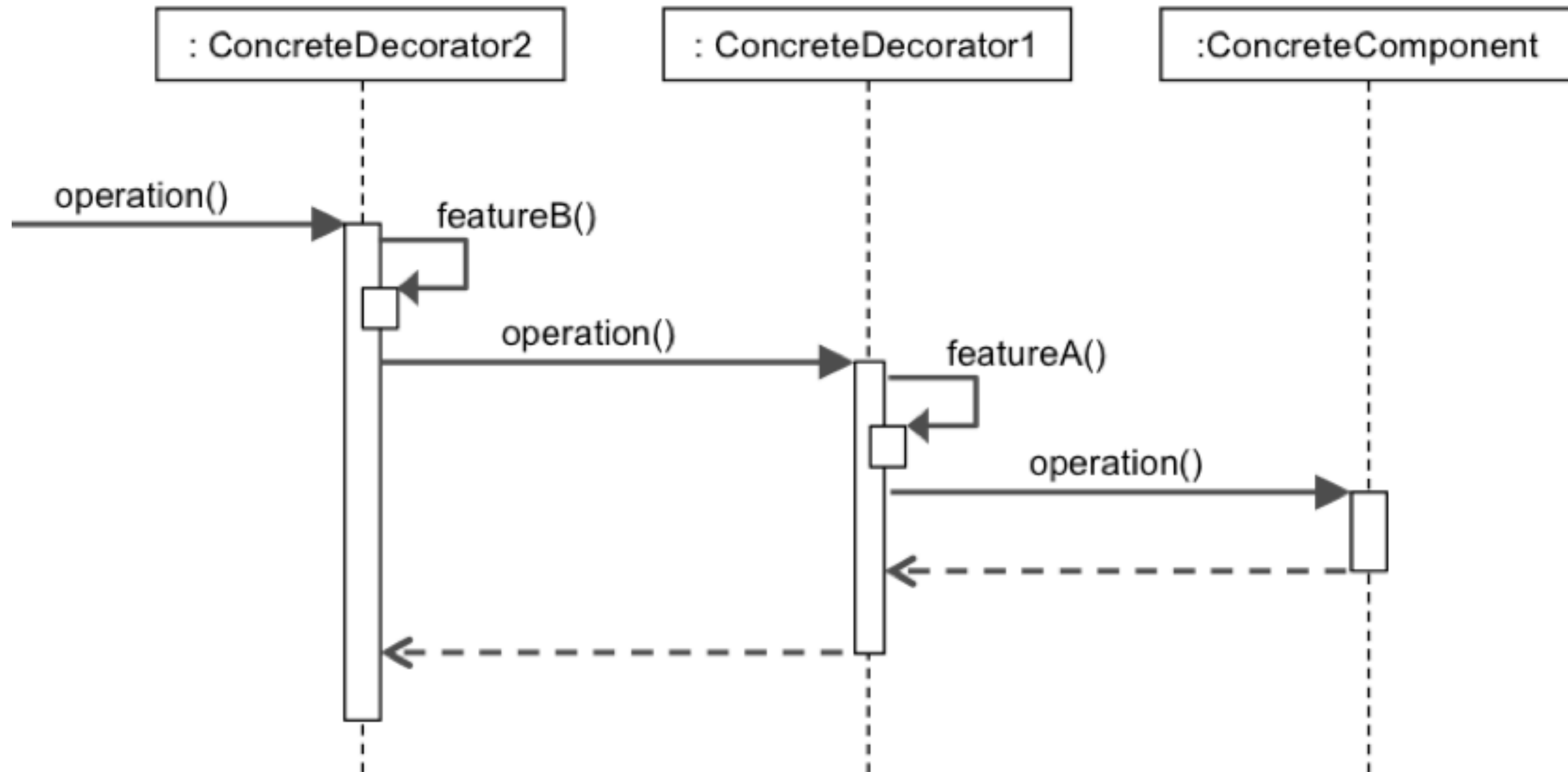
Decorator Pattern using Composition (from Burris)



Decorator Pattern Conceptual Diagram (from Burris)



Decorator Pattern Sequence Diagram (from Burris)



Comparison of Adapter and Decorator Patterns

	Adapter	**Decorator**
Interface	differs from adapted	same as decorated
Functionality	same as adapted	differs from decorated
Realization	inheritance or compostion	composition

Summary

- Dependency Inversion Principle (DIP)
- Flexible Callbacks Toolkit: adapt, distribute, decorate
- Strategy combined with Composite = Observer
- Handout: *From Callbacks to Design Patterns*
- Design Pattern: *Adapter*
- Design Pattern: *Decorator*
- Inheritance versus Composition