2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

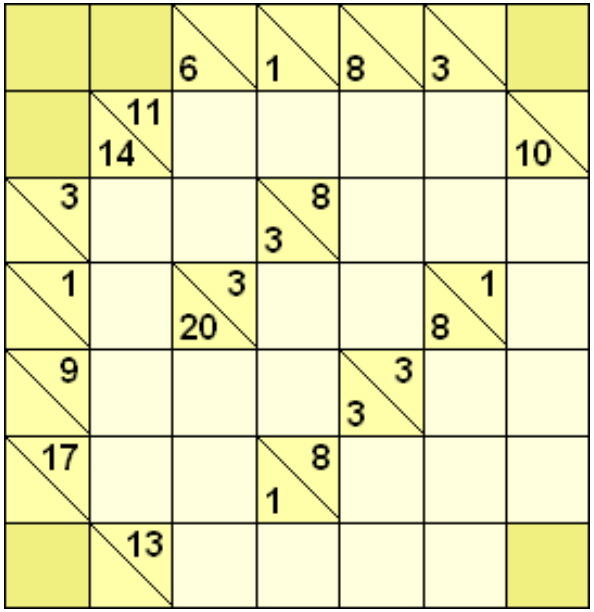
http://canvas.tue.nl/courses/473

Overview

- Dependency Inversion Principle (DIP), Dependency Injection (DI)
- Callbacks Toolkit
- Observer pattern combines Strategy with Composite
- Adapter design pattern
- Decorator design pattern
- Inheritance versus Composition
- See handout *From Callbacks to Design Patterns*

Assignments: Simple Kakuro Helper

- Recursion: base case, inductive step
- Performance: opportunity for *Branch & Bound*
- Avoid code duplication (DRY), especially in `Intersector`



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 5)
- **I**nterface Segregation Principle (ISP, treated later)
- **D**ependency Inversion Principle (DIP, treated in this lecture)

Dependency Inversion Principle (DIP)

Module A *depends on* module B:

- A cannot be compiled/used without having B
- This hinders testing and reuse of A

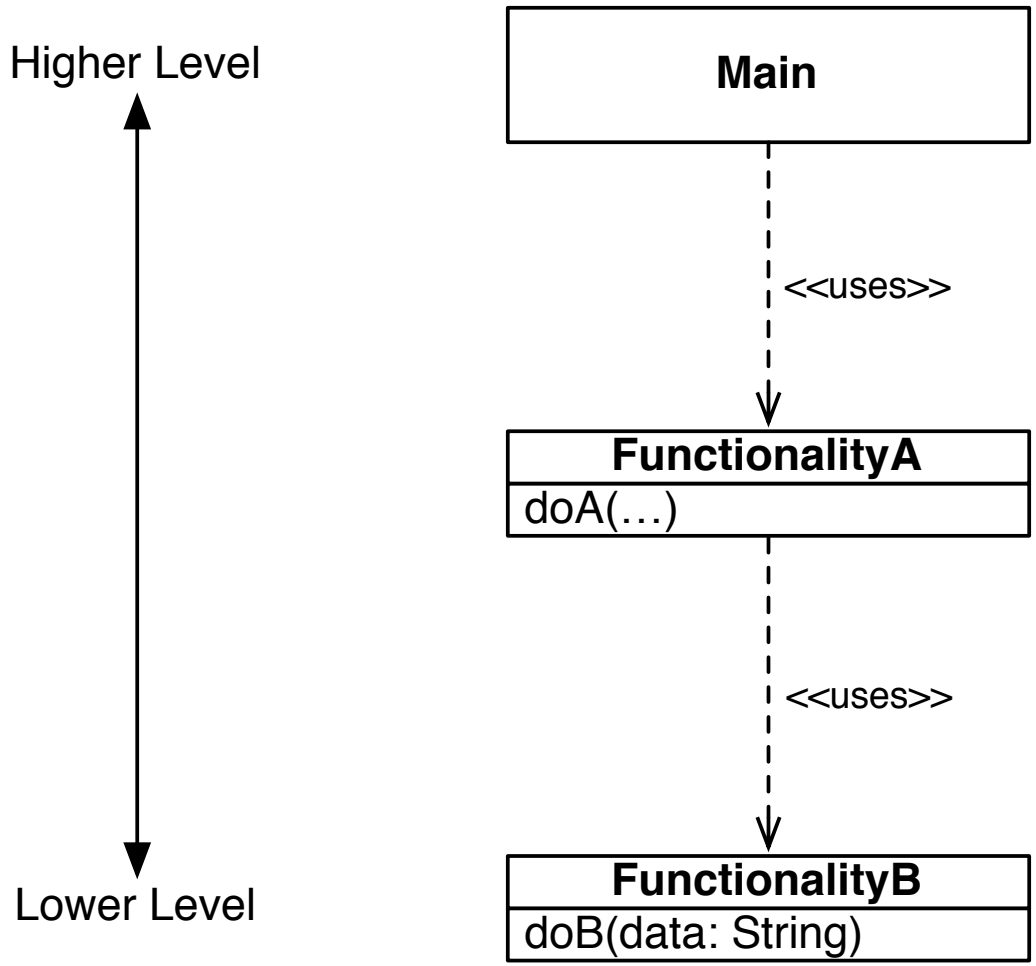
Dependency Inversion Principle (DIP):

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

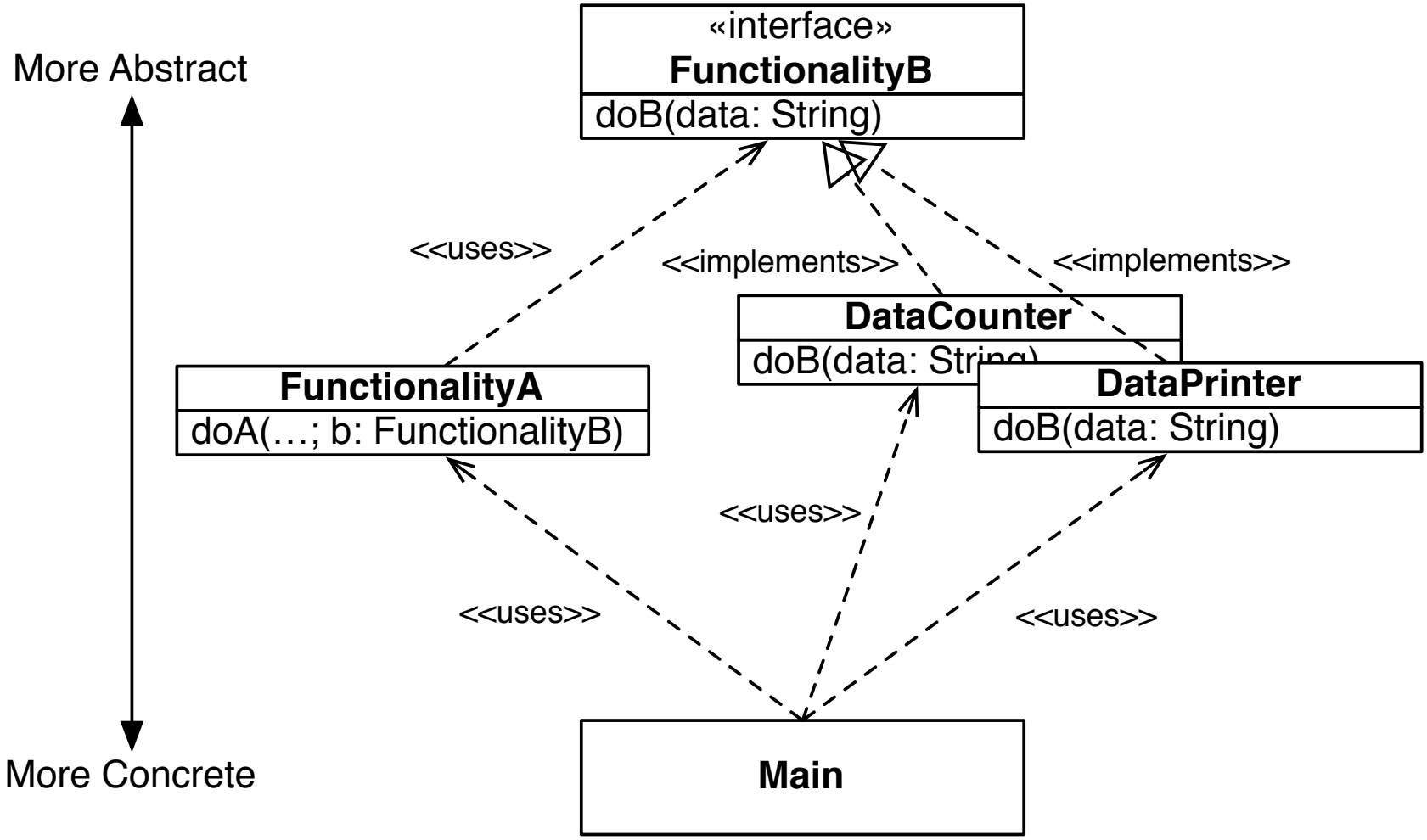
  Both should depend upon *abstractions*.
- Abstractions should not depend upon details.

  Details should depend upon abstractions.

Without Callback: Violates DIP



Strategy Design Pattern: Adheres more to DIP



Callback Interface

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

Could do same thing for `FunctionalityA` (but here we won't)

Using the Callback Interface via a Parameter

```
 1 public class FunctionalityA {
 2
 3     /**
 4      * Produces a number of data items and processes them.
 5      */
 6     public static void doA(int n, FunctionalityB 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 }
```

Implementing the Callback Interface

```
1  public class DataPrinter implements FunctionalityB {
2
3      /** Name to print in front of each data item. */
4      private final String name;
5
6      /**
7        * Prints data.
8        */
9      @Override
10     public void doB(String data) {
11         System.out.println(name + " processed " + data);
12     }
13
14 }
```

Integrating 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[1]);
 8         final FunctionalityB printer = new DataPrinter(args[2]);
 9         FunctionalityA.doA(n, printer);
10     }
11
12 }
```

- Line 9: *Depencency Injection* (DI) of `printer` into `FunctionalityA`
- At compile time, `FunctionalityA` does not depend on `printer`
- At run-time (after configuring), `FunctionalityA` depends on `printer`

What If We Already Have a Class with Another Interface?

?

What If We Already Have a Class with Another Interface?

```
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

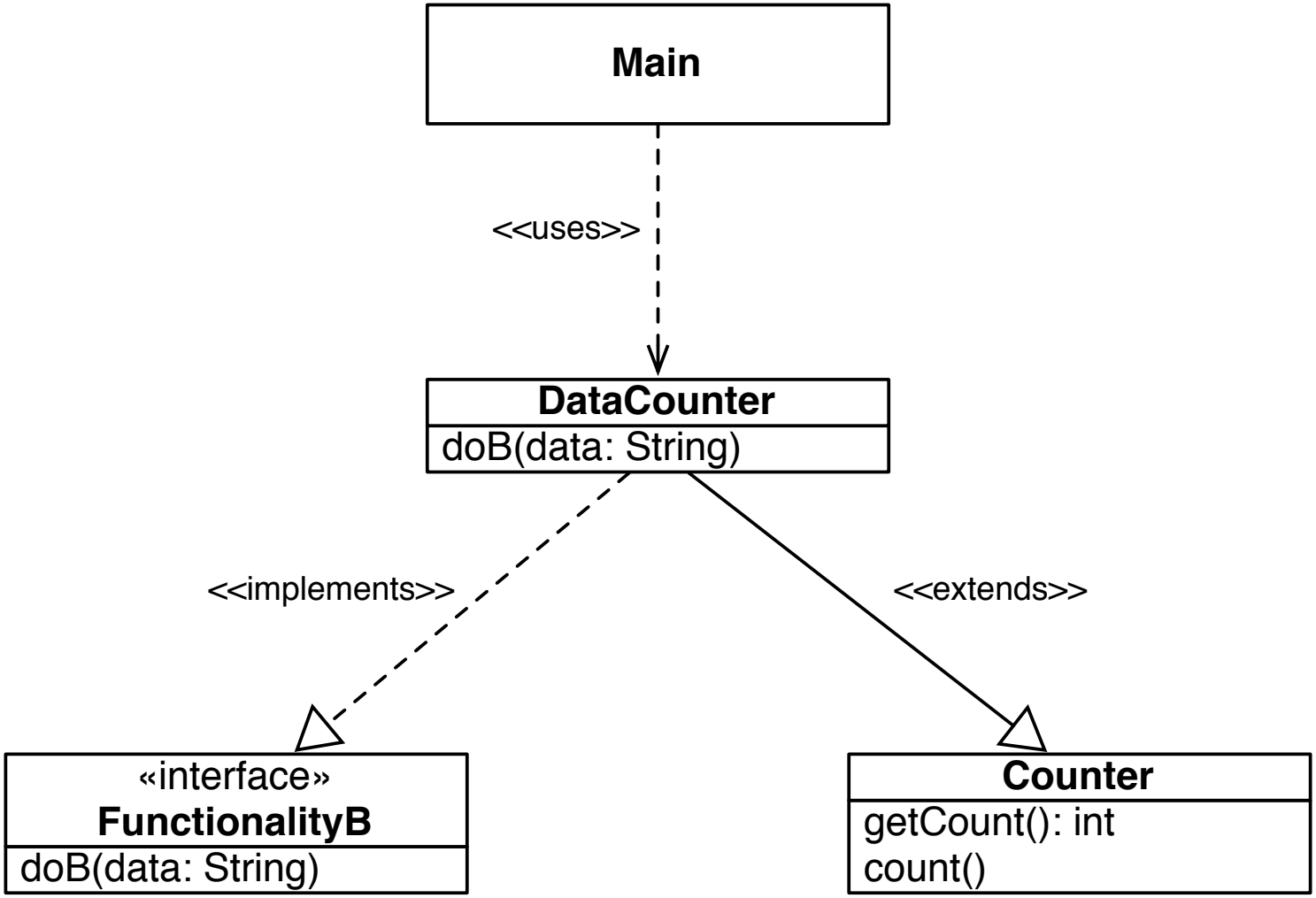
```
1  /**
2   * Wraps a {@code Counter} to obtain a {@code FunctionalityB},
3   * to count how many data items were processed.
4   */
5  class DataCounter extends Counter implements FunctionalityB {
6
7      /**
8       * Counts a data item.  The content of the item is ignored.
9       *
10      * @param data  data item to count
11      */
12     @Override
13     public void doB(String data) {
14         count();
15     }
16
17 }
```

Using a Class Wrapped via Inheritance

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

`DataCounter` offers two interfaces: `FunctionalityB` and `Counter`

Wrapping via Inheritance: Class Diagram



Wrapping via Inheritance: Disadvantage

- Adapted class `DataCounter` has hard coupling to implementation from `Counter`
- It cannot be used to adapt `SpecialCounter` which **`extends`** `Counter`

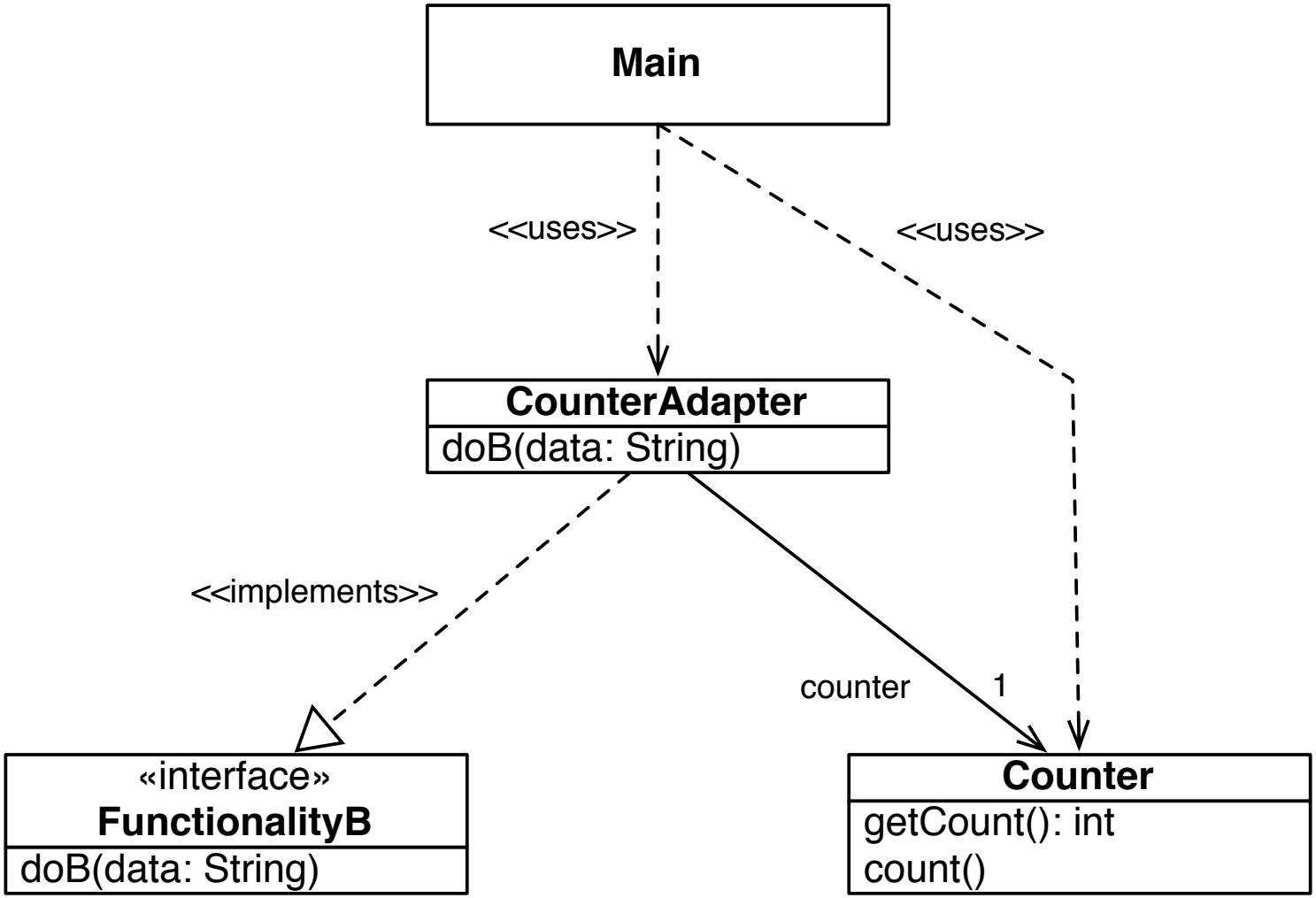
  You would need to define `SpecialDataCounter` for that

  This would lead to duplicated code

Adapting a Class to Provide a Callback, via Composition

```
1  class CounterAdapter implements FunctionalityB {
2      /** Counter being adapted. */
3      private final Counter counter;
4      /**
5       * Constructs a new adapter for a given counter
6       */
7      public CounterAdapter(Counter counter) {
8          this.counter = counter;
9      }
10
11     /**
12      * Counts a data item.  The content of the item is ignored.
13      */
14     @Override
15     public void doB(String data) {
16         counter.count();
17     }
18 }
```

Adapting via Composition: Class Diagram



Using a Class Adapted via Composition

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

Note the extra overhead compared to solution using inheritance:

- Extra indirection from `doB()` to `count()`, via `counter` object
- Client code has to create *two* objects and pass one to the other
- This buys you more independence and flexibility:

  Client can adapt a `Counter` *and any subclass of it*

Inheritance versus Composition

- In general: prefer *composition* over *inheritance*

  But beware of the price

- See slide 42 for details

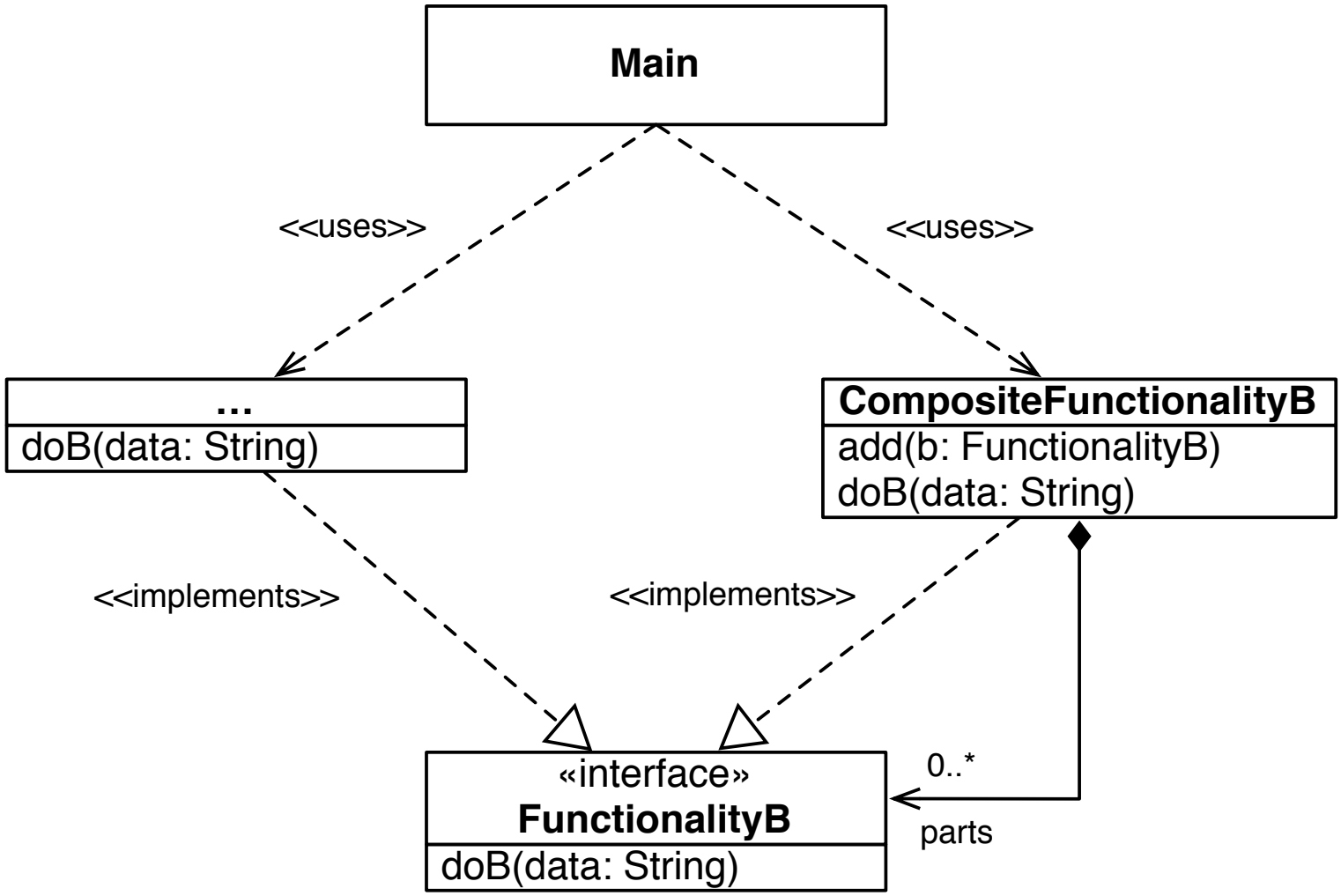
Distributing a Callback to Multiple Callback Handlers

?

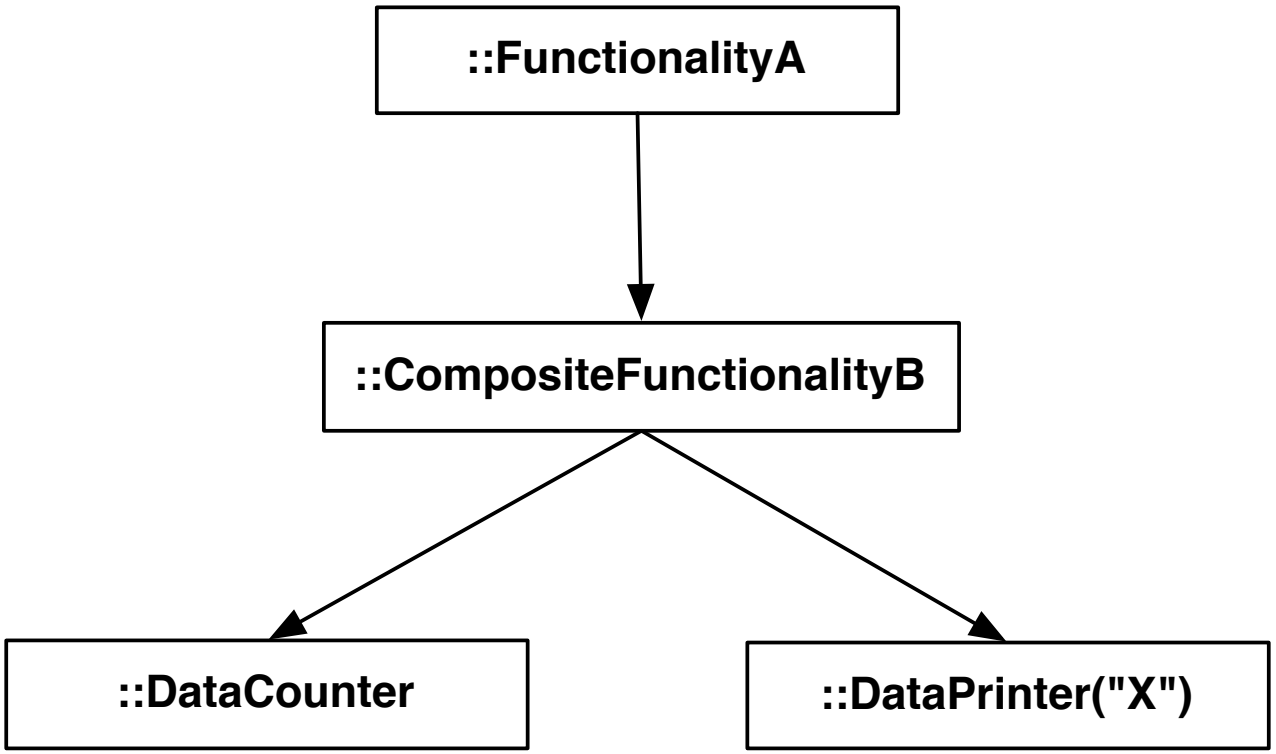
Distributing a Callback to Multiple Callback Handlers

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

Distributing a Callback = Application of Composite Pattern



Distribution of a Callback to Printer and Data Counter



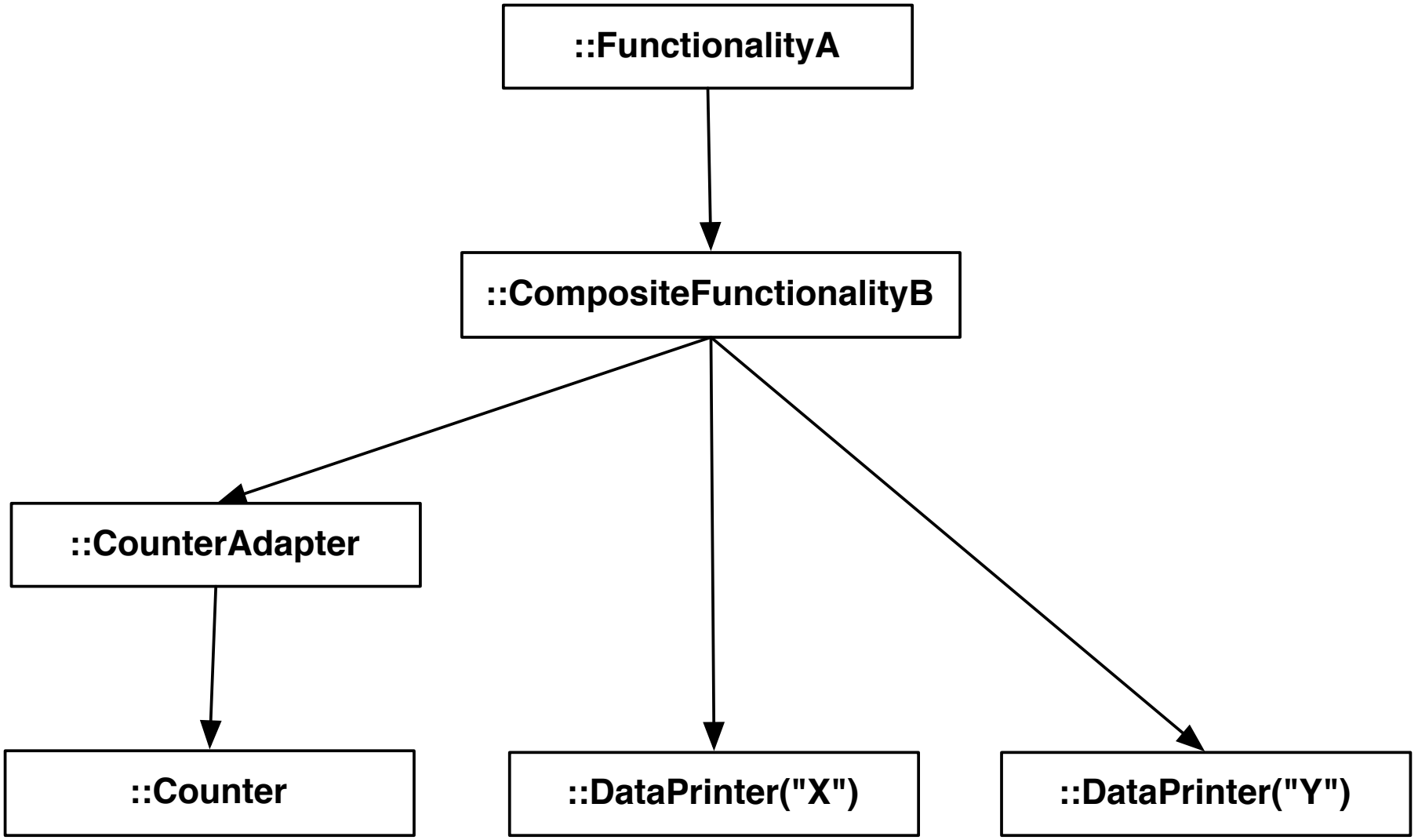
Using a Callback Distributor (Observer Pattern)

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

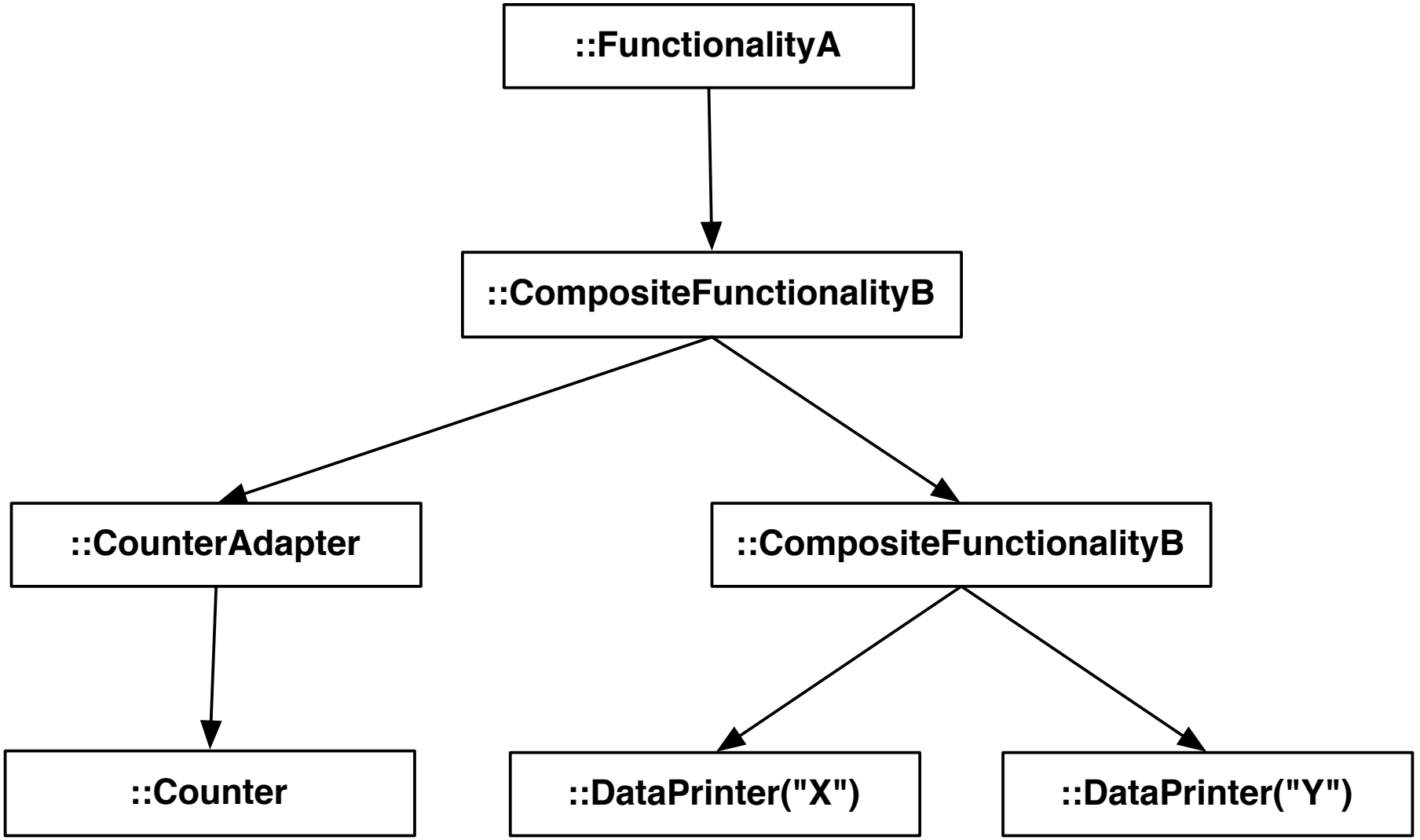
`FunctionalityA` + `CompositeFunctionalityB` = Observable Subject

Cf. Observer pattern: supports arbitrary-depth hierarchies

Distribution of a Callback, Including an Adapter



Hierarchical Distribution of a Callback



Modifying a Callback's Behavior

?

Modifying a Callback's Behavior

```
1  /**
2   * Print data passed, surrounded by quotes (using inheritance).
3   */
4  public class QuotedDataPrinter extends DataPrinter {
5
6      /**
7       * Prints data, surrounded by quotes.
8       */
9      @Override
10     public void doB(String data) {
11         super.doB('"' + data + '"');
12     }
13
14 }
```

Modifies behavior of `DataPrinter`; cannot be used generally

Modifying a Callback's Behavior Generally

?

Modifying a Callback's Behavior Generally

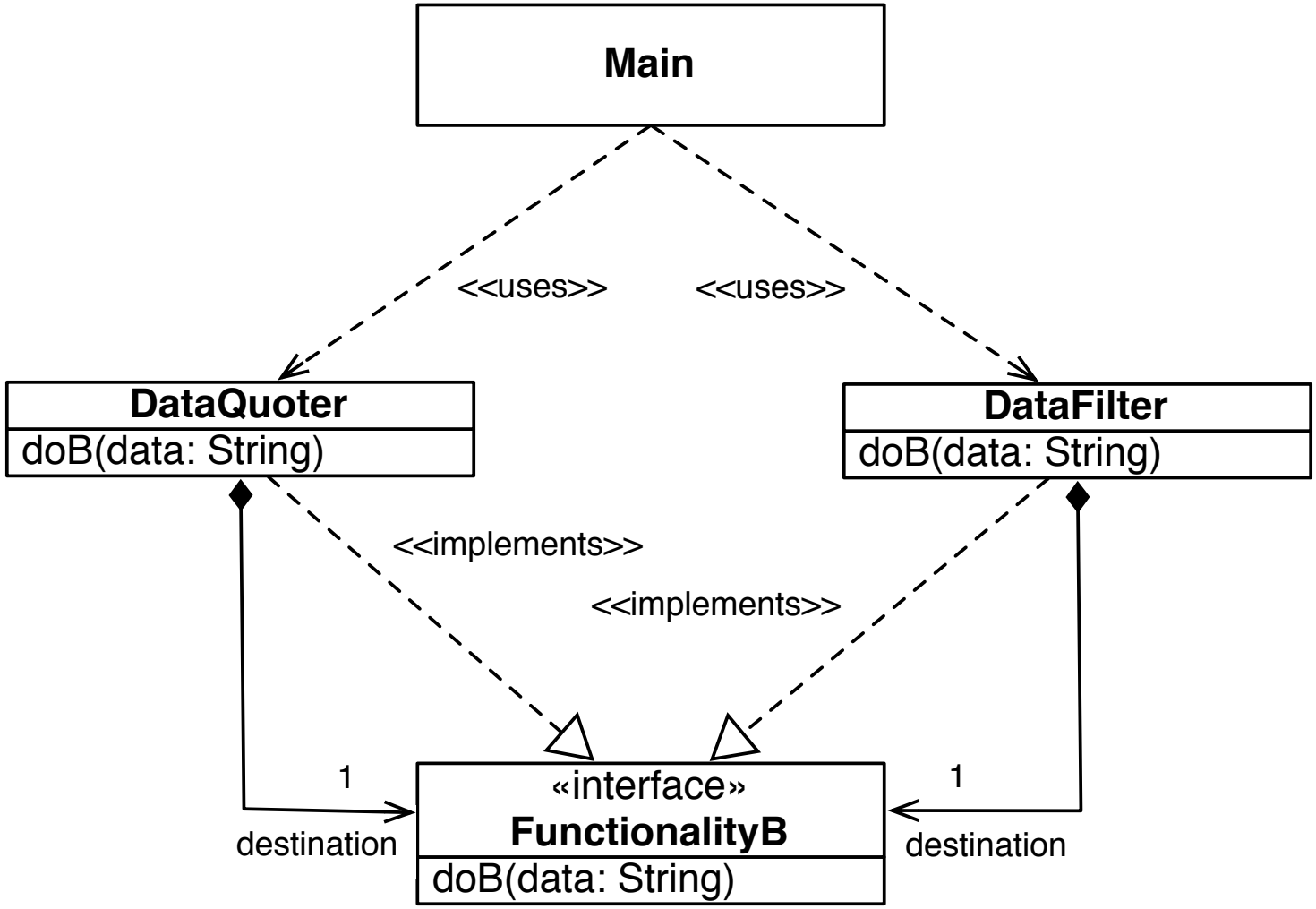
```
1  /**
2   * Put quotes around data and pass on to another {@code FunctionalityB}.
3   */
4  class DataQuoter implements FunctionalityB {
5      private FunctionalityB destination;
6      public DataQuoter(FunctionalityB destination) {
7          this.destination = destination;
8      }
9      public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

Use composition, and provide destination as parameter

Filtering a Callback

```
1  /**
2   * Filter to selectively pass data to another {@code FunctionalityB}.
3   */
4  class DataFilter implements FunctionalityB {
5      private FunctionalityB destination;
6      private String pattern;
7      public DataFilter(FunctionalityB 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



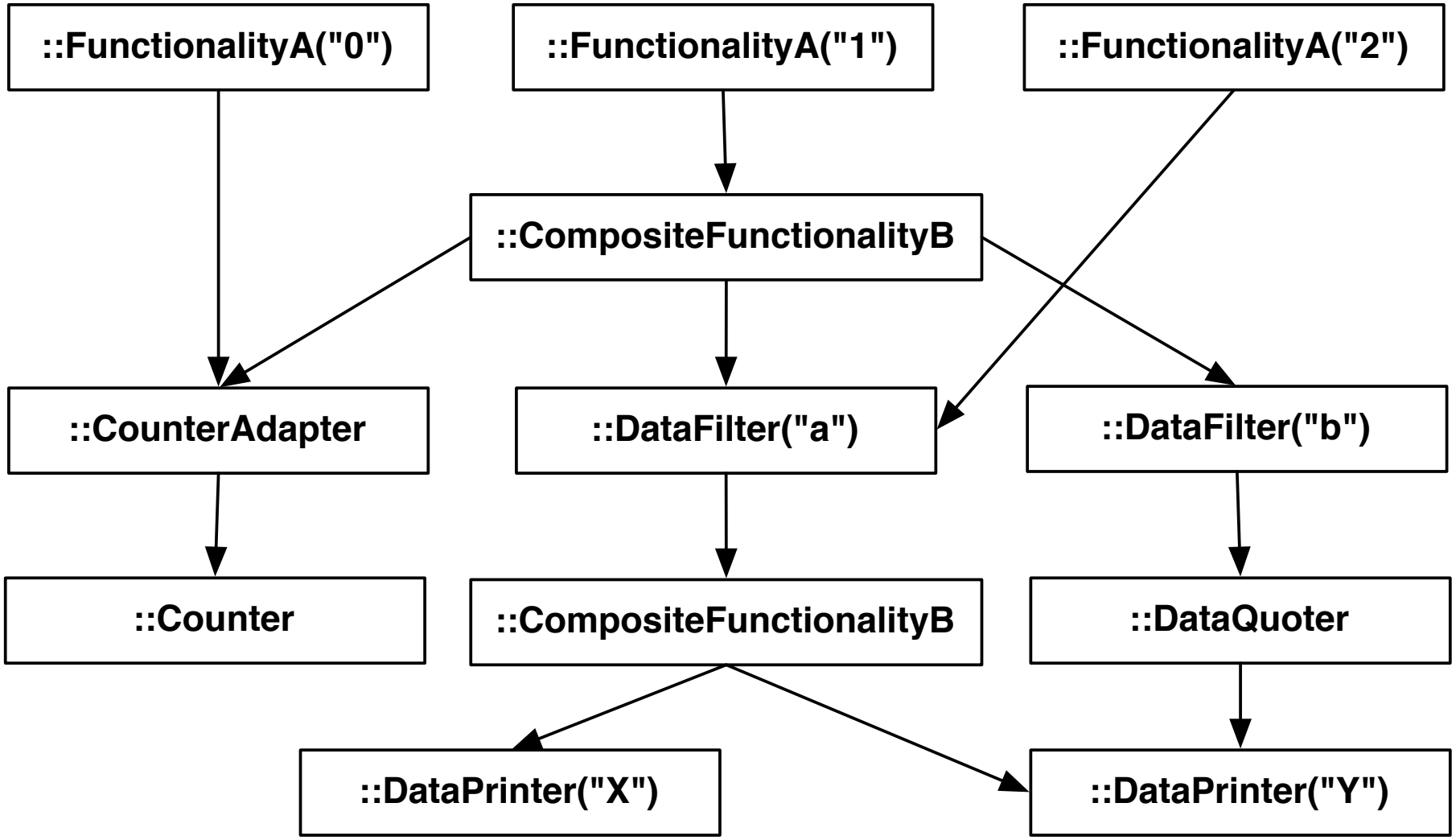
Combining Decorators

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

Prints quoted versions that contain a `1`

Combined decorators are impossible when decorating via inheritance

Flexible Callback Toolkit: `CallbackExamples/`



Flexible Callback Toolkit: Evaluation

Advantages:

- Small number of general 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 course: 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

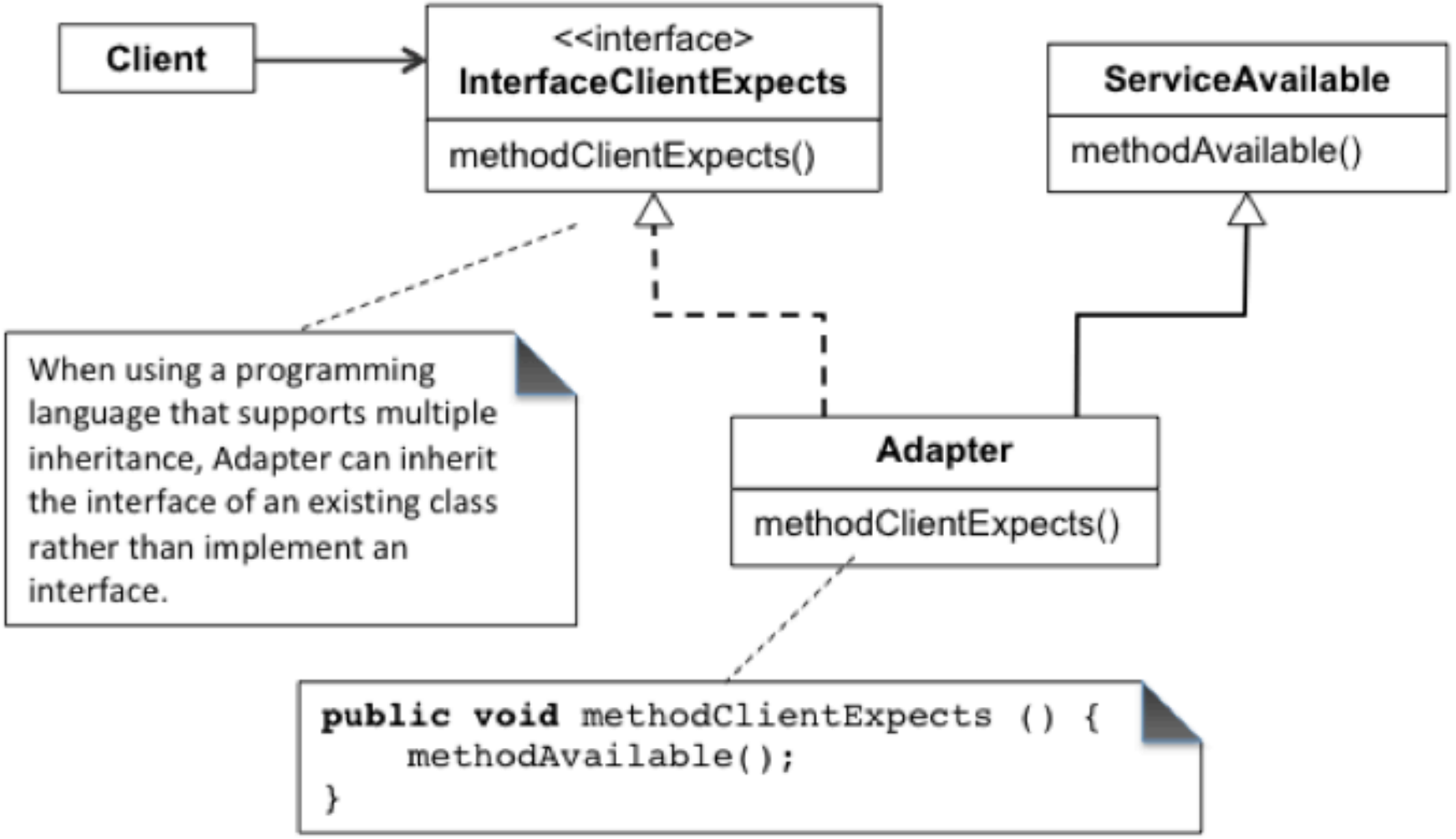
Why not modify existing class? Because

- 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?

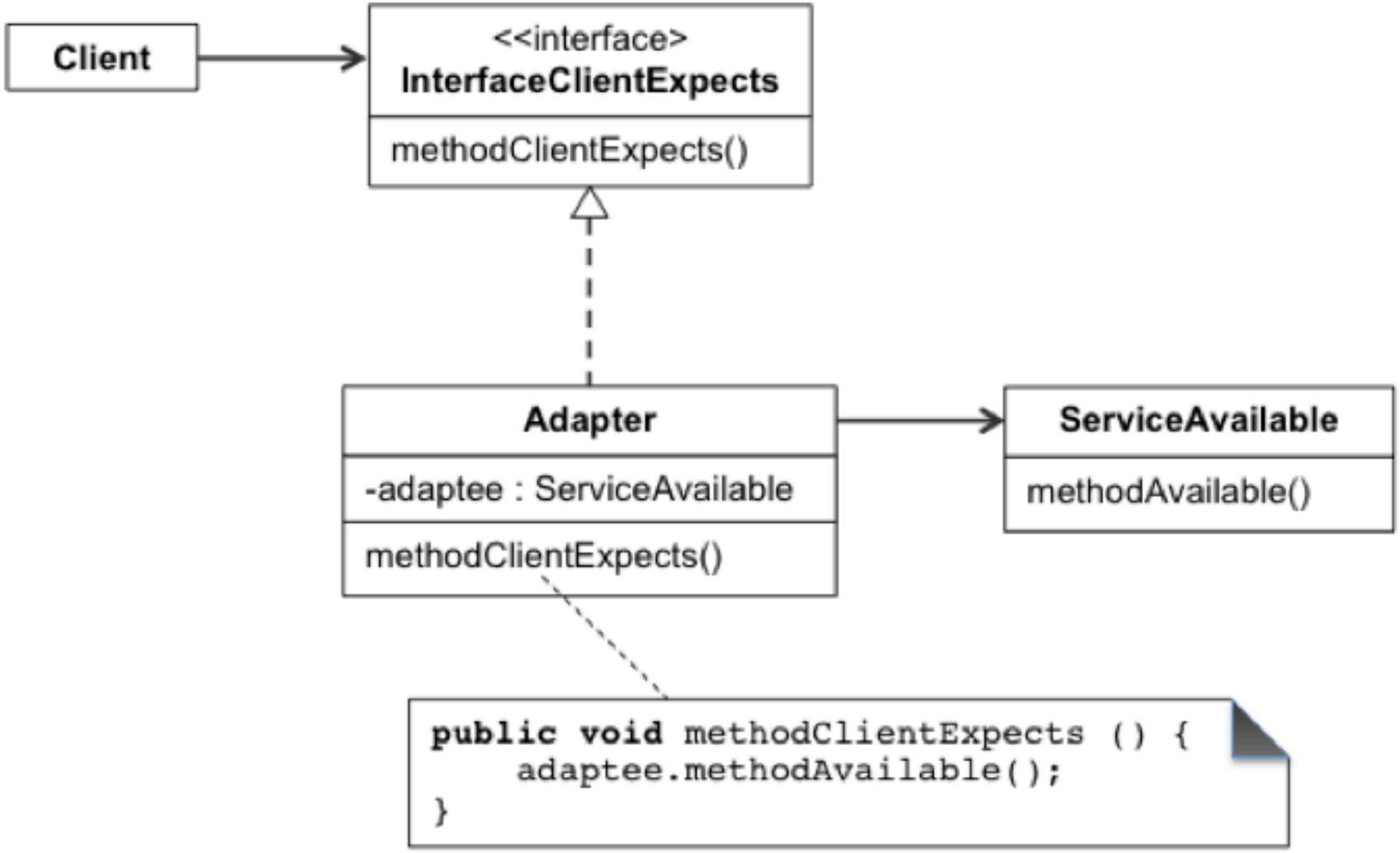
Adapter Pattern using Inheritance (from Burris)



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 underlying implementation of adapted class
  - exposes inherited **`protected`** internals
  - yields class with larger interface
    Violates *Interface Segregation Principle* (ISP)
- Composition offers looser coupling: couples to an *object*

  Can vary the underlying implementation of adapted class.

  Cost: indirection via stored reference to adapted (extra) object

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

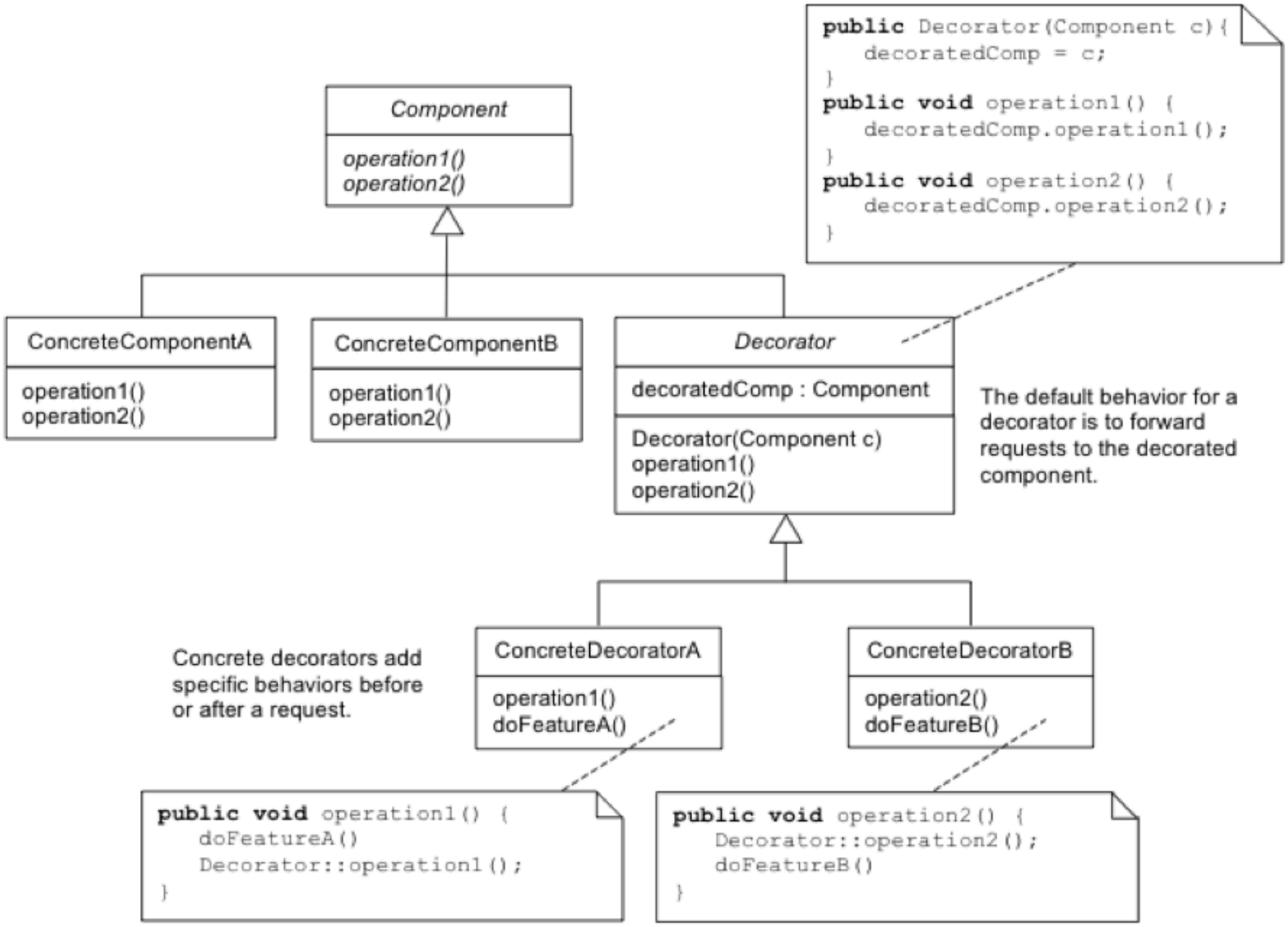
  Example: quoting, trimming, reversing, capitalizing a string

- Decorators obtained by composition can be mixed and matched

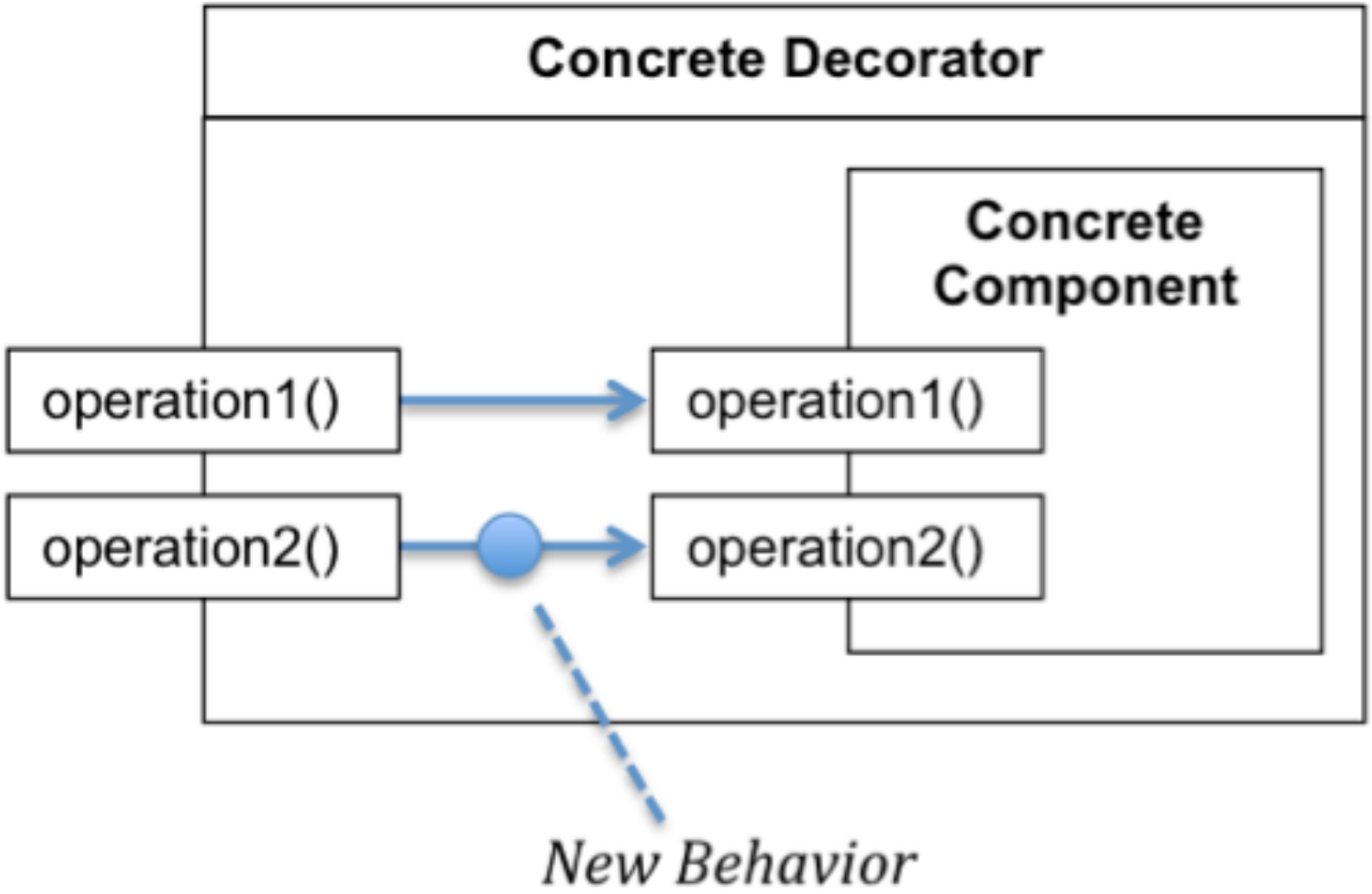
  Small set of decorators can be combined at will

  Similar to functional composition

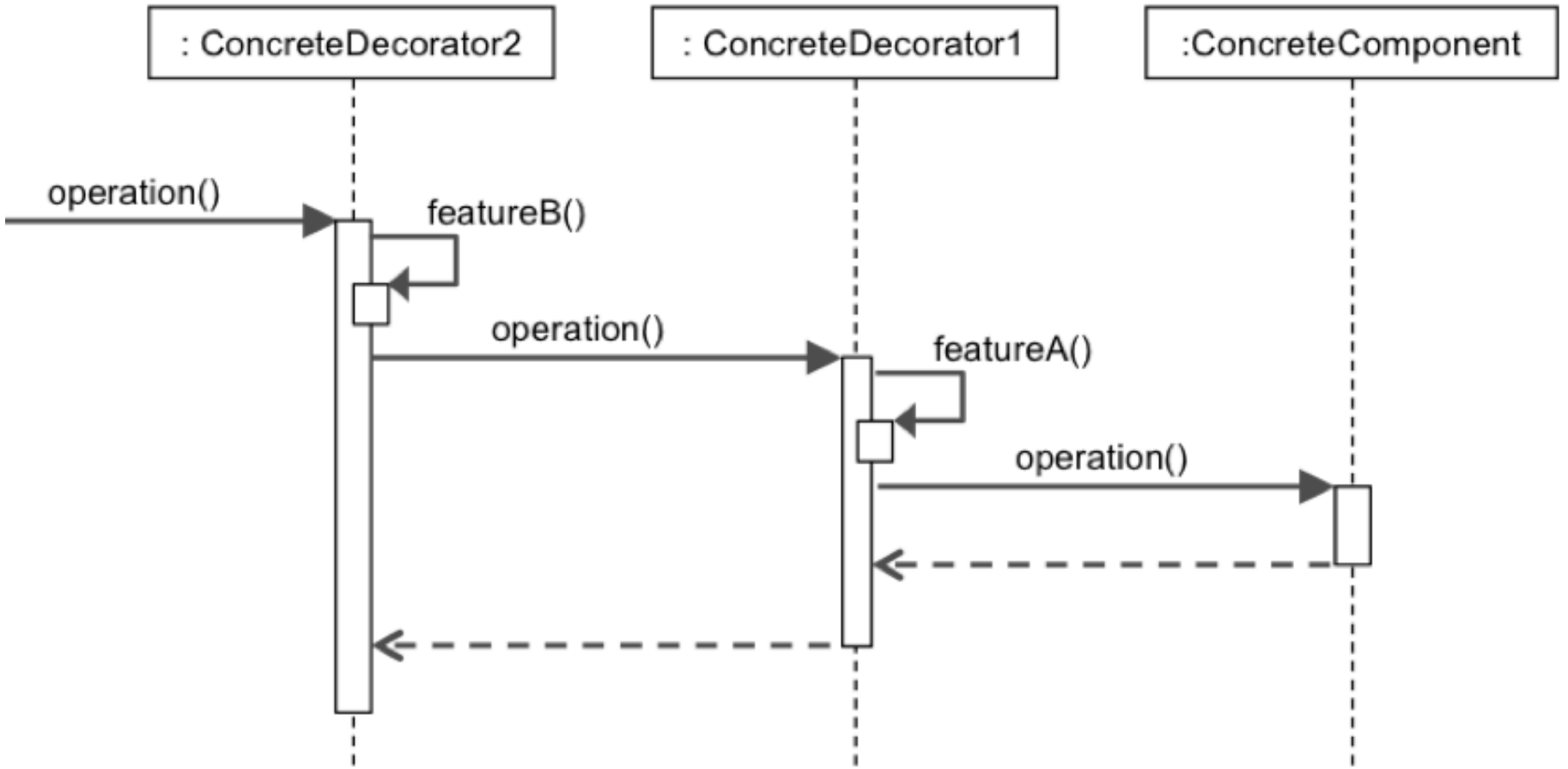
Decorator Pattern using Composition (from Burris)



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 composition	composition

Summary

- Dependency Inversion Principle (DIP), Dependency Injection (DI)
- 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