Summer Semester 2015

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Decorator Pattern

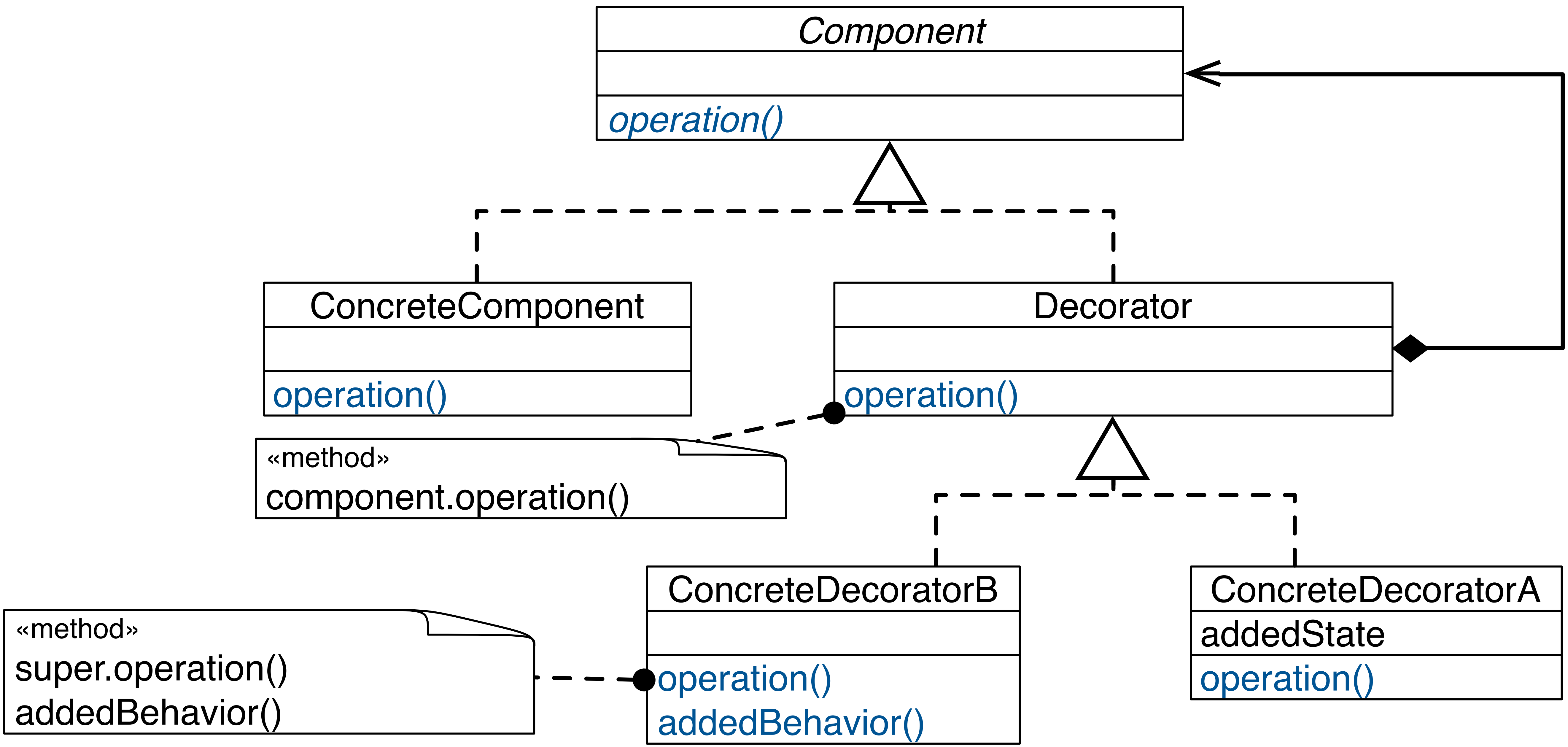
Intent of the Decorator Pattern

We need to **add functionality to existing objects!**

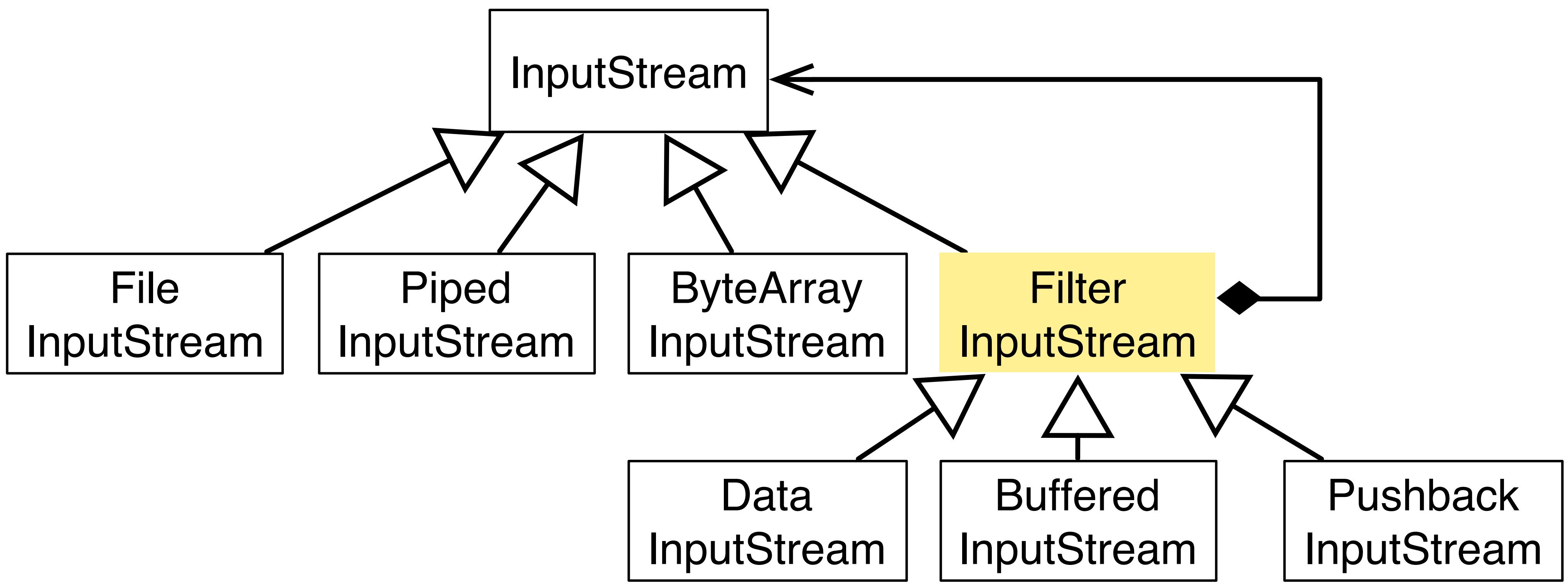
- dynamically, i.e., during runtime after the object is created,

- without having to implement conditional logic to use the new functionality.

The Structure of a Decorator-Based Design



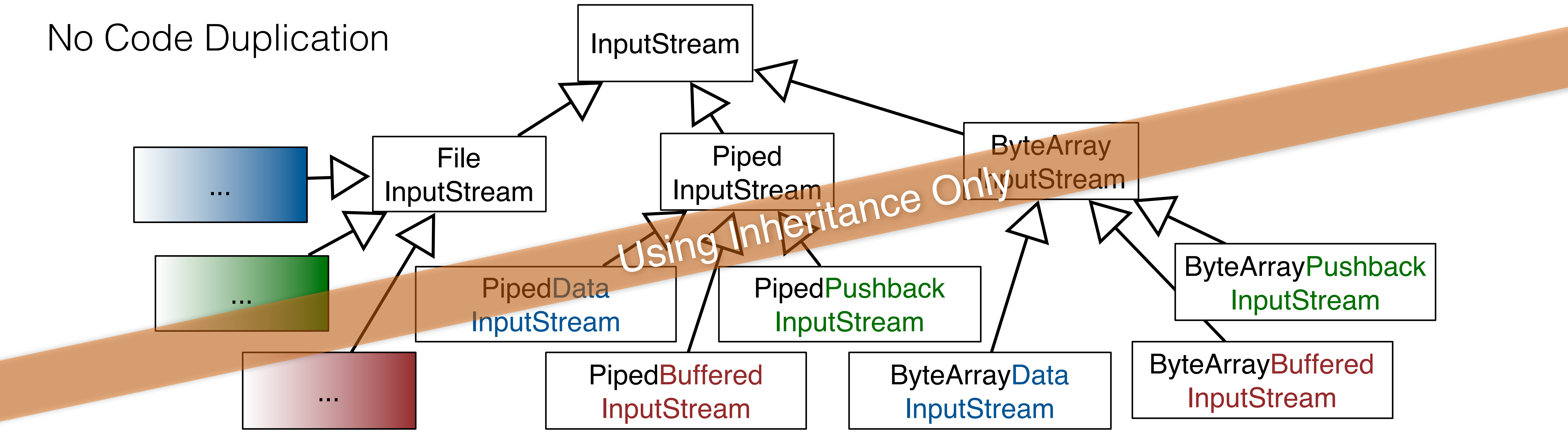
The Decorator Pattern - *by Example*



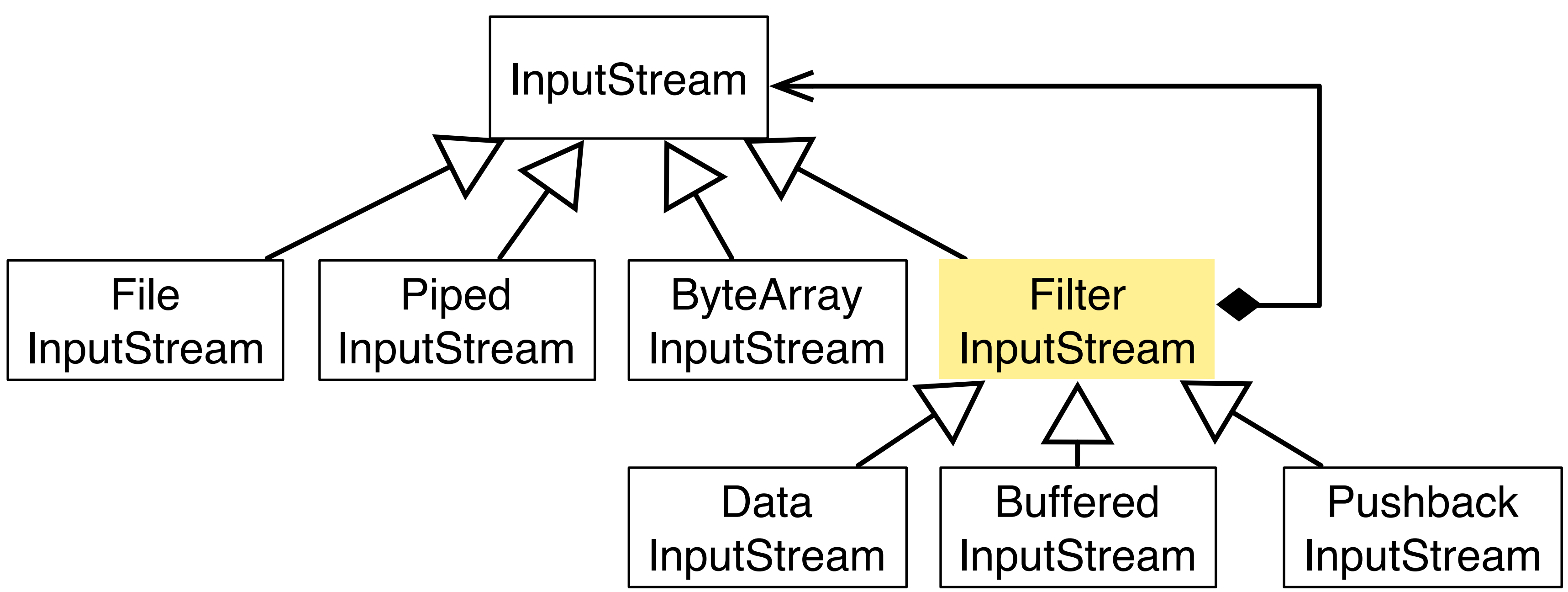

```
DataInputStream dis = new DataInputStream(new FileInputStream(file));

dis.readUnsignedByte();
```

Each Variation Defined Once

No Code Duplication



Using the Decorator Pattern



Improved Flexibility

- Decorative functionality can be added / removed at run-time.
- Combining different decorator classes for a component class enables to mix and match responsibilities as needed.

```
is = new FileInputStream(file);
is.read(…);
…
DataInputStream dis = new DataInputStream(is);
dis.readUnsignedByte();
…
(new BufferedInputStream(dis)).readLine(…);
```

- Easy to add functionality twice.
  E.g., given a class `BorderDecorator` for a `TextField`, to add a double border, attach two instances of `BorderDecorator`.

Decorator Avoids Incoherent Classes

- No need for feature-bloated classes positioned high up in the inheritance hierarchy to avoid code duplication.
- Pay-as-you-go approach: Do not bloat, but extend using fine-grained Decorator classes.
    - Functionality can be composed from simple pieces.
    - A client does not need to pay for features it does not use.

Advantages of Decorator-Based Designs

A fine-grained Decorator hierarchy is easy to extend.

Decorator helps to design software that better supports OCP.

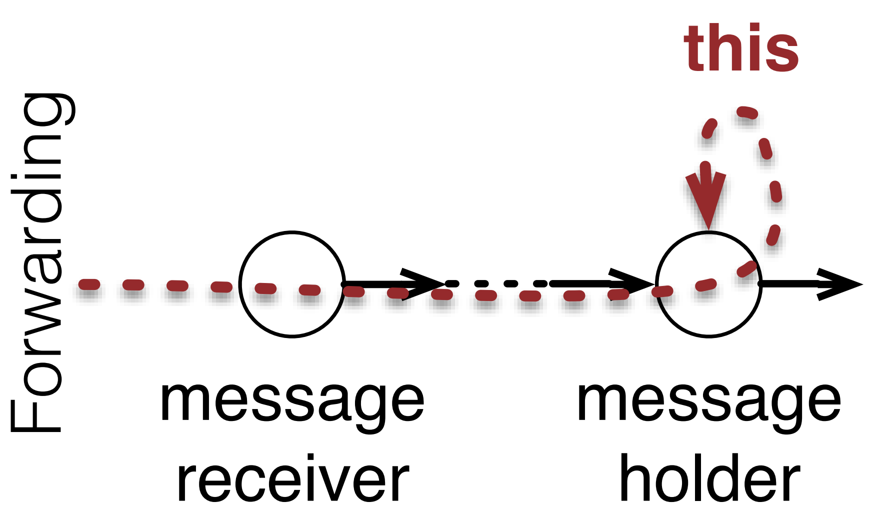
Consequences of Decorator-Based Designs

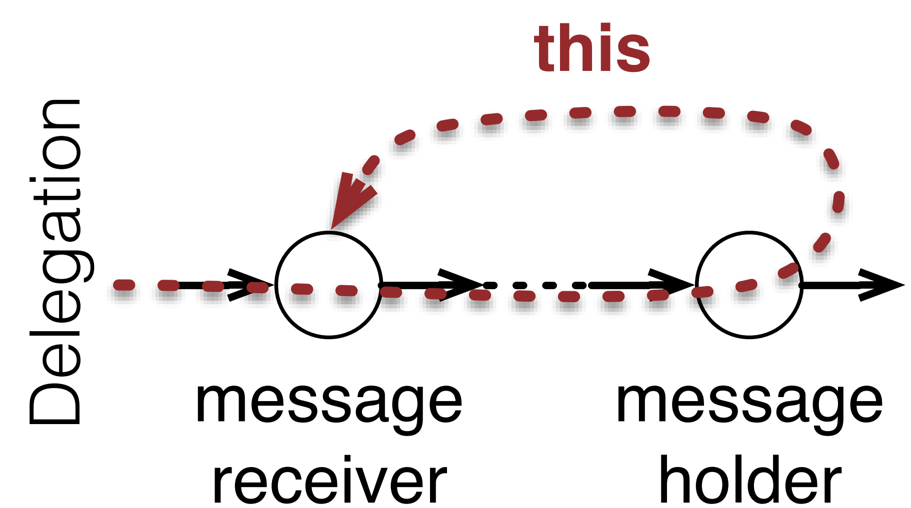
- Lots of Little Objects

- A decorator and its component are not identical (Object identity)

```
FileInputStream fin = new FileInputStream(“a.txt”);
BufferedInputStream din = new BufferedInputStream(fin);
```

- No Late Binding

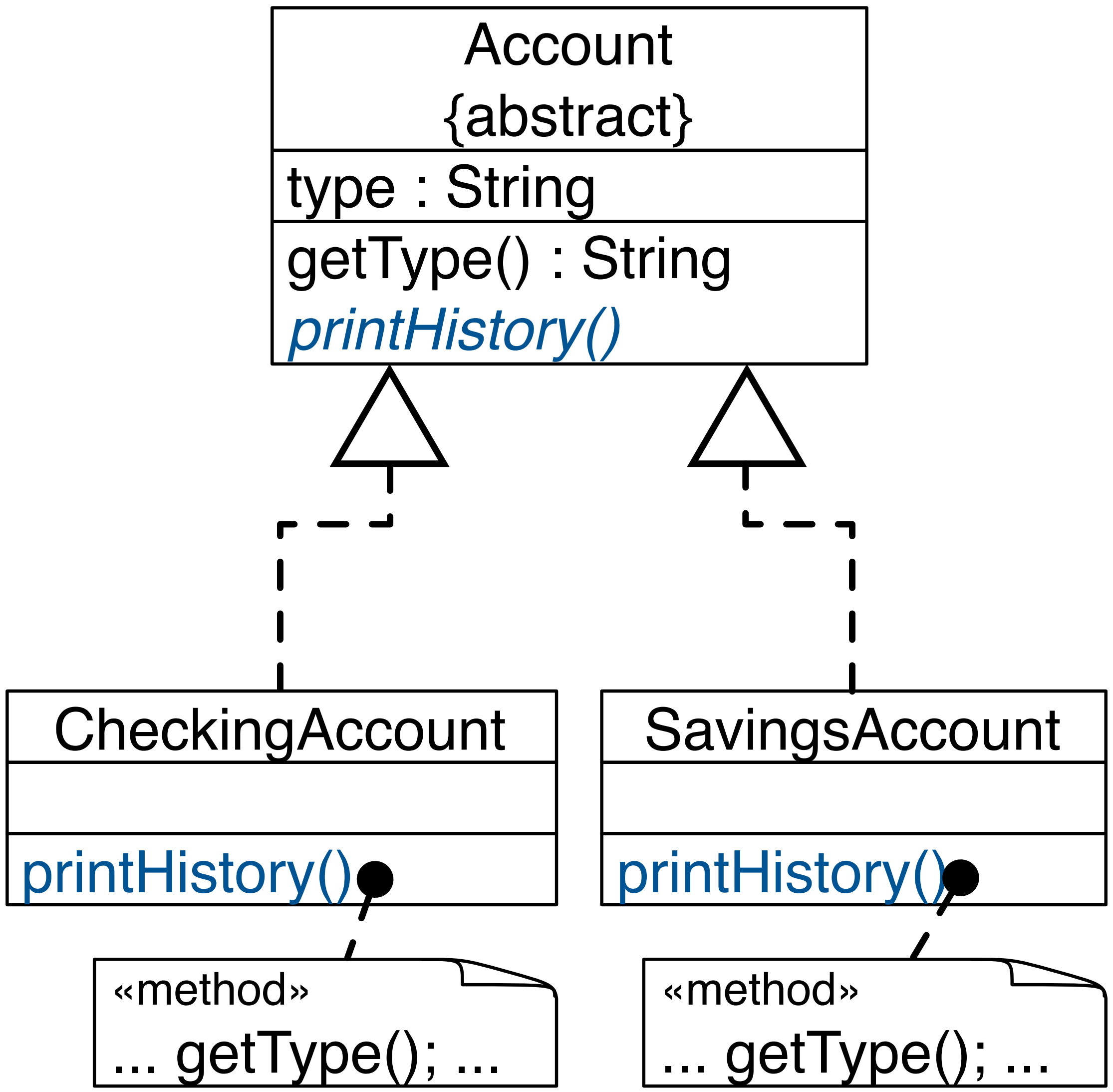




No Late Binding Illustrated

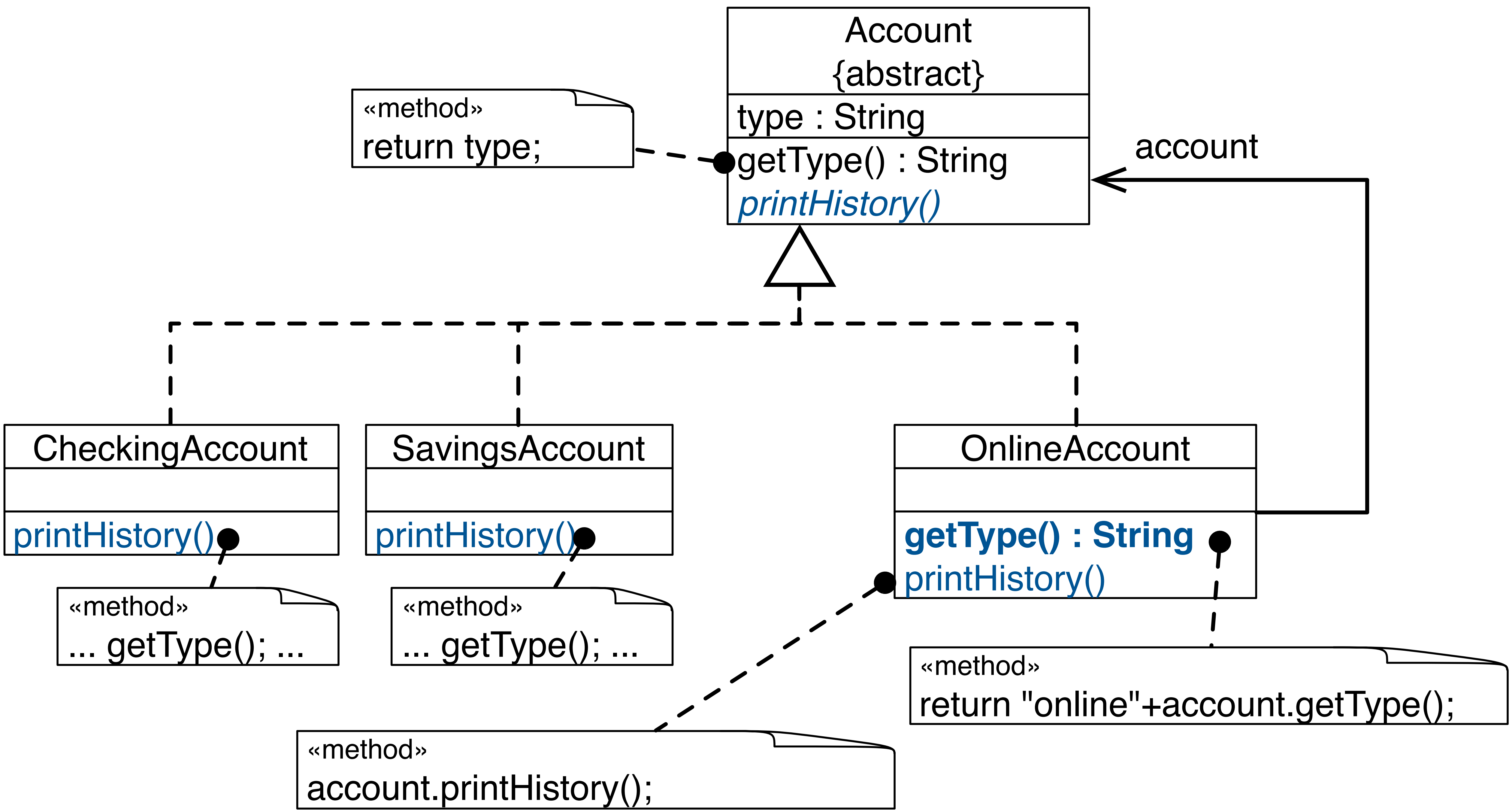
Task:

- Extend the design to enable online access to accounts.
- Decorator seems to be the right choice!
- Among other things, we decorate the description of accounts with the label “online”.
- The way the history is calculated does not need to be decorated, hence, the decorator just forwards.

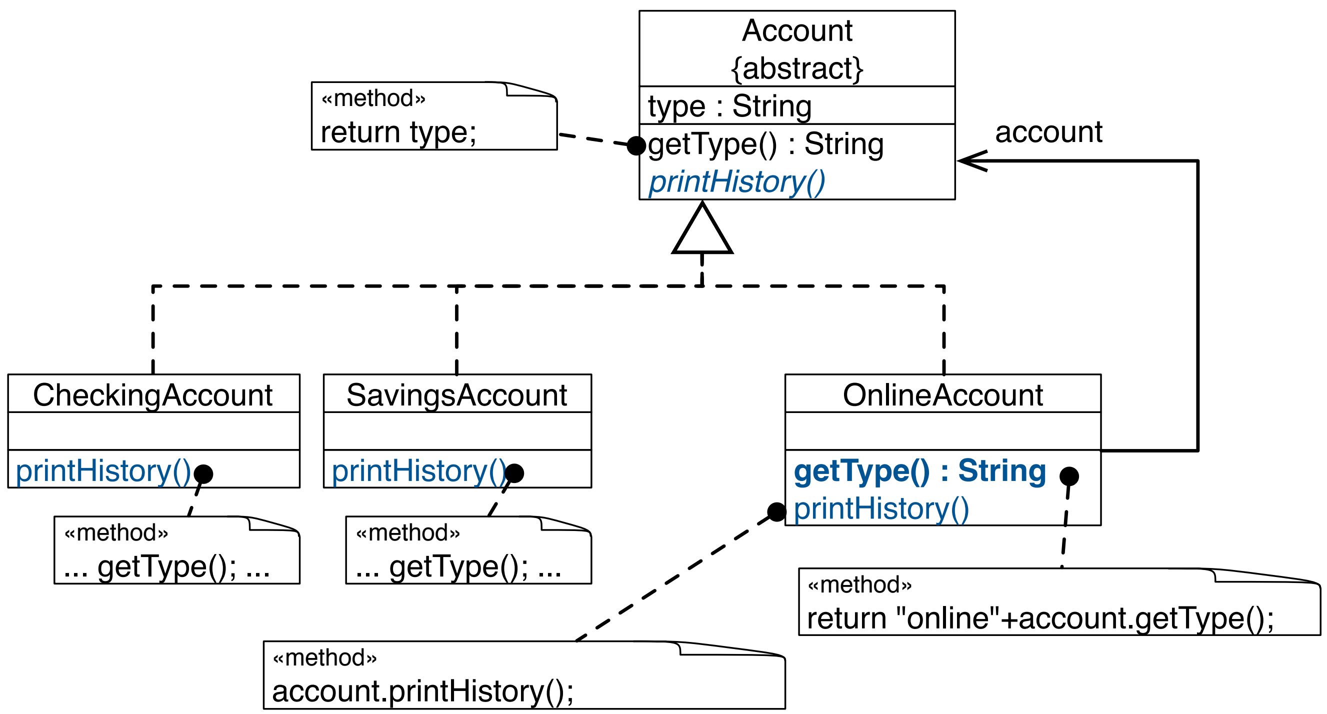


No Late Binding Illustrated

Do you see where we hit the "no-late binding" problem?



No Late Binding Illustrated



- Does the call to `printHistory` on `onlineAcc` behave as expected?

```
…
Account checkingAcc =
   new CheckingAccount(…);


…
Account onlineAcc =
   new OnlineAccount(
      checkingAccount);


…
onlineAcc.printHistory();
…
```

Implementation Issues

- Keep the common class (Component) lightweight!
- A decorator's interface must conform to the interface of the component it decorates.
- There is no need to define an abstract Decorator class when you only need to add one responsibility.

Decorator and the Fragile Base-Class Problem

Does the use of the Decorator Pattern solve the fragile base-class problem?

The InstrumentedHashSet again...

```
public class InstrumentedHashSet<E> extends java.util.HashSet<E> {
  private int addCount = 0;
  …
  @Override public boolean add(E e) {
    addCount++; return super.add(e);
  }
  @Override public boolean addAll(java.util.Collection<? extends E> c) {
    addCount += c.size(); return super.addAll(c);
  }
  public int getAddCount() { return addCount; }
}

public static void main(String[] args) {
  InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
  s.addAll(Arrays.asList("aaa", "bbb", "ccc"));
  System.out.println(s.getAddCount());
}
```

Ask yourself (again): What is printed on the screen?

A Decorator-Based `InstrumentedSet`

1. Declare an interface `Set<E>`
2. Let `HashSet<E>` implement `Set<E>`
3. Define `ForwardingSet<E>` as an implementation of `Set<E>`
4. `ForwardingSet<E>` (our root Decorator)
    1. Has a field `s` of type `Set<E>`
    2. Implements methods in `Set<E>` by forwarding them to `s`
5. `InstrumentedSet<E>` (a concrete Decorator) extends `ForwardingSet<E>` and overrides methods `add` and `addAll`

A ForwardingSet<E>

```
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 InstrumentedSet

```
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());
}
```

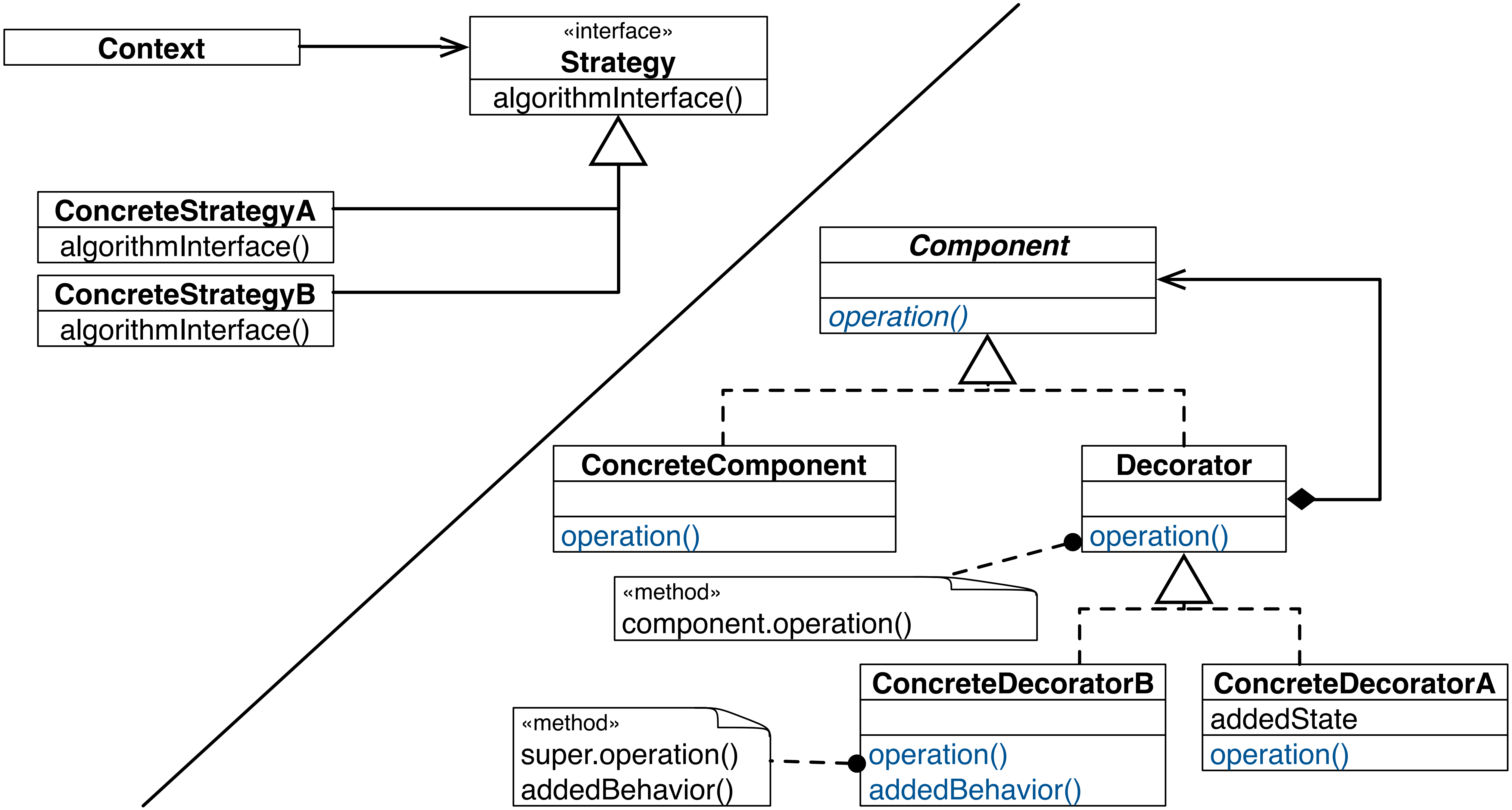
What is printed on the screen?

Decorator and the Fragile Base-Class Problem

Does the use of the Decorator Pattern **really** solve the fragile base-class problem?

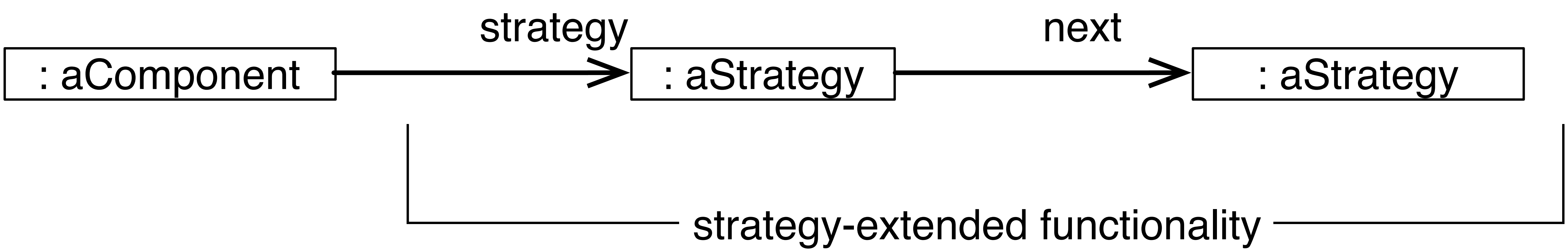
Decorator and Strategy

Decorator and Strategy share the goal of **supporting dynamic behavior adaptation.**



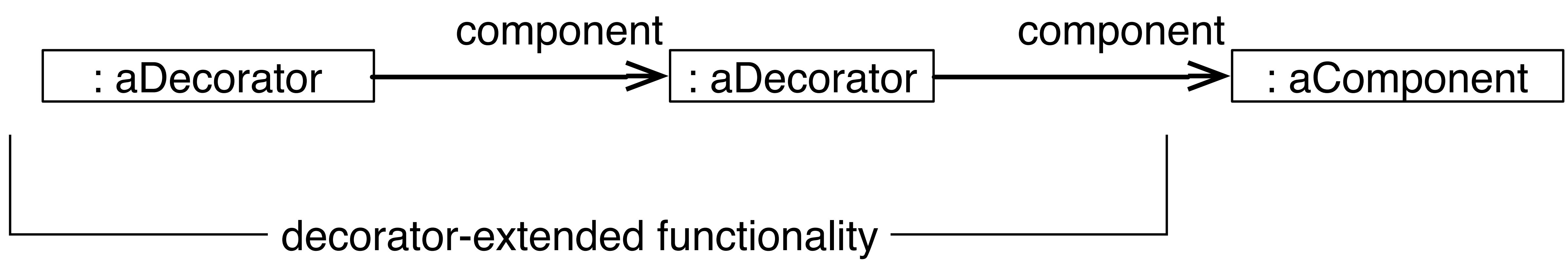
Simulate the Effect of Each Other

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

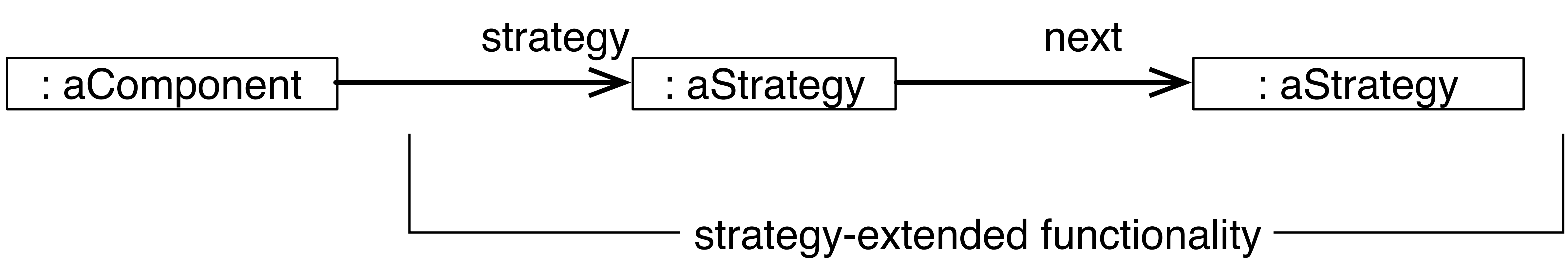


Transparent vs. Non-Transparent Change

Decorator changes a component from the outside:
The component does not know about its decorators.



Strategy changes a component from the inside:
Component knows about Strategy-based extensions.



Takeaway Decorator vs. Strategy

- Like the Strategy, the Decorator pattern also uses a combination of object composition and inheritance/subtype polymorphism to support dynamic and reusable variations.
- Unlike the Strategy, it adapts object behavior from the outside rather than inside.
- Unlike Strategy, variations encapsulated in decorator objects do not leave any footprint in the behavior of the objects being adapted.
- In that sense, it has a stronger “inheritance” resemblance than Strategy.

Takeaway

Decorator may lead to error-prone and hard to understand designs.

- Many little objects emulate the behavior of a conceptually single object.
- No object identity.
- No late-binding.
- Not appropriate for modeling the variability of heavy-weight objects with a lot of functionality.
- Might not be applicable to third-party library objects.
- It does not really solve the fragile base-class problem.

A "Static" Decorator

Using mixins we can statically decorate classes (class composition vs. object composition)
and also get delegation semantics.

```scala
trait Component {
  def state : String
  def name: String
}
case class AComponent (id : String) extends Component {
    def state = name+":"+id
    def name = "A"
}
trait ComponentDecoratorA extends Component {
    abstract override def name = "ByADecorated:"+super.name
}
trait ComponentDecoratorB extends Component {
    abstract override def name = "ByBDecorated:"+super.name
}

object DemoStructuralDecorator extends App {
  val c = new AComponent("42")   // static decoration
              with ComponentDecoratorA with ComponentDecor
  println(c.state)
}
```

Output: ByBDecorated:ByADecorated:A:42