Dr. Michael Eichberg
Software Engineering
Department of Computer Science
Technische Universität Darmstadt

Introduction to Software Engineering

# The Decorator Pattern

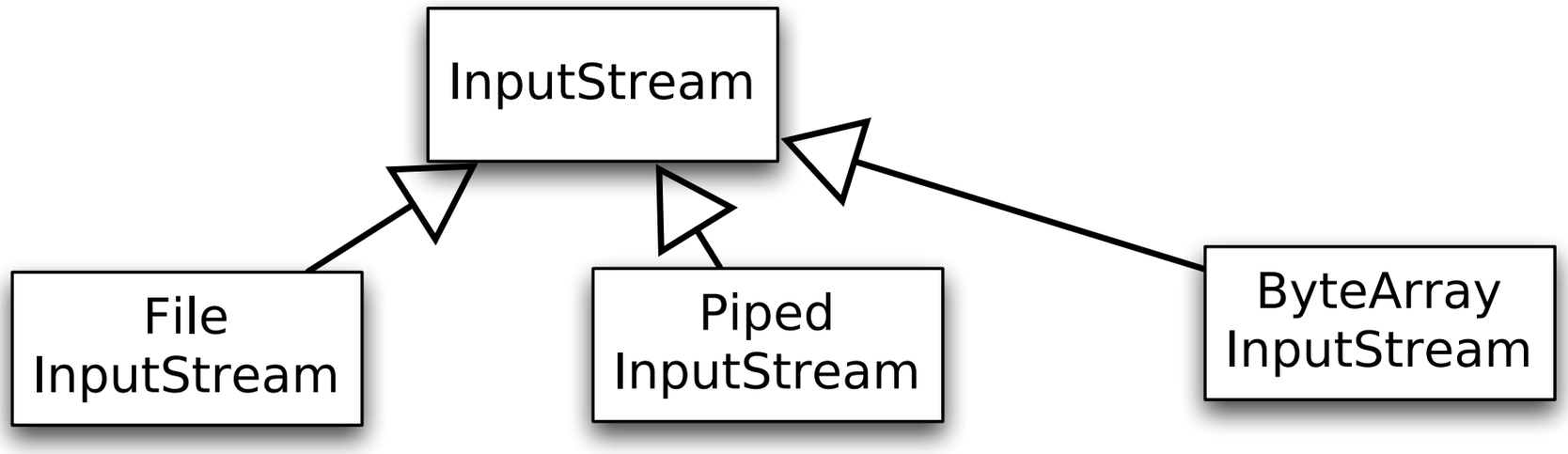
For details see Gamma et al. in “Design Patterns”

TECHNISCHE
UNIVERSITÄT
DARMSTADT

# The Decorator Design Pattern
## Example Scenario



Given:

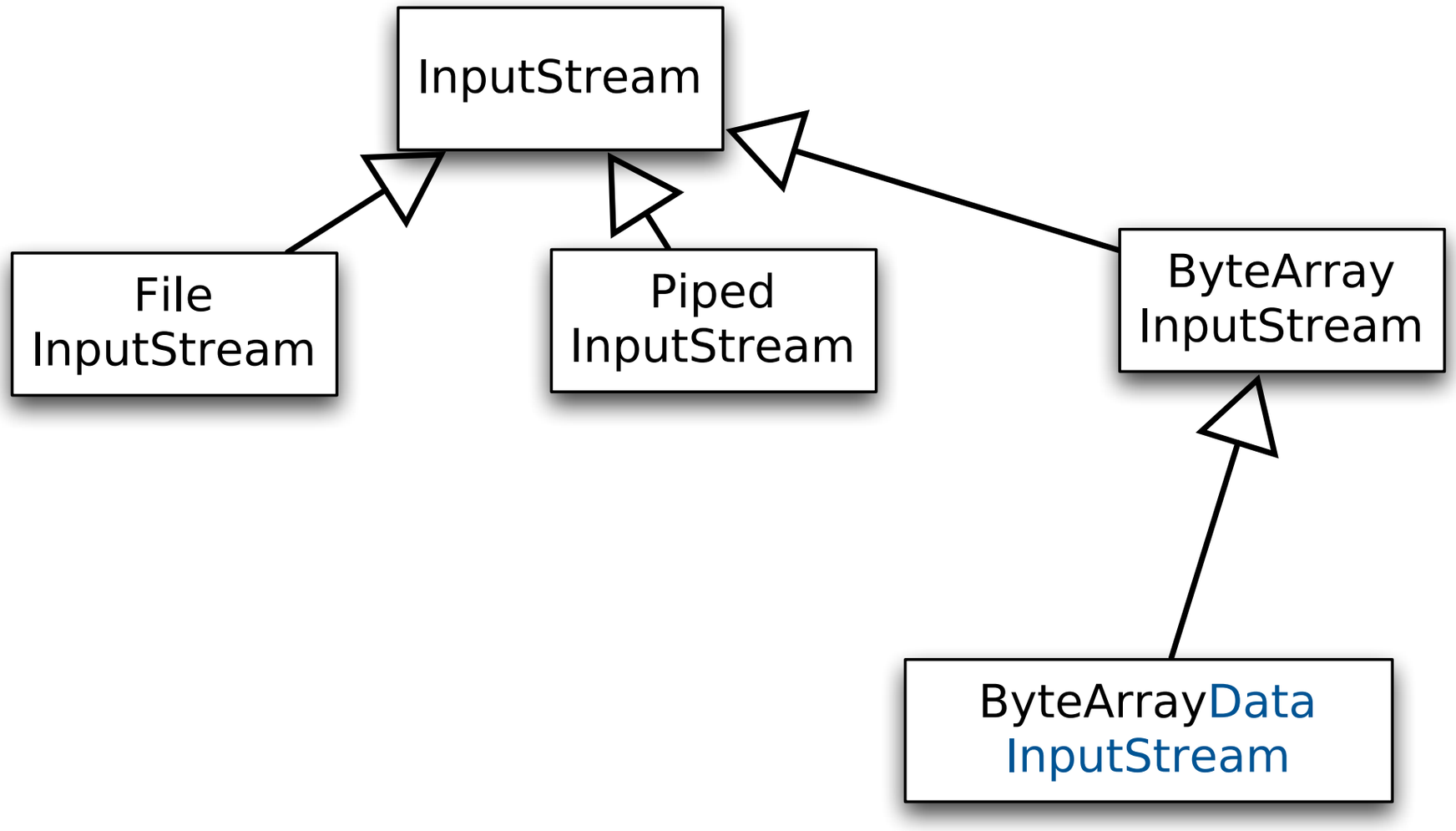



Goal:
Adding functionality to a **ByteArrayInputStream** to read whole sentences and not just single bytes.

# The Decorator Design Pattern
## Example Scenario



Given:

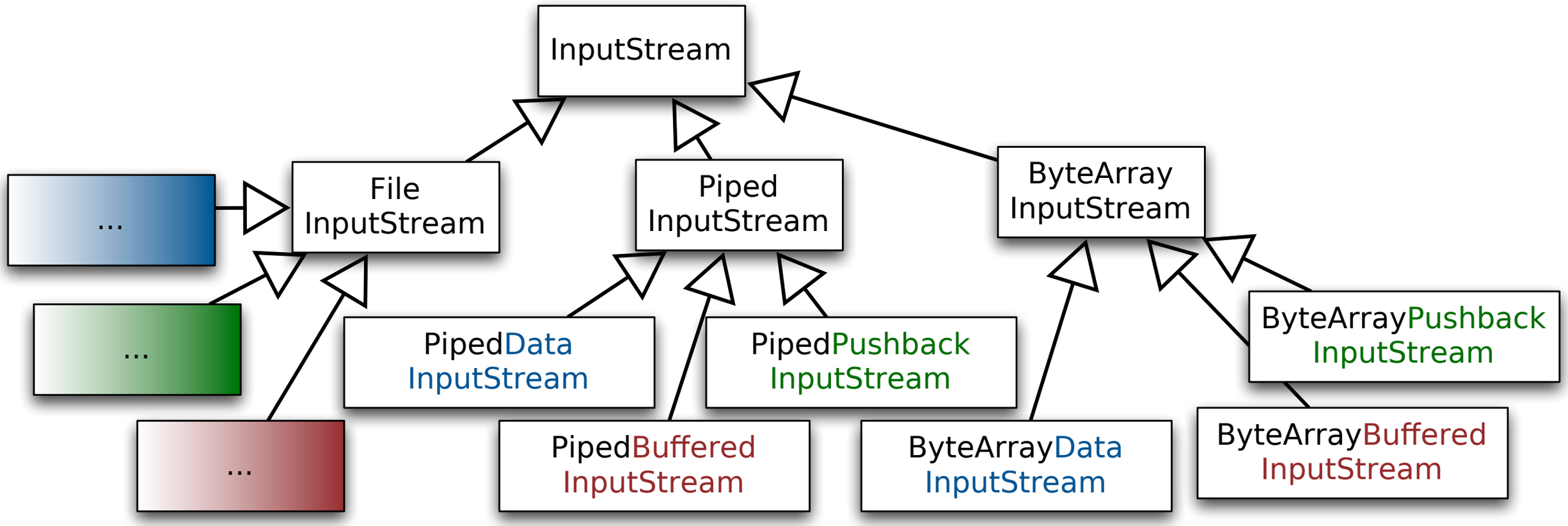



Goal:
We also want to have the possibility to read whole sentences using **FileInputStreams**...

# The Decorator Design Pattern
## Example Scenario



.. after n iterations:



# The Decorator Design Pattern
# Motivation
# Problems with (Single-)Inheritance



- Large number of independent extensions are possible
  - … a new class for each responsibility
    (How about PipedDataBufferedInputStream?)
    - explosion of subclasses to support every combination
- **Non-reusable extensions; code duplication;**
  Maintenance nightmare: exponential growth of number of classes!
- No support for dynamic adaptation
  Responsibility mix fixed statically
- A class definition may be hidden or otherwise unavailable for subclassing

Problems with Inheritance

# Fragile Base Class Problem



Base classes are considered fragile because... you can modify a base class in a seemingly safe way, but this new behavior, when inherited by the derived classes, might cause the derived classes to malfunction.

Problems with Inheritance

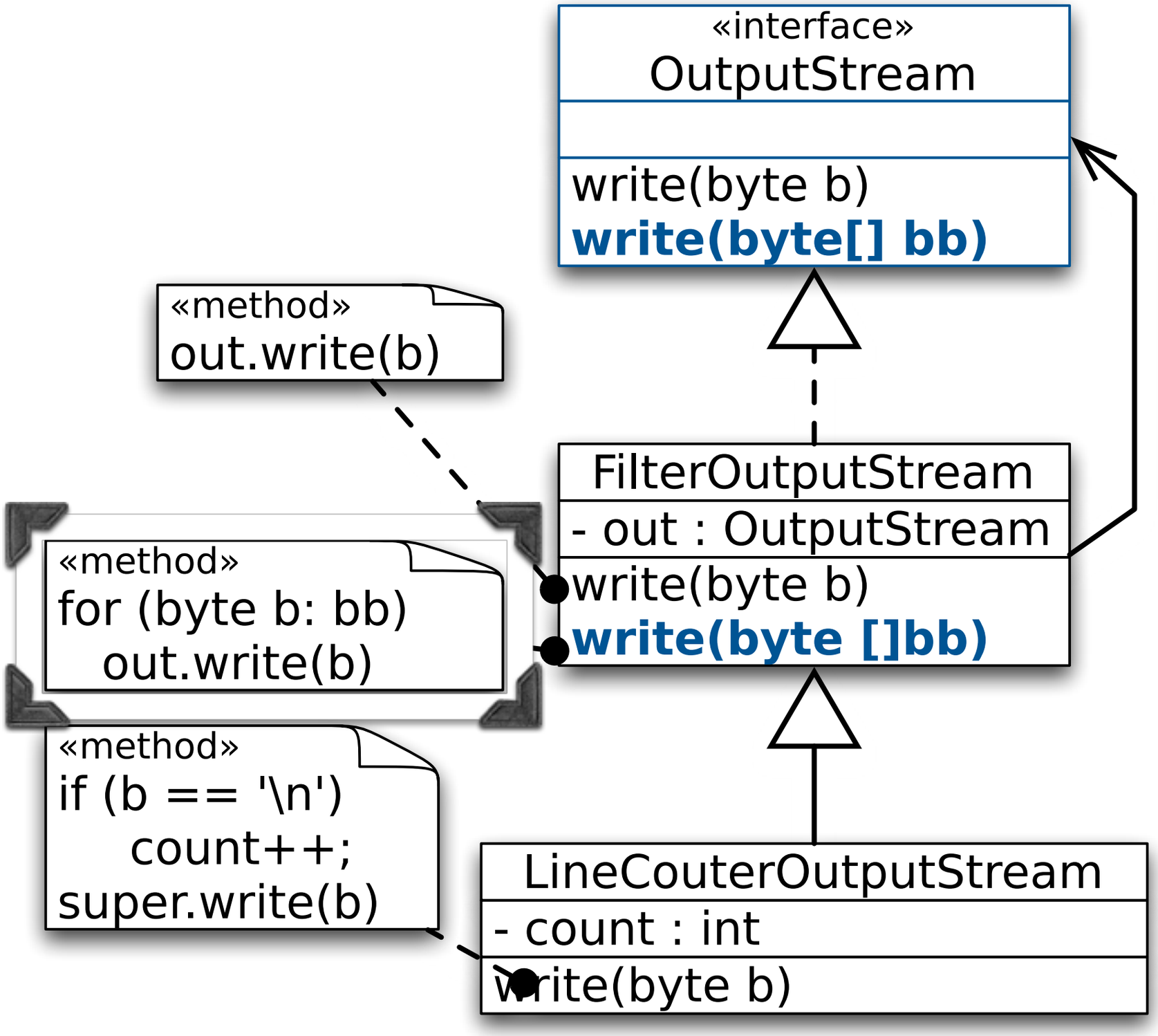
# Fragile Base Class Problem



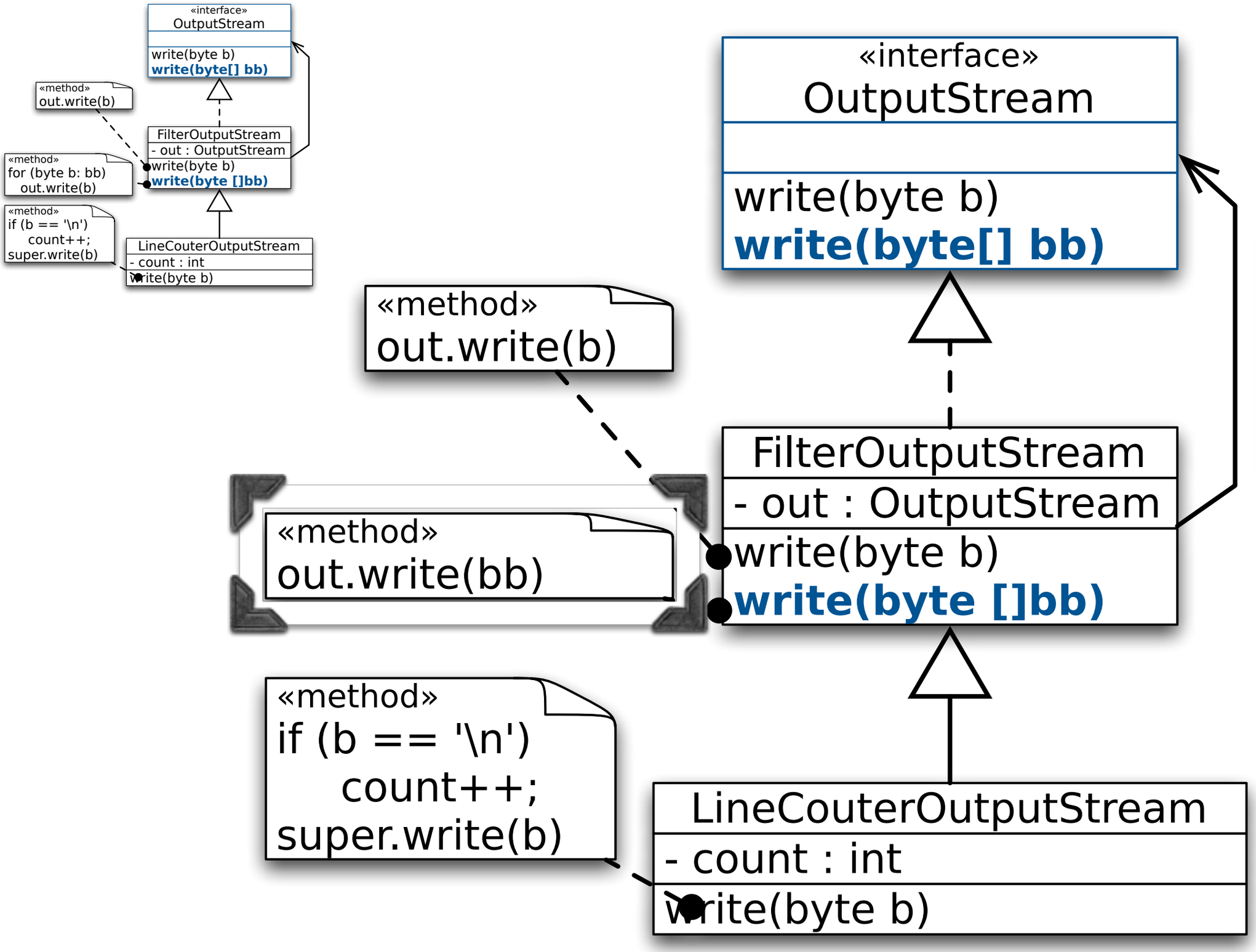
You can't tell whether a base class change is safe simply by examining the base class' methods in isolation.

- you must look at (and test) all derived classes as well
- moreover, you must check all code that uses the base class and also the derived class, since this code might also be broken by the new behavior

A simple change to a key base class can render an entire program inoperable.




**OutputStream** and **FilterOuputStream** are developed as part of a library; **LineCounterOuputStream** is a user extension

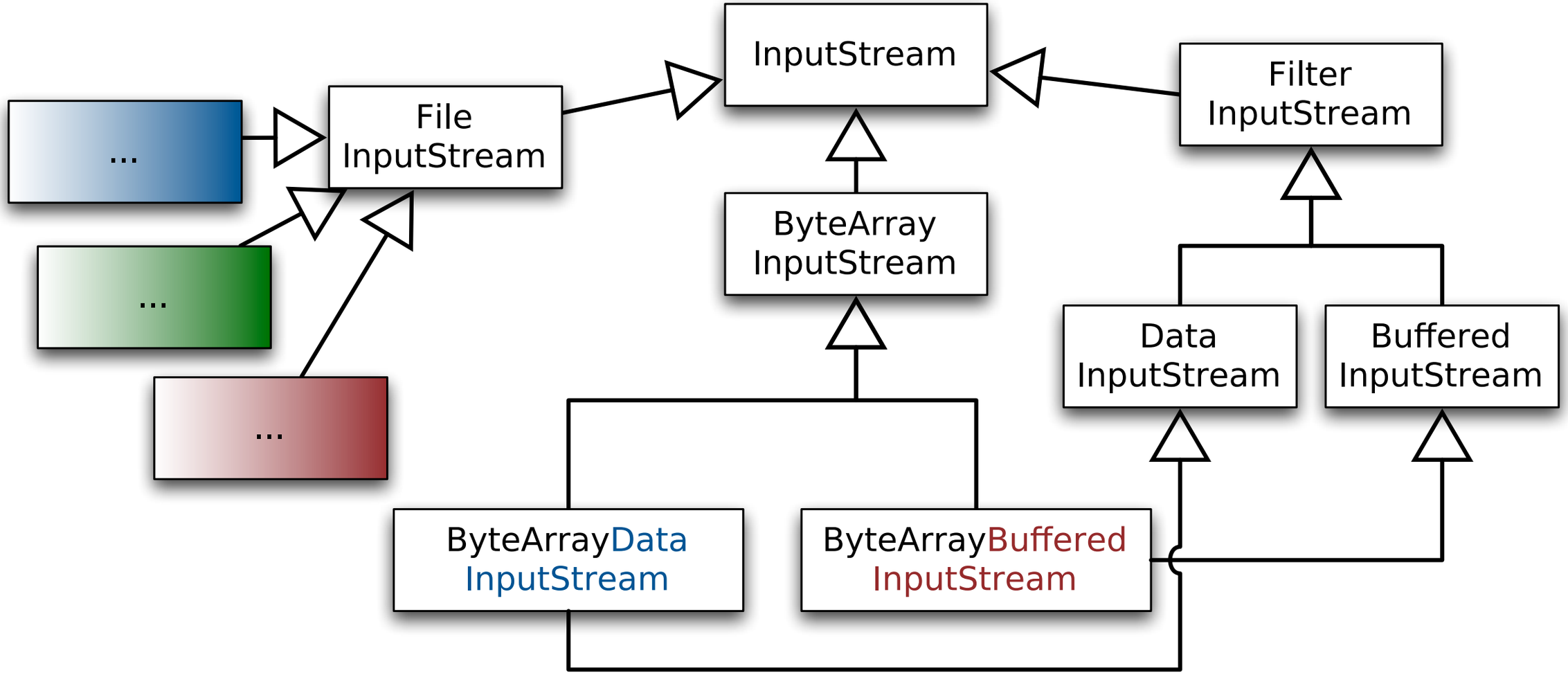



For performance reasons **write(byte [] bb)** is reimplemented to write all bytes at once. **LineCounterOutputStream** does no longer work correctly if **write(byte[])** is used.

# The Decorator Design Pattern
# Motivation
# Disadvantages of Multiple Inheritance



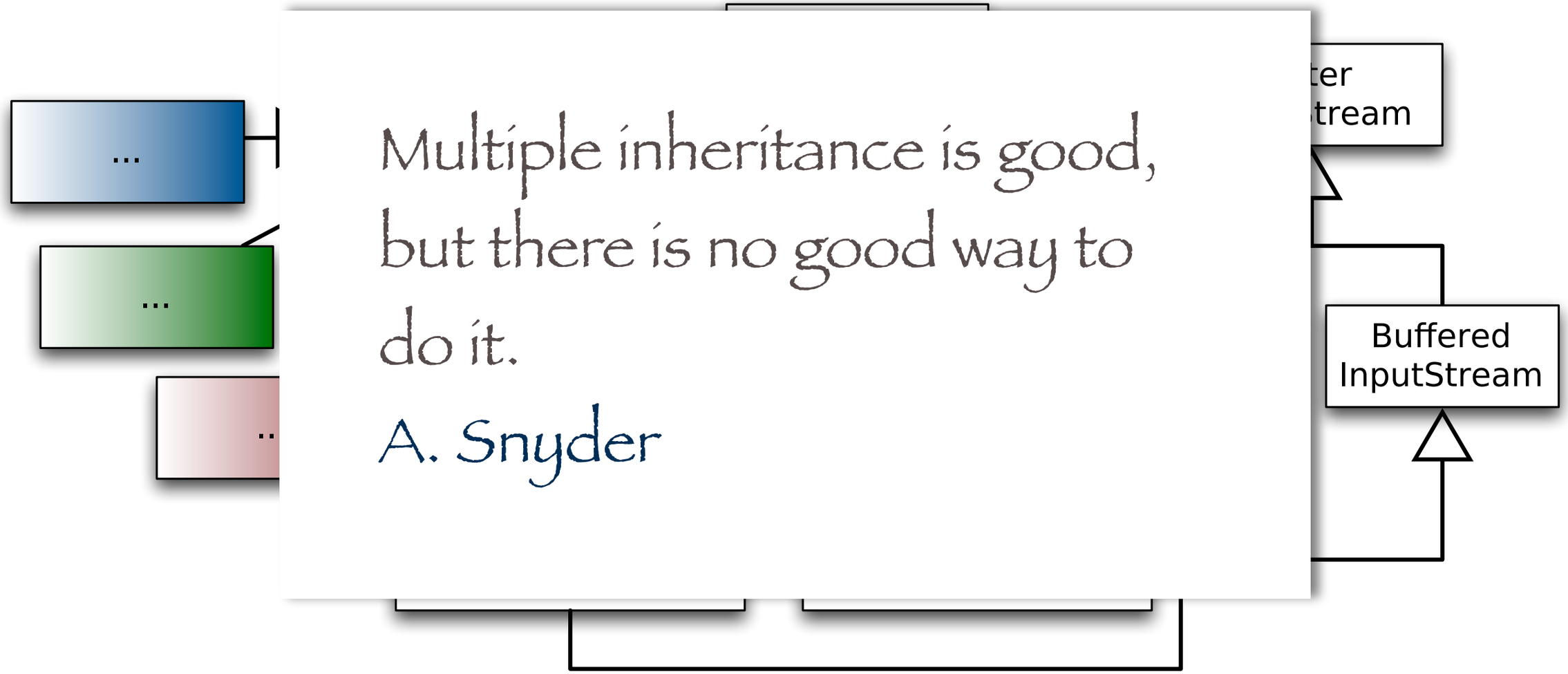
- static responsibility mix
- naming conflicts
- hard to dispatch super calls correctly



# The Decorator Design Pattern
# Motivation
# Disadvantages of Multiple Inheritance



- static responsibility mix
- naming conflicts
- hard to dispatch super calls correctly




Multiple inheritance is good, but there is no good way to do it.

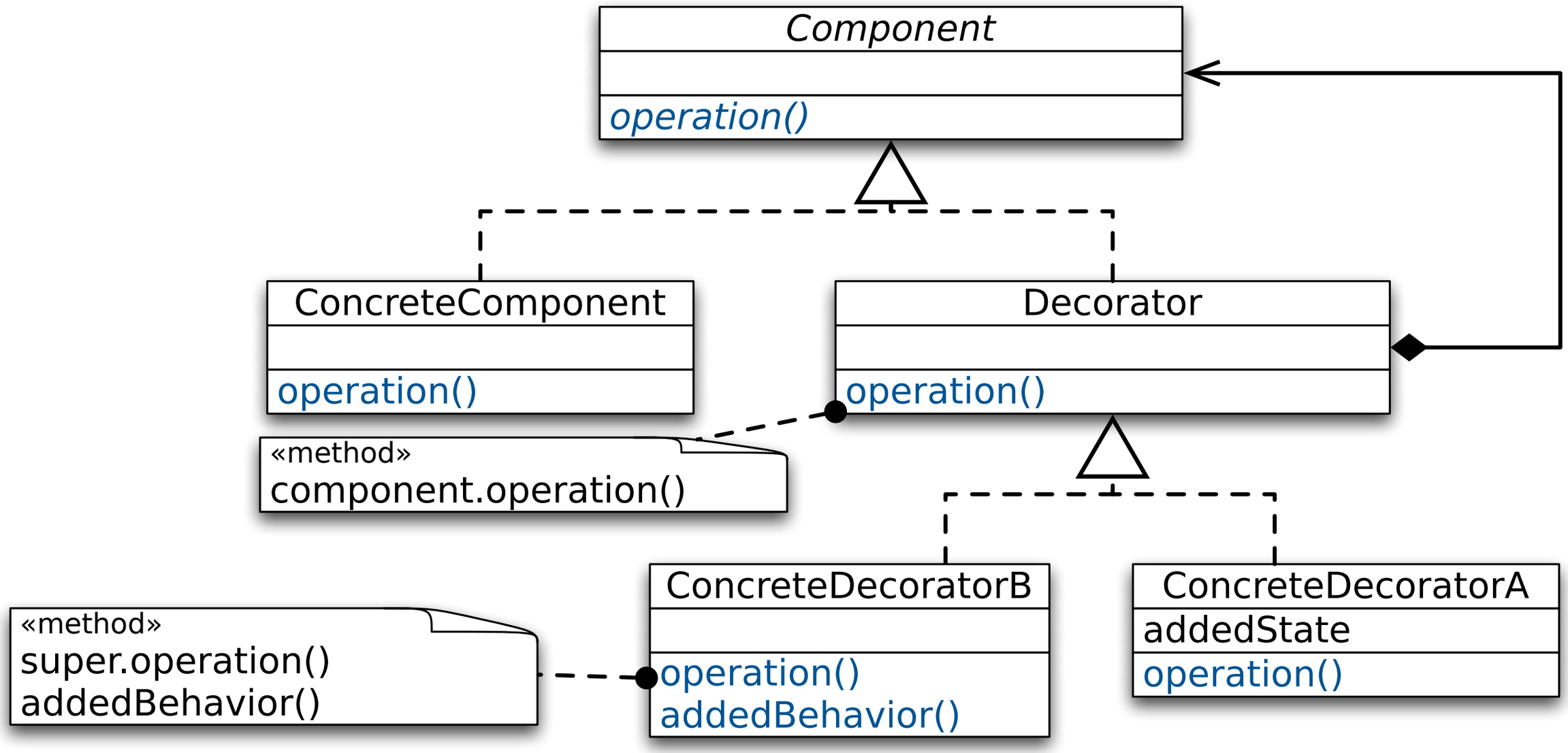
A. Snyder

## Intent

We need to add responsibilities to existing objects dynamically and transparently, without affecting other objects.

# The Decorator Design Pattern
## Structure

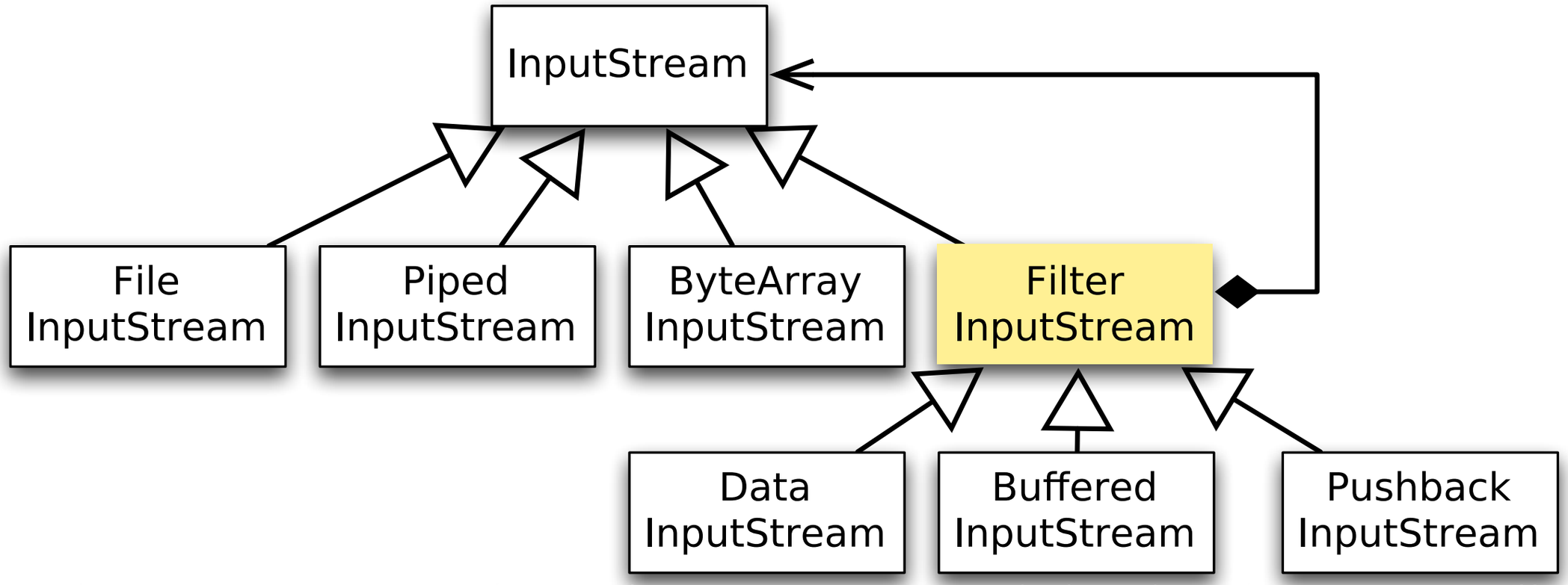




# The Decorator Design Pattern

## Example: The Decorator Pattern and “java.io.*”






java.io abstracts over various data sources and destinations, as well as processing algorithms:

- Programs **operate on stream objects ...**
- **independently of** ultimate data source / destination / shape of data.

Example:

```
new DataInputStream(new FileInputStream(“...”)).readUnsignedByte()
```

# The Decorator Design Pattern
## Advantages



**Decorator** enables more flexibility than inheritance

- Responsibilities 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 responsibility twice; e.g., for a double border, attach two **BorderDecorators**

# The Decorator Design Pattern
## Advantages



- **Decorator** avoids incoherent classes
  - feature-laden classes high up in the hierarchy (This also breaks encapsulation.)
  - pay-as-you-go approach don't 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 doesn't use
- a fine-grained **Decorator** hierarchy is easy to extend

# The Decorator Design Pattern
## 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

# The Decorator Design Pattern
## Problems



### Object identity

- A **decorator** and its component aren't identical
  From an object identity point of view, a decorated component is not identical to the component itself.
- You shouldn't rely on object identity when you use decorators

# The Decorator Design Pattern
## Example: The Decorator Pattern and “java.io.*”



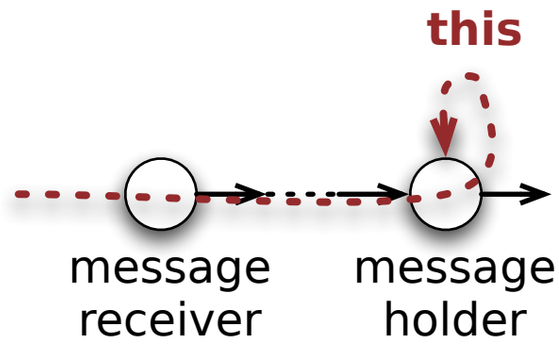
- A stream is normally addressed via the outermost Decorator
- Sometimes, a reference to one of the internal objects is maintained and operated on;
  good style: all read() operations are performed only to the head decorator in the composite stream object
- Reading from an internal object breaks the illusion of a single object accessed via a single reference, and makes the code more difficult to understand

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

# Delegation vs. Forwarding Semantics

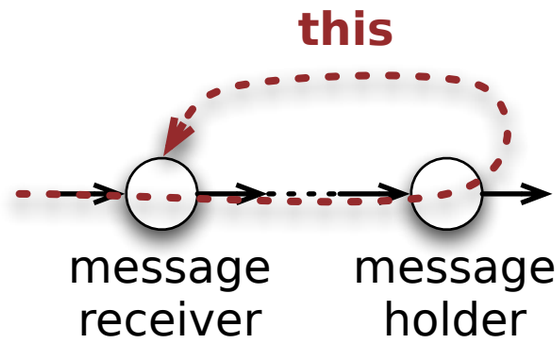


## Forwarding




Forwarding with binding of **this** to method holder; "ask" an **object to do something on its own**.

## Delegation

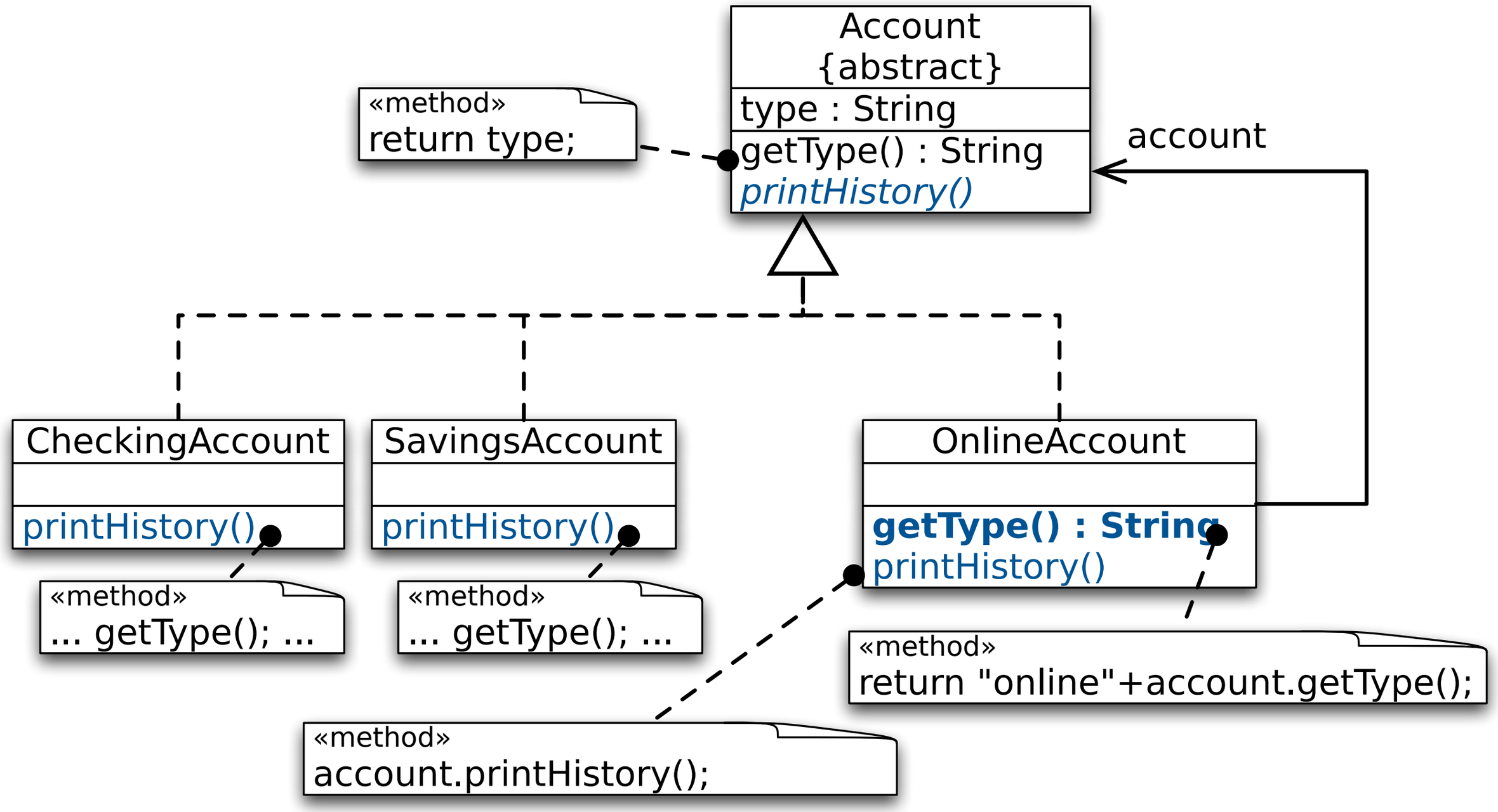



Binding of **this** to message receiver: “ask” an object to **do something on behalf of the message receiver**.

# The Decorator Design Pattern

## No Late Binding (Delegation vs. Forwarding Semantics)

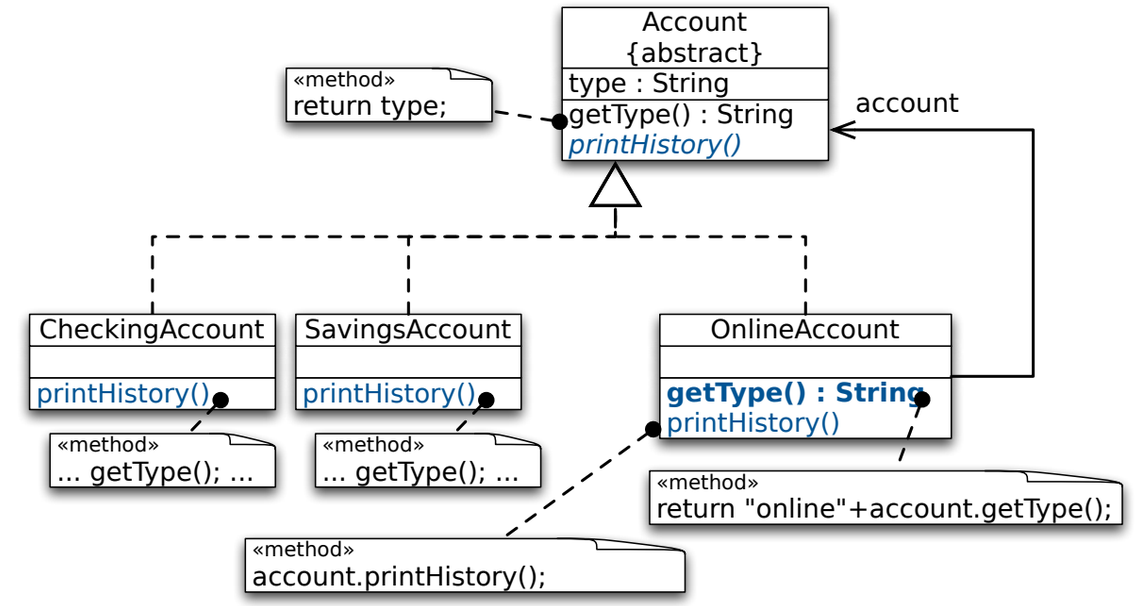




# The Decorator Design Pattern

## No Late Binding (Delegation vs. Forwarding Semantics)






Consider the following scenario:

1. A checking account, **ca**, is created
2. An online decorator, **od**, is created with **ca** as its account attribute
3. Call to **od.printHistory()**
   a. call to **ca.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**.

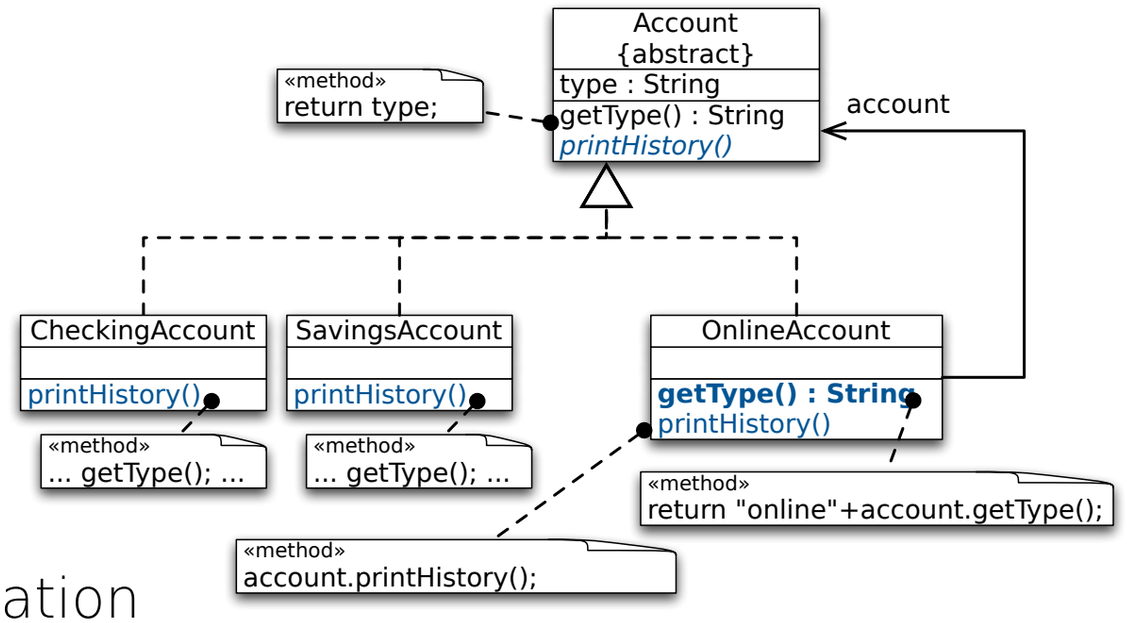
# The Decorator Design Pattern

## No Late Binding (Delegation vs. Forwarding Semantics)



Consider the following scenario:

1. A checking account, **ca**, is created
2. An online decorator, **od**, is created with **ca** as its account attribute
3. Call to **od.printHistory()**
   a. call to **ca.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**.




Problem:

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

# The Decorator Design Pattern
## Implementation



- 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 don't need
- These issues require pre-planning
  Difficult to apply decorator pattern to 3rd-party component class.

# The Decorator Design Pattern
## Implementation



- A decorator's interface must conform to the interface of the component it decorates; **ConcreteDecorator** 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 **ConcreteDecorator**