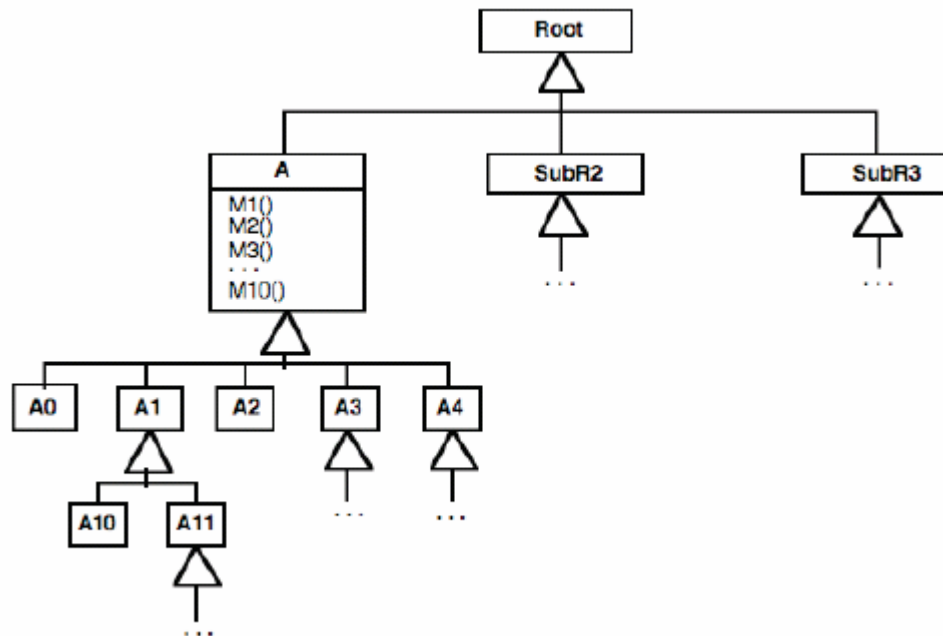


Construction of Flexible Software

- Composition
- **Decorator**

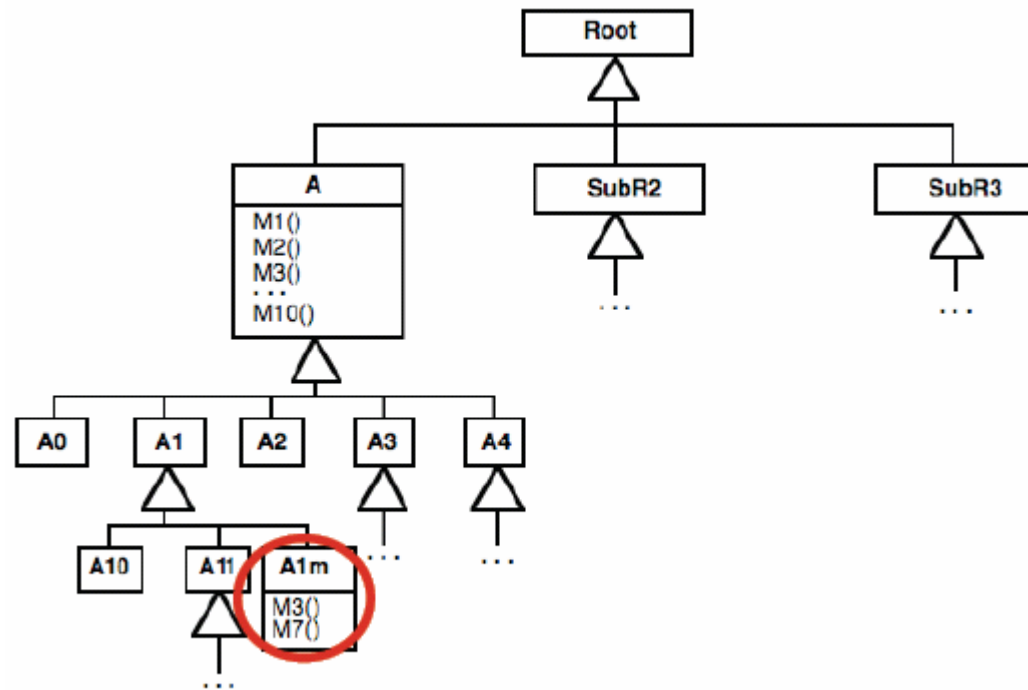
The Decorator Construction Principle

Motivation: Changes of a Class With Many Subclasses (I)



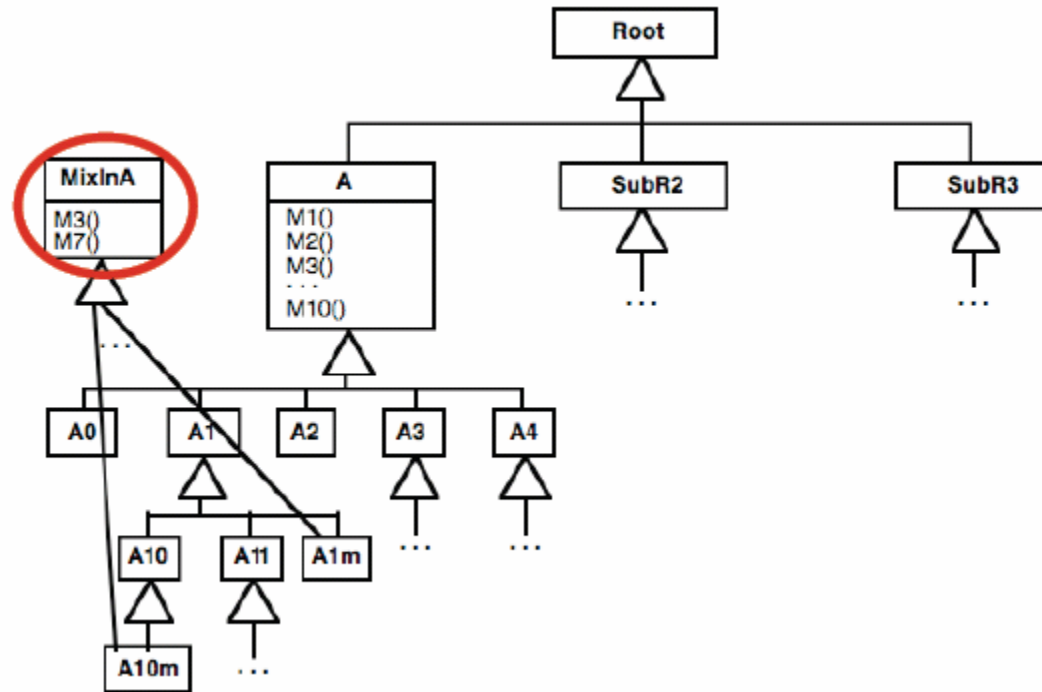
- Changes of M3() and M7() of class A necessary
- Change source text of A, if available?
- Change by inheritance?

Motivation: Changes of a Class With Many Subclasses (II)



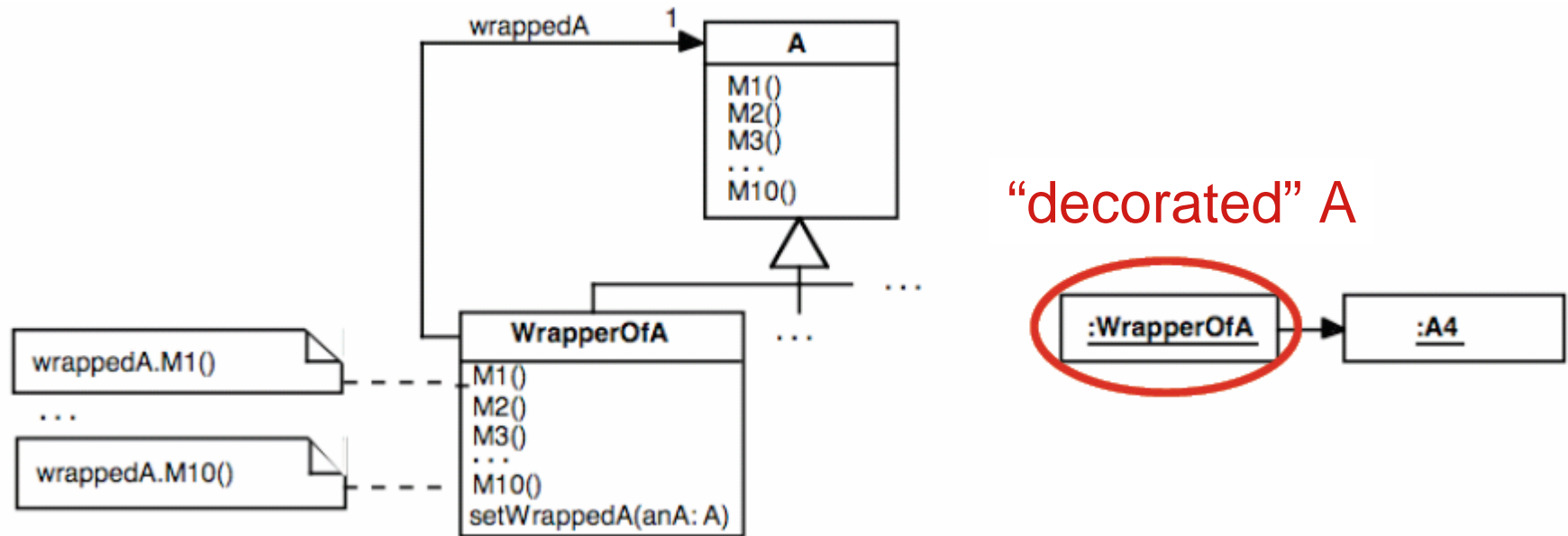
- For one class (e.g., A1m) the adaptation is meaningful
- For all subclasses of A this is too complicated

Motivation: Changes of a Class With Many Subclasses (III)



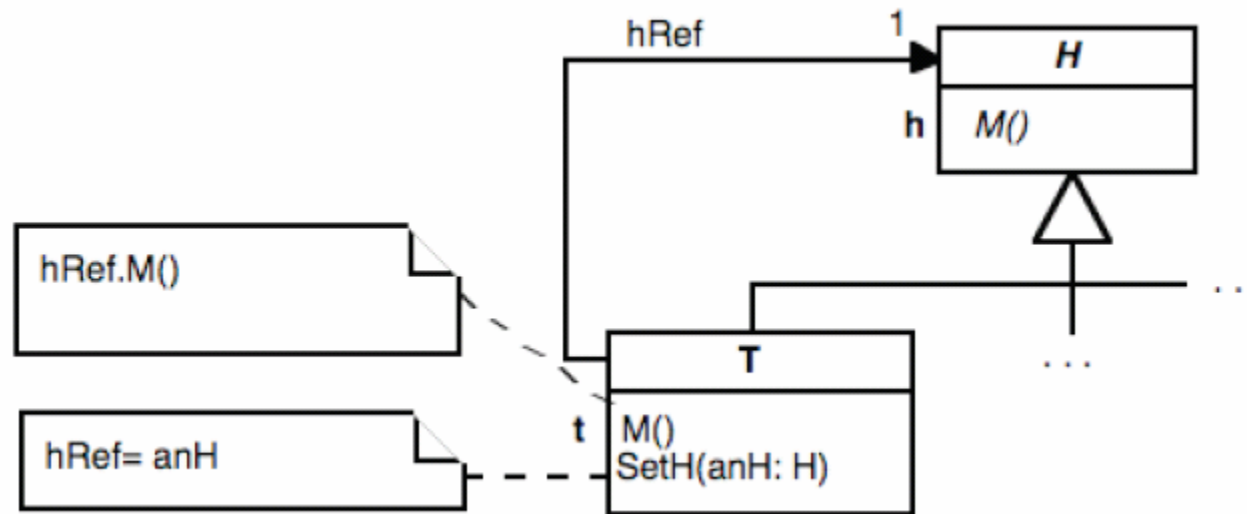
- In programming languages that support multiple inheritance, so-called **mixin** classes can be defined.
- Nevertheless a subclass must be formed for each class whose behavior is to be adapted.

Adapting a class by composition rather than by inheritance



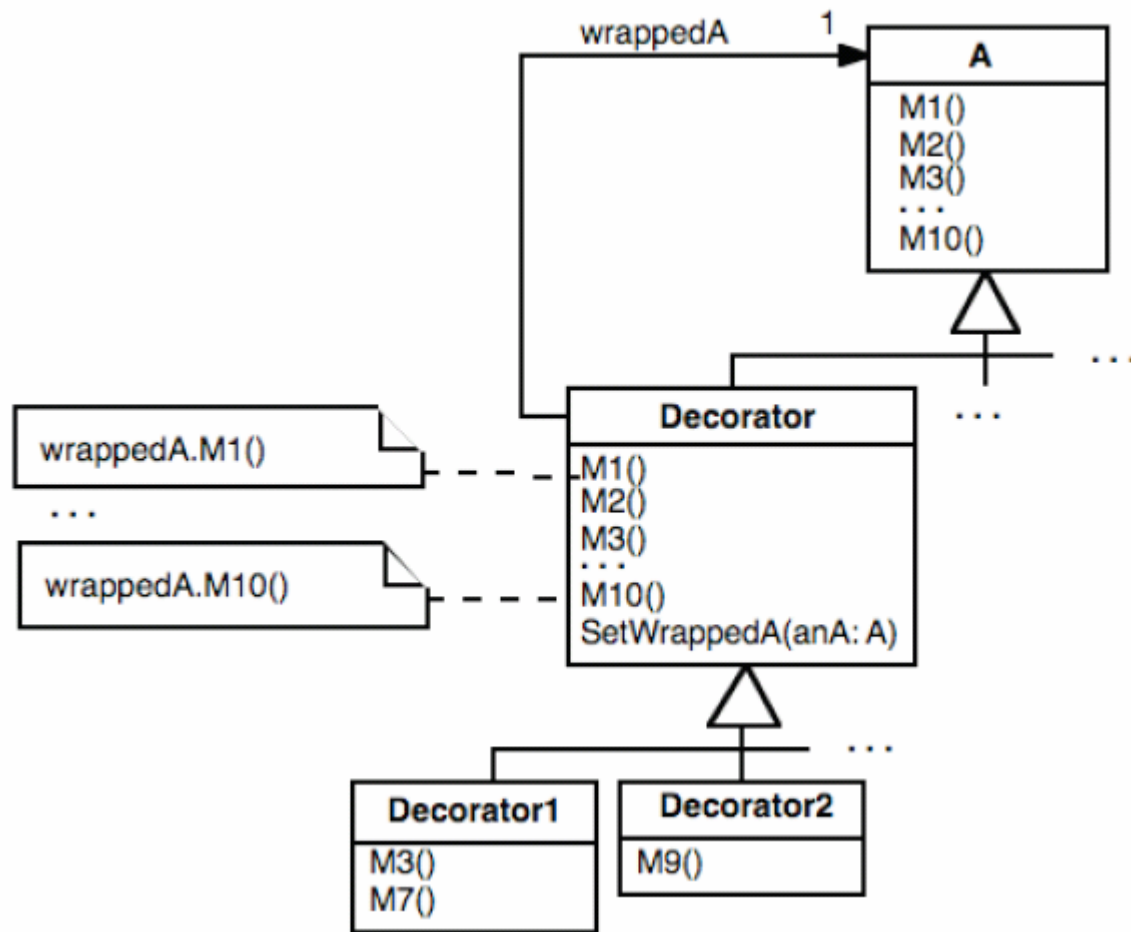
- All A's methods are overwritten in the class `WrapperOfA`, as the method call is delegated in each case to an object referenced by the instance variable `wrappedA` - with exception of those which are changed.
- Since `WrapperOfA` is a subclass of `A`, any time an object of static type `A` is demanded, an instance of the class `WrapperOfA` can be used. Since the instance variable `wrappedA` has the static type `A`, it can refer to each object of a subclass of `A`.

Decorator: Adaptation by composition with as many objects as desired

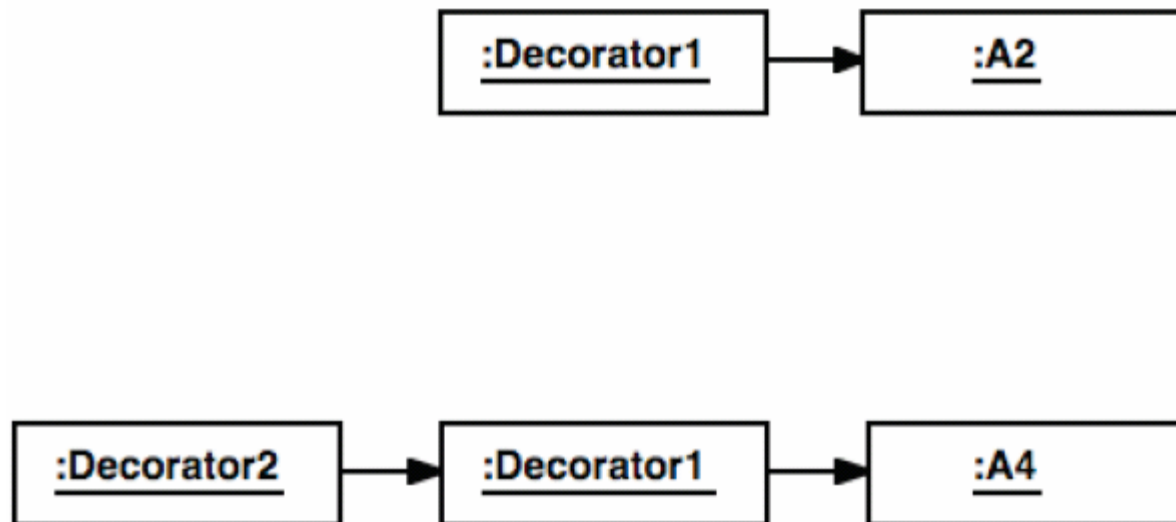


- The names of the template and hook methods are the same
- Setting of the decorated object with **SetH()**
- An instance of the decorator (filter) **T** as well as an instance of a subclass of **H** can be used by clients like an **H**-object. However, the behavior of the **H**-object is modified accordingly.
- **H** + decorator(s) can be used as a single object (see Composite).

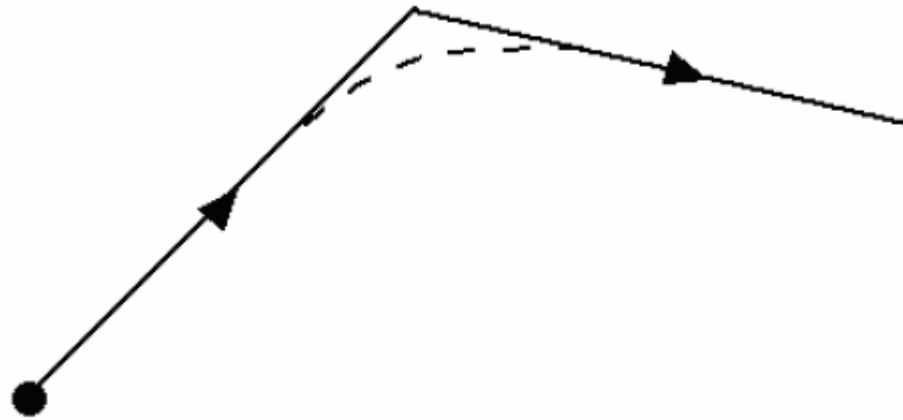
Design suggestion when using several Decorator classes

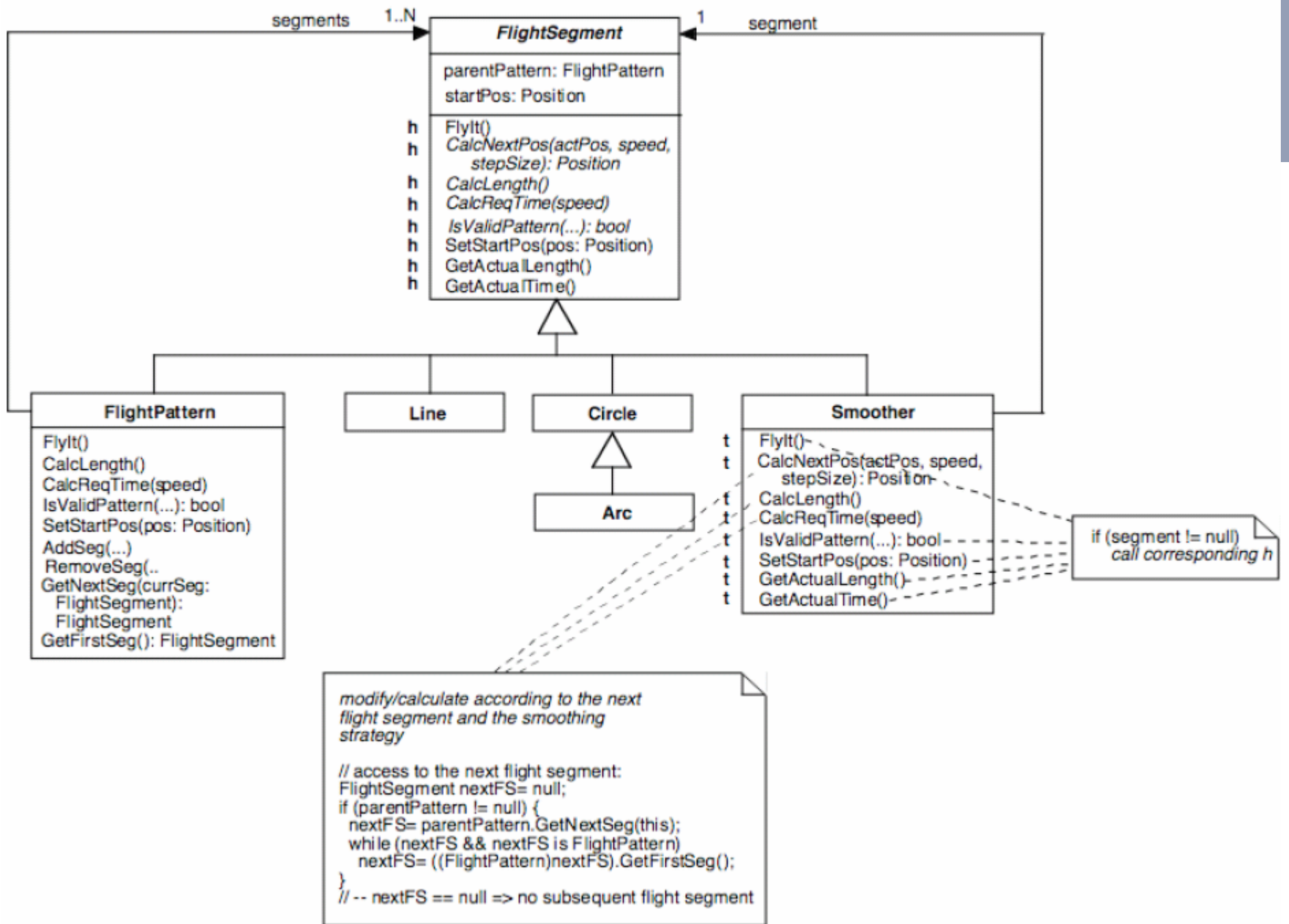


Two Exemplary Compositions



Example: Smoothing Flight Patterns





Use of the Decorator Class Smoother

```
FlightPattern triangle= new FlightPattern();  
triangle.SetStartPos(...);  
triangle.AddSeg(new Smoother(new Line(...)));  
triangle.AddSeg(new Smoother(new Line(...)));  
triangle.AddSeg(new Line(...));
```

Basic conditions for the application of the Decorator construction principle (I)

- The signature of H, which is the root class of the subtrees, is not to be extended by the subclasses of H. The reason for this is that additional methods in the subclasses cannot be considered into the Decorator.
- In order to guarantee fulfillment of this demand, it is necessary to transfer the common aspects of all subclasses of H into the root class. This requirement is not satisfied by many class libraries. In such cases, the application of the Decorator construction principle is not possible to full extent (see Decorator Smoother).

Basic conditions for the application of the Decorator construction principle (II)

In our example, some specific methods for the mentioned objects must be explicitly called – for example, `SetDirection()` in `Circle` - since they cannot be passed on by a `Smoother` instance:

```
Circle circle= new Circle (...);  
circle.SetDirection (cRight);  
Smoother smoother= new Smoother (circle);
```

If the mentioned demand would be fulfilled, a `Smoother` instance could be treated like each specific `FlightSegment` object:

```
Smoother smoother= new Smoother (new Circle (...));  
smoother.SetDirection (cRight);
```

A possibility of eliminating the flight-segment-specific methods is to let all characteristics be indicated only over the constructor of the respective class:

```
Smoother smoother= new Smoother (new Circle (... , cRight));
```

Application of the Decorator principle to the design of more “lightweight” root classes

- The Decorator construction principle can be used to make classes close of the root of the class tree more lightweight. Functionality that is not needed in all classes is implemented in Decorator classes. Only the objects which need this special functionality receive it by composition with the appropriate Decorator instance.
- The Decorator construction principle can be fruitfully used both with the (first) design of a class hierarchy and with the extensions of class hierarchies.

Example: Clipping Mechanism in GUI libraries

- Clipping mechanism: cutting a GUI element to its fixed size
- Since the Clipping mechanism is not needed for all GUI elements, it is meaningful to plan the Clipping mechanism by a decorator class Clipper rather than in the root of the subtree (see the Decorator example in Gamma et al., 1995).

Summary *Decorator*

- + Simple adaptation by object composition
- + New decorator elements (Template classes, which are subclasses of the Hook class) can be defined, without having to change the subclasses of the Hook class.
- + „More lightweight “classes can be realized elegantly
- The Hook class should fulfill the mentioned basic condition (factoring in behavior from all subclasses)
- Additional indirection in method calls
- Complex interactions between involved objects