Chapter 6. “Applications of Aspects”

Today‘s talk largely based on joint work with Tobias Rho

Often-Cited Aspect Applications

- Logging
  - Nice for demos but no killer application
- Persistence
  - Papers of Awais Rashid & al, and Paulo Borba & al.
  - Paperwork, no implementations available
- WebSphere
  - Paper and blog of Adrian Colyer
  - Serious implementation but code not publicly available (commercial)
- Design Patterns
  - Our focus today

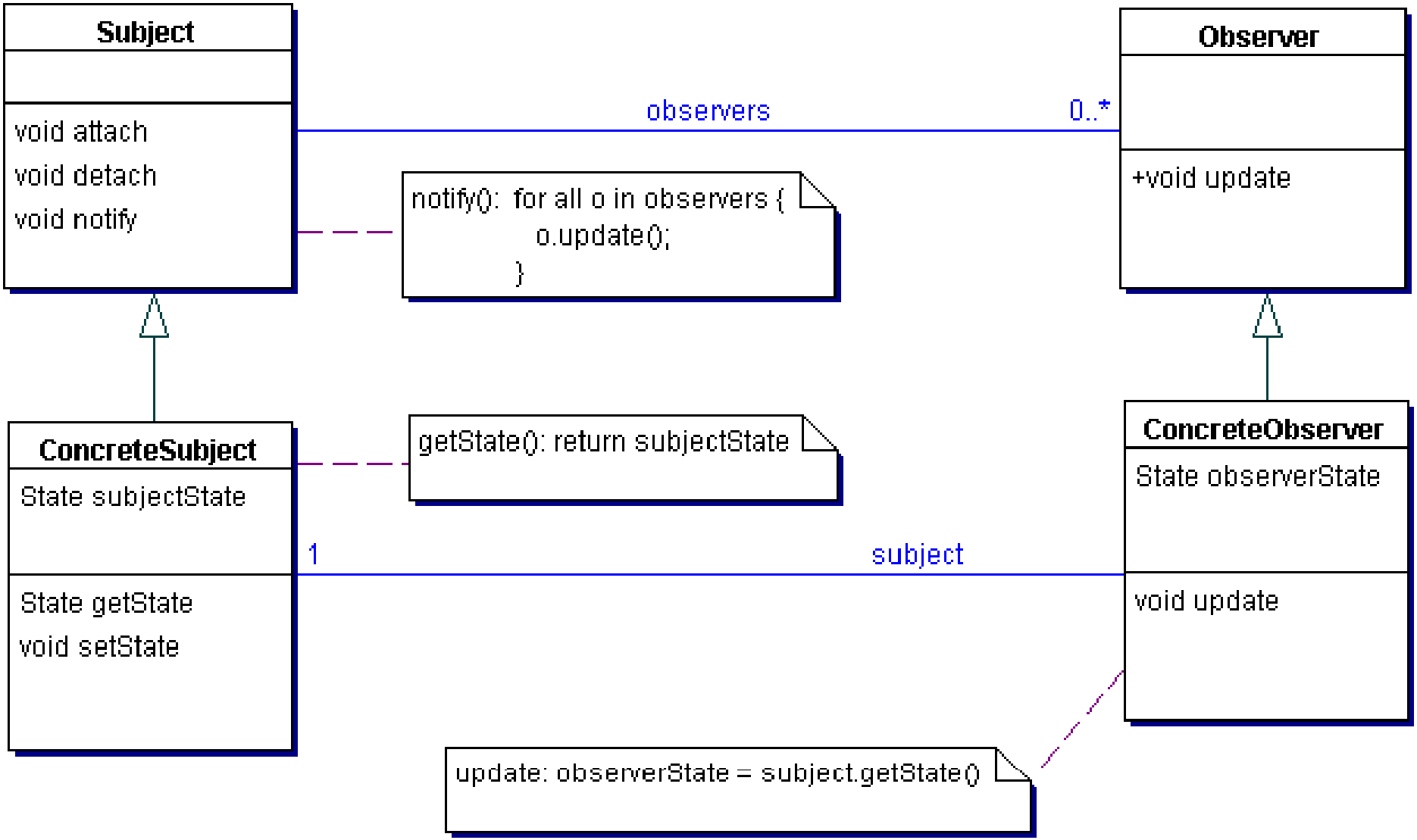
Design Patterns with AspectJ

Representation of Roles as Marker Interfaces

Use of Introductions on Interfaces

Centralized Management

The Observer Pattern



Abstract Observer Protocol aspect, Jan Hannemann 2002

```
public abstract aspect ObserverProtocol {

  protected interface Subject  { }
  protected interface Observer { }

  abstract protected pointcut
              subjectChange(Subject s);

  abstract protected void
     updateObserver(Subject s, Observer o);

  after(Subject s): subjectChange(s) {
    Iterator iter = getObservers(s).iterator();
    while ( iter.hasNext() ) {
      updateObserver(s,((Observer)iter.next()));}
...
}
```

Roles

Conceptual OPs

Observer update

Update logic

Observer Pattern in AspectJ [Hanneman & Kiczales, OOPSLA 2002]

- Roles as marker interfaces defined in the aspect

```
public abstract aspect ObserverProtocol {

    // This interface models the Subject role.
    protected interface Subject  { }

    // This interface models the Observer role.
    protected interface Observer { }
```

Observer Pattern in AspectJ

- The Subject-Observer Mapping maintained centrally, in the aspect

```
// Stores the mapping between Subjects and Observers.
    private WeakHashMap perSubjectObservers;

// Returns a Collection of the Observers of a subject.
    protected List getObservers(Subject subject) { … }

/* Adds an Observer to a Subject. This is the equivalent of
   attach() but is a method on the aspect, not the Subject. */
   public void  addObserver(Subject subject, Observer observer) {
        getObservers(subject).add(observer);
   }

/* Removes an observer. This is the equivalent of detach() … */
   public void delObserver(Subject subject, Observer observer) {
        getObservers(subject).remove(observer);
   }
```

Observer Pattern in AspectJ

- The `updateObserver()` method and the pointcut defining when it should be used are deferred to concrete subaspects.

```
/**
 * Defines how each Observer is to be updated when a change
 * to a Subject occurs. To be concretized by sub-aspects.
 */
protected abstract void updateObserver(
    Subject subject, Observer observer
);

/**
 * The join points after which to do the update.
 * It replaces the scattered calls to notify().
 * To be concretized by sub-aspects.
 */
protected abstract pointcut subjectChange(Subject s);
```

Observer Pattern in AspectJ

- The **`notify()`** method is replaced by an advice that invokes **`updateObserver()`** when the **`subjectChange()`** poincut matches

```
/**
 * Calls updateObserver(..) after a change of interest
 */
after(Subject subject): subjectChange(subject)
{
    Iterator iter = getObservers(subject).iterator();
    while ( iter.hasNext() ) {
        updateObserver(subject, ((Observer)iter.next()));
    }
}
```

Observer Pattern in AspectJ

- A concrete observer aspect defines the
  - `updateObserver()` method
  - `subjectChange()` poincut

  and introduces the marker interfaces to the classes that should play the corresponding roles.

```
// Where to call updateObserver():
pointcut subjectChange(Subject subject) :
    MyPackage.*.set*(..) &&
    target(subject);

void updateObserver(Subject s, Observer o) {...} ;
```

Observer Pattern in AspectJ

- A concrete observer aspect
  - introduces the marker interfaces to the classes that should play the corresponding roles
  - defines the `updateObserver()` method
  - defines the `subjectChange()` poincut
- See next slide for the code

Observer Pattern in AspectJ

```
public aspect ScreenObserver extends ObserverProtocol {

  // Assings the Subject role to the Screen class.
  declare parents: Screen implements Subject;

  //Assings the Observer role to the Screen
  declare parents: Screen implements Observer;

  // Specifies changes to the Subject.
  protected pointcut subjectChange(Subject subject):
     call(void Screen.display(String)) &&
     target(subject);

  // Defines how Observers are to be updated
  protected void updateObserver(Subject s, Observer o) {
     ((Screen)observer).display("Screen updated by " + s);
  }
}
```

Result: Observer Pattern, Jan Hannemann 2002

- Improved properties of implementation in AspectJ
  - Locality
    - All the code that implements the pattern is in the abstract and concrete aspects
    - No coupling between the participant classes exists
  - Reusability
    - Core pattern code (ObserverProtocol) is abstracted and reusable
    - For each pattern instance, only define one concrete aspect

Result: Observer Pattern, Jan Hannemann 2002

- Improved properties of implementation in AspectJ
  - Composition transparency
    - Even if a class takes part in multiple observing relationships, codes of the class and pattern are not confused
  - (Un)pluggability
    - Participants need not be aware of their role
    - It is easy to switch between using a pattern or not using it

Result Overview, Jan Hannemann 2002

Defining: The pattern is the only reason why the class is there.
Superimposed: An existing class takes a role in a pattern.

		Modularity Properties			
Pattern Name	**Kinds of Roles**	**Locality**	**Reusability**	**Composition Transparency**	**(Un)pluggability**
Abstract Factory, Bridge, Builder, Factory Method, Façade, Interpreter	Defining	✗	✗	✗	✗
Adapter, Command, Composite, Decorator, Flyweight, Iterator, Template Method, State, Proxy, Visitor, Memento, Strategy	Both Defining and Superimposed	✓	7 of 12	10 of 12	✓
Observer, Sinlgeton, Prototype, Chain of Responsibiltiy, Mediator	Superimposed	✓	✓	✓	✓

Conclusion, Jan Hannemann 2002

- Improvement from using AspectJ manifest themselves as a set of properties related to modularity
  - Locality, Reusability, Composition Transparency, (Un)pluggability

- Improvement is directly correlated to the presence of crosscutting structure in the patterns.
  - No improvement if the roles of the classes in the pattern are “defining”.
  - More improvement if the roles are “superimposed”.

Evaluation of the use of AspectJ on the Observer Design Pattern

- Good modularity
  - Centralized implementation of generic part
  - Concrete subaspect provides missing definitions for a particular pattern instance
- Good reusability
  - No hard-coded dependencies on base code
  - Independent evolution of design pattern and base code possible

Patterns beyond AspectJ:

Decorator Pattern asessed with Respect to Indpendent Evolvability

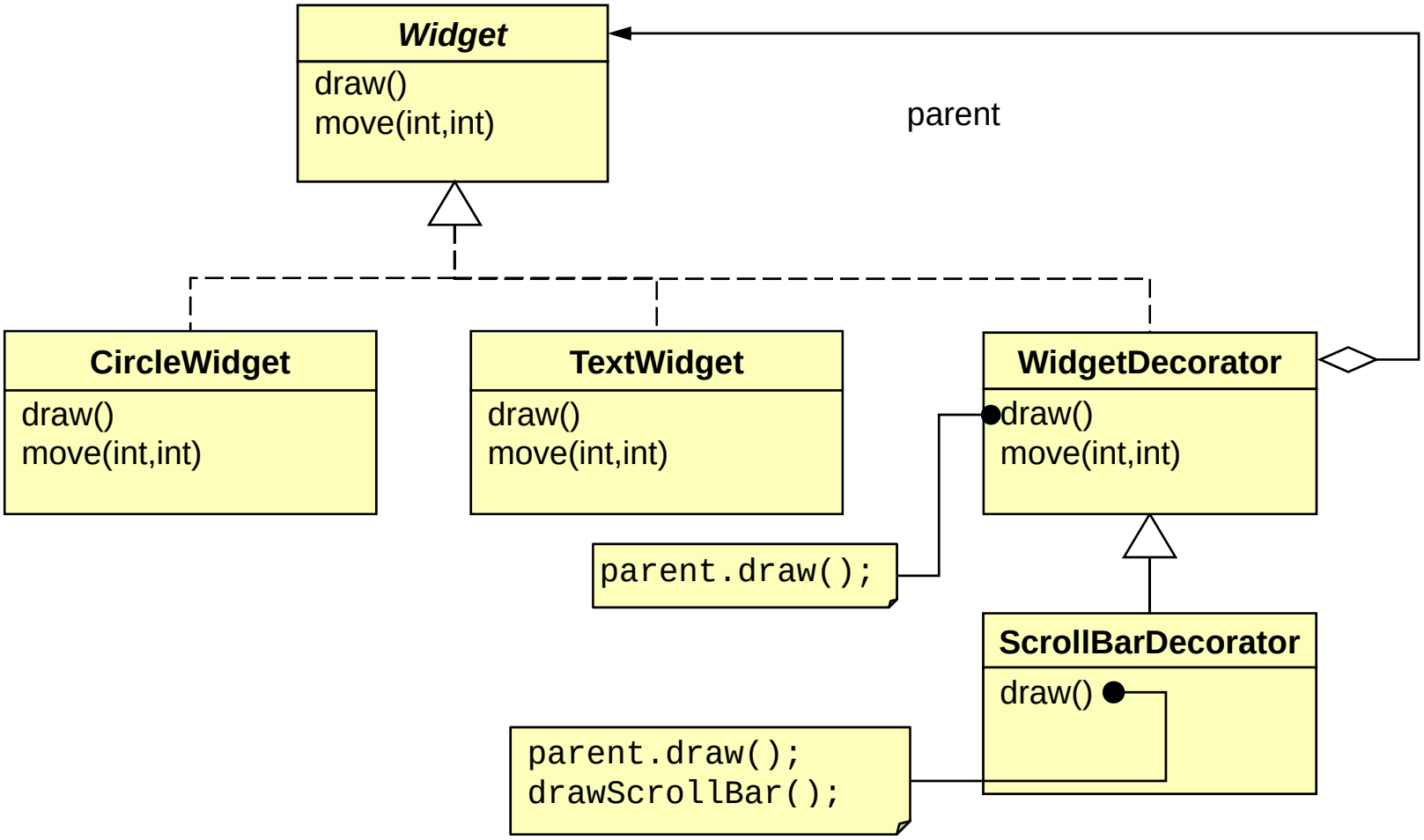
Decorator (GOF)

Evolution of Decorator and base code in GOF-Variant

Decorator with AspectJ

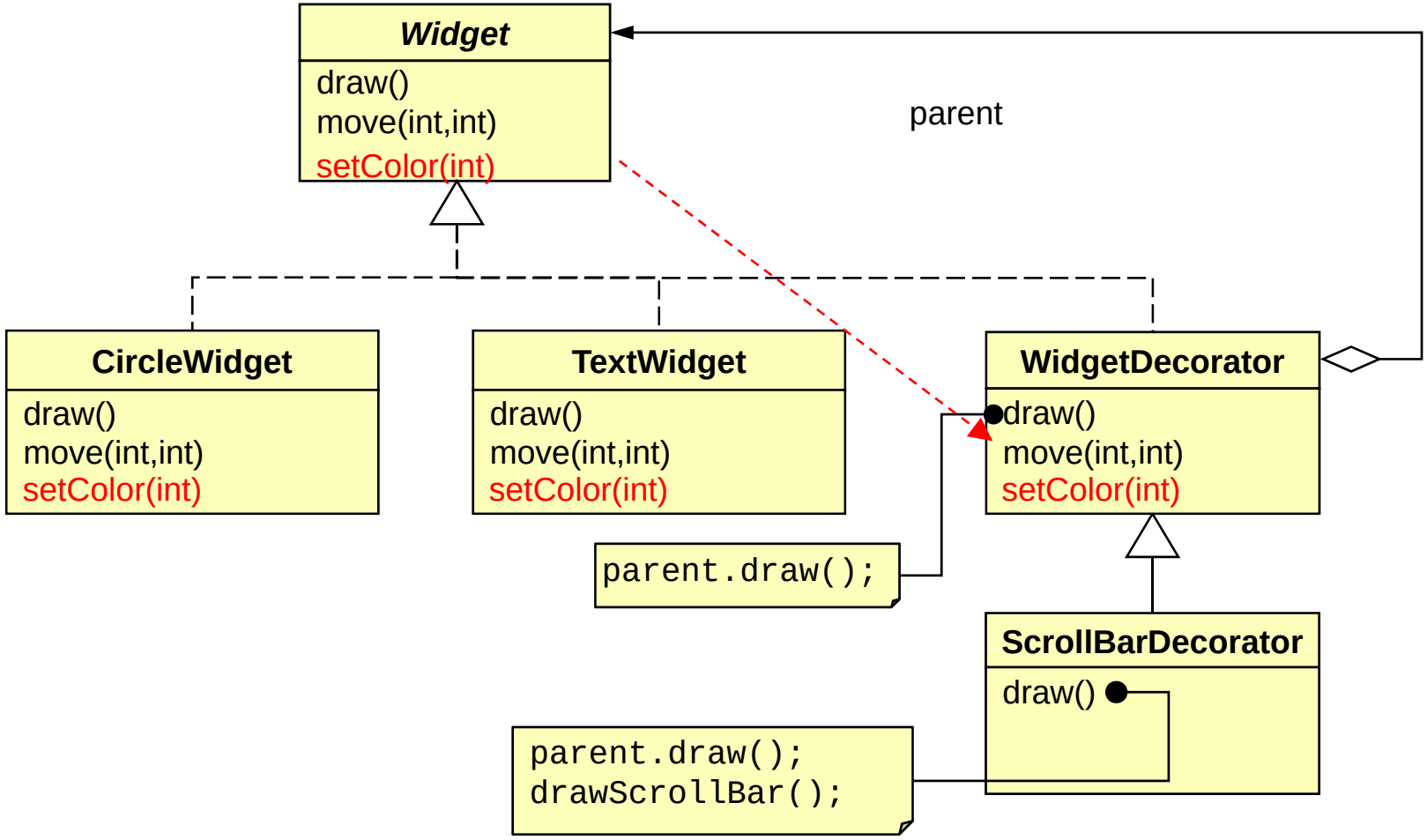
Decorator with LogicAJ

Motivation: Decorator Pattern



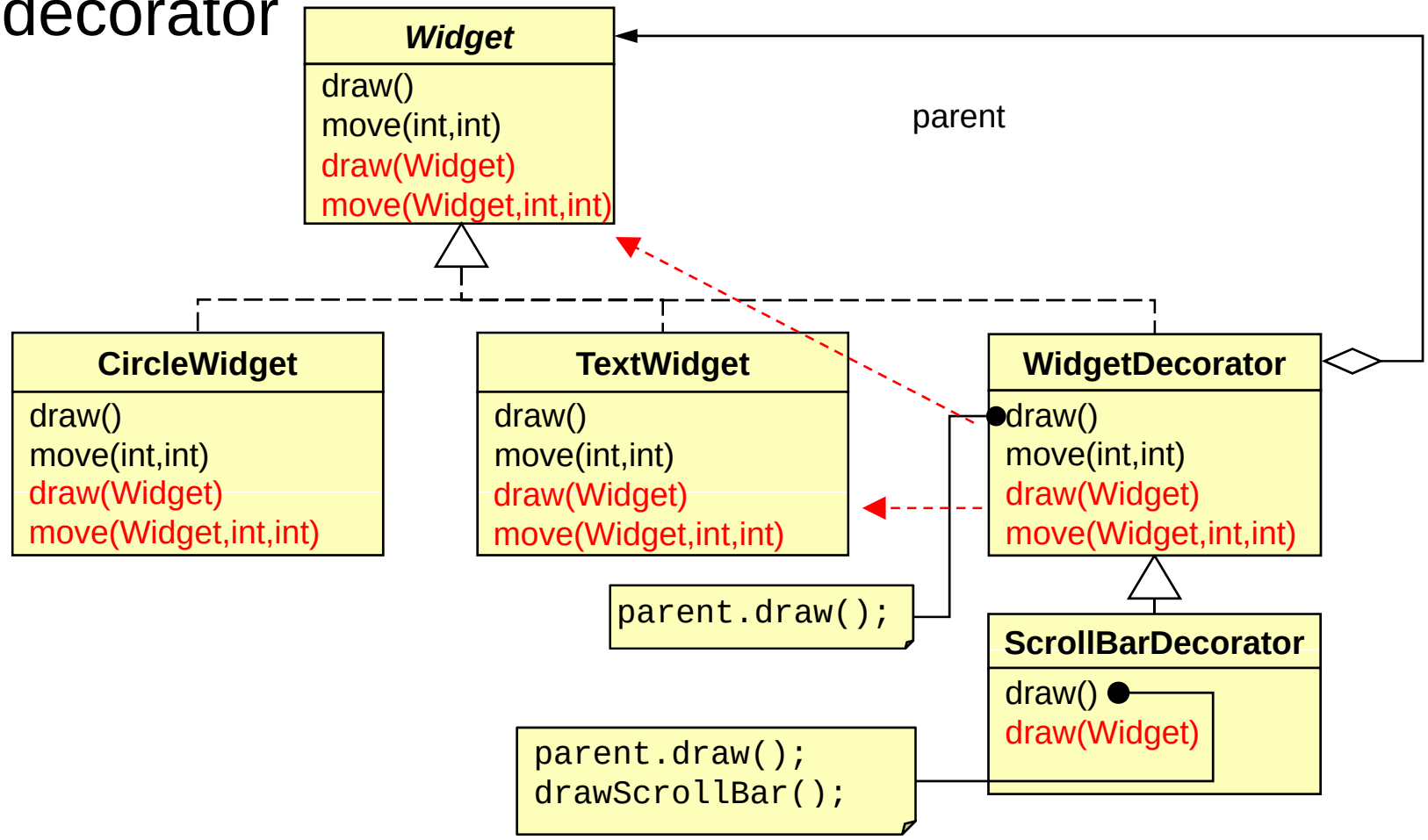
Core Logic Evolution

- Black widget toolkit → colored widget toolkit



Pattern Evolution

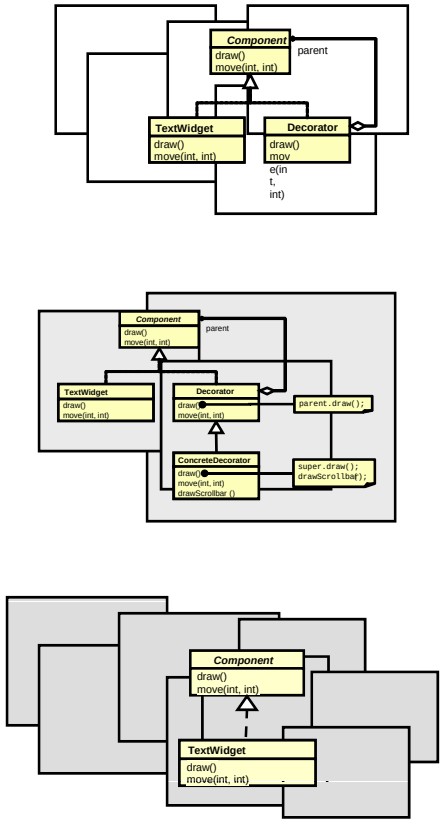
- Enable overriding → add back reference to decorator



Problem of OO implementation

- Core logic and design patterns are tangled
- No independent evolution possible

core logic evolution



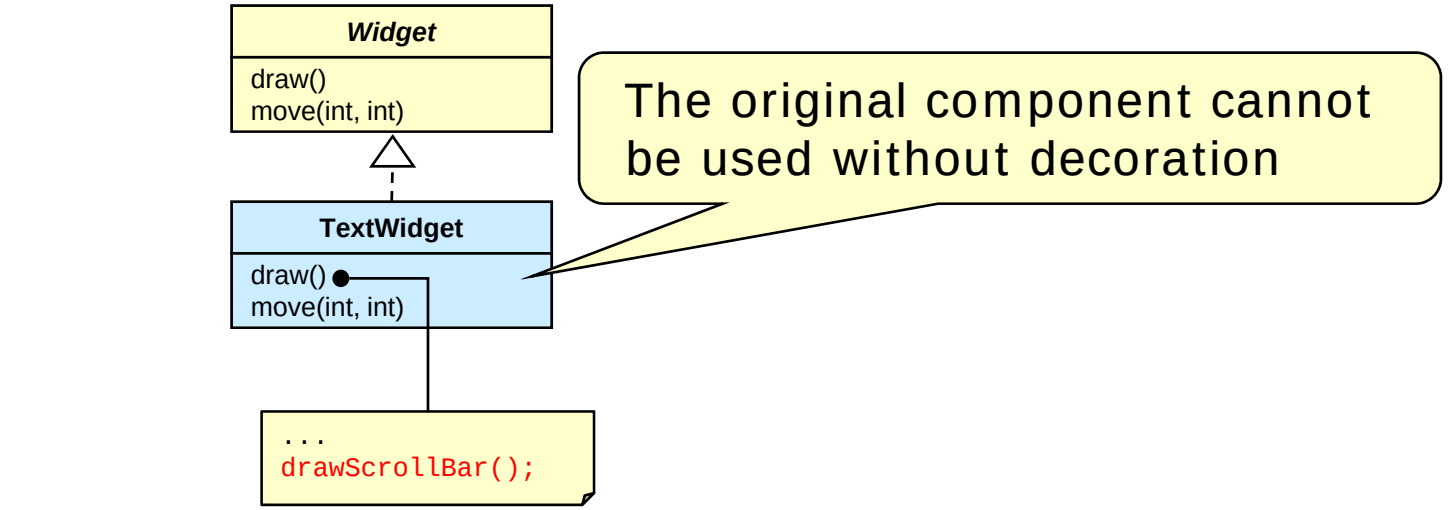
design pattern evolution

Solution?

- Observation
  - DPs are generic descriptions of object collaborations
  - AOP encapsulates object collaborations
- Basic idea
  - Implement design patterns as aspects
  - Specialize aspects for different DP variations

AspectJ Implementation Attempt [HK02]

- Decorating functionality is added into the concrete component

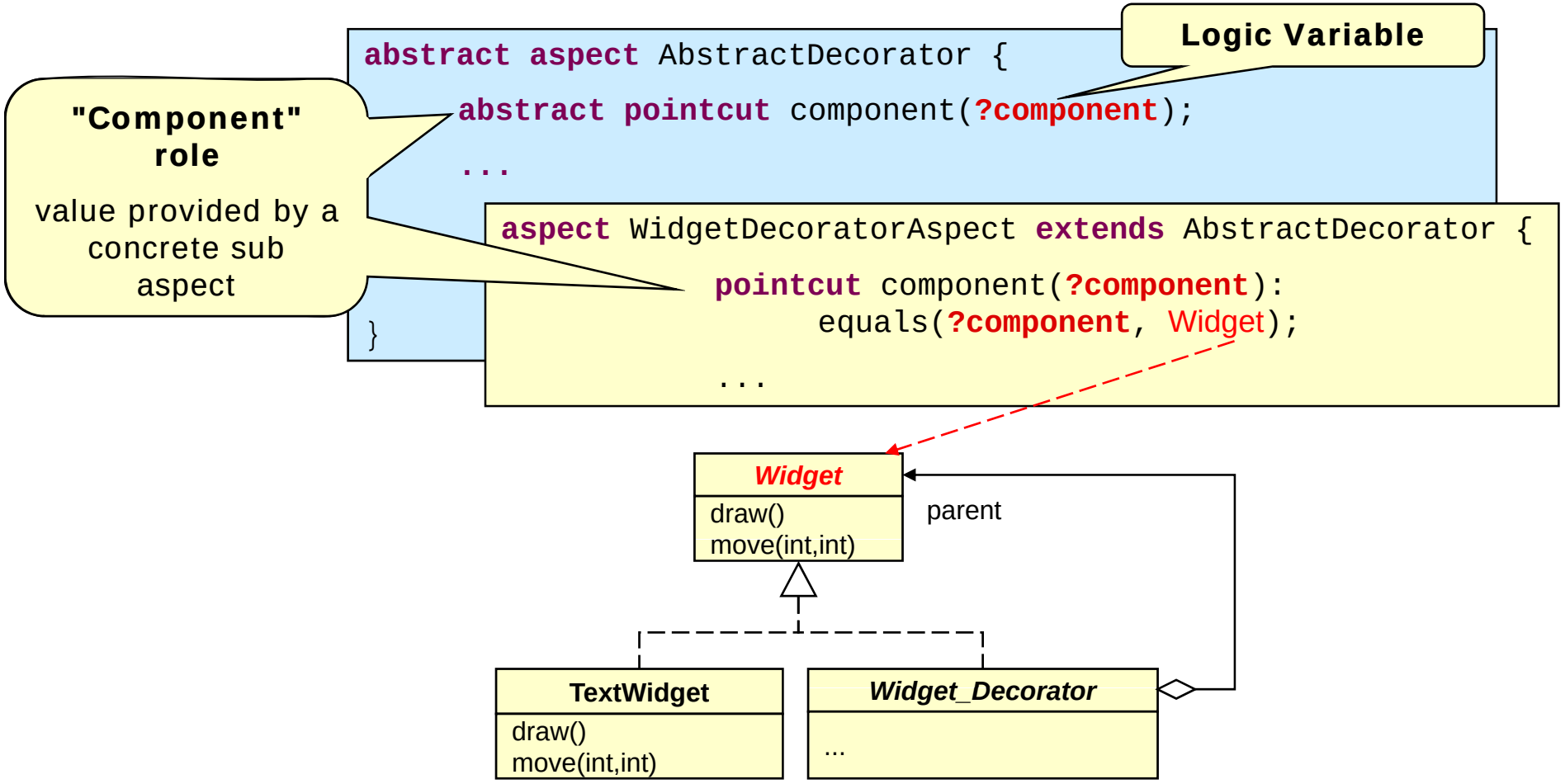



```
aspect ScrollBarAspect {
    void after(): execution(TextWidget.draw()) {
        drawScrollBar();
    }
    drawScrollbar() { … }
}
```

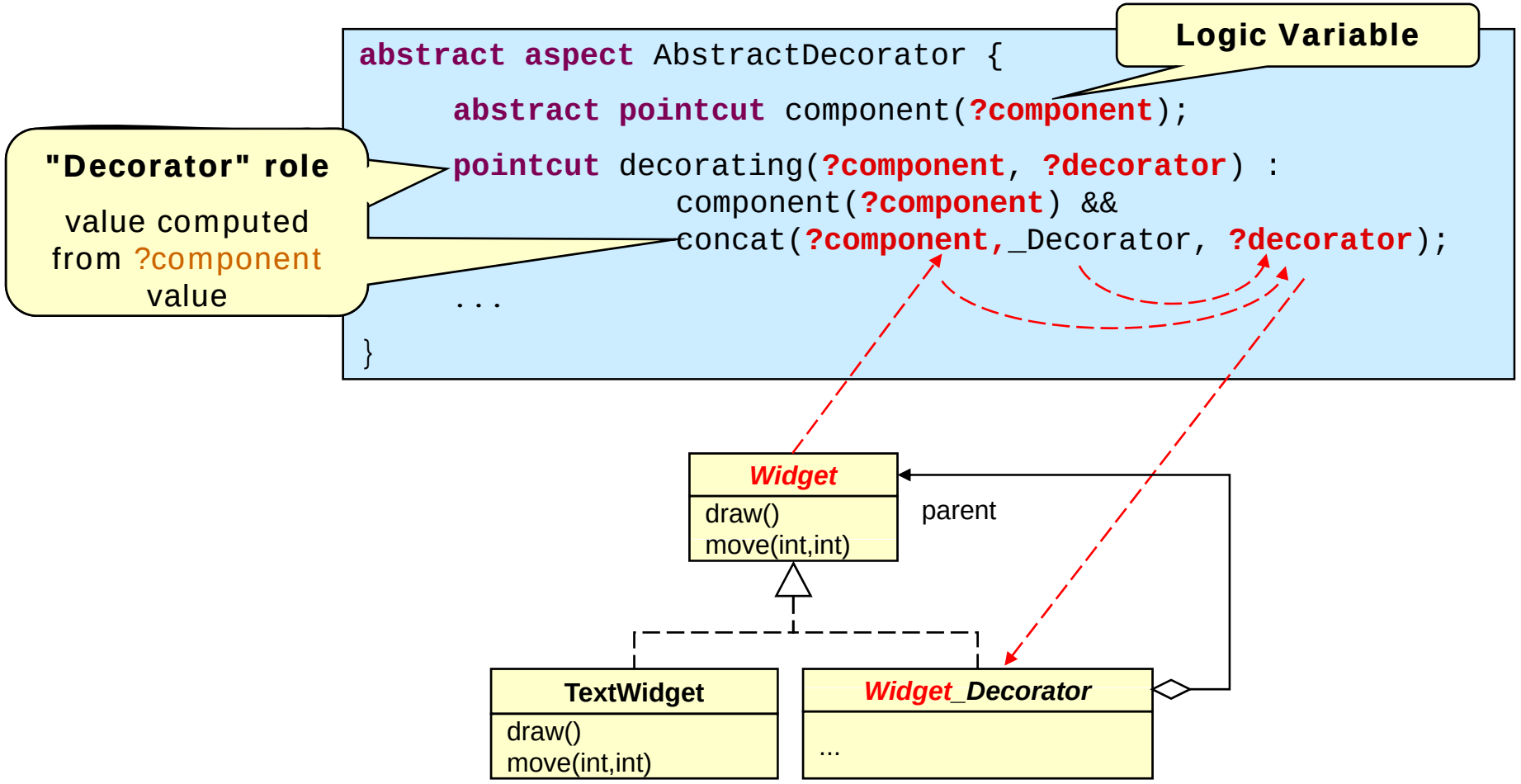
Roles are hardcoded
→ aspects are not reusable

[HK03] Hannemann, J.; Kiczales, G.: Design Pattern Implementation in Java and AspectJ, OOPSLA 2002.

1. Representing Roles by Logic Variables

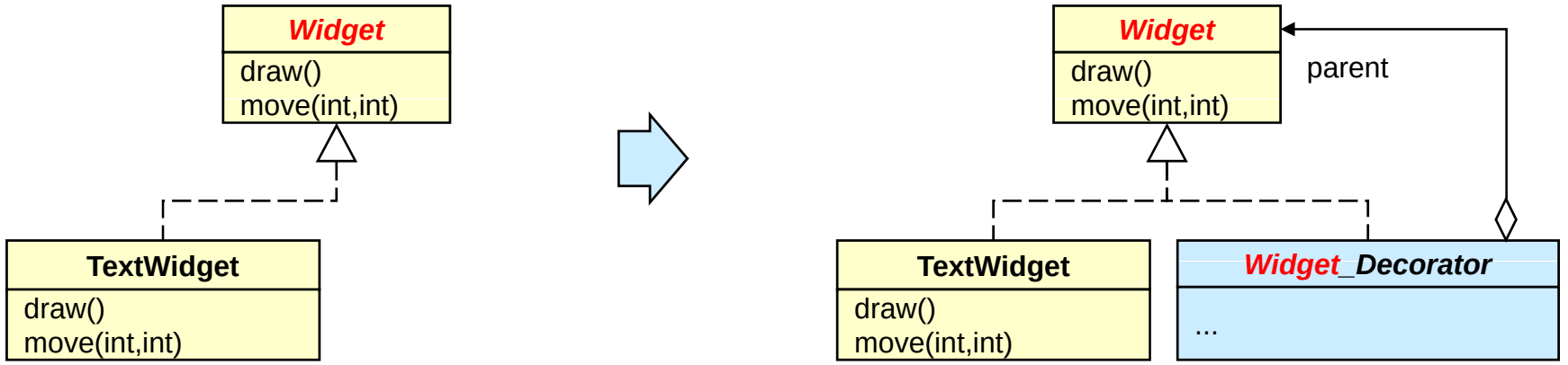


1. Representing Roles by Logic Variables



2. Creating a Class for the Decorator Role

- Generic type introduction
  - introduced type represented by a logic variable
  - ... determined by a pointcut
  - ... can depend on yet unknown program elements

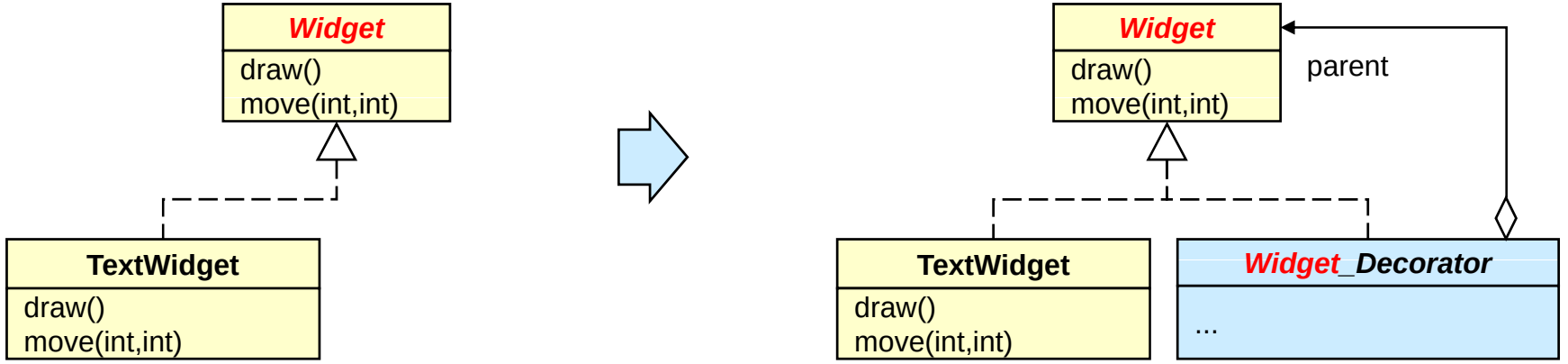


2. Creating a Class for the Decorator Role

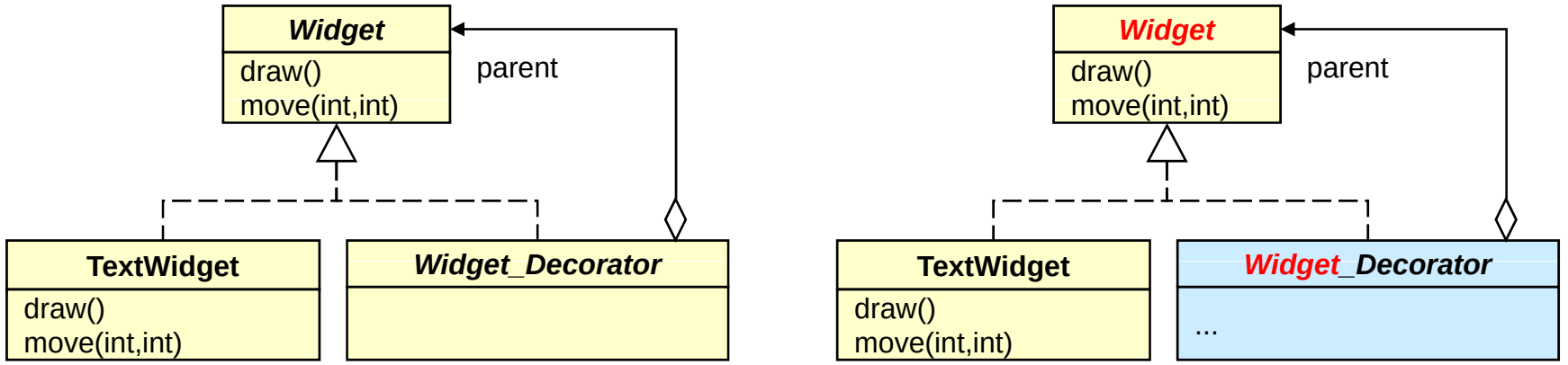
values of logic variables from pointcut

... are used in introduction

```
abstract aspect AbstractDecorator {

    ...

    introduce(?component,?decorator):
        decorating(?component, ?decorator) {
            abstract class ?decorator implements ?component {
                protected ?component parent;
            }
    }
}
```

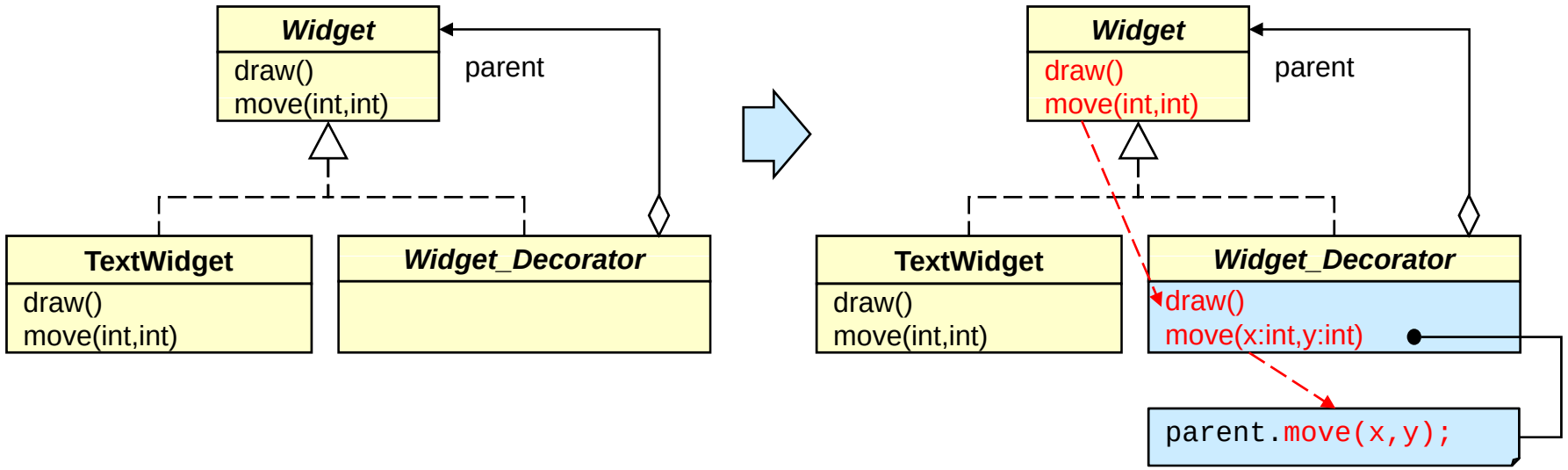


3. Adding Forwarding Methods



3. Adding Forwarding Methods

- Generic method introduction
  - elements of introduced methods represented by logic variables
  - ... determined by pointcuts
  - ... can depend on yet unknown program elements

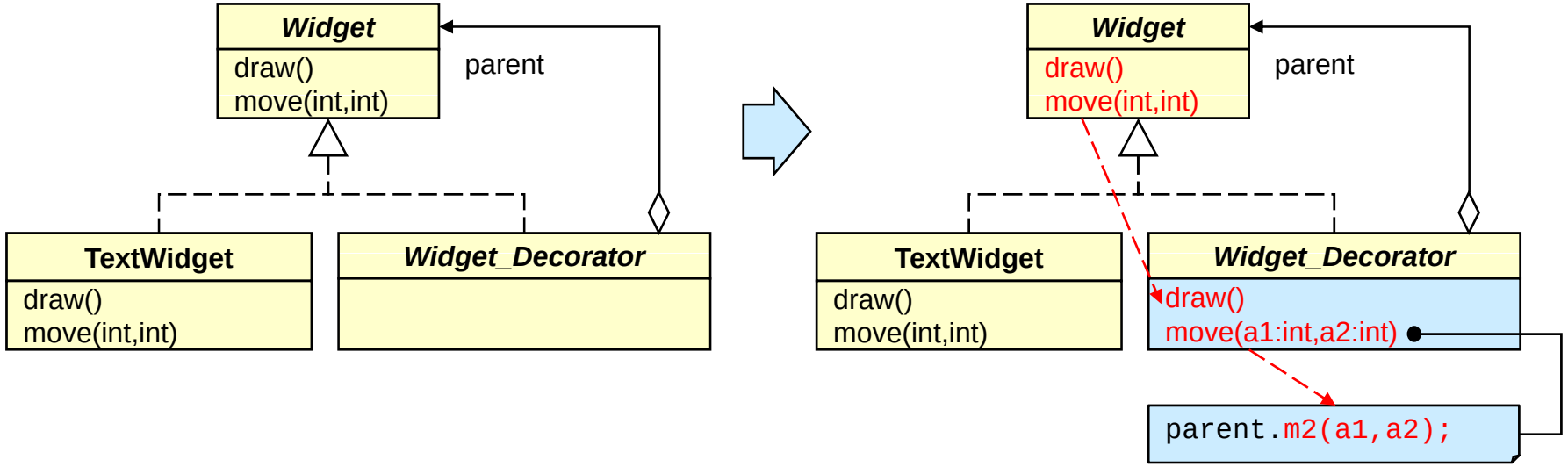


3. Adding Forwarding Methods

values of logic variables from pointcuts

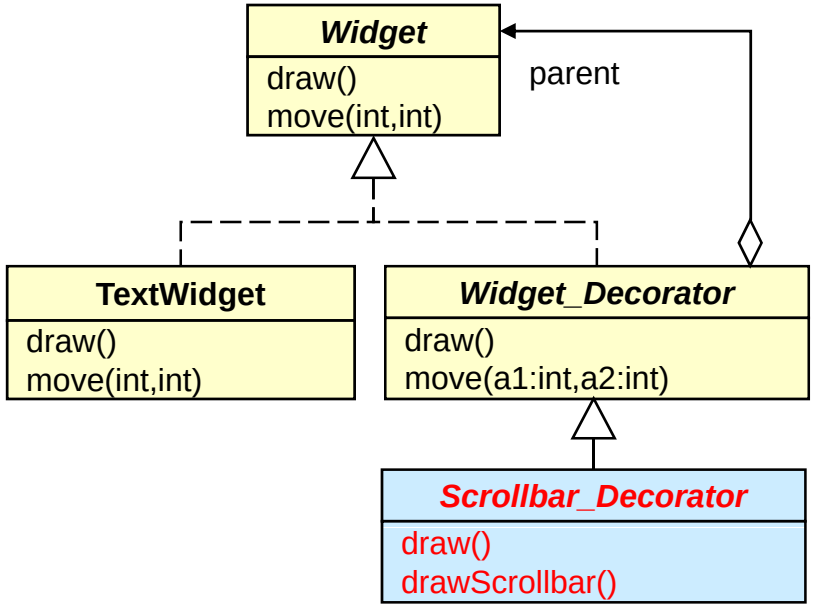
... are used in introduction

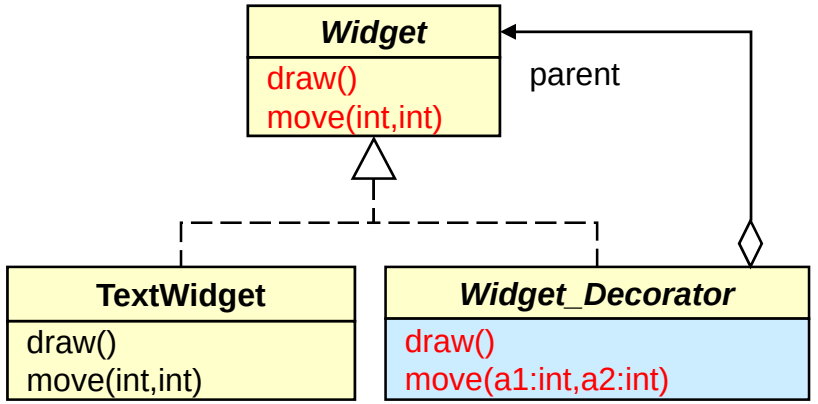
```
abstract aspect AbstractDecorator {
    ...
    introduce(?decorator, ?rettype, ?methodName, ?params) :
        decorating(?component, ?decorator) &&
        method(public ?rettype ?component.?methodName(?params)) {
            ?rettype ?decorator.?methodName(?params) {
                parent.?methodName(?params);
            }
    }
}
```



4. Creating Concrete Decorators

- Concrete Aspect
  - definition of concrete pointcuts
  - introduction of concrete decorator class(es)





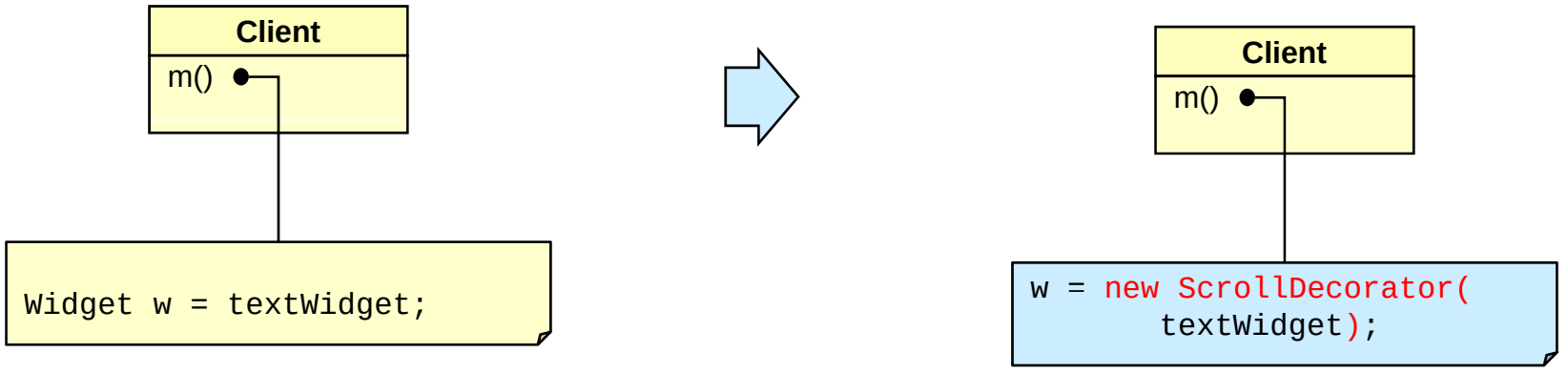
4. Creating Concrete Decorators

```
abstract aspect AbstractDecorator {

     abstract pointcut component(?component);

     ...
```

```
aspect WidgetDecoratorAspect extends AbstractDecorator {

     pointcut component(?component):
          equals(?component, Widget);

     introduce(?decorator) : decorating(?component,?decorator) {
          class ScrollbarDecorator extends ?decorator {

               public void draw() {
                    drawScrollbar();
                    super.draw();
               }
               void drawScrollbar() {
                    // ...
               }
          }
     }

        ...
```

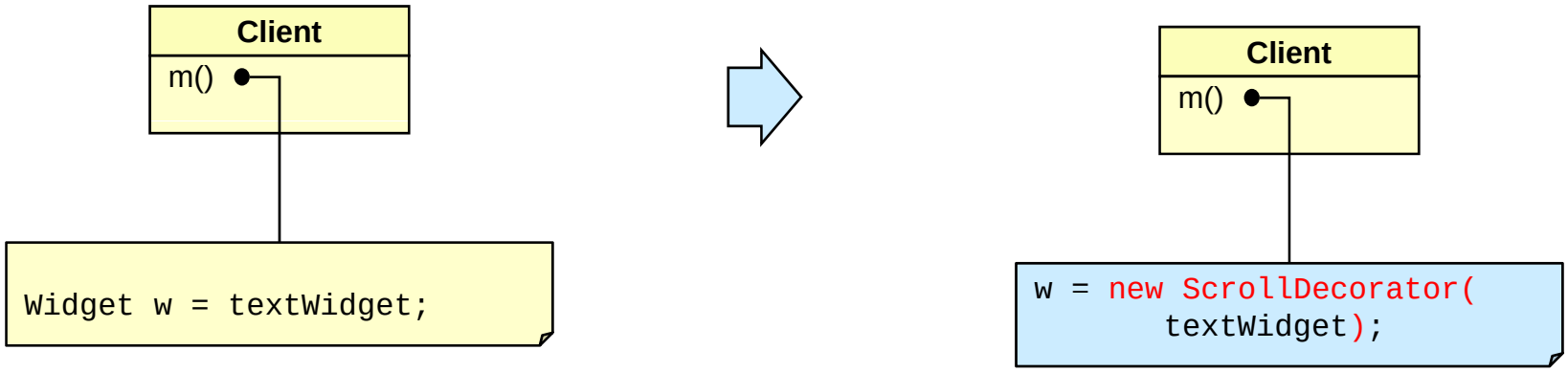
5. Using Decorators in Client Code

- Generic advice
  - elements of advice represented by logic variables
  - ... determined by pointcuts
  - ... can depend on yet unknown program elements



5. Using Decorators in Client Code

```
aspect WidgetDecoratorAspect extends AbstractDecorator {

     pointcut component(?component): equals(?component, Widget);

     introduce(?decorator) : decorating(?component,?decorator)...

     void around(?component comp, ?decorator) :
          decorating(?component,?decorator) &&
          set(?component *.*) && args(comp) {

          if (comp instanceof ?decorator)
               proceed(comp);
          else
               proceed(new ScrollDecorator(comp));
     }
}
```

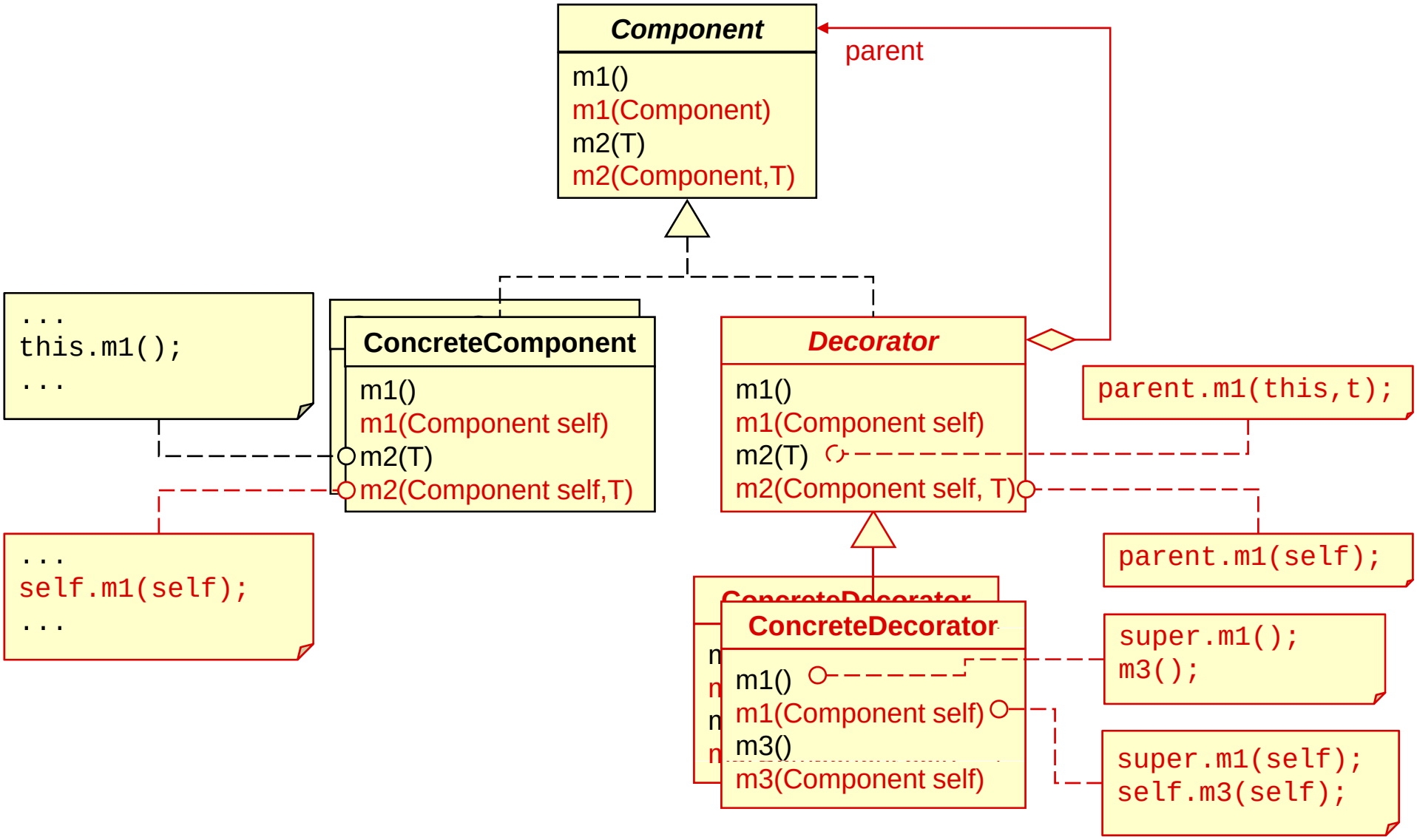


Decorator Variation – Back Reference

"Decorator with object-based inheritance"

- Add back reference to each method in Component
  - Create duplicate method with additional "back reference" parameter
- Change calls on "this" to calls on the back reference
  - Redirection of invocations on "this" lets the "context object" indicated by the back reference override definitions of the current object.

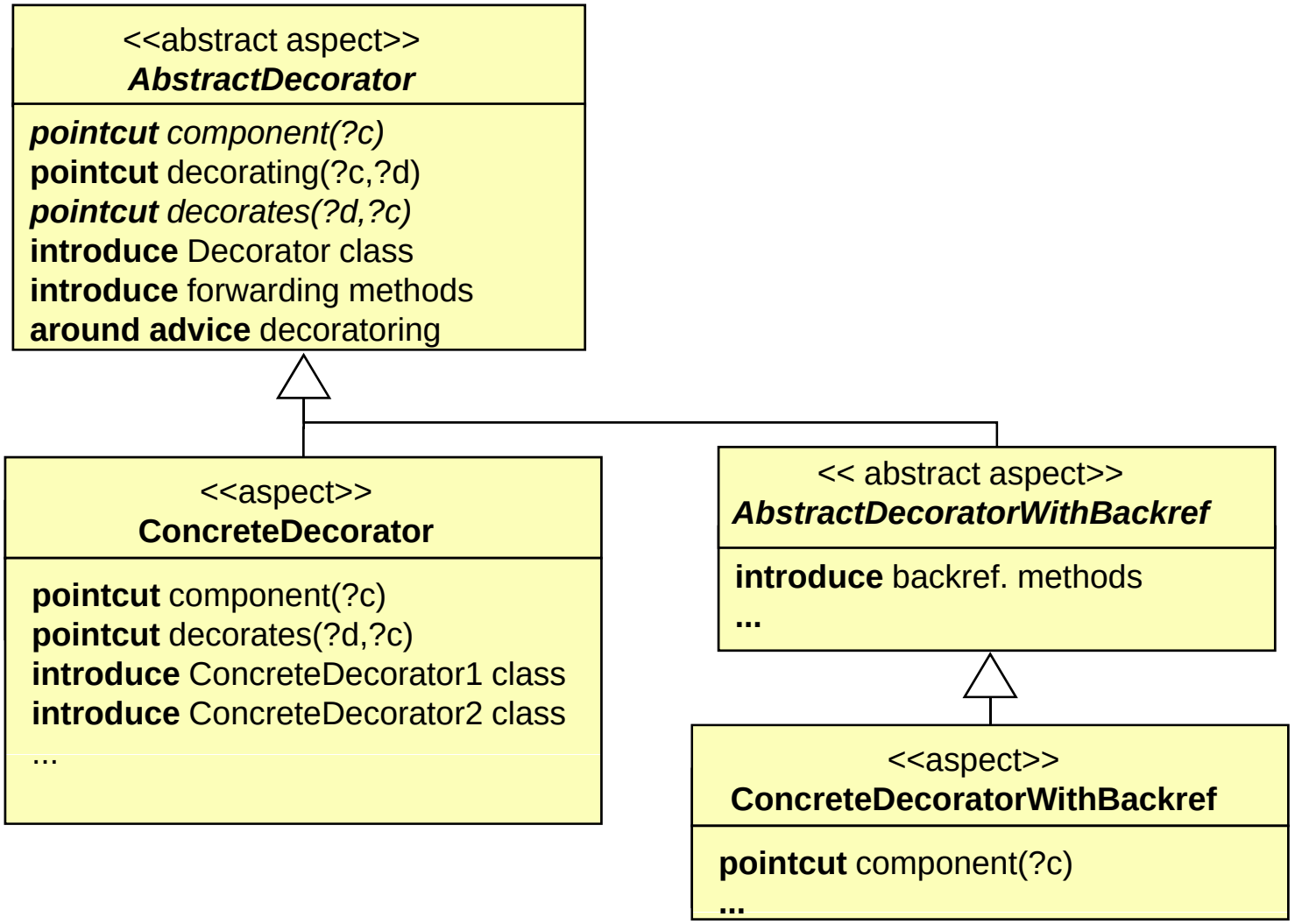
Decorator Variation – Back Reference



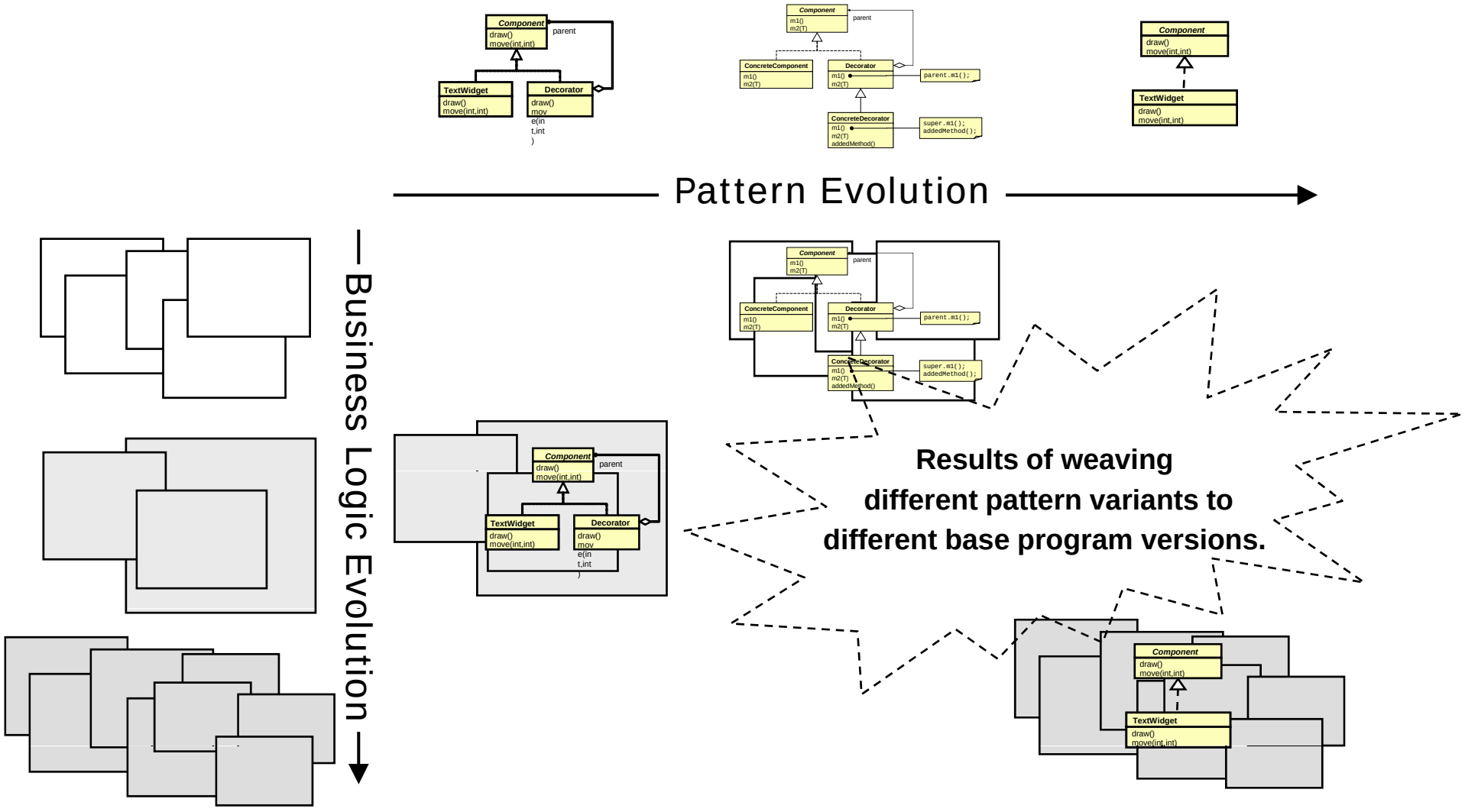
Decorator Variation – Back Reference

- For details see:
  - Günter Kniesel, Tobias Rho, Stefan Hanenberg, *Evolvable Pattern Implementations Need Generic Aspects*, Proc. of ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution
  - http://roots.iai.uni-bonn.de/research/logicaj/downloads/Kniesel%20Rho%20Hanenberg%20-%20RAM-SE04.pdf

Pattern Variations as Generic Aspects



Independent Evolution of Core Logic and DP



Conclusion

- Problem
  - tangled OO implementation of core logic and DPs
- Observation
  - Patterns describe collaborations → aspects
  - Patterns are inherently generic → logic variables
- Solution
  - Pattern implementations as uniformly generic aspects in LogicAJ
    - modular
    - reusable
    - evolvable