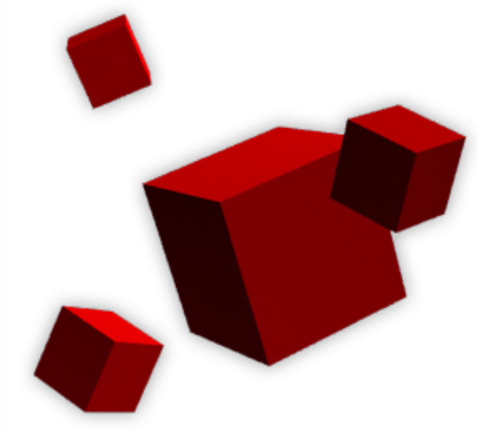


Modularity beyond inheritance

Alexandre Bergel
RMoD team, INRIA,
Lille, France
alexandre@bergel.eu

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Goal of this lecture

- To introduce research problems related to class-inheritance
- Present 2 state-of-the-art research topics related to class inheritance

Sources & references

- Wirfs-Brock & McKean, *Object Design — Roles, Responsibilities and Collaborations*, 2003.
- Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz, *Classbox/J: Controlling the Scope of Change in Java*, OOPSLA'05
- Damien Cassou, Stéphane Ducasse and Roel Wuyts, *Traits at Work: the design of a new trait-based stream library*, In Journal of Computer Languages, Systems and Structures, Elsevier, 2008
- Stephane Ducasse, Roel Wuts, Alexandre Bergel, and Oscar Nierstrasz, *User-Changeable Visibility: Resolving Unanticipated Name Clashes in Traits*, OOPSLA'07

Outline

1. Inheritance (single & multiple)
2. Classboxes: inheritance to express software evolution
3. Traits: inheritance to feature composition
4. Concluding words: complementing class inheritance as a major Software Engineering effort

Inheritance

- *Inheritance* in object-oriented programming languages is a mechanism to:
  - *derive new subclasses* from existing classes
  - where subclasses *inherit all the features* from their parent(s)
  - and may *selectively override* the implementation of some features.

Inheritance mechanisms

- OO languages realize inheritance in different ways:
  - *self*: dynamically access subclass methods
  - *super*: statically access overridden, inherited methods
  - *multiple inheritance*: inherit features from multiple superclasses
  - *abstract classes*: partially defined classes (to inherit from only)
  - *mixins*: build classes from partial sets of features
  - *interfaces*: specify method argument and return types
  - *subtyping*: guarantees that subclass instances can be substituted

Classboxes for evolution

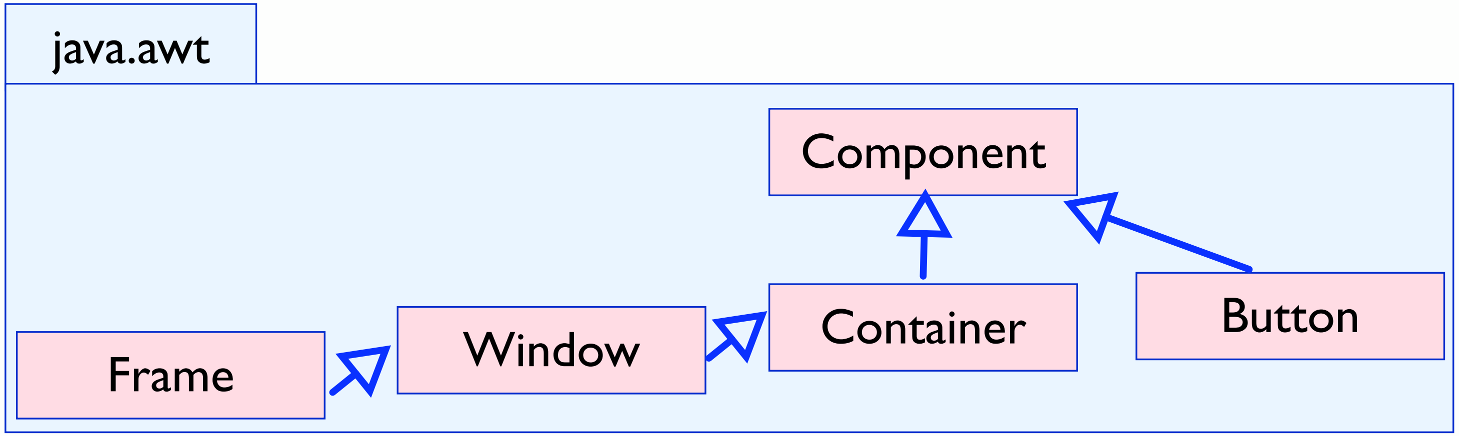
I. *Problem*: AWT and Swing anomalies

II. *Model*: Classbox/J

III. *Solution*: Swing as a classbox

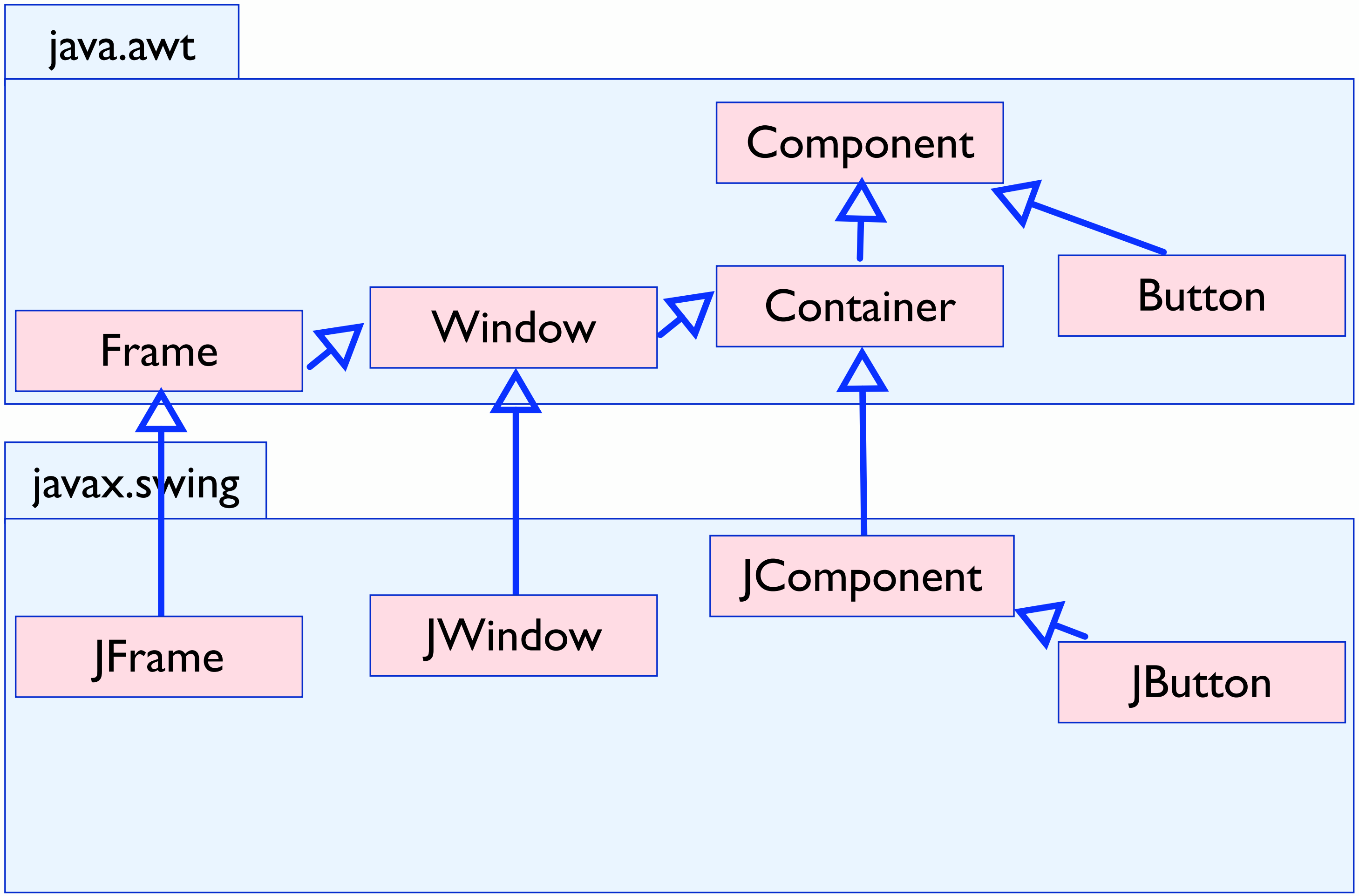
IV. *Ongoing work*: general scoping mechanism

Presentation of AWT

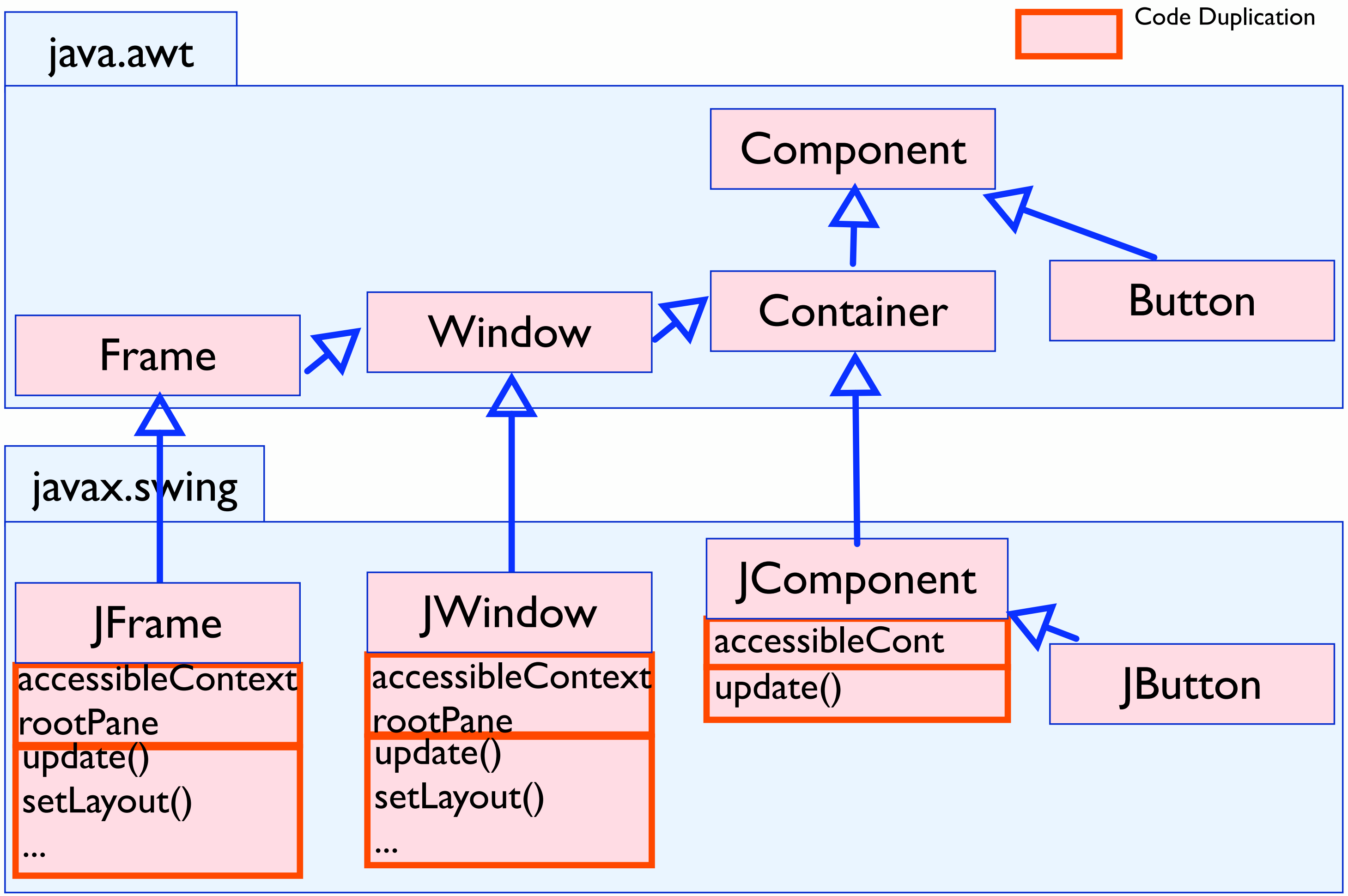


- In the AWT framework:
  - Widgets are components (i.e., inherit from Component)
  - A frame is a window (Frame is a subclass of Window)

Broken Inheritance in Swing



Problem: Code Duplication



Problem: explicit type operation

```
public class Container extends Component {
  Component components[] = new Component [0];
  public Component add (Component comp) {...}
}

public class JComponent extends Container {
  public void paintChildren (Graphics g) {
    for (; i>=0 ; i--) {
      Component comp = getComponent (i);
      isJComponent = (comp instanceof JComponent);
      ...
      ((JComponent) comp).getBounds();
    }
  }}
```

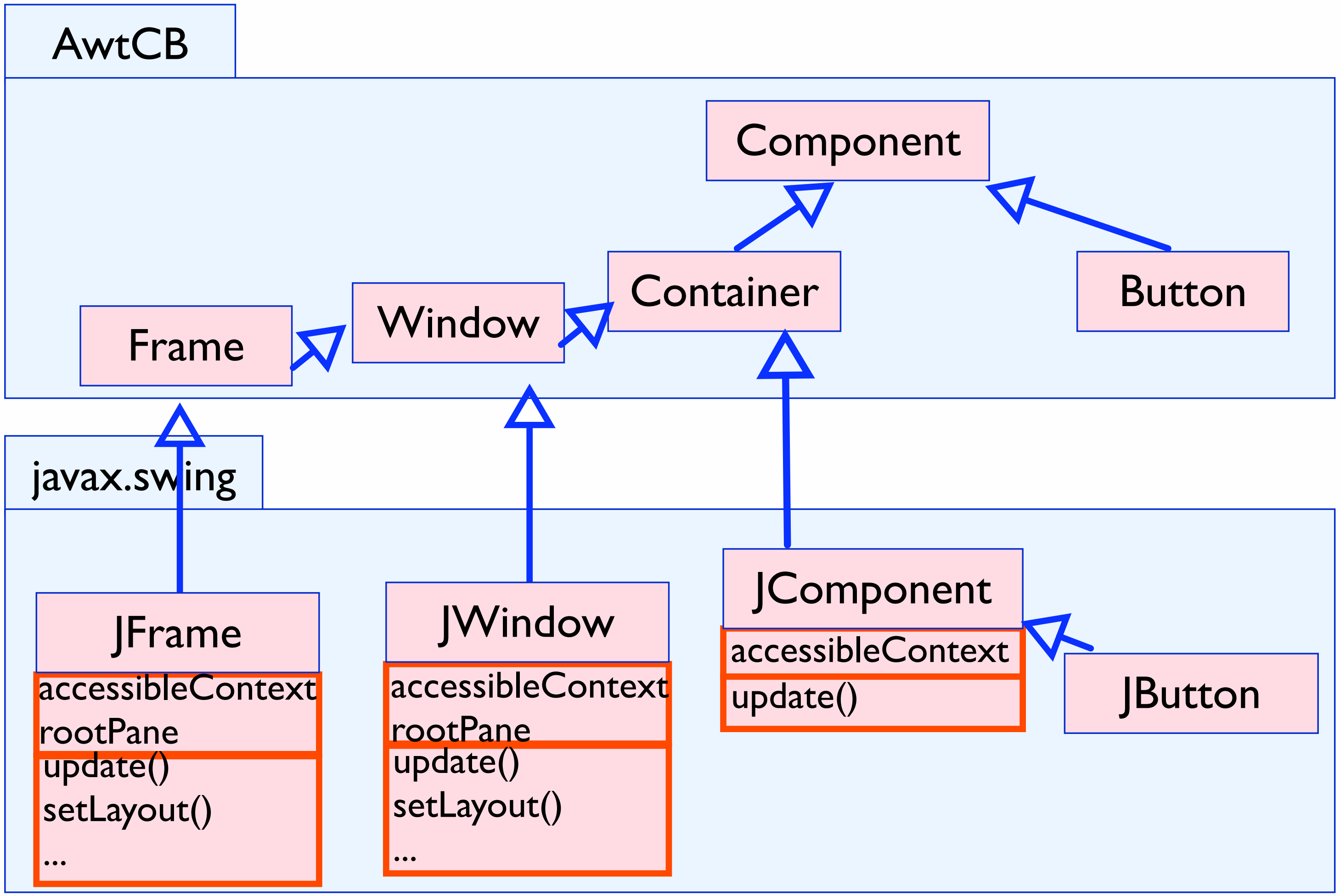
Alternative to inheritance

- *AWT couldn't be enhanced* without risk of breaking existing code.
- Swing is, therefore, *built on the top of AWT using subclassing*.
- As a result, *Swing is a big mess* internally!
- We *need an alternative to inheritance* to support unanticipated changes.

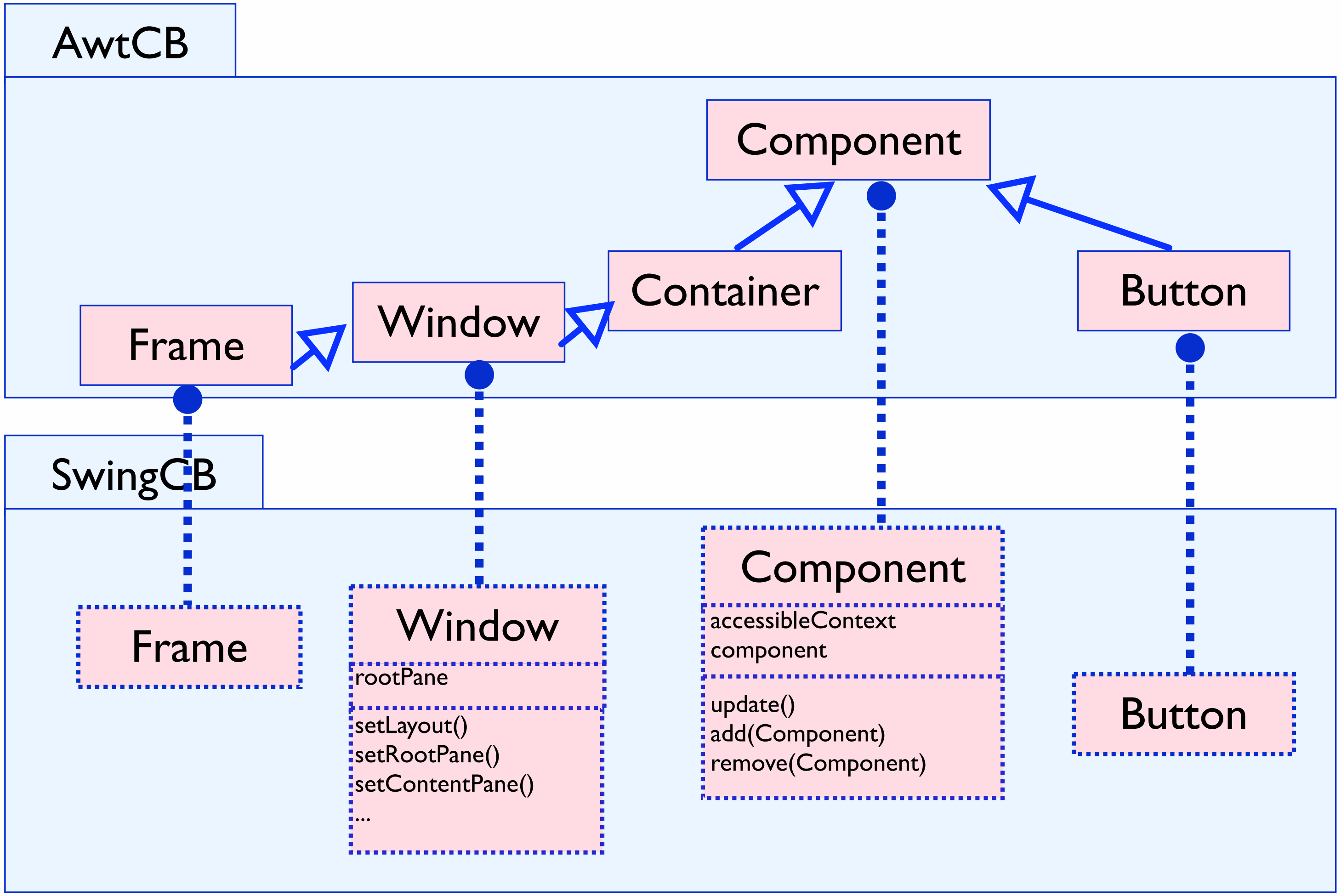
Classbox/J

- Module system for Java allowing classes to be refined without breaking former clients.
- A *classbox is like a package* where:
  - a class defined or imported within a classbox p can be imported by another classbox (*transitive import*).
  - class members can be added or redefined on an imported class with the keyword *refine*.
  - a refined method can access its original behavior using the *original* keyword

Swing Refactored as a Classbox



Swing Refactored as a Classbox



Swing Refactoring

- 6500 lines of code *refactored* over 4 classes.
- Inheritance defined in AwtCB is fully preserved in SwingCB:
  - In SwingCB, *every widget is a component* (i.e., inherits from the extended AWT Component).
  - The property "*a frame is a window*" is true in SwingCB.
- *Removed duplicated code*: the refined Frame is 29 % smaller than the original JFrame.
- Explicit type checks like *obj instanceof JComponent* and *(JComponent)obj* are *avoided*.

Properties of Classboxes

- Minimal extension of the Java syntax (transitive import, *refine* and *original* keywords).
- *Refinements are confined* to the classbox that define them and to classboxes that import refined classes.
- Method redefinitions have *precedence* over previous definitions.
- Classes can be refined *without risk of breaking* former clients.

Traits

I. *Problem*: Stream in Squeak anomalies

II. *Model*: Traits

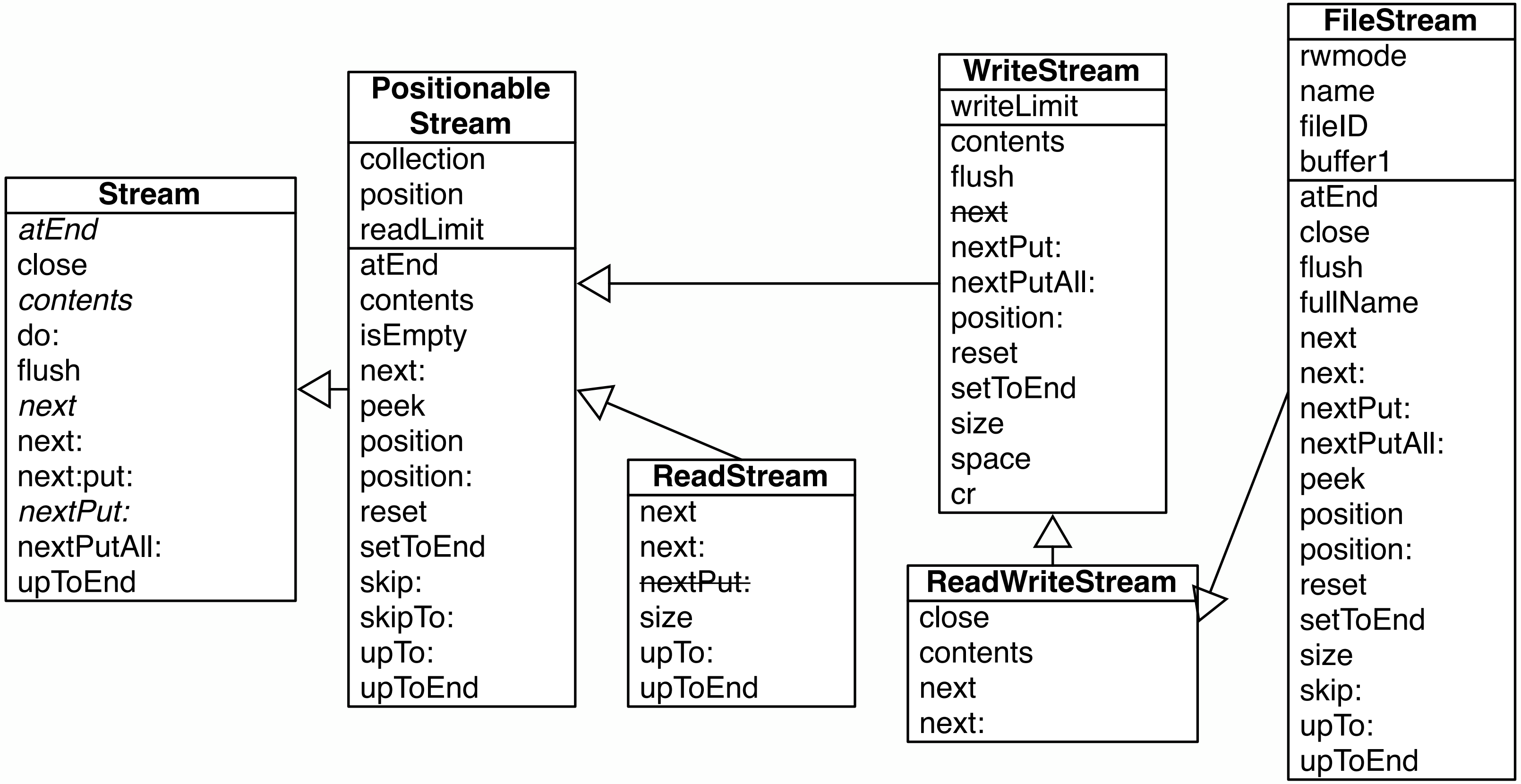
III. *Solution*: Refactoring with Traits

IV. *Ongoing work*: Pure trait language

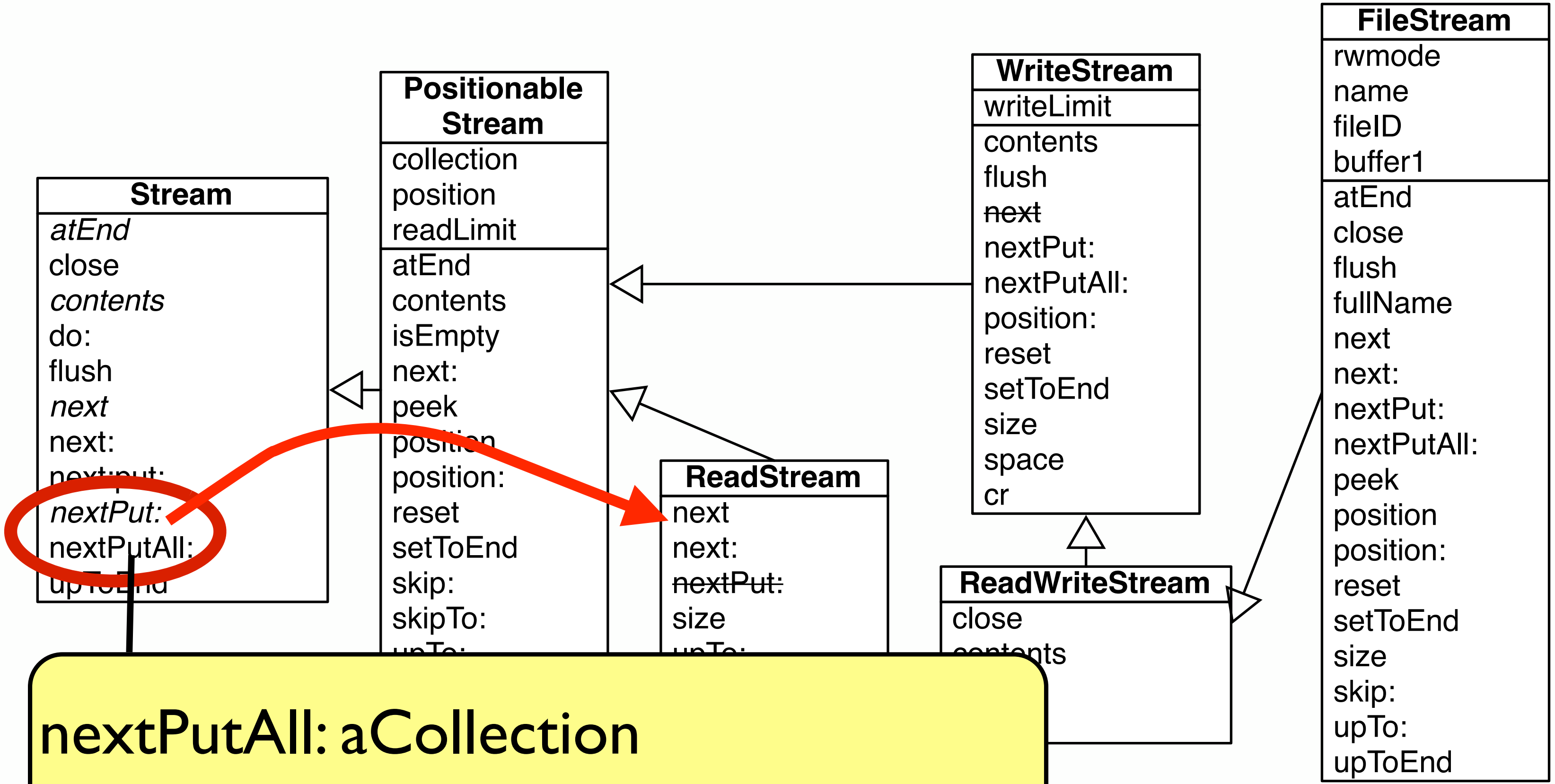
Stream in Squeak

- Example of a library that has been in use for almost 20 years
- Contains many flaws in its conception

Stream in Squeak



Methods too high

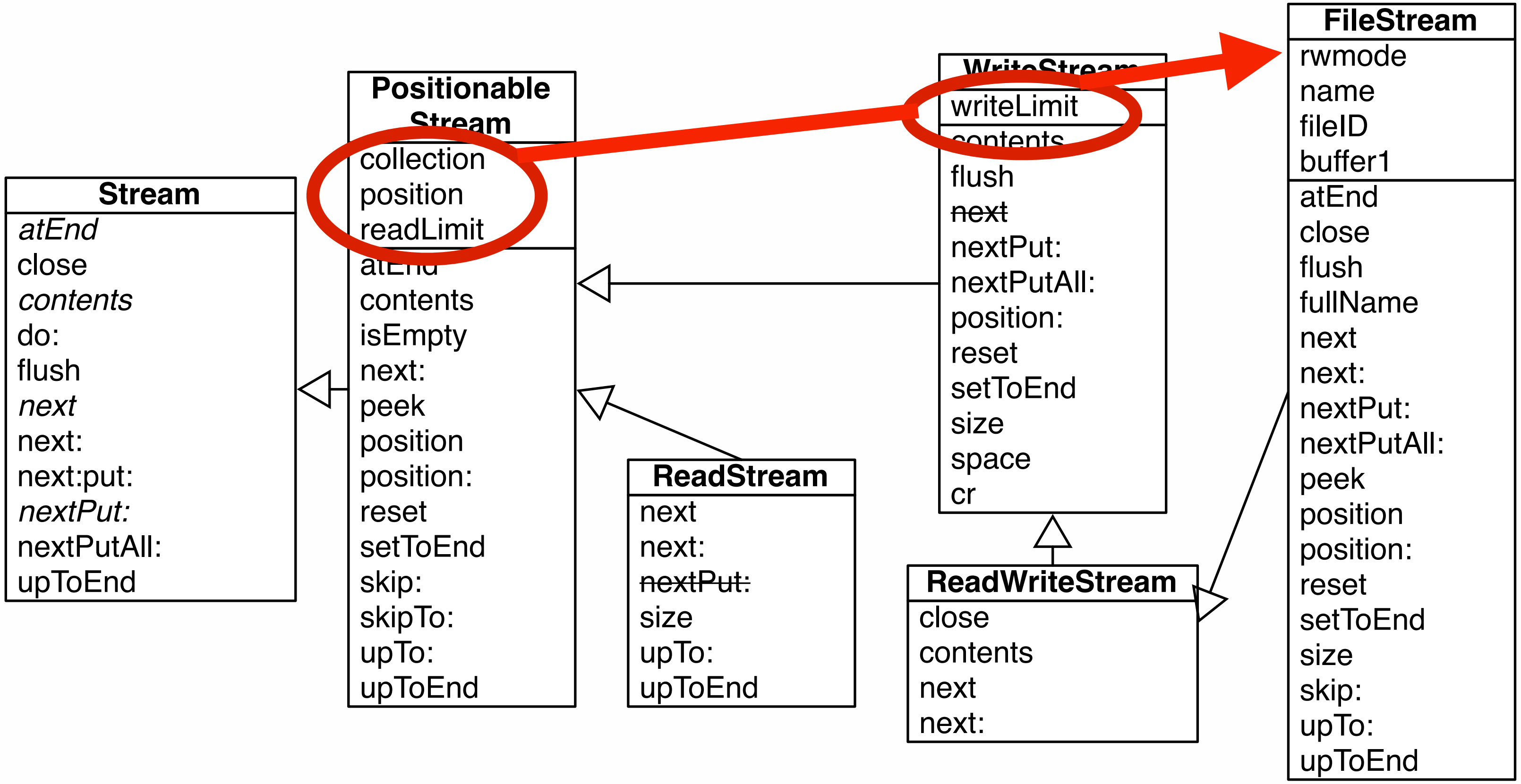



```
nextPutAll: aCollection
   aCollection do: [:v| self nextPut: v].
   ^ aCollection
```

Methods too high

- The *nextPut:* method defined in Stream allows for element addition
- The ReadStream class is read-only
- It therefore needs to "cancel" this method by redefining it and throwing an exception

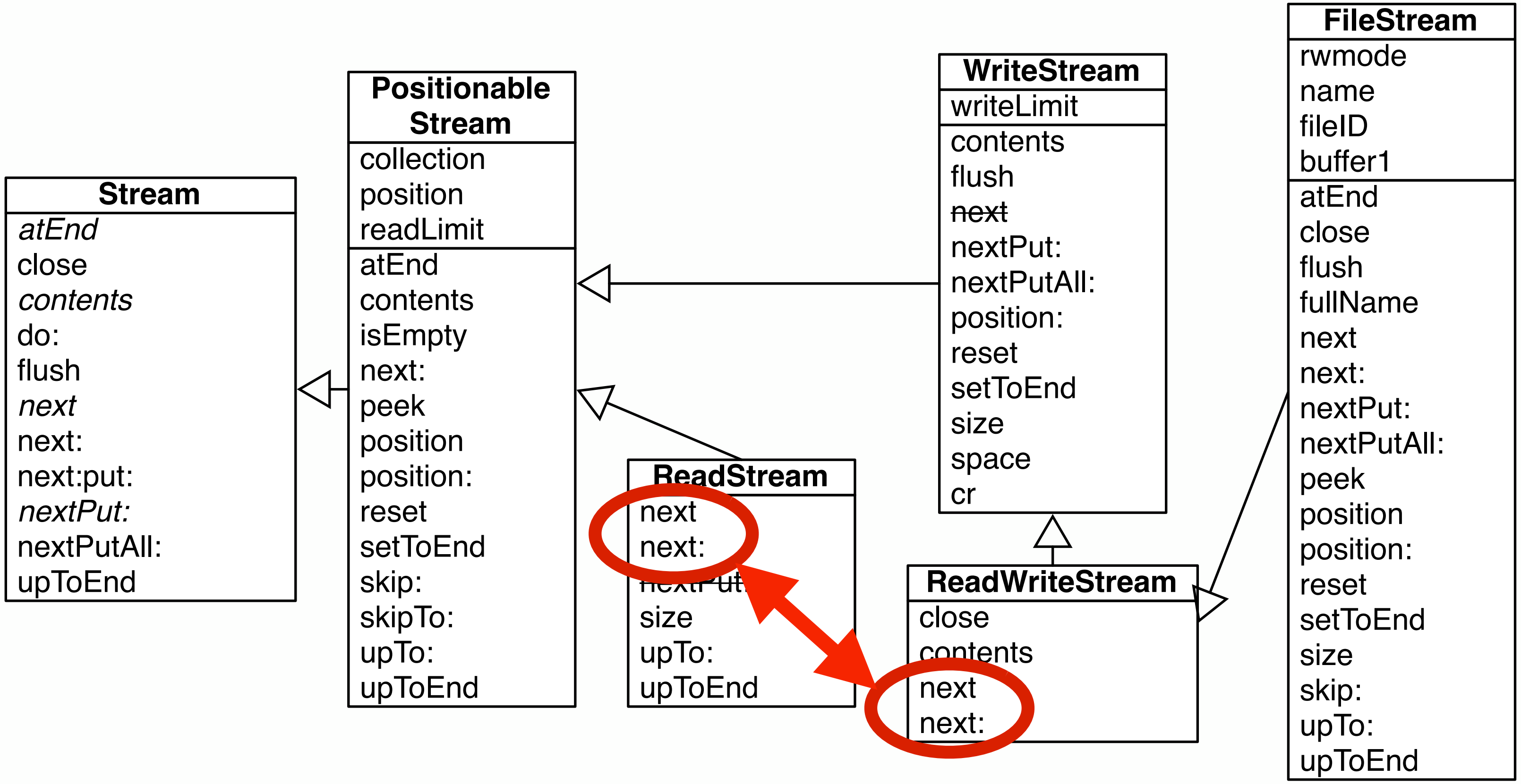
Unused state



Unused state

- State defined in the super classes are becoming irrelevant in subclasses
- FileStream does not use inherited variables

Multiple inheritance simulation



Multiple inheritance

- Methods are duplicated among different class hierarchies

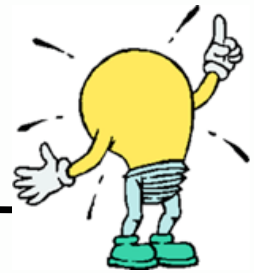
Class responsibilities

- Too many responsibilities for classes
  - object factories
  - group methods when subclassing

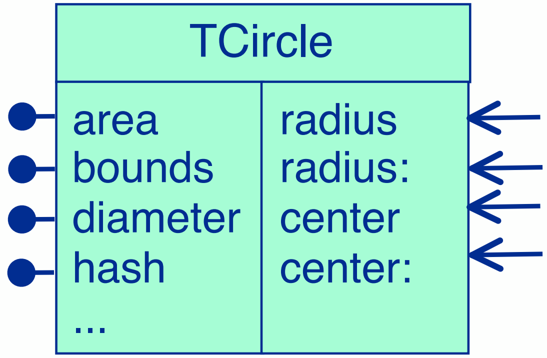
Class schizophrenia?

- Too many responsibilities for classes
  - object factories => *need for completeness*
  - group methods when subclassing => *need to incorporate incomplete fragments*

Traits

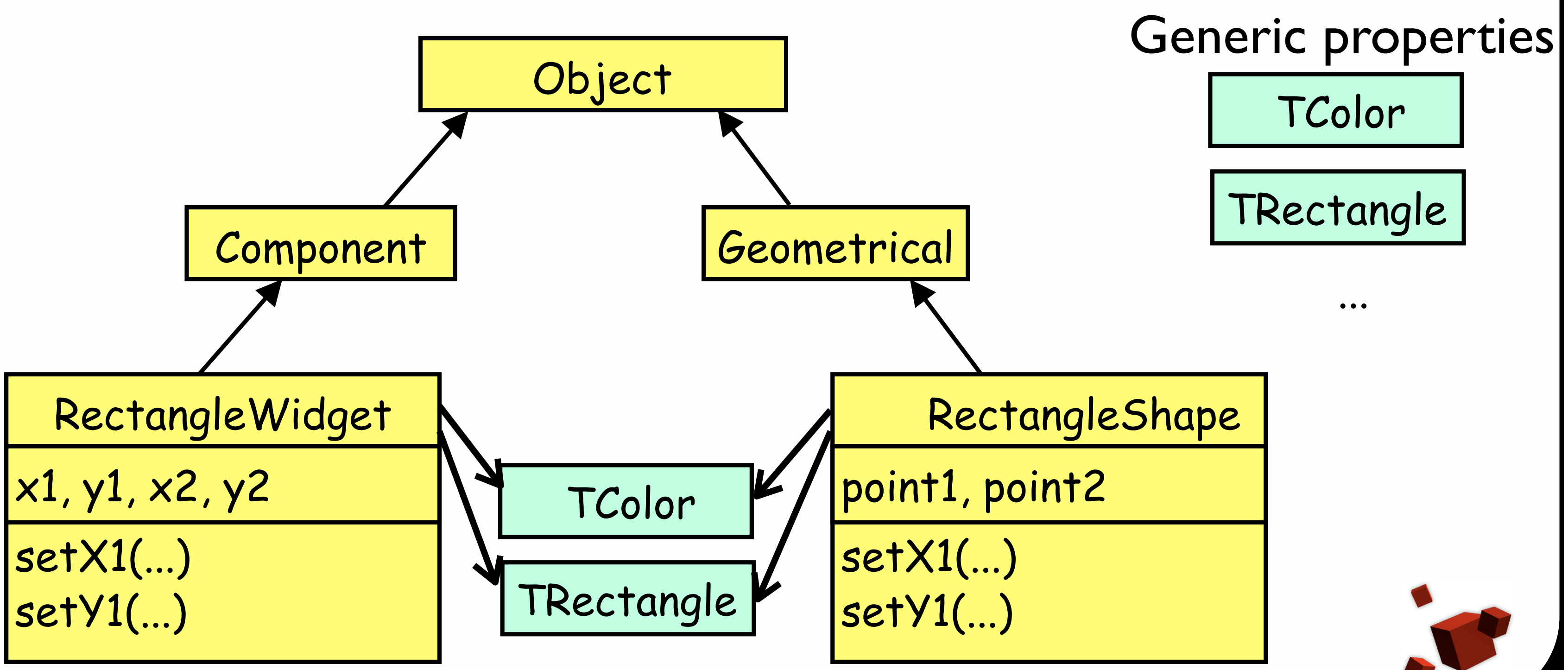


- Traits are parameterized behaviors
  - traits *provide* a set of methods
  - traits *require* a set of methods
  - traits are purely *behavioral*



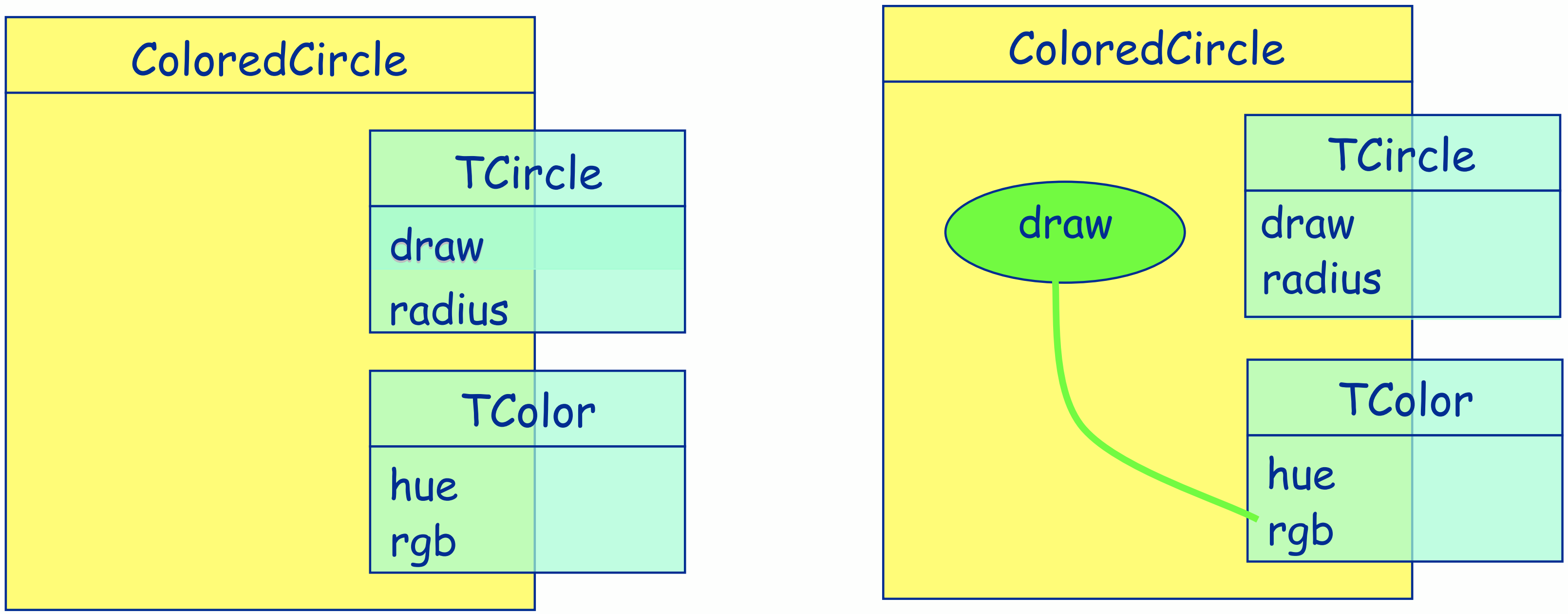
Class = Superclass + State + Traits + Glue Methods

- Traits are the behavioral building blocks of classes



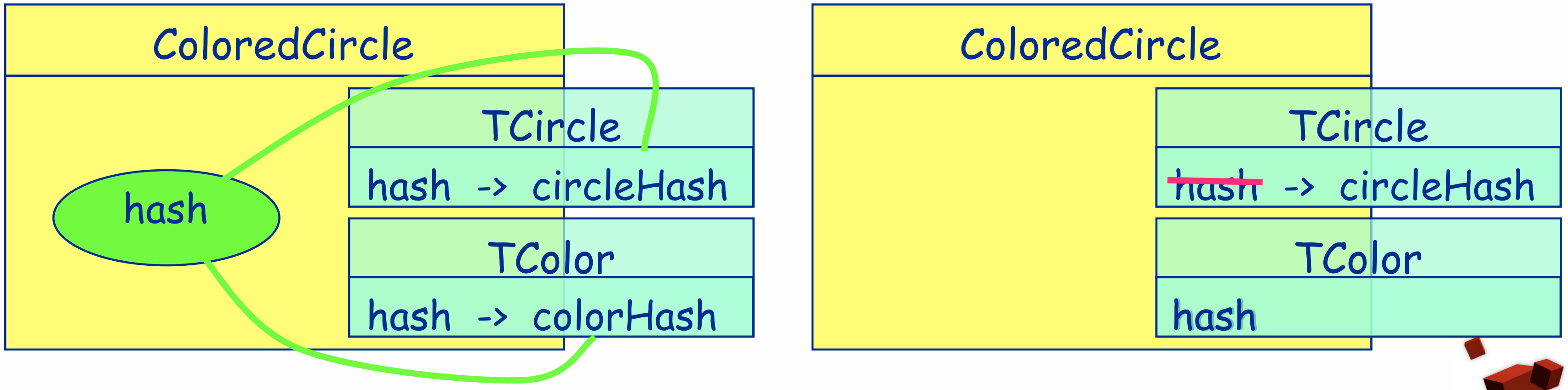
Composition rules

Class methods take precedence over trait methods

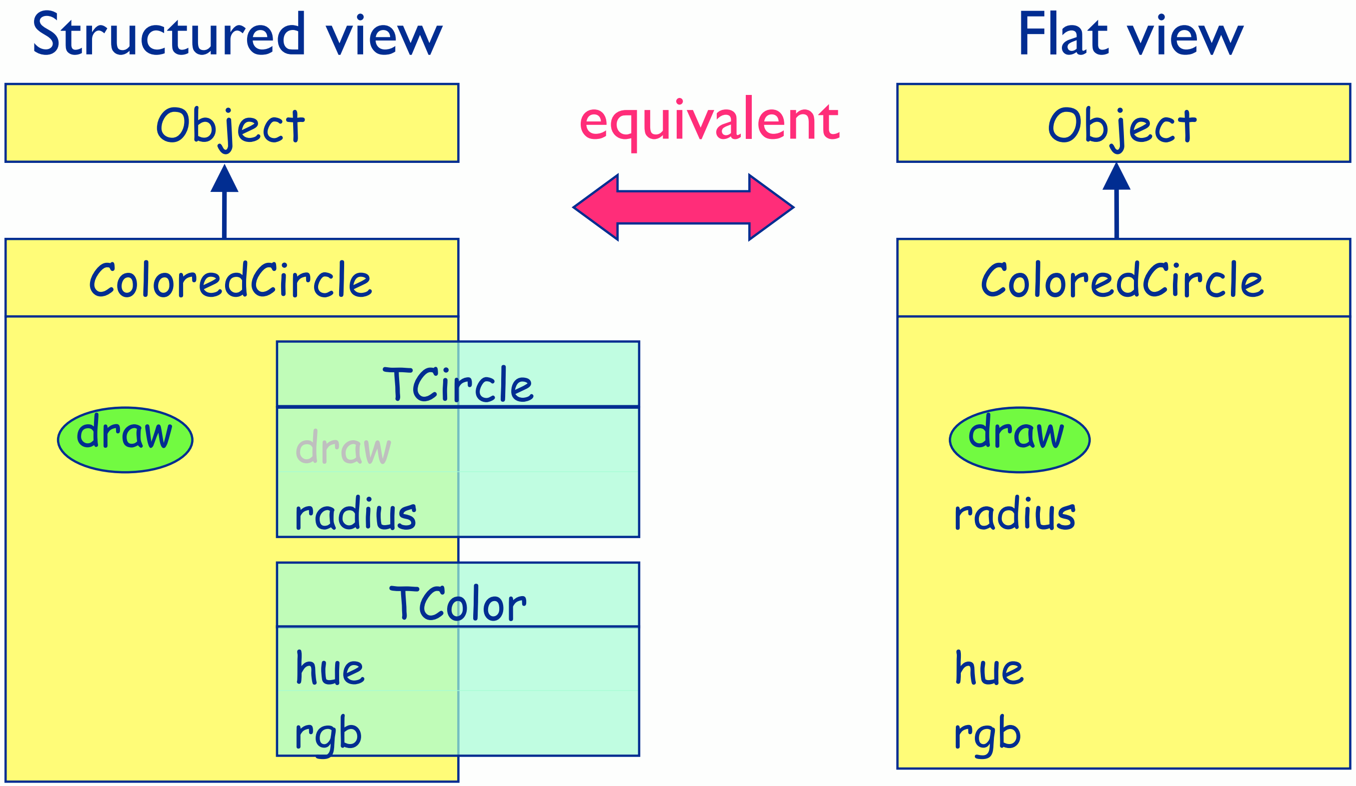


Conflicts are explicitly resolved

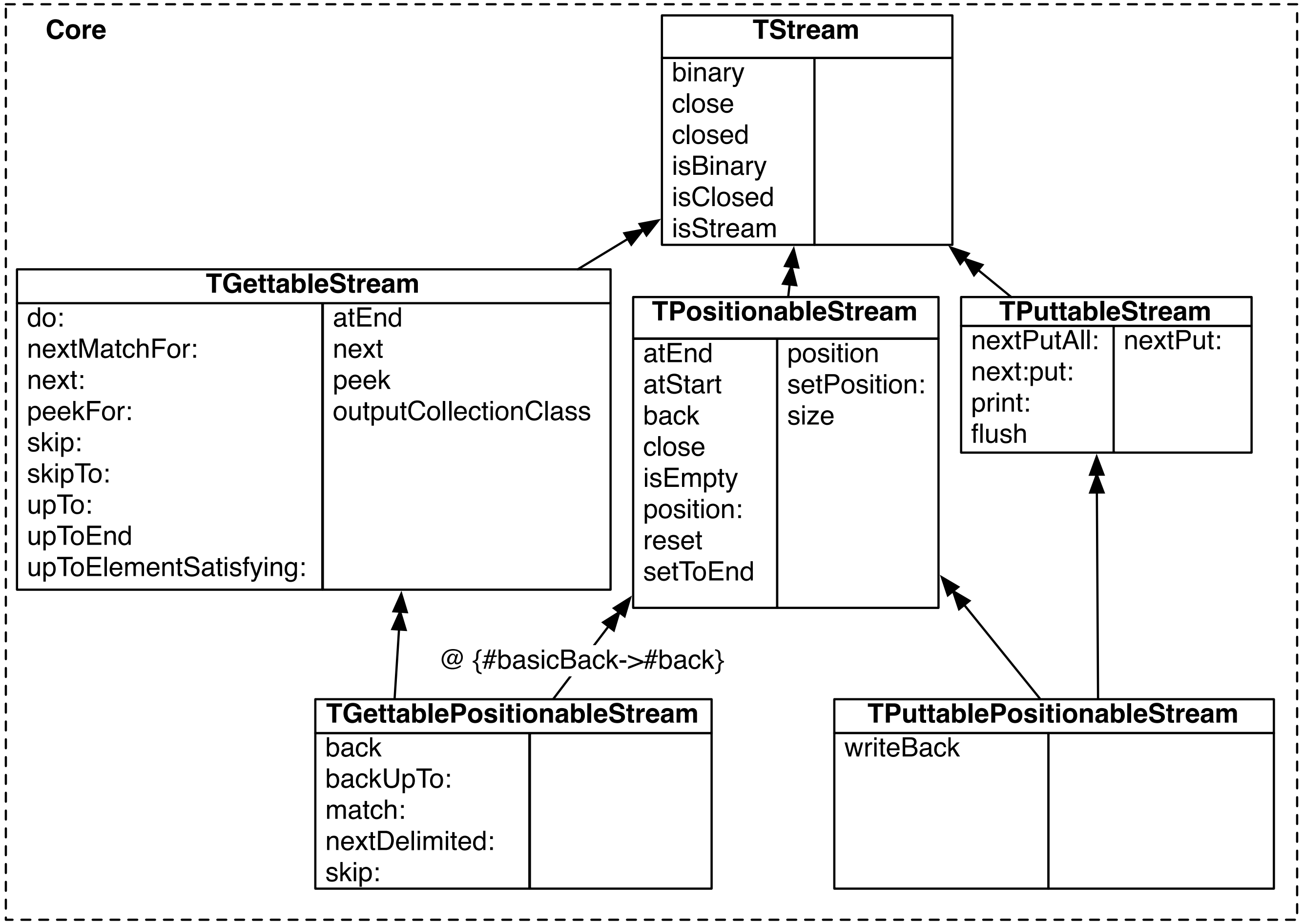
- Override the conflict with a glue method
  - Aliases provide access to the conflicting methods
- Avoid the conflict
  - Exclude the conflicting method from one trait



Flattening property



Stream revisited



Concluding words

- Hard topic where proving a better solution requires a significant effort
- Other hot topics: Virtual classes, Nested inheritance, Selector namespaces