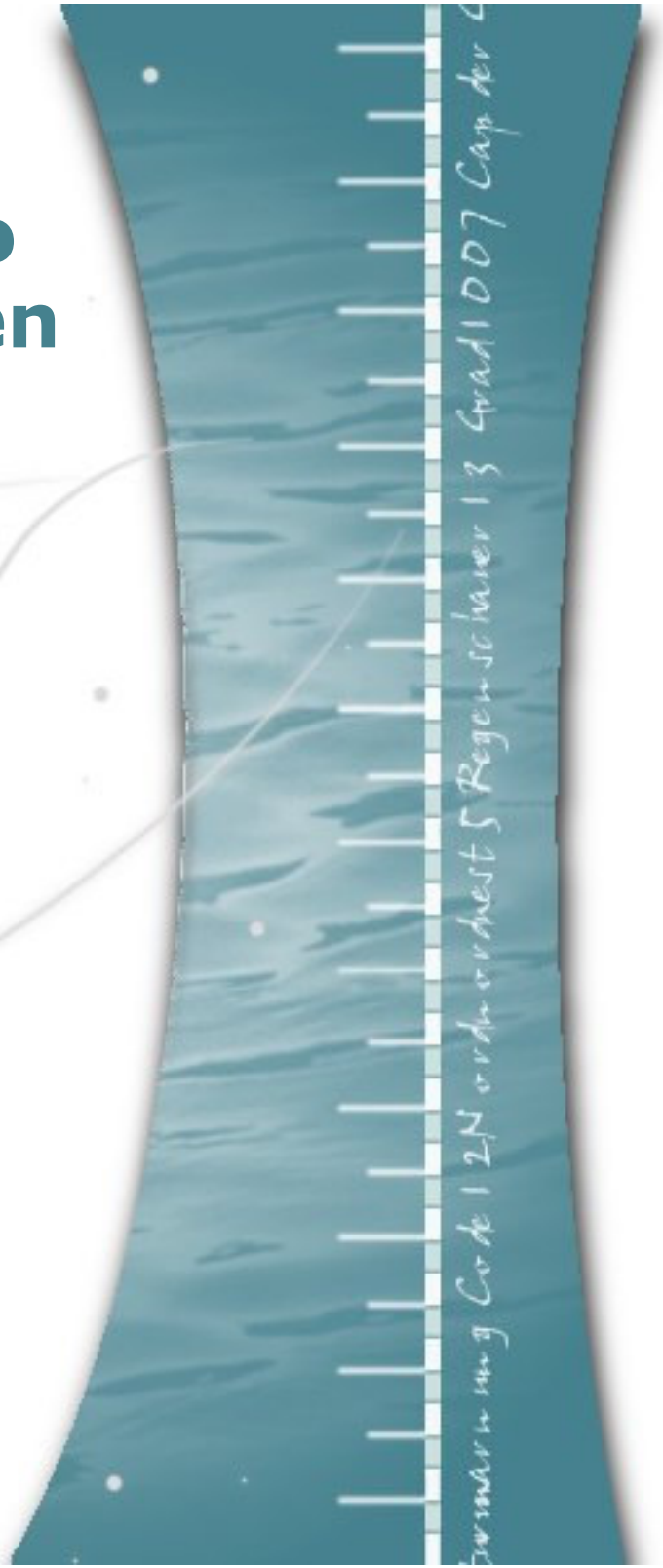
Specifying Design-Rules to improve modularity between OO/AO code with CaesarJ



Carlos Eduardo Pontual
M.Sc. student

Advisor: Paulo Borba

21/09/2009

Aspect-Oriented Software Development

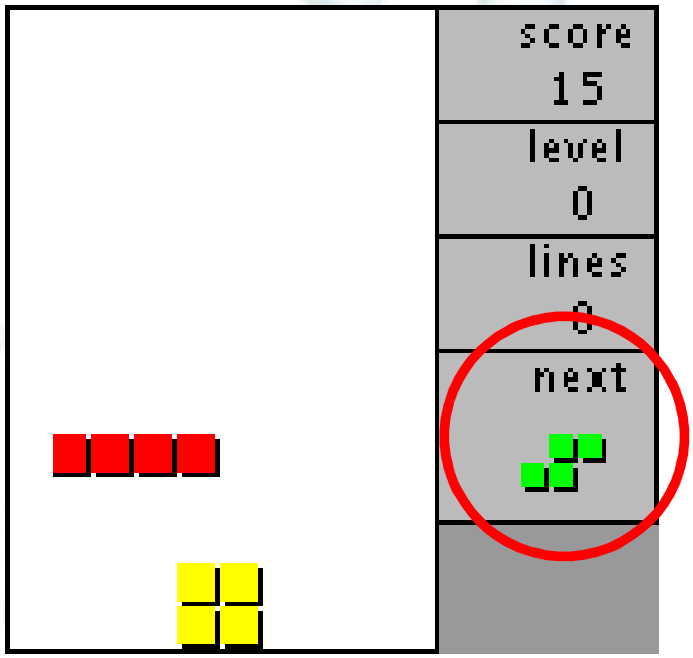
- Better modularize the *crosscutting* concerns
  - Transactional management, Persistence, ...
- However, aspects may break class modularity
  - It's not possible to reason about a class without consider all aspects that may advice this class
  - Envolving a class might break the intents of an Aspect
  - Programmers are not able to write the aspects until the related classes have been implemented
    - No parallel development of classes and aspects

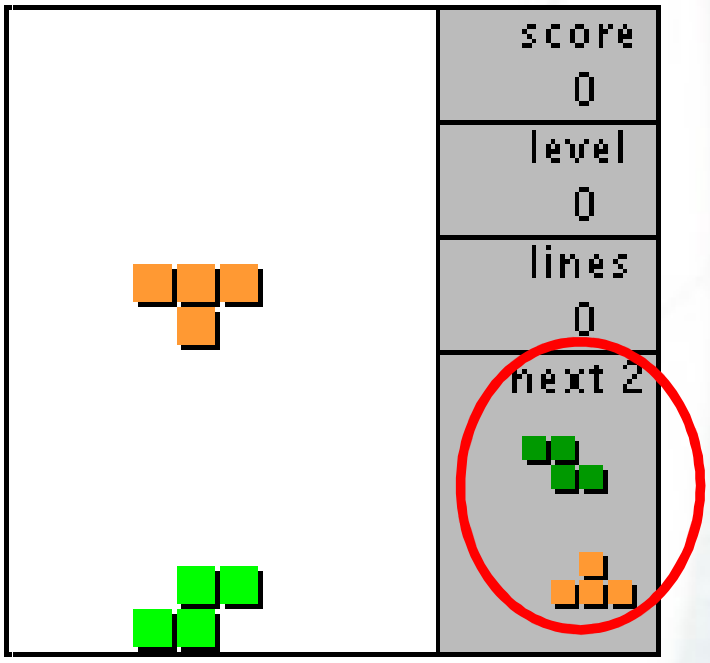
Improving Modularization OOxAO

- We need a brief specification of the relations/restrictions between classes and aspects
- Interfaces (Design-Rules) that enable the parallel development of classes and aspects
  - Guide the developers
  - Enable compiler checking
- Existing solutions (XPIs, Aspect-Aware Interfaces) are not enough for parallel development

Motivating Example

- Simple Tetris games SPL
  - Dificulty variant (easy, normal), among others.





Variation details

GameCanvas

```
NextPiece np;
...
void sideBoxes() { ...
  np.updatePiece(); ... }
void paintCanvas() { ...
  np.updatePiece(); ... }
...
```

NextPiece

```
...
void paint() { ...
  paintBox(); ... }
void drawPiece() {...}
void updatePiece() { ... }
void paintBox() { ... }
...
```

- Two versions of the following methods of *NextPiece:*
  - paintBox(), which is called by other methods of *NextPiece*
  - updatePiece(), which is called by other classes of the program (e.g. GameCanvas class)
- paintBox() has to access non-variant members of *NextPiece*
  - Bi-directional between base and variations

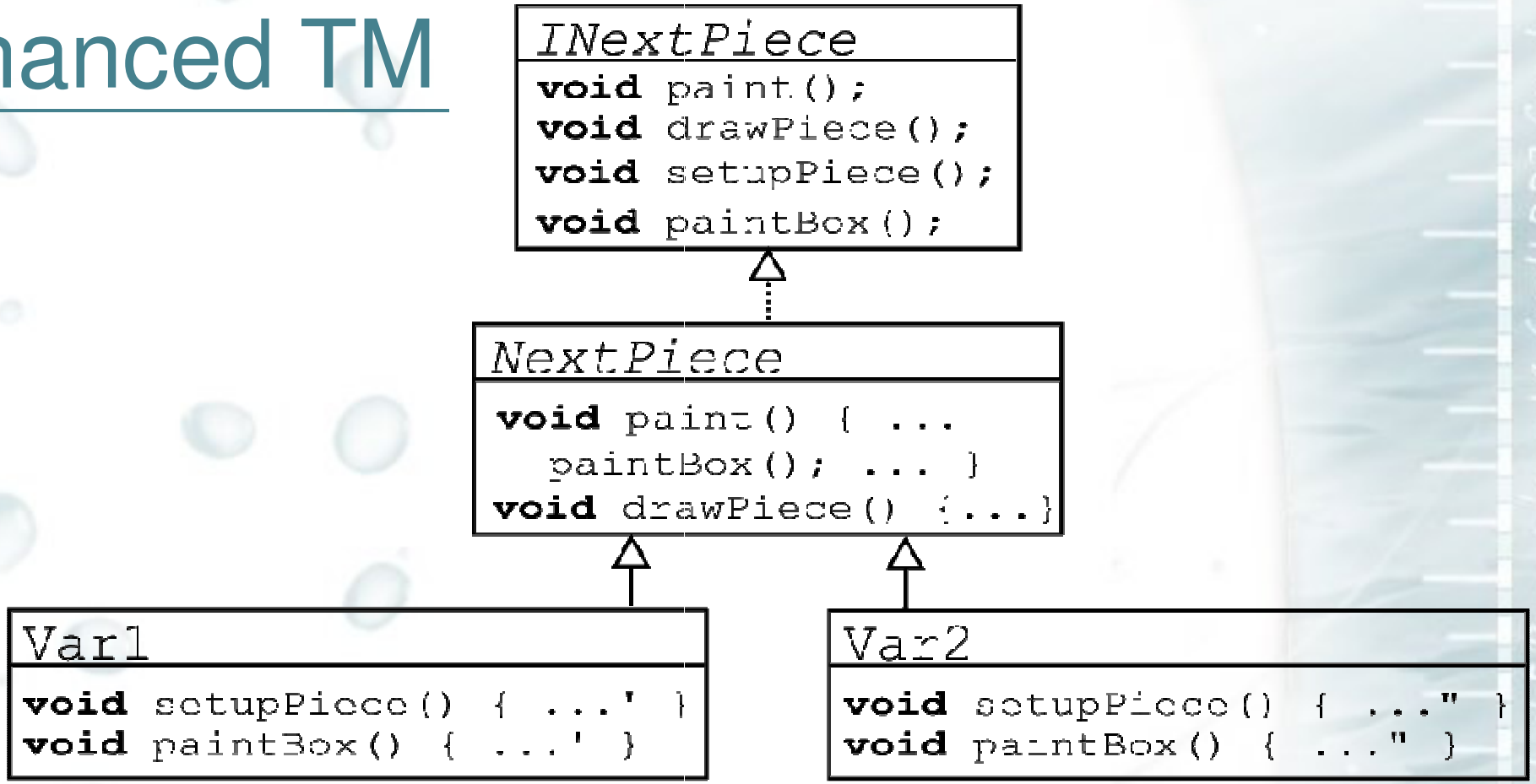
Possible Implementation - OO

```
abstract class NextPiece {
        void paint() { ...
                paintBox(); ... }
        void drawPiece() { ... }
        abstract void updatePiece();
        abstract void paintBox();
}
class Var1 extends NextPiece {
        void updatePiece() { ...' }
        void updateBox() { ...' }
}
```

```
class Var2 extends NextPiece {
        void updatePiece() { ..." }
        void updateBox() { ..." }
}
```

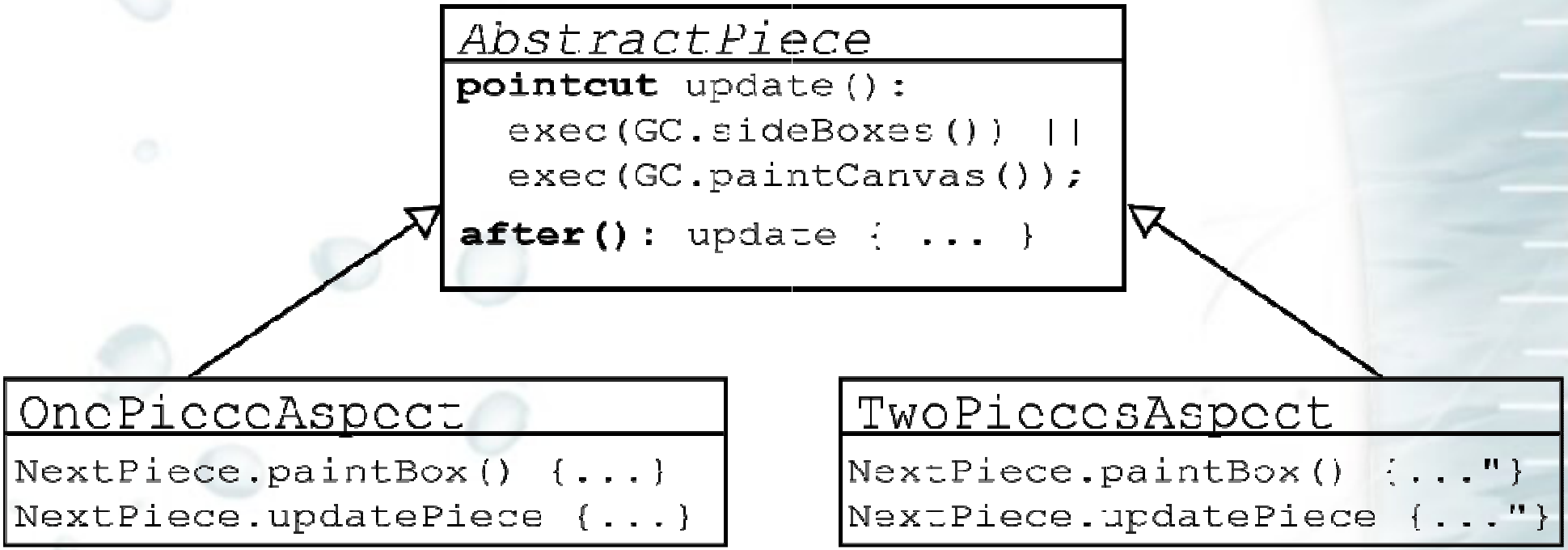
- Tangling of design (abstract signatures) and implementation
- The variation part can only be implemented after the implementation of the base code

Enhanced TM



- Its not clear on the interface which methods are from the base (one team) and which are from the variations
  - Not enough for parallel development
- Variation code can not be compiled independently of the base code (inheritance)

Implementation - AO

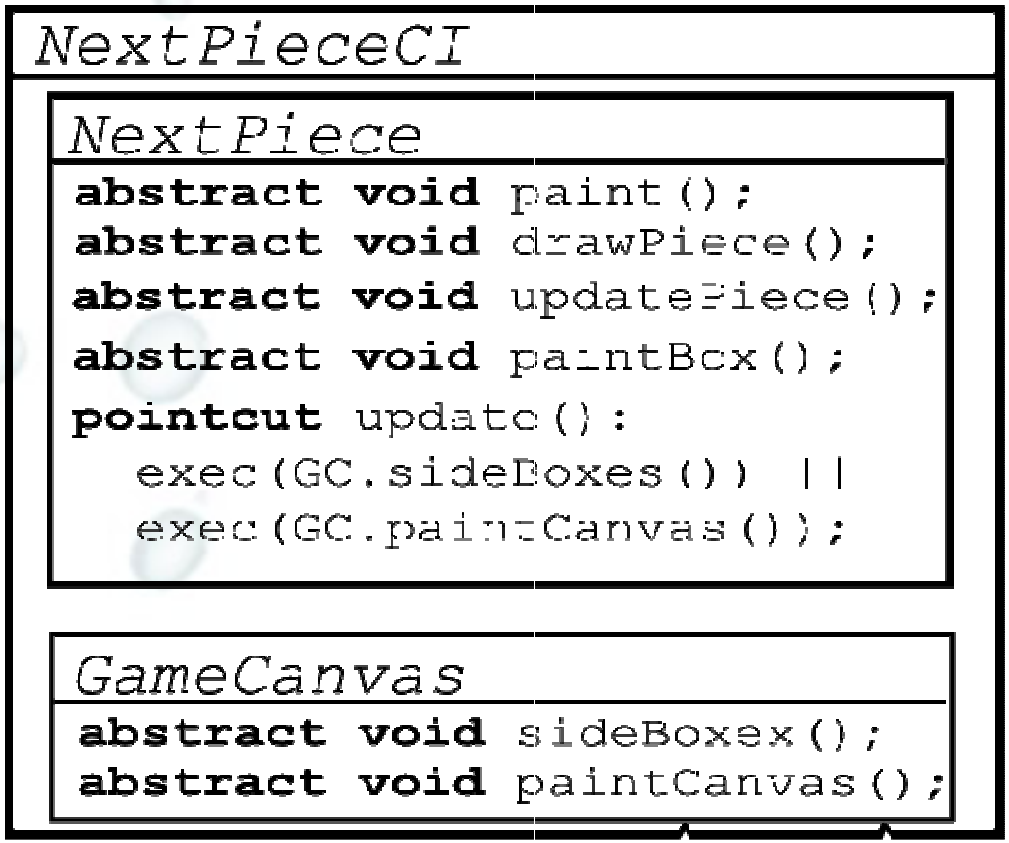


- Parallel development compromised
  - How specify the methods declared using ITD?
  - Independently compilation is not possible
- XPI can not guarantee that the methods nor the class exist.

CaesarJ

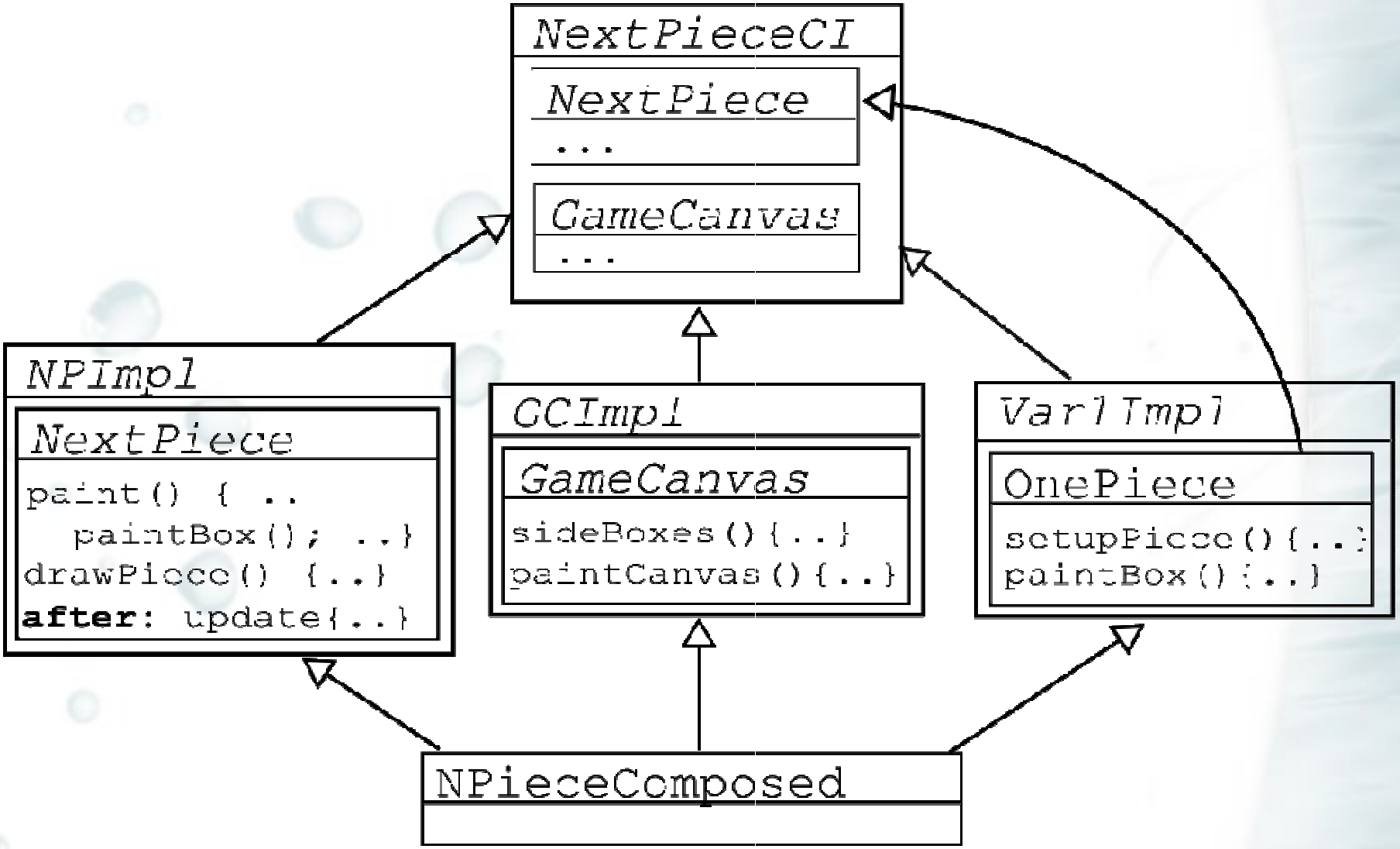
- No differentiation between classes and Aspects
  - An aspect is a Caesar class (cclass) with pointcuts / advices
- Aspect Collaboration Interfaces (ACI)
  - Interface that contains other interfaces (nested)
    - Virtual classes
  - Partial implementation
- Mixins are used to compose the partial implementations

Tetris Example ACI – CaesarJ



- ACI defining that two cclasses (NextPiece and GameCanvas) must exist
  - Defines the minimun content of both classes

Implementation of ACI



Selection of the variation

- Mixin composition

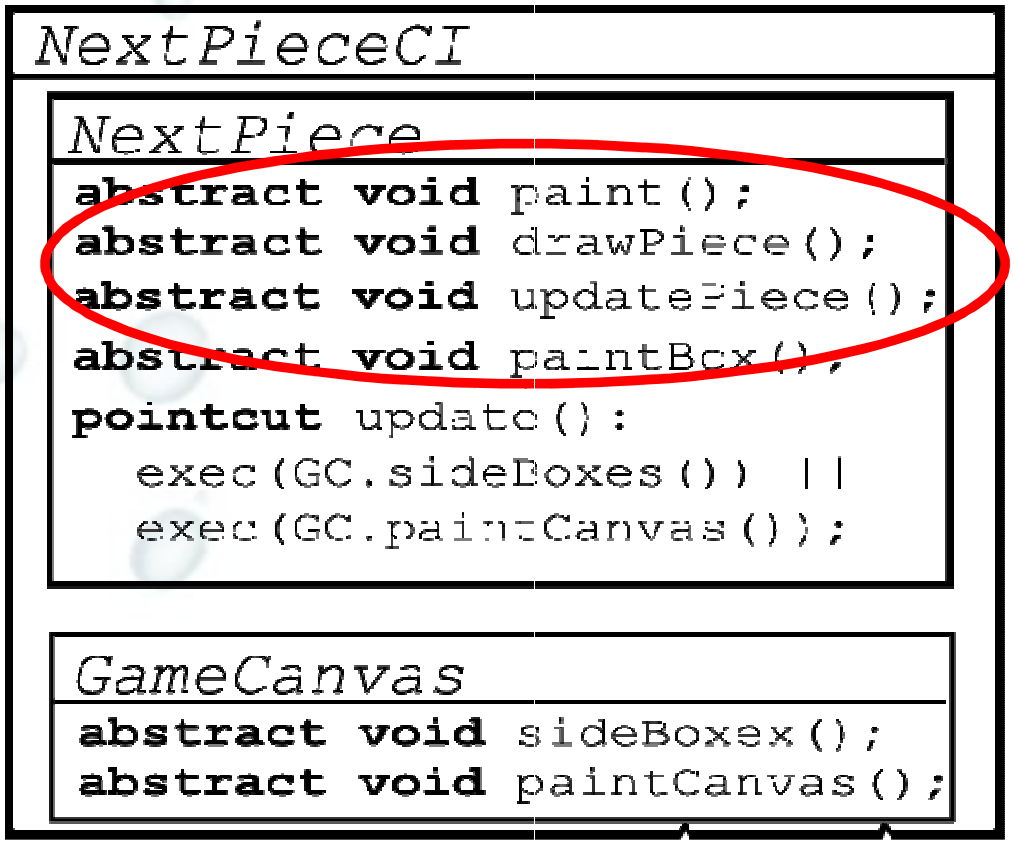
```
cclass NPComposition extends NPImpl & GCImpl & Var1Impl
```

- As on TM, we can use a Factory to instantiate the correct variation
- Each cclass can be independently compiled, just using the Interface (ACI)

Problems

- Its not possible to specify on the interface which methods are from each “role” (base or variation) (1)
  - Commentaries can not ensure the constrains
- Implementation or Refinement? (2)
  - Refine or implement a complete abstract class is the same thing
    - Different nomenclatures
- Overhead of the virtual classes (3)
  - All partial implementations contain an outer class

Problems (1)

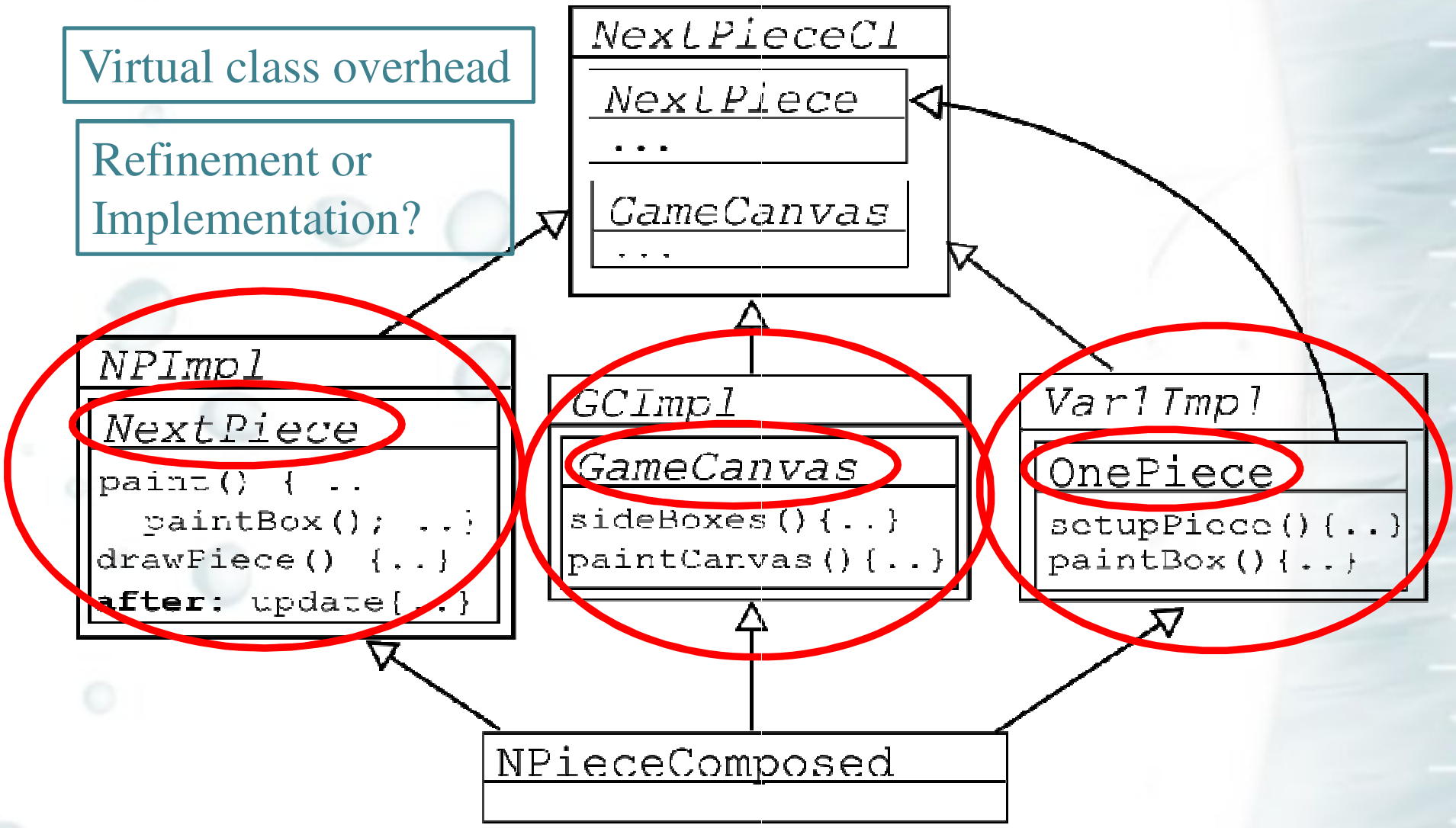


- paint() and drawPiece(): Methods of the "base"
- updatePiece() and paintBox(): Methods of the variations

Problems (2, 3)

Virtual class overhead

Refinement or Implementation?



Proposed Solution

- Extension to the actual concept of ACI

```
NextPieceCI [Base, Variation, Canvas] {
        Base {
            paint();
            drawPiece();
            pointcut update(): execution(Canvas.sideBoxes()) || ...;
        }
        Variation complements Base {
            updatePiece();
            paintBox();
        }
        Canvas {
            sideBoxes();
            paintCanvas();
        }
}
```

Implementation of DR

```
NextPiece extends NextPieceCI as Base {
paint() { ... }
drawPiece(){ ... }
after(): update { ... }
}

GameCanvas extends NextPieceCI as Canvas {
sideBoxes() { ... }
paintCanvas(){ ... }
}
```

Common part

```
OnePiece extends CI as Variation  {
updatePiece() { ...' }
paintBox() { ...' }
}

TwoPieces extends CI as Variation  {
updatePiece() { ..." }
paintBox() { ..." }
}
```

Variation part

cclass NPComposition **extends** NextPiece & GameCanvas & OnePiece

Current Stage and Future Work

- Using Stratego/XT to transform the code written on our extension into a valid CaesarJ code
- More examples
  - HealthWatcher
  - MobileMedia
- Propose new constructors to the DR extension
- Analysis of the proposed solution
  - Parallel with CaesarJ, LSD...

Thank you! Questions

Specifying Design-Rules to improve modularity between OO/AO code with CaesarJ

Carlos Eduardo Pontual

ceplc@cin.ufpe.br