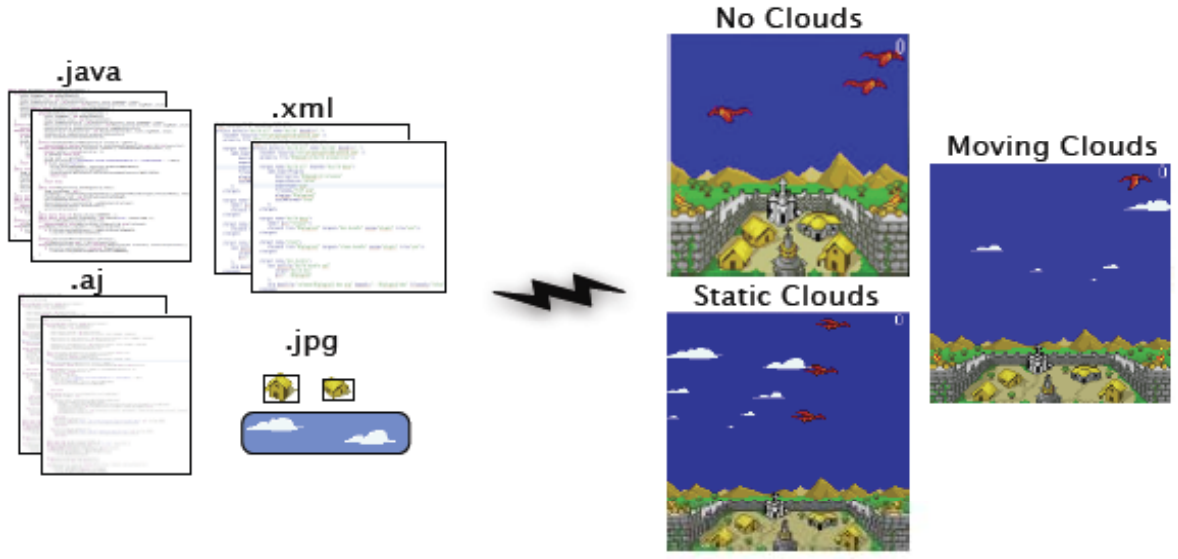
Modularizing Variabilities with CaesarJ Collaboration Interfaces

Carlos E. Pontual
Rodrigo Bonifácio
Henrique Rebêlo
Márcio Ribeiro
Paulo Borba

Software Product Lines (SPL)

- Aim to reduce the time to market of applications in a common domain
- Products are a combination of base (common) and variant behavior



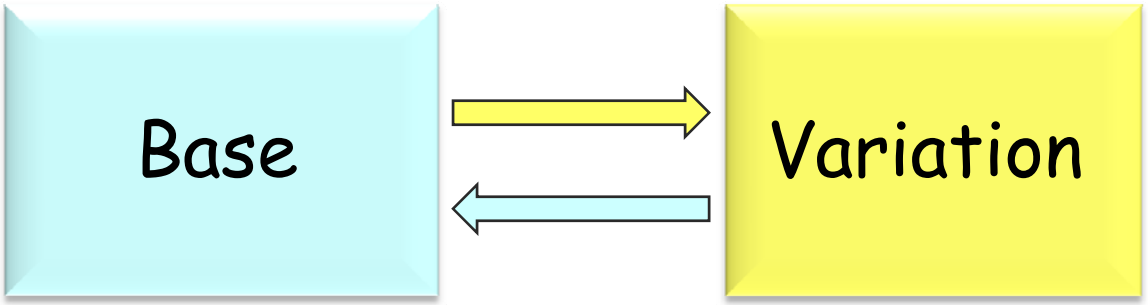
Rain of Fire: http://meantime.com.br/en/sec_game_rain.html

SPL Variabilities can be implemented with…

- Annotative approaches
  - Colored Ide (CIDE)
  - …
- Compositional Approaches
  - AOP
  - FOP

What is the problem?

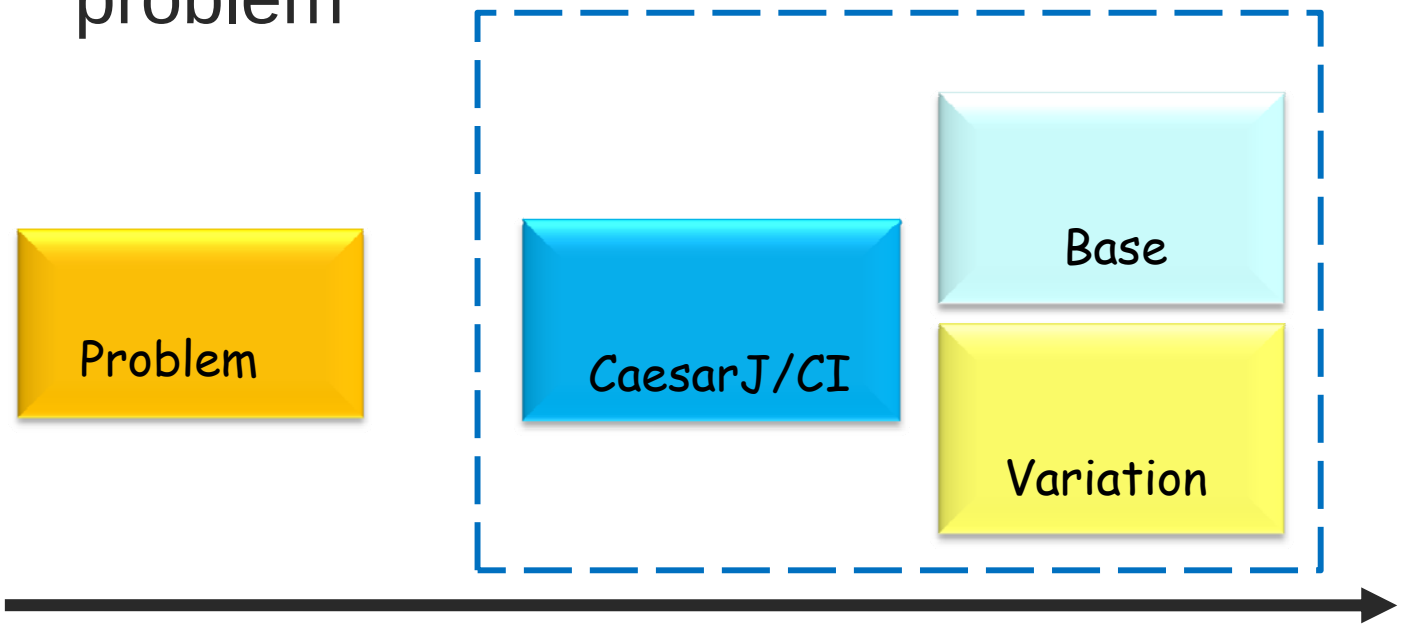
- Base and variation code are coupled!



- Symptoms
  - no parallel development
  - hard maintenance
  - …

Objectives

- CaesarJ Collaboration Interface (CI)
  - To implement (modularize) variabilities
  - To tackle the parallel development problem

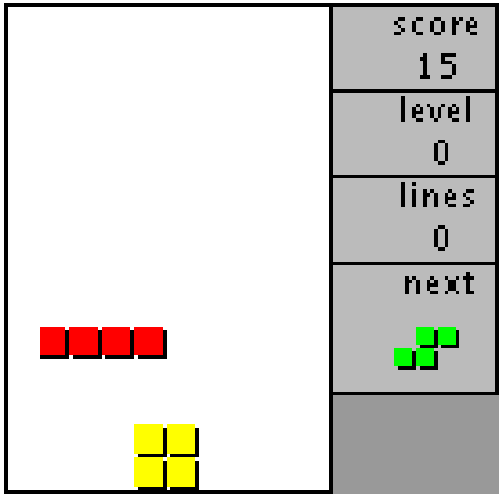


Outline

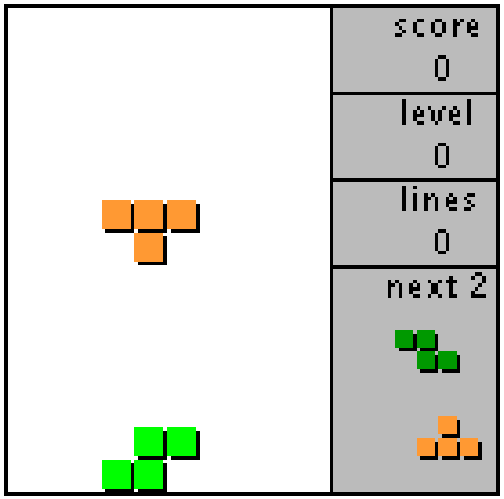
- Motivating Example
  - Variation Details
- TM Implementation
- CI implementation
- Final Remarks

Motivating Example

- Simple Tetris game SPL
- Variant behaviour
  - Game level

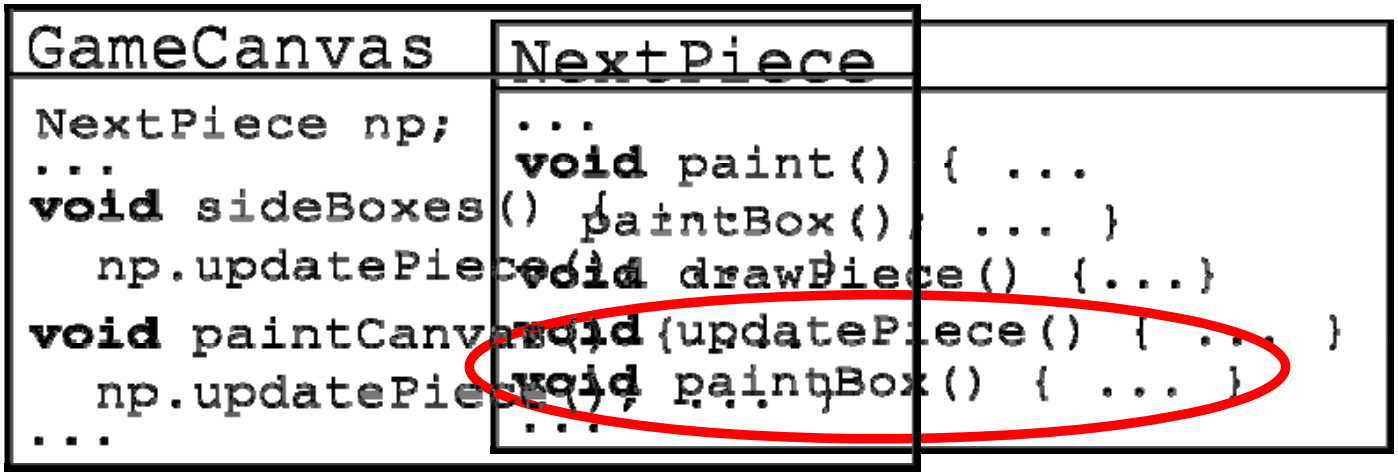


Normal



Easy

Variation Details



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

Implementation with TM

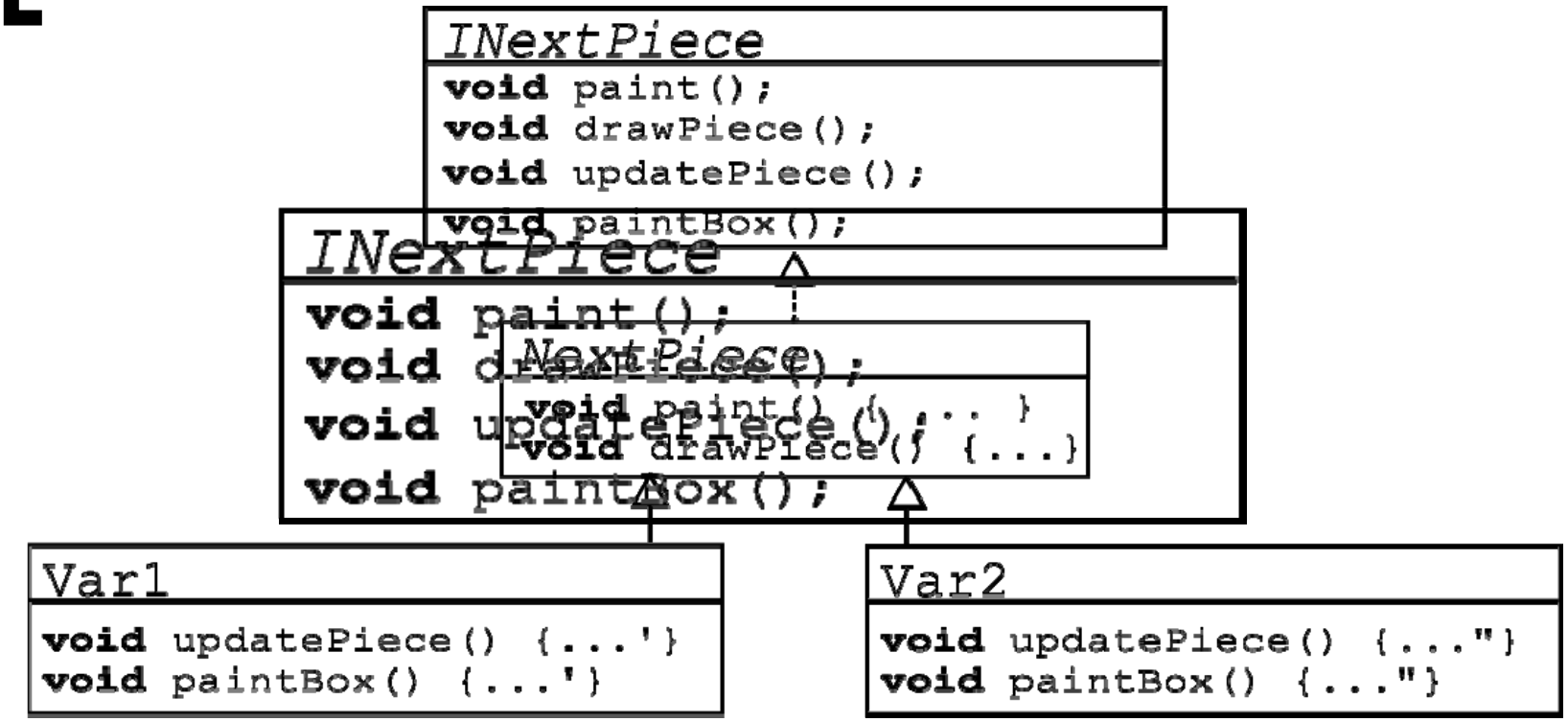
```
abstract class NextPiece {
    void paint() { ...
        paintBox(); ... }
    void drawPiece() { ... }
    abstract void updatePiece();
    abstract void paintBox();
}
```

```
class Var1 extends NextPiece {
    void updatePiece() { ... }
    void paintBox() { ... }
}
```

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

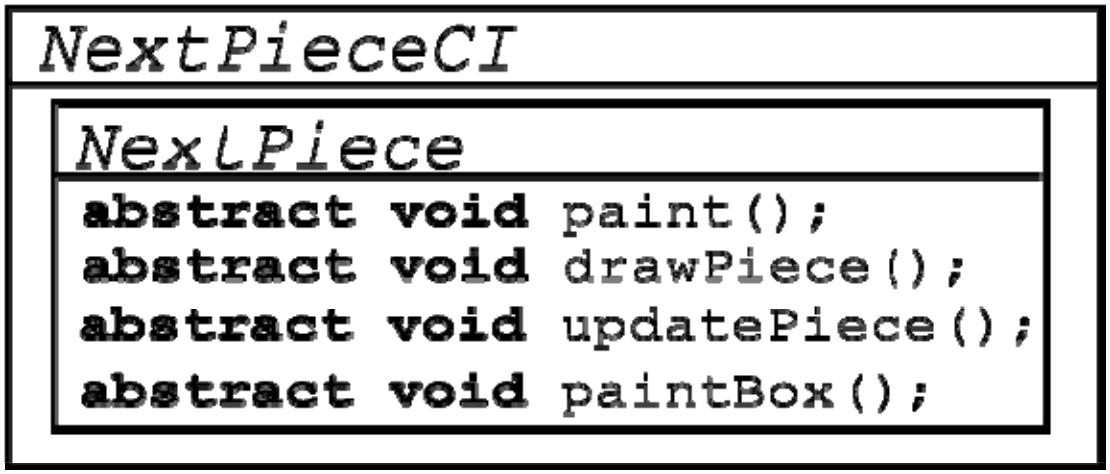
- Tangling of the design variation (abstract signatures) with base implementation (problem P1.1)
  - Variations can only be implemented after base code implementation

Enhancing TM



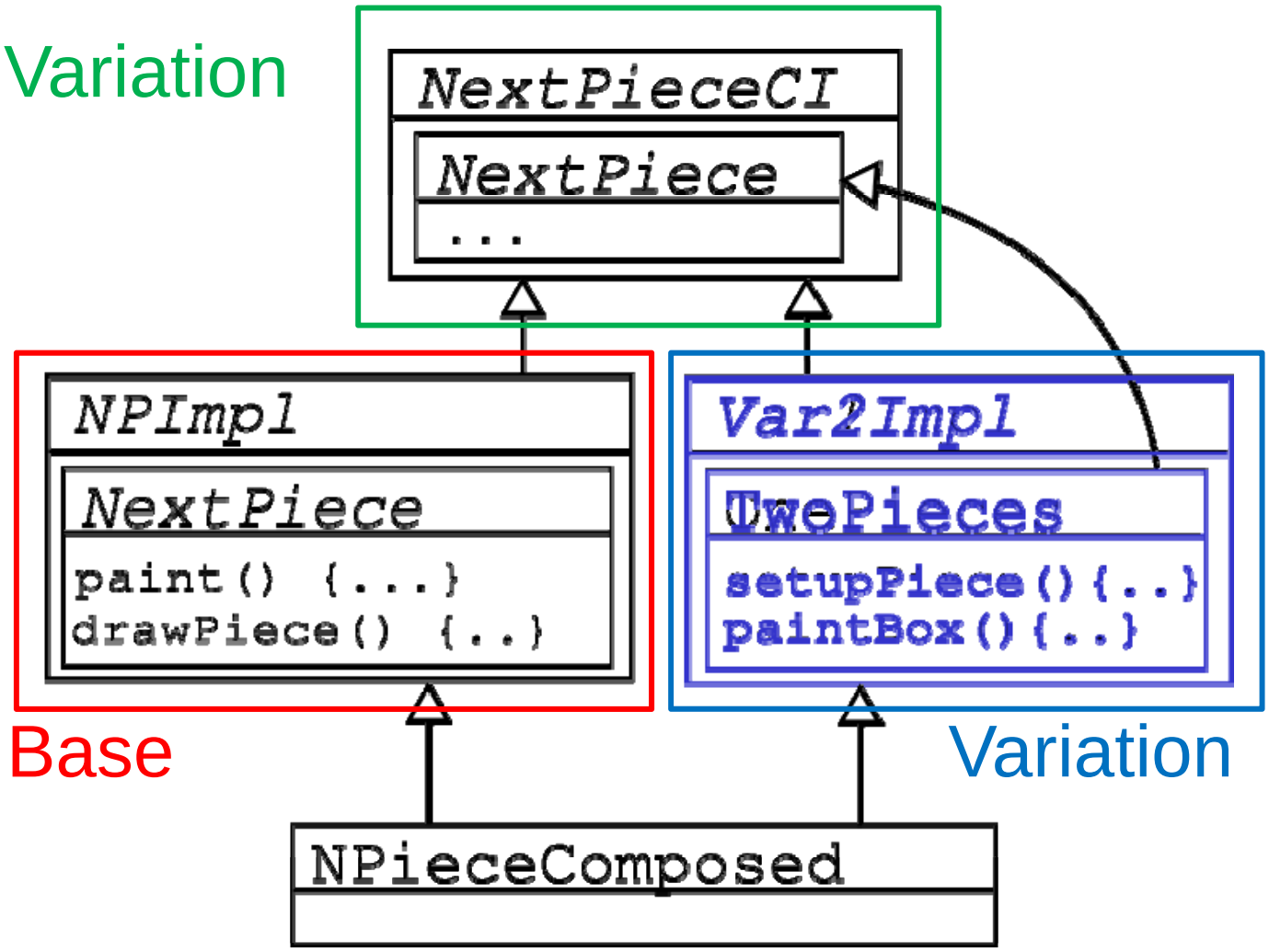
- Its not clear on the interface which methods are from the base role and which are from the variation (P2.1)
  - Not enough for parallel development
- Variation code can not be compiled independently of base code (P2.2)

Implementation with CI



- CI (interface) where the NextPiece class must have at least these four methods
  - This class can be partially implemented

Implementation with ACI



Analyzing CI Solution

- Interface enables independent compilation of base/variations
  - They depend only on the interface
- Difficult to explicitly specify multiple roles (base/variation) on the interface – P2.1
  - Compiler only check if all the methods of the interface are implemented on the composed mixin
- Overhead of the virtual classes
  - All partial implementations contain an outer class

Final Remarks

- CI is not enough to enable parallel development
  - Problem P2.1 remains open
- Working on extensions to the CI concept
- Using CIs to enable a more modular design OO/AO
  - Design-Rules

What we want is…

- To clearly define the obligations of different teams – to enable parallel development
- To specify design rules in such a way they can be statically checked by a compiler

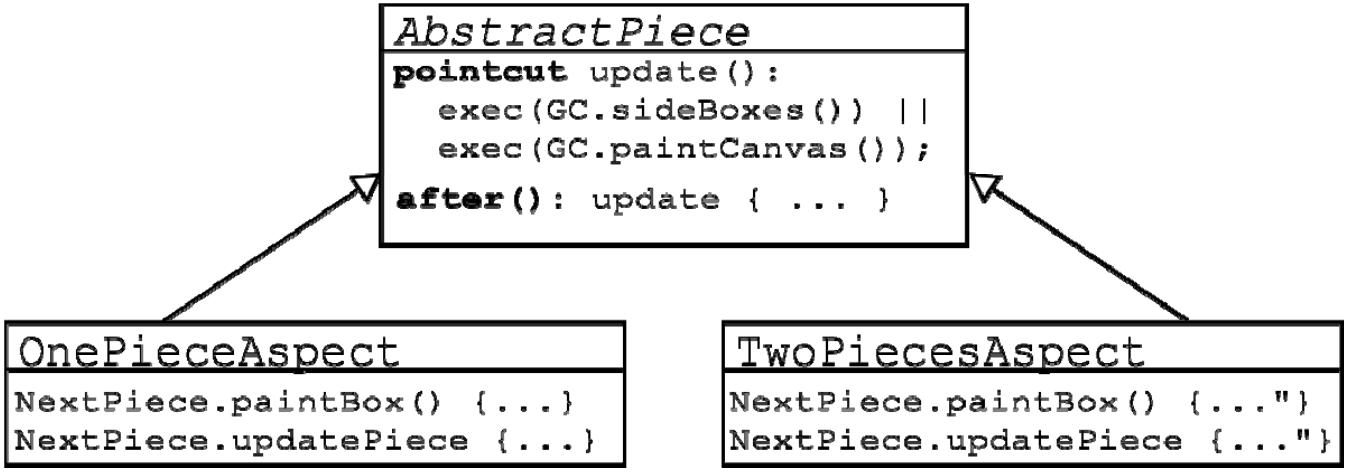
Modularizing Variabilities with CaesarJ Collaboration Interfaces

Carlos E. Pontual
Rodrigo Bonifácio
Henrique Rebêlo
Márcio Ribeiro
Paulo Borba

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

Implementation - AO



- Parallel development compromised
  - How specify the methods declared using ITD?
  - Independently compilation is not possible
- XPI can not guarantee that the methods or the class exist.