Fragility in Evolving Software

BENEVOL'13

16,17 Dec. 2013

Mons, Belgium

Prof. Kim Mens

ICTEAM

UCL, Belgium

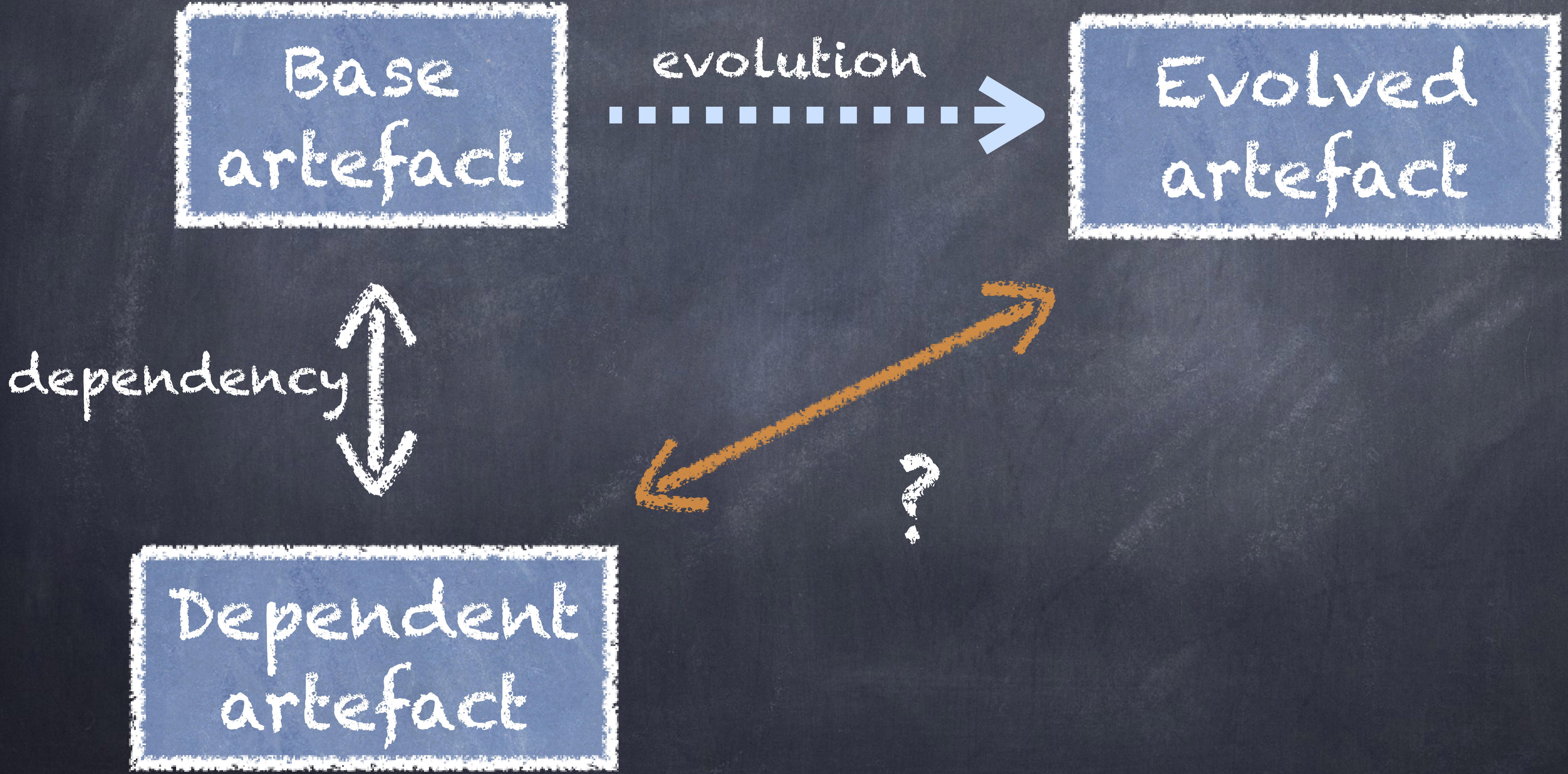
Fragility in Evolving Software

Fragility

- Software artefacts make many assumptions about other software artefacts
- Many of these assumptions are not documented, nor verified explicitly
- When an artefact evolves into one that doesn't respect the assumptions, fragility problems arise
- Fragility problems can thus be seen as a kind of substitutability problem

Fragility

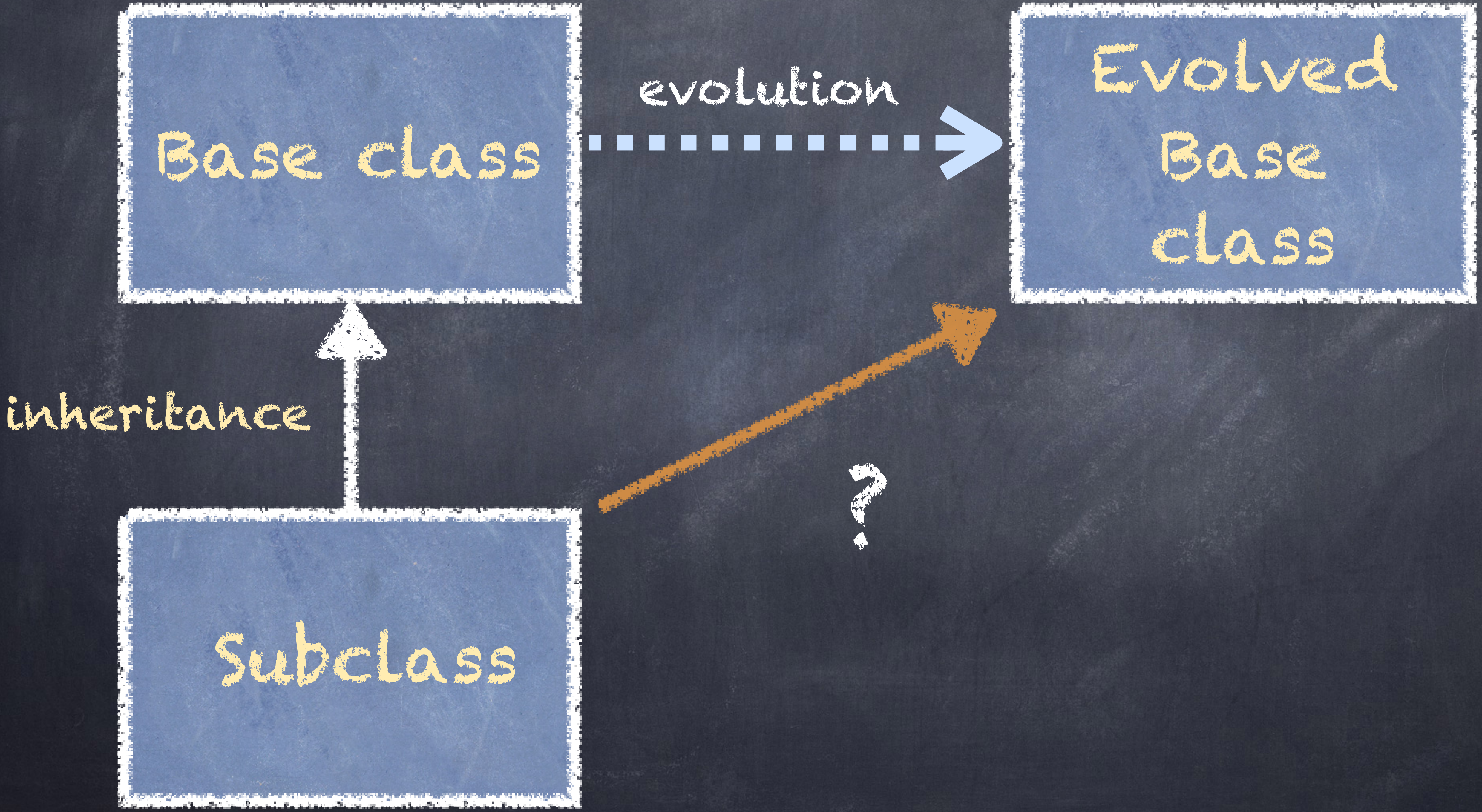
fragile


Base artefact
evolution
Evolved artefact
dependency
?
Dependent artefact

The Fragile Base Class Problem

- Object-oriented programs consist of many classes connected through inheritance
- Base classes make assumptions about how they can be reused by subclasses
- Subclasses make assumptions about the base classes they inherit from
- These assumptions are often not documented explicitly, nor verified automatically
- The fragile base class problem [5,7] occurs when a base class evolves into a new one that doesn't respect these assumptions

The Fragile Base Class Problem



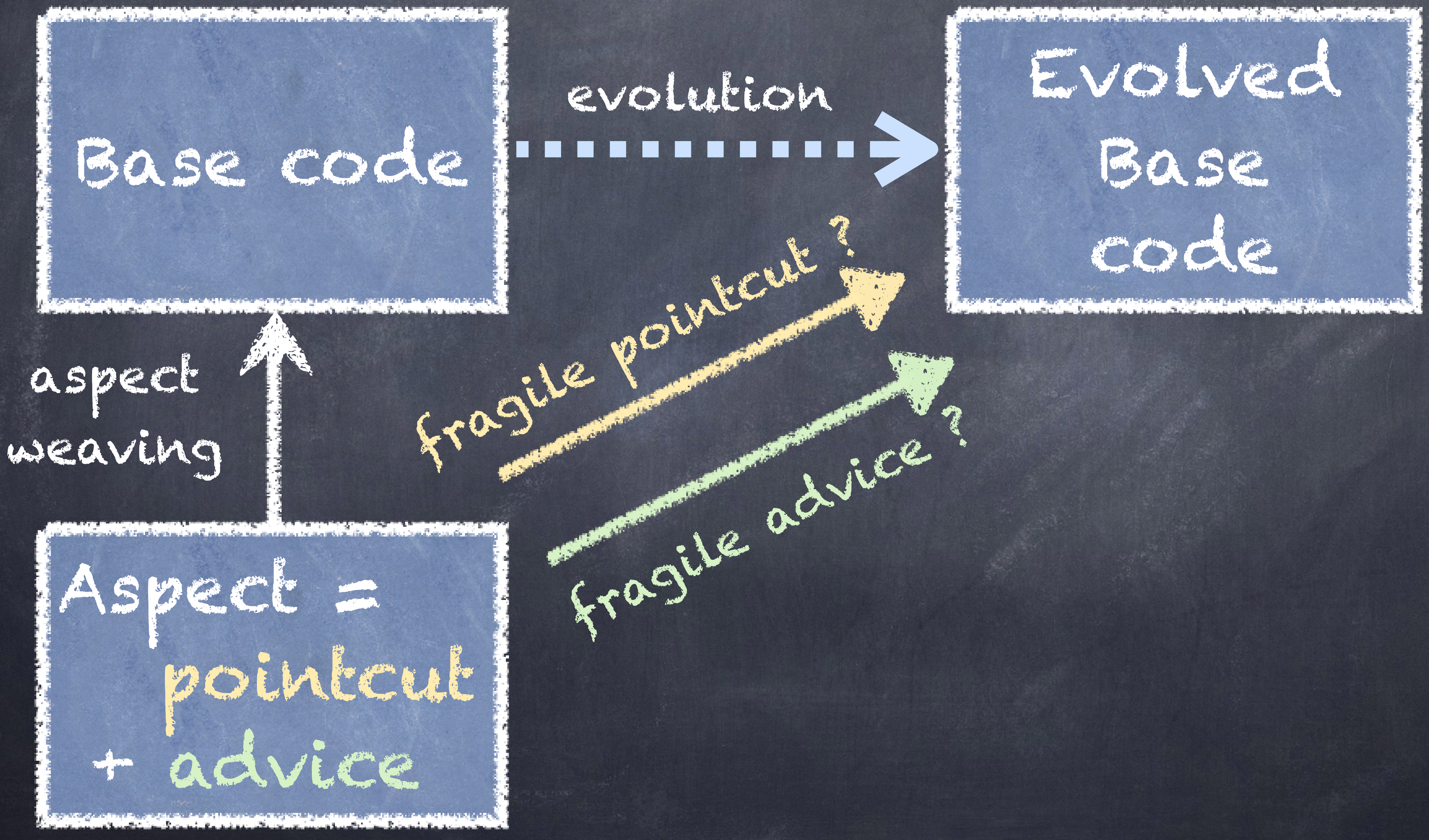
Fragility in Aspect Oriented Programming

- AOP = base code + aspects
- Aspects = pointcuts + advice code
- Aspects modify the behaviour of the base code by weaving in advice code at various join points, described by pointcut expressions
- Base code is oblivious to the aspects

Fragility in AOP

- Both pointcuts and advice code make assumptions about the base code they refer to or act upon
- These assumptions are not documented explicitly, nor verified automatically
- Subtle conflicts arise when the base code evolves in a way that breaks these assumptions
- These problems are known as the fragile pointcut problem [2,6] and the fragile advice problem [1]

Fragility in AOP



Dealing with fragility

- In general, fragility problems arise when implicit assumptions between dependent artefacts get broken when the system evolves
- Solution consists of documenting the assumptions explicitly, and verifying them upon evolution
- By defining some kind of evolution contract between the evolving artefact and its dependent artefacts
- A verifiable agreement between the two parties
- Detect fragility conflicts by verifying the contract

Dealing with fragility

- Different kinds of fragility conflicts can be distinguished, depending on :
  - how the artefact evolves
  - how the artefacts depend on each other
  - what conditions of the contract are breached
- Appropriate solutions to these conflicts can be proposed accordingly

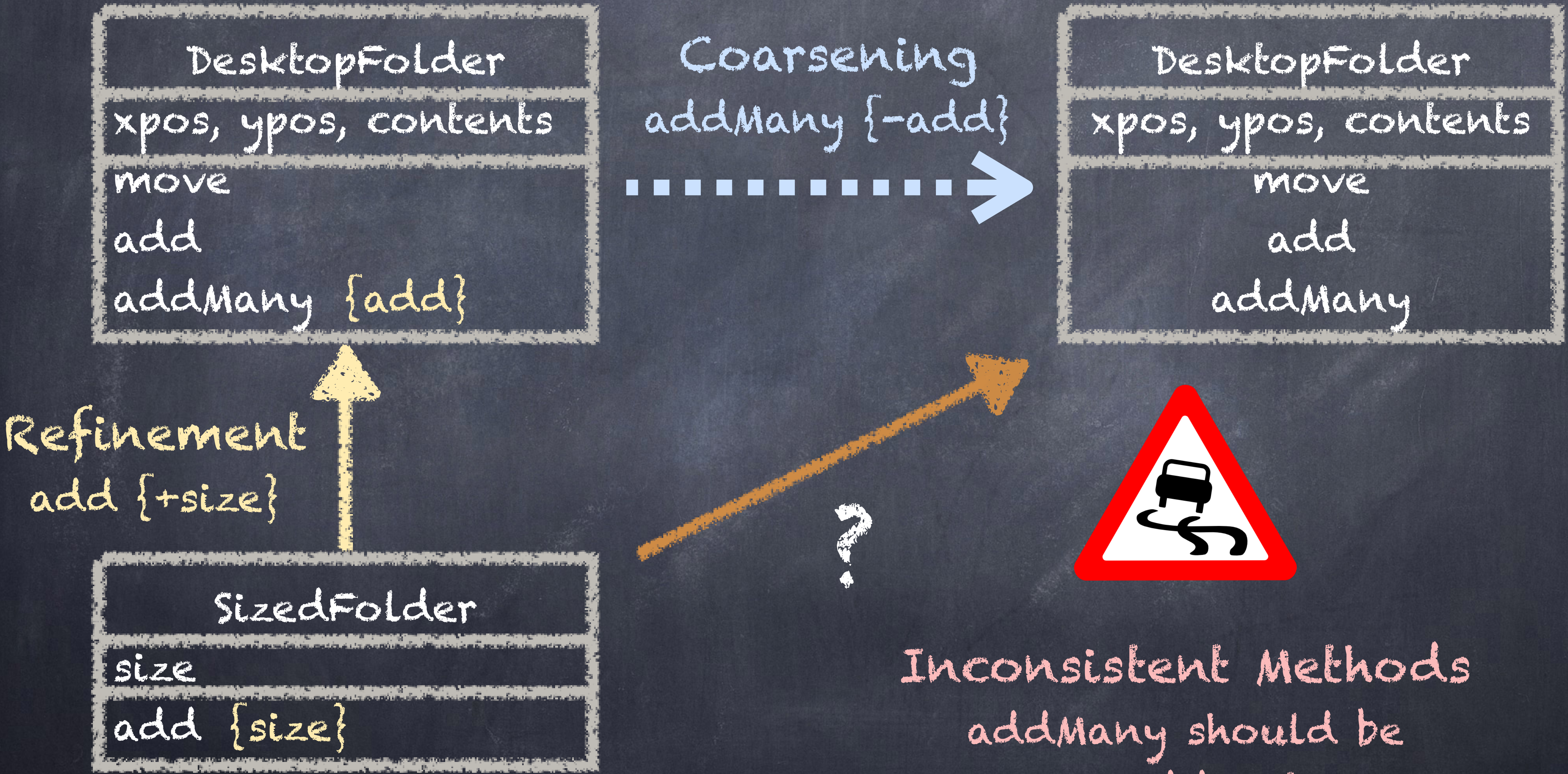
Handling the fragile base class problem

1. Define a reuse contract [7] between a derived class and the base class it depends on
   - how does the reuser specialize the base class?
2. Define a usage contract [4] between the base class and the derived classes that "use" it
   - what regularities does the base class expect the derived classes to respect ?

Reuse Contracts

- Reuse contracts define an evolution contract between a "reuser" and the base class it depends upon
    - the base class declares a kind of specialization interface [3] defining what reusers can rely upon
    - a reuse operator defines how derived classes reuse the base class
    - an evolution operator defines how the base class evolved

Reuse Contracts



Inconsistent Methods

addMany should be overridden too

Reuse Contracts

	Extension +m	Refinement m {+n}	Cancelation -m	Coarsening m {-n}	...
Extension +p	name con- flict [m=p]	method capture [n=p]	...	...	...
Refinement p {+q}	...	...	...	inconsistent method [n=p]	...
Cancelation -p	...	...	...	...	...
Coarsening p {-q}	...	...	...	...	...
...	...	...	...	...	...

Reuse Contracts

	Extension +m	Refinement m {+n}	Cancelation -m	Coarsening m {-n}	...
Extension +p	name con-flict [m=p]	method capture [n=p]	...	...	...
Refinement p {+q}	...			inconsistent method [n=p]	...
Cancelation -p			...	...	...
Coarsening p {-q}	...	...	...	...	...
...	...	...	...	...	...

Coarsening
addMany {-add}

Refinement
add {+size}

Handling the fragile base class problem

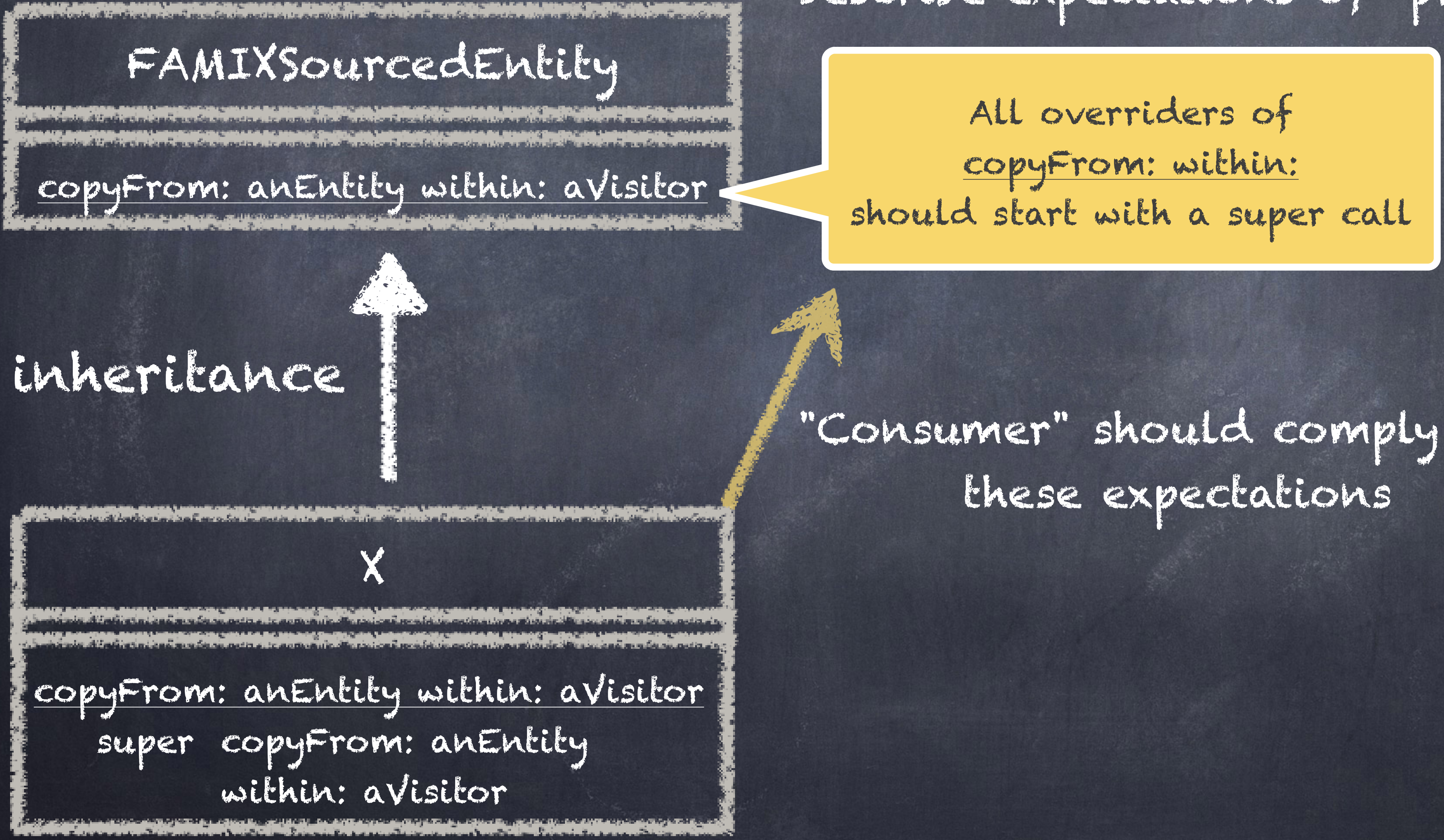
1. Define a reuse contract [7] between a derived class and the base class it depends on
   - how does the reuser specialize the base class?
2. Define a usage contract [4] between the base class and the derived classes that "use" it
   - what regularities does the base class expect the derived classes to respect ?

Usage Contracts

- Usage contracts [4] define an evolution contract between the base class and the classes that "use" it
- base class defines what regularities should be respected by its derived classes
- regularities are checked when modifying existing or creating new derived classes

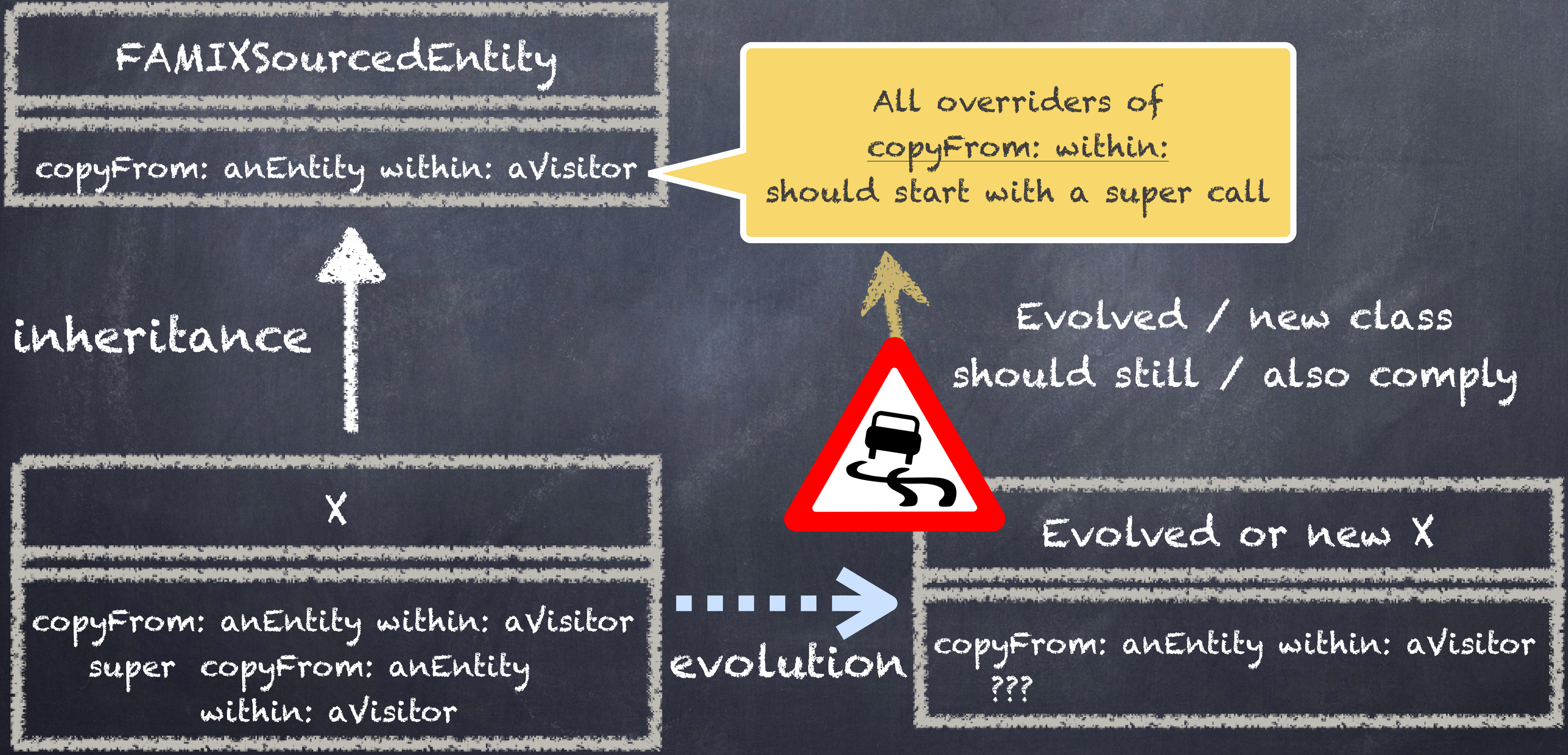
Usage Contracts

Describe expectations of "provider" :



"Consumer" should comply with these expectations

Usage Contracts


FAMIXSourcedEntity
copyFrom: anEntity within: aVisitor
All overriders of copyFrom: within: should start with a super call
inheritance
Evolved / new class should still / also comply
X
copyFrom: anEntity within: aVisitor
super copyFrom: anEntity
within: aVisitor
evolution
Evolved or new X
copyFrom: anEntity within: aVisitor
???

Usage Contracts

- A DSL for declaring usage contracts
- Specifying the liable entities of the contract
    - scope of classes or methods to which the contract is applicable
- Defining the structural regularity
    - structural constraints to be respected by the liable entities
- These lightweight contracts are checked and reported immediately during development and maintenance

Usage Contracts

All overriders of
copyFrom: within:
should start with a super call

- Specifying the liable entities

```
classesInFAMIXSourcedEntityHierarchy
    <liableHierarchy: #FAMIXSourcedEntity>
```

- Defining the structural regularity

```
copyFromWithinWithCorrectSuperCall
    <selector: #copyFrom:within:>
    contract:
        require:
            (condition beginsWith:
                (condition doesSuperSend: #copyFrom:within:)
        if: (condition isOverridden)
```

The fragile pointcut problem

- In aspect-oriented programming, the base program is oblivious of the aspects that act upon it
- Pointcut expressions describe at what join points in the base program advice code will be woven
- The fragile pointcut problem [2] occurs when pointcut expressions unintendedly capture or accidentally miss particular join points
    - as a consequence of their fragility with respect to seemingly safe evolutions of the base program

The fragile pointcut problem

- E.g., a pointcut expression to capture getters and setters
- An "enumeration" pointcut:

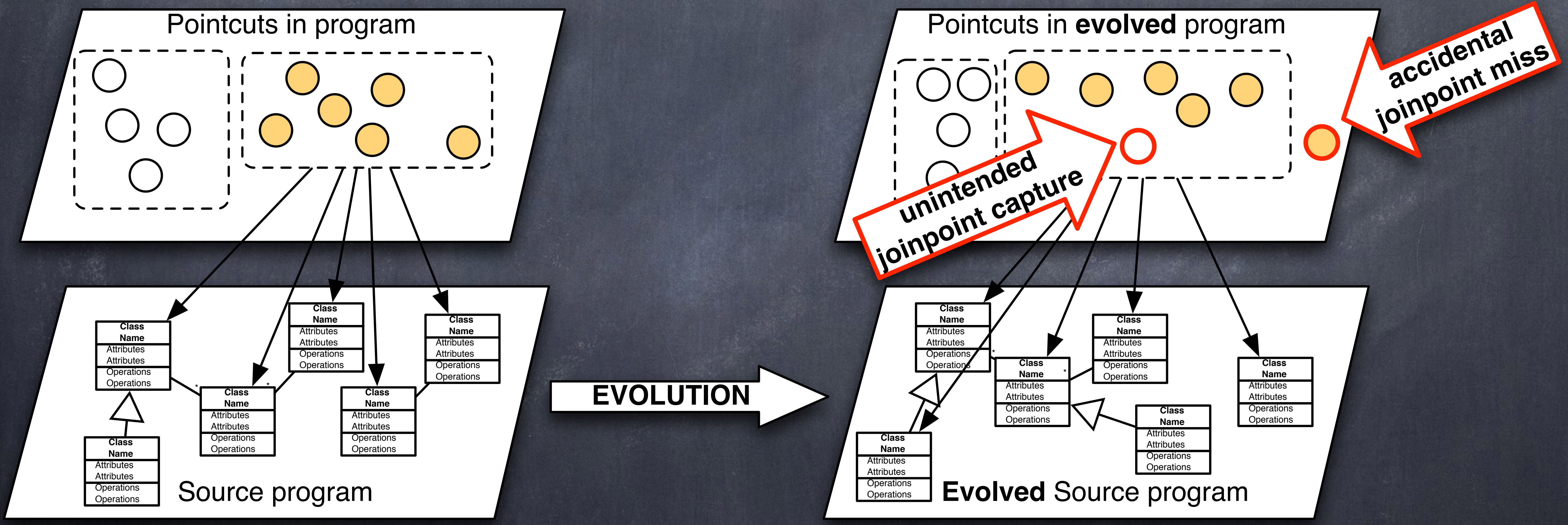
```
pointcut accessors()
   call(void Buffer.set(Object)) || call(Object Buffer.get());
```

  - May accidentally miss other relevant getters and setters
- A "pattern-based" pointcut:

```
pointcut accessors()
   call(* set*(..)) || call(* get*(..));
```

  - May unintendedly capture methods that aren't getters or setters
  - for example, a method named `setting`

The fragile pointcut problem



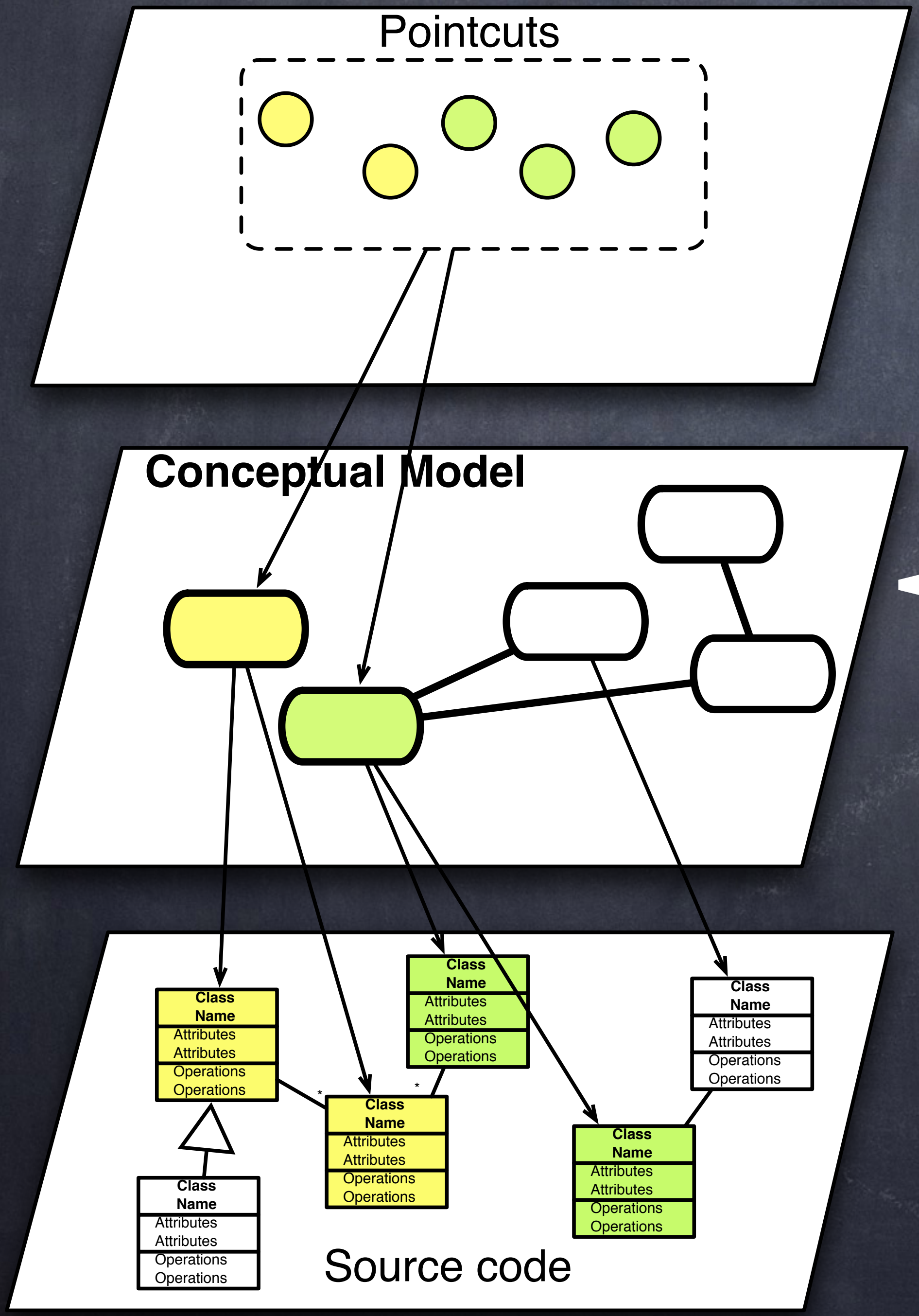
Model-based pointcuts

- Model-based poincuts [2] define an evolution contract between the pointcuts and the base code, in terms of an intermediate model that both agree upon.
- A "model-based" pointcut:

```
pointcut accessors()
   classifiedAs(?methSignature,AccessorMethods) &&
   call(?methSignature)
```

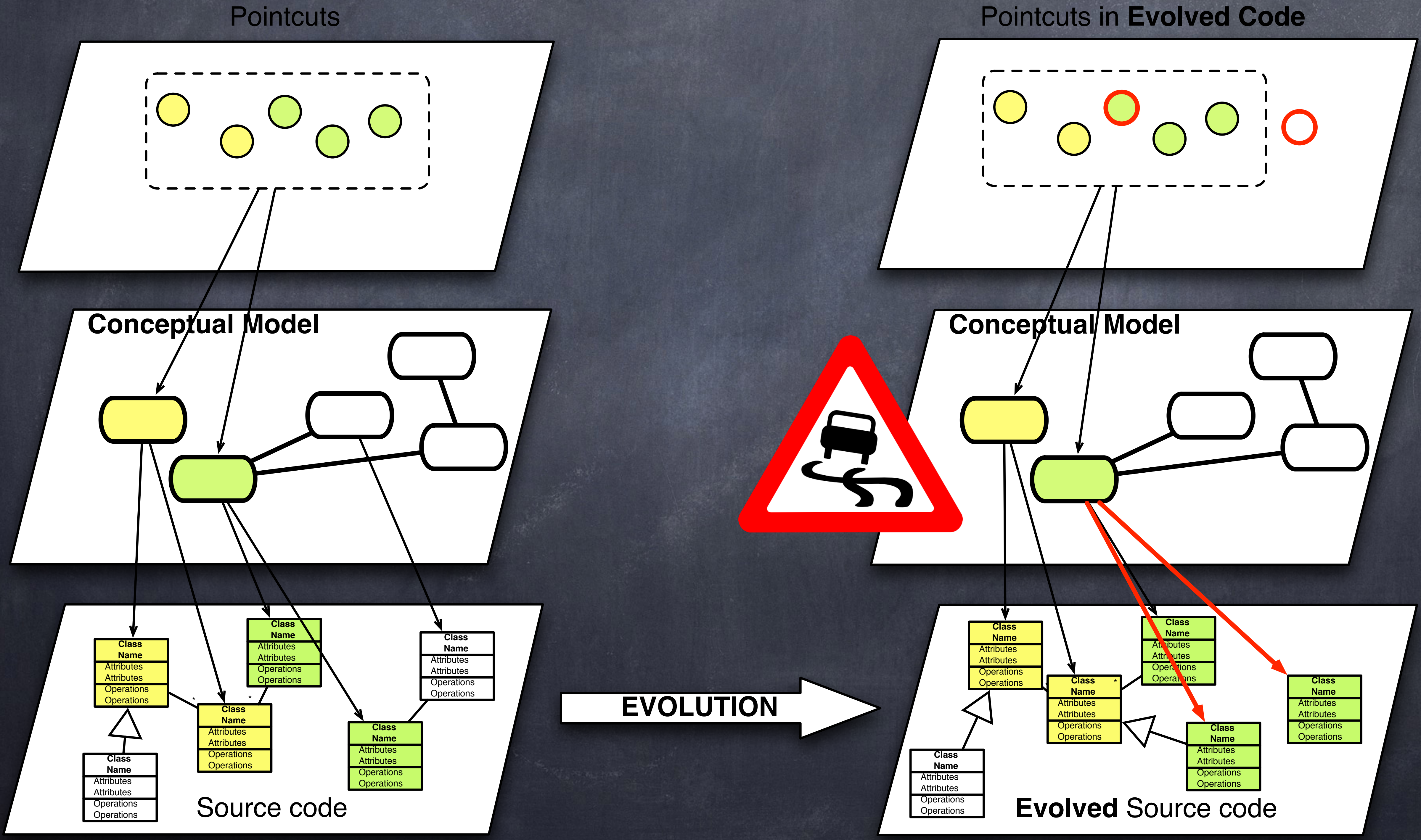
- The base code should comply with the model, but remains oblivious of the actual aspects
- Some fragility problems can be encountered by defining and verifying additional constraints at the level of the model
  - For example, every setter method should have a name of the form setX and assign a value to the variable X

Model-based pointcuts



Additional constraints such as:
{ methods named set* } =
{ methods assigning an instance variable }

Solving the fragile pointcuts problem: Model-based pointcuts



Summary

- Software fragility arises when implicit assumptions artefacts make about dependent and depending artefacts get broken upon evolution
- Solutions consists of documenting these assumptions explicitly as "evolution contracts", and verifying them whenever the software evolves
- Appropriate conflict resolutions can be suggested depending on how the artefacts evolved and on what assumptions got broken

Other potential application areas

- Until now we have applied these ideas mostly to OO, FW and AO software development
- We are studying the application of ideas to the area of dynamically adaptive systems (in particular: COP)
- Other envisaged application areas :
  - evolving configurations of network routers
  - data-intensive software systems
  - unit tests vs. source code
  - ...

Take-away message

- A good idea is timeless; to reinvent yourself it sometimes suffices to rediscover yourself.
- Revive research on software reuse and evolution by reusing or evolving previous research on software reuse and evolution.
- Context-oriented progamming is one potential new application area of these ideas.
- Ideas could be applied to any domain where you have potential fragility problems.

Timeline

Reuse Contracts: Managing the Evolution of Reusable Assets

[illegible]

Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts

[illegible]

Usage Contracts: Offering Immediate Feedback on Violations of Structural Source-code Regularities

Angela Lozano[a], Andy Kellens[b], Kim Mens[a]

[a] *Université catholique de Louvain*
[b] *Vrije Universiteit Brussel*

Abstract

Developers often encode design knowledge through structural regularities such as API usage protocols, coding idioms and naming conventions. As these regularities express how the source code should be structured, they provide vital information for developers using or extending that code. Adherence to such regularities tends to deteriorate over time when they are not documented and checked explicitly. This paper introduces *uContracts*, an internal DSL to codify and verify such regularities using 'usage contracts'. Whereas our DSL does not aim to be complete, it does cover common usage regularities, while providing a means through metaprogramming to express atypical regularities. Common regularities are identified based on regularities supported by existing approaches to detect bugs or suggest missing code fragments, techniques that mine for structural regularities, as well as on the analysis of an open-source project. We validate our DSL by documenting the usage regularities of an industrial case study, and analyse how useful the information provided by checking these regularities is for the developers of that case study.

1. Introduction

Being able to document and preserve architectural integrity and design knowledge of an application is important to increase its longevity [4]. Given that over time the actual implementation structure tends to drift away from

Email addresses: `angela.lozano@uclouvain.be` (Angela Lozano), `andy.kellens@vub.ac.be` (Andy Kellens), `kim.mens@uclouvain.be` (Kim Mens)

?

Reuse Contracts OOPSLA'96

Model-based Pointcuts AOSD'06

Usage Contracts (2013)

Dyn. evol. contracts

Some references

1. Cyment, Kicillof, Altman & Asteasuain. Improving AOP systems' evolvability by decoupling advices from base code. RAM-SE'06 WS @ ECOOP'06, pp 9-21, 2006.

2. Kellens, Mens, Brichau & Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. ECOOP'06, pp 501-525, 2006.

3. Lamping. Typing the specialization interface. OOPSLA '93, pp 201-214, 1993.

4. Lozano, Kellens & Mens. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. Under revision.

5. Mikhajlov & Sekerinski. A study of the fragile base class problem. ECOOP'98, pp 355-382, 1998.

6. Noguera, Kellens, Deridder & D'Hondt. Tackling pointcut fragility with dynamic annotations. RAM-SE'10 WS @ ECOOP'10, pp 1-6, 2010.

7. Steyaert, Lucas, Mens & D'Hondt. Reuse contracts: Managing the evolution of reusable assets. OOPSLA'96, pp 268-285, 1996.