Fragility in Evolving Software
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.
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

Base class

Evolved Base class

Subclass

inheritance

evolution

?
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

Base code

Aspect = pointcut + advice

Evolved Base code

Evolution

Fragile pointcut?

Fragile advice?
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**

- **DesktopFolder**
  - xpos, ypos, contents
  - move
  - add
  - addMany {add}

- **SizedFolder**
  - size
  - add {size}

- **Coarsening**
  - addMany {add}

- **Inconsistent Methods**
  - addMany should be overridden too
# Reuse Contracts

<table>
<thead>
<tr>
<th></th>
<th>Extension $+m$</th>
<th>Refinement $m {+n}$</th>
<th>Cancelation $-m$</th>
<th>Coarsening $m {-n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension $+p$</td>
<td>name conflict $[m=p]$</td>
<td>method capture $[n=p]$</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Refinement $p {+q}$</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>inconsistent method $[n=p]$</td>
</tr>
<tr>
<td>Cancelation $-p$</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Coarsening $p {-q}$</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>
# Reuse Contracts

<table>
<thead>
<tr>
<th>Extension</th>
<th>Refinement</th>
<th>Cancelation</th>
<th>Coarsening</th>
</tr>
</thead>
<tbody>
<tr>
<td>+m</td>
<td>m {+n}</td>
<td>-m</td>
<td>m {-n}</td>
</tr>
<tr>
<td>name conflict [m=p]</td>
<td>method capture [n=p]</td>
<td>inconsistent method [n=p]</td>
<td></td>
</tr>
</tbody>
</table>

**Extension**
- +p

**Refinement**
- p {+q}

**Cancellation**
- p

**Coarsening**
- p {-q}

- Refinement add {+size}

- Coarsening addMany {-add}
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":

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

"Consumer" should comply with these expectations
Usage Contracts

FAMIXSourcedEntity

- `copyFrom: anEntity within: aVisitor`

Inheritance

Evolved or new X

- `copyFrom: anEntity within: aVisitor`  
  - `super copyFrom: anEntity within: aVisitor`  

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

Evolved / new class should still / also comply

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

- 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 pointcuts 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:

```plaintext
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:

\{ \text{methods named set*} \} = \{ \text{methods assigning an instance variable} \}
Solving the fragile pointcuts problem:
Model-based pointcuts

Pointcuts

Evolved Pointcuts

Conceptual Model

Source code

Evolved Source code
Summary

- Software fragility arises when implicit assumptions artefacts make about dependent and depending artefacts get broken upon evolution.

- Solutions consist 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 programming is one potential new application area of these ideas.

- Ideas could be applied to any domain where you have potential fragility problems.
Usage Contracts: Offering Immediate Feedback on Violations of Structural Source-code Regularities

Angela Lozano, Andy Kellens, Kim Mens

1. Introduction

Being able to document and preserve architectural integrity and design knowledge of an application is important to ensure its longevity [4]. Given that over time the actual implementation structure tends to drift away from the original design, it is crucial to detect and correct violations of structural source-code regularities. While various techniques exist to detect such violations, offering immediate feedback to developers is essential to ensure adherence to architectural integrity.

Usage Contracts offer a mechanism to codify and verify structural source-code regularities using a system of declarative contracts. These contracts are intended to capture common structural regularities, and support atypical regularities by providing a mechanism for their expression.

Contracts are implemented using a declarative language that is based on pattern matching. This approach allows for the automatic identification of structural violations, which can then be communicated to developers in real-time.

This paper introduces Usage Contracts, a novel approach to documenting and verifying structural source-code regularities. The authors present a case study to demonstrate the effectiveness and potential benefits of using Usage Contracts in practice.
Some references


