Mining Aspectual Views using Formal Concept Analysis

Tom Tourwe
SEN1/CWI
The Netherlands

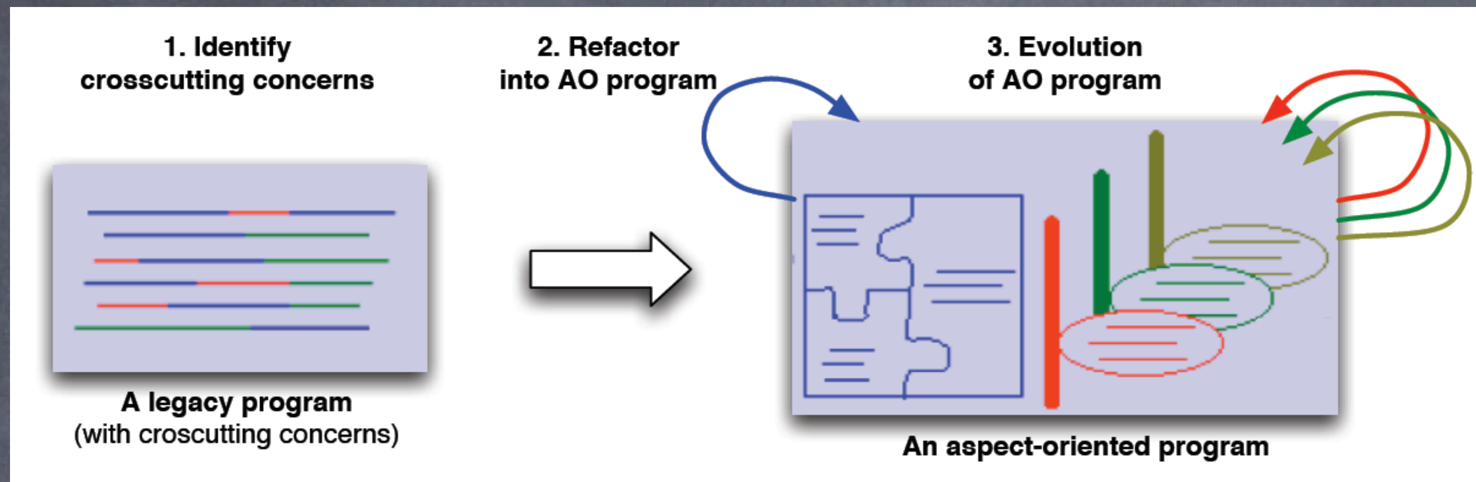
Kim Mens
INGI / UCL
Belgium

Source Code Analysis and Manipulation Workshop - Chicago - September, 2004.

Overview

- Research context
- Mining aspectual views with FCA
- Detailed approach
- Details of the experiment
- Conclusion

Research Context



Three important research goals

- Automatically identify crosscutting concerns
  - Based on pattern matching, clone detection, logic reasoning, concept analysis, clustering, ...
- Refactor / restructure legacy programs into aspect-oriented ones
- Support evolution of aspect-oriented programs
  - Aspect refactoring

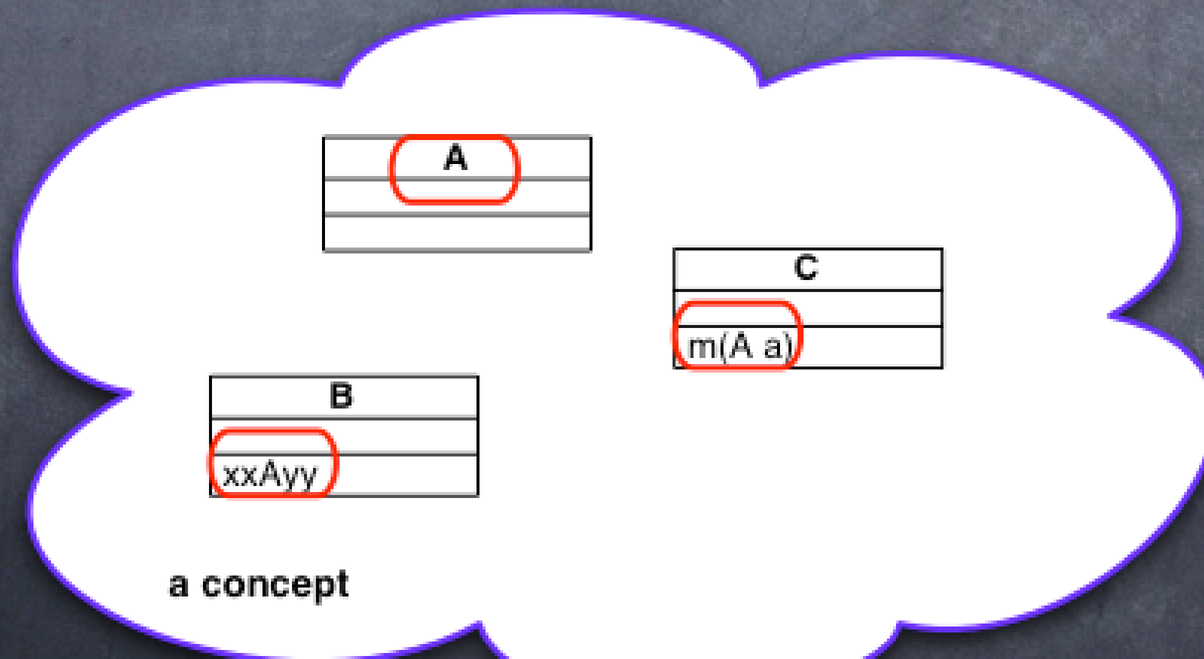
Aspectual View Mining with FCA

General idea

- Crosscutting concerns are often implemented using naming and coding conventions
- Assume that elements corresponding to a same concern have a similar name
- Try to group elements with similar names by using FCA

Schematically : Substring Concepts

- Elements : classes, methods, parameters
- Properties : substrings of classes, methods, ...



Overall approach

1. Generate the formal context
   - Elements, properties
2. Concept Analysis
   - Calculate the concepts
   - Organize them into a concept lattice
3. Filtering
   - Remove irrelevant concepts (false positives, noise, useless, ...)
4. Classification
   - Classify results according to relevance for user

The substring experiment
1. Generate the formal context

- We want to group elements that share a substring
- As elements we collect all classes, methods and parameters
- As properties we compute all "relevant" substrings of the names of those elements
  - Based on where uppercases occur in an element's name
    QuotedCodeConstant→ { quoted, code, constant }

The substring experiment
2. Concept Analysis

	unify	index	env	source	message	functo r	variabl e	...
Object>>unifyWithObject: inEnv: myIndex: hisIndex: inSource:	X	X	X	X	-		-	...
Variable>>unifyWithMessageFunctor: inEnv: myIndex: hisIndex: inSource:	X	X	X	X	X	X	-	...
AbstractTerm>>unifyWith: inEnv: myIndex: hisIndex: inSource:	X	X	X	X	-	-	-	...
AbstractTerm>>unifyWithVariable: inEnv: myIndex: hisIndex: inSource:	X	X	X	X	-	X	X	...
...	X	X	X	X	...	...	...	...

The substring experiment

2. Concept Analysis

	unify	index	env	source	message	functor	variable	…
Object>>unifyWithObject: inEnv: myIndex: hisIndex: inSource:	X	X	X	X	-		-	…
Variable>>unifyWithMessageFunctor: inEnv: myIndex: hisIndex: inSource:	X	X	X	X	X	X	-	…
AbstractTerm>>unifyWith: inEnv: myIndex: hisIndex: inSource:	X	X	X	X	-	-	-	…
AbstractTerm>>unifyWithVariable: inEnv: myIndex: hisIndex: inSource:	X	X	X	X	-	X	X	…
…	X	X	X	X	…	…	…	…

The substring experiment
3. Filtering

- Irrelevant substrings are already filtered
  - with little meaning : “do”, “with”, “for”, “from”, “the”, “ifTrue”, ...
  - too small (< 3 chars)
  - ignore plurals, uppercase and colons
- Extra filtering
  - Drop top & bottom concept when empty
  - Drop concepts with two elements or less
- More filtering needed (ongoing work)
  - Concepts higher in the lattice may be more relevant
    - More shared properties

The substring experiment
4. Classification

- Source code entities in single class
  - Accessors
  - Chained messages
  - Delegating methods
  - Similar signatures
- Source code entities in same hierarchy
  - Polymorphic methods
  - Similar signatures
  - Similar class names

The substring experiment

4. Classification

These seem most relevant when mining for aspects!

- “Crosscutting” source code entities
  - Polymorphic methods
  - Similar signatures
  - Similar class names
- Substring shared by method name & class name
- Substring shared by class name & parameter name

The substring experiment
Some quantitative results

Case study	#elements	#properties	#raw	#filtered	time (sec)
Soul	1469	434	1188	281	22
StarBrowser	527	266	491	73	4
CodeCrawler	1370	477	1419	327	24
DelfSTof	756	237	617	126	5
Ref.Browser	4779	729	4179	1234	414

- Remarks :
  - Time to compute = a few seconds / minutes
  - Still too much concepts remain after filtering

The substring experiment
Discovered aspectual views

- Programming idioms
  - Accessor methods (accessors)
  - Polymorphism (hierarchy methods)
- Design patterns (hierarchy methods)
  - Visitor, Abstract Factory, Observer
- Crosscutting features
  - "Unification" (hierarchy methods)
  - Crosscutting class-related behaviour (class name in keyword & class name in parameter)
  - "Bindings", "Horn clauses", "resolution" (unclassified)
- Opportunities for refactoring
  - Mainly code duplication

An aspectual view is a set of source code entities, such as classes, methods and parameters, that are structurally related and often crosscut the entire source code.

An example view

Star Browser on: #compoundVisit: (SimpleTermVisitor)

Services Help

Factory class (23)
Results class (2)
Frame class (2)
SimpleTermVisitor class (13)
SimpleTermVisitor
visit compound (4)
#compoundVisit: (SimpleTermVisitor)
#compoundVisit: (LexicalAddressVisitor)
#compoundVisit: (CopyingVisitor)
#compoundVisit: (CompoundTermRenamingV
cut visit (2)
#cutVisit: (SimpleTermVisitor)
#cutVisit: (CopyingVisitor)
object visit (3)
#objectVisit: (LexicalAddressVisitor)
#objectVisit: (CopyingVisitor)
#objectVisit: (SimpleTermVisitor)
visit constant (3)
#constantVisit: (CopyingVisitor)
#constantVisit: (LexicalAddressVisitor)
#constantVisit: (SimpleTermVisitor)
underscore visit variable (4)
#underscoreVariableVisit: (CopyingVisitor)
#underscoreVariableVisit: (NamedVariableVis
#underscoreVariableVisit: (SimpleTermVisitor
#underscoreVariableVisit: (VariableAndUnder
clause visit fact (2)
clause visit query (2)
clause rule visit (2)
visit sequence termsequence term (2)

Package | Hierarchy

Classifications
ConceptLattice
SCG StarBrow
SmaCC +
Soul +
SmalltalkAc
SoulGramm
SoulGramm

DepthFirstFram
DepthFirstRuleS
Environment
Factory
FailFrame
FixVisitor
Frame
NamedVariable\

Instance | Class | Shared Variable

visiting clauses
visiting terms

callTermVisit:
compoundVisit:
constantVisit:
cutVisit:
delayedVariable\
keywordFunctorV
messageFunctor'
multiPartFunctor\

Source | Rewrite | Code Critic | Statements

```
compoundVisit: aCompound
    aCompound functor accept: self.
    aCompound termSequence accept: self
```

Spawn results

Method: #compoundVisit: (vis | Parcel: none | Package: SoulKernel

Conclusion

- Substring experiment performed
  - Discovers interesting source-code regularities just based on names
  - Some refinement needed : mainly more advanced filtering
- Sufficient to detect aspects?
- Future work
  - Try to detect real aspects instead of aspectual views
  - Check it on a real aspect program : are the weaved aspects discovered by the approach?
  - Consider more dynamic information
    - E.g., examining the execution trace of the program
    - Perhaps in combination with examining the static structure