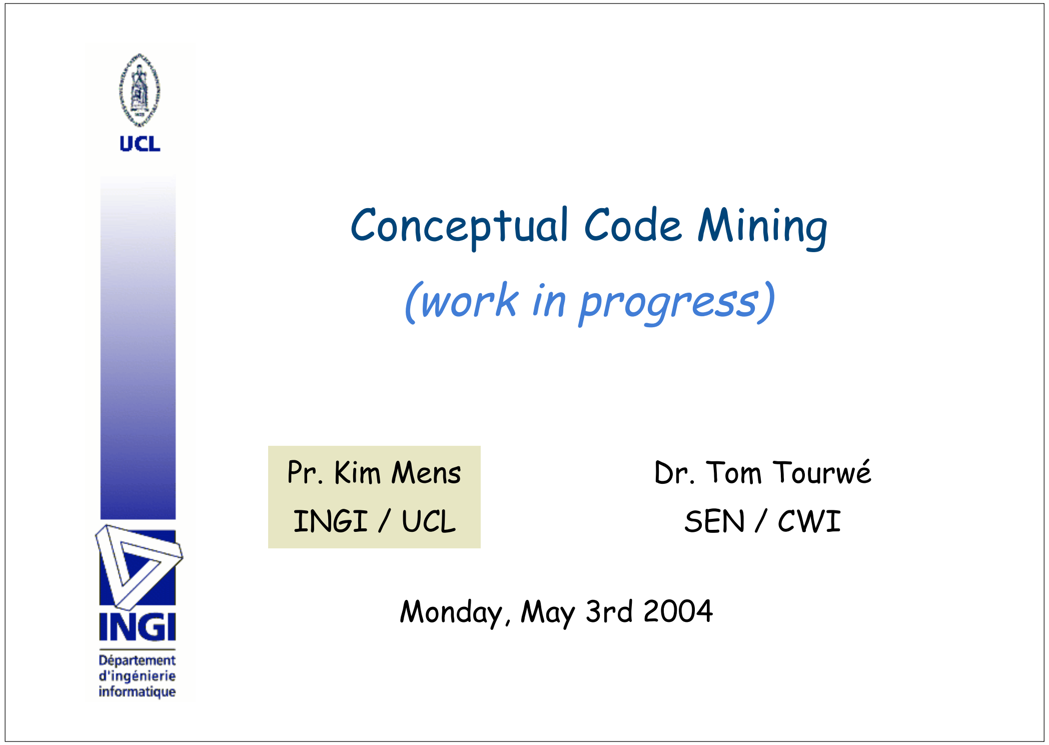


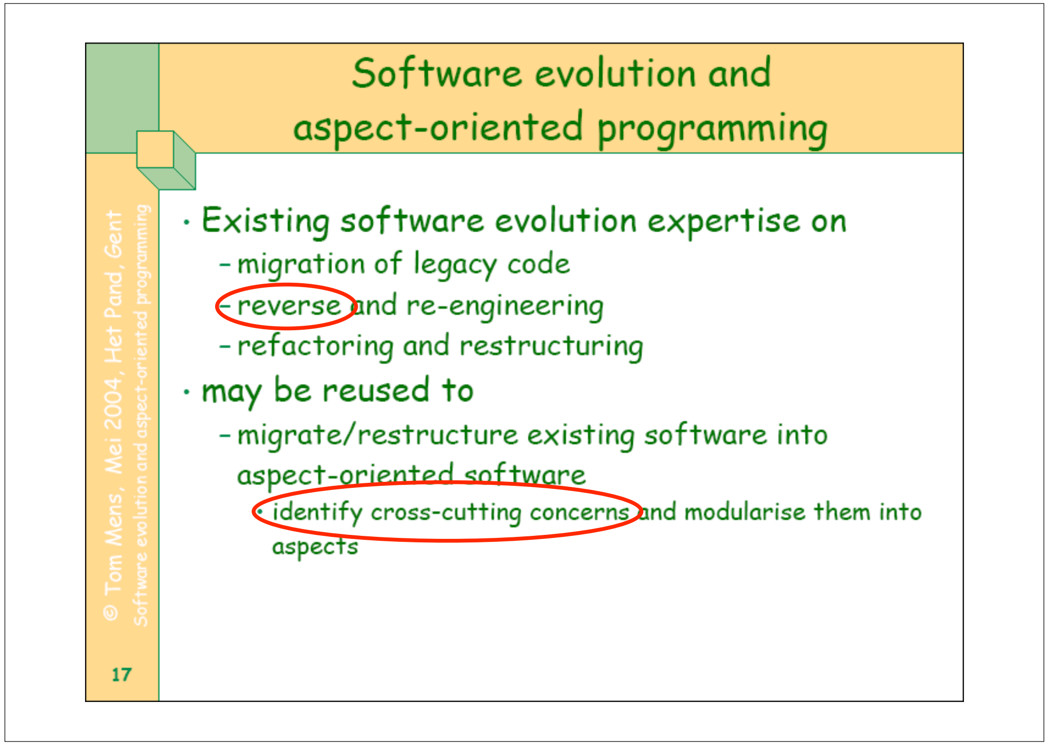
Conceptual Code Mining

(work in progress)

Pr. Kim Mens
INGI / UCL

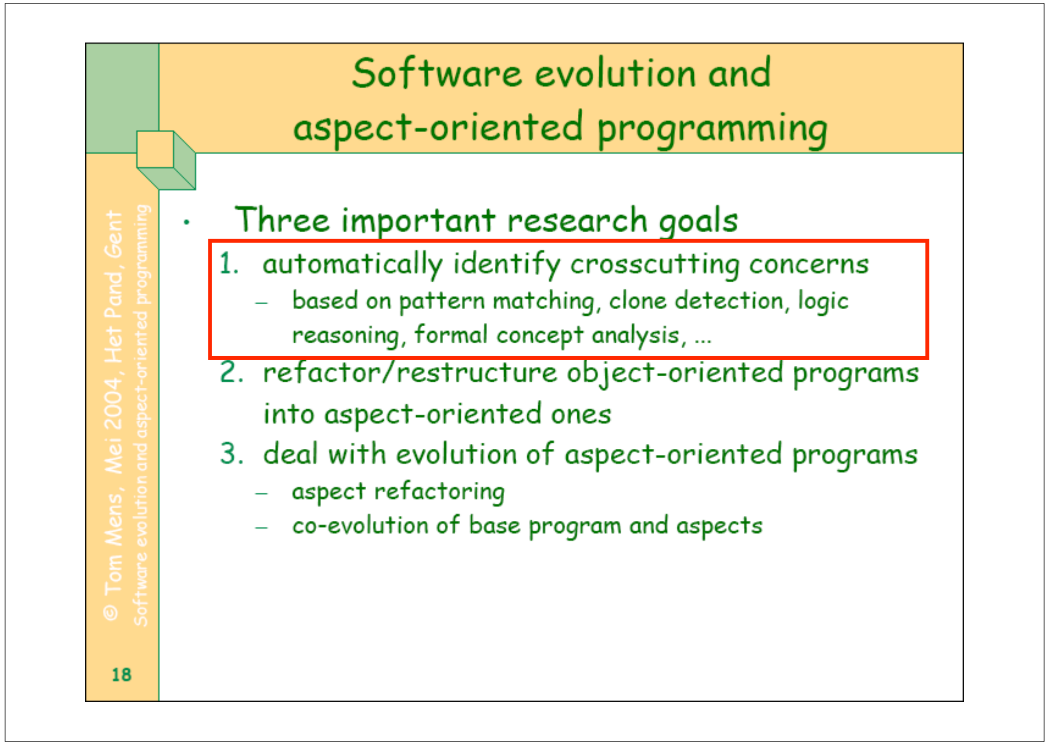
Dr. Tom Tourwé
SEN / CWI

Monday, May 3rd 2004



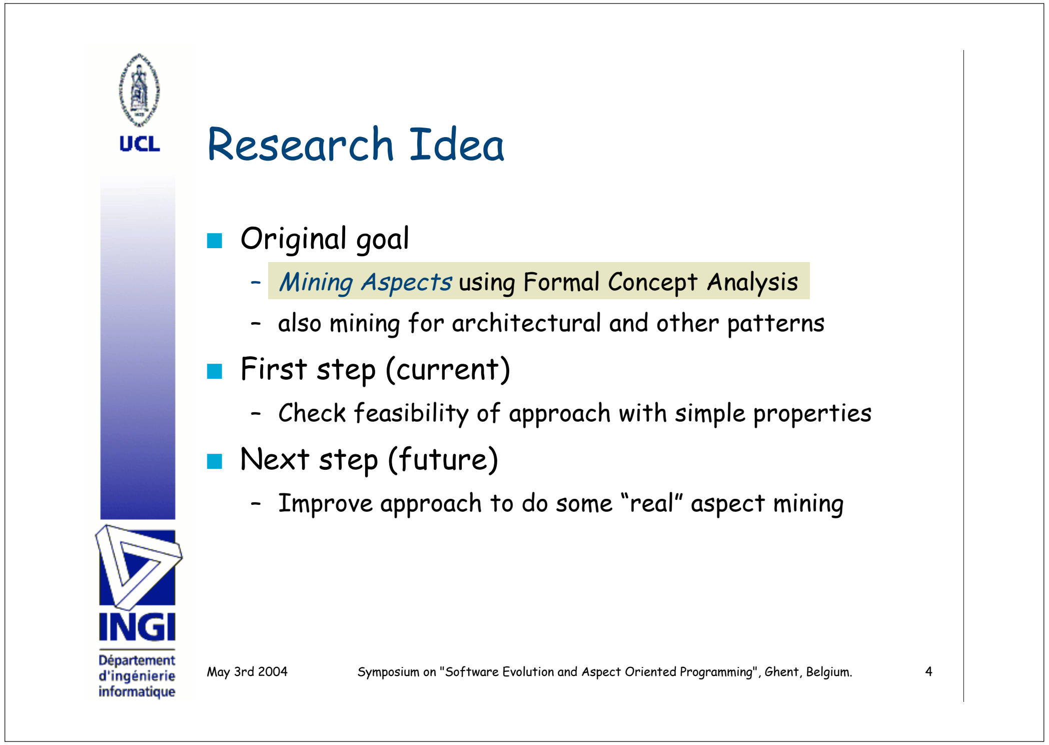
Software evolution and aspect-oriented programming

- Existing software evolution expertise on
  - migration of legacy code
  - reverse and re-engineering
  - refactoring and restructuring
- may be reused to
  - migrate/restructure existing software into aspect-oriented software
    - identify cross-cutting concerns and modularise them into aspects



Software evolution and aspect-oriented programming

- Three important research goals
  1. automatically identify crosscutting concerns
     - based on pattern matching, clone detection, logic reasoning, formal concept analysis, …
  2. refactor/restructure object-oriented programs into aspect-oriented ones
  3. deal with evolution of aspect-oriented programs
     - aspect refactoring
     - co-evolution of base program and aspects



Research Idea

- Original goal
  - *Mining Aspects* using Formal Concept Analysis
  - also mining for architectural and other patterns
- First step (current)
  - Check feasibility of approach with simple properties
- Next step (future)
  - Improve approach to do some “real” aspect mining

Overview

- Relation to “Software evolution and AOP”
- A crash course in formal concept analysis
- Mining for croscutting concerns with FCA
- Overall approach
- The substring experiment in detail
- The parsetree experiment
- Conclusion

Research idea :

Mining for croscutting concerns

using *Formal Concept Analysis*

Overview

- Relation to “Software evolution and AOP”
- A crash course in formal concept analysis
- Aspect mining with formal concept analysis
- Overall approach
- The substring experiment in detail
- The parsetree experiment
- Conclusion

A Brief Introduction to *AOP*

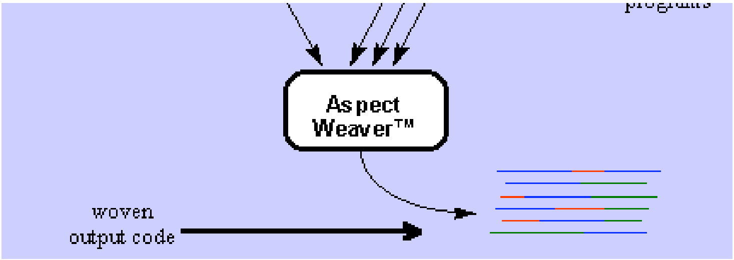
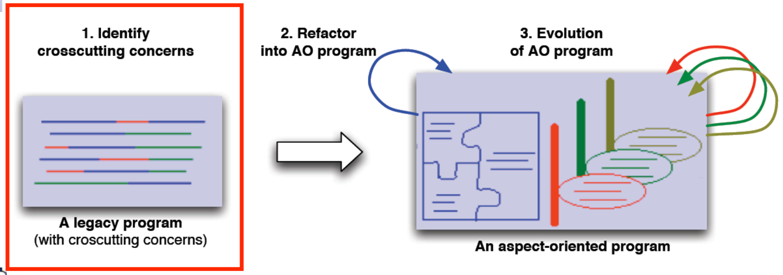


Figure 1 — The basic elements of an aspect-oriented programming system. The basic functionality (or primary aspect) is captured using a language that best suits it. Each of the cross-cutting aspects are captured using other appropriately specialized languages. The weaver takes all the programs as input and produces woven output code, which may itself be source code in a language like C.

Software Evolution and Aspect-Oriented Programming


1. Identify crosscutting concerns
A legacy program (with croscutting concerns)
2. Refactor into AO program
3. Evolution of AO program
An aspect-oriented program

Overview

- Relation to “Software evolution and AOP”
- A crash course in formal concept analysis
- Mining for croscutting concerns with FCA
- Overall approach
- The substring experiment in detail
- The parsetree experiment
- Conclusion

Formal Concept Analysis (*FCA*)

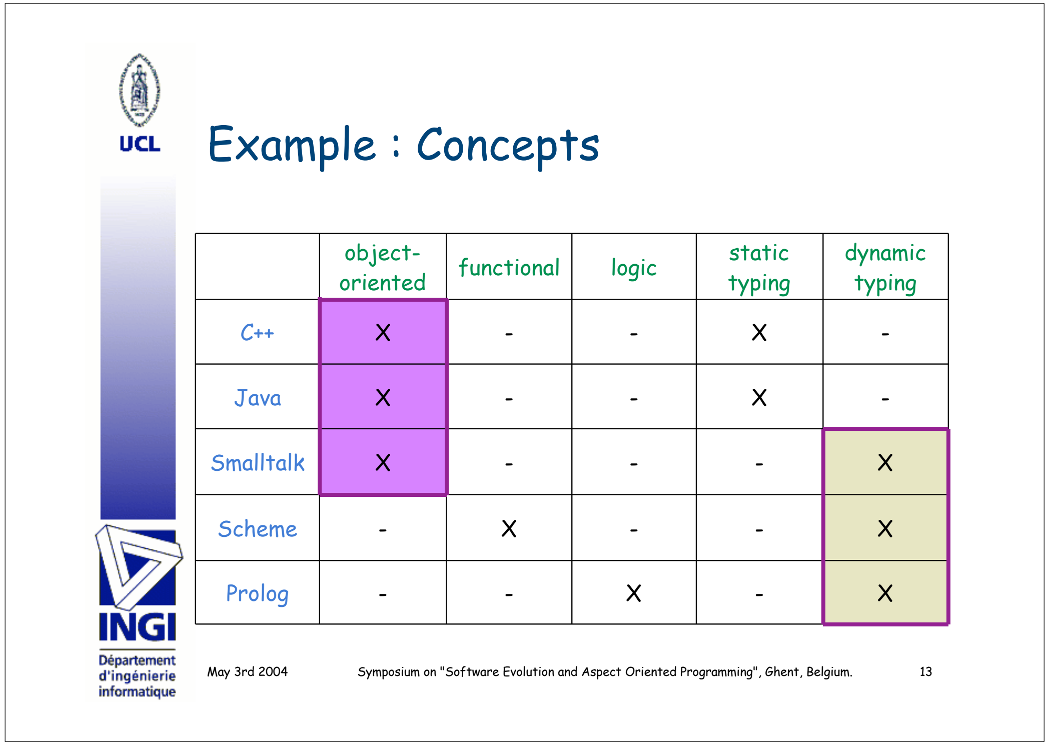
- Starts from
  - a set of elements
  - a set of properties of those elements
- Determines concepts
  - Maximal groups of elements and properties
  - Group:
    - Every element of the concept has those properties
    - Every property of the concept holds for those elements
  - Maximal
    - No other element (outside the concept) has those same properties
    - No other property (outside the concept) is shared by all elements

Example : Elements and Properties

	object-oriented	functional	logic	static typing	dynamic typing
C++	X	-	-	X	-
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X

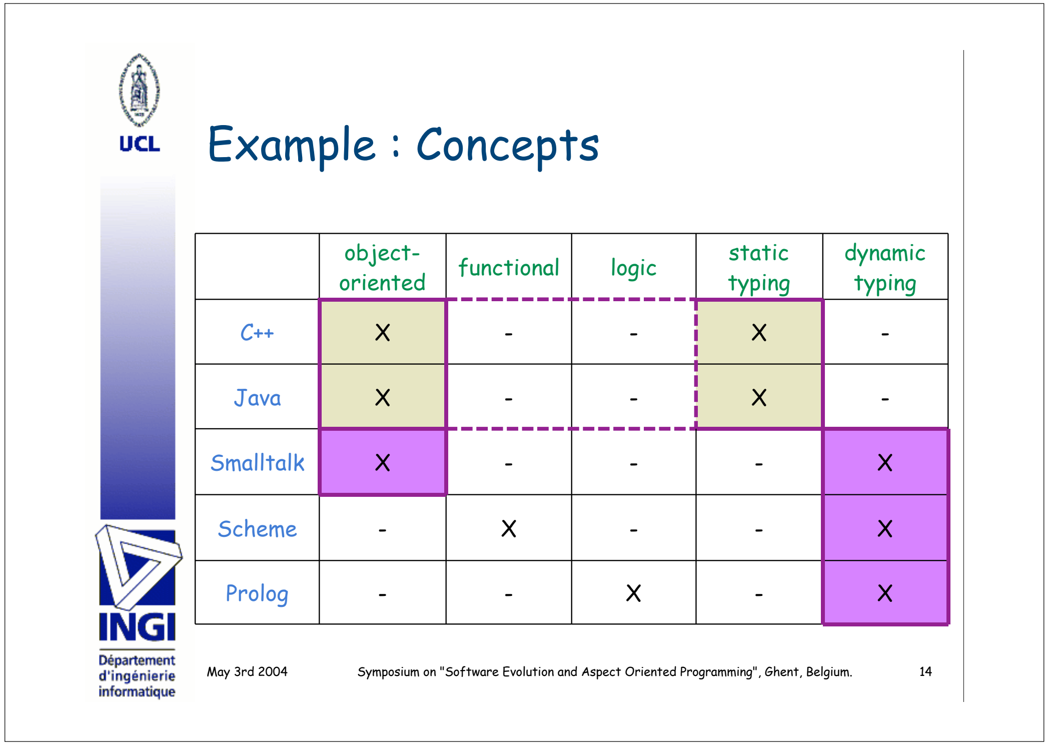
Example : Concepts

	object-oriented	functional	logic	static typing	dynamic typing
C++	X	-	-	X	-
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X



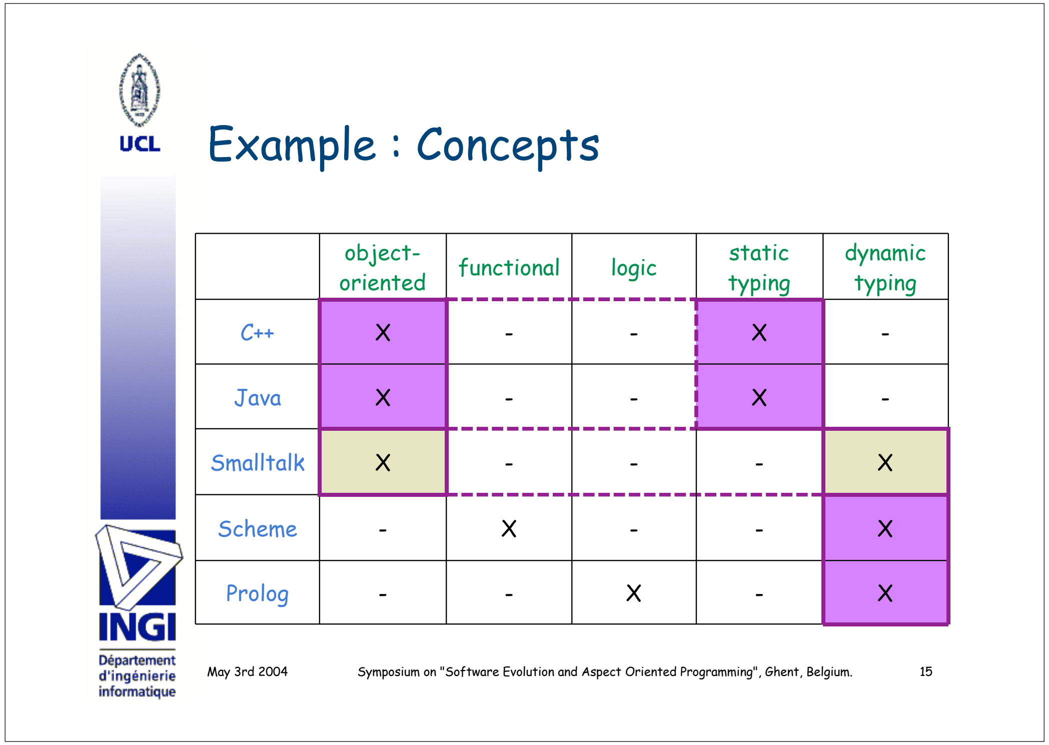
Example : Concepts

	object-oriented	functional	logic	static typing	dynamic typing
C++	X	-	-	X	-
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X



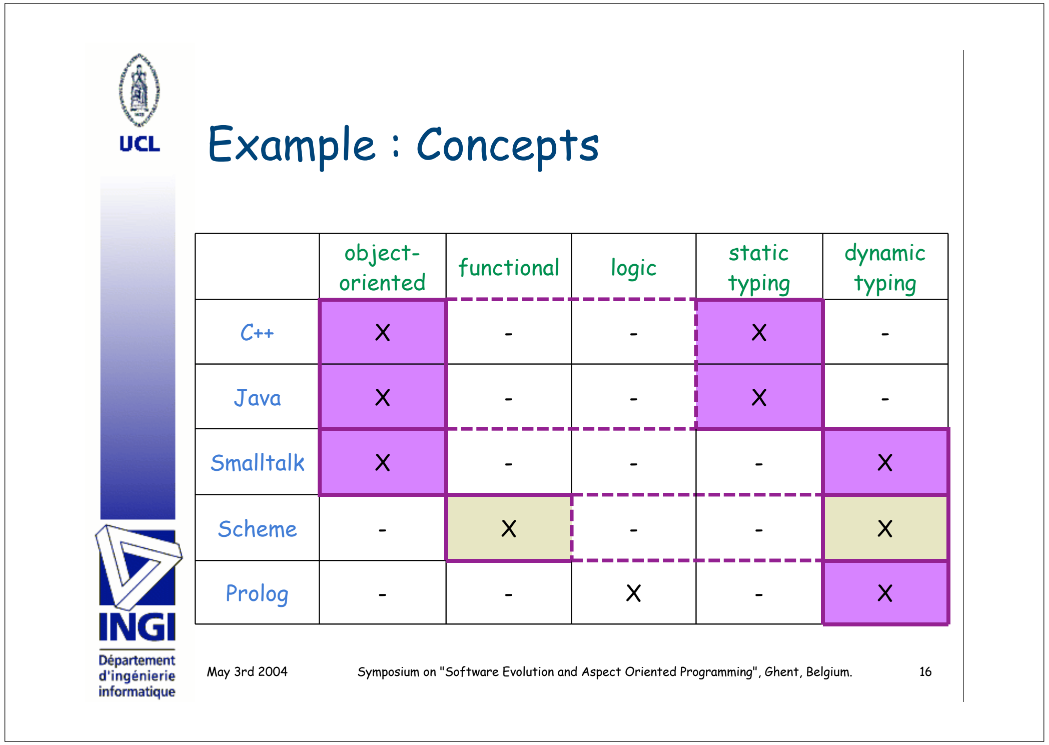
Example : Concepts

	object-oriented	functional	logic	static typing	dynamic typing
C++	X	-	-	X	-
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X



Example : Concepts

	object-oriented	functional	logic	static typing	dynamic typing
C++	X	-	-	X	-
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X



Example : Concepts

	object-oriented	functional	logic	static typing	dynamic typing
C++	X	-	-	X	-
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X

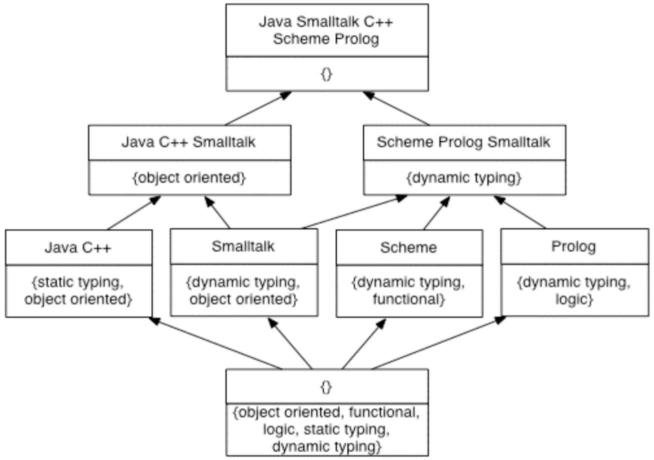
Example : Concepts

	object-oriented	functional	logic	static typing	dynamic typing
C++	X	-	-	X	-
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X

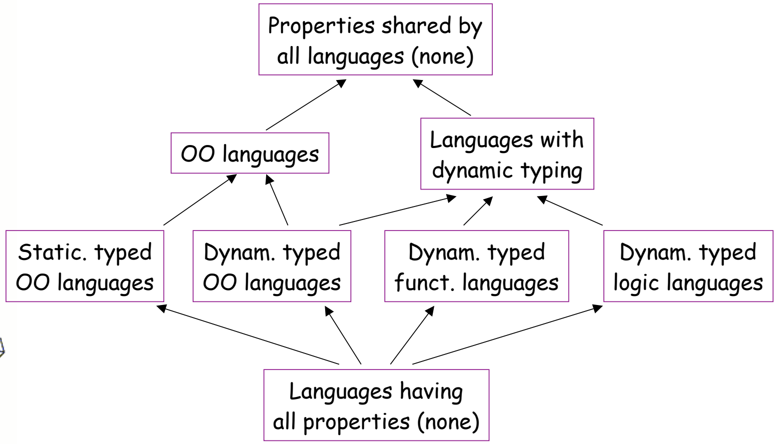
Example : Concepts

	object-oriented	functional	logic	static typing	dynamic typing
C++	X	-	-	X	-
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X

Concept Lattice


Java Smalltalk C++ Scheme Prolog
{}
Java C++ Smalltalk
{object oriented}
Scheme Prolog Smalltalk
{dynamic typing}
Java C++
{static typing, object oriented}
Smalltalk
{dynamic typing, object oriented}
Scheme
{dynamic typing, functional}
Prolog
{dynamic typing, logic}
{}
{object oriented, functional, logic, static typing, dynamic typing}

Discovered Concepts


Properties shared by all languages (none)

OO languages

Languages with dynamic typing

Static. typed OO languages

Dynam. typed OO languages

Dynam. typed funct. languages

Dynam. typed logic languages

Languages having all properties (none)

Overview

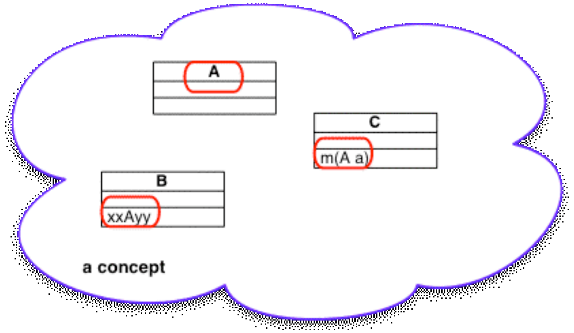
- Relation to “Software evolution and AOP”
- A crash course in formal concept analysis
- Mining for croscutting concerns with FCA
- Overall approach
- The substring experiment in detail
- The parsetree experiment
- Conclusion

Mining for crosscutting concerns with formal concept analysis

- First Step
  - Use substrings of class, method & parameter names to group related source code elements
  - Relies on coding conventions
  - Assumes that elements corresponding to a same concern will have a *similar name*
- Next step
  - Use generic parse trees to group source code that implements similar behaviour
  - Looks for recurring patterns in the source code
  - Similar to clone detection, but more advanced
  - Assumes that elements corresponding to a same concern will have *similar code*

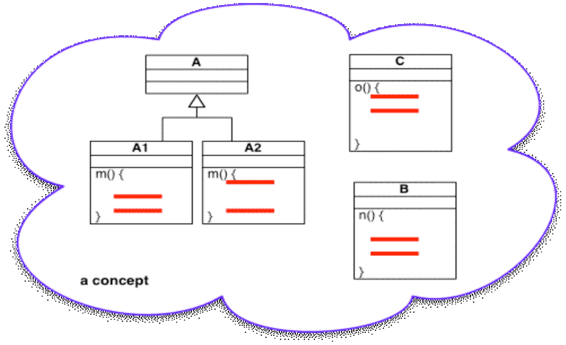
Substring Concepts

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



Parse tree Concepts

- Elements : methods
- Properties : generic parse tree elements



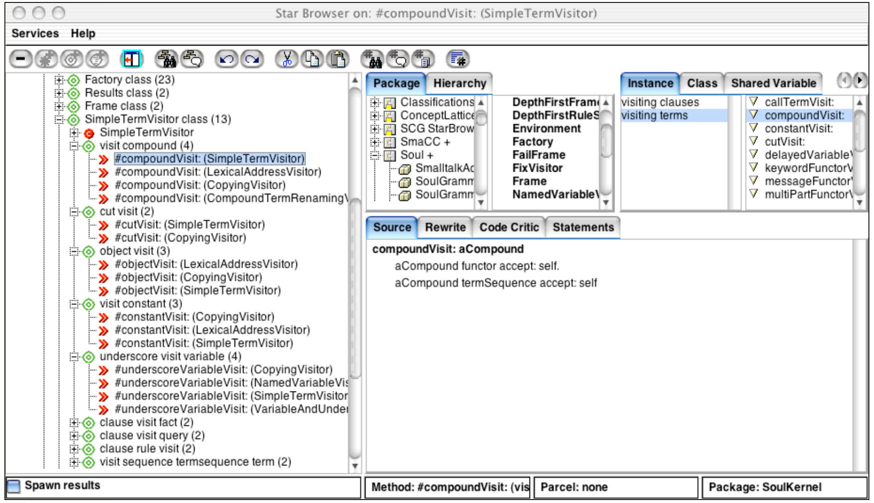
Overview

- Relation to “Software evolution and AOP”
- A crash course in formal concept analysis
- Mining for croscutting concerns with FCA
- Overall approach
- The substring experiment in detail
- The parsetree experiment
- Conclusion

Overall approach

1. Generate elements & properties for FCA algorithm
   - ✓ Pre-filter irrelevant ones
2. Concept Analysis
   - ✓ Find relevant groupings of elements in source code
3. Filtering
   - ✓ Remove irrelevant concepts (false positives, noise, useless, …)
4. Classification
   - ✓ Classify results according to relevance for user
5. Analyse unclassified concepts
   - ✓ Manually analyse concepts that were not classified automatically
6. Completion of concepts
   - ✓ Some concepts are relevant
     but need to be completed to represent reality correctly

Our Conceptual Code Mining Tool



```
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)
```

```
Classifications      DepthFirstFram      visiting clauses    callTermVisit:
ConceptLattice       DepthFirstRuleS     visiting terms      compoundVisit:
SCG StarBrow         Environment                             constantVisit:
SmaCC +              Factory                                 cutVisit:
Soul +               FailFrame                               delayedVariable\
  SmalltalkAc        FixVisitor                              keywordFunctorV
  SoulGramm          Frame                                   messageFunctor'
  SoulGramm          NamedVariable\                          multiPartFunctorV
```

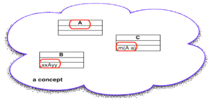
```
compoundVisit: aCompound
	aCompound functor accept: self.
	aCompound termSequence accept: self
```

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

Overview

- Relation to “Software evolution and AOP”
- A crash course in formal concept analysis
- Mining for croscutting concerns with FCA
- Overall approach
- The substring experiment in detail
- The parsetree experiment
- Conclusion

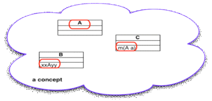
The substring experiment



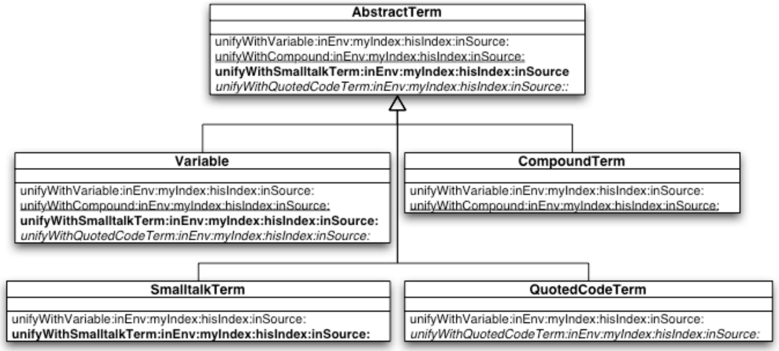
1. Generate elements & properties

- We want to group elements that share a substring
- Problem :
  - "Having a substring in common" is *binary*
  - FCA properties are *unary*
    - Does an element satisfy the property or not?
- Solution :
  - Every substring corresponds to an FCA property
    - Does an element have this substring in its name?
  - Generate relevant substrings
    - Based on where uppercases occur in an element's name
      - QuotedCodeConstant → { quoted, code, constant }
    - Filter substrings that produce too much noise

The substring experiment



2. Concept Analysis - a concept (1)



AbstractTerm

unifyWithVariable:inEnv:myIndex:hisIndex:inSource:
unifyWithCompound:inEnv:myIndex:hisIndex:inSource:
unifyWithSmalltalkTerm:inEnv:myIndex:hisIndex:inSource
unifyWithQuotedCodeTerm:inEnv:myIndex:hisIndex:inSource::

Variable

unifyWithVariable:inEnv:myIndex:hisIndex:inSource:
unifyWithCompound:inEnv:myIndex:hisIndex:inSource:
unifyWithSmalltalkTerm:inEnv:myIndex:hisIndex:inSource:
unifyWithQuotedCodeTerm:inEnv:myIndex:hisIndex:inSource:

CompoundTerm

unifyWithVariable:inEnv:myIndex:hisIndex:inSource:
unifyWithCompound:inEnv:myIndex:hisIndex:inSource:

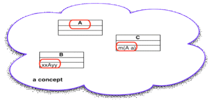
SmalltalkTerm

unifyWithVariable:inEnv:myIndex:hisIndex:inSource:
unifyWithSmalltalkTerm:inEnv:myIndex:hisIndex:inSource:

QuotedCodeTerm

unifyWithVariable:inEnv:myIndex:hisIndex:inSource:
unifyWithQuotedCodeTerm:inEnv:myIndex:hisIndex:inSource:

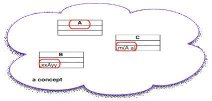
The substring experiment



2. Concept Analysis - a concept (2)

	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



2. Concept Analysis - a concept (2)

	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

2. Concept Analysis - some numbers

Case study	#elements	#properties	#raw concepts	#combined concepts	time (sec)
Soul	1469	439	1197	593	29
StarBrowser	512	262	500	196	5
CodeCrawler	1370	478	1502	699	37
CA tool	750	238	656	347	7

- Remarks :
  - Without filtering
  - | properties | < | elements | is a good sign
  - Time to compute = a few seconds
  - Lots of noise and some false positives
    - Better filtering & classification needed

The substring experiment

2. Concept Analysis - some numbers

- Same experiment but now with “structured substrings”
  - also consider combinations of primitive substrings
  - more confidence that you get better results
  - but more properties, since we keep the primitive ones too
  - More properties, concepts and computation
  - *But we hope that we can do better filtering on these*

Case study	#elements	#properties	#raw concepts	#combined concepts	time (sec)
Soul	1469	3427	1602	782	106
StarBrowser	512	1341	618	249	8
CodeCrawler	1370	478	1898	854	131
CA tool	750	1615	863	449	17

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
- More filtering needed
  - Drop top & bottom concept when empty
  - Drop concepts with only one element
  - Recombine substrings belonging together
  - Require some minimal coverage of element name by properties
  - Concepts higher in the lattice (more properties) may be more relevant
  - Avoid redundancy in discovered concepts
    - Make better use of the lattice structure (Now it is “flattened”)
- Ongoing work

The substring experiment

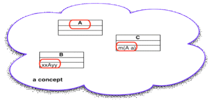
4. Classification

- In single class
  - Accessors
  - Chained messages
  - Delegating methods
  - Similar signatures
- Too few elements
- In same hierarchy
  - Polymorphic methods
  - Substring shared by method name & parameter name
  - Similar signatures
  - Similar class names

- Croscutting
  - Polymorphic methods
  - Substring shared by method name & parameter name
  - Similar signatures
  - Similar class names
- Substring shared by method name & class name
- Substring shared by class name & parameter name
- Unclassified

These seem most relevant when mining for concerns

The substring experiment

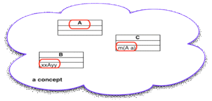


5. Unclassified Concepts

- Some concepts remain unclassified
  - but may still provide useful insights
  - e.g., for code comprehension
- Unclassified concepts for SOUL:
  - Everything that has to do with

√ Stacks	√ Lists
√ Bindings	√ Pairs
√ Horn clauses	√ Resolution
√ Native clauses	√ Term sequence

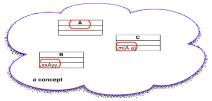
  - They seem to address a common feature



6. Completion

- Discovered classifications may require completion
  - E.g., we may discover an interesting set of polymorphic methods
  - But some methods are missing because, e.g.,
    - Their implementing class does not adhere to the right naming convention
    - One of their parameters they had was named differently
  - These classifications should be completed "a posteriori"
    - Can this be done (semi) automatically?

The substring experiment



Discovered aspectual views (Soul)

- Programming idioms
  - Accessor methods (*accessors*)
  - Polymorphism (*hierarchy methods*)
- Design patterns (*hierarchy methods*)
  - Visitor
  - Abstract Factory
- Features
  - "Unification" (*hierarchy methods*)
  - Crosscutting class-related behaviour (*class name in keyword* & *class name in parameter*)
  - "Bindings", "Horn clauses", "resolution" (*unclassified*)
- Code duplication

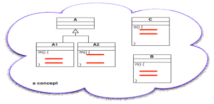
  (*methods in single class* & *crosscutting methods*)

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.

Overview

- Relation to "Software evolution and AOP"
- A crash course in formal concept analysis
- Mining for croscutting concerns with FCA
- Overall approach
- The substring experiment in detail
- The parsetree experiment
- Conclusion

Parse Tree Concepts



- Idea :
  - Use FCA to structure methods hierarchically according to their shared parse trees
- Technique :
  - We “abuse” some functionality provided by the *rewrite rule editor* of the *refactoring browser*
  - Allows us to write partial code, parameterized with an @ for those parse tree nodes that we want to leave generic
    - Variable names, expressions, …
- Experiment ongoing

Overview

- Relation to “Software evolution and AOP”
- A crash course in formal concept analysis
- Mining for croscutting concerns with FCA
- Overall approach
- The substring experiment in detail
- The parsetree experiment
- Conclusion

Conclusion

- Current status
  - Substring experiment already performed, but needs refinement
    - Mainly more advanced filtering
  - Parse tree experiment seems promising complement / extension to already existing experiment
  - Enough to detect aspects?
- Future work
  - Work out parse tree experiment
  - Check it on a real aspect program : are the weaved aspects discovered by the approach?