2IS55 Software Evolution

Implementing evolution: Aspect-Oriented Programming

Alexander Serebrenik

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Last week

- Assignment 8
  - How is it going?
  - Questions to Marcel: m.f.v.amstel@tue.nl
  - Deadline: Tuesday, June 22, 2010
- Today
  - 1st hour: Evolution and aspect-oriented programming
  - 2nd hour: General discussion of software evolution

Sources

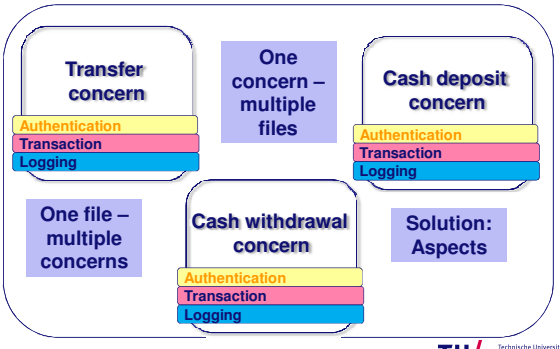
Systems are too large…

- Concern – an area of interest or focus in a system.
- Primary criteria for decomposing software
  - Object-oriented: concern = object
  - Procedural: concern = procedure
  - Service-oriented: concern = service
  - Functional: concern = function
- Separation of concerns:
  - each program element implements one concern only

Bank Transfer Example

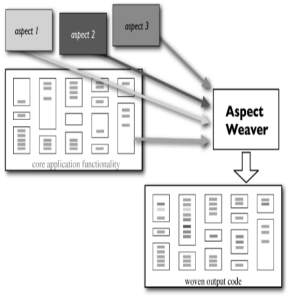
```
void transfer(Account toAccount, int amount) throws Exception {
f (!getCurrentUser().canPerform(OP_TRANSFER))          Authentication
        { throw new SecurityException(); }
Transaction tx = database.newTransaction();            Transaction
ry {
    if (this.getBalance() < amount)
        { throw new InsufficientFundsException(); }
    this.withdraw(amount);
    toAccount.deposit(amount);
    tx.commit();                                       Transaction
    systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
    }                                                  Logging
catch(Exception e) {
    tx.rollback();                                     Transaction
    throw e; }
```

Problem



Aspects

- Used in conjunction with other techniques (OO)
- Aspects implement cross-cutting concerns
- Aspect
  - advice – the code to be added
  - point cut – set of join points
    - code locations were the code will be woven



Example: Core application functionality

```
void transfer(Account toAccount, int amount) throws Exception {
try {                                                    Authentication
                                                         Transaction
    if (this.getBalance() < amount)
        { throw new InsufficientFundsException(); }
    this.withdraw(amount);
    toAccount.deposit(amount);                           Logging
    }                                                    Transaction
catch(Exception e) {
    throw e;                                             Transaction
    }
}
```

Point cuts: Where

- Extensional

```
pointcut authenticate(Account):
  (call (public void Account.transfer (Account, int))
  || call (public real Account.getBalance()) )
```

  - Problem: breaks when the core functionality evolves

- Intentional

```
pointcut authenticate(Account):
   call (public * Account.* (..))
```

  - All calls to public methods of Account
  - Better suited for evolution

Advice: What

Label (before/after returning/after throwing…)

Name of the pointcut

Java code

```
before(Account acc): authenticate(acc)  {
   int tries = 0 ;
   string userPwd = UI.GetPwd ( tries ) ;
   while (tries < 3 && userPwd != acc.pwd ( ) ) {…}
}
```


Aspects and evolution

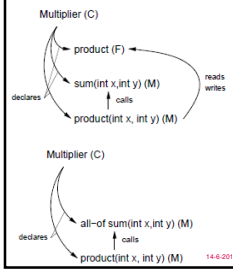
- From legacy to aspects
  - Aspect exploration
    - identify crosscutting concerns
    - propose aspect candidates
  - Aspect extraction
    - disentangle the code
    - ensure correctness of the result
- Once we are there…
  - Aspect evolution

Aspect exploration

- Early aspect discovery
  - Even before the code has been built
  - Requirements, domain analysis, architecture design
- Dedicated browsers
  - Location in the code belonging to the concern: seed
  - Browser suggests locations that can belong to the same concern
- Aspect mining
  - Automate aspect identification
  - Static: code
  - Dynamic: logs

Dedicated browsers: Concern Graphs
[Robillard, Murphy]

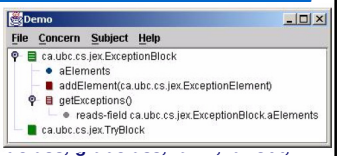
- Structural program model: graph
  - Nodes: classes, methods, fields
  - Arcs: calls, reads, writes, declares, superclass, creates and checks



- Concern graphs
  - If all subelements belong to the concern: "all-of"
  - If some subelements belong to the concern: "part-of"
  - Drop elements that do not belong to the concern

Dedicated browsers: Concern Graphs
[Robillard, Murphy]

- Start with a seed
- Select a node and explore it with built-in operations:
  - getSuperclass, expar… transitiveFanOut
  - grabClass(reads-field C.f) = C
- Label new nodes:
  - Filled rectangle: belongs to the concern
  - Striped rectangle: partially belongs to the concern



How successful was the approach?

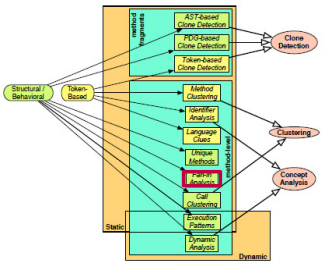
- Case study: Jex
- Oracle: the author of Jex
- 3 participants were asked to identify the concerns

Participant	1	2	3
Classes found (8)	7	6	8
Field found (1)	1	0	0
Methods found (15)	13	7	11
Code elements found (12)	11	3	7
False positives	2	0	0

- < 50 minutes
- Low number of false positives
- ~ 80% of the code viewed was relevant

Aspect mining: What can form a concern?

- A method called from "everywhere"
  - High fan-in
  - Unique functionality
- Group of related code fragments
  - Classes/methods with similar names
  - Similar methods being called
- Similar behaviour (call to A always followed by a call to B)
- Similar code (cloning)



Fan-in based detection

- Calculate fan-in for all methods
  - JHotDraw: 2800 methods
- Filter out methods with fan-in < threshold
  - JHotDraw: threshold = 10, 7% of the methods retained
- Filter out "non-interesting" methods
  - Access methods (get*/set*)
  - Utility methods (toString)
  - JHotDraw: half of the methods retained
- Manual inspection
  - JHotDraw: 52% of the remaining methods can be used as aspect seeds

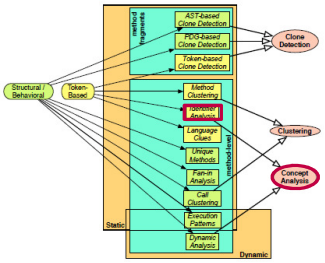
Fan-in results

- Multiple unrelated instances of the same type

Concern type	#	Seed's description
Consistent behavior	4	Methods implementing the consistent behavior shared by different callers, such as checking and refreshing figures that have been changed when executing a command.
Contract enforcement	4	Method implementing a contract that needs to be enforced, such as checking the reference to the editor's active view before executing a command.
Undo	1	Methods checking whether a command is undoable/redoable and the *undo* method in the super-class which is invoked from the overriding methods in subclasses.
Persistence and resurrection	1	Methods implementing functionality common to persistent elements, such as read/write operations for primitive types wrappers (e.g., Double, Integer, etc.) which are referenced by the scattered implementations of persistence/resurrection.
Command design pattern	1	The *execute* method in the command classes and command constructors.
Observer design pattern	1	The observers' manipulation methods and *notify* methods in classes acting as subject.
Composite design pattern	2	The composite's methods for manipulating child components, such as adding a new child.
Decorator design pattern	1	Methods in the decorator that pass the calls on to the decorated components.
Adapter design pattern	1	Methods that manipulate the reference from the adapter(*Handle*) to the adaptee(*Figure*).

Aspect mining: What can form a concern?

- A method called from "everywhere"
  - High fan-in
  - Unique functionality
- Group of related code fragments
  - Classes/methods with similar names
  - Similar methods being called
- Similar behaviour (call to A always followed by a call to B)
- Similar code (cloning)

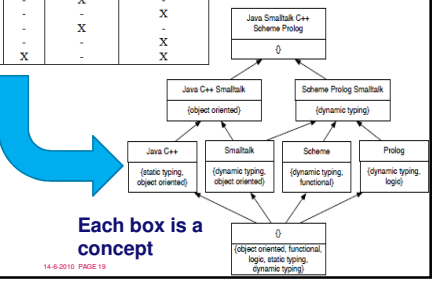


Example of the Formal Concept Analysis

Input: Objects and Attributes

PL	OO	Functional	Logic	Static typing	Dynamic typing
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
C++	X	-	-	X	-
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X

Formal concept analysis



Each box is a concept

Application of FCA [Tourwé Mens 2004]

- Objects: classes and methods
  - Except for test classes and accessor methods
- Attributes
  1. Split: createUndoableActivity ⇒ create, undoable, activity
  2. Strip suffices: undoable ⇒ undo [Porter 1980]
     - Technique based on distinguishing consonant/vowel patterns and simplification rules
     - Pattern: $[C](VC)^m[V]$
       - C, V – sequences of consonants/vocals
       - [] - optionality
     - Simplification rules (multiple rules can be applied):
       - "(m > 1) ABLE →"

From FCA to Concerns

- FCA identifies concepts as groups of classes/methods
- Performance
  - JHotDraw ~18K SLOC, 2193 objects, 507 attributes
  - ≥4 methods in a concept: 31 sec, 230 concepts
- Manual selection of relevant concepts as concerns
  - 41 concerns retained

Croscutting concern	Concept(s)	#elements	Some elements
Observer	change(d) / check / listener / release	67 / 14 / 65 / 12	figureChanged(e) / checkDamage() / createDesktopListener() / . . .
Undo	undo(able) / redo(able)	53 / 14	createUndoActivity() / redo()
Visitor	visit	12	visit(FigureVisitor)
Persistence	file / storable / load / register	15 / 5 / 8 / 7	registerFileFilters(c) / readStorable() / loadRegisteredImages
Drawing figures	draw	112	draw(g)
Moving figures	move	36	moveBy(x,y), moveSelection(dx,dy)
Iterating over collections	iterator	5	iterator(), listIterator(), . . .

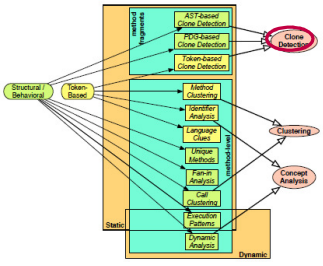
Fan-In vs. Identifier analysis: JHotDraw case

Concern	Fan-In Analysis	Identifier Analysis
Observer	+	+
Consistent behavior / Contract enforcement	+	-
Command execution	+	+
Bring to front / Send to back	-	-
Manage handles	-	+
Move Figures	+ (discarded)	+

- Both approaches still involve human judgement
  - "Moving figures" was not considered as a concern by the Fan-In team
- Different techniques reveal different concerns
- Some concerns are detected by both techniques

Aspect mining: What can form a concern?

- A method called from "everywhere"
  - High fan-in
  - Unique functionality
- Group of related code fragments
  - Classes/methods with similar names
  - Similar methods being called
- Similar behaviour (call to A always followed by a call to B)
- Similar code (cloning)



Code cloning: Do you still remember?

- Popular techniques:
  - Token-based (Baker (Dup), Kamiya (CCFinder))
  - AST-based (Baxter)
    - In combination with tokens: Koschke
  - Program Dependence Graph-based (Krinke)
- General study [Roy, Cordy, Koschke 2009]
  - Brief summary. 6 is maximal grade, 0 – minimal

			S1	S2	S3
Dup	Baker	Token	4	2.8	0
CCFinder	Kamiya		5	3.8	0.8
CloneDr	Baxter	AST	6	4.3	3.8
cpdetector	Koschke		6	3.8	0
Duplix	Krinke	Graph	5	4.8	4

Clone detection for aspect mining

- Case study: 19K LOC
- 5 concerns labelled by an expert
  - memory handling, null pointer checking, range checking, exception handling, tracing
- Code clones have been detected
- Precision = correct concern parts / all found
- Greedy algorithm
  - select clone groups such that precision is as good as possible

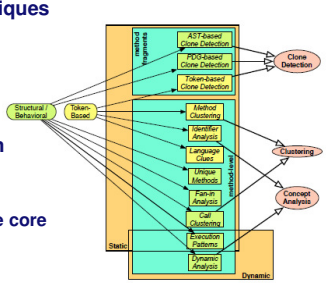
Clone detection for aspect mining

- Null pointer handling is quite "separate" from the core functionality
- Exception handling is intertwined with the core functionality
- AST and Graph are still leading but Tokens are not far behind

	Precision				Recall		
	AST	Token	PDG	AST+PDG	AST	Token	PDG
Memory	.65	.63	.81	.83	.96	.95	.98
Null	.99	.97	.80	.99	1.0	1.0	1.0
Range	.71	.59	.42	.72	.89	.96	.92
Exception	.38	.36	.35	.53	.79	.97	.95
Tracing	.62	.57	.68	.78	.76	.85	.90

Aspect mining: Summary

- Wide variety of techniques
- All involve manual inspection
- Precision depends on
  - the kind of concern considered
  - "separation" from the core functionality



Aspects and evolution

- From legacy to aspects
  - Aspect exploration
    - identify crosscutting concerns
    - propose aspect candidates
  - ➢ Aspect extraction
    - disentangle the code
    - ensure correctness of the result
- Once we are there…
  - Aspect evolution

First step: disentangle the concerns

- Manually using refactoring
  - Extract method
  - Replace Temp by Query
- Alternative: extract slice as a method
  - Slice = series of program statements that can affect the value of the slicing criterion (given variable at the given program point)

```
int i;
int sum = 0;
int product = 1;
for(i = 0; i < N; ++i) {
    sum = sum + i;
    product = product *i;
}
write(sum);
write(product);
```

```
int i;
int sum = 0;
for(i = 0; i < N; ++i) {
    sum = sum + i;
}
write(sum);
```

Join points

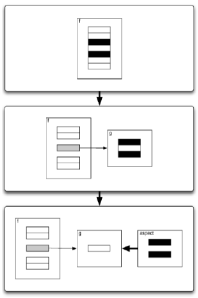
- Intention: add aspects at an arbitrary point
- Problem: limited support of AOP solutions
  - AspectJ: before, after returning, after throwing, after (finally), around (method call)
  - Impossible to weave aspects around a control structure
  - Before cannot prevent the execution unless an exception is thrown
  - Impossible to redefine before-advice in a subaspect
  - …

Solution? First attempt

- Extend the number of join points
  - At every edge in the control graph
  - With "correct" inheritance mechanism
  - …
- Complex join point model ⇒ performance penalty
- Local extensions are being proposed…

Solution? Second attempt

- Add more join points
- Extract a new method: g
  - before and after
- Separate the "black code" as an aspect
- Problem:
  - Core functionality should not have been aware of the aspects!



Point cuts

- Recall: point cut = set of join points
- Simple solution:
  - Extensional

```
pointcut authenticate(Account):
  (call (public void Account.transfer (Account, int))
  || call (public real Account.getBalance()) )
```

    - Easy to infer, but brittle
  - Intentional
    - Using machine learning techniques
    - Similar to Kim, Notkin on program differencing

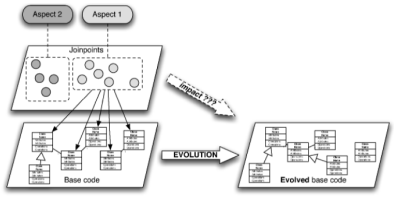
Aspect Extraction in Practice

- "Simple" concern: tracing
- Goal: extract the simplest aspect that describes the way the tracing concern is implemented
- Starting point: "standard way" prescribed by the company
- Observed: 5.7% of the functions adhere to the "standard way"
- Automatic aspect extraction techniques cannot cope with this variability!

Aspects and evolution

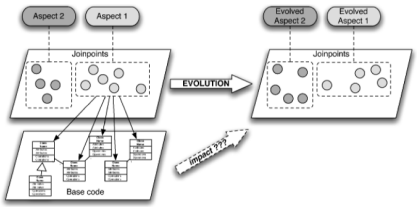
- From legacy to aspects
  - Aspect exploration
    - identify crosscutting concerns
    - propose aspect candidates
  - Aspect extraction
    - disentangle the code
    - ensure correctness of the result
- Once we are there…
  - ➢ Aspect evolution
    - ➢ In fact, co-evolution

Evolution of the base code can affect aspects



- Impact of the core code on the aspects
  - Core methods/classes are added/removed
  - Corresponding join points are added/removed
  - Affects intentional point cuts

Evolution of the aspects can affect the base code



- Impact of the aspects on the core code
  - Point cuts can be generalized to make them less brittle
  - Unintentional join points being included?
    - "Fragile point cut" problem

Evolution-related challenges in AOP

- Base code / aspects co-evolution
- Fragile point cuts
  - Missed / unintentionally captured join points
- Aspect composition
  - Log + synchronisation
  - Log synchronisation? Synchronise logging?
- Bad aspect smells
  - Refactorings [Monteiro, Fernandes]
  - High-level, hard to automate
  - Automatic support can be provided

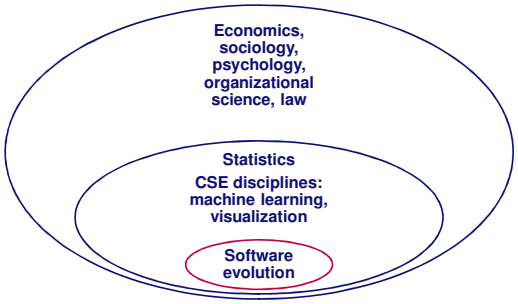
Conclusions: AOP

- AOP: promising technique
  - Specific (co-)evolution challenges
  - Migration
    - Aspect exploration
    - Aspect extraction
  - Automatic support is possible
  - Still manual check is required

Software evolution: conclusions

- Software evolution is inherently complex
  - Humans, software technology, market, non-software technology, multiple stakeholders, feedback loops
- Software evolution is inherently multi-faceted
  - Requirements, architecture, code
  - Code duplication and differencing
  - One version (evolvability) and multiple versions (evolution)
  - What happened vs. How to proceed?

Software evolution and other knowledge areas



Beyond 2IS55

- You: the first generation of the S.E. students
- Follow-up classes:
  - 2IS95: Seminar Software Eng & Technology
  - 2IM91: Master project
  - 2IS99: Capita Selecta Software Eng & Technology
- Going deeper:
  - Model transformation: 2IS15 Generic language techn.
  - Association rule mining: Dr. Toon Calders (DH)
  - Software visualization: Dr. Danny Holten (Vis)

What do you think about

Software Evolution

in general and specifically about

2IS55?