2IS55 Software Evolution

Implementing evolution: Aspect-Oriented Programming
Alexander Serebrenik

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

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

Source
Software Evolution
Ch. 9

Bank Transfer Example

Authentication

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

Problem

One file – multiple concerns
Cash withdraw concern
Cash deposit concern

One concern – multiple files
Transfer concern

Solution: Aspects

Each program element implements one concern only.

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 where the code will be woven

Example: Core application functionality

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

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

```java
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

- 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

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
      - Field found: 3
      - Methods found: 13
      - Code elements found: 11
      - False positives: 2
      - < 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
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

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: (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

Fan-In vs. Identifier analysis: JHotDraw case

<table>
<thead>
<tr>
<th>Concern</th>
<th>Fan-In Analysis</th>
<th>Identifier Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Consumer behaviour</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Contract enforcement</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bring to front: lead to back</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Message handlers</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Move Figures</td>
<td>(discarded)</td>
<td>-</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dup Baker</td>
<td>4</td>
<td>2.8</td>
<td>0</td>
</tr>
<tr>
<td>CCFinder Kamiya</td>
<td>5</td>
<td>3.8</td>
<td>0.8</td>
</tr>
<tr>
<td>CloneDr Baxter</td>
<td>6</td>
<td>4.3</td>
<td>3.8</td>
</tr>
<tr>
<td>cpdetector Koschke</td>
<td>6</td>
<td>3.8</td>
<td>0</td>
</tr>
<tr>
<td>Duplix Krinke Graph</td>
<td>5</td>
<td>4.8</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Null</th>
<th>Range</th>
<th>Exception</th>
<th>Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>.65</td>
<td>.99</td>
<td>.71</td>
<td>.38</td>
<td>.62</td>
</tr>
<tr>
<td>Recall</td>
<td>.63</td>
<td>.97</td>
<td>.59</td>
<td>.36</td>
<td>.57</td>
</tr>
<tr>
<td>Recall</td>
<td>.81</td>
<td>.80</td>
<td>.42</td>
<td>.35</td>
<td>.68</td>
</tr>
<tr>
<td>Recall</td>
<td>.83</td>
<td>.99</td>
<td>.72</td>
<td>.53</td>
<td>.78</td>
</tr>
<tr>
<td>Recall</td>
<td>.96</td>
<td>1.0</td>
<td>.89</td>
<td>.79</td>
<td>.76</td>
</tr>
<tr>
<td>Recall</td>
<td>.95</td>
<td>1.0</td>
<td>.96</td>
<td>.97</td>
<td>.85</td>
</tr>
<tr>
<td>Recall</td>
<td>.98</td>
<td>1.0</td>
<td>.92</td>
<td>.95</td>
<td>.90</td>
</tr>
</tbody>
</table>

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)
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

Economics, sociology, psychology, organizational science, law

Statistics
CSE disciplines: machine learning, visualization
Software evolution
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?