

Mining Control Flow Graphs for Crosscutting Concerns

Jens Krinke
FernUniversität in Hagen



October 24, 2006

- ... **cross-cutting concerns**, or **crosscutting concerns**, are *aspects* of a program which affect (crosscut) other concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and result in either scattering or tangling of the program, or both.
- ... an **aspect** is a part of a program that cross-cuts its core concerns, therefore violating its separation of concerns.

Aspect Mining – Search for Crosscutting Concerns

Crosscutting concerns in *traditional* systems are *tangled* and *scattered*.

When crosscutting concerns are identified,

- traditional systems are more comprehensible
- refactoring will be possible
- deviations can be symptoms for faults

→ *Can we identify crosscutting concerns automatically?*

→ *Can we identify typical previously unknown aspects?*

Aspect Mining – Search for Crosscutting Concerns

Crosscutting concerns in *traditional* systems are *tangled* and *scattered*.

When crosscutting concerns are identified,

- traditional systems are more comprehensible
 - refactoring will be possible
 - deviations can be symptoms for faults
- *Can we identify crosscutting concerns automatically?*
- *Can we identify typical previously unknown aspects?*

Mining Execution Relations

Fan-In Analysis [Marin et al.]

Method x is called from
(many) different methods a, b, c, \dots

Analysis of Traces or Control Flow Graphs [Breu, Krinke]

Method x is always called

- 1 *before* (many) different methods a, b, c, \dots
- 2 *after* (many) different methods a, b, c, \dots
- 3 *first* in (many) different methods a, b, c, \dots
- 4 *last* in (many) different methods a, b, c, \dots

→ “many” indicates crosscutting

→ “always” enables join points.

Mining Execution Relations

Fan-In Analysis [Marin et al.]

Method x is called from
(many) different methods a, b, c, \dots

Analysis of Traces or Control Flow Graphs [Breu, Krinke]

Method x is always called

- 1 *before* (many) different methods a, b, c, \dots
- 2 *after* (many) different methods a, b, c, \dots
- 3 *first* in (many) different methods a, b, c, \dots
- 4 *last* in (many) different methods a, b, c, \dots

→ “many” indicates crosscutting

→ “always” enables join points.

uniform

Mining Execution Relations

Fan-In Analysis [Marin et al.]

Method x is called from
(many) different methods a, b, c, \dots

Analysis of Traces or Control Flow Graphs [Breu, Krinke]

Method x is always called

- 1 *before* (many) different methods a, b, c, \dots
- 2 *after* (many) different methods a, b, c, \dots
- 3 *first* in (many) different methods a, b, c, \dots
- 4 *last* in (many) different methods a, b, c, \dots

→ “many” indicates crosscutting
“always” enables join points.

crosscutting

Mining Execution Relations

Fan-In Analysis [Marin et al.]

Method x is called from
(many) different methods a, b, c, \dots

Analysis of Traces or Control Flow Graphs [Breu, Krinke]

Method x is always called

- 1 *before* (many) different methods a, b, c, \dots
- 2 *after* (many) different methods a, b, c, \dots
- 3 *first* in (many) different methods a, b, c, \dots
- 4 *last* in (many) different methods a, b, c, \dots

→ “many” indicates crosscutting

→ “always” enables join points.

Static Aspect Mining

Idea

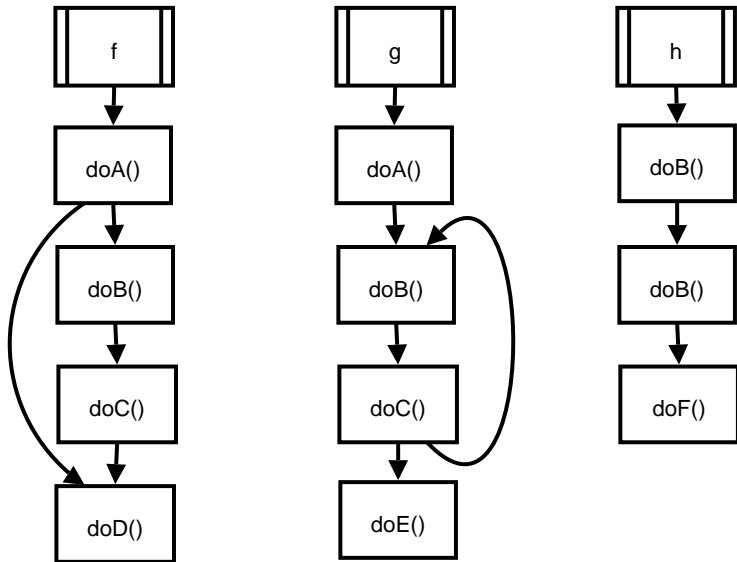
Search for patterns of recurring execution relations

Procedure

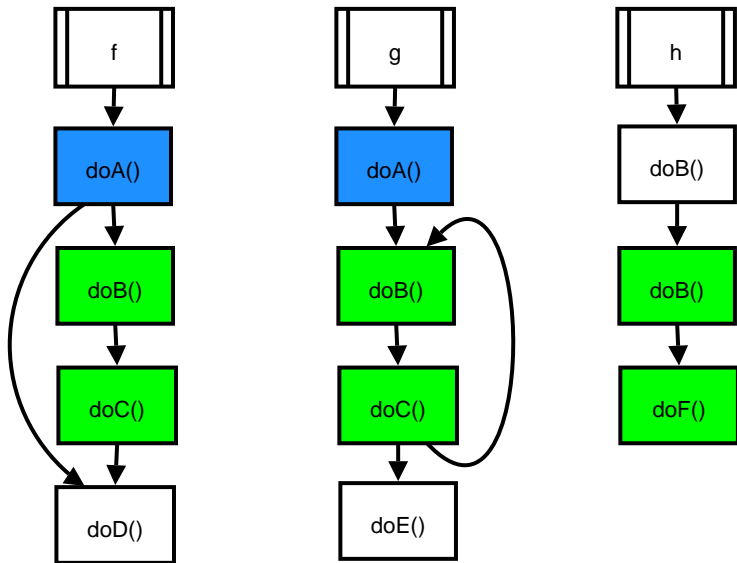
Analysis of control flow graphs for recurring patterns.

- Call to X is always followed by a call to Y .
- Inside X another method Y is always called first/last.

Example



Example



Example

- 1 Extract all execution relations from the control flow graph
- 2 Remove non uniform relations
- 3 Remove non crosscutting relations

<i>doA first in f</i>	<i>doD last in f</i>	<i>doA before doB</i>	<i>doB after doA</i>
<i>doA first in g</i>	<i>doE last in g</i>	<i>doA before doD</i>	<i>doB after doB</i>
<i>doB first in h</i>	<i>doF last in h</i>	<i>doB before doB</i>	<i>doB after doC</i>
		<i>doB before doC</i>	<i>doC after doB</i>
		<i>doC before doB</i>	<i>doD after doA</i>
		<i>doC before doD</i>	<i>doD after doC</i>
		<i>doC before doE</i>	<i>doF after doB</i>
		<i>doB before doF</i>	

Example

- 1 Extract all execution relations from the control flow graph
- 2 Remove non uniform relations
- 3 Remove non crosscutting relations

<i>doA first in f</i>	<i>doD last in f</i>	<i>doA before doB</i>	<i>doB after doA</i>
<i>doA first in g</i>	<i>doE last in g</i>	<i>doA before doD</i>	<i>doB after doB</i>
<i>doB first in h</i>	<i>doF last in h</i>	<i>doB before doB</i>	<i>doB after doC</i>
		<i>doB before doC</i>	<i>doC after doB</i>
		<i>doC before doB</i>	<i>doD after doA</i>
		<i>doC before doD</i>	<i>doD after doC</i>
		<i>doC before doE</i>	<i>doF after doB</i>
		<i>doB before doF</i>	

Example

- 1 Extract all execution relations from the control flow graph
- 2 Remove non uniform relations
- 3 Remove non crosscutting relations

<i>doA first in f</i>	<i>doD last in f</i>	<i>doA before doB</i>	<i>doB after doA</i>
<i>doA first in g</i>	<i>doE last in g</i>	<i>doA before doD</i>	<i>doB after doB</i>
<i>doB first in h</i>	<i>doF last in h</i>	<i>doB before doB</i>	<i>doB after doC</i>
		<i>doB before doC</i>	<i>doC after doB</i>
		<i>doC before doB</i>	<i>doD after doA</i>
		<i>doC before doD</i>	<i>doD after doC</i>
		<i>doC before doE</i>	<i>doF after doB</i>
		<i>doB before doF</i>	

Mining JHotDraw (“before”)

JHotDraw 5.4b1: 18 KLOC, 2900 methods, extensively analyzed

size	candidates	size	candidates
2	53	8	1
3	19	9	1
4	4	11	1
5	6	12	1
6	3	13	1
7	2		

m called before n :

294 relations in 92 candidates

Mining JHotDraw (“first in”)

size	candidates	size	candidates
2	127	13	4
3	55	15	2
4	30	16	1
5	12	17	2
6	9	18	1
7	7	19	1
8	7	20	1
9	3	22	1
10	3	24	2
11	3	32	1
12	4	49	1

m called first in n :

1236 relations in 277 candidates

Need for “filters”

...*CollectionsFactory.current* is called first in 49 different methods.

- accesses the current factory method
- ✗ crosscutting, but not not refactorable
- ➔ classified as a *utility* method by Marin et al.

...*DrawingView.view* is called first in 32 different methods.

- returns the currently active view
- ✗ crosscutting, but not refactorable
- ➔ classified as a *utility* method by Marin et al.

...*DecoratorFigure.getDecoratedFigure* and ...*AbstractHandle.owner* are called first in 24 different methods.

- are accessor methods
- classified as part of the observer crosscutting concern method by Marin et al.
- ✗ crosscutting, but not refactorable

Need for “filters”

...*CollectionsFactory.current* is called first in 49 different methods.

- accesses the current factory method
- ✗ crosscutting, but not not refactorable
- ➔ classified as a *utility* method by Marin et al.

...*DrawingView.view* is called first in 32 different methods.

- returns the currently active view
- ✗ crosscutting, but not refactorable
- ➔ classified as a *utility* method by Marin et al.

...*DecoratorFigure.getDecoratedFigure* and ...*AbstractHandle.owner* are called first in 24 different methods.

- are accessor methods
- classified as part of the observer crosscutting concern method by Marin et al.
- ✗ crosscutting, but not refactorable

Need for “filters”

...*CollectionsFactory.current* is called first in 49 different methods.

- accesses the current factory method
- ✗ crosscutting, but not not refactorable
- classified as a *utility* method by Marin et al.

...*DrawingView.view* is called first in 32 different methods.

- returns the currently active view
- ✗ crosscutting, but not refactorable
- classified as a *utility* method by Marin et al.

...*DecoratorFigure.getDecoratedFigure* and ...*AbstractHandle.owner* are called first in 24 different methods.

- are accessor methods
- classified as part of the observer crosscutting concern method by Marin et al.
- ✗ crosscutting, but not refactorable

Need for “filters”

...Iterator.next is always called before 13 different methods

✗ incidental crosscutting

→ classified as a *utility* method by Marin et al.

...AbstractCommand.execute is always called before 12 methods

- 11 different methods with the name “*createUndoActivity*”

- is the 12th method an anomaly? No.

- ✓ crosscutting and refactorable

...DrawingView.drawing, *...List.add*, and *...DrawingView.view* are always called before 11, 9, and 8 resp. methods.

✗ incidental crosscutting

→ can be regarded as utility methods

Need for “filters”

...Iterator.next is always called before 13 different methods

✗ incidental crosscutting

→ classified as a *utility* method by Marin et al.

...AbstractCommand.execute is always called before 12 methods

- 11 different methods with the name “*createUndoActivity*”

- is the 12th method an anomaly? No.

- ✓ crosscutting and refactorable

...DrawingView.drawing, *...List.add*, and *...DrawingView.view* are always called before 11, 9, and 8 resp. methods.

✗ incidental crosscutting

→ can be regarded as utility methods

Need for “filters”

...Iterator.next is always called before 13 different methods

- ✗ incidental crosscutting
- classified as a *utility* method by Marin et al.

...AbstractCommand.execute is always called before 12 methods

- 11 different methods with the name “*createUndoActivity*”
- is the 12th method an anomaly? No.
- ✓ crosscutting and refactorable

...DrawingView.drawing, *...List.add*, and *...DrawingView.view* are always called before 11, 9, and 8 resp. methods.

- ✗ incidental crosscutting
- can be regarded as utility methods

A simple filter

Without filtering, execution relation based approaches result in many false positives.

- Marin et al. filter accessors and manually specified utility methods (and some heuristics).
- A simple filter discards all non-void methods.

Assumption any method that returns a value is required and has been delegated a task to perform.

→ Delegation is a trivial form of crosscutting.

Results after filtering

size	candidates	size	candidates
2	30	8	2
3	15	10	1
4	11	11	1
5	1	13	1
6	1	17	1
7	2	20	1

m called first in *n*:
261 relations in 67 candidates

size	candidates
2	11
3	3
4	2
5	1
7	1
12	1

m called before *n*:
62 relations
in 19 candidates

Results after filtering

Manual inspection of all candidates that are always called first in at least seven different methods revealed:

- 9 candidates
 - 8 crosscutting concerns
 - 1 classified as delegation (contrasts to Ceccato et al.)
 - 3 have not been discussed before
- Classification of library methods as utility methods will ignore potential crosscutting concerns.

...*AbstractFigure.willChange* is called first in 20 different methods

- ✓ crosscutting and refactorable (observer)

...*AbstractCommand.execute* is called first in 17 different methods

- ✓ crosscutting and refactorable (command, contract enforcement)

...*UndoableAdapter.setUndoable* called first in 13 different methods

- is a setter method
- ✓ crosscutting and refactorable (undo)
- ✗ classified as utility method before

...*AbstractTool.mouseDown* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)

...*Rectangle.add* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)
- ✗ classified as utility method before

...*AbstractFigure.willChange* is called first in 20 different methods

- ✓ crosscutting and refactorable (observer)

...*AbstractCommand.execute* is called first in 17 different methods

- ✓ crosscutting and refactorable (command, contract enforcement)

...*UndoableAdapter.setUndoable* called first in 13 different methods

- is a setter method
- ✓ crosscutting and refactorable (undo)
- ✗ classified as utility method before

...*AbstractTool.mouseDown* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)

...*Rectangle.add* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)
- ✗ classified as utility method before

...*AbstractFigure.willChange* is called first in 20 different methods

- ✓ crosscutting and refactorable (observer)

...*AbstractCommand.execute* is called first in 17 different methods

- ✓ crosscutting and refactorable (command, contract enforcement)

...*UndoableAdapter.setUndoable* called first in 13 different methods

- is a setter method
- ✓ crosscutting and refactorable (undo)
- ✗ classified as utility method before

...*AbstractTool.mouseDown* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)

...*Rectangle.add* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)
- ✗ classified as utility method before

...*AbstractFigure.willChange* is called first in 20 different methods

- ✓ crosscutting and refactorable (observer)

...*AbstractCommand.execute* is called first in 17 different methods

- ✓ crosscutting and refactorable (command, contract enforcement)

...*UndoableAdapter.setUndoable* called first in 13 different methods

- is a setter method
- ✓ crosscutting and refactorable (undo)
- ✗ classified as utility method before

...*AbstractTool.mouseDown* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)

...*Rectangle.add* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)
- ✗ classified as utility method before

...*AbstractFigure.willChange* is called first in 20 different methods

- ✓ crosscutting and refactorable (observer)

...*AbstractCommand.execute* is called first in 17 different methods

- ✓ crosscutting and refactorable (command, contract enforcement)

...*UndoableAdapter.setUndoable* called first in 13 different methods

- is a setter method
- ✓ crosscutting and refactorable (undo)
- ✗ classified as utility method before

...*AbstractTool.mouseDown* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)

...*Rectangle.add* is called first in 11 different methods

- ✓ crosscutting and refactorable (consistent behavior)
- ✗ classified as utility method before

...*ObjectInputStream.defaultReadObject* is called first in 8 methods

- ✓ crosscutting and refactorable (consistent behavior)
- ✗ classified as utility method before

...*Rectangle.translate* is called first in 8 methods

- instances of *basicMoveBy* methods
- crosscutting as consistent behavior?
- No!
 - *translate* is used in 20 of >30 *basicMoveBy* methods
 - heterogeneous behavior
- ✗ neither crosscutting, nor refactorable

...*AttributeFigure.write* and ...*AttributeFigure.read*

- ✓ crosscutting (and refactorable)
- part of the composite design pattern
- ➔ not to be refactored

...*ObjectInputStream.defaultReadObject* is called first in 8 methods

- ✓ crosscutting and refactorable (consistent behavior)
- ✗ classified as utility method before

...*Rectangle.translate* is called first in 8 methods

- instances of *basicMoveBy* methods
- crosscutting as consistent behavior?
- No!
 - *translate* is used in 20 of >30 *basicMoveBy* methods
 - heterogeneous behavior
- ✗ neither crosscutting, nor refactorable

...*AttributeFigure.write* and ...*AttributeFigure.read*

- ✓ crosscutting (and refactorable)
 - part of the composite design pattern
- not to be refactored

...*ObjectInputStream.defaultReadObject* is called first in 8 methods

- ✓ crosscutting and refactorable (consistent behavior)
- ✗ classified as utility method before

...*Rectangle.translate* is called first in 8 methods

- instances of *basicMoveBy* methods
- crosscutting as consistent behavior?
- No!
 - *translate* is used in 20 of >30 *basicMoveBy* methods
 - heterogeneous behavior
- ✗ neither crosscutting, nor refactorable

...*AttributeFigure.write* and ...*AttributeFigure.read*

- ✓ crosscutting (and refactorable)
- part of the composite design pattern
- ➔ not to be refactored

Delegation

- **Delegation** is the handing of a task over to another person, usually a subordinate.
- In object-oriented programming there are two related notions of delegation:
 - ① In its original usage, delegation refers to a design pattern where one object relies upon another to provide a specified set of functionality.
 - ② Alternately, it can refer to a programming language feature making use of the method lookup rules for dispatching so-called self-calls.
- Delegation has also been suggested for advice resolution.
- In 1989, the “war” between delegation and inheritance ended (The Treaty of Orlando).

Execution Relation based Aspect Mining approaches

- try to identify crosscutting concerns
- mine for join points
- often find delegations

How can crosscutting concerns
be distinguished from delegation?