Resolving Feature Convolution in Middleware Systems

Talk about the paper from Zhang and Jacobsen (OOPSLA, 2004)

presented by Manuel Krucker
content

- Intro
- Re-factoring based implementation of Horizontal Decomposition (HD)
- Implementation evaluation
- Conclusions / Related Works
- Appendix
overview

- Intro
  - what’s mw ?
  - what’s the challenge ?
  - HD as a solution
- Re-factoring based implementation of Horizontal Decomposition (HD)
- Implementation evaluation
- Conclusions / Related Works
- Appendix
what’s middleware?

- what: platform facilitates development of distr. Applications
- why: - shorter development cycle
  - much smaller coding effort
- challenge: - no assumption about application domain
  - fast evolution of MW has created many problems
how is this solved in praxis?

There is a proliferation of specifications.
E.g: in CORBA:
- CORBA
- Realtime CORBA
- Minimum CORBA
- Data-parallel CORBA
- Fault-tolerant CORBA
- specific CORBA individually
But, real goals are …

- high degree of:
  - Configurability
  - Adaptability
  - Customizability

- Ultimate goal:
  - specific user need!

- Possible to reach:
  - often unattainable because of crosscutting concerns
define: “CROSSCUT”

Is any part of the design of an OOP application that occurs simultaneously in several different parts of the OOP application that are not otherwise related.
(also “crosscutting concern”)

Due to bad design?
No, due to limitations of conventional decomposition methods.
any solution?

“.. We introduce the principles of horizontal decomposition (HD) which addresses this problem with a mixed-paradigm MW architecture. HD provides guidance for the use of conventional decomposition methods to implement the core functionalities of middleware and the use of aspect orientation to address its orthogonal properties.”
Any solution? : Horizontal decomposition (HD)

Hierarchical decomposed architecture for a minimal, specialized, and commonly shared core.

**Vertical Decomposition (VD)**
- levels of abstraction
- stepwise refinement
- single indep. function performing single logical task

**Core Funct. ➔ GENERALITY**

Referring to domain-specific properties, which can be composed as aspects.

**Orth. properties ➔ SPECIALITY**

**Aspect Oriented Programming (AOP)**
- higher degree of separation
- handle crosscutting concerns
- new languages constructs needed

a mixed-paradigm architecture
overview

- Intro
- Re-factoring based implementation of Horizontal Decomposition (HD)
- Implementation evaluation
- Conclusions / Related Works
- Appendix
Distributed object system Architecture

ORB’s task:
- interface definition
- location and poss. activation of remote objects
- communication between clients and objects
ORBacus remote invocation

Abstraction:

- IDL-Layer
- Messaging-Layer
- Transport and Protocol-Layer
Remote and Local Invocation simultaneously in ORBacus
Convolution Code

deal with Interceptors

deal with one-way

deal with both
Implementation convolution means …

- loss of modularity and configurability
- it also increases runtime overhead

(Example: asynchronous one-way)

Fundamental question:

-> How do we untangle convolution features?

We need patterns/principles
5 principles on HD

1. Recognize the relativity of aspects.
2. Establish the coherent core decomposition.
3. Define the semantics of an aspect according to the core decomposition.
4. Maintain a class-directional architecture.
5. Apply a incremental refactoring.
Recognize the relativity of aspects (principle 1)

The semantics of an aspect is determined by the primary functionality of the application.

in a mw architecture: 
MW aspects are relative to the primary functionality of MW.

- define primary functionalities
- Mw aspects are mw features that crosscuts the implementation of its core functionality
Defining mw core

Three layer where each has **ONE** task:

- IDL layer (I-L): Stub and Skeleton
  Task: support remote statically typed, synchronous invocation

- Messaging layer (M-L): client-side and server-side
  Task: un-/marshalling

- Transport and Protocol layer (T-L)
  Task: Communication with peer ORB’s
Establish the coherent core decomposition (principle 2)

The basis of aspect oriented decomposition is the establishment of a functionally coherent and vertically decomposed core.

in a mw architecture:
The MW core consists of a set of components that support transparent remote invocations.

- MW core:
  Composing, transporting, and dispatching invocation requests (only synchronous com., static typed req.)
- Aspects:
  Asynchronous com., dynamically typed req.
Simple vs. complex

- **Functional cohesion**: one function within a module should perform one single task

- MW architecture should be functional coherent
- therefore: system contains no convoluted features.
- Adv: - easier to obtain VD with a small core
  - Good basis for further customization
Defining CORBA aspects

- One-way invocation semantic
- Dynamic typing
- Local invocation support
- ...

...
Define the semantics of an aspect according to the core decomposition (principle 3).

Using the core as a reference, a functionality is considered Orthogonal if both its semantics and its implementations are not local to a single component of the core. Only the orthogonal functionality is treated in the aspect oriented way.

**in a mw architecture:**
MW aspects are feature that cannot be encapsulated within an individual component of the core decomposition. (E.g.: customization of com. protocols / async. Invocation)
Oneway invocation semantic

- **Semantic:**
  
  best-effort and asynchronous delivery of client requests
  
  No response is expected.

- **Orthogonality:**
  
  core support synchronous invocation semantics

- **Crosscutting:**
  
  I-L: support of keyword “oneway”
  
  M-L: add. Logic in the downcall process for not expecting a response.
  
  P-L: setting of a time-out value
Dynamic typing

- **Semantic:**
  supports reflective composition of remote invocation.

- **Orthogonality:**
  core supports statically typed invocation requests.

- **Crosscutting:**
  I-L: support of dynamic IDL data types (Any, Dynamic Any) and associated stub/skeleton operations.
  M-L: un-/marshalling of these data types
  P-L: None
Local invocation support

Semantic:
supports different forwarding of request.

Orthogonality:
local invocation is orthogonal to remote invocation

Crosscutting:  
(Cf. Picture)
Marshalling
Protocol
Dispatch
Maintain a class-directional architecture (principle 4).

Crosscutting concerns should be implemented class – directional towards the core.

What means class-directional?

**in a mw architecture:**
The implementation of MW aspects are class-directional and super-impositional.

three different types of components:
- Multiple sets of VD (core classes)
- Exclusive application of AOP (aspect classes)
- Flexible combination of architectures (SI). Interaction between of an aspect, the core, other aspects individually (SI classes)
Untangling convoluted aspects

**Convolution Matrix**

<table>
<thead>
<tr>
<th></th>
<th>LI</th>
<th>Conv</th>
<th>Dyn</th>
<th>DPI</th>
<th>PI</th>
<th>OW</th>
<th>Wchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv.</td>
<td>x</td>
<td>x</td>
<td>N/A</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Dyn</td>
<td>x</td>
<td></td>
<td>N/A</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>PI</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>OW</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Wchar</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Conv:** Conversion  
**Dyn:** Dynamic Typing  
**OW:** oneway  
**CO:** collocated server  
**PI:** Portable Interceptor  
**Wchar:** wide character/string  
**LI:** Local invocation  
**DPI:** Dynamic Programming Interface
advantages

- Original design choice is fully respected and preserved. No shifting of programming paradigms.
- Crosscutting logic is isolated and therefore conveniently be analyzed.
- Separation of aspect functionality and its crosscutting logic is explicit and can be completely decoupled.
Apply incremental refactoring (principle 5)

Decomposition in the aspect dimension is assisted by incremental refactoring.

In a mw architecture:
MW aspects are implemented incrementally.

Refinements of the definition for the core decomposition give rise to the identification of new aspects.

• First level factoring: factoring the VD
• Second level factoring: to separate new aspects from both the MW core and previously identified aspects.
## History table

<table>
<thead>
<tr>
<th>Stage</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PI, CO</td>
<td>Dyn, Wchar, Conv, OW</td>
<td>DPI, PI CO</td>
</tr>
<tr>
<td>2</td>
<td>OW</td>
<td>Dyn, Wchar, Conv</td>
<td>DPI, PI CO</td>
</tr>
<tr>
<td>3</td>
<td>Dyn</td>
<td>Conv, Wchar</td>
<td>CO, DPI, PI</td>
</tr>
<tr>
<td>4</td>
<td>Wch</td>
<td>Conv</td>
<td>Dyn, PI</td>
</tr>
<tr>
<td>5</td>
<td>Conv</td>
<td></td>
<td>CO, OW, PI, Dyn, Wchar</td>
</tr>
</tbody>
</table>

**Table 3:** Incremental Decomposition of Aspects (A: Aspects being refactored. B: Aspects contained in core. C: Aspects being refactored in 2nd phase. Abbreviations are the same as in Table 2.)
comments

the advantages:

- 40% reduction of code size
- Several combinations can be build for different users
- application is simple
- core stays stable also when we evolve it further
- Much more efficient because not used option can be omitted
overview

- Introduction
- Re-factoring based implementation of horizontal decomposition
- **Implementation evaluation**
  - Structural comparison
  - Performance evaluation
- Conclusions / Related Work
- Appendix
Structural comparison

Reduction of the overall structure.

CC: cyclomatic complexity (number of methods), WC: weight of class (simplification in control flow), EC: efferent coupling

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Size</th>
<th>CC</th>
<th>WC</th>
<th>EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>23277</td>
<td>3.69</td>
<td>2404</td>
<td>2423</td>
</tr>
<tr>
<td>Re-factored</td>
<td>13524</td>
<td>3.05</td>
<td>1549</td>
<td>1899</td>
</tr>
<tr>
<td>Reduction</td>
<td>9753</td>
<td>0.64</td>
<td>855</td>
<td>525</td>
</tr>
</tbody>
</table>

Reduction of Code Elements caused by crosscutting.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Arguments</th>
<th>Conditional</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any &amp; TypeCode</td>
<td>0</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Encoding Conversion</td>
<td>6</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Oneway Call</td>
<td>8</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Wchar &amp; Wstring</td>
<td>4</td>
<td>44</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>18</td>
<td>68</td>
<td>20</td>
</tr>
</tbody>
</table>
performance evaluation

- **Invocation roundtrip:**
  8% performance gain on average
- **Data transport:**
  Improvements because of (un-)marshalling
- **Request dispatch:**
  Improvements because of no dispatching decision
- **Parallel execution:**
  Simpler and lighter mechanism of downcall
- **Combined execution:**
  See Appendix
- **Cache performance:**
  Consistent with our earlier benchmarking results

(See the appendix for detailed analysis of the evaluation.)
Concluding remarks

- Improvements not so dramatic as anticipated at the begin
- Validates that HD methods are effective in separating convoluted features from the MW core without compromising its functionality
- More improvements with further refactoring efforts
- HD applicable to any MW system or more generally any software system
- Virtual machine decreases speed-up
- Past has shown, features supported through aspects do not experience significant runtime overhead using AspectJ.
overview

- Introduction
- Re-factorizing based implementation of horizontal decomposition
- Implementation evaluation
- Conclusions / Related Work
- Appendix
conclusions

- The basis of AOP is defining the core decomposition
- Leads to reduce the core (here 40%)
- Leads to increase configurability and adaptability
- Leads to make the architecture super-impositional (here 60 possible combinations)
- No issue about ensure consistency across aspects
- Long term objective: fully aspect orientated mw platform
my conclusions

- The reader thinks, it must be very easy to maintain functional cohesion. But this is in practice not the case. What means one single task?
- I was surprised about the much better performance. I didn’t expect that.
- The “consistency-question” could be the biggest problem. And if this is not solved, then the hole theory of HD is useless.
Related work

- Existing mw architectures of AOP
- Adaptive MW (various forms of aspect definitions and interpretations, reflection, …)
- Feature Orientated Programming (FOP) (base obj., features that crosscut base obj. are exit in separate modules)

There are many reference papers. They are listed on the paper:
http://www.eecg.toronto.edu/~jacobsen/papers/oopsla04.pdf
questions?

I hope this is not necessary after my presentation ...

… and thank you to listen me.
overview

- Introduction
- Re-factoring based implementation of horizontal decomposition
- Implementation evaluation
- Conclusions / Related Work
- Appendix
  - performance evaluation
Performance evaluation

- Use open benchmarking suite OCBS
- Measures: invocation, marshalling, dispatching, parallelism, as well as combinations of these areas
- Compare:
  - None: refactored ORBacus core with aspects taken out
  - Original: original implementation
  - All: all aspects “woven” back via AspectJ
- Environment: Pentium 4 2GHZ, Redhat 8.0, 1 G of memory
1. Invocation roundtrip

Invocation cost in microseconds

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Median</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>157</td>
<td>203</td>
</tr>
<tr>
<td>Original</td>
<td>167</td>
<td>221</td>
</tr>
<tr>
<td>All</td>
<td>180</td>
<td>238</td>
</tr>
</tbody>
</table>

- client invocation “do nothing”
- average of 4500 sample invocations
- 8% performance gain on average
Data transport
Data transport (2)

Client-Server Data Send Average

Invocation Time (usec)

Payload Size (octet)

- None
- Origin
- All
data transport (3)

Client-Server Data Send Maximum

![Graph showing Client-Server Data Send Maximum with Payload Size (octet) on the x-axis and Invocation Time (usec) on the y-axis. The graph compares different scenarios: None, Origin, and All.]
data transport (4)
data transport (5)
data transport (6)
data transport (7)

- Weighting core functionality in this test is the efficiency of marshalling and unmarshalling
- X-axis: number of octets
- Y-axis: average invocation time
- 10(3): improvement because:
  - lighter-weight marshalling/unmarshalling layer supporting fewer number of CORBA data types
  - marshalling/unmarshalling with no need on character endcoding
  - weave back does not much increase overhead
request dispatch

Multi-server Dispatching Minimum

Invocation Time (usec)

No. of Server Objects

- None
- Origin
- All
request dispatch (2)

Multi-server Dispatching Average

Invocation Time (usec)

No. of Server Objects

Legend:
- None
- Origin
- All
request dispatch (3)

Multi-server Dispatching Maximum

Invocation Time (usec)

- None
- Origin
- All

No. of Server Objects
request dispatch (4)

- X-axis: number of instantiated objects
- Y-axis: invocation time in microseconds
- More objects $\Rightarrow$ average times increases
- refactored best is obvious, there is no dispatching decision (dynamic/local invocations)
- Min/avg case: all 5 % - 13 % penalty
- Worst case: all three versions are equivalent
parallel execution

Multi-threaded Invocation

Invocation Time (usec)

No. of Threads

0 50 100 150 200 250

1 2 3 4 5 6 7 8 9 10 11

None
Origin
All

remote
parallel execution (2)

- X-axis: number of threads created (client)
- Y-axis: invocation time in microseconds
- Overall mechanism of downcall much more simpler and lighter than the original.
- Refactored reduces shared data among threads, therefore it exists in process-wide singletons and includes codeset factories, conversation utilities, default dynamic servers, …
combined execution

multiple threads and exchanging messages of size 50 KB octets between client and server:

- A: message sending using 100 C threads
- B: message receiving using 100 C threads
- C: Remote invocations (Ping) by 100 C threads to 50’000 S
- D: message sending to 50’000 S
- E: message receiving from 50’000 S
- F: message sending by 100 C threads to 50’000 S
- G: message receiving of 100 C threads from 50’000 S

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-factored</td>
<td>7036</td>
<td>2161</td>
<td>193</td>
<td>954</td>
<td>1061</td>
<td>6020</td>
<td>2689</td>
</tr>
<tr>
<td>Original</td>
<td>2332</td>
<td>2395</td>
<td>199</td>
<td>976</td>
<td>1055</td>
<td>1686</td>
<td>3130</td>
</tr>
<tr>
<td>Combined</td>
<td>6142</td>
<td>2366</td>
<td>238</td>
<td>1003</td>
<td>1103</td>
<td>5396</td>
<td>3111</td>
</tr>
</tbody>
</table>
Combined execution (2)

- A, F anomalies possibly caused by OS scheduler
cache performance

- Use performance counter library to count various microprocessor events
- Decrease of instruction-cache miss is indicator of simpler control flow
- Better L2-cache performance represents better locality and a higher degree of cohesion in the program.
- Consistent with our previously presented benchmarking results

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>Original</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction Misses</td>
<td>368869</td>
<td>380404</td>
<td>404187</td>
</tr>
<tr>
<td>L2 Miss Rate</td>
<td>3.25%</td>
<td>3.83%</td>
<td>4.4%</td>
</tr>
</tbody>
</table>