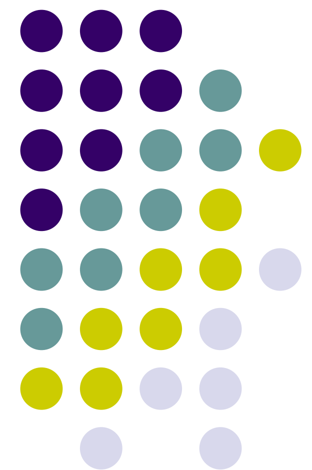
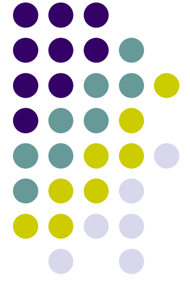
Resolving Feature Convolution in Middleware Systems

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

presented by Manuel Krucker

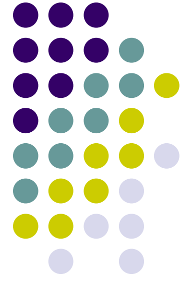


content



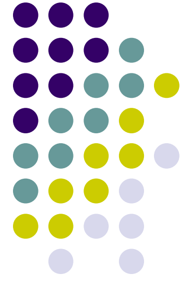
- Intro
- Re-factoring based implementation of Horizontal Decomposition (HD)
- Implementation evaluation
- Conclusions / Related Works
- Appendix

overview



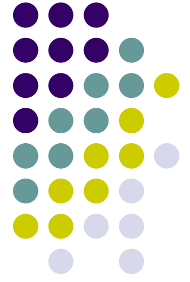
- **Intro**
  - **- what's mw ?**
  - **- what's the challenge ?**
  - **- HD as a solution**
- Re-factoring based implementation of Horizontal Decomposition (HD)
- Implementation evaluation
- Conclusions / Related Works
- Appendix

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?

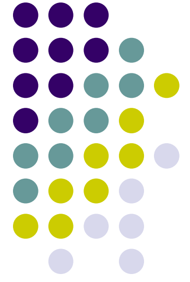


There is a proliferation of specifications .

E.g: in CORBA:

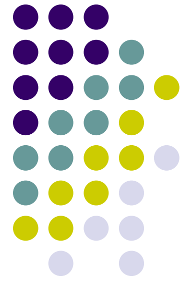
- CORBA
- Realtime CORBA
- Minimum CORBA
- Data-parallel CORBA
- Fault-tolerant CORBA
- specific CORBA induvidually

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”



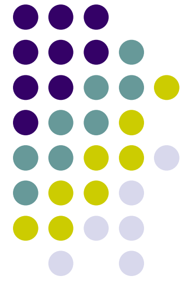
Is any part of the design of an OOP application that occurs simultanously in several different parts of the OOP application that are not otherwise related.

(also “crosscutting concern”)

Due to bad design?

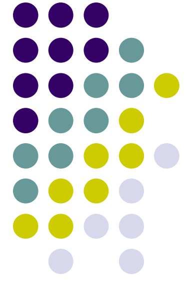
No, due to limitations of conventional decomposition methods.

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)



Core Funct. ➔ GENERALITY

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

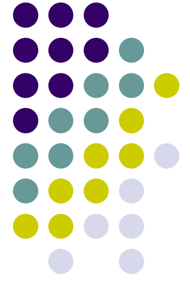
Orth. properties ➔ SPECIALITY

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

Aspect Oriented Programming (AOP)

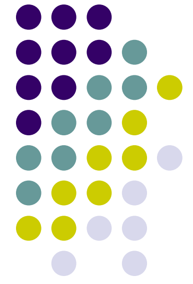
- higher degree of separation
- handle crosscutting concerns
- new languages constructs needed

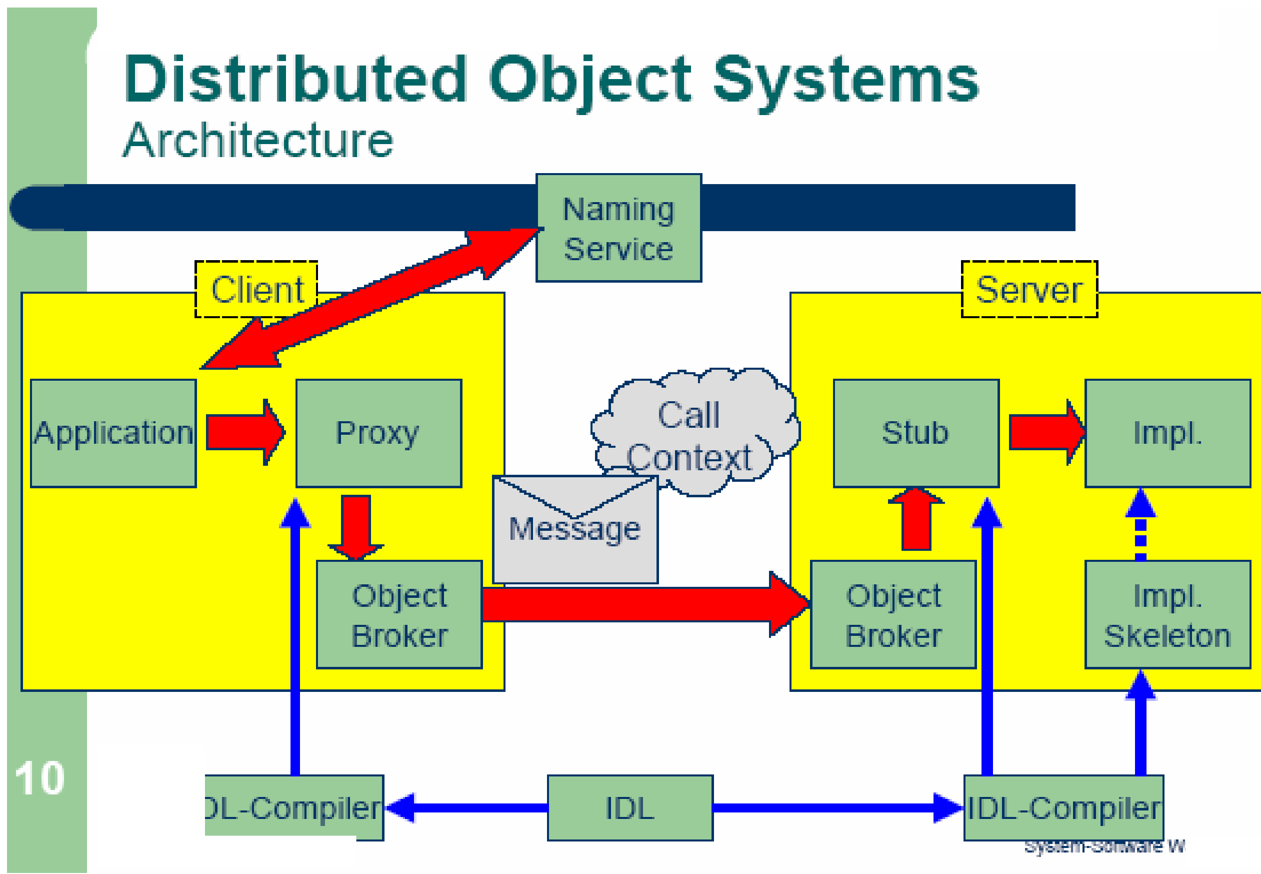
overview



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

Distributed object system Architecture

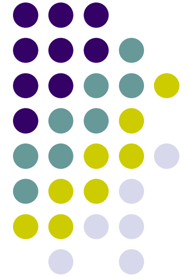


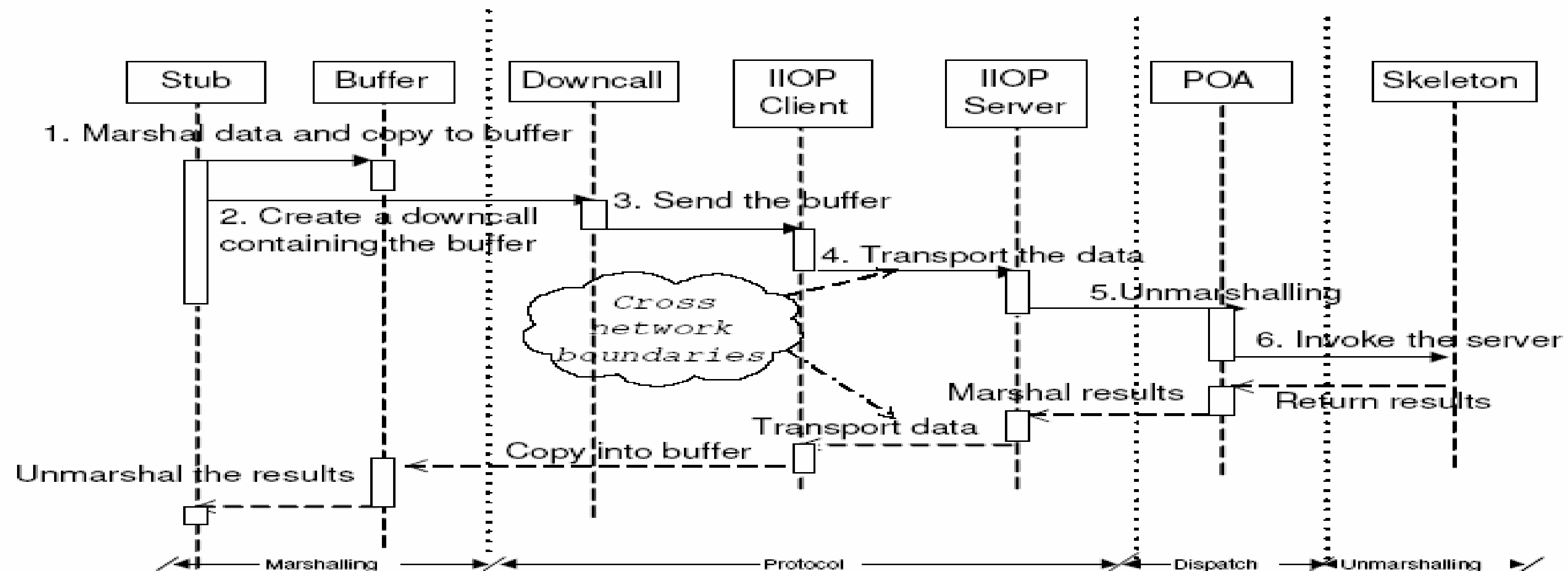


ORB's task:

- interface definition
- location and poss. activation of remote objects
- communication be-tween clients and objects

ORBacus remote invocation

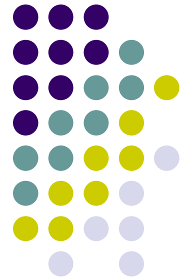




Abstraction:

- **IDL-Layer**
- **Messaging-Layer**
- **Transport and Protocol-Layer**

Remote and Local Invocation simultaneously in ORBacus



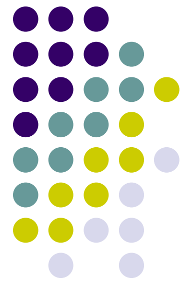
Stub
Buffer
Downcall
POAManager
Collocated Client
IIOP Client
Collocated Server
IIOP Server
POA
Skeleton

1. Marshal data and copy to buffer
2. Create a downcall containing the buffer
3. check if server is local
If server remote
4. send the buffer
4. send the buffer
Cross network boundaries
5. Transport the data
6.Unmarshalling
7. Invoke the server
If server local
Does not cross networks, simply copy data
5. Transport the data
Return results
Transport data
Marshal results
Unmarshal the results
Transport data
Marshal results
Copy into buffer

Marshalling
Protocol
Dispatch
Protocol
Dispatch
Unmar-shalling

Aspect Activitiy
Crosscutting Logic

Convolution Code



deal with both

deal with Interceptors

deal with one-way

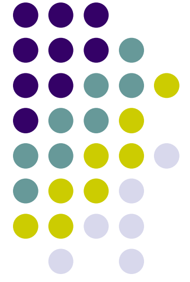
```
public Downcall
    createPIDIIDowncall(String op, boolean resp,
                        org.omg.CORBA.NVList args,
                        org.omg.CORBA.NamedValue result,
                        org.omg.CORBA.ExceptionList exceptions)
        throws FailureException
    {
        com.ooc.OCI.ProfileInfoHolder profile =
            new com.ooc.OCI.ProfileInfoHolder();
        Client client = getClientProfilePair(profile);
        Assert._OB_assert(client != null);

        if(!policies_.interceptor)
            return new Downcall(orbInstance_, client,
                                profile.value, policies_, op, resp);

        PIManager piManager = orbInstance_.getPIManager();
        if(piManager.haveClientInterceptors())
        {
            return new PIDIIDowncall(orbInstance_, client,
                        profile.value, policies_,
                        op, resp, IOR_, origIOR_, piManager,
                        args, result, exceptions);
        }
        else
        {
            return new Downcall(orbInstance_, client,
                                profile.value, policies_, op, resp);
        }
    }
```

(Callout markers 1–8 on the code.)

Implementation convolution means …



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

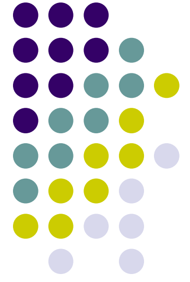
(Example: asynchronous one-way)

Fundamental question:

➔ How do we untangle convolution features?

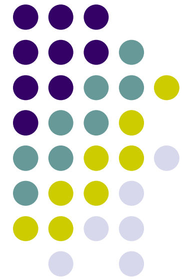
We need patterns/principles

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)

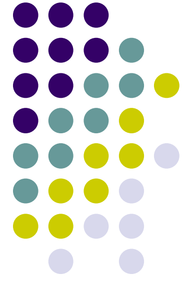


The semantics of an aspect is determined by the primary functionality of the application.

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:

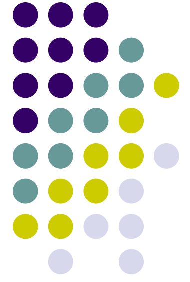
- IDL layer (I-L): Stub and Skeleton

  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)

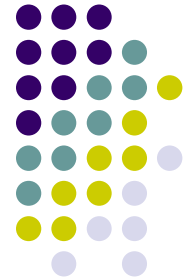


The basis of aspect oriented decomposition is the establishment of a functionally coherent and vertically decomposed core.

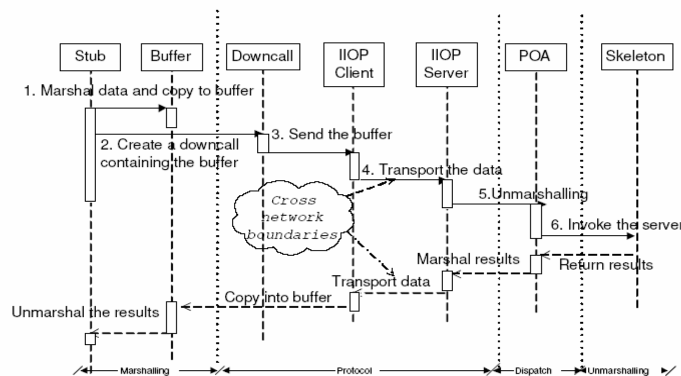
in a mw architecture:

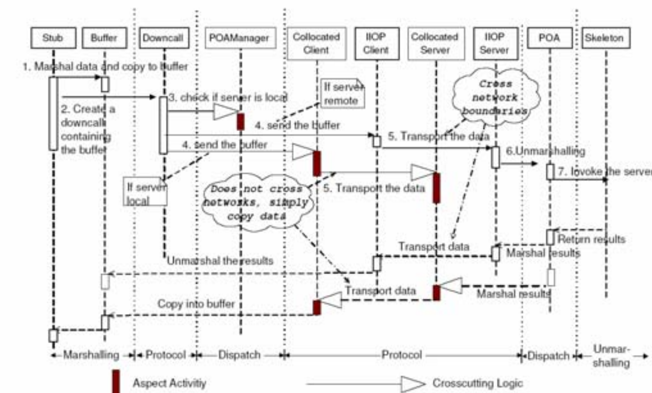
The MW core consists of a set of components that support transparent remote invocations.

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



Simple vs. complex



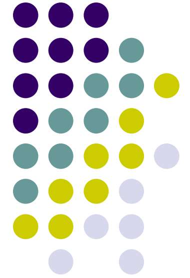


- **Functional cohesion**:

  one function within a module should perform one single task

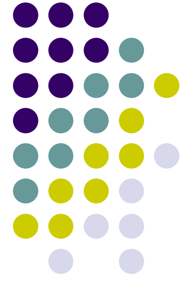
- MW architecture should be functional coherent
- therefore: system contains no convoluted features.
- Adv: - easier to obtain VD with a small core
  - Good basis for further customization

Defining CORBA aspects



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

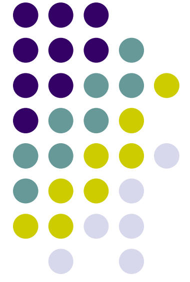
Define the semantics of an aspect according to the core decomposition (principle 3).



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

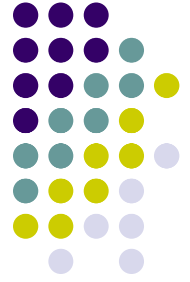
- **Crosscutting:**

  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.

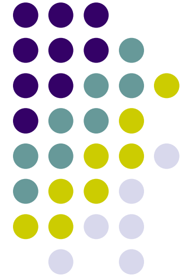
- **Crosscutting:**

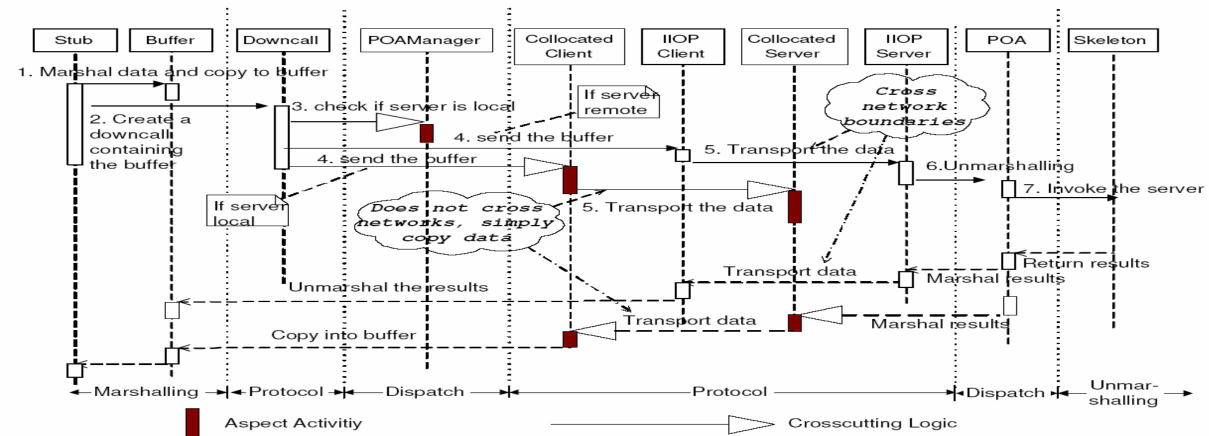
  I-L: support of dynamic IDL data types (Any, Dynamic Any) and sociated stub/skeleton operations.

  M-L: un-/marshalling of these data types

  P-L: None

Local invocation support





Semantic:

supports different forwarding of request.

Orthogonality:

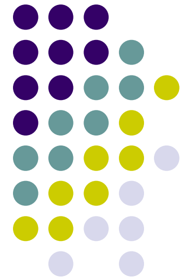
local invocation is orthogonal to remote invocation

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?

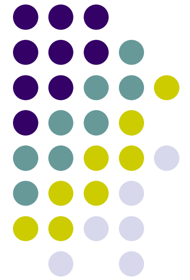
in a mw architecture:

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

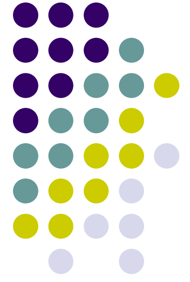


Convolution Matrix

	LI	Conv	Dyn	DPI	PI	OW	Wchar
Conv.	x	N/A	x		x	x	x
Dyn	x		N/A	x	x		x
PI				x	N/A		
OW	x			x		N/A	
Wchar		x	x		x		N/A

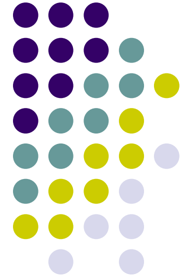
Conv:	**Conversion**	**PI:**	**Portable Interceptor**
Dyn:	**Dynamic Typing**	**Wchar:**	**wide character/string**
OW:	**oneway**	**LI:**	**Local invocation**
CO:	**collocated server**	**DPI:**	**Dynamic Programming Interface**

advantages



- Original design choice is fully respected and preserved. No shifting of programming paradigms.
- Crosscutting logic is isolated and therefore conveniently be analyezed
- 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.

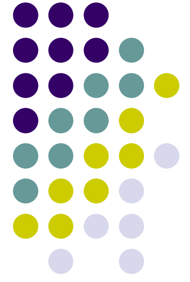
In a mw architecture:

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.

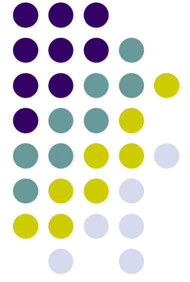
History table



Stage	A	B	C
1	PI, CO DPI	Dyn, Wchar, Conv, OW	
2	OW	Dyn, Wchar, Conv	DPI, PI CO
3	Dyn	Conv,Wchar	CO, DPI, PI
4	Wch	Conv	Dyn, PI
5	Conv		CO, OW, PI, Dyn, Wchar

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.

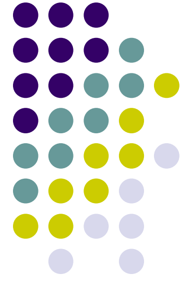
comments



the advantages:

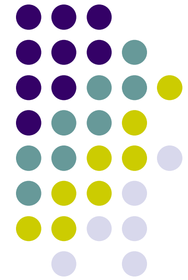
- 40% reduction of code size
- Several combinations can be build for different users
- application is simple
- core stays stable also when we evolve it further
- Much more efficient because not used option can be omitted

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.

	42%	35%	17%	22%
Implementation	*Size*	*CC*	*WC*	*EC*
Original	23277	3.69	2404	2423
Re-factored	13524	3.05	1549	1899
Reduction	9753	0.64	855	525

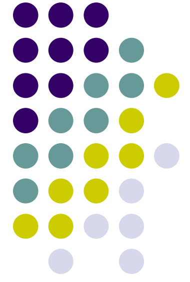
[in KB]

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

Reduction of Code Elements caused by crosscutting.

Aspect	*Arguments*	*Conditional*	*Attributes*
Any & TypeCode	0	8	2
Encoding Conversion	6	9	9
Oneway Call	8	7	1
Wchar & Wstring	4	44	8
Total	18	68	20

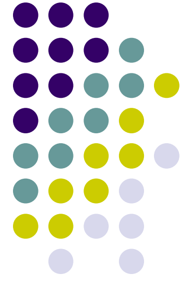
performance evaluation



- **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

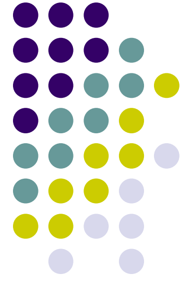
(See the appendix for detailed analysis of the evaluation.)

Concluding remarks



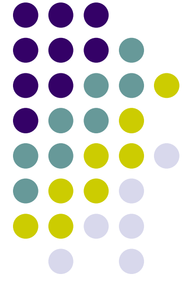
- Improvements not so dramatic as anticipated at the begin
- validates that HD methods are effective in separating convoluted features from the MW core without compromising its functionality
- More improvements with further refactoring efforts
- HD applicable to any MW system or more generally any software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

overview



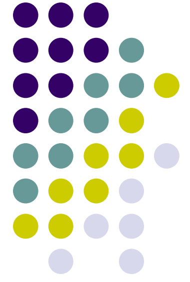
- Introduction
- Re-factoring based implementation of horizontal decomposition
- Implementation evaluation
- **Conclusions / Related Work**
- Appendix

conclusions



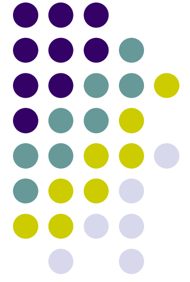
- The basis of AOP is defining the core decomposition
- Leads to reduce the core (here 40%)
- Leads to increase configurability and adaptability
- Leads to make the architecture super-impositional (here 60 possible combinations)
- No issue about ensure consistency across aspects
- Long term objective: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

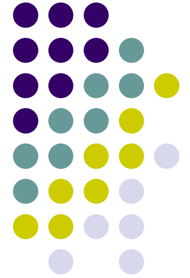
  (base obj., features that crosscut base obj. are exit in separate modules)

There are many reference papers.

They are listed on the paper:

http://www.eecg.toronto.edu/~jacobsen/papers/oopsla04.pdf

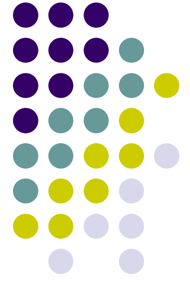
questions?



I hope this is not necessary after my presentation …

… and thank you to listen me.

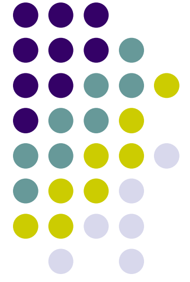
overview



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

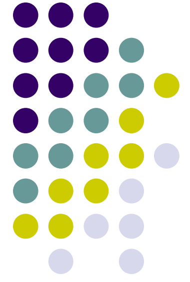
  - performance evaluation

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
- Enviroment: Pentium 4 2GHZ, Redhat 8.0, 1 G of memory

1. Invocation roundtrip

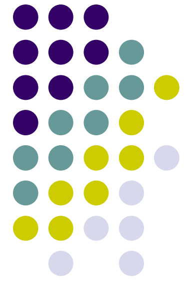


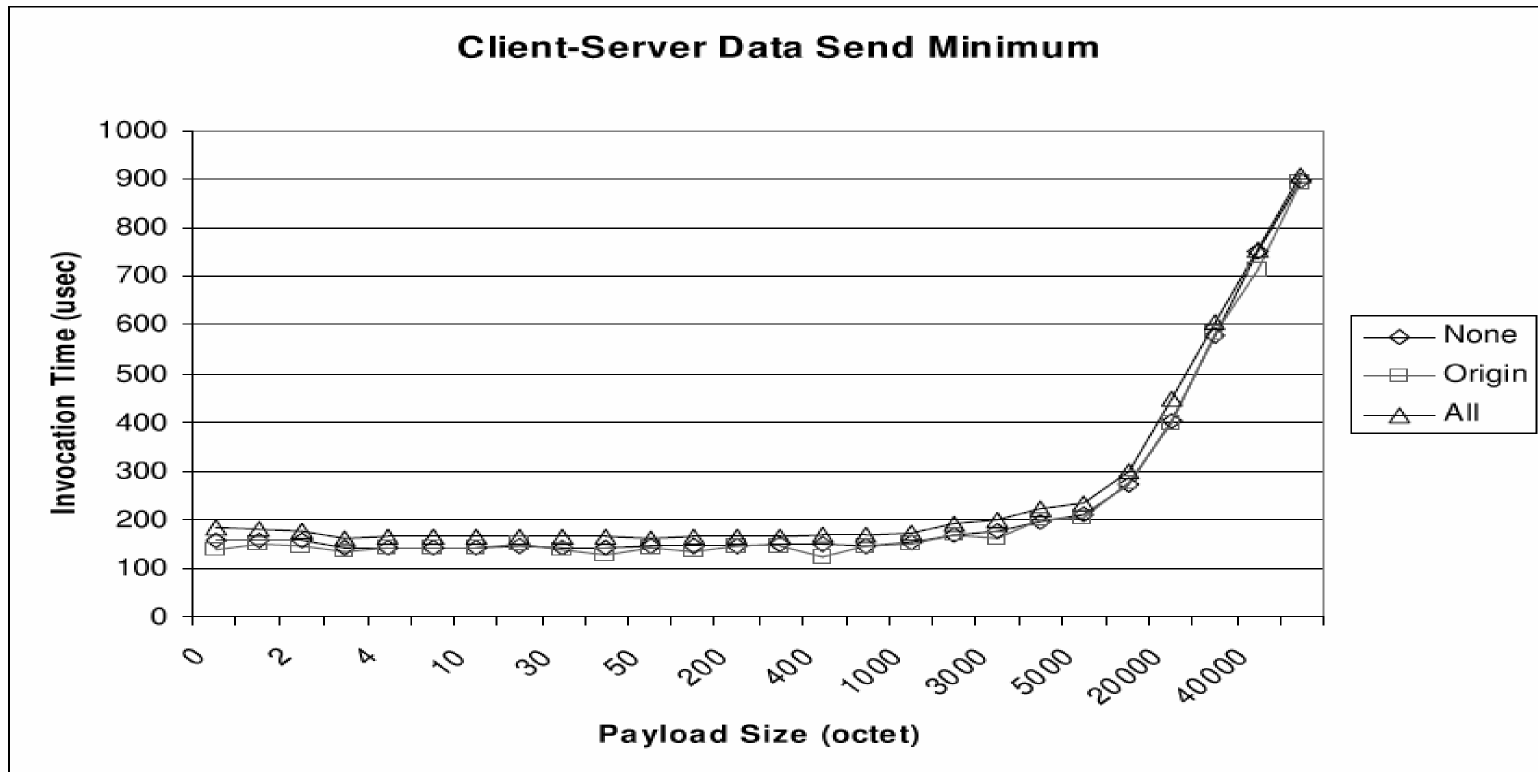
Invocation cost in microseconds

Implementation	*Median*	*Average*
None	157	203
Original	167	221
All	180	238

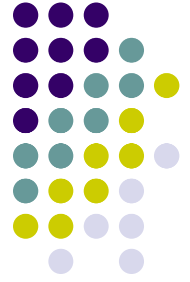
- client invocation “do nothing”
- average of 4500 sample invocations
- 8% performance gain on average

Data transport





Data transport (2)



Client-Server Data Send Average

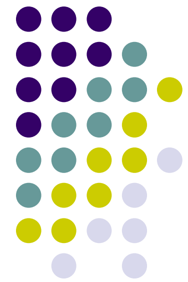
Invocation Time (usec)

1400
1200
1000
800
600
400
200
0

0 2 4 10 30 50 200 400 1000 3000 5000 20000 40000

Payload Size (octet)

None
Origin
All



data transport (3)

Client-Server Data Send Maximum

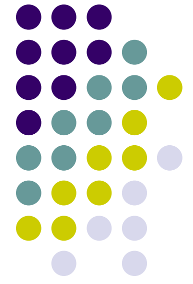
Invocation Time (usec)

2000
1800
1600
1400
1200
1000
800
600
400
200
0

0 2 4 10 30 50 200 400 1000 3000 5000 20000 40000

Payload Size (octet)

None
Origin
All



data transport (4)

Server-Client Data Send Minimum

Invocation Time (usec)

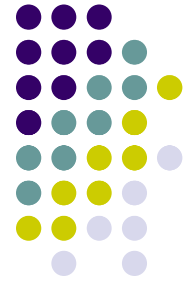
1200
1000
800
600
400
200
0

0 2 4 10 30 50 200 400 1000 3000 5000 20000 40000

Payload Size (octet)

None
Origin
All

data transport (5)



Server-Client Data Send Average

Invocation Time (usec)

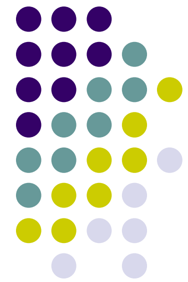
1200
1000
800
600
400
200
0

1 3 5 7 9 11 13 15 17 19 21 23 25

Payload Size (octet)

None
Origin
All

data transport (6)



Server-Client Data Send Maximum

Invocation Time (usec)

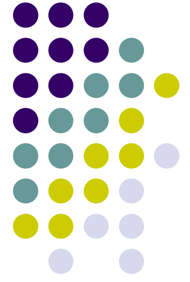
2500
2000
1500
1000
500
0

0
2
4
10
30
50
200
400
1000
3000
5000
20000
40000

Payload Size (octet)

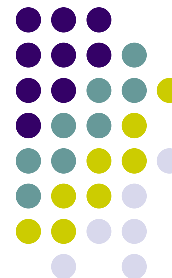
None
Origin
All

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

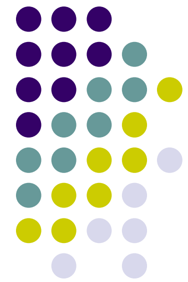
Invocation Time (usec)

200
180
160
140
120
100
80
60
40
20
0

1 3 5 20 40 100 300 500 2000 4000 10000 30000 50000

No. of Server Objects

None
Origin
All



request dispatch (2)

Multi-server Dispatching Average

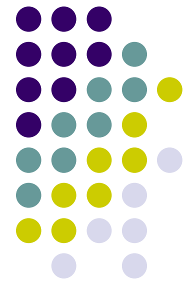
Invocation Time (usec)

450
400
350
300
250
200
150
100
50
0

1 3 5 20 40 100 300 500 2000 4000 10000 30000 50000

No. of Server Objects

None
Origin
All



request dispatch (3)

Multi-server Dispatching Maximum

Invocation Time (usec)

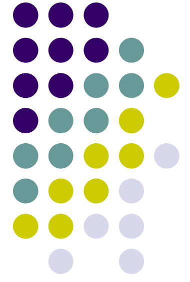
18000
16000
14000
12000
10000
8000
6000
4000
2000
0

1 3 5 20 40 100 300 500 2000 4000 10000 30000 50000

No. of Server Objects

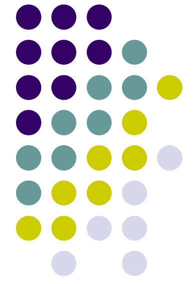
None
Origin
All

request dispatch (4)



- X-axis: number of instantiated objects
- Y-axis: invocation time in microsecunds
- More objects ➔ 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



remote

Multi-threaded Invocation

Invocation Time (usec)

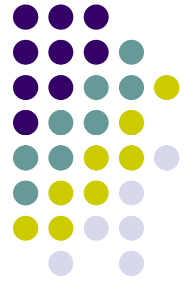
250
200
150
100
50
0

1 2 3 4 5 6 7 8 9 10 11

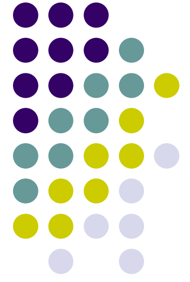
No. of Threads

None
Origin
All

parallel execution (2)



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



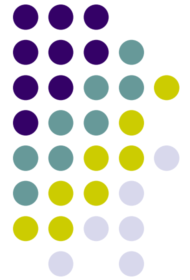
combined execution

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**

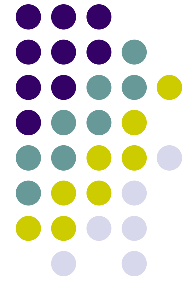
	A	B	C	D	E	F	G
Re-factored	7036	2161	193	954	1061	6020	2689
Original	2332	2395	199	976	1055	1686	3130
Combined	6142	2366	238	1003	1103	5396	3111

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

	None	*Original*	*All*
L1 Instruction Misses	368869	380404	404187
L2 Miss Rate	3.25%	3.83%	4.4%