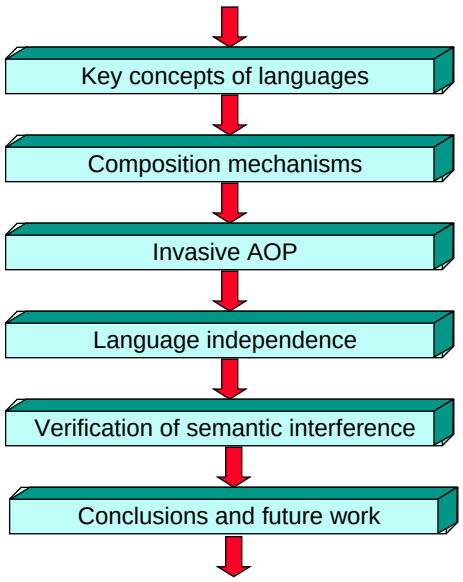
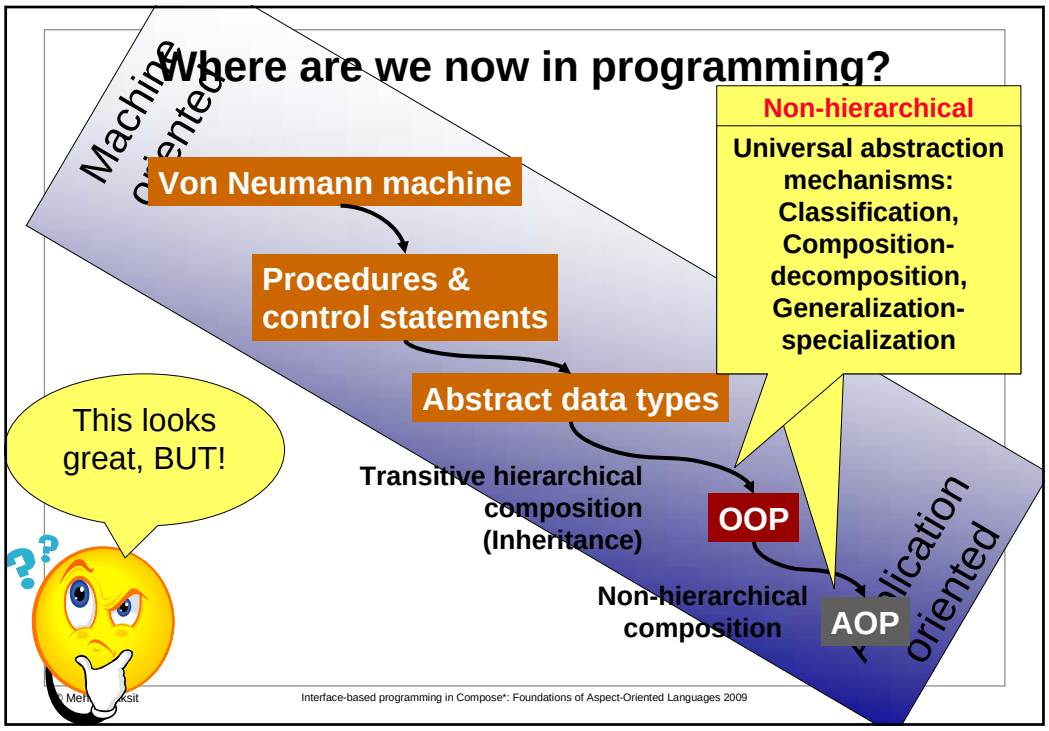
Interface-based aspect-oriented programming in Compose*: its language independency, semantic point-cuts and aspect-interface detection possibilities

Mehmet Aksit
Chair Software Engineering (TRESE)
Department of Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede, The Netherlands
aksit@ewi.utwente.nl
trese.cs.utwente.nl/

Table of contents

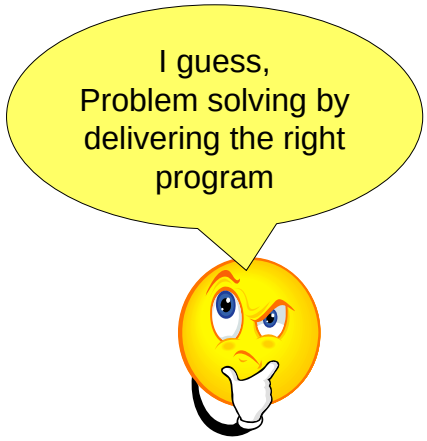




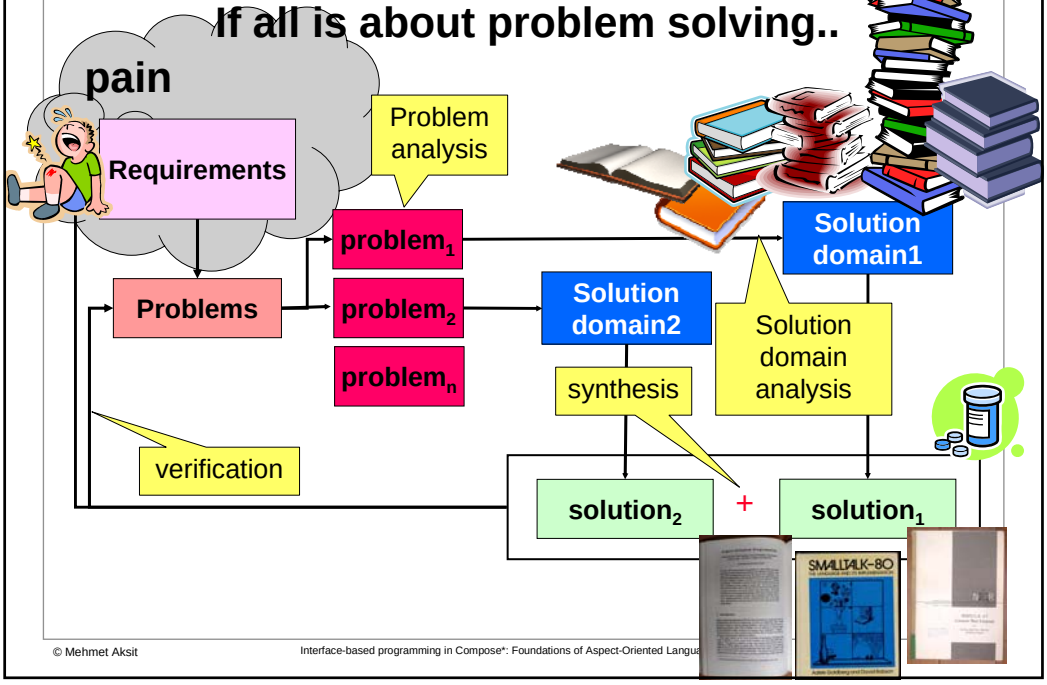
What are the key concepts of programming languages?

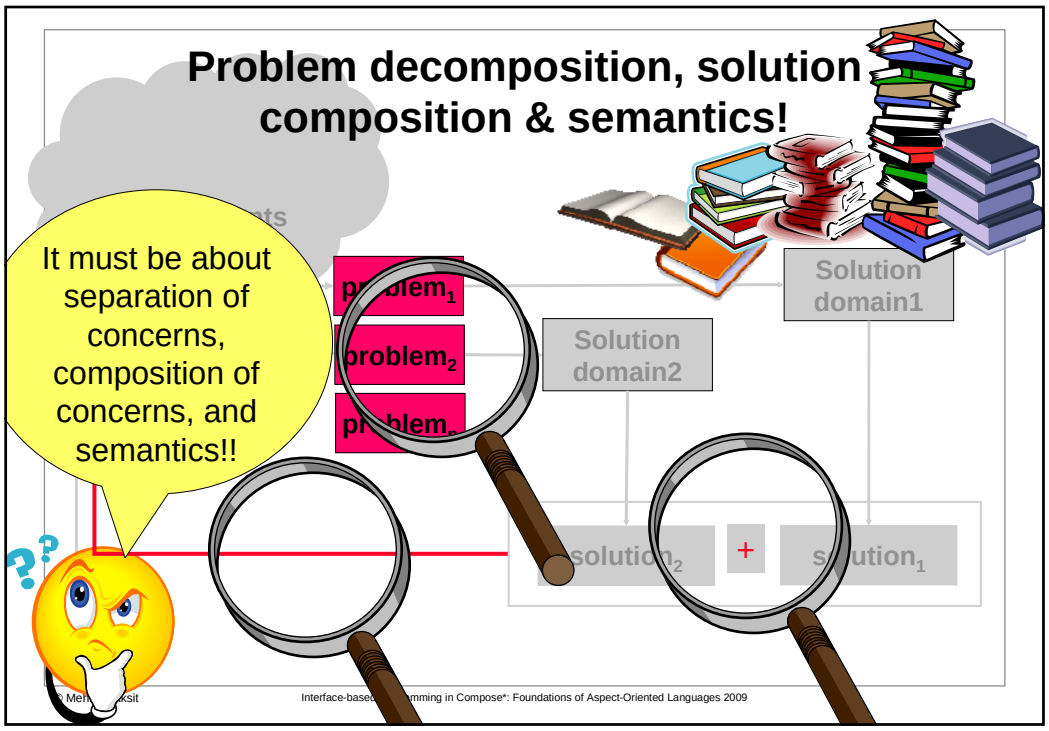
(for programming in the large!)

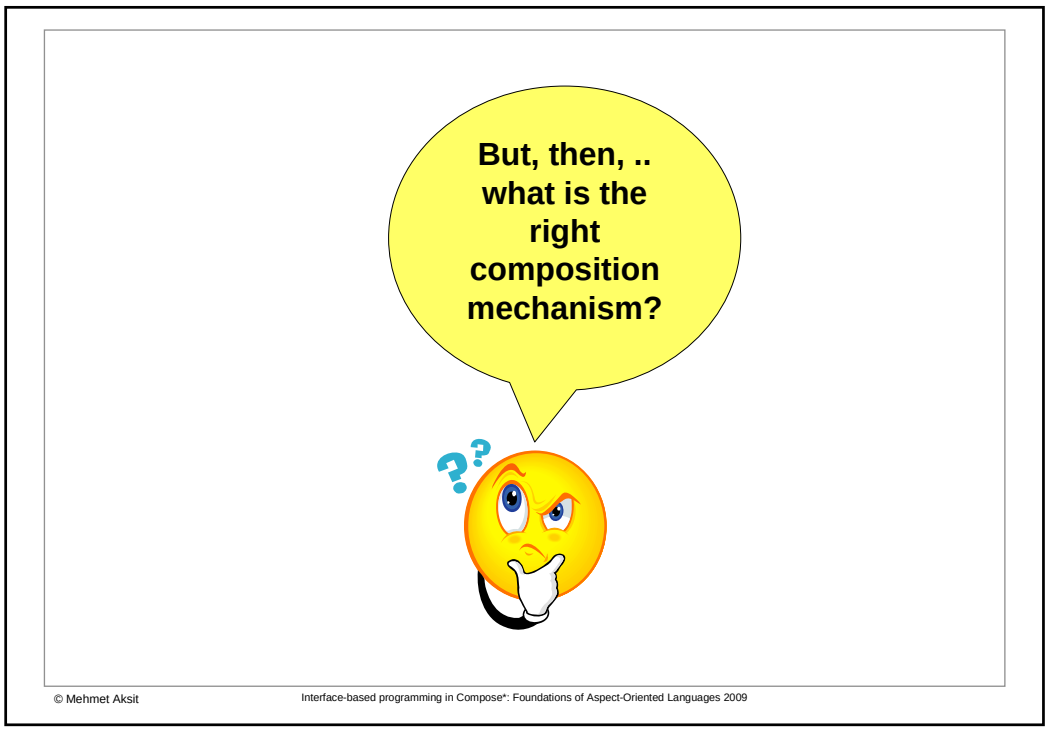
What are programming languages good for?



If all is about problem solving..




Problem decomposition, solution composition & semantics!
It must be about separation of concerns, composition of concerns, and semantics!!
problem1
problem2
Solution domain1
Solution domain2
+
© Mehmet Aksit
Interface-based programming in Compose*: Foundations of Aspect-Oriented Languages 2009


But, then, .. what is the right composition mechanism?
© Mehmet Aksit
Interface-based programming in Compose*: Foundations of Aspect-Oriented Languages 2009

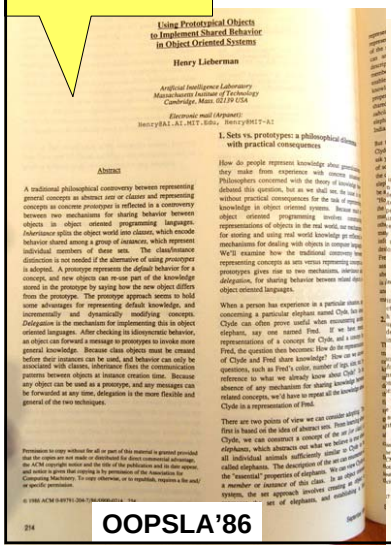
Once upon a time..

Delegation, inheritance & reflection debate

Delegation is better

Delegation is inheritance

I can do anything with reflection


Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems

Henry Lieberman

OOPSLA'86


Delegation Is Inheritance

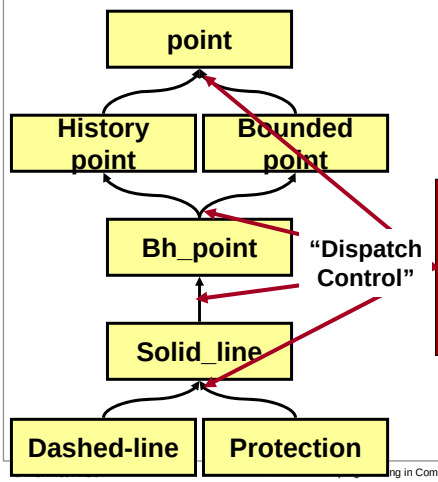
Lynn Andrea Stein
Brown University
Department of Computer Science
Providence, R.I. 02912

OOPSLA'87

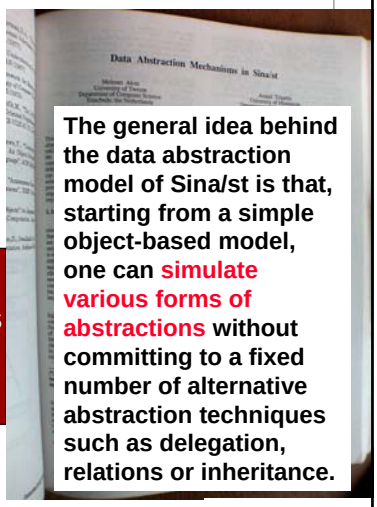

OOPSLA'87

Non-invasive generalization approaches (1)

The Sina language



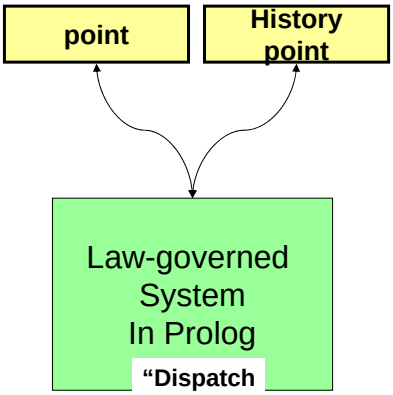
Reuse mechanisms "interface Predicates"

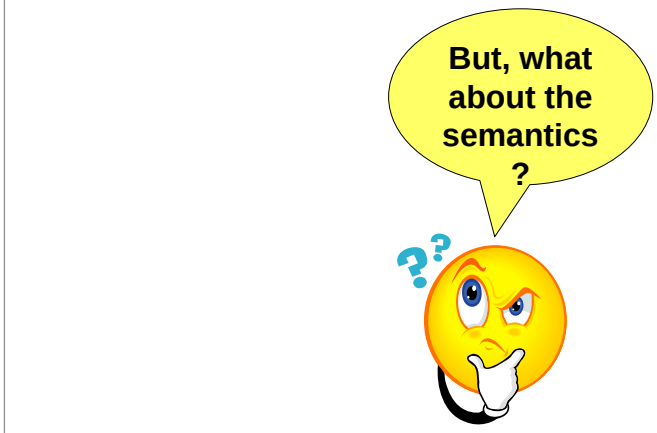


OOPSLA'88

Non-invasive generalization approaches (2)

The law-governed system





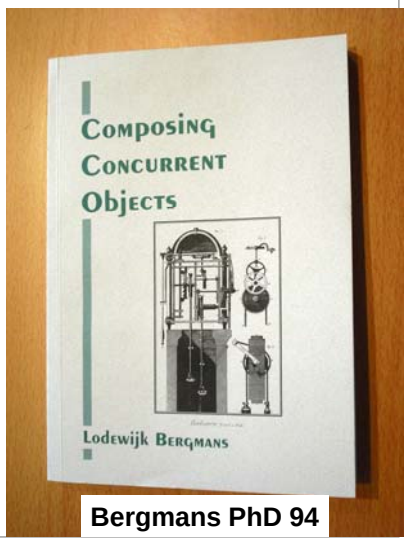
Then some time has passed..

Synchronization inheritance anomalies


4 Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages

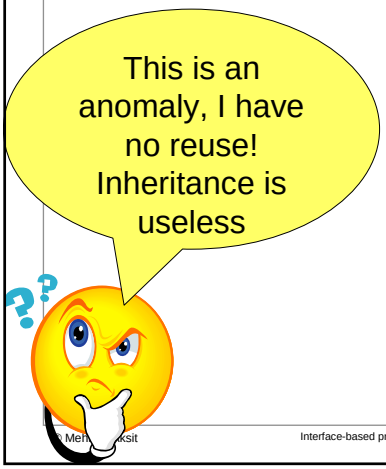
4.1 Introduction

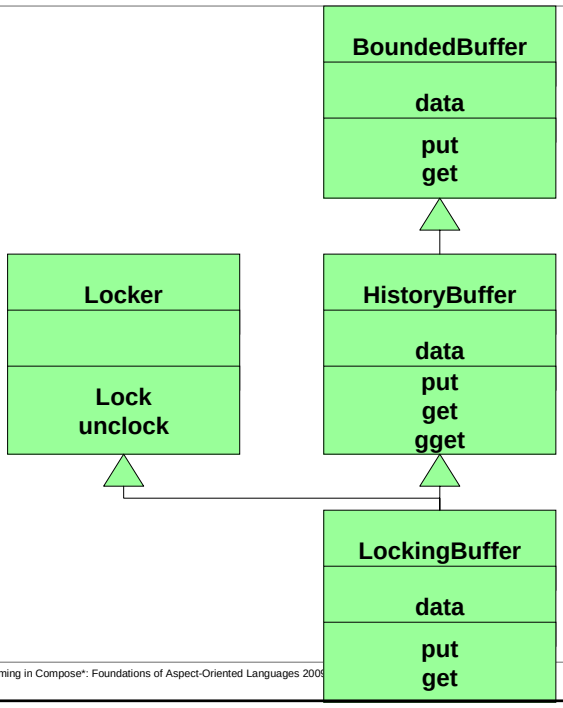
MIT Press: Research directions In concurrent OOP



Bergmans PhD 94

Synchronization inheritance anomalies





Research on synchronization inheritance anomalies attracted some attention


Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming

ECOOP'93

Real-time inheritance anomalies


Real-Time Specification Inheritance Anomalies and Real-Time Filters

ECOOP'94

Obstacles in object-oriented programming included seven “crosscutting-like” problems

Coordinated behavior: tangled & scattered

Inheritance & delegation: dispatching aspect

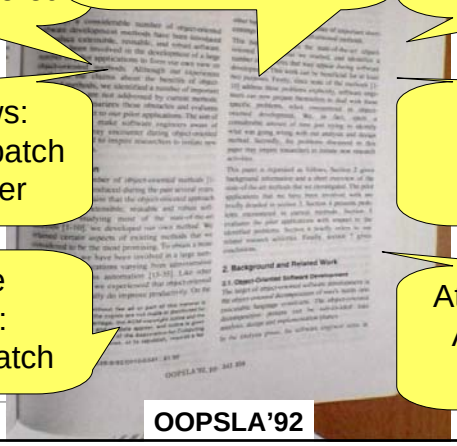
Synchronization inheritance: synchron. dispatch

Multiple views: conditional dispatch & logical order

Arbitrary inheritance: Any dispatch

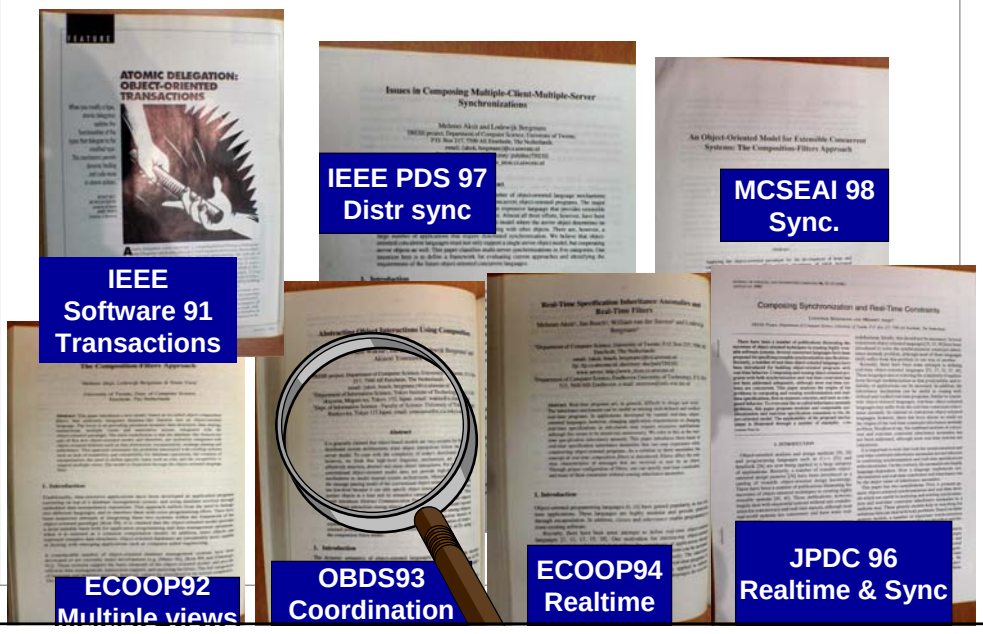
Associative inheritance: predicate dispatch

Atomic delegation: Atomic dispatch & logical order



OOPSLA’92

Composition-filters without superimposition



Aspect-identification and aspect-based re-engineering example (from OBDS'93)

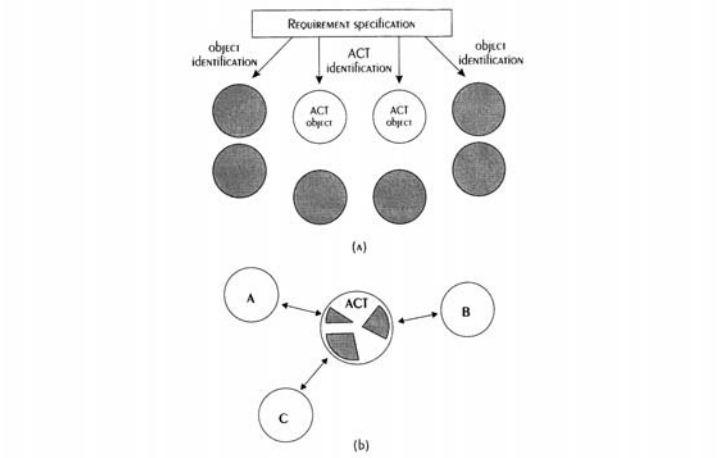
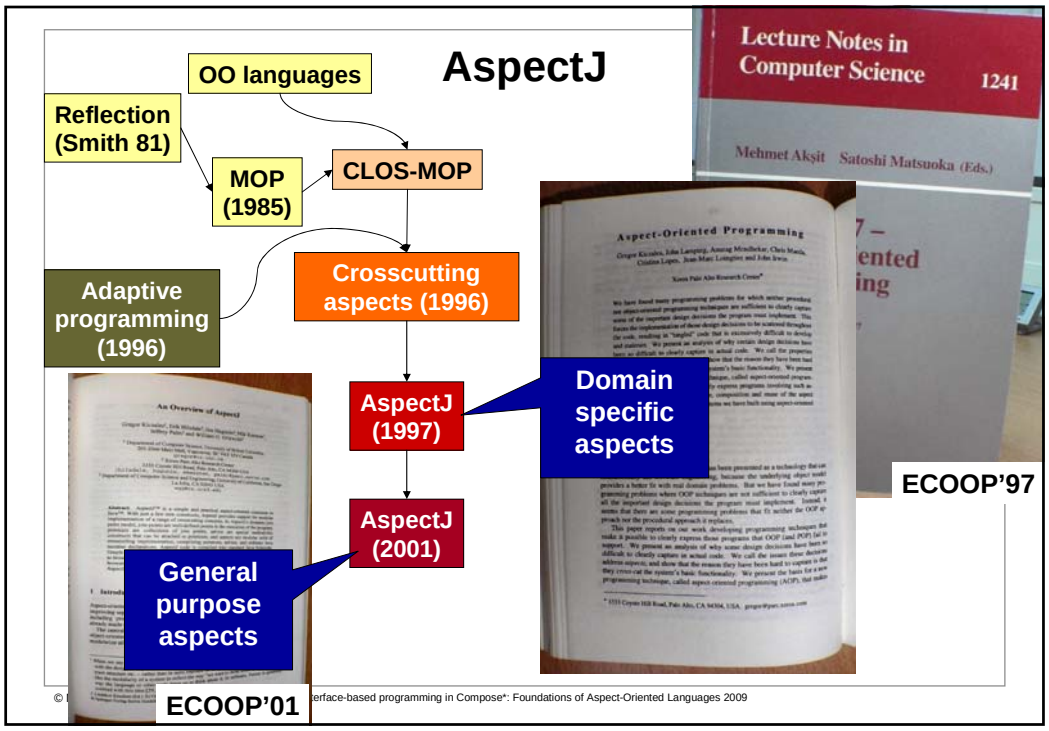


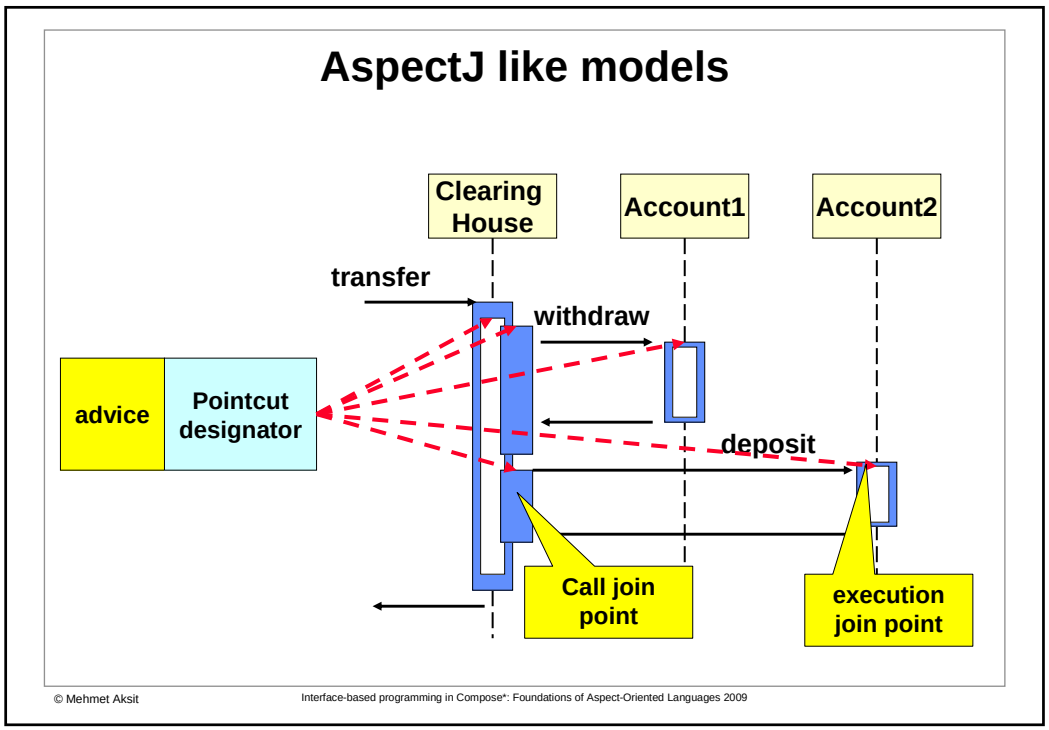
Figure 7. Identifying ACTs using (a) requirement specification and (b) object interaction patterns.

Then some more time has passed..

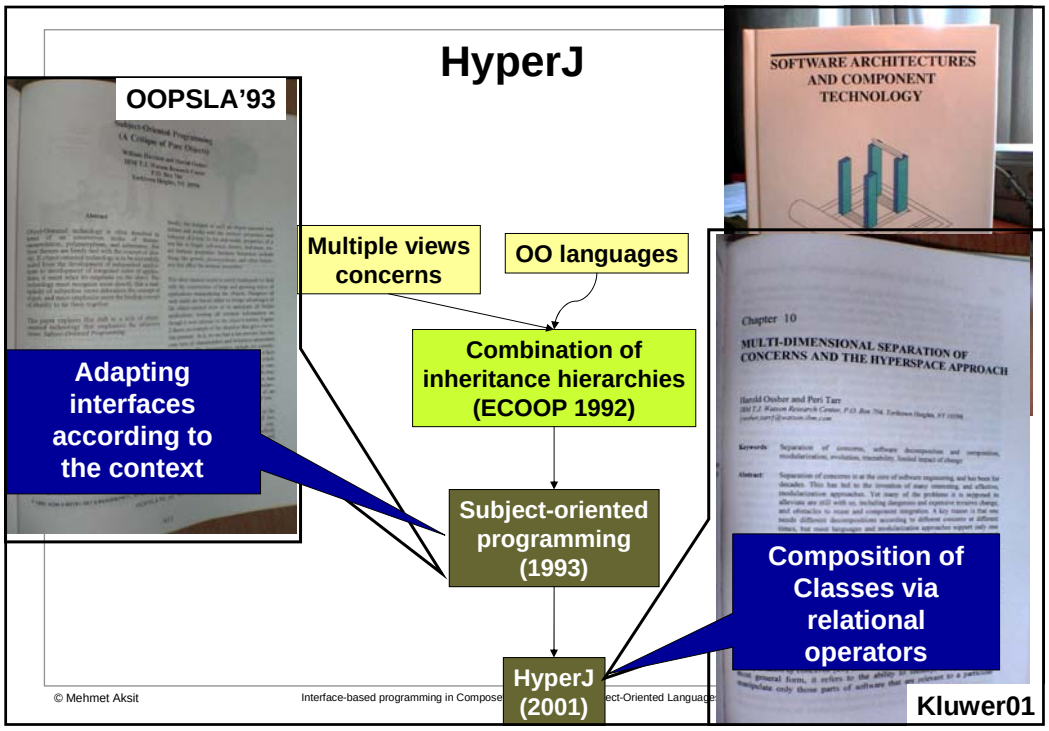
Invasive approaches

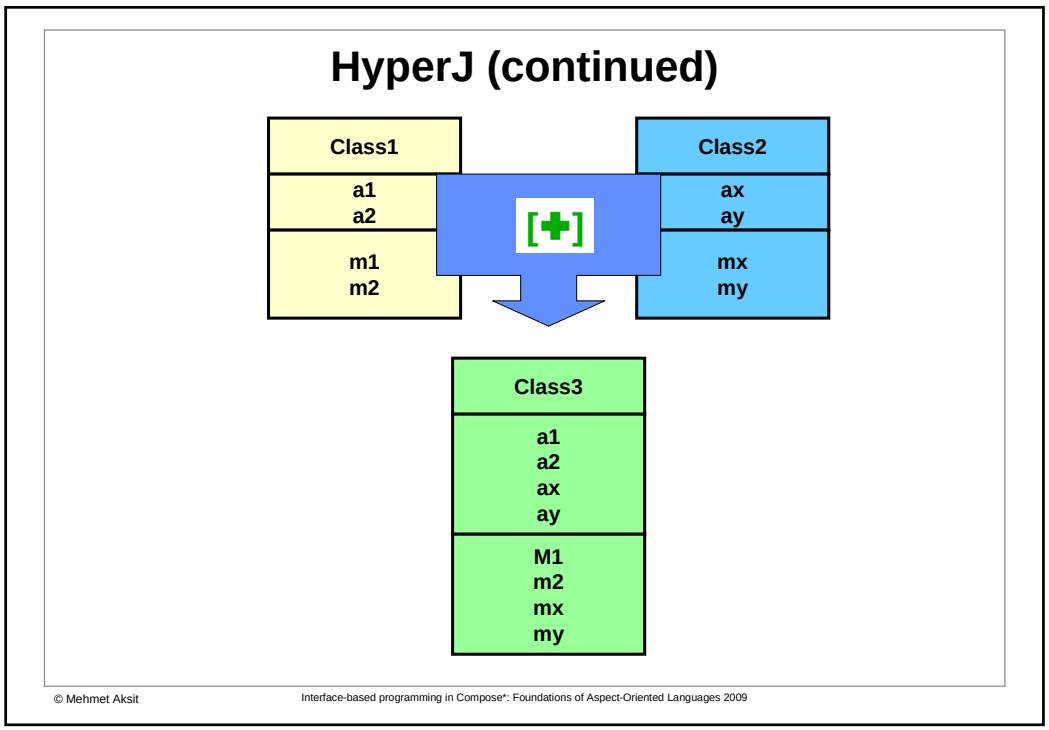
Examples of AOP languages: AspectJ, HyperJ and framework-based approaches


AspectJ
OO languages
Reflection (Smith 81)
MOP (1985)
CLOS-MOP
Adaptive programming (1996)
Crosscutting aspects (1996)
AspectJ (1997)
AspectJ (2001)
Domain specific aspects
General purpose aspects
ECOOP'01
ECOOP'97
Lecture Notes in Computer Science 1241
Mehmet Akşit Satoshi Matsuoka (Eds.)
Aspect-Oriented Programming


AspectJ like models
Clearing House
Account1
Account2
transfer
withdraw
deposit
advice
Pointcut designator
Call join point
execution join point

Interface-based aspect-oriented programming in Compose*:





Framework-based aspect-oriented approaches

- Supported within a platform with a number of libraries & tools
- Provides aspect weaving mechanisms as a tool
- Becoming more and more popular
- Mostly implement Composition Filters like mechanisms (using proxies/interceptors, etc.)

AOP languages have some concerns

Being able to compose crosscutting concerns is an additional benefit but, crosscutting concern mechanisms are "tangled with language semantics" (language dependent);

Extending existing languages with AOP constructs makes the languages too complex;

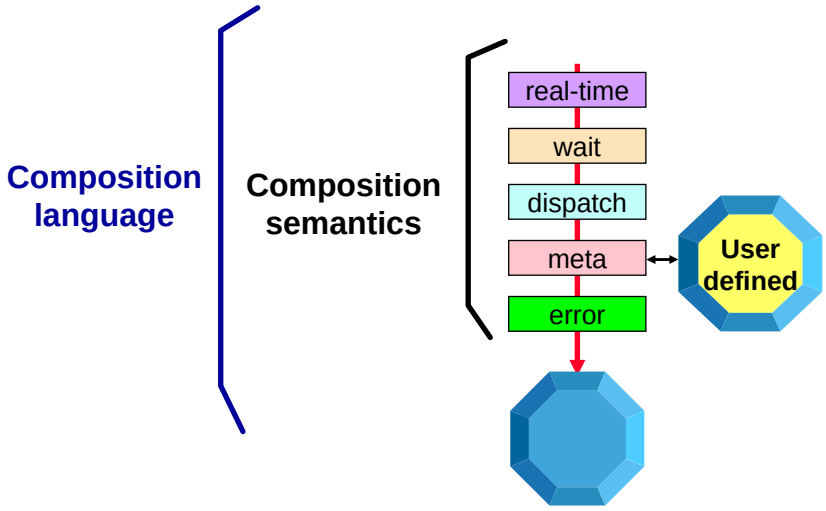
Verifying semantic correctness of compositions is still difficult;

Joinpoint level of composition is too low-level .

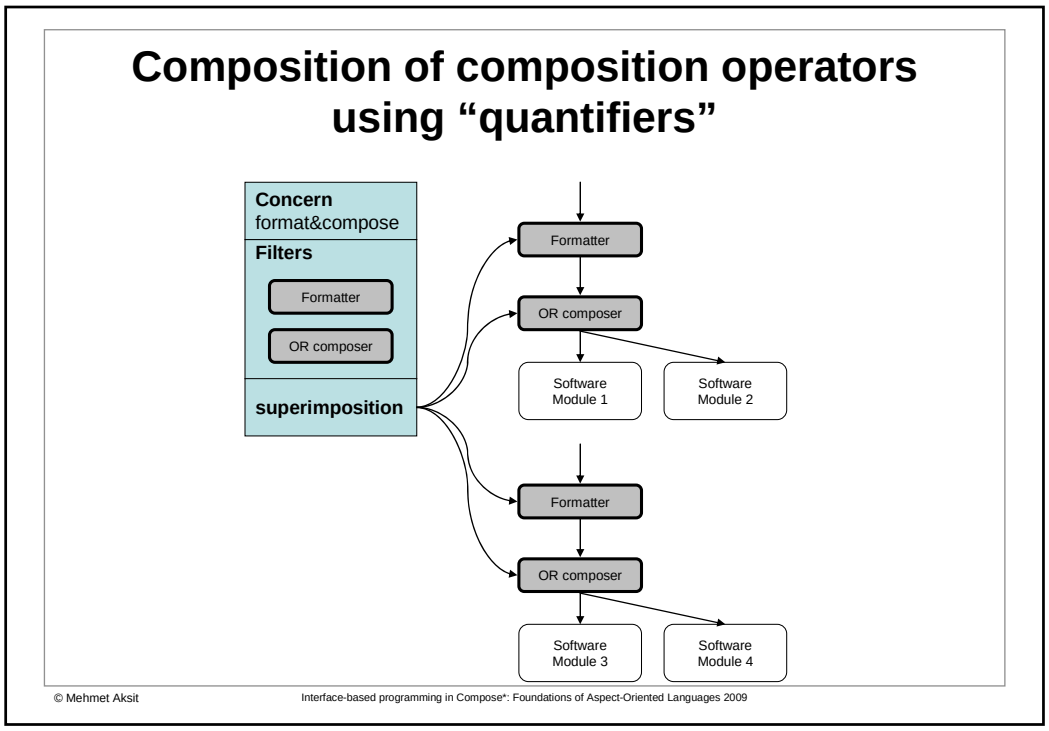
How to address these problems?

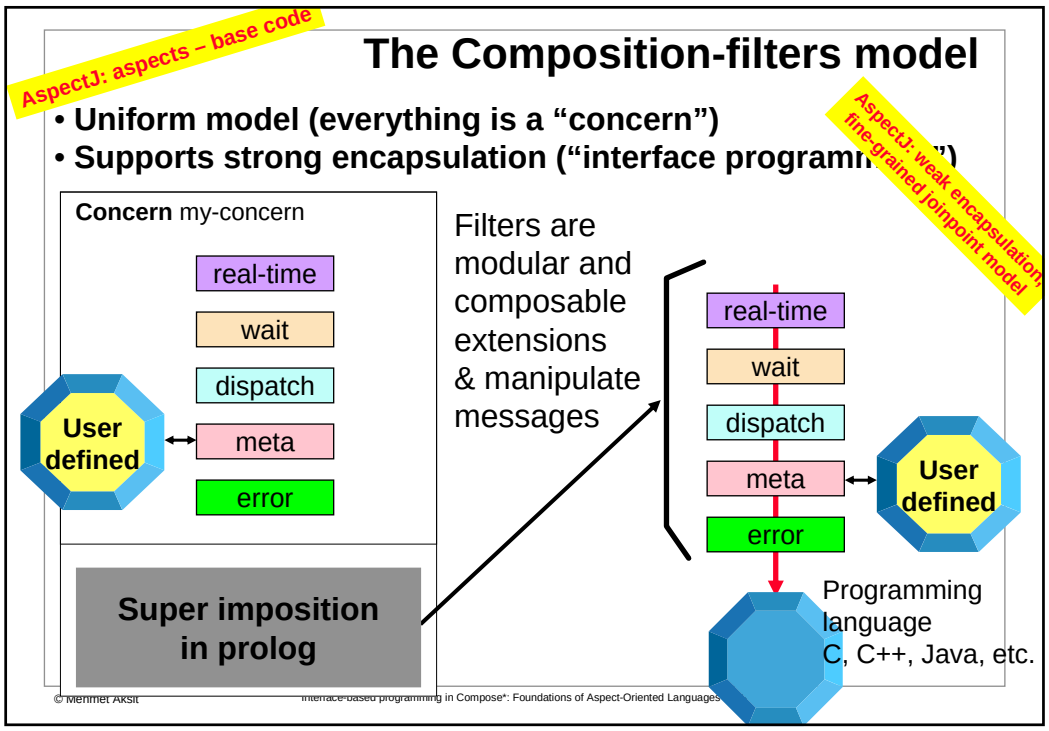
Step 1:
Separation of composition operators from concerns

Composition operators as dedicated interface extensions



Step 2: Composing composition operators with concerns


Composition of composition operators using “quantifiers”
Concern format&compose
Filters
Formatter
OR composer
superimposition
Formatter
OR composer
Software Module 1
Software Module 2
Formatter
OR composer
Software Module 3
Software Module 4
© Mehmet Aksit
Interface-based programming in Compose*: Foundations of Aspect-Oriented Languages 2009


AspectJ: aspects – base code
The Composition-filters model
• Uniform model (everything is a “concern”)
• Supports strong encapsulation (“interface programm
AspectJ: weak encapsulation, fine-grained joinpoint model
Concern my-concern
real-time
wait
dispatch
meta
error
User defined
Filters are modular and composable extensions & manipulate messages
real-time
wait
dispatch
meta
error
User defined
Super imposition in prolog
Programming language C, C++, Java, etc.
© Mehmet Aksit
Interface-based programming in Compose*: Foundations of Aspect-Oriented Languages

First two claims of composition filters:

language independence

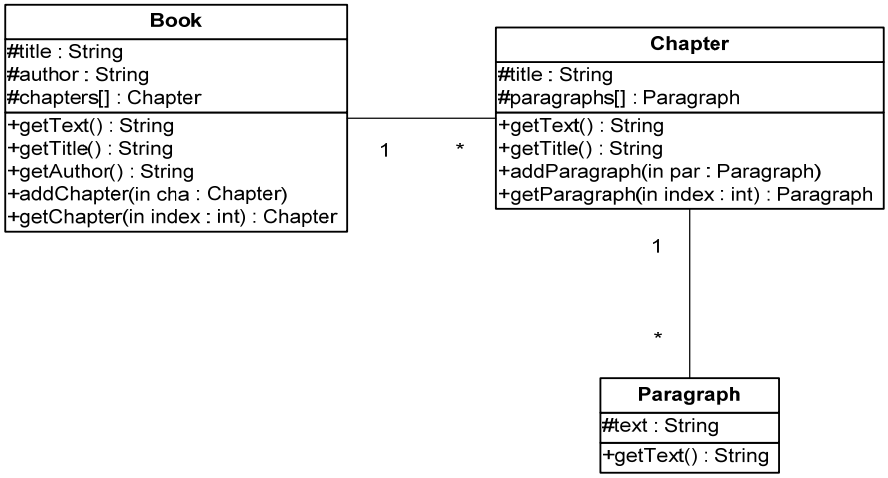
domain specific aspects as filters

Simulated demonstration of the proof of these two claims

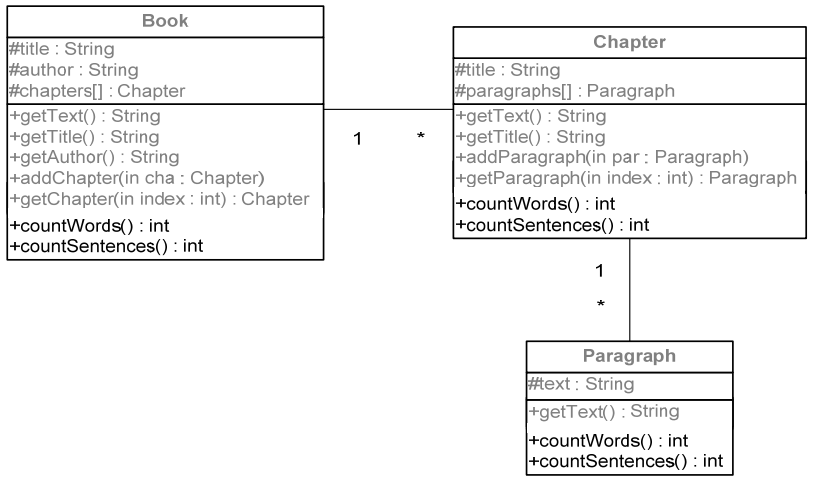
What we will show in the demo

- **Composing a *word counting* 'feature' with the *book shelf* application as a modular concern**
- **Composing a *cache* optimization concern with the w*ord counting* concern.**
- **Language/platform independence**

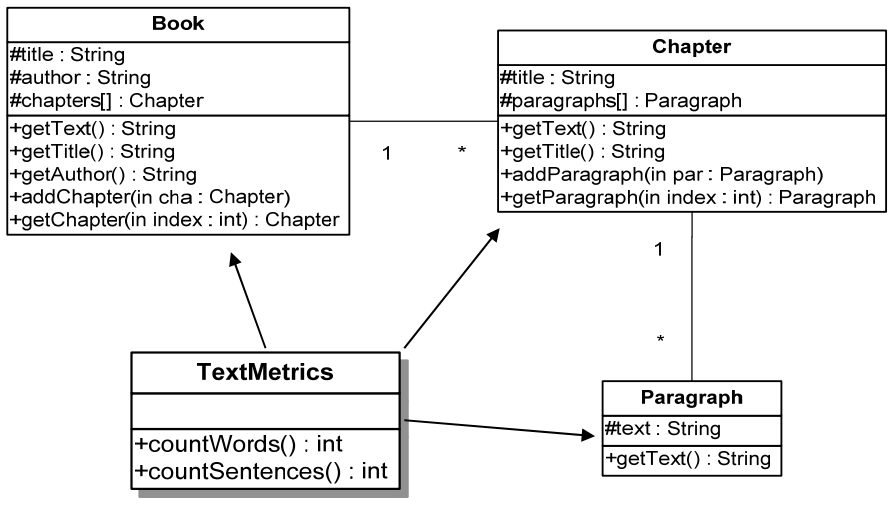
Example: Bookshelf



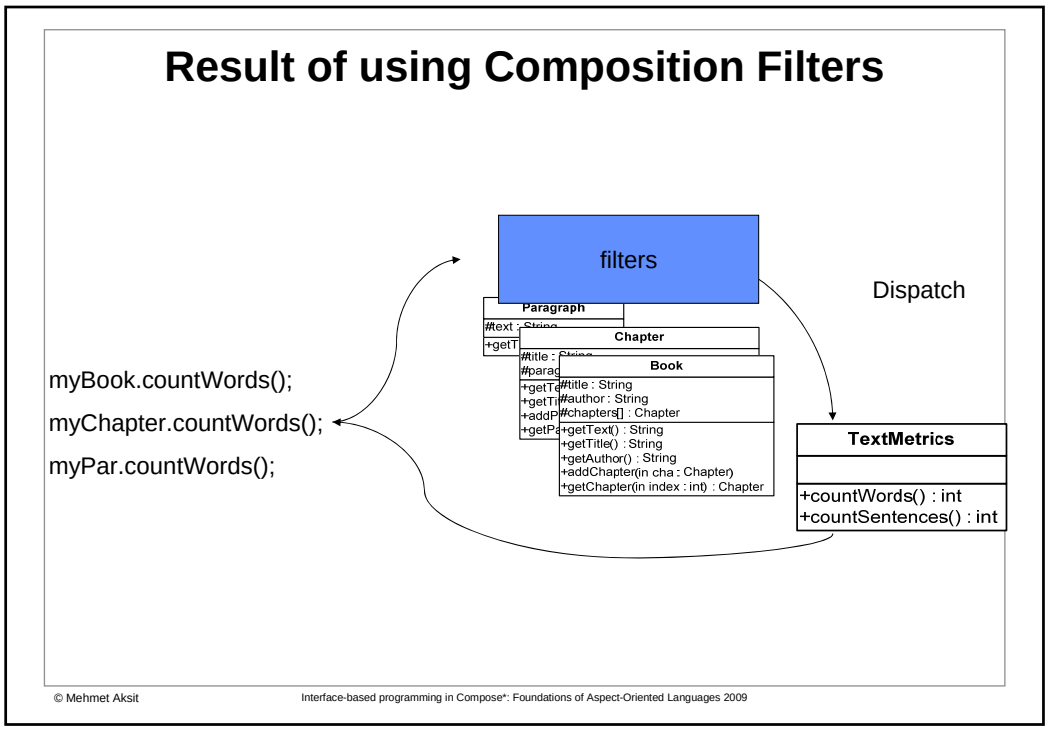
Counting words and sentences

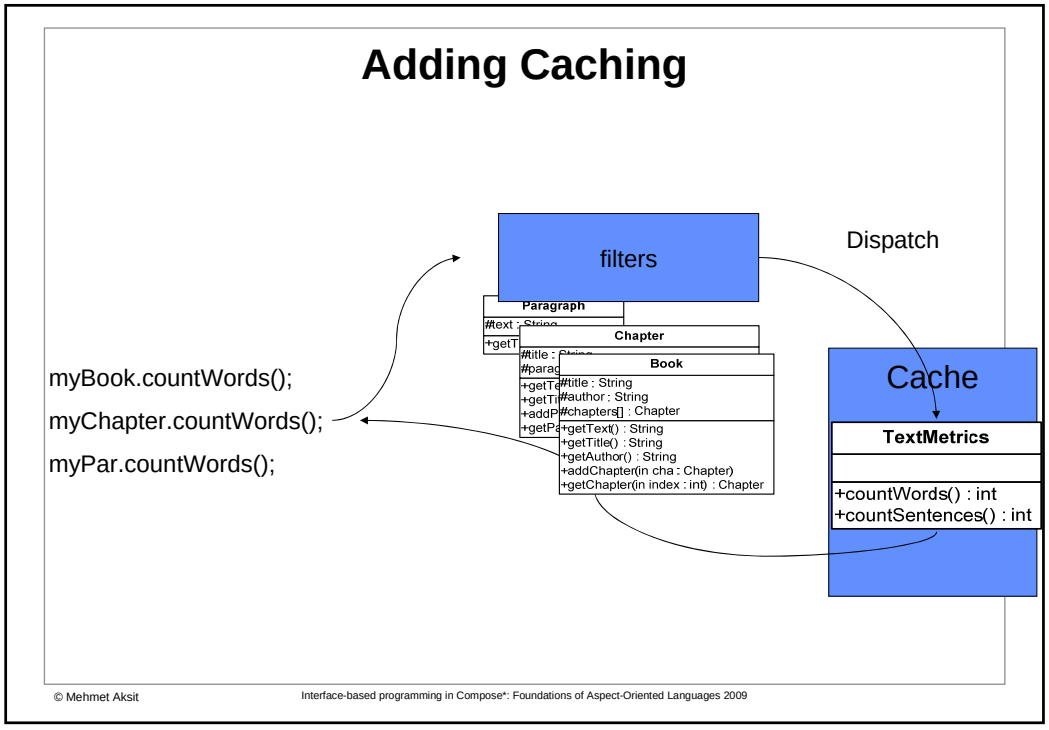


Design using Composition Filters



Interface-based aspect-oriented programming in Compose*:





Conclusion of the first demonstration

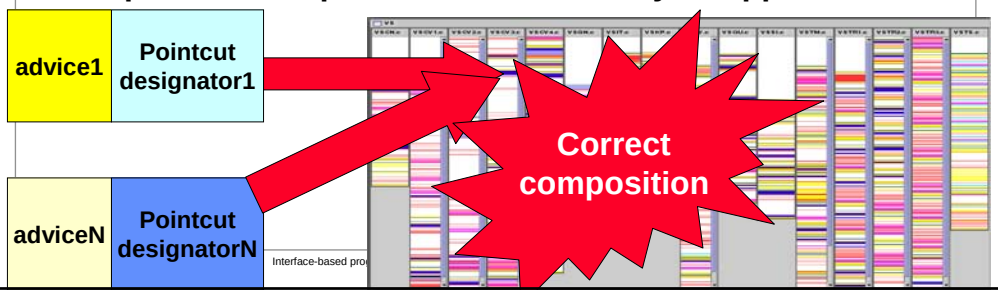
- Composing a *word counting* 'feature' with the *bookshelf* application
- Composing the *caching* optimization concern as a high level concern with the *word counting* concern
- Composition Filters tools work on C*/* .NET / Java
  - *BookShelf* + *WordCounting* + *Caching* in Java
  - *Fibonacci* + *Caching* in C
  - Same concern code (*Caching*) reused!
- Efficiency: filters can be in-lined -> low overhead

The third claim of the Composition-Filters

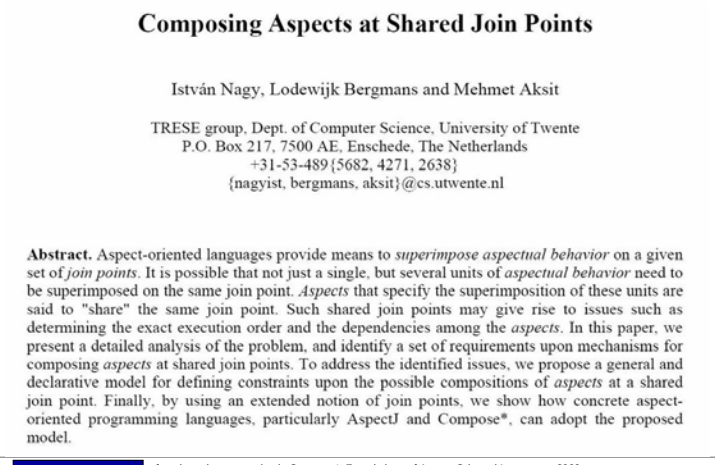
Should be easier to verify the semantic correctness of filter compositions since filters are modular extensions;

CF's: Should be easier to verify

- Controlling the order of aspect superimpositions using partial, advice-based priority specifications
- Resource-model based aspect-interference analysis approach
- Graph- based aspect-interference analysis approach



Controlling the order of aspect superimpositions using partial, advice-based priority specifications



Composing Aspects at Shared Join Points

István Nagy, Lodewijk Bergmans and Mehmet Aksit

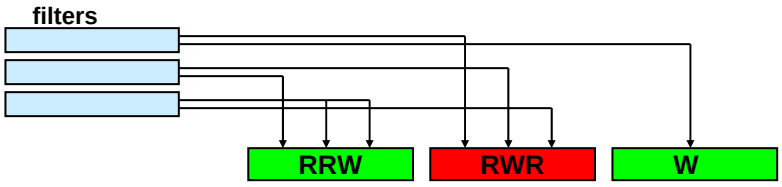
TRESE group, Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
+31-53-489{5682, 4271, 2638}
{nagyist, bergmans, aksit}@cs.utwente.nl

Abstract. Aspect-oriented languages provide means to *superimpose aspectual behavior* on a given set of *join points*. It is possible that not just a single, but several units of *aspectual behavior* need to be superimposed on the same join point. *Aspects* that specify the superimposition of these units are said to "share" the same join point. Such shared join points may give rise to issues such as determining the exact execution order and the dependencies among the *aspects*. In this paper, we present a detailed analysis of the problem, and identify a set of requirements upon mechanisms for composing *aspects* at shared join points. To address the identified issues, we propose a general and declarative model for defining constraints upon the possible compositions of *aspects* at a shared join point. Finally, by using an extended notion of join points, we show how concrete aspect-oriented programming languages, particularly AspectJ and Compose*, can adopt the proposed model.

Resource-model based aspect-interference analysis approach

Serialization-based aspect-interference approach (continued)

From the set of filters (from multiple filtermodules), per resource a sequence of operations is compiled, e.g.:



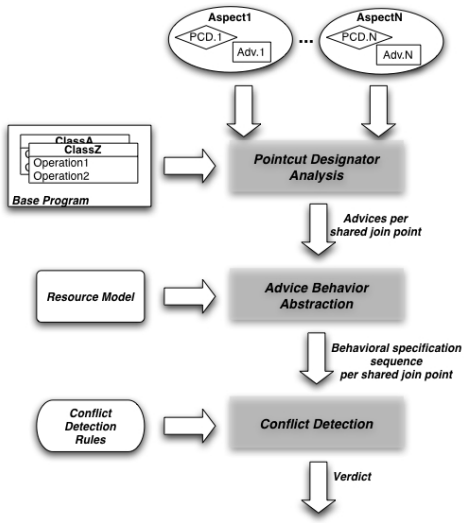
NB: not for all combinations of accept & reject a corresponding message exists: this is filtered out

These sequences are matched with conflict patterns (defined per resource), e.g. “R*WR

So: RR and RRRW don’t match, but RWR does match → conflict

Serialization-based aspect-interference approach (continued)

The process:



Graph-based aspect-interference analysis

Graph-based aspect-interface analysis approach

Detect aspect-interference on a shared joinpoints without:

- **(formally) specifying the aspect**
- **(formally) specifying the base program**
- **specifying the base language semantics**

“Only analyse the aspects, not the base program.”

Graph-based aspect-interface approach (continued)

generate abstract syntax graph
substitute filter-types with filter-actions
construct control flow graph
create runtime state (method call)

simulation: generate state space

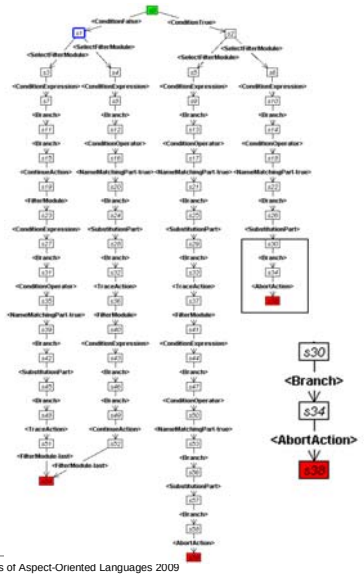
- **Fixed production system for operational semantics**

verification: analyse state space

Graph-based aspect-interface approach (continued)

Two kinds of non-determinism:

- Assignment of unknown runtime values (conditions)
- Selection of filter module order



Conclusions about the semantic interference detection

- Prioritization of compositions is sometimes necessary. Prioritization must be based on partial specifications. This is integrated in filters.
- Well-defined interfaces of concerns (filters) and/or concerns with well-defined semantics (domain specific concerns) make it easier to analyze & verify semantic interference among concerns (for example by using graph-based verification)
- In case of large state spaces, concerns can be abstracted as semantic signatures (related to a resource model) and be analyzed based on the references to the resource-model.

Our future work

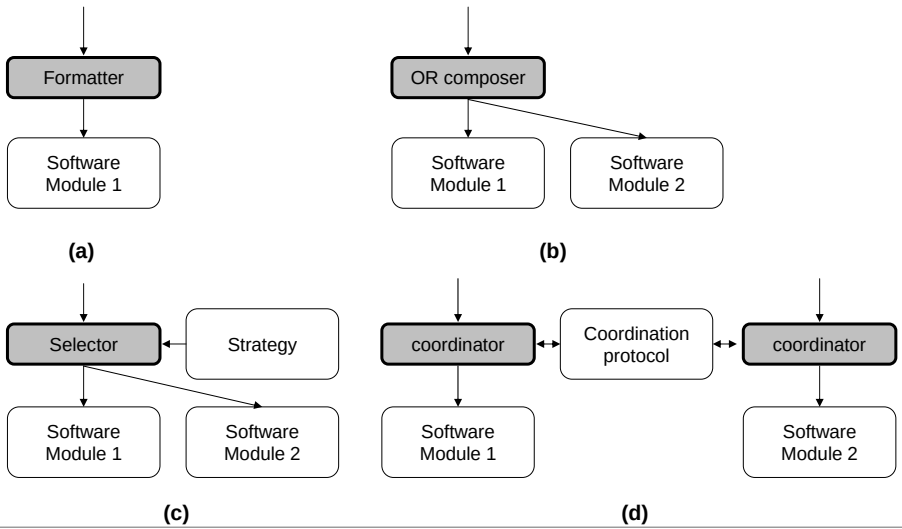
Define larger set of filters (filter library) identified from practical needs

Work further on semantic composition also for not shared joinpoints

Composition patterns

Higher-level compositions

Larger set of explicit composition operator



Conclusions

- **Non-invasive AOP has some advantages; language independence, semantic verification, etc.**
- **Non-invasive AOP must be supported with a high-level compositional language.**
- **Invasive AOP is probably good for applications like run-time verification kind of applications, but the ideal level of detail of the joinpoints is difficult to determine. Open joinpoint models can be the answer!**