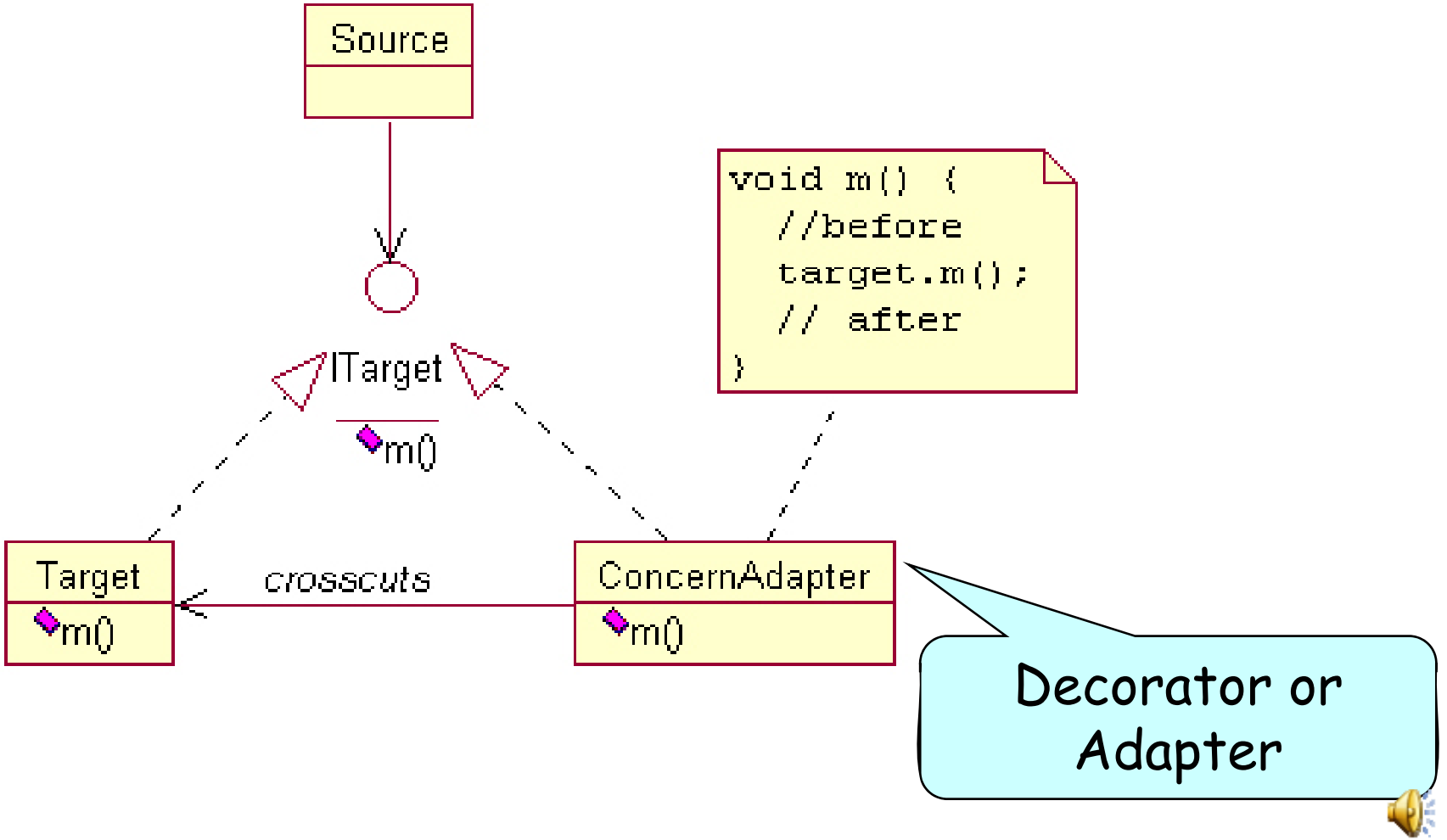
Aspect-oriented programming

Paulo Borba
Informatics Center
Federal University of Pernambuco

Aspect-oriented patterns and refactoring

Paulo Borba
Informatics Center
Federal University of Pernambuco

AOP or good OO design?



The real question is...

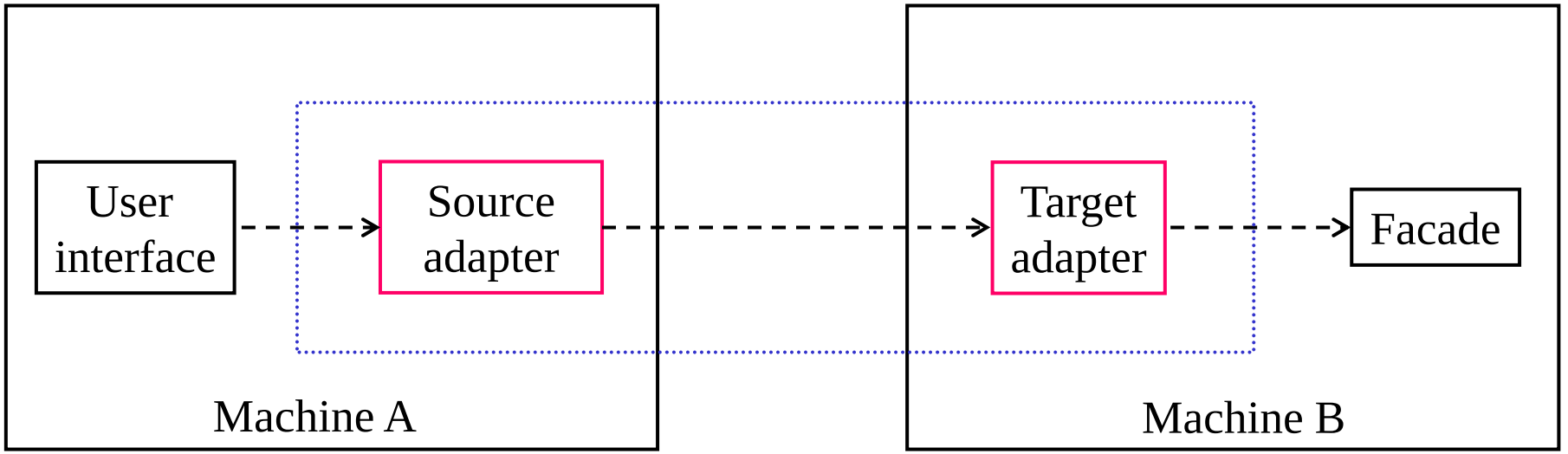
shall we implement typical (like GoF's) design patterns with...

- AOP?
- OOP?

Com decoradores OO...

- Escrevemos mais código
- A ligação entre o decorador e o objeto decorado
  - é explicita e invasiva
  - não altera o comportamento de chamadas internas para o objeto decorado
  - não tem acesso ao objeto fonte (*source*)
  - pode ser modificada dinamicamente

Same with the distributed adapters pattern (DAP)



But AOP is not always the best choice... (Garcia et al 2006)

	CDC		CDO		CDLOC		Scalability		Superior
Design pattern	OO	AO	OO	AO	OO	AO	OO	AO	solution
Abstract Factory	14	16	35	35	34	34	No	No	OO
Adapter*	8	7	30	22	32	16	No	Yes	AO+
Bridge	12	13	24	26	16	16	No	No	OO
Builder	9	10	29	30	8	8	Yes	Yes	OO
CoR*	9	3	15	21	50	4	No	Yes	AO
Command*	17	11	23	16	38	21	No	Yes	AO
Composite*	18	9	149	28	70	48	No	No	AO
Decorator*	18	8	31	8	38	6	No	Yes	AO+
Façade	Same implementations for Java and AspectJ								
Factory Method	14	16	23	23	18	18	No	No	OO
Flyweight*	10	13	10	12	20	26	No	No	OO
Interpreter	13	13	26	26	38	38	No	No	=
Iterator*	10	6	20	20	18	14	No	No	AO
Mediator*	13	5	18	6	36	10	No	Yes	AO
Memento*	11	10	23	24	44	40	No	No	AO
Observer*	14	9	49	9	92	20	No	Yes	AO
Prototype*	7	3	7	2	30	8	No	Yes	AO+
Proxy*	11	11	38	19	8	2	No	Yes	AO+
Singleton*	6	6	6	1	6	2	Yes	Yes	AO+
State*	10	10	78	78	30	30	No	No	=
Strategy*	14	12	20	17	18	16	No	No	AO
Template Method*	15	16	24	24	20	20	No	No	OO
Visitor*	20	9	50	23	34	14	No	Yes	AO+
Success total	6 vs. 12		5 vs. 11		1 vs. 14		2 vs. 11		6 vs. 14

considering different metrics…

	CBC		DIT		LCOO		Superior
Design pattern	OO	AO	OO	AO	OO	AO	solution
Abstract Factory	37	44	7	7	1	1	OO
Adapter*	5	5	2	1	–	–	AO
Bridge	17	18	2	2	0	0	OO
Builder	2	3	2	2	12	6	OO
CoR*	29	28	2	2	1	13	OO
Command*	21	34	7	7	3	4	OO
Composite*	47	23	2	2	463	82	AO
Decorator*	3	14	3	1	0	0	AO
Façade	Same implementations for Java and AspectJ						
Factory Method	22	24	2	2	3	0	OO
Flyweight*	11	17	2	2	0	1	OO
Interpreter	17	23	5	5	0	0	OO
Iterator*	12	13	2	2	0	0	=
Mediator*	41	34	2	2	5	1	AO
Memento*	13	18	1	2	0	0	OO
Observer*	45	40	2	2	80	30	AO
Prototype*	7	13	2	2	0	0	OO
Proxy*	11	39	2	2	0	0	AO
Singleton*	11	22	2	2	5	0	AO
State*	17	10	2	2	106	93	AO
Strategy*	18	32	2	2	–	–	OO
Template Method*	2	3	2	2	–	–	OO
Visitor*	41	28	2	2	27	2	AO
Success total	**15 vs. 6**		**1 vs. 2**		**3 vs. 8**		**12 vs. 9**

including size

	NOA		WOC		LOC		Superior
Design pattern	OO	AO	OO	AO	OO	AO	solution
Abstract Factory	9	9	37	41	231	265	OO
Adapter*	3	1	34	32	67	61	AO
Bridge	1	1	40	44	156	161	OO
Builder	7	7	50	51	168	177	=
CoR*	8	2	40	64	213	234	=
Command*	6	4	26	29	198	206	=
Composite*	19	12	169	63	501	283	AO
Decorator*	1	0	34	16	88	69	AO
Façade	Same implementations for Java and AspectJ						
Factory Method	1	1	17	17	135	146	=
Flyweight*	7	7	30	36	119	132	OO
Interpreter	14	14	99	99	216	219	=
Iterator*	9	9	50	53	164	163	=
Mediator*	21	17	51	40	253	253	AO
Memento*	6	6	32	31	128	179	OO
Observer*	26	21	134	117	363	265	AO
Prototype*	6	6	38	33	142	147	AO
Proxy*	9	3	105	38	248	190	AO
Singleton*	30	26	25	21	238	251	AO
State*	13	20	164	110	367	374	=
Strategy*	5	1	62	58	251	264	AO
Template Method*	0	0	46	46	125	128	=
Visitor*	13	13	105	57	289	222	AO
Success total	**1 vs. 10**		**7 vs. 12**		**14 vs. 7**		**4 vs. 10**

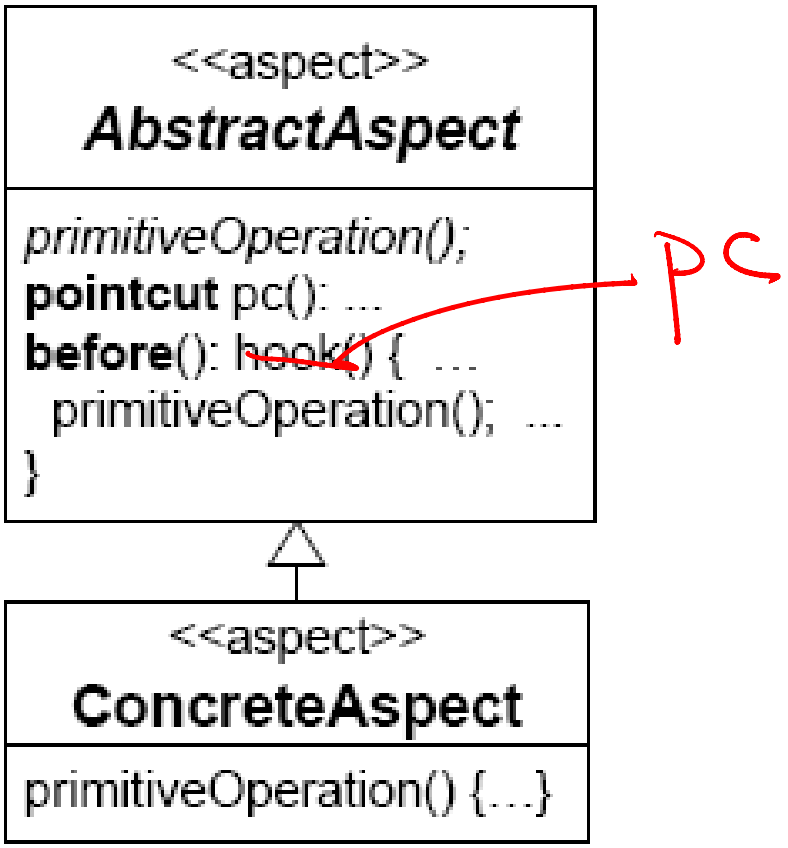
Metrics suite

Attributes	Metrics	Definitions
Separation of concerns	Concern diffusion over components (CDC)	Counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them
	Concern diffusion over operations (CDO)	Counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them
	Concern diffusion over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch"
Coupling	Coupling between components (CBC)	Counts the number of other classes and aspects to which a class or an aspect is coupled
	Depth inheritance tree (DIT)	Counts how far down in the inheritance hierarchy a class or aspect is declared
Cohesion	Lack of cohesion in operations (LCOO)	Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same instance variable
Size	Lines of code (LOC)	Counts the lines of code
	Number of attributes (NOA)	Counts the number of attributes of each class or aspect
	Weighted operations per component (WOC)	Counts the number of methods and advices of each class or aspect and the number of its parameters

AO patterns and AspectJ idioms

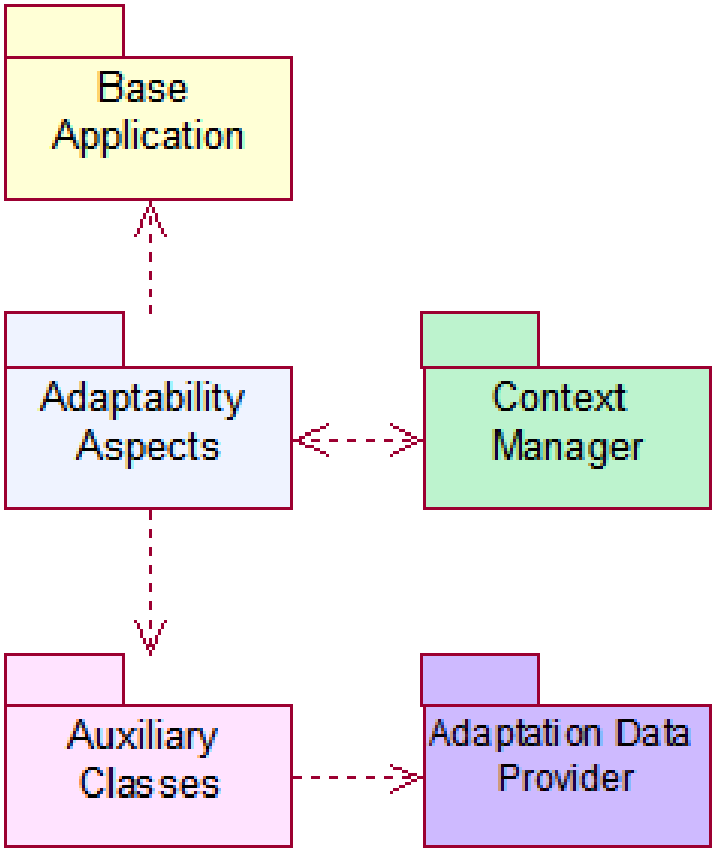
- Tag (marker) interface
  - como na interface **`Trans`** do aspecto de transações
- Glue aspects
  - advices invocando serviços auxiliares
- Template (abstract) pointcut
  - com aspectos abstratos

Template advice



From AspectJ Idioms for Aspect-Oriented Software Construction,
de Hanenberg, S.; Schmidmeier, A.; Unland, R.

The adaptability aspects pattern



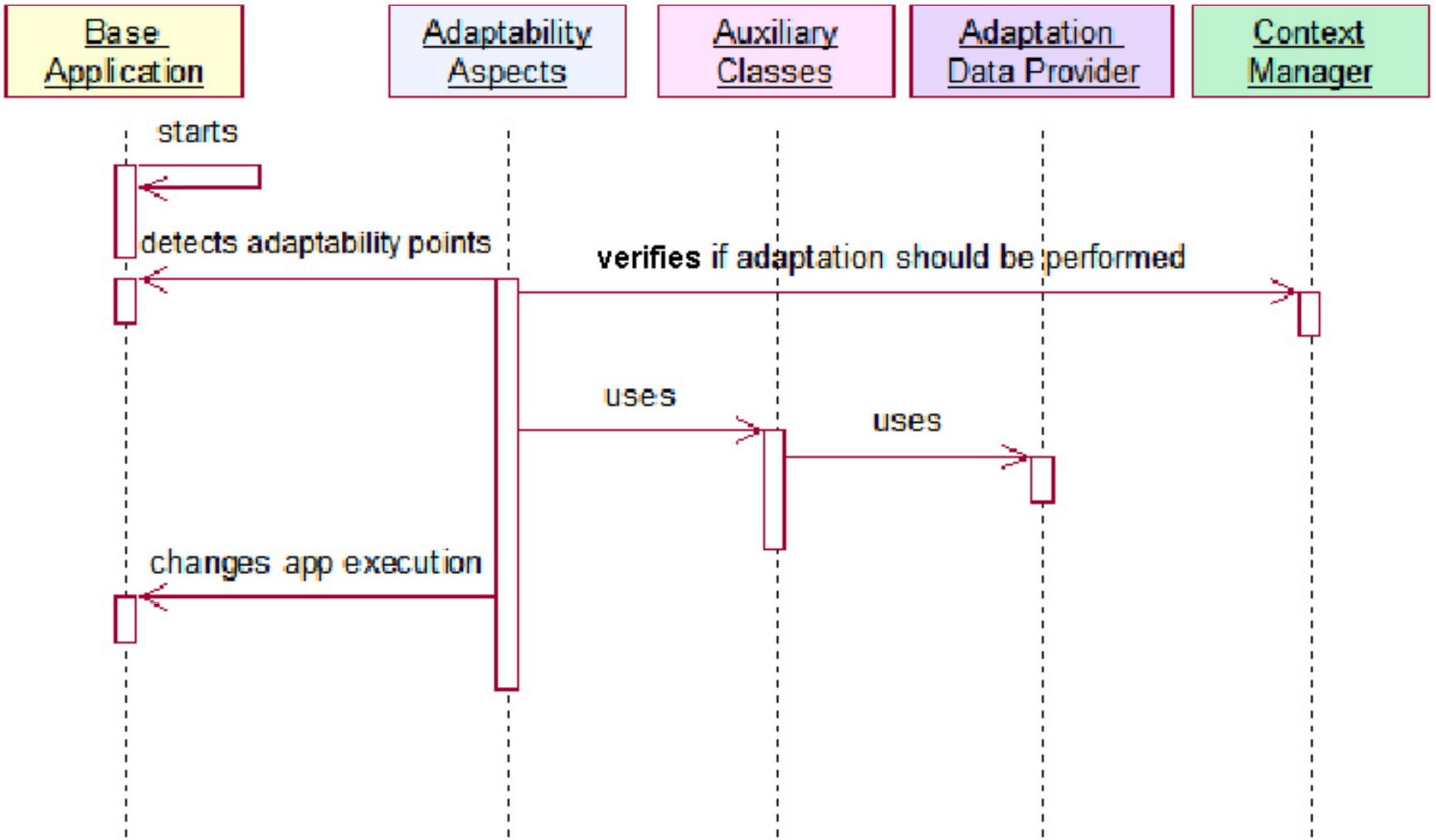
Improving modularity and reuse …

Plugging in/out concerns…

Organizing the development team …

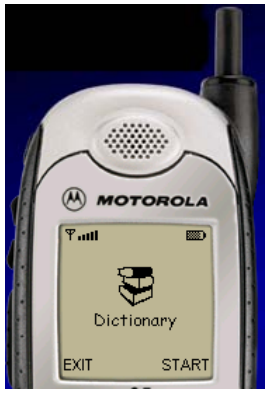
Using lightweight aspects …

Dynamics…


Base Application
Adaptability Aspects
Auxiliary Classes
Adaptation Data Provider
Context Manager
starts
detects adaptability points
verifies if adaptation should be performed
uses
uses
changes app execution

Making systems adaptive…

% versions	Initialization	Execution	Used Memory	JAR size	Source
AspSol x TangSol	25,0	30,9	23,0	35,7	2,6
AspSol x PatSol	10,4	12,5	14,1	16,0	-11,6



+ **Adaptability Concerns** → **AspSol**, **TangSol**, **PatSol**

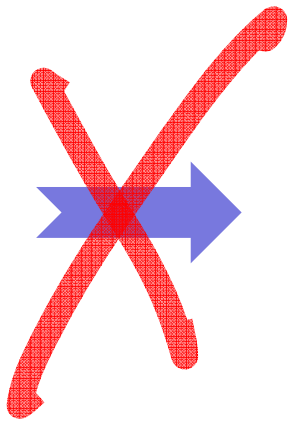
Refactorings and aspects

- Refactorings OO must be adapted to be aware of aspects
- New refactorings for turning OO code into AO code
- New refactorings for improving AO code

Aspect-aware OO refactorings

```
class C {
  void n(){…body…}
}

aspect A {
  after():
    call(* *m()){
    extra
  }
}
```



```
class C {
  void n(){…m()…}
  void m(){body}
}
aspect A {
  after():
    call(* *m()){
    extra
  }
}
```

Mas falta saber...

- Como introduzir aspectos a uma aplicação orientada a objetos?
- Como reestruturar aplicações orientadas a aspectos?
- Refactorings para aspectos
  - Como verificar se preservam comportamento?

Problem

- Aspect-oriented developers are identifying common transformations for aspect-oriented programs
- Refactorings are generally complex and global
- It is difficult to verify if a defined refactoring preserves behaviour

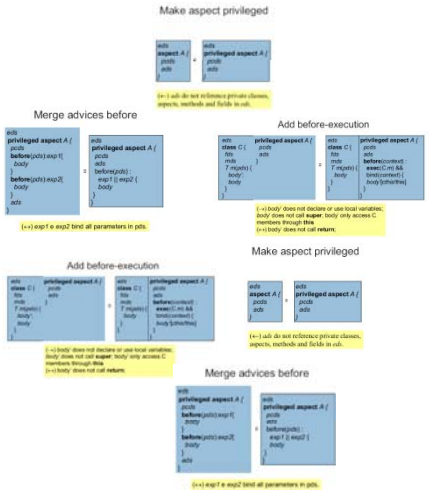
Our Approach

- Primitive laws of programming
  - simple, localized, intuitive and easier to understand
  - Guarded by pre-conditions

We use the laws to
- Derive complex and global refactorings
- Verify that an existing refactoring preserves behaviour

Deriving Refactorings

Laws



Composing

Refactoring

```
eds
aspect A {
  pcds
  ads
  before(pds) : exp {
    body
  }
  after(pds) : exp {
    body'
  }
}
```

=

```
eds
aspect A {
  Pcds
  ads
  pointcut p(pds) : exp
  before(pds) : p(αpds) {
    body
  }
  after(pds) : p(αpds){
    body'
  }
}
```

(→) There is no pointcut named *p* in *pcds*;
(←) *eds*, *ads* and *pcds* do not reference *p*

Aspect-Oriented Programming Laws

- Simple and localized
- Two transformations
- Bi-directional
- One direction may not increase code quality

Our laws define equivalences between two programs given that the preconditions are respected

Make Aspect Privileged

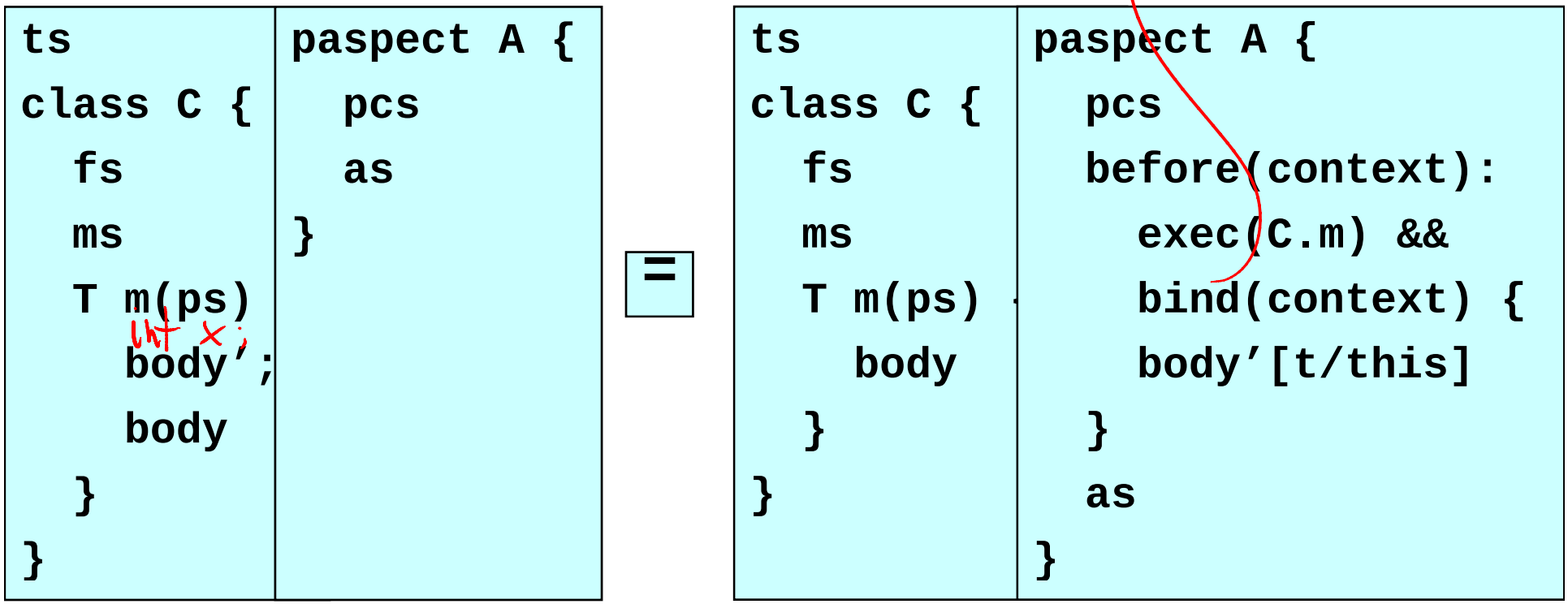
```
ts
aspect A {
  pcs
  as
}
```

=

```
ts
privileged aspect A {
  pcs
  as
}
```

(←) Advice bodies from *as* do not refer to private members declared in *ts*.

Add before-execution



(→) *body'* does not declare or use local variables;

body' does not call super;…

Example

```
public class Account {
  private double balance;...
  public void debit(double amount) {
    Access.check(new Permission(…));
    //debit logic
  }...
}
privileged aspect PermissionAspect {
}
```

Source: Laddad, 2003. Aspect Oriented Refactoring Series

Applying add before-execution

```
public class Account {
  private double balance;...
  public void debit(double amount) {
    //debit logic
  }...
}
privileged aspect PermissionAspect {
  before(Account cthis, double amount):
     execution(void Account.debit(double)) &&
     this(cthis) && args(amount) {
     Access.check(new Permission(..));
  }
}
```

Remove this Parameter

```
ts
paspect A {…
  before(T t, ps):
    this(t)&&… {
    body
  } …
}
```

=

```
ts
paspect A {…
  before(ps):
    this(T)&&… {
    body
  } …
}
```

(→) *t* is not referenced from *body*.

Merge Advices

```
ts
paspect A { ...
  before(ps): exp1 {
    body
  }
  before(ps): exp2 {
    body
  }
  ...
}
```

=

```
ts
paspect A { ...
  before(ps) :
    exp1 || exp2 {
    body
  }
  ...
}
```

(↔) The set of join points captured by *exp1* and *exp2* are disjoint. …

Move field to aspect

```
ts
class C {
  fs; T field
  ms
}
paspect A {
  ...
}
```

=

```
ts
class C {
  fs
  ms
}
paspect A {
  T C.field
  ...
}
```

(→) The field *field* of class *C* does not appear in *ts* and *ms*.

Adicionar before-call

```
eds
class C {
  fds
  mds
  T n(pds) {
    body;
    e.m(args)
  }
}
```

```
priv aspect A
{
  pcds
  ads
}
```

=

```
eds
class C {
  fds
  mds
  T n(pds) {
     e.m(αpds)
  }
}
```

```
priv aspect A {
  pcds
  ads
  before(context) :
    wc(C.n) &&
    call(C.m) &&
    bind(context) {
    body[cthis/this]
  }
}
```

(→) …

Summary of laws

Law	Name	Law	Name
1	Add empty aspect	16	Remove argument parameter
2	Make aspect privileged	17	Add catch for softened exception
3	Add before-execution	18	Soften exception
4	Add before-call	19	Remove exception from throws clause
5	Add after-execution	20	Remove exception handling
6	Add after-call	21	Move exception handling to aspect
7	Add after returning-execution	22	Move field to aspect
8	Add after returning-call	23	Move method to aspect
9	Add after throwing-execution	24	Move implements declaration to aspect
10	Add after throwing-call	25	Move extends declaration to aspect
11	Add around-execution	26	Extract named pointcut
12	Add around-call	27	Use named pointcut
13	Merge advices	28	Move field introduction up to interface
14	Remove this parameter	29	Move method introduction up to interface
15	Remove target parameter	30	Remove method implementation

Assumptions

- No packages
- `this` is mandatory
- No abstract aspects
- No concurrency
- No reflection
- ...

Applications

- We derived existing aspect-oriented refactorings from the laws
- We restructured two object-oriented applications modularizing crosscutting concerns with aspects

Refactorings: Extract Pointcut

```
ts
aspect A {
  …
  before(ps): exp {
    body
  }
  …
  after(ps): exp {
    body'
  }
  …
}
```

=

```
ts
aspect A {
  pointcut p(ps): exp
  …
  before(ps): p(αps) {
    body
  }
  …
  after(ps): p(αps) {
    body'
  }
  …
}
```

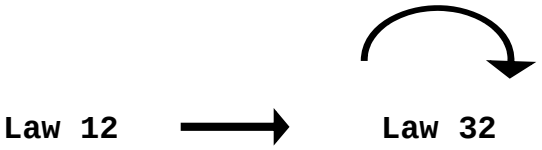
Extract Pointcut

- Preconditions

(→) **There is no pointcut named *p* in *pcs*;**

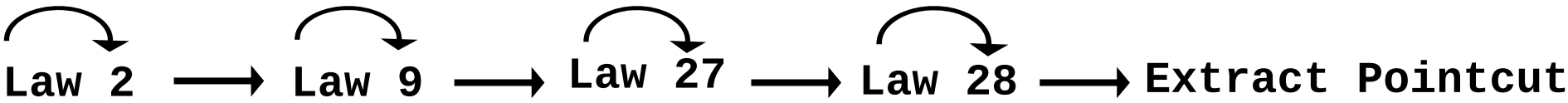
(←) **There is no reference to *p* in *ts* and *A*.**

- Summary
  - Law 12 – Extract named pointcut
  - Law 32 – Use named poitncut



Extract Method Calls

- Summary
  - Law 2 – Add before-execution
  - Law 9 – Merge advices
  - Law 27 – Remove `this` parameter
  - Law 28 – Remove argument parameter



Other Refactorings

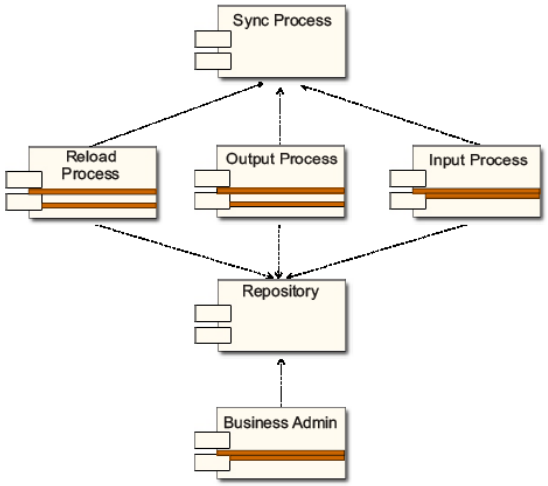
- Extract Interface Implementation
- Extract Exception Handling
- Extract Concurrency Control
- Evolving Product Lines
- …

Refactoring systems

- Two commercial applications
- First
  - Mobile Server
  - Concurrency control
- Second
  - HealthWatcher
  - Distribution
  - Problem with Extract Exception Handling

Refactoring to AspectJ

Before: OO



General OO Refactorings

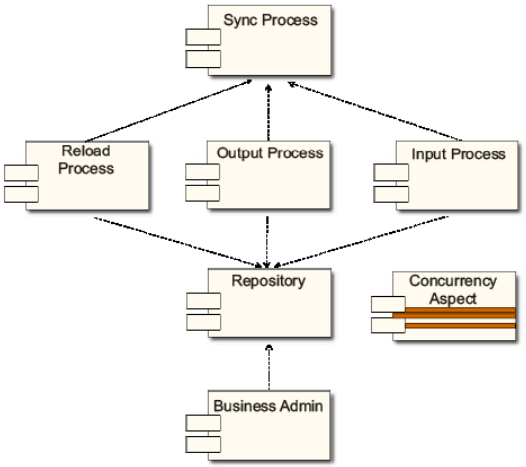
+

AO Derived Refactorings

+

Laws

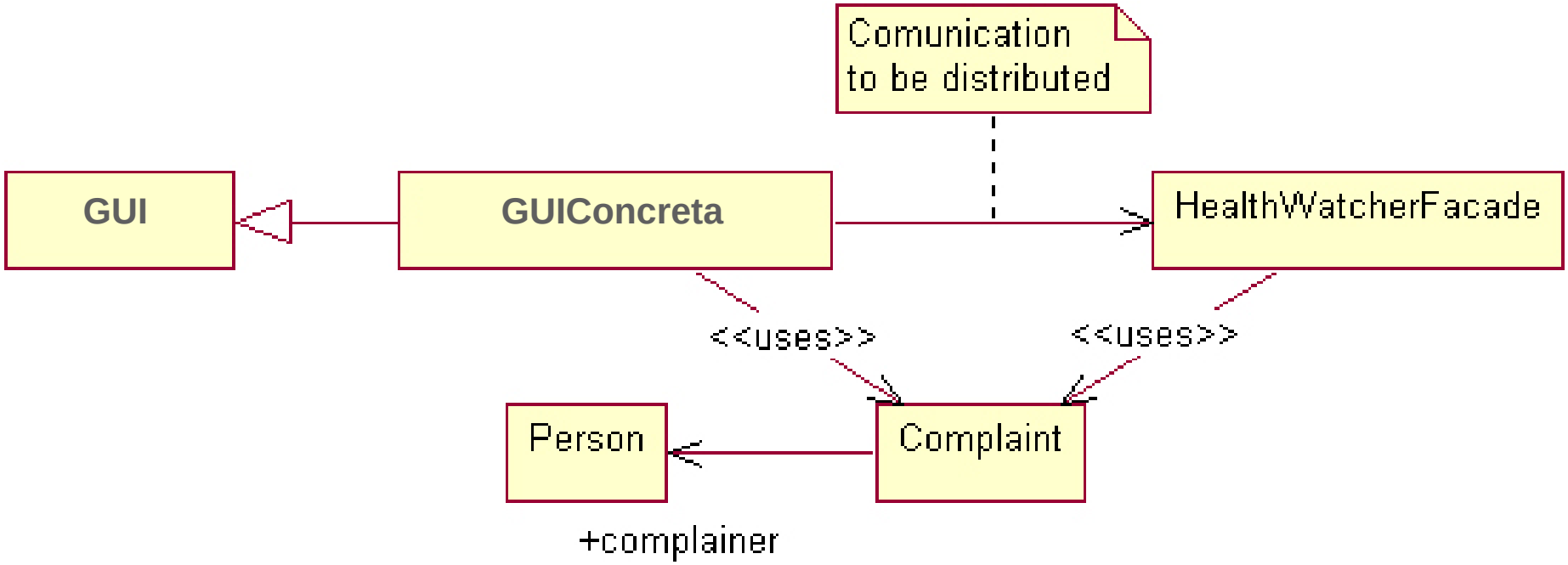
After: AO



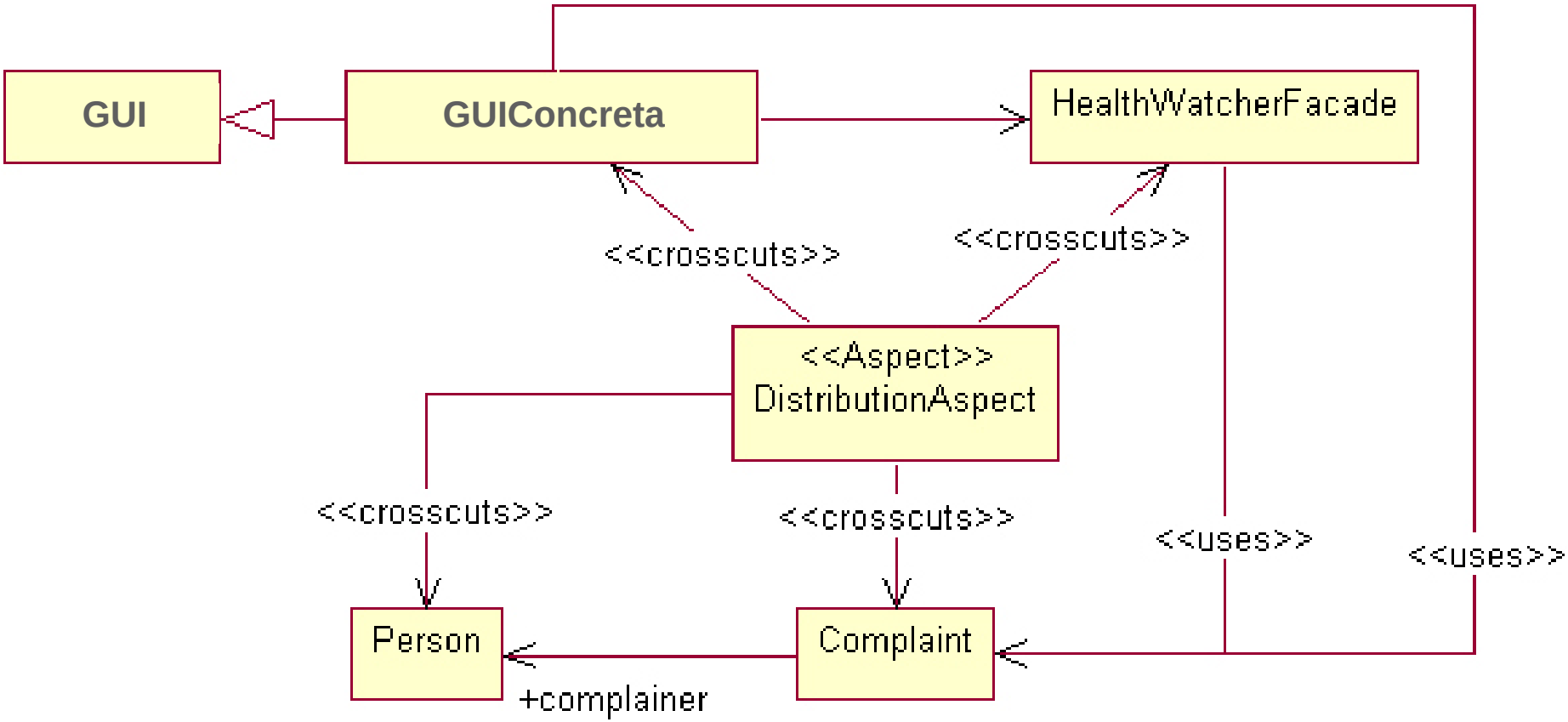
Mobile server, resultados

- Código mais limpo e legível
- Modularidade e manutenabilidade
- Redução no número de linhas de código
  - Mais visível nos casos em que advices capturam muitos join points
- 2 aspectos (concorrência e controle de exceções) afetando 4 classes

Local health watcher



Health watcher with AOP



Health watcher, resultados

- Aspectos de distribuição
- Mesmos ganhos do Mobile Server, porém mais visíveis
- 18 servlers afetados no cliente, mais 14 classes afetadas no servidor (fachada e classes básicas)
- Aspectos podem ser generalizados para outras aplicações de distribuição similares

Aspect-oriented patterns and refactoring

Paulo Borba
Informatics Center
Federal University of Pernambuco