Aspect-Oriented Programming is Quantification and Obliviousness

Robert E. Filman, Daniel P. Friedman
Workshop on Advanced Separation of Concerns,
OOPSLA 2000

AOP= Quantification + Obliviousness

- AOP can be understood as the desire:
  - to make quantified statements about the behavior of programs
  - to have these quantifications hold over programs written by oblivious programmers

- Quantified program statements: statements that have effect on many loci in the underlying code

- AOP should work with oblivious programmers: ones who don't have to expend any additional effort to make the AOP mechanism work

Quantification

- AOP is the desire to make programming statements of the form:
  - In programs P, whenever condition C arises, perform action A

- Quantification:
  - Over the static structure of the system
  - Over its dynamic behavior

- The static structure is the program as text:
  - Black-box AOP: quantify over the public interface of components
  - Clear-box AOP: quantification over the parsed structure of components. Example: AspectJ

- Dynamic quantification: tying the aspect behavior to something that happens at run-time.
  - Example: Call to a function X within the temporal scope of a call of Y

Obliviousness

- “Just program like you always do, and we’ll be able to add the aspects later.”

- Obliviousness is desirable because it allows greater separation of concerns in the system creation process
  - Concerns can be separated not only in the structure of the system, but also in the heads of the creators.

AOP: Avaliação Crítica

Mais de uma década de AOP

- Kiczales, ECOOP 1997:
  - "*We present an analysis of why certain design decisions have been so difficult to clearly capture in actual code. We show that the reason ... is that they cross-cut the system's basic functionality*."

- AOP teve um grande sucesso:
  - Acadêmico (~6800 citações no Google Scholar, 23/03/2012)
  - Na indústria (SpringAOP, JBossAOP, RubyAOP etc)

- Combinação de conceitos e princípios:
  - Quantificação
  - Obliviousness
  - Invocação implícita
  - Decomposição horizontal

Muitas críticas também ...

- Steimann, OOSPLA 2006:
  - "*Much of aspect-oriented programming's success seems to be based on the conception that it improves both modularity and the structure of code, while in fact, it works against the primary purposes of the two, namely independent development and understandability of programs*."

- Mais especificamente, dois problemas principais:
  - Raciocínio modular
  - Fragilidade dos conjuntos de junção

Problema #1: Raciocínio Modular

- Aspectos podem modificar qualquer ponto de junção
- Exemplo: chamada de C.m2() vai imprimir 55?

```
class C {
   static int foo;
   static void m1() { foo= 55;}
   static void m2() { m1(); println(foo);  }
}
```

Problema #1: Raciocínio Modular

- Aspectos podem modificar qualquer ponto de junção
- Exemplo: chamada de C.m2() vai imprimir 55?

```
class C {
   static int foo;
   static void m1() { foo = 55;}
   static void m2() { m1(); println(foo);  }
}
```

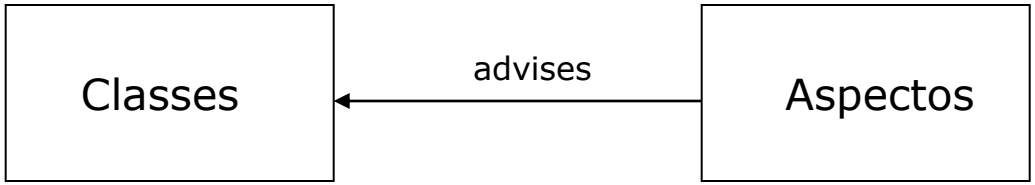
- E se existir o seguinte aspecto?

```
aspect A {
   after(C c): call(void C.m1()) && withincode(void C.m2())
                  && target(c)
   { c.foo = 66; }
}
```

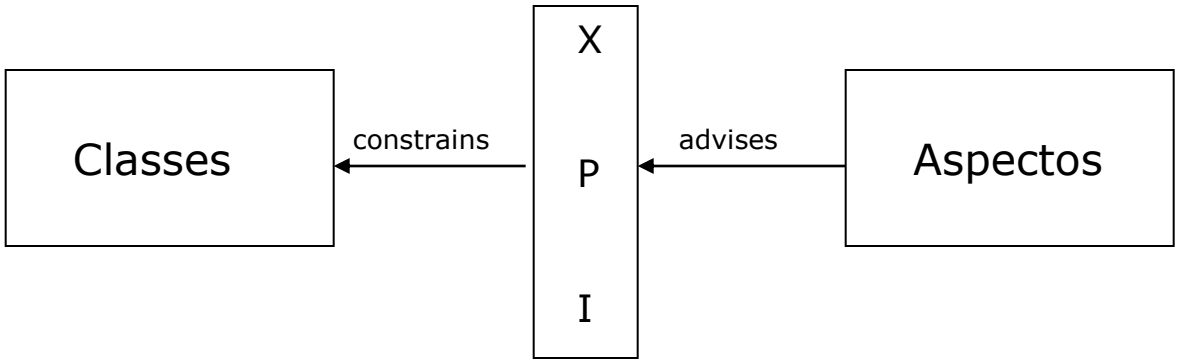
Adaptado de
Wand, ICFP 2003

XPIs

- Crosscutting interfaces (XPIs)
  - Propostas por Sullivan (FSE 2005, IEEE SW 2006)
  - Regras de projeto que classes e aspectos devem seguir
- AOP sem XPIs:



- AOP + XPIs:



XPIs

- Mérito: restaurar conceitos de modularidade em AOP
  - Desenvolvimento independente: aspectos e classes podem evoluir em paralelo
  - Raciocínio modular: aspectos não podem interferir em qualquer parte do código das classes

- Problema:
  - Não existe consenso sobre soluções para definição de XPIs
  - Tentativas existentes: regras informais, AspectJ

Problema #2: Fragilidade de PCs

- Em orientação por objetos, o acoplamento entre o código cliente e o código que implementa uma classe é controlado por meio de interfaces.
- O cliente adquire o direito de usar os métodos de uma interface e a classe assume o compromisso de implementar tais métodos.
- Caso a assinatura dos métodos de uma interface seja alterada, esta alteração impacta tanto os clientes da interface, como as classes que implementam a mesma.
- Além disso, este impacto é verificado pelo compilador:
  - Avisa os clientes sobre a necessidade de utilizar os métodos com suas novas assinaturas
  - Avisa as classes implementadoras sobre a necessidade de prover uma implementação para os novos métodos.

Problema #2: Fragilidade de PCs

- No caso de orientação por aspectos, é interessante observar que aspectos podem ser vistos como clientes do programa base

- Isto é, embora aspectos complementem o programa base com código para implementação modular de requisitos transversais, este código não é explicitamente chamado pelo programa base.

- Em vez disso, para seu correto funcionamento, aspectos pressupõem que o programa base disponibiliza determinados pontos de junção, os quais funcionam como ganchos para chamada implícita de *advices*

Problema #2: Fragilidade de PCs

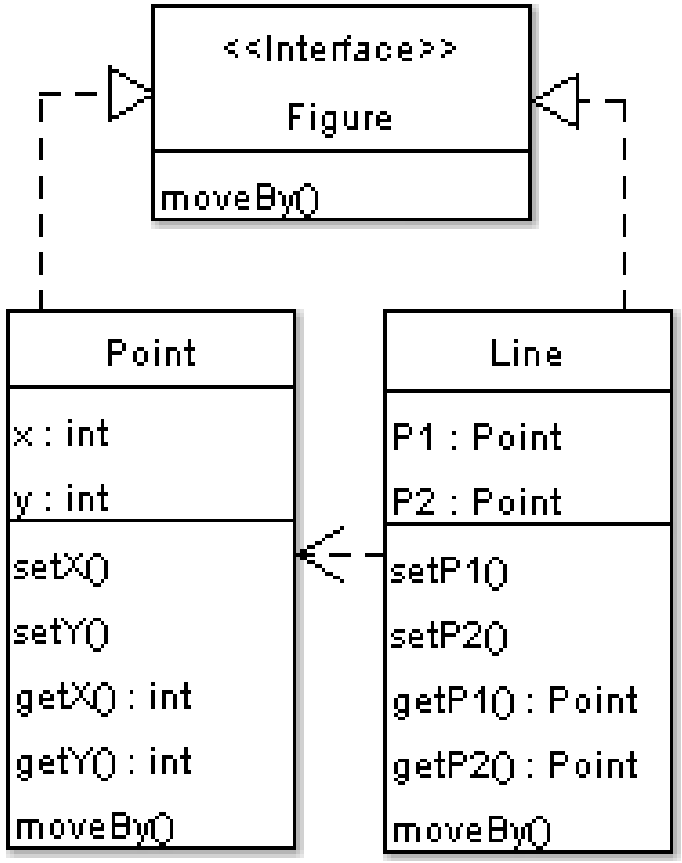
- Esta inversão de responsabilidades constitui uma diferença fundamental entre OO e OA
- Em OO, uma classe cliente explicitamente invoca métodos implementados por uma classe servidora.
- Em OA, cabe a um aspecto -- que, do ponto de vista desta explicação, constitui um cliente -- implementar métodos que serão implicitamente chamados pelo código base.
- No entanto, esta dependência entre aspectos e programa base não é documentada nem verificada por um contrato (ou interface).
- Com isso, estabelecem-se as condições para ocorrência do problema dos conjuntos de junção frágeis.

Problema #2: Fragilidade de PCs

- Mais especificamente, o problema dos conjuntos de junção frágeis em sistemas orientados por aspectos se manifesta quando, em razão de uma modificação no programa base, conjuntos de junção silenciosamente passam a capturar pontos de junção indesejados ou deixam de capturar determinados pontos de junção essenciais ao correto funcionamento de um sistema.

- Assim, de acordo com esta definição, um conjunto de junção é caracterizado como frágil por dois motivos:
  - por não ser robusto a alterações no programa base;
  - por ter sua semântica silenciosamente alterada em função de uma manutenção no programa base.

Exemplo




```
aspect UpdateSignaling {

    pointcut change() :
        execution(void Figure+.set*(..))
      || execution(void Figure+.moveBy(..));

   after() returning: change() {
        Display.update();
   }
}
```

Fragile pointcut problem

- Pointcuts are defined by names and regular expressions

- Changes in the base system may cause:
  - Unintended capture of join points
    - e.g. addition of Point.setDate()
  - Accidental join point miss
    - e.g. renaming Point.setX() to Point.changeX()

Refactoring Crosscutting Concerns using Aspects: Is it Always Worthwhile?

II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP),

2008

In this talk

- Show interesting and not so interesting uses of aspects
- Judgment based on qualitative, plausible arguments
- Convince by showing code before and after aspectization
- Focus:
  - Existing systems
  - Real and non-trivial
  - Medium to large size
  - Available for download
  - Previously used in AOP papers

Considered systems

- Jaccounting (https://jaccounting.dev.java.net)
  - Web-based business accounting system
  - Size: 11 KLOC
  - Concern: transactions
- Jspider (http://j-spider.sourceforge.net)
  - Web robot engine that supports downloading and validation of web pages
  - Size: 14 KLOC
  - Concern: logging
- Used to evaluate the AOP-Migrator tool:
  - David Binkley et al. Tool-Supported Refactoring of Existing OO Code into Aspects. IEEE Trans. Software Eng. 2006.

Considered systems

- Prevayler (http://www.prevayler.org)
  - Object persistence library for Java (2.5 KLOC)
  - Concerns: snapshots, clocks, replication, threads etc
  - Non-trivial software product line using aspects
  - Godil and Jacobsen: Horizontal decomposition of Prevayler. CASCON 2005.

JAccounting

- Transaction handling in the original implementation:

```
sess= openSession();              // opens a database session
tx= sess.beginTransaction();      // starts a transaction
try {
  ...                             // performs database operations
  tx.commit();                    // commits
}
catch (...) {                     // handles database exceptions
  tx.rollback();                  // rollback
}
finally {
  sess.close();
}
```

Jaccounting: TransactionMngt Aspect

- First aspectization attempt:

```
after(): call(Session SessionFactory.openSession() {
  tx= sess.beginTransaction();
}

before(): handler(Exception) && withincode(...) {
  tx.rollback();
}

before(): ????? {
  tx.commit();
}
```

- Problem: how to define the pointcut specifying join points where commit must be called?

OO Transformations

- AspectJ only supports the introduction of crosscutting behavior in well-defined join points
- In legacy systems we should not expect that crosscutting code is located precisely before, around or after join points
- Programmers usually need to transform the base program
- Goal: associate crosscutting statements with parts of the program that can be captured by AspectJ pointcuts.
- OO Transformations:
  - Statement reordering
  - Method extraction

Jaccounting: OO Transformations

```
try {
  ...                          // performs database operations
  tx.commit();                 // commits
}
catch (...) {                  // handles database exceptions
  tx.rollback();               // rollback
}
finally {
  sess.close();
}
```

```
try {
  ...                          // performs database operations
}
catch (...) {                  // handles database exceptions
  tx.rollback();               // rollback
}
finally {
  tx.commit();                 // commits before sess.close()
  sess.close();
}
```

Jaccounting: OO Transformations

```
try {
  ...                          // performs database operations
}
catch (...) {                  // handles database exceptions
  tx.rollback();               // rollback
}
finally {
  tx.commit();                 // commits
  sess.close();
}
```

```
try {
  ...                          // performs database operations
}
catch (...) {                  // handles database exceptions
  tx.rollback();               // rollback
  tx= null;
}
finally {
  if (tx != null) tx.commit();
  sess.close();
}
```

Jaccounting: TransactionMngt Aspect

```
after(): call(Session SessionFactory.openSession() {
  tx= sess.beginTransaction();
}

before(): handler(Exception) && withincode(...) {
  tx.rollback();                // rollback
  tx= null;
}

before(): call(Connection Session.close())  {
  if (tx != null) tx.commit();
}
```

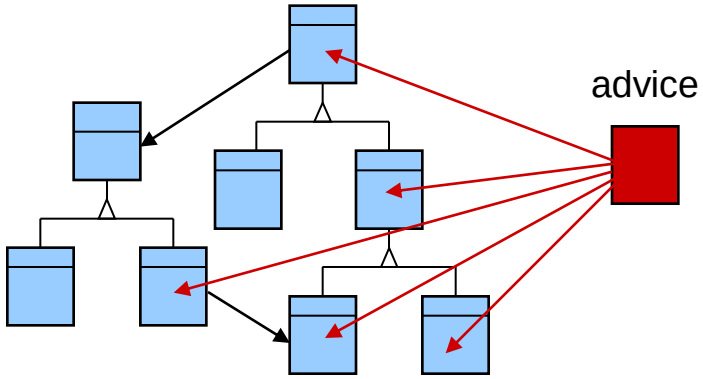
JAccounting: Evaluation

- OO Transformations have been fundamental:
  - Transactions are homogeneous concerns
  - But they not appear on aspectizable points
  - 30 statement reordering transformations

- Automatating OO transformations is a challeging task
  - Main reason: ad hoc transformations

JAccounting: Evaluation

- Aspects making reasonable use of quantification:
  - Transaction code is confined in a single aspect
  - Advices affect many points of the base program
  - 4 advices and 45 join point shadows



Jaccounting: Evaluation

- It is a good example of using aspects
- Duplicated and tangled code moved to few advices
- More simple to understand, evolve and change
- Equivalent form of generalization is provided by traditional modularization techniques:
  - Procedures and functions abstract out computations that are needed in many parts of a system
  - Inheritance allows developers to implement in superclasses methods required by subclasses
- Small reduction in LOC: 11676 => 11477 (-1.7%)

JSpider

- Logging calls in the original JSpider implementation:

```
log.info("Loading " + pluginCount + " plugins.");
...
log.info("Loading plugin configuration '" + pluginInstance + "'...");
...
log.info("Plugin class "+ getString(PLUGIN_CLASS, "") +" not found");
.....
log.info("Plugin uses local event filtering");
.....
log.info("Plugin not configured for local event filtering");
.....
log.info("Plugin Name    : " + plugin.getName());
.....
log.info("SQL Exception during JDBC Connect” + e);
```

- Always call the same method, but using different strings as arguments (logging messages)

JSpider: OO Transformations

- Several method extractions required, in order to enable join points in the base code

```
if (interval < INTERVAL_MIN) {
    log.warn("Throttle interval < " + INTERVAL_MIN +        // aspect
      " ms is dangereous - set to minimum allowed of " +    // aspect
      INTERVAL_MIN + " ms");                                // aspect
    interval = INTERVAL_MIN;
  } ...
}
```

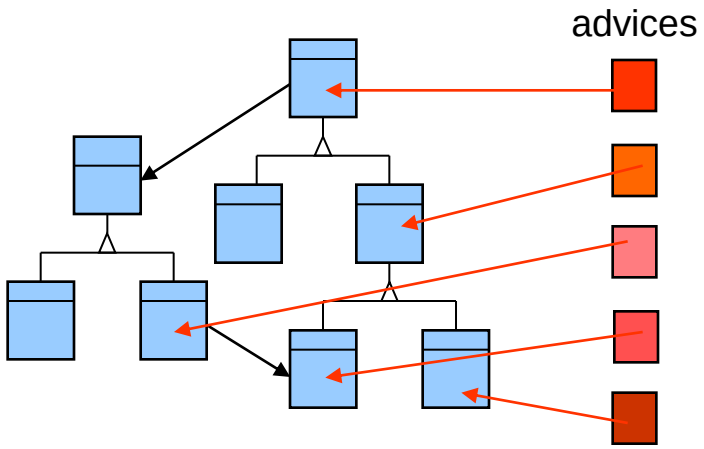
```
if (interval < INTERVAL_MIN) {
    log.warn("Throttle interval < " + INTERVAL_MIN +        // aspect
    " ms is dangereous - set to minimum allowed of " +      // aspect
    INTERVAL_MIN + " ms");                                  // aspect
    interval = setToMininum();
  } ...
}
private int setToMininum() {                          // extracted method
  return INTERVAL_MIN;
}
```

JSpider: LogAspect

- Most advices modularizes a single logging call
- Defined pointcuts are very fragile

```
pointcut p_27(DBUtil _this, SQLException e ):
  this(_this)
  && execution(void DBUtil.sqlException(SQLException)) && args(e);

before(DBUtil _this, SQLException e): p_27(_this, e) {
  _this.log.error("SQL Exception during JDBC Connect", e);
}
```

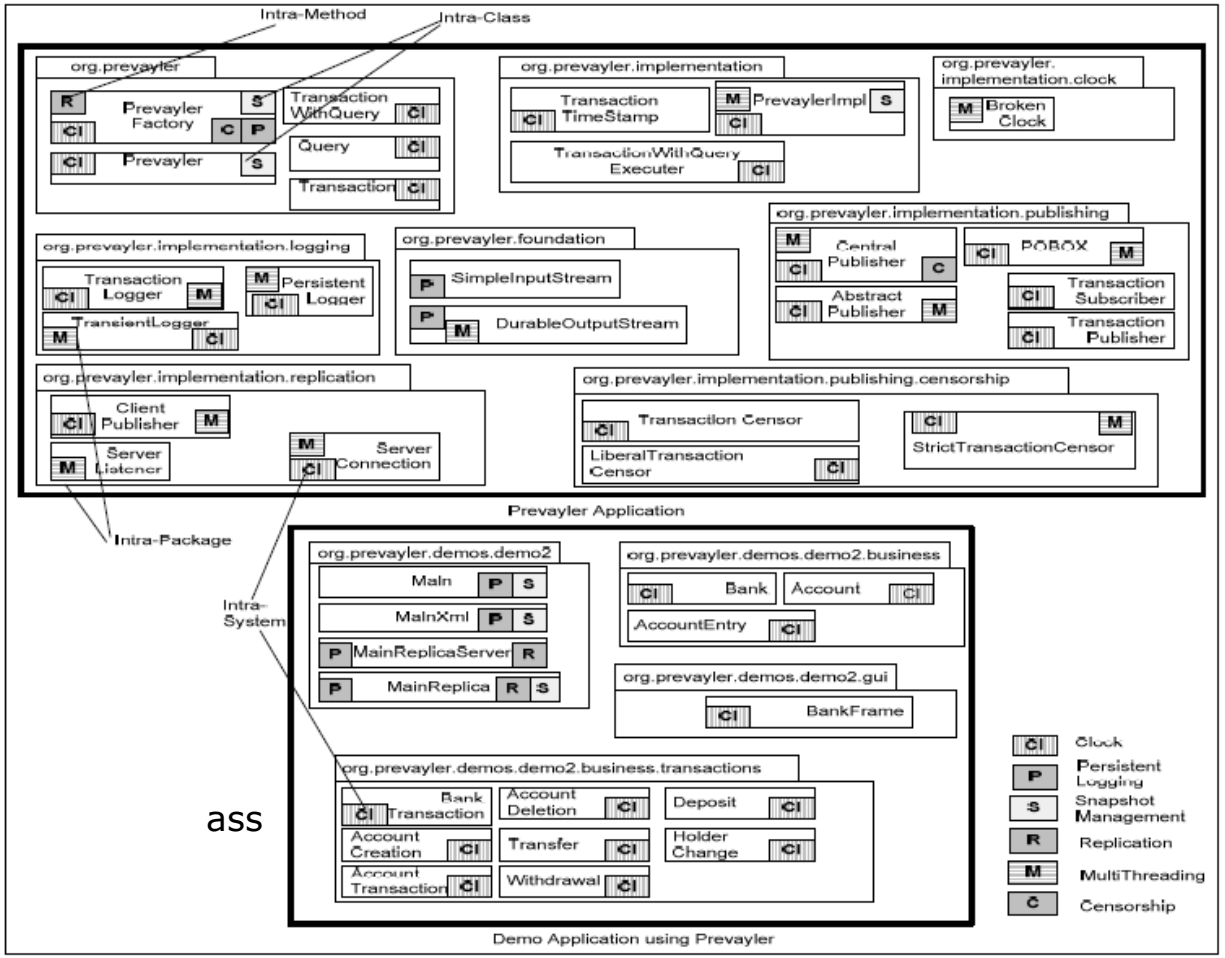


JSpider: Evaluation

- Several OO transformations (31 method extractions)
- Mostly, one advice per logging call
  - 234 advices, 246 logging calls
- Increase in size: 9 KLOC => 11 KLOC (+19.92%)
- In summary, it is not a good example of using aspects
  - +2 KLOC to modularize a single concern
- Futhermore:
  - This conclusion is valid for any alternative design preserving the same level of detail in log messages
  - Logging has similar behavior in other systems (e.g. Tomcat)

Prevayler

- SPL including may internal DB features (36 configurations)



Godil, Jacobsen, CASCON 2005

Prevayler

- Observer pattern employed in many parts of the system
- Problem: *executionTime* parameter is optional (feature Clock):

```
void receive(Transaction transaction, Date executionTime) {
   ...
}
...
subscriber.receive(entry.transaction(), getTimeStamp());
```

- Optional argument is removed from the SPL core:

```
void receive(Transaction transaction) {
   ...
}
...
subscriber.receive(entry.transaction());
```

ClockPrevayler aspect

```
pointcut receiveCall(ClientPublisher p): this(p) &&
  call(public void TransactionSubscriber.receive(Transaction));

pointcut inReceiveHelp():
  withincode(* ClientPublisher.helperReceiveTransactionFromServer(..));

pointcut timeStampCall(): call(private Date getTimeStamp());

after() returning (Date d ): timeStampCall() &&  inReceiveHelp() {
  timestamp= d;
}

void around(ClientPublisher p): receiveCall(p) &&  inReceiveHelp()  {
  ... p._subscriber.receive(p._myTransaction, timestamp); ...
}
```

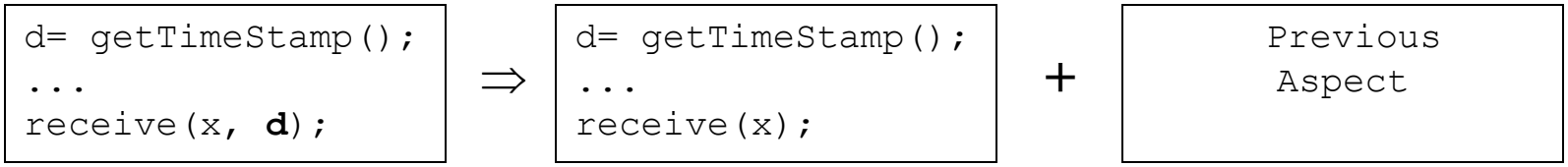
Saves the second argument

calls receive using two arguments (implementation is introduced by another aspect)

captures receive calls from the core (with a single argument)

Prevayler: Evaluation

- Complex system (database engine)
- Fine-grained features: required in very specific points of the code, including method signatures
- Strategy used to extend method signatures:
  - Use the minimal signature in the base code
  - Intercept all calls to save optional arguments



- Complexity increases with more optional arguments
- Increase in size (LOC): 3320 LOC => 4119 LOC (+20%)

Prevayler: Evaluation

- Prevayler is not a good example of using aspects
  - Regarding its current design and implementation:

- But we can cannot conclude that it is impossible to modularize Prevayler features using aspects:
  - Maybe there are alternative designs more friendly to aspects
  - Radical change in the current design of the system

Summary

- Crosscutting concerns associated to single method calls

- Jaccounting: no-args calls
  - Example: `tx.rollback() e tx.commit();`
  - AspectJ provides a interesting solution
  - OO transformations are crucial

- Jspider: calls using string as arguments
  - Example: `log.info("Loading plugins.");`
  - Example: `log.info("Plugin uses local event ...");`
  - AspectJ does not provide a very interesting solution

- Prevayler: calls using a variable number of arguments
  - Example: `receive(transaction(), getTimeStamp());`
  - Example: `receive(transaction());`
  - AspectJ solution is very complex and hard to follow

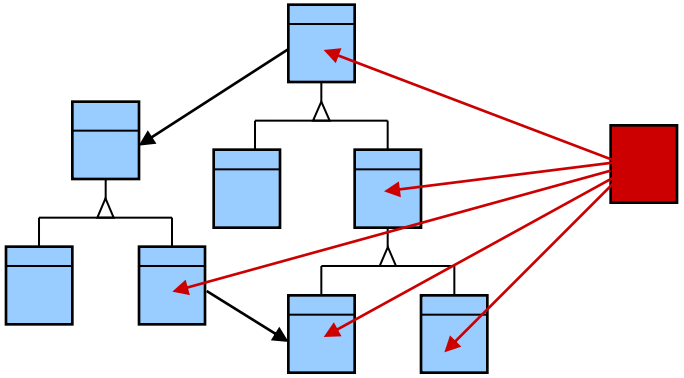
Quantification Degree

Motivation

- AOP= quantification + obliviousness [Filman and Friedman]
- Obliviousness is under revision:
  - Design rules
  - XPIs
  - Open modules
  - Ownership type systems
  - etc

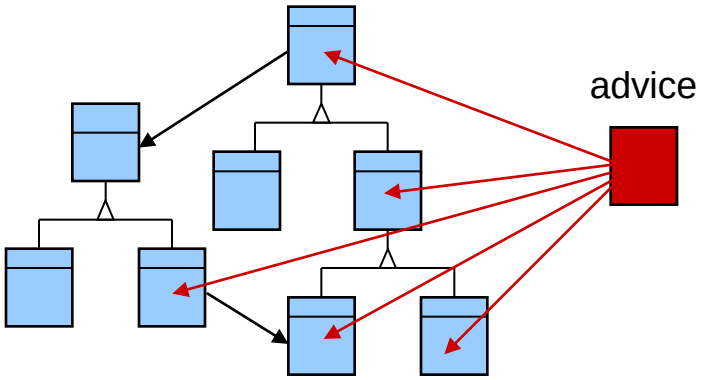
And Quantification?

- Definition: abstractions that have “effect on many *loci* in the base code”



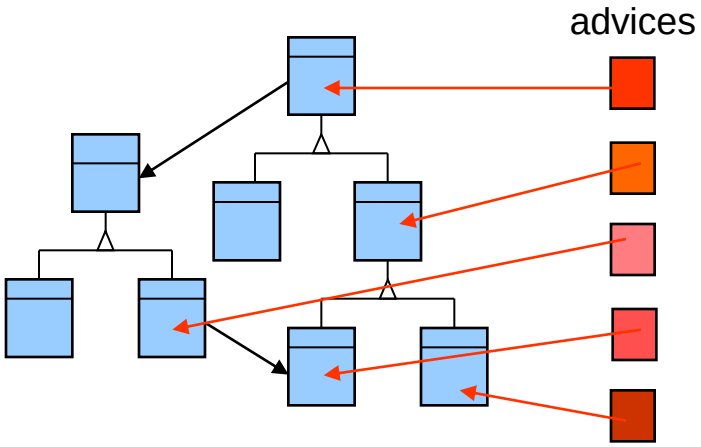
- We are trying to answer the following question:
  - On average, advices affect how many *loci* in the base code?

Quantification Degree



Single advice modularizes the implementation of a given concern

Quantification degree: five

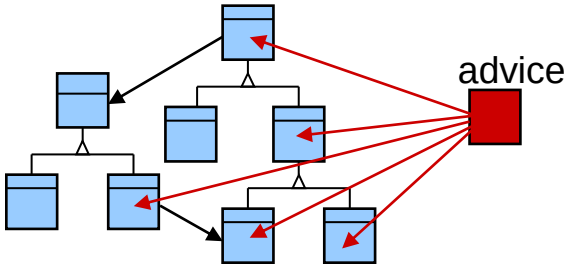


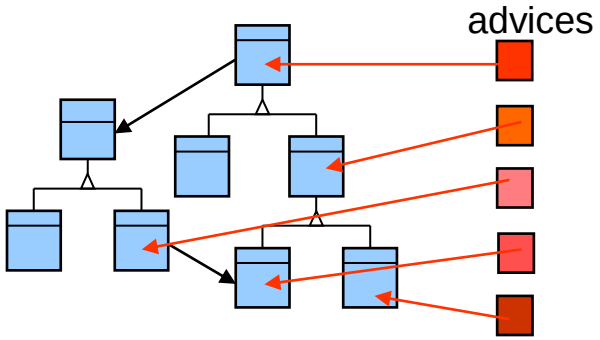
Multiple advices are needed to modularize the implementation of a given concern (due to variations in the crosccutting code)

Each advice affetcs a single *locus* of the base program

Quantification degree: one (minimal)

Quantification Degree

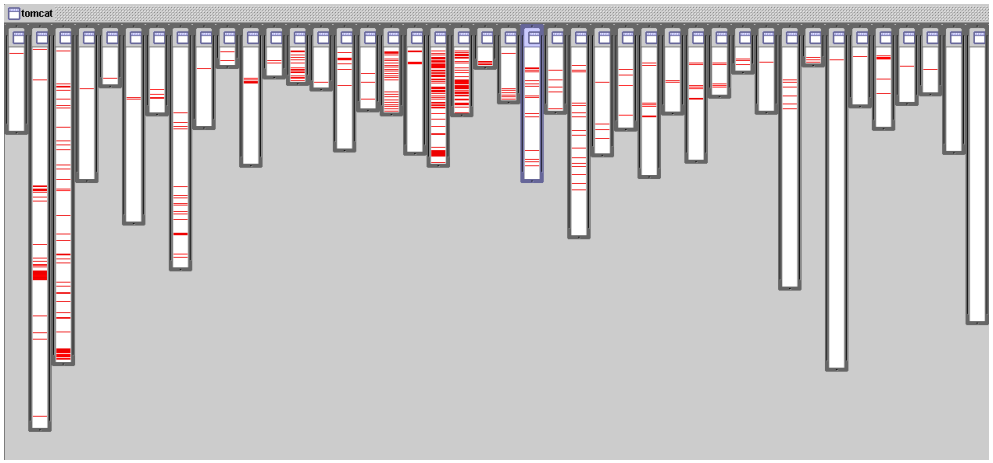




- Both designs achieve:
  - Implementation in a single program unit (aspect)
- But, high degrees of quantification favor:
  - Comprehensibility
  - Changeability
- What is the most common situation in medium to large size AspectJ systems?

Preliminary Case Study

- JHotdraw: 40 KLOC, concern: undo
- JAccounting: 11 KLOC, concern: transactions
- Tomcat: 45 KLOC, concern: logging



Results

	JHotDraw	Tomcat	JAccounting
Join point shadows	86	258	45
Advices	86	236	4
Quantification degree	1	1.09	11.25
LOC (OO version)	40022	45107	11676
LOC (AO version)	40805(+2%)	45642(+1.2%)	11477(-1.7%)

- Quantification degree= join point shadows / advices

Discussion

- High quantification degree ⇔ homogeneous concern
  - e.g. transactions
- Low quantification degree ⇔ heterogeneous concern
  - e.g. logging and undo
- High degrees of quantifification are not so common
- Other systems:
  - JSpider (Binkley et al)
  - JavaPetStore (Binkley et al)
  - Prevayler (Jacobsen et al)
  - Oracle Berkley DB (Apel et al)

Discussion

- Fillman and Friedman´s quantification definition:
  - “effect on many loci in the base code”
- And when the effect is on a single locus?
- Is it worth to use AOP when QD ≈ 1?
- Pros:
  - Implementation in a single program unit
  - Pluggability
- Cons:
  - Coupling (aspects + base code)
  - Pointcut fragility