Towards Typesafe Join Points for Modular Reasoning in Aspect-Oriented Programs

Eric Bodden
joint work with Milton Inostroza, Éric Tanter
Aspect-oriented programming successfully modularizes crosscutting concerns.
Aspect-oriented programming successfully modularizes crosscutting concerns.

Aspect-oriented programming fails to preserve modular reasoning.
AOP is ...
AOP is ...

Programming support for implementing a separation of concerns
Dependencies in traditional AOP

Main-stream software developer

Class C
join point (shadow)

Aspect A
pointcut
advice

Global reasoning

AOP Expert
Join Point Interfaces (JPI)

Main-stream software developer

Class C

- join point (shadow)
- exhibit
- pointcut

Aspect A

- advice

jpi

Separate evolution
"just like a method signature"

- Fixed contract
- Separate evolution
- Modular reasoning
- No weave time errors
Birthday discount in an online shopping system:

5% off for purchases on your birthday
Aspect code

jpi void CheckingOut(float price, Customer c)
Aspect code

```java
@override
void CheckingOut(float price, Customer c)

aspect Discount{
  void around CheckingOut(float price,
                          Customer cus){
    int factor = cus.hasBirthday() ? 0.95 : 1;
    proceed(price*factor, cus);
  }
}

No reference to "base code"!
No reference to Discount aspect!
class ShoppingSession{
  ...
  void checkOut(Item item, float price, int amount, Customer cus){
    sc.add(item, amount);
    inv.add(item, amount, cus);
  }
}

exhibits void CheckingOut(float price, Customer cus):
  call(* checkOut(..))
  && args(*,price,*),cus);
JPIs give you...

- complete de-coupling of base and aspects
- therefore code can evolve independently
- no weave-time errors:
  language-semantics of Java preserved
- increases potential to re-use aspects
Sometimes unable to define a pointcut at all...
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    System.out.println("seed was:" + seed);
    int result = monteCarlo(seed);
    return result;
}
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    System.out.println("seed was:" + seed);
    int result = monteCarlo(seed);
    return result;
}

void around(long seed): monteCarloCall(seed) {
    proceed(0);
}
```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    System.out.println("seed was:" + seed);
    int result = monteCarlo(seed);
    return result;
}

void around(long seed): monteCarloCall(seed) {
    proceed(0);
}

pointcut monteCarloCall(long seed):
```
int monteCarloAlg() {
  long seed = System.currentTimeMillis();
  System.out.println("seed was:"+seed);
  int result = monteCarlo(seed);
  return result;
}

void around(long seed): monteCarloCall(seed) {
  proceed(0);
}

pointcut monteCarloCall(long seed):
  withincode(* monteCarloAlg())
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    System.out.println("seed was:" + seed);
    int result = monteCarlo(seed);
    return result;
}

void around(long seed): monteCarloCall(seed) {
    proceed(0);
}

pointcut monteCarloCall(long seed):
    withincode(* monteCarloAlg())
    && call(* monteCarlo(long)) && args(seed)
```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    System.out.println("seed was:" + seed);
    int result = monteCarlo(seed);
    return result;
}

void around(long seed): monteCarloCall(seed) {
    proceed(0);
}

pointcut monteCarloCall(long seed):
    withincode(* monteCarloAlg())
    && call(* monteCarlo(long)) && args(seed)
    .. call(* println(..)) && args(???)
```
Solution: Block Joinpoints!
Non-Solution: Block Joinpoints!

```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    exhibit SomeAspect.JP(seed) {
        System.out.println("seed was: "+seed);
        result = monteCarlo(seed);
    }
    return result; }

joinpointtype JP{ long theSeed; }
void around(JP j) {
}
```

```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    exhibit SomeAspect.JP(seed) {
        System.out.println("seed was:" + seed);
        result = monteCarlo(seed);
    }
    return result;
}

joinpointtype JP{ long theSeed; }
void around(JP j) {
    j.theSeed = 0;
    proceed(j);
}
```

```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    exhibit SomeAspect.JP(seed) {
        System.out.println("seed was:" + seed);
        result = monteCarlo(seed);
    }
    return result;
}

joinpointtype JP{
    long theSeed;
}
void around(JP j) {
    j.theSeed = 0;
    proceed(j);
}
```

```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    exhibit SomeAspect.JP(seed) {
        System.out.println("seed was:");
        result = monteCarlo(seed);
    }
    return result;
}
```

```java
joinpointtype JP{ long seed; }
void around(JP j) {
    j.seed = 0;
    proceed(j);
}
```

```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    exhibit SomeAspect.JP(seed) {
        System.out.println("seed was:" + seed);
        result = monteCarlo(seed);
    }
    return result;
}
```

```java
joinpointtype JP{
    long seed;
}
void around(JP j) {
    new Thread() {
        public void run() {
            proceed(j);
        }
    }.start();
}
```

```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    exhibit SomeAspect.JP(seed) {
        System.out.println("seed was:" + seed);
        result = monteCarlo(seed);
    }
    return result; }
```

```java
joinpointtype JP{
    long seed;
}
void around(JP j) {
    j.seed = 0;
    proceed(j);
}
```

```java
new Thread() {
    public void run() {
        proceed(j);
    }
}.start();
```
Key finding
Key finding

code can be joinpoint

\[\iff\]

code can be extracted into a method
Solution 1: Extract-method refactoring
Solution 1: Extract-method refactoring
Solution 2: Closure Joinpoints
Closure Joinpoints

mark code with closures instead of blocks

- more verbose than blocks
- also more restrictive

but: very strong static guarantees

+ allows for modular type checking
+ calling a closure joinpoint can never fail
+ no data races on local variables
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:" + theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;
}

jpi int JP(long s);
int around JP (long mySeed) {
```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:" + theSeed);
        return monteCarlo(theSeed);
    })(seed);
    return result;
}

jpi int JP(long s);
int around JP (long mySeed) {
}
```
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:");
        return monteCarlo(theSeed);
    }(seed);
    return result;
}

int JP(long s);
int around JP (long mySeed) {
}
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:");
    return monteCarlo(theSeed);
    }(seed);
    return result;
}

jpi int JP(long s);
int around JP (long mySeed) {
    return proceed(0);
}
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:" + theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;
}

jpi int JP(long s);
int around JP (long mySeed) {
    return proceed(0);
}
```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:" + theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;
}

int JP(long s);
int around JP (long mySeed) {
    return proceed(0);
}
```
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:"+theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;
}

jpi int JP(long s);
int around JP (long mySeed) {

    return proceed(0);
}
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:",theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;
}

int JP(long s);
int around JP (long mySeed) {
    return proceed(0);
}
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:" + theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;
}

jpi int JP(long s);
int around JP (long mySeed) {
    return proceed(0);
}
```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:"+theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;
}
```

```java
int JP(long s);
int around JP (long mySeed) {
    return proceed(0);
}
```
```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:" + theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;
}
```

```java
jpi int JP(long s);
int around JP (long mySeed) {
    new Thread() {
        public void run() {
            proceed(mySeed);
        }
    }.start();
}
```
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:"+theSeed);
        return monteCarlo(theSeed);
    }(seed);
    return result;  

}

int JP(long s);
int around JP (long mySeed) {
    new Thread() {
        public void run() {
            proceed(mySeed); }
    }.start();

CJPs are expressions, not statements!
```java
int monteCarloAlg() {
    long seed = System.currentTimeMillis();
    int result;
    result = exhibit JP(long theSeed) {
        System.out.println("seed was:");
        return monteCarlo(theSeed);
    }(seed);
    return result;  // result==42
}

int JP(long s);
int around JP (long mySeed) {
    new Thread() {
        public void run() {
            proceed(mySeed);
        }
    }.start();
    return 42;
}
```

CJPs are expressions, not statements!
Closure joinpoints: aspect side

JPIs, as before

advice directly refers to that joinpoint type

no pointcuts required

```java
jpi int JP(long s);

int around JP(long seed) {
    return proceed(0);
}
```
Closure joinpoints: base-code side

reference to same JPI

base code exhibits joinpoint as call to closure

result = exhibit JP(long theSeed) {
    println("seed was:"+theSeed);
    return monteCarlo(theSeed);
}(seed);

jpi int JP(long s);
Variable-access rules

class C {
    Field f;

    void foo(Param fp) {
        Local l;
        final Local L;
        exhibit JP(Param cp) {
            
        }(..);
    }
}
Variable-access rules

```java
class C {
    Field f;
    void foo(Param fp) {
        Local l;
        final Local L;
        exhibit JP(Param cp) {
            f = null;
        }(..);
    }
}
```

May read and write fields
Variable-access rules

```java
class C {
    Field f;

    void foo(Param fp) {
        Local l;
        final Local L;
        exhibit JP(Param cp) {
            println(cp);
        }(..);
    }
}
```

May read (and write) closure parameters
Variable-access rules

class C {
    Field f;

    void foo(Param fp) {
        Local l;
        final Local L;
        exhibit JP(Param cp) {
            println(L);
        }(..);
    }
}

May read final locals
Variable-access rules

```java
class C {
    Field f;

    void foo(Param fp) {
        Local l;
        final Local L;
        exhibit JP(Param cp) {
            println(l);
            println(fp);
        }(..);
    }
}
```

May NOT access non-final locals
Control-flow rules

class C {
    void foo() {
        for(int i=0; i<5; i++) {
            exhibit JP() {
                println(i);
            }
        }
    }
}
Control-flow rules

class C {
    void foo() {
        for(int i=0;i<5;i++) {
            exhibit JP()
            return;
        }
        println(i);
    }
}
Control-flow rules

```java
class C {
    void foo() {
        for(int i=0; i<5; i++) {
            exhibit JP(){
                return;
            }();
            println(i);
        }
    }
}
```

void around JP() {
    new Thread() {
        public void run() {
            proceed();
        }
    }.start();
}

“upward FUNARG problem” (Weizenbaum 1968, Moses 1970)
Control-flow rules

```java
class C {
    void foo() {
        for(int i=0; i<5; i++) {
            exhibit JP() {
                return;
            }();
            println(i);
        }
    }
}
```

break/continue/return always bind to closure, not to declaring method!
Control-flow rules

```java
class C {
    void foo() {
        for(int i=0; i<5; i++) {
            exhibit JP()
            break;
            ()
            println(i);
        }
    }
}
```

break/continue/return always bind to closure, not to declaring method!
Control-flow rules

```java
class C {
    void foo() {
        for(int i=0;i<5;i++) {
            exhibit JP() {
                continue;
            }();
            println(i);
        }
    }
}
```

break/continue/return always bind to closure, not to declaring method!
Syntactic sugar

\[ \text{exhibit } \text{JP}\{ \}
\] \[ \equiv \]

\[ \text{exhibit } \text{JP(\()\{ \}
\] \[ \equiv \]

\[ \text{exhibit } \text{JP()}\}; \]
CJPs and JPIs

Tight integration:
A closure joinpoint is just a special joinpoint and is processed like any other joinpoint.
Will it blend?
Will it blend?

www.willitblend.com
Will it blend? type?
Checked Exceptions

jpi void JPNone();
jpi void JPEx() throws Exception;
Checked Exceptions

```java
jpi void JPNone();

jpi void JPEx() throws Exception;
```

```java
before JPNone() throws Exception { }

before JPEx() throws Exception { }
```

Will it type?
Checked Exceptions

```java
jpi void JPNone();

jpi void JPEx() throws Exception;
```

Will it type?

```java
before JPNone() throws Exception { }
before JPEx() throws Exception { }
```
Checked Exceptions

```java
jpi void JPNone();

jpi void JPEx() throws Exception;

before JPNone() throws Exception { }
before JPEx() throws Exception { }

void foo() {
    ... exhibit JPNone() { ... }
}

void bar() {
    ... exhibit JPEx() { ... }
}
```

Will it type?
Checked Exceptions

jpi void JPNone();

jpi void JPEx() throws Exception;

before JPNone() throws Exception { }
before JPEx() throws Exception { }

void foo() {
    ... exhibit JPNone() { ... }
}

void bar() {
    ... exhibit JPEx() { ... }
}
public aspect TestCase {

    static void correct() {
        HashSet s = exhibit JP {
            ...
        };
    }

    HashSet JP();

    Set around JP() {
        return new TreeSet();
    }
}

public aspect TestCase {

    static void correct() {
        HashSet s = exhibit JP {
            ...
        };
    }

    jpi HashSet JP();

    Set around JP() {
        return new TreeSet();
    }

}
public aspect TestCase {

    static void correct() {
        HashSet s = exhibit JP {
            ...
        };
    }

    HashSet JP();

    Set around JP() {
        return new TreeSet();
    }
}

public aspect TestCase {

    static void correct() {
        HashSet s = exhibit JP {
            ...
        };
    }

    jpi HashSet JP();

    HashSet around JP() {
        return new TreeSet();
    }

}
... the same applies to argument types.

(Alternative: StrongAspectJ, De Fraine et al., AOSD 08)
Invariant Pointcuts

```java
jpi void JP(Number n);

aspect A{
    exhibits void JP(Number n) : call(void *(..)) && args(n);

    public static void main(String[] args){
        foo(new Integer(2));
    }

    void around JP(Number l){
        proceed(new Float(3));
    }

    public static void foo(Integer a){}
}
Invariant Pointcuts

```java
jpi void JP(Number n);

aspect A{
    exhibits void JP(Number n) : call(void *(..)) && args(n);
    public static void main(String[] args){
        foo(new Integer(2));
    }
    void around JP(Number l){
        proceed(new Float(3));
    }
    public static void foo(Integer a){}
}
```

AspectJ: match
Invariant Pointcuts

jpi void JP(Number n);

aspect A{

  exhibits void JP(Number n) : call(void *(..)) && argsinv(n);

  public static void main(String[] args){
    foo(new Integer(2));
  }

  void around JP(Number l){
    proceed(new Float(3));
  }

  public static void foo(Integer a){}
}
Invariant Pointcuts

- Same for thisinv and targetinv
- Warning if exhibits uses this/target/args
- Not nice but maybe AspectJ should have used different semantics in the first place...
More flexible typing through type parameters

```java
void printSet(Set s) { ... }

Set around LogMe() {
    Set ret = proceed();
    printSet(ret);
    return ret;
}
```

```java
jpi Set LogMe();
```

```java
exhibits Set LogMe(): call(* foo());
HashSet foo() { .. }
HashSet s = foo();
```
void printSet(Set s) { ... }

Set around LogMe() {
    Set ret = proceed();
    printSet(ret);
    return new TreeSet();
}

exhibits Set LogMe(): call(* foo());
HashSet foo() { .. }
HashSet s = foo();

jpi Set LogMe();
More flexible typing through type parameters

```java
void printSet(Set s) { ... }

Set around LogMe() {
    Set ret = proceed();
    printSet(ret);
    return new TreeSet();
}

jpi Set LogMe();

HashSet foo() { .. }
HashSet s = foo();

exhibit: Set LogMe(): call(* foo());
```
More flexible typing through type parameters

```java
void printSet(Set s) { ... }

Set around LogMe() {
    Set ret = proceed();
    printSet(ret);
    return new TreeSet();
}

exhibit Set LogMe(): call(* foo());
HashSet foo() { .. }
HashSet s = foo();
```

Prevent error by not matching!
More flexible typing through type parameters

```java
void printSet(Set s) { ... }

<S extends Set> S around LogMe() {
    S ret = proceed();
    printSet(ret);
    return ret;
}

<S extends Set> exhibits S LogMe(): call(* foo());
HashSet foo() { .. }
HashSet s = foo();
```

StrongAspectJ, De Fraine et al., AOSD 08
More flexible typing through type parameters

void printSet(Set s) { ... }

<S extends Set> S around LogMe() {
    S ret = proceed();
    printSet(ret);
    return ret;
}

jpi <S extends Set> S LogMe();

<S extends Set> exhibits S LogMe(): call(* foo());
HashSet foo() { .. }
HashSet s = foo();
More flexible typing through type parameters

void printSet(Set s) { ... }

<S extends Set> S around LogMe() {
    S ret = proceed();
    printSet(ret);
    return new TreeSet();
}

<S extends Set> exhibits S LogMe(): call(* foo());
HashSet foo() { .. }
HashSet s = foo();
More flexible typing through type parameters

StrongAspectJ, De Fraine et al., AOSD 08

```java
void printSet(Set s) { ... }

<S extends Set> S around LogMe() {
    S ret = proceed();
    printSet(ret);
    return ret;
}
```
Supporting logging-like concerns through global pointcuts

```java
<R> R around LogMe() {
    long timeBef = time();
    R ret = proceed();
    print(timeBef - time());
    return ret;
}
```

class A { <R> exhibits R LogMe(): ... }

class B { <R> exhibits R LogMe(): ... }

class C { <R> exhibits R LogMe(): ... }

...
Supporting logging-like concerns through global pointcuts

\[ \text{jpi } <R> \text{ R LogMe(): call(* *(..));} \]

Introduces default:

\[ \text{class A { \}} \]

\[ \equiv \]

\[ \text{class A { \} \text{ <R> exhibits R LogMe(): call(* *(..));} \}} \]
Do allow for refinements...

Seal a class:

```java
class A {
  <R> exhibits R LogMe();
}
```

Add joinpoints:

```java
class A {
  <R> exhibits R LogMe():
      global() || set(* *);
}
```
Do allow for refinements...

Seal a class:

```java
class A {
    <R> exhibits R LogMe();
}
```

Refine joinpoints:

```java
class A {
    <R> exhibits R LogMe():
        global() && call(* foo());
}
```
Result of typing rules

Invocation of a closure can never fail at runtime!

Strong typing!
Result of typing rules

Invocation of a closure can never fail at runtime!
Join Point Polymorphism

\[
\text{jpi \ void \ CheckingOut(float \ price, \ Customer \ c)}
\]
Join Point Polymorphism

```
void CheckingOut(float price, Customer c)

void Buying(Item i, float price, Customer cust)
  extends CheckingOut(price,cust);
```
Join Point Polymorphism

```java
// void CheckingOut
float price,
Customer cust)

// extends CheckingOut(price,cust);

// void Buying
Item i,
float price,
Customer cust)

// extends CheckingOut(price,cust);

"width subtyping"
```
Join Point Polymorphism

\begin{align*}
\text{jpi void CheckingOut} & \text{(float price, Customer c)} \\
\text{jpi void Buying} & \text{(Item i, float price, Customer cust) extends CheckingOut(price,cust);} \\
\text{jpi void Renting} & \text{(float price, int amt, Customer c) extends CheckingOut(price,c);} \\
\end{align*}
Advice-dispatch semantics

The most specific advice gets executed.

Aspect Discount
- around CheckingOut
- around Buying

Join Point
- jpi CheckingOut
- jpi Buying
- jpi Renting
The most specific advice gets executed.

Aspect Discount

around CheckingOut

around Buying

Advice-dispatch semantics

Join Point

Renting

jpi CheckingOut

jpi Buying

jpi Renting

The most specific advice gets executed.
Advice-dispatch semantics

The most specific advice gets executed.

Aspect Discount
around CheckingOut
around Buying

The most specific advice gets executed.
NO depth subtyping

jpi void CheckingOut(Customer c)

jpi void GoodCheckout(GoodCustomer c) extends CheckingOut(c);
NO depth subtyping

```java
jpi void CheckingOut(Customer c)

jpi void GoodCheckout(GoodCustomer c)
  extends CheckingOut(c);
```
NO depth subtyping

```java
jpi void CheckingOut(Customer c)

jpi void GoodCheckout(GoodCustomer c) extends CheckingOut(c);

void around CheckingOut(float price, Customer cus){
  proceed(price, new BadCustomer());
}
```
NO depth subtyping

```java
jpi void CheckingOut(Customer c)

jpi void GoodCheckout(GoodCustomer c)
  extends CheckingOut(c);

void around CheckingOut(float price,
  Customer cus){
  proceed(price, new BadCustomer());
}

void around GoodCheckout(float price,
  GoodCustomer cus){ ... }
```
NO depth subtyping

```java
jpi void CheckingOut(Customer c)

jpi void GoodCheckout(GoodCustomer c) extends CheckingOut(c);
```

Asp. A
```java
void around CheckingOut(float price, Customer cus){
  proceed(price, new BadCustomer());
}
```

Asp. B
```java
void around GoodCheckout(float price, GoodCustomer cus){ ... }
```
Only apparent solution

Forbid re-assignment of proceed values

e.g. Ptolemy, see Session 3
Static overloading

```cpp
void CheckingOut(Customer c)

void CheckingOut(float price, Customer c)
```
# Feature summary

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPIs as method signatures</td>
<td>preserves lexical scoping</td>
</tr>
<tr>
<td>CJPs</td>
<td>when pointcut awkward</td>
</tr>
<tr>
<td>Invariant typing (args, ret, exceptions)</td>
<td>no more ClassCastException</td>
</tr>
<tr>
<td>Invariant pointcuts</td>
<td></td>
</tr>
<tr>
<td>Width subtyping</td>
<td>better advice reuse</td>
</tr>
<tr>
<td>Generic Type Parameters</td>
<td></td>
</tr>
<tr>
<td>Global pointcuts</td>
<td>fewer exhibit clauses</td>
</tr>
</tbody>
</table>
Implementation

- All implemented within abc
- Type-checking pass (JastAdd)
- All constructs flattened into plain AJ
  - CJPs extracted into methods
  - Associate correct pointcut with each advice
- Resulting runtime overhead: **Zero**!

Thanks Milton!
Closely Related Work

Pointcut Interfaces (Gudmundson & Kiczales)
- refactoring only, no language support

IIIA (Steimann et al.)
- First attempt to de-couple aspects from base code through types

Ptolemy
- Only explicit events
- Hence no quantification (incl. global)
- No re-assignment of proceed values
  - Hence: depth subtyping
Evaluation

- Study subjects: AJHotDraw, Glassbox, SpaceWar, LawOfDemeter (LoD)
- JPIs applicable in all cases
- Subtyping surprisingly useful (e.g. Glassbox)
- Generics avoid most redefinitions
- Global Pointcuts really useful for LoD
Join Point Interfaces

FOAL Keynote

Eric Bodden will be giving a keynote talk about JPIs at FOAL 2012.

Introduction

Join point interfaces (JPIs) are contracts between aspects and advised code. JPIs are an extension and refinement of the notion of join point types recently introduced by Steiman et al. JPIs support a programming methodology where aspects only specify the types of join points they advise based on a JPI, not on concrete pointcuts. It is the responsibility of the programmer maintaining the advised code to specify, through an `exhibits` clause, which join points are of which type. Aspects and advised code can be developed and evolved independently.
Join Point Interfaces for Modular Reasoning in Aspect-Oriented Programs

Milton Nistruțu | Eric Bodden
PUC-Rio Lab | PUC-Rio Lab
Computer Science Department | University of Chile
São Paulo, Brazil | Santiago, Chile

ABSTRACT

Join point interfaces (JPIs) are a way to modularize common concerns in aspect-oriented programming (AOP) by defining a contract between normal code and advice. JPIs are a way to modularize common concerns in aspect-oriented programming (AOP) by defining a contract between normal code and advice. JPIs are a way to modularize common concerns in aspect-oriented programming (AOP) by defining a contract between normal code and advice. JPIs are a way to modularize common concerns in aspect-oriented programming (AOP) by defining a contract between normal code and advice.

1. JOIN POINT INTERFACES

Inheritance, aspect-oriented programming (AOP), and modular reasoning. Inheritance, aspect-oriented programming (AOP), and modular reasoning. Inheritance, aspect-oriented programming (AOP), and modular reasoning. Inheritance, aspect-oriented programming (AOP), and modular reasoning.

1.1. Aspects and modular reasoning

Aspects are a way to modularize common concerns in aspect-oriented programming (AOP) by defining a contract between normal code and advice. Aspects are a way to modularize common concerns in aspect-oriented programming (AOP) by defining a contract between normal code and advice. Aspects are a way to modularize common concerns in aspect-oriented programming (AOP) by defining a contract between normal code and advice. Aspects are a way to modularize common concerns in aspect-oriented programming (AOP) by defining a contract between normal code and advice.

Categories and Subject Descriptors

1. Software Engineering/1.3. Programming Languages/1.3. Languages Constructs

General Terms

1. Design, Languages

Keywords

Aspect-oriented programming, modularity
Join Point Interfaces for Modular Reasoning in Aspect-Oriented Programs

Milton Inostroza, Eric Tanter

ABSTRACT

While aspect-oriented programming supports the modular definition of concerning concerns, most approaches to aspects use base code that is difficult to reason about. The basic problem is that aspects usually require a different programming model and require a different set of tools. These differences make aspects fragile. Changes in one of the components of a system, which is a frequent event, often break a functional definition of an aspect. While it is important to have a different definition for aspects, the separate development of aspects and base code is challenging. This work introduces a new way of reasoning about aspects called Join Point Interfaces. Join Point Interfaces (JPIs) are used to define a new functional interface for base code that can define points that can be leveraged by aspects. This interface is able to be reused by base code that can be leveraged by aspects. This allows the base code to define points that are used by aspects. The JPI approach supports the modular reasoning about aspects, and it can be used to simplify the programming of aspects. It also supports the modular reasoning about aspects. In this way, JPIs can be used to simplify the programming of aspects. In this way, JPIs can be used to simplify the programming of aspects. In this way, JPIs can be used to simplify the programming of aspects.
Open problems

- Pointcuts in classes defeat the purpose of quantification
  => Lift “exhibits” declaration to modules

- What about inter-type declarations?

- Interplay with execution layers/membranes
  (see next talk)
Separate evolution though strong typing

Class C
- cjp
- join point (shadow)
- exhibit
- pointcut

Aspect A
- advice

http://bodden.de/jpi
http://bodden.de/cjp