

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 ...



Aspect-Oriented Programming is Quantification and Obliviousness

**Robert E. Filman
Daniel P. Friedman**

RIACS Technical Report 01.12

May 2001

**Workshop on Advanced Separation of Concerns, OOPSLA 2000,
October 2000**

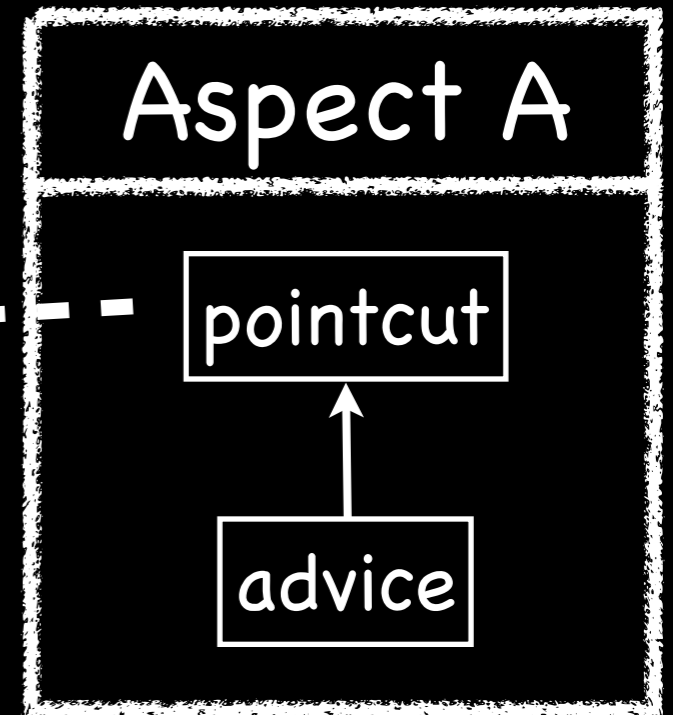
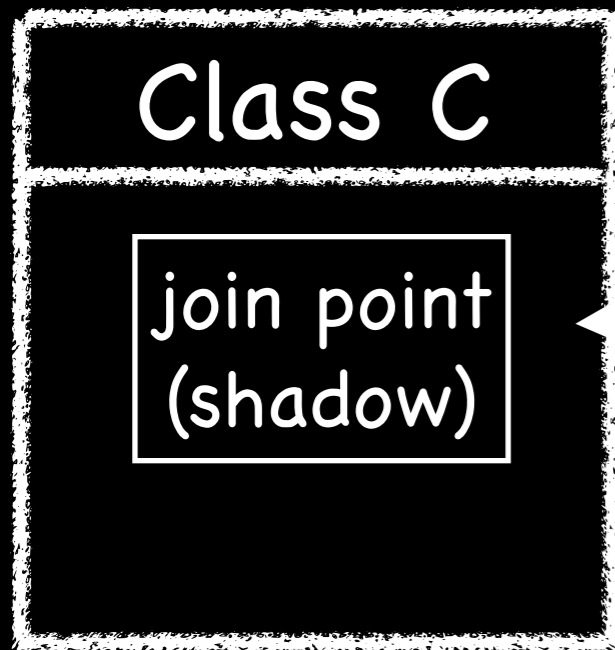
AOP is ...

Programming support for
implementing a separation of concerns

Dependencies in traditional AOP

Main-stream
software developer

AOP
Expert



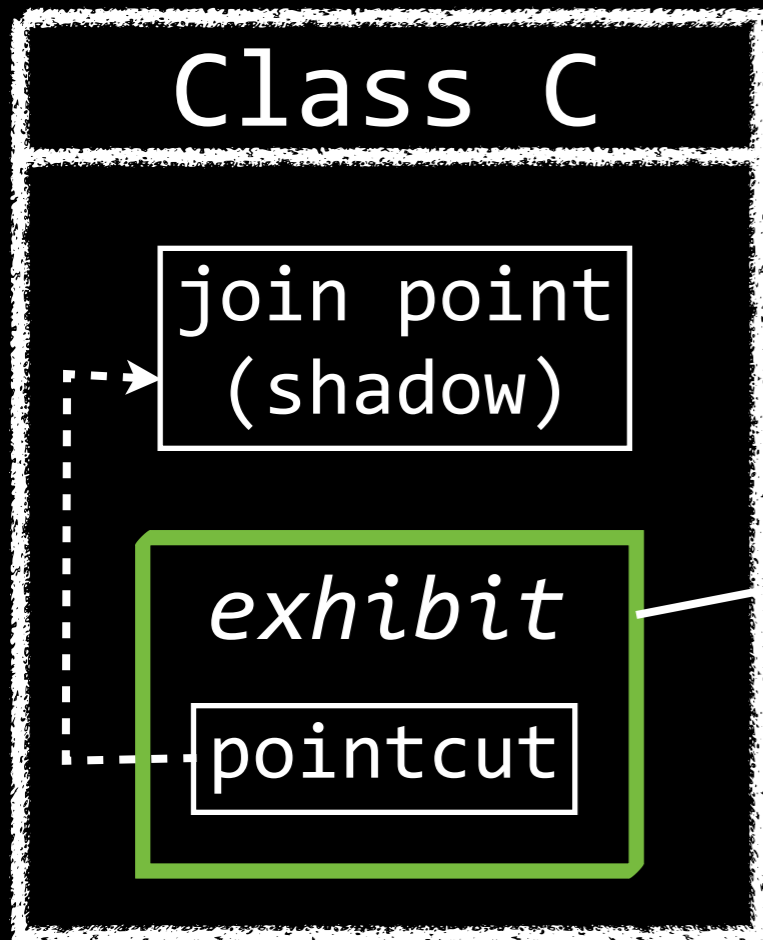
Global
reasoning

Join Point Interfaces



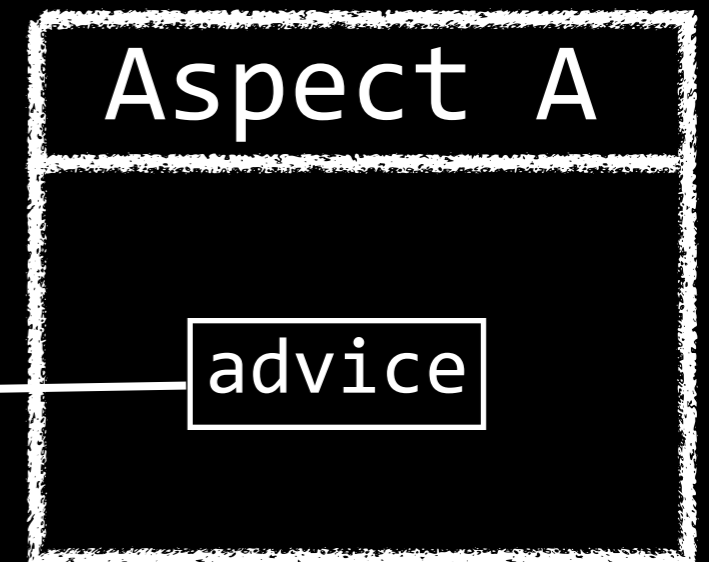
Join Point Interfaces (JPI)

Main-stream
software developer



jpi

AOP
Expert



Separate
evolution

`jpi` ReturnType Name (FormalParameters)* CheckedExceptions*

“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

```
jpi 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" !

Base code

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

```
class ShoppingSession{  
    ShoppingCart sc = new ShoppingCart();  
    Invoice inv = new Invoice();  
  
    void checkOut(Item item, float price,  
                 int amount, Customer cust){  
        sc.add(item, amount);  
        inv.add(item, amount, cus);  
    }  
}
```

No reference to Discount aspect !

Base code

```
jpi void CheckingOut(Item i, float price, int amt, Customer c)
```

```
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);  
}
```

```
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)
```

```
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 Joints!

Non-Solution: Block Joints!

IIIA

Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. TOSEM, 20(1):1–43, 2010.

```
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);  
}
```

IIIA

Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. TOSEM, 20(1):1–43, 2010.

```
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);  
}
```

IIIA

Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. TOSEM, 20(1):1–43, 2010.

```
int monteCarloAlg() {  
    long seed = System.currentTimeMillis();  
    int result; ————— seed == <time>  
    exhibit SomeAspect.JP(seed) {  
        System.out.println("seed was:" + seed); ————— seed == 0  
        result = monteCarlo(seed); ————— seed == <time>  
    }  
    return result; }  
}
```

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

IIIA

Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. TOSEM, 20(1):1–43, 2010.

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

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

IIIA

Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. TOSEM, 20(1):1–43, 2010.

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

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

```
}
```

Key finding

Key finding

code can be joinpoint

\Leftrightarrow

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) {

}
```

```
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:" + 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:"+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;
}
```



```
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);

}
```

```
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;  
}
```

seed==<time>
theSeed==0

```
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;  
}
```

seed==<time>
theSeed==0

```
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;  
}
```

seed==<time>
theSeed==0

```
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; ———— result==???
```

```

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; ———— result==???
```

```

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

CJPs are
expressions,
not statements!

```

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; result==42
}

```

```

jpi 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

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

advice directly
refers to that
joinpoint type

```
int around JP(long seed) {  
    return proceed(0);  
}
```

no pointcuts required

Closure joinpoints: base-code side

reference to
same JPI

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

base code
exhibits joinpoint
as call to closure

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

Variable-access rules

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


Variable-access rules

```
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


```
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

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

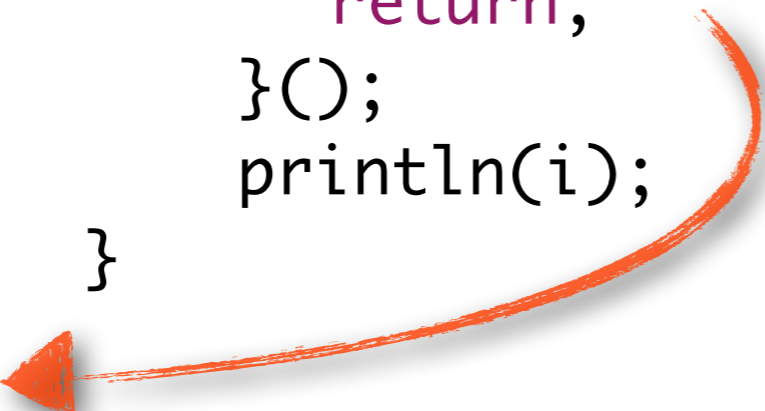
May NOT access non-final locals

Control-flow rules

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

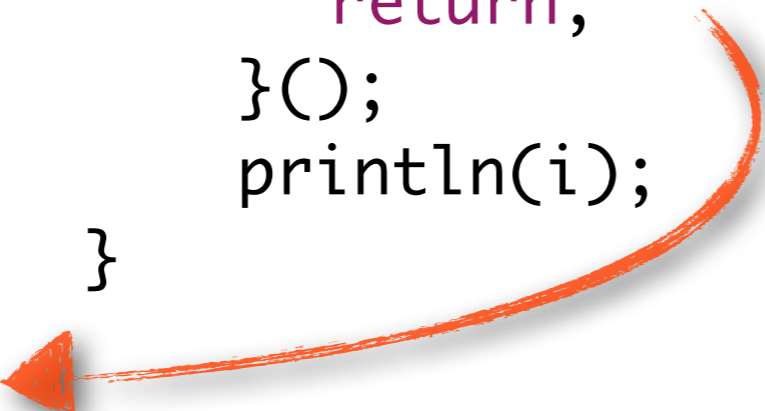

Control-flow rules

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



Control-flow rules

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

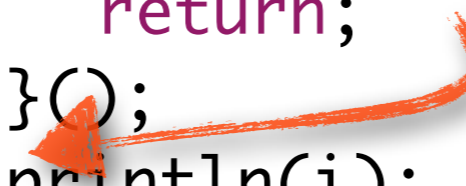


```
void around JPC() {  
    new Thread() {  
        public void run() {  
            proceed(); }  
    }.start();  
}
```

“upward FUNARG problem”
(Weizenbaum 1968,
Moses 1970)

Control-flow rules

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



prints:

```
1  
2  
3  
4  
5
```

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

Control-flow rules

```
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

```
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

```
exhibit JP{  
};
```

≡

```
exhibit JP() {  
};
```

CJPs and JPIs

Tight integration:

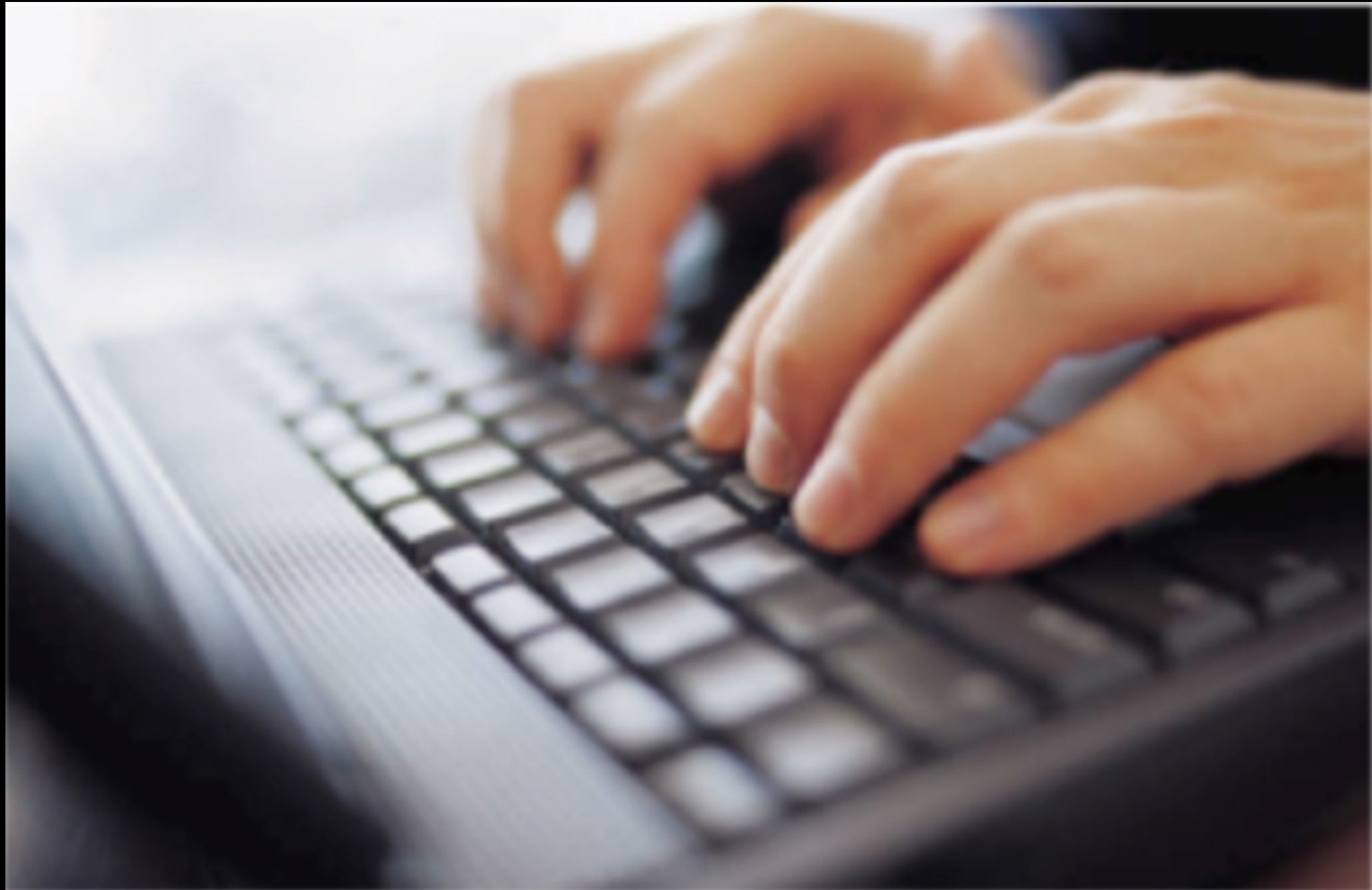
A closure joinpoint is just a special joinpoint and is processed like any other joinpoint.

Will it blend?

Will it blend?



Will it ~~blend~~? *type*?



Checked Exceptions

```
jpi void JPNone();
```

```
jpi void JPEx() throws Exception;
```

Checked Exceptions

```
jpi void JPNone();
```

```
jpi void JPEx() throws Exception;
```

Will it
type?

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

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

Checked Exceptions

```
jpi void JPNone();
```

```
jpi void JPEx() throws Exception;
```

Will it
type?

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

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

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() { ... }  
}
```

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() { ... }  
}
```

Will it
type?

Invariant Return Types

```
public aspect TestCase {  
  
    static void correct() {  
        HashSet s = exhibit JP {  
            ...  
        };  
    }  
  
    jpi HashSet JPC();  
  
    Set around JPC() {  
        return new TreeSet();  
    }  
  
}
```




Invariant Return Types

```
public aspect TestCase {  
  
    static void correct() {  
        HashSet s = exhibit JP {  
            ...  
        };  
    }  
  
    jpi HashSet JPC();  
  
    Set around JPC() {  
        return new TreeSet();  
    }  
  
}
```

The diagram consists of three orange arrows. A long, curved arrow starts from the word 'exhibit' in the line 'HashSet s = exhibit JP { ... };' and points to the 'HashSet' type. A shorter, double-headed arrow is positioned between the 'HashSet' type in the 'jpi HashSet JPC();' line and the 'Set' type in the 'Set around JPC() { return new TreeSet();' line. A third arrow starts from the 'return new TreeSet();' line and points to the 'HashSet' type in the 'HashSet s = exhibit JP { ... };' line, indicating that TreeSet is a subtype of HashSet.

Invariant Return Types

```
public aspect TestCase {  
  
    static void correct() {  
        HashSet s = exhibit JP {  
            ...  
        };  
    }  
  
    jpi HashSet JPC();  
  
    Set around JPC() {  
        return new TreeSet();  
    }  
  
}
```



Invariant Return Types

```
public aspect TestCase {  
  
    static void correct() {  
        HashSet s = exhibit JP {  
            ...  
        };  
    }  
  
    jpi HashSet JP();  
  
    HashSet around JP() {  
        return new TreeSet();  
    }  
  
}
```

Invariant Return Types

```
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

```
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

```
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){}  
}
```



argsinv: no match

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

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

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

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

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

More flexible typing through type parameters

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

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

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

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

More flexible typing through type parameters

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

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

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



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

More flexible typing through type parameters

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

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

Prevent error by
not matching!

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

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

More flexible typing through type parameters

StrongAspectJ, De Fraine et al., AOSD 08

```
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

StrongAspectJ, De Fraine et al., AOSD 08

```
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

StrongAspectJ, De Fraine et al., AOSD 08

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

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

```
<S extends Set> S LogMe();
```



```
<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

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

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

Supporting logging-like concerns through global pointcuts

```
<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

```
jpi <R> R LogMe(): call(* *(..));
```

Introduces default:

```
class A {  
  
}
```



```
class A {  
  <R> exhibits R LogMe(): call(* *(..));  
}
```

Do allow for refinements...

Seal a class:

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

Add joinpoints:

```
class A {  
  <R> exhibits R LogMe():  
    global() || set(* *);  
}
```

Do allow for refinements...

Seal a class:

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

Refine joinpoints:

```
class A {  
  <R> exhibits R LogMe():  
    global() && call(* foo());  
}
```

Result of typing rules

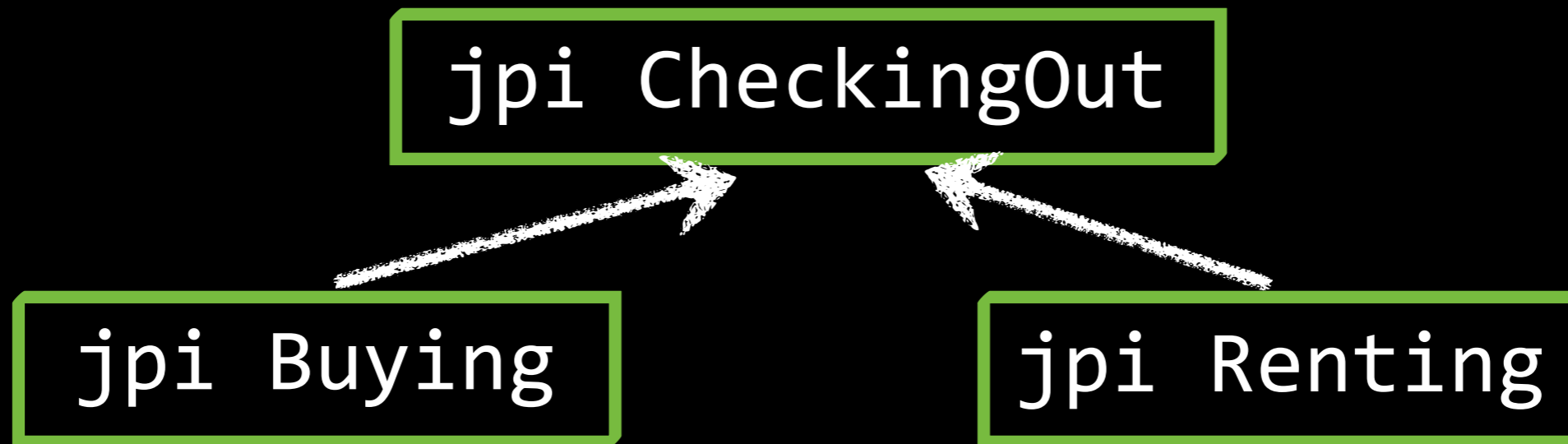


Strong typing!

Result of typing rules

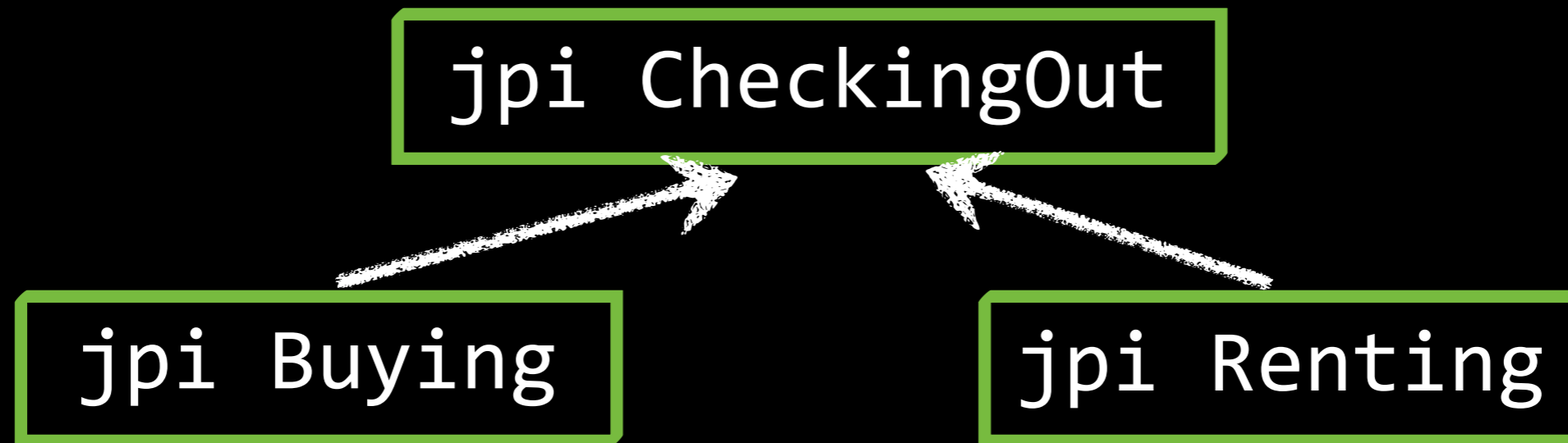
Invocation of a closure
can never fail
at runtime!

Join Point Polymorphism



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

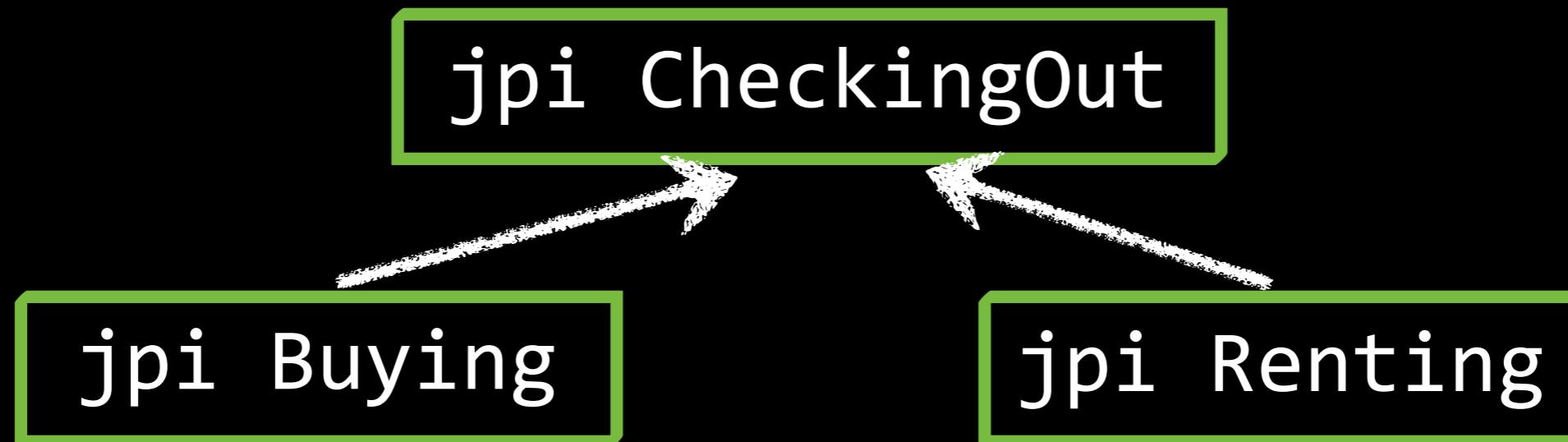

Join Point Polymorphism



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

```
jpi void Buying(Item i, float price, Customer cust)  
extends CheckingOut(price, cust);
```

Join Point Polymorphism

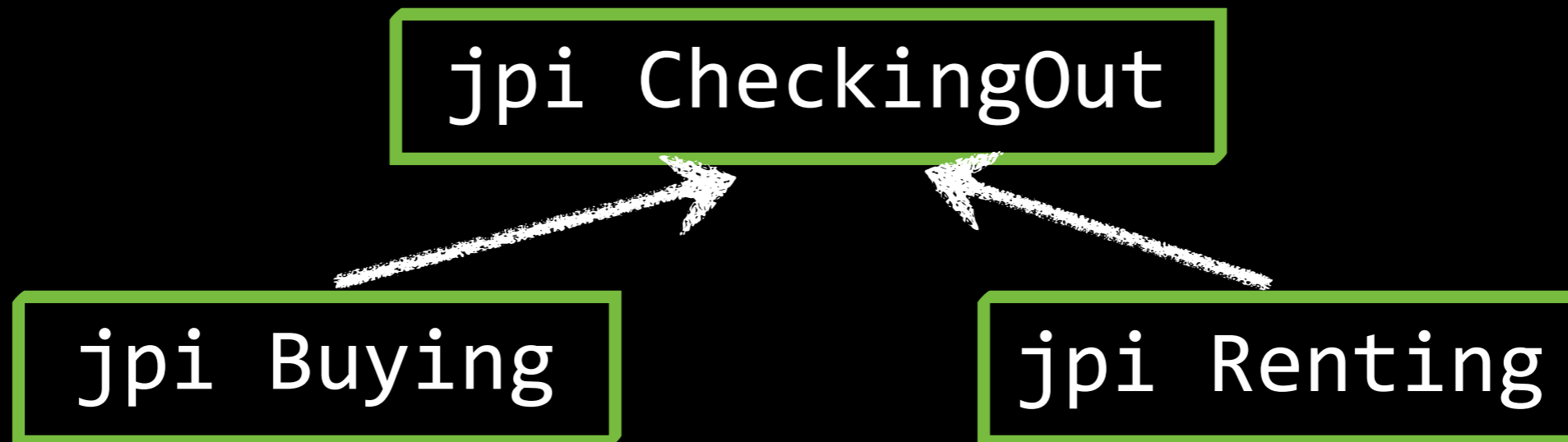


```
jpi void CheckingOut(float price, Customer c)
```

```
jpi void Buying(Item i, float price, Customer cust)  
extends CheckingOut(price, cust);
```

“width subtyping”

Join Point Polymorphism



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

```
jpi void Buying(Item i, float price, Customer cust)  
extends CheckingOut(price,cust);
```

```
jpi void Renting(float price, int amt, Customer c)  
extends CheckingOut(price,c);
```

Advice-dispatch semantics



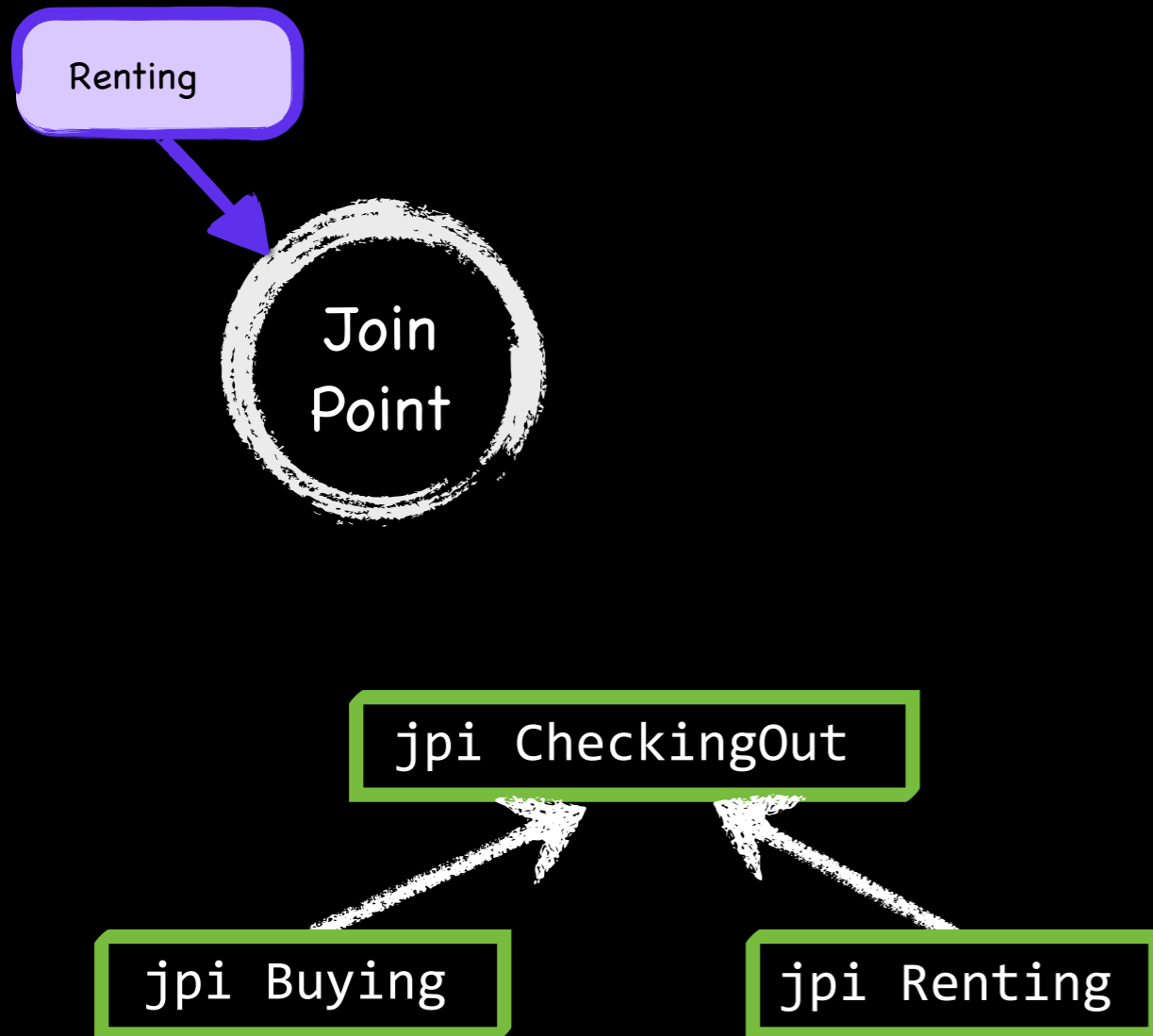
Aspect Discount

around CheckingOut

around Buying

The most specific advice gets executed.

Advice-dispatch semantics



The most specific advice gets executed.

Advice-dispatch semantics

Aspect Discount

around CheckingOut

around Buying



The most specific advice gets executed.

NO depth subtyping

```
jpi void Checkout(Customer c)
```

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

NO depth subtyping

```
jpi void Checkout(Customer c)
```

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



NO depth subtyping

```
jpi void Checkout(Customer c)
```

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



Asp. A

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

NO depth subtyping

```
jpi void Checkout(Customer c)
```

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



Asp. A

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

Asp. B

```
void around GoodCheckout(float price,  
                         GoodCustomer cus){ ... }
```

NO depth subtyping

```
jpi void Checkout(Customer c)
```

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




Asp. A

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

Asp. B

```
void around GoodCheckout(float price,  
                          GoodCustomer cus){ ... }
```



Only apparent solution

Forbid re-assignment of proceed values

e.g. Ptolemy, see Session 3

Static overloading

```
jpi void CheckingOut(Customer c)
```

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

Feature summary

JPIs as method signatures	preserves lexical scoping
CJPs	when pointcut awkward
Invariant typing (args, ret, exceptions)	no more ClassCastExceptions
Invariant pointcuts	
Width subtyping	better advice reuse
Generic Type Parameters	
Global pointcuts	fewer exhibit clauses

Implementation

- All implemented within abc
- Type-checking pass (JastAdd)
- All constructs flattened into plain AJ
 - CJP's 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: AHotDraw, 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

<http://bodden.de/jpi>

Eric Bodden, Ph.D.

Head of Secure Software Engineering Group at EC SPRIDE
Principal Investigator in Secure Services at CASED



19°C

rss

HOME

RESEARCH

TOOLS

TEACHING

ABOUT ME

OPEN PHD POSITIONS IN EC SPRIDE

PHOTOS

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.

WELCOME

Welcome to my website. Interested in my research? Click here for details or jump directly to my [publications](#).

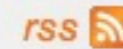
UPCOMING CONFERENCES



<http://bodden.de/jpi>

Eric Bodden, Ph.D.

Head of Secure Software Engineering Group at EC SPRIDE
Principal Investigator in Secure Services at CASED



Join Point Interfaces for Modular Reasoning in Aspect-Oriented Programs

Milton Inostroza, Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile - Santiago, Chile
{minostro,etanter}@dcc.uchile.cl

Eric Bodden
Software Technology Group
Center for Advanced Security Research
Darmstadt (CASED)
Technische Universität Darmstadt - Germany
bodden@acm.org

ABSTRACT

While aspect-oriented programming supports the modular definition of crosscutting concerns, most approaches to aspect-oriented programming fail to improve, or even preserve, modular reasoning. The main problem is that aspects usually carry, through their pointcuts, explicit references to the base code. These dependencies make programs fragile. Changes in the base code can unwittingly break a pointcut definition, rendering the aspect ineffective or causing spurious matches. Conversely, a change in a pointcut definition may cause parts of the base code to be advised without notice. Therefore separate development of aspect-oriented programs is largely compromised, which in turn seriously hinders the adoption of aspect-oriented programming by practitioners.

We propose to separate base code and aspects using Join Point Interfaces, which are contracts between aspects and base code. Base code can define pointcuts that expose selected join points through a Join Point Interface. Conversely, an aspect can offer to advise join points that provide a given Join Point Interface. Crucially, however, aspect themselves cannot contain pointcuts, and hence cannot refer to base code elements. In addition, because a given join point can provide several Join Point Interfaces, and Join Point Interfaces can be organized in a subtype hierarchy, our approach supports join point polymorphism. We describe a novel advice dispatch mechanism that offers a flexible and type-safe approach to aspect reuse.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs

General Terms

Design, Languages

Keywords

Aspect-oriented programming, modularity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'11, September 3-8, 2011, Szeged, Hungary.
Copyright 2011 ACM 978-1-4503-0443-6/11/09...\$5.00.

1. JOIN POINT INTERFACES

Inevitably, aspect-oriented programming [3] aids modularizing crosscutting code: it helps programmers to put together code that belongs together. So far, however, most approaches to aspect-oriented programming have failed to improve, or even preserve, modular reasoning. Such modular reasoning is easily established in traditional procedural languages. In such languages, programmers can reason about one procedure at a time and in isolation. The procedure's signature establishes a strong contract with the contexts in which the procedure may be used. Object-oriented programming already gives up modular reasoning to some extent. Object-oriented programs carry virtual method calls. For such calls, at least in statically typed languages, the signature of the call is known at the call site, and hence the usage contract for the called method is known as well. It is unknown, however, which concrete implementation the virtual method call will eventually be dispatched to in the running program.

1.1 Aspects and modular reasoning

With aspect-oriented programming, modular reasoning becomes even harder, as aspect-oriented programs add a crucial feature: implicit invocation with implicit announcement (IIIA) [7]. Through IIIA, an aspect can become active at many different program points (called join points) without any explicit call to the aspect being present at these points. This is problematic because maintainers of the base code may be unaware of the program point being advised, and hence may refactor this point or change the point otherwise, unwittingly breaking the connection to the advising aspect. There has been several attempts to discuss and enhance the possibilities for modular reasoning in presence of pointcuts and advice, but they all eventually fall short in supporting full separate development with static and modular typechecking.

Running Example

As a running example, we will consider an e-commerce system with a set of discount rules. (Deliberately, we keep the example similar to a motivating example by Steiman et al. [7].) In the initial system, a customer can check out a product by either buying or renting the product. On the customer's birthday, the customer will be given a 5% discount when checking out a product. We will be adding further rules later.

HOME

OPEN PHD POSITIONS IN EC SPRIDE

PHOTOS

Join P

FOAL

Eric Bodd

Introc

Join point

and refine

programm

ing technology, which depends only upon the type 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.

WELCOME

Welcome to my website. Interested in my research? Click here for details or jump directly to my publications.

UPCOMING CONFERENCES

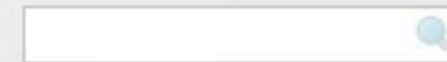
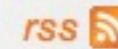


<http://bodden.de/jpi>

Eric Bodden, Ph.D.

Head of Secure Software Engineering Group at EC SPRIDE
Principal Investigator in Secure Services at CASED

19°C



Join Point Interfaces for Modular Reasoning in Aspect-Oriented Programs

Milton Inostroza, Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile - Santiago, Chile
{minostro,etanter}@dcc.uchile.cl

Eric Bodden
Software Technology Group
Center for Advanced Security Research
Darmstadt (CASED)
Technische Universität Darmstadt - Germany
bodden@acm.org

ABSTRACT

While aspect-oriented programming supports the modular definition of crosscutting concerns, most approaches to aspect-oriented programming fail to improve, or even preserve, modular reasoning. The main problem is that aspects usually carry, through their pointcuts, explicit references to the base code. These dependencies make programs fragile. Changes in the base code can unwittingly break a pointcut definition, rendering the aspect ineffective or causing spurious matches. Conversely, a change in a pointcut definition may cause parts of the base code to be advised without notice. Therefore separate development of aspect-oriented programs is largely compromised, which in turn seriously hinders the adoption of aspect-oriented programming by practitioners.

We propose to separate base code and aspects using Join Point Interfaces, which are contracts between aspects and base code. Base code can define pointcuts that expose selected join points through a Join Point Interface. Conversely, an aspect can offer to advise join points that provide a given Join Point Interface. Crucially, however, aspects themselves cannot contain pointcuts, and hence cannot refer to base code elements. In addition, because a given join point can provide several Join Point Interfaces, and Join Point Interfaces can be organized in a subtype hierarchy, our approach supports join point polymorphism. We describe a novel advice-dispatch mechanism that offers a flexible and type-safe approach to aspect reuse.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs

General Terms

Design, Languages

Keywords

Aspect-oriented programming, modularity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'11, September 5-9, 2011, Szeged, Hungary.
Copyright 2011 ACM 978-1-4503-0443-6/11/09...\$5.00.

1. JOIN POINT INTERFACES

Inevitably, aspect-oriented programming [2] aids modularizing crosscutting code: it helps programmers to put together code that belongs together. So far, however, most approaches to aspect-oriented programming have failed to improve, or even preserve, modular reasoning. Such modular reasoning is easily established in traditional procedural languages. In such languages, programmers can reason about one procedure at a time and in isolation. The procedure's signature establishes a strong contract with the contexts in which the procedure may be used. Object-oriented programming already gives up modular reasoning to some extent. Object-oriented programs carry virtual method calls. For each call, at least in statically typed languages, the signature of the call is known at the call site, and hence the usage contract for the called method is known as well. It is unknown, however, which concrete implementation the virtual method call will eventually be dispatched to in the running program.

1.1 Aspects and modular reasoning

With aspect-oriented programming, modular reasoning becomes even harder, as aspect-oriented programs add a crucial feature: implicit invocation with implicit announcements (IIIA) [7]. Through IIIA, an aspect can become active at many different program points (called join points) without any explicit call to the aspect being present at these points. This is problematic because maintainers of the base code may be unaware of the program point being advised, and hence may refactor this point or change the point otherwise, unwittingly breaking the connection to the advising aspect. There has been several attempts to discuss and enhance the possibilities for modular reasoning in presence of pointcuts and advice, but they all eventually fall short in supporting full separate development with static and modular typechecking.

Running Example

As a running example, we will consider an e-commerce system with a set of discount rules. (Deliberately, we keep the example similar to a motivating example by Steiman et al. [7].) In the initial system, a customer can check out a product by either buying or renting the product. On the customer's birthday, the customer will be given a 5% discount when checking out a product. We will be adding further rules later.

Technical Report

No. TUD-CS-2011-0272
October 3rd, 2011



Modular Reasoning with Join Point Interfaces

Authors
Milton Inostroza (Universidad de Chile)
Éric Tanter (Universidad de Chile)
Eric Bodden (CASED)

HOME

Join F

FOAL

Eric Bodd

Introc

Join point

and refine

programm

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.

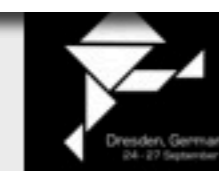
OPEN PHD POSIT

2.

advised code. JPIs a

d by Steiman et al.

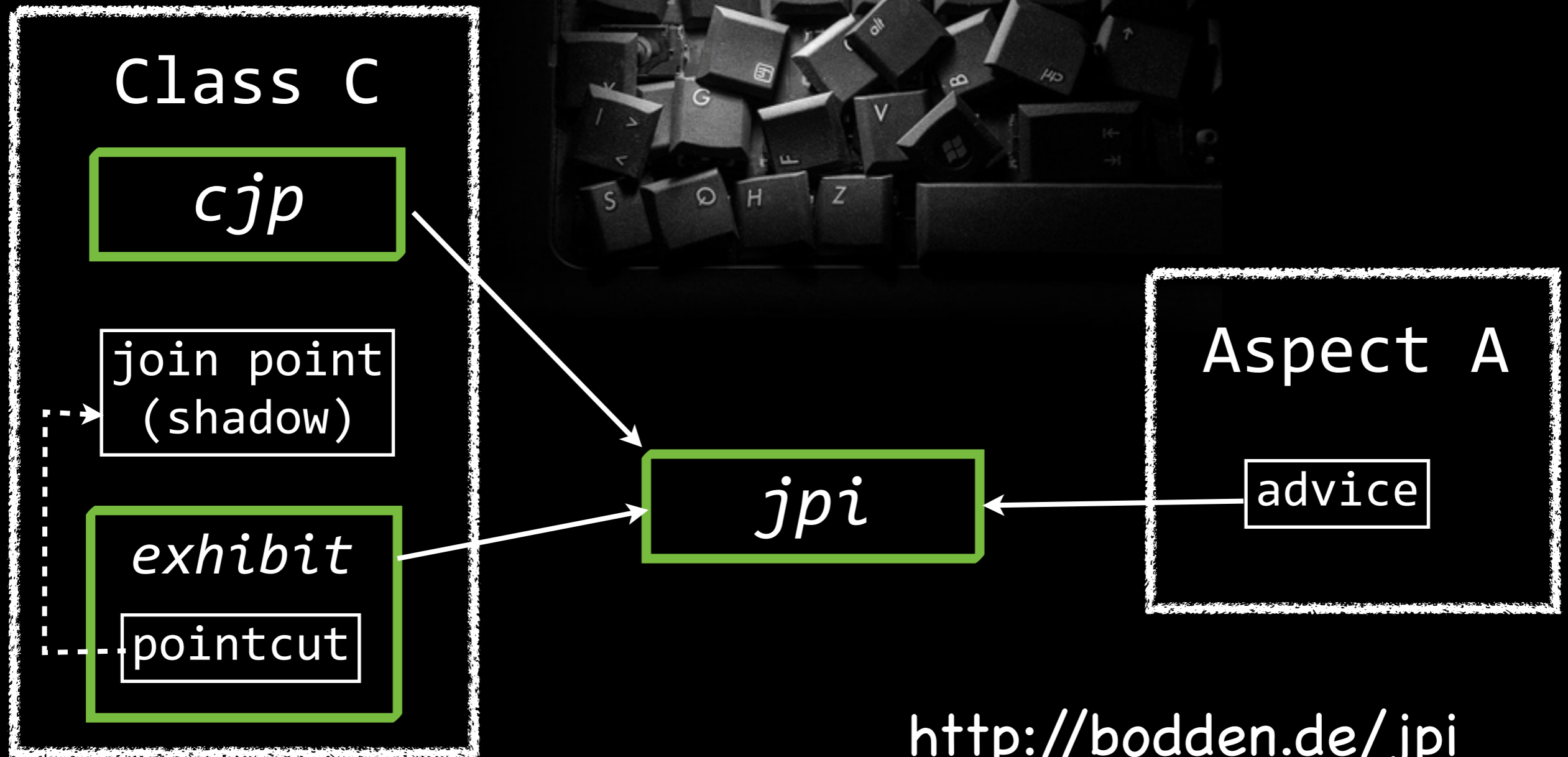
of join points they a



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 through strong typing



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