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

Eric Bodden
joint work with Milton Inostroza, Éric Tanter

SECURE SOFTWARE ENGINEERING GROUP
EC SPRIDE EUROPEAN CENTER FOR SECURITY AND PRIVACY BY DESIGN
TECHNISCHE UNIVERSITÄT DARMSTADT

Aspect-oriented programming successfully modularizes crosscutting concerns.

Aspect-oriented programming successfully modularizes crosscutting concerns.

Aspect-oriented programming fails to preserve modular reasoning.

AOP is …

riacs Research Institute for Advanced Computer Science

Aspect-Oriented Programming is Quantification and Obliviousness

Robert E. Filman
Daniel P. Friedman

RIACS Technical Report 01.12

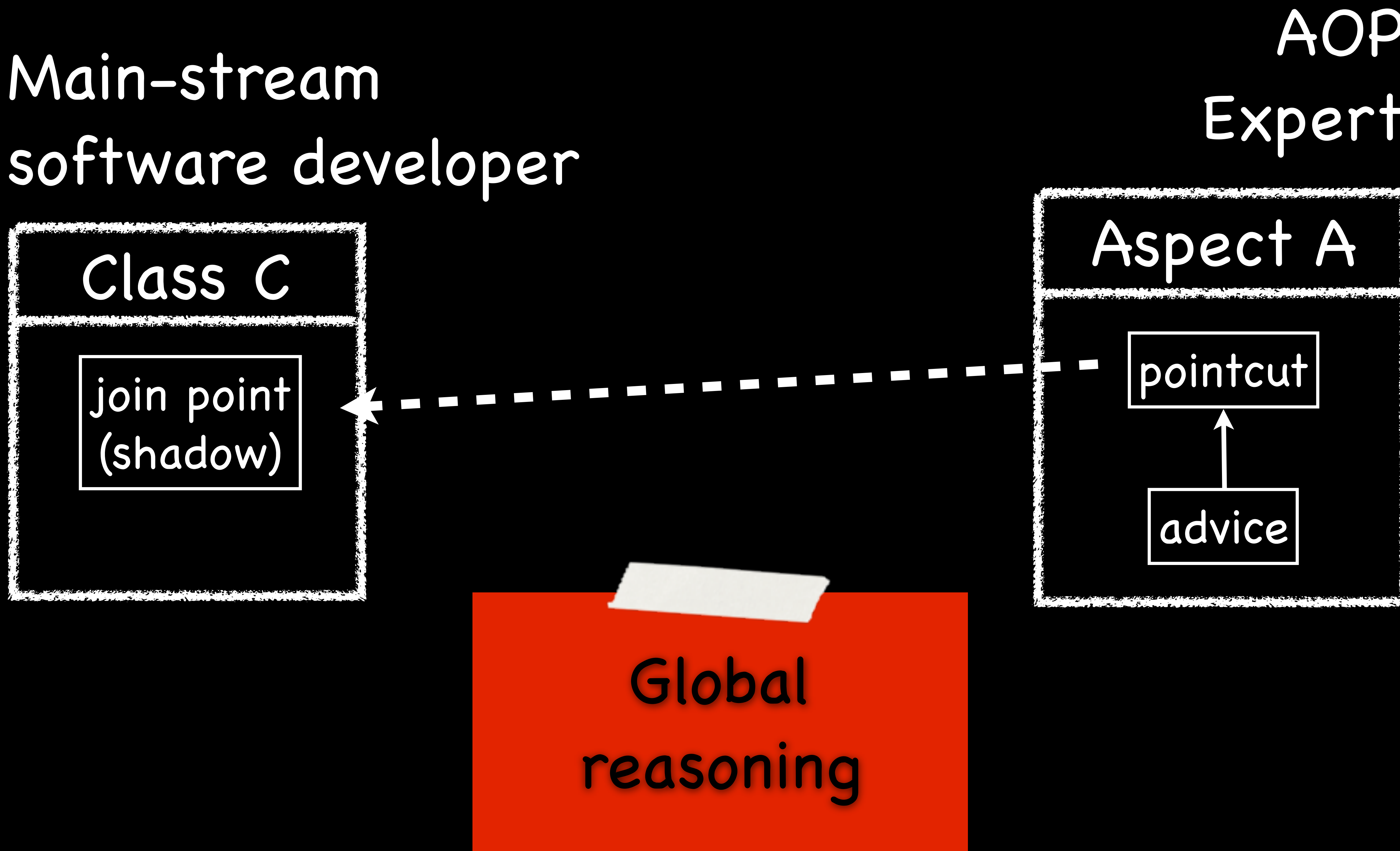
May 2001

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

AOP is …

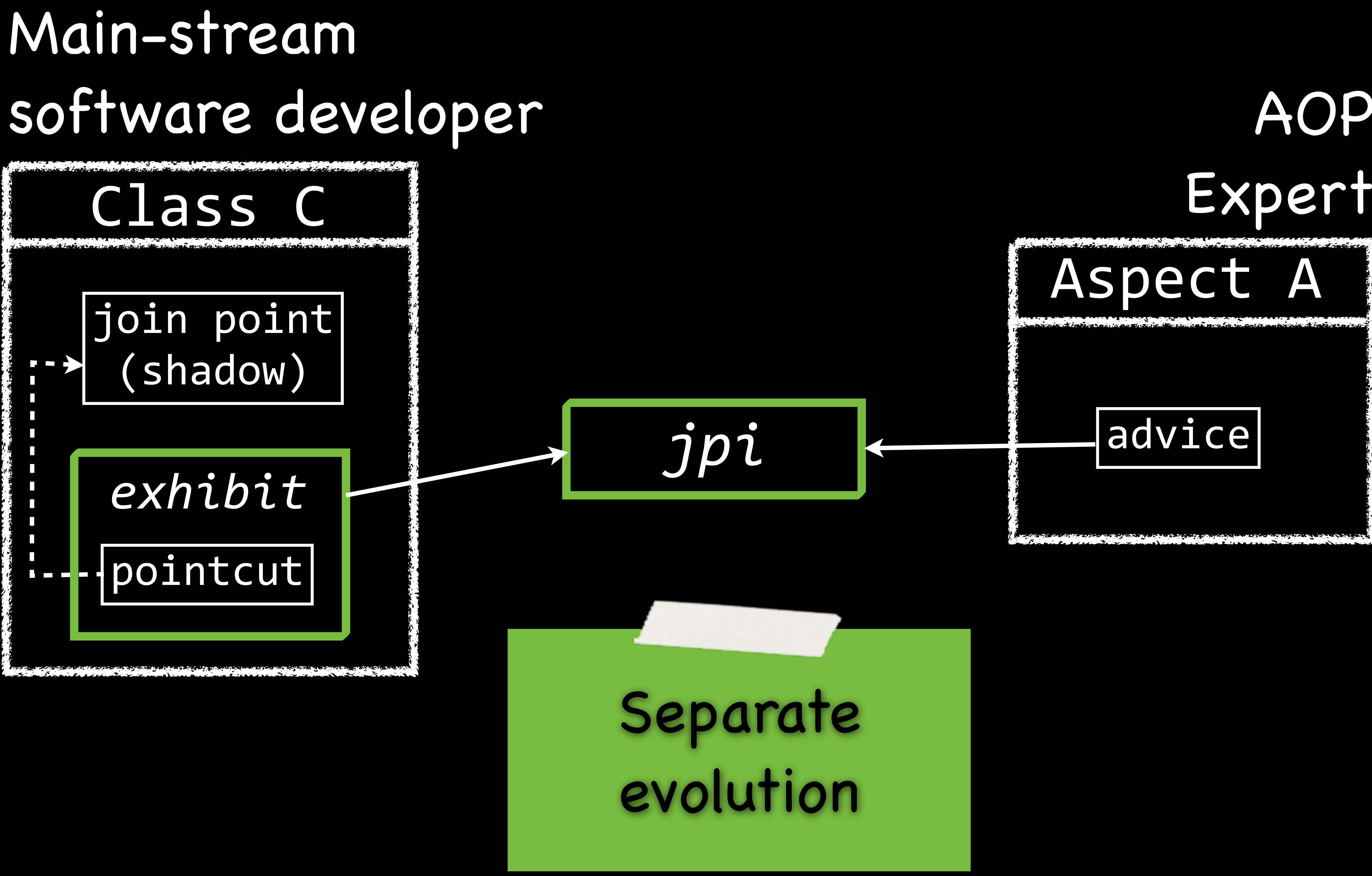
Programming support for
implementing a separation of concerns

Dependencies in traditional AOP



Join Point Interfaces

Join Point Interfaces (JPI)



```
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...

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

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

```java
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):
```

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

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

```java
pointcut monteCarloCall(long seed):

    withincode(* monteCarloAlg())
```

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

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

```java
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 Joinpoints!

Non-Solution: Block Joinpoints!

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) {




}
```

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

seed==<time>

seed==0

seed==<time>

```
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

<=>

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);



}
```

```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) {


     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 JP(){
              return;
            }();
            println(i);
        }
    }
}
```

Control-flow rules

```
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

```
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

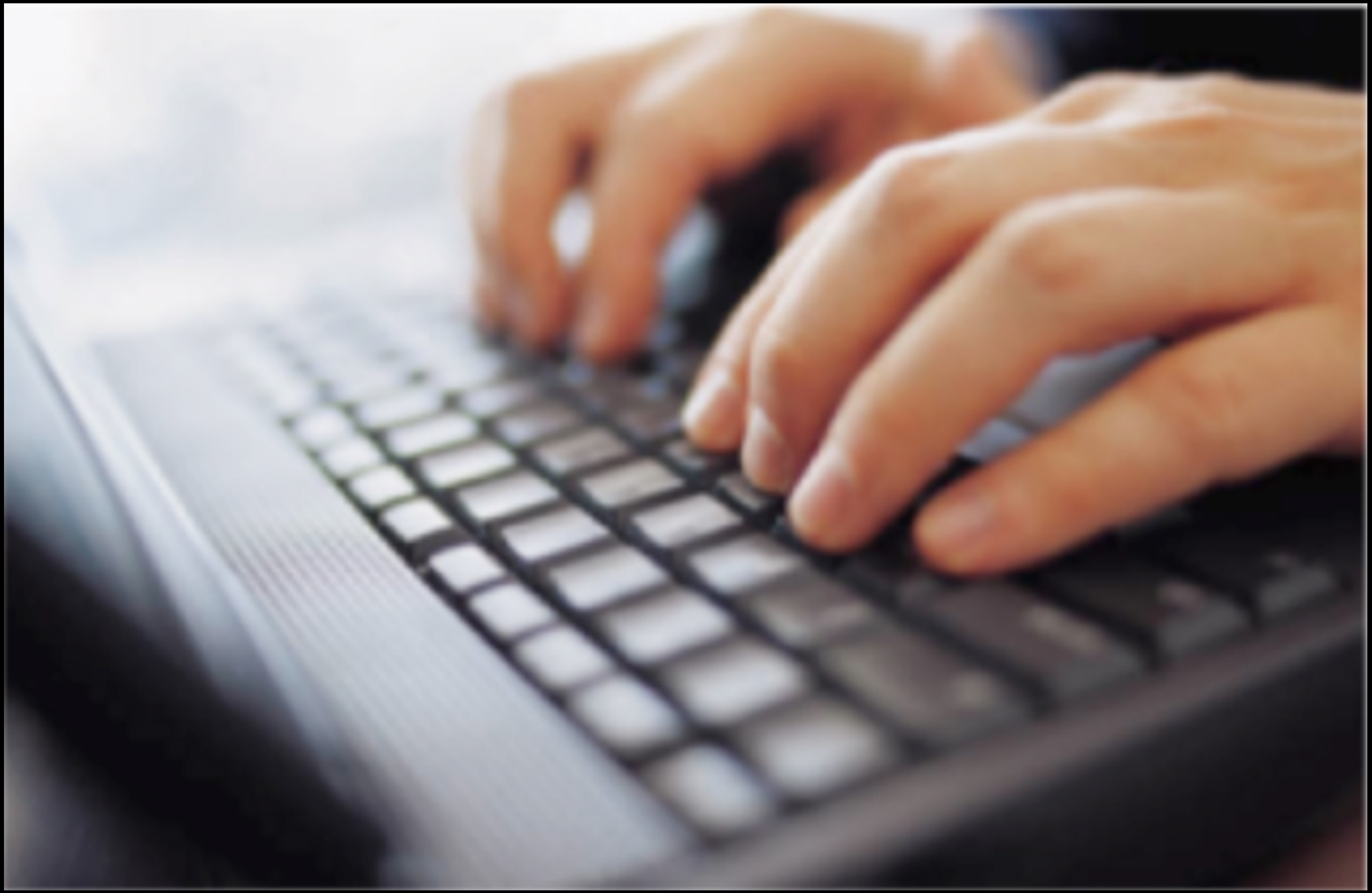
Tight integration:
A closure joinpoint is just a special joinpoint and is processed like any other joinpoint.

Will it blend?

Will it blend?

www.willitblend.com

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 { }
```

Will it type?

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

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

Checked Exceptions

```
jpi void JPNone();

jpi void JPEx() throws Exception;
```

```
before JPNone() throws Exception { }

before JPEx() throws Exception { }
```

Will it type?

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

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

Invariant Return Types

```
public aspect TestCase {

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

   jpi HashSet JP();


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

}
```

Invariant Return Types

```
public aspect TestCase {

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

  jpi HashSet JP();

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

}
```

Invariant Return Types

```
public aspect TestCase {

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

   jpi HashSet JP();

   Set 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();
  }

}
```

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();
```

```
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();
}
```

Prevent error by not matching!

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

```
exhibits 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());
}
```

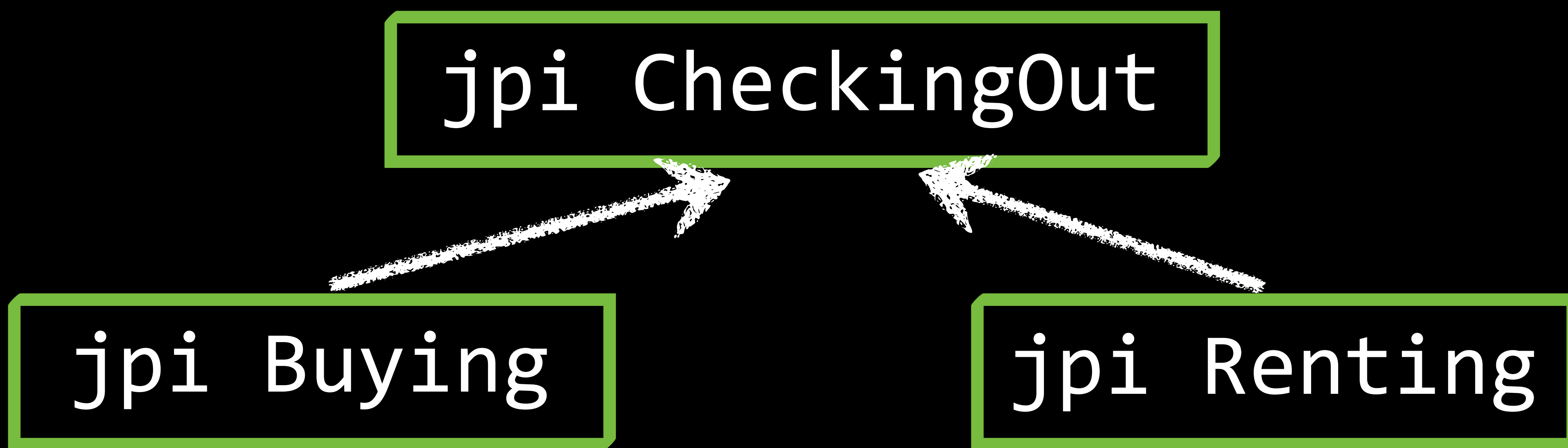
Result of typing rules

Strong typing!

Result of typing rules

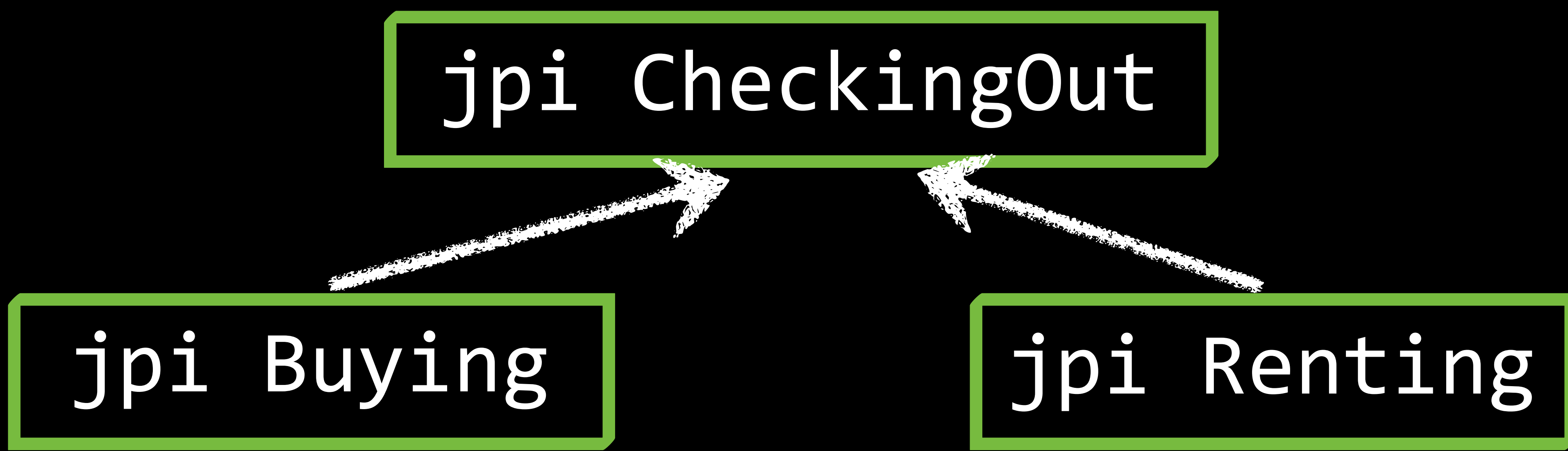
Invocation of a closure
can never fail
at runtime!

Join Point Polymorphism




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

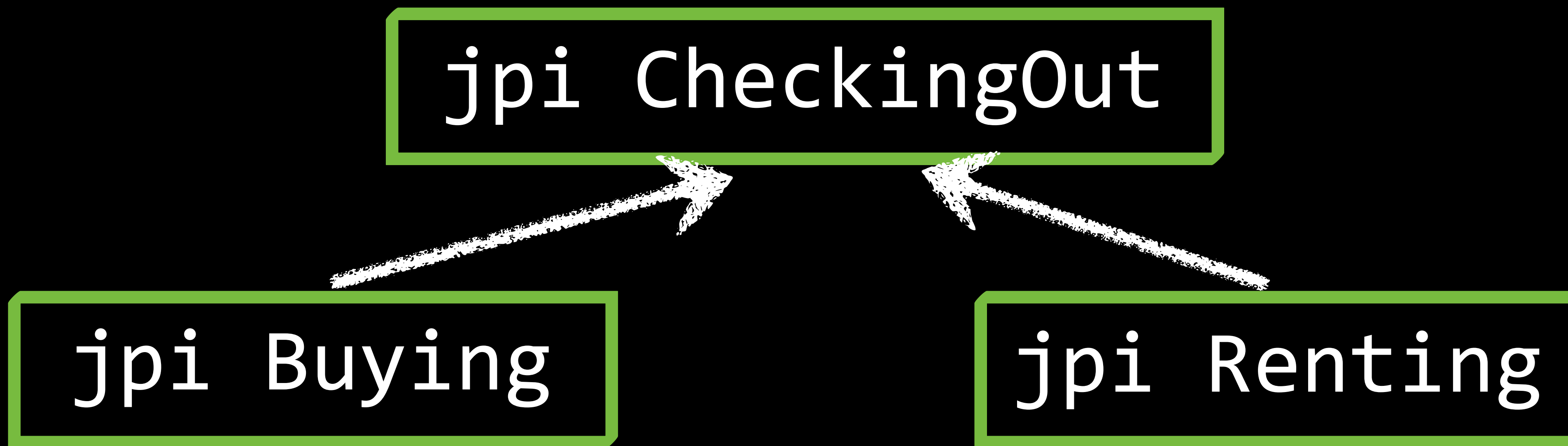
Join Point Polymorphism




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

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

Join Point Polymorphism

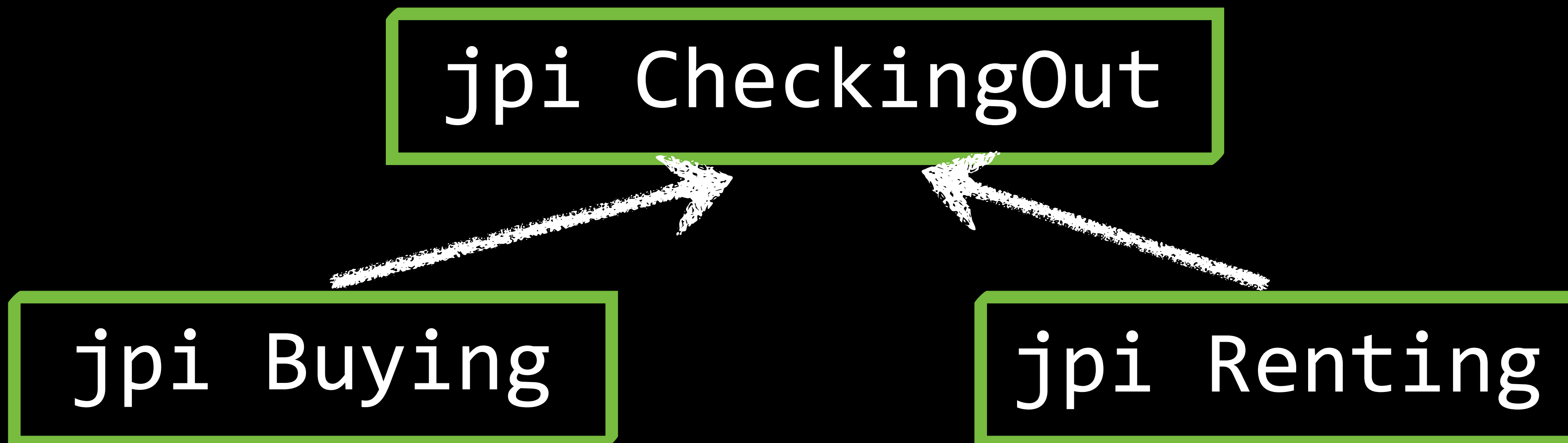



```
jpi void CheckingOut(float price, Customer c)
```

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

“width subtyping”

Join Point Polymorphism

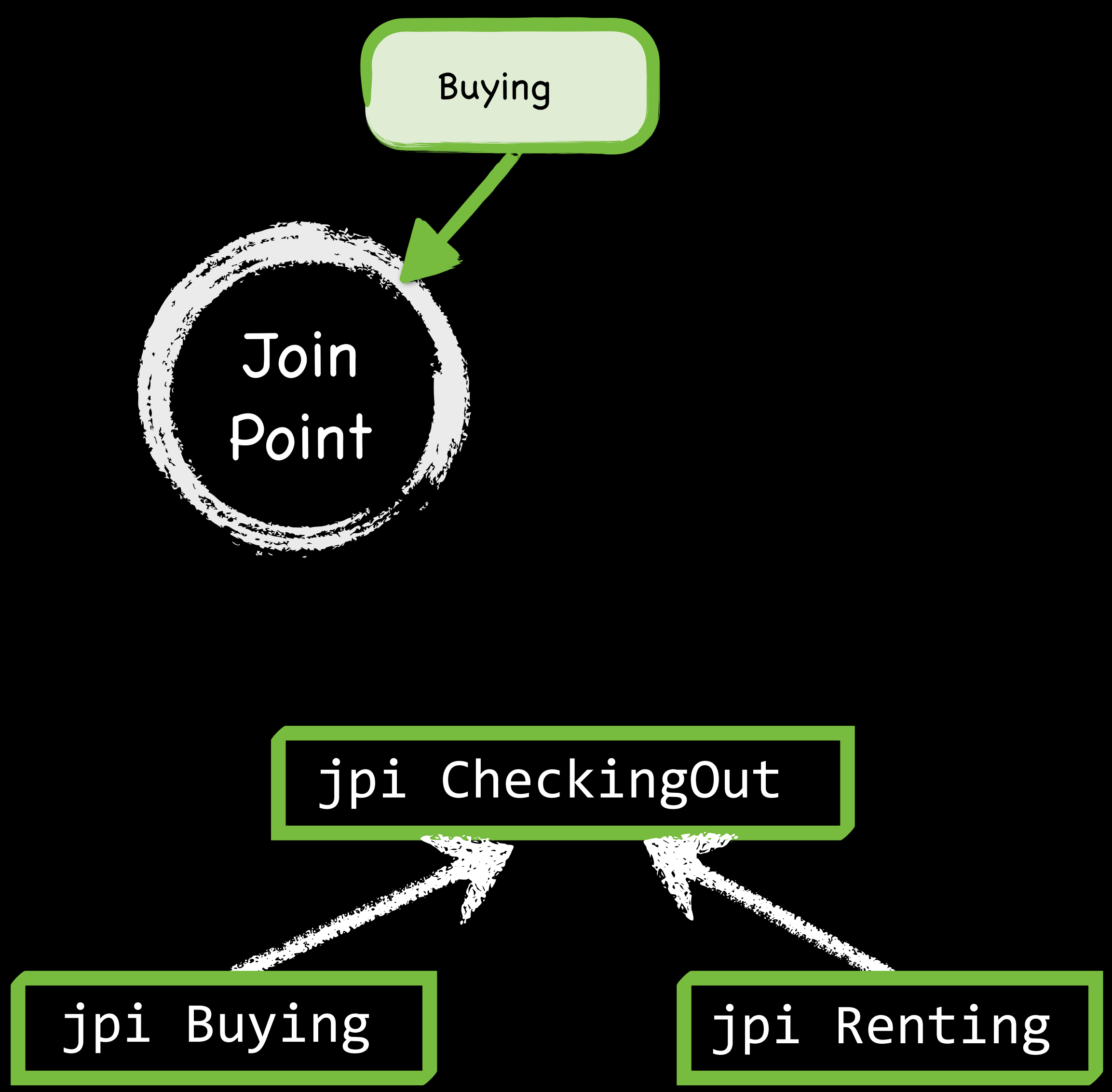



```
jpi void CheckingOut(float price, Customer c)
```

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

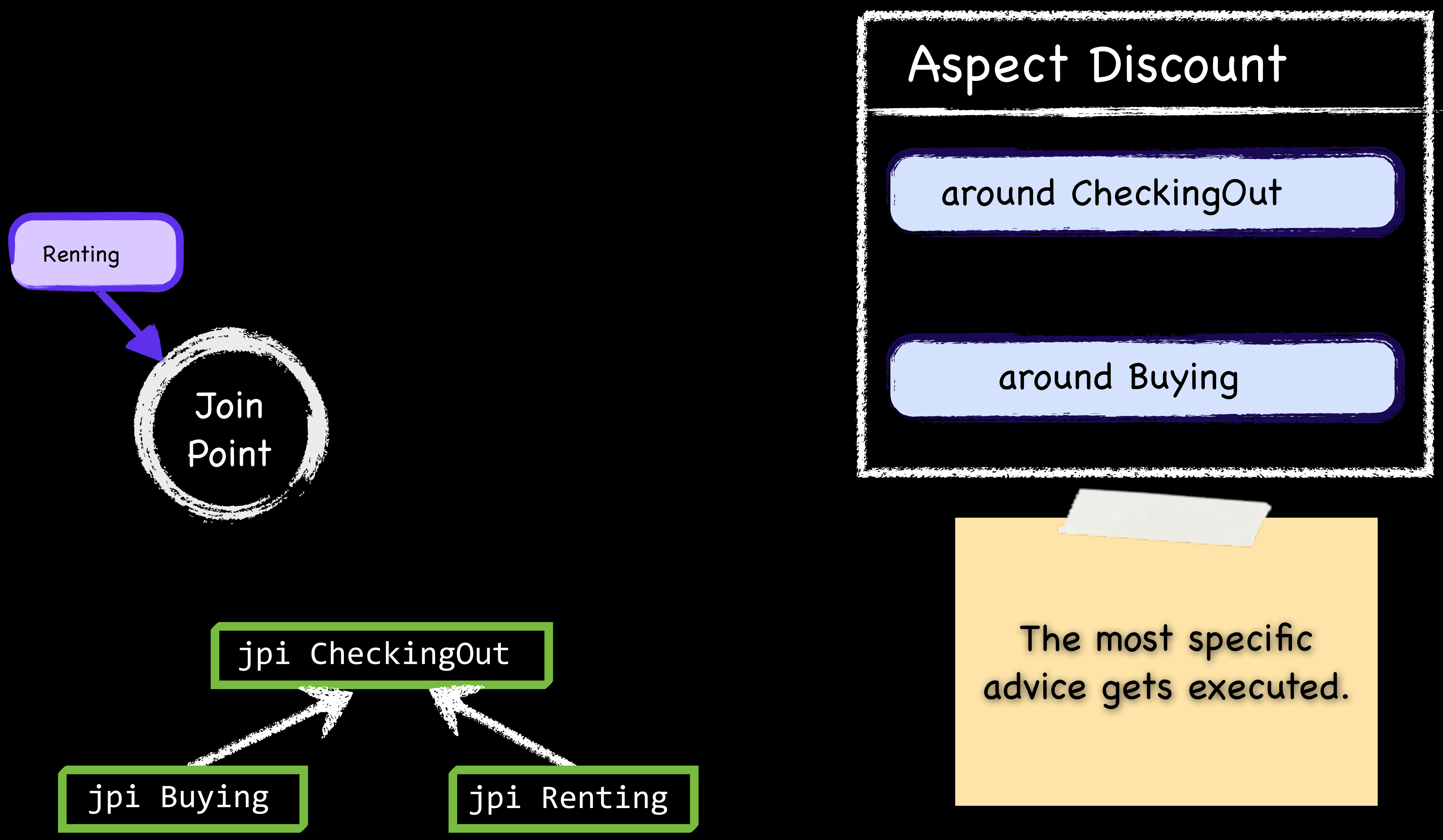
```
jpi void Renting(float price, int amt, Customer c)
extends CheckingOut(price,c);
```

Advice-dispatch semantics

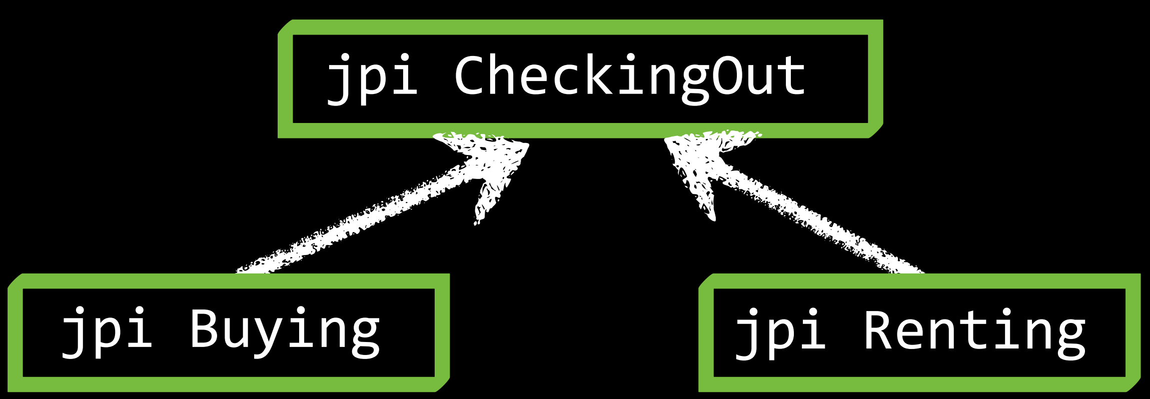


The most specific advice gets executed.

Advice-dispatch semantics



Advice-dispatch semantics



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

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

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

NO depth subtyping

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

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

Asp. A

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

NO depth subtyping

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

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

Asp. A

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

Asp. B

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

NO depth subtyping

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

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

Asp. A

```
void around CheckingOut(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
  - CJPs extracted into methods
  - Associate **correct** pointcut with each advice
- Resulting runtime overhead: **Zero!**

Thanks Milton!

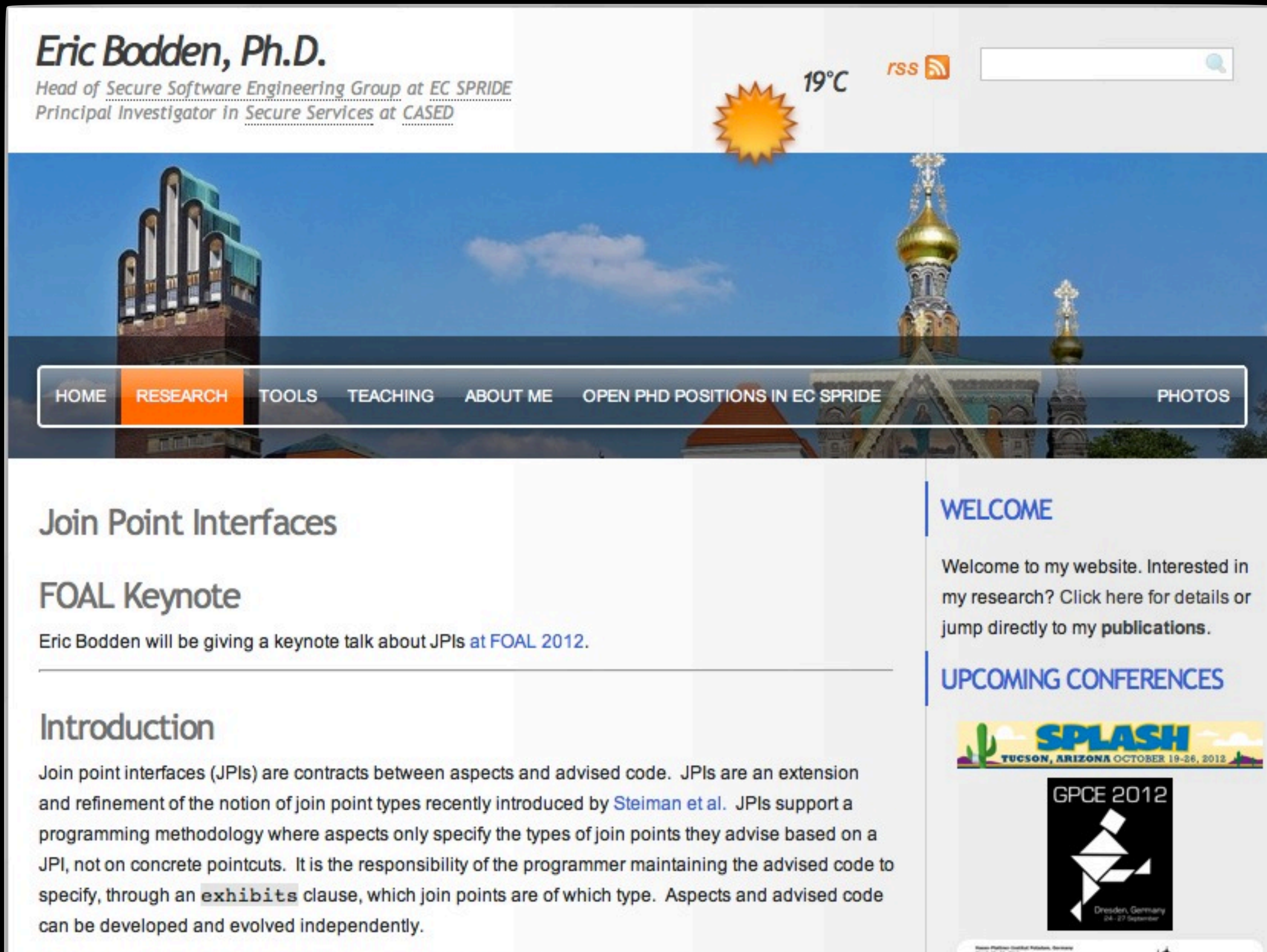
Closely Related Work

- 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

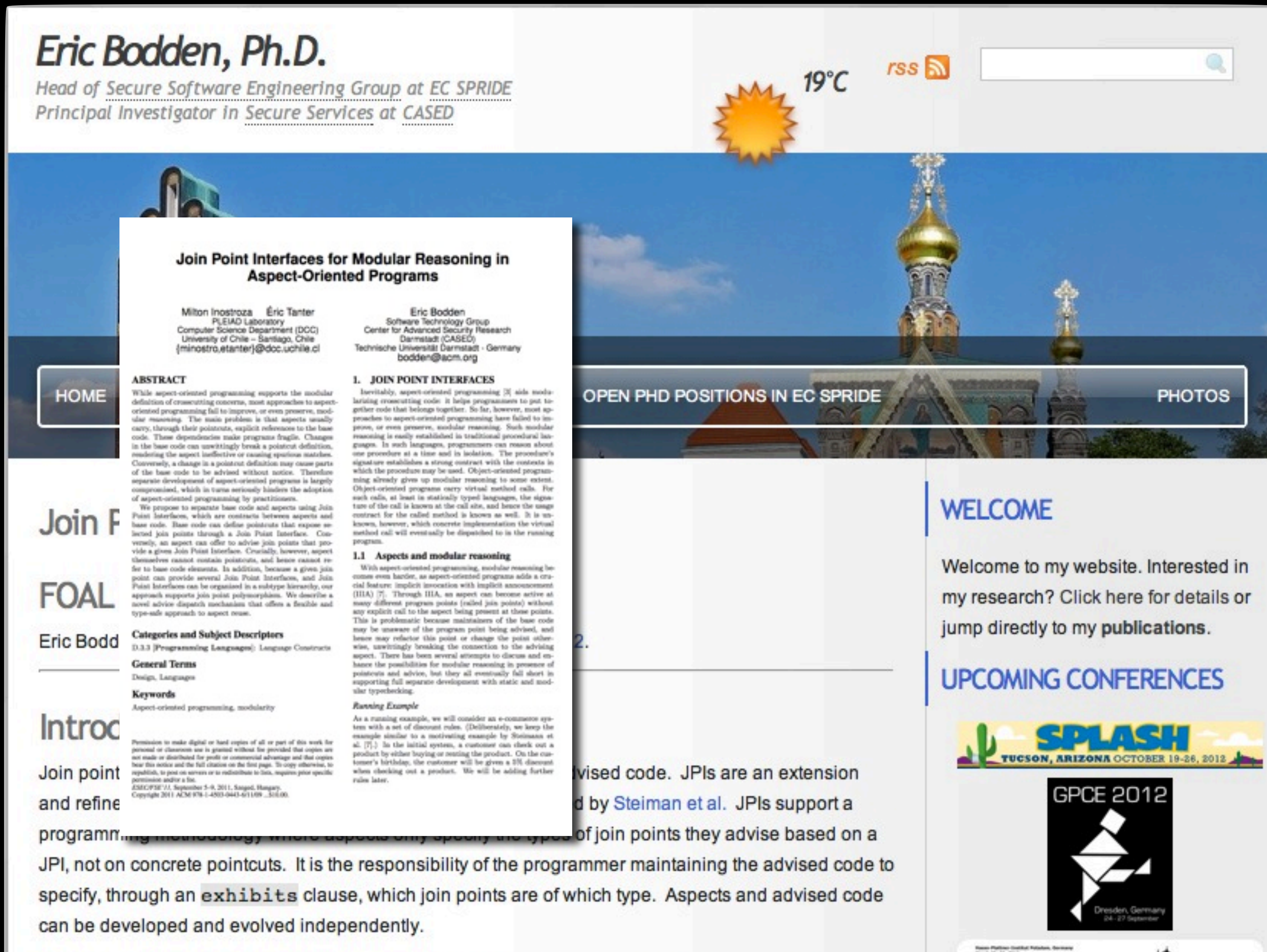
Evaluation

- 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

http://bodden.de/jpi



http://bodden.de/jpi

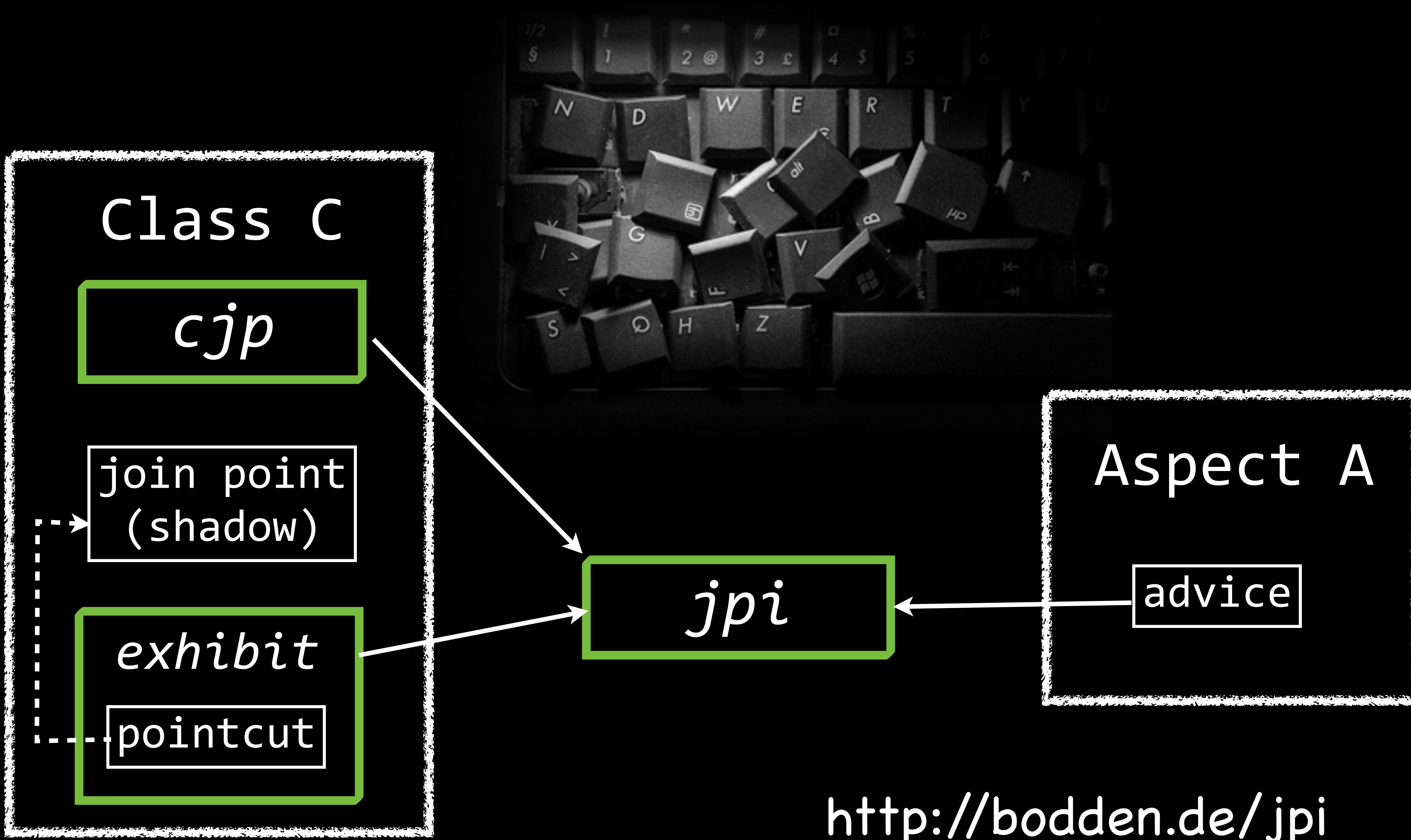


http://bodden.de/jpi

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



http://bodden.de/jpi

http://bodden.de/cjp