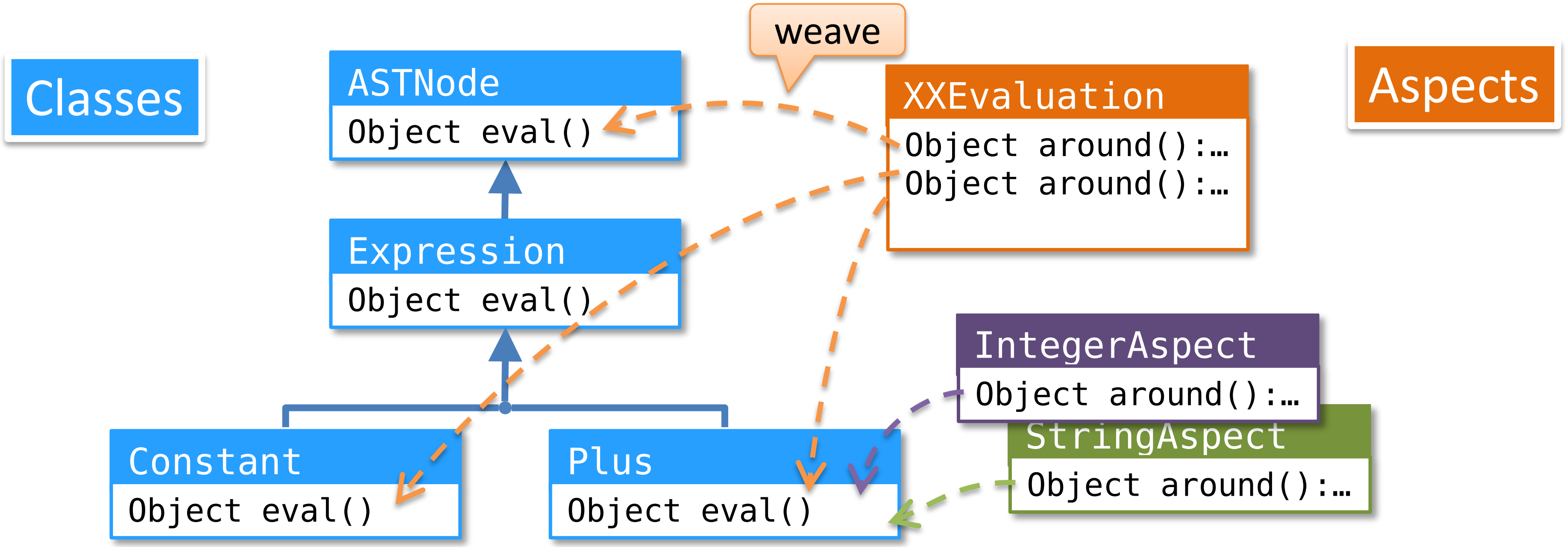
An Advice for Advice Composition in AspectJ

Fuminobu Takeyama
<u>Shigeru Chiba</u>

Tokyo Institute of Technology, Japan

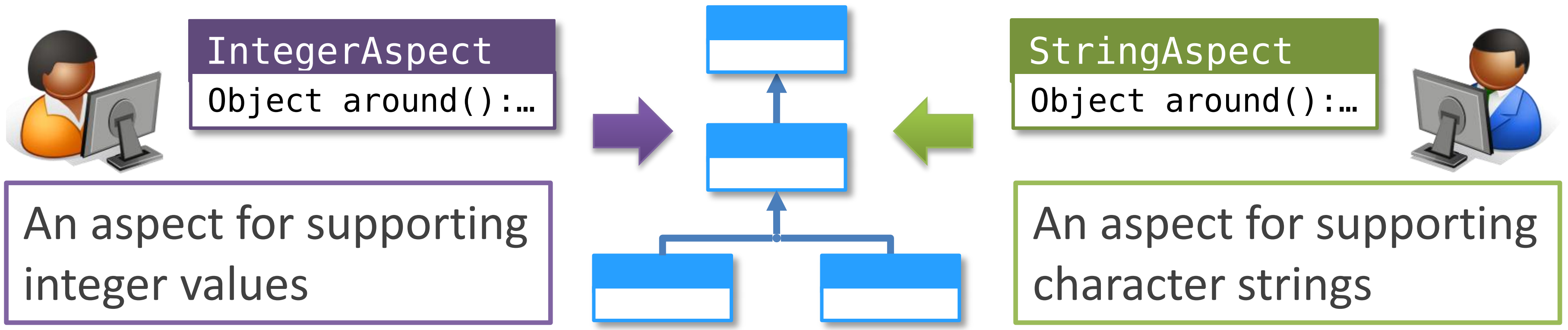
AOP: Aspect Oriented Programming

- Application = Classes + Aspects
  - ex) JastAddJ [T. Ekman, et al, OOPSLA 07]
    - Classes represents ASTs and aspects implement evaluation
  - Programmers can extend an application by reusing original one
    - No need to modify existing code



An aspect-oriented development scenario

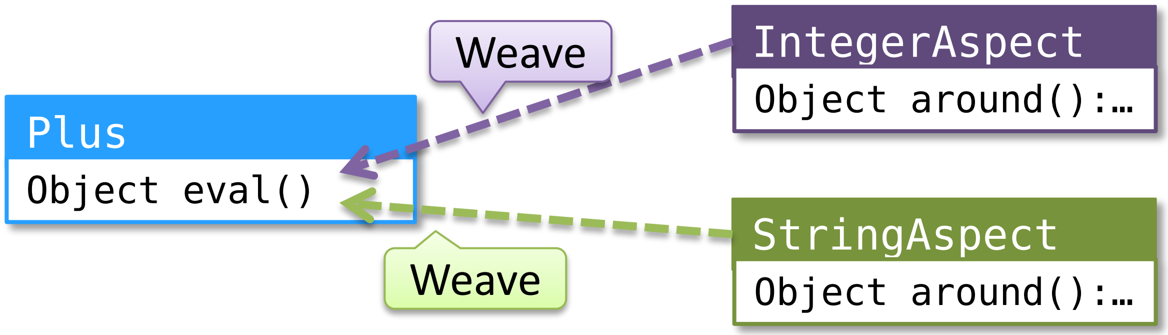
- 2 programmers extends the interpreter by aspects


IntegerAspect
Object around():…
An aspect for supporting integer values
StringAspect
Object around():…
An aspect for supporting character strings

- How can we get an interpreter supporting integers and strings?
  - Naive implementation of aspects above cause interference

Aspect interference: a crucial issue in AOP

- Conflict may cause aspect interference
  - Aspects show unexpected behaviour
    - even if each aspects are correct
  - Conflict: multiple advices are woven into the same join point
- “AOP is useless because aspects conflict”



Existing approaches

- Precedence rule does not work

- Considering composition when writing aspects
  - Programmers must design each advices so that they works with other advices
    - proceed calls executes the advice that has next precedence
  - Other advices might be unknown
    - The author of IntegerAspect does not know StringAspect

Composition-aware code

- Composition code scatters over aspects

IntegerAspect

```
aspect IntegerAspect {
  Object around(Plus t): target(t) && execution(Object Plus.eval()) {
    Object left = t.getLeft().eval();
    Object right = t.getLeft().eval();
    if (left instanceof Integer && right instanceof Integer) {
      return (Integer)left + (Integer)right;
    } else {
      return proceed(t);
}}}
```

composition code!

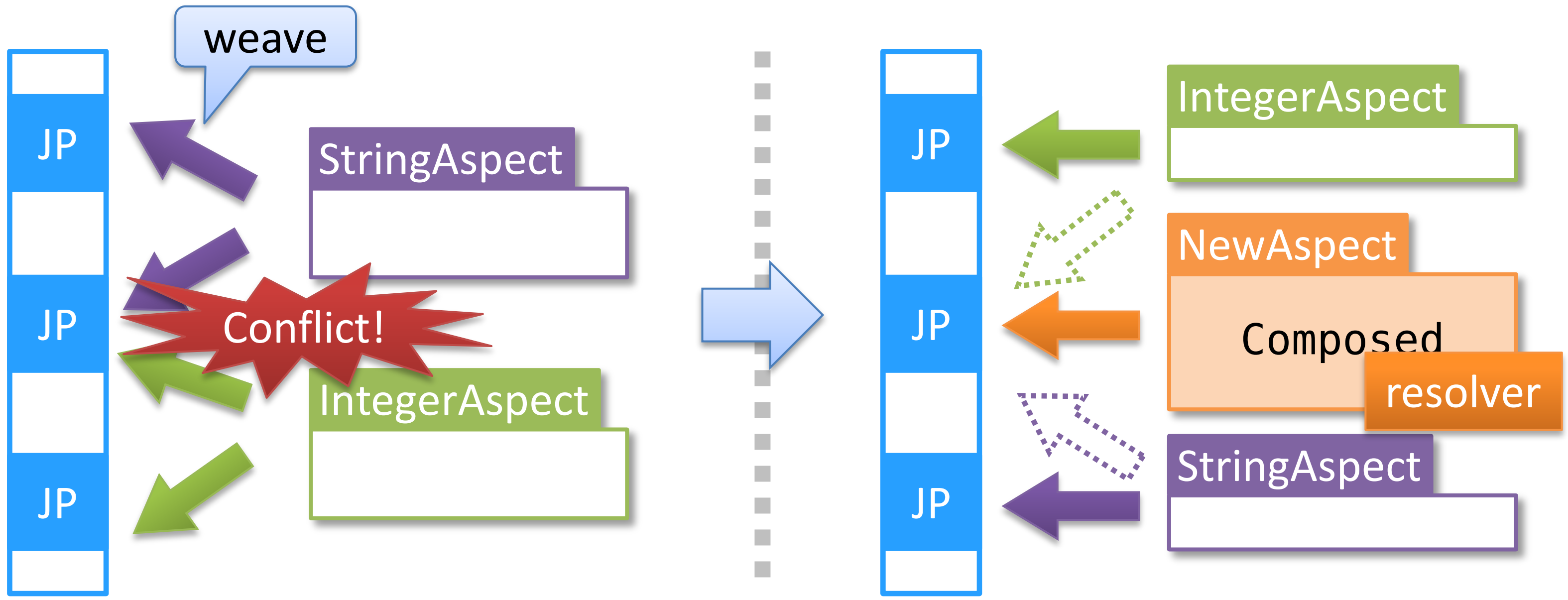
StringAspect

```
aspect StringAspect {
  Object around(Plus t): target(t) && execution(Object Plus.eval()) {
    Object left = t.getLeft().eval();
    Object right = t.getLeft().eval();
    if (left instanceof String || right instanceof String) {
      return left.toString() + right.toString();
    } else {
      return proceed(t);
}}}
```

composition code!

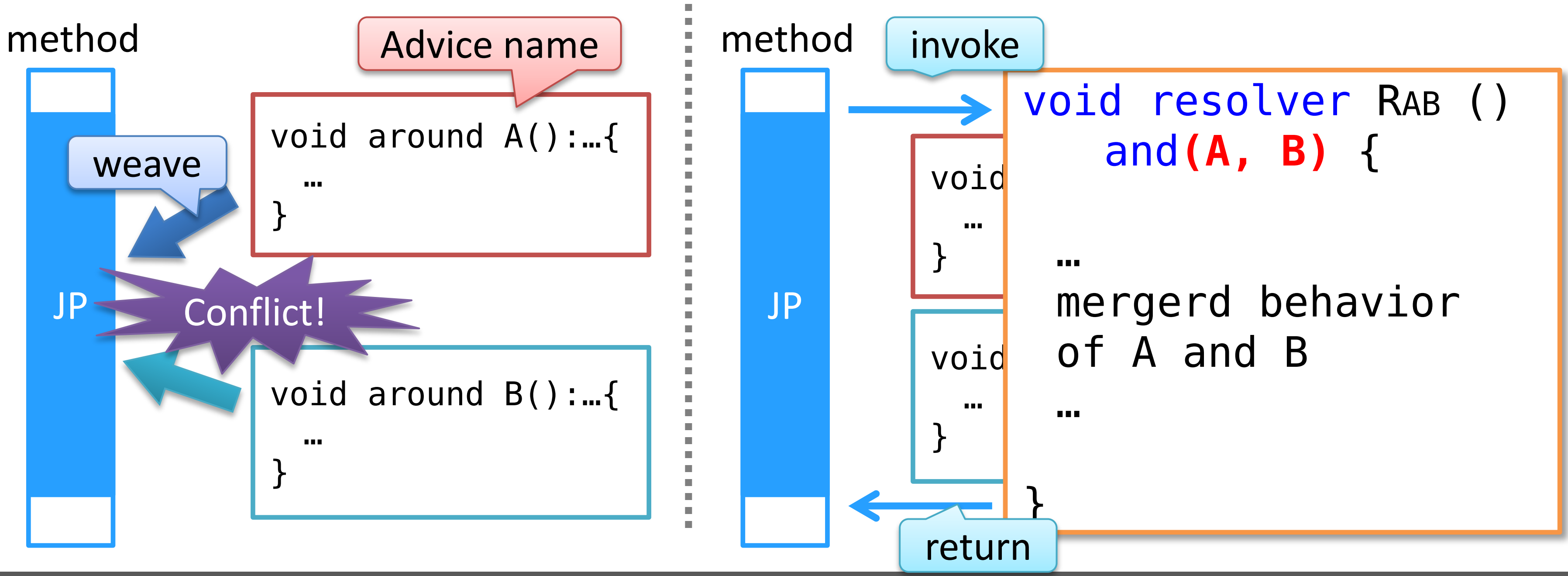
Airia: an extension of AspectJ

- Describe composed behaviour as a special case by a resolver
  - A resolver is executed only when advices conflict
  - Manually implemented



Resolver

- A resolver selects join points by and/or clause
  - and: when all the specified advices are woven
  - or: when one of specified advices is woven

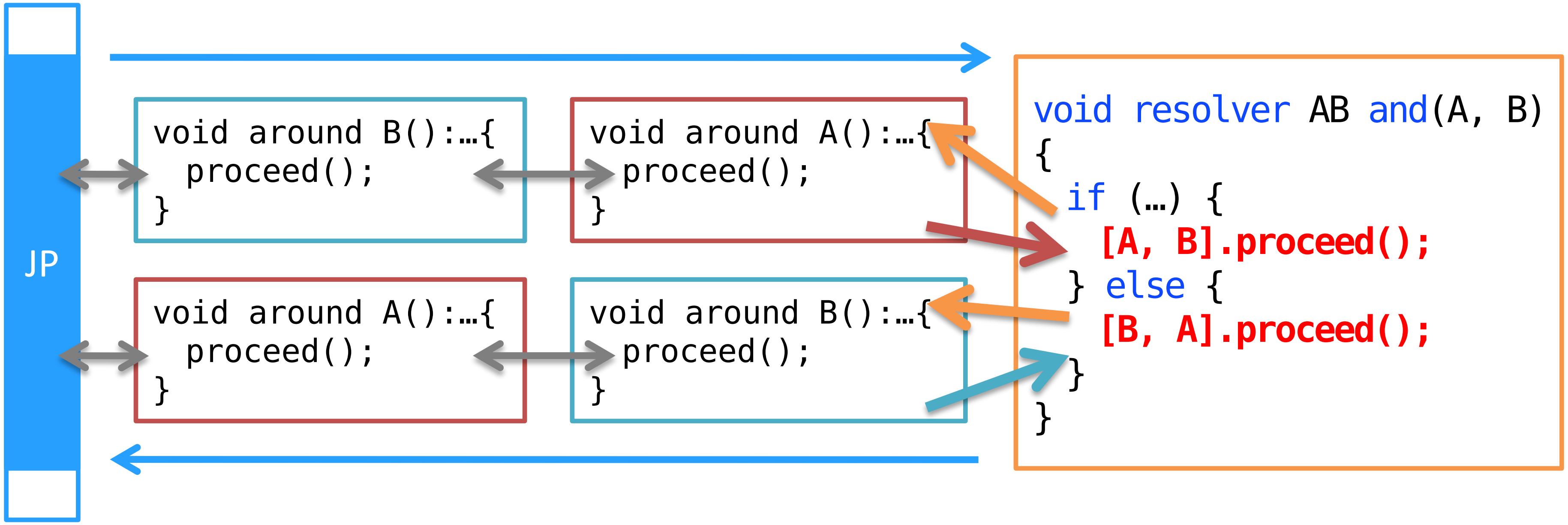


```
void resolver RAB ()
    and(A, B) {

  …
  mergerd behavior
  of A and B
  …
}
```

Proceed call with precedence 1/2

- Resolver can reuse existing advices by proceed call
  - The advice with the next highest precedence is invoked
- Can change precedence depending on dynamic context
  - The advice invoked by proceed call changes



```
void around B():…{
  proceed();
}
```

```
void around A():…{
  proceed();
}
```

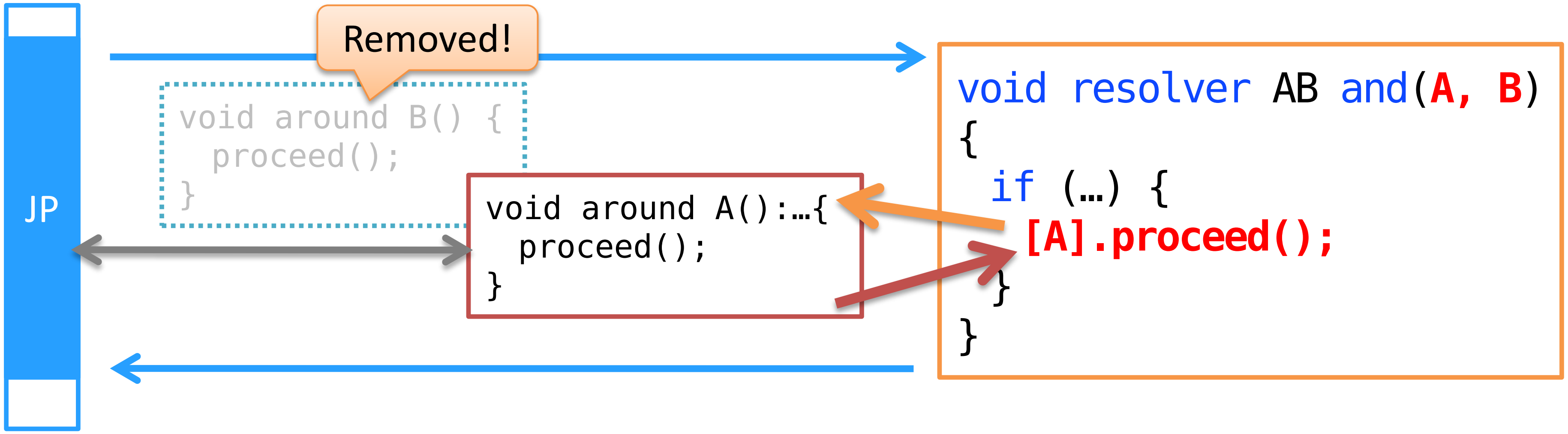
```
void around A():…{
  proceed();
}
```

```
void around B():…{
  proceed();
}
```

```
void resolver AB and(A, B)
{
  if (…) {
    [A, B].proceed();
  } else {
    [B, A].proceed();
  }
}
```

Proceed call with precedence 2/2

- Can also remove advices from remaining advices
  - To overwrite existing advices
  - The advices given in and/or clause but not on proceed call are removed



```
void around B() {
  proceed();
}
```

```
void around A():…{
  proceed();
}
```

```
void resolver AB and(A, B)
{
  if (…) {
   [A].proceed();
  }
}
```

Precedence relation is simple in Airia

- Only 2 precedence declaration mechanisms in Airia
  1. A resolver has higher precedence than advices in and/or

     ```
     void resolver A(): and(B, C, D) {…}
     ```

     →A precedes B, C, and D

  2. A proceed call with precedence

     ```
     [B, C, D].proceed();
     ```

     →B precedes C and C precedes D

  - Airia does NOT support `declare precedence` etc.

Aspects are free from composition code

- IntegerAspect and StringAspect in Airia

IntegerAspect

Advice name

```
aspect IntegerAspect {
  Object plusEvalInt around(Plus t):
       target(t) && execution(Object Plus.eval()) {

     return (Integer)t.getLeft().eval() + (Integer)t.getRight().eval();
  }
}
```

StringAspect

```
aspect StringAspect {
  Object plusEvalStr around(Plus t):
       target(t) && execution(Object Plus.eval()) {

     return t.getLeft().eval().toString() + t.getRight().eval().toString();
  }
}
```

Complex composition code is here

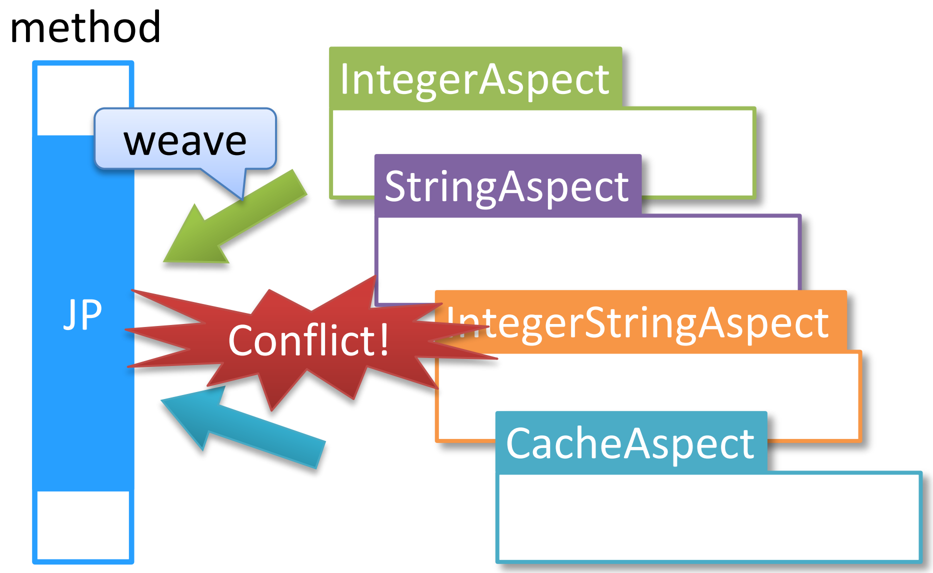
- The composition is defined in `IntegerStringAspect`

IntegerStringAspect

```
aspect IntegerStringAspect {
  Object resolver plusEvalIntStr(Plus t)
        and(IntegerAspect.plusEvalInt(t), StringAspect.plusEvalStr) {
    Object lvalue = t.getLeft().eval();
    Object rvalue = t.getRight().eval();
    if (lvalue instanceof String && rvalue instanceof Integer ||
        lvalue instanceof String && rvalue instanceof String) {
      return [StringAspect.plusEvalStr].proceed(t);
    } else if (lvalue instanceof Integer && rvalue instanceof Integer) {
      return [IntegerAspect.plusEvalInt].proceed(t);
    } else {
      throw new RuntimeException();
}}}
```

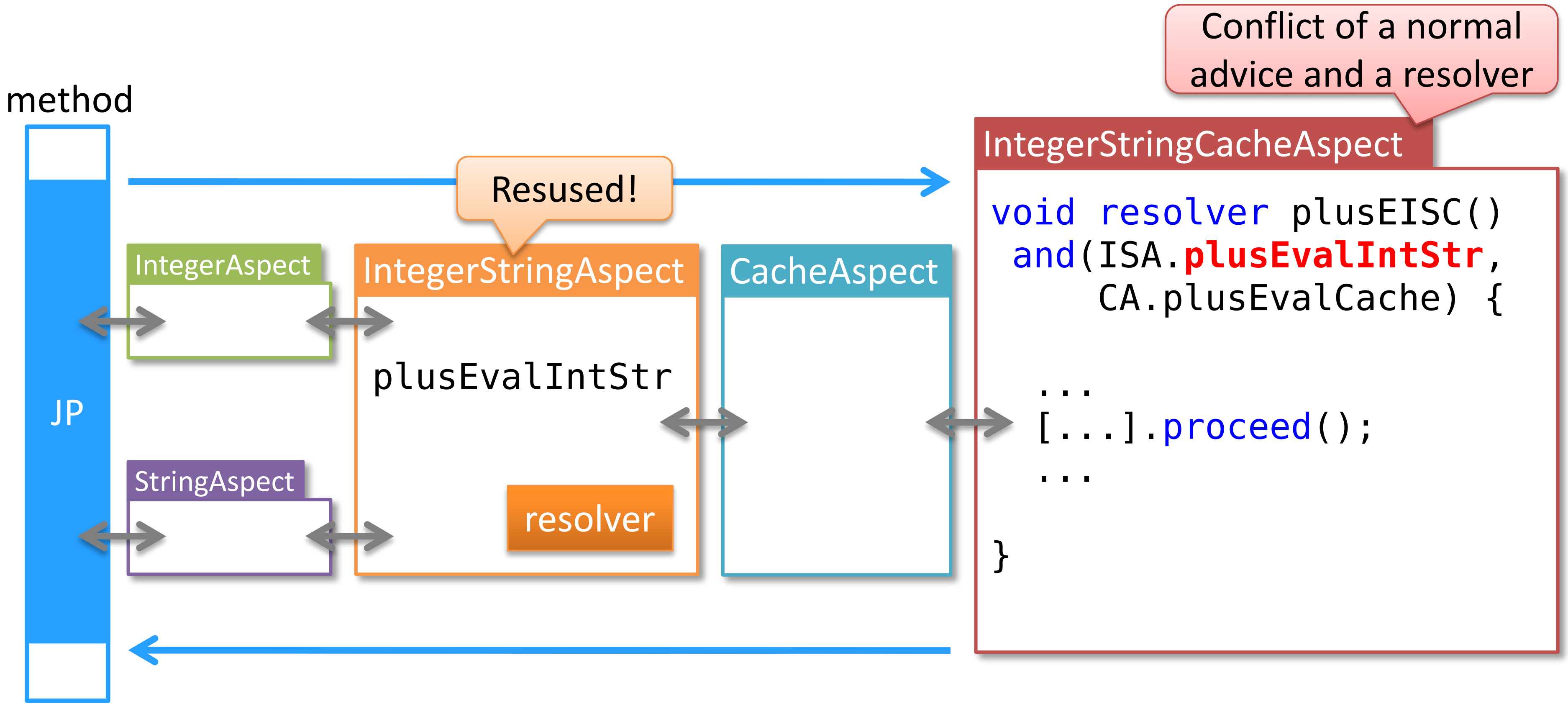
Resolvers are composable 1/2

- A resolver can resolve a conflict among resolvers and advices
- Let's consider new advice in `CacheAspect`
  - Conflicts with `IntegerStringAspect`



Resolvers are composable 2/2

- The existing resolver and advices can be reused
  - Using a proceed call with precedence



```
void resolver plusEISC()
 and(ISA.plusEvalIntStr,
     CA.plusEvalCache) {

    ...
    [...].proceed();
    ...

}
```

Compile time check of conflict resolution 1/2

- All conflict among advices must be resolved by resolvers
  - However there is some runtime factors
- Limitation for enabling compile time check
  - Static conflict: overlap of shadow
  - Our checking algorithm is conservative
    - All control statements such as `if` are ignored
    - All possible control path

Compile time check of conflict resolution 2/2

- OK

```
void resolver R() and(A, C, S) {
   if (flag) {
       [S, C, A].proceed();
   }
}
void resolver S() and(A, B) {
   [A, B].proceed();
}
```

1. R because R < S
2. S because S < C
3. C because C < A
4. A because A < B
5. B

- Compile error

```
void resolver R() and(C, S) {
   if (flag) {
       [S, C].proceed();
   }
}
void resolver S() and(A, B) {
   [A, B].proceed();
}
```

- R < S
- However there is no precedence order between A and C

X < Y means X precedes Y

Ideas of Airia

- Aspects are free from composition code
  - Separating composition code into a resolver
- Resolvers are composable
  - Resolvers can be resolved in the same way with normal advices
- Precedence relation is checked statically

Related work

- Stateful aspect [R. Douence, et al, GPCE 02 & AOSD 04]
  - Programmers can define composition operator that explicitly replace conflict with composed behaviour
  - Airia provides new language constructs into AspectJ
- Context-Aware Composition Rules [A. Marot, et al, DSAL 08 and SPLAT 08]
  - provides an advice-like construct for specifying precedence at selected join points
  - Some composition of advices requires additional code

Related work

- Existing meta programming approaches for composition
  - POPART [T. Dinkelaker, et al, AOSD 09], JAsCo [D. Suvée, et al, AOSD 03]
    - Change precedence at runtime when conflict
  - OARTA [A. Marot, et al, AOSD 10]
    - Provides construct for modifying pointcuts of already defined advices
  - They does not support composition of composition code
- Traits [N. Schärli, et al, ECOOP 03]
  - resolved similar problem of mixin inheritance in OOP
  - Method conflicts must be explicitly resolved by overriding the methods
    - We are inspired by this idea

Conclusion

- Resolvers
- proceed calls with precedence
  - A resolver defines composed behaviour as a special case
  - Advices are free from composition code
- The Airia compiler is available
  - http://www.csg.is.titech.ac.jp/projects/airia/