An Aspect-Oriented Approach to implement JML Features

Henrique Rebêlo
Informatics Center
Federal University of Pernambuco

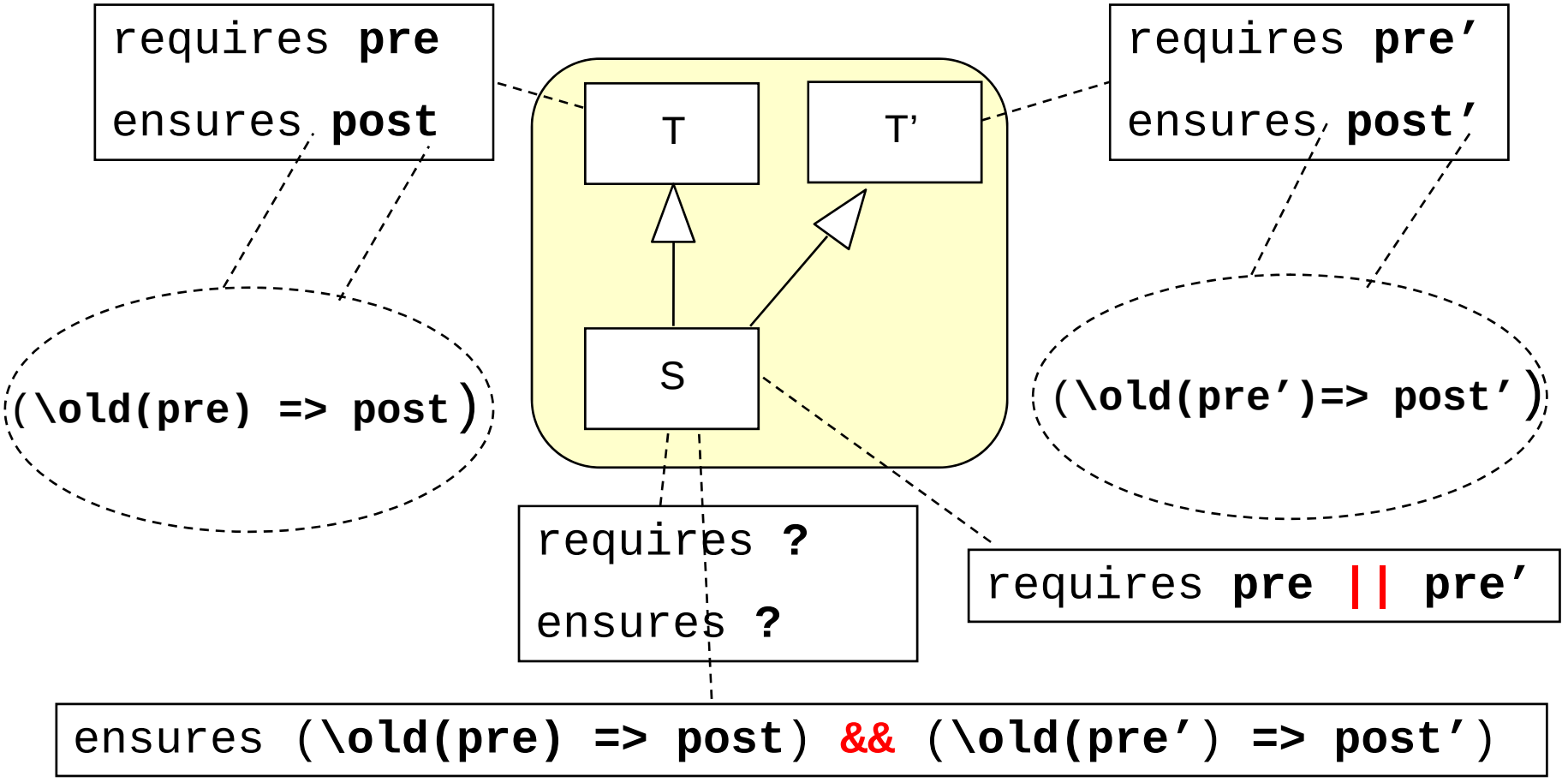
Summary

- jmlc problems
  - bigger code, slower code, no suppport for Java ME, and bad error messages
- Approach
  - aspect-oriented programming
- Contributions
  - smaller code, faster code, Java ME support, and better error messages

Java Modeling Language—JML

- Formal specification language for Java
  - behavioral specification of Java modules
  - Sequential Java
- Adopts design by contract based on Hoare-style with assertions
  - pre-, postconditions and invariants
- Main goal → Improve functional software correctness of Java programs

Specification Inheritance



Specification Inheritance

```
requires 0 <= a && a <= 150;
ensures age == a;
```

also

```
requires a < 0;
ensures age == \old(age);
```

means

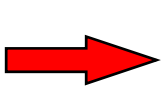
conjunction

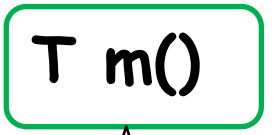
disjunction

```
requires (0 <= a && a <= 150) || a < 0 ;
ensures (\old(0 <= a && a <= 150) ==> age == a)
     && (\old(a < 0) ==> age == \old(age)) ;
```

Join of Specifications

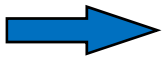
```
requires 0 <= a && a <= 150;
ensures age == a;
```

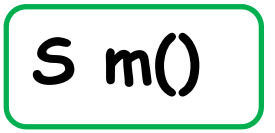




also

```
requires a < 0;
ensures age == \old(age);
```

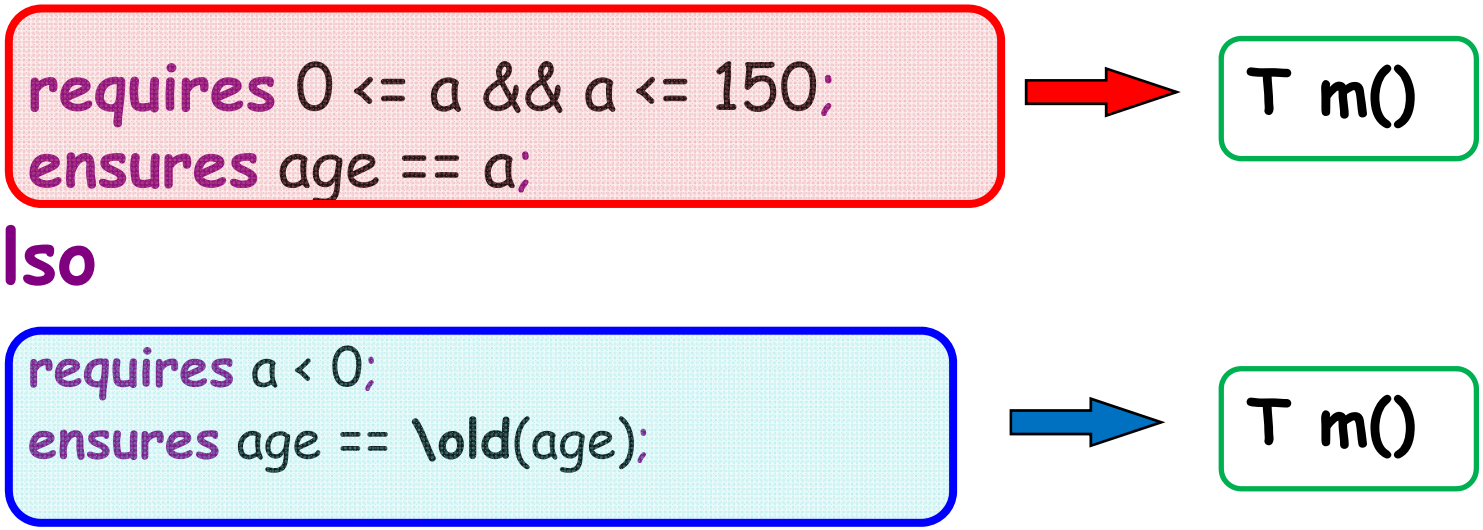




means

Specification inheritance

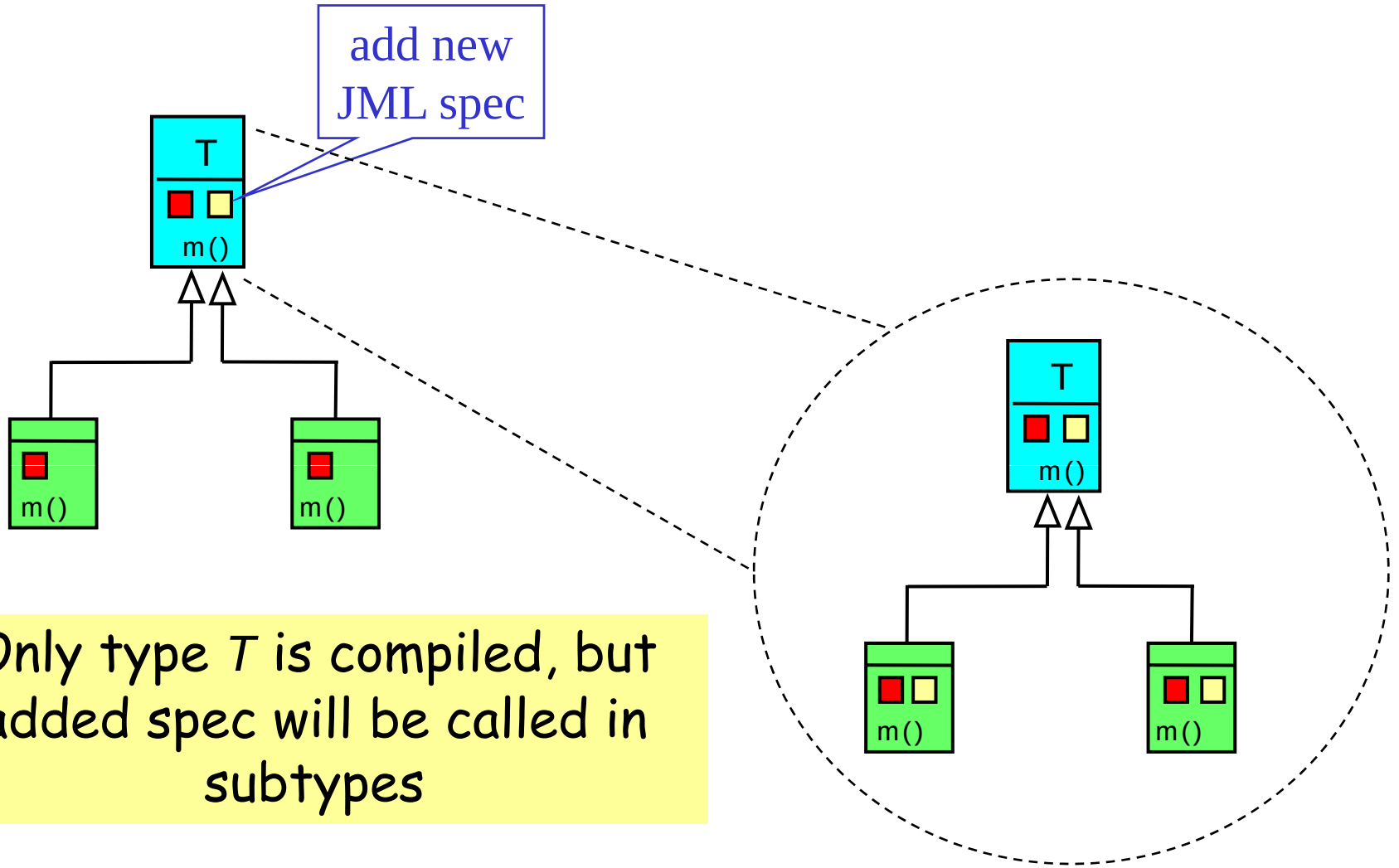
Join of Specifications



means

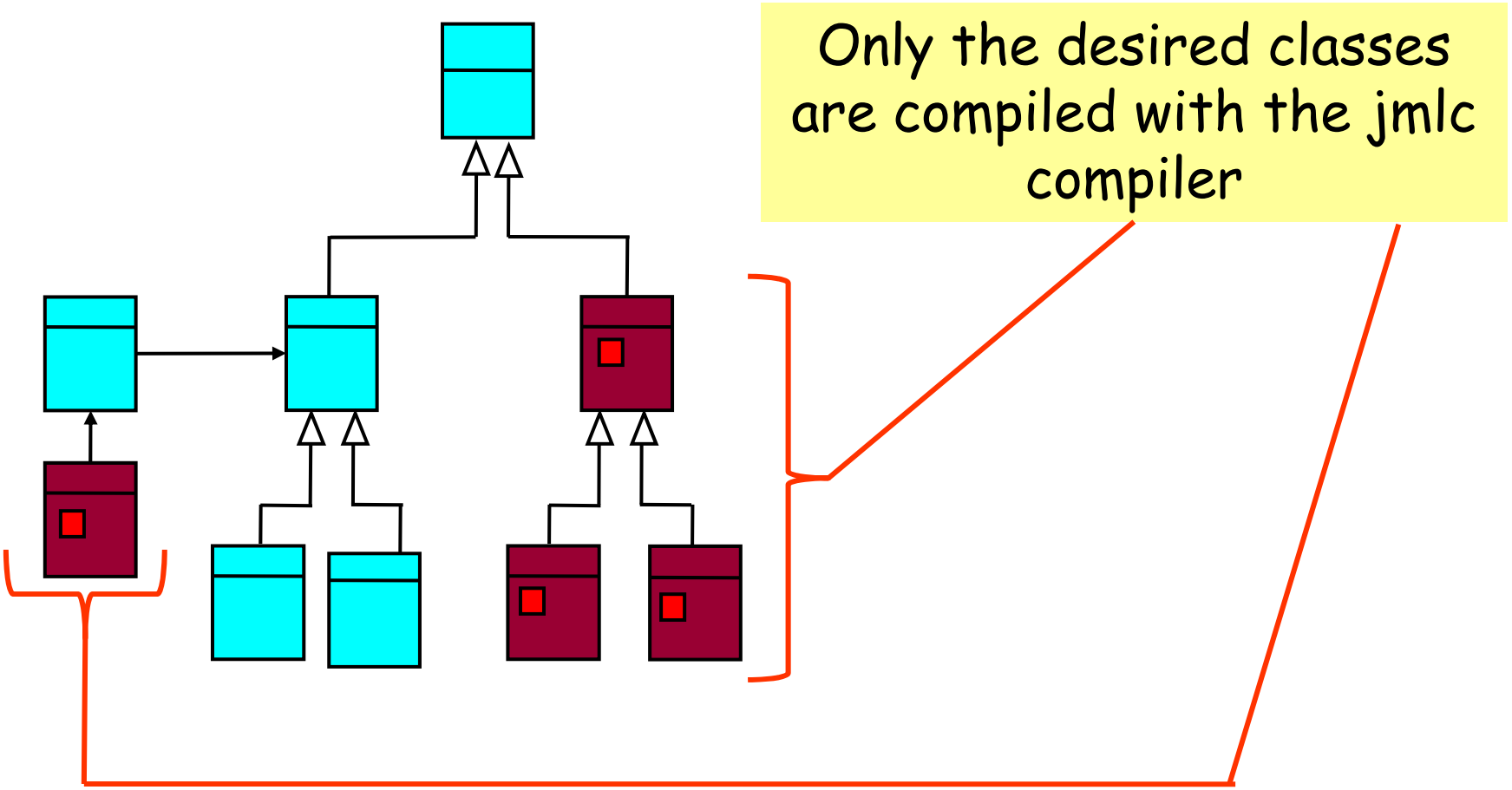
Specification cases

Modular compilation in jmlc



Only type *T* is compiled, but added spec will be called in subtypes

Separate compilation in jmlc



Tool Support

- Many tools, one language

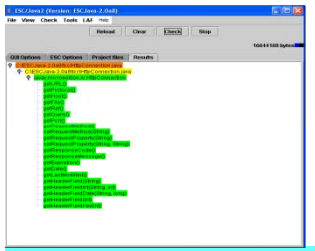
jmldoc

Web pages

JML Annotated Java

```
public class Animal implements Gendered {
  // ...
  protected /*@ spec_public @*/ int age = 0;
  /*@    requires 0 <= a && a <= 150;
    @    ensures age == a;
    @ also
    @    requires a < 0;
    @    ensures age == \old(age);   @*/
  public void setAge(final int a) {
    if (0 <= a) { age = a; }
  }
}
```

ESC/Java2



```
Bag.java:15: Warning: Possible null
dereference (Null)
      if (elements[i] < min) {
          ^
Bag.java:15: Warning: Array index
possibly too large (..
      if (elements[i] < min) {
```

Warnings

jmlunit

Unit tests

jmlc

Class file

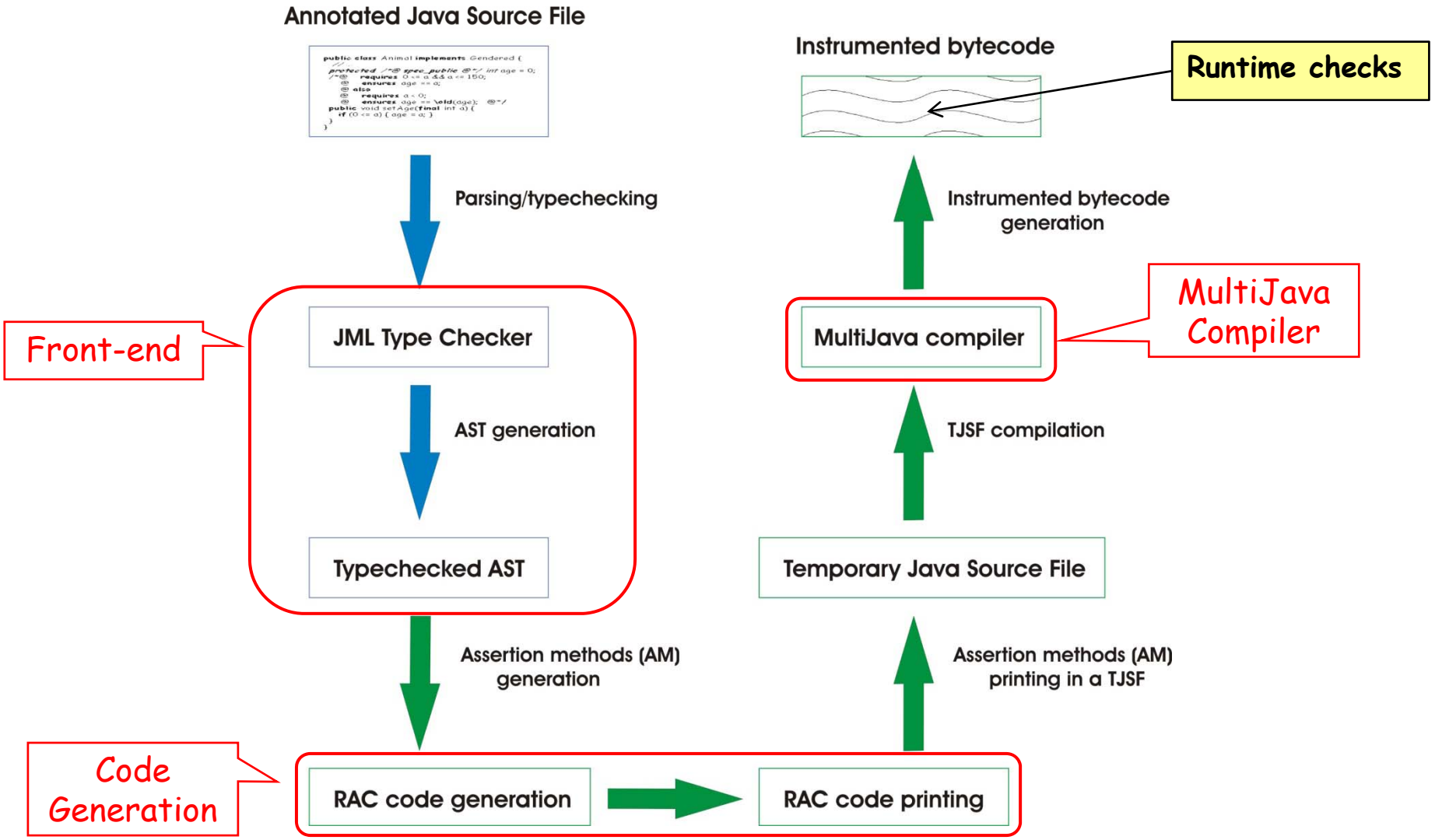
jmlrac

runtime assertion checker

JACK, Krakatoa, LOOP

Correctness proof

jmlc: compilation passes



Before understanding our approach with AOP…

we will look at the

problem

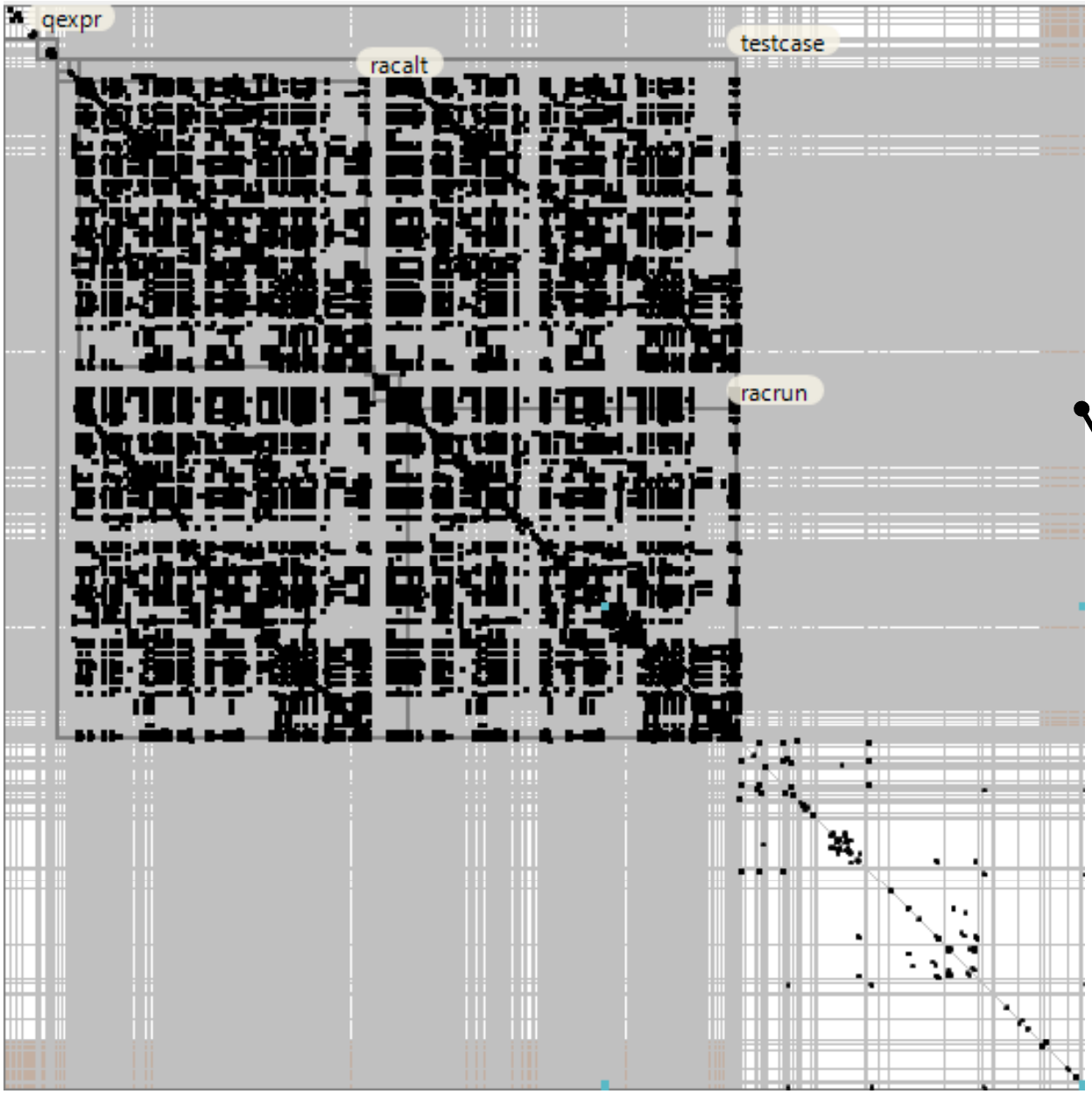
that motivated its use

The failure of JML (user perspective)

- The generated instrumented code is not complaint with Java ME
- The overhead in bytecode size is significant in relation to constrained environments
- Bad performance due to many reflective calls
- Dependence of the reflection API
- Wrong line numbers in error messages

More failures of JML… (implementor perspective)

- Many parts of the compiler code present too many clones
- Generation code is invaded by changes because the generated code is tangled (to the contract enforcement concern) and scattered
- Contract enforcement cannot be understood in isolation
  - reasoning about core class code and contract concern is difficult



Was detected approximately 40960 clone pairs in the jmlrac implementation

Source: CCFinder, 2003. http://www.ccfinder.net

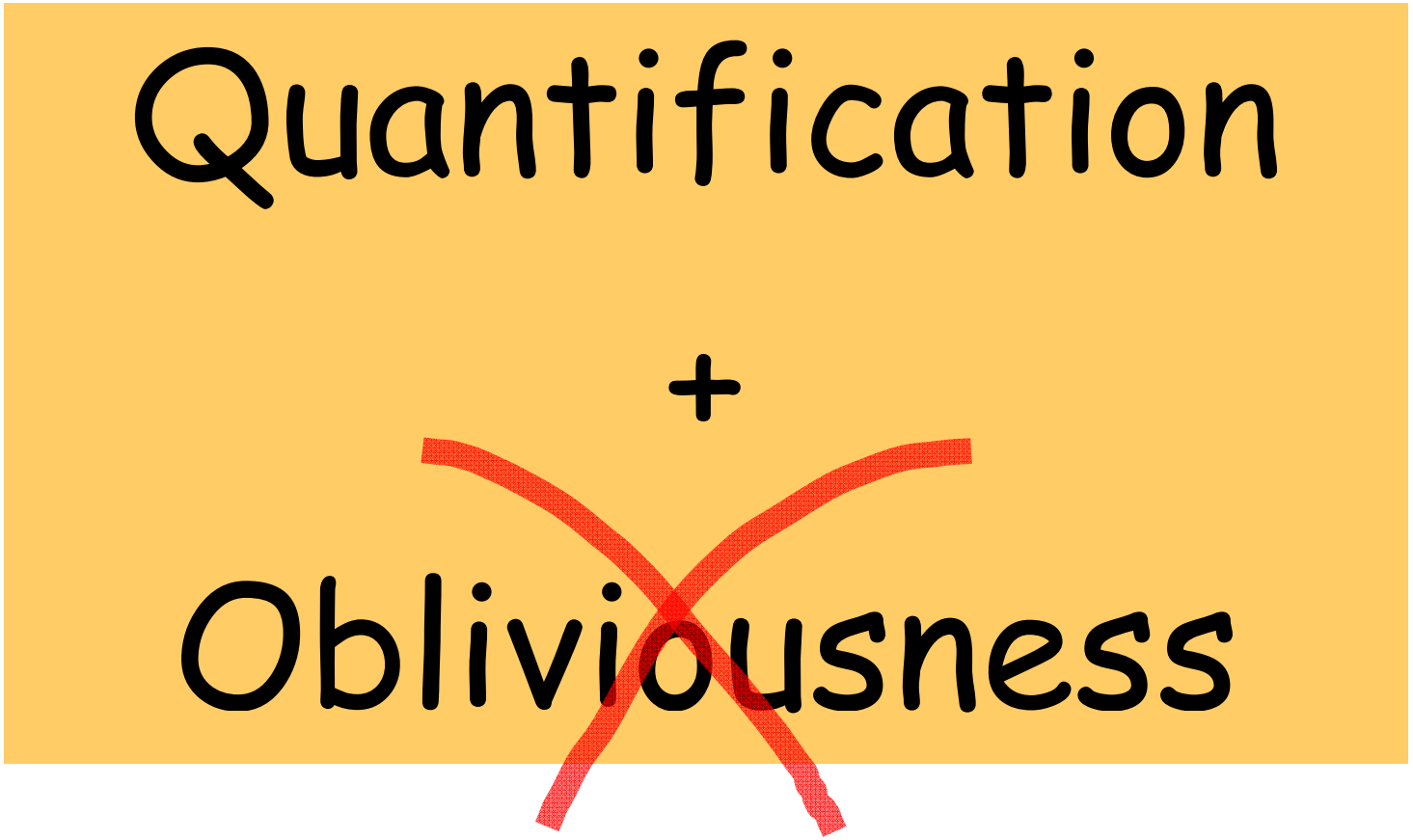
Aspect-oriented programming…

is the

solution

to the discussed problems of JML

AOP is not…



(Filman, Elrad, Clarke, and Aksit 2005)

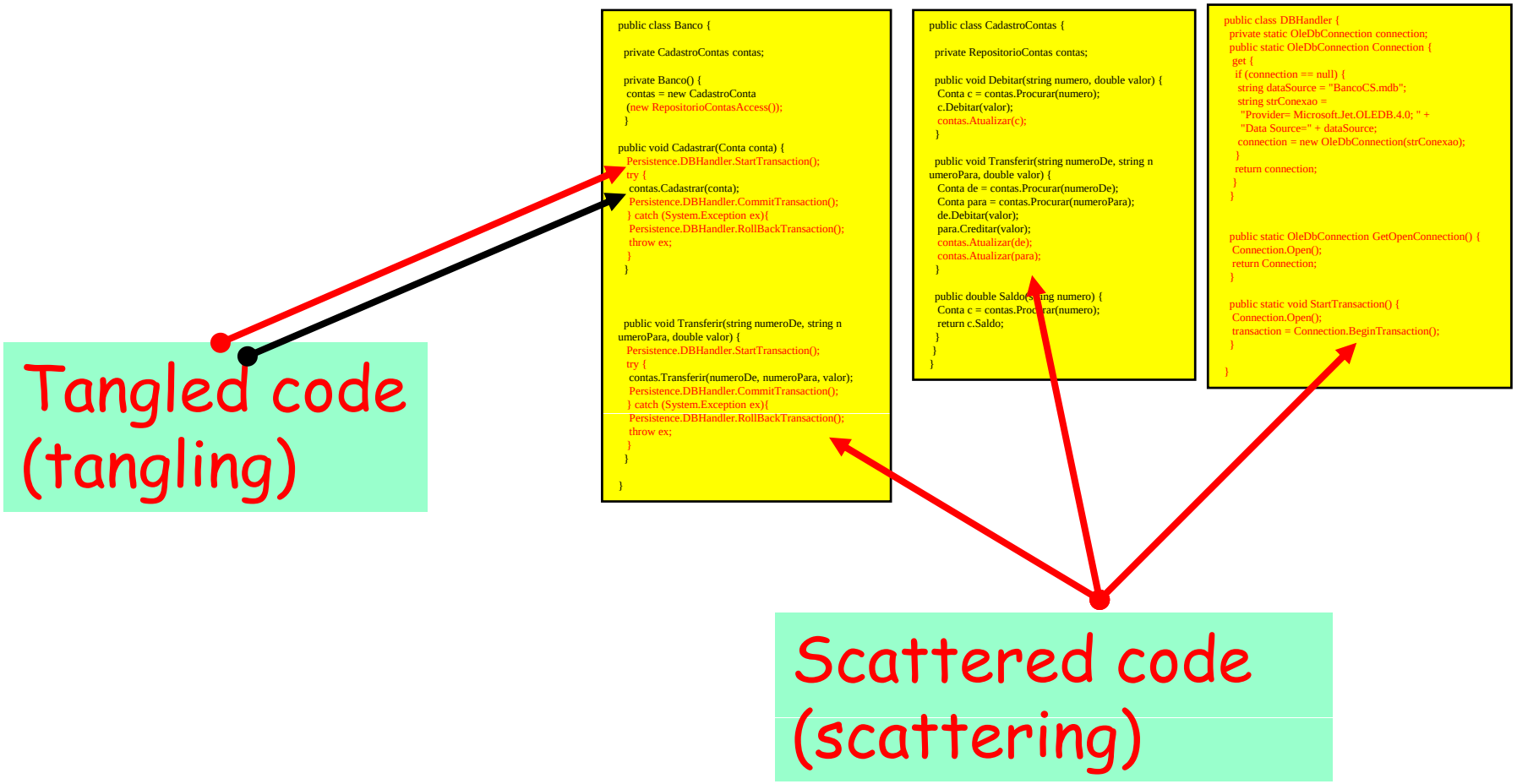
AOP is…

Quantification

+

Non invasiveness

Problems with OO implementation



The failure of OOP

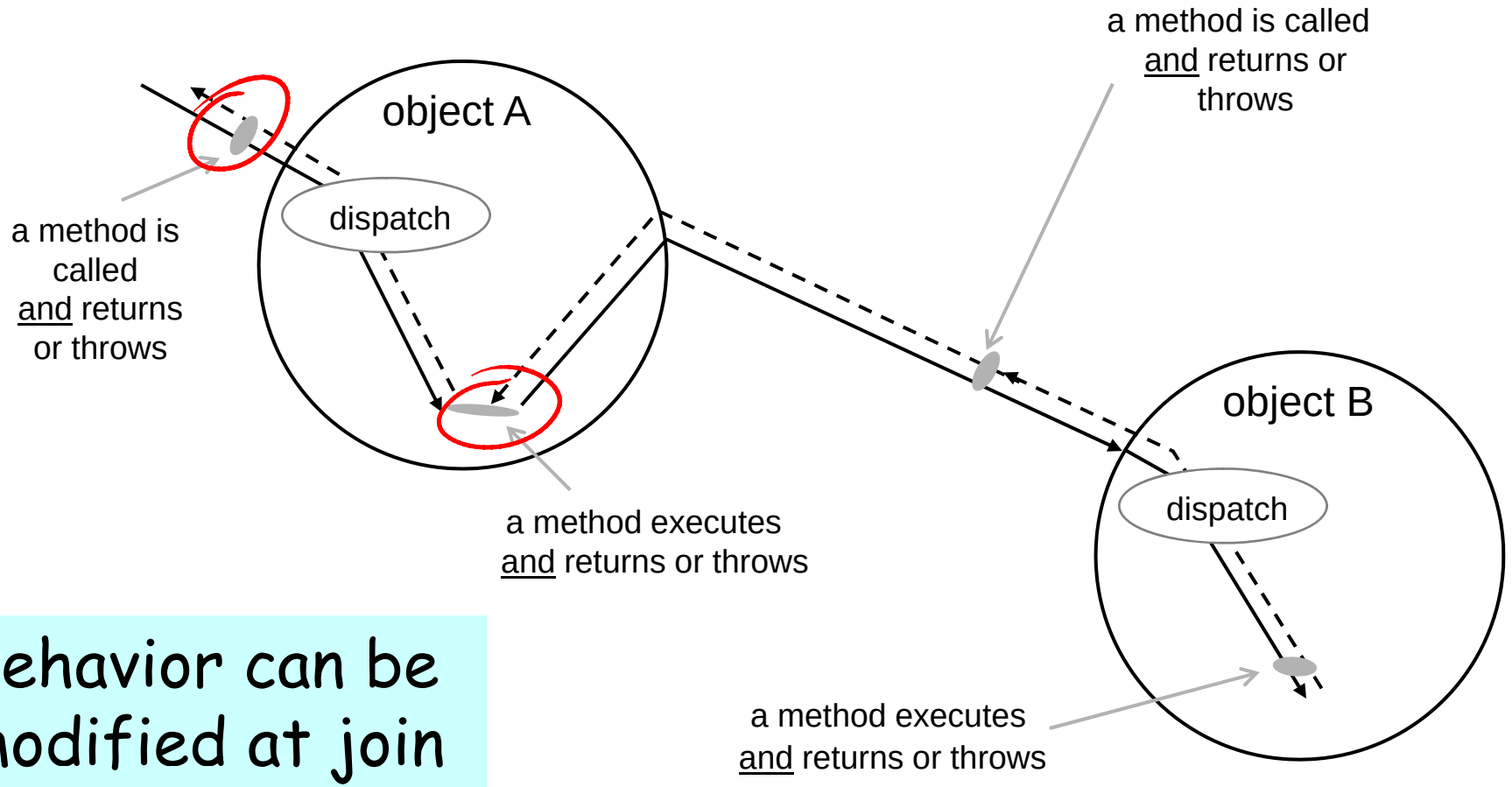
- Part of the assertion checking code is tangled (to the business concern) and scattered, cannot be reused
- Business code cannot be reused to work with other assertion checking APIs
- Business code is invaded by changes to assertion checking APIs
- Assertion checking policy cannot be understood in isolation

Tangling and scattering of…

Concerns

- Persistence
- Concurrency control
- Logging
- Business rules
- Performance
- Distribution
- Use cases
- Assertion checking
- …

Composition at join points



Behavior can be modified at join points…

Source: AspectJ Programming Guide

Advice in AspectJ provides extra behavior at join points

- Define additional code that should be executed...
  - `before`
  - `after`
    - `after returning`
    - `after throwing`
  - `or around`

  join points

Around-execution example

```
class T {

  void m() {
    pre
    body
    post
  }
}
```

=

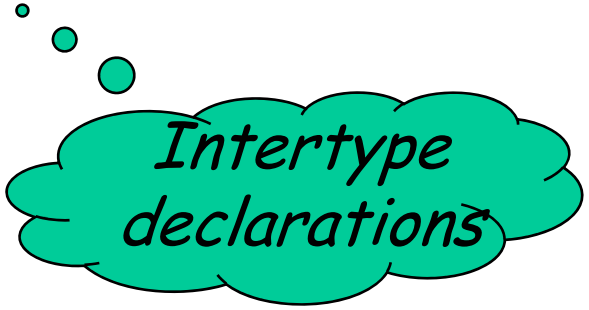
```
class T {

  void m(){
    body
  }
}
```

```
aspect T {

  void around() :
    exec(T.m) {
    pre
    proceed();
    post
  }
}
```

It is useful when you want to surround the method with extra behaviors

Besides *dynamic crosscutting* with *advice*...

- AspectJ also supports ***static crosscutting***
  - change the relation of subtypes
  - add members into classes



The success of AOP

- Assertion checking code is **localized**, can be understood in isolation, part of it can be reused
- Business code **can** be reused to work with other assertion checking APIs
- Business code is **not** invaded by changes to assertion checking APIs
- Less code, more code units

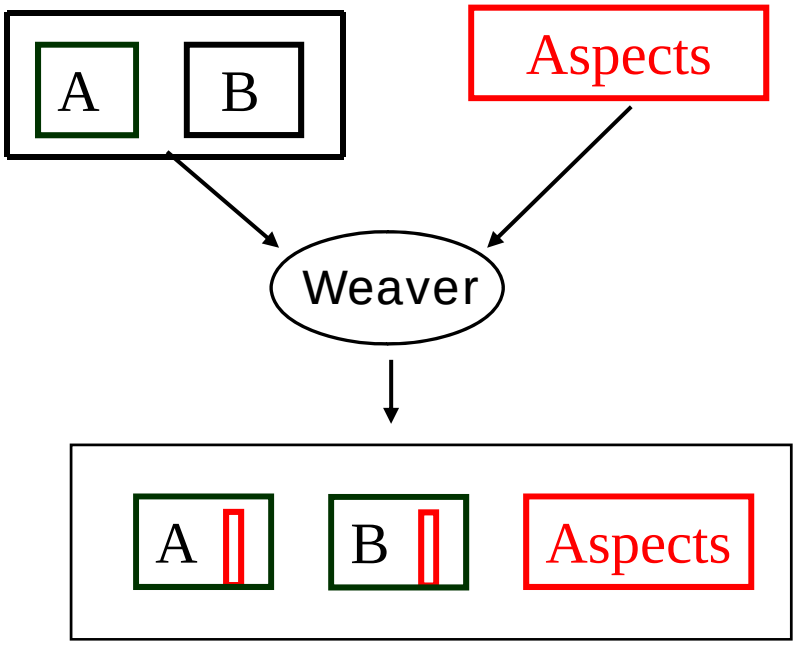
Weaving is used to…

- Compose the base system with aspects

Original system decomposed...

Composition process

Final system composed....

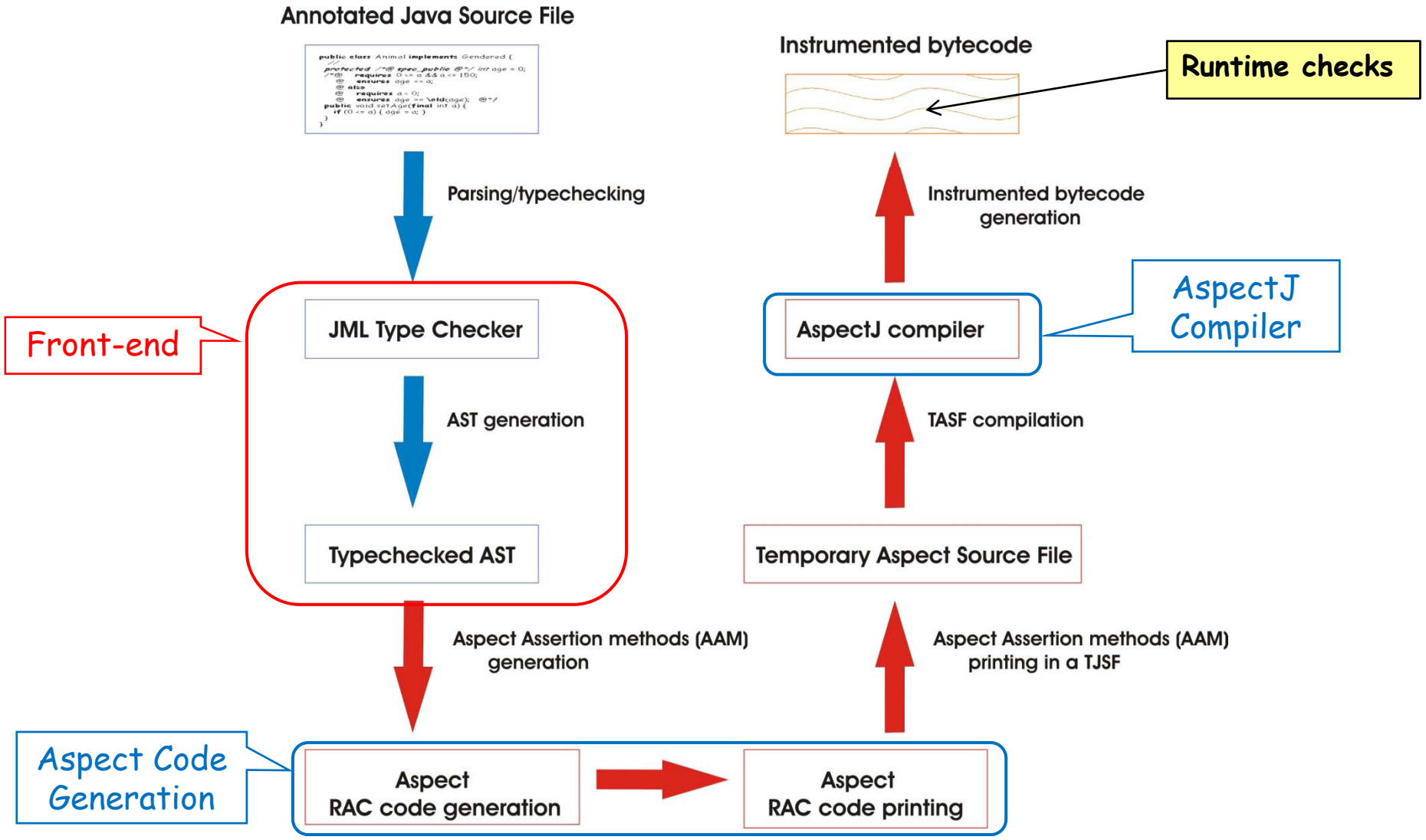


In relation to our approach…

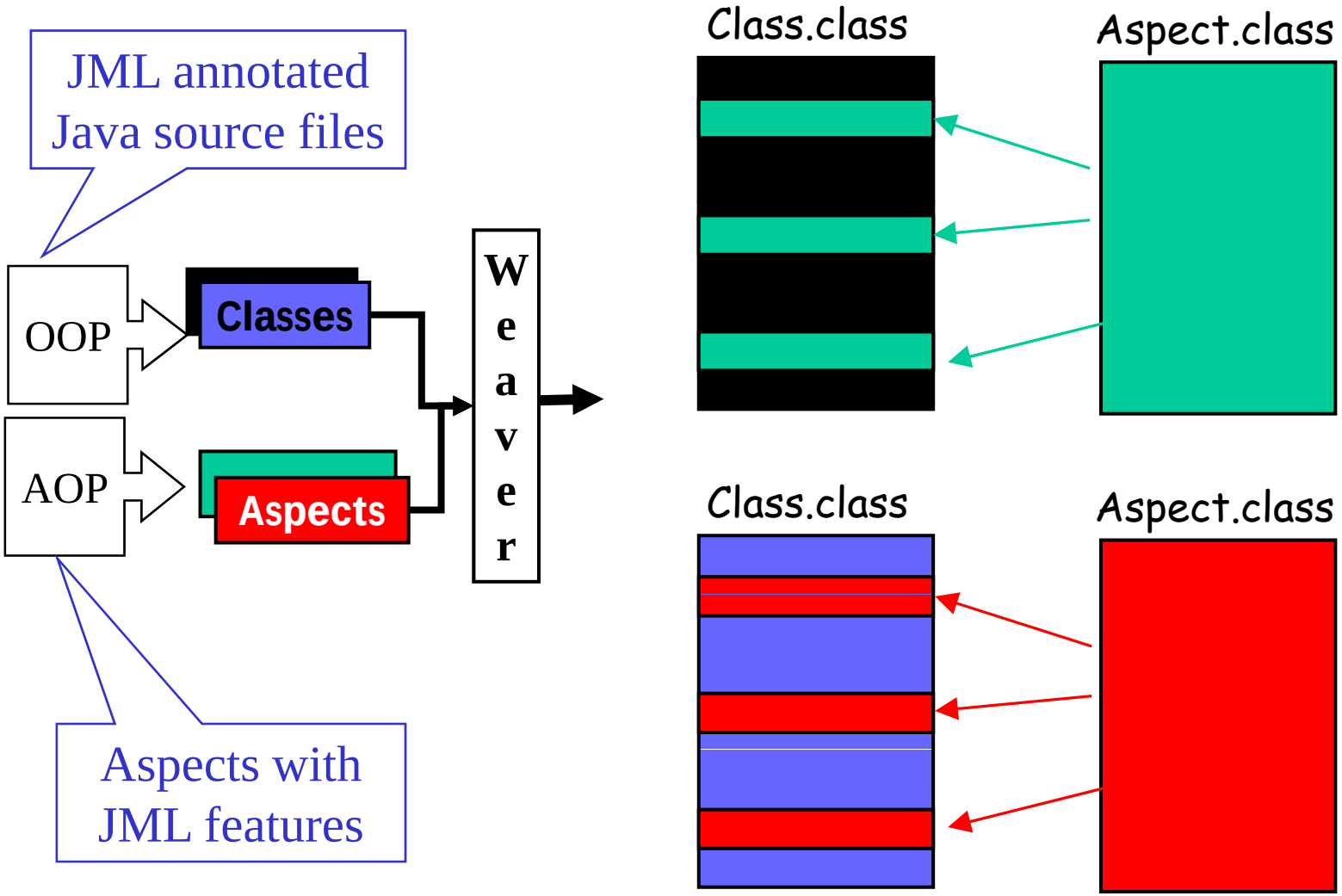
We use AOP/AspectJ to…

- Translate JML features into aspects
- Generate bytecode compliant with both Java SE/ME
- Check if the program respects the JML features during runtime

ajmlc: implementation Strategy



ajmlc: more code units generated…



Add before-execution as precondition

```
class T {

 /*@ pre a >= 0;
   @ also
   @ pre a <= 10;
   @*/
  void m(int a) {
    body
  }
}
```

=

```
class T {

 /*@ pre a
   @ also
   @ pre a
   @*/
  void m(
    body
  }
}
```

```
aspect T {

  before(T obj) :
    exec(T.m) && within(T)&&
    this(obj){
    if(!obj.checkPre$m$T()){
      throw new Error();
    }
  }
  boolean T.checkPre$m$T(){
  return (a>=0) || (a<=10);
  }
}
```

Add after-return-execution as normal postcondition

```
class T {

 /*@ pre true;
   @ post \return > 10;
   @*/
  int m(int a) {
    body
    return a+5;
  }
}
```

=

```
class T

 /*@ pre a
   @ also
   @ pre a
   @ post
   @ a+5;
   @*/
  void m
    body
    return a+5;
  }
}
```

```
aspect T {

  after(T obj)return(int r):
    exec(T.m) && this(obj){
    if(!(!(true)||(r>10))){
      throw new Error();
    }
  }
}
```

Add before and after-execution as invariants

```
class T {

  int x = 10;
//@ invariant x >= 10;

  void m() {
    body
  }
}
```

=

```
class T

  int x

//@ invari

  void m
    body
  }
}
```

```
aspect T {

  before(T obj) :
    exec(!static * T.*) &&
    this(obj){
    if(!obj.x >=10){
      throw new Error();
    }
  }
  after(T obj) returning :
    exec(!static * T.*) &&
    this(obj){
    if(!obj.x >=10){
      throw new Error();
    }
  }
}
```

Research questions

- Does AOP represent the JML features?
- What is the order and relationship between the generated aspects?
- How to check Java ME apps using ajmlc (with aspects)?
- …

The analogy between JML and Aspects

- AspectJ — An AOP extension for Java
  - dynamic crosscutting (e.g., before advice)
  - static crosscutting — ITD (e.g., new fields)
  - quantification
  - property-based crosscuting — wildcarding (*)

execution (* T.*(..))

Identifies executions to any method, with any return and parameters type, defined on type T.

The invariants analogy

```
class T {
  int i = 10;
   //@ invariant i == 10;

  void m() {...}
  void n() {...}
  void o() {...}
}
```

=

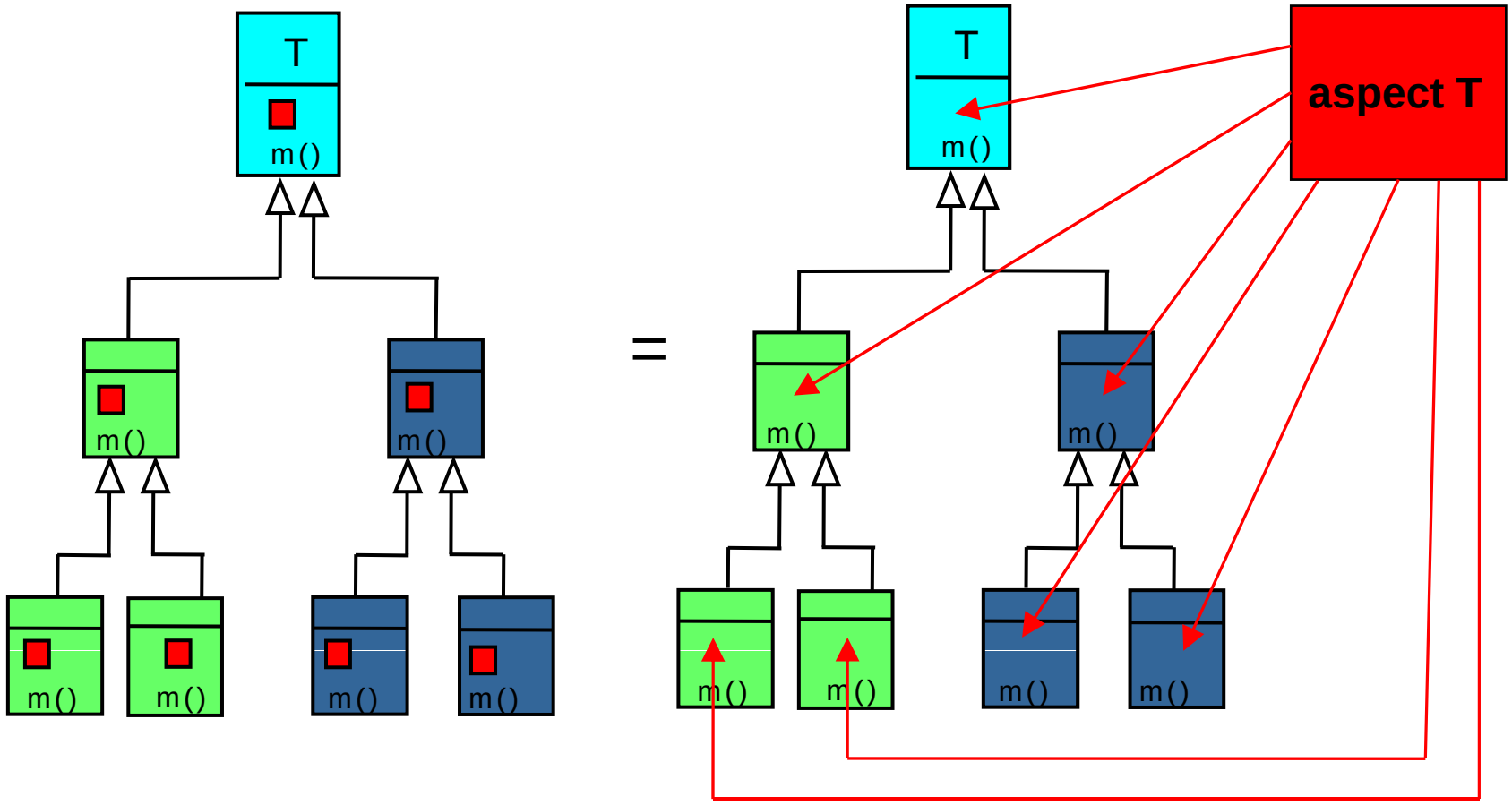
```
class T {
  int i = 10;


  void m() {...}
  void n() {...}
  void o() {...}
}
```

(→) JML feature as an aspect

(←) An aspect feature as JML spec

Both JML spec and aspect quantify the same points on type T

Behavioral subtyping analogy



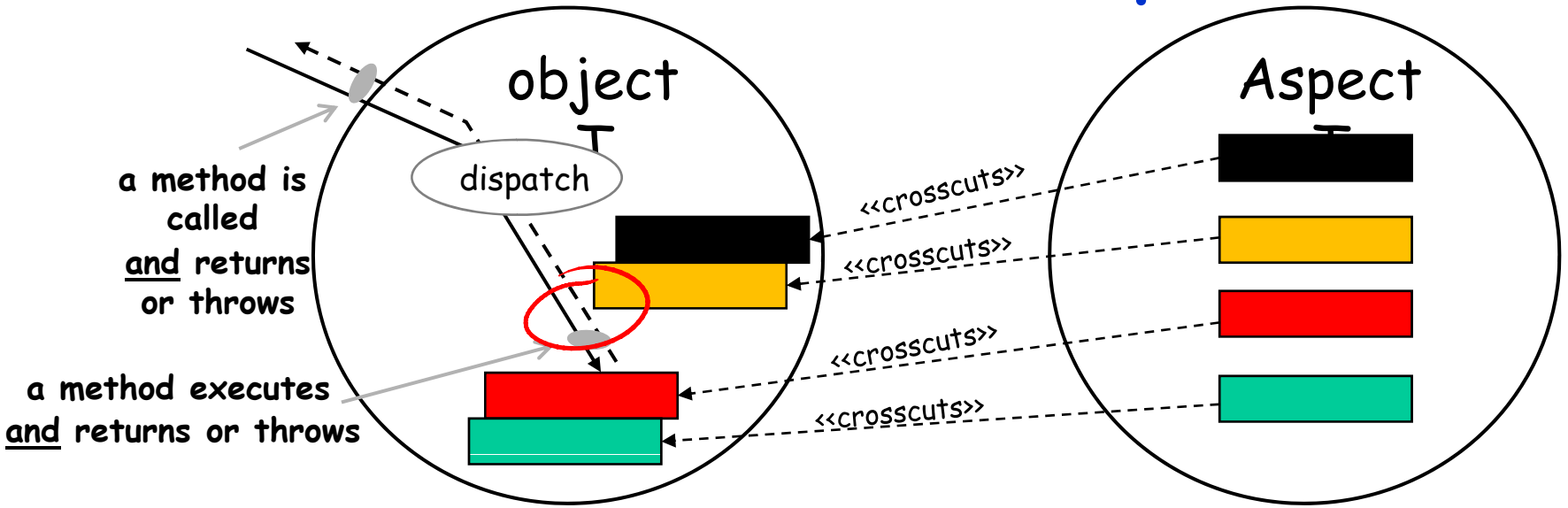
Both JML spec and aspect quantify the same points on type T and its subtypes

Other analogies

- Not limited to:
  - constraint specifications
  - refinement
  - model-programs
  - …

Other quantification points in JML that can be implemented using AspectJ

Ordering of advice executions into an aspect



Before advices to check invariants

Before advice to check preconditions

After or around advices to check postconditions

After advices to check invariants

ajmlc and Java ME applications

To verify Java ME applications, our compiler only generates aspects that avoid AspectJ constructs that are not supported by Java ME

- Avoids AspectJ constructs such as…
  - cflow pointcut
  - cflow below pointcut
  - thisJoinPoint, …

ajmlc optimizations

- Compiling empty classes
  - ajmlc generates no code
  - jmlc
    - generates 11.0 KB (source code instrumentation)
    - generates 5.93 KB (bytecode instrumentation)

```
public class T {
}
```

jmlc VS ajmlc

JML clauses	jmlc generates	ajmlc generates
requires	yes ✓	no
ensures	yes ✓	no
signals	yes ✓	no
invariant	yes ✓	no

ajmlc provides better error messages

```
1:  class C {
2:
3:    //@ pre x  > 0;
4:    void m(int x) {
5:       ...
6:    }
7:  }
```

```
1:  class Client {
2:
3:    static void main(){
4:       C c = new C();
5:       c.m(-10);
6:    }
7:  }
```

jmlc-5.6

```
c:\JML\JML-5.6\bin>jmlrac Client
Exception in thread "main" org.jmlspecs.jml
ror: by method C.m
        at Client.main(Client.java:293)
```

ajmlc-1.0

```
<terminated> Client [Java Application] C:\Java\jdk1.5.0_09\bin\ja
Exception in thread "main" org.jmlspecs.ajm
        at AspectJMLRac_C.ajc$before$Aspect
        at C.m(C.java:5)
        at Client.main(Client.java:5)
```

Study

- 3 Java SE applications
  - annotated with JML
  - extracted from JML literature
- We have compiled these programs
  - using ajmlc with two different weavers
    - —ajc
    - —abc
  - using jmlc (classical JML compiler)

Considered metric

- Code size
  - instrumented source code size
  - instrumented bytecode size
  - Jar size (bytecode size + JML lib)

Results

Source code instrumentation

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
Animal	28.8	4.8	4.8
Person	27.4	0.5	0.5
Patient	26.2	9.6	9.6
IntMathOps	18.2	2.0	2.0
StackAsArray	55.7	9.2	9.2

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
hierarchy classes	33.6	18.7	10.7
IntMathOps	20.6	7.5	4.7
StackAsArray	25.2	11.7	6.6

Jar size

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
Animal	13.3	17.0	5.5
Person	11.7	2.3	0.7
Patient	12.7	25.3	7.4
IntMathOps	9.39	5.4	2.3
StackAsArray	21.7	23.2	6.2

Bytecode instrumentation

Conclusion

- Benefits to use AOP to instrument JML
  - suitability (implementors perspective)
  - flexibility (user perspective)
  - evidence to be less complex (implementor perspective)
  - better error messages (user perspective)
- Answers to research questions
- ajmlc optimizations

Our current Work

- Extendind the ajmlc compiler to treat other JML features (e.g., model programs)
- Supporting assertion checking in a concurrent environment
- ...

An Aspect-Oriented Approach to implement JML Features

Henrique Rebêlo
Informatics Center
Federal University of Pernambuco