Specifying Design Rules in Aspect-Oriented Systems

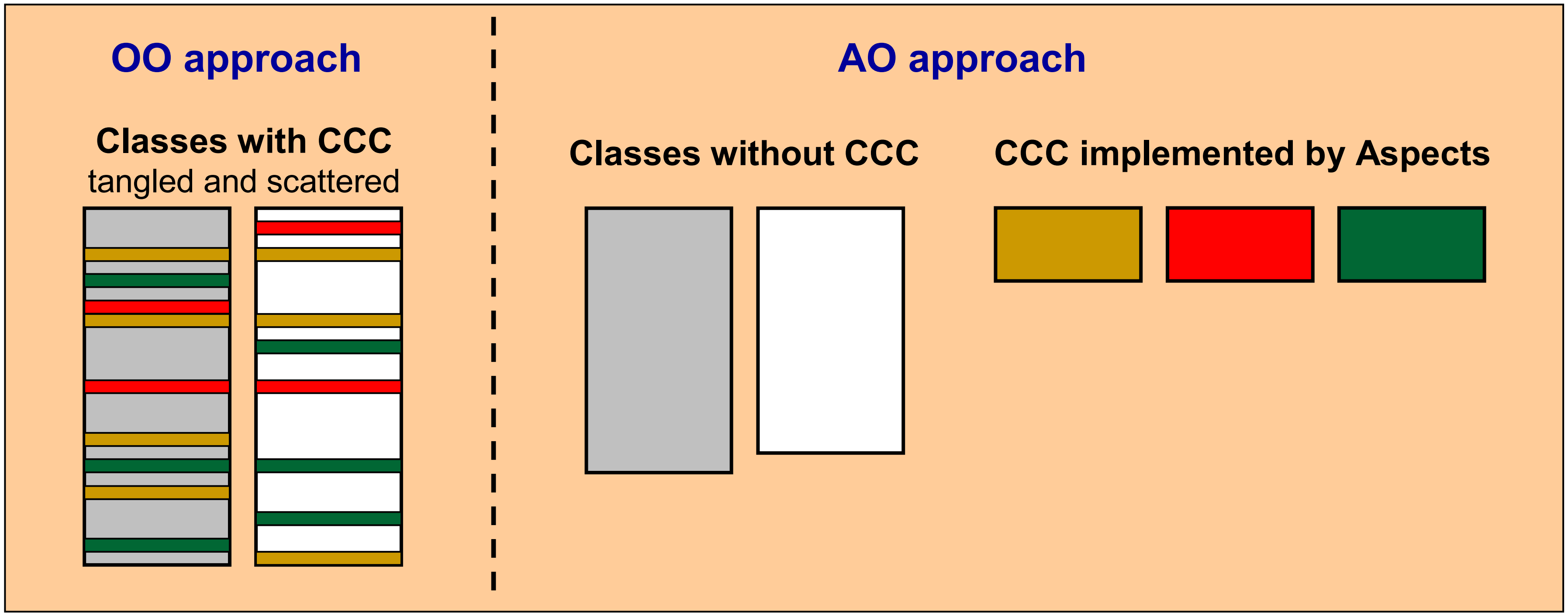
Alberto Costa Neto

Supervisor: Paulo Borba
Co-Supervisor: Fernando Castor

Federal University of Pernambuco
{acn, phmb, castor}@cin.ufpe.br

Modularizing of Crosscutting Concerns with AOP

- Examples
  - Logging, distribution, tracing, security, persistence and transactional management



Benefits of AOP

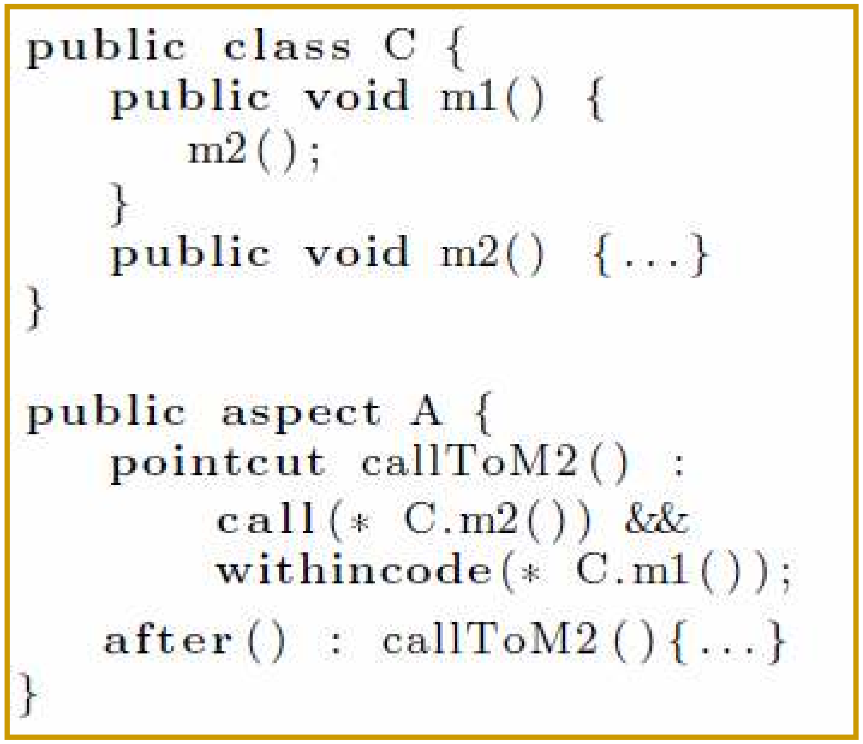
- Better code **localization** (non-scattered)
- **Less code** (in most cases)
- **More** implementation **units**
- **Classes** are more **reusable**

- Does it lead to
  well modularized systems?

Does AOP lead to systems that...

- Are easier to **understand**?
- Can be modified without **ripple effects**?
- Support **parallel development** and **independent evolution** of modules?

Are AO systems Well Modularized?

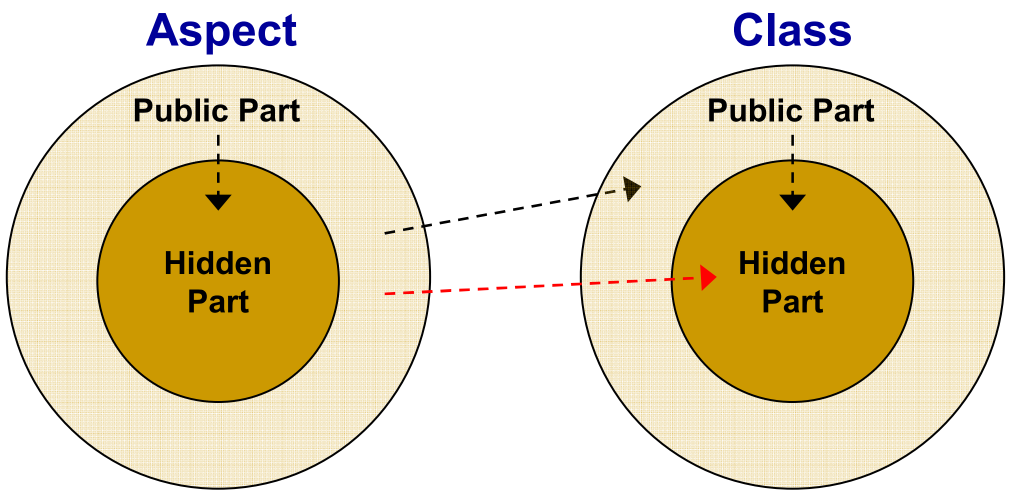


```
public class C {
    public void m1() {
        m2();
    }
    public void m2() {...}
}

public aspect A {
    pointcut callToM2() :
        call(* C.m2()) &&
        withincode(* C.m1());
    after() : callToM2(){...}
}
```

- *Comprehensibility?*
  It is difficult to reason about a Class or Aspect in isolation
- *Changeability?*
  Changes in a Class may break Aspects
- *Parallel Development?*
  Classes must be developed before Aspects

What causes modularity issues in AOP

- AOP introduces new types of dependencies (hidden parts)
  - Privileged aspects (access to private members)
  - within and withincode
  - (Un)expected join points and members



Problem

AOP → Crosscutting Concerns Modularization

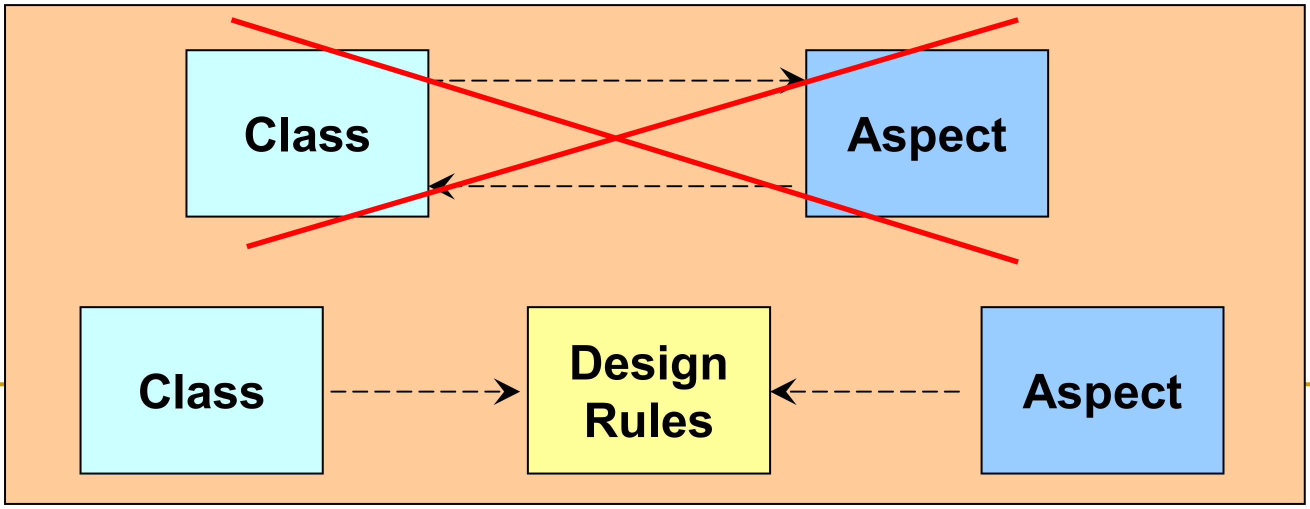
↑ Better Crosscutting Modularity

↓ Breaks Class Modularity

Is there any solution?

Design Rules

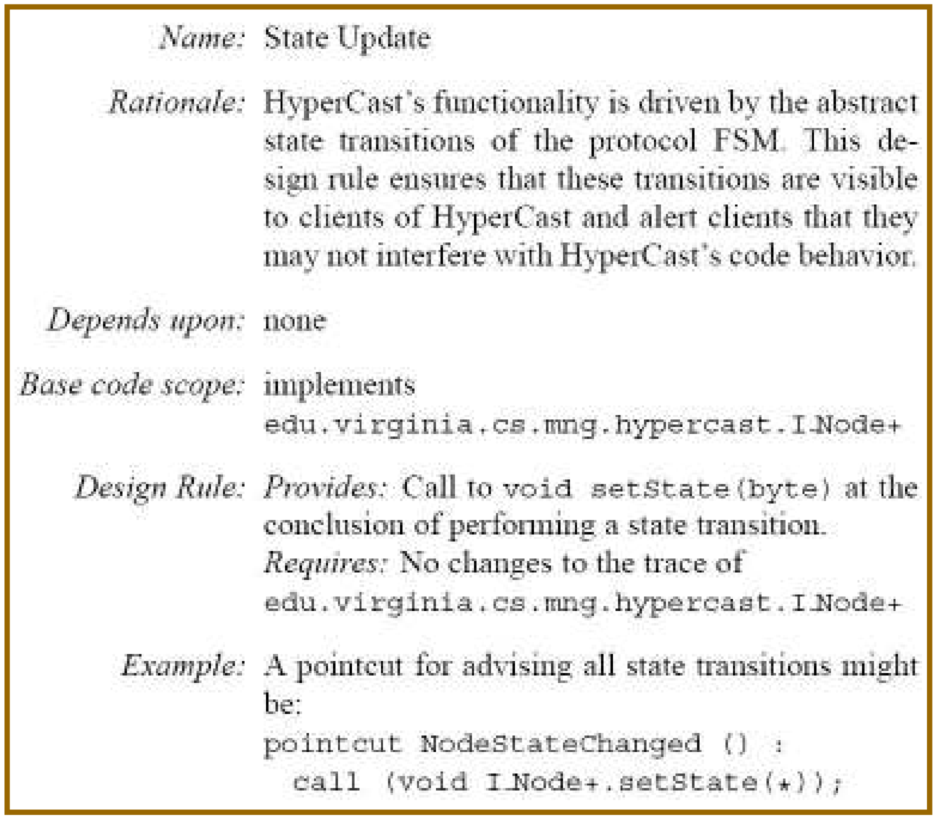
- They generalize the notion of information hiding and interfaces
- Constitute the interfaces that designers use to connect modules with each other
- By defining Design Rules we can recover Class Modularity



Expressing Design Rules

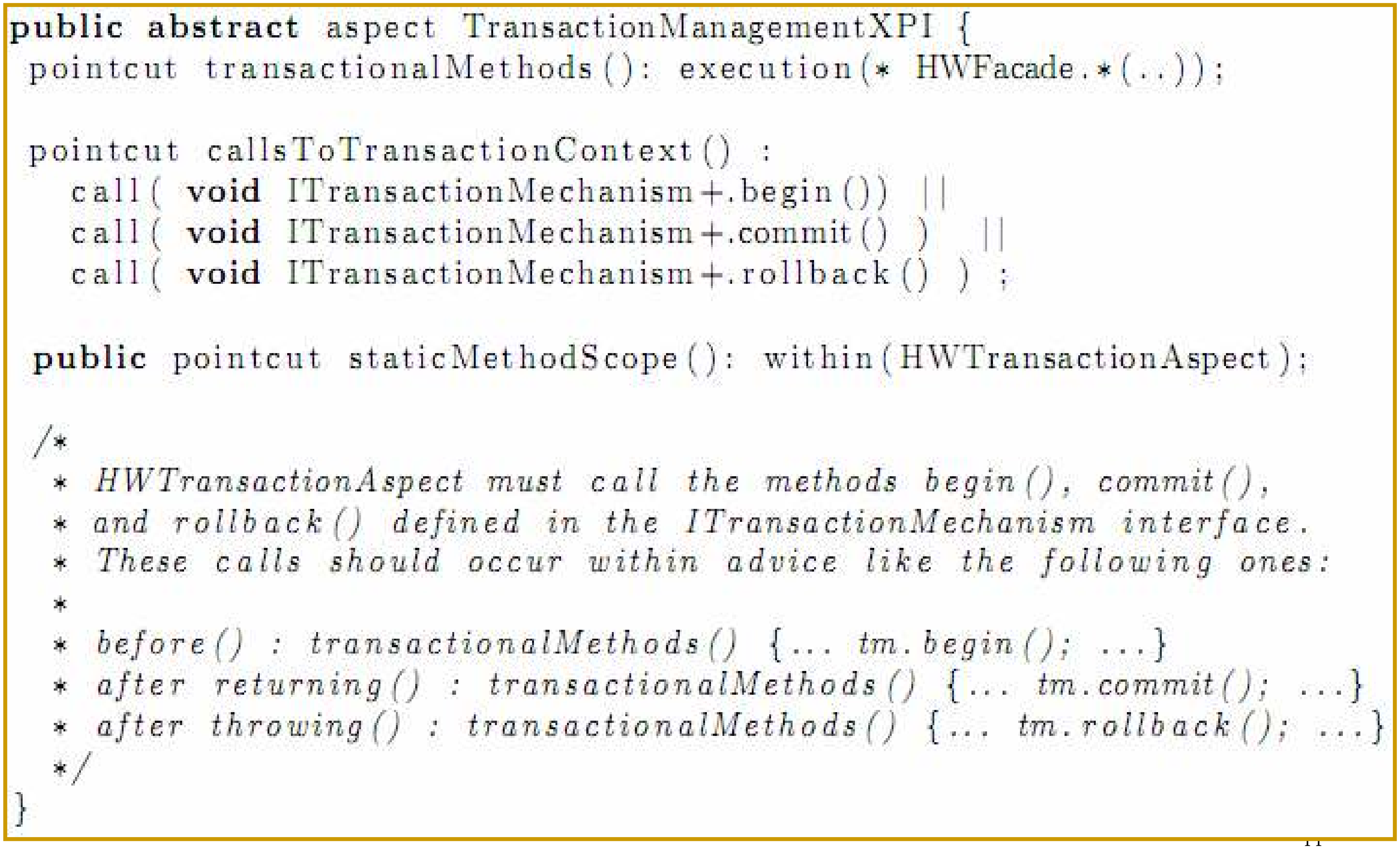
- Sullivan documented DRs using natural language
  - Verbose, incomplete, inconsistent and ambiguous specifications
  - Too expressive but cannot be checked automatically

DR Update State from HyperCast



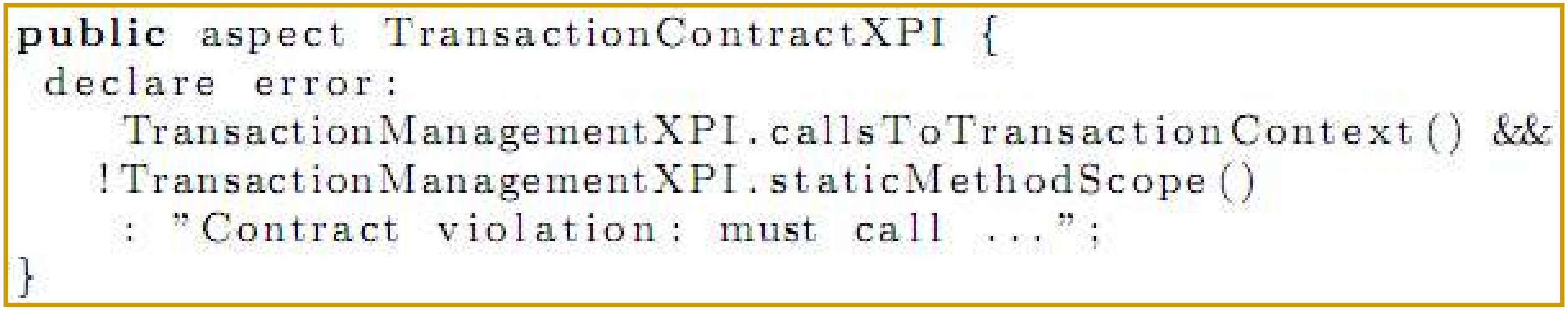
Name:	State Update
Rationale:	HyperCast's functionality is driven by the abstract state transitions of the protocol FSM. This design rule ensures that these transitions are visible to clients of HyperCast and alert clients that they may not interfere with HyperCast's code behavior.
Depends upon:	none
Base code scope:	implements `edu.virginia.cs.mng.hypercast.I_Node+`
Design Rule:	*Provides:* Call to `void setState(byte)` at the conclusion of performing a state transition. *Requires:* No changes to the trace of `edu.virginia.cs.mng.hypercast.I_Node+`
Example:	A pointcut for advising all state transitions might be: `pointcut NodeStateChanged () : call (void I_Node+.setState(*));`

Expressing DRs with XPIs (Griswold)



```
public abstract aspect TransactionManagementXPI {
 pointcut transactionalMethods(): execution(* HWFacade.*(..));

 pointcut callsToTransactionContext() :
   call( void ITransactionMechanism+.begin()) ||
   call( void ITransactionMechanism+.commit() )  ||
   call( void ITransactionMechanism+.rollback() ) ;

 public pointcut staticMethodScope(): within(HWTransactionAspect);

 /*
  * HWTransactionAspect must call the methods begin(), commit(),
  * and rollback() defined in the ITransactionMechanism interface.
  * These calls should occur within advice like the following ones:
  *
  * before() : transactionalMethods() {... tm.begin(); ...}
  * after returning() : transactionalMethods() {... tm.commit(); ...}
  * after throwing() : transactionalMethods() {... tm.rollback(); ...}
  */
}
```

XPI Contract



```
public aspect TransactionContractXPI {
 declare error:
    TransactionManagementXPI.callsToTransactionContext() &&
   !TransactionManagementXPI.staticMethodScope()
    : "Contract violation: must call ...";
}
```

- Not expressive enough
- Error-prone

Natural Language and XPI Limitations

Verbose, **Incomplete**, **Inconsistent, Ambiguous** Specifications + **No automatic checking (NL)**

Not expressive enough and **error-prone (XPI)**

No language with the specific purpose of **describing DRs in AO systems!**

Hypothesis

The use of a **language** that was designed with the **sole purpose of specifying design rules**, with

a **clearly defined semantics** and **expressive enough** to specify **most** of the **design rules** in AO systems,

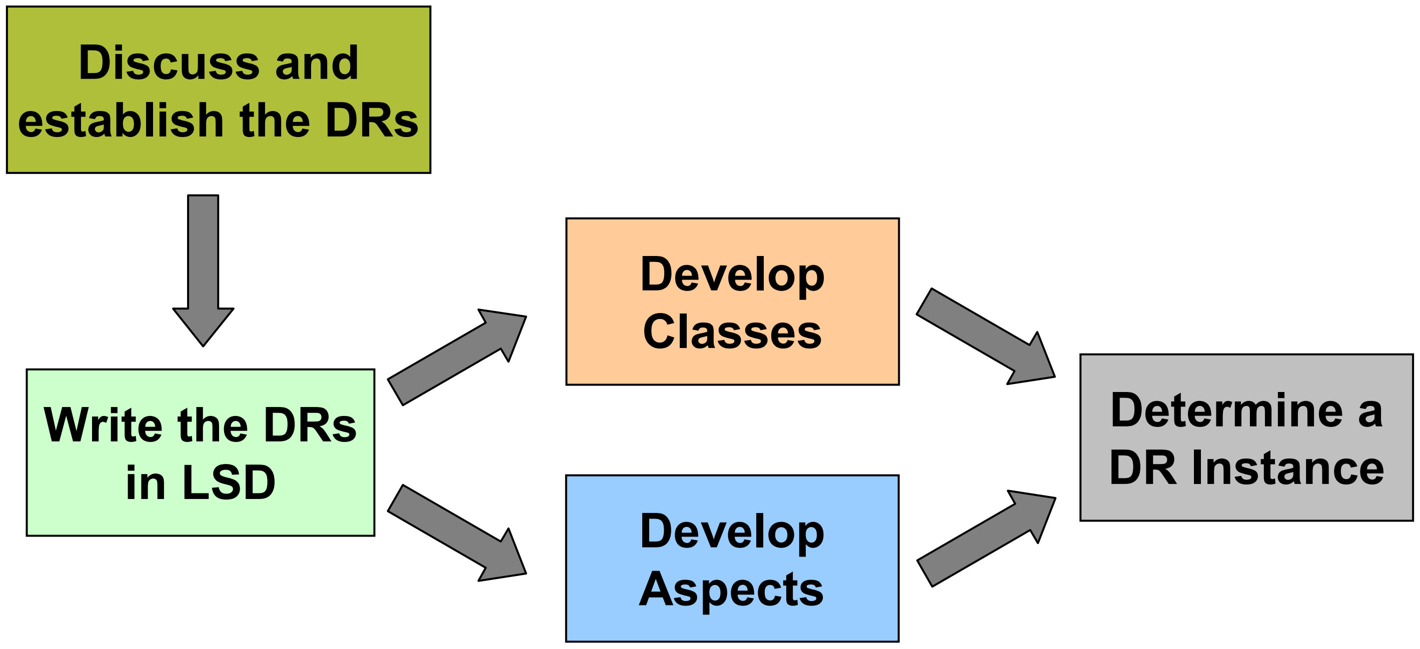
improves both class and crosscutting **modularity**, when **compared** to an **oblivious approach**,

but **does not present the problems** of **informal design rules and XPIs**.

Our Approach

- **A Language for Specifying Design Rules (LSD)**
  - **Decouples** classes and aspects
  - **Improves** Class and Crosscutting **Modularity**
  - **Syntactically similar** to Java/AspectJ
  - **Unambiguous semantics**
  - **Automatic checking**

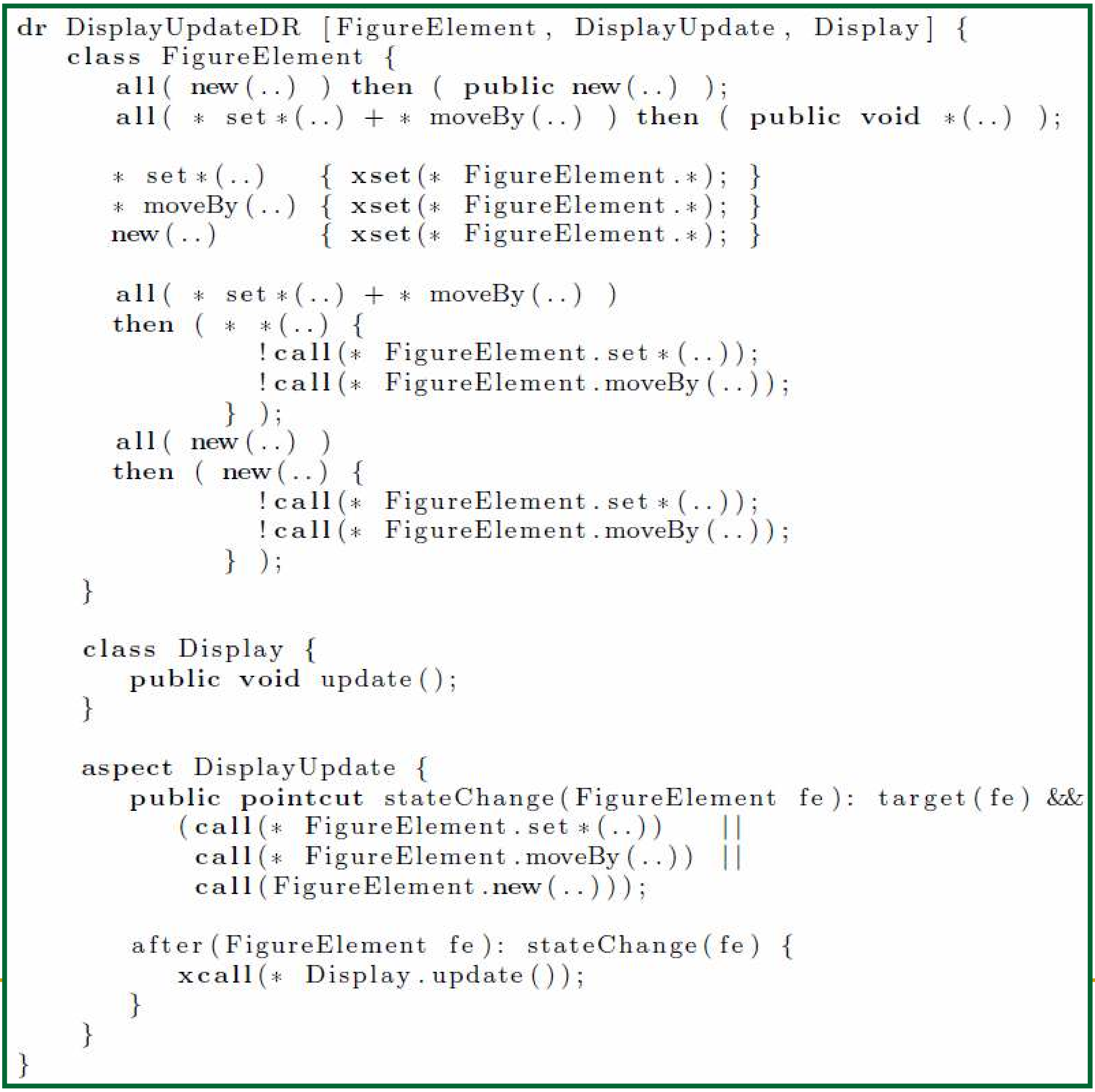
Major Steps of the Development Process with LSD



Discussing a DR - Display Update Concern of a Drawing Tool

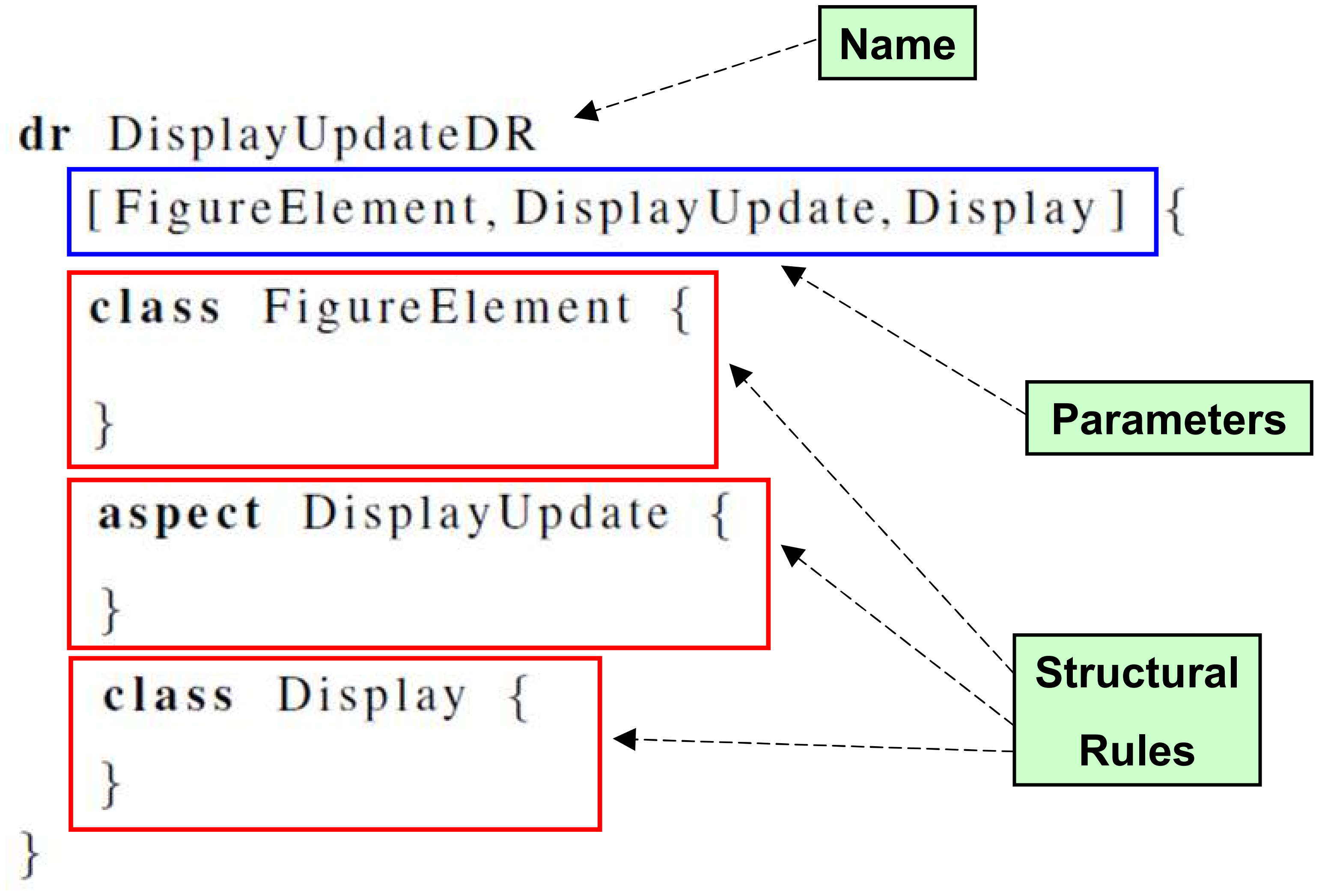
1. **FigureElement** methods called **set*** (starting with set, like setX) **and moveBy** must be **public and return void**. Also, **all constructors must be public**.
2. **FigureElement constructors and methods** called **set*** or **moveBy** are the **only possible points of state change** in figure elements.
3. Methods called **set*** or **moveBy** and **constructors must change some attribute of the figure element**.
4. Methods called **set*** or **moveBy** and **constructors cannot call** any method called **set* or moveBy from a FigureElement**.
5. A **Display** class must have a **public void update() method**.
6. The **aspect** responsible for updating the display **must declare a pointcut called stateChange** that **intercepts calls to** the **methods/constructors that change figure elements state** based on their names (predetermined).
7. The aspect must also contain an **advice** that **calls Display.update()**. This method **cannot be called from any other place** in the system.

Writing a DR in LSD



```
dr DisplayUpdateDR [FigureElement, DisplayUpdate, Display] {
   class FigureElement {
      all( new(..) ) then ( public new(..) );
      all( * set*(..) + * moveBy(..) ) then ( public void *(..) );

      * set*(..)    { xset(* FigureElement.*); }
      * moveBy(..)  { xset(* FigureElement.*); }
      new(..)       { xset(* FigureElement.*); }

      all( * set*(..) + * moveBy(..) )
      then ( * *(..) {
                  !call(* FigureElement.set*(..));
                  !call(* FigureElement.moveBy(..));
               } );
      all( new(..) )
      then ( new(..) {
                  !call(* FigureElement.set*(..));
                  !call(* FigureElement.moveBy(..));
               } );
    }

    class Display {
       public void update();
    }

    aspect DisplayUpdate {
       public pointcut stateChange(FigureElement fe): target(fe) &&
          (call(* FigureElement.set*(..))    ||
           call(* FigureElement.moveBy(..))  ||
           call(FigureElement.new(..)));

       after(FigureElement fe): stateChange(fe) {
          xcall(* Display.update());
       }
    }
}
```

DR Overview



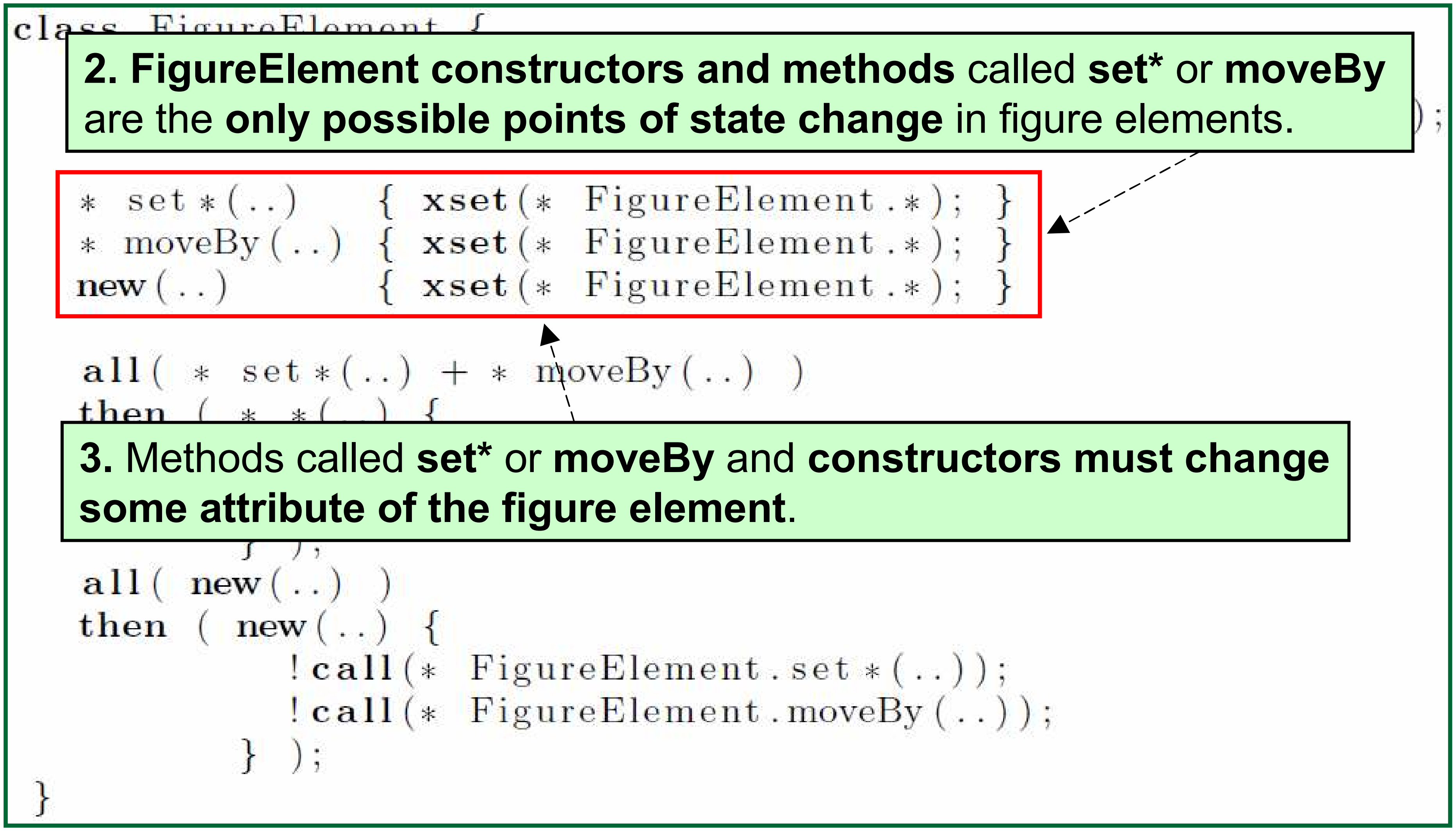
```
dr DisplayUpdateDR
   [FigureElement, DisplayUpdate, Display] {
   class FigureElement {

   }
   aspect DisplayUpdate {

   }
   class Display {

   }
}
```

Display Structural Rule

```
class Display {
    public void update();
}
```

5. Display class must have a **public void update()** method

FigureElement Structural Rule



```
class FigureElement {
                                                                        );

  *  set*(..)     { xset(*  FigureElement.*); }
  *  moveBy(..)  { xset(*  FigureElement.*); }
  new(..)          { xset(*  FigureElement.*); }

  all( * set*(..) + * moveBy(..) )
  then ( * *(..) {

          } );
  all( new(..) )
  then ( new(..) {
              !call(* FigureElement.set*(..));
              !call(* FigureElement.moveBy(..));
          } );
}
```

2. FigureElement constructors and methods called **set*** or **moveBy** are the **only possible points of state change** in figure elements.

3. Methods called **set*** or **moveBy** and **constructors must change some attribute of the figure element**.

DisplayUpdate Structural Rule

6. The aspect responsible for updating the display must declare a **pointcut called stateChange** that **intercepts** calls to the **methods/constructors** that **change figure elements state** based on their names (predetermined).

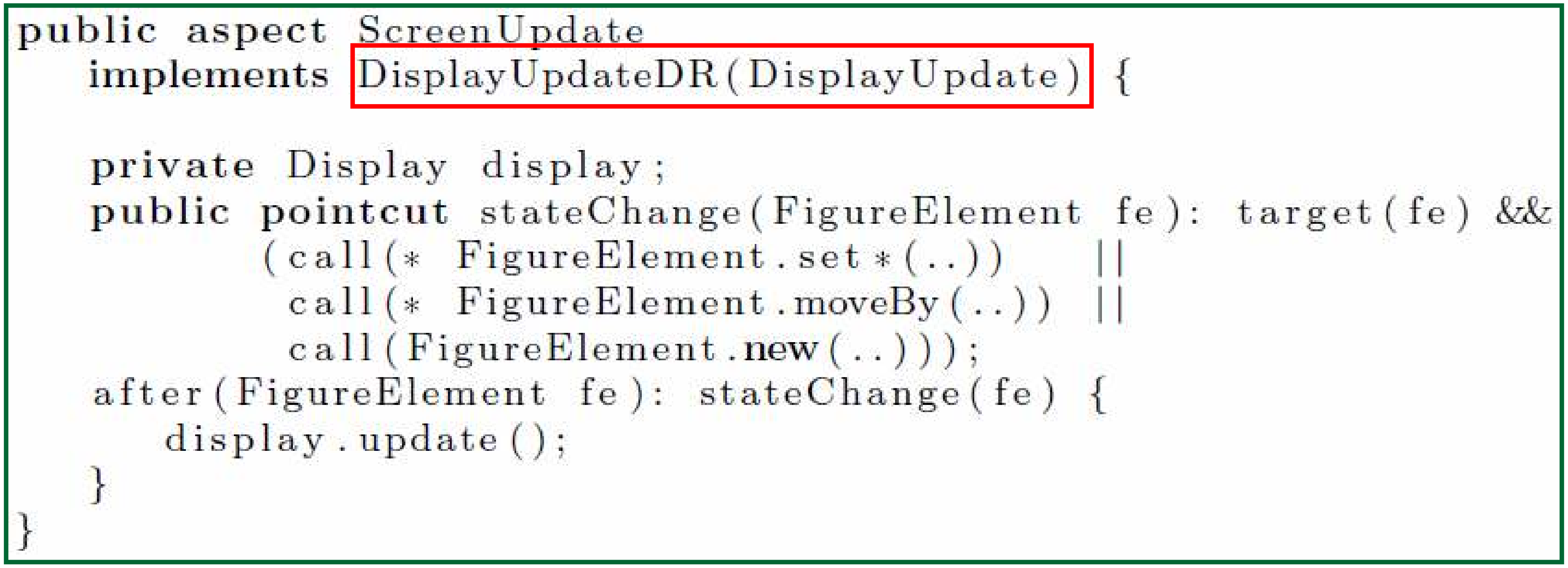
```
aspect DisplayUpdate {
  public pointcut stateChange(FigureElement fe): target(fe) &&
     (call(* FigureElement.set*(..))      ||
      call(* FigureElement.moveBy(..))  ||
      call(FigureElement.new(..)));

  after(FigureElement fe): stateChange(fe) {
     xcall(* Display.update());
  }
}
```

7. The aspect must also contain an **advice** that **calls Display.update()**. This method **cannot be called from any other place** in the system.

Implementing Classes/Interfaces

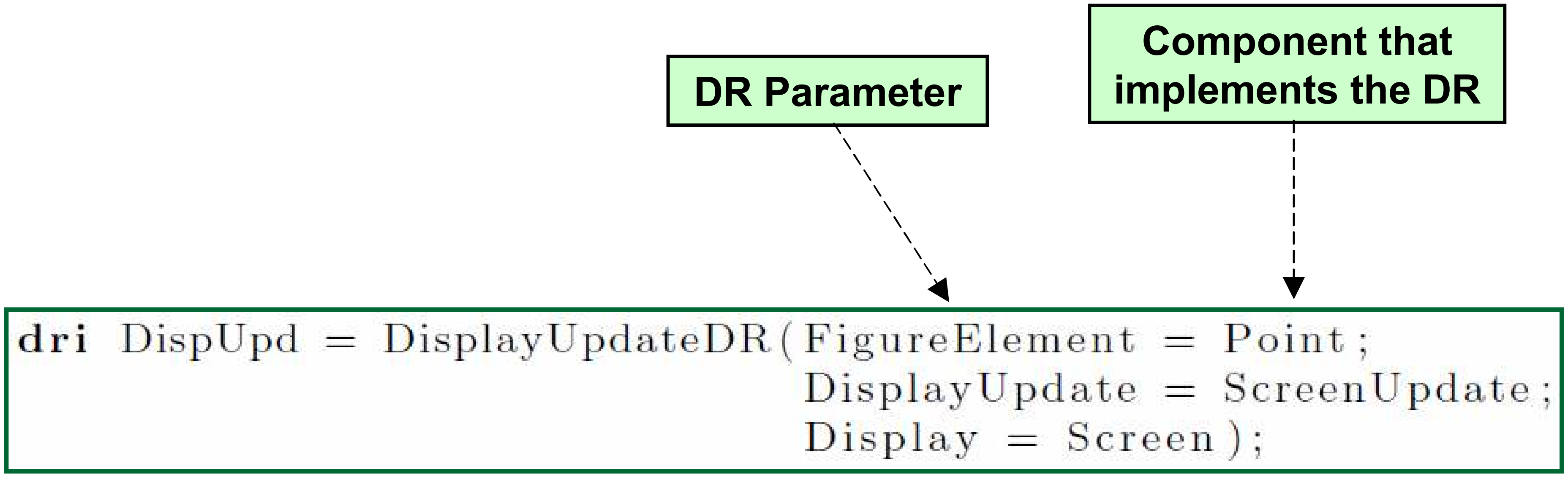
```
public class Point implements DisplayUpdateDR(FigureElement) {
    protected int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public moveBy(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Implementing Aspects



```
public aspect ScreenUpdate
   implements DisplayUpdateDR(DisplayUpdate) {

   private Display display;
   public pointcut stateChange(FigureElement fe): target(fe) &&
           (call(* FigureElement.set*(..))    ||
            call(* FigureElement.moveBy(..))  ||
            call(FigureElement.new(..)));
   after(FigureElement fe): stateChange(fe) {
      display.update();
   }
}
```

Defining a DR Instance



```
dri DispUpd = DisplayUpdateDR(FigureElement = Point;
                              DisplayUpdate = ScreenUpdate;
                              Display = Screen);
```

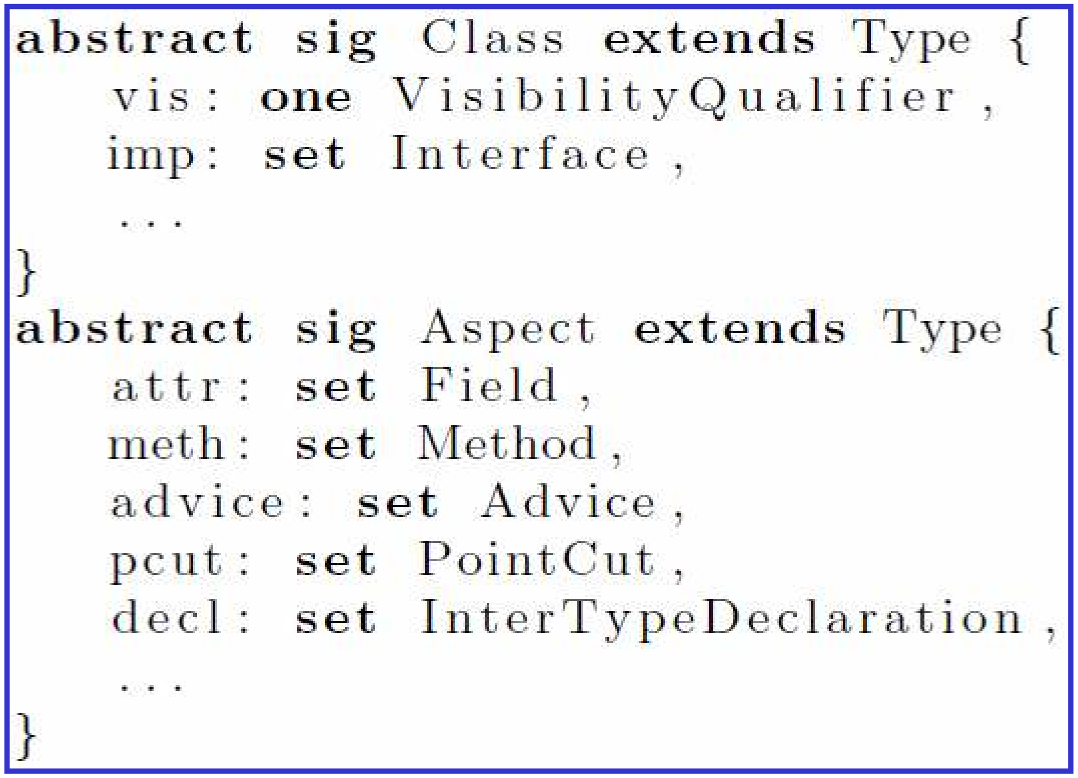
LSD Formal Semantics

Specifying LSD semantics in Alloy

- Alloy is a formal modeling language
  - **Signatures:** describe the elements of a model
  - **Facts:** describe relationships between signatures and their elements
- We chose Alloy due to:
  - Previous **experience**
  - **Tool support** to perform analysis in specifications (**Alloy Analyzer**)
  - Simplicity in expressing **first-order logic** constraints
- We mapped **LSD constructs** to a **Theory specified in Alloy**

Theory

- Abstract Syntax of all elements in our theory
  - Classes
  - Aspects
  - Methods
  - Fields
  - Advices
  - …



```
abstract sig Class extends Type {
    vis: one VisibilityQualifier,
    imp: set Interface,
    ...
}
abstract sig Aspect extends Type {
    attr: set Field,
    meth: set Method,
    advice: set Advice,
    pcut: set PointCut,
    decl: set InterTypeDeclaration,
    ...
}
```

Translating Display to Alloy

```
class Display {
   public void update();
}
```

Translation to Alloy

```
one sig Display extends Class {}{}
one sig update extends Method {} {
    vis = public
    return = void
    no update.param
    update in Display.meth
}
```

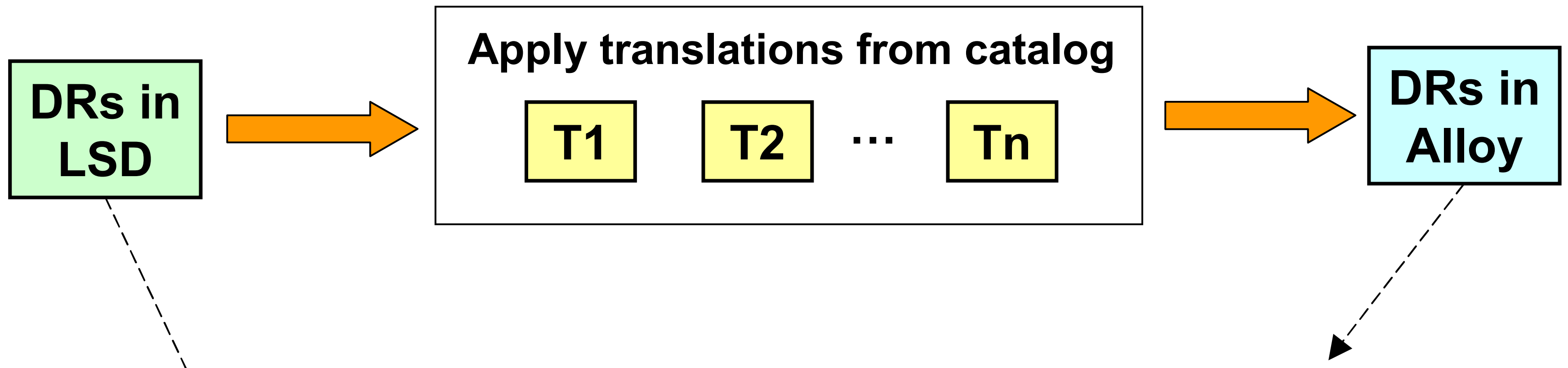
General Translations: Method Declaration

```
cds
class C {
    ... M(...){
        ...
    }
}
```

$\longrightarrow$

```
ps
one sig C ext Class {}
{ ... }
one sig M ext Method
{...} {
    M in C.meths
    ...
}
```

Applying Translations



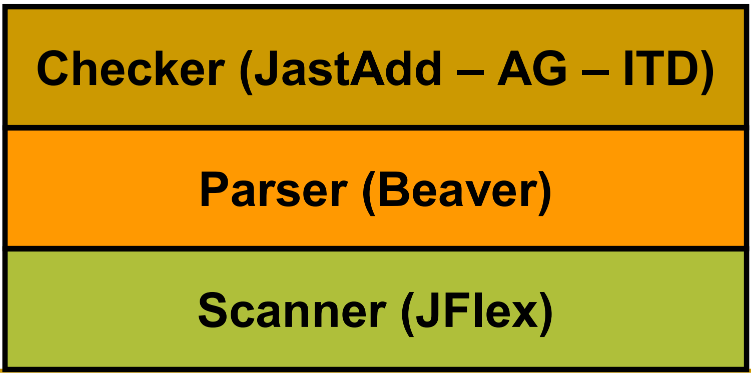
```
class Display {
   public void update();
}
```

```
one sig Display extends Class {}{}
one sig update extends Method {} {
    vis = public
    return = void
    no update.param
    update in Display.meth
}
```

COLA:
Compiler for LSD and AspectJ

COLA Overview

- Tool for checking DRs in AspectJ programs
- Checks DRs at compilation time
- AspectBench Compiler (abc) extension
  - **Polyglot**: First version
  - **JastAdd**: Second and current version



Using COLA

- Command line examples

```
abc -ext abc.lsd *.java *.aj *.dr *.dri
abc -ext abc.lsd *.*
```

Example: Extract Method breaks aspect

```
public class C {
    public void m1() {
        m2();
    }
    public void m2() {...}
}

public aspect A {
    pointcut callToM2() :
            call(* C.m2()) &&
            withincode(* C.m1());
    after() : callToM2(){...}
}
```

```
public class C {
    public void m1() {
        m3();
    }
    public void m3() {
        m2();
    }
    public void m2() {...}
}
public aspect A {
    pointcut callToM2() :
            call(* C.m2()) &&
            withincode(* C.m1());
    after() : callToM2(){...}
}
```

Using LSD to prevent the error

```
dr DREx [C,A] {
  class C {
    void m1() {
      call(* C.m2());
    }
    void m2();
  }
  public aspect A {
    pointcut callToM2() :
      call(* C.m2()) &&
      withincode(* C.m1());
  }
}
```

```
dri DRIEx = DREx(C = C;
                 A = A);
```

```
Message generated by COLA:
[Error in class C] Method declaration with required behavior not found:
void m1() { call(* C.m2()); }
(Check structural rule C within design rule DREx)
Found 1 error(s)!
```

Evaluation

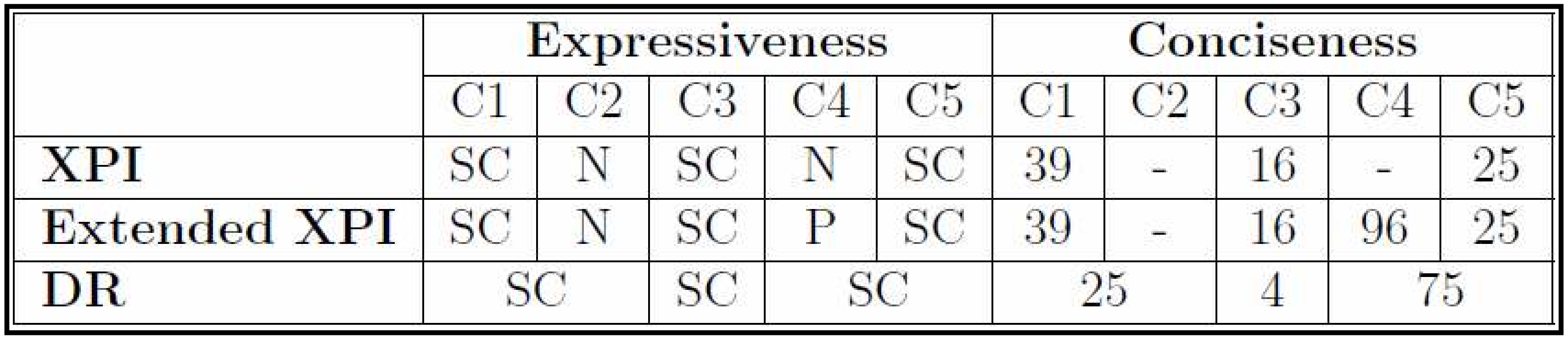
Evaluation

- Comparison between LSD and XPI
- Health Watcher concerns
  - Transaction and Distribution
  - Repository, Persistence and Exception Handling (similar)
- Design Quality Checking

Evaluation Criteria

- **Expressiveness**: quantifies the degree to which a language is able to express a constraint.
  - Three level factor - a language **supports**, **does not support**, or **partially supports** a specific rule
- **Conciseness**: measures how simple is to express a constraint in a language
  - Number of tokens required to express a constraint

Transaction Management Concern

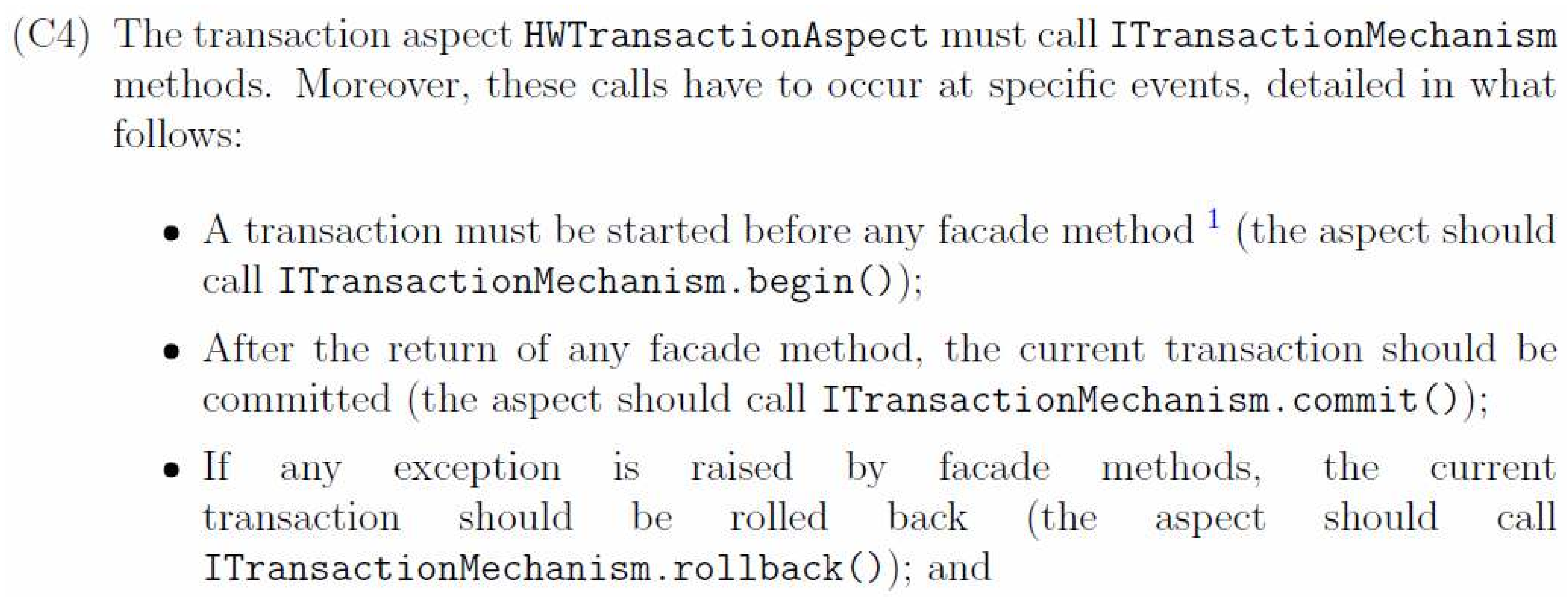


	Expressiveness					Conciseness				
	C1	C2	C3	C4	C5	C1	C2	C3	C4	C5
XPI	SC	N	SC	N	SC	39	-	16	-	25
Extended XPI	SC	N	SC	P	SC	39	-	16	96	25
DR	SC		SC	SC		25		4	75	

Where:

- **SC** = Statically checked
- **P** = Partially checked
- **N** = No checking (only expressible by means of natural language)

Enforcing transactional methods calls



(C4) The transaction aspect `HWTransactionAspect` must call `ITransactionMechanism` methods. Moreover, these calls have to occur at specific events, detailed in what follows:

- A transaction must be started before any facade method [1] (the aspect should call `ITransactionMechanism.begin()`);
- After the return of any facade method, the current transaction should be committed (the aspect should call `ITransactionMechanism.commit()`);
- If any exception is raised by facade methods, the current transaction should be rolled back (the aspect should call `ITransactionMechanism.rollback()`); and

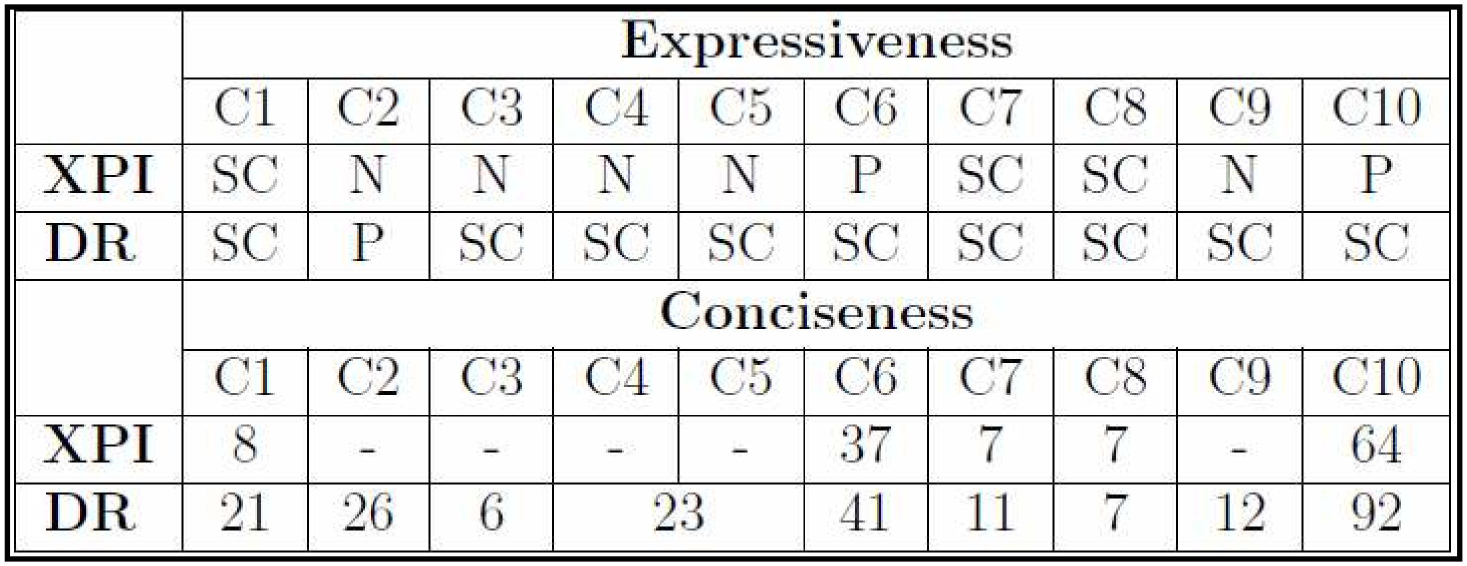
Partially checking C4 with XPI (extended version)

```
public abstract aspect XPITransaction {
  ...
 public pointcut expectedCallToBegin() :
   within(HWTransactionAspect) &&
   call(void ITransactionMechanism+.begin());

 public pointcut expectedCallToCommit() :
   within(HWTransactionAspect) &&
   call(void ITransactionMechanism+.commit());

 public pointcut expectedCallToRollback() :
   within(HWTransactionAspect) &&
   call(void ITransactionMechanism+.rollback());

 before(): expectedCallToBegin() { }

 before(): expectedCallToCommit() { }

 before(): expectedCallToRollback() { }
}
```

Checking C4 with LSD

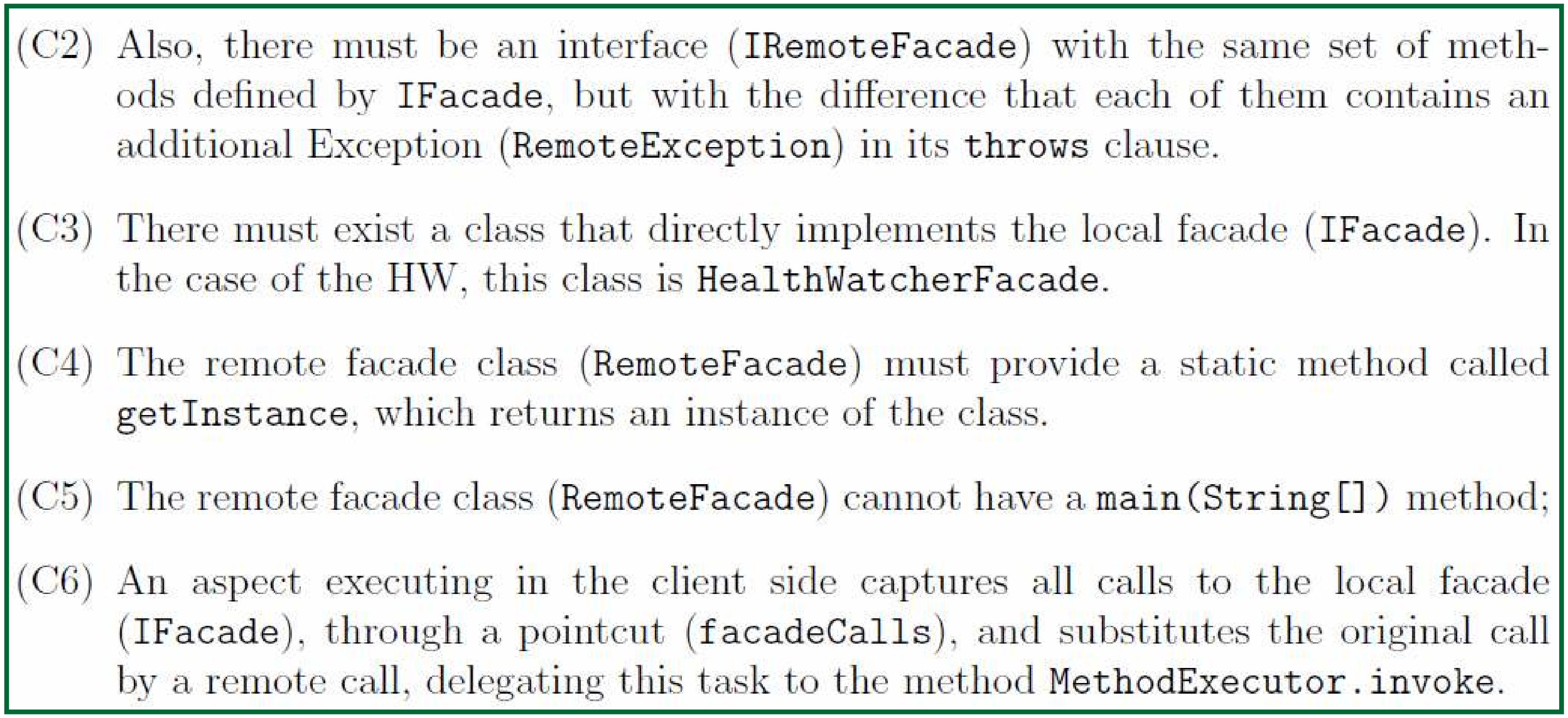
```
dr TransactionManagementDR
   [ITransactionMechanism, TransactionManagement, Facade] {

    interface ITransactionMechanism {
        void begin() throws TransactionException;
        void commit() throws TransactionException;
        void rollback() throws TransactionException;
    }

    aspect TransactionManagement {

      pointcut transactionalPoints(): call(* Facade.*(..));

      before(): transactionalPoints() {
           xcall(void ITransactionMechanism.begin());
      }
      after() returning: transactionalPoints() {
           xcall(void ITransactionMechanism.commit());
      }
      after() throwing: transactionalPoints() {
           xcall(void ITransactionMechanism.rollback());
      }
   }

   class Facade {}
}
```

Distribution Concern



	Expressiveness									
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
XPI	SC	N	N	N	N	P	SC	SC	N	P
DR	SC	P	SC	SC	SC	SC	SC	SC	SC	SC
	Conciseness									
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
XPI	8	-	-	-	-	37	7	7	-	64
DR	21	26	6	23		41	11	7	12	92

Distribution Design Rules (C2 – C6)



(C2) Also, there must be an interface (`IRemoteFacade`) with the same set of methods defined by `IFacade`, but with the difference that each of them contains an additional Exception (`RemoteException`) in its `throws` clause.

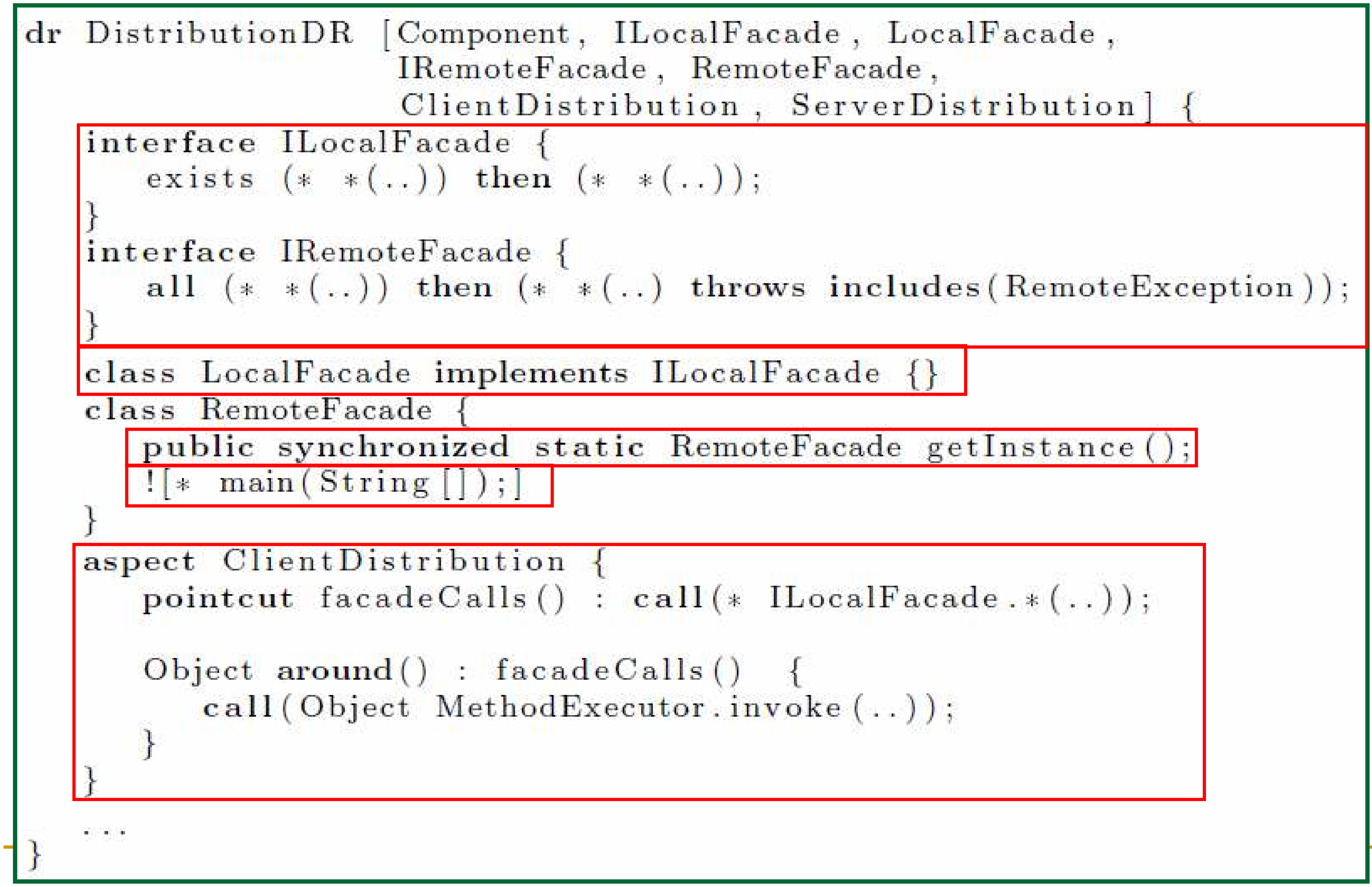
(C3) There must exist a class that directly implements the local facade (`IFacade`). In the case of the HW, this class is `HealthWatcherFacade`.

(C4) The remote facade class (`RemoteFacade`) must provide a static method called `getInstance`, which returns an instance of the class.

(C5) The remote facade class (`RemoteFacade`) cannot have a `main(String[])` method;

(C6) An aspect executing in the client side captures all calls to the local facade (`IFacade`), through a pointcut (`facadeCalls`), and substitutes the original call by a remote call, delegating this task to the method `MethodExecutor.invoke`.

Checking C2 – C6 with LSD



```
dr DistributionDR [Component, ILocalFacade, LocalFacade,
                   IRemoteFacade, RemoteFacade,
                   ClientDistribution, ServerDistribution] {
  interface ILocalFacade {
     exists (* *(..)) then (* *(..));
  }
  interface IRemoteFacade {
     all (* *(..)) then (* *(..) throws includes(RemoteException));
  }
  class LocalFacade implements ILocalFacade {}
  class RemoteFacade {
     public synchronized static RemoteFacade getInstance();
     ![* main(String[]);]
  }
  aspect ClientDistribution {
     pointcut facadeCalls() : call(* ILocalFacade.*(..));

     Object around() : facadeCalls() {
        call(Object MethodExecutor.invoke(..));
     }
  }
  ...
}
```

Design Quality Checking

(C1) The number of public methods must range from 1 to 10;
(C2) No public attribute is allowed.

	Expressiveness		Conciseness	
	C1	C2	C1	C2
XPI	N	P	-	28
DR	SC	SC	23	12

```
public aspect NonPublicAttributesXPI {
    declare error : get(public * *.*) || set(public * *.*) :
        "Public attributes are prohibited";
}
```

```
dr QualityDR [C] {
    class C {
        range[1..10] (* *(..)) then (public * *(..));
        none (* *) then (public * *);
    }
}
```

Evaluation Results

- Some evidence that LSD enhances the XPI approach
  - More expressive and concise
- LSD does not hinder the use of XPIs
  - More constraints than XPIs

Conclusion & Future Work

Conclusion

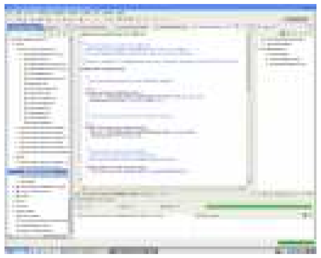
- The definition of a language for specifying design rules (**LSD**)
- A **formal specification** of the language semantics in Alloy
- **Evaluation** of the proposed language in real case studies and its **comparison with XPIs**
- Tool to support the use of design rules (**COLA**)

Future Work

- Add support to invariants, pre- and post conditions checked dynamically in DRs
- Tool for checking DR consistency
- Define a complete set of Translations from LSD to Alloy
- Translation tool from LSD to Alloy

Future Work (2)

- IDE extension to support DRs
  - Visualization based on the DR instance



- DR-based component generation

- Build a completely new system using LSD and XPIs

Questions