# Modularity, information hiding, and interfaces for aspect-oriented languages

Paulo Borba
Informatics Center
Federal University of Pernambuco

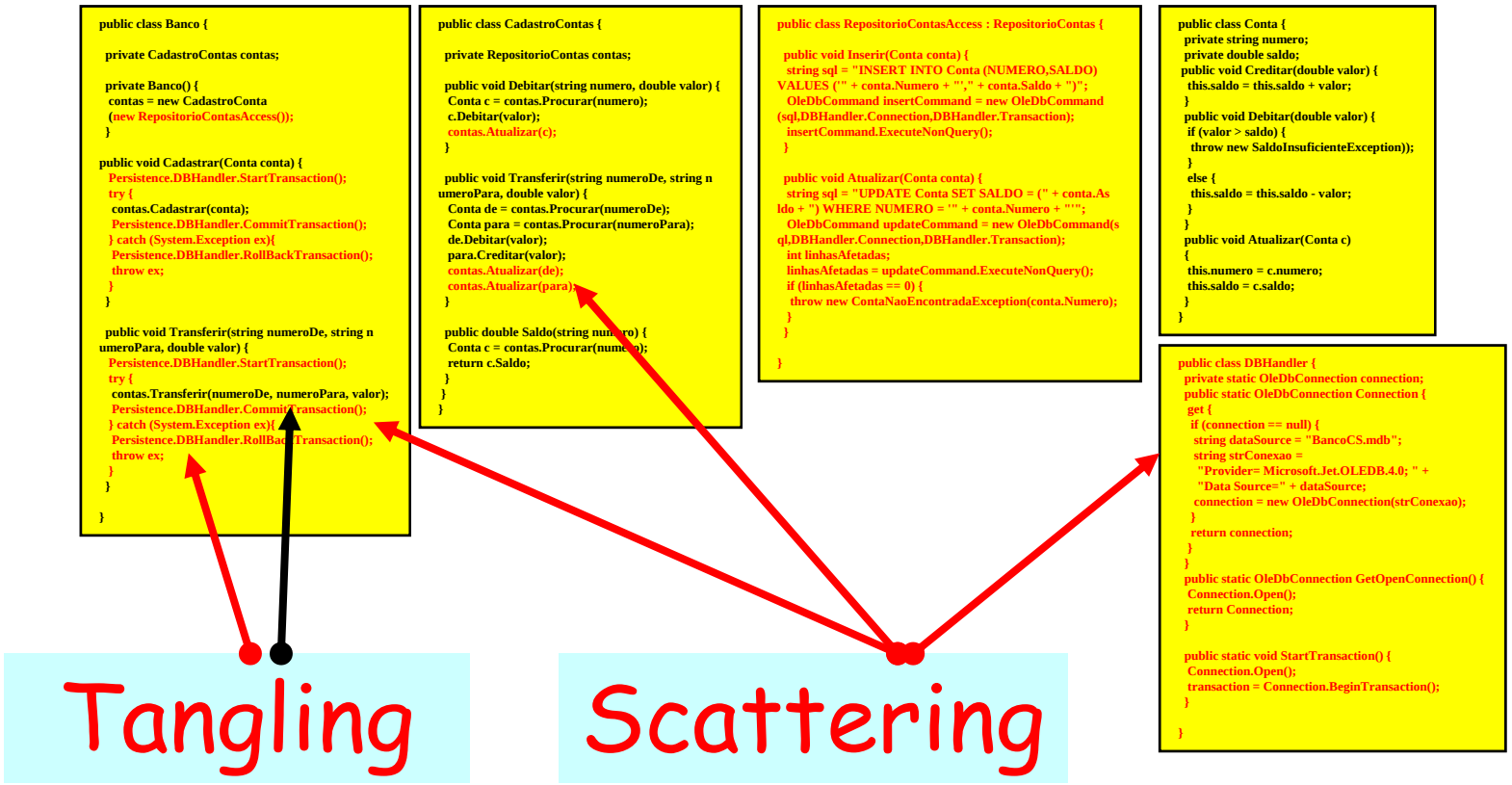

Aspect-oriented languages are quite popular...

due to the **promise** of

modularizing

crosscutting concerns

# Persistence without aspects...



# The failure of OOP

- Most persistence code is tangled and scattered, cannot be reused
- Business code cannot be reused to work with other persistence APIs
- Business code is invaded by changes to persistence APIs

# Persistence with aspects

```
public class Banco {

  private CadastroContas contas;

  private Banco() {
   contas = new CadastroConta
}

public void Cadastrar(Conta conta) {
contas.Cadastrar(conta);
}

  public void Transferir(string numeroDe, string n
umeroPara, double valor) {
contas.Transferir(numeroDe, numeroPara, valor);
}

}
```

```
public class CadastroContas {

  private RepositorioContas contas;

  public void Debitar(string numero, double valor) {
   Conta c = contas.Procurar(numero);
   c.Debitar(valor);
}

  public void Transferir(string numeroDe, string n
umeroPara, double valor) {
   Conta de = contas.Procurar(numeroDe);
   Conta para = contas.Procurar(numeroPara);
   de.Debitar(valor);
   para.Creditar(valor);
}

  public double Saldo(string numero) {
   Conta c = contas.Procurar(numero);
   return c.Saldo;
  }
 }
}
```

```
public class Conta {
  private string numero;
  private double saldo;
 public void Creditar(double valor) {
   this.saldo = this.saldo + valor;
  }
  public void Debitar(double valor) {
   if (valor > saldo) {
    throw new SaldoInsuficienteException));
   }
   else {
    this.saldo = this.saldo - valor;
   }
  }
  public void Atualizar(Conta c)
  {
   this.numero = c.numero;
   this.saldo = c.saldo;
  }
 }
```

Business code

```
public class RepositorioContasAccess : RepositorioContas {

  public void Inserir(Conta conta) {
   string sql = "INSERT INTO Conta (NUMERO,SALDO)
VALUES ('" + conta.Numero + "'," + conta.Saldo + ")";
   OleDbCommand insertCommand = new OleDbCommand
(sql,DBHandler.Connection,DBHandler.Transaction);
   insertCommand.ExecuteNonQuery();
  }

  public void Atualizar(Conta conta) {
   string sql = "UPDATE Conta SET SALDO = (" + conta.As
ldo + ") WHERE NUMERO = '" + conta.Numero + "'";
   OleDbCommand updateCommand = new OleDbCommand(s
ql,DBHandler.Connection,DBHandler.Transaction);
   int linhasAfetadas;
   linhasAfetadas = updateCommand.ExecuteNonQuery();
   if (linhasAfetadas == 0) {
    throw new ContaNaoEncontradaException(conta.Numero);
   }
  }

}
```

```
public class DBHandler {
 private static OleDbConnection connection;
 public static OleDbConnection Connection {
  get {
   if (connection == null) {
    string dataSource = "BancoCS.mdb";
    string strConexao =
     "Provider= Microsoft.Jet.OLEDB.4.0; " +
     "Data Source=" + dataSource;
    connection = new OleDbConnection(strConexao);
   }
   return connection;
  }
 }
 public static OleDbConnection GetOpenConnection() {
  Connection.Open();
  return Connection;
 }

 public static void StartTransaction() {
  Connection.Open();
  transaction = Connection.BeginTransaction();
 }

}
```

```
public class Conta {
  private string numero;
  private double saldo;
 public void Creditar(double valor) {
   this.saldo = this.saldo + valor;
  }
  public void Debitar(double valor) {
```

```
public class Conta {
  private string numero;
 ;
  }
  public void Debitar(double valor) {
```

Persistence code

# The success of AOP

- Persistence code is **localized**, part of it can be reused
- Business code **can** be reused to work with other persistence APIs
- Business code is **not** invaded by changes to persistence APIs
- Less code, more code units


AOP is not...
Quantification
+
Obliviousness

# Obliviousness (Filman and Friedman, 2000)

**For true AOP, we want our system to work with *oblivious programmers – ones who don't have to* expend any additional effort to make the AOP mechanism work.**

Can they be aware of aspects?

# Being precise about obliviousness (Sullivan et al, FSE 2005)

- Language-level obliviousness
- Feature obliviousness
- Designer obliviousness
- Pure obliviousness

# Persistence at façade methods

```
public class Bank {
  private AccountCollection accounts;...
  public void register(Account account) {
    try {
      getPM().startTransaction();
      accounts.register(account);...
      getPM().commitTransaction();
    } catch (Exception ex){
      getPM().rollBackTransaction();...
    }
  }...
```

Business code mixed with transaction code

# Persistence aspect, advice

```
public aspect PersistenceAspect {
  before(): transMethods() {
    getPM().startTransaction();
  }
  after() returning(): transMethods() {
    getPM().commitTransaction();
  }
  after() throwing(): transMethods() {
    getPM().rollBackTransaction();
  }...
```

# AOP is not...

**Quantification**

**+**

**Designer obliviousness**

AOP is...

**Quantification**

**+**

**Language-level obliviousness**

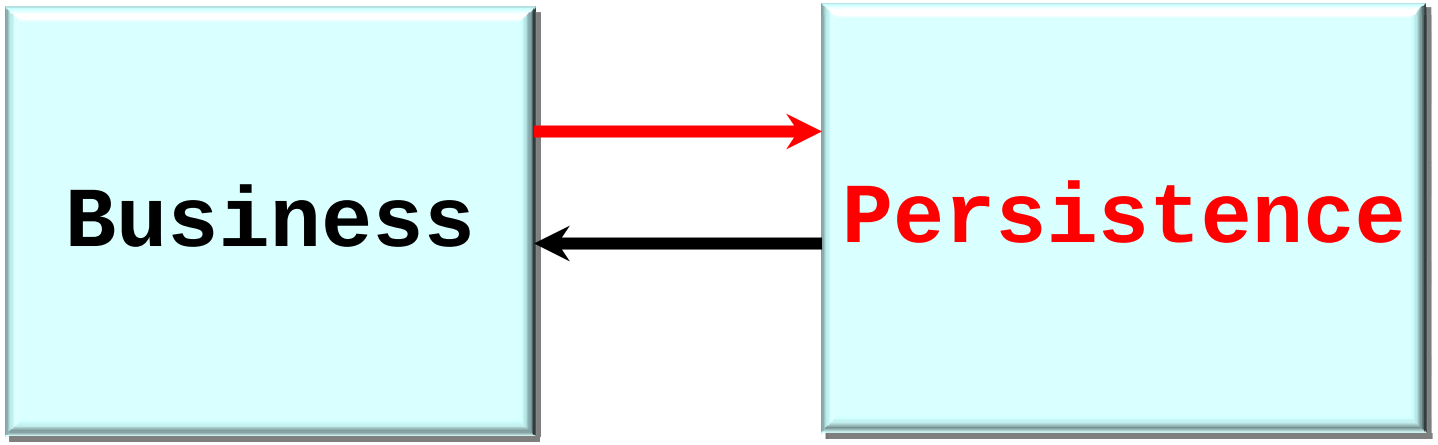
AOP is...

**Quantification**

**+**

**Non invasiveness**

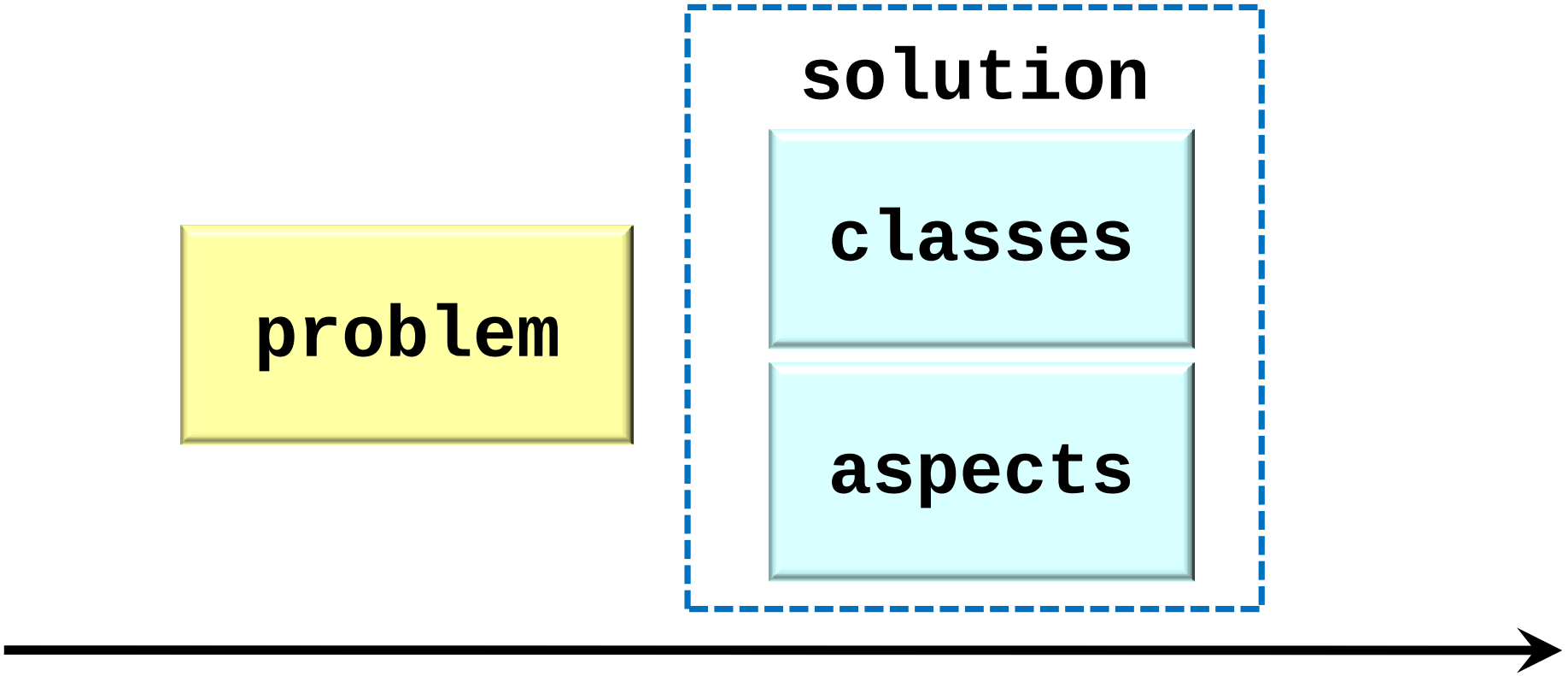
# But persistence and business code are coupled!



# The failure of AOP

- No true modularity!
- It is difficult to achieve
  - Independent development
  - Independent maintenance
  - Independent understanding

# No parallel development and evolution



# Designer obliviousness wouldn't help (Filman and Friedman, 2000)

**It's a really nice bumper sticker to be able to say, "Just program like you always do, and we'll be able to add the aspects later."**

## Business code developer should be aware of persistence code...

- to expose proper joinpoints
- to know that aspects declare some class members

# Persistence code developer should be aware of business code...

- to exploit exposed joinpoints and private members
- to know that it should declare some class members in aspects

# Changes to business code might impact persistence code...

- when added members conflict with members added by aspects
- when there are changes in joinpoints and private members
  - inline method refactoring

# What about modular reasoning?

- None for classes
  - modifications to a class should be fully aware of the aspects that affect that class
- Some for aspects
  - modifications to a crosscutting concern are mostly local

# The paradoxical success of AOP (Friedrich Steimann, OOPSLA 2006)

Constructs aimed to support modularity actually break modularity

# No paradox, but refined view on modularity...

Constructs aimed

to support

**crosscutting modularity**

actually break

**class modularity**

# Design rules

- General notion of interfaces and modularity
- Reconciles class and crosscutting modularity

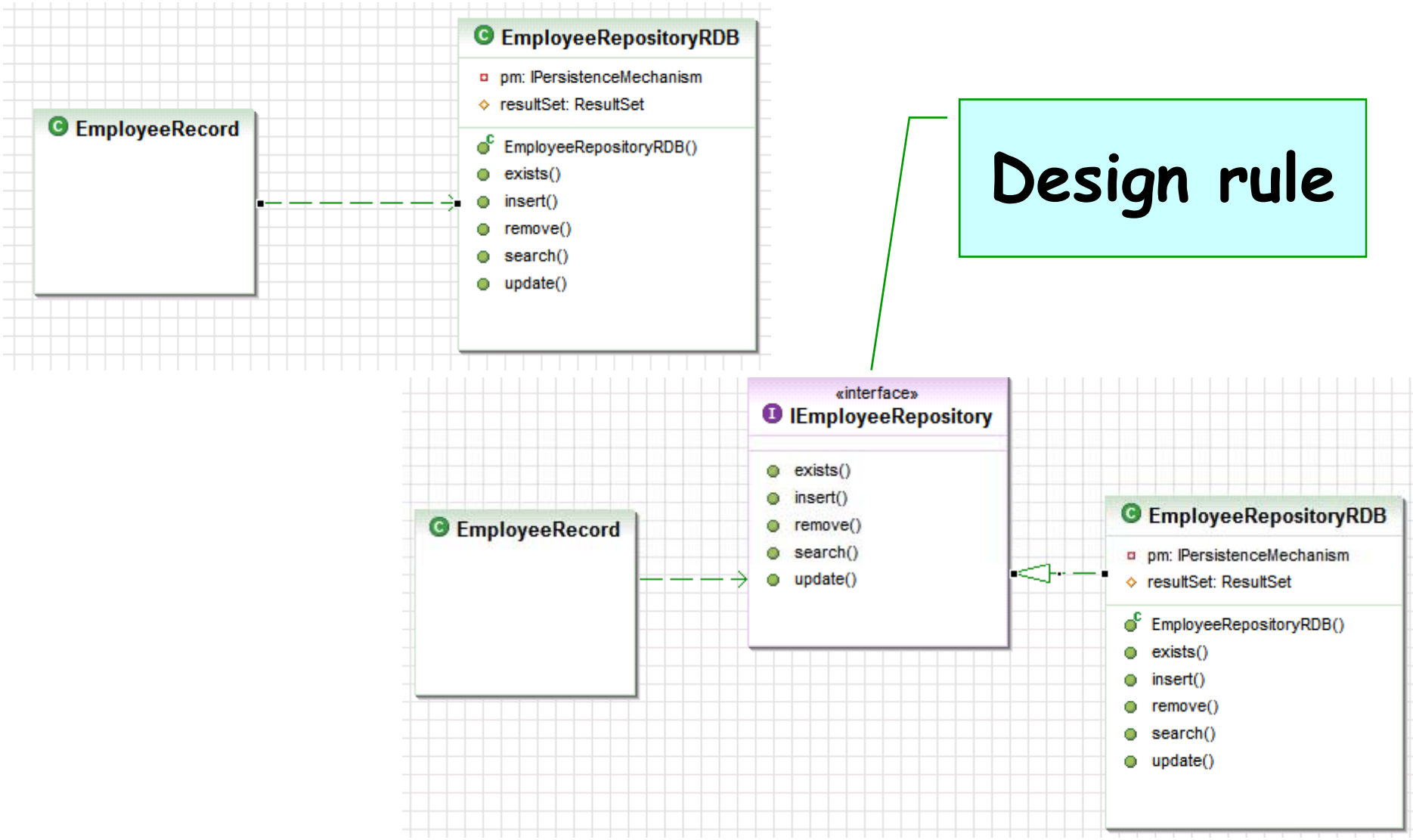
# Information hiding and interfaces (David Parnas, CACM 1972)

Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.

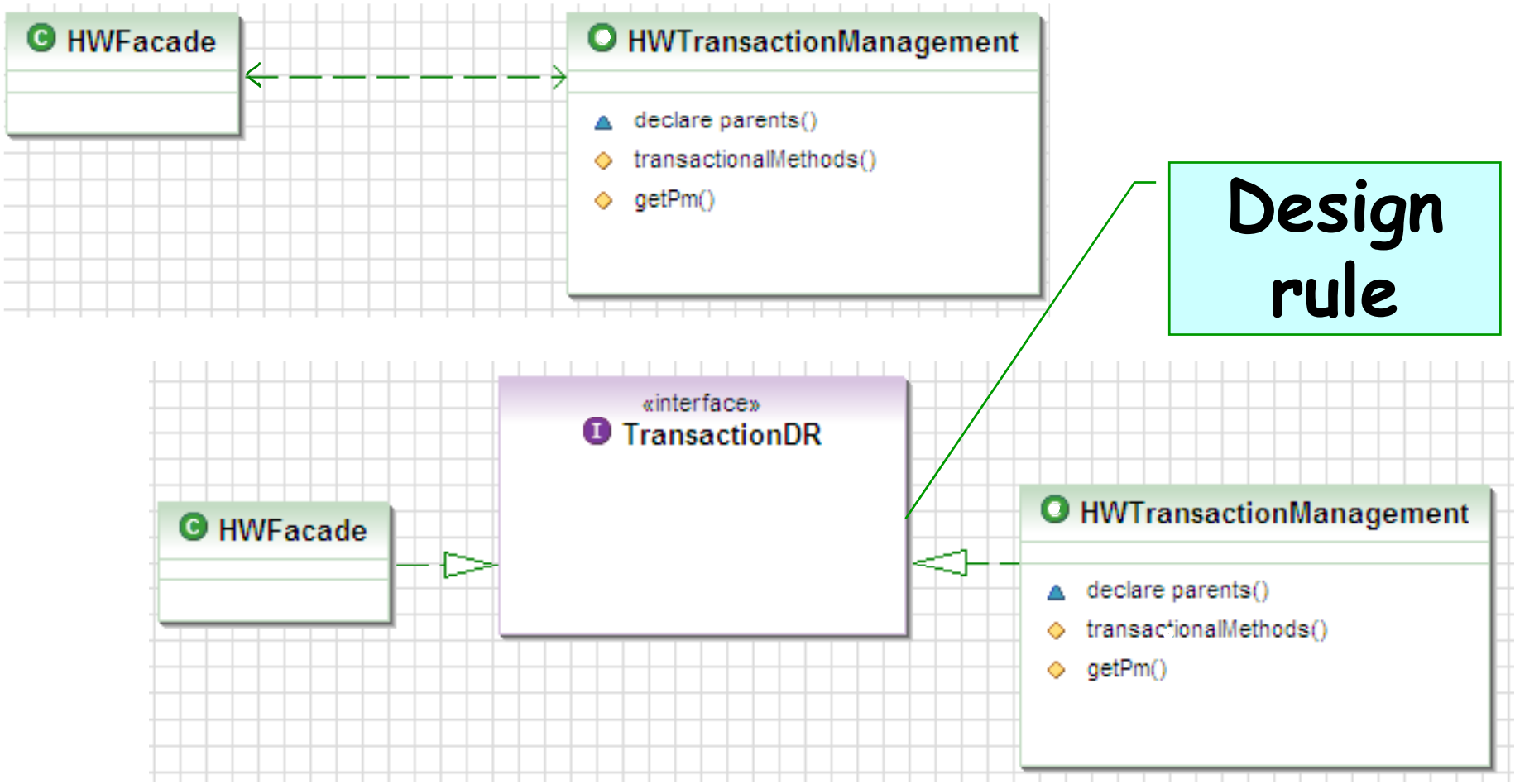
Not necessarily public methods!

Not necessarily data representation!

# Interfaces decouple classes...

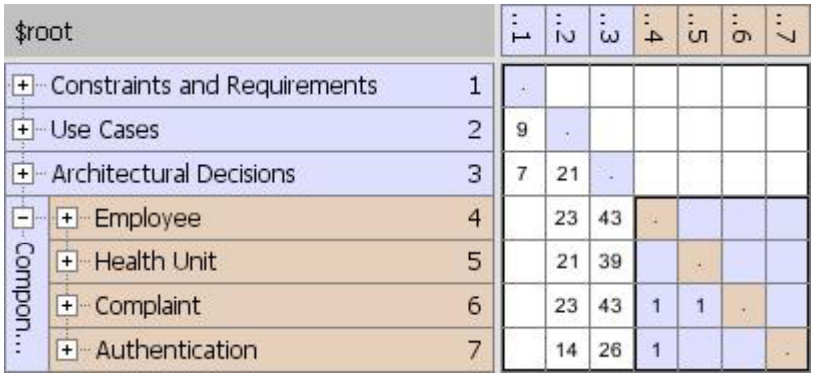


# and aspects too!

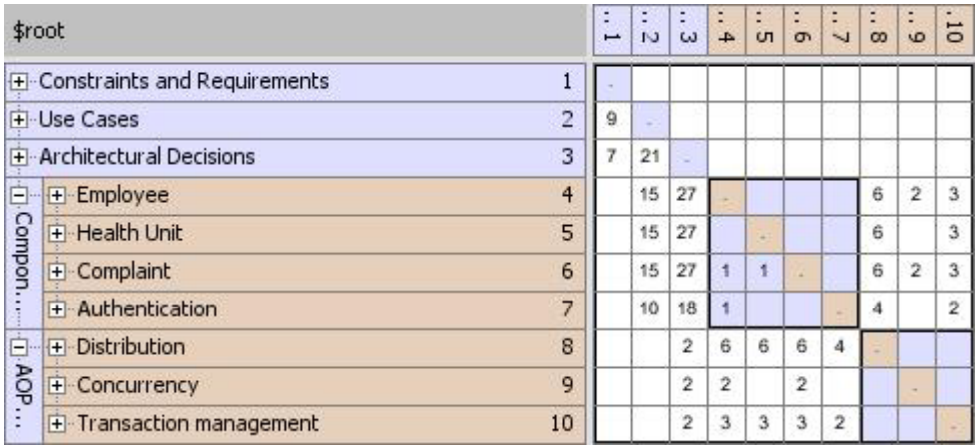


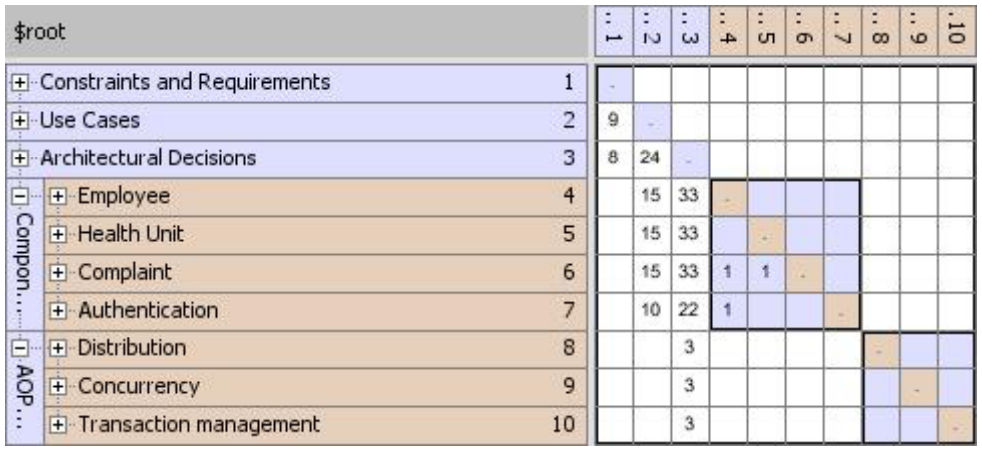
# Reconciling modularity dimensions

## OO



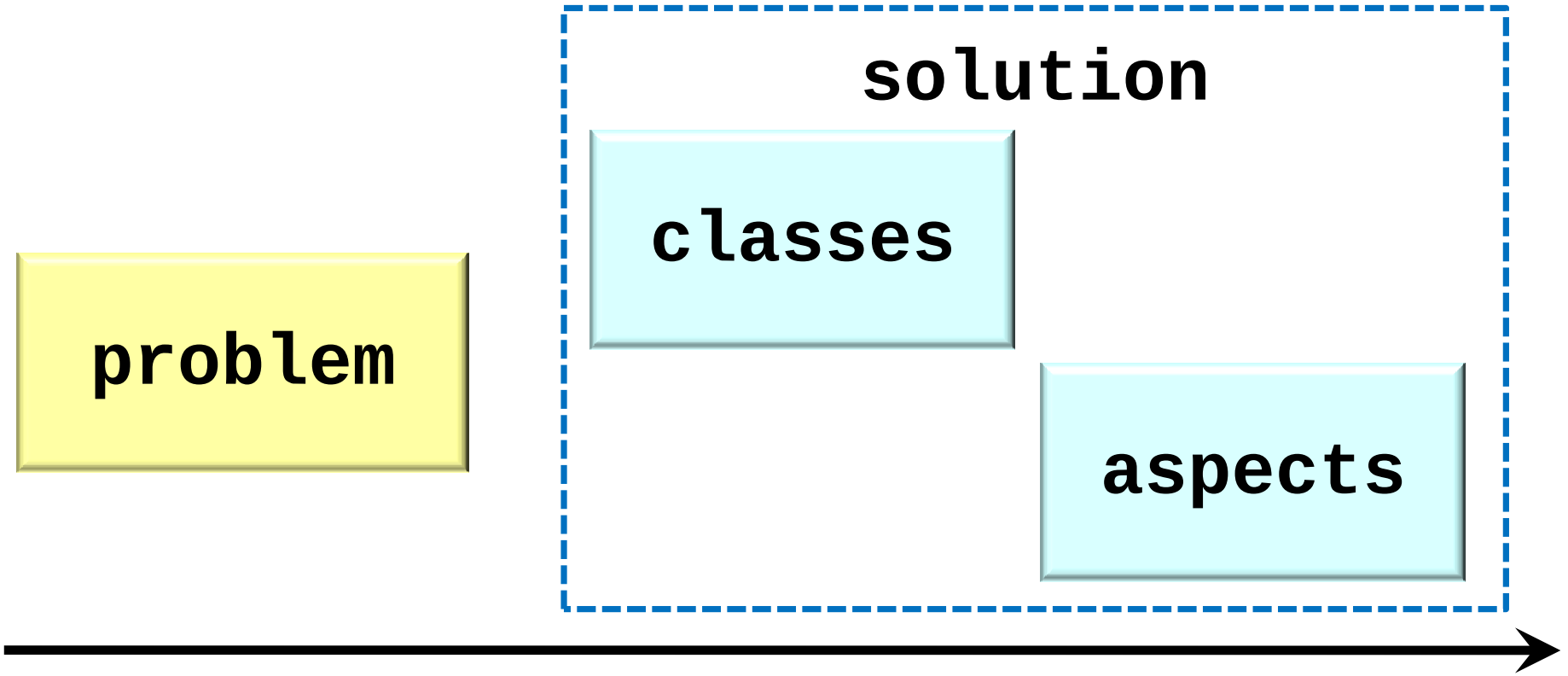

## AO



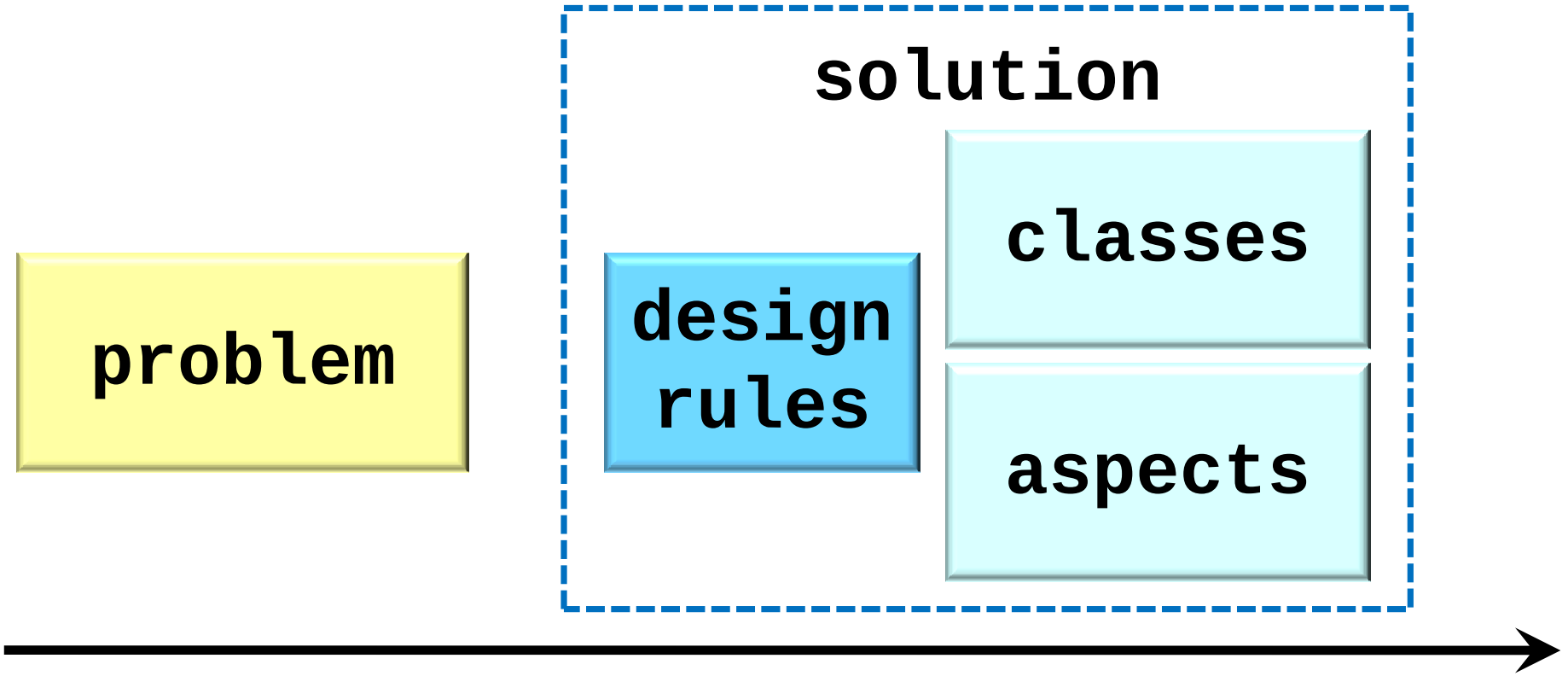




## AO with DRs

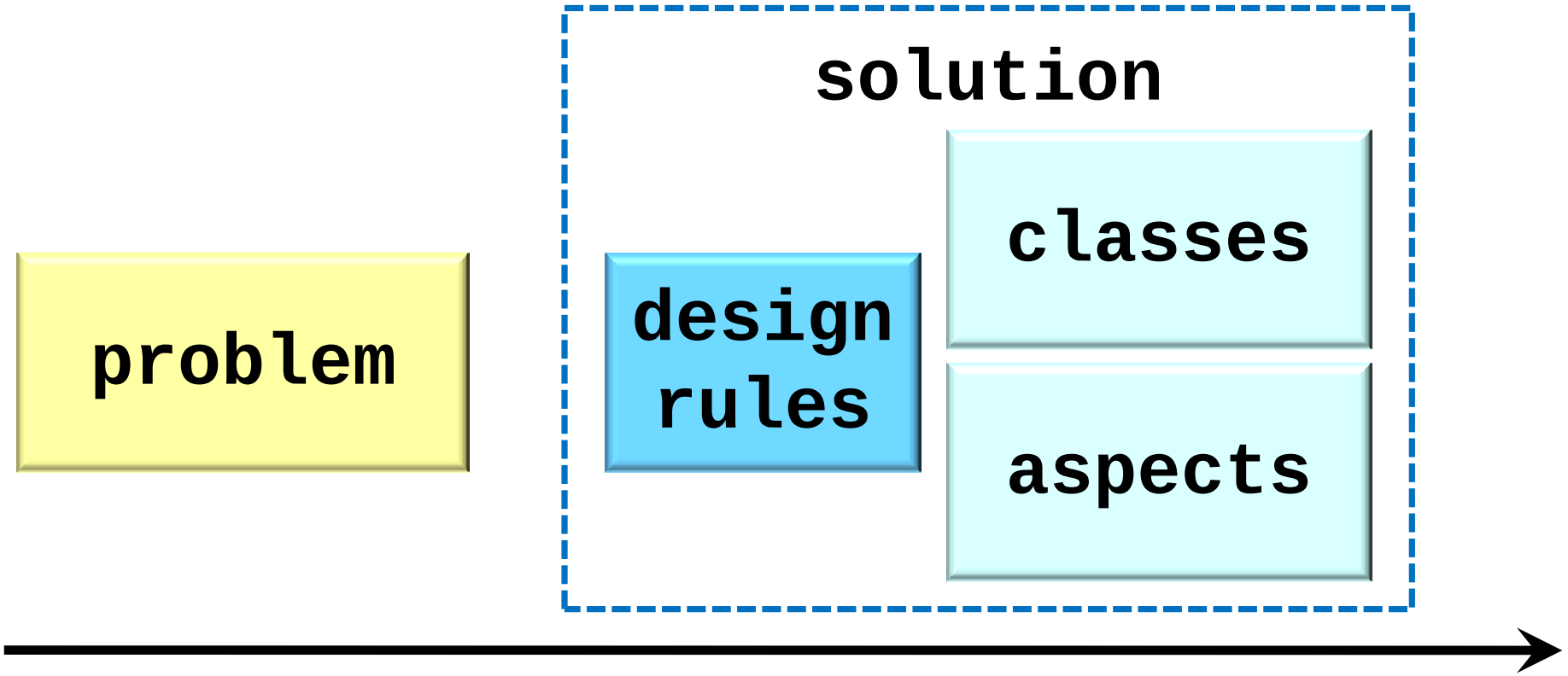
# Sequential development, no need for design rules



# Parallel development, design rules are essential



# Parallel evolution, design rules are also essential

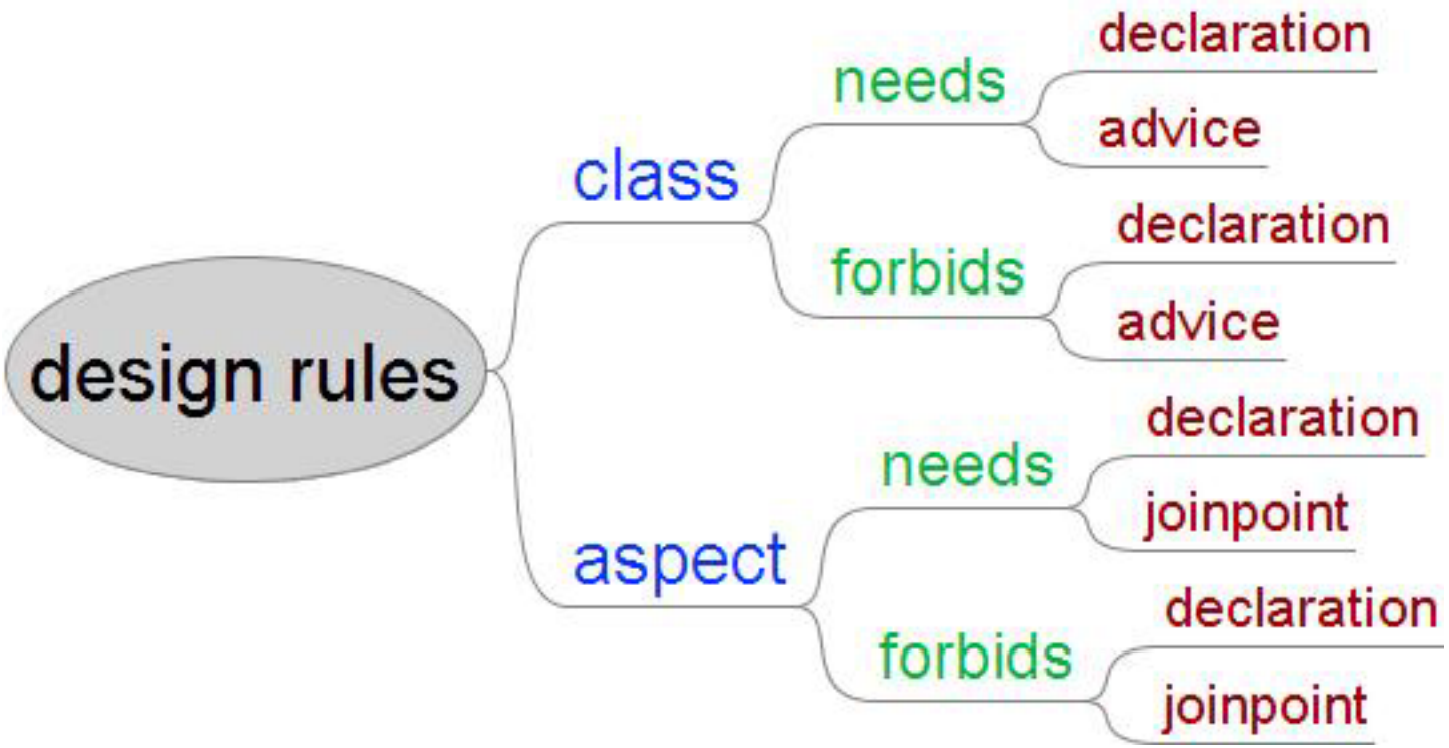


# Design rules behind AOP success! (Soares, Laureano and Borba, OOPSLA 2002)

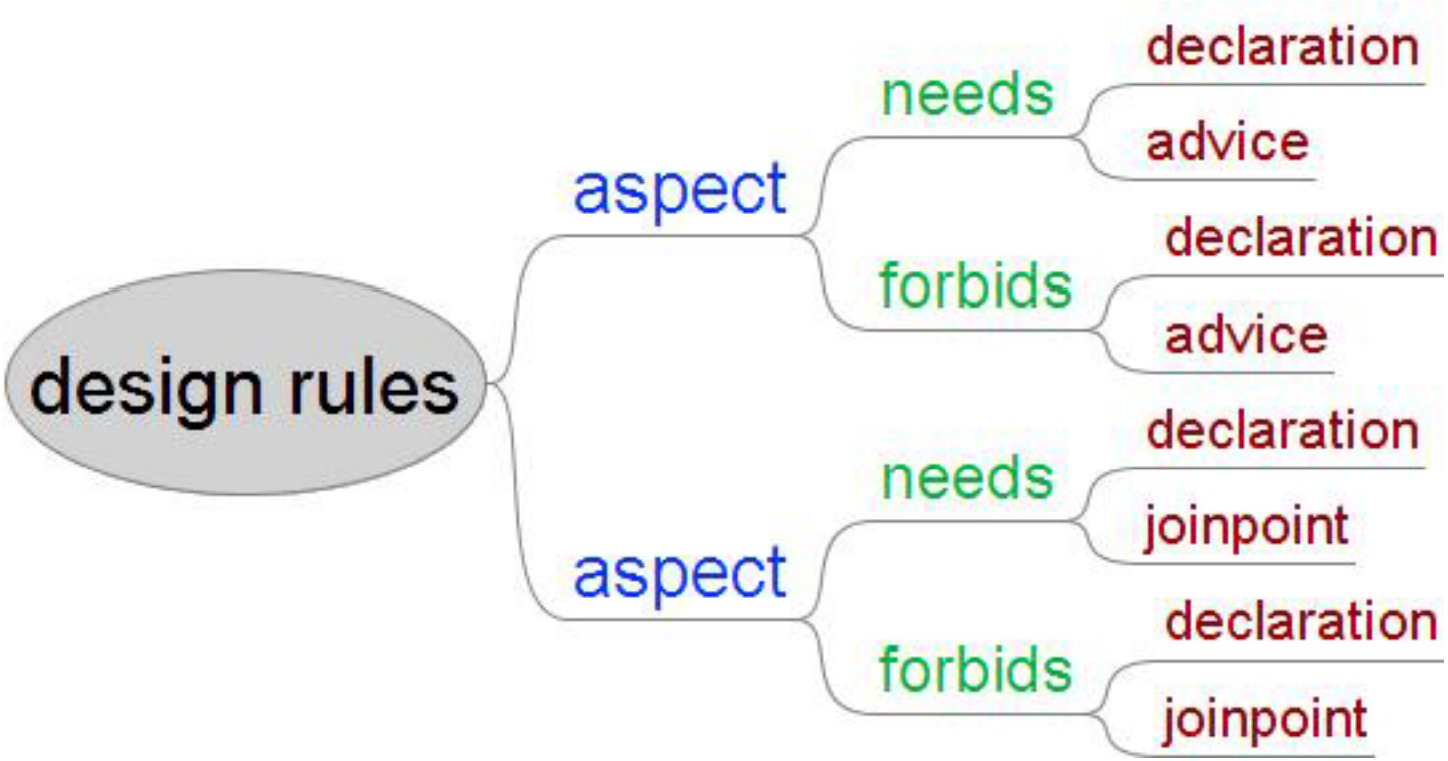
The name conventions simplify the pointcut definitions, but they are not essential. In fact, more complex pointcuts can be defined when naming conventions are not followed.

Moreover, as the definition of a pointcut identifies (by using methods signatures, class names, etc.) specific points of a given system, the aspects become specific for that system, or for systems adopting the same naming conventions, decreasing reuse possibilities.

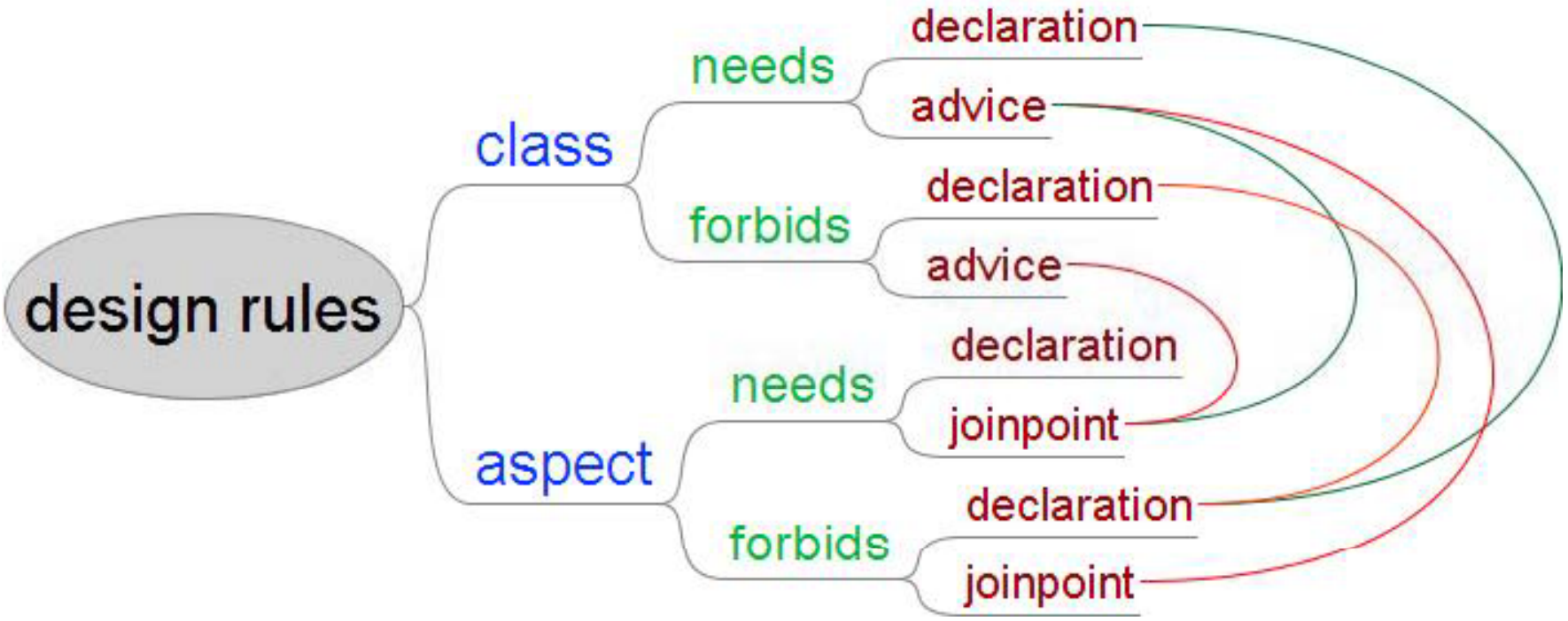
# Identified crosscutting design rules for product lines



# Aspects might intercept aspects too



# Noticing combinations



# Being concise with declarations

```
C needs m && A forbids m

C calls m
C calls m && A calls m

A declares m
```

needs and forbids

calls

declares

# Being concise with behavior too!

- Focus on whom is responsible for implementing behavior
  - Class exposes joinpoints satisfying constraints
  - Aspect intercepts joinpoints implementing behaviour and preserving conditions

# Mechanisms for crosscutting design rules

- Similar to mechanisms for OO
  - Interfaces
  - Specifications
- But generalized in several dimensions
  - and supporting aspect interaction

# Interfaces as partial OO specifications

```
interface I {
  void m(int i);
}
```

```
interface I {
  class ? {
    void m(int i);
  }
}
```

- Unidirectional, one module
- Public methods only
- Hides class name and constructors

# Interfaces as partial AO specifications

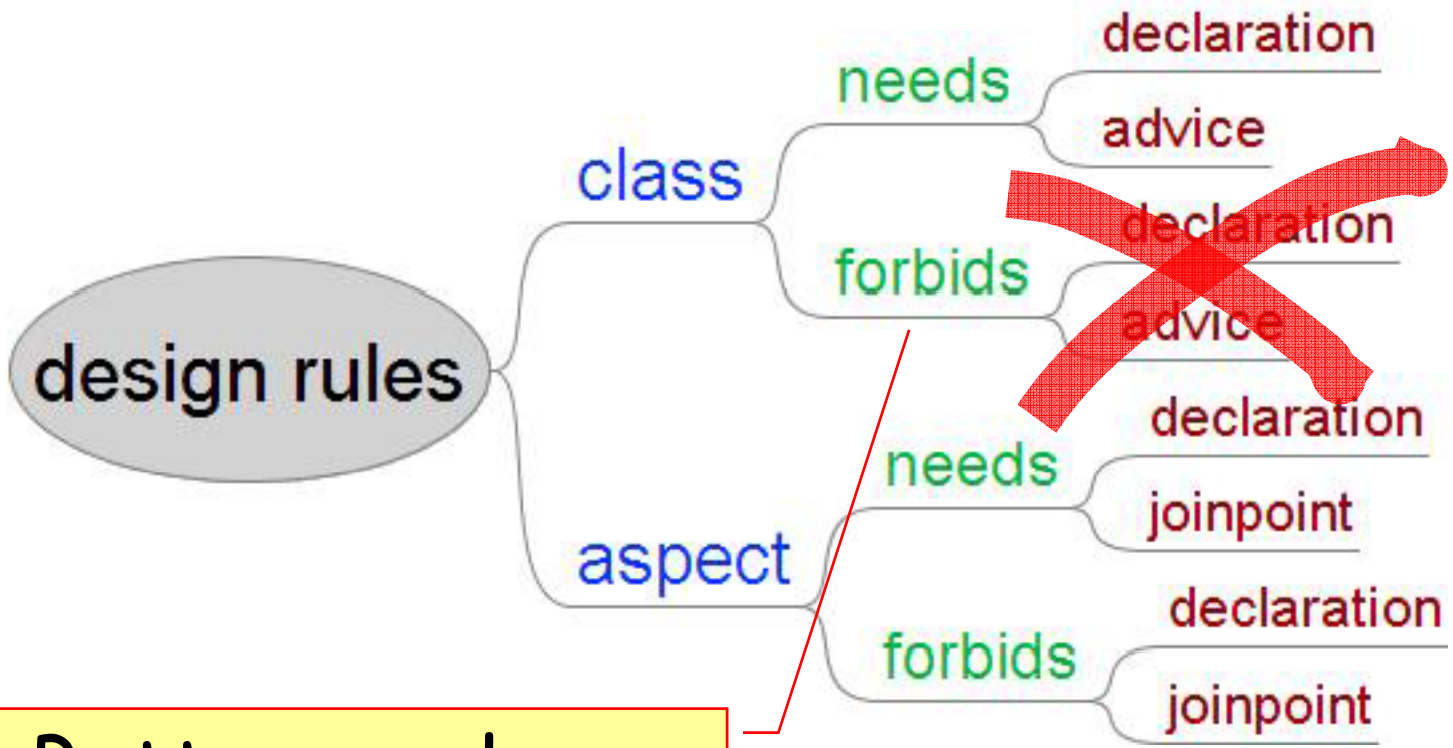
```
dr Synchronization {
  class Employees {}
  aspect SynchronizationController {
    pointcut critical(Employees c):
          execution(* Employees.*(..))
      && this(c);
  }
}
```

# No overspecification!

```
aspect Manager {
  pointcut critical(Complaint c):
      execution(*
         Complaints.*(Complaint))
    && args(c);
}
}
```

```
perthis(critical(Complaint))
```

# Supported crosscutting design rules




Better as class operators

# Related work

- Other AO interfaces
  - XPIs [Sullivan et al]
  - Crosscutting interfaces [Chaves et al]
  - Aspect aware interfaces [Kickzales and Mezini]
  - Open modules [Aldrich]

# Conclusion

Aspects and design rules
support both

**crosscutting modularity**

and

**class modularity...**