Migrating Object-Oriented Software to Component-Based Ones

Présentation extraite de la soutenance de thèse de

Zakarea Al Shara

Evolution of Large OO Software Systems

- Example: in Mozilla.org
  - Composed of **30K classes**
  - More than **1 million changes**
  - Performed by **hundreds of developers**
  - Over more than **6 years**


Do not have explicit architecture

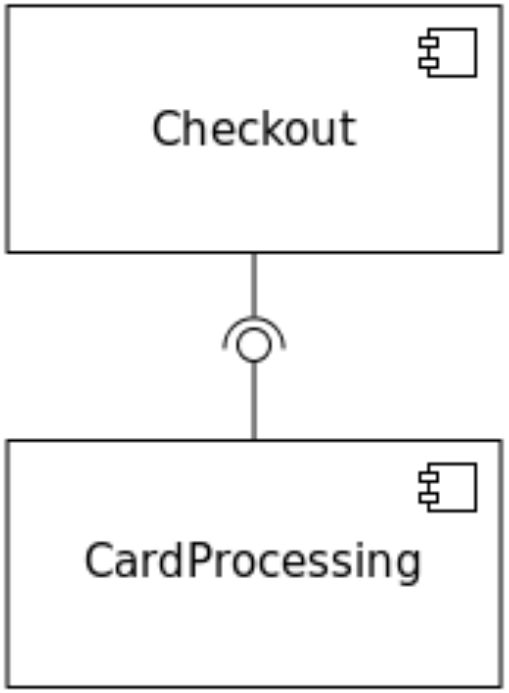

Fine-grained entities (objects)


Numerous implicit dependencies


Hard to maintain, understand and reuse

Characteristics of CB Software

- Component-Based Software Engineering
  - Definition: an approach for developing software systems by choosing **off-the-shelf software components** and then **assembling** them using a **well-defined software architecture**
- **Software component** characteristics
  - Coarse-grained entity
  - Encapsulated
  - Composable
- Easy to maintain, understand and reuse



Strategy Solutions for Software Evolution

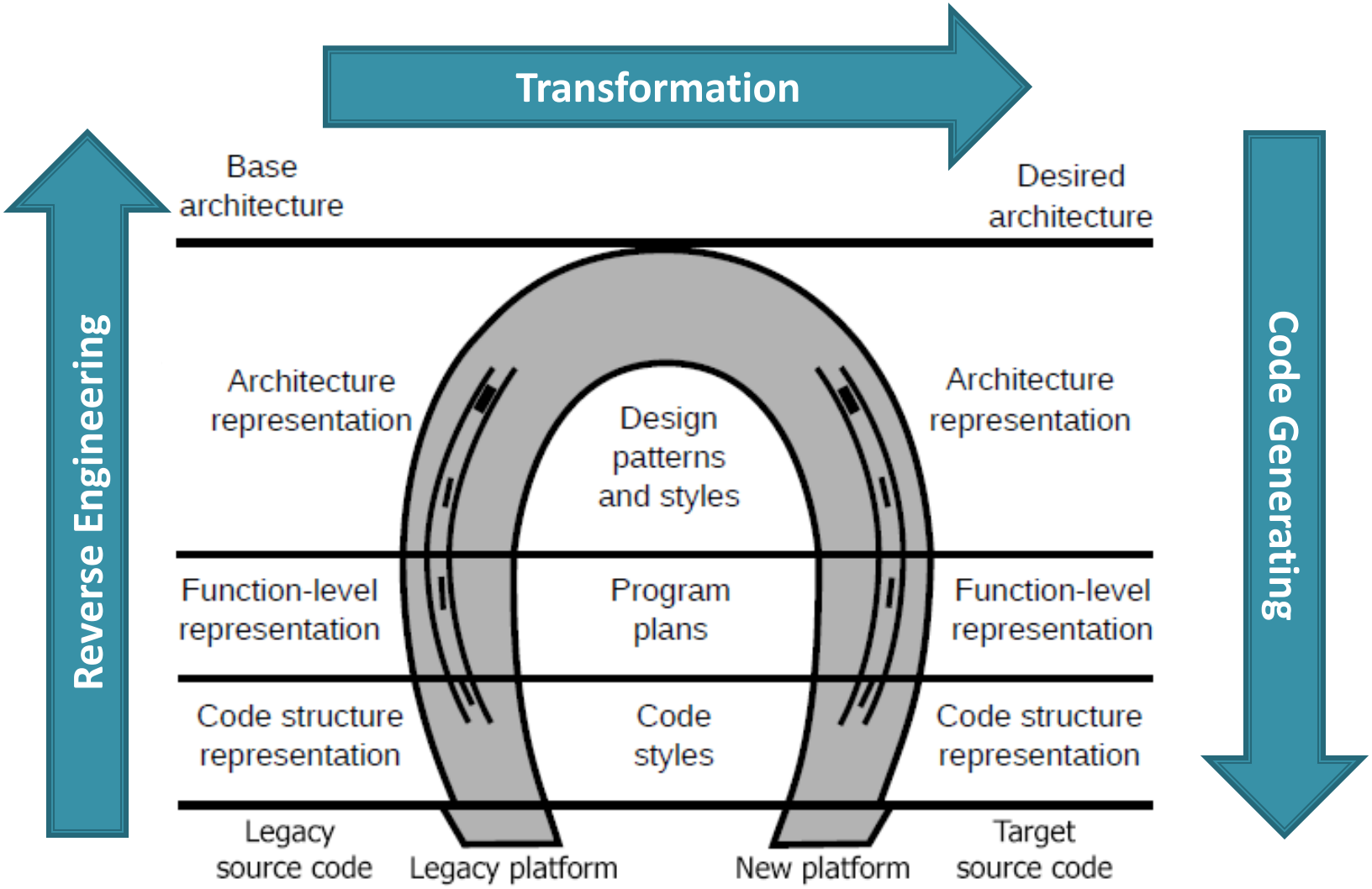
1. ✗ Replacing the software
2. ✗ Continuing maintenance despite cost

3. ✔ Software migration
   - Allows to move from original form to a new one
   - Keeps same functionality and data
   - Without having to completely redevelop the software

Cost effective

- Migrating OO software to CB ones:
  - Improve understandability, maintainability and reusability

Steps of Software Migration

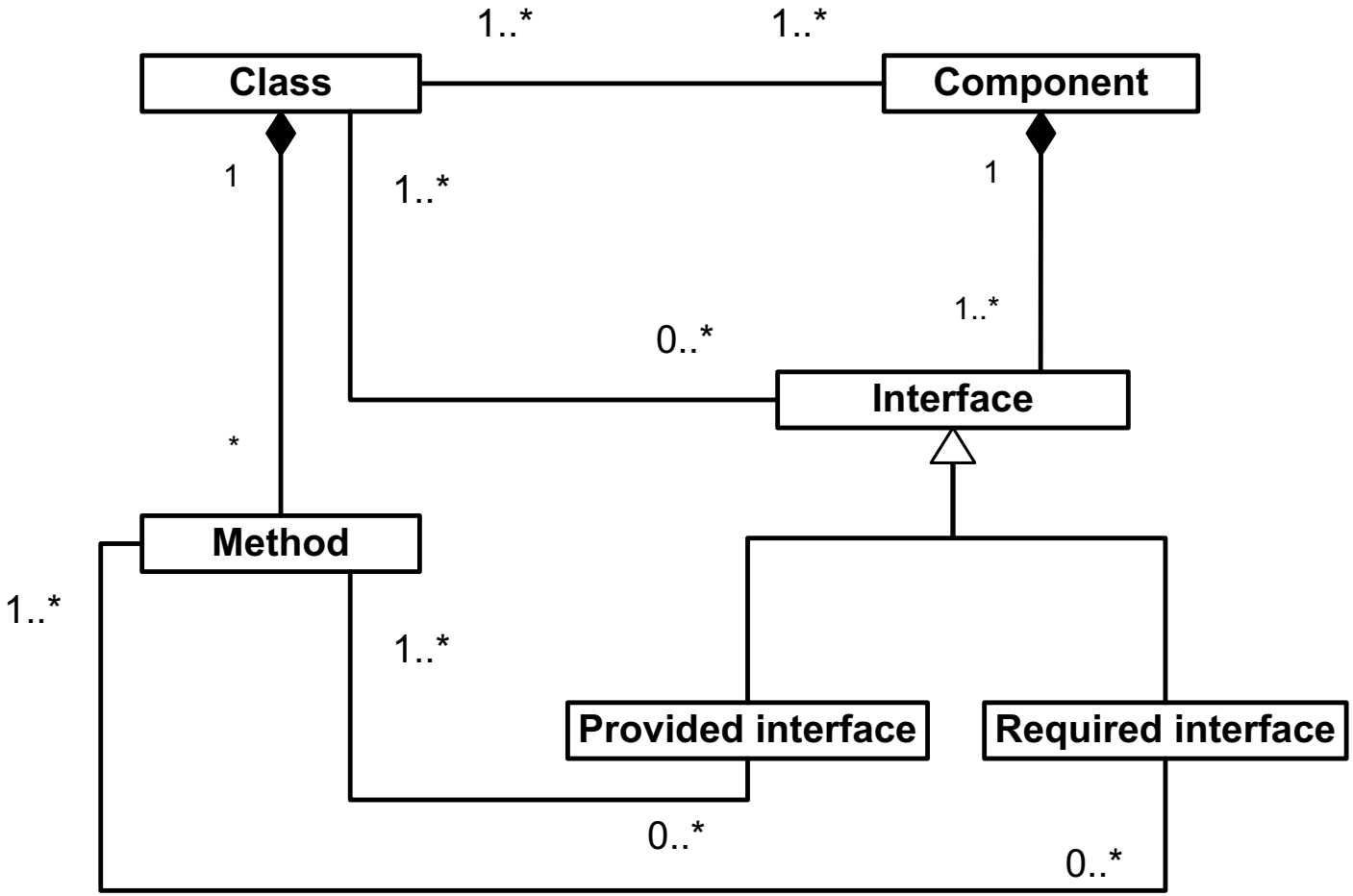


Steps of Migrating OO Software to CB Ones

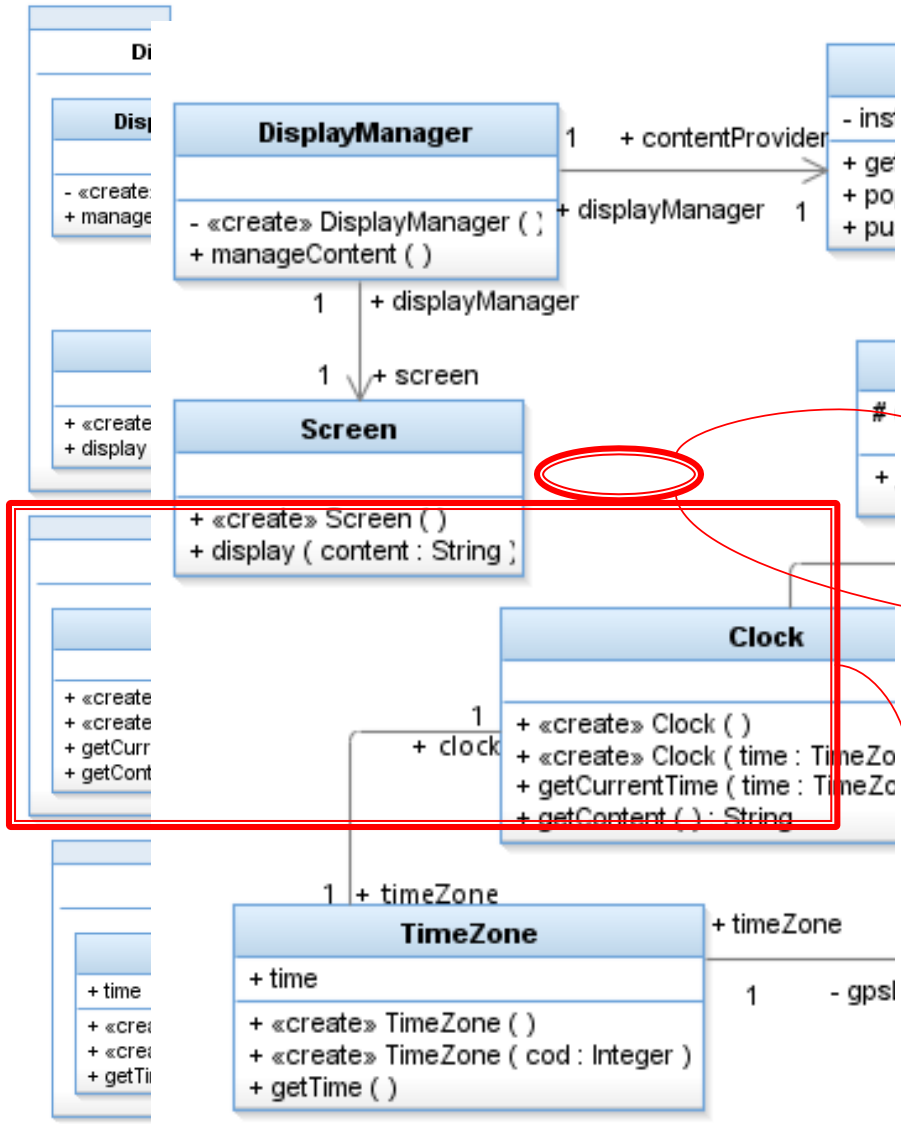
1. Reverse engineering: CB architecture recovery
   - Identifying components and their inter-relationships

2. Transformation: from OO code to CB code
   - Transforming OO code to create programming level components

OO-to-CB Mapping Model



Toy Example: Information Screen




```
//DisplayedInformation and ContentProvider
//by Darwin ADL

interface I1{
  long : getCurrentTime(TimeZone : time)
  String : getContent()
}

interface I2{
  String : getContent()
}

Component information{
  Require I1, I2
}

Component content{
  Provide I1, I2
}

Component information_screan{
  inst //instantiate component instances
  DisplayedInformation : information
  ContentProvider : content
  ... // other component instances
  bind
  information.I1 -- content.I1
  information.I2 -- content.I2
  ... // other bindings
}
```

Healing Component Encapsulation

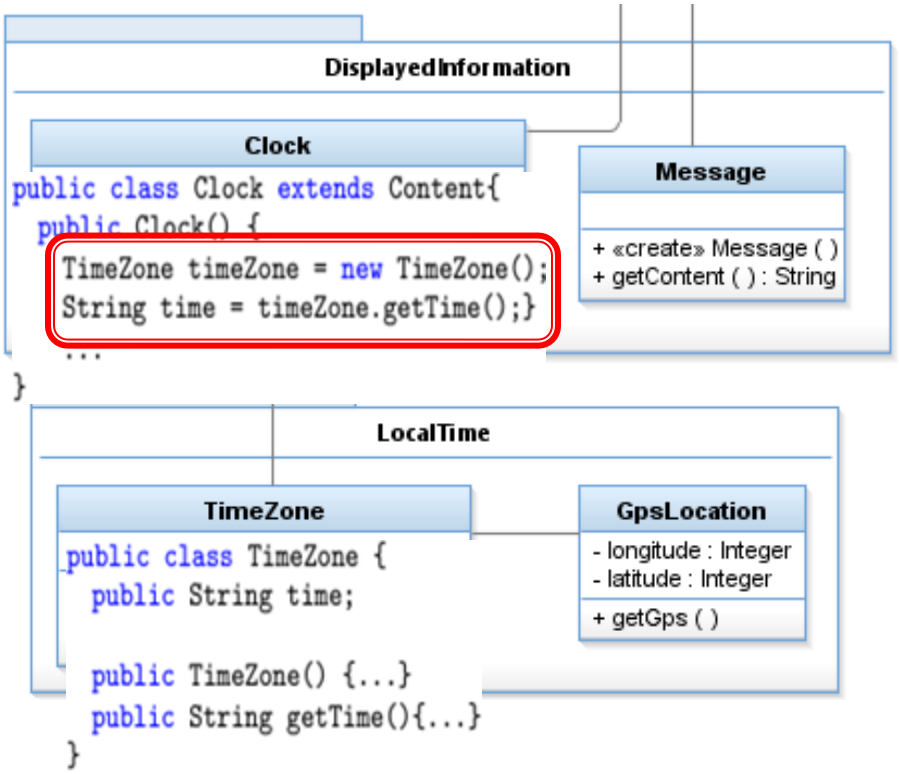
Healing Component Encapsulation

- Input: CB architecture recovery
- The transformation needs to solve two main problems:

1. Explicit component encapsulation violation
   - Explicit dependencies between components caused by direct access to its internal implementation
   - i.e. Instantiation and method invocation
2. Implicit component encapsulation violation
   - Implicit dependencies between components caused by OO mechanisms
   - i.e. Inheritance and exception handling

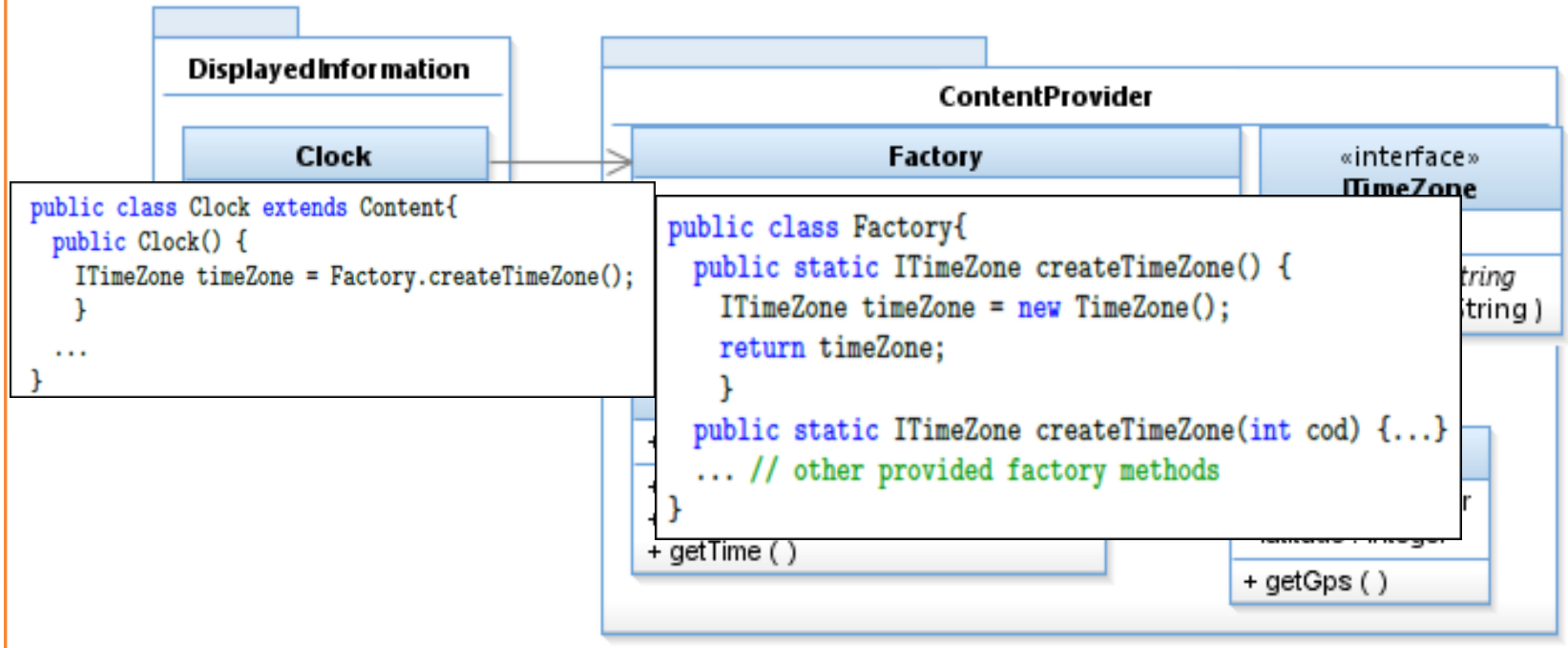
Instance Handling Problem

A class belonging to a component (cluster) can be directly instantiated/called in a method of class belonging to another component



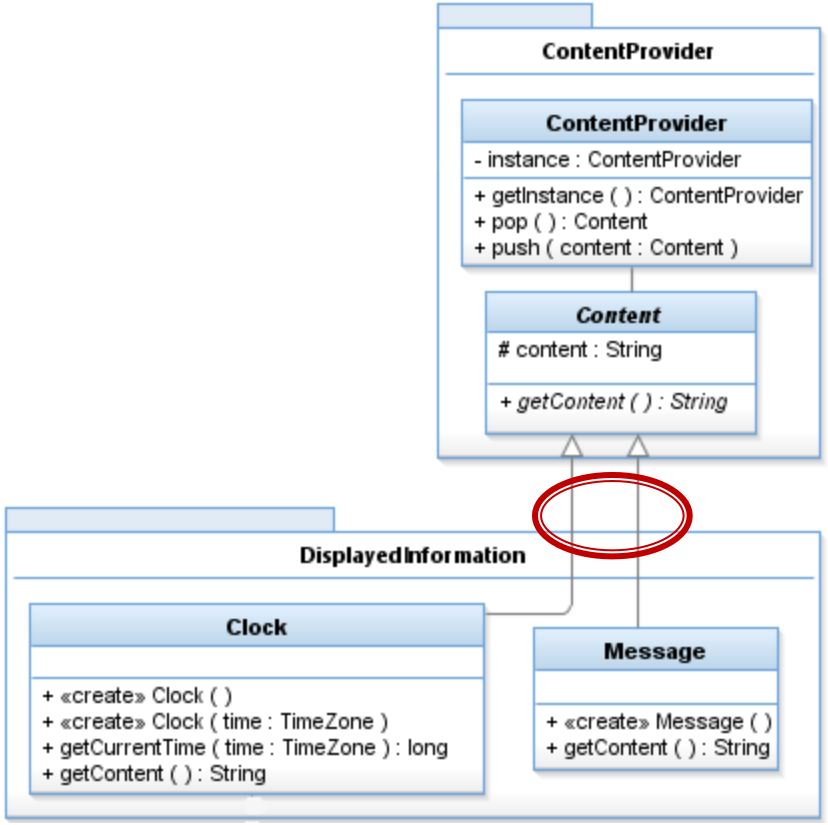
Solution: Using Component Interfaces through the Factory Pattern

1. Uncoupling classes by creating object interfaces
2. Implement factory design pattern to provide object interfaces



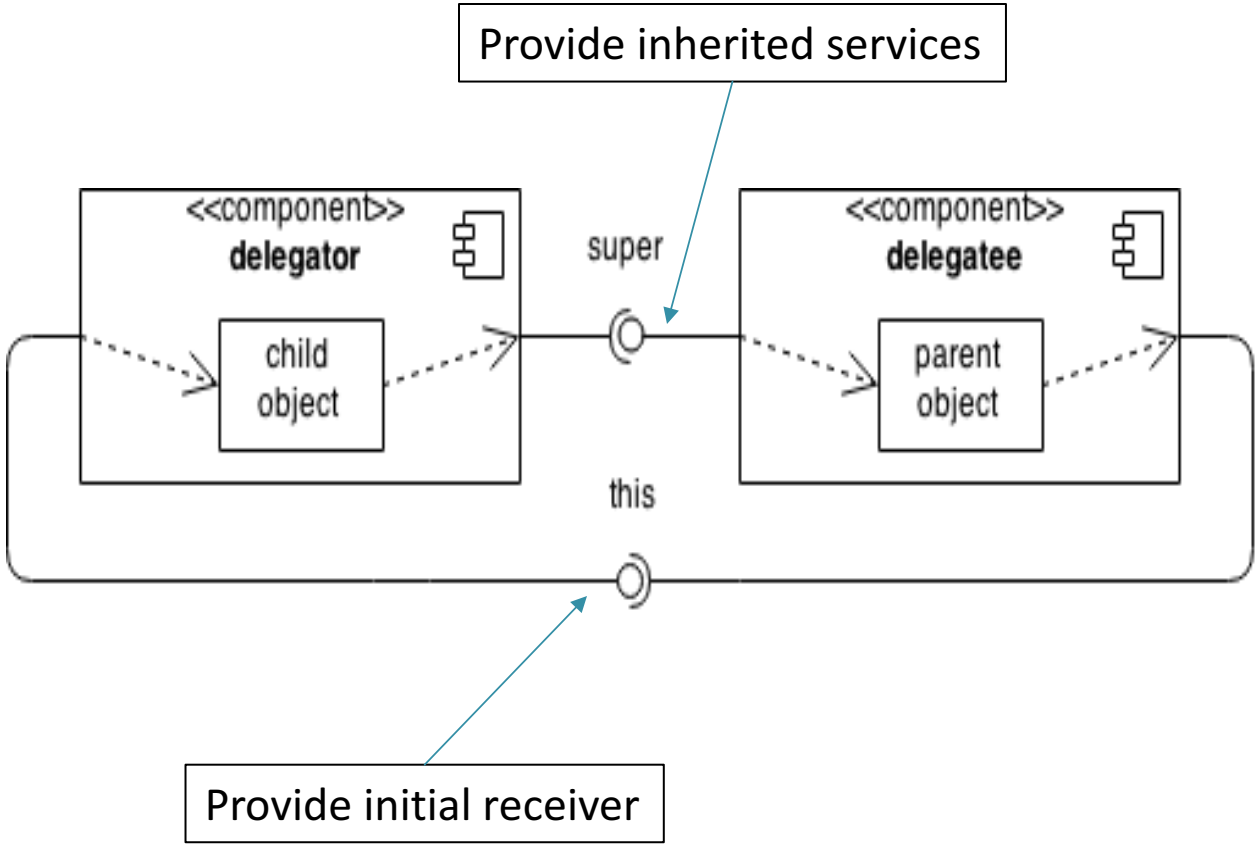
Inheritance Problem

- Inheritance links between classes belonging to different components need to be transformed

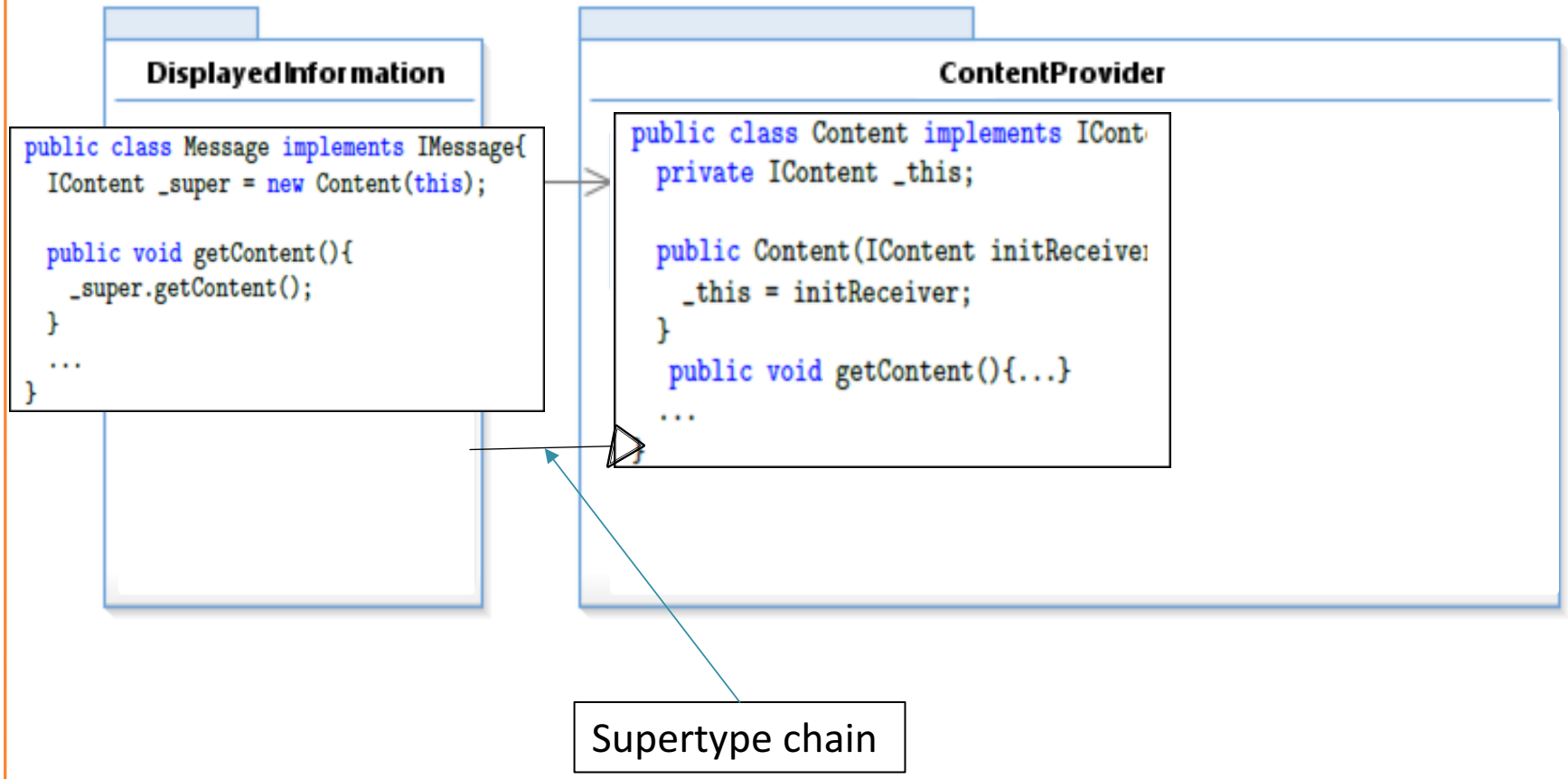


Solution: Replacing Inheritance by Delegation

- Replace inheritance with delegation between components

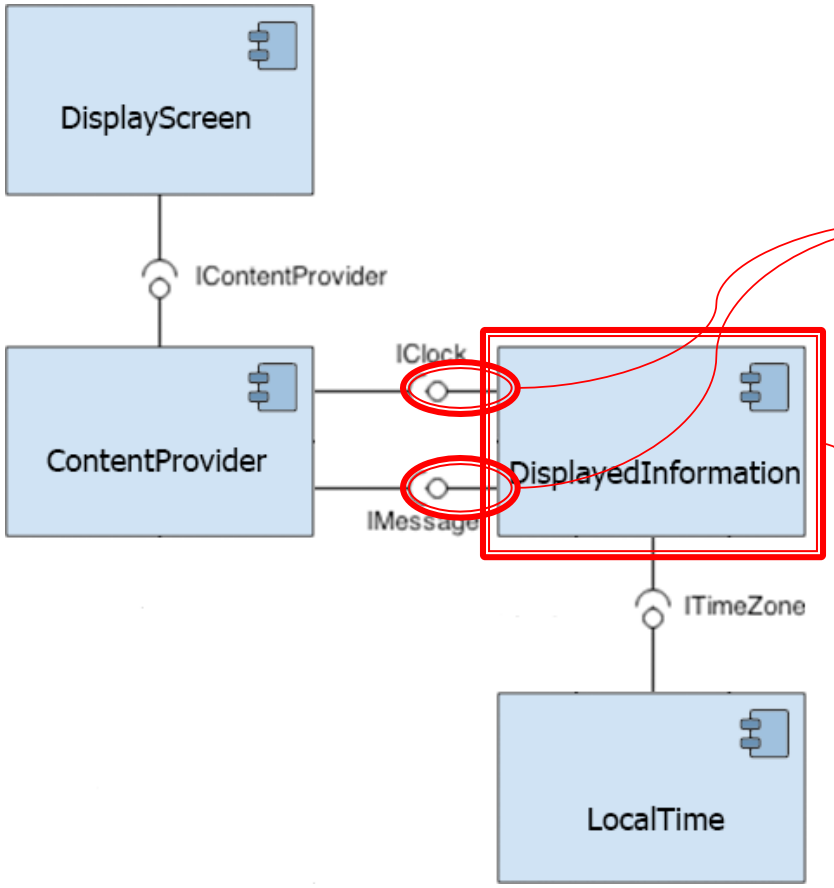


Solution: Replacing Inheritance by Delegation



Reveal Component Instance

Reveal Component Instance




```
//DisplayedInformation and ContentProvider
//by Darwin ADL

interface I1{
  long : getCurrentTime(TimeZone : time)
  String : getContent()
}

interface I2{
  String : getContent()
}

Component information{
  Require I1, I2
}

Component content{
  Provide I1, I2
}

Component information_screan{
  inst //instantiate component instances
  DisplayedInformation : information
  ContentProvider : content
  ... // other component instances
  bind
  information.I1 -- content.I1
  information.I2 -- content.I2
  ... // other bindings
}
```

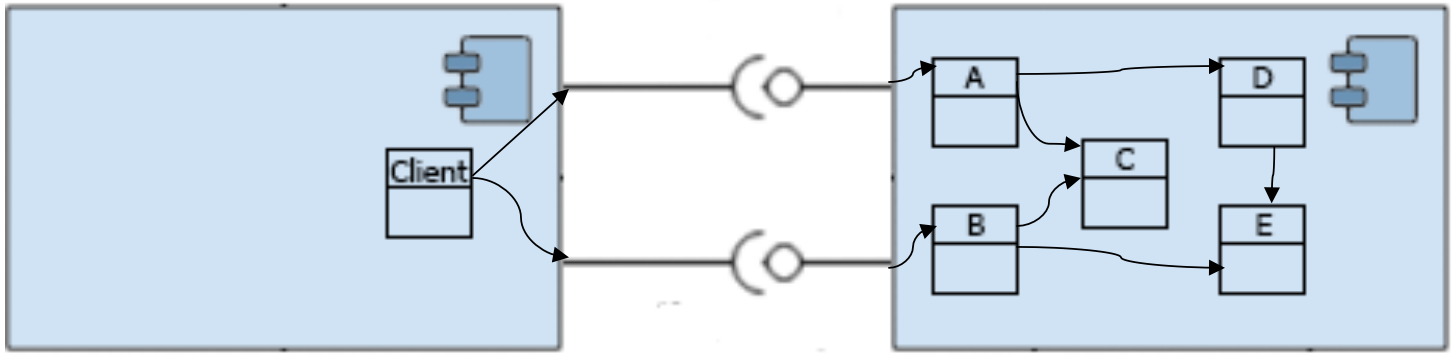
Problem and Motivation

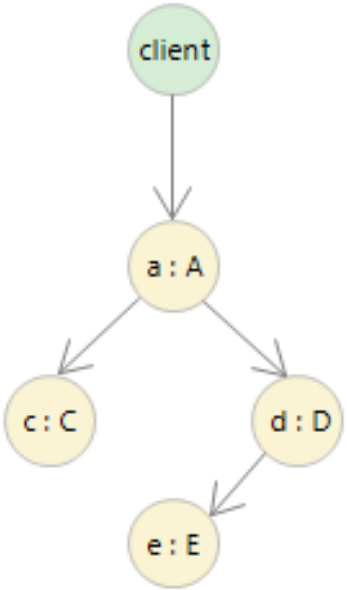
- A cluster should not be considered as simple packaging and deployment units
- Moving from the concept of object to a concept of component instance.
- Solving the gap between CB architecture and its running components

Component Instance

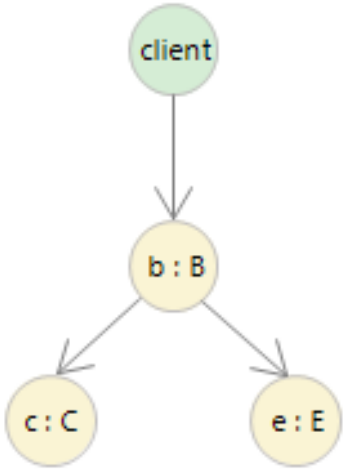
- In OO, an instance consist of **state** (attributes) and **behavior** (reified by methods)
- Infer component instance from a set of classes instances belonging to the same component
  - A component state is the aggregated state of these instances
  - A component behavior is published through the component interfaces

Component Instance

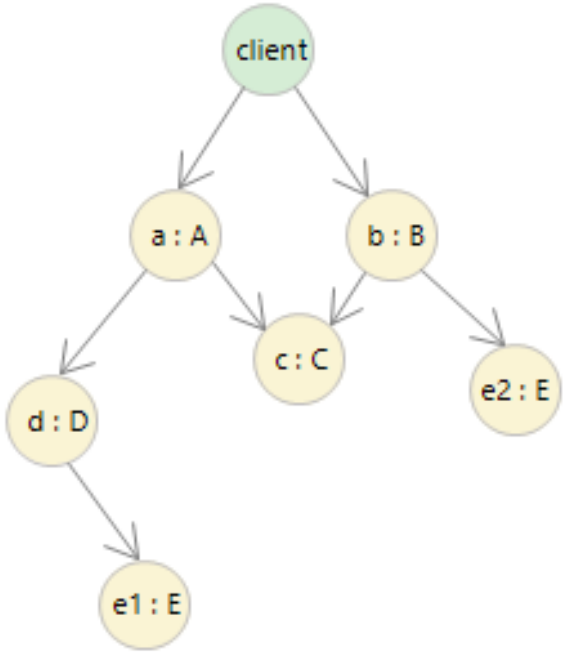




(a)



(b)



(c)

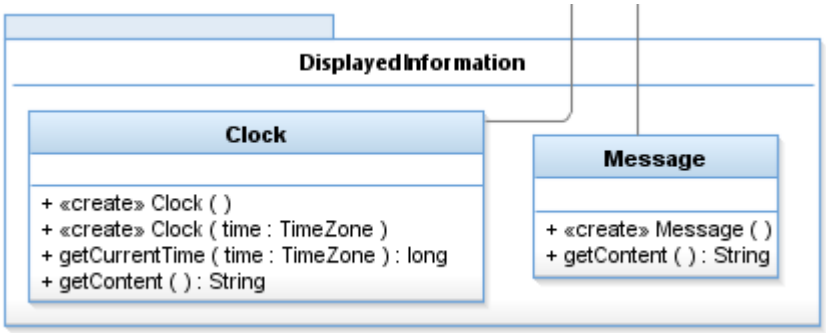
Solution: Component Descriptor

Component descriptor consists of:

1. **Component interfaces**: the component descriptor needs to define provided and required interfaces
2. **Implementation reference**: the component descriptor needs to define references of its component implementation source code
3. **Component instantiation**: the component descriptor needs to describe how its component is instantiated

Component Descriptor: interfaces

```
public class DisplayedInformation{

  public static ITime portTime;
  public static IMessage portMessage;

  private class PortTime implements ITime{
      @Override
      public String getContent() {
      //TODO: add behaviore implementation
      }

        @Override
      public long getCurrentTime(ITimeZone timeZone) {
      //TODO: add behaviore implementation
      }
  }

  private class PortMessage implements IMessage{
      @Override
      public String getContent() {
      //TODO: add behaviore implementation
      }
  }

}
```



Component Descriptor: Implementation reference

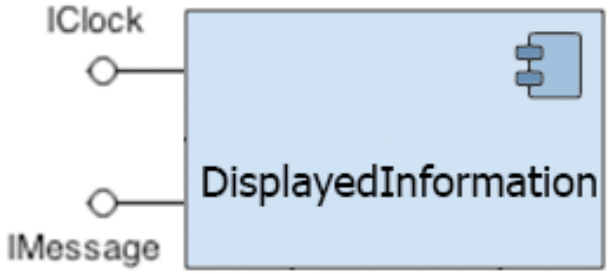
Delegating provided services

```
public class DisplayedInformation{
    ...
  private class PortTime implements ITime{

    @Override
    public String getContent() {
      if(clock == null){ //lazy instantiation
        clock = new Clock();}
      return clock.getContent();
    }

    @Override
    public long getCurrentTime(ITimeZone timeZone) {
      if(clock == null){ //lazy instantiation
        clock = new Clock();}
      return clock.getCurrentTime(timeZone);
    }
  }


  private class PortMessage implements IMessage{

    @Override
    public String getContent() { //lazy instantiation
      if(message == null){
        message = new Message();}
      return message.getContent();
    }
  }

}
```

Component Descriptor: instantiation

- How component instances can be created?
  - Component constructors
    1. Default constructor
    2. Initializing component state

```
public class DisplayedInformation{
...
  public DisplayedInformation() {
    //initializing component ports
    portTime = new PortTime();
    portMessage = new PortMessage();
  }

  public initialize(ITimeZone timeZone) {
    clock.setTimeZone(timeZone);
  }
}
```



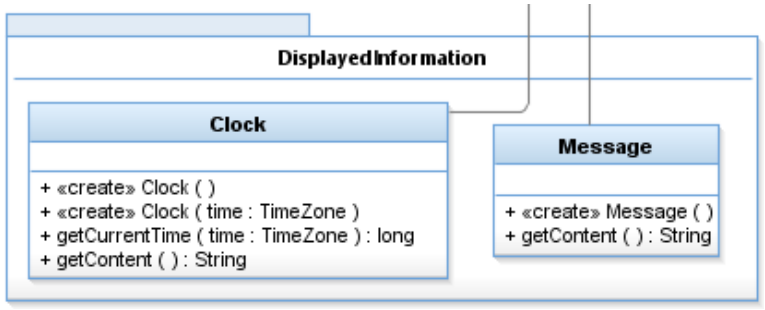
Component Descriptor: instantiation

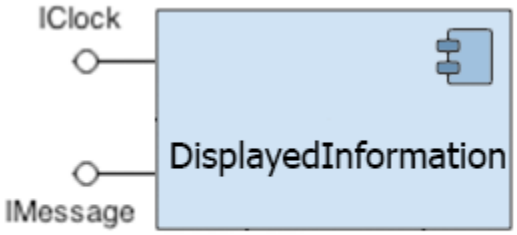
Before transformation

```
Clock clock = new Clock(timeZone);
clock.getCurrentTime();
```

After transformation

```
DisplayedInformation info = new DisplayedInformation();
info.initialize(timeZone);
info.portTime.getCurrentTime();
```

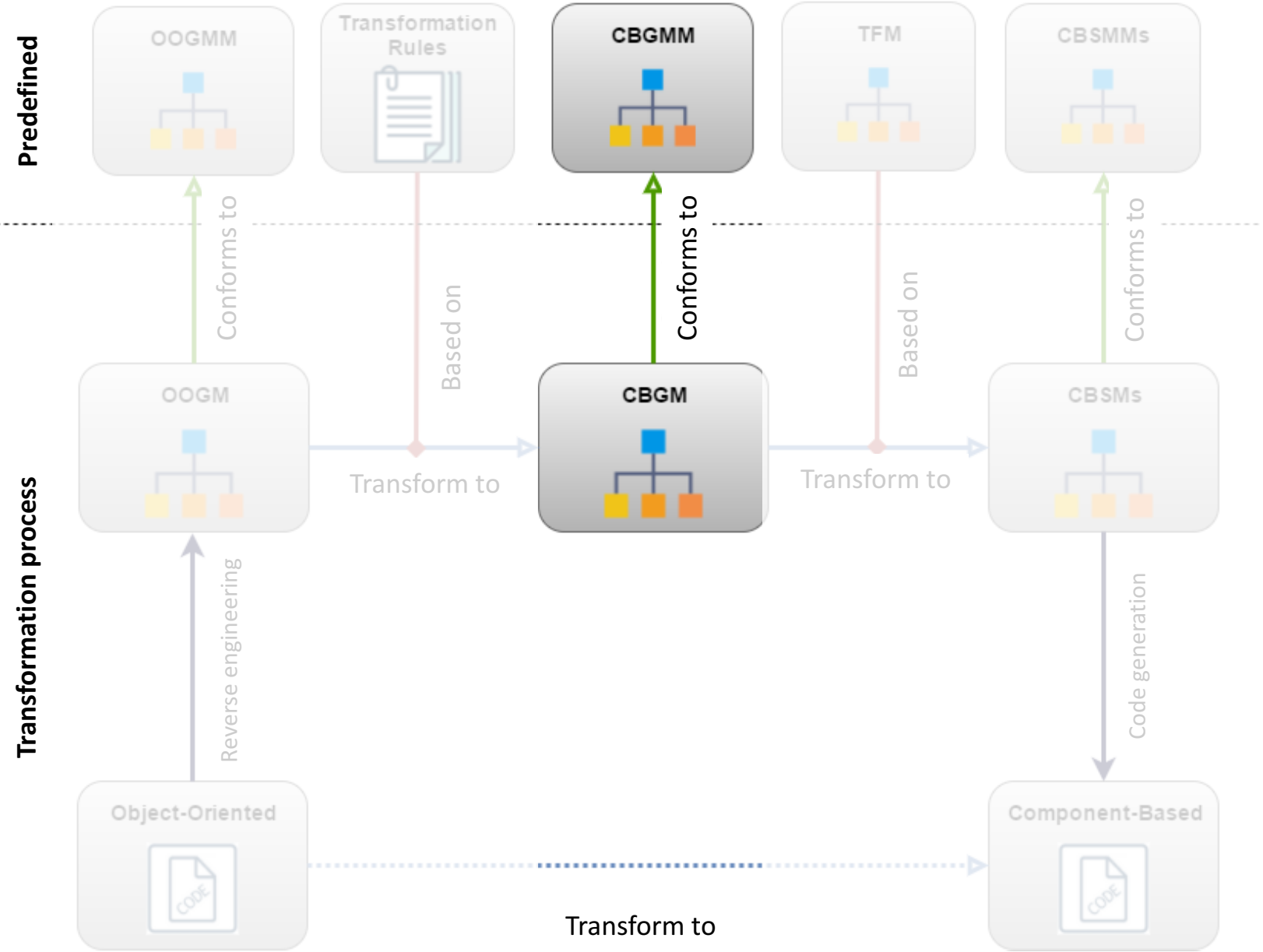




Outline

Model-Driven Transformation: OO Models to CB Models

MDT: OO Models to CB Models



Transforming OOGM to CBGM

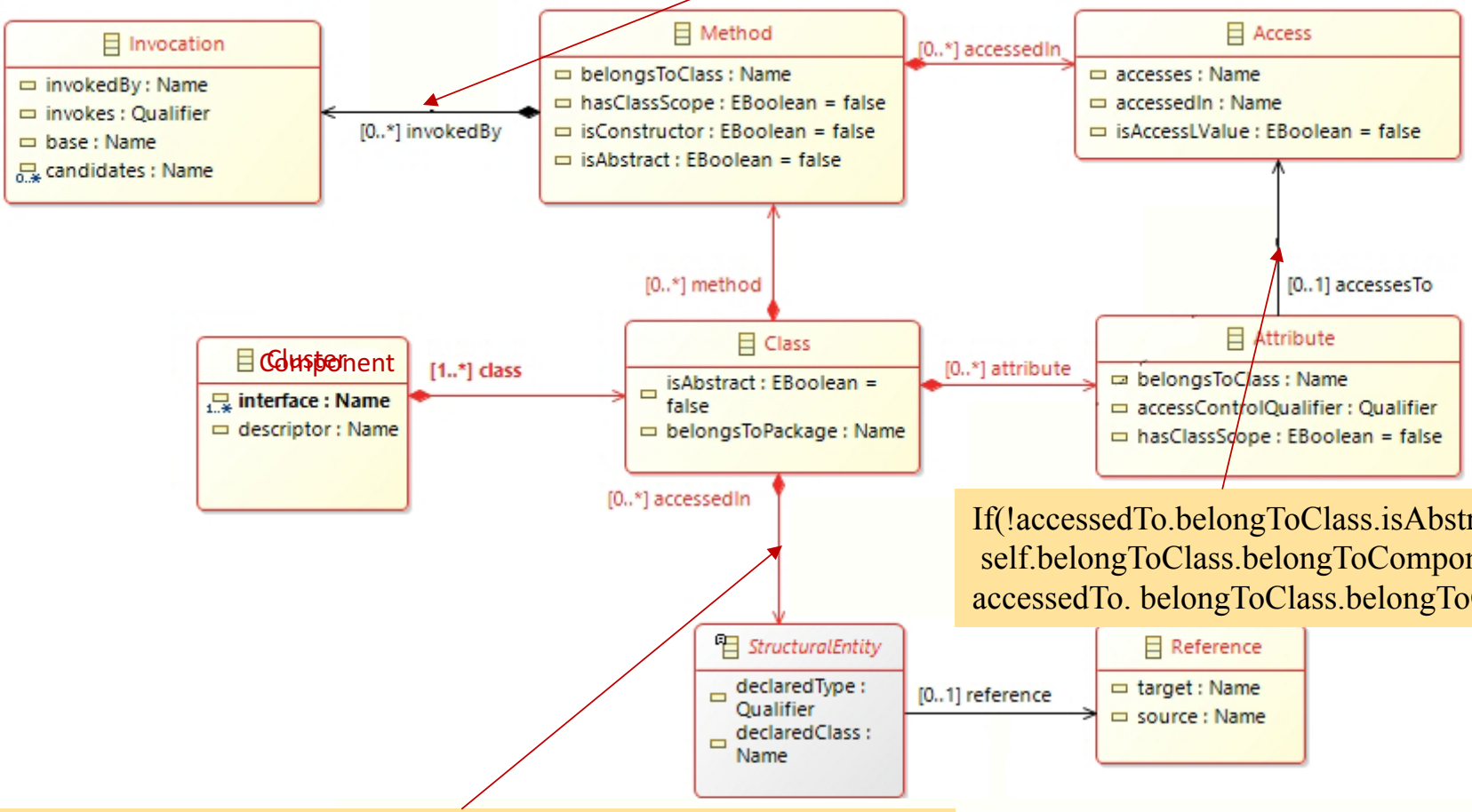
1. Metamodeling: Defining OOGMM and CBGMM
   - **FAMIX** is a family of metamodels for object-oriented languages
     - Extensible
     - Language independent
     - Existing parsing technology to export the meta information of OO languages to FAMIX (e.g. JFamix for Java)

2. Rules for transforming OOGM to CBGM
   - Define the transformation rules to transform OO dependencies to interface-based ones.

INSTANTIATION

OBGMM

If(!invokedBy.belongToClass.isAbstract){
self.belongToClass.belongToComponent == invokedBy. belongToClass.belongComponent

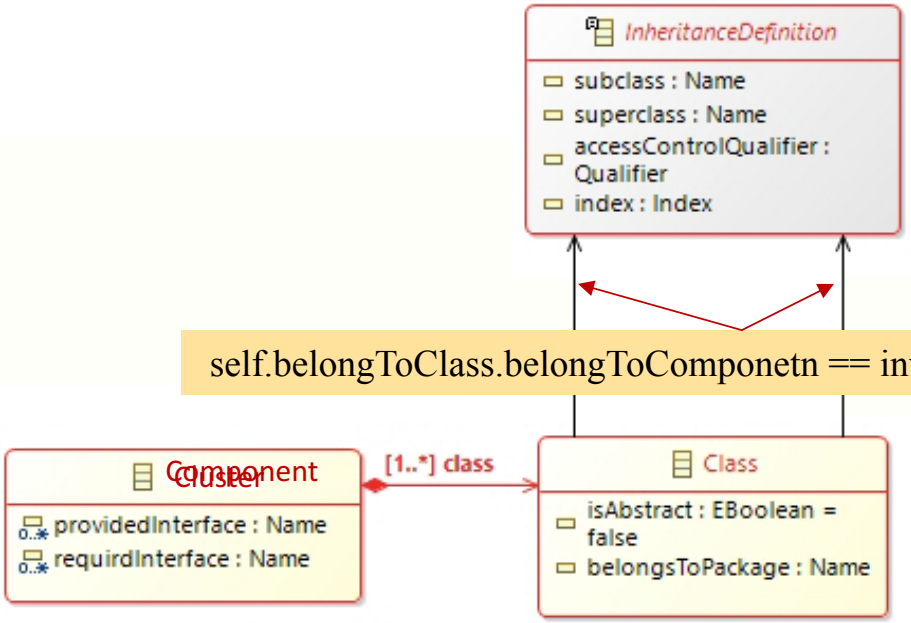


If(!accessedTo.belongToClass.isAbstract){
self.belongToClass.belongToComponent ==
accessedTo. belongToClass.belongToComponent}

If(!reference.belongToClass.isAbstract){
self.belongToClass.belongToComponent == reference.
belongToClass.belongToComponent}

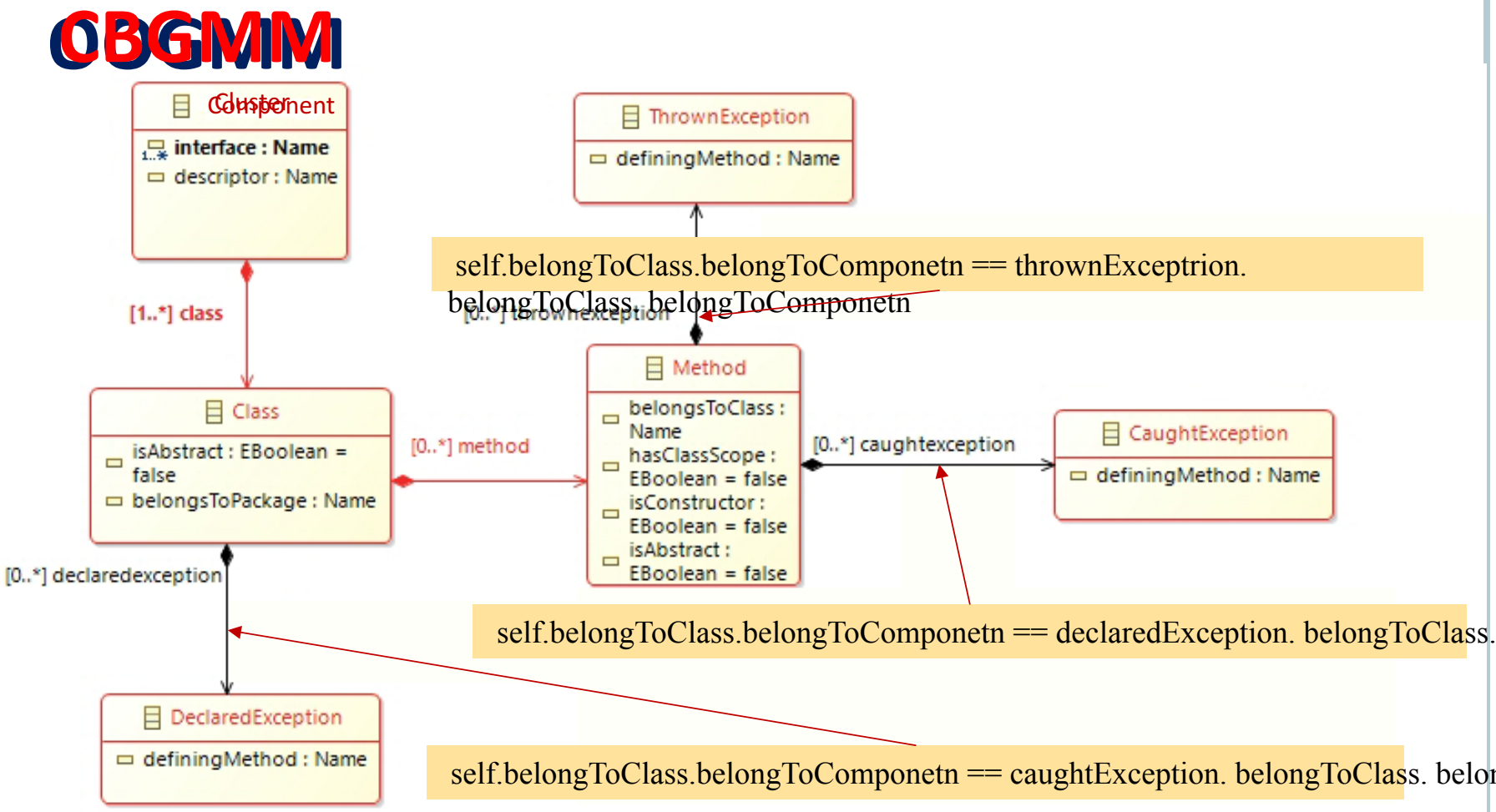
CBGMM : Inheritance Relationship

CBGMM



self.belongToClass.belongToComponetn == invokedBy. belongToClass. belongToComponetn

CBGMM: Exception Handling



Example: Transformation Rules

Listing 5.1: Main QVT definitions and mapping functions in our migration

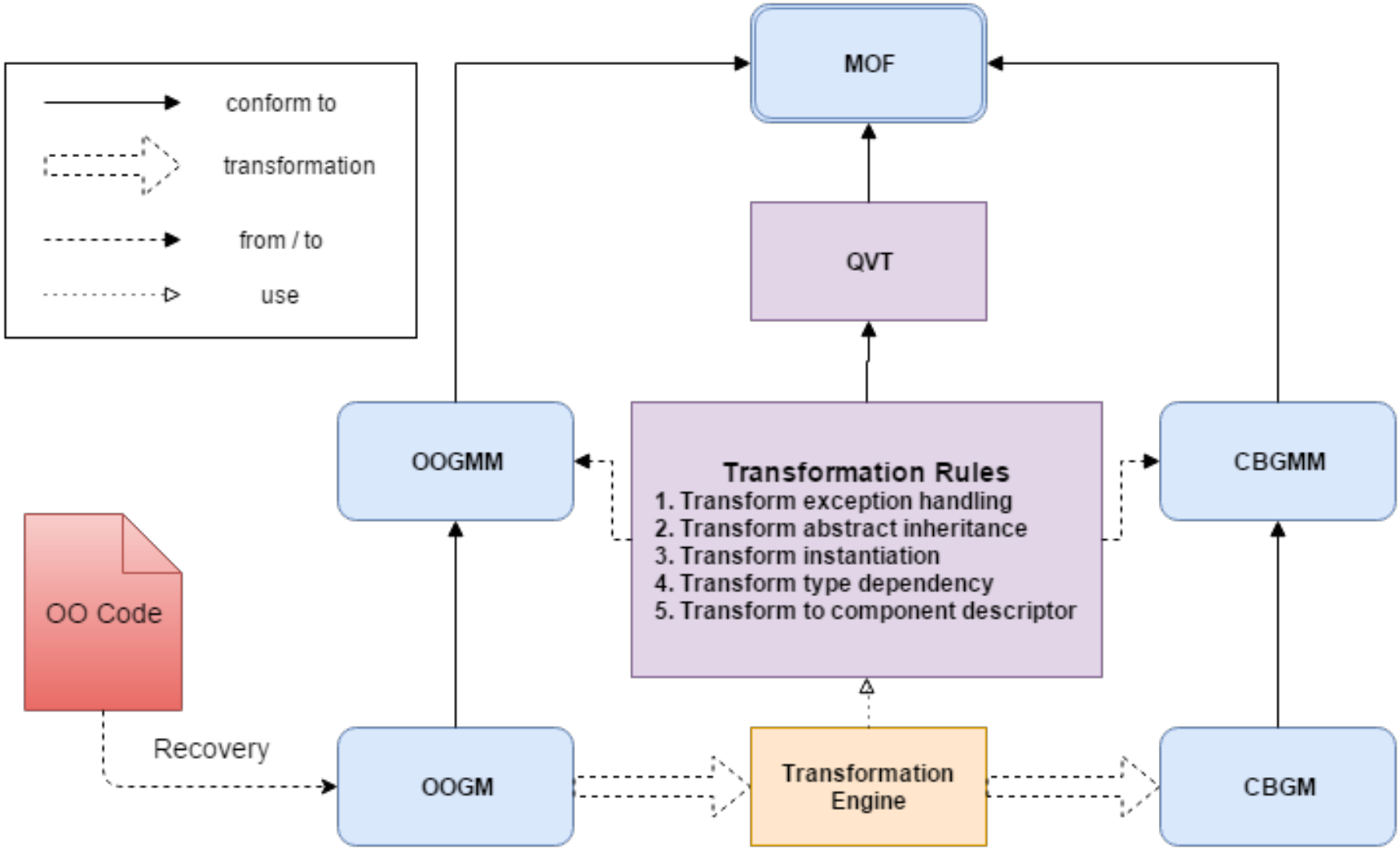
```
/*
```

Algorithm 3 Transforming inheritance relationship

1: **procedure** INHERITANCE–TRANSFORMATION
2: **Pre-Conditions:**
3: $subClass \in Cluster1$ & $superClass \in Cluster2$
4: $subClass$.isInherit($superClass$) = true
5: $superClass$.isAbstract() = false
6: **Rules:**
7: $subClass$.removeInherit($superClass$)
8: $interfaceSub \leftarrow ExtractInterface(subClass)$
9: $interfaceSuper \leftarrow ExtractInterface(superClass)$
10: $interfaceSub$.inherit($interfaceSuper$)
11: applyDelegationPattern($subClass$, $superClass$)
12: $subClass \leftarrow addAttribute(interfaceSuper, \_super)$
13: $superClass \leftarrow addAttribute(interfaceSuper, \_this)$
14: **end procedure**

```
-

mapping OOAR::Invocation::instantiation2factory() : GCM::Class {
```

Transforming OOGM to CBGM Rules



Transforming CBGM into CBSMs

1. Defining CBSMMs
   - Studying the common component-based metamodels
2. Identifying the variability of transformation rules
   - Identifying the variability between component-based metamodels
3. Model-driven transformation feature model
   - Modeling the common and variability transformations for component-based metamodels

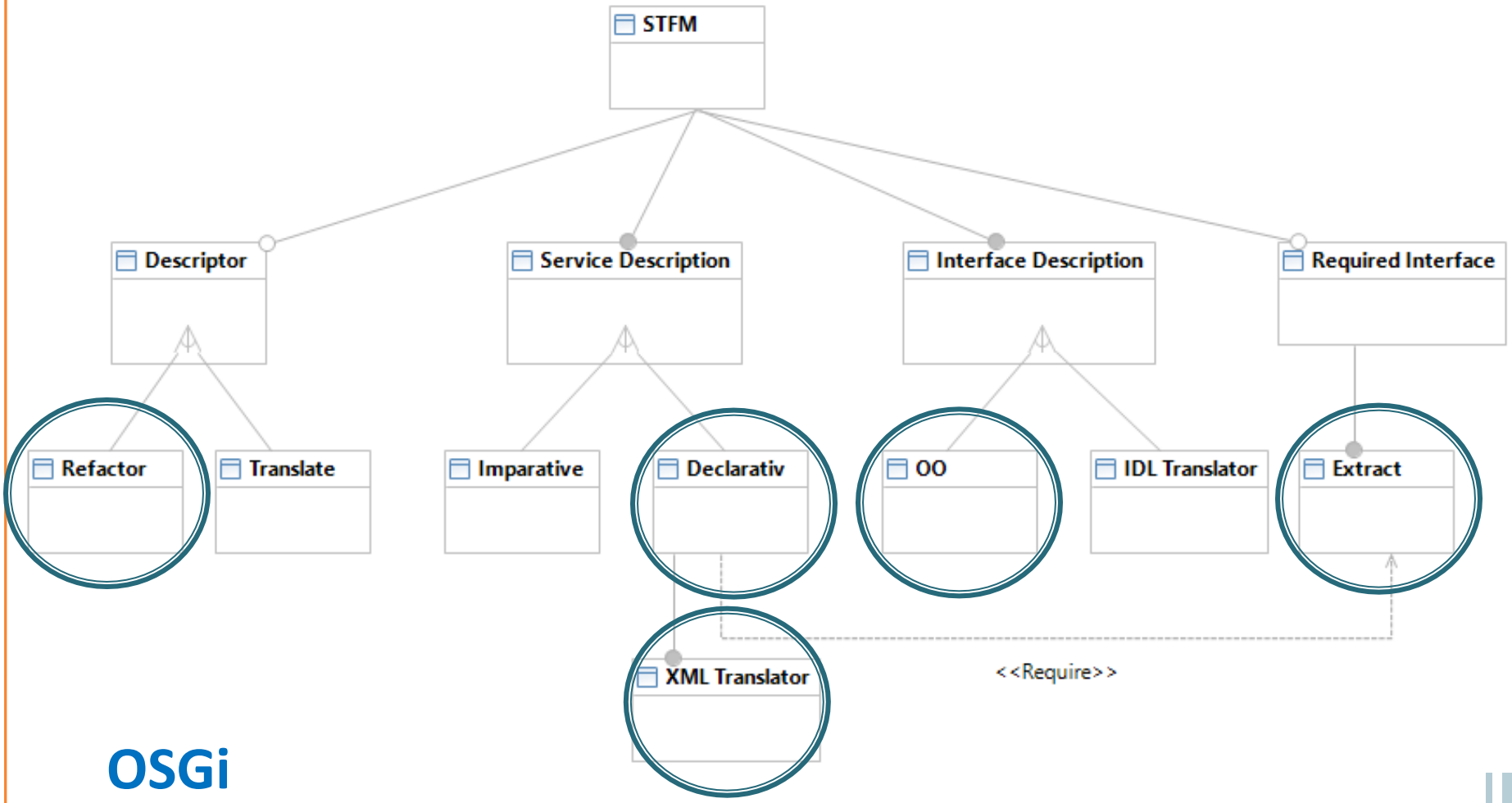
The Variability of Transformation Rules

- Component descriptor
  - Implicit: Do not have component descriptor
  - Explicit: OO, CDL, ADL
- Service description
  - Declarative: Provided and required references of services are described in XML-like files
  - Imperative: Using the standard method call in object oriented
- Interface description
  - Independent: Using independent languages to describe component interfaces IDL
  - Dependent: using a standard object oriented interface
- Required interface
  - Explicit: Explicitly declare the required services
  - Implicit: Embedded in the component source code

The Variability of Transformation Rules

Variability		OSGi	SOFA	CCM	Fractal	COM	OpenCOM	JB	EJB
Component Descriptor	Implicit			✔		✔			✔
	Explicit	✔	✔		✔		✔	✔	
Service Description	Declarative	✔	✔						
	Imperative	✔		✔	✔	✔	✔	✔	✔
Interface Description	Independent			✔					
	Dependent	✔	✔		✔	✔	✔	✔	✔
Required interface.	Implicit	✔	✔	✔	✔		✔		
	Explicit	✔				✔		✔	✔

Transformation Feature Model (TFM)



OSGi

Outline

- Contributions
  1. Healing Component Encapsulation
  2. Reveal Component Instance
  3. Model-Driven Transformation: OO Models to CB Models
- **Experimental Evaluation**
- Conclusion and Future Work

Experimental Evaluation

- Research Questions
  - **RQ1**: Does the transformation result avoid **component encapsulation** violation?
  - **RQ2**: To which extent does the automatic transformation reduce the developer's **effort**?
- Evaluation Methods
  - **Answer to RQ1**: The **Abstractness** metric proposed by Martin [Martin 2011]
    - Evaluate how much the OO dependencies are transformed to interface-based ones
  - **Answer to RQ2**: Compared the estimated efforts expressed by time spent by developers through **manual** transformation **automatic** transformation

Experimental Evaluation: Data Collection

- Conducted our transformation approach on 9 Java projects [Qualitas Corpus]
- Selection criteria
  - Different project size
  - Different domains
  - Different Development team

Application	Version	Domain	# of classes	Code size (KLOC)
Tomcat	7.0.71	middleware	1359	196
Ant	1.9.4	parsers/generators/make	1233	135
Checkstyle	6.5.0	IDE	897	63
Freecol	0.11.3	games	669	113
JFreeChart	1.0.19	tool	629	98
HyperSQL	2.3.2	database	539	168
Colt	1.2.0	SDK	288	35
Log4j	1.2.17	testing	220	21
Galleon	0.0.0-b7	3D/graphics/media	137	26

Experimental Evaluation: Protocol

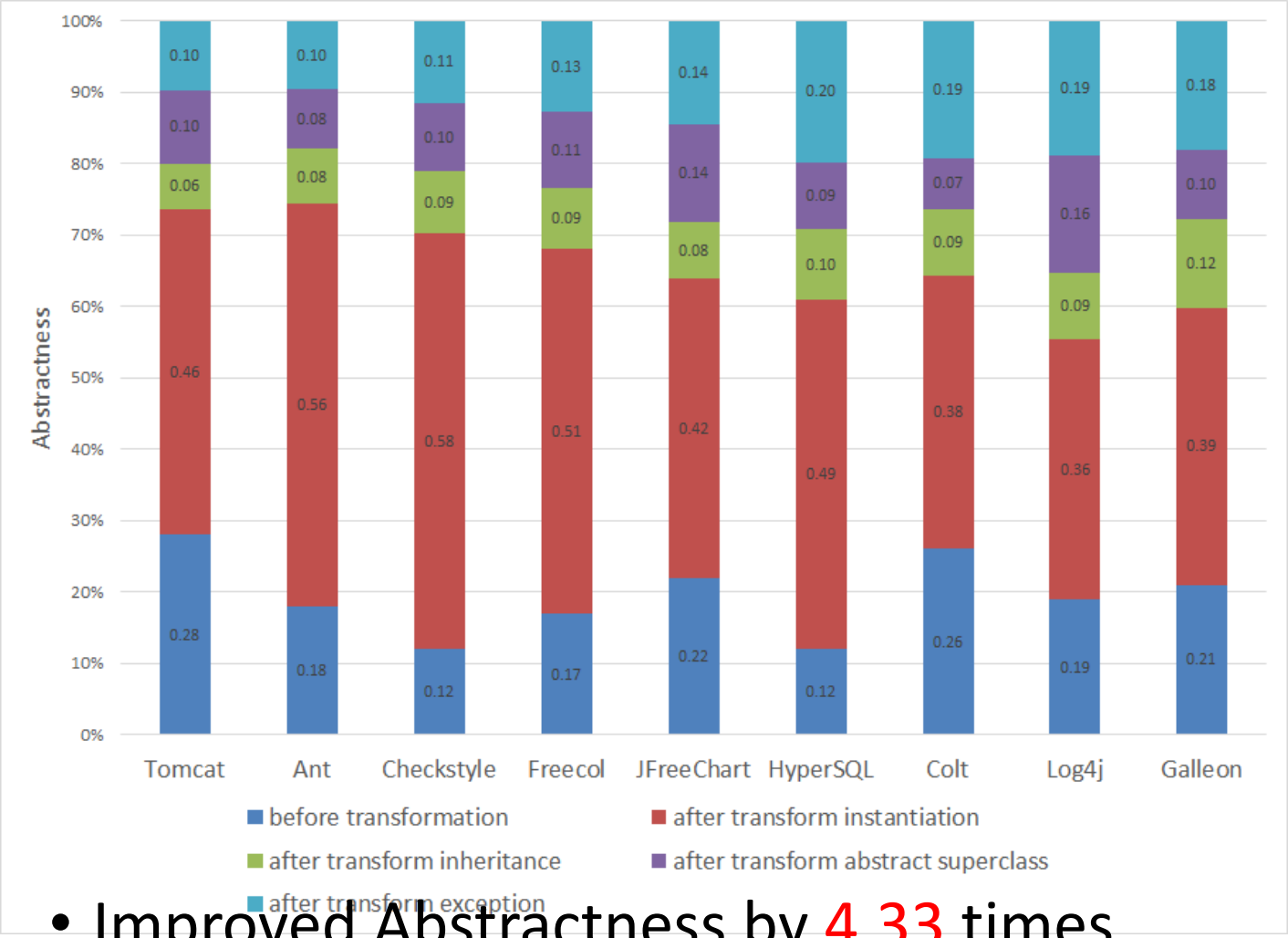
Abstractness(*C*) = *Na* / *Np*

Na: # of interface and abstract $\in$ provided types by *C*

Np: # provided types by *C*

Persons	# persons	Group	Experience in Java
Ph.D Students	5	1	3-6 years
Developers	5	2	4-6 years
M.S. Students	5	3	2-4 years

RESULTS: ABSTRACTNESS



- Improved Abstractness by 4.33 times

Results: Manual vs. Automatic Transformation

Application	Transformation type	# of needed trans.	# of manual trans.	# of different manual trans.	# of wrong trans.	AVG. time (s)	Min/Max time (s)	STD time (s)	AVG. estimated time (h)
Tomcat	Instansiation	350	35	20	2	367	230/1008	126	35.68
	Inheritance	49	3	3	6	1106	928/1380	241	15.05
	Abstract superclass	79	16	9	2	1310	1019/1803	195	28.75
	Exception	74	13	8	2	1255	989/1747	173	25.80
JFreeChart	Instansiation	116	37	16	0	395	192/901	169	12.73
	Inheritance	22	16	15	2	1053	862/1301	148	6.44
	Abstract superclass	38	11	3	2	1198	1012/1405	135	12.65
	Exception	40	7	5	3	1077	1002/1359	104	12.00
Log4j	Instansiation	62	34	17	0	377	248/869	158	6.49
	Inheritance	16	9	6	11	1054	892/1401	188	4.68
	Abstract superclass	28	6	6	5	989	982/1106	64	7.69
	Exception	32	7	4	1	1033	932/1203	120	9.18

Example:

- Automatically transforms **Tomcat** in a **few minutes** (about 6 minutes) without any **wrong** transformation.
- The ratio between the manually and the automatically transformation times for Tomcat is **795**

Conclusion

- Transforming object-oriented code into component-based code
  I. Transforming object-oriented dependencies into interface-based ones using design patterns
  II. Materialize Component instance
  III. A model-driven approach to automatically transform object-oriented code to component-based code

- Threats to validity
  I. Evaluate other software quality attributes (e.g. performance)
  II. The coverage of test cases