Master II Informatique
Parcours : Génie logiciel

SILOC : Spécification et Implémentation des Langages à Objets et à Composants - Spécification et transformation de contraintes sur les architectures à base de composants -

Chouki.TIBERMACINE@lirmm.fr
Maître de conférences en informatique

http://www.lirmm.fr/~tibermacin/ens/siloc/

Plan du cours

- **Partie 1 :** Généralités sur les composants et architectures logicielles à base de composants
- **Partie 2 :** Spécification des contraintes architecturales
- **Partie 3 :** Un modèle de composants pour les contraintes architecturales
- **Partie 4 :** Transformation des contraintes architecturales
- **Partie 5 :** Génération de code à objets à partir de contraintes architecturales
- **Partie 6 :** Génération de composants à partir des contraintes

Plan du cours

- **Partie 1 :** Généralités sur les composants et architectures logicielles à base de composants
- **Partie 2 :** Spécification des contraintes architecturales
- **Partie 3 :** Un modèle de composants pour les contraintes architecturales
- **Partie 4 :** Transformation des contraintes architecturales
- **Partie 5 :** Génération de code à objets à partir de contraintes architecturales
- **Partie 6 :** Génération de composants à partir des contraintes

Problématique

- Contraintes d'architecture écrites dans la phase de conception (sur les modèles) doivent être ré-écrites **manuellement dans la phase d'implémentation (sur le code)**

- Sémantique identique dans les deux phases. Pourquoi ne pas dériver les unes à partir des autres ?

Objectif

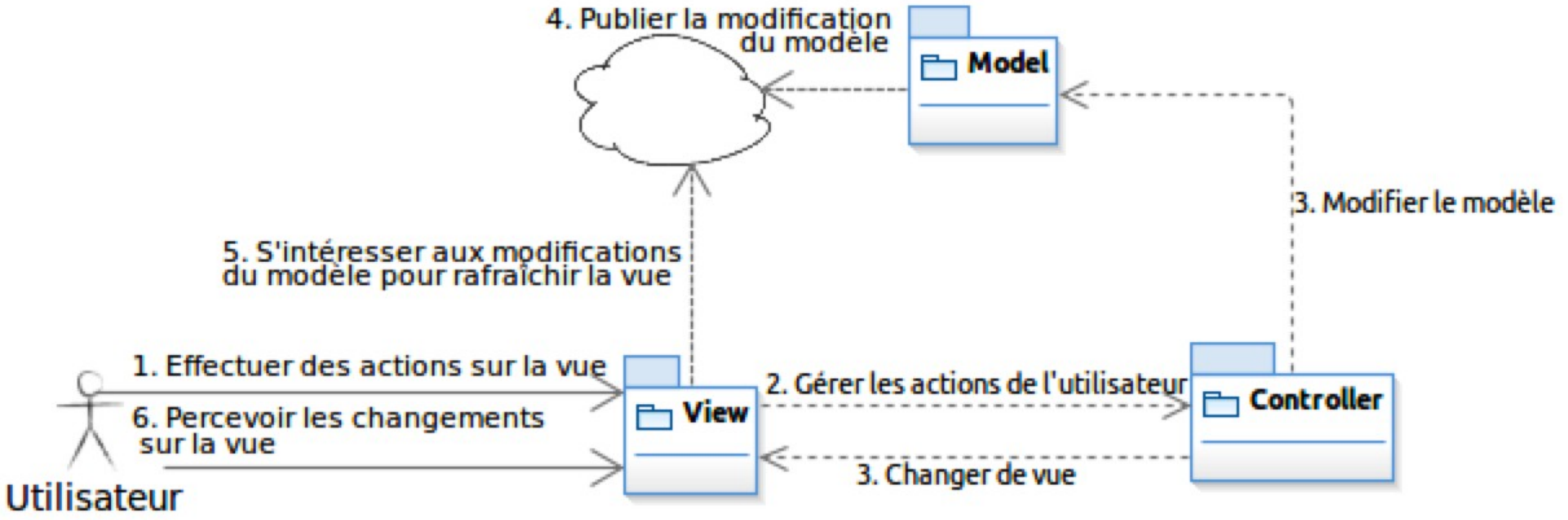
- Génération automatique de contraintes exprimées sur du code à partir de contraintes exprimées sur des modèles

- Intérêt :
  - Compléter le forward-engineering des artefacts métiers (génération de code à partir des modèles métier)
  - Offrir la possibilité donc aux développeurs :
    - de se concentrer sur la logique métier (compléter le code métier de leur application)
    - de vérifier les contraintes architecturales sur le code

- Application à UML (conception) et Java (implémentation)

Exemple de contrainte architecturale

- Patron MVC :



Exemple de contrainte architecturale

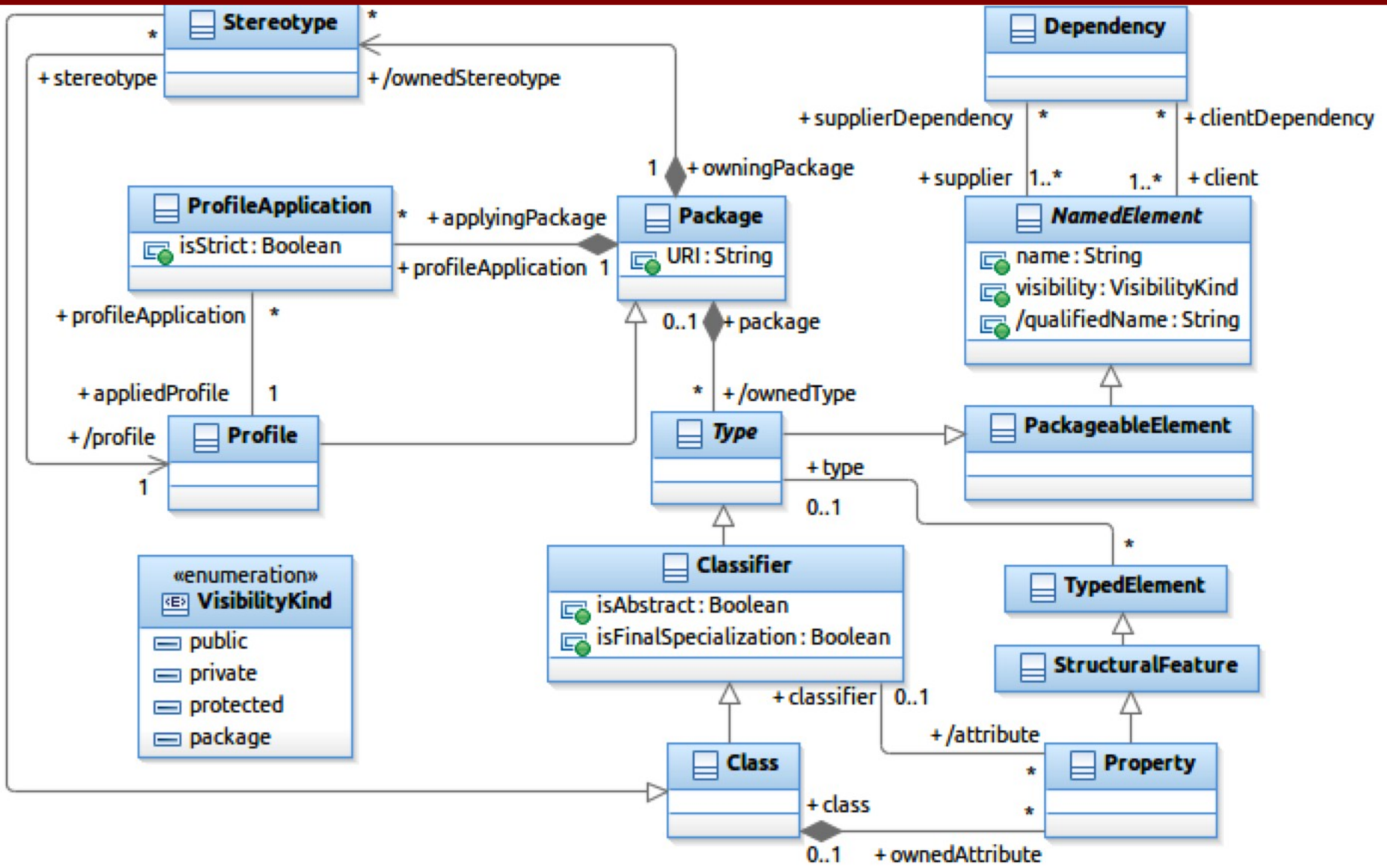
- Patron MVC : Pas de dépendances :
  - des classes stéréotypées Model vers les classes stéréotypées View
    - possibilité d'avoir plusieurs vues pour un même modèle
    - la vue peut jouer le rôle d'écouteur des modifications qui ont lieu sur le modèle, afin qu'elle se mette à jour
  - des classes stéréotypées Model vers les classes stéréotypées Controller (possibilité d'avoir +ieurs contrôleurs pour le modèle)
  - des classes stéréotypées View vers les classes stéréotypées Model (les contrôleurs doivent faire le relai entre la vue et le modèle

Exemple de contrainte architecturale : applicable sur un modèle UML

- Patron MVC : contrainte partielle

```
context MonApplication:Package inv:
let model:Set(Type) = self.ownedType->select(t:Type |
  t.getAppliedStereotypes()->exists(name='Model'))
in let view:Set(Type) = self.ownedType->select(t:Type |
  t.getAppliedStereotypes()->exists(name='View'))
in let controller:Set(Type) = self.ownedType->select(t:Type |
  t.getAppliedStereotypes()->exists(name='Controller'))
in
self.ownedType->forAll(t : Type | (model->includes(t)
  implies t.clientDependency.supplier->forAll(tt |
    view->excludes(tt) and controller->excludes(tt)))
and (view->includes(t) implies t.clientDependency.supplier->forAll(tt |
    model->excludes(tt))))
```

Extrait du méta-modèle UML utilisé



Contrainte du patron MVC : les types

- Pas de dépendance = pas d'attribut d'un certain type

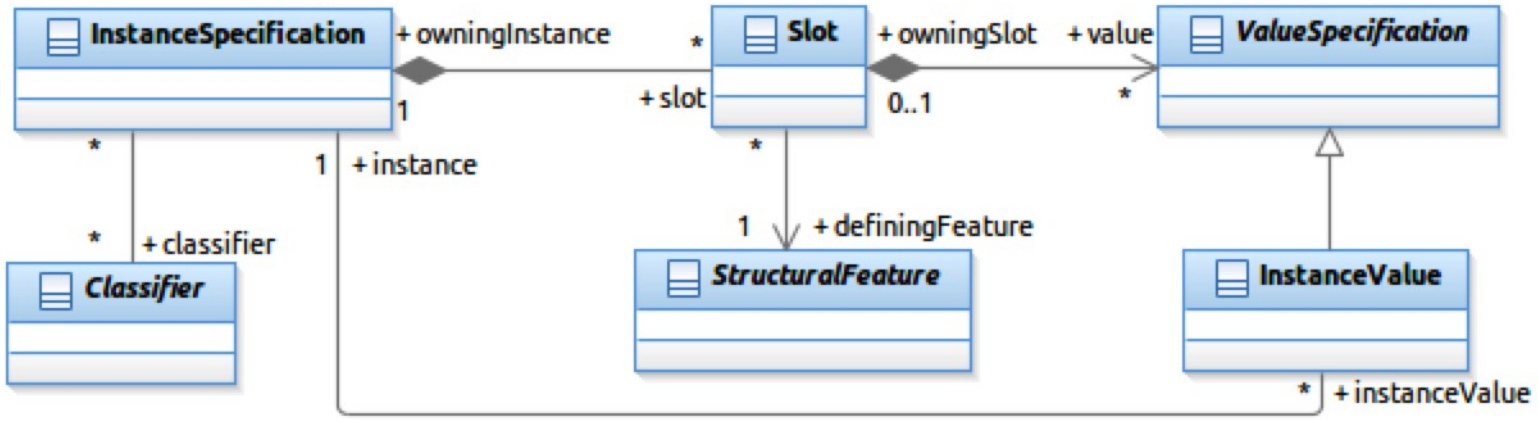
```
self.ownedType->forAll(t : Type | (model->includes(t)
  implies if t.oclIsKindOf(Classifier)
  then
    t.oclAsType(Classifier).attribute->forAll(a |
      view->excludes(a.type) and controller->excludes(a.type))
  else true endif )
  and
  (view->includes(t) implies
  if t.oclIsKindOf(Classifier)
  then t.oclAsType(Classifier).attribute->forAll(a |
    model->excludes(a.type))
  else true
  endif ))
```

Contrainte du patron MVC : les instances

- Pas de dépendance = Pas de valeurs d'attributs d'un certain type

```
context InstanceSpecification inv:
if model->includes(self.classifier)
then
  self.slot->forAll(s : Slot |
    -- Mêmes vérifications que précédemment en utilisant s.definingFeature
    -- pour accéder à l'attribut (StructuralFeature) qui définit le slot
    if s.value.isOclKindOf(InstanceValue)
    then
      controller->excludes(s.value.oclAsType(InstanceValue)
        .instance.classifier)
    else true
    endif
  )
else true
endif
```

Extrait du méta-modèle UML utilisé



Exemple de contrainte architecturale : applicable sur du code Java

- Patron MVC : contrainte partielle

```
public boolean invariant(Class<?>[] classesMonApplication) {
  for(Class<?> uneClasse : classesMonApplication) {
    if(uneClasse.isAnnotationPresent(Model.class)) {
      Field[] attributs = uneClasse.getDeclaredFields();
      for(Field unAttribut : attributs) {
        if(unAttribut.getType().isAnnotationPresent(View.class) ||
        unAttribut.getType().isAnnotationPresent(Controller.class))
          return false;
      }
    } // ...
  }
  return true;
}
```

Checker unique des contraintes architecturales sur le code

- Checker de l'invariant :

```
public boolean check(String classInv, Object objToCheck) {
  try {
    Class<?> c = Class.forName(classInv);
    if(! c.isAnnotationPresent(Constraint.class))
      throw new RuntimeException("Not a valid constraint class");
    Class[] params = {objToCheck.getClass()};
    return (Boolean)(c.getMethod("invariant",Object.class)
    .invoke(c.newInstance(),params));
  }
  catch(ClassNotFoundException e) {
    System.err.println("La classe "+classInv+" n'existe pas.");
  }
  catch(NoSuchMethodException e) {
    System.err.println("Pas de méthode (invariant) pour tester
    les contraintes architecturales dans la classe "+classInv);
  }
```

Checker unique des contraintes architecturales sur le code

- Checker de l'invariant : suite

```
...
catch(InvocationTargetException e) {
    System.err.println("Impossible d'invoquer la méthode (invariant)
    pour tester les contraintes architecturales dans la classe "+classInv);
  }
  catch(InstantiationException e) {
    System.err.println("Impossible de créer une instance de la classe : "
    +classInv);
  }
  catch(IllegalAccessException e) {
    System.err.println("Impossible de créer une instance de la classe : "
    +classInv);
  }
  return false;
}
```

Où et comment utiliser le checker ?

- Gérer cette vérification dans un IDE comme Eclipse, après modification du code : identifier les parties du code qui, lorsqu'elles sont modifiées, la vérification des contraintes est déclenchée ? -Solution statique-

- Injecter à la fin d'un constructeur ? -Solution dynamique-

```
if(! (new Checker()).check("PatronMVC",this))
   throw new ViolatedConstraintException("Patron MVC non respecté");
```

Méthode de génération

- Processus multi-étapes :
  1. réifier les contraintes architecturales

  2. appliquer des règles de transformation pour générer des contraintes équivalentes sur le méta-modèle Java

  3. générer du code Java

  4. vérifier la cohérence du code généré

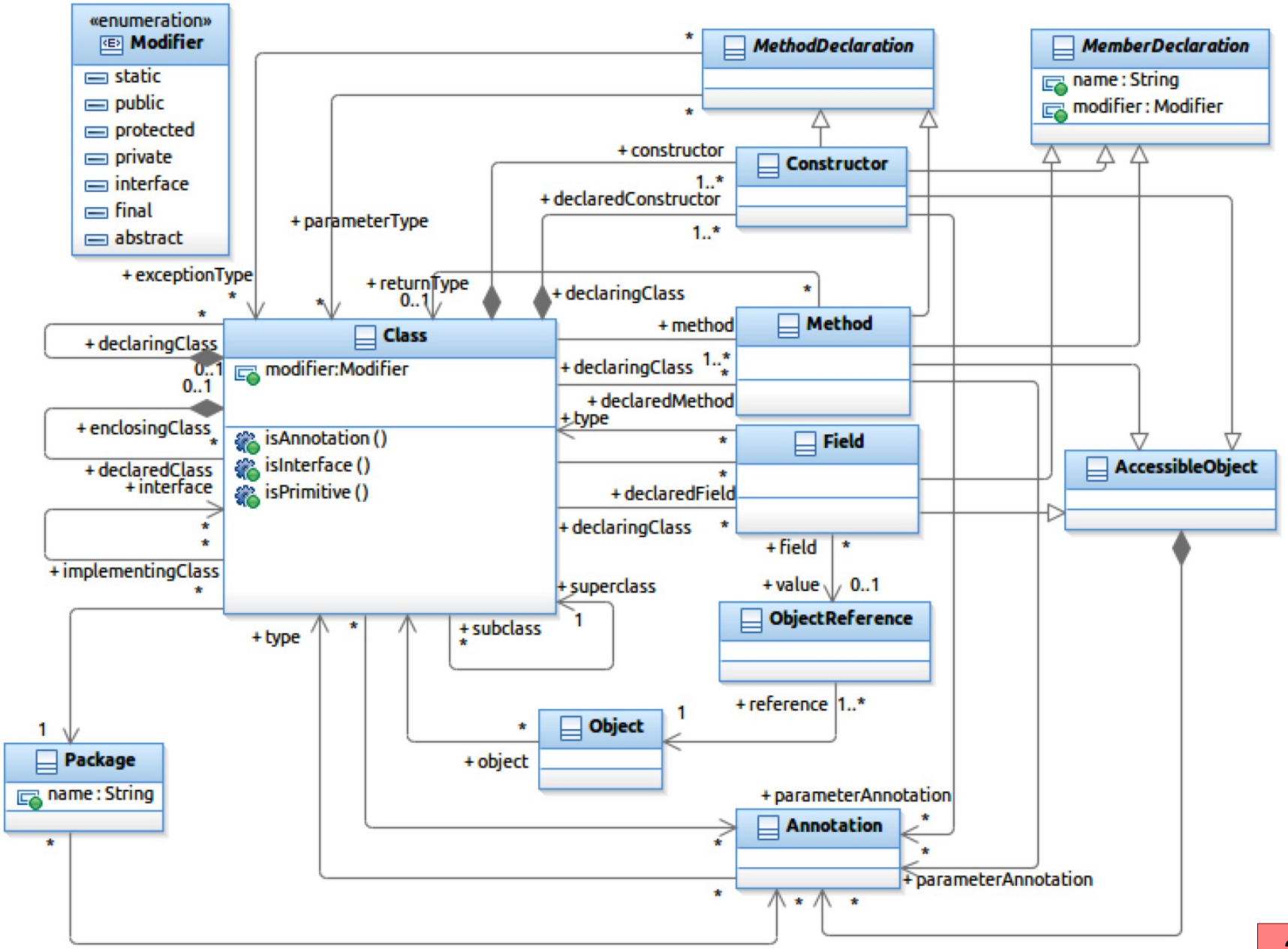
Réification des contraintes architecturales

- Il s'agit simplement de passer les contraintes à un compilateur qui va générer l'arbre syntaxique abstrait
- Implémentation actuelle : DresdenOCL
  http://www.dresden-ocl.org/index.php/DresdenOCL

Transformation des contraintes vers le méta-modèle Java

- Objectif : projeter les concepts avant de changer de syntaxe
- Raison : réduire la complexité de la génération de code (la faire en plusieurs étapes simples)
- Besoin qui en découle : un méta-modèle de Java
  - Il en existe quelques uns dans la littérature
  - Aucun ne répond à notre besoin : méta-modèle « reflétant » Java Reflect (puisque le code qui est généré repose sur cette librairie)

Méta-modèle de Java



Contrainte du patron MVC sur le méta-modèle Java

- Patron MVC :

```
context Class inv:
self.annotation->exists(a:Annotation | a.type.name='Model')
implies
self.field->forAll(f : Field |
  not (f.type.annotation->exists(type.name='View')
  or f.type.annotation->exists(type.name='Controller'))
)
and
self.annotation->exists(type.name='View')
implies
self.field->forAll(not (type.annotation->exists(type.name='Model')))
```

Contrainte du patron MVC sur le MM Java : les instances

- Patron MVC :

```
context Object inv:
if self.class.annotation->exists(name='Model')
then
  self.class.field->forAll(f:Field |
    -- Mêmes vérifications que précédemment en utilisant f.type
    -- pour accéder au type de l'attribut
    if f.value <> null
    then f.value.object.class.annotation = 'Controller'
    else true
    endif
  )
else true
endif
```

Utilité des langages de transformation de modèles ici ?

- Des langages existent pour la transformation de modèles :
  - Modèles vers modèles
  - Modèles vers texte
- Exemples : Kermeta, ATL, QVT, ... (cours Ingénierie des modèles)
- Dans notre cas :
  - Transformer les contraintes textuelles en modèles conformes au méta-modèle OCL
  - Transformer le modèle de la contrainte en un autre modèle (transformation endogène : même méta-modèle OCL)
  - Reproduire le texte de la contrainte cible à partir de son modèle
- Transformation trop lourde

Réalisation de la transformation

- Méthode ad-hoc (pas de langages de transformation particulier)
- Transformation basée sur des mappings :
  - Mappings de concepts (méta-classes)
  - Mappings d'accès aux propriétés des concepts (méta-attributs et méta-rôles)
  - Mappings de navigations

Réalisation de la transformation

- Extrait des mappings :

Méta-classe en UML	Équivalent en Java
Class	Class
Package	Package
Property	Field
Operation	Method
Stereotype	Annotation

Rôle en UML	Équivalent en Java
ownedAttribute	field
ownedOperation	method
nestedType	declaredClass
superClass	superClass
interfaceRealization	interface
package	package
class	declaringClass
type	type
raisedException	exceptionType
type	returnType

Navigation en UML	Équivalent en Java
package.profileApplication.appliedProfile.ownedStereotype	annotation
profileApplication.appliedProfile.ownedStereotype	annotation

Génération de code

- Génération de code de façon ad-hoc à partir d'exemples
- Prise en compte d'une partie d'ACL
- Difficultés multiples (opérations ensemblistes imbriquées, combinées avec des forAll et exists, par exemple)

Travaux existants : Génération de code à partir de contraintes

- Aucun travail ne traite la génération de code à partir de contraintes d'architecture
- Il existe des générateurs de code à partir de contraintes OCL, mais pour des contraintes fonctionnelles :
  - Outil : OCL Dresden
  - Le code généré ne fait pas de l'introspection
- Il existe des langages de contraintes d'architecture au niveau code, mais sans lien avec des modèles (conception) de ce code

Travaux existants : Langages de contraintes d'architecture au niveau code

- CoffeeStrainer : du code Java écrit sous la forme de commentaires particuliers qui utilisent une librairie d'analyse statique de code
- CCEL (C++ Constraint Expression Language) : contraintes sur les structures de programmes C++
- CDL (Category Description Language) et SCL (Structural Constraint Language) : prédicats du 1er ordre définis sur des programmes à objets (API pour l'analyse statique de code)
- DCL (Dependency Constraint Language) : naviguer dans du code à objets et écrire des règles du type : only A can-access B
- Langages issus de dialectes à la Prolog : LogEn, Law Governed Architecture et Spine
- Outils industriels : Sonar, Checkstyle, Lattix, Macker, Classycle, Architexa et JArchitect

Travaux futurs

- Raffiner la génération de code
- Développer une librairie de contraintes d'architecture pour les patrons et styles les plus connus : application à UML objet
- Développer une librairie de programmes Java contenant des invariants représentant les contraintes
- Valider la génération de code en comparant les résultats avec la librairie d'invariants
- Mettre en œuvre l'évaluation des contraintes sur du code en utilisant des aspects

Questions