Génération de Code & Reverse Engineering

Reda Bendraou
reda.bendraou{{@}}Lip6.fr
http://pagesperso-systeme.lip6.fr/Reda.Bendraou/

Le contenu de ce support de cours a été influencé par les lectures citées à la fin de ce support.

Avant de commencer!

- Des techniques qui permettent d'augmenter la productivité et la qualité des applications
- Font partie de nombreux travaux de recherche en cours
- Techniques pas forcement très matures mais prometteuses
- Le résultat peut différer (souvent) d'un outilleur à un autre
- Dans ce cours, UML et Java seront utilisés comme source/cible

Petits problèmes!

- La différence entre les niveaux d'abstraction des deux langages (i.e., UML et Java)
  - Le tout graphique Vs. Le tout code!
  - Le corps des opérations sous forme de diagrammes?

- Des modèles 100% UML Vs des modèles pollués
  - Ex. JFrame et Compte bancaire dans le même modèle !!

- L'expressivité de chacun des langages
  - Des concepts UML n'ont pas d'équivalent en Java et vice-versa

Incompatibilité entre UML & Java: exemples

- Les Associations
  - Impossibilité d'exprimer en Java qu'une référence est l'opposé d'une autre
- L'héritage
  - Simple en Java contre multiple dans UML
- Les types de base
  - Beaucoup plus restreints dans UML qu'en Java
- Direction des paramètres des opérations
  - En Java seulement « in » et « return » contre « inout » et « out » en plus dans UML
- Import de Package
- Le corps des méthodes!!!!!
  - Diagramme d'activités avec Action Semantics? xUML?

Génération de Code

Génération de Code ou Forward Engineering

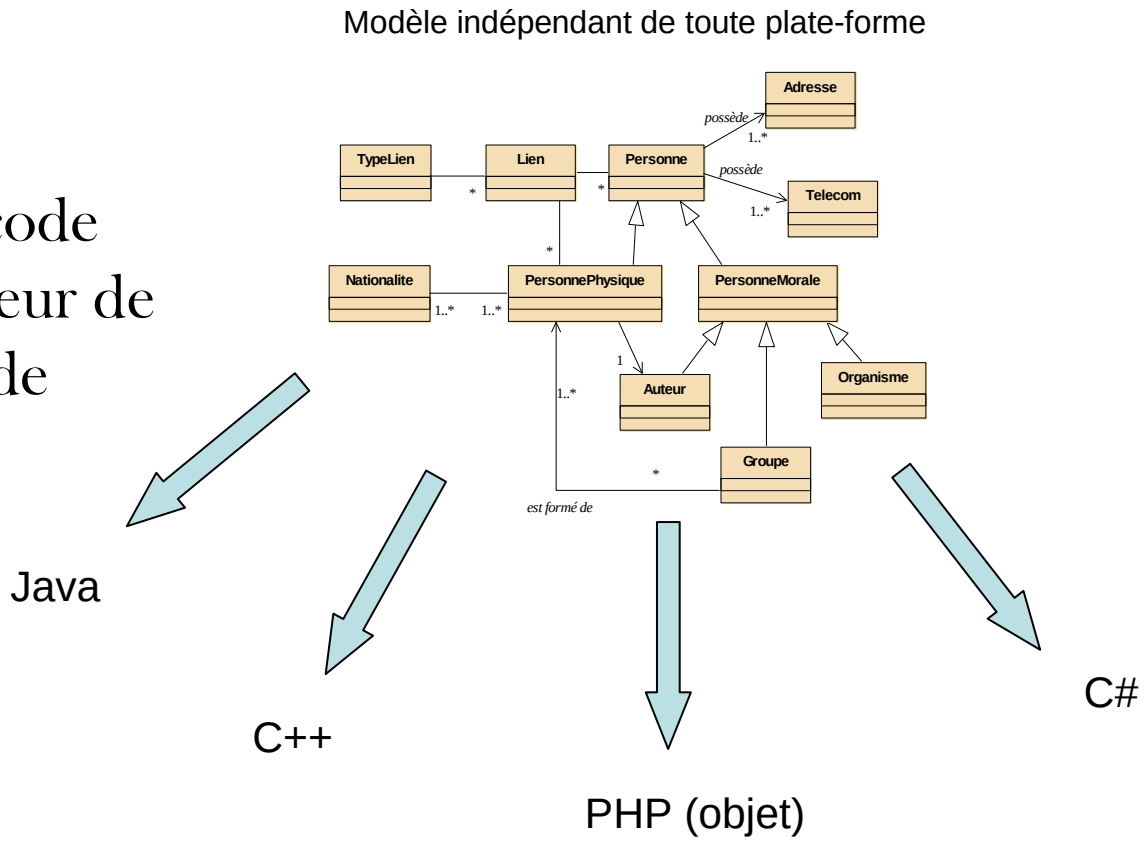
- Plus de productivité (ex. la structure d'une centaine de classes, d'opérations, etc. générée en un click)
  - Du 100% code généré dans le domaine du Web, BD, fichier de configuration, etc.

- Moins d'erreurs: un savoir-faire codé dans les générateurs de code
  - Je code un générateur une fois, je génère du code à volonté

- Vision MDE: Un modèle => plusieurs plates-formes d'exécution

Génération de Code

- Le principe du Pivot
- Le métier de développeur de code passe à développeur de générateur de code



Génération de Code: Principe

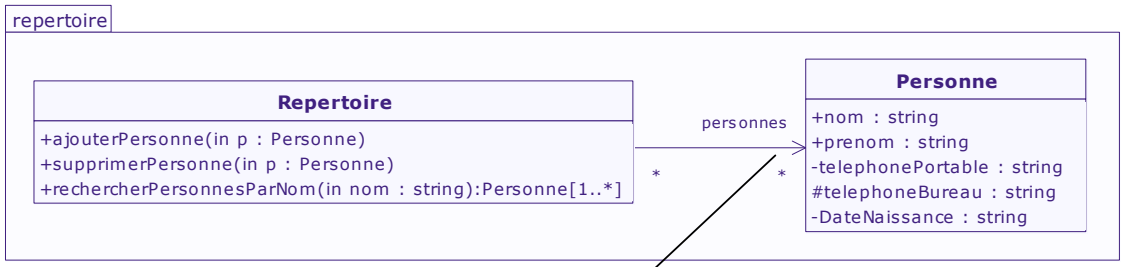
- Un ensemble de règles décrivant la correspondance entre concepts du modèle et concepts du langage (Java par ex.)
  - Éviter les incompatibilités décrites avant dans vos modèles UML (diapo 4)

- Dans certains cas/outils, des options sont à paramétrer
  - Choix des constructions dans le langage cible (ex. Vector ou ArrayList en Java)
  - Générer les accesseurs, constructeurs
  - Implanter les méthodes abstraites, etc.

- Un moteur de génération de code
  - Input: modèle, output: Code (Java par ex.)

Génération de Code: Exemples

- À noter: visibilités, navigabilités, multiplicités, opérations

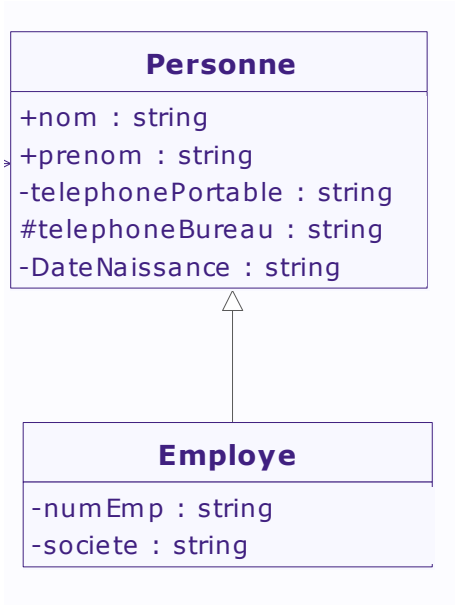



```
package repertoire;

import java.util.ArrayList;
import java.util.List;

public class Repertoire {

 public List<Personne> personnes = new ArrayList<Personne>();

 public void ajouterPersonne(Personne p){}

 public void supprimerPersonne(Personne p){}

 public List<Personne> rechercherPersonnesParNom(String nom){}
}
```

```
package repertoire;

public class Personne {

   public String nom;

   public String prenom;

   private String telephonePortable;

   protected String telephoneBureau;
}
```

Génération de Code: Exemples

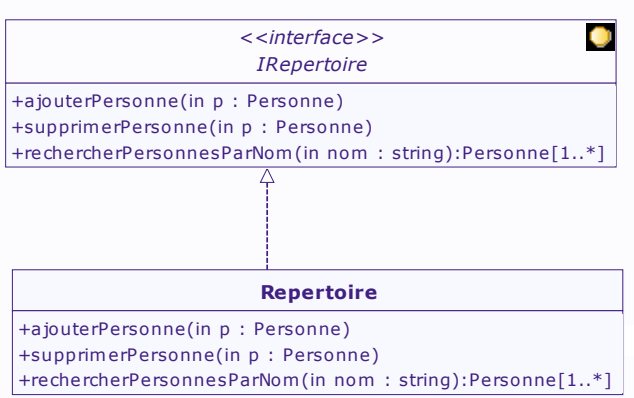
- L'héritage
  - En java, c'est le mot clé « extends »
  - Dans cet exemple: génération automatique des *getters*




```
public class Employe extends Personne {

    private String numEmp;

    private String societe;

    public String getNumEmp (){
        return this.numEmp;
    }


    public String getSociete (){
        return this.societe;
    }

}
```

Génération de Code: Exemples

- Réalisation d'interface
  - En java, c'est le mot clé « implements »




```
import java.util.List;

public interface IRepertoire{

 void ajouterPersonne(Personne p);
 void supprimerPersonne(Personne p);
 List<Personne> rechercherPersonnesParNom(String nom);
}
```

```
package repertoire;

import java.util.List;

public class Repertoire implements IRepertoire{

 public void ajouterPersonne(Personne p) {}
 public void supprimerPersonne(Personne p) {}
 public List<Personne> rechercherPersonnesParNom(String nom){}
}
```

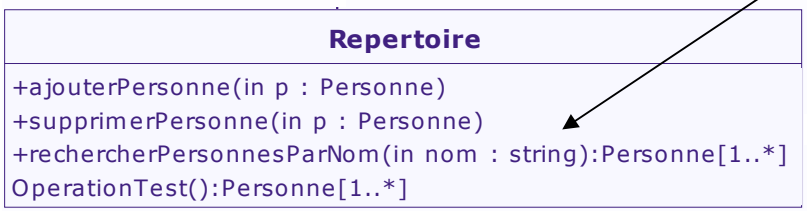
Génération de Code: Pour plus de productivité!

- Générer le corps des opérations directement des modèles
- Deux solutions:
  - Spécifier le corps des opérations avec les activités et actions UML2.0
    - Un diagramme d'activités par opération
    - Trop lourd, trop compliqué, moins lisible que du code
    - Aucune implantation de générateur de code pour D.A (seulement des prototypes)
    - Encore beaucoup d'ambiguïtés dans le standard
  - Associer des notes aux opérations contenant le code des opérations
    - En Java, C++ par ex. ou bien d'autres langages tels que xUML

Génération de Code: Pour plus de productivité!

- Exemple

```
ArrayList success = new ArrayList();
for (Iterator it = personnes.iterator() ; it.hasNext();) {
  Personne current = (Personne) it.next();
  if (current.getNom().compareTo(nom)==0) success.add(current);
 }
Personne[] res = new Personne[0];
return (Personne[]) success.toArray(res);
```



Règles de Génération de code UML 2 JAVA utilisées sans ce cours

- À toute classe UML doit correspondre une classe Java portant le même nom que la classe UML.

- À toute interface UML doit correspondre une interface Java portant le même nom que l'interface UML.

- Si une classe UML est associée à une autre classe UML et que l'association soit navigable, il doit se trouver un attribut dans la classe Java correspondant à la classe UML. Le nom de l'attribut doit correspondre au nom du rôle de l'association. Si l'association spécifie que le nombre maximal d'objets pouvant être reliés est supérieur à 1, le type de l'attribut Java est de type ArrayList. Sinon, le type de l'attribut doit être une correspondance Java de la classe UML associée. Si l'association n'est pas navigable, nous considérons qu'il n'est pas possible de générer du code Java.

Règles de Génération de code UML 2 JAVA utilisées sans ce cours

- À toute opération d'une classe UML doit correspondre une opération appartenant à la classe Java correspondant à la classe UML. Les noms des opérations doivent être les mêmes. Étant donné que Java ne supporte que les directions in et return, si l'opération contient des paramètres de direction out ou inout, nous considérons qu'il n'est pas possible de générer du code Java. Sinon, pour chaque paramètre de l'opération UML dont la direction est in doit correspondre un paramètre de l'opération Java. Les noms des paramètres doivent être les mêmes. Les types des paramètres doivent être une correspondance Java des types des paramètres UML. Si l'opération UML contient un paramètre de direction return, l'opération Java doit définir un retour qui lui correspond. Si l'opération UML ne contient pas de paramètre de direction return, l'opération Java retourne void.

Règles de Génération de code UML 2 JAVA utilisées sans ce cours

- Si une classe UML A est associée à une classe UML B et que l'association soit navigable, il doit correspondre un attribut dans la classe Java correspondant à la classe UML A. Le nom de l'attribut doit correspondre au nom du rôle de l'association. Le type de l'attribut doit être une correspondance Java de la classe UML B associée. Si l'association spécifie que le nombre maximal d'objets pouvant être reliés est supérieur à 1, l'attribut Java est un tableau. Si l'association n'est pas navigable, nous considérons qu'il n'est pas possible de générer du code Java.

- Si une classe UML hérite d'une autre classe UML, il doit correspondre une relation d'héritage (extends en Java) entre les classes Java correspondantes. Comme Java ne supporte pas l'héritage multiple, si une classe UML hérite de plusieurs autres classes UML, nous considérons qu'il n'est pas possible de générer du code Java.

Règles de Génération de code UML 2 JAVA utilisées sans ce cours

- Si une classe UML réalise une ou plusieurs interfaces UML, il doit correspondre une relation de réalisation entre la classe et les interfaces Java correspondantes.

- Si une classe UML est contenue dans un package, la classe Java correspondante doit déclarer qu'elle appartient à un package Java. Le nom du package Java doit être le même que le nom du package UML.

- Si un package UML importe un autre package UML, toutes les classes Java correspondant aux classes UML incluses dans le package UML doivent déclarer un import Java vers toutes les classes Java correspondant aux classes incluses dans le package UML importé.

- Si des notes de code Java sont associées aux opérations des classes UML, ce code est recopié dans les opérations Java correspondantes.

Reverse Engineering

Reverse Engineering

- Permet de générer des représentations d'un système dans un langage de plus haut niveau d'abstraction que le code source du système.

- Permet de réfléchir sur la structure du système ainsi que sur son comportement dynamique
  - Longtemps focalisé sur l'aspect structurel, certains outils proposent le reverse engineering du corps des opérations sous forme de diagramme de séquences (ex. NetBeans, IBM RSA)

- Le sujet de nombreux travaux de recherche

Reverse Engineering: Objectifs

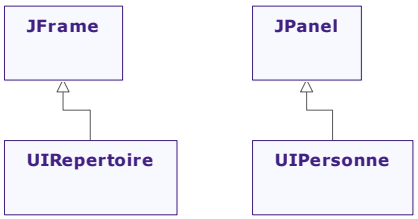
- Documenter, raisonner, communiquer autour de l'application dans un formalisme autre que du code
  - Ouverture de l'application à des non-développeurs
- Comprendre la conception ce qui permet de l'améliorer, la corriger, l'optimiser
  - Ex. Casser les dépendances cycliques entre packages (abordé dans ce cours)
  - Appliquer des design patterns (abordé dans ce cours)
- Identifier facilement des points d'extension de l'application
- Extraire des vues sur l'application
  - Vue structurelle
  - Vue dynamique (Opérations)
  - Vues spécifiques sur les généralisation entre classes, dépendances, associations, etc.

Reverse Engineering: Principe

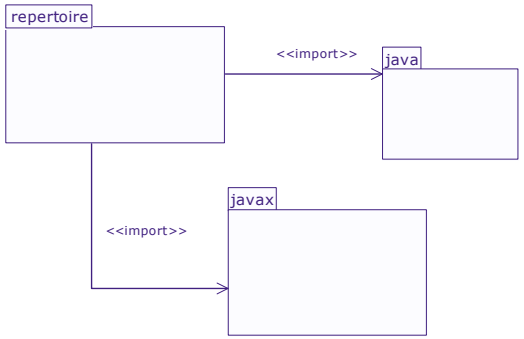
- Un ensemble de règles décrivant les correspondances entre le code source (Java dans notre cas) et le langage de plus haut niveau d'abstraction (UML dans notre cas).
- Règles souvent différentes d'un outilleur à un autre. Pas de norme standard
- Paramétrable dans certains outils
- Input: Code Output: Modèle UML (attention Modèle Vs. Diagrammes)

Reverse Engineering: Aspect structurel

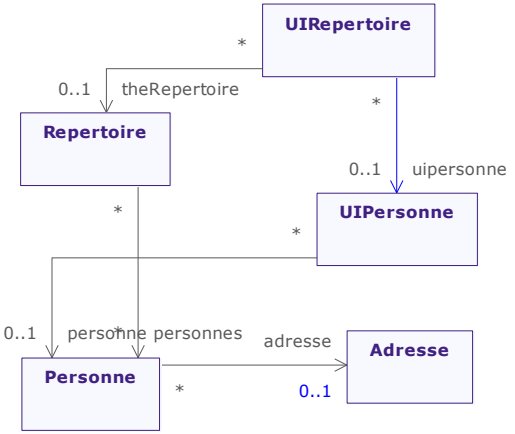
- L'opération de réverse génère un modèle UML à partir duquel il est possible d'extraire des vues (diagrammes)



Vue sur les généralisations

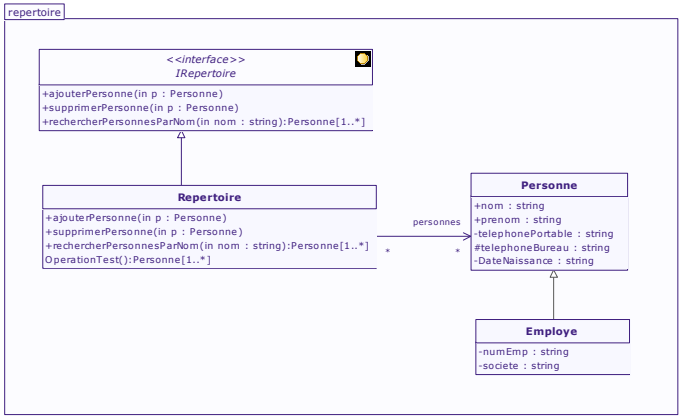


Vue sur les packages



Vue sur les associations

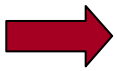
Zoom sur le contenu d'un package

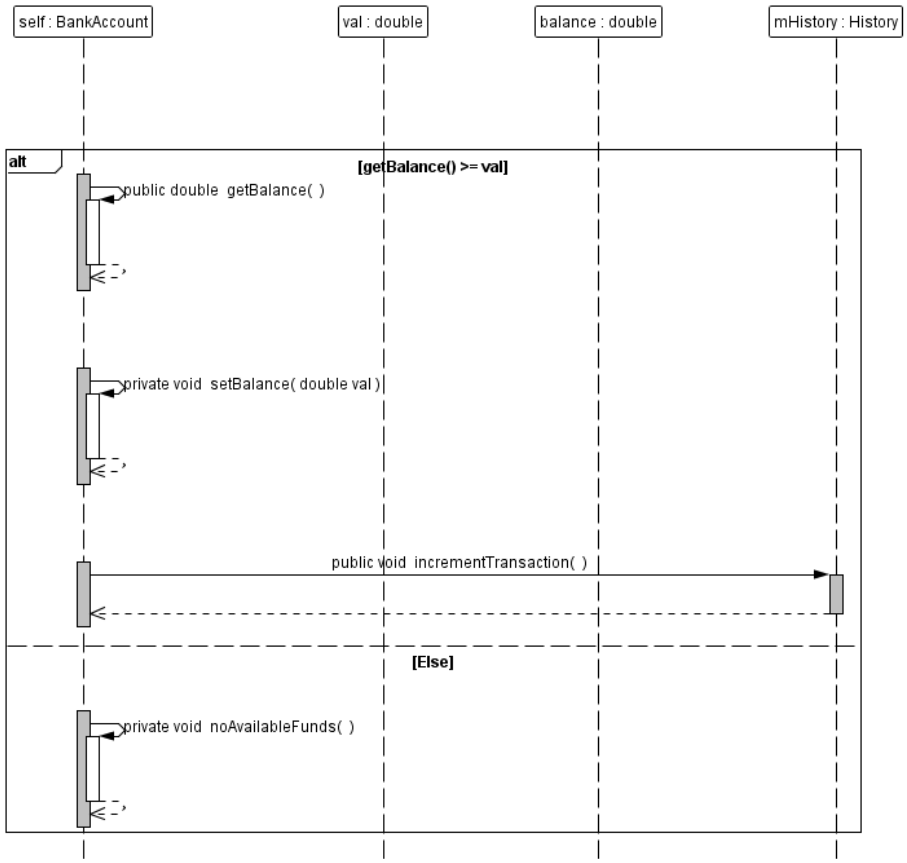


Reverse Engineering: Aspect comportemental

- Un diagramme de séquences à partir d'une opération en Java
- Ne capture que la séquence d'appels d'opérations
- Independent d'un scénario en particulier

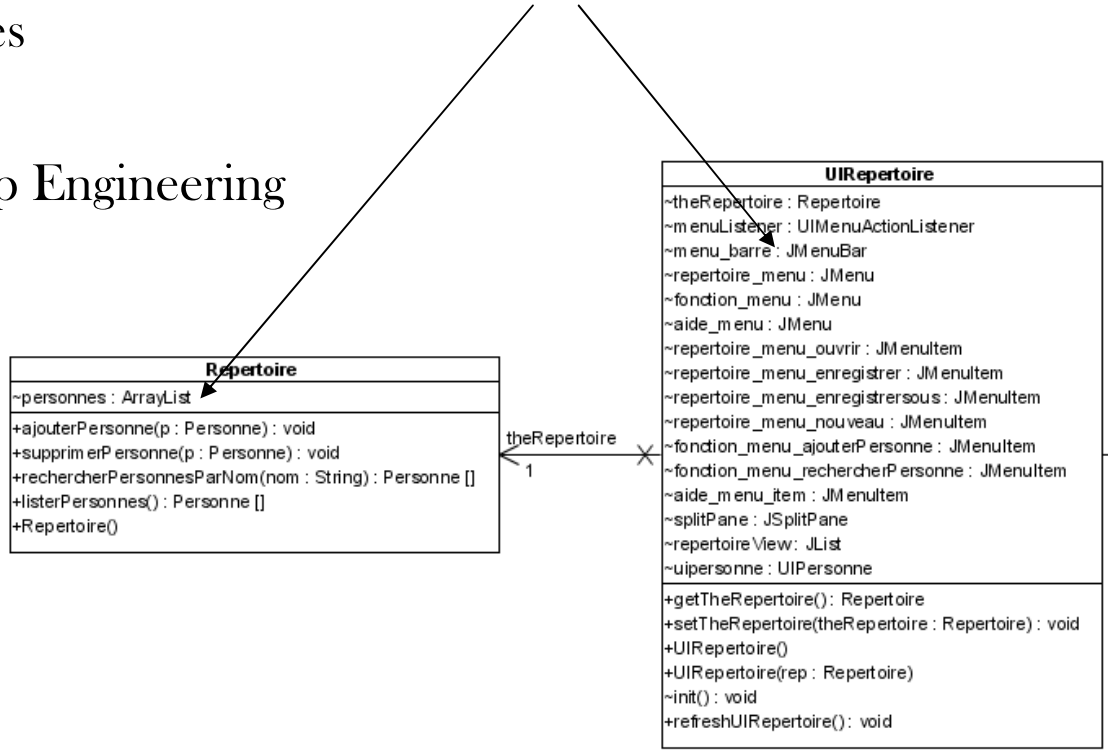
```
public void withdraw(double val) {

if (getBalance() >= val) {
    setBalance(balance - val);
    mHistory.incrementTransaction();
}
else
    noAvailableFunds();
}
```





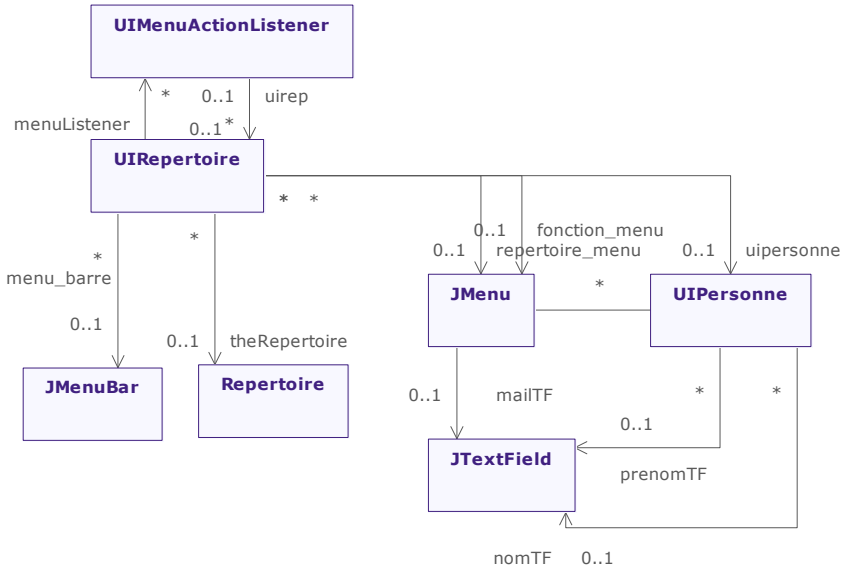
Reverse Engineering

- Certains outils arrivent à générer un diagramme de classes sans les classes propres aux APIs Java (Visual P., NetBeans par ex.)
  - Elles sont utilisées comme types pour les attributs. Ne polluent pas le diagramme de classes
  - Incorporées à l'outil
  - Facilite le Round trip Engineering



Reverse Engineering: des modèles pas 100% UML

- Certains outils font le reverse de toute l'API Java et créent les classes Java dans le diagramme de classes UML
  - Inc. Pollue le diagramme de classe



Reverse Engineering: Exemples de règles

- À toute classe Java doit correspondre une classe UML portant le même nom que la classe Java.

- À toute interface Java doit correspondre une interface UML portant le même nom que l'interface Java.

- À tout attribut d'une classe Java *dont le type est un type primitif* doit correspondre une propriété appartenant à la classe UML correspondant à la classe Java. Le nom de la propriété doit être le même que le nom de l'attribut. Le type de la propriété doit être une correspondance UML du type de l'attribut Java. Si l'attribut est un tableau, la propriété peut avoir plusieurs valeurs (en fonction de la taille du tableau).

Reverse Engineering: Exemples de règles

- À tout attribut d'une classe Java *dont le type est une autre classe Java* doit correspondre une association UML entre la classe UML correspondant à la classe Java de l'attribut et la classe UML correspondant au type de l'attribut Java. Cette association doit être navigable vers la classe UML correspondant au type de l'attribut Java. Le nom de rôle de la classe correspondant au type de l'attribut doit être le même que le nom de l'attribut Java. Si l'attribut Java est un tableau, l'extrémité de l'association qui porte sur la classe UML correspondant au type de l'attribut Java doit spécifier que plusieurs objets peuvent être liés. Sinon, nous considérons que la multiplicité est 0..1

- À toute opération d'une classe Java doit correspondre une opération appartenant à la classe UML correspondant à la classe Java. Le nom de l'opération UML doit être le même que celui de l'opération Java. Pour chaque paramètre de l'opération Java doit correspondre un paramètre UML de même nom, dont la direction est in et dont le type est le type UML correspondant au type du paramètre Java.

Reverse Engineering: Exemples de règles

- Si une classe Java appartient à un package Java, ce dernier doit correspondre à un package UML correspondant au package Java qui doit contenir la classe UML correspondant à la classe Java.

- Si une classe Java importe un package Java, ce dernier doit correspondre à une relation d'import entre le package UML de la classe UML correspondant à la classe Java et le package UML correspondant au package Java importé.

- Si une classe Java hérite d'une autre classe Java, les classes UML correspondantes doivent avoir elles aussi une relation d'héritage.

- Si une classe Java réalise une interface, la classe UML correspondante doit aussi réaliser l'interface UML correspondante.

Round-Trip Engineering

Round-Trip Engineering

- Du code aux modèles et inversement
  - Java 2 UML et de UML 2 Java
- Correspondrait plus aux besoins de développement d'aujourd'hui
- Permettrait de garder modèles & code en parfaite synchronisation
  - Le talon d'Achilles des projets utilisant UML de nos jours
- Malheureusement, techno pas très mature encore
  - même si pas mal d'outils le proposent déjà (ex. RSA, Objecteering VP, NetBeans, Eclipse, etc.)
  - Pas de normes. Résultats différents d'un outil à un autre

Round-Trip Engineering: Attention!

- Si le point de départ est le code, Reverse + Génération de code retournera le même code
  - Dans le même outil (même règles). Pas de garanties autrement!

- L'opération inverse n'est pas garantie (Génération + reverse)
  - Certains concepts Java qui n'étaient pas présents avant l'opération de génération peuvent apparaître dans le modèle après le reverse

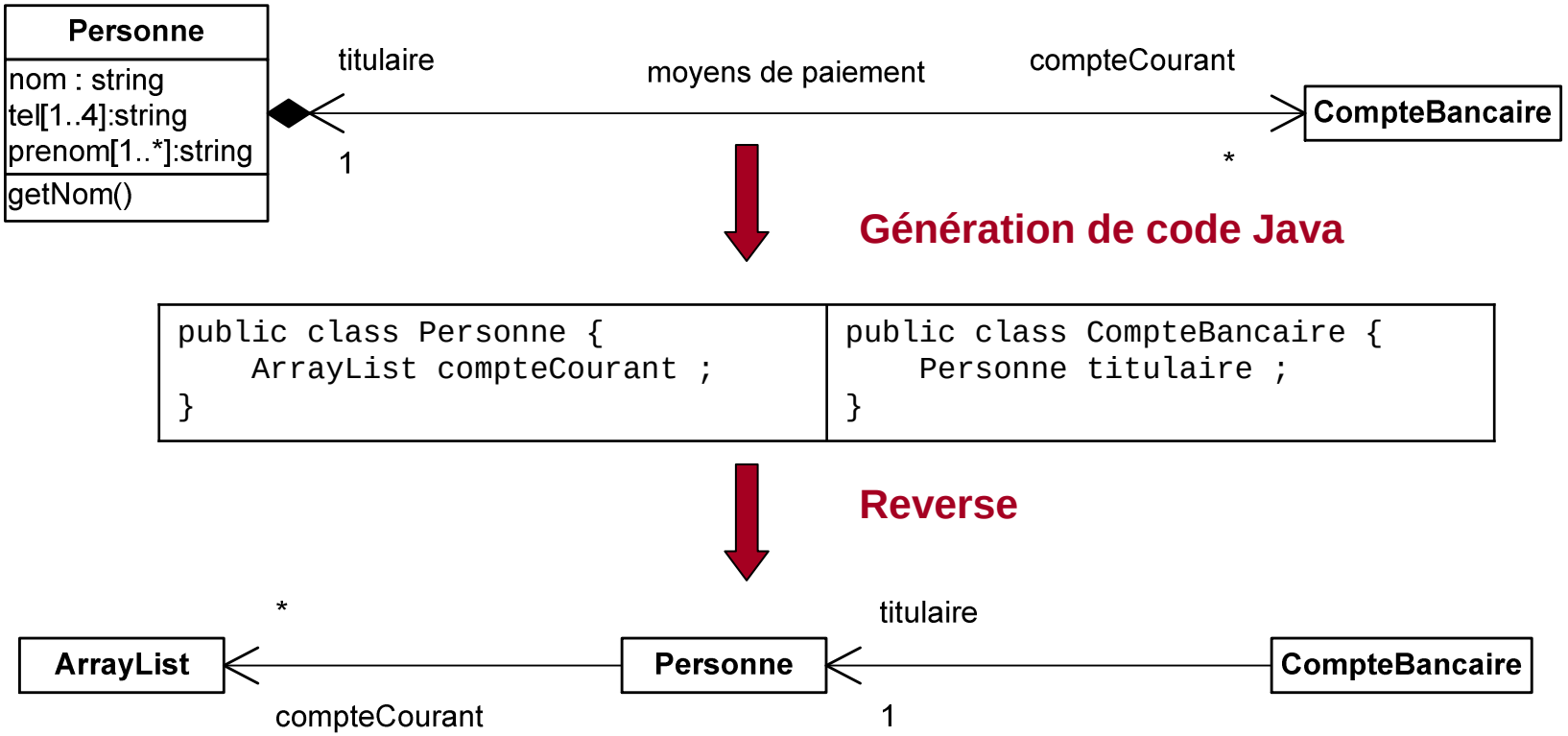
Round-Trip Engineering: Attention!

- Peu d'outils réussissent une synchronisation parfaite
  - Double association,
  - la modification du nom d'un l'attribut dans le code => un nouvel attribut dans le modèle (Objecteering utilise des id pour éviter ça!)

- Préférez des outils qui proposent une fonction Round-Trip plutôt qu'une combinaison de Génération + Reverse
  - Meilleure synchronisarion!

Round-Trip Engineering: Attention!

- Exemple où Génération + Reverse ne retourne pas le même modèle (ex. réalisé avec l'outil Objecteering). Source (X.Blanc et al. 06)



Conclusion

- Techniques phares pour la réussite de la vision MDE
- Une meilleure productivité
- Des efforts à faire coté normalisation

- Attention aux incompatibilités entre UML et les langages OO
- Certains outils permettent de paramétrer les moteurs de reverse et génération de code
- Peu d'outils actuellement avec des résultats satisfaisants à 100%

Lectures

- Software Engineering,
  - Ian Sommerville, Addison Wesley; 8 edition (15 Jun 2006), ISBN-10: 0321313798
- The Mythical Man-Month
  - Frederick P. Brooks JR., Addison-Wesley, 1995
- Cours de Software Engineering du Prof. Bertrand Meyer à cette @:
  - http://se.ethz.ch/teaching/ss2007/252-0204-00/lecture.html
- Cours d'Antoine Beugnard à cette @:
  - http://public.enst-bretagne.fr/~beugnard/

- UML Distilled 3rd édition, a brief guide to the standard object modeling language
  - Martin Fowler, Addison-Wesley Object Technology Series, 2003, ISBN-10: 0321193687
- UML2 pour les développeurs, cours avec exercices et corrigés
  - Xavier Blanc, Isabelle Mounier et Cédric Besse, Edition Eyrolles, 2006, ISBN-2-212-12029-X
- UML 2 par la pratique, études de cas et exercices corrigés,
  - Pascal Roques, 6ème édition, Edition Eyrolles, 2008
- Cours très intéressant du Prof. Jean-Marc Jézéquel à cette @:
  - http://www.irisa.fr/prive/jezequel/enseignement/PolyUML/poly.pdf
- La page de l'OMG dédiée à UML: http://www.uml.org/

- Design patterns. Catalogue des modèles de conception réutilisables
  - Richard Helm (Auteur), Ralph Johnson (Auteur), John Vlissides (Auteur), Eric Gamma (Auteur), Vuibert informatique (5 juillet 1999), ISBN-10: 2711786447