

# Correspondance/Mappage du Modèle au Code

Phase d'Implémentation

Reference: Bernd Bruegge & Allen H. Dutoit. **Object-Oriented Software Engineering using UML, Patterns and Java**

# Génie Logiciel base sur les modèles

- La vision
  - Pendant la conception est construit un modèle objet conceptuel qui réalise le modèle des use case et c'est la base de l'implementation (model-driven design)
- La réalité
  - Travailler sur le modèle de conception objet implique de nombreuses activités sujettes aux erreurs
  - Exemples:
    - Un nouveau paramètre doit être ajouté à une opération. En raison de la pression du temps, il est ajouté au code source, mais pas au modèle objet
    - Des attributs supplémentaires sont ajoutés à une classe, mais la table de base de données n'est pas mise à jour (par conséquent, les nouveaux attributs ne sont pas persistants).

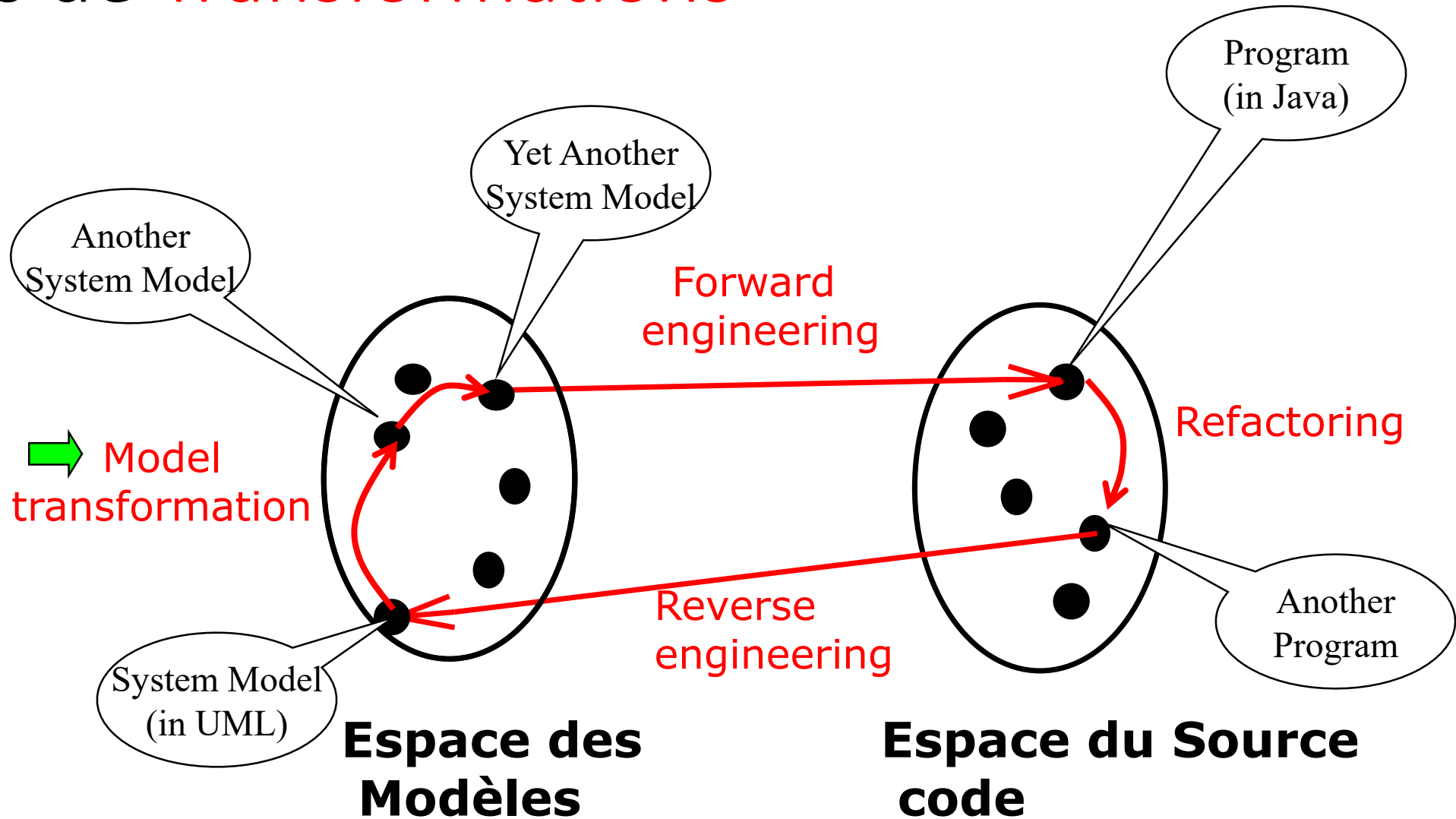
# Autres activités de conception

- Les langages de programmation ne prennent pas en charge le concept d'**association** UML
  - Les associations du modèle objet doivent être transformées en collections de références d'objets
- De nombreux langages de programmation ne prennent pas en charge les **contrats** (invariants, pré et post conditions)
  - Les développeurs doivent donc transformer manuellement la spécification du contrat en code source pour détecter et gérer les violations de contrat
- Le client **modifie les exigences** lors de la conception de l'objet
  - Le développeur doit modifier la spécification d'interface des classes concernées
- Toutes ces activités de conception d'objets **posent des problèmes**, car elles doivent être effectuées manuellement.

# Espaces et Transformations

- Prenons en main ces problèmes
- Pour ce faire, nous distinguons deux types d'espaces
  - l'espace modèle et l'espace code source
- et 4 types de transformations différents
  - Transformation de modèle,
  - Forward engineering (Ingénierie avancée),
  - Reverse engineering (Ingénierie inverse),
  - Refactoring (Refactorisation).

# Types de Transformations



# Quelques exemples

- Transformations de modèle

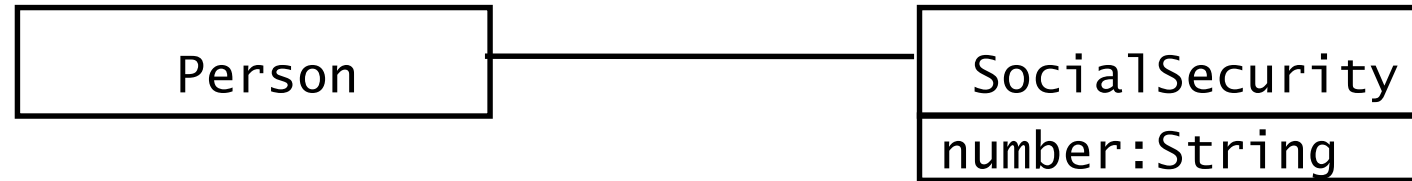
- Objectif : Optimiser le modèle de conception d'objets
  - ➡ • Réduire des objets
  - Retarder les calculs coûteux

- Forward Engineering

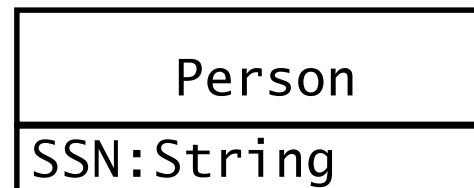
- Objectif : Implémenter le modèle de conception objet dans un langage de programmation
- Mappage de l'héritage
- Mappage des associations
- Mappage des contrats aux exceptions
- Mappage de modèles d'objets sur des tables

# Réduction des objets

Modèle objet avant transformation:



Modèle objet après transformation:



Transformer un objet en attribut d'un autre objet est généralement fait, si **l'objet n'a pas de comportement dynamique intéressant** (uniquement les opérations get et set).

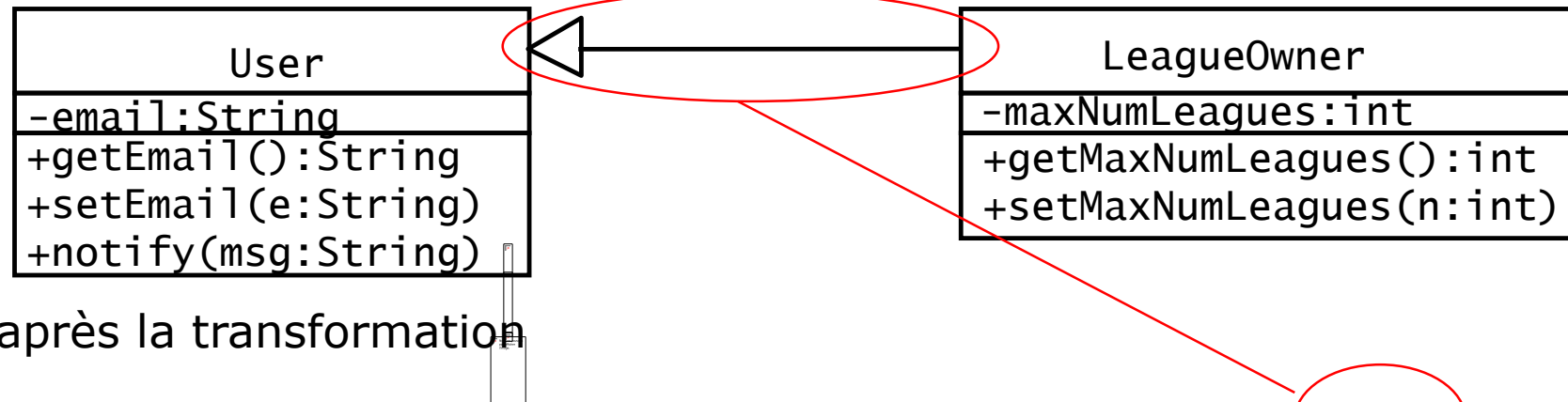
# Forward Engineering: cas de l'héritage

- **Objectif** : Nous avons un modèle UML avec héritage. Nous voulons le traduire en code source
- **Question**: Quels mécanismes du langage de programmation peuvent être utilisés ?
  - Interressons nous à Java
- Java offre les mécanismes suivants:
  - Remplacement (Overriding) des méthodes (par défaut en Java)
  - Classes finales
  - Méthodes finales
  - Méthodes abstraites
  - Classes abstraites
  - Interfaces



# Exemple avec l'héritage

Modèle objet avant la transformation:



Code Source après la transformation

```
public class User {
    private String email;
    public String getEmail() {
        return email;
    }
    public void setEmail(String value){
        email = value;
    }
    public void notify(String msg) {
        // ....
    }
}
```

```
public class LeagueOwner extends User {
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
        return maxNumLeagues;
    }
    public void setMaxNumLeagues
        (int value) {
        maxNumLeagues = value;
    }
}
```

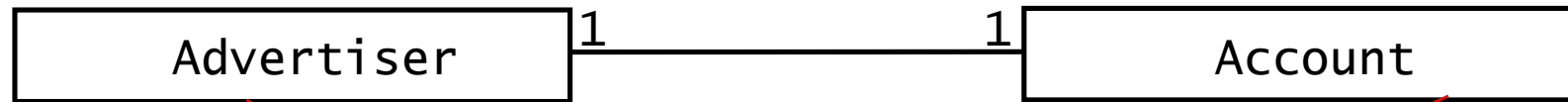
# Réaliser l'héritage en Java

- **Realisation de la spécialisation et généralization**
  - Définition des sous classes
  - Mot clé Java: **extends**
- **Realisation de l'héritage simple**
  - Overriding des méthodes interdit
  - Mot clé Java : **final**
- **Realisation de l'implémentation de l'héritage**
  - Aucun mot clé n'est requis:
    - Overriding des méthodes par défaut
- **Realisation de la spécification de l'héritage**
  - Spécification d'une interface
  - Mots clés Java: **abstract, interface**

# Mappage des Associations

1. Unidirectionnel association un-à-un
2. Bidirectionnel association un-à-un
3. Bidirectionnel association un-à-plusieurs
4. Bidirectionnel association plusieurs-à-plusieurs
5. Bidirectionnel association qualifiée.

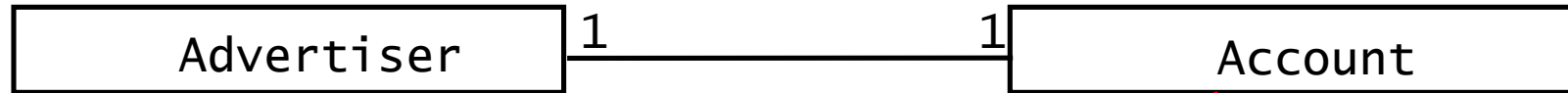
# Association unidirectionnel 1-à-1



Code source après transformation:

```
public class Advertiser {  
    private Account account;  
    public Advertiser() {  
        account = new Account();  
    }  
    public Account getAccount() {  
        return account;  
    }  
}
```

# Association Bidirectionnelle 1-à-1

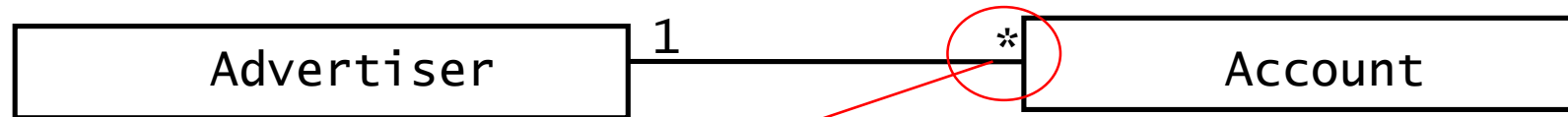


Code source après transformation:

```
public class Advertiser {
    /* account is initialized
    * in the constructor and never
    * modified. */
    private Account account;
    public Advertiser() {
        account = new Account(this);
    }
    public Account getAccount() {
        return account;
    }
}
```

```
public class Account {
    /* owner is initialized
    * in the constructor and
    * never modified. */
    private Advertiser owner;
    public Account(owner:Advertiser) {
        this.owner = owner;
    }
    public Advertiser getOwner() {
        return owner;
    }
}
```

# Association Bidirectionnelle 1-à-n



Source code after transformation:

```
public class Advertiser {
    private Set accounts;
    public Advertiser() {
        accounts = new HashSet();
    }
    public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
    }
    public void removeAccount(Account a) {
        accounts.remove(a);
        a.setOwner(null);
    }
}
```

```
public class Account {
    private Advertiser owner;
    public void setOwner(Advertiser newOwner) {
        if (owner != newOwner) {
            Advertiser old = owner;
            owner = newOwner;
            if (newOwner != null)
                newOwner.addAccount(this);
            if (oldOwner != null)
                old.removeAccount(this);
        }
    }
}
```

# Association Bidirectionnelle n-à-n



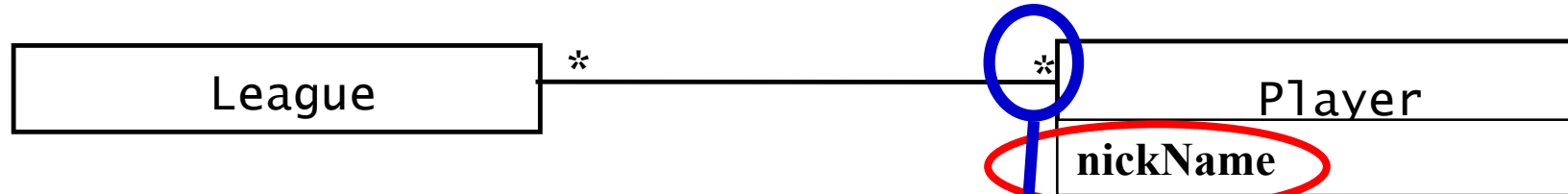
Code Source après transformation

```
public class Tournament {  
    private List players;  
    public Tournament() {  
        players = new ArrayList();  
    }  
    public void addPlayer(Player p) {  
        if (!players.contains(p)) {  
            players.add(p);  
            p.addTournament(this);  
        }  
    }  
}
```

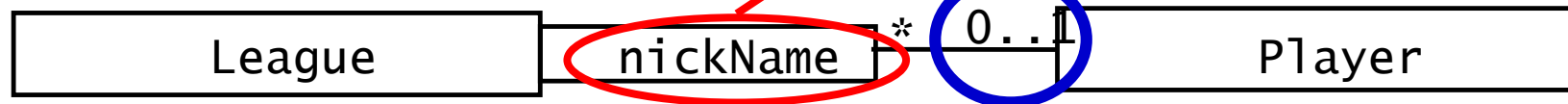
```
public class Player {  
    private List tournaments;  
    public Player() {  
        tournaments = new ArrayList();  
    }  
    public void addTournament(Tournament t) {  
        if (!tournaments.contains(t)) {  
            tournaments.add(t);  
            t.addPlayer(this);  
        }  
    }  
}
```

# Association bidirectionnelle qualifiée

Avant la transformation **de modèle**

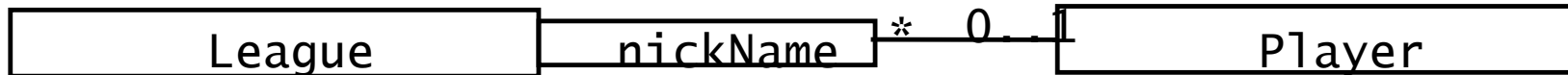


Après la transformation de modèle





# Association qualifiée cntd.



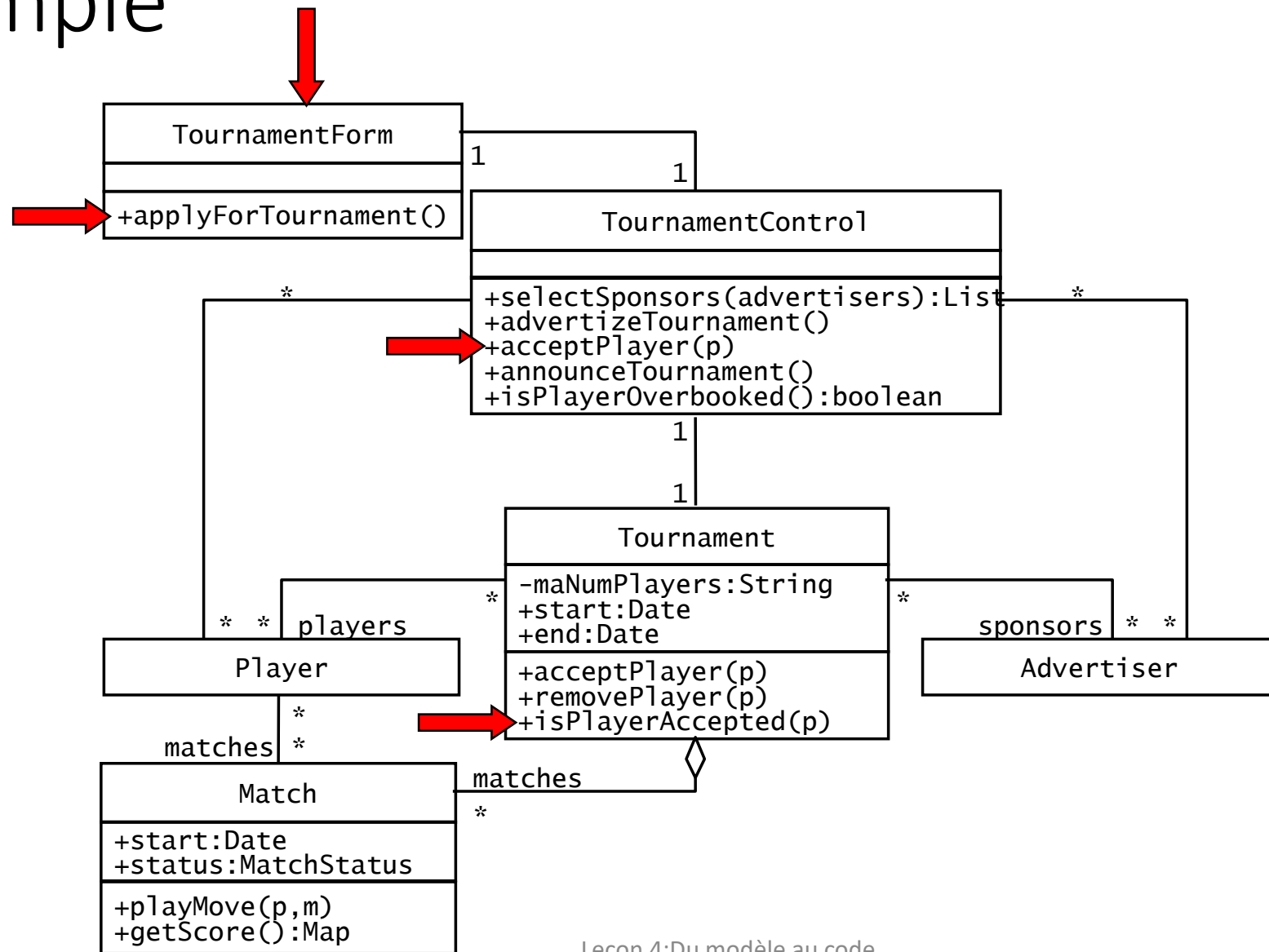
```
public class League {
    private Map players;
    public void addPlayer
        (String nickName, Player p) {
        if
        (!players.containsKey(nickName)) {
            players.put(nickName, p);
            p.addLeague(nickName, this);
        }
    }
}
```

```
public class Player {
    private Map leagues;
    public void addLeague
        (String nickName, League l) {
        if (!leagues.containsKey(l)) {
            leagues.put(l, nickName);
            l.addPlayer(nickName, this);
        }
    }
}
```

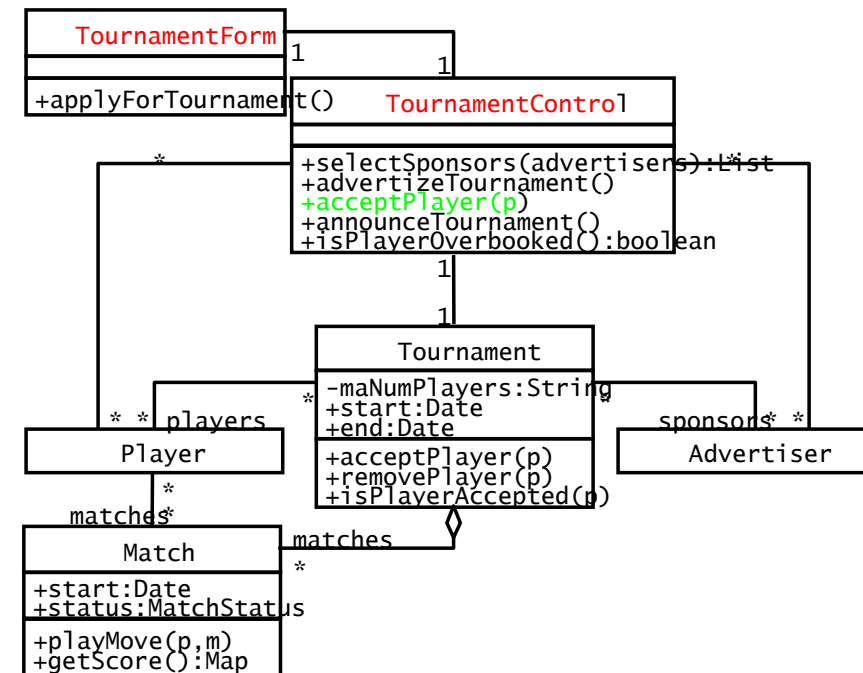
# Implementation des Contracts

- De nombreux langages orientés objet n'ont pas de support intégré pour les contrats
- Cependant, si les exceptions sont pris en charge, nous pouvons utiliser leurs **mécanismes d'exception** pour signaler et gérer les violations de contrat.
- En Java, nous utilisons le mécanisme **try-throw-catch**
- Exemple:
  - Supposons que l'opération `acceptPlayer()` de `TournamentControl` est invoquée avec un joueur qui fait déjà partie du tournoi
  - Modèle UML
  - Dans ce cas, `acceptPlayer()` dans `TournamentControl` devrait lever une exception de type `KnownPlayer`
    - Code source Java.

# Exemple



# Implementation en Java



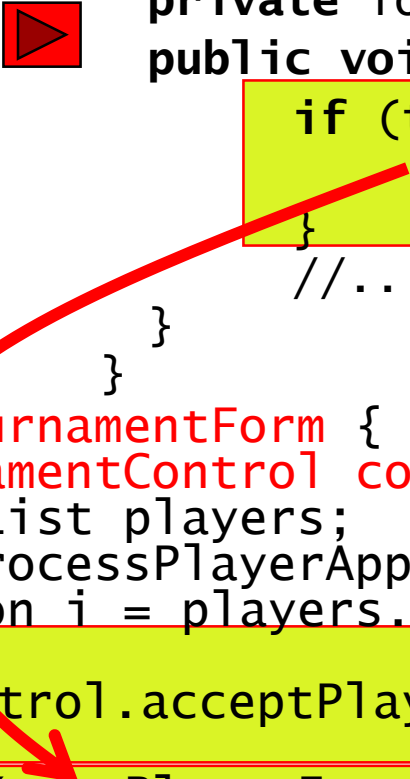
```

public class TournamentForm {
    private TournamentControl control;
    private ArrayList players;
    public void processPlayerApplications() {
        for (Iterator i = players.iterator(); i.hasNext();) {
            try {
                control.acceptPlayer((Player)i.next());
            }
            catch (KnownPlayerException e) {
                // If exception was caught, log it to console
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}

```

# Mécanisme try-throw-catch en Java

```
public class TournamentControl {  
    private Tournament tournament;  
    public void addPlayer(Player p) throws KnownPlayerException {  
        if (tournament.isPlayerAccepted(p)) {  
            throw new KnownPlayerException(p);  
        }  
        //... Normal addPlayer behavior  
    }  
}  
  
public class TournamentForm {  
    private TournamentControl control;  
    private ArrayList players;  
    public void processPlayerApplications() {  
        for (Iteration i = players.iterator(); i.hasNext();) {  
            try {  
                control.acceptPlayer((Player)i.next());  
            }  
            catch (KnownPlayerException e) {  
                // If exception was caught, log it to console  
                ErrorConsole.log(e.getMessage());  
            }  
        }  
    }  
}
```

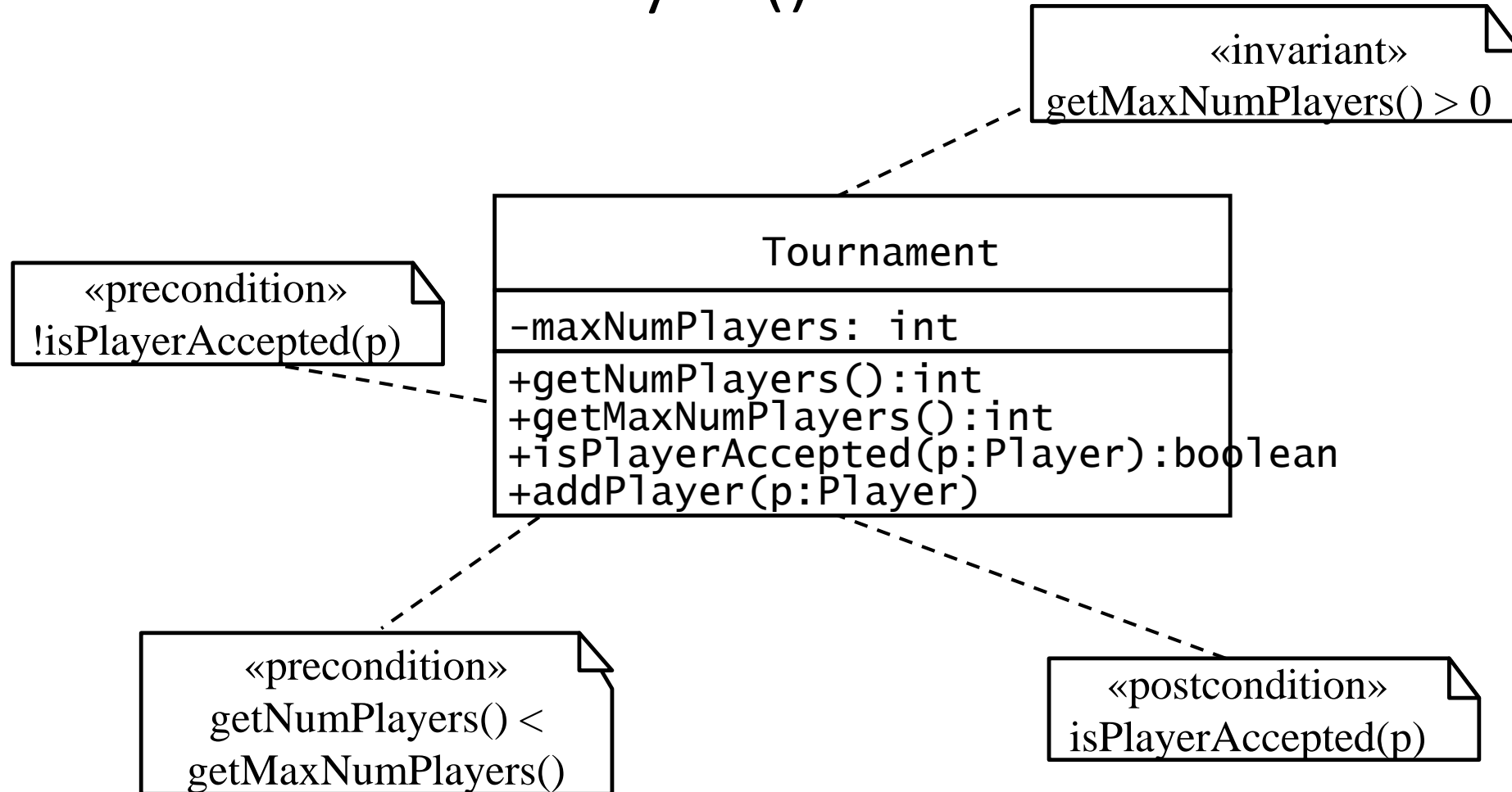


A red arrow originates from the `throw new KnownPlayerException(p);` line in the `addPlayer` method of `TournamentControl` and points to the `catch (KnownPlayerException e)` block in the `processPlayerApplications` method of `TournamentForm`, illustrating the exception handling mechanism.

# Implementer un Contract

- **Vérifiez chaque condition préalable :**
  - Avant le début de la méthode avec un test pour vérifier la condition préalable pour cette méthode
    - Lever une exception si la condition préalable est évaluée comme fausse
- **Vérifiez chaque postcondition :**
  - A la fin de la méthode écrire un test pour vérifier la postcondition
    - Lève une exception si la postcondition est évaluée comme fausse. Si plus d'une postcondition n'est pas satisfaite, lever une exception uniquement pour la première violation.
- **Vérifiez chaque invariant :**
  - Vérifier les invariants en même temps lors de la vérification des préconditions et lors de la vérification des postconditions
- **Traiter l'héritage :**
  - Ajoutez le code de vérification pour les préconditions et les postconditions également dans les méthodes qui peuvent être appelées depuis la classe.

# Une implementation du contrat Tournament.addPlayer()



# Heuristiques: Contracts -> Exceptions

- L'exécution du code de vérification ralentit votre programme
  - S'il est trop lent, omettez le code de vérification pour les méthodes privées et protégées
  - S'il est encore trop lent, privilégiez les composants ayant la durée de vie la plus longue
    - Omettez de vérifier le code pour les postconditions et les invariants pour tous les autres composants.
- Pour une transformation donnée, utilisez toujours le même outil
- Gardez les contrats dans le code source, pas dans le modèle de conception d'objet
- Utiliser les mêmes noms pour les mêmes objets
- Avoir un guide de style pour les transformations (Martin Fowler)



# Résumé

- Quatre **concepts de mappage** :
  - Transformation de modèle
  - Forward engineering
  - Refactoring
  - Reverse engineering
- **Techniques de transformation** de modèles et de forward engineering:
  - Optimiser le modèle de classe
  - Mappage des associations aux collections
  - Mappage des contrats aux exceptions
  - Mappage **du modèle de classe aux schémas de stockage** (en exercice)