IUT Lyon 1

l'excellence technologique

Refactoring

LP IEM

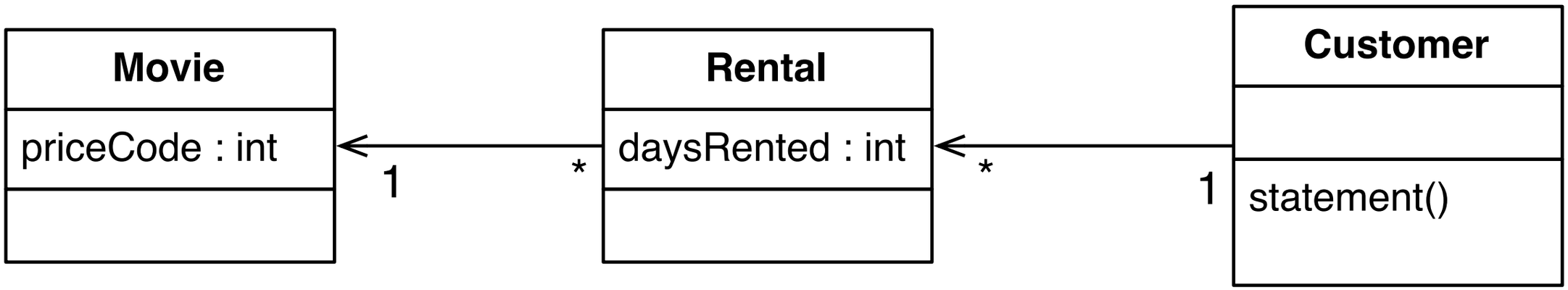
Refactoring

- Refactoring (reconception)
  - Définition par un exemple

- Nous partons d'un programme simple de gestion d'un videoclub qui doit permettre d'imprimer un reçu (statement) affichant le montant correspondants aux films loués ainsi que le nombre de points de fidélités gagnés

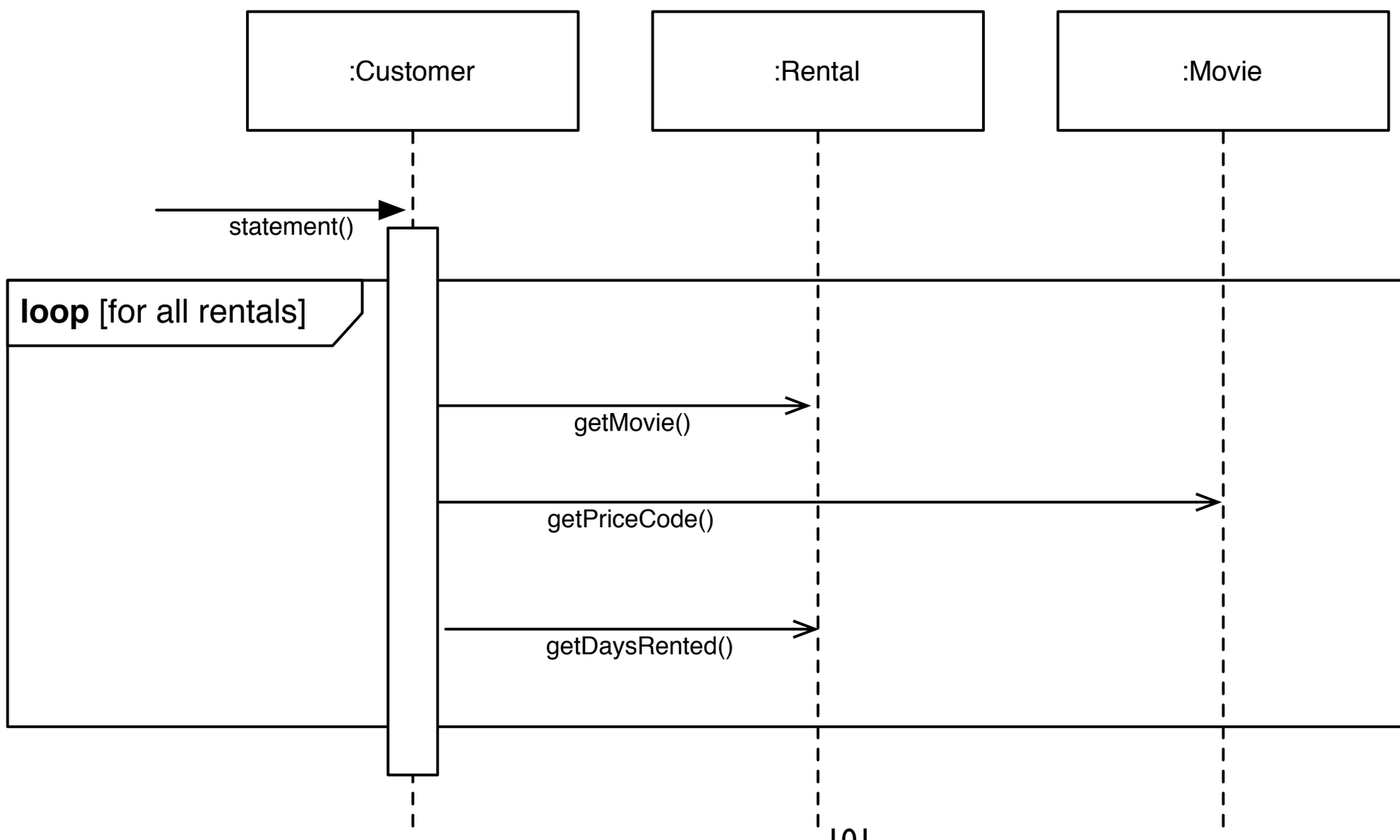
- Le montant et les points dépendent de la durée de location et du type de film (Regular / Children / New Release)

- Refactoring video club
  - Diagramme de classe initial simplifié



Refactoring video club

Diagramme de séquence initial simplifié



Refactoring

- Refactoring video club
  - Récupérer le code sous SPIRAL et l'analyser
  - Premières impressions
    - Pas très bien conçu / pas très orienté objet
    - Pour un petit programme comme celui-ci ce n'est pas gênant mais s'il s'agissait d'un morceau d'un programme plus important cela deviendrait problématique
    - La méthode statement() est longue et effectue du travail qui devrait être fait dans d'autres classes (elle utilise des informations propres aux films ou à la location)

- Refactoring video club
  - Le compilateur se moque de savoir si le code est moche ou élégant.
  - Cette mauvaise conception devient un véritable problème si on veut modifier des choses dans le programme ➙ il y a un humain dans la boucle qui lui, accorde de l'importance à la propreté du code
  - Un programme mal conçu sera difficile à modifier
    - les modifications prendront plus de temps
    - les chances d'introduire des bugs sont grandes

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand" M. Fowler

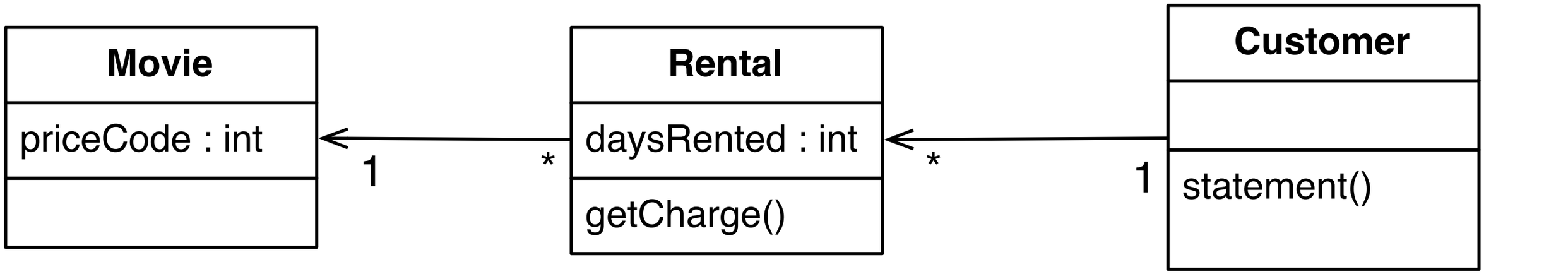
- Refactoring video club
  - Nous savons déjà que ce programme va être modifié
    - Ajout de la génération d'un reçu au format html
      - En l'état aucun morceau de code ne pourra être réutilisé pour faire ce changement alors qu'une part importante de la logique est identique.
    - La seule solution consisterait à écrire une méthode htmlStatement() qui dupliquerait une part importante du code
    - Pour l'instant, cela pourrait se faire facilement (un copier/coller et le remplacement de quelques lignes pour adapter au html)
    - Mais que se passera-t-il lorsque les règles de tarification changeront ?

- Refactoring video club
- Quand vous vous rendez compte que l'ajout d'une fonctionnalité ne cadre pas avec la structure du programme, c'est un bon signe qu'un refactoring est nécessaire
- Avant de se lancer dans un refactoring, il faut disposer d'une bonne suite de tests unitaires afin de vérifier à chaque étape que l'on n'a rien cassé !

- Refactoring video club
  - Première étape
    - Première cible : la méthode statement() ➙ trop longue
      - Il faut la décomposer en plus petits morceaux
      - Utilisation du refactoring "Extract Method"
        - Extraction du switch vers une nouvelle méthode amountFor qui calcule le montant pour l'un des films rendus en utilisant le menu Refactor d'Eclipse
        - On valide par l'exécution des tests unitaires que tout fonctionne toujours bien

- Refactoring video club
  - Deuxième étape
    - Renommage (avec le menu refactor d'Eclipse) de variables dans la méthode amountFor
      - each → aRental et thisAmount → result
      - Test !
  - Troisième étape
    - amountFor utilise des informations de la classe Rental mais pas de la classe Customer
    - Refactoring "Move Method"
      - Déplacement de la méthode vers la classe Rental en la renommant au passage getCharge()
    - Test !

- Refactoring video club
  - Troisième étape



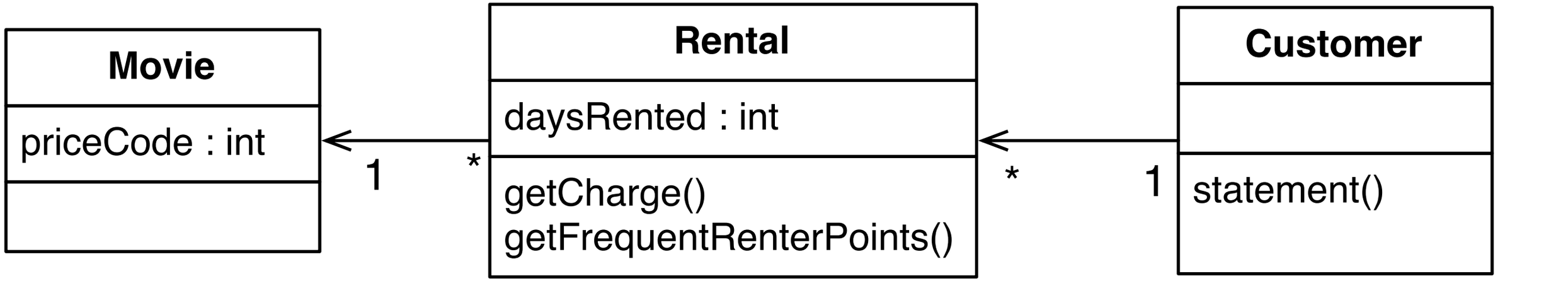
- Refactoring video club
  - Quatrième étape
    - La variable locale thisAmount se voit assigner le résultat du calcul each.getCharge et ne change pas après
    - Refactoring "Replace Temp with Query"
      - On supprime cette variable locale et on la remplace par un appel de méthode et on teste !
  - Motivation
    - Les variables temporaires locales génèrent le besoin de passer beaucoup de paramètres entre méthodes et peuvent complexifier la lecture du programme
    - Inconvénient : l'indirection ralentie un peu l'exécution mais il reste à prouver que cela a un impact global sur les performances de l'application

- Refactoring video club
  - Remarques sur l'optimisation
    - Un code bien "refactorisé" sera plus facile à optimiser si nécessaire
      - Exemple : dans la méthode getCharge on pourrait aisément garder en cache le résultat du dernier calcul si celui-ci est particulièrement long
    - En règle général, il ne faut pas essayer d'optimiser de façon prématurée
      - "First make it right, then make it fast"
      - L'expérience montre que les programmes passent l'essentiel de leur temps dans une toute petite fraction du programme (≃10 %)
      - Ne pas perdre de temps à optimiser les 90 % du programme qui n'en ont pas besoin au risque de les rendre moins clairs et plus dur à faire évoluer

- Refactoring video club
  - Remarques sur l'optimisation
    - Une fois que le programme fonctionne comme il faut, utiliser un profiler pour analyser ses performances et identifier les endroits du code sur lesquels porter les efforts d'optimisation
      - Valider après chaque optimisation l'effet sur les performances avec le profiler et annuler ces modifications si elles n'apportent pas de gain
    - Un code bien conçu vous laissera plus de temps pour la phase d'optimisation et vous facilitera ce travail

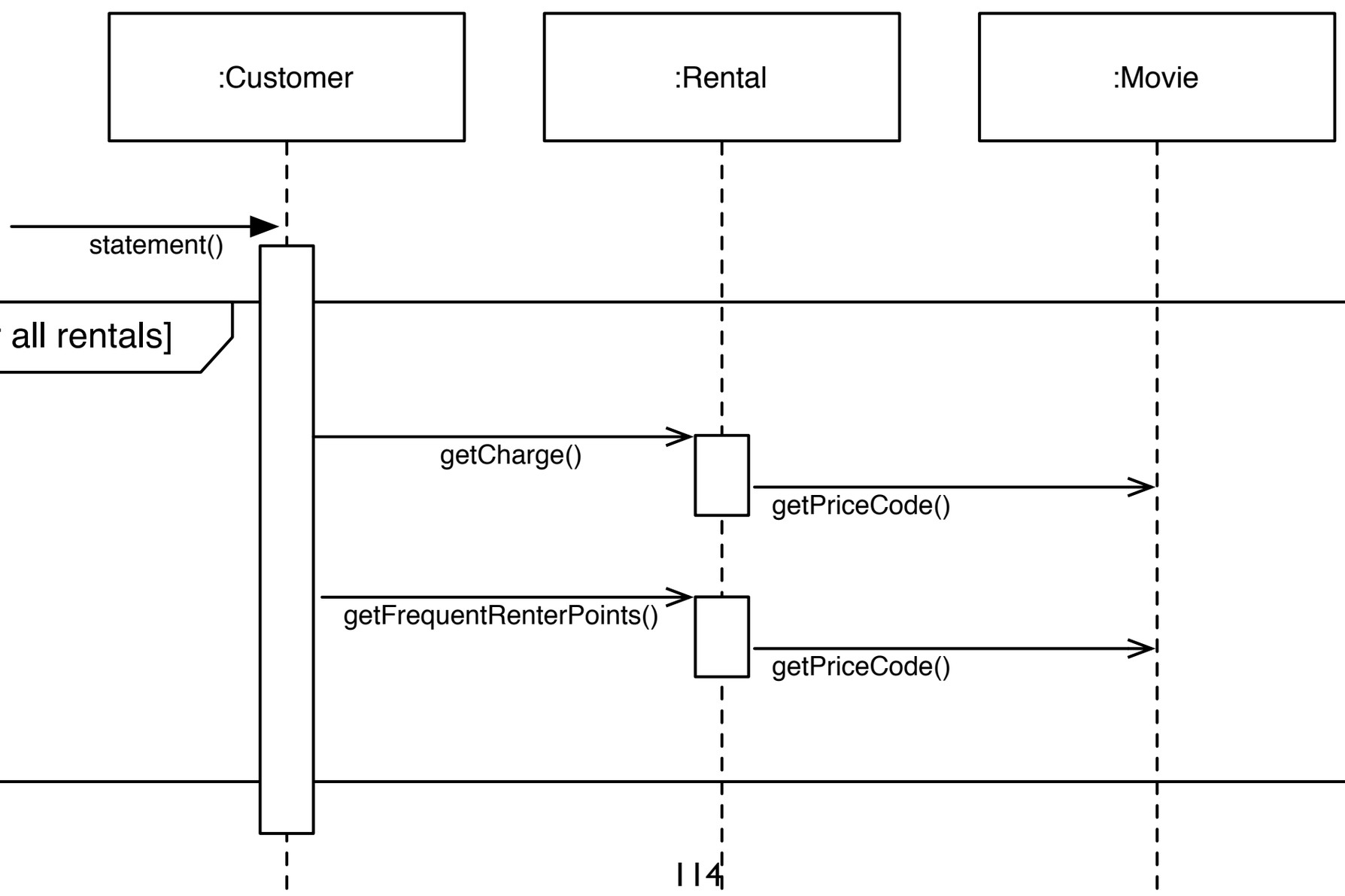
- Refactoring video club
  - Cinquième étape
    - On procède de la même manière avec le calcul des points de fidélités et on extrait une méthode getFrequentRenterPoints()
    - Cette méthode a également sa place dans la classe Rental
    - Test !

- Refactoring video club
  - Cinquième étape



Refactoring video club

Cinquième étape

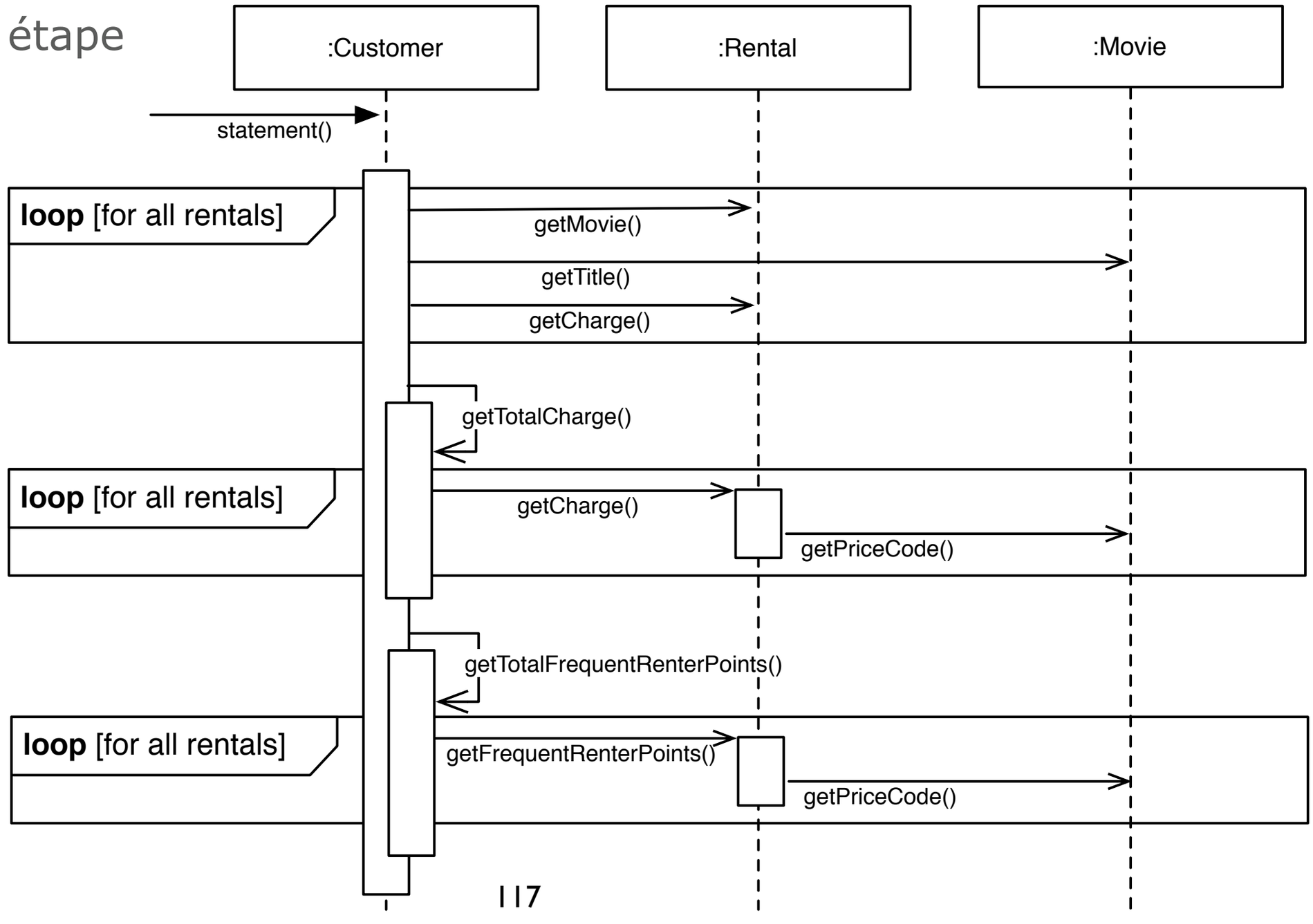


- Refactoring video club
  - Sixième étape
    - Refactoring "Removing Temps"
      - Remplacement des variables temporaires totalAmount et frequentRentalPoints par des méthodes
        - ces calculs seront nécessaires pour les versions ASCII et HTML
        - en déplaçant ce code vers des méthodes accessibles depuis n'importe quelle méthode de la classe cela encourage un code plus propre et permet d'éviter les méthodes longues et complexes
      - Test !

- Refactoring video club
  - Sixième étape
    - Ce dernier refactoring rajoute des lignes de code et conduit à la multiplication des structures de boucles
      - Une seule boucle avant le refactoring
      - Trois boucles après
    - Cette modification pourrait avoir un impact sur les performances
      - tout est dans le "pourrait"
      - tant que l'on n'a pas déterminé avec un profiler que ces boucles ont un impact sur les performances du systèmes il n'y a pas de raison de se priver d'un refactoring qui rend possible l'écriture de la nouvelle méthode htmlStatement() en réutilisant un maximum de code

Refactoring video club

Sixième étape



- Refactoring video club
  - Septième étape
    - Ajout de la méthode htmlStatement()

```
public String htmlStatement() {
        String result = "<META HTTP-EQUIV=\"Content-Type\" CONTENT=\"text/html; charset=UTF-8\">";
    result+= "<H1>Rentals for <EM>" + getName() + "</EM></ H1><P>\n";
    for(Rental each : _rentals) {
        //show figures for each rental
        result += each.getMovie().getTitle()+ ": " +
                    String.valueOf(each.getCharge()) + "<BR>\n";
    }
    //add footer lines
    result +=  "<P>You owe <EM>" + String.valueOf(getTotalCharge()) + "</EM><P>\n";
    result += "On this rental you earned <EM>" +
        String.valueOf(getTotalFrequentRenterPoints()) +
        "</EM> frequent renter points<P>";
    return result;
}
```

- Refactoring video club
  - Septième étape
    - Aucun code de calcul n'a été copié/collé
    - Le changement des règles de calcul n'impactera pas les deux méthodes d'édition d'un reçu
    - On pourrait ajouter une nouvelle méthode dans les tests unitaires pour valider l'édition du reçu au format html

- Refactoring video club
  - Huitième étape
    - Les patrons du videoclub sont en train de réfléchir à une nouvelle classification des films et une redéfinition des règles de calcul des tarifs de location et de l'attribution de points de fidélités en fonction du type de film
      - En l'état actuel du code, effectuer ce type de modification n'est pas commode, il faut aller dans getCharge et getFrequentRenterPoints pour modifier un code avec des conditions sur les types de films

- Refactoring video club
  - Huitième étape
    - Cette méthode doit être modifiée à chaque nouvel ajout d'un type de film ou à chaque modification
    - Il faut aussi effectuer des modifications similaires dans la méthode getFrequentRenterPoints()
    - Mauvaise idée de faire un switch sur un attribut d'une autre classe (Movie)
      - cette logique devrait se trouver dans la classe Movie

```
double getCharge() {
        double result;
        result = 0;
        // determine amounts for each line
        switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
             result += 2;
             if (getDaysRented() > 2)
                  result += (getDaysRented() - 2) * 1.5;
             break;
        case Movie.NEW_RELEASE:
             result += getDaysRented() * 3;
             break;
        case Movie.CHILDRENS:
             result += 1.5;
             if (getDaysRented() > 3)
                  result += (getDaysRented() - 3) * 1.5;
             break;

        }
        return result;
}
```

- Refactoring video club
  - Huitième étape
    - Déplacement dans la classe Movie
      - il faudra passer le nombre de jours de la location en paramètre et supprimer les appels à getDaysRented() pour les remplacer par le paramètre
    - Effectuer le changement localement (en surchargeant getCharge) et déplacer la méthode
    - Test !

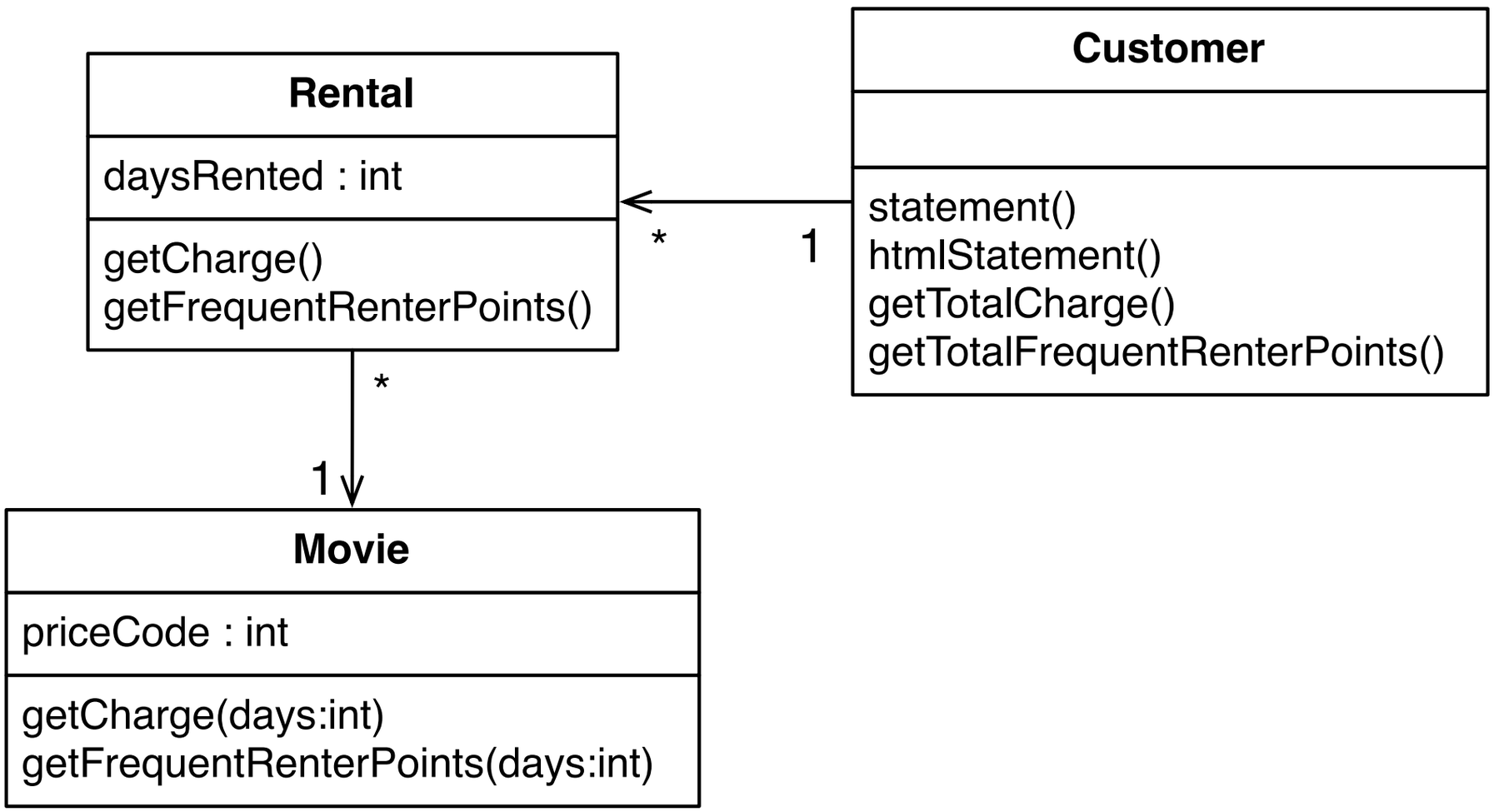
```
double getCharge() {
    double result;
    result = 0;
    // determine amounts for each line
    switch (getMovie().getPriceCode()) {
    case Movie.REGULAR:
        result += 2;
        if (getDaysRented() > 2)
            result += (getDaysRented() - 2) * 1.5;
        break;
    case Movie.NEW_RELEASE:
        result += getDaysRented() * 3;
        break;
    case Movie.CHILDRENS:
        result += 1.5;
        if (getDaysRented() > 3)
            result += (getDaysRented() - 3) * 1.5;
        break;

    }
    return result;
}
```

Refactoring

- Refactoring video club
  - Huitième étape
    - On peut se demander s'il vaut mieux passer le nombre de jours de la location à la classe Movie ou s'il vaut mieux passer le type de film à la classe Rental
  - Le champ qui est le plus susceptible de changer est le type de film c'est donc au niveau de la classe film qu'il faut le confiner

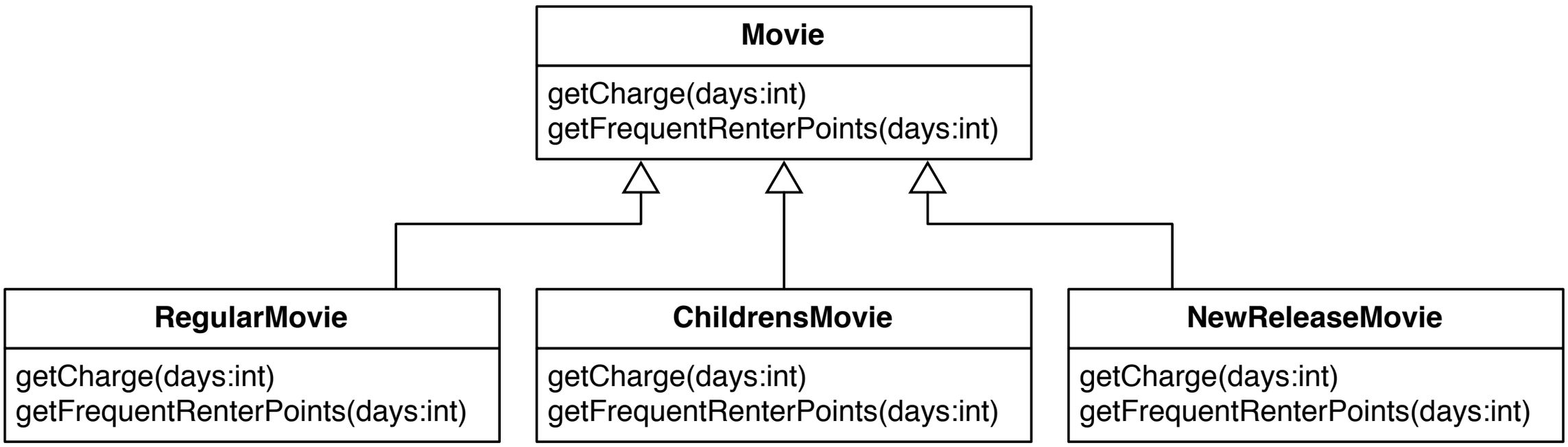
```
double getCharge(int daysRented) {
        double result;
        result = 0;
        // determine amounts for each line
        switch (getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (daysRented > 2)
                result += (daysRented - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented > 3)
                result += (daysRented - 3) * 1.5;
            break;

        }
        return result;
    }
```

- Refactoring video club
  - Huitième étape
    - Déplacer de la même manière la méthode getFrequentRenterPoints()

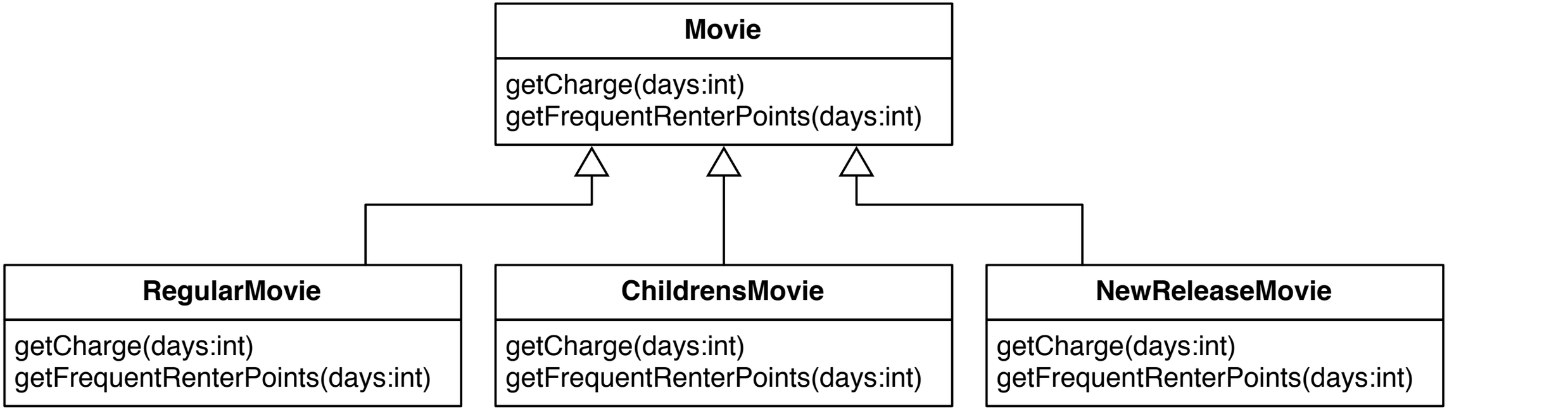


- Refactoring video club
  - Neuvième étape
    - Refactoring "Replace conditional with polymorphism"
      - Création de nouveaux types pour gérer les spécificités de chaque type de film de manière orientée objet
      - Nous avons différents types de films qui ont différentes manières de répondre aux même questions
        - cela doit faire penser à l'héritage !

- Refactoring video club
  - Neuvième étape
    - Refactoring "Replace conditional with polymorphism"
      - Première idée

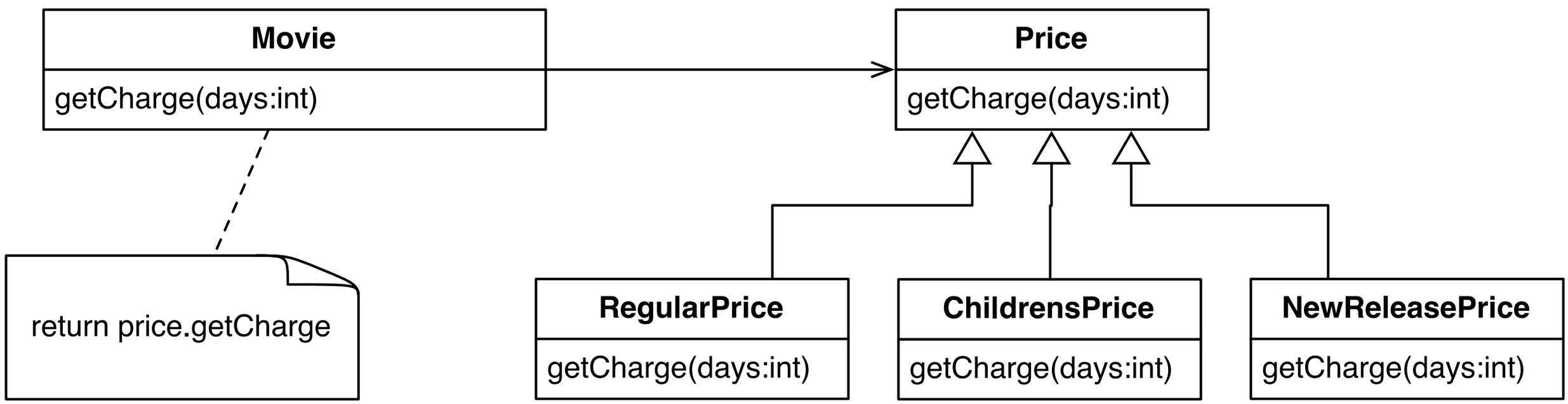


- Refactoring video club
  - Neuvième étape
    - Refactoring "Replace conditional with polymorphism"
      - Première idée



Un film peut changer de classification au cours de sa vie mais un objet ne peut pas changer de type

- Refactoring video club
  - Neuvième étape
    - Refactoring "Replace conditional with polymorphism"
      - Solution : Design Pattern State



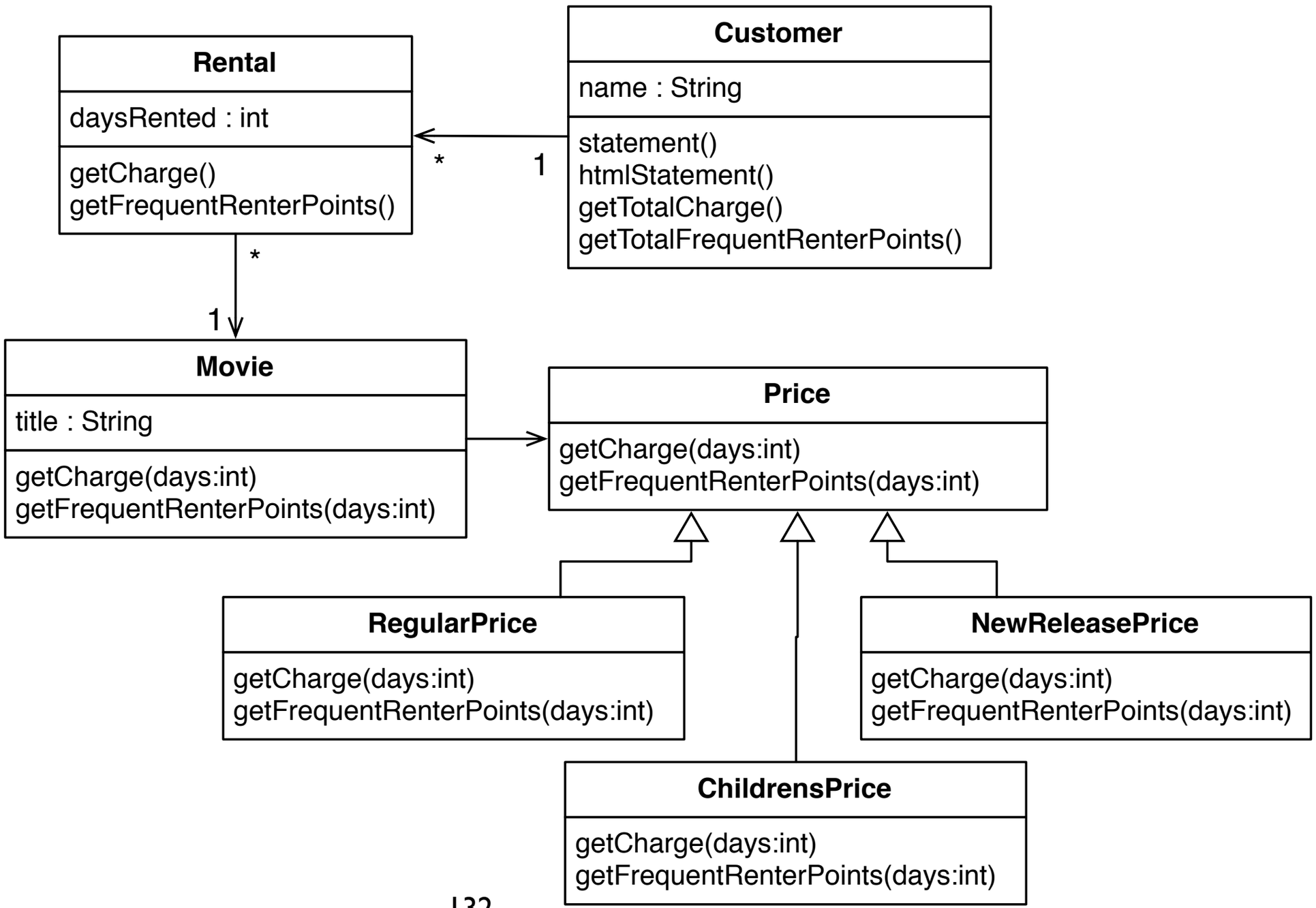
- Refactoring video club
  - Neuvième étape
    - Refactoring "Replace conditional with polymorphism"
      - Solution : Design Pattern State
        - Permet à un objet de modifier son comportement lorsque son état interne change. L'objet en question donne l'illusion de changer de classe
        - Utilisation de la composition : adjonction d'un objet auquel le travail qui dépend de l'état actif est délégué
        - Utilisation du polymorphisme pour définir une "interface" commune et une hiérarchie d'héritage implémentant les méthodes dont l'exécution doit dépendre de l'état

- Refactoring video club
  - Neuvième étape
    - Refactoring "Replace conditional with polymorphism"
      - Encapsuler les accès au champ priceCode (passage obligatoire par des setters/getters) : utiliser le menu refactor d'eclipse
      - Créer une classe Price avec une méthode abstraite getPriceCode
      - Remplacer la variable d'instance priceCode de la classe Movie par une référence vers un objet de type Price
      - Modifier getPriceCode() et setPriceCode() dans la classe Movie en faisant appel à cet objet et tester

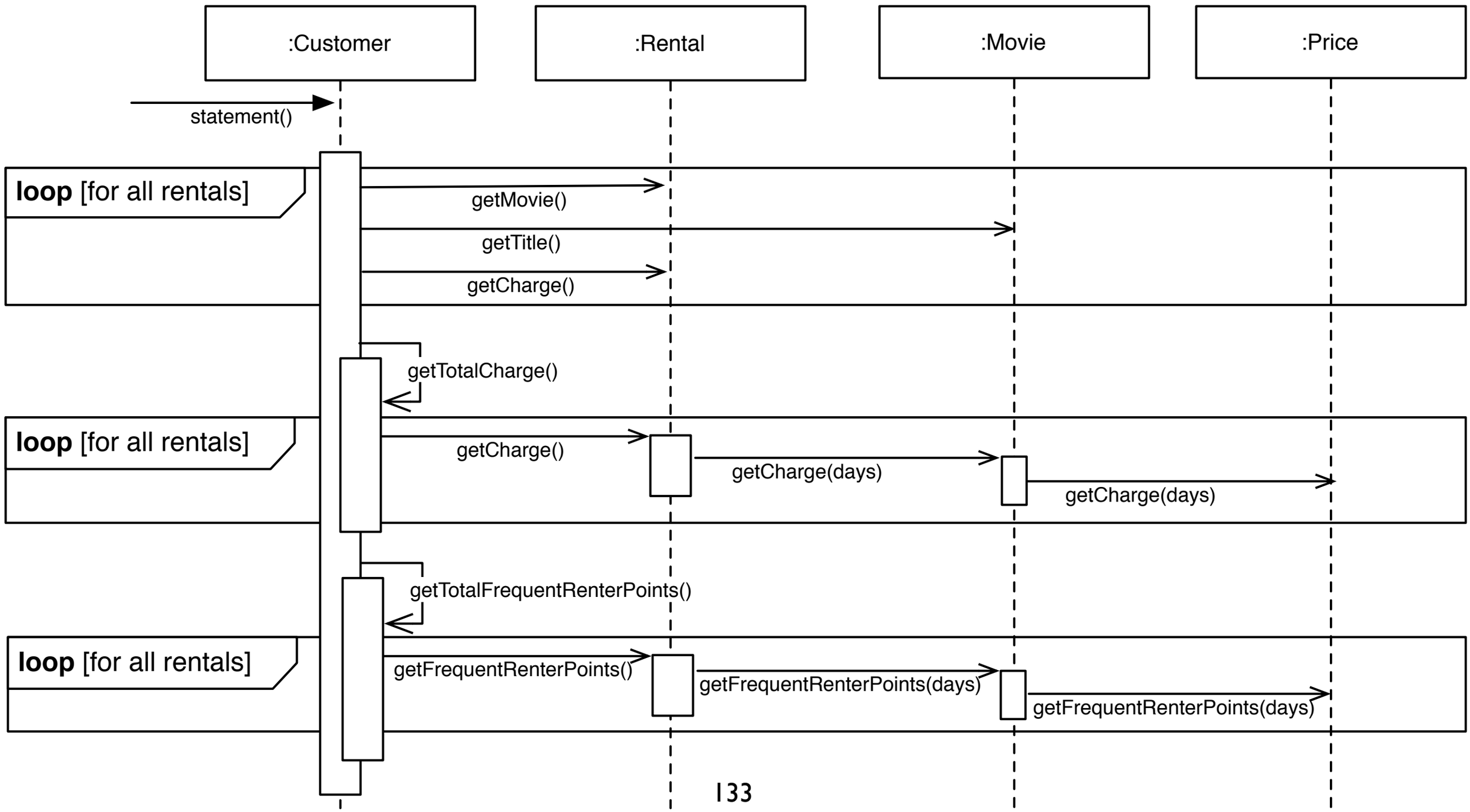
```
abstract class Price {
     abstract int getPriceCode();
}

class ChildrensPrice extends Price {
     int getPriceCode() {
          return Movie.CHILDRENS;
     }
}

class NewReleasePrice extends Price {
     int getPriceCode() {
          return Movie.NEW_RELEASE;
     }
}

class RegularPrice extends Price {
     int getPriceCode() {
          return Movie.REGULAR;
     }
}
```

- Refactoring video club
  - Neuvième étape
    - Refactoring "Replace conditional with polymorphism"
      - Déplacer la méthode getCharge de la classe Movie vers la classe Price
      - Tester
      - Faire "descendre" chaque branche du swith dans la sous-classe de Price appropriée redéfinissant getCharge
      - Tester
      - Déclarer abstract la méthode getCharge dans la classe Price
      - Tester
      - Faire de même avec getFrequentRenterPoints

Refactoring video club



Refactoring video club



Refactoring

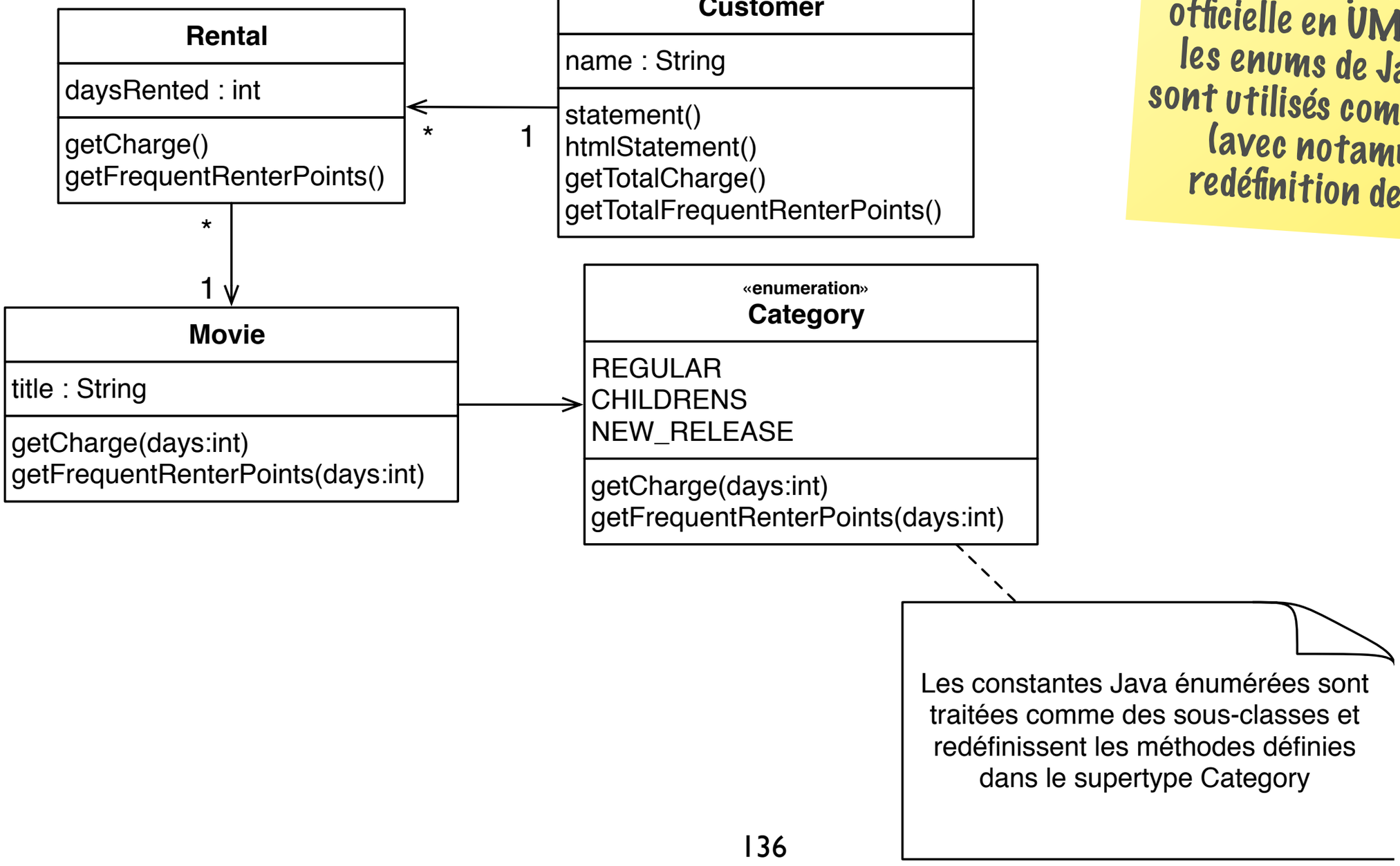
- Refactoring video club
  - Dixième étape
    - l'utilisation d'un code de type entier ne permet une vérification par le compilateur
    - Oblige à générer une exception
    - Solution plus élégante possible à base d'enum
    - Effectuer la modification et valider par les tests unitaires

```
public class Movie {

      public static final int CHILDRENS = 2;
      public static final int REGULAR = 0;
      public static final int NEW_RELEASE = 1;

      public Movie(String title, int priceCode) {
            _title = title;
            setPriceCode(priceCode);
      }

      public void setPriceCode(int priceCode) {
            switch (priceCode) {
            case CHILDRENS:
                  _price = new ChildrensPrice();
                  break;

            case REGULAR:
                  _price = new RegularPrice();
                  break;

            case NEW_RELEASE:
                  _price = new NewReleasePrice();
                  break;

            default:
                  throw new IllegalArgumentException("Incorrect
                                                      price code");
            }
      }
}
```

- Refactoring video club
  - Onzième étape
    - Le nom de l'enum Price paraît mal choisi, du point de vue du modèle il renvoie plutôt à la classification du film qu'à son prix
    - Le film "Nemo" est dans la catégorie "Enfants" plutôt que de prix "Enfants"
    - Ne pas hésiter à utiliser le refactoring pour renommer des classes et diminuer la distance avec le vocabulaire du modèle
    - Renommage de Price en Category

Refactoring video club

Onzième étape

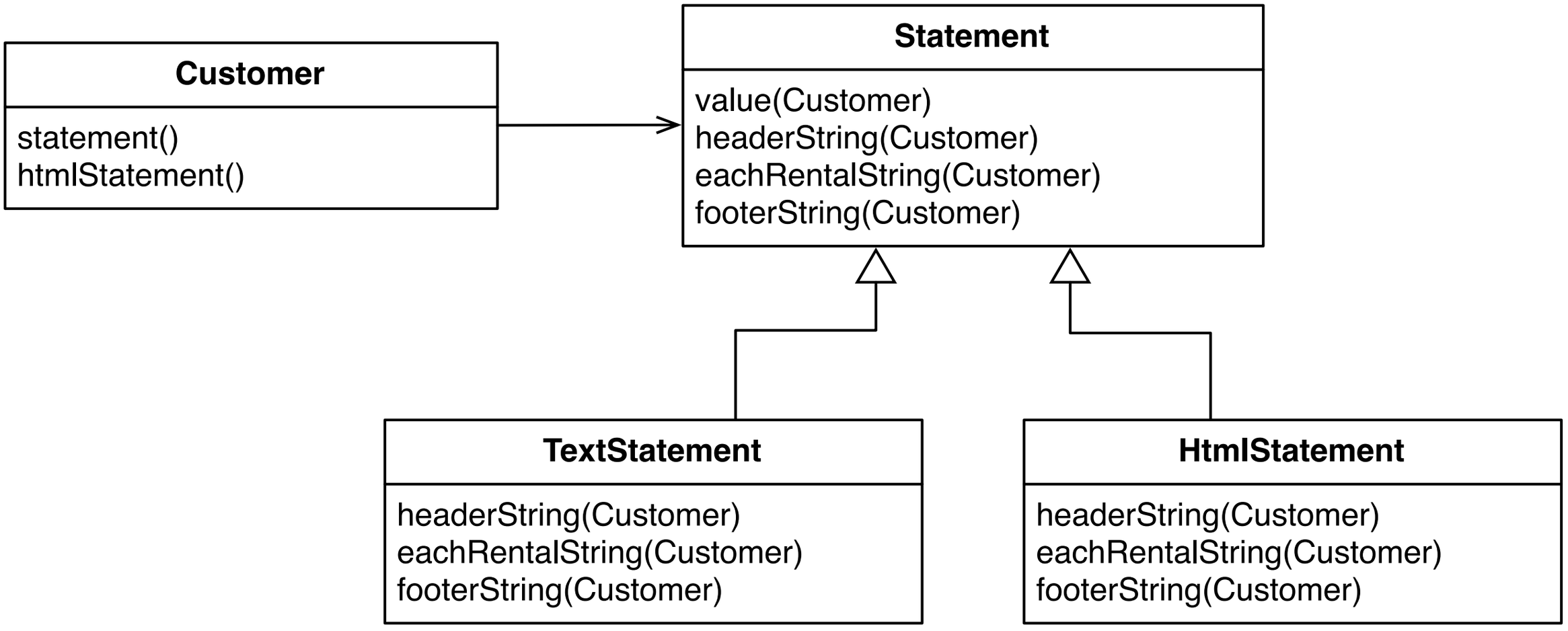


Attention

Il n'existe pas de notation officielle en UML pour décrire les enums de Java quand ils sont utilisés comme des classes (avec notamment de la redéfinition de méthodes)

- Refactoring video club
  - Dernière étape
    - Les méthodes statement et htmlStatement possèdent une structure similaire
    - Elles partagent une même logique avec des étapes similaires mais diffèrent dans la gestion de chacune des étapes
  - Solution
    - Application du design pattern "Template Method" (Patron de conception)
    - La séquence des étapes de la méthode est déplacée dans une super classe et on s'appuie sur le polymorphisme pour s'assurer que chacune des étapes est bien effectuée différemment
  - Permet de limiter la duplication de code

- Refactoring video club
  - Dernière étape



- Refactoring video club
  - Dernière étape
    - Ecrire une méthode tests unitaires pour le reçu au format html
    - Créer les classes Statement, TextStatement et HtmlStatement
    - La stratégie pour s'appuyer un maximum sur eclipse afin de déplacer le contenu des méthodes statement et htmlStatement vers leurs classes respectives est de disposer d'une instance d'une de ces classes
      - Créer une variable d'instance de type TextStatement et instancier un objet
      - Déplacer la méthode statement vers la classe TextStatement
      - Supprimer la variable d'instance et instancier l'objet TextStatement au moment de l'appel
      - Procéder de la même manière pour htmlStatement()

- Refactoring video club
  - Dernière étape
    - Dans les deux classes TextStatement et HtmlStatement
      - Utiliser le refactoring Extract Method pour isoler des méthodes
        - String headerString(Customer c)
        - String eachRentalString(Rental r)
        - String footerString(Customer c)
    - Utiliser le refactoring Pull Up pour déplacer la méthode value dont le corps est maintenant identique dans les deux sous-classes vers la classe Statement
    - Ajouter les méthodes abstract headerString, eachRentalString et footerString dans la classe Statement
    - Tester !

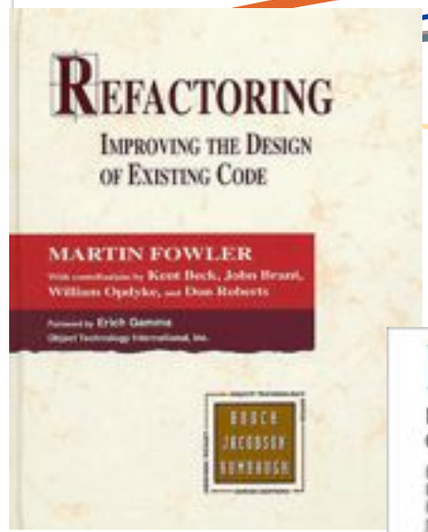
IUT Lyon 1

l'excellence technologique

Bibliographie

LP IEM

Bibliographie



- Effective Java (2nd edition) - J. Bloch
- Refactoring - Improving the design of existing code - M. Fowler
- Design patterns elements of reusable object-oriented software - E. Gamma, R. Helm, R. Johnson, J. Vlissides ("Gang of four")
- UML 2 et les design patterns - C. Larman
- Clean Code - Robert C. Martin (Uncle Bob)