

DVD Miage: Module Programmation Objet

Chapitre 10 - Éléments de Conception à objets

Un exemple illustré en Smalltalk et Java

Pascal André, Gilles Ardourel



1 Introduction

Dans cette fiche nous montrons comment coder un programme en

- Java avec Eclipse
- Smalltalk avec VisualWorks

L'exemple support est le jeu de Nim.

1.1 Le jeu de Nim

Le jeu de Nim se joue entre deux joueurs et avec un tas d'allumettes. Les joueurs enlèvent alternativement 1, 2 ou 3 allumettes. Le perdant est celui qui épuise le tas.

1.2 Organisation

Nous commençons par une modélisation abstraite à la UML, puis nous codons l'application en fonction des caractéristiques de chaque langage.

2 Conception à objets

Nous distinguons la description indépendante des langages de programmations, à la UML, de la conception détaillée qui prend en compte des caractéristiques des langages cibles.

2.1 Conception abstraite à objets

Le diagramme de classes de la figure 1 représente une conception abstraite du programme dans la notation uml [AV01]. Une conception abstraite à objets plus formelle est présentée dans [AR98], qui explicite la spécification des opérations. Quatre classes composent l'application : **Personne**, **Personne**, **Joueur**, **Arbitre** et **JoueurIntelligent**.

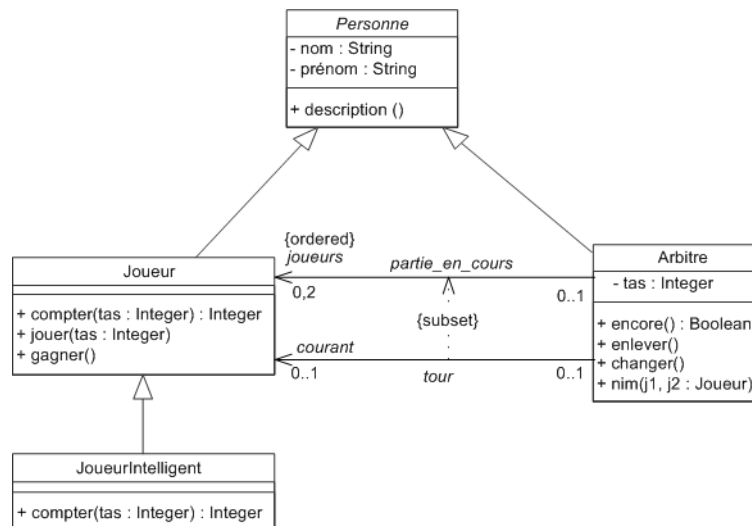


Figure 1 : *Environnement initial*

La classe abstraite¹ **Personne** définit les participants au jeu. Les participants sont caractérisés par deux attributs : le nom et le prénom. Nous avons défini une opération **description()** qui rend une description textuelle de la personne. On distingue les deux joueurs de l'arbitre en utilisant la relation de spécialisation.

La classe **Joueur** définit les opérations (les responsabilités) suivantes :

- **compter** : le joueur détermine le nombre d'allumettes à ôter,
- **jouer** : le joueur joue un coup,
- **gagner** : le joueur se proclame gagnant.

La sous-classe **JoueurIntelligent** décrit des joueurs qui savent "mieux compter". Au lieu de tirer au hasard, le joueur choisi en fonction du nombre restant. L'opération **compter** est redéfinie.

L'arbitre a la maîtrise du jeu, c'est lui qui détient le tas d'allumettes et organise les actions des joueurs. Il définit les opérations (les responsabilités) suivantes :

- **encore** : indique si la partie n'est pas finie,
- **enlever** : modifie le tas d'allumettes après qu'un joueur ait joué un coup,
- **changer** : l'arbitre passe la main à un autre joueur,
- **nim** : l'arbitre démarre une nouvelle partie. Si une partie était en cours, elle est annulée.

La partie n'est possible que lorsque les joueurs sont mis en relation avec l'arbitre. Les deux associations **partie_en_cours** et **tour** symbolisent les liens entre joueurs et l'arbitre. L'association **partie_en_cours** indique les deux joueurs de la partie en cours (ou aucun s'il n'y a pas de partie en cours). La contrainte **{ordered}** signifie qu'ils sont distingués selon un ordre donné. L'association **tour** indique le joueur qui doit jouer un coup. La contrainte **{subset}** met en évidence le fait que le joueur joue un coup dans la partie en cours. Autrement dit, le joueur qui a la main est un des participants du jeu. La navigation est unidirectionnelle pour les deux associations, cela signifie que les joueurs n'ont pas connaissance de l'arbitre ou encore que les communications se font dans le sens arbitre vers joueurs. Ce choix dans la navigation a une conséquence en conception détaillée (ou concrète). Il n'y a pas non plus de communication directe entre deux joueurs.

Noter qu'on respecte le principe d'encapsulation : les attributs sont privés et les opérations sont publiques. Ainsi seul l'arbitre peut modifier le tas d'allumettes (par l'opération **enlever**), les noms et prénoms ne peuvent être changés car il n'y a pas d'opération associées. Les opérations d'accès en lecture et de création d'instance n'ont pas été explicitement modélisées. Les types des attributs sont ceux du langage Object Constraint Language (OCL), qui fait partie de la notation UML.

¹Une classe abstraite n'a pas d'instances. Voir les chapitres 3 et 4.

2.2 Conception détaillée

Ce que nous appelons conception détaillée (ou concrète) est la mise en place de la conception dans l'environnement de développement cible. Il ne s'agit pas encore de coder le programme mais de faire l'adéquation entre les concepts de la conception abstraite et ceux de la conception détaillée. En principe, les concepts d'objets et de classes se retrouvent dans l'environnement cible.

2.2.1 Principes généraux

En supposant, un seul langage de programmation cible², voici les grandes lignes d'une telle transition :

- **Vocabulaire** : on adapte le vocabulaire à celui du langage cible. Par exemple, un attribut (resp. une opération) d'instance sera appelée variable (resp. méthode) d'instance en Java ou donnée (resp. fonction) membre en C++.
- **Typage** : on adapte les règles de typage et les conventions associées à celles du langage cible. Par exemple, en Java, le typage est statique alors qu'en Smalltalk il est dynamique. Le polymorphisme des méthodes est implicite en Java alors que celui des fonctions C++ est explicite par le qualificatif `virtual`.
- **Association** : ce point est détaillé dans la section suivante.
- **Héritage multiple** :
 - Si le langage cible n'autorise pas l'héritage multiple (Java par exemple), il faut mettre en œuvre une politique de traduction adaptée. Par exemple, on choisit un chemin d'héritage principal et on duplique les caractéristiques issues des chemins secondaires. La duplication en Java se fait en déclarant de nouvelles interfaces.
 - Si le langage cible n'autorise pas l'héritage multiple, il faut adapter la stratégie de gestion des conflits à celle du langage cible (renommage et redéfinition en Eiffel, tri topologique en CLOS, implantation multiple d'interfaces en Java).
- **Contrôle des objets** : La plupart des langages à objets manipulent des objets séquentiels passifs. Lorsque la conception définit des objets actifs, il faut mettre en place un environnement d'exécution distribué simulé (processus en Java) ou intégré (*threads* Java). Ce point, qui inclue les variantes d'envois de messages et d'événements, constitue une étape essentielle de la conception pour les systèmes temps-réels.
- **Méta-objets** : certains langages autorisent des protocoles pour méta-objets (Java), pour les autres, il faut simuler en fonction des possibilités du langage (routines en Eiffel).
- **Assertions** : si le langage autorise les assertions (Eiffel par exemple), la traduction de contraintes OCL est simplifiée, sinon il faut programmer explicitement les contraintes.
- **Réutilisation** : une partie de la conception est réécrite en fonction des éléments qui existent dans l'environnement cible. C'est le cas par exemple des collections OCL.

2.2.2 Conception des associations

En programmation à objets, une association s'implante habituellement avec des pointeurs. On peut s'inspirer des règles de transformation du schéma E-A-P dans le modèle relationnel. Examinons les alternatives de modélisation.

1. L'association ne possède pas de propriétés. Les liens sont représentés par des attributs de navigation, un par classes de la relation. Par exemple, *arbitre.courant* donne le joueur qui a la main. Ces attributs prennent en général les noms des rôles. Il y a quelques cas particuliers :
 - (a) Navigation unidirectionnelle : un seul attribut de lien, dans la classe origine de la navigation. C'est le cas des deux associations ici.
 - (b) Cardinalité maximale de 1 dans un sens : on peut choisir une navigation unidirectionnelle.

²Avec CORBA, des objets issus de différents langages peuvent coopérer.

2. L'association possède des propriétés et

- (a) une cardinalité égale à 1. Les propriétés migrent dans la classe correspondant à la cardinalité. Après migration, on se retrouve dans le cas 1.
- (b) aucune cardinalité maximale de 1. L'association donne lieu à une nouvelle classe. Elle est reliée aux classes sous-jacentes par des associations binaires sans propriétés (cas 1).
- (c) une cardinalité dans 0..1. Les deux cas précédents sont applicables. Si les propriétés migrent dans la classe, il se pose le problème des valeurs partiellement définies.

Nous déclinons les principes ci-dessus pour chaque langage.

3 Programmation en Java

3.1 Conception détaillée de l'exemple en Java

Les attributs sont représentés par des attributs en Java. Le type des attributs est noté dans le commentaire de la classe. Les types `Integer`, `String` et `Boolean` existent tels quels en Java, mais on peut utiliser `int`, `StringBuffer` et `boolean` selon le besoin ou l'habitude. L'héritage ne pose pas de problèmes dans cet exemple puisqu'il n'y a pas de cas d'héritage multiple.

Dans la classe `Personne`, on définit deux variables d'instance : `nom` et `prenom`. Le commentaire de la classe indique que leur type est `String`. Il n'y a pas de définition de variables d'instance dans les classes `Joueur` et `JoueurIntelligent` car les variables sont implicitement héritées en Java.

Dans la classe `Arbitre`, la situation est plus compliquée pour les raisons suivantes :

1. Le tas d'allumettes est représenté par un entier qui donne le nombre d'allumettes dans le tas. C'est une variable d'instance "conditionnelle". Le nombre d'allumettes est fixé aléatoirement au départ du jeu et non à la création de l'arbitre. Le nombre initial d'allumettes varie entre 10 et `NbMaxAllu` où la constante `NbMaxAllu` sera représenté par une variable de classe.
2. Les associations sont représentées sous forme de variables d'instances. Le nom de la variable est le rôle associé s'il existe. Il faut modéliser les contraintes portant sur les associations.
 - (a) La contrainte d'ordre se modélise en choisissant une collection ordonnée pour ranger les joueurs. La taille de cette collection est au maximum de deux, plus précisément elle contient deux ou aucun objet de la classe `Joueur`. **Toute mise à jour de cette variable devra vérifier cette condition.**
 - (b) Le joueur qui a la main est représenté par une variable d'instance. En principe, nous devrions utiliser un objet de la classe `Joueur`. Attention, ici aussi, il peut ne pas y avoir de joueur courant (cardinalité 0..1).

Ces deux variables sont aussi une variable d'instance "conditionnelle" dont l'existence est conditionnée au fait qu'une partie est en cours.

Pour prendre en compte les variables conditionnelles, nous ajoutons une nouvelle variable d'instance booléenne `partieEnCours` dont la valeur conditionnera les autres variables d'instance. Ainsi tout accès en lecture ou écriture des variables conditionnées implique de vérifier leur existence. En cas d'échec, une erreur est levée.

Cette traduction "systématique" peut être améliorée. En effet, nous n'avons pas représenté la contrainte d'inclusion du joueur courant dans les joueurs de la partie en cours. Nous n'avons pas non plus représenté le mécanisme de changement de joueur. La structure la plus pratique en termes de manipulation est en fait le tableau. Le joueur courant sera représenté par un indice dans le tableau à deux dimensions. Sachant que les tableaux ont des indices de 1 à n en Java, le changement de joueur s'écrit arithmétiquement pas `courant` :
`:= ((courant mod 2)+1`.

Les opérations sont traduites en méthodes. Le nommage des méthodes en Java est similaire à celui des opérations en UML. Par exemple, la méthode associée à l'opération `compter(tas : Integer) : Integer` est `public int compter (int tas)`. Les méthodes d'instances d'accès en lecture sont générées de manière automatique avec Eclipse.

La méthode de description par défaut des objets Java est `toString()`. Nous remplaçons donc la méthode `description()` de la classe `Personne` par cette méthode.

La méthode d'instance `compter()` de la classe `Joueur` est redéfinie avec le même type dans la classe `JoueurIntelligent` et masque de ce fait la définition héritée de la classe `Joueur`.

Les méthodes de classes, implicites dans la modélisation abstraite, sont définies, en tenant compte les variables d'instances des classes correspondantes et des contraintes du schéma. La classe `Personne` est abstraite et n'a donc pas de méthode d'instanciation. Dans la classe `Joueur`, la méthode d'instanciation sera le constructeur `Joueur (String nom, String prenom)`. Dans la classe `Arbitre`, la définition tient compte de la représentation choisie : il n'y a pas d'arguments car au départ il n'y a pas de partie en cours. Par contre, le tableau des joueurs est créé avec deux positions et la variable `partieEnCours` est initialisée à `false`. C'est la méthode d'instance `nim`, qui initialise les autres variables pour la partie qui démarre.

3.2 Codage

Cette section illustre étape par étape l'écriture du code dans l'environnement Eclipse.

3.2.1 Création d'un projet et/ou paquetage

Ouvrir l'application Eclipse. Créer un nouveau projet `Nim` ou ouvrir un projet existant par le menu `File > New...` ou par un menu contextuel (ou dynamique) dans la vue des paquetages. Puis créer un nouveau paquetage `nim` dans un projet existant par le menu `File > New > Package` ou par un menu contextuel dans la vue des paquetages. Nous conseillons de ne pas utiliser le paquetage par défaut pour mieux maîtriser la structure du code.

Le paquetage joue le rôle de classement modulaire (la catégorie en Smalltalk).

3.2.2 Création d'une classe

Créer une nouvelle classe `Personne` dans le paquetage `nim` par le menu `File > New > Class` ou par un menu contextuel dans la vue des paquetages. Selon les options choisies, certaines déclarations sont générées automatiquement ou pas.

Tout ce travail peut être réalisé par un gestionnaire de fichier hôte et un éditeur de texte simple. L'importation se fait soit en déplaçant les répertoires sous l'arborescence d'un projet existant dans un répertoire `workspace` d'Eclipse soit importé sous Eclipse par le menu contextuel `Import...` sur un projet ou un paquetage. On obtient :

```
package nim;
public class Personne {
    public Personne() {
        super();
        // TODO Auto-generated constructor stub
    }
}
```

Insérer au clavier la définition des variables d'instance `nom` et `prenom`.

3.2.3 Création d'une classe abstraite

Rajouter le mot-clé `abstract` pour indiquer que la classe `Personne` est abstraite. Modifier le contenu de son constructeur pour invalider l'instanciation.

```
public abstract class Personne {
    String nom, prenom;
}
```

3.2.4 Création d'une méthode de classe

Les méthodes de classe indispensables sont au moins les constructeurs. En l'absence de constructeur (pour une classe normale en tout cas), un constructeur par défaut est défini. Comme la classe est abstraite, nous pourrions décider, dans un but pédagogique, d'invalider l'instanciation, en modifiant le constructeur de base.

```
public abstract class Personne {
    String nom, prenom;
    public Personne() {
        // méthode abstraite
    }
}
```

3.2.5 Création des méthodes d'instance

Nous allons maintenant créer les méthodes d'accès en lecture et écriture des variables d'instance et la méthode de description.

Générer automatiquement les méthodes d'accès publique en lecture et protégée (pour les sous-classes) en écriture par le menu contextuel **Source > Generate Getters and Setters...**

Pour la méthode de description, on redéfinit la méthode `toString`. Ajouter au clavier la méthode de transformation en texte `toString`. Pour aller plus vite, copier ce code depuis une autre classe.

3.2.6 Classement des méthodes

Java ne dispose pas comme Smalltalk d'un outil de classement des méthodes (le protocole). Le développeur doit lui-même ordonner ses méthodes. Il peut s'appuyer sur des commentaires pour séparer les éléments de code et utiliser des noms de protocoles comme dans Smalltalk.

3.2.7 Création des commentaires Javadoc

Générer automatiquement les commentaires pour la classe, les variables d'instance et les méthodes par le menu contextuel **Source > Add Comment** ou le raccourci clavier.

Indenter automatiquement le code par le menu contextuel **Source > Generate Format**.

Listing 1: Code Java de la classe `Personne`

```
package nim;
/**
 * @author pascal andre
 * @date Janvier 2006
 * @version 1 Classe abstraite définissant des personnes pour le jeu de Nim.<br>
 *      Pas de constructeur par défaut pour une classe abstraite .
 */
public abstract class Personne {
    /**
     * Une personne est définie simplement par un nom et un prénom.
     */
    protected String nom, prenom;
    /**
     * @return nom de la personne : String
     */
    public String getNom() {
        return nom;
    }
    /**
     * @param nom
     *      de la personne : String
     */
    protected void setNom(String nom) {
        this.nom = nom;
    }
    /**
     * @return prénom de la personne : String
     */
    public String getPrenom() {
        return prenom;
    }
    /**
     * @param prénom
     *      de la personne : String
     */
    protected void setPrenom(String prenom) {
        this.prenom = prenom;
    }
    /**
     * (non-Javadoc) méthode d'affichage par défaut
     *
     * @see java.lang.Object#toString()
     */
    public String toString() {
        // redéfinition de la méthode ((Object)this).toString()+
        return ("Je m'appelle " + nom + " " + prenom);
    }
}
```

```
}  
}
```

3.2.8 Sauvegarde des travaux

La sauvegarde se fait en enregistrant les fichiers java.

3.2.9 Héritage par extension

Procéder de même pour écrire les autres classes de l'application (**Joueur**, **Arbitre** et **JoueurIntelligent**) selon le code suivant. Ces classes sont définies par héritage d'extension sur la classe **Personne** dans le même paquetage. En Java, une classe ne peut être créée que si sa superclasse existe. Ce n'est pas le cas sans IDE car les classes sont dans des fichiers séparés.

La classe **Joueur** ajoute uniquement de nouvelles méthodes à la classe **Personne**. La méthode **compter** calcule combien d'allumettes il doit enlever. Lorsque le nombre d'allumettes est supérieur à deux, il choisi aléatoirement. Deux possibilités se présentent en Java : définir une suite aléatoire avec la classe `java.util.Random` ou plus simplement demander un nombre aléatoire entre 0 et 1 à la classe `java.lang.Math`. Nous choisissons cette dernière car c'est une demande ponctuelle.

La classe **JoueurIntelligent** est créée après la classe **Joueur**. Les méthodes correspondant au jeu sont placées dans un protocole **jeu**. Pour l'arbitre, on considère que les variables d'instances sont privées (pas de méthodes d'accès). L'initialisation de la variable de classe **NbMaxAllu** (nombre maximal d'allumettes dans le tas) est réalisée par une valeur par défaut (et non une méthode de classe **initialize** comme en Smalltalk - on voit ici la différence avec un vrai protocole de réflexion.). Les méthodes annexes de la méthode **nim** sont définies en **protected** pour être accessibles dans d'éventuelles sous-classes.

3.2.10 Evaluer un envoi de message

Pour évaluer du code, le plus simple est de définir une méthode **main** qui évalue des expressions. Il n'y a pas d'outil de type **Workspace** comme en Smalltalk pour évaluer n'importe quelle expression car l'environnement de travail est vide par défaut en Java.

Voici le code de ces différentes classes.

Listing 2: Code Java de la classe Joueur

```
package nim;  
/**  
 * @author pascal andre  
 * @date Janvier 2006  
 * @version 1 Classe définissant des joueurs pour le jeu de Nim.<br>  
 */  
public class Joueur extends Personne {  
    /**  
     * Méthode d'instanciation : constructeur  
     */  
    public Joueur(String nom, String prenom) {  
        super();  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
    /**  
     * Le joueur détermine combien d'allumettes il enlève du tas. Le joueur  
     * décide aléatoirement ce nombre s'il y a au moins trois allumettes dans le  
     * tas. Il retire de 1 à 3 allumettes.  
     * @param tas :  
     *         nombre d'allumettes du tas  
     * @return nombre d'allumettes à enlever  
     */  
    public int compter(int tas) {  
        int combien;  
        // variable locale qui détermine le nombre d'allumettes à tirer  
        if (tas == 1 || tas == 2) {  
            combien = 1;  
        } else {  
            // on peut aussi utiliser le temps Time  
            combien = (((int) (Math.random() * 1000)) % 3) + 1;  
        }  
    }  
}
```

```

        return combien;
    }

    public int jouer(int tas) {
        int combien = this.compter(tas);
        // variable locale qui détermine le nombre d'allumettes à tirer
        System.out.print("Il y a " + tas + " allumette(s)");
        System.out.println("    moi, " + this.toString() + " j'enlève "
            + combien + " allumette(s)");
        return combien;
    }

    /**
     * Le joueur se proclame gagnant.
     */
    public void gagner() {
        System.out.println("→ Moi, " + this.nom + " " + this.prenom
            + " j'ai gagné");
    }
}

```

Listing 3: Code Java de la classe Arbitre

```

package nim;
import java.util.ArrayList;
/**
 * @author pascal andre
 * @date Janvier 2006
 * @version 1
 * Classe définissant l'arbitre du jeu de Nim.<br>
 * Certaines méthodes peuvent générer des erreurs.
 */
public class Arbitre extends Personne {
    /**
     * variable de classe définissant le nombre maximum d'allumettes
     * dans le tas
     */
    public static int NbMaxAllu = 30;
    /**
     * variables d'instance privées
     * les variables sont conditionnées par le fait qu'une partie est en cours.
     * tas : le nombre courant d'allumettes dans le tas
     * compris entre 0 et NbMaxAllu
     * partie_en_cours : liste de joueurs
     * de taille 2
     * courant : numéro du joueur courant
     * compris entre 0 et 1
     * ces trois variables sont valides si une partie est en cours
     * nous le gérons par une variable booléenne
     * partieEnCours : vrai si une partie est en cours
     */
    private boolean partieEnCours;
    private int tas;
    private ArrayList<Joueur> partie_en_cours;
    private int courant;

    /**
     * @param nom
     * @param prenom
     * au départ, in n'y a pas de partie en cours
     */
    public Arbitre(String nom, String prenom) {
        super();
        this.nom = nom;
        this.prenom = prenom;
        partieEnCours = false;
        // ce qui suit n'a pas de pertinence
        // car il n'y a pas de partie en cours
        tas = 0;
        courant = 0;
        partie_en_cours = new ArrayList<Joueur>();
    }
}

```



```

/**
 * @param j1 un joueur
 * @param j2 un autre joueur
 * Cette méthode démarre une partie de Nim.<br>
 * Le tas est défini aléatoirement.
 * Le jeu est une itération qui se termine quand il
 * n'y a plus d'allumettes.
 */
public void nim(Joueur j1, Joueur j2) {
    partieEnCours = true;
    partie_en_cours = new ArrayList<Joueur>();
    partie_en_cours.add(j1);
    partie_en_cours.add(j2);
    tas = 10 + (((int) (Math.random() * 1000)) % (NbMaxAllu - 9));
    courant = 0; // le choix pourrait être aléatoire
    System.out.println("La partie débute avec " + tas +
        " allumettes \n et " + partie_en_cours.toString());
    while (this.encore()) {
        this.enlever();
        this.changer();
    }
    partie_en_cours.get(courant).gagner();
}

// méthodes utilisées pour implanter le jeu
/**
 * @return true si une partie se continue
 */
protected boolean encore () {
    if (partieEnCours) return (tas > 0);
    else throw new Error("Pas de partie en cours");
}

/**
 * un joueur prend des allumettes
 */
protected void enlever () {
    if (partieEnCours) {
        tas = tas - (partie_en_cours.get(courant).jouer(tas));
    }
    else throw new Error("Pas de partie en cours");
}

/**
 * on change de joueur
 */
protected void changer () {
    if (partieEnCours) courant = (courant + 1) % 2;
    else throw new Error("Pas de partie en cours");
}

// méthodes pour tester le jeu
/**
 * @param args
 * Crée une partie avec un arbitre et deux joueurs.
 * Lance le jeu et affiche les résultats.
 */
public static void main(String args[]) {
    /* programme principal */
    Arbitre arbitre;
    Joueur j1, j2, j3;
    arbitre = new Arbitre("Sémoi", "Lechef");
    j1 = new Joueur("Alain", "Térier");
    j2 = new Joueur("Alex", "Térier");
    j3 = new JoueurIntelligent("Alex", "LeFort");
    arbitre.nim(j1,j2);
    arbitre.nim(j1,j3);
}
}

```

Listing 4: Code Java de la classe JoueurIntelligent

package nim;

```

/**
 * @author pascal andre
 * @date Janvier 2006
 * @version 1
 * Classe définissant des joueurs intelligents pour le jeu de Nim.<br>
 * A la base ce sont des joueurs.
 */
public class JoueurIntelligent extends Joueur {
    /**
     * @param nom
     * @param prenom
     */
    public JoueurIntelligent(String nom, String prenom) {
        super(nom, prenom);
        // TODO Auto-generated constructor stub
    }
    /**
     * @param tas : nombre d'allumettes du tas
     * @return nombre d'allumettes à enlever
     * Le joueur intelligent détermine combien d'allumettes il enlève du tas.
     * Le joueur décide directement ce nombre quel que soit le nombre d'allumettes dans le tas.
     * Il retire de 1 à 3 allumettes.
     */
    public int compter (int tas) {
        int combien = tas % 4;
        // variable locale qui détermine le nombre d'allumettes à tirer
        switch (tas) {
            case 0:
                combien = 3;
                break;
            case 1:
            case 2:
                combien = 1;
                break;
            default:
                combien = 2;
                break;
        }
        return combien;
    }
}

```

3.2.11 Compiler et exécuter des travaux

La compilation du code se fait en ligne par la commande `javac`. Sous Eclipse, on utilise un menu **Project>Build...** ou un script dans la fenêtre **Ant**. La compilation produit des fichiers `.java` archivable par l'utilitaire `jar` pour en faire des bibliothèques utilisables (sans les sources) dans d'autres programmes.

Pour exécuter le code, le plus simple est d'activer le menu **Run As > Java Application** ou le menu contextuel **Run As > Java Application** dans la fenêtre d'édition.

L'évaluation du message `arbitre.nim(j1,j2)` du programme `Arbitre.main()` affiche le résultat suivant dans la fenêtre Eclipse de la console :

```

La partie débute avec 30 allumettes
et [Je m'appelle Alain Térieur, Je m'appelle Alex Térieur]
Il y a 30 allumette(s) moi, Je m'appelle Alain Térieur j'enlève 2 allumette(s)
Il y a 28 allumette(s) moi, Je m'appelle Alex Térieur j'enlève 2 allumette(s)
Il y a 26 allumette(s) moi, Je m'appelle Alain Térieur j'enlève 2 allumette(s)
Il y a 24 allumette(s) moi, Je m'appelle Alex Térieur j'enlève 1 allumette(s)
Il y a 23 allumette(s) moi, Je m'appelle Alain Térieur j'enlève 2 allumette(s)
Il y a 21 allumette(s) moi, Je m'appelle Alex Térieur j'enlève 2 allumette(s)
Il y a 19 allumette(s) moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 18 allumette(s) moi, Je m'appelle Alex Térieur j'enlève 3 allumette(s)
Il y a 15 allumette(s) moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 14 allumette(s) moi, Je m'appelle Alex Térieur j'enlève 1 allumette(s)
Il y a 13 allumette(s) moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 12 allumette(s) moi, Je m'appelle Alex Térieur j'enlève 3 allumette(s)

```

```

Il y a 9 allumette(s)    moi, Je m'appelle Alain T rieur j'enl ve 3 allumette(s)
Il y a 6 allumette(s)    moi, Je m'appelle Alex T rieur j'enl ve 3 allumette(s)
Il y a 3 allumette(s)    moi, Je m'appelle Alain T rieur j'enl ve 3 allumette(s)
-> Moi, Alex T rieur j'ai gagn 

```

L' valuation du message `arbitre.nim(j1,j3)` affiche le r sultat suivant :

```

La partie d bute avec 22 allumettes
et [Je m'appelle Alain T rieur, Je m'appelle Alex LeFort]
Il y a 22 allumette(s)    moi, Je m'appelle Alain T rieur j'enl ve 1 allumette(s)
Il y a 21 allumette(s)    moi, Je m'appelle Alex LeFort j'enl ve 2 allumette(s)
Il y a 19 allumette(s)    moi, Je m'appelle Alain T rieur j'enl ve 3 allumette(s)
Il y a 16 allumette(s)    moi, Je m'appelle Alex LeFort j'enl ve 2 allumette(s)
Il y a 14 allumette(s)    moi, Je m'appelle Alain T rieur j'enl ve 2 allumette(s)
Il y a 12 allumette(s)    moi, Je m'appelle Alex LeFort j'enl ve 2 allumette(s)
Il y a 10 allumette(s)    moi, Je m'appelle Alain T rieur j'enl ve 1 allumette(s)
Il y a 9 allumette(s)     moi, Je m'appelle Alex LeFort j'enl ve 2 allumette(s)
Il y a 7 allumette(s)     moi, Je m'appelle Alain T rieur j'enl ve 1 allumette(s)
Il y a 6 allumette(s)     moi, Je m'appelle Alex LeFort j'enl ve 2 allumette(s)
Il y a 4 allumette(s)     moi, Je m'appelle Alain T rieur j'enl ve 1 allumette(s)
Il y a 3 allumette(s)     moi, Je m'appelle Alex LeFort j'enl ve 2 allumette(s)
Il y a 1 allumette(s)     moi, Je m'appelle Alain T rieur j'enl ve 1 allumette(s)
-> Moi, Alex LeFort j'ai gagn 

```

3.3 Conclusion

Dans cette section, nous avons mis en  uvre un programme Java   partir d'une description abstraite pour illustrer l'utilisation des outils de Java pour le codage. Nous n'avons pas pr sent  les outils de mise au point (inspecteurs et d bogueurs) bien que nous les ayons utilis  pour tester le programme.

Cet exemple montre qu'il est rapide de programmer en Java, c'est pourquoi le langage est utilis  pour le prototypage rapide. Il montre aussi que la connaissance des classes du syst me influence fortement le r sultat.

4 Programmation en Smalltalk

4.1 Conception d taill e de l'exemple en Smalltalk

Les attributs sont repr sent s par des variables d'instances. Le type des attributs est not  dans le commentaire de la classe. Les types `Integer`, `String` et `Boolean` existent tels quels en Smalltalk. L'h ritage ne pose pas de probl mes puisqu'il n'y a pas de cas d'h ritage multiple.

Dans la classe `Personne`, on d finit deux variables d'instance : `nom` et `prenom`. Le commentaire de la classe indique que leur type est `String`. Il n'y a pas de d finition de variables d'instance dans les classes `Joueur` et `JoueurIntelligent` car les variables sont implicitement h rit es en Smalltalk.

Dans la classe `Arbitre`, la situation est plus compliqu e pour les raisons suivantes :

1. Le tas d'allumettes est repr sent  par un entier qui donne le nombre d'allumettes dans le tas. C'est une variable d'instance "conditionnelle". Le nombre d'allumettes est fix  al atoirement au d part du jeu et non   la cr ation de l'arbitre. Le nombre initial d'allumettes varie entre 10 et `NbMaxAllu` o  la constante `NbMaxAllu` sera repr sent  par une variable de classe.
2. Les associations sont repr sent es sous forme de variables d'instances. Le nom de la variable est le r le associ  s'il existe. Il faut mod liser les contraintes portant sur les associations.
 - (a) La contrainte d'ordre se mod lise en choisissant une collection ordonn e pour ranger les joueurs. La taille de cette collection est au maximum de deux, plus pr cis ment elle contient deux ou aucun objet de la classe `Joueur`. **Toute mise   jour de cette variable devra v rifier cette condition.**
 - (b) Le joueur qui a la main est repr sent  par une variable d'instance. En principe, nous devrions utiliser un objet de la classe `Joueur`. Attention, ici aussi, il peut ne pas y avoir de joueur courant (cardinalit  0..1).

Ces deux variables sont aussi une variable d'instance "conditionnelle" dont l'existence est conditionnée au fait qu'une partie est en cours.

Pour prendre en compte les variables conditionnelles, nous ajoutons une nouvelle variable d'instance booléenne `partieEnCours` dont la valeur conditionnera les autres variables d'instance. Ainsi tout accès en lecture ou écriture des variables conditionnées implique de vérifier leur existence. En cas d'échec, une erreur est levée.

Cette traduction "systématique" peut être améliorée. En effet, nous n'avons pas représenté la contrainte d'inclusion du joueur courant dans les joueurs de la partie en cours. Nous n'avons pas non plus représenté le mécanisme de changement de joueur. La structure la plus pratique en termes de manipulation est en fait le tableau. Le joueur courant sera représenté par un indice dans le tableau à deux dimensions. Sachant que les tableaux ont des indices de 1 à n en Smalltalk, le changement de joueur s'écrit arithmétiquement pas `courant := ((courant mod 2)+1)`.

Les opérations sont traduites en méthodes. Le nommage des méthodes varie par rapport à celui des opérations. Nous utilisons la convention suivante :

1. Opération sans arguments `op():TypeRes` : le sélecteur de la méthode associée à `op` est `op`.
2. Opération avec arguments `op(arg1 : T1, ..., argn : Tn) : TypeRes` : le sélecteur de la méthode associée à `op` est `op + maj1(arg1) + ':' + article(T1) ... maj1(argn) + ':' + article(Tn)`, où `maj1` est une fonction qui met en majuscule la première lettre de son argument et `article` est une fonction qui préfixe par 'a' ou 'an' son argument selon qu'il s'agit d'une consonne ou d'une voyelle. Par exemple, la méthode associée à l'opération `compter(tas : Integer) : Integer` est `compterTas:anInteger`.

Les méthodes d'instances d'accès en lecture sont générées de manière automatique.

La méthode de description par défaut des objets Smalltalk est `printOn: aStream()`. Nous remplaçons donc la méthode `description()` de la classe `Personne` par cette méthode.


La méthode d'instance `compter()` est redéfinie dans la classe `JoueurIntelligent` et masque de ce fait la définition héritée de la classe `Joueur`.

Les méthodes de classes, implicite dans la modélisation abstraite, sont définies, qui tient compte les variables d'instances des classes correspondantes et des contraintes du schéma. La classe `Personne` est abstraite et n'a donc pas de méthode d'instanciation. Dans la classe `Joueur`, la méthode d'instanciation sera `newNom: aString prenom: aString`. Dans la classe `Arbitre`, la définition tient compte de la représentation choisie : il n'y a pas d'arguments car au départ il n'y a pas de partie en cours. Par contre, le tableau des joueurs est créé avec deux positions et la variable `partieEnCours` est initialisée à `false`. C'est la méthode d'instance `nim`, qui initialise les autres variables pour la partie qui démarre.

4.2 Codage

Cette section illustre étape par étape l'écriture du code dans l'environnement `VisualWorks`.

4.2.1 Création d'une classe

Ouvrir un flâneur système depuis la fenêtre `Launcher` par le menu `Browse>All Classes` ou le bouton .

Ajouter une nouvelle catégorie par le menu central `add...`. Une fenêtre de dialogue s'ouvre. Taper 'Nim' dans cette fenêtre et valider. Si aucune catégorie n'était sélectionnée, la nouvelle catégorie se place à la fin de la liste, sinon elle se place juste au dessus de la catégorie sélectionnée. En principe on range les catégories utilisateur vers la fin de la liste pour laisser les catégories de base de Smalltalk en début de liste.

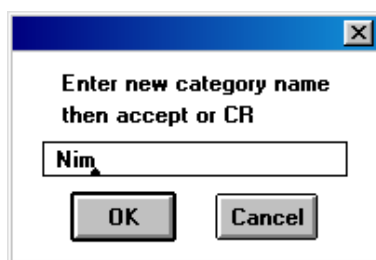


Figure 2 : *Création de la catégorie, jeu de Nim*

Sélectionner si nécessaire la catégorie **Nim**. La liste des classes de cette catégorie est vide. Dans la fenêtre de texte remplacer le prototype de création de classe

```
NameOfSuperclass subclass: #NameOfClass
instanceVariableNames: 'instVarName1 instVarName2'
classVariableNames: 'ClassVarName1 ClassVarName2'
poolDictionaries: ''
category: 'Jeux'
```

par

```
Object subclass: #Personne
instanceVariableNames: 'nom prenom'
classVariableNames: ''
poolDictionaries: ''
category: 'Nim'
```

Valider par la fonction **accept** du menu central. La liste des classes de cette catégorie contient maintenant une nouvelle classe **Personne**.

Dans la liste des classes, activer la fonction **comment** du menu central. Dans la fenêtre de texte remplacer le prototype de création de commentaire par

Cette classe implante la classe abstraite **Personne** du jeu de Nim.

Les variables d'instance sont :

```
nom : String
prenom : String
```

La méthode de classe **new** aurait du être redéfinie pour interdire la création d'instances. Cependant pour pouvoir appeler la méthode d'instanciation **new** de la classe **Object**, sans passer par un artifice supplémentaire, nous n'avons pas redéfini la méthode **new**. Par défaut, la méthode **new** est donc celle de la classe **Object**.

Valider par la fonction **accept** du menu central.

4.2.2 Création d'une méthode d'instance

Nous allons maintenant créer les méthodes d'accès en lecture et écriture des variables d'instance et la méthode de description.

Dans la liste des protocoles, activer la fonction **add...** du menu central. Une fenêtre de dialogue s'ouvre. Taper 'accessing' dans cette fenêtre et valider. Un nouveau protocole est créé.

Astuce : le nom affiché dans la fenêtre est celui du dernier protocole utilisé. Pour éviter de taper ce nom, passer dans une autre classe, sélectionner le protocole **accessing**, puis revenir dans la classe **Personne**, activer la fonction **add...** du menu central de la fenêtre des protocoles puis valider.

Dans la fenêtre de texte remplacer le prototype de création de méthode

```
message selector and argument names
"comment stating purpose of message"

| temporary variable names |
statements
```

par

```
nom
"rend le nom de la personne"

↑nom
```

Formater par la fonction **format** du menu central et valider par la fonction **accept** du menu central. Le formatage et l'acceptation effectuent des contrôles sur l'existence des variables et des méthodes employées. Modifier ensuite le texte précédent par

```
nom: aString
"affecte le nom de la personne"

nom ← aString
```

La valeur rendue par défaut est la variable `self` (`^self`). Formater par la fonction `format` du menu central et valider par la fonction `accept` du menu central. Faire de même pour les méthodes `prenom`, `prenom:`.

Pour la méthode de description, on redéfinit la méthode `printOn:`. Pour gagner du temps, sélectionner cette méthode dans le protocole `printing` de la classe `Object` (ou d'une autre classe). Puis copier le contenu de la méthode dans la fenêtre de texte par la fonction `copy` du menu central. Revenir dans la classe `Personne`, activer la fonction `add...` du menu central de la fenêtre des protocoles puis valider. Dans la fenêtre de texte, remplacer le prototype par le texte copié fonction `paste` du menu central et compléter le texte de la méthode par :

```
printOn: aStream
    "Ecrit le nom et le prénom dans le flot."

    aStream cr ; nextPutAll: 'Je m''appelle ', nom, ' ', prenom
```

Formater valider. Les doubles quotes servent à placer une quote. La méthode `cr` place un retour chariot dans le flot de sortie. La méthode `nextPutAll:` place son argument (une collection) dans le flot de sortie. Dans notre exemple, la collection est une chaîne de caractères, obtenue par concaténation (opérateur `,`) d'autres chaînes de caractères.

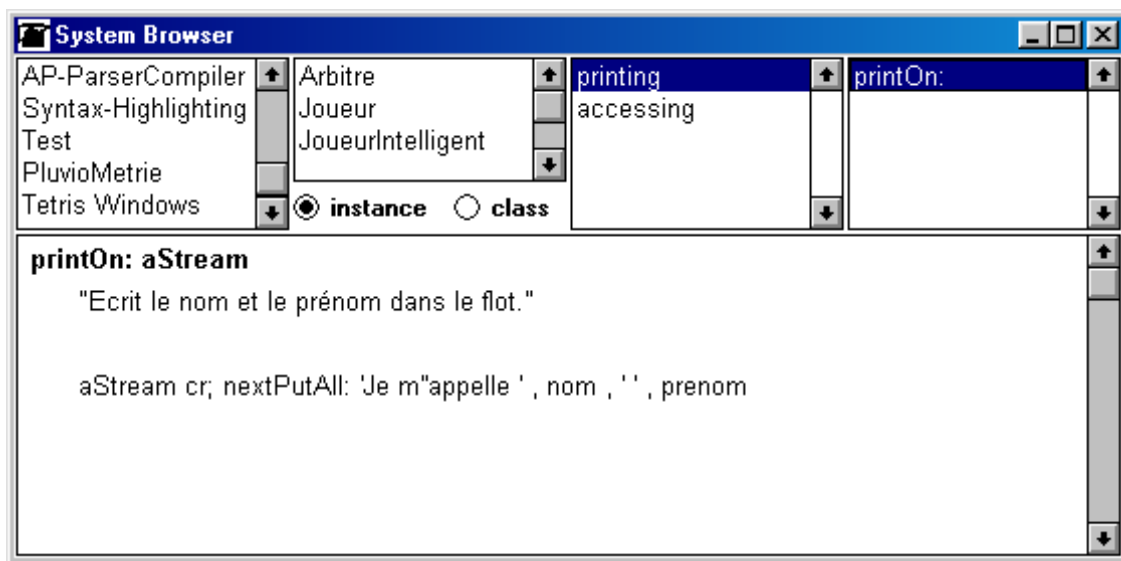


Figure 3 : Une fenêtre flâneur système

4.2.3 Création d'une méthode de classe

Bien que la classe soit abstraite, nous décidons, dans un but pédagogique, de créer une personne par une méthode de classe. Dans le flâneur système, sélectionner le protocole de classe (par le bouton radio `class`). Créer un nouveau protocole (de classe) appelé `examples`. Créer une nouvelle méthode `albert` avec le code suivant :


albert

"Cette méthode crée une personne et affiche ses caractéristiques sur la console. Il s'agit d'un exemple jouet, car en principe la classe Personne est abstraite"

```
Transcript show: ((Personne new) nom: 'Albert'; prenom: 'Michel';
                  printString)
```

La méthode `printString` fait appel à la méthode `printOn:` en redirigeant le flot `aStream` vers une chaîne de caractères.

4.2.4 Evaluer un envoi de message

Ouvrir une fenêtre de travail depuis la fenêtre `Launcher` par le menu `Tools>Workspace` ou le bouton . Puis taper l'envoi de message `Personne albert` dans la zone de texte. Sélectionner ce texte et évaluer la

fonction `do it` du menu central. Le texte `Je m'appelle Albert Michel` apparaît dans la console de la fenêtre `Launcher`.

4.2.5 Sauvegarde des travaux

La sauvegarde du code se fait soit en sauvegardant l'environnement (l'image) soit en générant une version textuelle du code écrit.

Pour enregistrer l'image, activer la fonction `File>Save As...` de la fenêtre `Launcher`.

Pour enregistrer le code dans un fichier texte, activer la fonction `file out as...` du menu central d'une des fenêtres de listes dans un flâneur système. Par exemple, sélectionner la catégorie `Nim`, activer la fonction `file out as...` pour créer un fichier comprenant l'ensemble du code des classes de cette catégorie.

4.2.6 Autres classes

Les classes `Joueur`, `Arbitre` et `JoueurIntelligent` sont définies dans la même catégorie dans un flâneur de classes, en respectant les relations d'héritage. En `Smalltalk`, une classe ne peut être créée que si sa superclasse existe. Ce n'est pas le cas en `C++` car les classes sont dans des fichiers séparés. Les classes `Joueur` et `Arbitre` sont créées après la classe `Personne`. La classe `JoueurIntelligent` est créée après la classe `Joueur`. Les méthodes correspondant au jeu sont placées dans un protocole `jeu`. Les méthodes d'instanciation sont placées dans le protocole de classe `instance creation`.

Pour l'arbitre, on considère que les variables d'instances sont privées (pas de méthodes d'accès). L'initialisation de la variable de classe `NbMaxAllu` (nombre maximal d'allumettes dans le tas) est réalisée par une méthode `initialize` placées dans le protocole de classe `initialize release`.

```
initialize
  "Arbitre initialize "

NbMaxAllu ← 30
```

Attention Pour que l'initialisation soit effective, il faut évaluer le message `Arbitre initialize`. Par convention, on place ce message dans un commentaire de la méthode et on l'évalue par la fonction `do it` du menu central. Noter que lors d'une inclusion d'une classe dans l'image depuis un fichier texte la méthode `initialize`, lorsqu'elle est définie, est exécutée.

Voici le code de ces différentes classes.

```
Personne subclass: #Joueur
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Nim'

!Joueur methodsFor: 'jeu'

compterTas: anInteger
  "le joueur détermine combien il enlève d'allumettes du tas"
  "Le calcul aléatoire est possible par la classe Random (résultat dans 0..1)
  ou l' utilisation d'une horloge Time now réduite dans l'intervalle 1..3"

  | combien |
  anInteger = 1 | (anInteger = 2)
    ifTrue: [combien ← 1]
    ifFalse: [anInteger = 3
      ifTrue: [combien ← 2]
      ifFalse: [combien ← (Random new next * 1000 rem: 3) truncated + 1]].
  ↑combien!

gagner
  "le joueur se proclame gagnant"

  Transcript cr; show: '--> Moi, ', self nom, self prenom, ' j''ai gagné'.
  ↑self!

jouerTas: anInteger
  "le joueur joue un coup"

  | combien |
  combien ← self compterTas: anInteger.
```

```

Transcript cr; show: 'Il y a ', anInteger printString, ' allumette(s)'.
Transcript show: ' moi, ', self nom, self prenom, ' j''enlève ', combien printString, ' allumette(s)'.
↑combien!!
"-----"!

Joueur class
instanceVariableNames: ''!

!Joueur class methodsFor: 'instance creation'!

newNom: aString prenom: aString1
↑(super new) nom: aString; prenom: aString1!

```

Figure 4 : *Implantation Smalltalk de la classe Joueur*

```

Joueur subclass: #JoueurIntelligent
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Nim'!

!JoueurIntelligent methodsFor: 'jeu'!

compterTas: anInteger
"le joueur intelligent détermine combien il enlève d'allumettes du tas"
"Le calcul n'est plus aléatoire"

| combien reste |
reste ← anInteger rem: 4.
reste = 0
ifTrue: [combien ← 3]
ifFalse: [reste = 1 | (reste = 2)
ifTrue: [combien ← 1]
ifFalse: [combien ← 2]].
↑combien!!

```

Figure 5 : *Implantation Smalltalk de la classe JoueurIntelligent*

```

Personne subclass: #Arbitre
instanceVariableNames: 'partieEnCours tas courant joueurs '
classVariableNames: 'NbMaxAllu '
poolDictionaries: ''
category: 'Nim'!
Arbitre comment:
'Cette classe implante la classe Arbitre du jeu de Nim.

Les variables d''instance sont :
partieEnCours : Boolean
tas : Integer [0..NbMaxAllu]
courant : Integer 1..2
joueurs : Array [1..2] of Joueur

La variable partieEnCours indique si une partie est démarrée, elle conditionne
les trois variables suivantes.
- tas indique le nombre d''allumettes restant dans le tas
- courant donne l''indice du joueur courant dans le tableau joueurs
- joueurs de la partie en cours '!

!Arbitre methodsFor: 'jeu'!

changer
"l'arbitre donne la main à l'autre joueur"

partieEnCours
ifTrue: [courant ← (courant rem: 2) + 1]
ifFalse: [self error: 'Pas de partie en cours']!

encore
"rend vrai si la partie n'est pas terminée"

```



```

partieEnCours
  ifTrue: [↑tas > 0]
  ifFalse: [self error: 'Pas de partie en cours']!

enlever
  "l'arbitre demande au joueur courant le nombre d'allumettes à enlever
  et il enlève ces allumettes."

partieEnCours
  ifTrue: [tas ← tas - ((joueurs at: courant) jouerTas: tas)]
  ifFalse: [self error: 'Pas de partie en cours']!

nimJ1: aJoueur1 j2: aJoueur2
  "l'arbitre crée un nouveau jeu avec un nombre quelconque d'allumettes
  compris dans 10 .. NbMaxAllu. Si une partie était en cours, elle est
  implicitement annulée."

partieEnCours ← true.
joueurs ← Array with: aJoueur1 with: aJoueur2.
courant ← 1. "on aurait pu choisir au hasard"
tas ← 10 + (Random new next * 10000 rem: NbMaxAllu - 9) truncated + 1.
[self encore]
  whileTrue:
    [self enlever.
     self changer].
(joueurs at: courant) gagner.
↑self !!

!Arbitre methodsFor: 'initialize-release'!

initialize
  "Au départ, il n'y a aucune partie en cours"

  partieEnCours ← false! !
"-----"!

Arbitre class
  instanceVariableNames: ''!

!Arbitre class methodsFor: 'instance creation'!

newNom: aString prenom: aString1
  ↑(super new) nom: aString; prenom: aString1; initialize !!

!Arbitre class methodsFor: 'initialize-release'!

initialize
  "Arbitre initialize "

  NbMaxAllu ← 30! !

!Arbitre class methodsFor: 'examples'!

unePartie
  "Cette méthode crée un arbitre et deux joueurs, lance le jeu et affiche les résultats sur la console."
  "Arbitre unePartie"

  | j1 j2 arbitre |
  j1 ← Joueur newNom: 'Alain' prenom: 'Térier'.
  j2 ← Joueur newNom: 'Alex' prenom: 'Andrin'.
  arbitre ← self newNom: 'Sémoi' prenom: 'Lechef'.
  arbitre nimJ1: j1 j2: j2! !

Arbitre initialize !

```

Figure 6 : *Implantation Smalltalk de la classe Arbitre*

L'évaluation du message `Arbitre unePartie` affiche le résultat suivant :

```

Il y a 20 allumette(s)   moi, AlainTérier j'enlève 3 allumette(s)
Il y a 17 allumette(s)  moi, AlexAndrin j'enlève 3 allumette(s)

```

```

Il y a 14 allumette(s)    moi, AlainTérier j'enlève 2 allumette(s)
Il y a 12 allumette(s)    moi, AlexAndrin j'enlève 1 allumette(s)
Il y a 11 allumette(s)    moi, AlainTérier j'enlève 2 allumette(s)
Il y a 9 allumette(s)     moi, AlexAndrin j'enlève 1 allumette(s)
Il y a 8 allumette(s)     moi, AlainTérier j'enlève 1 allumette(s)
Il y a 7 allumette(s)     moi, AlexAndrin j'enlève 3 allumette(s)
Il y a 4 allumette(s)     moi, AlainTérier j'enlève 3 allumette(s)
Il y a 1 allumette(s)     moi, AlexAndrin j'enlève 1 allumette(s)
--> Moi, AlainTérier j'ai gagné

```

L'évaluation du message `Arbitre unePartieInt` affiche le résultat suivant :

```

Il y a 26 allumette(s)    moi, AlainTérier j'enlève 2 allumette(s)
Il y a 24 allumette(s)    moi, AlexLefort j'enlève 3 allumette(s)
Il y a 21 allumette(s)    moi, AlainTérier j'enlève 2 allumette(s)
Il y a 19 allumette(s)    moi, AlexLefort j'enlève 2 allumette(s)
Il y a 17 allumette(s)    moi, AlainTérier j'enlève 1 allumette(s)
Il y a 16 allumette(s)    moi, AlexLefort j'enlève 3 allumette(s)
Il y a 13 allumette(s)    moi, AlainTérier j'enlève 2 allumette(s)
Il y a 11 allumette(s)    moi, AlexLefort j'enlève 2 allumette(s)
Il y a 9 allumette(s)     moi, AlainTérier j'enlève 3 allumette(s)
Il y a 6 allumette(s)     moi, AlexLefort j'enlève 1 allumette(s)
Il y a 5 allumette(s)     moi, AlainTérier j'enlève 3 allumette(s)
Il y a 2 allumette(s)     moi, AlexLefort j'enlève 1 allumette(s)
Il y a 1 allumette(s)     moi, AlainTérier j'enlève 1 allumette(s)
--> Moi, AlexLefort j'ai gagné

```

Faire de même pour les classes `Joueur`, `Joueur`, `Joueur`,

4.3 Conclusion

Dans cette section, nous avons mis en œuvre un programme Smalltalk à partir d'une description abstraite pour illustrer l'utilisation des outils de Smalltalk pour le codage. Nous n'avons pas présenté les outils de mise au point (inspecteurs et débogueurs) bien que nous les ayons utilisés pour tester le programme.

Cet exemple montre qu'il est rapide de programmer en Smalltalk, c'est pourquoi le langage est utilisé pour le prototypage rapide. Il montre aussi que la connaissance des classes du système influence fortement le résultat.

References

- [AR98] Pascal André and Jean-Claude Royer. Modélisation par objets. Rapport de Recherche RR-179, IRIN, Octobre 1998. (54 pages).
- [AV01] Pascal André and Alain Vailly. *Spécification des logiciels ; Deux exemples de pratiques récentes : Z et UML*, volume 2 of *Collection Technosup*. Editions Ellipses, 2001. ISBN 2-7298-0774-8.