Chapitre 2

Le modèle objet avec UML

2.1 Les raisons d'une méthodologie objet

2.1.1 L'objet, un élément de programme dynamique opératoire qui simplifie la complexité d'un problème

– Objet : modules cohérants regroupant des données et des opérations associées
– Ils sont créés dynamiquement au cours d'un programme (instanciation) à partir d'une classe, abstraction statique qui définit les opérations réalisables par ses objets.

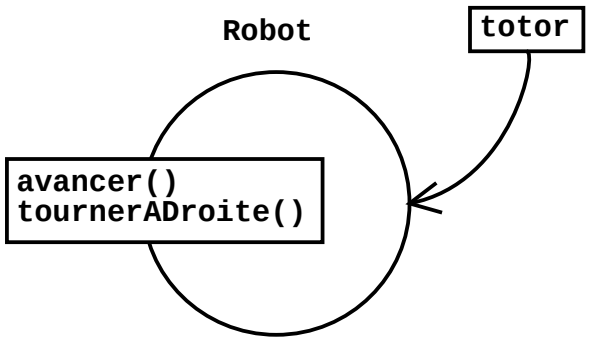

FIG. 2.1: totor est une instanciation de Robot

– Approche décentralisée d'un développement logiciel : des unités élémentaires, objets, qui interagissent
  – simplifie la complexité d'une description globale parfois non souhaitable ou non accessible
  – pas nécessaire de connaître le système complet pour en développer une partie

2.1.2 L'encapsulation

– Un objet est une coquille cachant à l'utilisateur son contenu (données ou opérations)
– Interface publique : elle décrit les données ou opérations accessibles à l'extérieur de l'objet (par les autres objets).
– Interface privée : elle décrit les données ou méthodes non visibles hors de l'objet.

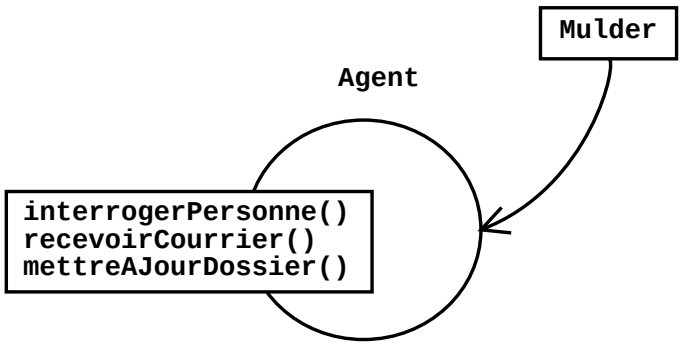

FIG. 2.2: interface publique de l'agent mulder

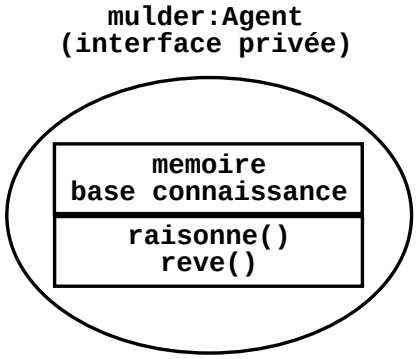

FIG. 2.3: interface privée de l'agent mulder

2.1.3 L'héritage

- Une classe A peut se définir comme spécialisation d'une autre classe B. On dit que A hérite de B. A hérite de toutes les données de B et peut en définir d'autres ou en adapter certaines à ses caractéristiques.
- Permet la réutilisabilité
- Permet de "factoriser" des parties communes
- Facilite les évolutions du programme

2.1.4 Bilan : réutilisabilité, adaptabilité et sureté

- Réutilisabilité : grâce à l'héritage et à la définition de classes génériques pouvant être dérivée.
- Sureté : grâce à l'encapsulation. Le développeur ne possède que des contextes locaux délimités [1] et plus faciles à gérer.

2.2 Décrire les utilisateurs et les cas d'utilisation : diagramme de cas

- Objectif : conceptualiser le problème du point de vue du client/utilisateur
- Représentation orientée utilisateur

[1] la vérité est ailleurs

– Découper le système en grandes tâches à répartir entre les équipes de développement
– Outil privilégié de communication entre les équipes et avec les clients

2.2.1 Les utilisateurs

On recense les différents types d'utilisateurs en interaction avec le système.

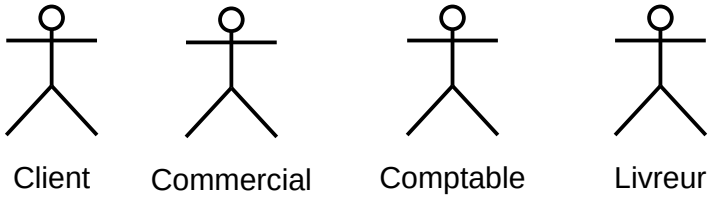

FIG. 2.4: exemple d'utilisateurs

2.2.2 Les cas d'utilisation

On recense les actions ou évènements demandés ou réalisés par les utilisateurs autour du système.

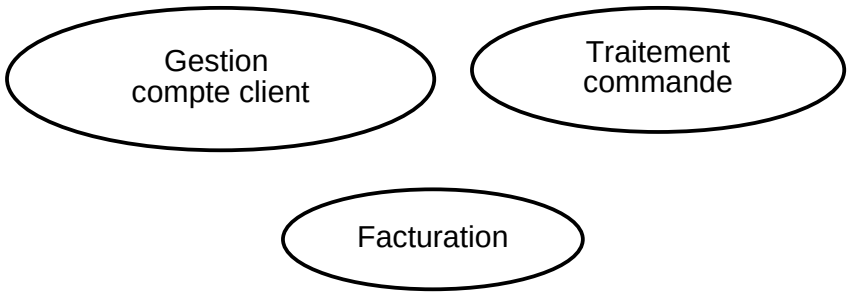

FIG. 2.5: exemple de cas d'utilisation

2.2.3 Les diagrammes de cas

– Liaisons entre utilisateurs et cas
– Liaisons entre cas d'utilisation ("uses" ou "includes"). Sur l'exemple, la gestion de compte du client "utilise" le traitement de commande qui "utilise" la facturation.
– On peut aussi introduire une liaison "extends" entre des cas : c'est analogue à de l'héritage.

2.3 Décrire des objets : diagramme de classe

Une classe est un modèle à partir duquel on construire des objets. Elle est représentée par un rectangle découpé en 3 parties :

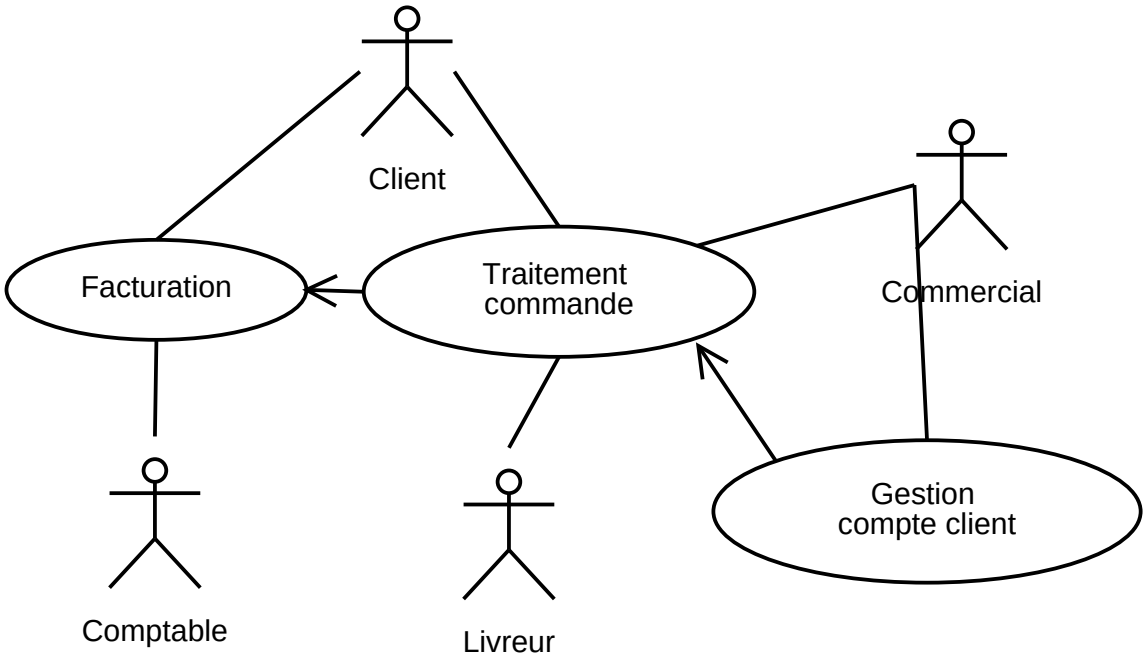

FIG. 2.6: exemple de diagramme de cas

– Identifiant de la classe
– Liste des attributs
– Liste des méthodes

Les attributs ou méthodes sont précédés d'un symbole désignant sa visibilité :
- l'attribut ou la méthode est privé (visible que dans la classe)
+ l'attribut ou la méthode est public (visible partout)
# l'attribut ou la méthode est protégé (visible dans la classe et ses sous-classes)

– Les méthodes sont suivie par " :" et le type de retour, s'il y en a un (non indiqué si de type "void").
– Les attributs peuvent être suivis de "=" et d'une valeur par défaut.
– Attention : Au niveau de la description UML, les attributs ne peuvent être que de type élémentaire (int, double, boolean, string, ...) et il ne peuvent être de type objet. Pour définir une telle dépendance, on utilise les associations.

La figure 2.7 est un exemple de classe "compte bancaire" :
La traduction en Java de cette classe, sans son implémentation, est la suivante :

```
class Compte {
   string titulaire;
   double solde = new Somme(0.0);

   public void ouvrir() { ... };
   public void fermer() { ... };
   public void crediter(double x) { ... };
   public void debiter(double x) { ... };
```

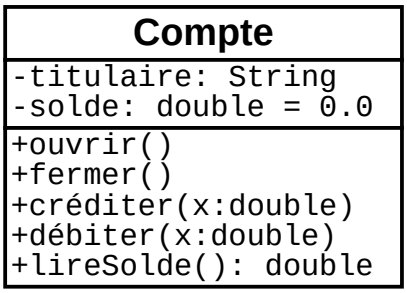

FIG. 2.7: Représentation d'une classe Compte

```
   public double lireSolde() { ... };
}
```

Pour désigner un objet d'une classe, on le spécifiera de la manière suivante :

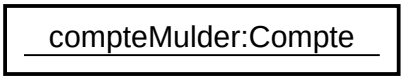

FIG. 2.8: Représentation d'un objet

2.3.1 Les associations

– Les associations correspondent à des relations entre objets.
– Description alternative des attributs-classes, interdits en UML
– Elles sont nommées avec un sens de lecture, si nécessaire
– Un rôle est attribué à chaque classe associée. Il est indiqué aux extrémités des relations.

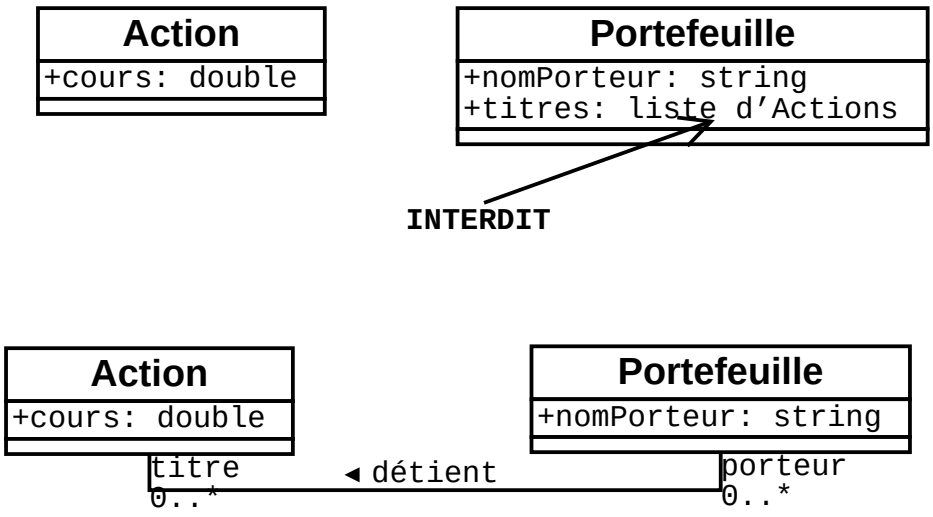

FIG. 2.9: Exemple d'association

Cardinalités
Les cardinalités sont associées aux rôles et indique le nombre d'objets d'une même classe participant à l'association :
– 1 : obligatoire (un et un seul)
– 0..1 : optionnel (O ou 1)
– 0..* ou * : quelconque
– n..m : entre n et m
– l,n,m : l, n ou m

2.3.2 Sous-type et généralisation

Correspond à la notion d'héritage en C++ et Java. Sur l'exemple suivant, "CompteEpargne" est une spécialisation de "Compte" ("Compte" est une généralisation de "CompteEpargne").

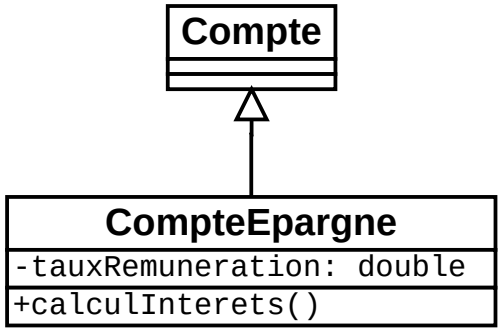

FIG. 2.10: Exemple de généralisation

2.3.3 Agrégation et composition

Concerne les relations "partie de" qui peuvent être vues de 2 manières.
– La composition, représentée par un losange noir, signifie que l'objet est une partie indissociable de l'objet auquel il est lié. Par exemple, une chambre d'hôtel et son hôtel de rattachement : si ce dernier disparait, la chambre disparaitra aussi.
– L'agrégation, représentée par un losange blanc, signifie que l'objet est référencé par l'objet auquel il est lié, il n'en est pas la propriété. Par exemple, les wagons d'un train peuvent être ultérieurement rattachés à un autre train.

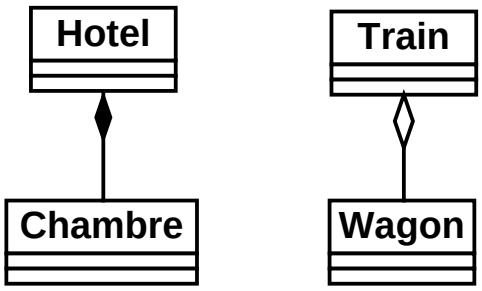

FIG. 2.11: Exemple de composition et d'agrégation

2.3.4 Compléments et notions avancées

– Classe abstraite : classe générique qui n'est pas instanciée dans l'application
– Classe paramétrée : correspond à des templates en C++.
– Interface : décrit un comportement générique de classe. Similaire aux interfaces Java.
– Associations n-aire
– Attributs et classes d'association

2.4 Décrire des programmes : diagrammes de séquence, d'état et d'activité

2.4.1 Collaboration entre les objets, diagrammes de séquence

– Les diagrammes de séquences ont pour objet de décrire des scénario particuliers de comportements des acteurs vis-à-vis du système.
– Ensemble de colonnes représentant chacun un objet ou un acteur. La longueur de la colonne correspond à la durée de vie de son interaction avec les autres.
– Les flèches entre les colonnes correspondent aux messages envoyés.

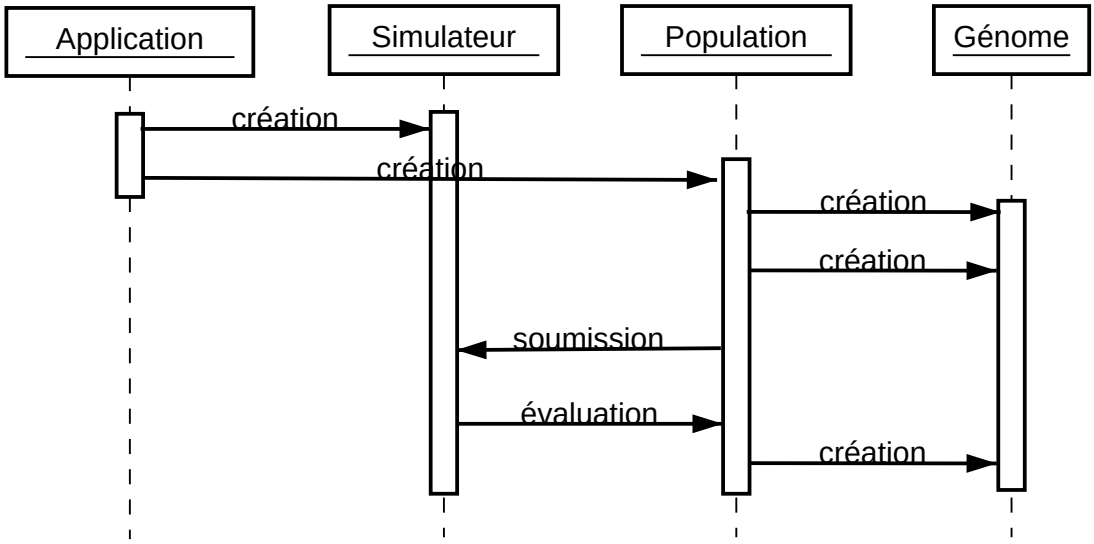

FIG. 2.12: Exemple de diagramme de séquence

– Un programmeur doit pouvoir coder à partir d'un diagramme de séquence.
– Eviter les diagrammes tentaculaires. Un diagramme de séquence doit correspondre à un algorithme simple.

2.4.2 Aspect dynamique d'un objet : automate et diagramme d'état

Les diagrammes d'état indiquent les changement d'état d'un objet à travers les cas d'utilisation dans lesquels il est impliqué.
– Etat-transition De la forme
<nomEvenementDeclancheur>[<contrainte>]/<nomActionDéclanchée>.

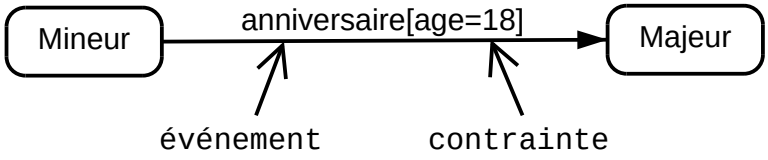

FIG. 2.13: Exemple d'état-transition

– Une transition peut provoquer une activité, notée "do/action" : opération durable, interruptible, associée à un état. Exemple de la modélisation d'un réveil à 3 boutons : alarm on/ alarm off/ stop sonnerie

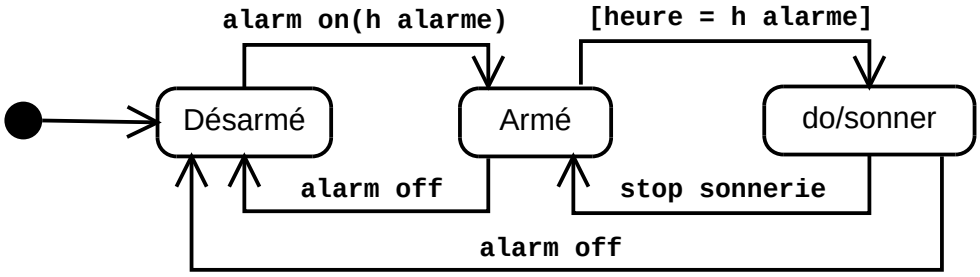

FIG. 2.14: Automate réveil

– Action en entrée et en sortie :
  – Action en entrée (entry) exécutée à chaque entrée dans l'état (ex : redessiner le contenu de la fenêtre quand la souris y est).
  – Action en sortie (exit) exécutée à chaque sortie d'un état (ex : mettre à jour un compteur de visite).

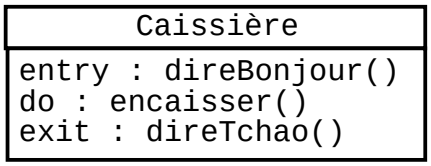

FIG. 2.15: Représentation des actions d'entrée et de sortie

– Diagrammes hiérarchiques
  – Permettent de structurer la description et en facilite la lecture.
  – On entre dans l'état englobant puis dans ses sous-états.

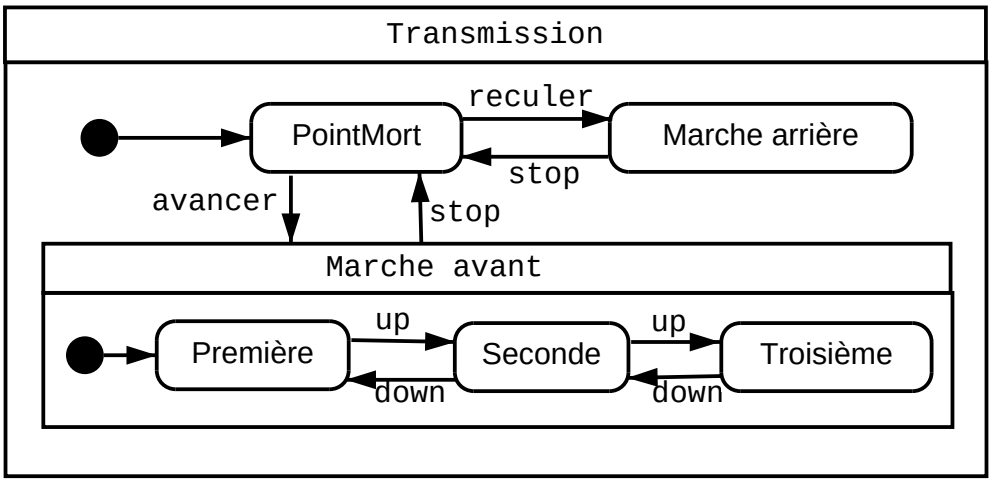

FIG. 2.16: Diagramme hiérarchique

– Parallélisme et synchronisme

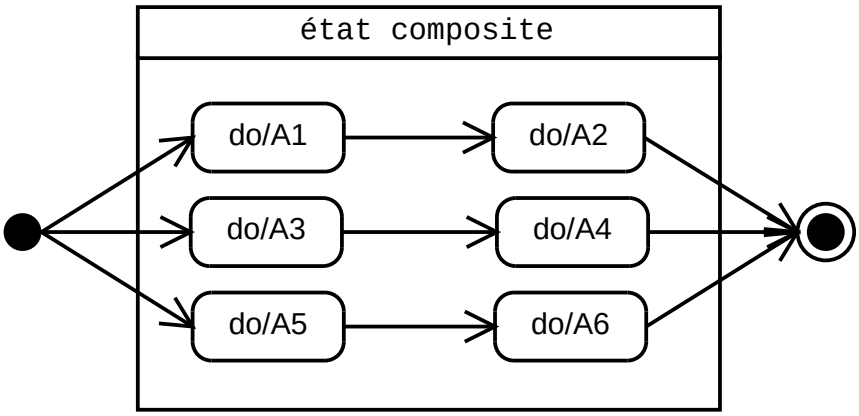

FIG. 2.17: Etat à traitement parallèle

2.4.3 Diagramme d’activité

– Doit correspondre au comportement d’une seule méthode
– Doit permettre de mettre en évidence les contraintes de séquentialité et de parallélisme

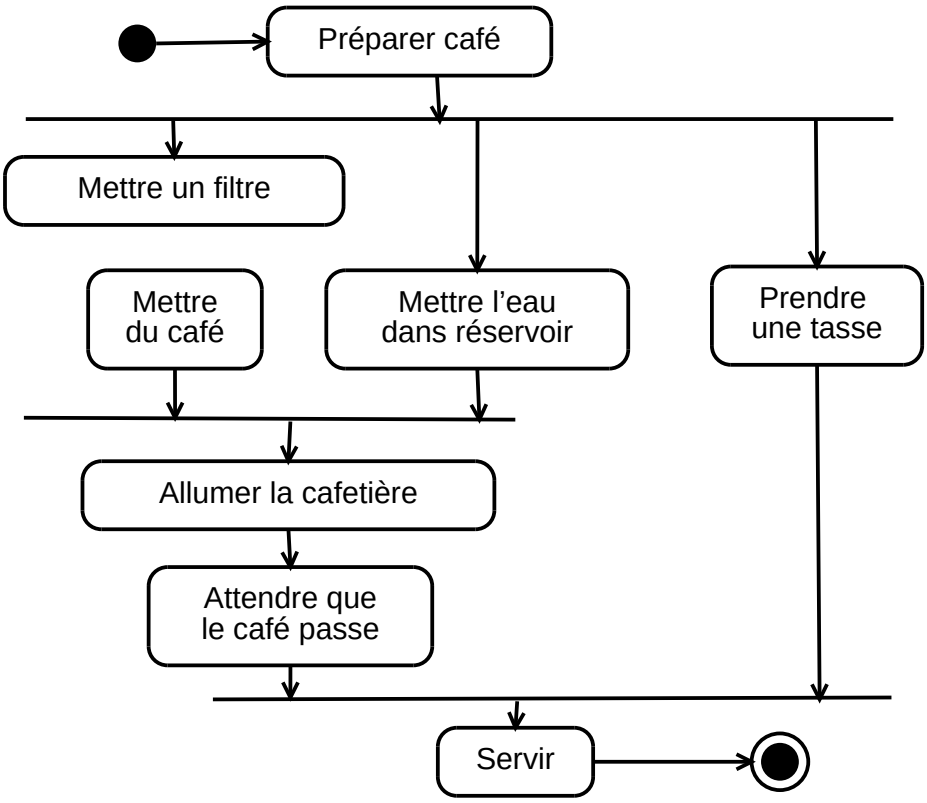

FIG. 2.18: Exemple de diagramme d’activité, la pause café

2.5 Présentation rapide de BlueJ

BlueJ est un outil de développement pour l'apprentissage du langage Java. C'est un interface utilisateur qui contient un éditeur et des accès faciles aux outils du JDK.

Il permet aussi de gérer interactivement des objets dont il fait une représentation graphique qui s'appuie sur UML. Sur la figure 2.19, on visualise l'interface qui représente une hiérarchie de classe.

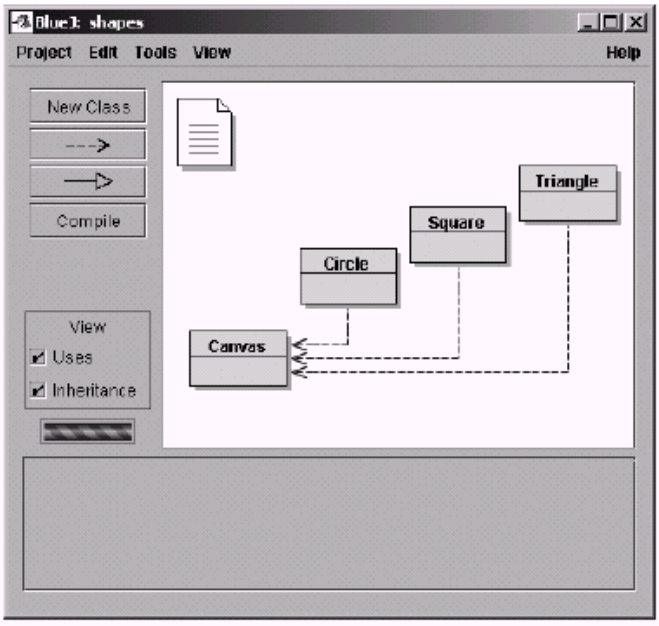

FIG. 2.19: Interface de BlueJ représentant une hiérarchie de classes

A partir de cette représentation, on peut alors, par exemple, cliquer-droit sur la classe "Circle" et choisir, à la souris, "new Circle()" : une représentation de l'instanciation apparaît alors dans le rectangle bas de l'interface (cf. figure 2.20).

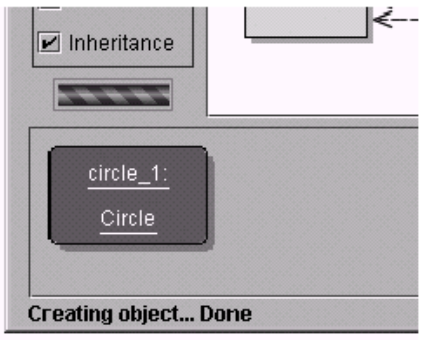

FIG. 2.20: Visualisation de l'instanciation d'un objet de classe Circle

On peut ensuite appeler des méthodes des objets créés, en cliquant-droit sur ceux-ci et choisir la méthode dans la liste des méthodes accessibles. Sur la figure 2.21, on a invoqué la méthode

“makeVisible()” de l’objet cercle. La visualisation se fait alors dans une fenêtre de visu.

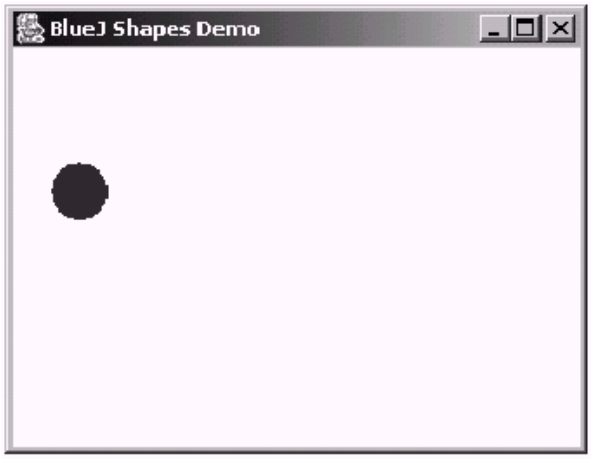

FIG. 2.21: Tracé graphique du cercle instancié dans une fenêtre de visu

Lorsqu’une méthode attend des paramètres, à la sélection de celle-ci, une fenêtre apparaît, avec un champ de formulaire pour pouvoir saisir des valeurs aux paramètres. La figure 2.22 montre comment affecter une valeur au paramètre de la méthode “moveHorizontal”.

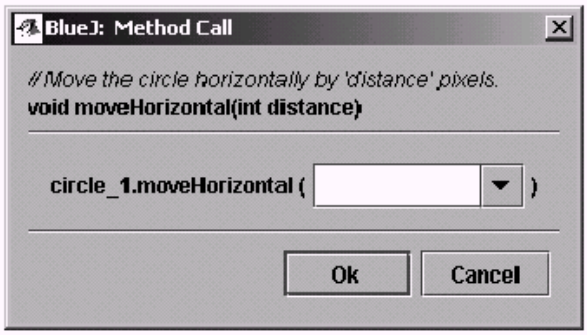

FIG. 2.22: Saisie de la valeur d’un paramètre

BlueJ propose aussi des fenêtres d’inspection des différents attributs des objets (cf. figure 2.23). Vous serez donc amené à manipuler et à utiliser BlueJ en TD/TP et à en connaître ainsi ses fonctionalités complémentaires.

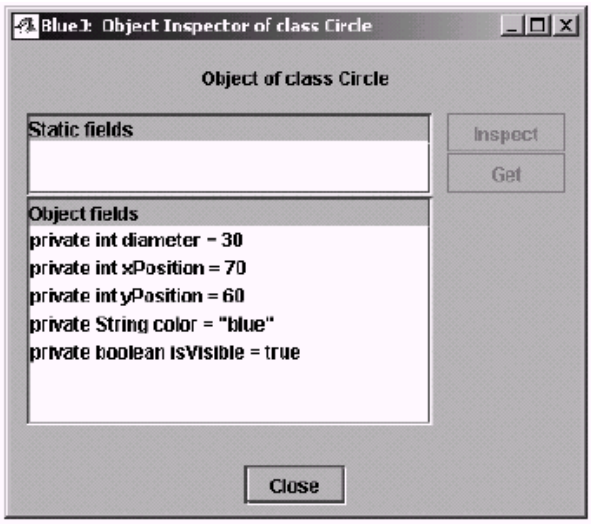
Blue1: Object Inspector of class Circle
Object of class Circle
Static fields
Inspect
Get
Object fields
private int diameter = 30
private int xPosition = 70
private int yPosition = 60
private String color = "blue"
private boolean isVisible = true
Close

FIG. 2.23: