Chapitre 6
La modélisation des objets

1. Introduction

L'objectif de ce chapitre est de vous faire découvrir les techniques UML de modélisation statique des objets.

Cette modélisation est statique car elle ne décrit pas les interactions ou le cycle de vie des objets. Les méthodes sont introduites d'un point de vue statique, sans description de leur enchaînement.

Nous découvrirons le diagramme de classes. Ce diagramme contient les attributs, les méthodes et les associations des objets. Comme nous l'avons vu au chapitre Les concepts de l'approche par objets, cette description est réalisée par les classes.

Ce diagramme est central lors d'une modélisation par objets d'un système. De tous les diagrammes UML, il est le seul obligatoire lors d'une telle modélisation.

Nous étudierons comment le langage OCL (*Object Constraint Language* ou langage de contraintes objet) peut étendre le diagramme de classes pour exprimer de façon plus riche les contraintes. Ensuite, le diagramme d'objets nous montrera comment illustrer la modélisation réalisée dans le diagramme de classes. Enfin, nous découvrirons comment décrire les objets composés au moyen du diagramme de structure composite.

L'emploi d'OCL, du diagramme d'objets ou du diagramme de structure composite est optionnel. Leur utilisation dépend des contraintes du projet de modélisation.

2. Découvrir les objets du système par décomposition

Au chapitre La modélisation de la dynamique, nous avons étudié comment découvrir les objets d'un point de vue dynamique. Nous avons présenté les cas d'utilisation sous forme de diagrammes de séquence. Puis nous avons enrichi ces diagrammes au niveau de l'envoi de message pour découvrir les objets du système.

La décomposition des messages fait apparaître les objets du système, car elle conduit à des messages plus fins dont il convient de rechercher le destinataire.

Une autre approche possible est la décomposition de l'information contenue dans un objet. Souvent, cette information est trop complexe pour n'être représentée que par la structure d'un seul objet. Elle doit parfois être également répartie entre plusieurs objets.

Exemple

Dans l'exemple du chapitre La modélisation de la dynamique, le directeur recherche les papiers (dans le sens d'informations) de la jument à vendre dans la base de données de l'élevage. Cette base constitue un objet à grosse granularité composé lui-même d'autres objets, comme les papiers des chevaux, les informations financières et comptables, les documents d'achat et de vente de chevaux. Les papiers d'une jument sont composés, entre autres, de son carnet de vaccination et des papiers de ses descendants. Les papiers des descendants sont partagés par d'autres objets, comme les papiers de leur père étalon. Cette décomposition est guidée par les données et non par des aspects dynamiques. La figure 6.1 illustre la composition de `PapiersJument` dans le diagramme de classes.

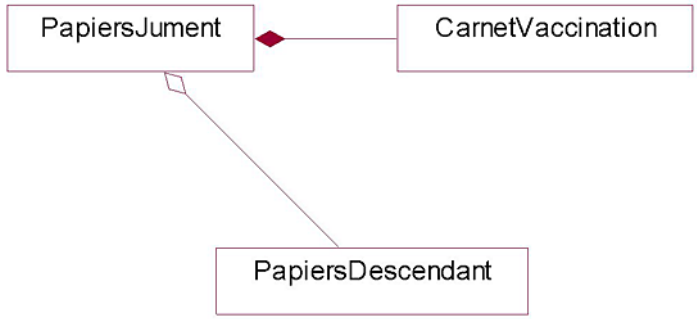

Figure 6.1 - Composition de `PapiersJument`

Remarque

Rappelons que la granularité d'un objet définit sa taille. Le système pris comme un objet est de gros grain ou de granularité importante. À l'opposé, le carnet de vaccination d'un cheval est un objet de grain beaucoup plus fin que le système.

Exemple

La décomposition d'un cheval pour faire apparaître ses différents organes peut se faire soit par la décomposition d'un diagramme de séquence, soit par la décomposition guidée par les données.

La décomposition par le diagramme de séquence consiste à analyser différents envois de message : faire peur, courir, manger, dormir. Ces derniers feront apparaître progressivement les différents organes du cheval. Elle est illustrée à la figure 6.2 pour le message `fairePeur`. Un cheval dilate ses naseaux pour marquer l'alerte, la surprise ou la peur. Il pince la bouche pour indiquer tension, peur ou colère. Enfin, la ruade est un mouvement défensif.

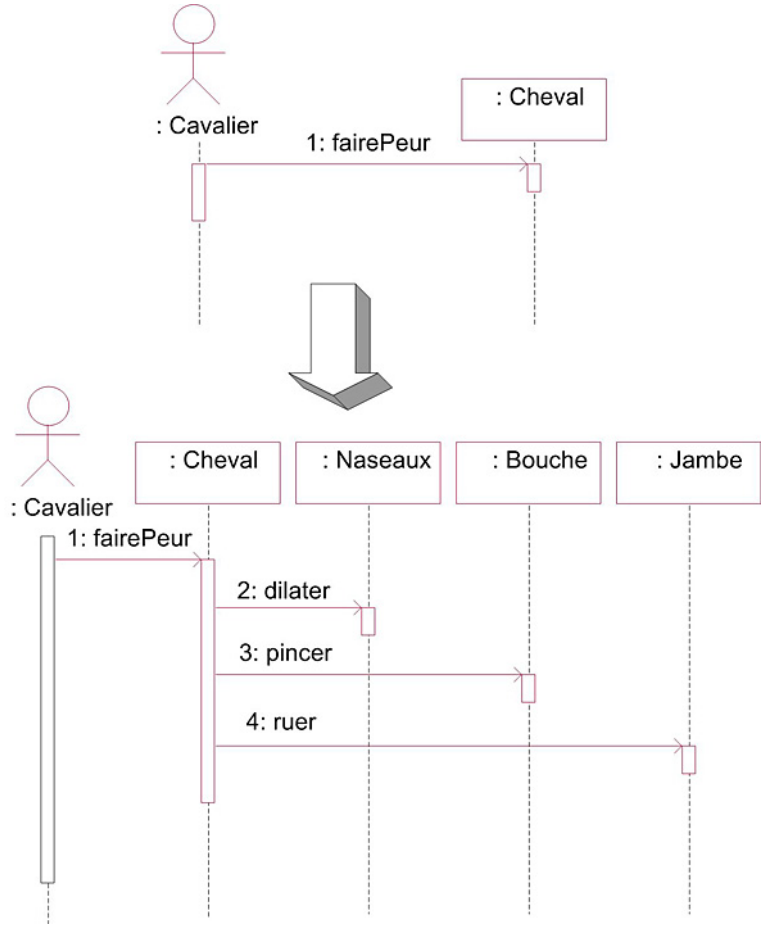

Figure 6.2 - Découverte des objets par enrichissement du diagramme de séquence

La décomposition guidée par les données consiste à étudier directement les différents organes d'un cheval et à les prendre en compte dans le diagramme de classes. La figure 6.3 illustre un cheval composé de ses différents organes.

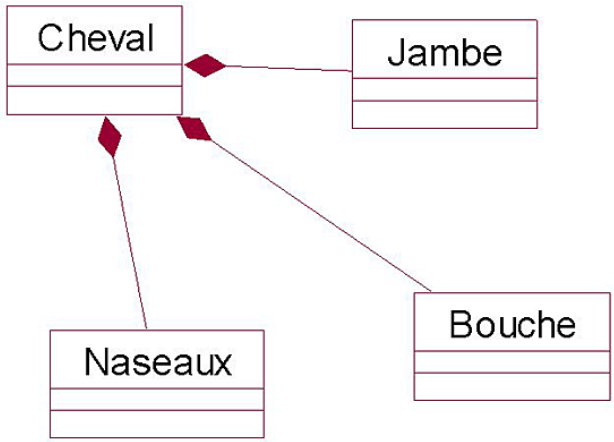

Figure 6.3 - Composition de `Cheval`

Remarque

La décomposition par le diagramme de séquence et la décomposition guidée par les données apparaissent naturelles pour découvrir les objets, ce qui est normal car un objet est l'assemblage d'une structure et d'un comportement. Il convient enfin de remarquer que ces deux approches ne sont pas incompatibles.

Remarque

La décomposition guidée par les données est plus efficace quand la personne chargée de la modélisation connaît bien le domaine. La décomposition en objets est alors immédiate.

3. La représentation des classes

3.1 La forme simplifiée de représentation des classes

Les objets du système sont décrits par des classes dont une forme simplifiée de la représentation en UML est donnée à la figure 6.4. Cette représentation est constituée de trois parties.

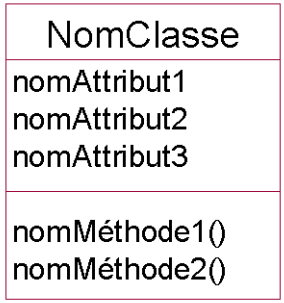

Figure 6.4 - Représentation simplifiée d'une classe en UML

La première partie contient le nom de la classe.

Remarque

Rappelons que le nom d'une classe est au singulier. Il est constitué d'un nom commun précédé ou suivi d'un ou plusieurs adjectifs qualifiant le nom. Ce nom est significatif de l'ensemble des objets constituant la classe. Il représente la nature des instances d'une classe.

La deuxième partie contient les attributs. Ceux-ci contiennent l'information portée par un objet. L'ensemble des attributs forme la structure de l'objet.

La troisième partie contient les méthodes. Celles-ci correspondent aux services offerts par l'objet. Elles peuvent modifier la valeur des attributs. L'ensemble des méthodes forme le comportement de l'objet.

Partie 5
Application des patterns

Chapitre 5-1
Compositions et variations de patterns

1. Préliminaire

Les vingt-trois patterns de conception introduits dans cet ouvrage ne constituent bien évidemment pas une liste exhaustive. Il est possible de créer de nouveaux patterns soit ex nihilo, soit en composant ou adaptant des patterns existants. Ces nouveaux patterns peuvent avoir une portée générale, à l'image de ceux introduits dans les chapitres précédents ou être spécifiques à un environnement de développement particulier. Nous pouvons citer ainsi le pattern de conception des JavaBeans ou les design patterns des EJB.

Dans ce chapitre, nous allons vous montrer trois nouveaux patterns obtenus par composition et variation de patterns existants.

2. Le pattern Pluggable Factory

2.1 Introduction

Nous avons introduit dans un précédent chapitre le pattern `Abstract Factory` pour abstraire la création (instanciation) de produits de leurs différentes familles. Une fabrique est alors associée à chaque famille de produits. Sur le diagramme de la figure 5-1.1, deux produits sont exposés : les automobiles et les scooters, décrits chacun par une classe abstraite. Ces produits sont organisés en deux familles : traction essence et traction à l'électricité. Chacune de ces deux familles engendre une sous-classe concrète de chaque classe de produit.

Il existe donc deux fabriques pour les familles `FabriqueVéhicule-Essence` et `FabriqueVéhiculeÉlectricité`. Chaque fabrique permet de créer l'un des deux produits à l'aide des méthodes appropriées.

Ce pattern organise de façon très structurée la création d'objets. Chaque nouvelle famille de produits oblige à ajouter une nouvelle fabrique et donc une nouvelle classe.

À l'opposé, le pattern `Prototype` introduit dans le chapitre du même nom offre la possibilité de créer des nouveaux objets de façon très souple.

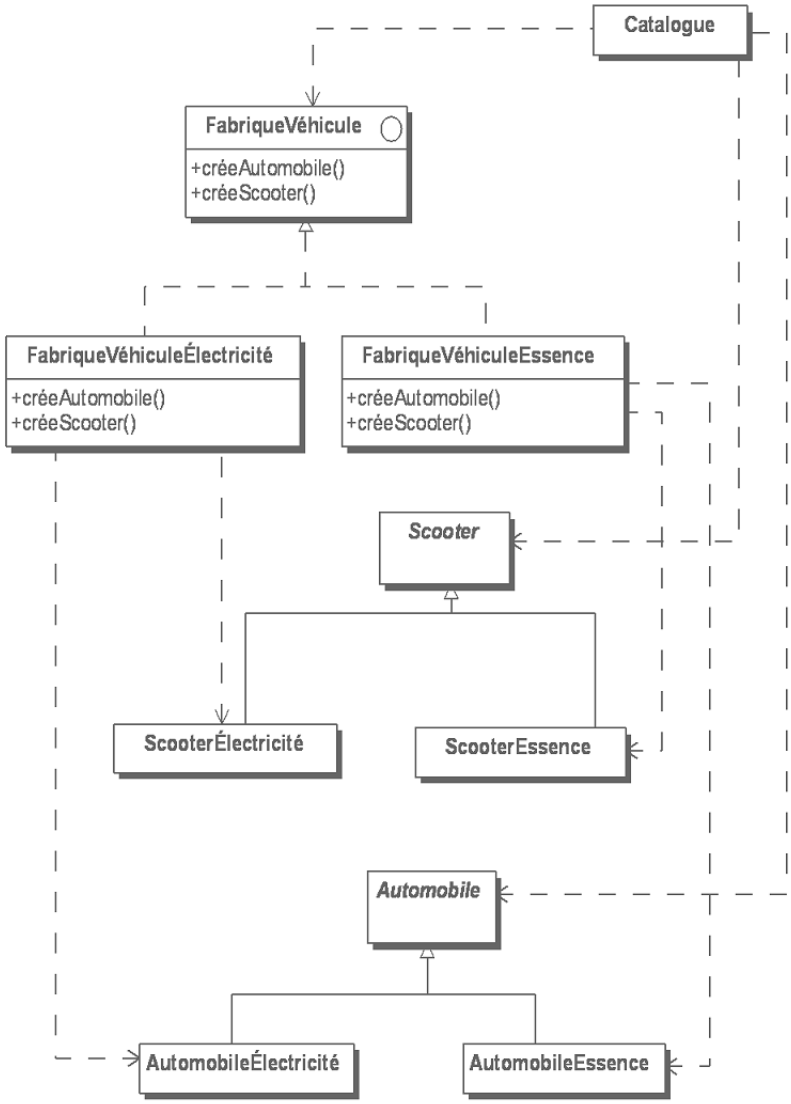

Figure 5-1.1 - Exemple d'utilisation du pattern `Abstract Factory`

La structure du pattern `Prototype` est décrite à la figure 5-1.2. Un objet initialisé afin d'être prêt à l'emploi et détenant la capacité de se dupliquer est appelé un prototype.

Le client dispose d'une liste de prototypes qu'il peut dupliquer lorsqu'il le désire. Cette liste est construite dynamiquement et peut être modifiée à tout moment lors de l'exécution. Le client peut ainsi construire de nouveaux objets sans connaître la hiérarchie de classes dont ils proviennent.

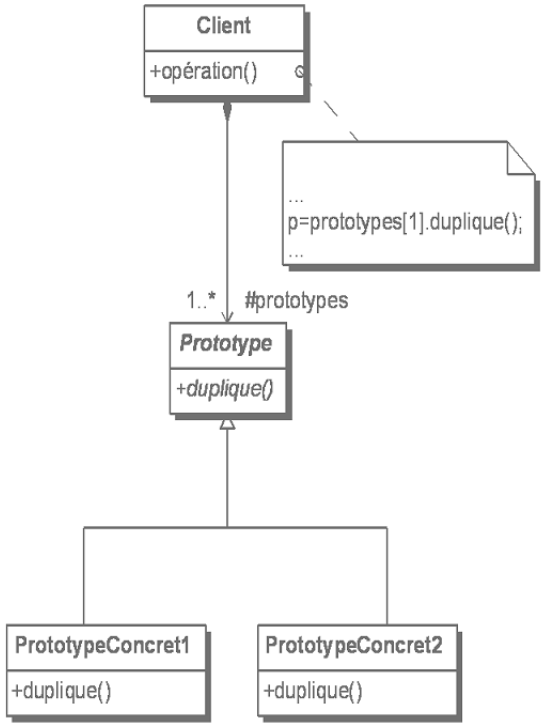

Figure 5-1.2 - Exemple d'utilisation du pattern `Prototype`

L'idée du pattern `Pluggable Factory` est de composer ces deux patterns pour conserver l'idée de création d'un produit par l'invocation d'une méthode de la fabrique et d'autre part, de pouvoir changer dynamiquement la famille à créer. Ainsi, la fabrique ne doit plus connaître les familles d'objets, le nombre de familles peut être différent pour chaque produit et enfin, il est possible de faire varier le produit à créer non plus uniquement par sa sous-classe (sa famille) mais aussi par des valeurs différentes de certains attributs. Nous reviendrons sur ce dernier point dans l'exemple Java.

La figure 5-1.3 reprend l'exemple du chapitre Le pattern Abstract Factory, structuré cette fois à l'aide du pattern `Pluggable Factory`. La classe de fabriques d'objets `FabriqueVéhicule` n'est plus une interface comme à la figure 5-1.1 mais une classe concrète qui permet la création des objets et qui n'a donc plus besoin de sous-classes. Chaque fabrique possède un lien vers un prototype de chaque produit. De façon plus précise, il s'agit d'un lien vers une instance de l'une des sous-classes de la classe `Automobile` et d'un lien vers une instance de l'une des sous-classes de la classe `Scooter`.

C'est ici qu'intervient le pattern `Prototype`. Chaque produit devient un prototype. La classe abstraite qui introduit et décrit chaque famille de produits leur confère la capacité de clonage. Elle joue ainsi le rôle de la classe abstraite `Prototype` de la figure 5-1.2.

Les deux liens présents dans `FabriqueVéhicule` vers chaque prototype peuvent être modifiés dynamiquement à l'aide des méthodes `setPrototype Automobile` et `setPrototypeScooter`. La fabrique nécessite d'ailleurs que ces liens soient initialisés soit à l'aide de ces deux méthodes, soit par un autre moyen (comme le constructeur de la classe) pour pouvoir fonctionner. Le fonctionnement de la fabrique est réalisé par les deux méthodes `créeAutomobile` et `créeScooter` qui s'appuient sur la capacité de clonage des deux prototypes.

La classe `Test` représente le client de la fabrique et des classes de produits. Nous verrons son rôle dans l'exemple Java.

Remarque

Le nom du pattern provient d'une part du fonctionnement de la fabrique de produits qui est dépendant des prototypes fournis et d'autre part de la possibilité de changer (« plugger ») dynamiquement ces prototypes.

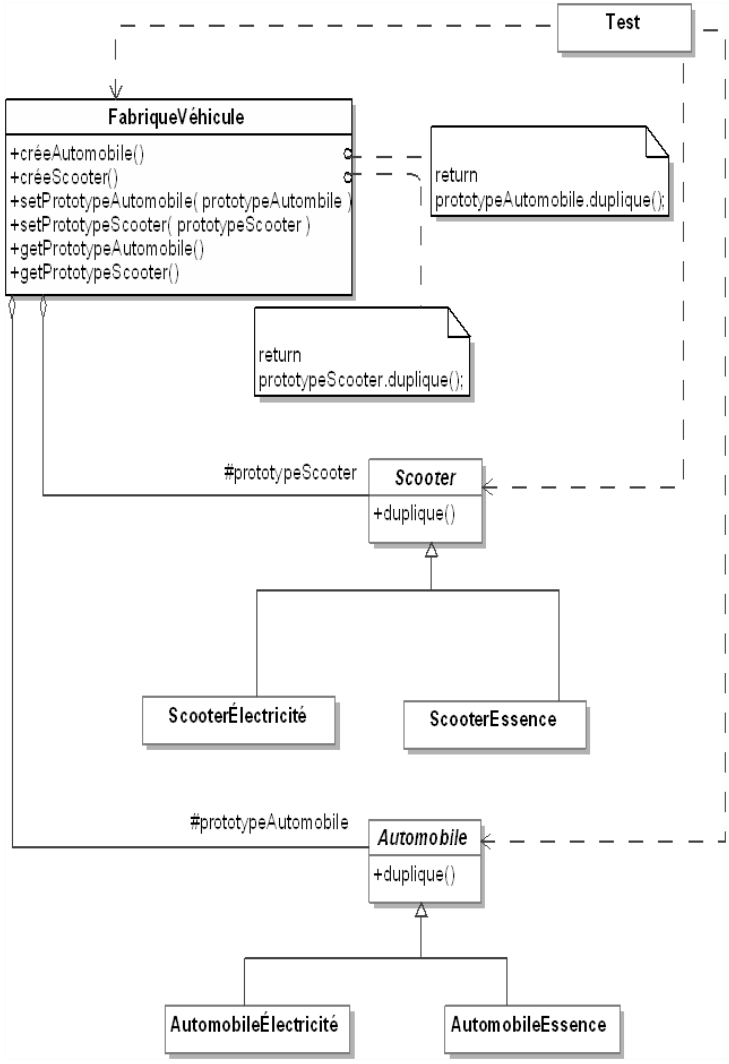

Figure 5-1.3 - Exemple d'utilisation du pattern `Pluggable Factory`

2.2 Structure

La figure 5-1.4 illustre la structure générique du pattern `Pluggable Factory`.

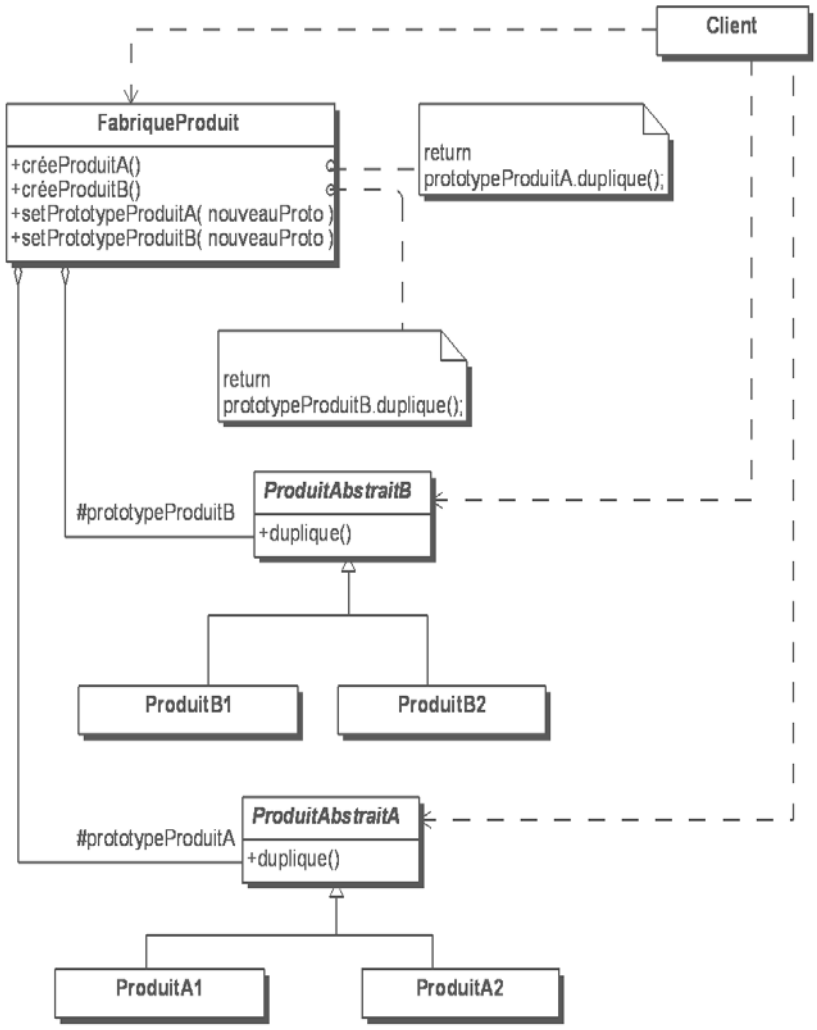

Figure 5-1.4 - Structure du pattern `Pluggable Factory`