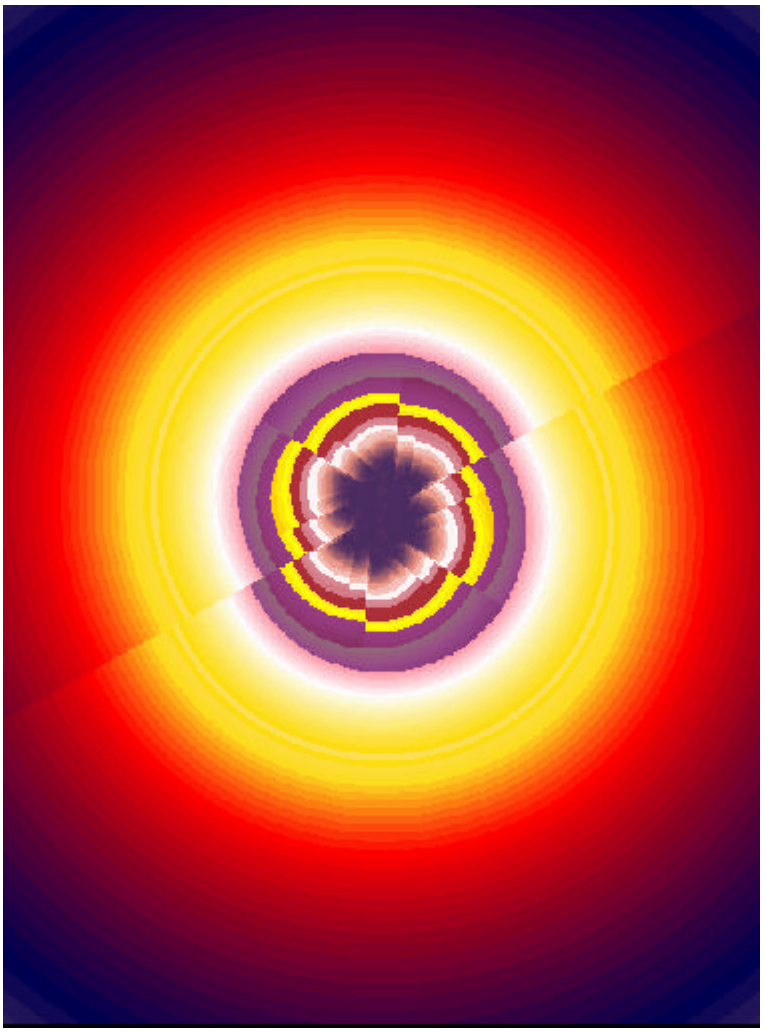
MCOURS.COM
Le N°1 du cours et exercices sur Internet

CHAPITRE 9

UML, diagrammes de classes



9.1 *Les diagrammes de classe*

Le diagramme de classes constitue un élément très important de la modélisation : il permet de définir quelles seront les composantes du système final : il ne permet en revanche pas de définir le nombre et l'état des instances individuelles. Néanmoins, on constate souvent qu'un diagramme de classes proprement réalisé permet de structurer le travail de développement de manière très efficace; il permet aussi, dans le cas de travaux réalisés en groupe (ce qui est pratiquement toujours le cas dans les milieux industriels), de séparer les composantes de manière à pouvoir répartir le travail de développement entre les membres du groupe. Enfin, il permet de construire le système de manière correcte (***Build the system right***).

9.1.1 *Les paquetages*

Les paquetages permettent typiquement de définir des sous-systèmes. Un sous-système est formé d'un ensemble de classes ayant entre elles une certaine relation logique. Souvent, un paquetage fait l'objet d'une réalisation largement indépendente, et peut être confiée à un groupe, ou à un individu n'ayant pas un contact étroit avec les responsables d'autres paquetages.

- Ils regroupent des éléments de modélisation, selon des critères purement logiques.
- Ils permettent d'encapsuler des éléments de modélisation (ils possèdent une interface).
- Ils permettent de structurer un système en catégories (vue logique) et sous-systèmes (vue des composants).
- Ils servent de "briques" de base dans la construction d'une architecture.
- Ils représentent le bon niveau de granularité pour la réutilisation.

Les paquetages sont aussi des espaces de noms.

FIGURE 9.1 Notation utilisée pour un paquetage

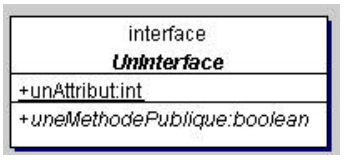
On peut ouvrir un package; par défaut, le package choisi est "default".

9.1.2 *Les interfaces*

Les interfaces représentent l'élément le plus abstrait du diagramme de classes. La manière habituelle pour représenter un interface est représentée à la figure9.2, page97. Un

membre est accompagné de son type, comme il est d'usage dans beaucoup de langages de projection.

FIGURE 9.2 Représentation d'un interface



A cette représentation est associée du code, dans le langage de projection choisi. Dans le cadre de ce chapitre, nous utiliserons systématiquement Java comme langage de projection. Il est bien entendu que la modélisation est largement indépendante du langage de projection; néanmoins, certaines différences d'implémentation peuvent apparaître, comme par exemple l'absence de la notion d'interface en C++, et son remplacement par la notion de classe abstraite purement virtuelle.

FIGURE 9.3 Traduction de la représentation en code Java

```
package UnPackage;

public interface UnInterface {
    boolean uneMethodePublique();

    int unAttribut = 0;
}
```

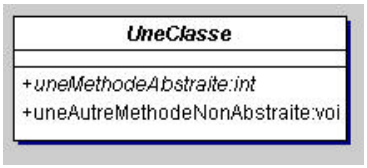
9.2 *Les classes*

La notion de classe est essentielle en programmation orientée objets : elle définit une abstraction, un type abstrait qui permettra plus tard d'instancier des objets. On distingue généralement entre classes abstraites (qui ne peuvent pas être instanciées) et classes "normales", qui servent à définir des objets.

9.2.1 *Classes abstraites*

Une classe abstraite ne peut donc pas être utilisée pour fabriquer des instances d'objets; elle sert uniquement de modèle, que l'on pourra utiliser pour créer des classes plus spécialisées par dérivation (héritage). Une classe abstraite est assez proche de la notion d'interface; d'ailleurs, la notion d'interface est généralement implémentée par une classe abstraite, dont toutes les méthodes sont purement virtuelles, en C++ (rappelons que C++ ne connaît pas la notion d'interface).

FIGURE 9.4 Représentation d'une classe abstraite



Le + dénote la publication du membre concerné. Une classe abstraite peut posséder des membres privés ou des attributs privés ou publics; d'autre part, les méthodes peuvent faire l'objet d'une implémentation, même si la méthode est purement virtuelle.

FIGURE 9.5 La même classe abstraite, projetée en Java

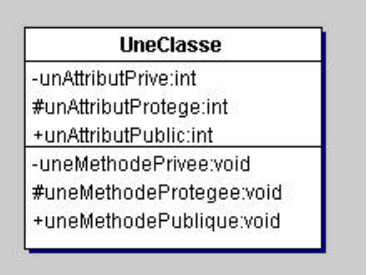
```
package UnPackage;

abstract public class UneClasse {
    public abstract int uneMethodeAbstraite();

    public void uneAutreMethodeNonAbstraite() {
    }
}
```

9.2.2 *Classes non abstraites*

Une classe "normale" ne contient pas de membres abstraits. Sa représentation en UML correspond à la figure9.6, page99.

MCOURS.COM
Le N°1 du cours et exercices sur Internet

FIGURE 9.6 Représentation d'une classe



Noter les petits signes "cabalistiques" précédant l'identification des méthodes :

- \- dénote un membre privé
- \+ dénote un membre public
- # dénote un membre protégé

D'ailleurs, la projection en Java est parfaitement claire à ce sujet (figure9.7, page99).

FIGURE 9.7 Code correspondant

```
public class UneClasse {
    private int unAttributPrive;
    protected int unAttributProtege;
    public int unAttributPublic;
    private void uneMethodePrivee() {};
    protected void uneMethodeProtegee() {};
    public void uneMethodePublique() {};
}
```

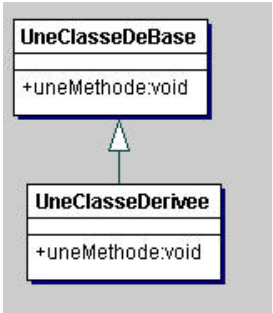
9.2.3 *Les relations entre les classes*

Les diverses classes possèdent des relations de dépendance entre elles. Ces relations possèdent en principe un équivalent syntaxique dans le langage de projection. Les principales de ces relations sont énumérées dans la suite. Il est néanmoins important de noter que certaines relations peuvent ne pas avoir d'équivalent dans le langage de projection considéré. Ainsi, C++ introduit la notion de template, ou classe paramétrable, qui peut être modélisée en UML, mais qui représente une notion inconnue en tant que telle en Java. A l'inverse, Java permet de définir une classe qui implémente un interface, alors que la notion d'interface est inexistante en C++. Dans les deux cas, l'outil de modélisation, s'il génère du code, choisira le mode de représentation le mieux adapté en considération du langage de projection choisi. Un interface UML pourrait, par exemple, se traduire par une classe abstraite en C++.

9.2.4 *Héritage*

L'héritage constitue une relation de spécialisation. Elle est notée, en UML, par une flèche allant de la classe spécialisée vers la classe originale (de la classe vers la superclasse).

FIGURE 9.8 Relation d'héritage



On notera que la relation inverse n'est en principe pas documentée; c'est que cette relation n'est en principe pas connue au moment de la conception d'un produit, et qu'elle est susceptible de changer au cours du temps. De plus, une superclasse n'est pas censée dépendre de ses dérivées : alors, à quoi bon le documenter ?

FIGURE 9.9 Code généré par la modélisation précédente

```
/* Generated by Together */

package UnPackage;

public class UneClasseDeBase {
    public void uneMethode() {
    }
}
```

```
/* Generated by Together */

package UnPackage;

public class UneClasseDerivee extends UneClasseDeBase {
    public void uneMethode() {
    }
}
```

9.2.5 *Implémentation*

Une classe peut implémenter un interface; elle peut aussi en implémenter plusieurs. En notation UML, cette relation est dénotée par une flèche en traitillés (figure9.10, page101)

FIGURE 9.10 Implémentation d'un interface

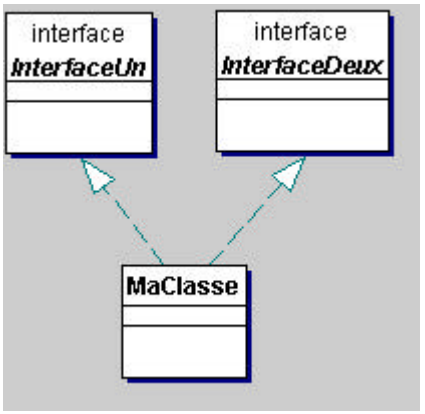


FIGURE 9.11 Code correspondant

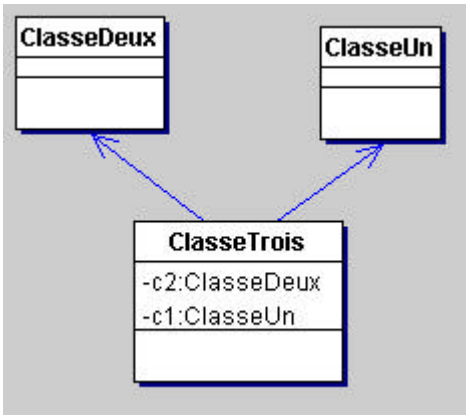
```
/* Generated by Together */

package UnPackage;

public class MaClasse implements InterfaceUn, InterfaceDeux {
}
```

9.2.6 *La relation d'aggrégation*

Lorsqu'un objet en contient d'autres, on parle d'aggrégation. Le diagramme de classes d'UML décrit cette relation par une flèche pleine, comme indiqué à la figure9.12, page102.

Il faut noter que l'aggrégation est parfois appelée "relation de contenance". On peut signaler aussi que UML ne propose pas de représentation spécifique pour l'héritage privé ou protégé, notions propres au langage C++, ce qui de l'avis de l'auteur est une bonne chose.

FIGURE 9.12 Relation d'aggrégation et code correspondant




```
/* Generated by Together */

package UnPackage;

public class ClasseTrois {
    private ClasseDeux c2;
    private ClasseUn c1;
}
```

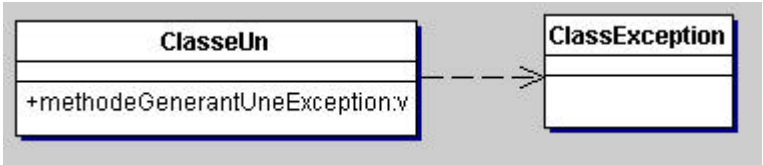
La relation d'aggrégation est souvent (et de manière fort judicieuse) implémentée par des membres privés. Il va de soi que ces derniers peuvent être protégés ou publics.

9.2.7 *Relation de dépendance*

La notion de dépendance est plus floue que les précédentes. Il est difficile de faire une nomenclature complète des possibles relations de dépendance. Certains outils de modélisation offrent la possibilité de tracer automatiquement les relations directes : hélas, les relations indirectes sont souvent négligées, parceque difficiles à détecter; d'autre part, le schéma résultant de ces outils est souvent encombré de relations évidentes qu'il eût peut-être mieux valu, pour des raisons de lisibilité, passer sous silence.

C'est pourquoi beaucoup d'outils laissent à l'utilisateur le soin de définir ce type de relation. La figure9.13, page103 montre un exemple de dépendance, étant bien entendu que ceci n'est qu'un exemple parmi de nombreux autres possibles.

FIGURE 9.13 Relation de dépendance et exemple de code correspondant




```
/* Generated by Together */

package UnPackage;

public class ClassException {
}
```

```
package UnPackage;

public class ClasseUn {
    public void methodeGenerantUneException() throws ClassException  {
    }

    /** @link dependency */
    /*#ClassException lnkClassException;*/
}
```

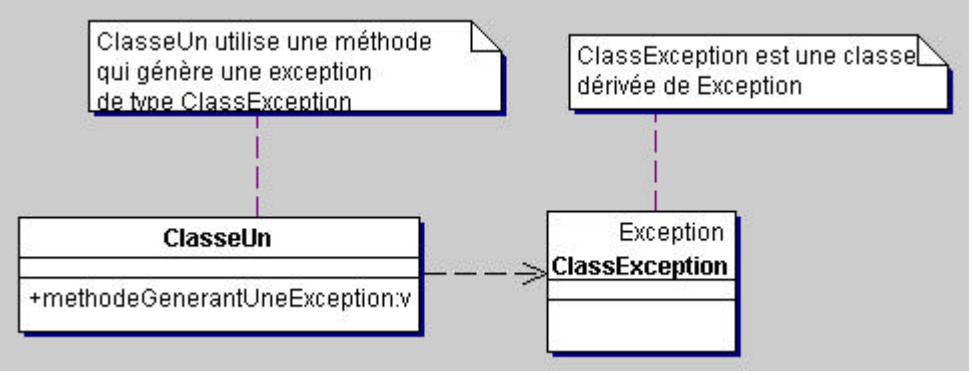
9.2.8 *Commentaires*

UML permet d'ajouter des notes (commentaires) à la représentation graphique. Les annotations peuvent être nécessaires pour préciser les intentions du concepteur.

Les annotations peuvent être ou non traduites en commentaires dans le langage de projection: tout dépend de l'outil utilisé. Certains commentaires de plus peuvent être insérés de manière systématique dans les fichiers sources : il en va en particulier ainsi pour le langage de projection Java, qui permet une extraction automatique de la documentation grâce à ses balises (*tag*) de documentation par les commentaires, et son utilitaire *javadoc* qui se sert de ces balises pour générer un fichier HTML.

MCOURS.COM
Le N°1 du cours et exercices sur Internet

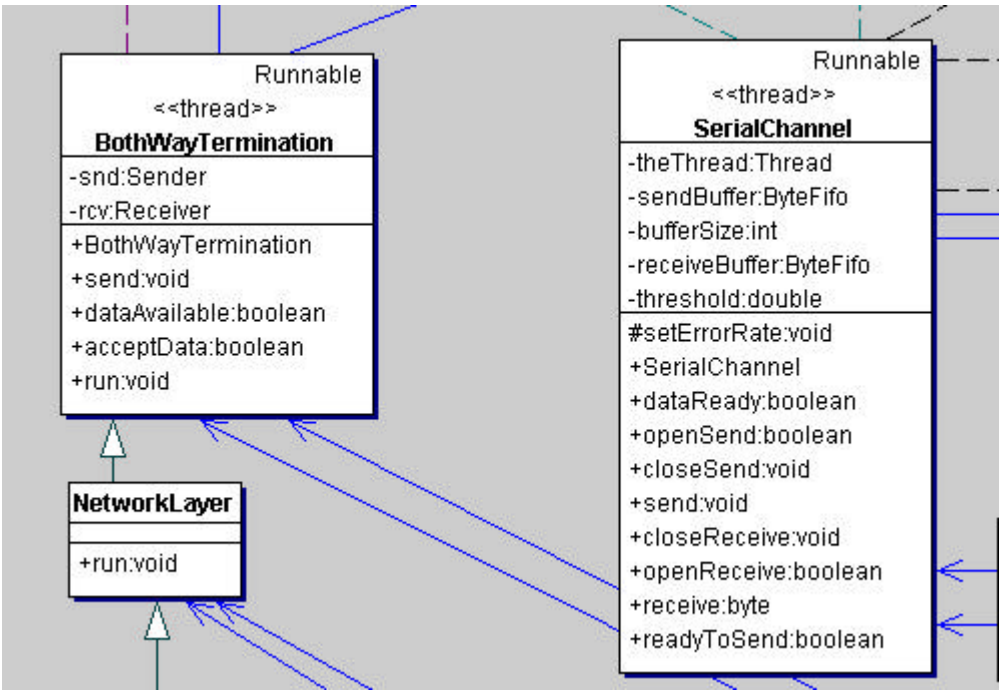
FIGURE 9.14 Commentaires dans le diagramme de classe UML



9.2.9 *Stéréotypes*

Un stéréotype dénote une appartenance à une catégorie de classes déterminée. Un développeur peut définir ses propres stéréotypes, mais il est souvent judicieux de parcourir la nomenclature des stéréotypes standards définis par UML, avant de commencer à inventer de nouveaux stéréotypes.

FIGURE 9.15 Exemples de stéréotypes



Certains outils permettent de colorier de manière spécifique certains stéréotypes (Together, par exemple). Cette possibilité peut paraître au premier abord un peu vaine : pourtant, elle devient intéressante dés que l'on utilise un développement à l'aide de "patrons" (*patterns*). Le stéréotype peut en effet identifier un patron d'un certain type, et la couleur permet de détecter plus facilement l'utilisation de patrons particuliers.

Le stéréotype n'a pas d'influence sur la génération de code; tout au plus y aura-t-il génération d'un commentaire. En Java, le stéréotype sera documenté sous forme de balise de documentation *javadoc*, ce qui permettra de le faire figurer automatiquement dans la documentation de la classe.

FIGURE 9.16 Code correspondant au stéréotype de la figure précédente

```
package SerialChannel;

/**
 * @stereotype thread
 */
public class BothWayTermination implements Runnable {
    public BothWayTermination(Sender snd, Receiver rcv, int bufsize) {
    }

    private Sender snd;
    private Receiver rcv;
    public void send(byte b) throws SendBufferFullException {}

    public boolean dataAvailable() {
    }

    public boolean acceptData() {
    }

    public void run() {
    }
}
```