CHAPITRE III

LIENS ENTRE LES CLASSES

Dans ce chapitre III, où vous allez découvrir qu'il existe des liens entre les classes, vous allez apprendre à :

- Expliquer ce qu'est un lien d'**association** entre deux classes,
- Faire la distinction entre la **composition** et l'**agrégation**,
- Expliquer ce que l'on entend par **généralisation** et **spécialisation**,
- Expliquer ce qu'apporte un lien d'**héritage** entre deux classes,
- Créer une **classe abstraite** (et pour quelles raisons !),
- Comment vous servir des **interfaces** pour retrouver les avantages de l'héritage multiple, impossible à implémenter directement dans la plupart des langages,
- Comment mettre toutes ces notions en place dans un **langage Objet** tel que Java ou C#.

Et… vous allez retrouver la notion de **polymorphisme**, mais cette fois dans le cadre de l'héritage !

SOMMAIRE

1. LES LIENS ENTRE CLASSES

Un des apports importants de l'orienté objets est la possibilité de définir de nouveaux types de liens entre les classes.

En plus du lien d'association, déjà présent dans les méthodes d'analyse et de représentation systémique du type de MERISE, ces nouveaux liens sont de deux types :

- la **généralisation** et la **spécialisation,** qui introduisent la notion de hiérarchie entre les classes,
- l'**agrégation** et la **composition**, qui permettent de définir une classe comme étant composée de classes plus élémentaires.

2. AGREGATION ET COMPOSITION

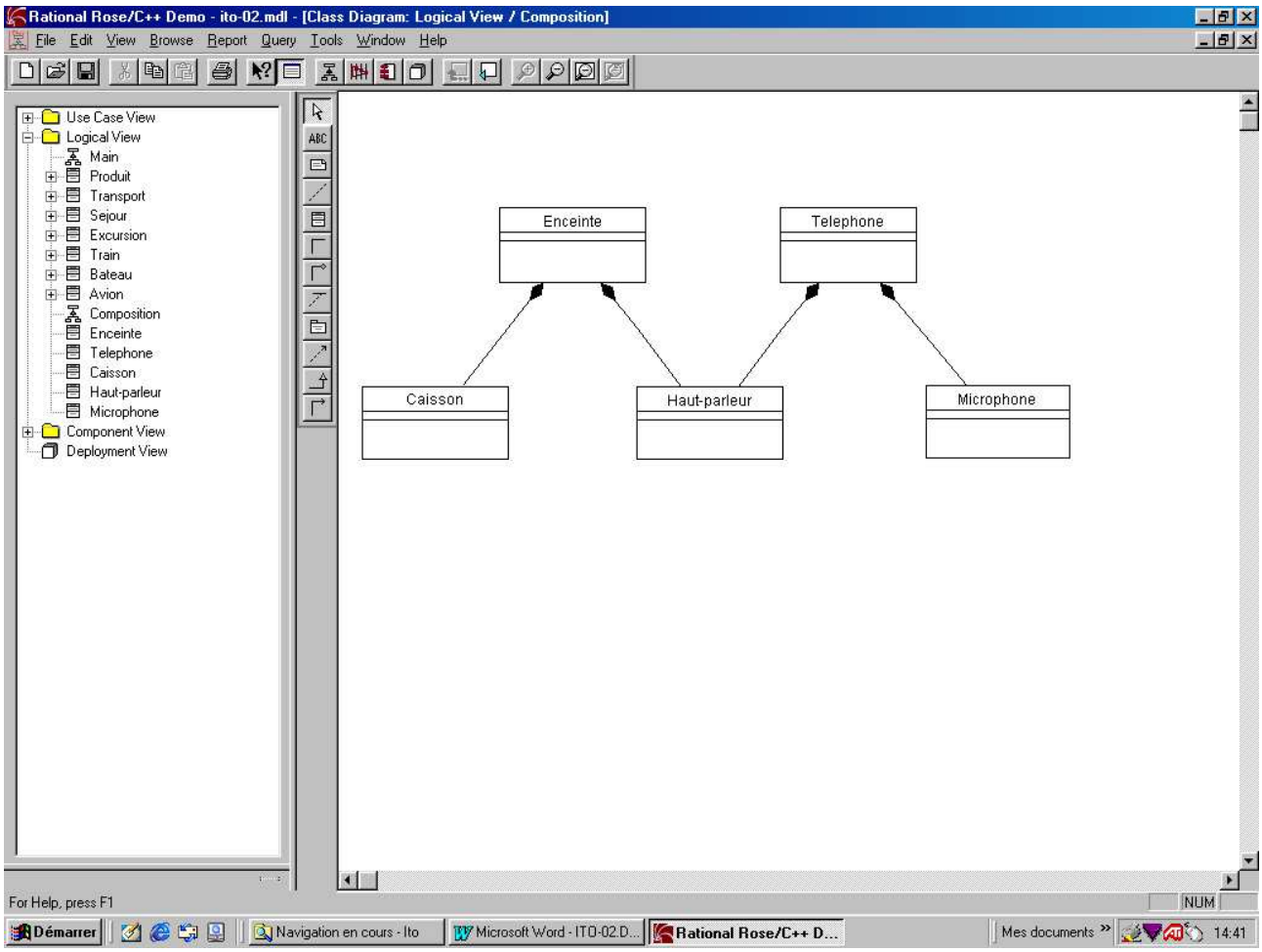
2.1. DEFINITION

L'**agrégation** comme la **composition,** est une forme d'association forte dans laquelle un objet est composé d'objets.

L'agrégat est un objet composé logiquement d'objets.

Le composé est physiquement constitué d'objets.

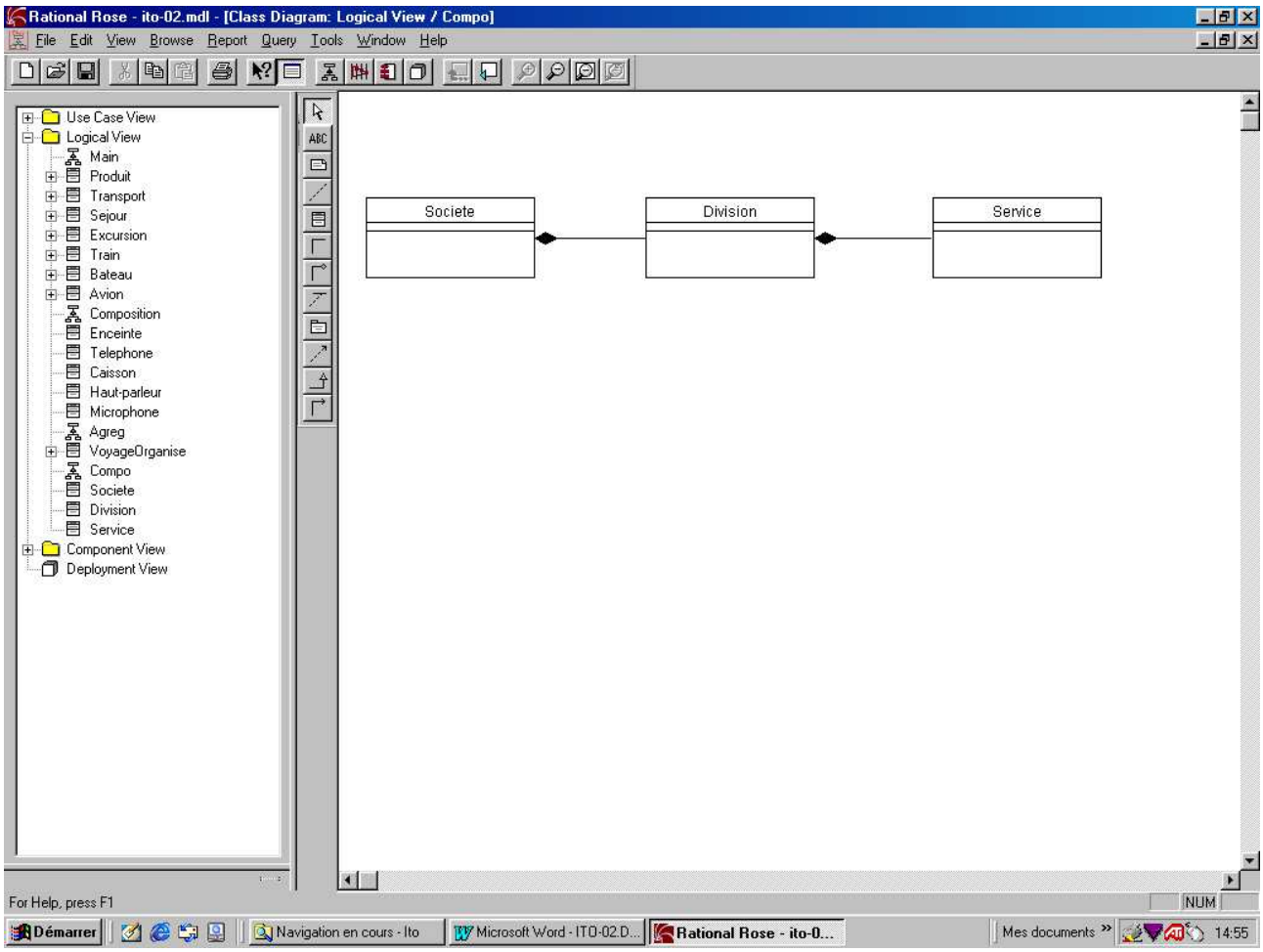
Exemple : Composants électroniques



Le principal intérêt de ce type de lien est l'élimination de la redondance.

Un lien de composition peut être établi entre seulement deux classes :

Exemple : Composition d'une société



Tout lien de composition doit pouvoir se traduire par la sémantique : **"Est constitué de"**

2.2. DIFFERENCE ENTRE AGREGATION ET COMPOSITION.

Composition

En UML, le losange représentant le lien de composition est **noir**. Cela permet de traduire que :

- L'objet agrégé n'existe pas sans l'objet qui l'inclut.
- La relation est de **1** à 0 ou plusieurs.

Exemple : un répertoire est constitué de plusieurs fichiers.

- Le fichier n'appartient qu'à **1** seul répertoire.
- Si l'on supprime le répertoire, on supprime tous les fichiers.

Agrégation

En UML, le losange représentant le lien d'agrégation est **blanc**. Cela permet de traduire que :

- L'objet agrégé peut exister indépendamment de l'objet qui l'inclut.
- La relation est de 0 ou 1 à plusieurs.

Exemple : une pièce est composée de 0 à plusieurs murs.

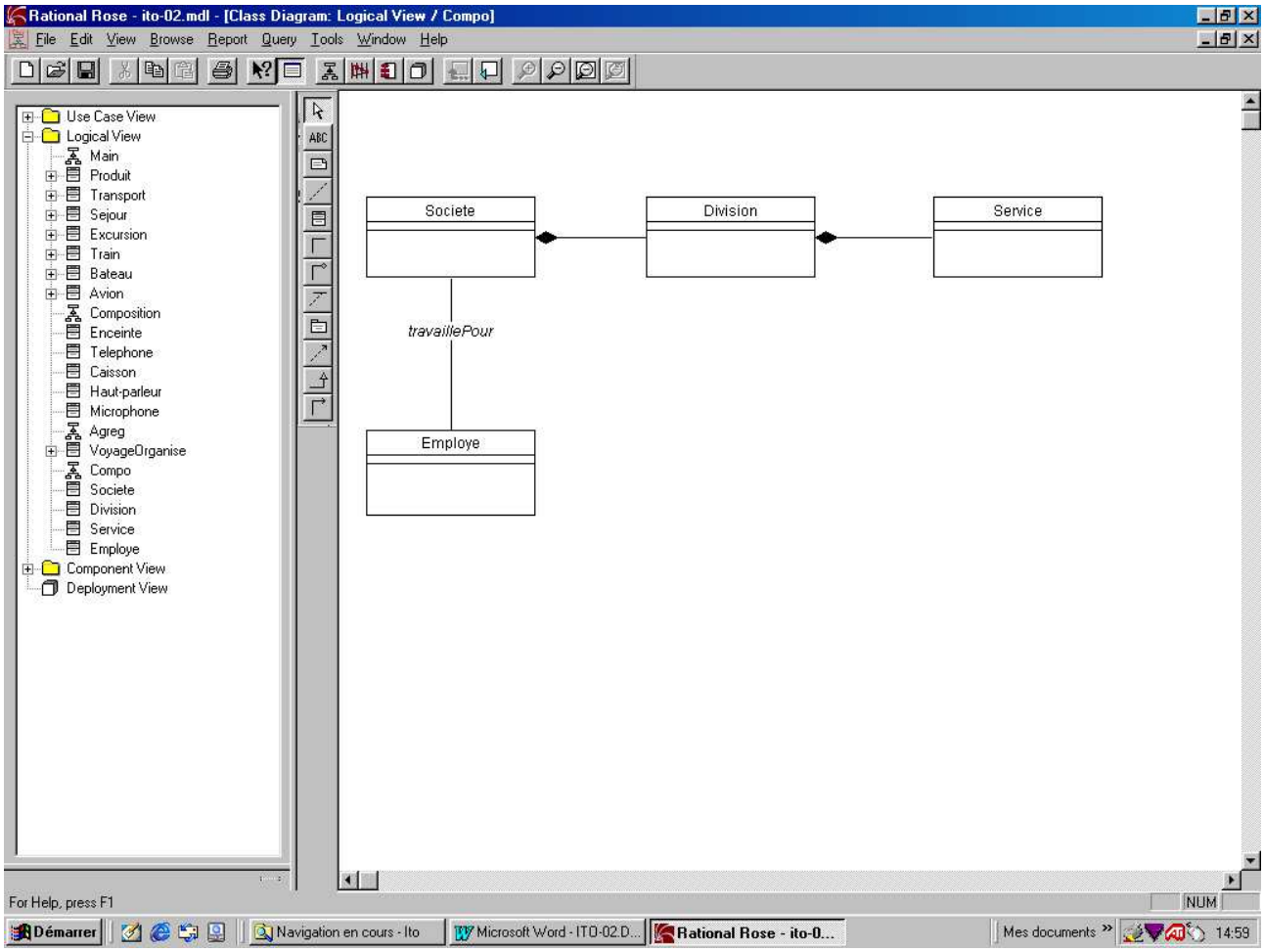
- Le mur mitoyen n'appartient pas qu'à une pièce.
- Si l'on supprime la pièce, on ne supprime pas les murs.

Tout lien d'agrégation doit pouvoir se traduire par la sémantique : **"Est composé de"**

2.3. DIFFERENCE ENTRE AGREGATION ET ASSOCIATION

Le lien d'agrégation est reconnu par les AGL (Atelier de Génie Logiciel) pour la génération de code.

Exemple : Agrégation et association pour une société



Bien que nous ne disposions pas des règles de gestion ayant conduit au modèle ci-dessus, on peut le juger *a priori* correct ; une société est le résultat de l'agrégation de plusieurs divisions, chacune d'entre elles étant le résultat de l'agrégation de plusieurs services. Il semble en revanche peu naturel de tenter d'établir le même type de lien entre une société et ses employés : on utilisera donc ici une association ordinaire.

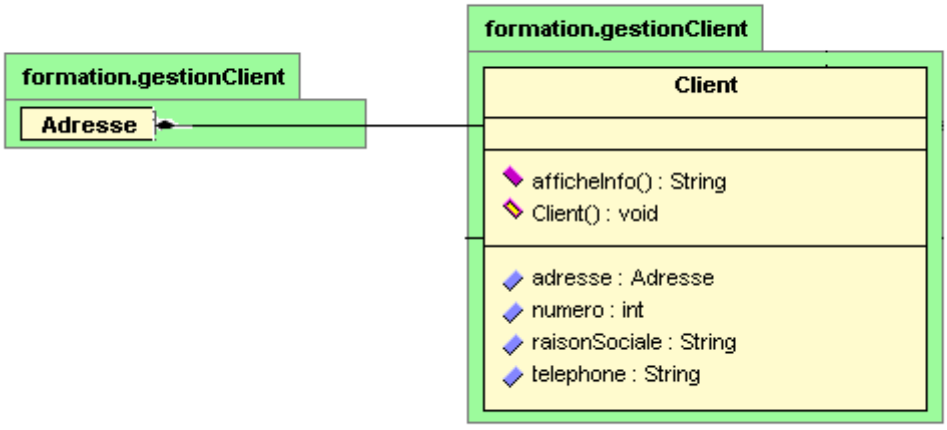
Une telle distinction est contextuelle. D'autres modèles (statistiques sur la population active par exemple), pourraient considérer une société comme n'étant que le résultat de l'agrégation de ses employés.

2.4. DU CONCEPT AGREGATION / COMPOSITION AU CODE

2.4.1. EXPLICATIONS

Dans l'exemple Client précédent, on décide de mémoriser l'adresse du client. On peut donc créer des variables d'instance supplémentaires pour la rue, la ville, le code postal…
On préfère créer une classe Adresse définissant l'ensemble des informations d'une adresse. Ainsi notre classe Client sera composée d'une adresse.

2.4.2. DIAGRAMME UML



2.4.3. CODE JAVA DE LA CLASSE CLIENT

```
package formation.gestionClient ;
public class Client {
        // Déclaration des variables d'instance
        private int numero ;
        private String raisonSociale ;
        private String telephone ;
        private Adresse  adresse;

        // Constructeur : permet d'initialiser les variables d'instance
        public Client (int num) {
                numero = num ;
                adresse = new Adresse() ;
        }
        //Accesseurs
        public Adresse getAdresse () {
                return adresse;
        }
        public void setAdresse(Adresse adr) {
                adresse = adr;
        }
…/… (suite du code précédemment écrit)
}
```

2.4.4. CODE JAVA DE LA CLASSE ADRESSE

```
package formation.gestionClient;

public class Adresse
{
	private int numero ;
	private String rue;
	private String ville;
	private int codePostal;

	// Constructeur
	public Adresse()
	{
	}

	//Accesseurs
	public int getCodePostal() {
		return codePostal;
	}
	public int getNumero() {
		return numero;
	}
	public String getRue() {
		return rue;
	}
	 public String getVille() {
		return ville;
	}
	public void setCodePostal(int code) {
		codePostal = code;
	}
	public void setNumero(int num) {
		numero = num;
	}
	public void setRue(String ru) {
		rue = ru;
	}
	public void setVille(String vil) {
		ville = vil;
	}
}
```

2.4.5. REFERENCE A L'OBJET

Dans la classe Adresse, nous créons une méthode qui permet d'afficher l'adresse, sous forme de chaîne de caractères.

Il faut donc utiliser l'ensemble des méthodes *get*. Cependant, dans la définition de la méthode, l'objet de type Adresse n'existe pas. On utilise donc un pseudo code permettant de référencer l'objet en cours. Ce pseudo code est *this*.

```
package formation.gestionClient;

public class Adresse
{
	private int numero;
	private String rue;
	private String ville;
	private int codePostal;
	// Constructeur
	public Adresse() {}

	public String afficheAdresse() {
		return this.getNumero() + " " + this.getRue()
			+ " " + this.getCodePostal()+ " " + this.getVille();
	}
	// Suite des méthodes get et set
		….
}
```

Par ailleurs, lorsque l'on écrit les setters, on passe en paramètre une variable dont le nom est identique à la variable d'instance.

Etant donné que l'on ne peut pas écrire :

```
public void setRue(String rue) {
	rue = rue;
}
```

On fait référence à la variable d'instance de l'objet en cours :

```
public void setRue(String rue) {
	this.rue = rue;}
```

La classe Adresse s'écrit donc de la manière suivante :

```
package formation.gestionClient;

public class Adresse {
	private int numero ;
	private String rue;
	private String ville;
	private int codePostal;
	// Constructeur
	public Adresse() {
	}
	// Affichage de l'adresse sous forme de chaîne de caractères
	public String afficheAdresse() {
	return this.getNumero() + " " + this.getRue() + " "
		+ this.getCodePostal()+ " " + this.getVille();
	}
	// Définition des getters
	public int getCodePostal() {
		return codePostal;
	}
	public int getNumero() {
		return numero;
	}
	public String getRue() {
		return rue;
	}
	public String getVille() {
		return ville;
	}
	// Définition des setters
	public void setCodePostal(int codePostal) {
		this.codePostal = codePostal;
	}
	public void setNumero(int numero) {
		this.numero = numero;
	}
	public void setRue(String rue) {
		this.rue = rue;
	}
	public void setVille(String ville) {
		this.ville = ville;
	}
}
```

2.4.6. CODE JAVA DU TRAITEMENT MANIPULANT LA CLASSE CLIENT

Le client Test manipule donc les nouvelles méthodes :

```
package formation.gestionClient;

public class TestClient
{
	// Déclaration du constructeur
	TestClient()
	{
	}

	// Programme principal
	public static void main(String args[])
	{
	// Création d'un objet client
	Client client2 = new Client(2000);
	// Modification de la raison sociale de l'objet client2
	client2.setRaisonSociale("Etablissement D");
	// Affichage à la console système du traitement d'affichage du client2
	System.out.println("AFFICHAGE " + client2.afficheInfo());

	// Récupération de l'adresse du client
	Adresse adr = client2.getAdresse();
	// Mise à jour des valeurs
	adr.setNumero(100);
	adr.setRue("rue de la justice");
	adr.setCodePostal(91230);
	adr.setVille("Montgeron");
	// Affichage de l'adresse
	System.out.println("Adresse du client "  + adr.afficheAdresse());
	}
}
```

Résultat obtenu à la console système :

```
AFFICHAGE le client est : 2000  Etablissement D
Adresse du client 100 rue de la justice 91230 Montgeron
```

3. LA GENERALISATION, LA SPECIALISATION

3.1. DEFINITION

Le processus classique de modélisation des données et des traitements conduit à introduire des redondances volontairement ou involontairement.

Ainsi pour un assureur, modéliser un contrat d'assurance automobile et un contrat d'assurance habitation induira une double définition du numéro de contrat, du nom de l'assuré, de son adresse, des informations concernant son état civil...

De même un banquier, pour qui un compte chèque de particulier et un compte épargne logement sont deux entités de gestion différentes, sera vraisemblablement amené à introduire des redondances dans les traitements de mouvement sur les comptes, de consultation du solde,...

Une telle redondance peut paraître anodine au moment du développement des applicatifs et se révèler particulièrement coûteuse lors d'une maintenance. Quelle est la répercussion d'une modification des règles de gestion des mouvements de crédit de comptes bancaires pour le banquier précédent ?

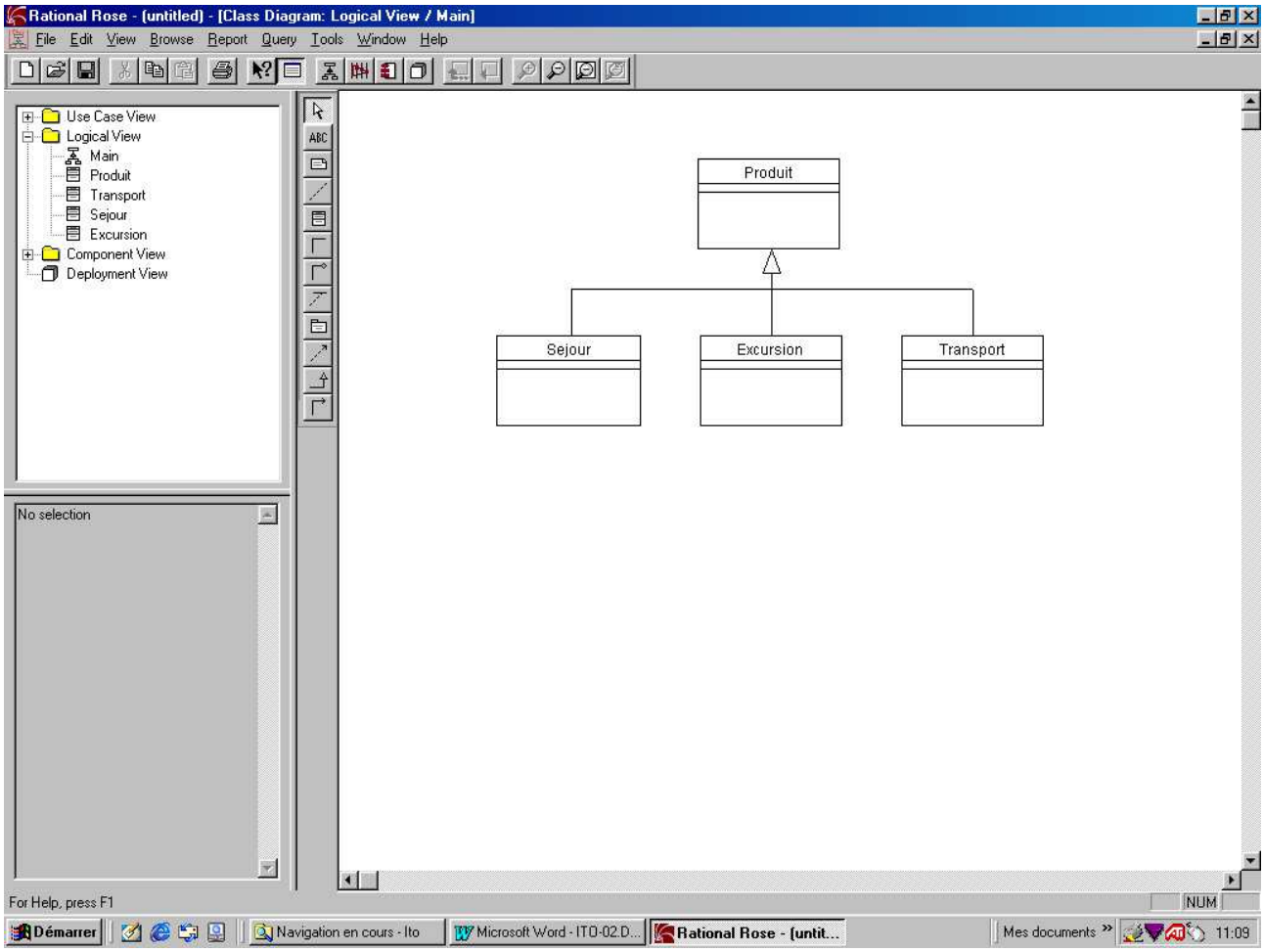
```
sémentique pour savoir s'il s'agit d'un héritage :
ex ==> est ce qu'un compte courant est une sorte de compte?
Si oui alors il s'agit d'un héritage
```

On peut donc admettre que deux classes de gestion, dont la distinction est justifiée, peuvent néanmoins présenter un certain nombre de similitudes, qu'il peut être intéressant de factoriser afin d'éliminer tout risque de redondance et de faciliter les maintenances ultérieures.

Ce processus est connu sous le nom de **généralisation** et se traduit par l'établissement d'un lien hiérarchique entre une classe, dénommée **sur-classe** ou **classe-mère**, et sa **sous-classe** ou **classe-fille**.

Exemple :

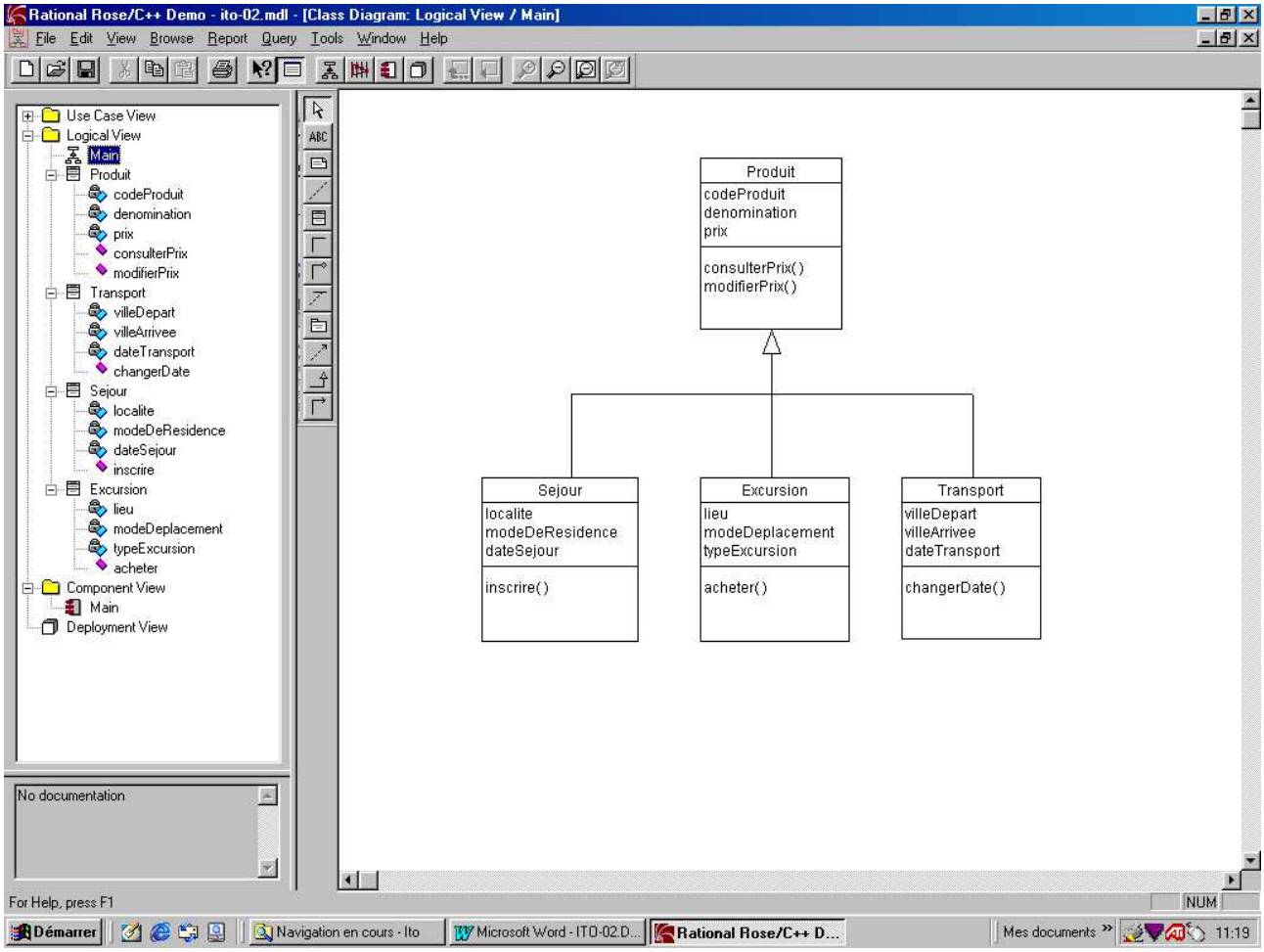
Produits commercialisés par une agence de voyage



Cet exemple introduit la notion de **classe abstraite**, la classe *Produit*, à partir de laquelle aucun objet ne pourra être instancié. Une classe abstraite n'a donc qu'un rôle de fédérateur dans le but d'éliminer la redondance. Nous verrons plus loin qu'une classe-mère n'est pas forcément une classe abstraite.

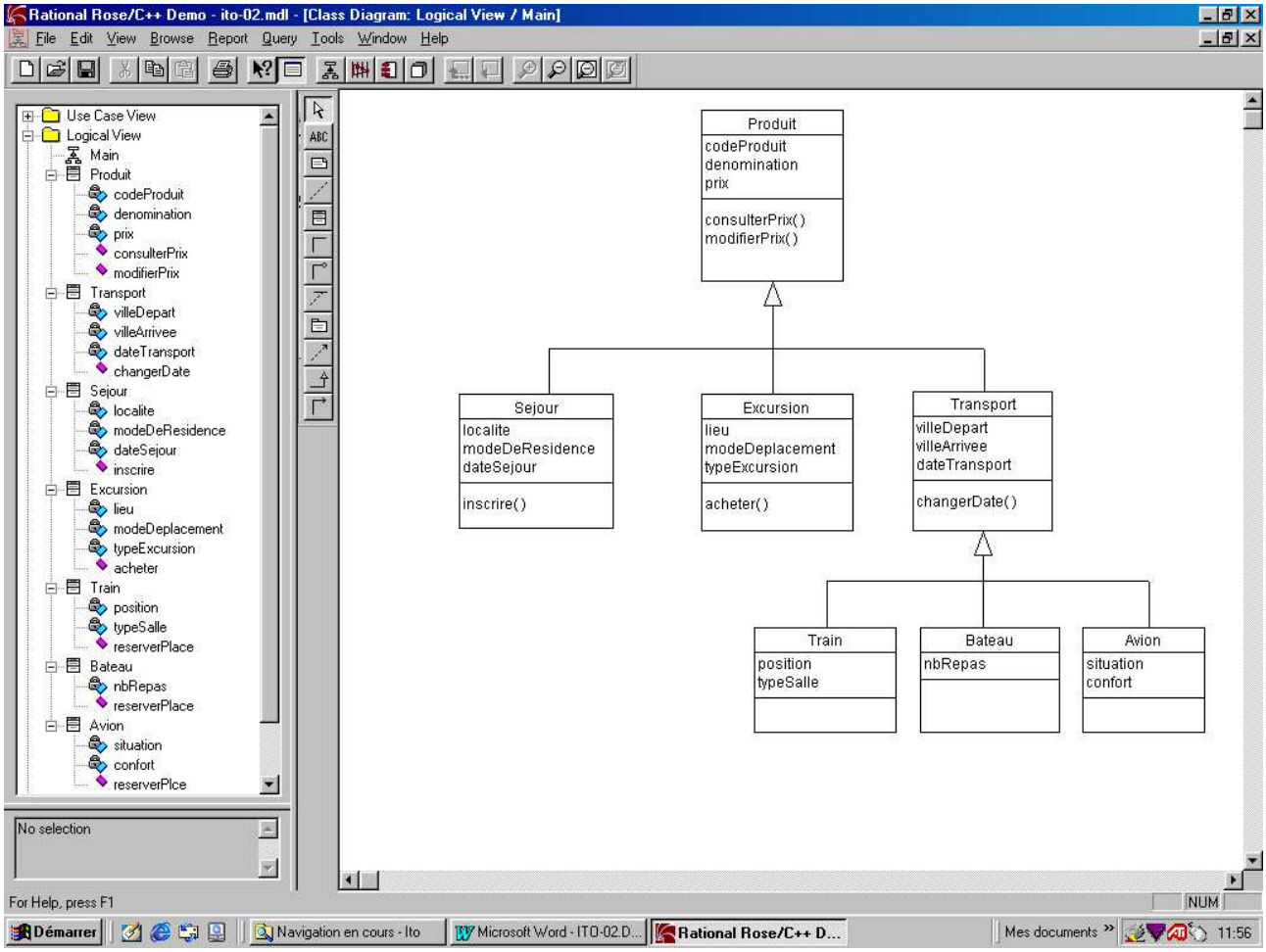
La classe ayant pour but de déclarer la liste des attributs et la liste des méthodes des objets qu'elle caractérise, l'exemple ci-dessus est incomplet. Une classe-mère est une classe au même titre qu'une autre et doit donc elle aussi décrire un état et un comportement.

Exemple :



Le processus inverse de la généralisation est la **spécialisation**. Une classe définie de manière trop globale peut être détaillée si les règles de gestion à implémenter le justifient.

Exemple :



Tout lien de généralisation/spécialisation doit pouvoir se traduire par la sémantique : **"Est une sorte de"**

Dans l'exemple précédent, même si les attributs déclarés au niveau de la classe *Séjour* sont uniquement *Localité*, *Mode de résidence* et *Dates*, tout objet instancié à partir de cette classe peut accéder aux attributs *Code*, *Dénomination*, *Prix*, *Localité*, *Mode de résidence* et *Dates*.

Autrement dit, **l'état d'un objet est décrit à partir de la valeur des attributs de la classe à partir de laquelle il a été instancié et de toutes ses sur-classes**.

Ainsi l'état d'un objet instancié à partir de la classe *Bateau* est décrit par les valeurs des attributs *Code*, *Dénomination*, *Prix*, *Ville départ*, *Ville arrivée*, *Date*, *Nb de repas*.

Il en va de même pour les méthodes. Un objet instancié à partir de la classe *Excursion* peut accéder aux méthodes *Consulter prix()*, *Modifier prix()*, et *Acheter()*.

Autrement dit, **le comportement d'un objet est constitué des comportements de la classe à partir de laquelle il a été instancié ainsi que de ceux de toutes ses sur-classes**.

Cette capacité d'une classe à bénéficier de la description d'une classe-mère se nomme l'**héritage**.

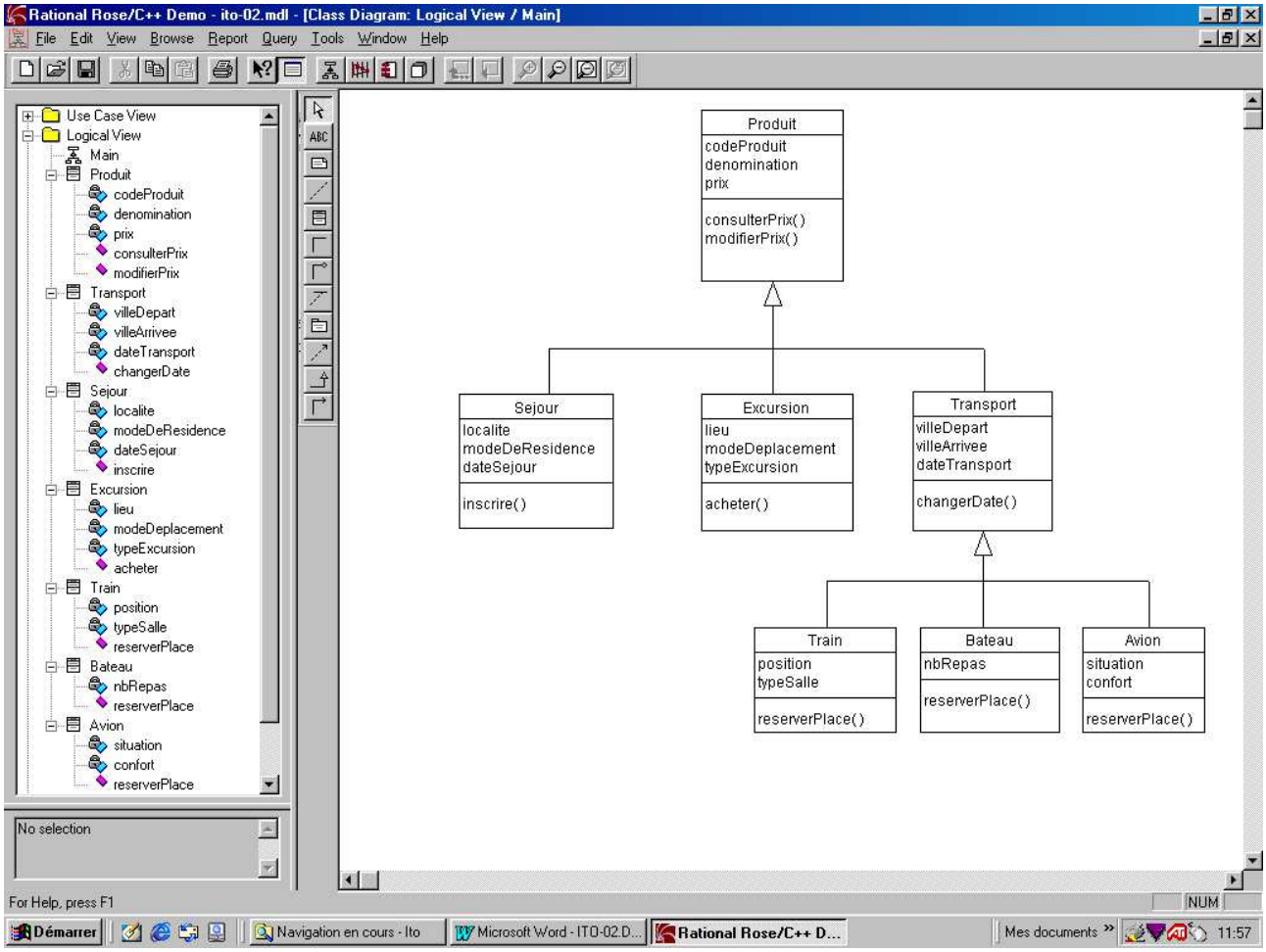
3.2. LE POLYMORPHISME

Le modèle de l'exemple précédent ne permet pas de réserver une place dans un moyen de transport, la méthode adéquate n'ayant pas été déclarée.

Coder la méthode *Réserver place()* dans la classe *Transport* n'est pas satisfaisant puisque celle-ci n'a pas accès aux attributs spécialisés tels que *Couloir/fenêtre*, *Nb de repas*,... De plus, réserver une place dans un train suppose de se mettre en relation avec la SNCF alors que la réservation d'un billet d'avion peut d'abord nécessiter l'entrée dans une application de choix de la compagnie aérienne.

La méthode *Réserver place()*, même si elle désigne globalement une réalité similaire dans les trois cas de l'exemple, recouvre en pratique trois types d'action (et donc de codage) complètement distincts.

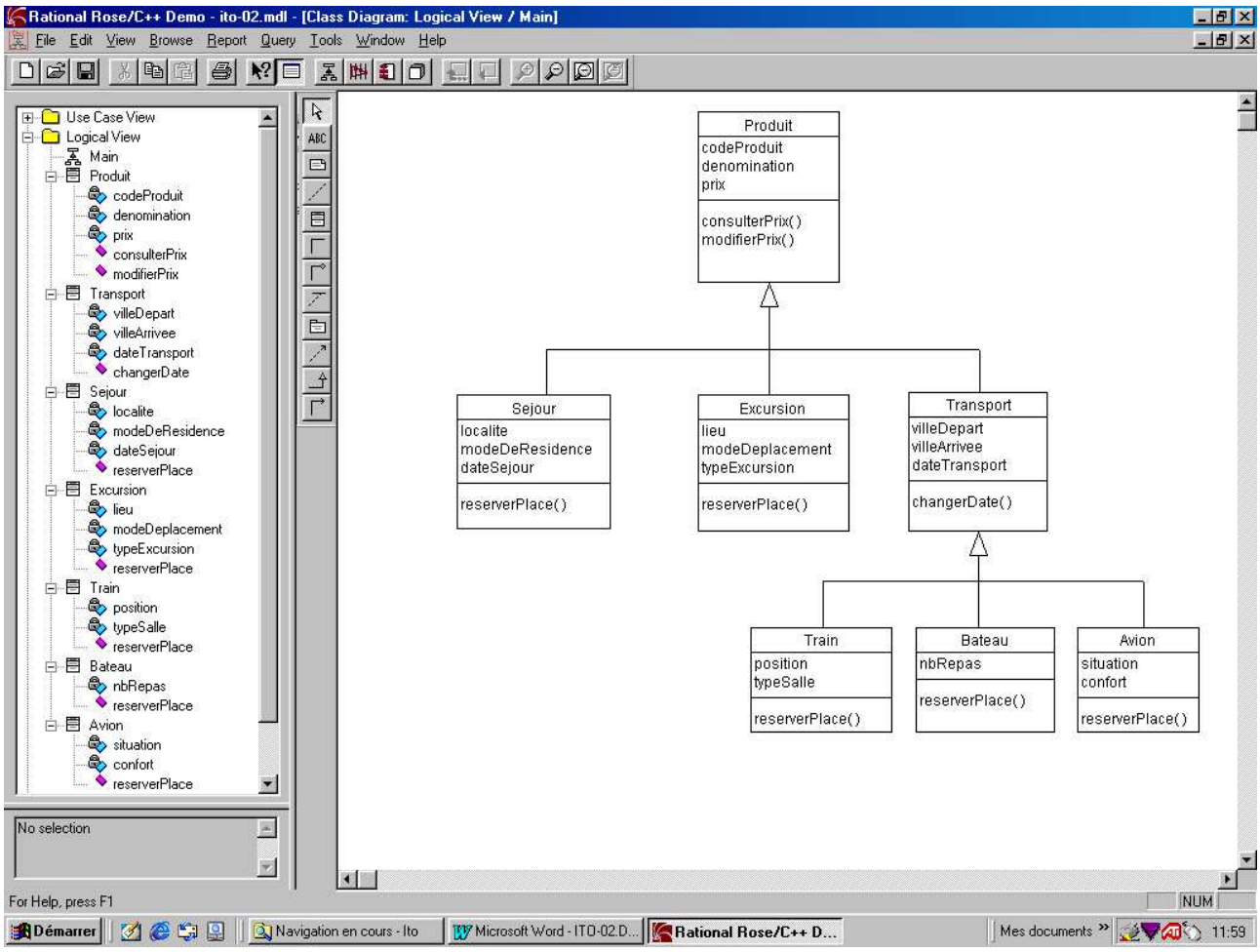
La solution retenue est donc celle-ci :



Cette possibilité est particulièrement puissante. Lorsque le développeur d'applications aura besoin d'activer la méthode qui réserve une place dans un transport, il pourra ignorer totalement le type de l'objet pour lequel il active la méthode, le routage étant résolu par l'implémentation.

On appelle **polymorphisme**, la caractéristique d'une méthode à recouvrir des réalités différentes de manière transparente pour l'utilisateur des classes.

Le modèle précédent peut d'ailleurs être rendu encore plus efficace en rebaptisant la méthode *Inscrire()* de la classe *Séjour* et la méthode *Acheter()* de la classe *Excursion* :



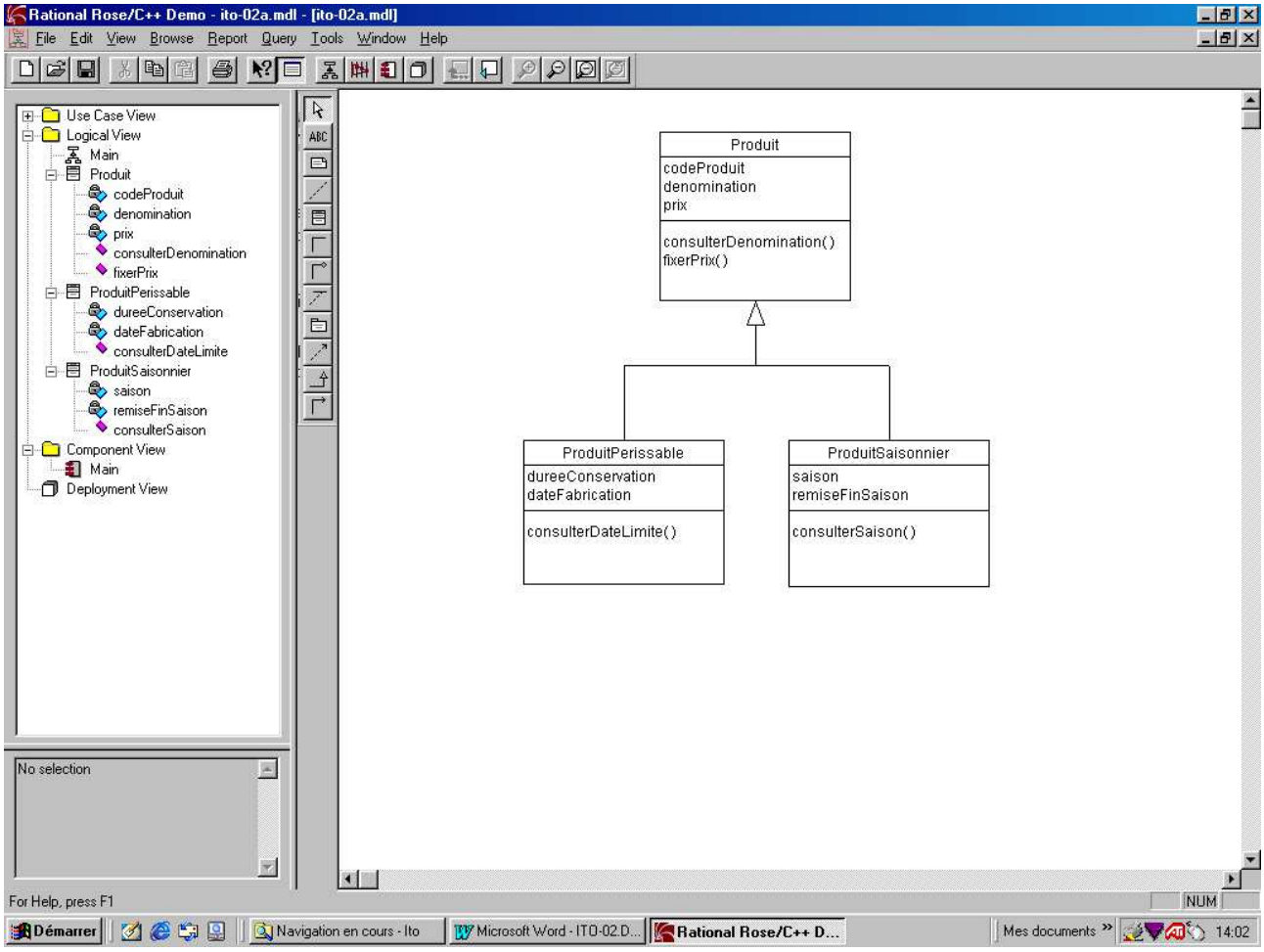
3.3. LA REDEFINITION

Exemple : Grossiste multi-domaines

Un grossiste gère trois types de produits :

- des produits périssables (produits laitiers, viandes...),
- des produits saisonniers (bonnets, écharpes, parapluies...),
- des produits ni périssables, ni saisonniers (fournitures de bureau...).

L'étude des règles de gestion permet de modéliser les classes *Produit saisonnier* et *Produit périssable* comme étant sous-classes de la classe *Produit* :

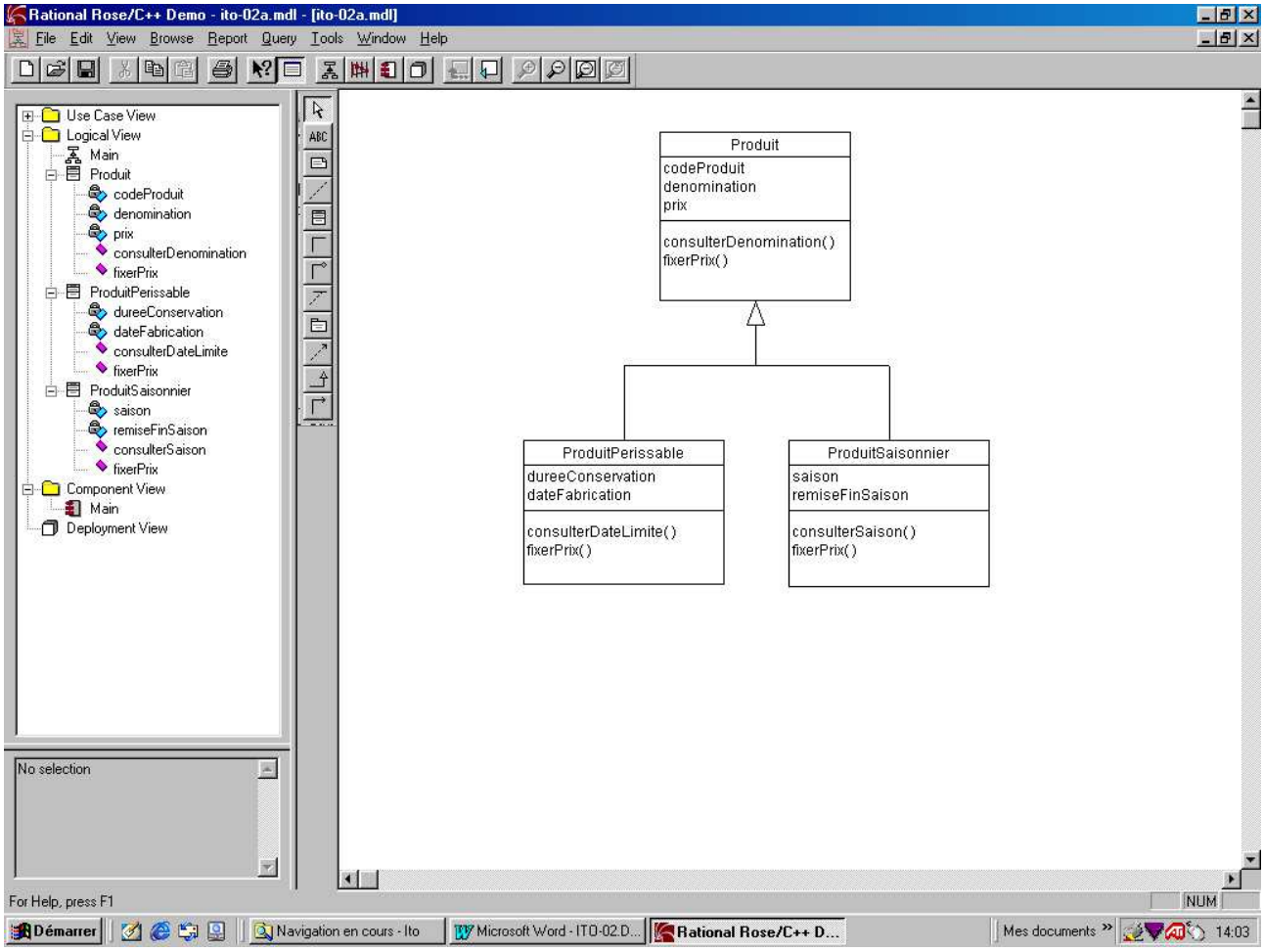


Remarque :

Dans cet exemple, contrairement à celui de l'agence de voyages, la classe *Produit* n'est pas une classe abstraite. Elle représente la quasi-totalité des produits gérés par le grossiste, hormis les produits périssables et les produits saisonniers.

Le grossiste peut souhaiter affiner les règles de gestion régissant le prix d'un produit en effectuant une remise sur les produits périssables qui atteignent leur date limite de vente et en soldant les produits saisonniers proportionnellement à l'approche de la fin de saison.

Une fois encore, c'est le polymorphisme qui va permettre de répondre à ce nouveau besoin, la méthode *Fixer prix()* ayant des comportements différents en fonction de l'objet sur lequel elle s'applique, et ce de manière complètement transparente pour le développeur d'applications :



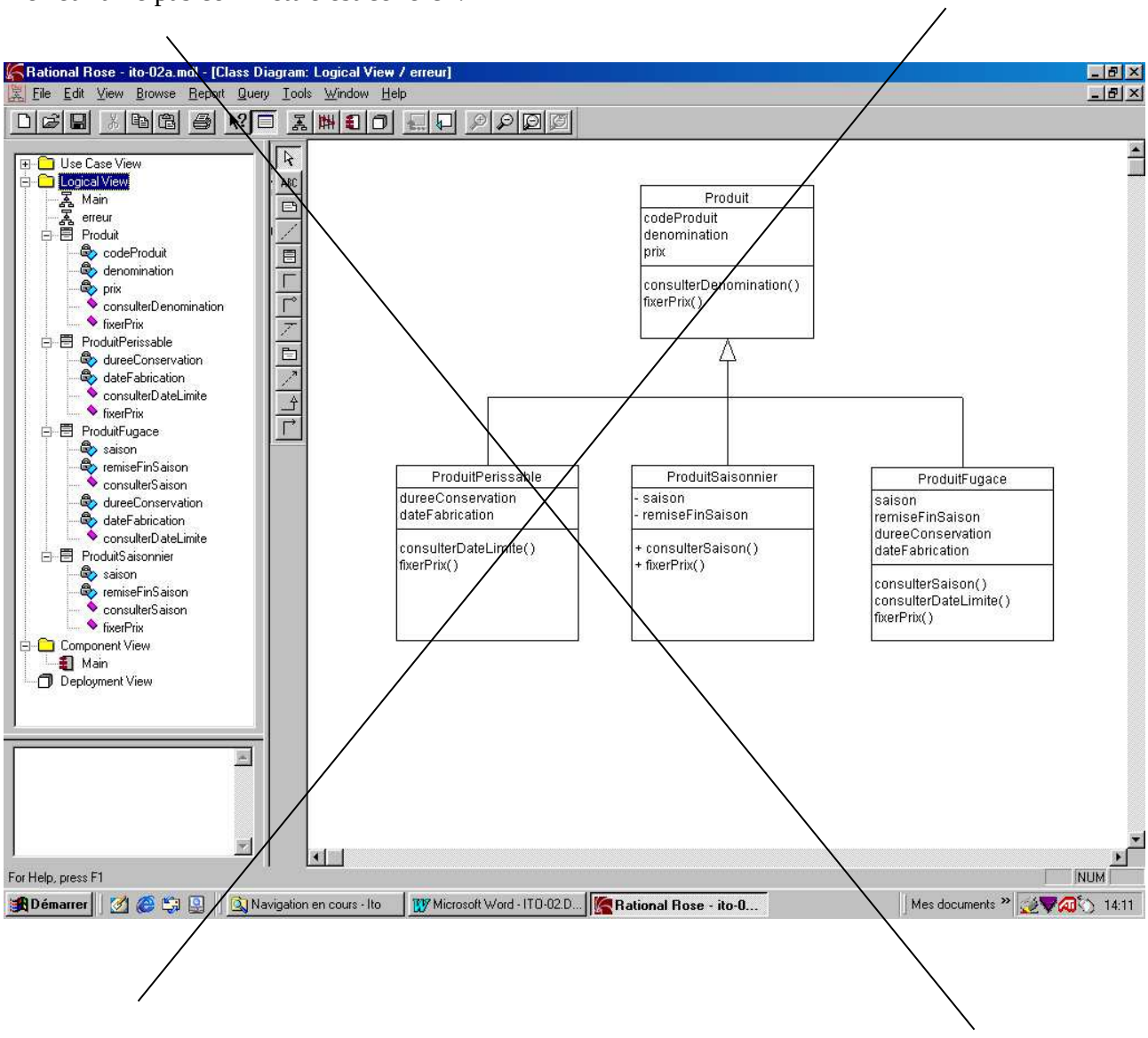
Le mécanisme d'héritage s'effectue naturellement de haut en bas. La classe *Produit périssable* hérite des attributs *Code*, *Dénomination*, *Prix* et déclare les attributs *Durée conservation* et *Date fabrication*. De même, elle hérite des méthodes *Consulter dénom.()* et *Fixer prix()* de la classe *Produit*, mais cette dernière est annulée et remplacée par la méthode de même nom déclarée dans la classe même.

On dit qu'il y a **redéfinition** d'une méthode lorsqu'une méthode héritée d'une classe mère est annulée et remplacée par une autre méthode d'une sous-classe.

3.4. L'HERITAGE MULTIPLE

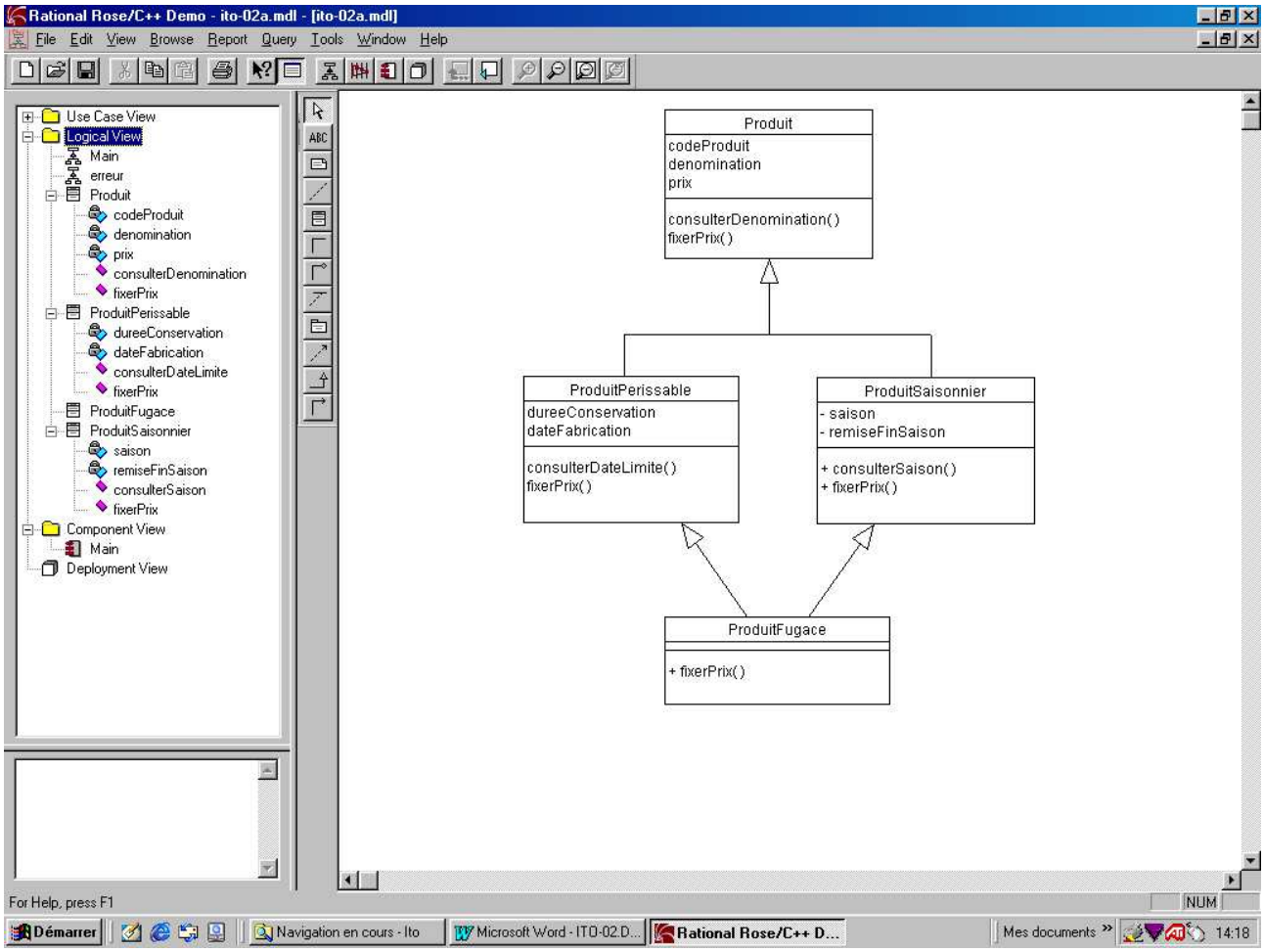
Une opportunité économique incite le grossiste à se lancer dans les huîtres. Or une huître est un produit qui est à la fois saisonnier et périssable...

L'erreur à ne pas commettre est celle-ci :



Une telle modélisation, même si elle résout le problème de la méthode *Fixer prix()*, génère une redondance inacceptable.

Le modèle objet autorise l'**héritage multiple**, capacité dont dispose une sous-classe d'hériter de plusieurs classes mères **de même niveau** :



La méthode *Fixer prix()* peut être soit surchargée au niveau de la classe *Produit fugace*, soit choisie statiquement ou dynamiquement entre les méthodes de même nom des classes *Produit périssable* et *Produit saisonnier*.

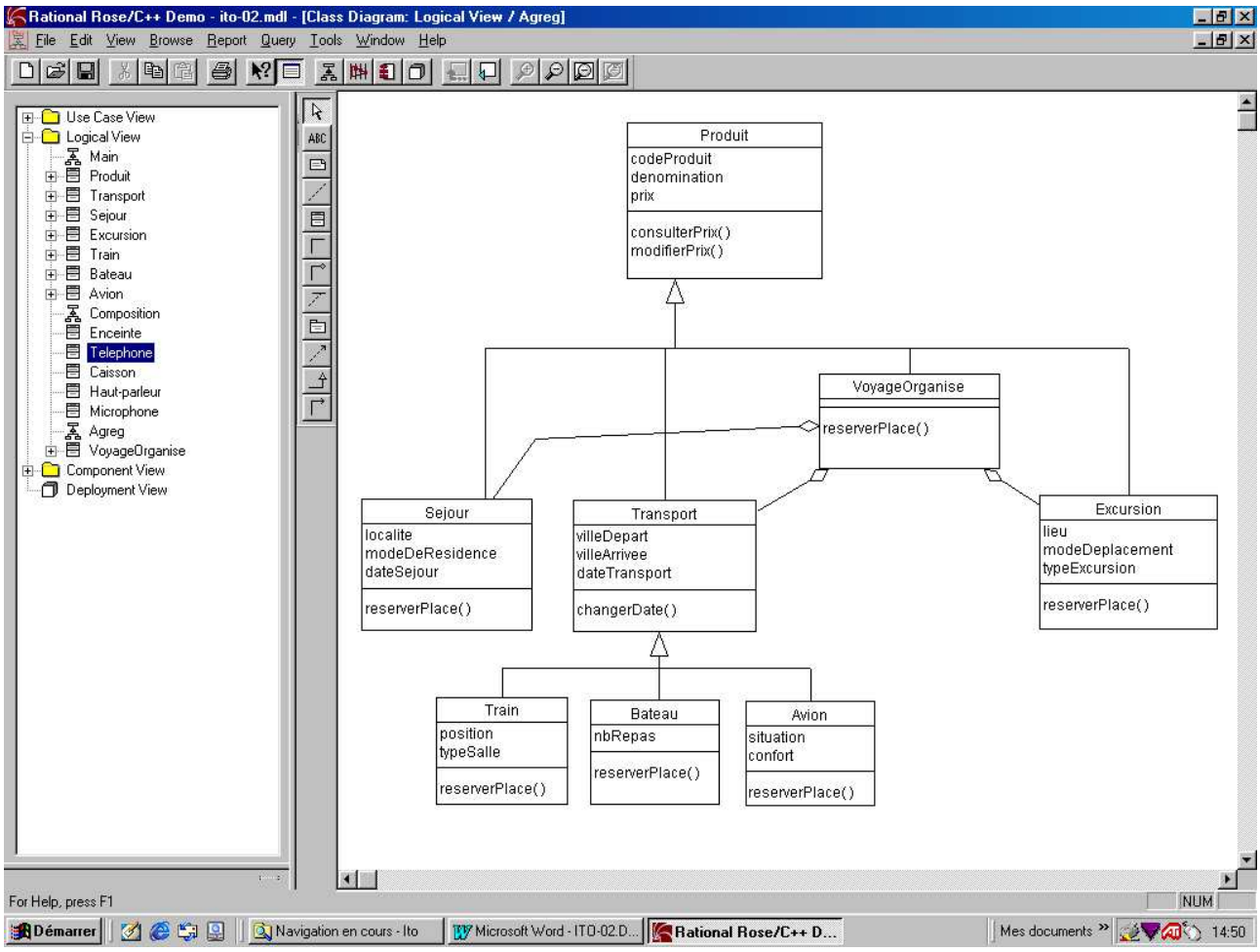
Attention !...

L'héritage multiple est en fait un lien assez rare, qu'il ne faut pas confondre avec le lien de composition.

4. RECAPITULATIF DES LIENS

Exemple : Agence de voyage

En plus de commercialiser des séjours, des excursions et des transports, l'agence de voyage propose à ses clients des voyages organisés (un transport + un séjour + une ou plusieurs excursions) :



Tout lien d'agrégation doit pouvoir se traduire par la sémantique : **"Est composé de"**

Tout lien de généralisation/spécialisation doit pouvoir se traduire par la sémantique : **"Est une sorte de"**

Il n'y a pas, à proprement parler, de mécanisme d'héritage dans une relation d'agrégation. Toutefois, la classe *Voyage organisé* étant composée logiquement des classes *Séjour*, *Transport* et *Excursion*, elle inclut tous les attributs et toutes les méthodes de ces classes.

En cas de conflit, par exemple sur la méthode r*éserver place()* ci-dessus, les traditionnelles règles de masquage des langages de programmation s'appliquent, chacune des méthodes restant toutefois accessible à condition de la préfixer correctement.

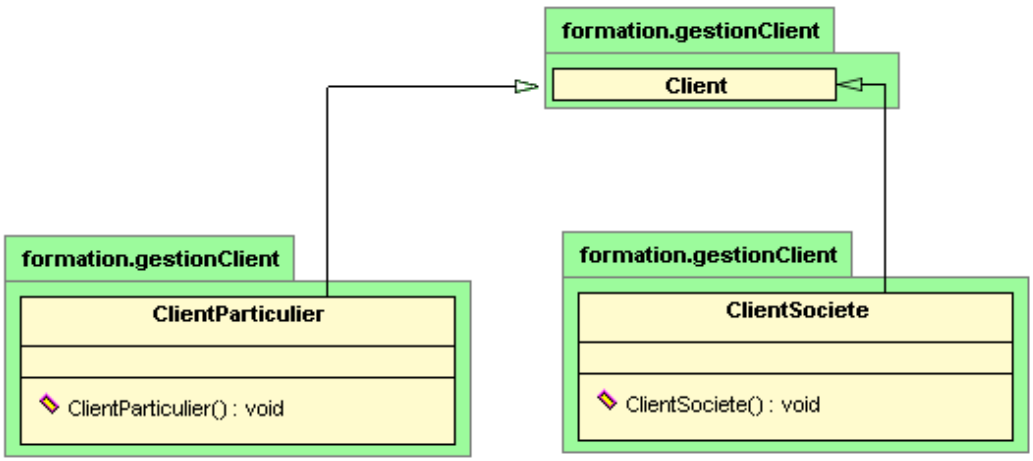
5. DES CONCEPTS AU CODE

5.1. HERITAGE SIMPLE

5.1.1. EXPLICATIONS

Dans l'exemple Client précédent, on décide de gérer des clients particuliers, afin de connaître leurs enfants et de gérer des clients « société » pour connaître le nombre d'employés. Ces deux types de clients sont des clients, ils vont donc hériter de la classe Client.

5.1.2. DIAGRAMME UML



5.1.3. CODE JAVA

```
package formation.gestionClient;

public class ClientSociete extends Client
{
	// Définition de la variable d'instance spécialisant l'information

	private int nombreEmployes;

	// Définition des getters et des setters

	public int getNombreEmployes()
	{
		return nombreEmployes;
	}

	public void setNombreEmployes(int nombreEmployes)
	{
		this.nombreEmployes = nombreEmployes;
	}
}
```

5.2. REDÉFINITION

5.2.1. EXPLICATIONS

On désire redéfinir la méthode « AfficheInfo() » de la classe mère. Dans la classe fille, la méthode affichera le nombre de salariés.

5.2.2. CODES JAVA

5.2.2.1. Code de la classe ClientSociete

```
package formation.gestionClient;

public class ClientSociete extends Client
{
	private int nombreEmployes;

	// Constructeur
	public ClientSociete(int num)
	{
		// code du constructeur (voir plus loin..)
		…

	}

	public int getNombreEmployes() {
		return nombreEmployes;
	}

	public void setNombreEmployes(int nombreEmployes) {
		this.nombreEmployes = nombreEmployes;
	}

	// Envoi d'une chaine de caractères affichant le nombre de salariés.
	public String afficheInfo()
	{
		return "Il y a : " + this.getNombreEmployes() + " employés";
	}
}
```

5.2.2.2. Code du client Test

```
package formation.gestionClient;

public class TestClientSociete
{
	public TestClientSocieté()
	{
	}

	public static void main(String[] args)
	{
		ClientSociete client3 = new ClientSociete(3000);
		client3.setRaisonSociale("Societe Durand");
		client3.setNombreEmployes(30);

		System.out.println(client3.afficheInfo());
	}

}
```

Résultat obtenu à la console système :

```
Il y a : 30 employés
```

5.2.3. REFERENCE A L'OBJET EN COURS

5.2.3.1. Dans une méthode

En fait, on souhaite récupérer le code existant à la définition de la classe mère et ajouter le nombre d'employés. Pour eviter la réécriture de code, on fait référence au code de la classe mère en spécifiant que l'on applique la méthode de la classe mère par le pseudo code « *super* ».

```
…/…

// Envoi d'une chaine de caractères affichant le nombre de salariés en plus des informations
// données dans la classe mère….

public String afficheInfo()
{
        String retour;
        retour = super.afficheInfo();
        return retour + " Il y a : " + this.getNombreEmployes() + " employés";
}

…/…
```

Le code du client test reste inchangé mais lorsque l'on exécute le test on obtient :

```
le client est : 3000  Societe Durand Il y a : 30 employés
```

5.2.3.2. Dans le constructeur

On n'hérite pas des constructeurs, cependant on veut utiliser le constructeur de la classe mère. L'appel d'un constructeur est particulier. On fait référence au constructeur de la classe mère dans le constructeur de la classe fille en utilisant aussi le pseudo code super.

```
package formation.gestionClient;

public class ClientSociete extends Client
{
	private int nombreEmployes;

	// Constructeur
	public ClientSociete(int num)
	{
		super(num) ;
	}

	public int getNombreEmployes()
	{
		return nombreEmployes;
	}

	public void setNombreEmployes(int nombreEmployes)
	{
		this.nombreEmployes = nombreEmployes;
	}

	// Envoi d'une chaine de caractères affichant le nombre de salariés.
	public String afficheInfo()
	{
		return "Il y a : " + this.getNombreEmployes() + " employés";
	}
}
```

5.3. POLYMORPHISME

5.3.1. EXPLICATIONS

La redéfinition est un polymorphisme.

On enregistre dans un tableau de type Client, des clients, des sociétés, des particuliers.

On effectue une boucle afin d'afficher les informations du client quelqu'il soit.

On va donc avoir un polymorphisme, car la méthode afficheInfo() va s'appliquer de manière différente en fonction du type de client.

5.3.2. CODE JAVA DU CLIENT TEST

```
package formation.gestionClient;
public class TestDifferentClient
{
	public TestDifferentClient() {
	}
	public static void main(String[] args)
	{
		// Création d'un objet client
		Client client1 = new Client(1000);
		// Modification de la raison sociale de l'objet client1
		client1.setRaisonSociale("Client A");
		// Création d'un objet client
		Client client2 = new Client(2000);
		// Modification de la raison sociale de l'objet client2
		client2.setRaisonSociale("Client B");
		// Création d'un objet client société
		ClientSociete client3 = new ClientSociete(3000);
		client3.setRaisonSociale("Societe Durand");
		client3.setNombreEmployes(30);
		// Création d'un objet client particulier
		ClientParticulier client4 = new ClientParticulier(4000);
		client4.setRaisonSociale("Particulier Durand");
		client4.setNombreEnfants(2);

		// Enregistrement des clients crées dans un tableau
		Client clients[] = new Client[4];
		clients[0] = client1;
		clients[1] = client2;
		clients[2] = client3;
		clients[3] = client4;

		// Polymorphisme
		for (int i = 0; i < clients.length; i++)
		{	System.out.println("AFFICHAGE élement " + i + " " +
clients[i].afficheInfo()); }
	}
}
```

Résultat obtenu à la console système :

```
AFFICHAGE élement 0 le client est : 1000  Client A
AFFICHAGE élement 1 le client est : 2000  Client B
AFFICHAGE élement 2 le client est : 3000  Societe Durand Il y a : 30 employés
AFFICHAGE élement 3 le client est : 4000  Particulier Durand Il a : 2 enfants
```

5.4. HERITAGE MULTIPLE : INTERFACES

5.4.1. INTERET

En Java, l'héritage multiple n'existe pas. Les interfaces sont la solution pour cette représentation.

Une **interface** est similaire à une **classe abstraite**, c'est donc juste une déclaration de **méthodes abstraites** et de **variables statiques** et **finales** (des constantes), pas une définition.

Une interface n'est donc pas instanciable.

Une interface comme une classe définit un type, utilisable ensuite comme un type de données.

5.4.2. DECLARATION

La déclaration s'effectue par le mot clé « **interface** » et l'on indique le nom de l'interface

Par convention un nom d'interface débute en majuscule, et termine par « able ». Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.

Devant le mot clé, on peut trouver des modificateurs. Derrière on note l'héritage par le mot clé « **extends »**. Une interface peut hériter de plusieurs interfaces (séparées par des virgules).

```
interface Interfacable {…}

// Exemple avec héritage
interface InterfacableA extends Interfacable , InterfacableC {…}

// Exemple avec modificateurs
public interface InterfacableC{…}

Syntaxe
[Modificateurs] interface Nom extends Inteface1, Interface2 {…}
```

5.4.3. MODIFICATEURS

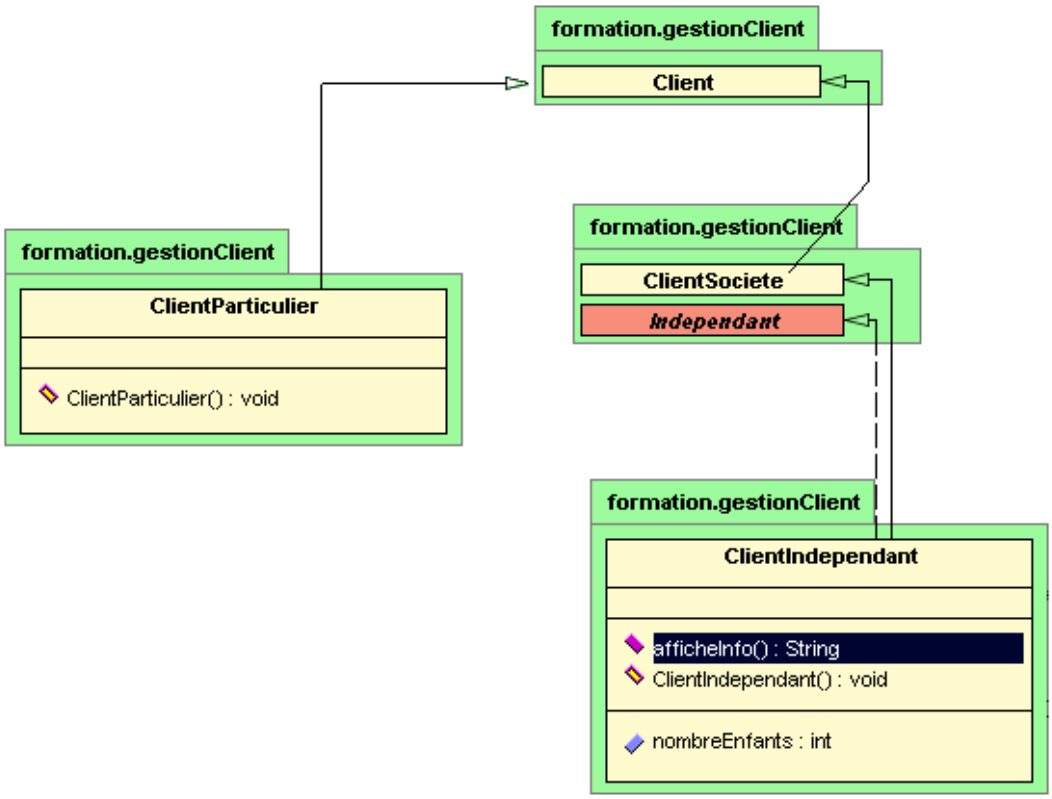
Par défaut une interface n'est visible que pour les classes définies à l'intérieur de son package.

- public : l'interface est visible pour les classes des autres packages.
- abstract : l'interface est par définition abstraite.

5.4.4. EXEMPLES

Nous désirons gérer des indépendants, il s'agit de client société et de client particulier. L'héritage multiple n'existant pas en java, on va donc hériter de la classe ClientSociete et se créer une interface Independant que l'on implémentera dans la classe ClientIndependant. Cette interface contient des déclarations de afficheInfo(), getNombreEnfants, setNombreEnfants.

5.4.4.1. Diagramme UML



5.4.4.2. Code java de l'interface

```
package formation.gestionClient;

public interface Independant
{
        public String afficheInfo();
        public int getNombreEnfants();
        public void setNombreEnfants(int nombreEnfants);
}
```

5.4.5. UTILISATION D'UNE INTERFACE DANS UNE CLASSE

Une classe qui implémente une ou plusieurs interfaces s'engage à définir toutes les méthodes déclarées dans les interfaces.

Une classe qui n'implémente pas l'ensemble des méthodes de l'interface devient abstraite.

5.4.5.1. Code java

```
package formation.gestionClient;
public class ClientIndependant extends ClientSociete implements Independant
{       private int nombreEnfants;
        public ClientIndependant(int num) {super(num); }
        // Envoi d'une chaine de caractères affichant le nombre de salariés.
        public String afficheInfo() {
                String retour;
                retour = super.afficheInfo() + " Affichage info supplementaires " +
                        this.getNombreEnfants() ;
                if (this.getNombreEnfants() > 1)      {return  retour + " enfants";}
                else                                  {return  retour +  " enfant";}  }
        public int getNombreEnfants() { return nombreEnfants;}
        public void setNombreEnfants(int nombreEnfants)  {this.nombreEnfants =
nombreEnfants;}
}
```

5.4.5.2. Code du client test

Dans notre client test précédent on crée une instance de ClientIndependant, on l'ajoute dans notre tableau de client pour afficher les informations.

```
package formation.gestionClient;
public class TestDifferentClient
{
        public TestDifferentClient() {}
        public static void main(String[] args)
        {
        // Création d'un objet client
        Client client1 = new Client(1000);
        // Modification de la raison sociale de l'objet client1
        client1.setRaisonSociale("Client A");
        // Création d'un objet client societe
        ClientSociete client3 = new ClientSociete(3000);
        client3.setRaisonSociale("Societe Durand");
        client3.setNombreEmployes(30);
        // Création d'un objet client particulier
        ClientParticulier client4 = new ClientParticulier(4000);
        client4.setRaisonSociale("Particulier Durand");
        client4.setNombreEnfants(2);
        // Création d'un objet client indépendant
        ClientIndependant client5 = new ClientIndependant(5000);
        client5.setRaisonSociale("Indépendant Martin");
        client5.setNombreEnfants(0);
        client5.setNombreEmployes(5);
        // Enregistrement des clients crées dans un tableau
        Client clients[] = new Client[4];
        clients[0] = client1;
        clients[1] = client3;
        clients[2] = client4;
        clients[3] = client5;
        // Polymorphisme
        for (int i = 0; i < clients.length; i++)
        { System.out.println("AFFICHAGE élement " + i + " " + clients[i].afficheInfo()); }
        }
}
```

Résultat obtenu à la console système :

```
AFFICHAGE élement 0 le client est : 1000  Client A
AFFICHAGE élement 1 le client est : 2000  Client B
AFFICHAGE élement 2 le client est : 3000  Societe Durand Il y a : 30 employés
AFFICHAGE élement 3 le client est : 4000  Particulier Durand Il a : 2 enfants
AFFICHAGE élement 4 le client est : 5000  Indépendant Martin Il y a : 1 employé Affichage info supplementaires 0 enfant
```

5.4.6. UTILISATION D'UNE INTERFACE COMME TYPE DE BASE

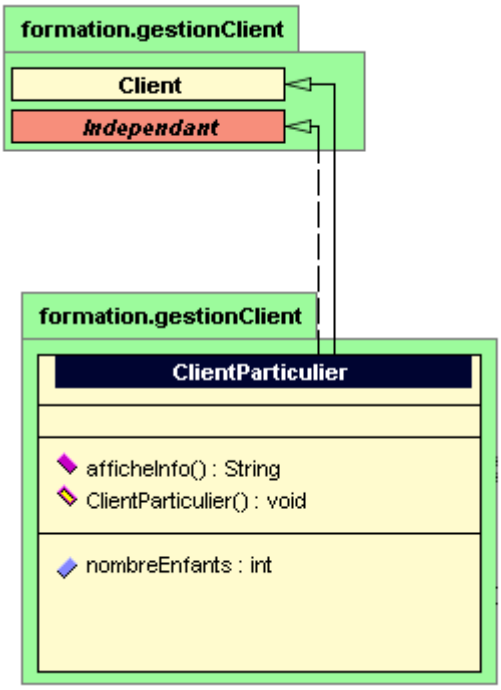
Il est possible de déclarer des objets de type interface.

Ces objets ne sont pas instanciables mais ils référenceront un objet qui implémente l'interface et donc qui comprendra les méthodes.

Ceci est pratique pour gérer des objets de classes différentes, mais qui réagissent identiquement.

On modifie donc la classe ClientParticulier afin qu'elle implémente l'interface Independant.

5.4.6.1. Diagramme UML



5.4.6.2. Code java

```
package formation.gestionClient;

public class ClientParticulier extends Client implements Independant
{
	private int nombreEnfants;

	public ClientParticulier(int num)
	{
		super(num);
	}

	// Envoi d'une chaine de caractères affichant le nombre d'enfants.
	public String afficheInfo()
	{
		String retour;
		retour = super.afficheInfo() + " Il a : " + this.getNombreEnfants() ;
		if (this.getNombreEnfants() > 1)
		{return  retour + " enfants";}
		Else
		{return  retour +  " enfant";}
	}

	public int getNombreEnfants()
	{
		return nombreEnfants;
	}

	public void setNombreEnfants(int nombreEnfants)
	{
		this.nombreEnfants = nombreEnfants;
	}

}
```

5.4.6.3. Interface comme type de base

On crée un tableau de type Independant, dans lequel, on place des objets de types ClientParticulier et ClientIndependant.

```
package formation.gestionClient;

public class TestEnfants
{
	public TestEnfants() {}

	public static void main(String[] args)
	{
		// Création d'un objet client particulier
		ClientParticulier client4 = new ClientParticulier(4000);
		client4.setRaisonSociale("Particulier Durand");
		client4.setNombreEnfants(2);
		// Création d'un objet client indépendant
		ClientIndependant client5 = new ClientIndependant(5000);
		client5.setRaisonSociale("Indépendant Martin");
		client5.setNombreEnfants(0);
		client5.setNombreEmployes(5);

		// Enregistrement des clients crées dans un tableau

		Independant clients[] = new Independant[2];
		clients[0] = client4;
		clients[1] = client5;

		// Polymorphisme
		for (int i = 0; i < clients.length; i++)
		{
			System.out.println("AFFICHAGE élement " + i +
					" " clients[i].getNombreEnfants());
		}
	}
}
```

Résultat obtenu à la console système :

```
AFFICHAGE élement 0 2
AFFICHAGE élement 1 0
```