# TP 6

Thème :

- *Pattern Décorateur*
- le chapitre 3
  extrait de Head First Design Patterns
- créer une classe "ensemble d'entiers"
- et pouvoir vérifier à l'exécution que ses méthodes "fonctionnent" correctement.

---


## Les boissons

### Question 1.1)

#### a) Présentation

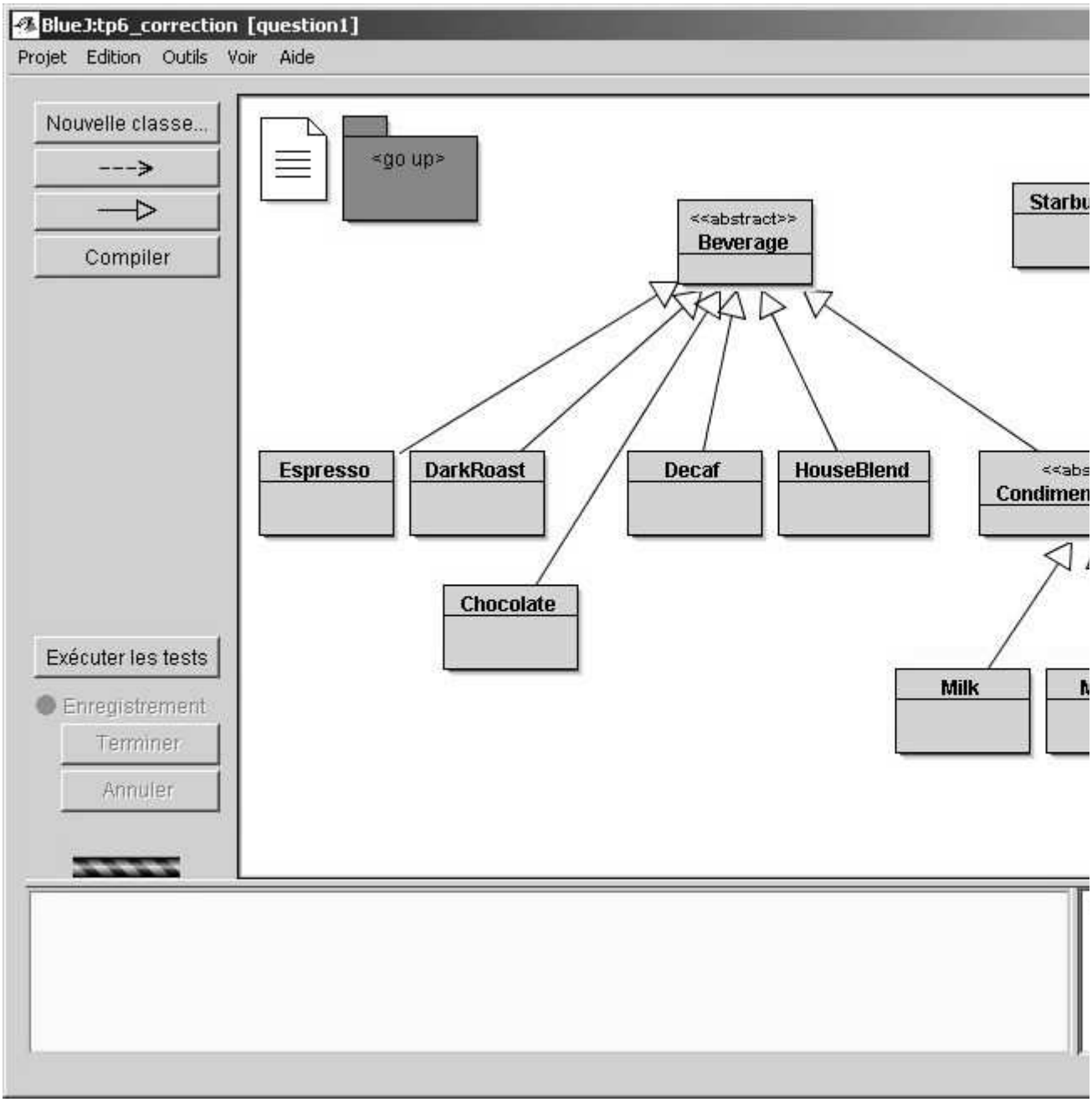
L'exemple présenté dans le chapitre 3 extrait de Head First Design Patterns, décrit une boisson et des compléments possibles. Le pattern décorateur est utilisé afin, de "décorer" la boisson choisie avec les souhaits d'un client d'une part, et de fournir au client le prix exact de la boisson qu'il a commandée d'autre part.

Exemple : un café corsé avec du lait, s'écrit en Java :

```
Beverage darkRoastWithMilk = new Milk( new DarkRoast());
```

et l'obtention de son prix :

```
double price = darkRoastWithMilk.cost();
```




Les différentes boissons héritent de la classe abstraite **Beverage**, La classe abstraite **CondimentDecorator**, instance du pattern Décorateur, représente les exigences possibles du client...

**b) Exercice**

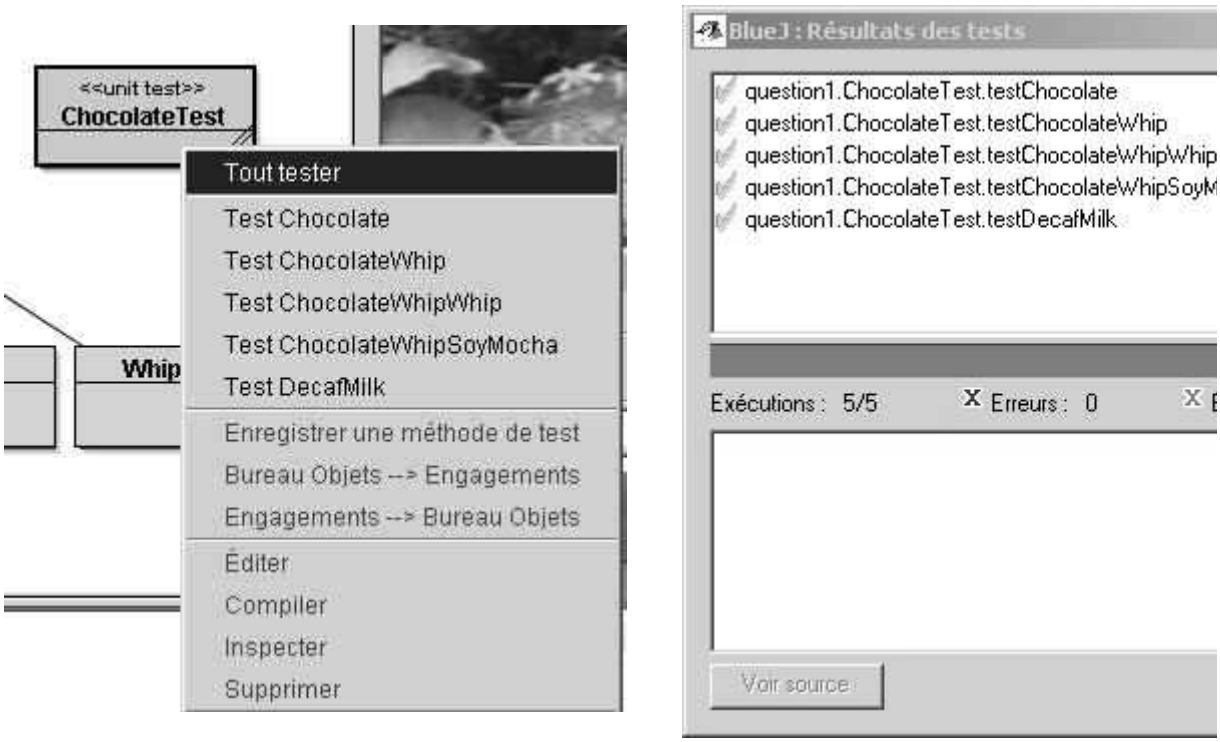
On demande de compléter cette architecture en ajoutant le chocolat (classe **Chocolate**) comme boisson, le coût de cette boisson est de **2.10** .

**Question 1.2)**

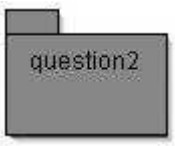
Compléter toutes les méthodes de la classe de test ChocolateTest, les méthodes correspondent aux boissons suivantes (rappel: BlueJ permet

d'enregistrer des séquences d'actions pour constituer des méthodes de test) :

- un chocolat seul
- un chocolat avec de la crème(Whip)
- un chocolat avec deux rations de crème
- un chocolat avec de la crème, du soja(soy) et du moka(mocha),... ( les goûts ne se discutent pas ...)
- un café décaféiné avec du lait




---




## Assertions

### Question 2.1)

#### a) Présentation

L'interface EnsembleDEntiersI ci-dessous spécifie en Java un "ensemble d'entiers"*. Elle contient les signatures des méthodes que devront respecter toutes les classes qui implémenteront un "ensemble d'entiers" :

EnsembleDEntiersI

```
import java.util.Iterator;

public interface EnsembleDEntiersI extends Iterable<Integer>{

	void ajoute( int el ); // ajout d'un élément

	void retire( int el ); // retrait d'un élément
	Attention! il existe une méthode qui retire un élément à partir de son index de type
	int ...

	boolean contient( int el ); // test de présence d'un élément

	boolean estSousEnsemble( EnsembleDEntiersI en ); // test d'inclusion dans
	en

	int cardinal(); // nombre d'éléments

	Iterator<Integer> iterator(); // accès à tous les éléments successivement

	boolean invariant(); // invariant de représentation **

	String toString(); // pour faciliter la mise au point

}
```

*Cette question est inspirée de "Program Development in Java" de Barbara Liskov (Addison-Wesley, 2000).
**L'invariant de représentation est une propriété qui doit être vraie "en permanence" à vérifier au début et à la fin du corps de chaque méthode et à la sortie du constructeur.

**b) Exercice**

**Complétez la classe concrète EnsembleDEntiers qui implémente l'interface EnsembleDEntiersI et qui utilise un vecteur (cf. java.util.Vector<T>) pour stocker ses éléments. Les entiers étant représentés par le type référence Integer.**

### Question 2.2)

**a) Présentation**

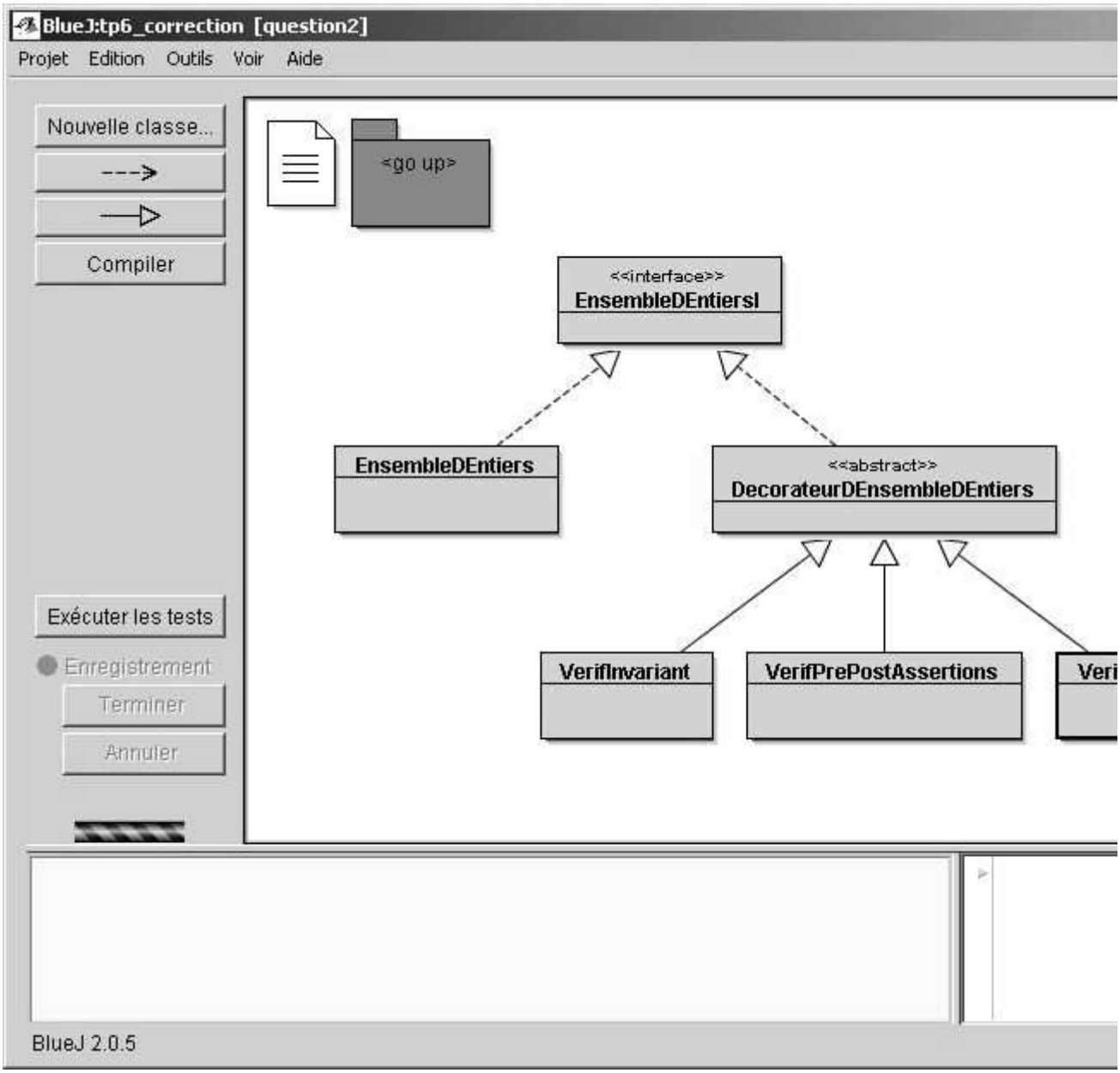
On désire maintenant vérifier l'invariant de représentation dans toutes les méthodes de la classe EnsembleDEntiers. On veut aussi vérifier d'autres propriétés spécifiques à chaque méthode.

*Rappel : L'invariant de représentation est une propriété qui doit être vraie " en permanence" i.e. vraie à chaque appel de méthode et vraie aussi au retour d'une*

*méthode. Il en est de même pour les autres propriétés particulières de chaque méthode.*

Une première solution serait d'insérer ces vérifications au début et à la fin du corps de chaque méthode. Si on insère tous ces tests dans la classe EnsembleDEntiers, celle-ci deviendra " très peu performante " et peu lisible donc difficile à maintenir. Le pattern Décorateur permet d'éviter cet inconvénient majeur.

> Le pattern Décorateur conseille d'envelopper la classe EnsembleDEntiers par la classe DecorateurEnsembleDEntiers (wrapper) qui contiendra une instance d'une classe qui implémente l'interface EnsembleDEntiersI et lui délèguera toutes les opérations.




**b) Exercice**

Complétez la classe **VerifInvariant** en ajoutant à chaque méthode la vérification de l'invariant de classe, cette vérification est effectuée en utilisant la clause assert.

exemple : à la sortie du constructeur de cette classe l'invariant doit être vérifié

```
public VerifInvariant(EnsembleDEntiersI ens){
  super(ens);
  assert invariant():"invariant en échec, en sortie du co
}
```

assert *condition:String;* lorsque la condition échoue une exception est levée et le message est affiché.

**Question 2.3)**

Écrire la classe **VerifPrePostAssertions** en ajoutant à chaque méthode les post assertions induites des propriétés suivantes :

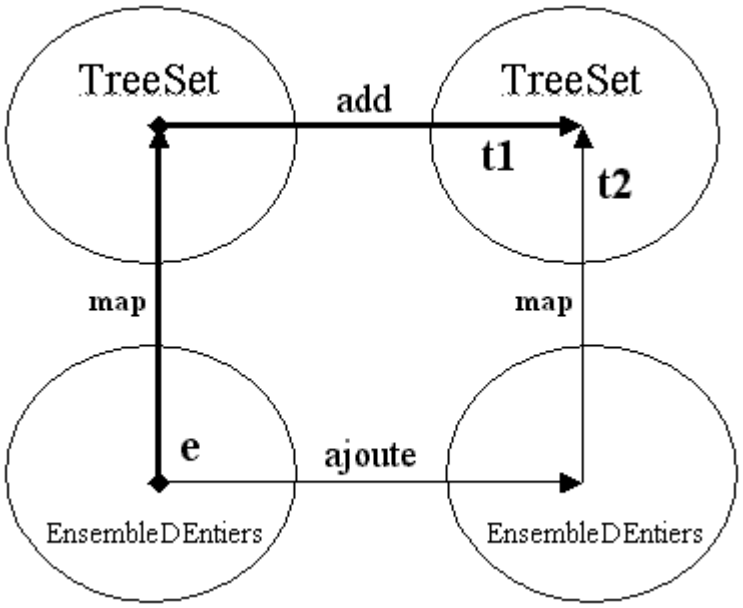
- ajoute : le cardinal augmente de 1 si l'élément n'existait pas ; inchangé sinon
- retire : le cardinal diminue de 1 si l'élément existait ; inchangé sinon
- contient : résultat vrai SI cardinal > 0
- estSousEnsemble : résultat vrai SI cardinal de l'objet =< cardinal du paramètre
- cardinal : résultat >= 0

**Question 2.4)**

**a) Présentation**

# AUTRE Stratégie de vérification :

On imagine maintenant une autre manière de tester notre classe EnsembleDEntiers : si on est capable de transformer n'importe quel EnsembleDEntiers en TreeSet ou en HashSet (qui peuvent être considérées comme deux implémentations de référence fournies par Sun), il suffira alors de comparer les résultats obtenus par nos méthodes et par celles de Sun ; si on obtient les mêmes résultats, on considèrera que nos méthodes fonctionnent correctement. Si on choisit les TreeSet, cette méthodologie peut être illustrée par le schéma suivant. Par exemple vérifier la méthode ajoute : consiste à répondre à la question t1 est-il égal à t2 ? (ou map(e).add(X) est-il égal à map(e.ajoute(X)))




**e** est l'EnsembleDEntiers de départ
**t1** est le TreeSet obtenu en passant par **map** puis la méthode add des TreeSet
**t2** est le TreeSet obtenu en passant par notre méthode ajoute puis par **map**

**b) Exercice**

Écrire la méthode de classe **map**() de la classe **VerifTreeSet** qui retourne un TreeSet contenant les mêmes éléments que l'EnsembleDEntiers passé en paramètre.

La signature de la méthode **map** :

```
public static TreeSet<Integer> map( EnsembleDEntiersI e)
```

**Question 2.5)**

Écrire la classe **VerifTreeSet** qui, au lieu d'effectuer les vérifications précédentes, utilise les TreeSet pour vérifier les propriétés suivantes :

1. - ajoute : voir schéma ci-dessus (t1 est-il égal à t2 ?)
1. - retire : adapter le schéma ci-dessus (t1 est-il égal à t2 ?)
1. - contient : résultat booléen identique si opération appliquée au TreeSet ?
1. - estSousEnsemble : résultat booléen identique si opération appliquée au TreeSet ?
1. - cardinal : taille du TreeSet identique ?