4 pages annexes (-1, ex 15 ; -3, ex 16) 5,5

TE Génie Logiciel – *14 Avril 2015* Nom : MINDER VALENTIN

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
4	1	3	2	1	2	3	2	3	3,5	3	3	3	6	8	15	62,5
4	1	3	4	1	5	3	2	3	5	3	3	5	8/6	8	15	73

21 — 18,5 — 31 — 73-31=42 — 39,5

EXERCICE 1 – 4 POINTS

- 1.5 POINT PAR RÉPONSE FAUSSE

La crise du logiciel des années 70 a eu pour conséquence - dans le courant des deux ou trois années qui ont suivi ce constat - d'opérer une remise en question du processus de développement logiciel et des outils utilisés. Notamment :

- ☐ L'arrêt de la programmation procédurale au profit d'une programmation orientée objet.
- ☐ La réduction systématique de la taille des équipes de développement
- ☐ La promotion d'un cycle de vie de type itératif
- ☒ La promotion de la phase d'analyse des besoins

4/4

EXERCICE 2 – 1 POINT

Les gens du Génie Logiciel sont parfois tordus. Ils opèrent une distinction assez nette entre le fait de *vérifier* le logiciel et le fait de le *valider. Qu'est-ce que cela signifie ?*

Valider = répond aux besoins du client (le "quoi") – "the right thing"
Vérifier = fait les choses correctement (le "comment") – "the thing right"

1/1

EXERCICE 3 – 3 POINTS

- 1POINT PAR RÉPONSE FAUSSE

Parmi les différents critères de qualités énumérés ci-dessous, indiquez s'il s'agit d'une qualité « externe », « interne » ou alors ayant trait au processus de développement. Une seule réponse possible par critère de qualité.

	Externe	Interne	Processus
Réutilisation	o	☒	o
Correction	☒	o	o
Respect des délais	o	o	☒
Evolutivité	o	☒	o
Robustesse	☒	o	o
Maintenabilité	o	☒	o

3/3

EXERCICE 4 - 4 POINTS

- 1POINT PAR RÉPONSE FAUSSE

Méthode UP. Indiquer par une croix dans quelle phase l'intensité de chaque discipline/activité est maximum (choix exclusif !)

8

Init : Initialisation – **Elab** : Elaboration – **Constr** : Construction – **Trans** : Transition

	Init	Elab	Constr	Trans
Réalisation de l'architecture centrale	o	o	o	o
Génération d'un sous-ensemble exécutable	o	o	o	o
Rédaction des cas d'utilisation	o	o	o	o
Livraison finale	o	o	o	o
Réalisation d'un module	o	o	o	o
Etude de faisabilité	o	o	o	o
Modélisation de domaine	o	o	o	o
Planification des itérations	o	o	o	o

2/4

EXERCICE 5 – 1 POINT

Quel est le principal intérêt du cycle de vie itératif ?

1/1 Il permet une gestion des risques efficace car le client a très vite (dans le projet) une première version à tester/valider afin de continuer correctement

EXERCICE 6 – 5 POINTS – THREADS

- 1.5POINT PAR RÉPONSE FAUSSE

2/5

Parmi les propositions énoncées ci-dessous, cochez la ou les proposition(s) vous paraît (paressent) correcte(s) ?

- ☒ Deux threads Java peuvent se partager des instructions
- ☐ Deux threads Java de la même application peuvent s'exécuter en vrai parallélisme
- ☐ La machine virtuelle de Java est responsable de l'ordonnancement des threads et notamment du time-slicing
- ☒ Le système d'exploitation alloue une zone mémoire pour chacun des threads de l'application Java
- ☒ Un thread Java corrompu peut corrompre l'exécution des autres threads

EXERCICE 7 – 3 POINTS

- 1.5POINT PAR RÉPONSE FAUSSE

3/3

Parmi les propositions énoncées ci-dessous, cochez la ou les proposition(s) vous paraît (paressent) correcte(s) ?

Un cas d'utilisation est :

- ☒ Une fonctionnalité mise à disposition par le système
- ☐ Un cas spécial d'utilisation nécessitant l'intervention d'un acteur
- ☒ Une utilisation possible du système qui arriverait de cas en cas

EXERCICE 8 – 2 POINTS

La méthode UP s'appuie sur quel(s) type(s) de programmation ?

2/2 OO (orienté-objet)

La méthode UP s'appuie sur quelle(s) notation(s) ?

UML (unified modelling language)

EXERCICE 9 – 3 POINTS - THREADS

Donnez-moi au moins 4 exemples de threads lancés automatiquement lorsque l'on lance un programme Java.

3/3 | 13

'idle (thread "inactif") ✓ GC (garbage collector)

main (thread principal du programme lancé) ✓

VM (thread machine virtuelle Java sur système hôte)

Début un processus Processus créé

EXERCICE 10 – 5 POINTS *(- 1.5 POINTS PAR RÉPONSE FAUSSE)*

LE VRAI ET LE FAUX DE LA RELATION EXTENDS

Dans le cadre des « cas d'utilisation », et plus spécifiquement de la relation « extends », cocher les assertions qui sont vraies.

3,5/5 -1,5

- ☐ La relation « extends » permet de modéliser des activités asynchrones, pouvant interrompre le cas de base étendu
- ☑ La relation « extends » permet de modéliser une variante au cas de base étendu.
- ☐ La relation « extends » permet d'étendre un cas d'utilisation de base, d'où le sens de la flèche.
- ☐ La relation « extends » permet de compléter un cas d'utilisation de base dont la rédaction à été bloquée, en lui rajoutant une fonctionnalité considérée comme obligatoire.
- ☐ La rédaction du cas de base doit se référer aux différentes extensions qu'il possède

EXERCICE 11 – 3 POINTS *(-1.5 POINTS PAR RÉPONSE FAUSSE)*

3/3

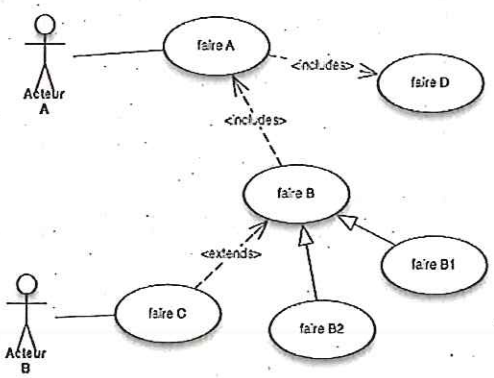
Acteurs et cas d'utilisation. Parmi les assertions suivantes, cochez celles qui reflètent correctement le concept d'acteur.

- ☐ Un rôle peut être associé à plusieurs personnes physiques, mais la même personne physique sera associée à un rôle au maximum. FAUX
- ☐ L'héritage entre acteurs implique un héritage des droits
- ☐ Un acteur externe est situé dans un endroit géographiquement éloigné du système à développer FAUX

EXERCICE 12 – 3 POINTS *(- 1.5 POINTS PAR RÉPONSE FAUSSE)*

3/3

Spécialisation et cas d'utilisation



Les propositions suivantes sont-elles vraies ou fausses ?

- ☐ Pour « Faire C », il faut « Faire B » FAUX
- ☐ Pour « Faire B1 », il faut « Faire D » VRAI
- ☐ Pour « Faire B1 », il faut « Faire C » FAUX

9,5

EXERCICE 13–5 POINTS *(- 1 POINT PAR RÉPONSE FAUSSE)*

Vous avez été embauché pour créer le document des exigences d'un nouveau système – le système QUEST - de définition, d'impression et de récolte de QCM papiers, relié au système de gestion des étudiants d'un institut de formation (qui recueille les résultats) et à la base de données de questions.

3/5

Décidez si chaque élément de la liste suivante est un *acteur externe*, un *acteur principal*, un *acteur secondaire* (auxiliaire), ou s'il n'est pas du tout un acteur.

	Acteur externe	*Acteur principal*	*Acteur secondaire*	*Pas un acteur*
La souris de l'ordinateur				X ✓
Le professeur (qui définit les QCM)		X ✓		
L'analyseur optique de QCM (analyse des réponses) au moment où le professeur lance l'évaluation des résultats.		X →		
La base de données des questions qui permet de sélectionner une question au moment de la confection d'un QCM			X ✓	
Le système QUEST				X ✓
Le directeur de l'école à qui sont transmises les statistiques se rattachant aux résultats des étudiants.	X ✓			
L'imprimante utilisée pour l'impression des QCM.			X ✓	
Le responsable de l'entretien de l'imprimante		X →		
La définition d'une nouvelle question				X ✓
Le système informatique de gestion des étudiants (pour que le professeur puisse accéder à la liste des étudiants)			X ✓	

élément du système

- 1

système lui-même

- 1

(si Base de Données au moment de la définition: secondaire)

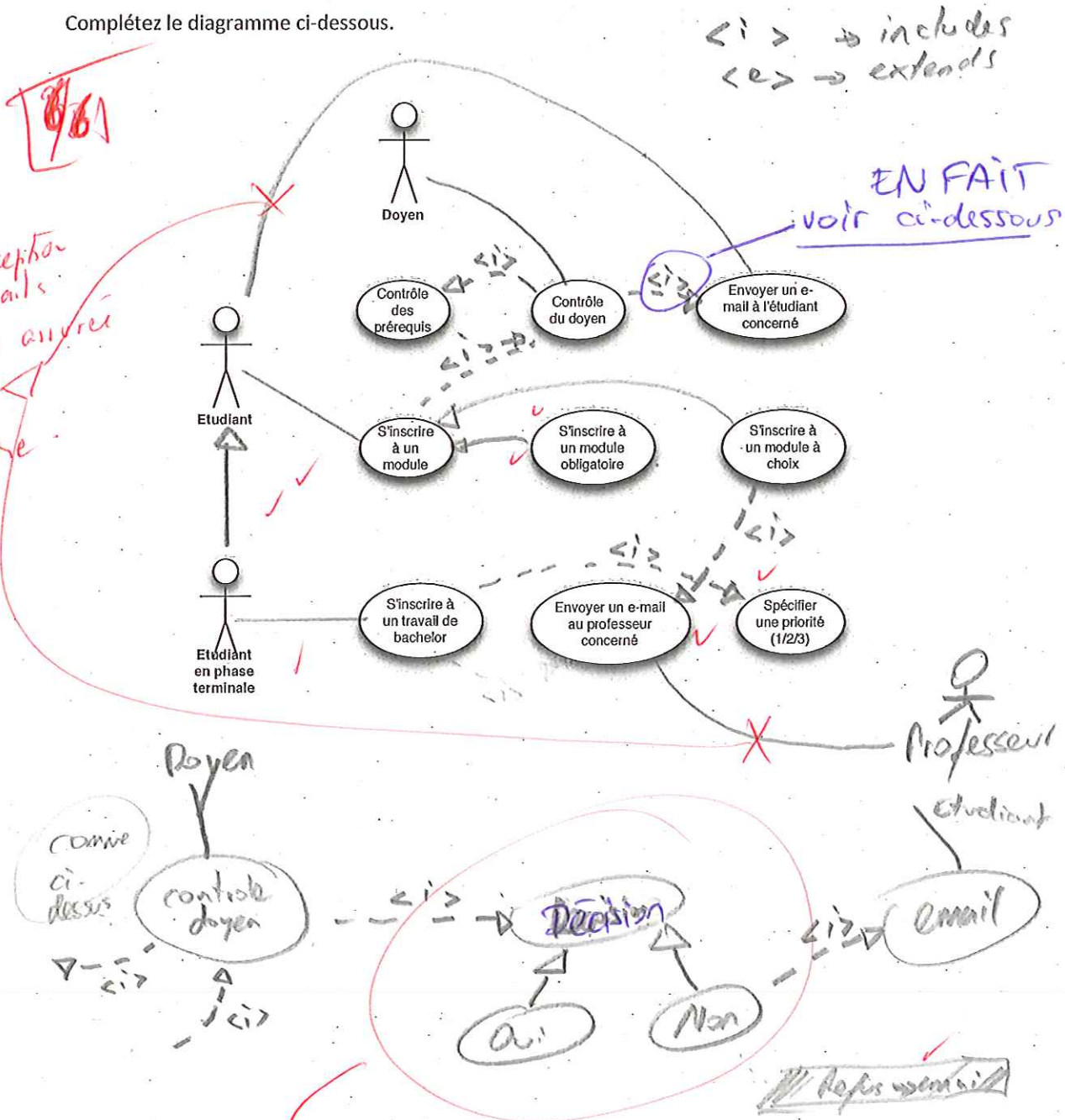
3

EXERCICE 14 – 8 POINTS

Dessinez le diagramme de cas d'utilisation correspondant à la description suivante.

- Un étudiant peut s'inscrire à un module, il peut s'agir d'un module à choix ou d'un module obligatoire.
- L'inscription à un module nécessite le contrôle du doyen. Suivant le type de module, ce contrôle peut nécessiter un contrôle des prérequis. En cas de refus, un mail est envoyé à l'étudiant et l'inscription est annulée.
- L'inscription à un module à choix implique l'envoi d'un e-mail au professeur concerné.
- Un étudiant en phase terminale peut faire tout ce que peut faire normalement un étudiant. En plus, il a toutefois la possibilité de s'inscrire à un travail de bachelor en indiquant une priorité entre 1et 3.

Complétez le diagramme ci-dessous.

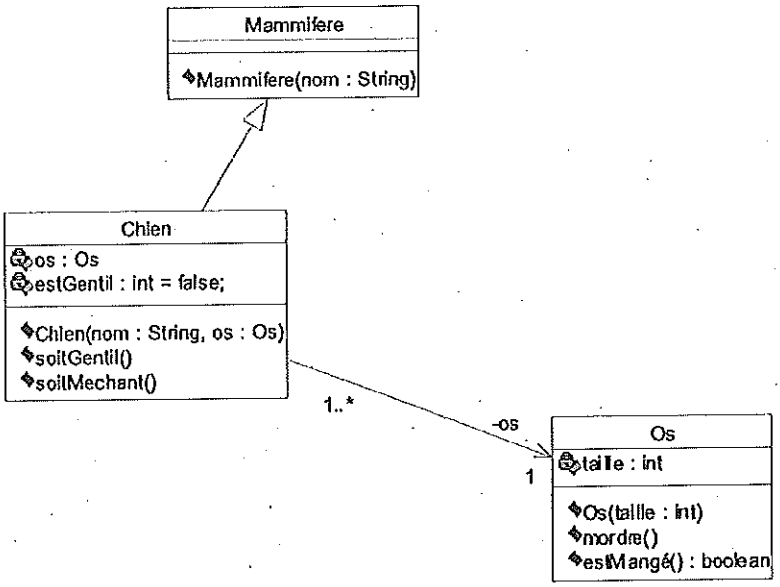


EXERCICE 15 – 8 POINTS

Des chiens et un os

On aimerait modifier la Classe «Chien» de la hiérarchie «Animal».

Considérons le diagramme de classes représenté ci-après :




```
//-----------------------------------------------------------------------
class Mammifere {
  public Mammifere (String nom) {
    super(nom);
  }
}
//-----------------------------------------------------------------------
class Chien extends Mammifere {

  private boolean estGentil = false;
  private Os os;              // os associé

  public Chien (String nom, Os os) {
     super (nom);
     this.os = os;
  }

  public void soitGentil() { estGentil = true; }
  public void soitMechant () { estGentil = false; }
}
//-----------------------------------------------------------------------
class Os {
  private int taille;

  public Os (int taille) {this.taille = taille;}

  public void mordre() {
   // Une fois mordu, la taille de l'os diminue d'une unité
   // Si l'os est déjà mangé, une exception est levée
    if (taille <= 0) throw new RuntimeException("Os déjà mangé");
    taille--;
  }
  public boolean estMangé() {return taille == 0; }
}
```

//----------

On aimerait que toute instance de la classe «Chien» soit un **objet actif** : sa tâche, démarrée dès que le chien est créé, doit passer son temps, une fois par seconde, à mordre dans l'os qui lui a été associé à la naissance.

Ainsi, toutes les secondes, le message «os.mordre()» sera envoyé à l'os associé, et ceci **jusqu'à ce que l'os soit entièrement mangé.**

Voici, en gros, **l'idée** de l'algorithme :

```
while (! os.estMangé()) {
   os.mordre() ;
   try {Thread.sleep(1000) ;}
   catch (InterruptedException e) {}
}
```

☞ Modifiez en conséquence les classes «Chien» et «Os» (éventuellement).

Pour gagner du temps, vous pouvez « oublier » les méthodes «soitGentil()» et «soitMechant()», qui ne sont pas concernées par la modification.

Contraintes!

On aimerait qu'il soit possible de créer deux chiens, **associés au même os**, auquel ils s'attaqueront en même temps:

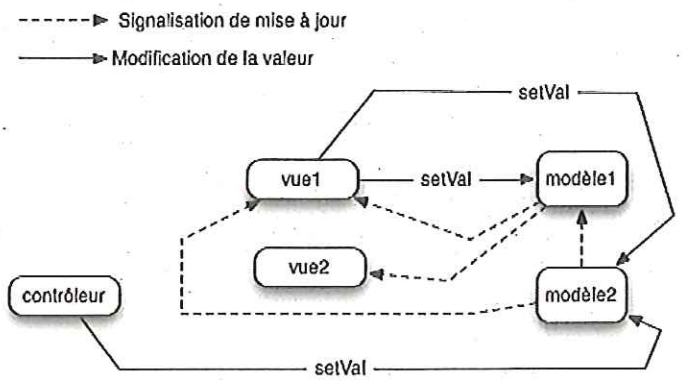
```
Os unOs = new Os(10) ;  // Os de taille 10
Chien ch1 = new Chien("Medor", unOs) ;
Chien ch2 = new Chien("toutou", unOs) ;
```

La levée d'exception, au cas où le chien mord dans un os déjà mangé, doit continuer à être levée. Toutefois, si votre solution est correcte, cette dernière ne doit jamais être levée (en effet, le test de la boucle while du chien devrait l'assurer).

COMMENTEZ VOTRE SOLUTION : notamment, vos décisions de mise en œuvre doivent être expliquées !!

EXERCICE 16– 15 POINTS

Supposons un programme dont l'architecture s'appuie sur le schéma présenté ci-dessous :



Les flèches en points-tillés illustrent les dépendances entre vues et modèles :

1. La `vue1` est concernée par les mises à jour des modèles `modèle1` et `modèle2`
2. La `vue2` est concernée uniquement par la mise à jour du modèle `modèle1`
3. Le `modèle1` est concerné par les mises à jour du modèle `modèle2`

Les flèches « `setVal` » illustrent les mises à jour possibles des modèles par la `vue1` et/ou le `contrôleur` : le `modele1` sera mis à jour uniquement par `vue1`, le `modele2` par `vue1` et/ou le `contrôleur`.

Que faire ??

- Ecrire les classes `Controleur`, `Vue1` et `Vue2`
  Note : `Contrôleur` et `Vue1` doivent tous deux mettre à disposition une méthode publique « `changeVal (int val)` » qui a pour effet – pour le contrôleur - d'envoyer le message `setVal (int val)` à `modèle2`, et – pour `Vue1` - d'envoyer le message `setVal (int val)` à `modèle1` et `modèle2`.
- Ecrire la classe `Modele` depuis laquelle sont créées les instances `modele1` et `modele2`
- A la réception d'un signal de mise à jour, `vue1`, `vue2` et `modele1` doivent simplement répondre en affichant la nouvelle valeur du modèle au moyen d'un « `System.out.println` ».

Exercice 15 — MINDER

```
class Chien implements Runnable {
    private boolean running = true;
    public void tryMordre() {
        synchronized (os) {
            if (os.estMange()) {
                running = false;
            } else {
                os.mordre();
            }
        }
    }
    public void run() {
        while (running) {
            tryMordre();
            Thread.sleep(1000);
            } catch (InterruptedExc e) {}
        }
    }
}
```

// uniquement modification

flag de continuité pour le chien

Ceci constitue une 1ère solution - fonctionnelle - qui a un problème de conception. Voir au verso version améliorée.

Très bien ! 8/8 !

```
// programme principal:
new Thread(ch1).start();
new Thread(ch2).start();
```

} démarrage des objects actifs "chiens"

// conception:

l'opération atomique à effectuer chaque seconde (par chaque chien) est 1. vérifier qu'on peut mordre l'os (!estMange())
2. mordre l'os (os.mordre())

Entre les 2, il faut que l'os soit "acquis" (cela synchronized)
Comme il s'agit de 2 méthodes différentes dans os, il m'a semblé plus facile de gérer du côté du chien : (tryMordre() dans

1. j'acquiers l'os (bloc synchronized sur os)
2. os.estMange() ? oui → ~~sortir~~ changer le flag du chien et → 4
   non → 3
3. os.mordre()
4. libérer l'os

Chaque seconde dans run(), tryMordre() est appelé (tant que c'est possible // que le chien est running)

Afin d'arrêter les chiens "proprement" lorsque leur os est mangé, j'ai utilisé un boolean flag privé (→ donc pas de protection particulière pas de concurrence) afin que le chien s'arrête de mordre lorsque son os est terminé.

Toutefois cette solution souffre d'un problème : la section de synchronized est extérieure à l'objet critique (os) ... et un nouveau client sur os pourrait mal l'utiliser ... ⚠

[Autre solution :] implémenter un mordreSafe() de côté de l'os (qui retourne true si succès de mordre)

```
// Chien implements Runnable
  public void run (){
    while (os.mordreSafe()){
      // sleep comme avant
    }
  }
// Os
public bool synchronized mordreSafe (){
    if (os.estMange()){
      return false;
    } else {
      os.mordre()
      return true;
    }
}
```

// mordre() et estMange() sont mis en PRIVATE (pas accessible de l'extérieur)

Ici, c'est l'os qui s'assure qu'il est libre avant de ~~mordre~~ se laisser mordre et renvoit un flag au chien qui saura s'il peut continuer ou non à mordre.

Cette méthode est plus simple que la première et surtout plus élégante : l'objet critique (os) gère lui-même sa synchronisation → évite qu'un nouveau client sur os (p.ex. un autre type de Chien) doive s'en occuper

Exercice 16 (1/3)

```
class Controleur {
    Modele m1, m2; Vue1 v1; Vue2 v2;
    Controlleur () {
        int val = 3;
        m1 = new Modele (val);
        m2 = new Modele (val);

        v1 = new Vue1 (m1, m2);
        v2 = new Vue2 ();

        m1.addObserver (v1);
        m1.addObserver (v2); -(2)
        m2.addObserver (v1);
        m2.addObserver (m1); -(3)

    }
public changeVal (int val) {
    m2.setVal (val);
  }
}
```

15/15

(1) (2) (3) : données

MINDER

Exercice 16 (3/3)

observent les modèles (V1 → m1, m2 ; V2 → m1)

```
class Vue1 implements Observer {
privateModele m1, m2
public Vue1 (Modele m1, Modele m2) {
        this.m1 = m1;
        this.m2 = m2;
    }
public changeVal (int val) {
        m1.setVal (val);
        m2.setVal (val);
    }
public update (Observable o, Object arg) {
        system.out.println ("MAJ@vue1 : " + arg + "/from" + o.getClass());
    }
}
```

auto-conversion grâce à toString (comme Integer.toString(arg))

```
class Vue2 implements Observer {
public Vue2 () {}
public update (Observable o, Object arg) {
        System.out.println ("MAJ@vue2 :" + arg + "/from" + o.getClass());
}
```

MINDER

Exercice 16. (2/3)

observable car observé par les 2 vues (et pour m2 par m1)

observer car m1 observe m2

```
class Modele extends Observable implements Observer {
private int val;
public Modele (int val) {
        this.val = val;
}
// utilisé depuis vue 1 pour m1 et m2, depuis controleur pour m2
public setVal (int val) {
    this.val = val;
    setChanged();
    notifyObservers(val);
}
// dans utilisé Modele 1 (par l'appel notify de modele 2)
public update (Observable o, Object arg) {
    System.out.println("MAJ@Model:" + arg + "from " + o.getClass());
    setVal ((Integer) arg);   // pour mettre à jour le modele 1 en fonction de la notif de modele 2 (→ notification récursive
}
}
```

MINDER