CPOA - Support TD 5

Dut/Info-S3/M3105 - (Semaine 49)

Version corrigée 

Cette version comporte des indications pour les réponses aux exercices.

PreReq	1. Je sais programmer en Java. 2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage. 3. Je maîtrise quelques patrons de conception.
ObjTD	Aborder le patron **Observer**.
Durée	**1** TD et **2** TPs

1. Motivation

Dans cet exercice, nous allons explorer comment on peut maintenir la cohérence entre plusieurs objets qui sont liés sans pour autant maintenir d'association fortes entre ces objets.

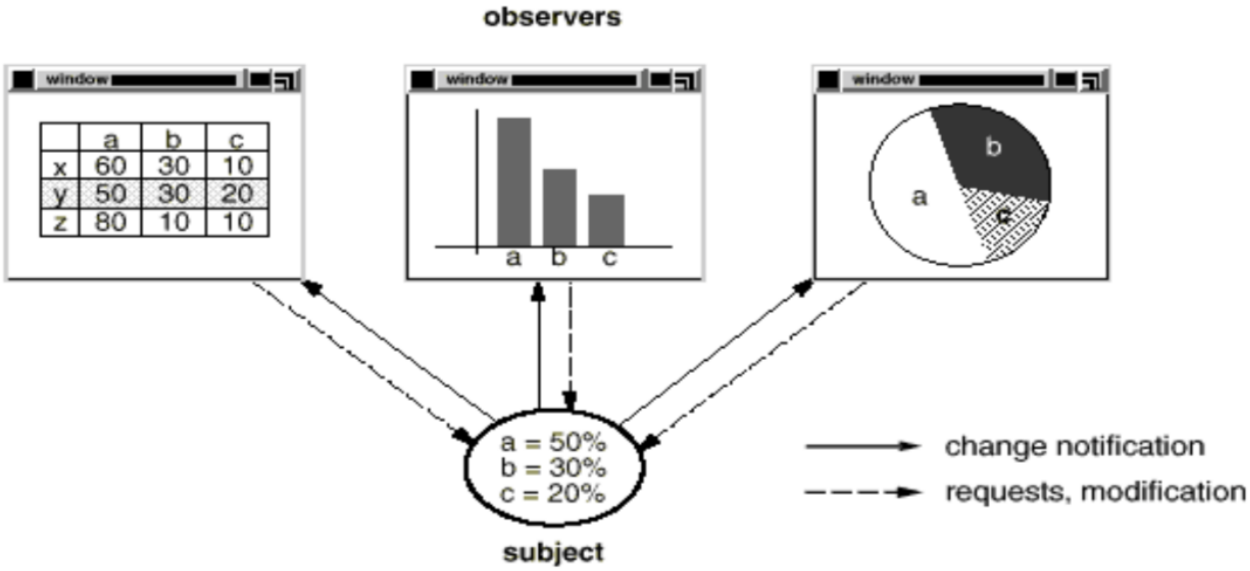


Figure 1. L'illustration classique du patron Observer

2. Le patron Observer

2.1. Définition

Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

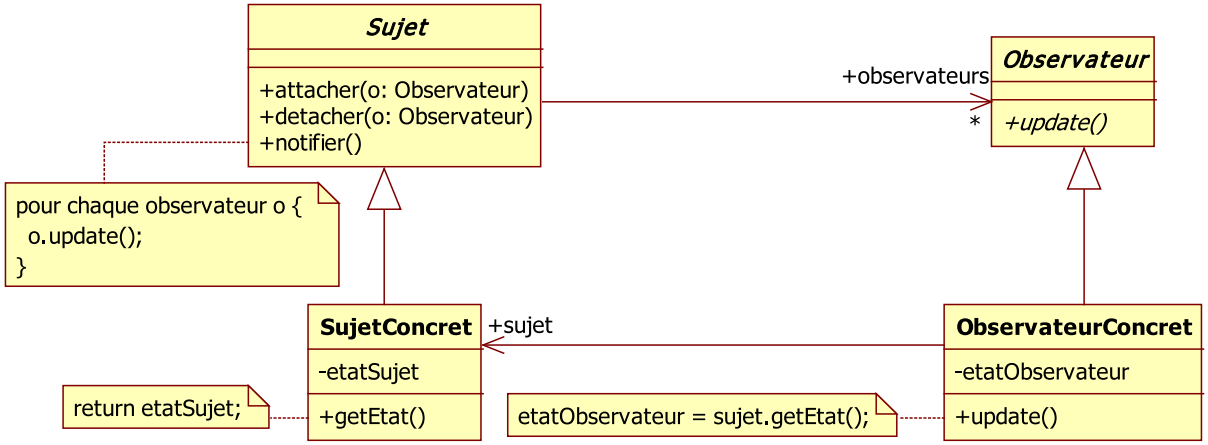


Figure 2. Modèle UML du patron Observer

Quelques précisions :

- les observateurs doivent être enregistrés
- le sujet est chargé de prévenir les observateurs (d'un changement).

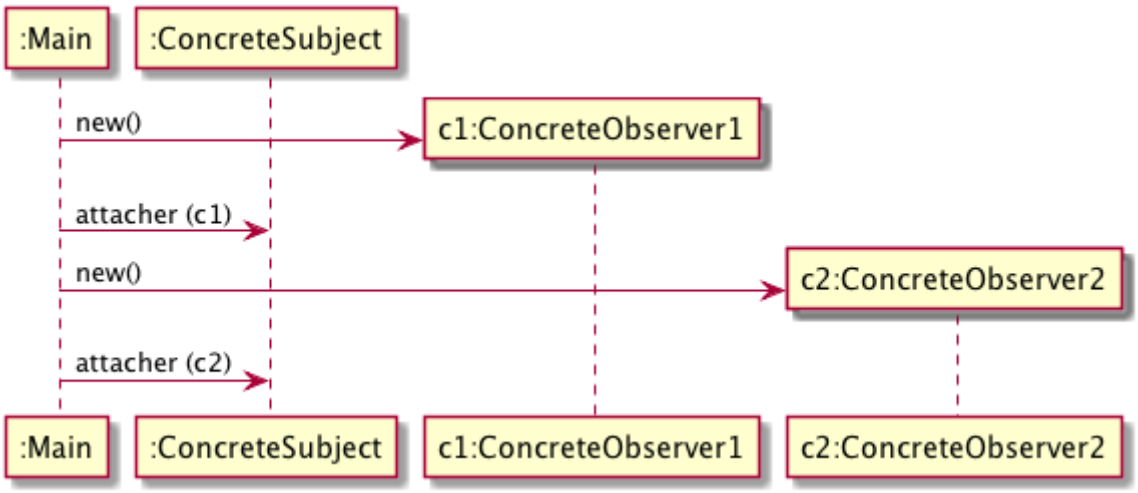


Figure 3. Exemple d'utilisation

On peut réaliser de plusieurs façon le `attacher`, mais il doit être fait pour que ca marche :

- par un objet extérieur,
- par le `concreteObserver` lui-même.

2.2. Objectif

- Définir une dépendance un-plusieurs entre un objet observé et des objets observateurs de telle façon que quand l'observé évolue, ces observateurs sont informés et mis à jour automatiquement.

2.3. Application

Le patron *Observer* est utilisable dans de nombreuses situations :

- Quand un concept a deux aspects, l'un dépendant de l'autre. Encapsuler ces aspects dans des objets séparés permet de les utiliser et les laisser évoluer de manière indépendante.
- Dès que le changement d'un objet entraîne le changement de plusieurs autres.
- Dès qu'un objet doit en notifier un certain nombre d'autres sans les connaitre.

2.4. Participants

- `Subject`
  - Garde trace de ses `observers`
  - Fournit une interface pour attacher et détacher les objets `Observer`
- `Observer`
  - Définit une interface pour les notifications `update`
- `ConcreteSubject`
  - Les objets observés.
  - Gère les objets `ConcreteObserver` intéressés.
  - Envoit une notification à ses observateurs quand il change d'état
- `ConcreteObserver`
  - Les objets observateurs.
  - Gère la cohérence avec l'état de l'observé.
  - Implémente l'interface `update()` d'`Observer`.

2.5. Scenario

QUESTION

Pour vérifiez que vous avez compris son utilité, réalisez un diagramme de séquence illustrant l'utilisation du patron *Observer*.

Solution

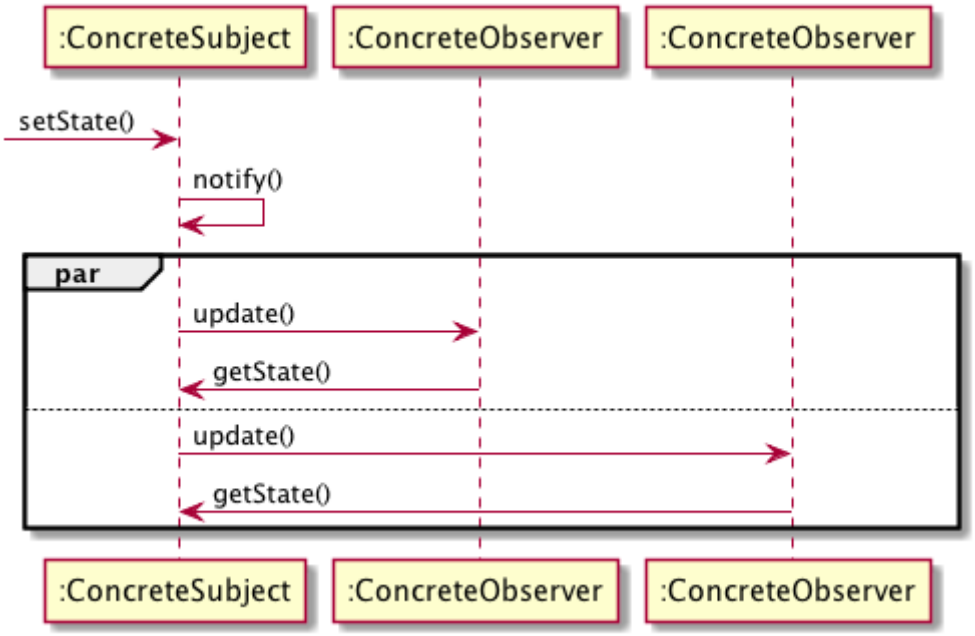


Figure 4. Exemple d'utilisation

2.6. Bénéfices

- Couplage minimal entre `Subject` et `Observer`
  - On peut utiliser les sujets sans se soucier des observateurs et vice-versa
  - Les observateurs peuvent être ajoutés sans modifier nécessairement le sujet
  - Tout ce que le sujet connait de ses observateurs, c'est leur liste
  - Le sujet n'a pas connaissance de la classe concrète des observateurs, juste qu'ils implémente l'interface `update`
  - Sujet et observateur peuvent appartenir à des couches d'abstraction différentes
- Support à la diffusion d'événements
  - Le sujet envoie des notifications à tous ses observateurs
  - Les observateurs peuvent être ajoutés/retirés à la volée

2.7. Limites

- Possibilité d'effets en cascade
  - Les observateurs ne sont pas nécessairement au courant les uns des autres. Il faut donc être prudent sur les effets des `update`
- Implication pour observateurs de déduire les changements à partir d'un simple `update`.

3. Implémentation

3.1. Problèmes

Nous allons maintenant nous intéresser à l'implémentation.

Si vous connaissez l'implémentation `Observer` Java, essayez de l'oublier!

QUESTION

Voici une liste des éléments, liés à l'implémentation, à résoudre. Pour chacune, essayez de fournir des éléments de réponse :

1. Comment le sujet garde la trace de ses observateurs?
2. Et si un observateur veut observer plus d'un sujet?
3. Qui déclence les `update`?

4. Comment s'assurer que les sujets mettent bien à jour leur état avant de notifier leur changement.
5. Quel niveau de détail au sujet de son changement doit transmettre le sujet à ses observateurs?
6. Est-ce que les observateurs peuvent souscrire à des événements spécifiques?
7. Est-ce qu'un observateur peut lui-même être un sujet?
8. Peut-on faire en sorte qu'un observateur soit notifié qu'après qu'un certain nombre de sujets aient changé d'état?

Solution

1. Array, liste chaînée, ...
2. Le subject peut s'identifier auprès des observateurs via l'interface `update`
3. Par exemple :
    - Le subjet quand il change d'état
    - Les observateurs après qu'ils ait provoqué un ou plusieurs changements
    - Des objets quelconques!

4. Difficile autrement que par checking/testing
5. Par exemple :
    - Push Model ⇒ Beaucoup
    - Pull Model ⇒ Le moins possible
6. On parle alors de *publish-subscribe*
7. Yes!
8. Par exemple :
    - Utiliser un objet intermédiaire qui agit comme médiateur

3.2. *Observer* en Java

Java fournit des classes *Observable/Observer* pour le patron *Observer*. La classe `java.util.Observable` est la classe de base pour les sujets. Ainsi, toute classe qui veut être observée étant cette classe dont voici les caractéristiques :

- fournit des méthodes pour ajouter/enlever des observateurs
- fournit des méthodes pour notifier les observateurs
- une sous-classe concrète doit seulement s'occuper de notifier à chque méthode modifiant l'état des objets (*mutators*)
- utilise un vecteur stoquant les références des observateurs

L'interface `java.util.Observer` correspond aux observateurs qui doivent implémenter cette interface.

3.2.1. La classe java.util.Observable

QUESTION

Voici la liste des méthodes de `java.util.Observable` :

```
public Observable()
public synchronized void addObserver(Observer o)
protected synchronized void setChanged()
public synchronized void deleteObserver(Observer o)
protected synchronized void clearChanged()
public synchronized boolean hasChanged()
public void notifyObservers(Object arg)
public void notifyObservers()
```

Retrouver les commentaires correspondants :

1. Adds an observer to the set of observers of this object
2. If this object has changed, as indicated by the hasChanged() method, then notify all of its observers and then call the clearChanged() method to indicate that this object has no longer changed. Each observer has its update() method called with two arguments: this observable object and the arg argument. The arg argument can be used to indicate which attribute of the observable object has changed.
3. Indicates that this object has changed
4. Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by `notifyObservers()`.
5. Same as above, but the arg argument is set to null. That is, the observer is given no indication what attribute of the observable object has changed.
6. Tests if this object has changed. Returns true if setChanged() has been called more recently than clearChanged() on this object; false otherwise.
7. Construct an `Observable` with zero observers
8. Deletes an observer from the set of observers of this object

Solution

```
public Observable()
// => Construct an `Observable` with zero Observers
public synchronized void addObserver(Observer o)::
// => Adds an observer to the set of observers of this object
protected synchronized void setChanged()
// => Indicates that this object has changed
public synchronized void deleteObserver(Observer o)
// => Deletes an observer from the set of observers of this object
protected synchronized void clearChanged()
/* => Indicates that this object has no longer changed, or that it has
already
notified all of its observers of its most recent change. This method is
called
automatically by notifyObservers(). */
public synchronized boolean hasChanged()
/* => Tests if this object has changed. Returns true if setChanged()
has been
called more recently than clearChanged() on this object; false
otherwise. */
public void notifyObservers(Object arg)
/* => If this object has changed, as indicated by the hasChanged()
method, then
notify all of its observers and then call the clearChanged() method to
indicate that this object has no longer changed. Each observer has its
update() method called with two arguments: this observable object and
the
arg argument. The arg argument can be used to indicate which attribute
of
the observable object has changed. */
public void notifyObservers()
/* => Same as above, but the arg argument is set to null. That is, the
observer
is given no indication what attribute of the observable object has
changed. */
```

3.2.2. L'interface java.util.Observer

java.util.Observer

```
/**
* This method is called whenever the observed object is changed. An
* application calls an observable object's notifyObservers method to have all
* the object's observers notified of the change.
*
* Parameters:
* o - the observable object
* arg - an argument passed to the notifyObservers method
*/
public abstract void update(Observable o, Object arg)
```

3.3. Observable/Observer par l'exemple

ConcreteSubject.java

```
/**
* A subject to observe!
*/
public class ConcreteSubject extends Observable {

  private String name;
  private float price;
  public ConcreteSubject(String name, float price) {
    this.name = name;
    this.price = price;
    System.out.println("ConcreteSubject created: " + name + " at " + price);
  }
  public String getName() {return name;}
  public float getPrice() {return price;}
  public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
  }
  public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
  }
}
```

NameObserver.java

```
// An observer of name changes.
public class NameObserver implements Observer {
  private String name;
  public NameObserver() {
    name = null;
    System.out.println("NameObserver created: Name is " + name);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      name = (String)arg;
      System.out.println("NameObserver: Name changed to " + name);
    } else {
      System.out.println("NameObserver: Some other change to
      subject!");
    }
  }
}
```

PriceObserver.java

```
// An observer of price changes.
public class PriceObserver implements Observer {
  private float price;
  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      price = ((Float)arg).floatValue();
      System.out.println("PriceObserver: Price changed to " +
      price);
    } else {
      System.out.println(”PriceObserver: Some other change to
      subject!");
    }
  }
}
```

TestObservers.java

```
// Test program for ConcreteSubject, NameObserver and PriceObserver
public class TestObservers {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

QUESTION

Donnez la trace d'exécution du code ci-dessus.

Solution

Test program output

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
NameObserver: Some other change to subject!
PriceObserver: Price changed to 9.22
NameObserver: Some other change to subject!
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```

On oublie souvent que le moindre changement informe les deux observateurs.

Attention: l'exécution des méthodes observateur.update() n'est pas déterministe car chaque appel est effectué dans un thread spécifique.

3.4. Limitations de Observable/Observer en Java

Problème

Supposons que la classe que nous voulons rendre observable est déjà une sous-classe, par

exemple :

```
class SpecialSubject extends ParentClass
```

QUESTION

Puisque Java ne supporte pas l'héritage multiple, comment pouvons-nous avoir `ConcreteSubject` qui étende à la fois `Observable` et `ParentClass` ?

1. Proposez une solution
2. Définissez le diagramme de classe correspondant
3. Écrire l'implémentation Java (principalement `SpecialSubject`)

Deux méthodes de `java.util.Observable` sont **protected** : `setChanged()` et `clearChanged()`.

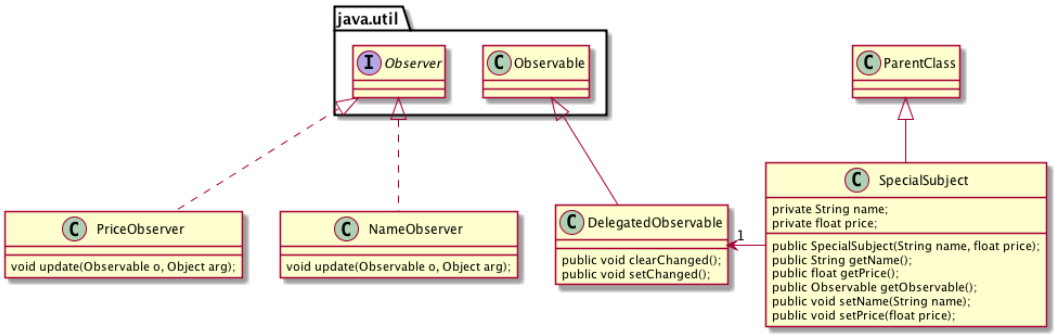
Vous pourrez pour vous aider utiliser le code de test suivant :

```
// Test program for SpecialSubject with a Delegated Observable.
public class TestSpecial {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    SpecialSubject s = new SpecialSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.getObservable().addObserver(nameObs);
    s.getObservable().addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

Qui produit la même trace qu'à l'exercice précédent.

Solution

1. Une solution :
   - Utiliser une délégation
   - Nous aurons un `SpecialSubject` qui contient un `Observable`
   - Nous déléguons les aspects "observabilité" dont `SpecialSubject` a besoin à son objet contenu, de type `Observable`
2. Diagramme de classe :



3. Code Java :

```
/**
 * A subject to observe!
 * But this subject already extends another class.
 * So use a contained DelegatedObservable object.
 * Note that in this version of SpecialSubject we do
 * not duplicate any of the interface of Observable.
 * Clients must get a reference to our contained
 * Observable object using the getObservable() method.
 */
import java.util.Observable;

public class SpecialSubject extends ParentClass {
   private String name;
   private float price;
   private DelegatedObservable obs;

   public SpecialSubject(String name, float price) {
     this.name = name;
     this.price = price;
     obs = new DelegatedObservable();
     System.out.println("ConcreteSubject created: " + name + " at " +
price);
   }
   public String getName() {return name;}
   public float getPrice() {return price;}
   public Observable getObservable() {return obs;}
   public void setName(String string) {
     name = string;
     obs.setChanged();
     obs.notifyObservers(name);
   }
   public void setPrice(float f) {
     price = f;
     obs.setChanged();
     obs.notifyObservers(price);
   }
}
```

Explications (désolé pour l'anglais) :

What's this `DelegatedObservable` class? It implements the two methods of `java.util.Observable` that are protected methods: `setChanged()` and `clearChanged()`.

Apparently, the designers of Observable felt that only subclasses of Observable (that is, "true" observable subjects) should be able to modify the state-changed flag.

If `SpecialSubject` contains an `Observable` object, it could not invoke the `setChanged()` and `clearChanged()` methods on it. So we have `DelegatedObservable` extends `Observable` and override these two methods making them have public visibility.

Java rule: A subclass can changed the visibility of an overridden method of its superclass, but only if it provides more access.

QUESTION

Après avoir étudié avec votre prof préféré la solution à la question précédente :

1. Voyez-vous un problème dans cette implementation?
2. Pouvez-vous y apporter une solution?

Solution

1. The problem: this version of `SpecialSubject` did not provide implementations of any of the methods of `Observable`. As a result, it had to allow its clients to get a reference to its contained `Observable` object using the `getObservable()` method. This may have dangerous consequences. A rogue client could, for example, call the `deleteObservers()` method on the `Observable` object, and delete all the observers! Let's have `SpecialSubject` not expose its contained `Observable` object, but instead provide "wrapper" implementations of the `addObserver()` and `deleteObserver()` methods which simply pass on the request to the contained `Observable` object.
2. The solution

```
import java.util.Observer;

public class SpecialSubject2 extends ParentClass {
  private String name;
  private float price;
  private DelegatedObservable obs;
  public SpecialSubject2(String name, float price) {
    this.name = name;
    this.price = price;
    obs = new DelegatedObservable();
  }
  public String getName() {return name;}
  public float getPrice() {return price;}
  public void addObserver(Observer o) {
    obs.addObserver(o);
  }
  public void deleteObserver(Observer o) {
    obs.deleteObserver(o);
  }
  public void setName(String name) {
    this.name = name;
    obs.setChanged();
    obs.notifyObservers(name);
  }
  public void setPrice(float price) {
    this.price = price;
    obs.setChanged();
    obs.notifyObservers(new Float(price));
  }
}
```

QUESTION

Quelles modifications doit-on apporter au test?