

Les tests unitaires avec Java

Philippe PRADOS

pp@philippe.prados.name

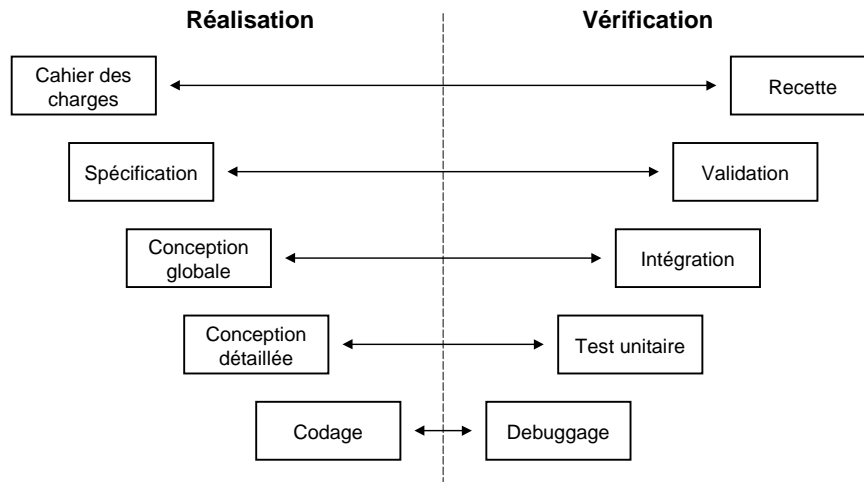
TABLE DES MATIERES

1.	Invariant, Pré et Postconditions.....	3
2.	Test unitaire	7
2.1	Méthodes.....	8
2.2	Méthodes redéfinissables	12
2.3	Héritage	13
2.4	Comment rédiger les tests unitaires	13
2.4.1	Équivalence	13
2.4.2	Inverse.....	14
2.4.3	Réversibilité.....	14
2.4.4	Ordres	14
2.4.5	Commutativité	14
2.4.6	Ensemble de valeur.....	14
2.4.7	Tests aux limites.....	15
2.5	Résumé.....	15
3.	Intégration	15
4.	Pour terminer	16

Avant-propos

Comment améliorer la qualité des logiciels ? En rédigeant des tests unitaires puissants. Ce document explique différentes techniques permettant de valider un programme java. Il propose une démarche objet des tests unitaires. Cela permet d'hériter des tests unitaires d'une classe pour rédiger les tests unitaires d'une sous-classe.

Pour valider un développement en Java, il faut utiliser l'approche traditionnelle. Le développement est découpé en plusieurs phases. Celles-ci sont de plus en plus précises. Chacune d'entre elles doit être validée.



Il n'est pas possible d'écrire du premier coup une classe importante sans que plusieurs erreurs apparaissent. Celles-ci peuvent venir d'une mauvaise utilisation du langage ou d'une résolution erronée ou incomplète des spécifications de la classe.

Les débogueurs ne permettent de détecter que les erreurs fatales. Les erreurs sur la valeur d'un attribut ou sur la sémantique d'une méthode ne peuvent pas être détectées par un débogueur. Tout au plus, vous pouvez parcourir le code pour constater l'appel erroné d'un service. Avec l'aide de quelques outils très simples, vous pouvez détecter un nombre très important d'erreurs que le débogueur ne trouvera pas, et cela, dès qu'elles arrivent.

1. INVARIANT, PRE ET POSTCONDITIONS

Chacune des méthodes d'une classe doit répondre à un contrat. Elles reçoivent des paramètres en entrées répondant à certaines contraintes, et elles doivent effectuer les traitements attendus par l'appelant. Pour cela, à la fin de celles-ci, il est envisageable de vérifier si les traitements se sont correctement effectués. Bertrand Meyer, le concepteur du langage objet « Eiffel », a introduit les trois principes suivants :

- pré-conditions, les conditions devant être remplies avant d'appeler une méthode,
- post-conditions, les conditions devant être présentes à la fin d'une méthode,
- invariants, les conditions étant toujours vraies.

Ces tests devant toujours être vrais sont des assertions. Les assertions servent :

- à la production de programme correct
- à enrichir la documentation
- sont une aide à la mise au point

L'appel d'une méthode est un contrat. Ce contrat peut par exemple se traduire comme ceci :

	Obligations	Bénéfice
Programmeur du client	N'appeler la routine que si $x \geq 0$, $\epsilon \geq 10^{-6}$	Obtenir en retour l'approximation de la racine carrée.
Rédacteur de la méthode	Renvoyer l'approximation demandée	Inutile de traiter les cas où $x < 0$ et $\epsilon \leq 10^{-6}$

Chacun des protagonistes doit respecter sa part du contrat. Les assertions sont écrites pour vérifier l'exécution de celui-ci. Il ne s'agit pas de vérifier lors de l'exécution les paramètres pour renvoyer un code d'erreur. Cela doit être codé classiquement. Une précondition permet juste-ment d'éviter de gérer les cas d'erreurs. Les assertions ne sont utiles que dans la phase de développement, pas dans la phase de production. L'utilisateur final ne doit pas être pénalisé par les tests d'assertions.

Le langage Eiffel est le premier langage qui offre dans sa syntaxe la prise en compte de ces principes. Les vérifications de ces conditions ne sont nécessaires que lors de la phase de débogage et d'intégration. Il faut pouvoir les supprimer lors des phases de validation et/ou de recette.

Nous allons créer une classe pour pouvoir facilement tester les invariants.

```

package dbg;

public final class Dbg
{
    private static boolean IsFatal_ = true;
    private static boolean IsAssert_ = true;
    private static boolean IsPreCondition_ = true;
    private static boolean IsPostCondition_ = true;

    public static class AssertionException extends RuntimeException
    {
        public AssertionException(String msg)
        {
            super(msg);
        }
    }

    private Dbg() { }

    private static void Fail(String s, Object x) throws AssertionException
    {
        if (s != null)
            System.err.print(s);
            System.err.println(x);
            if (IsFatal_)
                { throw new AssertionException(s + ' ' + x);
                } else
                { new Throwable().printStackTrace();
                }
    }

    public static final void Assert(boolean b) throws AssertionException
    { Assert(b, "Assertion fail");
    }

    public static final void Assert(boolean b, Object x)
        throws AssertionException
    {
        if (IsAssert_ && (!b))
            Fail(null, x);
    }

    public static final void Assert(boolean b, String s, Object x)
        throws AssertionException
    {
        if (IsAssert_ && (!b))
            Fail(s, x);
    }

    public static boolean IsAssert()
    { return IsAssert_;
    }

    public static boolean IsFatal()
    { return IsFatal_;
    }

    public static boolean IsPostCondition()
    { return IsPostCondition_;
    }

    public static boolean IsPreCondition()
    { return IsPreCondition_;
    }

    public static final void PostCondition(boolean b)
        throws AssertionException
    {
        if (IsPostCondition_)
            Assert(b, "Post-condition");
    }

    public static final void PostCondition(boolean b, Object x)
        throws AssertionException
    {
        if (IsPostCondition_)
            Assert(b, "Post-condition:", x);
    }

    public static final void PostCondition(boolean b, String s, Object x)
        throws AssertionException
    {
        if (IsPostCondition_)
            Assert(b, "Post-condition:" + s, x);
    }

    // Idem pour les preconditions...

    public static void SetAssert(boolean state)
    { IsAssert_ = state;
    }
}

```

```

    }
    public static void SetFatal(boolean aState)
    { IsFatal_ = aState;
    }
    public static void SetPostCondition(boolean state)
    { IsPostCondition_ = state;
    }
    public static void SetPreCondition(boolean state)
    { IsPreCondition_ = state;
    }
}

```

Avec cet outil, il est facile de vérifier les assertions, et de les débrancher si nécessaire. Si une assertion n'est pas vérifiée, le programme génère une exception. Cela peut être débranché à l'aide de la méthode `setFatal()`. Dans ce cas, l'erreur est affichée, mais le traitement continue. Dieu seul sait ce que le programme fera par la suite.

Après la phase de débogage, les « Post-conditions » et les « Invariants » peuvent être considérées comme corrects. Il suffit alors de modifier les drapeaux `IsAssertion` et `IsPostCondition` pour ne supprimer de l'exécution que ces deux tests. Les « pré-conditions » sont très importantes lors de la phase d'intégration. En effet, l'utilisation erronée de votre classe par les autres modules sera détectée à l'aide des pré-conditions. Le programme se testera de lui-même lors de son exécution. Qui mieux que lui peut le faire ?

Comment intégrer cela dans une classe ? Nous allons prendre un exemple simple. Prenons une classe `Adulte` ayant différents attributs.

```

public class Adulte
{ int _age;
  char _nom;
  String _prenom;
  public Adulte(String nom,String prenom,int age)
  { _age=age;
    _nom=nom;
    _prenom=prenom;
  }
}

```

Un adulte doit toujours avoir son âge compris entre dix-huit et cent cinquante ans. Pour le moment, il n'existe pas de personne ayant vécu plus de cent cinquante ans. L'invariant de la classe `Adulte` doit être écrit dans une méthode particulière. Celle-ci ne fera rien lors de la phase de recette.

```

void invariant()
{ Dbg.Assert ((18<=_age) && (_age<=150),"Age incorrect");
}

```

Le constructeur d'`Adulte` reçoit différents paramètres. Il est possible de vérifier ceux-ci en précondition.

```

public Adulte(String nom,String prenom,int age)
{ Dbg.PreCondition(nom!=null,"nom est à null !");
  Dbg.PreCondition(prenom!=null,"prenom est à null !");
  _age=age;
  _nom=nom;
  _prenom=prenom;
  invariant();
}

```

A la fin du constructeur, les invariants de la classe doivent être résolus. Chaque méthode autre que le constructeur peut commencer par appeler la méthode `invariant()` pour vérifier l'état courant de la classe. Il est préférable de découvrir les erreurs le plus tôt possible. Bien sûr, cela ralentit le programme. Tout dépend de la complexité de la méthode `invariant()`. Si par exemple, pour une relation `n..m` vous testez en `invariant()` si tous les objets en relations possèdent bien la relation inverse, cela est très consommateur en temps CPU. Dans ce cas, vous pouvez choisir de ne vérifier les invariants que dans les méthodes de modification.

La méthode `finalize()` doit vérifier avant toute destruction les invariants. C'est un passage obligé pour les utilisateurs de la classe, donc le lieu idéal pour vérifier la classe.

Vous pouvez également rendre public la méthode `invariant()`, ce qui permet aux appelants de vérifier l'objet. Une méthode `invariant()` peut ainsi vérifier l'état de son objet, et l'état de tous les objets en relation. Le test d'invariant peut être propagé dans tous les objets. La validation de l'invariant d'un objet entraîne le test des invariants de tous les attributs de l'objet et de tous les objets en agrégation. Au final, il existe un invariant de l'application elle-même qui consiste à vérifier les invariants de tous les objets de l'application.

Pour vérifier une post-condition, il est parfois nécessaire de comparer le résultat d'une méthode avec l'état de l'instance avant celle-ci. Ce qui peut se traduire en Java comme suit :

```

public class MaClass
{
  Object clone()
  { ...
  }
  void f()
  {
    Dbg.PreCondition (... )
    MaClass old;
  }
}

```

```

    if (Dbg.IsPreCondition())
    { old=(MaClass)clone();
    }
    // ... Traitement
    if (Dbg.IsPreCondition())
    { Dbg.PostCondition (x==old.x * 2,"Calcul errone");
    }
}
}

```

Lors de la rédaction des post-conditions, il n'est pas nécessaire de vérifier les traitements simples. Par exemple, il n'est pas utile de vérifier qu'une variable a bien la valeur que l'on vient d'y mettre. Il faut avoir un minimum de confiance dans le compilateur. Seuls les traitements complexes doivent être vérifiés. Par exemple, pour optimiser une méthode, il est utile de rédiger une version simple de l'algorithme qui sera facilement vérifiable mais lente. Cette version servira d'étalon à la version optimisée. En post-condition, il est important de vérifier que la routine obtient le même résultat que la version non optimisée. L'optimisation consiste à détecter des cas particuliers pour améliorer les traitements ou à modifier l'algorithme pour éviter les redondances. Dans les deux cas, il peut y avoir des fuites sur des combinaisons particulières. Certaines situations peuvent être dirigées vers un traitement par erreurs. La réduction de la redondance peut supprimer par erreur une redondance justifiée. Pour détecter ces situations, il faut comparer le résultat avec la version simple de l'algorithme. Pour écrire une version optimisée, il faut dans un premier temps écrire une version lente mais simple. Une fois la version lente écrite, le programme peut fonctionner sans rédiger la version rapide. Un utilitaire peut alors détecter les routines consommant beaucoup de temps. Vous devrez alors ne modifier que les routines critiques ayant été identifiées. Commencez toujours par écrire une version simple de l'algorithme. Au pire, elle servira à valider la version rapide.

Lors de la phase de débogage, vous devez déclarer :

	Au début	A la fin
Constructeur	Pré-condition	Invariant Post-condition
Méthode de consultation	(Invariant) Pré-condition	(Invariant) Post-condition
Méthode de modification	Invariant Pré-condition	Invariant Post-condition
Destructeur	Invariant Pré-condition	Post-condition

Une variante consiste à supprimer les Invariants des méthodes de consultation car elles ne modifient pas l'objet. En phase d'intégration, vous pouvez ne laisser que les pré-conditions car le corps de la classe est considérée comme validé par les tests unitaires. Si vous êtes pessimiste, continuez à utiliser tous les tests présents lors de la phase de débogage. Lors de la phase de validation ou de recette, tous les tests des assertions doivent être supprimés.

Lors de la rédaction d'une assertion, il ne faut pas appeler de méthodes de modification de la classe, car il ne doit pas y avoir d'effet de bord. De même, vous ne devez pas utiliser les mêmes variables que le corps de la classe. Encadrez le code de vérification d'un `if` et d'accolade.

```

if (Dbg.IsAssert())
{
    // test des assertions
    ...
}

```

Le langage Eiffel utilise un environnement extérieur à la fonction pour décrire les assertions. En Java, il faut un minimum de rigueur pour obtenir le même effet. Il ne faut pas qu'en enlevant le code d'une assertion, le programme ait un comportement erroné. Les assertions ne sont pas là pour corriger le code, mais pour le vérifier. Les critères d'optimisations ou de places mémoires ne doivent pas impacter la rédaction de ce code, car l'application finale ne vérifiera plus les assertions. Les assertions ne doivent qu'ajouter du code, mais pas le modifier.

Il est tentant, lors de la rédaction des post-conditions, de vouloir utiliser une autre méthode de la classe afin de confirmer le traitement par réversibilité. Par exemple, pour vérifier la méthode `ajoute()`, il semble intéressant de vérifier en post-condition, si en appelant la méthode `soustrait()` sur le résultat, on obtient bien la valeur avant le traitement.

```

class Fraction
{ // ...
    public Fraction ajoute(Fraction x)
    {
        Fraction old=new Fraction(this);
        //... Traitement
        Fraction tmp=new Fraction(*this);
        Dbg.PostCondition(tmp.soustrait(x).equals(old,"ajoute() faux");
        return this;
    }
}

```

Mais que va faire la méthode `soustrait()` ? De même, elle va appeler la méthode `ajoute()` pour vérifier son comportement. Nous sommes en présence d'une récursivité infinie. La méthode `ajoute()` appelle la méthode `soustrait()`, qui appelle la méthode `ajoute()`, qui appelle etc.

Il ne faut pas vérifier la réversibilité d'une méthode en post-condition. Ce sera fait en test unitaire. Par contre, certaines méthodes peuvent judicieusement appeler d'autres méthodes pour vérifier les post-conditions. Ces méthodes ne doivent pas modifier l'instance. Par exemple, le constructeur de copie peut vérifier son comportement à l'aide de la méthode `equals()`.

```
class Fraction
{
    public Fraction(Fraction x)
    { //... Traitement
      Dbg.PostCondition>equals(x),"Constructeur de copie faux");
    }
}
```

Les post-conditions doivent être rédigées avec rigueur, elles doivent être valides quelles que soient les conditions d'appel. Pour les tests aux limites ou les tests de réversibilité, utilisez les tests unitaires. Peut-être qu'à l'avenir, Java possédera une extension gérant syntaxiquement les invariants.

2. TEST UNITAIRE

Le test unitaire est la première étape de validation d'un développement. Il ne faut pas confondre le test unitaire et le débogage. Le test unitaire cherche à prouver qu'une classe est correcte, le débogage permet de localiser une erreur. Le test unitaire est un complément des pré-conditions et des invariants.

Les tests unitaires sont des exemples d'utilisation correcte de la classe. Ils vérifient que la classe fonctionne si on l'utilise correctement. Pour des scénarios d'utilisation « valide », si une précondition, une post-condition ou un invariant échoue, la classe et/ou le test sont erronés. Un test unitaire vérifiera qu'un appel à une méthode avec un jeu de paramètres connus donne le résultat attendu. Dans les tests unitaires, il faut également tester les méthodes aux limites.

Les tests unitaires permettent de vérifier la non-régression d'un développement. Avant chaque nouvelle intégration d'une classe, les tests unitaires doivent être refaits pour vérifier que la nouvelle version ne régresse pas par rapport à la précédente.

Quand concevoir les tests unitaires ? Dès la phase de conception. En effet, la rédaction des tests unitaires fait apparaître des erreurs possibles qui pourront influencer la modélisation. Plus tôt les erreurs sont trouvées, moins l'impact est important. La correction de certaines erreurs peut entraîner une modification importante de l'interface ou de la modélisation d'une classe.

« Si vous n'avez pas la patience de tester votre programme, celui-ci testera votre patience ! »

Pour le modèle objet, il semble naturel d'effectuer les tests unitaires sur les classes. Il faut tester toutes les méthodes d'une classe et les transitions d'état de celle-ci. Nous allons prendre un exemple de classe dont nous voulons vérifier la conformité avec ses spécifications. Testons la classe `Magnetophone`. Celle-ci possède un état lui permettant d'autoriser certaines méthodes.

```
import dbg.*;
class Magnetophone
{
    static public final int STOP=0;
    static public final int START=1;
    static public final int REWIND=2;

    public void start()
    { Dbg.PreCondition(cassette_==true,"Pas de cassette");
      Dbg.PreCondition(etat_==STOP,"Deja en marche");
      etat_=START;
    }

    public void stop()
    { etat_=STOP;
    }

    public void ajouterCassette()
    { Dbg.PreCondition(cassette_==false,"Deja une cassette");
      cassette_=true;
    }
    public void enleverCassette()
    { Dbg.PreCondition(cassette_==true,"Il n'y a pas de cassette");
      cassette_=false;
    }
    public boolean getCassette()
    { return cassette_;
    }

    public int getEtat()
    {
      return etat_;
    }
    private int etat_=STOP;
}
```

```
protected boolean cassette_=false;
}
```

Généralement, on rédige les tests unitaires dans une méthode `main()` de la classe. Cette méthode est modifiée au fur et à mesure de la rédaction de la classe. Lors de la rédaction des tests unitaires d'une classe dérivée de `Magnetophone`, on ne peut pas bénéficier des tests unitaires de la super classe. Pour éviter cela, est normaliser la rédaction des tests unitaires, il faut procéder différemment.

2.1 Méthodes

Dans un premier temps, nous allons tester les méthodes publiques de la classe. Les tests unitaires ne sont pas nécessaires à l'application finale. Pour pouvoir facilement les supprimer lors de l'intégration de l'application, nous créons une nouvelle classe de même nom que la classe à tester, suffixer par `UnitTest`.

Cette nouvelle classe permet de faire un test en boîte noire. Elle est dans la même situation qu'un client de la classe à tester. Elle ne peut accéder qu'aux méthodes publiques.

```
class MaClass
{
...
}
class MaClassUnitTest
{
...
public static void main(String[] args)
{
...
}
}
```

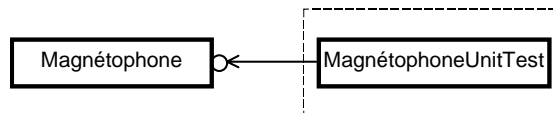
Pour avoir un test en boîte blanche, il faut rédiger une inner classe appelée `UnitTest`. Cette classe interne peut consulter les attributs privés.

```
class MaClass
{
...
static class UnitTest
{
...
public static void main(String[] args)
{
...
}
}
}
```

La classe de test s'occupera de tester les méthodes. Elle maintiendra l'état courant du test, et l'état de tous les objets testés.

Lors du déploiement, pour supprimer tous les tests unitaires, il faut supprimer les fichiers `*UnitTest*.class`.

Pour pouvoir appeler une méthode, il faut dans un premier temps créer un objet. La classe `UnitTest` gardera l'information de la création de l'instance. Ainsi, le test d'une méthode pourra vérifier auparavant que l'instance est bien présente.



```
public class Magnetophone
{
//...
}
public static class MagnetophoneUnitTest
{
protected Magnetophone magneto_=null;
public static void main(String[])
{
...
}
}
```

La class utilitaire `UnitTest` suivante, propose les outils nécessaires aux tests unitaires.

```
package dbg;

import java.lang.reflect.*;

public class UnitTest
{
public static final UnitTest Ok = new UnitTest();
public static final UnitTest Bad = new UnitTest();
public static final UnitTest PreCondition = new UnitTest();

public static Method[] toMethods(Class base, String[] names)
throws RuntimeException
{
try
```

```

    {
        Method[] rc = new Method[names.length];
        for (int i = 0; i < names.length; ++i)
        {
            rc[i] = base.getDeclaredMethod(names[i], new Class[] {});
        }
        return rc;
    } catch (NoSuchMethodException x)
    {
        throw new RuntimeException("Method not found : " + x);
    }
}
}
public String toString()
{
    if (this == Ok)
        return "Ok";
    else
        if (this == Bad)
            return "Bad";
        else
            return "PreCondition";
}
//...
}

```

Les tests unitaires seront implantés à l'aide de méthodes renvoyant trois types de valeur :

- test OK
- test erroné
- et condition de test incorrecte.

Pour effectuer un test, il faut que certaines conditions soient réunies. Chaque méthode de test vérifiera les conditions courantes et effectuera les appels aux méthodes de l'objet. Elles vérifieront les réponses de celles-ci suivant les valeurs attendues en sortie par rapport aux valeurs d'entrée. Les transitions nécessaires sur l'objet `Magnetophone` seront également vérifiées. La plupart des méthodes de tests renverront par défaut la valeur `Ok` car les vérifications auront été faites dans les post-conditions des méthodes testées. Parfois, une post-condition ne peut pas tester la validation d'une méthode car le résultat dépend d'un contexte qu'elle ne maîtrise pas. Dans ce cas, la méthode de test vérifiera le résultat attendu.

```

import dbg.*;
import java.lang.reflect.*;

public class MagnetophoneUnitTest
{
    protected Magnetophone magneto_=null;

    // Test de la construction
    public UnitTest ctrl()
    {
        if (magneto_!=null) return UnitTest.PreCondition;
        magneto_=new Magnetophone();
        if (magneto_.getCassette()) return UnitTest.Bad;
        return UnitTest.Ok;
    }

    // Test de la destruction
    public UnitTest dtrl()
    {
        if (magneto_==null) return UnitTest.PreCondition;
        magneto_=null;
        System.gc();
        return UnitTest.Ok;
    }

    // Test methode Start dans n'importe quel etat
    public UnitTest startl()
    {
        if (magneto_==null) return UnitTest.PreCondition;
        if (!magneto_.getCassette())return UnitTest.PreCondition;
        if (magneto_.getEtat()!=Magnetophone.STOP)
            return UnitTest.PreCondition;
        magneto_.start();
        return UnitTest.Ok;
    }

    // Test methode Stop dans n'importe quel etat
    public UnitTest stopl()

```

```

{
    if (magneto_==null) return UnitTest.PreCondition;
    magneto_.stop();
    return UnitTest.Ok;
}

// Test methode Stop dans l'état Start
public UnitTest stop2()
{
    if (magneto_==null) return UnitTest.PreCondition;
    if (magneto_.getEtat()!=Magnetophone.START)
        return UnitTest.PreCondition;
    magneto_.stop();
    return UnitTest.Ok;
}

// Test methode ajouteCassette
public UnitTest ajouterCassette1()
{
    if (magneto_==null) return UnitTest.PreCondition;
    if (magneto_.getCassette()) return UnitTest.PreCondition;
    magneto_.ajouterCassette();
    if (!magneto_.getCassette()) return UnitTest.Bad;
    return UnitTest.Ok;
}

// Test methode enleverCassette
public UnitTest enleverCassette1()
{
    if (magneto_==null) return UnitTest.PreCondition;
    if (!magneto_.getCassette()) return UnitTest.PreCondition;
    magneto_.enleverCassette();
    if (magneto_.getCassette()) return UnitTest.Bad;
    return UnitTest.Ok;
}
}

```

Toutes les méthodes de test vérifient le contexte courant. Elles effectuent le test et vérifient le comportement des méthodes de `Magnetophone`. Il n'est pas toujours possible de vérifier le comportement d'une méthode à l'extérieur de la classe. Les post-conditions s'occuperont de cela. Le test unitaire est un exemple d'utilisation de la classe. La méthode `start1()` appelle la méthode correspondante de la classe `Magnetophone`, mais est incapable de vérifier si le comportement de la méthode est correct. Par contre, la méthode `ajouteCassette1()` vérifie le comportement de la méthode.

Il est possible, par la suite, d'écrire des scénarios de test de la classe `Magnetophone`. Pour cela, il suffit d'appeler successivement différentes méthodes de tests. Les scénarios peuvent eux-mêmes être des tests de la classe `Magnetophone`.

```

public class MagnetophoneUnitTest
{ //...
    // Scenario 1
    public UnitTest scenariol()
    {
        if (magneto_==null)                return PreCondition;
        UnitTest rc=Ok;
        if ((rc=ctrl())!=Ok)                return rc;
        if ((rc=ajouterCassette1())!=Ok)    return rc;
        if ((rc=start1())!=Ok)              return rc;
        if ((rc=stop2())!=Ok)               return rc;
        if ((rc=start1())!=Ok)              return rc;
        if ((rc=stop1())!=Ok)               return rc;
        rc=dtrl();
        return rc;
    }
}

```

Les tests des méthodes étant tous faits sous le même format, il est possible de décrire un scénario en indiquant uniquement le nom des méthodes de test à appeler et dans quel ordre. Pour cela, on ajoute deux méthodes à la classe `UnitTest`.

```

public class UnitTest
{
    //...
    public static UnitTest RunScenario(Method[] scenario, Object test)
        throws Throwable
    {
        try
        {
            UnitTest rc = Ok;
            for (int i = 0; i < scenario.length; ++i)
            {

```

```

        if ((rc = (UnitTest) scenario[i].invoke(test, null)) != Ok)
            return rc;
    }
    return Ok;
} catch (IllegalAccessException x)
{
} catch (InvocationTargetException x)
{
    throw x.getTargetException();
}
return Bad;
}
public static UnitTest RunScenario(Class cl, String[] scenario,
    Object test)
    throws Throwable
{
    return RunScenario(toMethods(cl, scenario), test);
}
}

```

Il suffit ensuite de construire un tableau de méthode et d'appeler `RunScenario()`.

```

static Method[] methodsScenario2=
    UnitTest.toMethods(MagnetophoneUnitTest.class,
        new String[]
        {
            "ctrl",
            "start1",
            "ajouterCassette1",
            "stop2",
            "start1",
            "dtrl"
        }
    );
public UnitTest scenario2()
    throws Throwable
{
    return UnitTest.RunScenario(methodsScenario2, this);
}

```

La méthode `toMethods()` construit un tableau de méthode à partir d'un tableau de `String`.

Il est également envisageable d'effectuer des tests aléatoires en s'appuyant sur l'état `TstCondition` des méthodes. Un tirage aléatoire choisit un test ou un scénario au hasard, puis celui-ci est exécuté. Si le test retourne l'état `TstCondition`, celui-ci est considéré comme n'ayant pas pu être fait, et un nouveau tirage est effectué. Une boucle de durée ou d'itération finie peut tester la classe dans les contextes les plus variés. Nous ajoutons pour cela les méthodes `RunTorture()` à la classe `UnitTest`.

```

public class UnitTest
{
    //...
    public static UnitTest RunTorture(Method[] scenario, Object test,
        int loop)
        throws Throwable
    {
        try
        {
            UnitTest rc = Ok;
            for (int i = 0; i < loop; ++i)
            {
                do
                {
                    int idx = (int) (Math.random() * scenario.length);
                    rc = (UnitTest) scenario[idx].invoke(test, null);
                } while (rc == PreCondition);
                if (rc == Bad)
                    return Bad;
            }
            return Ok;
        } catch (IllegalAccessException x)
        { } catch (InvocationTargetException x)
        {
            throw x.getTargetException();
        }
        return Bad;
    }
    public static UnitTest RunTorture(Class cl, String[] scenario,
        Object test, int loop)
        throws Throwable
    {
        return RunTorture(toMethods(cl, scenario), test, loop);
    }
}

```

Il faut alors construire un tableau de méthode et appeler `RunTorture()` en indiquant un nombre d'itérations.

```

static Method[] methodsTorture=
    UnitTest.toMethods(MagnetophoneUnitTest.class,
        new String[]
        {
            "ctrl",

```

```

        "dtrl",
        "start1",
        "stop1",
        "stop2",
        "ajouterCassette1",
        "enleverCassette1",
    });

public UnitTest torture()
    throws Throwable
{ return UnitTest.RunTorture(methodsTorture,this,100);
}

```

Le torture-test est très utile pour les classes techniques. Cela permet de vérifier profondément les couches techniques de l'application.

Il est également possible d'envisager un parcours exhaustif de toutes les transitions des états d'une classe, en rédigeant un test pour chaque changement d'état. Un automate parcourt ensuite l'ensemble des transitions possibles en appelant successivement les tests correspondants.

Pour tester une classe en condition multi-tâches, il faut offrir la méthode `RunScenarioMT`.

```

class UnitTest
{ ...
    public static UnitTest RunScenarioMT(final Method[] scenario,
                                        final Object test,
                                        int nbThread)

        throws Throwable
    {
        Thread[] task = new Thread[nbThread];
        for (int i = 0; i < task.length; ++i)
        {
            final int num = i;
            task[i] = new Thread(new Runnable()
            {
                public void run()
                {
                    try
                    {
                        RunScenario(scenario,test);
                    } catch (Throwable x)
                    {
                        x.printStackTrace();
                    }
                }
            });
            task[i].start();
        }
        return UnitTest.Ok;
    }
}

```

Avec cet outil, il est possible de lancer des scénarios simultanément dans plusieurs tâches. Pour désynchroniser les tâches, il faut ajouter une méthode `sleep()` dans le script.

```

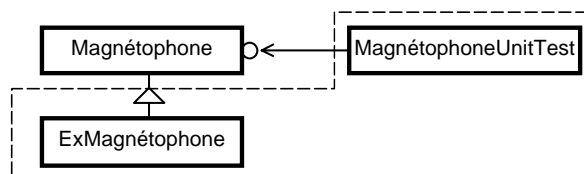
UnitTest sleep()
{
    try { Thread.sleep((int) (1000 * Math.random())); }
    catch (InterruptedException x) { }
    return UnitTest.Ok;
}

```

L'utilisation multi-tâches combiné avec un torture-test permet de démontrer que la classe est solide.

2.2 Méthodes redéfinissables

Pour tester les méthodes non `final`, il faut vérifier leurs comportements si une classe dérivée modifie celles-ci. Pour cela, il faut déclarer une classe dérivée et redéfinir toutes les méthodes. Les méthodes peuvent individuellement ou collectivement changer leurs comportements selon l'état de cette nouvelle classe.



```

public static class MagnetophoneUnitTest
{ //...

```

```

static class ExMagnetophone extends Magnetophone
{
    boolean original_=false;
    public void chgMode(boolean original)
    {
        original_=original;
    }
    public void start()
    {
        if (original_) super.start();
        else
        { // Modification du comportement
        }
    }
    public void stop()
    {
        if (original_) super.stop();
        else
        { // Modification du comportement
        }
    }
    //...
}
}

```

La méthode `chgMode()` permet de basculer l'objet d'une version normale vers une version avec les méthodes modifiées. Les tests unitaires des méthodes de la classe doivent continuer à être valides avec ou sans modification. Il est pertinent de tester la classe `Magnetophone` seule, puis de parcourir les mêmes tests avec la classe `ExMagnetophone`.

```

public static class MagnetophoneUnitTest
{
    //...
    // Test de la construction
    public UnitTest ctr2()
    {
        if (magneto_!=null) return UnitTest.PreCondition;
        magneto_=new ExMagnetophone();
        if (magneto_.getCassette()) return UnitTest.Bad;
        return UnitTest.Ok;
    }
}

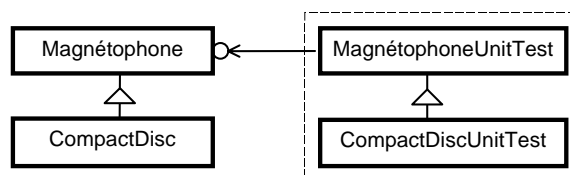
```

Le test `ctr2()` construit un objet `ExMagnetophone` en demandant la modification des méthodes virtuelles. Celles-ci peuvent modifier l'automate d'état. Dans ce cas, les tests doivent être adaptés pour tenir compte de ces changements.

Les classes abstraites peuvent également être testées par ce mécanisme. Les méthodes `abstract` seront redéfinies dans la classe `ExMagnetophone`.

2.3 Héritage

Si une classe `CompactDisc` hérite de `Magnetophone`, celle-ci doit également être validée par un test unitaire. Elle doit être capable de répondre correctement à une majorité des tests de la classe `Magnetophone`. Nous allons construire une classe `CompactDiscUnitTest` qui hérite de la classe `MagnetophoneUnitTest`. Les tests supplémentaires peuvent alors être ajoutés pour les nouvelles méthodes de `CompactDisc`, et les tests hérités peuvent être inclus dans les nouveaux scénarios.



Certains tests de `MagnetophoneUnitTest` peuvent être adaptés dans la classe `CompactDiscUnitTest` en redéfinissant les méthodes.

Les mêmes principes, suite aux différents types de méthodes, sont à utiliser pour tester correctement la classe `CompactDisc`. Au fur et à mesure de l'enrichissement du modèle, les classes héritées bénéficieront des tests des classes de base. Si par la suite la classe `Magnetophone` évolue, les tests correspondants seront modifiés. La classe `CompactDiscUnitTest` bénéficiera de ces modifications.

2.4 Comment rédiger les tests unitaires

Lors de la rédaction des tests unitaires, il faut vérifier les différents cas typiques d'utilisation de la classe, mais également les cas exceptionnels. Les tests aux limites seront présents ici. Il faut vérifier plusieurs scénarios de comportements.

2.4.1 Équivalence

Les tests d'équivalence permettent de vérifier que plusieurs méthodes ont un comportement équivalent. Cela permet de vérifier simultanément deux ou plusieurs méthodes. Par exemple, pour toutes valeurs `a` et `b`, les tests suivants doivent être résolus :

```

(a+b) == (a+=b)
(a-b) == (a-=b)

```

```

(a*b) == (a*=b)
(a/b) == (a/=b)
(a<b) == (b>a)
(a>b) == (b<a)
...

```

Des méthodes équivalentes doivent être testées l'une par rapport à l'autre. Par exemple, vérifier la méthode `equals()` et le constructeur de copie : `new A(a).equals(a)`. De même, la méthode `hashCode()` peut être validée à l'aide de la méthode `equals()`.

Si `x.equals(y)==true` alors `x.hashCode()==y.hashCode()`.

2.4.2 Inverse

Les tests d'*inverse* vérifient que les opérations contradictoires le sont bien. Pour toutes valeurs a et b, les tests suivants doivent être résolus :

```

(a<b) != (a>=b)
(a>b) != (a<=b)
(a==b) != (a!=b)
(!a) != (a)
...

```

2.4.3 Réversibilité

Les tests de *réversibilité* vérifient qu'une opération est réversible. Cela est très utile pour vérifier les commandes « Undo ». Les opérateurs arithmétiques sont également réversibles. Pour toutes valeurs a et b, les tests suivants doivent être résolus :

```

a == (a+b-b)
a == (a-b+b)
a == (a*b/b)
a == (a/b*b)
a == (a+=b, a-=b)
a == (!(!a))
...

```

2.4.4 Ordres

Les tests d'*ordre* vérifient qu'une opération impacte correctement les relations d'ordre entre les objets. Pour toutes valeurs a, b et c, les tests suivants doivent être résolus :

```

si b>=0 alors a<=(a+b)
si b<=0 alors a>=(a-b)
(a<c) == (a<b && b<c)
...

```

2.4.5 Commutativité

Les tests de *commutativité* vérifient que l'ordre de rédaction d'une opération n'a pas d'impact sur le résultat. Pour toutes valeurs a et b, les tests suivants doivent être résolus :

```

(a+b) == (b+a)
(a*b) == (b*a)
...

```

Un exemple type de cette situation est la méthode `equals()`. On doit avoir `a.equals(b)==b.equals(a)`.

2.4.6 Ensemble de valeur

Les tests d'*équivalence* vérifient des valeurs caractéristiques d'un ensemble de valeurs. Des tranches de valeurs peuvent être identifiées comme ayant le même comportement. Par exemple, tester une valeur négative peut être représentatif de toutes les valeurs négatives. Tester une valeur positive peut être également représentatif des valeurs positives. Les valeurs inférieures et supérieures aux limites peuvent également être testées par un représentant de ces valeurs.

	-11	-10	-1	+1	+10	+11	
Limite inférieure			Valeurs négative	Valeurs positive			Limite supérieure

Tester un représentant de chaque partition permet de tester l'ensemble des valeurs.

2.4.7 Tests aux limites

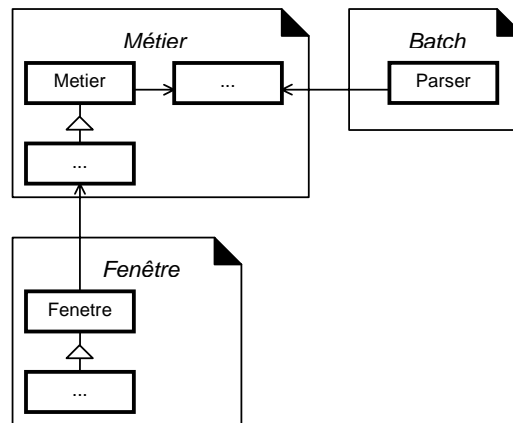
Les *tests aux limites* vérifient le comportement des méthodes avec des valeurs limites. Cela permet d'adapter les pré-conditions pour tenir compte des valeurs initiales valides entraînant un résultat final correct. La plupart des expressions de tests des chapitres précédents fonctionnent avec des valeurs normales pour les types de base du compilateur, mais échouent avec des valeurs limites. Par exemple, l'expression « $a \leq (a+b)$ » n'est pas vraie si $a+b$ est supérieure à `Integer.MAX_VALUE`. Une précondition de l'opérateur d'addition d'un entier devrait vérifier si le résultat attendu est compatible avec la valeur maximale d'un entier. Les tests unitaires vérifieront les expressions précédentes avec des valeurs courantes, puis les mêmes tests seront effectués avec des valeurs aux limites.

2.5 Résumé

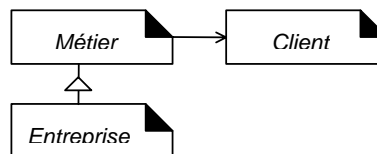
Les tests unitaires permettent de vérifier la rédaction des classes. Lors d'un développement avec une équipe importante, la plupart des classes doivent être simulées. Lors de l'intégration, les tests unitaires doivent être refaits après chaque ajout de nouveaux composants logiciels. Les erreurs éventuelles seront ainsi très rapidement localisées. La rédaction des tests unitaires permet au développeur de vérifier sa classe et également de savoir si les erreurs apparues en intégration viennent de celle-ci ou des autres composants logiciels précédemment simulés.

3. INTEGRATION

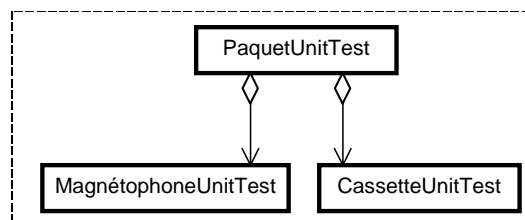
Après avoir passé avec succès les tests unitaires de chaque classe, il faut ensuite intégrer celles-ci dans des « paquets de classes ». Un paquet de classes est un ensemble de classes formant une couche logicielle. C'est une sorte de librairie. Cela peut correspondre à un `package`. Un paquet de classes est un regroupement de classes en relations fortes entre elles. Par exemple, les classes métier forment un paquet de classe. Ce paquet peut être utilisé par différents paquets qui offriront différentes interfaces aux classes du métier. Un paquet pourra par exemple offrir une interface fenêtre, et un autre offrira un langage auteur permettant de manipuler par un traitement par lot les objets du métier.



Un paquet peut lui-même utiliser d'autres paquets. Dans ce cas, il dépend de ceux-ci. Il existe des liens d'utilisations entre paquets. Un paquet peut également hériter d'un autre paquet si une de ces classes hérite d'une classe d'un autre paquet.

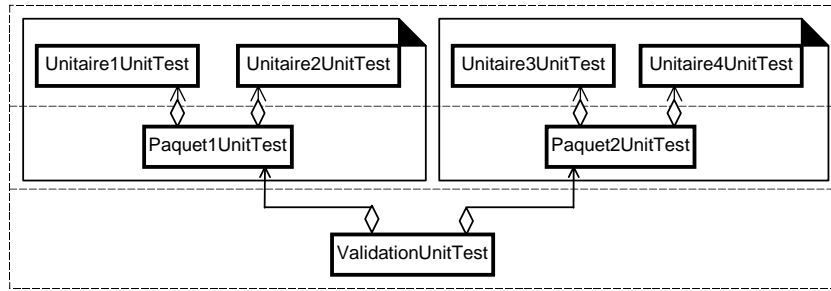


Lors des tests unitaires, les classes non encore intégrées sont simulées. Maintenant, il faut réunir les vraies classes et vérifier que les tests unitaires continuent à fonctionner. Il est possible qu'il faille modifier légèrement les tests pour pouvoir placer les classes précédemment simulées dans le bon état. Après avoir réuni les classes d'un paquet, il faut vérifier individuellement chaque classe, puis vérifier l'ensemble des classes du paquet. Pour cela, nous allons utiliser les tests unitaires de chaque classe. Nous allons par exemple déclarer la classe `MagnetophoneUnitTest` pour tester la classe `Magnetophone`, et la classe `CassetteUnitTest` pour tester la classe `Cassette`. Nous allons écrire un test du paquet de classe agrégeant deux instances de test unitaire.



Cela permet de bénéficier du contexte des deux tests. Il faut ensuite ajouter de nouveaux scénarios s'appuyant sur les tests de ces deux classes mais combinant les interactions entre celles-ci. Il est possible d'écrire un test d'un paquet de classes ayant son propre contexte qui n'a rien à voir avec les contextes des tests unitaires. Les scénarios des tests de paquet de classes utilisent la même technique que les tests unitaires.

Cette technique est applicable également lors de l'intégration de paquet de classes. Un test de l'intégration de plusieurs paquets de classes pourra agréger des tests de chacun des paquets. Un test de validation pourra agréger des tests de chacun des paquets de classes.



Une hiérarchie parallèle aux classes de l'application va progressivement être rédigée, permettant de tester sérieusement toute l'application.

4. POUR TERMINER

Une fois l'exécutable terminé, il faut procéder aux tests *a posteriori*. Il faut dans un premier temps effectuer les tests fonctionnels, c'est-à-dire chercher à piéger le programme par une utilisation de celui-ci. Dans un deuxième temps, il faut effectuer un test de « stress ». Vérifiez que le programme supporte les charges prévues et a un comportement correct aux limites des ressources. Si la mémoire vient à manquer, le programme tolérera-t-il cela sans perte d'information ? Si le réseau est interrompu, les erreurs seront-elles correctement détectées et gérées ?

Il est également possible de rédiger des scénarios d'utilisations permettant de vérifier la non-régression de l'application.