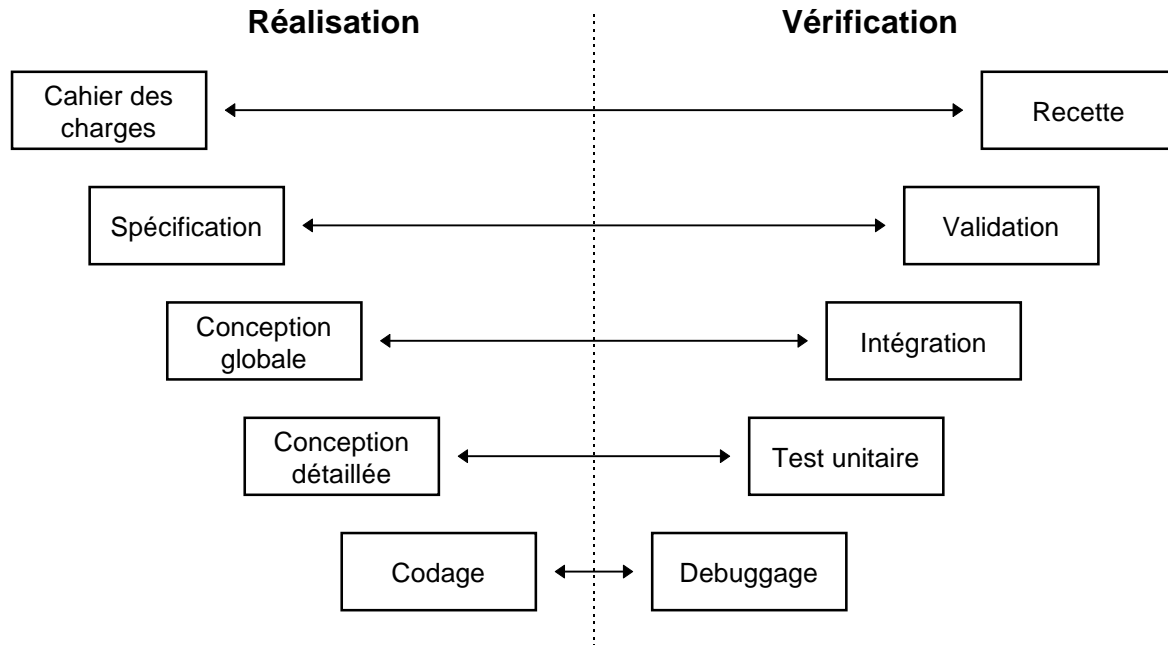


# Les tests

Pour valider un développement en C++, 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.

Le débogage est une phase difficile du C++. Il faut posséder de très bons outils pour pouvoir suivre précisément toutes les étapes cachées par le compilateur. Il n'est pas raisonnable d'ajouter des traces par-ci, par-là, pour vérifier les différents appels. La compilation est très consommatrice en temps CPU car le compilateur effectue énormément de travail. L'édition de lien est également très coûteuse. Recompiler successivement pour n'ajouter que quelques traces, n'est pas envisageable.

Certains outils peuvent être écrits facilement, d'autres devront être achetés. Un bon débogueur, spécifique au C++, est nécessaire. 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.

## 1.1 Invariant, Pré 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 :

- préconditions, les conditions devant être remplies avant d'appeler une méthode,
- postconditions, 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 cela :

	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 ou $x < 0$ , ou $\epsilon < 10^{-6}$

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 justement 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. En C++, il est facile de traduire cela. Le préprocesseur va nous aider.

Le C ANSI déclare une macro appelée `assert()` permettant de vérifier une condition devant être vraie. Si celle-ci n'est pas vérifiée, le programme est interrompu et un message d'erreur indique le fichier et la ligne où l'erreur s'est produite. Cette macro peut être écrite comme cela :

```
#define                                assert(TST)
\
    ((TST)                               ?                               (void)0
\
    :   (cerr << __FILE__ " (" << __LINE__
\
    << "): Assertion failed " #TST
\
    << endl,abort()))
```

La norme C++ demande à ce que `assert()` soit une expression. L'utilisation de celle-ci est extrêmement simple.

```
void                               f(char*                               pt)
{   assert(pt!=NULL); //           abort()                               si           pt==NULL
    //...
}
```

Un `#define` particulier permet de supprimer l'ensemble des appels à `assert()` lors de la compilation avant la recette. Si vous déclarez `#define NDEBUG`, tous les `assert` sont supprimés.

```

#ifndef NDEBUG
#define assert(TST) ...
#else
#define assert(TST) ((void)0)
#endif

```

Si vous désirez effectuer un traitement particulier pour vérifier les conditions d'un traitement, encadrer celui-ci par :

```

#ifndef NDEBUG
// Traitement spécial de vérification du programme
#endif

```

Nous allons utiliser cet outil pour écrire des macros spécifiques de vérification des invariants, des pré et postconditions.

```

#include <assert.h>
#define INVARIANT(TST) assert(TST)
#define PRECONDITION(TST) assert(TST)
#define POSTCONDITION(TST) assert(TST)

```

Une version étendue peut recevoir un message explicatif supplémentaire.

```

#define ASSERTMSG(TST,MSG)
\
\ ((TST) ? (void)0
\
\ : (cerr << __FILE__ "(" << __LINE__
\
\ << "): Assertion failed " #TST
\
\ << MSG << endl,abort()))

```

On constate l'avantage du C++ sur le C. En effet, il est possible d'écrire :

```

ASSERTMSG(i>100,"Valeur trop petites");
ASSERTMSG(pt!=NULL,"this=" << *this);

```

Si les opérateurs de flux ont été écrits pour chaque objet, il est très facile d'afficher l'état de ceux-ci.

Les macros de tests deviennent :

```

#define INVARIANT(TST,MSG) ASSERTMSG(TST,"Invariant " <<
MSG)
#define PRECONDITION(TST,MSG) ASSERTMSG(TST,"Precondition "
<<
MSG)
#define POSTCONDITION(TST,MSG) ASSERTMSG(TST,"Postcondition "
<< MSG)

```

Après la phase de débogage, les " Postconditions " et les " Invariants " peuvent être considérés comme corrects. Il suffit alors d'adapter les macros précédentes, pour ne supprimer de la compilation que ces deux tests. Les " Préconditions " 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 préconditions. 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 CAdulte ayant différents attributs.

```

class CAdulte
{
    int _age;
    char _nom[40];
    char _prenom[40];
}

```

```

public:
    CAdulte(const char* nom,const char* prenom,int age)
    :                               _age(age)
    {                               strcpy(_nom,nom);
      strcpy(_prenom,prenom);
    }
};

```

Un adulte doit toujours avoir son âge compris entre 18 et 150 ans. Pour le moment, il n'existe pas de personne ayant vécu plus de 150 ans. L'invariant de la classe `CAdulte` doit être écrit dans une méthode particulière. Celle-ci sera vide lors de la phase de recette.

```

#ifndef NDEBUG
void invariant() const
{ INVARIANT((_age>=18) && (_age<=150), "Age incorrect");
}
#else
void invariant() const {}
#endif

```

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

```

CAdulte(const char* nom,const char* prenom,int age)
:                               _age(age)
{ PRECONDITION(strlen(nom)<sizeof(_nom)-1,"Nom trop grand");
  PRECONDITION(strlen(prenom)<sizeof(_prenom)-1,"Prenom trop
grand");
  strcpy(_nom,nom);
  strcpy(_prenom,prenom);
  POSTCONDITION(!strcmp(_nom,nom),"Erreur sur le nom");
  POSTCONDITION(!strcmp(_prenom,prenom),"Erreur sur le
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  $0..n \leftrightarrow 0..n$  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 non constantes. Les méthodes constantes ne devant pas modifier l'objet, il ne devrait pas y avoir d'effet de bord.

Le destructeur 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 relation. Au final, il existe un invariant de l'application elle-même qui consiste à vérifier les invariants de tous les objets de l'application.

Il est difficile de vérifier toutes les préconditions dans les constructeurs. En effet, ceux-ci commencent par appeler les constructeurs des classes de bases, et les constructeurs de leurs

attributs. Les préconditions ne peuvent être rédigées que dans les accolades. Certains traitements sont alors déjà effectués, peut-être de façon erronée. Le constructeur ne pourra que constater, après coup, l'exécution erronée.

```
class A
{
    B _b;
public:
    A(int b) : _b(b)
    { PRECONDITION(b>=0,"b positif"); // B::B() déjà appelé !
    }
    //...
};
```

Le test de précondition est effectué après la construction de l'attribut `_b`. Celui-ci n'aurait jamais dû être construit. Mais, si un entier négatif est valide pour cet objet, le constructeur de `_b` ne réagira pas. Sinon, la précondition de `B::B(int)` aurait refusé la construction. La précondition du constructeur de `A` sera testée un peu trop tard mais sera quand même détectée. Dans de très rares cas, il est possible qu'un traitement erroné soit exécuté avant le test de la précondition.

Pour vérifier une postcondition, 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 C++ comme suit :

```
void A::f()
{
    PRECONDITION(...)
#ifdef NDEBUG
    A old(*this); // Constructeur de copie
#endif
    // ... Traitement
    POSTCONDITION(x==old.x * 2,"Calcul errone");
}
```

Il est parfois difficile de vérifier les postconditions des méthodes si celles-ci retournent des objets.

```
A B::getA()
{ return f(); // cctr
}
```

Comment tester que la fonction `f()` retourne un objet répondant aux postconditions ? Il n'est pas correct de garder le résultat de `f()` dans une variable de la méthode pour ensuite la renvoyer, car cela n'est pas efficace.

```
A B::getA() const
{ A tmp=f(); // cctr
  POSTCONDITION(A==1,"A doit être égal à 1");
  return tmp; // cctr
}
```

Avec ce type d'écriture, il y a deux appels au constructeur de copie à la place d'un seul. Une écriture alternative serait celle-là :

```
A B::getA() const
{
#ifdef NDEBUG
    A tmp=f();
    POSTCONDITION(A==1,"A doit être égal à 1");
    return tmp;
#else
```

```

return                                     f();
#endif
}

```

L'inconvénient de cette approche est que le code est découpé en deux versions distinctes. Il n'est pas utile de vérifier la postcondition d'une version ne devant pas être exécutée en phase d'intégration. Les assertions doivent vérifier le vrai code. Il est parfaitement envisageable, que la version vérifiant la postcondition soit correcte, mais pas la version sans le test de postcondition. Seul un langage spécialisé peut vérifier au bon moment les assertions. Avec le C++ actuel, vous pouvez tester la plupart des assertions, mais pas toutes.

Lors de la rédaction des postconditions, 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 postcondition, 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 de traitement 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 debugage, vous devez déclarer :

	Au début	A la fin
Constructeur	Précondition	Invariant Postcondition
Méthode <code>const</code>	(Invariant) Précondition	(Invariant) Postcondition
Méthode <code>non const</code>	Invariant Précondition	Invariant Postcondition
Destructeur	Invariant Précondition	Postcondition

Une variante consiste à supprimer les Invariants des méthodes `const` car elles ne modifient pas l'objet. En phase d'intégration, vous pouvez ne laisser que :

	Au début	A la fin
Constructeur	Précondition	Rien
Méthode <code>const</code>	Précondition	Rien

Méthode const	non	Précondi- tion	Rien
Destructeur		Précondi- tion	Rien

car le corps de la classe est considéré comme validé par le test unitaire. 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. Il faut recompiler tout le programme avec l'option /DNDEBUG.

Lors de la rédaction d'une assertion, il ne faut pas appeler de méthodes non `const` 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'accolade.

```
#ifndef NDEBUG
{
    //...
}
#endif
```

Le langage Eiffel utilise un environnement extérieur à la fonction pour décrire les assertions. En C++, 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 postconditions, de vouloir utiliser une autre méthode de la classe afin de confirmer le traitement par réversibilité. Par exemple, pour vérifier l'opérateur plus-égal, il semble intéressant de vérifier en postcondition, si en appelant l'opérateur moins-égal sur le résultat on obtient bien la valeur avant le traitement.

```
CFraction& operator +=(const CFraction& x)
{
    #ifndef NDEBUG
    CFraction old=*this;
    #endif
    //... Traitement
    #ifndef NDEBUG
    CFraction tmp=*this;
    POSTCONDITION((tmp-=x)==old, "Opérateur += ou -= faux");
    #endif
    return *this;
}
```

Mais que va faire l'opérateur moins-égal ? De même, il va appeler l'opérateur plus-égal pour vérifier son comportement. Nous sommes en présence d'une récursivité infinie. L'opérateur plus-égal appelle l'opérateur moins-égal, qui appelle l'opérateur plus-égal, qui appelle etc.

Il ne faut pas vérifier la réversibilité d'une méthode en postcondition. Ce sera fait en test unitaire. Par contre, certaines méthodes peuvent judicieusement appeler d'autres méthodes pour vérifier les postconditions. Ces méthodes doivent être constantes. Le constructeur de copie et l'opérateur d'affectation peuvent vérifier leurs comportements à l'aide de l'opérateur de comparaison d'égalité.

```
CFraction(const CFraction& x)
{
    //... Traitement
    POSTCONDITION(*this==x, "Constructeur de copie faux");
}
```

```
CFraction& operator =(const CFraction& x)
{
    //...
    POSTCONDITION(*this==x, "Constructeur de copie faux");
    return *this;
}
```

Les postconditions 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, le C++ possédera une extension gérant cela.

## 1.2 Test unitaire

Le test unitaire est la première étape de validation d'un développement. Il ne faut pas confondre le test unitaire et le déverminage. Le test unitaire cherche à prouver qu'une classe est correcte, le déverminage permet de localiser une erreur. Le test unitaire est un complément des préconditions 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 postcondition 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 assertions sont un bon moyen de trouver les erreurs dans les tests unitaires.

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 CMagnetophone. Celle-ci possède un état lui permettant d'autoriser certaines méthodes.

```
class CMagnetophone
{
public:
    enum TEtat { STOP, START, REWIND };
    CMagnetophone()
    : _etat(STOP) {}
    virtual ~CMagnetophone() {}

    virtual void start()
    {
        PRECONDITION(_cassette==true, "Pas de cassette");
        PRECONDITION(_etat==STOP, "Deja en marche");
        _etat=START;
    }
    virtual void stop()
    {
        _etat=STOP;
    }
    int compteur() const;
```

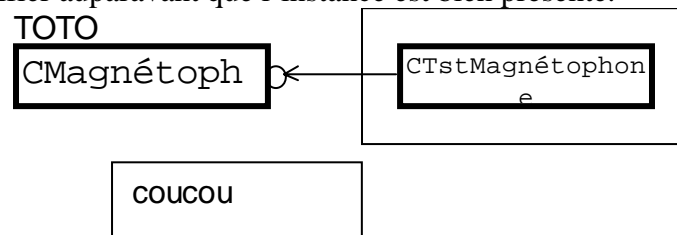
```

void                                         ajouteCassette()
{   PRECONDITION(!_cassette==false,"Deja   une   cassette");
    _cassette=true;
}
void                                         enleveCassette()
{   PRECONDITION(_cassette==true,"Il n'y a pas de cassette");
    _cassette=false;
}
bool                                         cassette() const
{   return _cassette;
}
protected:
bool                                         finBande() const;
TEtat                                       _etat;
bool                                         _cassette;
};

```

### 1.2.1 Méthodes public

Dans un premier temps, nous allons tester les méthodes publiques de la classe. Pour cela, nous allons écrire une classe `CTstMagnetophone` s'occupant de tester ces méthodes. Cette classe maintiendra l'état courant du test, et l'état de tous les objets testés. Pour pouvoir appeler une méthode, il faut dans un premier temps créer un objet. La classe `CTstMagnetophone` gardera l'information de la création de l'instance. Ainsi, le test d'une méthode pourra vérifier auparavant que l'instance est bien présente.



```

class                                         CTstMagnetophone
{
    CMagnetophone*                          _pMagneto;
    enum { IS_STOP, IS_START }              _etat;
public:
    CTstMagnetophone() : _pMagneto(NULL) {}
};

```

L'attribut `_etat` de `CTstMagnetophone` garde la trace de l'état courant de la classe `CMagnetophone`. Il n'est pas toujours possible de connaître l'état réel d'une classe. Il faut dans ce cas, maintenir un état hypothétique de la classe testée. Cela permet aux méthodes de vérifier les conditions de test avant de se lancer. Ensuite, nous allons ajouter des 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 du test 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 `CMagnetophone` seront également

vérifiées. La plupart des méthodes de tests renverront par défaut la valeur Ok car les vérifications auront été faites dans les postconditions des méthodes testées. Parfois, une postcondition 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.

```

enum          CTstRet          {          Ok,Bad,TstCondition          };
class
{
    enum          {          IS_STOP,IS_START          }          CTstMagnetophone
    {          CMagnetophone*          _pMagneto;
    public:
        CTstMagnetophone()
    :          _pMagneto(NULL),          _etat(IS_STOP)          {          }

    //          Test          de          la          construction
    virtual          CTstRet          ctrl()
    {          if          (_pMagneto!=NULL)          return          TstCondition;
      _pMagneto=new          CMagnetophone();
      if          (_pMagneto==NULL)          return          Bad;
      if          (_pMagneto->cassette())          return          Bad;
      _etat=IS_STOP;
      return          Ok;
    }

    //          Test          de          la          destruction
    virtual          CTstRet          dtrl()
    {          if          (_pMagneto==NULL)          return          TstCondition;
      delete
      _pMagneto=NULL;
      return          Ok;
    }

    //          Test          méthode          Start          dans          n'importe          quel          etat
    virtual          CTstRet          start1()
    {          if          (_pMagneto==NULL)          return          TstCondition;
      if          (!_pMagneto->cassette())          return          TstCondition;
      if          (_etat!=IS_STOP)          return          TstCondition;
      _pMagneto->start();
      _etat=IS_START;
      return          Ok;
    }

    //          Test          méthode          Stop          dans          n'importe          quel          etat
    virtual          CTstRet          stop1()
    {          if          (_pMagneto==NULL)          return          TstCondition;
      _pMagneto->stop();
      _etat=IS_STOP;
      return          Ok;
    }

    //          Test          méthode          Stop          dans          l'état          START
    virtual          CTstRet          stop2()
    {          if          (_pMagneto==NULL)          return          TstCondition;
  
```

```

    if      (_etat!=IS_START)      return      TstCondition;
    _pMagneto->stop(); //      Stop      si      etat      START
    _etat=IS_STOP;
    return                                     Ok;
}

//      Test      méthode      ajouteCassette
virtual      CTstRet      ajouteCassette1()
{
    if      (_pMagneto==NULL)      return      TstCondition;
    if      (_pMagneto->cassette())      return      TstCondition;
    _pMagneto->ajouteCassette();
    if      (_pMagneto->cassette()==false)      return      Bad;
    return                                     Ok;
}

//      Test      méthode      enleveCassette
virtual      CTstRet      enleveCassette1()
{
    if      (_pMagneto==NULL)      return      TstCondition;
    if      (!_pMagneto->cassette())      return      TstCondition;
    _pMagneto->enleveCassette();
    if      (_pMagneto->cassette())      return      Bad;
    return                                     Ok;
}
};

```

Toutes les méthodes de test vérifient le contexte courant du test. Elles effectuent le test et vérifient le comportement des méthodes de CMagnetophone. Il n'est pas toujours possible de vérifier le comportement d'une méthode à l'extérieur de la classe. Les postconditions 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 CMagnetophone, 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 CMagnetophone. Pour cela, il suffit d'appeler successivement différentes méthodes de tests. Les scénarios peuvent eux-mêmes être des tests de la classe CMagnetophone.

```

class      CTstMagnetophone
{
    virtual      CTstRet      scenario1()
    {
        if      (_pMagneto!=NULL)      return      TstCondition;
        CTstRet      rc;
        if      ((rc=ctrl())!=Ok)      return      rc;
        if      ((rc=ajouteCassette1())!=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;
    }
};

```

Une autre façon d'écrire ce scénario en utilisant les pointeurs de membres est la suivante :

```

class CTstMagnetophone
{
//...
virtual CTstRet scenario1()
{
    static CTstRet (CTstMagnetophone::* tab[])()=
    {&CTstMagnetophone::ctrl, &CTstMagnetophone::start1,
    &CTstMagnetophone::ajouteCassette1,
    &CTstMagnetophone::stop2,
    &CTstMagnetophone::start1,&CTstMagnetophone::dtrl
    };
    if (_pMagneto!=NULL) return TstCondition;
    CTstRet rc;
    for (int i=0;i<sizeof(tab)/sizeof(tab[0]);++i)
    {
        if ((rc=(this->*tab[i])())!=Ok) return rc;
    }
    return Ok;
}
};

```

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.

```

class CTstMagnetophone
{
//...
virtual CTstRet scenario2()
{
    static CTstRet (CTstMagnetophone::* tab[])()=
    {&CTstMagnetophone::ctrl,&CTstMagnetophone::dtrl,
    &CTstMagnetophone::ajouteCassette1,
    &CTstMagnetophone::start1,
    &CTstMagnetophone::stop1,&CTstMagnetophone::stop2
    };
    CTstRet rc;
    for (int i=0;i<50;++i)
    {
        do
        {
            rc=(this->*tab[rand()
(sizeof(tab)/sizeof(tab[0]))]());
        } while (rc==TstCondition);
        if (rc==Bad) return Bad;
    }
    return Ok;
}
};

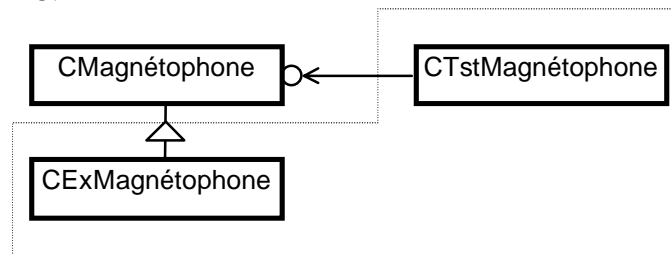
```

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.

### 1.2.2 Méthodes protected

Cela permet de tester les méthodes public d'une classe. Il faut également tester les méthodes protégées de celles-ci. Pour cela, si l'on ne désire pas modifier la classe testée pour effectuer le test (En déclarant toutes les méthodes publiques par exemple), il faut construire une

nouvelle classe qui héritera de la classe à tester. Celle-ci rendra toutes les méthodes protégées de la classe en `public`.



```

class CExMagnetophone : public CMagnetophone
{
    bool finBande() const
    {
        return CMagnetophone::finBande();
    }
};
  
```

Le test de la classe devra instancier une classe `CExMagnetophone` à la place de `CMagnetophone`, mais pourra alors, tester les méthodes `protected`. Il peut être préférable de séparer les tests des méthodes `protected` des tests des méthodes `public`. Dans ce cas, ajouter un mode dans la classe `CTstMagnetophe` et vérifier dans les conditions de test de chacun si ce mode correspond au type courant. Par exemple, si ce mode est à `TstProtected`, les méthodes `public` et `protected` peuvent être testées. Par contre, si ce mode est à `TstPublic`, seuls les tests des méthodes `publics` seront testés.

### 1.2.3 Méthodes `private`

Il n'est pas possible d'avoir un accès aux méthodes `private` d'une classe. Pour cela, il faudrait modifier la classe `CMagnetophone` pour déclarer la classe `CTstMagnetophone` en `friend`.

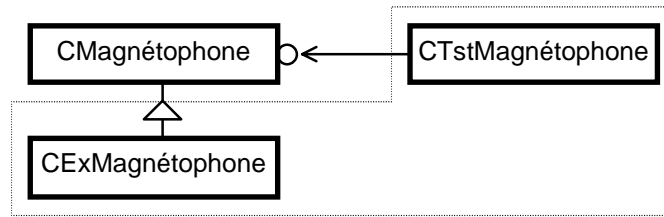
```

class CMagnetophone
{
    ...
    friend class CTstMagnetophone;
    ...
};
  
```

Avec cette modification minimale, il n'est plus nécessaire de construire la classe `CExMagnetophone`, car les méthodes `protected` et `private` deviennent accessibles au test. Il peut être raisonnable de ne pas tester les méthodes `private` car celles-ci ne sont utilisables que par les autres méthodes de la classe. Elles ne font pas partie de l'interface de la classe et seront de toute façon testées par effet de bord, en appelant les méthodes `public` et `protected`. Il est très probable que ces méthodes évoluent par la suite sans remettre en cause l'utilisation de la classe. Les tests de ces méthodes deviendraient obsolètes. Seul l'interface de la classe doit être testée. Si la classe évolue, les tests de l'interface doivent toujours être valides.

### 1.2.4 Méthodes `virtual`

Pour tester les méthodes virtuelles, il faut vérifier leurs comportements si une classe dérivée modifie celles-ci. Pour cela, il faut déclarer cette classe dérivée et redéfinir toutes les méthodes virtuelles. C'est l'occasion d'offrir l'accès aux méthodes `protected`. Les méthodes virtuelles peuvent individuellement ou collectivement changer leurs comportements selon l'état de cette nouvelle classe.



```

class      CExMagnetophone      :      public      CMagnetophone
{
  enum          {Normal,Modif}          _etat;

  CExMagnetophone()
  :          _etat(Normal)
  {
  }

  void          chgMode()
  {  _etat=((_etat==Normal) ? Modif : Normal);  }

  //      Acces      public      pour      finBande()
  bool          finBande()      const
  {      return      CMagnetophone::finBande();      }

  virtual      void      start()
  {      if      (_etat==Normal)      CMagnetophone::start();
    else
    {      //      Modification      du      comportement
    }
  }

  virtual      void      stop()
  {      if      (_etat==Normal)      CMagnetophone::stop();
    else
    {      //      Modification      du      comportement
    }
  }
};
  
```

La méthode `chgMode()` permet de basculer l'objet d'une version normale vers une version avec les méthodes virtuelles 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 `CMagnetophone` seule, si elle n'est pas virtuelle pure, puis de parcourir les mêmes tests avec la classe `CExMagnetophone`.

```

class          CTstMagnetophone
{
  CMagnetophone*      _pMagneto;
  enum      {      IS_NOT, IS_STOP, IS_START, IS_REWIND      }      _etat;
  enum      {IS_NORMAL, IS_VIRTUAL}      _vir;
  public:
  CTstMagnetophone()
  :  _pMagneto(NULL),  _etat(IS_NOT),  _vir(IS_NORMAL)  {  }

  //      Construction      de      CMagnetophone
  virtual      CTstRet      ctrl()
  
```

```

{
    if      (_pMagneto!=NULL)      return      TstCondition;
    _pMagneto=new                  CMagnetophone();
    _vir=IS_NORMAL;
    if      (_pMagneto==NULL)      return      Bad;
    _etat=IS_STOP;
    return      Ok;
}

//      Construction      de      CExMagnetophone      etat      Normal
virtual      CTstRet      ctr2()
{
    if      (_pMagneto!=NULL)      return      TstCondition;
    _pMagneto=new                  CExMagnetophone();
    if      (_pMagneto==NULL)      return      Bad;
    _vir=IS_NORMAL;
    _etat=IS_STOP;
    return      Ok;
}

//      Construction      de      CExMagnetophone      etat      Virtuel
virtual      CTstRet      ctr3()
{
    if      (_pMagneto!=NULL)      return      TstCondition;
    _pMagneto=new                  CExMagnetophone();
    if      (_pMagneto==NULL)      return      Bad;
    ((CExMagnetophone*)_pMagneto)->chgMode();
    _vir=IS_VIRTUAL;
    _etat=IS_STOP;
    return      Ok;
}
//...
};

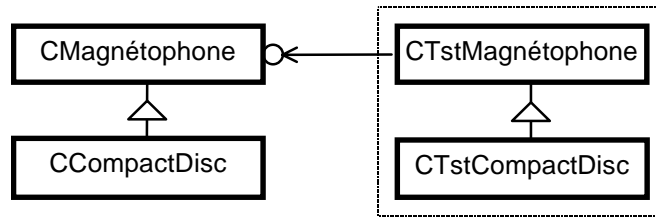
```

Les trois tests de construction vérifient le comportement de la classe dans différents cas d'utilisation. Le test `ctr2()` vérifie que la classe `CExMagnetophone` n'a pas ajouté d'erreur lors de l'utilisation normale. Le comportement de la classe doit être strictement le même que la classe originale. Le test `ctr3()` construit un objet `CExMagnetophone` 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 virtuelles pures seront redéfinies dans la classe `CExMagnetophone`.

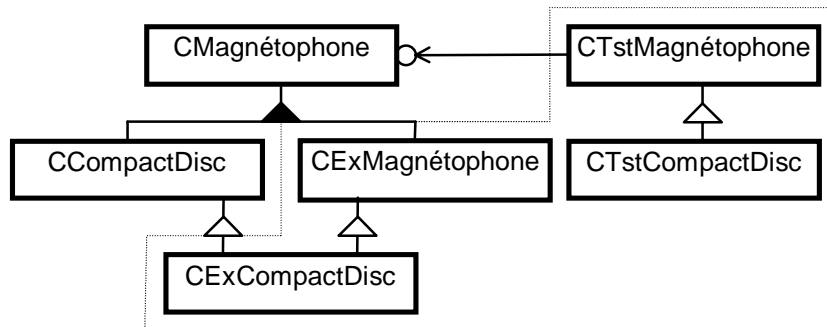
### 1.2.5 Héritage

Si une classe `CCompactDisc` hérite de `CMagnetophone`, 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 `CMagnetophone`. Nous allons construire une classe `CTstCompactDisc` qui hérite de la classe `CTstMagnetophone`. Les tests supplémentaires peuvent alors être ajoutés pour les nouvelles méthodes de `CCompactDisc`, et les tests hérités peuvent être inclus dans les nouveaux scénarios.

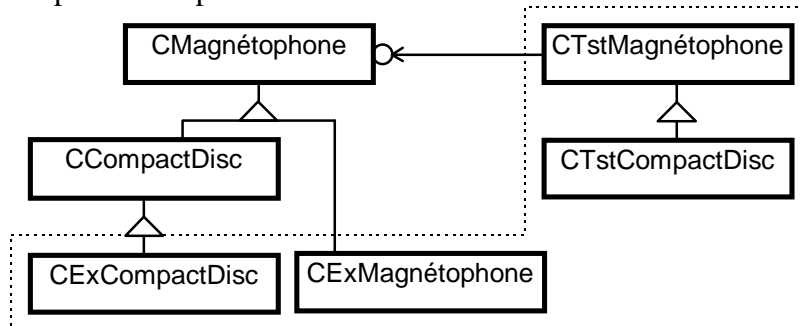


Certains tests de CTstMagnétophone peuvent être adaptés dans la classe CTstCompactDisc en redéfinissant les méthodes virtuelles.

On aimerait pouvoir recycler CExMagnétophone pour vérifier les méthodes protected de CCompactDisc et y ajouter les tests des nouvelles méthodes virtuelles. Pour cela, il faut que les classes CExMagnétophone et CCompactDisc héritent virtuellement de CMagnétophone. Cela donnerait le modèle suivant :



Malheureusement, la classe CCompactDisc n'hérite pas virtuellement de CMagnétophone. Il faut alors réécrire les méthodes de CExCompactDisc à l'aide, éventuellement, d'un copier-coller pour avoir ce modèle :



Les mêmes principes, suite aux différents types de méthodes, sont à utiliser pour tester correctement la classe CCompactDisc. 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 CMagnétophone évolue, les tests correspondants seront modifiés. La classe CTstCompactDisc bénéficiera de ces modifications. Il faudra par contre, vérifier la classe CExCompactDisc pour vérifier l'ensemble des méthodes virtuelles déclarées. Il est possible que de nouvelles méthodes soient ajoutées dans la classe CMagnétophone. Les classes CExMagnétophone et CExCompactDisc doivent alors être adaptées.

### 1.2.6 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 présent ici. Il faut vérifier plusieurs scénarios de comportements.

#### 1.2.6.1 Equivalence

Les tests d'*équivalence* permettent de vérifier que plusieurs méthodes ont un comportement équivalent. Cela permet de vérifier simultanément deux ou plusieurs méthodes. Pour les opérateurs de base, il existe plusieurs *patterns* standard à vérifier. 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)
...
```

### 1.2.6.2 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)
...
```

### 1.2.6.3 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))
...
```

### 1.2.6.4 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)
...
```

### 1.2.6.5 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)
...
```

### 1.2.6.6 Tests d'équivalence

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.

		0			
-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.

### 1.2.6.7 Tests aux limites

Les *tests aux limites* vérifient le comportement des méthodes avec des valeurs limites. Cela permet d'adapter les préconditions 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 pour les entiers non signés, si  $a$  est différent de zéro et que  $b$  est égale à `UINT_MAX`. Une précondition de l'opérateur d'addition d'un entier devrait vérifier si le résultat attendu est compatible avec la valeur maximum d'un entier. Les tests unitaires vérifieront les expressions précédente avec des valeurs courantes, puis les mêmes tests seront effectués avec des valeurs aux limites. Par exemple, un test unitaire pour les objets `unsigned int` peut être rédigé comme cela :

```
enum CTstRet {Ok, Bad, TstCondition};

class CTstunsigned
{
    unsigned* _a;
    unsigned* _b;
public:
    CTstunsigned() : _a(NULL), _b(NULL) {}
    CTstRet ctral()
    {
        if (_a!=NULL) return TstCondition;
        _a=new unsigned(10);
        return Ok;
    }
    CTstRet ctralLimitMax()
    {
        if (_a!=NULL) return TstCondition;
        _a=new unsigned(UINT_MAX);
        return Ok;
    }
    CTstRet dtra()
    {
        if (_a==NULL) return TstCondition;
        delete _a;
        _a=NULL;
        return Ok;
    }

    CTstRet ctrbl()
    {
        if (_b!=NULL) return TstCondition;
        _b=new unsigned(20);
        return Ok;
    }
    CTstRet ctrblLimitMax()
    {
        if (_b!=NULL) return TstCondition;
        _b=new unsigned(UINT_MAX);
        return Ok;
    }

    CTstRet dtrb()
    {
        if (_b==NULL) return TstCondition;
    }
};
```

```

        delete                _b;
        _b=NULL;
        return                Ok;
    }

    CTstRet                ordre()
    {
        if ((*_a==NULL) || (*_b==NULL)) return TstCondition;
        if ((*_b>=0) && !(*_a <= (*_a+*_b))) return Bad;
        if ((*_b>=0) && !(*_a >= (*_a-*_b))) return Bad;
        return                Ok;
    }

    CTstRet                scenario1();
    CTstRet                scenario2();
};

CTstRet                CTstunsigned::scenario1()
{
    static CTstRet (CTstunsigned::* tab[])()=
    {
        &CTstunsigned::ctral,&CTstunsigned::ctrbl,
        &CTstunsigned::ordre,
        &CTstunsigned::dtra,&CTstunsigned::dtrb
    };
    if ((*_a!=NULL) || (*_b!=NULL)) return TstCondition;
    CTstRet                rc;
    for (int i=0;i<sizeof(tab)/sizeof(tab[0]);++i)
    {
        if ((rc=(this->*tab[i]))!=Ok) return rc;
    }
    return                Ok;
}

CTstRet                CTstunsigned::scenario2()
{
    static CTstRet (CTstunsigned::* tab[])()=
    {
        &CTstunsigned::ctral,&CTstunsigned::ctrblLimitMax,
        &CTstunsigned::ordre,
        &CTstunsigned::dtra,&CTstunsigned::dtrb
    };
    if ((*_a!=NULL) || (*_b!=NULL)) return TstCondition;
    CTstRet                rc;
    for (int i=0;i<sizeof(tab)/sizeof(tab[0]);++i)
    {
        if ((rc=(this->*tab[i]))!=Ok) return rc;
    }
    return                Ok;
}

```

Le scénario “ un ” fonctionne correctement car les valeurs des objets sont normales. Le scénario “ deux ” ne fonctionne pas, car aux limites, les tests d’ordres ne sont plus valides. Les opérations sur les entiers non signés ne sont pas garanties !

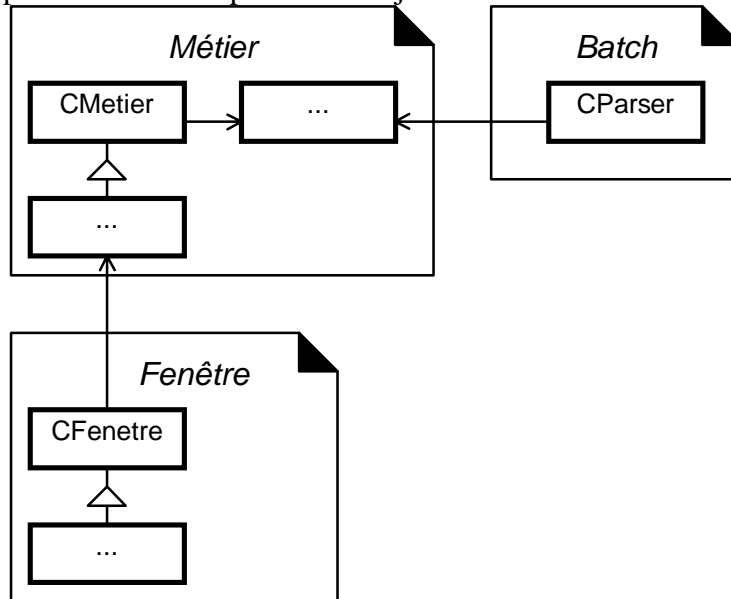
### 1.2.7 Résumé

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

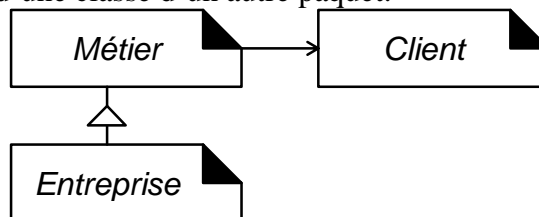
permet au développeur de vérifier sa classe et également de savoir si les erreurs apparues en intégration viennent de celle-ci ou des autres composants logiciels précédemment simulés.

### 1.3 Intégration

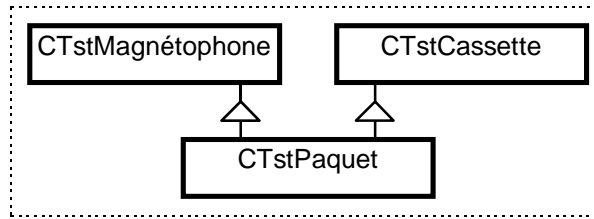
Après avoir passé avec succès les tests unitaires de chaque classe, il faut ensuite intégrer celles-ci dans des “paquets de classes”. Un paquet de classes est un ensemble de classes formant une couche logiciel. C’est une sorte de librairie. Cela peut correspondre à un namespace. Un paquet de classes 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.

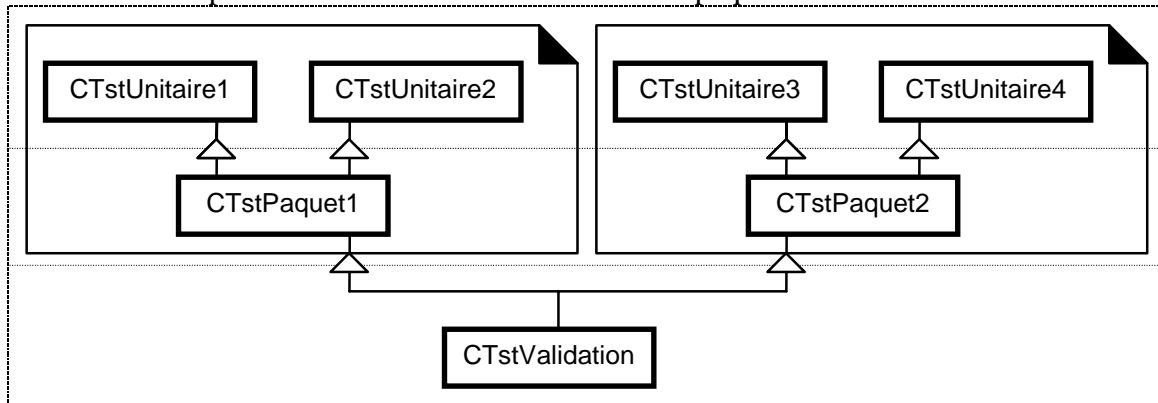


Lors des tests unitaires, les classes non encore intégrées sont simulées. Maintenant, il faut réunir les vraies classes et vérifier que les tests unitaires continuent à fonctionner. Il est possible qu’il faille modifier légèrement les tests pour pouvoir placer les classes précédemment simulées dans le 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 CTstMagnetophone pour tester la classe CMagnetophone, et la classe CTstCassette pour tester la classe CCassette. Nous allons écrire un test du paquet de classe héritant des deux classes 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. L'héritage des classes de tests est optionnel. Cela est un confort, mais n'est pas obligatoire. 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. Les tests aléatoires sont possibles, soit en utilisant les méthodes de tests héritées, soit en utilisant les nouveaux tests rédigés au sein de la classe CTstPaquet.

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

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.