

Test orienté objet

Test des classes

Test orienté objet des classes

- ❑ Introduction
- ❑ Considérations sur l'héritage
- ❑ Test des séquences de méthodes
- ❑ Test base sur le **modèle d'état**
- ❑ Driver, Oracles et Stubs

Motivations

- ❑ Les concepts orientés objet *facilitent l'analyse et la conception des grands systèmes.*
- ❑ Mais les études empiriques semblent indiquer qu'ils *demandent plus de tests.*
- ❑ Le logiciel OO a des *éléments spécifiques* que nous devons considérer durant le test.
- ❑ Les tests unitaires et d'intégration sont spécialement affectés parce qu'ils dépendent de la structure du logiciel sous test.

Test de classe vs test de procédure

- ❑ Programmation procédurale
 - Composant de base : fonction (procédure) ;
 - Méthode de test : basée sur la relation entre entrées et sorties.
- ❑ Programmation orientée objet
 - Composant de base : classe = attributs (**état**) + ensemble d'opérations (**comportement**);
 - les objets (instances de classes) sont testés ;
 - la justesse ne peut pas simplement être définie comme étant une relation entrée/sortie, mais **doit aussi inclure l'état de l'objet**.
- ❑ L'état n'est pas directement accessible (**encapsulation**), mais peut seulement être accessible en utilisant les opérations publiques des classes.

Exemple

```
class Watcher {  
    private:  
        ...  
        int status;  
        ...  
    public:  
        void checkPressure() {  
            ...  
            if (status == 1)  
                ...  
            else if (status ...)  
                ...  
        }  
        ...  
}
```

- ❑ Tester la méthode *checkPressure()* de manière isolée, n'a aucun sens :
 - génération des données de test ;
 - mesure de couverture.
- ❑ Créer des oracles est plus difficile, par exemple :
 - la valeur retournée par la méthode *check_pressure* dépend des états des instances de la classe *Watcher* (variable **status**) ;
 - les défaillances à cause des valeurs incorrectes de la variable **status** – peuvent être révélées seulement par des tests qui ont un contrôle et une visibilité sur cette variable **status** .

Nouveaux niveaux d'abstraction

- ❑ Les fonctions (sous-programmes) sont les unités de base dans un logiciel procédural.
- ❑ Les classes introduisent un nouveau niveau d'abstraction :
 - *test unitaire de base* : le test d'une opération simple (méthode) d'une classe (test intra-méthode) ;
 - *test unitaire* : le test d'une classe (test intra-classe).
- ❑ *Test d'intégration* : le test d'intégration des classes (test inter-classes), reliées via des dépendances: associations, agrégations, spécialisations.

Nouveaux modèles de fautes

- ❑ Mauvaise instance de la méthode héritée en présence d'héritage multiple.
- ❑ Mauvaise redéfinition d'un attribut / donnée membre
- ❑ Mauvaise instance de l'opération appelée à cause des liaisons dynamiques ("dynamic binding") et aux erreurs de type.
- ❑ Manque d'information statistique sur la fréquence des erreurs et sur les coûts de leur détection et leur suppression.
- ❑ Les nouveaux modèles de fautes sont vitaux pour définir les méthodes de tests et techniques ciblant les fautes spécifiques OO.

Test structurel dans le contexte OO

- ❑ Dans les systèmes OO, la plupart des méthodes contiennent quelques LOCs – la complexité réside au niveau des interactions entre méthodes.
- ❑ Le comportement de la méthode est insignifiant à moins d'être analysé en relation avec d'autres opérations et leur effets associés sur un état partagé (valeurs d'attributs).
- ❑ On considère que toute unité significative à tester ne peut pas être plus petite que les instanciations d'une classe.

Test orienté objet des classes

- ❑ Introduction
- ❑ Considérations sur l'héritage
- ❑ Test des séquences de méthodes
- ❑ Test base sur le **modèle d'état**
- ❑ Driver, Oracles et Stubs

Test et héritage

- ❑ **Modification d'une superclasse**

- Nous avons à tester de nouveau ses sous-classes

- ❑ **Ajout d'une sous-classe** (ou modification d'une sous-classe existante).

- Nous pouvons avoir à tester de nouveau les méthodes héritées de chacun de ses ancêtres superclasses car les sous-classes produisent un nouveau contexte pour les méthodes héritées.

- ❑ Pas de problème si la nouvelle sous-classe est une extension pure de la superclasse, i.e., elle ajoute de nouvelles variables de classe et des méthodes, et il n'y a aucune interaction dans les deux directions entre les nouvelles variables de classe et méthodes, et les variables de classe et méthodes héritées.

Héritage : Exemple I (1)

```
class refrigerator {  
public:  
    void set_desired_temperature(int temp);  
    int get_temperature();  
    void calibrate();  
private:  
    int temperature;  
};
```

- ❑ *set_desired_temperature* permet à la température d'être entre 5C et 20C centigrade.
- ❑ *calibrate* met le réfrigérateur actuel dans des cycles de refroidissement et utilise le lecteur du capteur pour calibrer l'unité de refroidissement.

Héritage : Exemple I (2)

- ❑ Un nouveau modèle de réfrigération plus sophistiqué est créé et peut refroidir jusqu'à – 5 C centigrade.
- ❑ La classe *better_refrigerator* a une nouvelle version de *set_desired_temperature*.
- ❑ La méthode *calibrate* est inchangée.
- ❑ Est-ce que *better_refrigerator::calibrate* doit être testée de nouveau malgré qu'il s'agisse du même code exactement?

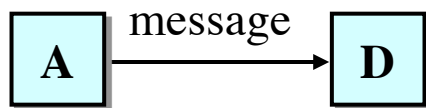
Héritage : Exemple I (3)

- ❑ Oui, elle doit être testée de nouveau.
- ❑ Supposons que *calibrate* fonctionne en **divisant les lectures du senseur par température.**
- ❑ Que faire si la *temperature* est 0 ?
- ❑ C'est possible dans *better_refrigerator*.
- ❑ Cela causera une division par 0, anomalie qui ne devrait pas arriver dans *refrigerator*.

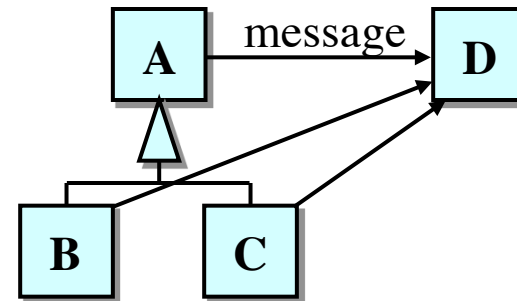
Surcharge des méthodes

- ❑ Les langages OO permettent à une sous-classe de remplacer une méthode héritée par une méthode du même nom.
- ❑ La méthode surchargée de la sous-classe doit être testée.
- ❑ Mais *différents ensembles de tests sont nécessaires !*
 - *Raison 1* : Si les cas de test sont dérivés de la structure du programme (les données et le flux de contrôle), la structure de la méthode surchargée peut être différente.
 - *Raison 2* : **Le comportement de la méthode surchargée est aussi probablement différent.**

Intégration et polymorphisme

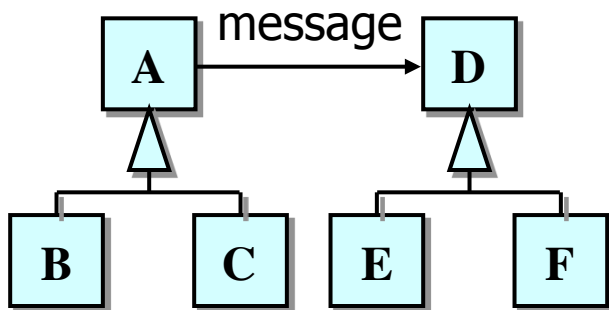


1 ensemble de test

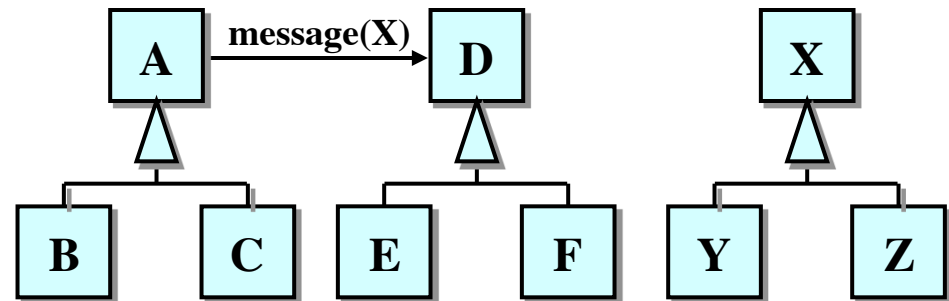


3 ensembles de test

Aucune classe n'est ABSTRAITE



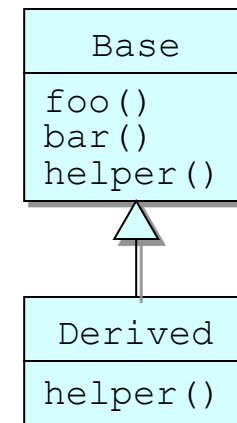
9 ensembles de test



27 ensembles de test

Exemple II : Code

```
class Base      {
    public:
        void foo()      { ... helper(); ...}
        void bar()      { ... helper(); ...}
    private:
        virtual void helper()    {...}
};
class Derived : public Base {
    private:
        virtual void helper()    {...}
};
void test_driver()    {
    Base base;
    Derived derived;
    base.foo();        // Test case 1
    derived.bar();     // Test case 2
}
```



Exemple II : Discussion

- ❑ **Cas de test 1** : Appelle *Base::foo()* qui à son tour appelle *Base::helper()*
- ❑ **Cas de test 2** : La méthode héritée *Base::bar()* est appelée dans l'objet *Derived*, qui à son tour appelle *helper()* sur l'objet *Derived*, appelant *Derived::helper()*.
- ❑ Supposons que toutes les méthodes contiennent seulement un flux de contrôle linéaire, est-ce que les cas de test exécutent complètement le code de *Base* et de *Derived* ?
- ❑ Les mesures de la couverture traditionnelle (e.g., statement, flux de contrôle) devraient répondre *oui*.

Exemple II : Manque-t-il quelque chose ?

- ❑ Nous n'avons pas complètement testé des interactions entre *Base* et *Derived* :
 - *Base::bar()* et *Base::helper()*
 - *Base::foo()* et *Derived::helper()*
- ❑ **Ce n'est pas parce que** *Base::foo()* **fonctionne avec** *Base::helper()* **qu'il fonctionnera automatiquement avec** *Derived::helper()*.
- ❑ Nous avons besoin d'exercer *foo()* et *bar()* pour la classe *Base* et la classe *Derived*.

Exemple II : Nouveau Driver de test

```
void better_test_driver()    {  
  
    Base base;  
    Derived derived;  
  
    base.foo();  
    base.bar();  
  
    derived.foo();  
    derived.bar();  
}
```

Vous pouvez voir comment l'héritage doit être utilisé avec précaution – il mène à plus de tests !

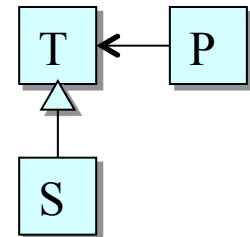
Test incrémental hiérarchique – *Harrold McGregor*

- ❑ Vise à tester la hiérarchie de l'héritage :
- ❑ **Étape 1** : Tester complètement toutes les méthodes dans le contexte d'une classe particulière (classe de base ou classe dérivée des classes de base abstraites).
- ❑ **Étape 2** : Couverture l'interaction : Toutes les méthodes qui sont héritées par une classe dérivée et qui interagissent avec des méthodes redéfinies (ou nouvelles méthodes via les attributs hérités) doivent être testées de nouveau dans le contexte de la classe dérivée.
- ❑ Re-exécuter tous les cas de test de la classe de base (e.g., basé sur 100% couverture de branches) dans le contexte de la classe dérivée par laquelle elle est héritée.
- ❑ Ceci réduit le coût de test des méthodes héritées dans plusieurs contextes et aide de vérifier la conformité des hiérarchies de l'héritage au *principe de substitution de Liskov* : **Dans les hiérarchies de classe, il devrait être possible de traiter un objet spécialisé comme s'il était un objet de classe de base.**

(... *Principe de substitution de Liskov*

- ❑ Ce principe définit les notions de généralisation / spécialisation d'une manière formelle.
- ❑ La classe S est correctement définie comme une spécialisation de la classe T si ce qui suit est vrai :

pour chaque objet s de la classe S, il y a un objet t de la classe T tel que le comportement de n'importe quel programme P défini en termes de T est inchangé si s est substitué par t.



- On dit que S est un *sous-type* de T.
- Toutes les instances d'une sous-classe peuvent remplacer les instances d'une superclasse sans effet sur les classes clientes.
- Toute future extension (nouvelles sous-classes) n'affecteront pas les clients existants.

Manque de substituabilité

```
class Rectangle : public Shape {
private: int w, h;
public:

    virtual void set_width(int wi) {
        w=wi;
    }

    virtual void set_height(int he) {
        h=he;
    }
}
```

```
class Square : public Rectangle {
public:

    void set_width(int w) {
        Rectangle::set_height(w);
        Rectangle::set_width(w);
    }

    void set_height(int h) {
        set_width(h);
    }
}
```

```
void foo(Rectangle *r) { // This is the client
    r->set_width(5);
    r->set_height(4);
    assert((r->get_width()*r->get_height()) == 20); // Oracle
}
```

- Si *r* est instanciée au moment d'exécution avec une instance du Square, le comportement observé par le client est différent (largeur*hauteur == 16).
- Peut mener à des problèmes.
- Le Square devrait être défini comme sous-classe de Shape, pas de Rectangle.

Règles

- ❑ **Règle de signature** : Les sous-types doivent avoir toutes les méthodes du super-type, et les signatures des méthodes des sous-types doivent être *compatibles* avec les signatures des méthodes des super-types correspondants.
- ❑ En Java, il est forcé que le sous-type doit avoir toutes les méthodes du super-type, avec des signatures identiques à l'exception qu'une méthode de sous-type peut avoir moins d'exceptions (compatibilité est ici plus stricte que nécessaire).
- ❑ **Règle des Méthodes** : Les appels des méthodes de sous-types doivent "se comporter comme" des appels des méthodes des super-types correspondants.
- ❑ **Règle de Propriétés** : Le sous-type doit préserver l'invariant du super-type.

Contrats - Définitions

- ❑ Buts : spécifient les opérations afin que l'appelant/client et l'appelé/serveur partagent les mêmes suppositions.
- ❑ Un contrat spécifie les contraintes que l'appelant doit rencontrer avant d'utiliser la classe ainsi que les contraintes qui sont assurées par l'appelé quand utilisé.
- ❑ Il y a trois types de **contraintes concernés dans les contrats** :
 - Invariant (classe), Pré-condition, Post-condition (opérations).
- ❑ Les contrats devraient être spécifiés, pour des opérations connues, pendant les phases de l'analyse et de la conception.
- ❑ En UML, un langage a été défini dans ce but : Le Langage de Contrainte d'Objet (*Object Constraint Language* (OCL)).

L'invariant de la classe

- ❑ La condition qui doit toujours être satisfaite par toutes les instances de la classe.
- ❑ Décrit en utilisant une expression qui est évaluée à vraie si l'invariant est satisfait.
- ❑ Les invariants doivent être vrais en tout temps, sauf durant l'exécution d'une opération où l'invariant peut être temporairement violé.
- ❑ Un invariant violé suggère un état illégal du système.

SavingsAccount

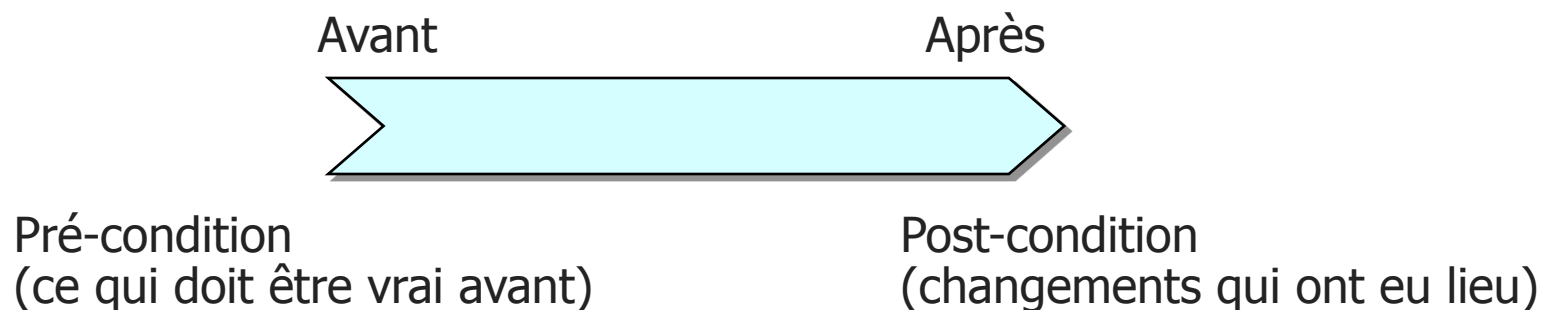
balance
{balance>0 and
balance<25000}

Contexte SavingsAccount **inv:**

self.balance > 0 and self.balance < 25000

Pré- et post-conditions d'opérations

- ❑ Pré-condition : Qu'est-ce qui doit être vrai avant l'exécution d'une opération ?
- ❑ Post-condition : Supposant que la pré-condition est vraie, qu'est-ce qui doit être vrai au sujet de l'état du système et quels sont les changements qui apparaissent après l'exécution de l'opération ?
- ❑ Ces conditions doivent être écrites comme expressions logiques (Booléennes).
- ❑ Ainsi, les opérations du système sont traitées comme des boîtes noires. Rien n'est dit concernant les états intermédiaires des opérations et les détails algorithmiques.
- ❑ Si les pré- et post-conditions sont satisfaisantes, alors l'invariant de classe doit être préservée.



Spécifications des Contrats

- ❑ Spécifier les requis des opérations du système en termes d'entrées et d'état du système (pré-condition).
- ❑ Spécifier les effets des opérations du système en termes de changements d'état et sorties (post-condition).
- ❑ L'état du système est représenté par l'état des objets et les relations entre eux.
- ❑ Une opération du système peut
 - créer une nouvelle instance d'une classe ou en éliminer une existante ;
 - changer une valeur d'attribut d'un objet existant ;
 - ajouter ou enlever des liens entre les objets, et
 - envoyer un événement/message à un objet.

Règle de la méthode

La règle peut être exprimée en pré- et post-conditions.

□ La pré-condition est *affaiblie* :

- Affaiblir la pré-condition implique que la méthode du sous-type requiert moins du appelant.
- Si les méthodes $T::m()$ et $S::m()$ (surcharge) ont des pré-conditions $PrC1$ et $PrC2$, respectivement, $PrC1 \Rightarrow PrC2$.

□ La post-condition est *renforcée* :

- Renforcer signifie que la **méthode sous-type retourne** plus que la méthode super-type.
- Si les méthodes $T::m()$ et $S::m()$ (surcharge) ont des post-conditions $PoC1$ et $PoC2$, respectivement, $(PrC1 \wedge PoC2) \Rightarrow PoC1$.

□ Le code appelant dépend de la post-condition de la méthode du super-type, mais seulement si la pré-condition est satisfaite.

IntSet

```
public class IntSet {
    private Vector els; /// the elements

    public IntSet() {...}
        // Post: Initializes this to be empty

    public void insert (int x) {...}
        // Post: Adds x to the elements of this

    public void remove (int x) {...}
        // Post: Remove x from the elements of this

    public boolean isIn (int x) {...}
        //Post: If x is in this returns true else returns false

    public int size () {...}
        //Post: Returns the cardinality of this

    public boolean subset (IntSet s) {...}
        //Post: Returns true if this is a subset of s else returns false
}
```

Post-conditions : MaxIntSet

```
public class MaxIntSet extends IntSet {

    private int biggest; // biggest element if set not empty

    public MaxIntSet () {...} // call super()

    public Max () throws EmptyException {...} // new method

    public void insert (int x) {...}
        // overrides InSet::insert()
        // Additional Post: update biggest with x if x > biggest

    public void remove (int x) {...}
        // overrides InSet::remove()
        // Additional Post: update biggest with next biggest element in this if
        x = biggest
}
```

Pré-conditions : LinkedList & Set

```
public class LinkedList {
    ...
    /** Adds an element to the end of the list
     * PRE:  element != null
     * POST: this.getLength() == old.getLength() + 1
     *       && this.contains(element) == true
     */
    public void addElement(Object element) { ... }
    ...
}

public class Set extends LinkedList {
    ...
    /** Adds element, provided element is not already in the set
     * PRE:  element != null && this.contains(element) == false
     * POST: this.getLength() == old.getLength() + 1
     *       && this.contains(element) == true
     */
    public void addElement(Object element) { ... }
    ...
}
```

Règle des Propriétés

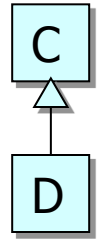
- ❑ Toutes les méthodes du sous-type doivent préserver l'invariant du super-type.
- ❑ L'invariant du sous-type doit impliquer l'invariant du super-type.
- ❑ Supposons que le *FatSet* est un ensemble d'entiers dont la taille est toujours au moins 1. Les méthodes constructeur et Remove garantissent ceci.
- ❑ *ThinSet* est aussi un ensemble d'entiers mais peut être vide et alors ne peut pas être un sous-type légal de *FatSet*.

IntSet, MaxIntSet ...)

- ❑ L'invariant du *IntSet*, pour n'importe quel instance de *i*:
i.els \neq *null* et
Tous les éléments de *i.els* sont de entiers et il n'y a pas de duplicata dans *i.els*.
- ❑ L'invariant du *MaxIntSet*, pour n'importe quelle instance *i* :
l'invariant de *IntSet* et *i.size* > 0 et pour tous les entiers *x* in *els*, *x* \leq *i.biggest*.
- ❑ L'invariant du *MaxIntSet* inclut l'invariant de *IntSet* et alors l'implique.
- ❑ Nous nous conformons avec la règle de propriété.

Test incrémental hiérarchique (II)

- ❑ Supposons que C est la classe de base et que D est une sous-classe de C.
- ❑ **Surcharger dans D une méthode de C mais aucun changement dans la spécification.**
 - Reutiliser tous les cas de test hérités et basés sur la spécification.
 - Mais nous aurons besoin des cas de test basés sur l'implémentation pour satisfaire le critère de couverture.
- ❑ **Changer dans D la spécification d'une opération de C :**
 - Des cas de test additionnels pour exercer les conditions des nouvelles entrées (pré-condition affaiblie) et vérifier les nouveaux résultats attendus (post-condition renforcée).
 - Les cas de test pour C s'appliquent encore.
 - **Raffiner l'oracle (post-condition renforcée).**
- ❑ Nouvelles opérations introduisent une nouvelle fonctionnalité et code à tester.
- ❑ Nouveaux attributs sont ajoutés en connexion avec les opérations nouvelles ou surchargées – ceci peut mener à retester les méthodes héritées.
- ❑ Nouvel invariant de classe : tous les cas de test ont besoin d'être reexécutés pour vérifier que le nouvel invariant est satisfait.



Couverture de contexte d'héritage

- ❑ Étendre l'interprétation des mesures de la couverture structurelle traditionnelle.
- ❑ Considérer le niveau de couverture dans le contexte de chaque classe comme mesures séparées.
- ❑ 100% de couverture du contexte d'héritage requiert que le code doit être *complètement* exécuté (pour tout critère sélectionné, e.g., tous les branches) dans *chaque* contexte approprié.
- ❑ Les contextes appropriés peuvent être déterminés en utilisant les principes HIT (Hierarchical Inheritance Testing) déjà vus.

Test orienté objet des classes

- ❑ Introduction
- ❑ Considérations sur l'héritage
- ❑ Test des séquences de méthodes
- ❑ Test base sur le **modèle d'état**
- ❑ Driver, Oracles et Stubs

Motivations & problèmes

- ❑ Il est avancé que tester une classe vise à trouver une *séquence d'opérations* pour laquelle une classe sera dans un état qui est *contradictoire* à son invariant ou à la sortie attendue.
- ❑ Tester les classes pour toutes les séquences possibles n'est pas souvent possible.
 - 10 méthodes → 10! Séquences
 - Si chaque test requiert 1 sec, nous avons besoin de 1000 heures ...
- ❑ Les ressources nécessaires pour tester une classe augmentent exponentiellement avec l'augmentation du nombre de ses méthodes.
- ❑ Il est nécessaire de trouver un moyen pour réduire le nombre de séquences en fournissant un degré de confiance suffisant.

Exemple

- ❑ Une boîte de monnaie d'une machine vendeuse implémentée en C++.
- ❑ La boîte à monnaie a une fonctionnalité simple et le code pour contrôler le dispositif physique est omis.
- ❑ Elle accepte seulement les vingt-cinq cents et permet la vente quand deux vingt-cinq cents sont reçus.
- ❑ Elle garde trace de tous les vingt-cinq cents déjà reçus (*totalQrts*), les vingt-cinq cents reçus pour la transaction en cours (*curQrts*) et de la permission de vente (*allowVend*).
- ❑ Fonctions : ajouter un vingt-cinq cents, retourner les vingt-cinq cents courants, remettre la boîte de monnaie à son état initial, et vendre.

Code CCoinBox

Code manquant
(allowVend=0;)

```
class CCoinBox
{
    unsigned totalQtrs;
    unsigned curQtrs;
    unsigned allowVend;

public:
    Ccoinbox() {Reset();}
    void AddQtrs();
    void ReturnQtrs() {curQtrs=0;}
    unsigned isAllowedVend()
        {return allowVend;}
    void Reset() {
        totalQtrs = 0;
        allowVend = 0;
        curQtrs = 0;
    }
    void Vend();
};
```

```
void CCoinBox ::AddQtr()
{
    curQtrs = curQtrs + 1;
    if (curQtrs > 1)
        allowVend = 1;
}

void CCoinBox::Vend()
{
    if(isAllowedVend())
    {
        totalQtrs = totalQtrs + 2;
        curQtrs = curQtrs - 2;
        if (curQtrs < 2) allowVend=0;
    }
}
```

Diagramme de transitions d'état UML

- ❑ Transitions d'état causées par des événements, permis par les conditions de garde.



- ❑ Ont la forme générale suivante (tous les éléments sont optionnels)

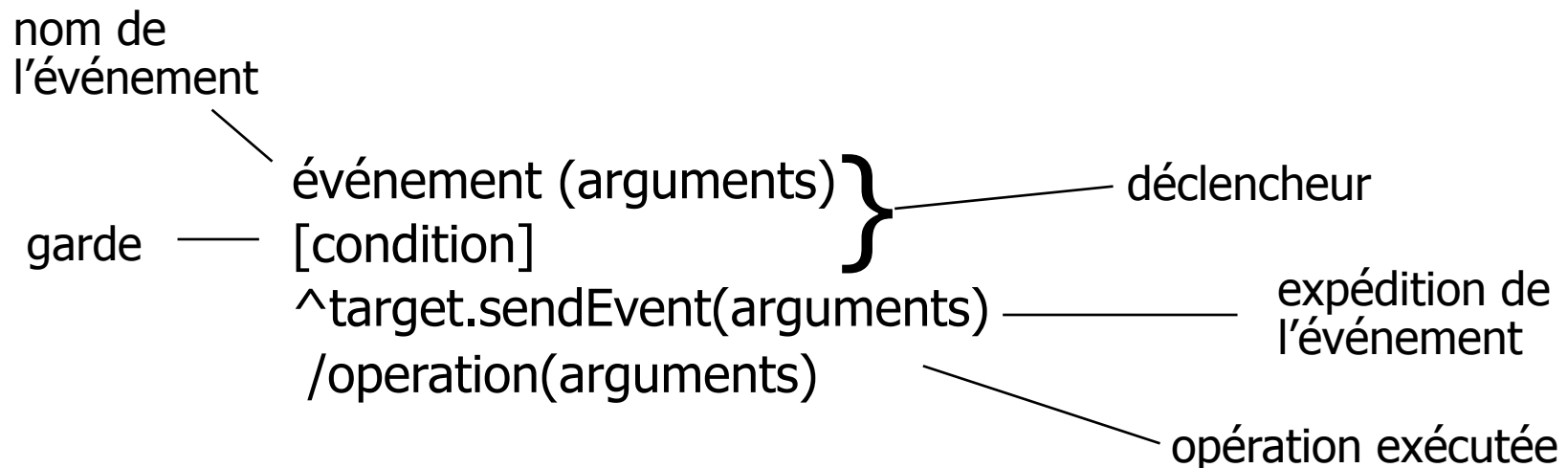


Diagramme d'état de CCoinBox

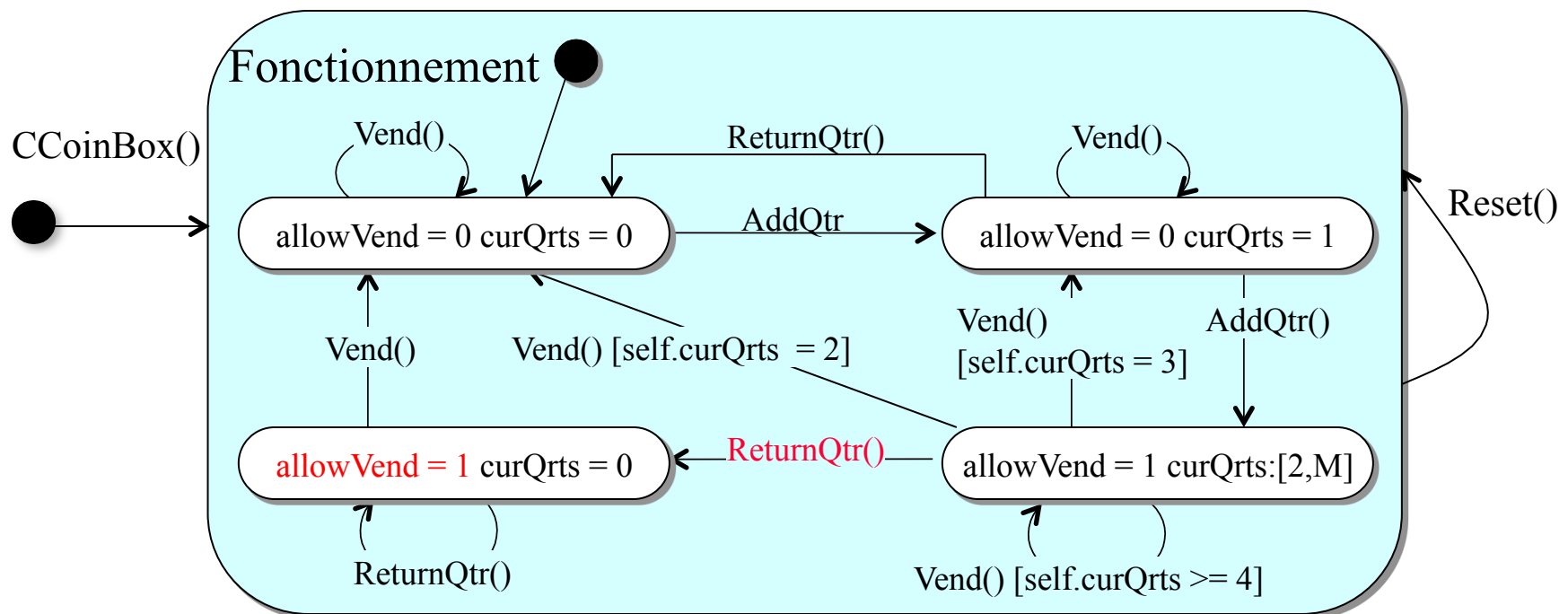


Diagramme d'état correspondant **au code**, pas complètement spécifié

- ❑ Un état corrompu manquant : allowVend = 1 curQrts = 1
- ❑ Une transition manquante dans l'état allowVend = 1 curQrts=0 : AddQrts()
- ❑ allowVend=0 curQrts=0: ReturnQtr() allowVend=1 curQrts:[2,M]: AddQrt()

Détection de fautes

- ❑ Le diagramme d'état indique que le scénario `AddQtrs()` ; `AddQtrs()` ; `ReturnQtrs()` ; `Vend()` est possible ! (BOISSON GRATUITE !)
- ❑ C'est une faute **dépendante d'état**, dans le sens qu'exécuter certaines opérations s'avère impossible dans certains états où elles sont légales ou, alternativement, que certaines séquences d'opération qui sont théoriquement impossibles sont exécutées.
- ❑ Le test structurel de méthode devrait conclure que e.g., chaque branche a été exécutée; l'omission peut demeurer imprévue même avec un test fonctionnel
- ❑ **La défaillance apparaîtra seulement pour une certaine séquence d'opérations.**
- ❑ Le test de méthode seul n'est pas suffisant!

Dérivation des séquences

- ❑ Tranches de données
- ❑ Pré-conditions et post-conditions des opérations
- ❑ Spécification formelle, e.g., spécification algébrique
- ❑ Diagrammes d'état

Test orienté objet des classes

- ❑ Introduction
- ❑ Considérations sur l'héritage
- ❑ Test des séquences de méthodes
 - Tranches de données
- ❑ Driver, Oracles et Stubs

Tranches de données : principes de base

- ❑ But : réduire le nombre de séquences des méthodes à tester.
- ❑ L'état (concret) d'un objet à un moment donné est équivalent à l'état agrégé de chacun de ses attributs.
- ❑ La **justesse d'une classe** dépend de :
 - si les attributs membres représentent correctement l'état voulu d'un objet,
 - si les fonctions membres manipulent correctement la représentation de l'objet.

Tranche – Slice (Bashir et Goel)

- ❑ Une classe peut être considérée comme une *composition* de tranches.
- ❑ Une tranche est un quantum d'une classe avec un seul attribut et le sous-ensemble de méthodes pouvant manipuler les valeurs associées à cet attribut.
- ❑ La stratégie de test de classe de Bashir et Goel est de tester une *tranche* à la fois.
- ❑ Pour chaque tranche, tester les séquences possibles des méthodes appartenant à cette tranche – **équivalent à tester un attribut spécifique**, i.e., la justesse partielle de la classe.
- ❑ Répéter pour chacune des tranches pour démontrer la justesse de la classe.

Formalisme de la tranche

- Une classe K encapsule un ensemble d'attributs et fournit un ensemble d'opérations

- ✓ $K = \langle D(K), M(K) \rangle$

- ✓ $D(K) = \{d_i \mid d_i @ K\}$

- ✓ $M(K) = \{m_i \mid m_i @K\}$

- (où @ indique la relation entre une classe et ses éléments i.e., est dans, appartient à ...).*

- Une tranche de données

- ✓ $\text{Slice}_{d_i}(K) = \langle d_i, M_{d_i}(K) \rangle$

- ◆ $d_i \in D(K)$

- ◆ $M_{d_i}(K) = \{m_i \mid m_i @@ d_i\}$

- (où @@ indique la relation 'utilise').*

Générer le MaDUM

- ❑ *MaDUM* : Matrice minimale d'utilisation des données membres (*Minimal Data Member Usage Matrix*)
- ❑ *matrice* $n*m$ où n est le nombre d'attributs et m représente le nombre de méthodes membres : signale l'utilisation des données membres par les méthodes.
- ❑ *Usages différents* : lire, signaler, transformer.
- ❑ Signale l'utilisation *indirecte* des données membres, à partir des fonctions membres intermédiaires.
- ❑ Utilisation de MaDUM pour concevoir une stratégie de test, un ordre de tests !

Catégoriser les fonctions membres

- ❑ Catégories : Constructeurs (C), Reporteurs (R), Transformateurs (T), Autres (O)
- ❑ $M_{di}(K) = \{R_{di}, C, T_{di}, O_{di}\},$
 - $R_{di} = \{r_{di} \mid r_{di} \text{ est une méthode reporteur (getter) pour la donnée membre } d_i\}$
 - $C = \{c_i \mid c_i \text{ est un constructeur de la classe } K\}$
 - $T_{di} = \{m_{di} \mid m_{di} \notin R_{di} \text{ et } m_{di} \notin C \text{ et } m_{di} \rightarrow d_i\} \text{ (setter)}$
 - $O_{di} = \{o_{di} \mid o_{di} @@ d_i \text{ et } o_{di} \notin R_{di} \text{ et } o_{di} \notin C \text{ et } o_{di} \notin T_{di}\}$
- ❑ Autre(s): `printError()`, destructeurs, ...

Exemple C++ (I)

```
class SampleStatistic {
    friend TestSS; //test driver!

protected :
    int n;
    double x;
    double x2;
    double minValue, maxValue;

public :
    sampleStatistic();
    virtual ~SampleStatistic();
    virtual void reset();
```

```
    virtual void operator += (double);
    int samples();
    double mean();
    double stdDev();
    double var();
    double min();
    double max();
    double confidence(int
        p_percentage);
    double confidence (double p_value);
    void error(const char * msg);
};
```

❑ Partie de la librairie C++ de GNU

Exemple C++ (II)

```
SampleStatistic::mean() {  
    if (n>0)  
        return (x/n);  
    else  
        return (0.0);  
}
```

```
SampleStatistic::operator+=(double value) {  
    n += 1;  
    x += value;  
    x2 += (value * value);  
    if(minValue > value)  
        minValue = value;  
    if(maxValue < value)  
        maxValue = value;  
}
```

```
double SampleStatistics::stdDev() {  
    if (n<=0 || this->var()<=0)  
        return(0);  
    else  
        return (double)sqrt(var());  
}
```

```
double SampleStatistics::var() {  
    if (n>1)  
        return ((x2 - ((x*x)/2)) / (n-1));  
}
```

MaDUM pour SampleStatistic

	Sample Statistic	reset	+ =	samples	mean	stdDev	var	min	max	confi- dence (int)	confi- dence (dbl)	error
n	c	t	t	r	o	o	o			o	o	
x	c	t	t		o	o	o			o	o	
x2	c	t	t			o	o			o	o	
min- Value	c	t	t					r				
max- Value	c	t	t						r			

Compte Matrice pour utilisation indirecte par méthodes appelées.

- stdDev() n'accède pas directement x et x2, mais appelle var() qui le fait.
- Tout ce que SampleStatistic() fait est d'appeler reset().

Procédure de test

- ❑ La classification des méthodes est utilisée pour décider des étapes de test à prendre :
 - 1. *Tester les reporteurs***
 - 2. *Tester les constructeurs***
 - 3. *Tester les transformateurs***
 - 4. *Tester les autres***
- ❑ Nous aimerions automatiser cette procédure le plus possible.

Tester les reporteurs

- ❑ Toutes les classes devraient avoir des méthodes 'set' et 'get' pour tous leurs données membres (**interface de classe standard**).
- ❑ Nous allons systématiquement "setter" et "getter" des valeurs des données membres et comparer l'entrée du set avec la sortie du get.
- ❑ **Notez que Bashir et Goel ont pris une approche différente (plus complexe).**

Tester les constructeurs

- ❑ Les constructeurs initialisent les données membres.
- ❑ Nous testons que :
 - Toutes les données membres sont correctement initialisées.
 - Tous les données membres sont initialisées dans un ordre correct.
- ❑ Exécuter le constructeur et ajouter la ou les méthode(s) reporteur(s) pour chaque donnée membre.
- ❑ Vérifier s'ils sont dans un état initial correct (état invariant).
- ❑ Un seul constructeur dans *SampleStatistic*.

Tester les transformateurs

- ❑ Pour chaque tranche d_i :
 1. Instancier l'objet sous test avec le constructeur (déjà testé).
 2. Créer les séquences, e.g., toutes les permutations de méthodes légales dans T_{d_i} :
 - ✓ Maximum # séquences: $|C| * |T_{d_i}|!$
 - ✓ Par exemple, si on a 7 fonctions membres dans T_{d_i} , avec un constructeur, cela mène à 5040 permutations possibles !
 - ✓ Question : est-ce que c'est suffisant pour tester les interactions entre les méthodes ?
 3. Ajouter la ou les méthode(s) reporters (déjà testée(s)).
- ❑ Si plusieurs chemins dans la fonction membre : **Tous les chemins où la tranche d_i est manipulée doivent être exécutés.**

Tester les autres

- ❑ Ils peuvent ne pas utiliser les données membres.
- ❑ Ils ne changent pas l'état de l'objet.
- ❑ Ils ne reportent l'état d'aucune donnée membre ni ne transforment l'état de la classe d'aucune façon.
- ❑ Ils devraient être testés comme toute autre fonction.
- ❑ Seule leur fonctionnalité en tant qu'*entité autonome* a besoin d'être vérifiée.
- ❑ Toute technique standard de test peut être utilisée.

Problèmes

- ❑ Approche basée sur le code, donc plusieurs problèmes potentiels ...
- ❑ Que dire si le code est défectueux et **une méthode qui devrait accéder à une donnée membre ne le fait pas** ?
Alors, les séquences de méthodes qui interagissent via cette donnée membre ne seront pas testées et l'anomalie peut demeurer non détectée.
- ❑ Comment identifier les séquences de méthodes légales ?
- ❑ Quelle stratégie utiliser quand il y a un grand nombre de séquences légales, possiblement infini ?
 - nous n'avons pas à utiliser "une méthode" une fois seulement dans une séquence !

Tranches de CCoinBox

- ❑ La tranche allowVend :

Selon le code original (buggé) ReturnQrts ne devrait pas être dans cette tranche car il ne manipulait pas allowVend.

➤ $T_{\text{allowVend}}(\text{CCoinBox}) = \{\text{Reset}, \text{AddQrt}, \text{ReturnQrts}, \text{Vend}\}$

- ❑ La tranche curQrts :

➤ $T_{\text{curQrts}}(\text{CCoinBox}) = \{\text{Reset}, \text{AddQrt}, \text{ReturnQrts}, \text{Vend}\}$

- ❑ Comme résultat de l'anomalie, possiblement importante, les séquences n'auraient pas été testées.
- ❑ Quand le code est correct, les deux tranches montrent le même ensemble de méthodes !

Tester les classes dérivées

- ❑ Quand deux classes sont reliées par héritage, une classe est *dérivée* d'une autre classe.
- ❑ La classe dérivée peut ajouter des fonctionnalités ou modifier celles fournies par la classe de *base* – elle hérite des données membres et des méthodes de sa classe de base.
- ❑ Deux options extrêmes pour tester une classe dérivée :
 - **Aplatir (Flatten)** la classe dérivée et tester de nouveau toutes les tranches de la classe de base dans le nouveau contexte.
 - Tester seulement les tranches **nouvelles/redéfinies** de la classe dérivée.

La stratégie de Bashir et Goel

- ❑ Supposons que la classe de base ait été *adéquatement* testée, qu'a-t-on besoin de tester dans la classe dérivée ?
- ❑ Étendre le MaDUM de la classe de base pour générer le $\text{MaDUM}_{\text{derived}}$:
 - Obtenir le MaDUM de la classe de base.
 - Ajouter une ligne pour chaque donnée membre nouvellement défini ou redéfini de la classe dérivée.
 - Ajouter une colonne pour chaque fonction membre nouvellement définie ou redéfinie de la classe dérivée.

Remplir le MaDUM_{Derived}

- Si une fonction membre nouvellement définie m_{derived} appelle une fonction membre héritée m_{base} de la classe de base, alors **la colonne des deux méthodes est combinée** et les résultats sont emmagasinés dans la colonne m_{derived} .

$$C(m_{\text{derived}}) = C(m_{\text{derived}}) \cup C(m_{\text{base}})$$

- Même si, a priori, la classe de base n'a aucune connaissance concernant ses données membres définies dans sa classe dérivée, elle peut encore agir sur eux par dynamic binding et polymorphisme. Un tel scénario arriverait si une fonction membre $m1_{\text{base}}$ appelle une autre fonction membre $m2_{\text{base}}$ et la méthode $m2$ est redéfinie dans une des classes dérivées.

$$C(m1_{\text{base}}) = C(m1_{\text{base}}) \cup C(m2_{\text{derived}})$$

**C(méthode) = la colonne correspondant à la méthode dans le MaDUM.*

Exemple : SampleHistogram

```
Class SampleHistogram : public SampleStatistic {
protected:
    short howmanybuckets;
    int *bucketCount;
    double *bucketLimit;
public:
    SampleHistogram(double low, double hi, double bucketWidth = -1.0);
    ~SampleHistogram();
    virtual void reset();//re-defined
    virtual void operator+=(double); //re-defined
    int similarSamples(double);
    int buckets();
    double bucketThreshold(int i);
    int inBucket(int i);
    void printBuckets(ostream&);
}
```

MaDUM (*SampleHistogram*)

	Samp-Stat	reset	+=	sam- ples	mean	stdDev	var	min	max	conf. (int)	conf. (dbl)	error	Samp- Histo	reset	+=	similar- samples	buckets	bucket- Thres- hold	inBucket	print- buckets
n		t	t	r	o	o	o			o	o			t	t					
x		t	t		o		o							t	t					
x2		t	t				o							t	t					
minValue		t	t					r						t	t					
maxValue		t	t						r					t	t					
howMany- Buckets													t	o	o	o	r	o	o	o
bucket Count													t	t	t	r			r	o
bucket Limit													t		o	o		r		o
	C	T	T	R	O	O	O	R	R	O	O	O	C	T	T	O	R	O	O	O

- ❑ Le MaDUM de (*SampleHistogram*) a huit lignes, trois d'entre elles pour les membres de données locaux, et vingt colonnes, huit pour les fonctions membres locales (2 d'entre elles redéfinies).
- ❑ reset et le += sont tous les deux redéfinis dans *SampleHistogram* et provoquent leur contrepartie dans *SampleStatistics* – ils accèdent conséquemment indirectement à tous les membres de données *SampleStatistics*

La procédure de test d'une classe dérivée

- ❑ *Les attributs locaux* : Similaire au test de la classe de base.
- ❑ *Retester les attributs hérités (I.e., leurs tranches)* :
 - S'ils sont directement ou indirectement accessibles par une méthode nouvelle ou redéfinie de la classe dérivée.
 - Vérification du quadrant du haut à droite du MaDUM_{derived}.
 - Nous devons nous assurer quel attribut hérité doit être mandaté à être testé de nouveau – le MaDUM_{derived} peut être utilisé pour l'automatisation.
 - Un fois que ces attributs hérités sont identifiés, le test de procédure est similaire au test de tranche dans la classe de base, mais en utilisant les méthodes héritées *et* nouvelles/redéfinies.

Discussions

- ❑ La mise en tranche peut ne pas être utile pour plusieurs classes (voir l'exemple CCoinBox).
- ❑ Le # de séquences peut encore être grand.
- ❑ Plusieurs séquences peuvent être impossibles (illégales).
- ❑ L'automatisation ? , e.g., Oracle, séquences impossibles.
- ❑ Manquons-nous beaucoup d'anomalies en testant des tranches indépendamment ? Par exemple, si les anomalies mènent à des tranches incorrectement définies.
- ❑ Implicitement, ciblé vers les classes sans comportement dépendant d'état ? (les transformateurs peuvent avoir besoin d'être exécutés plusieurs fois pour atteindre certains états et révéler des anomalies d'état).

Test orienté objet des classes

- ❑ Introduction
- ❑ Considérations sur l'héritage
- ❑ Test des séquences de méthodes
 - Tranches de données
 - Méthodes de pré-conditions et post-conditions
- ❑ Test basé sur les états
 - Testabilité pour le Test basé sur les états
- ❑ Driver, Oracles et Stubs

Méthodes de pré- et post-conditions

❑ Méthode de pré-condition :

- Un prédicat doit être vrai avant qu'une opération soit invoquée.
- Les contraintes spécifiques qu'un appelant doit satisfaire avant d'appeler une opération.

❑ Méthode de post-condition :

- Un prédicat doit être vrai après qu'une opération soit invoquée.
- Les contraintes spécifiques que l'objet doit assurer après l'invocation de l'opération.

❑ Notations possibles :

- Langage naturel.
- Langage de contrainte d'objet (*Object Constraint Language* (OCL)) - UML.
- Extensions de langages de programmation : Eiffel, JDK 1.4

Exemple - Queue

Class Queue (unbounded queue)

Attribute: Number of elements in the queue, count

Init(q:Queue)

pre: Queue q does not exist

post: Queue q exists and is empty

Empty(q:Queue)

pre: Queue q exists

post: Returns 1 if q is empty (count=0), 0 otherwise (count>0)

Eque(q:Queue, e:Element)

pre: Queue q exists

post: Element e is added to the tail of queue q, and q is not empty (count=old(count)+1)

Dque(q:Queue, e:Element)

pre: Queue q exists and is not empty (count>0)

post: Element e is removed from q (count=old(count)-1)

Top(q:Queue, e:Element)

pre: Queue q exists and is not empty (count>0)

post: The first element is returned (e)

Contrats dans une forme informelle

Approche – Tai et Daniels

1. Obtenir les pré- et post-conditions
 - e.g., des documents UML
 - ou les concevoir
2. Dériver les Contraintes pour les séquences de méthodes à partir des pré- et post-conditions
 - Elles indiquent quelles séquences de méthodes (paires) sont permises ou non et sous quelles conditions.
3. Choisir quel critère
 - Nous définirons 7 critères
4. Dériver les Séquences de méthodes satisfaisant le critère à partir des contraintes des séquences des méthodes.

Contraintes séquentielles (de séquencement)

- Les pré- et post-conditions impliquent des contraintes de séquencement de méthodes pour les paires de méthodes.
- Soient m_1 et m_2 2 méthodes d'une classe, les contraintes de séquencement entre m_1 et m_2 sont définies comme un triplet (m_1, m_2, C)
 - Un tel triplet indique que m_2 peut être exécutée après m_1 sous la condition C .
- C est une expression Booléenne ou un littéral booléen (Vrai, Faux).
 - $C = \text{Vrai} \Rightarrow m_2$ peut toujours être exécutée après m_1 .
 - ✓ La post-condition de m_1 implique la pré-condition de m_2 .
 - $C = \text{Faux} \Rightarrow m_2$ ne peut jamais être exécutée après m_1 .
 - ✓ La post-condition de m_1 implique la négation de la pré-condition de m_2 .
 - $C = \text{BoolExp} \Rightarrow m_2$ peut être exécutée après m_1 sous certaines conditions.
 - ✓ La BoolExp liste les conditions sous lesquelles la séquence est possible (forme normale disjonctive) : $C = C_1 \vee C_2 \vee \dots$

Critères (I)

- ❑ Couverture toujours-valide (T).
 - Chaque contrainte toujours-valide doit être couverte au moins une fois.
 - ✓ i.e., **chaque** $(m1, m2, T)$.
- ❑ Couverture toujours/peut-être vraie (T/pT).
 - Chaque contrainte toujours-valide et chaque contrainte peut-être vraie doivent être couvertes au moins une fois.
 - ✓ i.e., chaque $(m1, m2, T)$ et chaque $(m1, m2, C)$ **utilisant une des disjonctions** dans C (C_1 or C_2 , ou ...).
- ❑ Couverture toujours/peut-être vraie Plus (T/pT+).
 - Chaque contrainte toujours-valide et chaque contrainte peut-être vraie (**utilisant à leur tour chacune des disjonctions**) doivent être couvertes au moins une fois.
 - ✓ i.e., chaque $(m1, m2, T)$ et chaque $(m1, m2, C_i)$ **utilisant chacune des disjonctions** dans C (C_1, C_2 , et ...).

Critères (II)

□ Couverture jamais-valide (F).

- Chaque contrainte jamais-valide doit être couverte au moins un fois.
 - ✓ i.e., chaque $(m1, m2, F)$.

□ Jamais valide/possiblement faux (F/pF).

- Chaque contrainte jamais-valide et **utilisant une des contraintes de disjonction fausses** doit être couverte au moins une fois.
 - ✓ i.e., chaque $(m1, m2, F)$ et chaque $(m1, m2, \text{not}(C))$.

□ Jamais valide/possiblement faux plus (F/pF+).

- Chaque contrainte jamais-valide et **chaque contrainte possiblement fausse** doivent être couvertes au moins une fois.

- Pas $C = C'_1 \vee C'_2 \vee \dots$

- ✓ i.e., chaque $(m1, m2, F)$ et chaque $(m1, m2, \text{not}(C'_1))$, $(m1, m2, \text{not}(C'_2))$,
...

□ Toujours/possiblement vrai plus/jamais/possiblement fausse plus (T/pT +/F/pF+).

Exemple – Queue

Contraintes séquentielles(de séquençement) :

C ₁ . (#,Init,T)	C ₆ . (#,Eque,F)	C ₁₁ . (#,Dque,F)	C ₁₆ . (#,Top,F)
C ₂ . (Init,Eque,T)	C ₇ . (Eque,Dque,T)	C ₁₂ . (Dque,Eque,T)	C ₁₇ . (Top,Dque,T)
C ₃ . (Init,Dque,F)	C ₈ . (Eque,Top,T)	C ₁₃ . (Dque,Init,F)	C ₁₈ . (Top,Eque,T)
C ₄ . (Init,Init,F)	C ₉ . (Eque,Eque,T)	C ₁₄ . (Dque,Dque,C)	C ₁₉ . (Top,Init,F)
C ₅ . (Init,Top,F)	C ₁₀ . (Eque,Init,F)	C ₁₅ . (Dque,Top,C)	C ₂₀ . (Top,Top,T)

où **C** = count>0

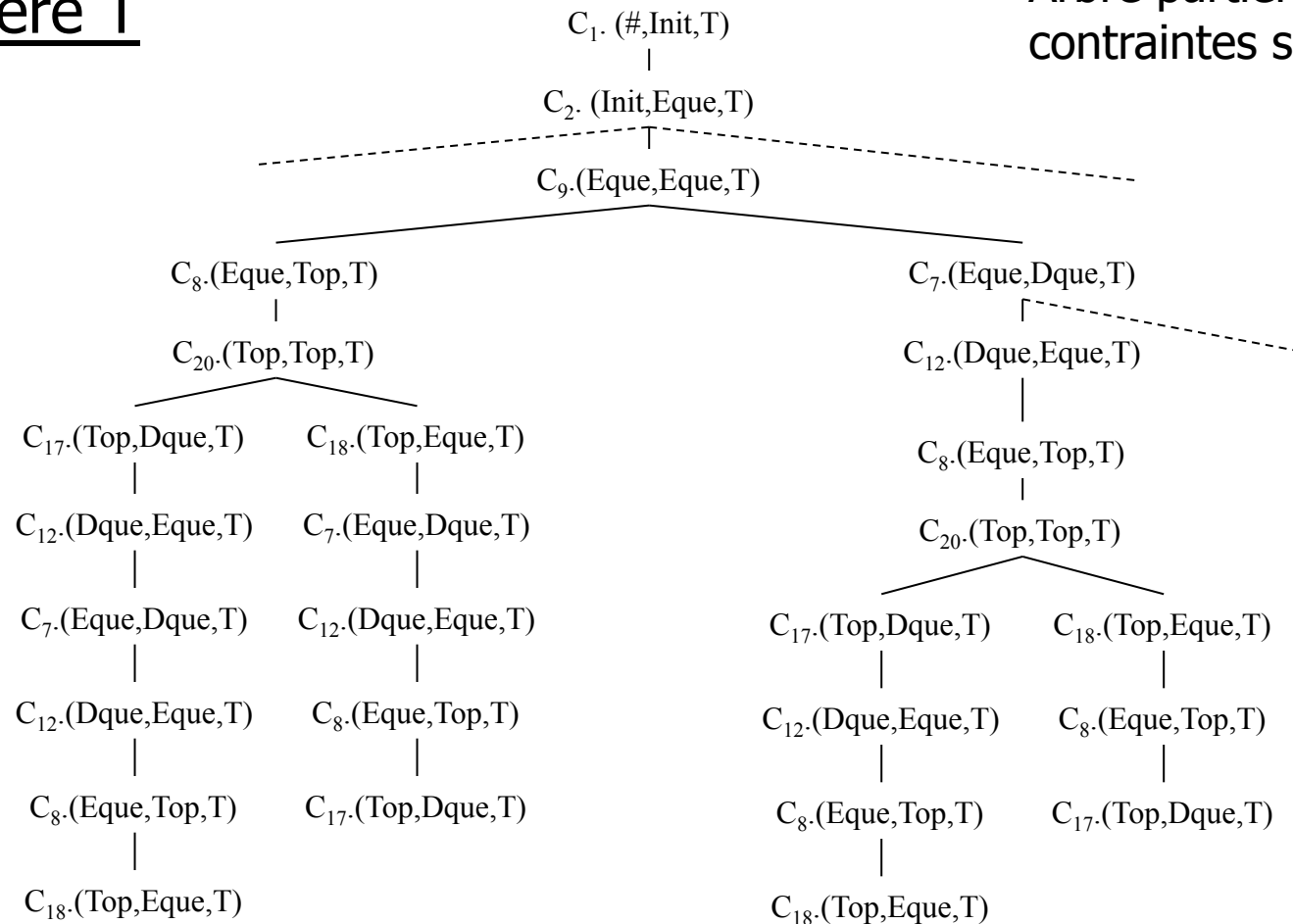
Le critère T requiert l'utilisation de :

C ₁ . (#,Init,T)	C ₂ . (Init,Eque,T)	C ₇ . (Eque,Dque,T)	C ₁₂ . (Dque,Eque,T)
C ₁₇ . (Top,Dque,T)	C ₈ . (Eque,Top,T)	C ₁₈ . (Top,Eque,T)	C ₉ . (Eque,Eque,T)
C ₂₀ . (Top,Top,T)			

Exemple - Queue

Critère T

Arbre partiel basé sur les contraintes séquentielles



Discussion

□ Automatisation ?

➤ Pré/post → contraintes séquentielles → séquences 'complètes'.

□ Plusieurs séquences peuvent être adéquates pour un critère particulier.

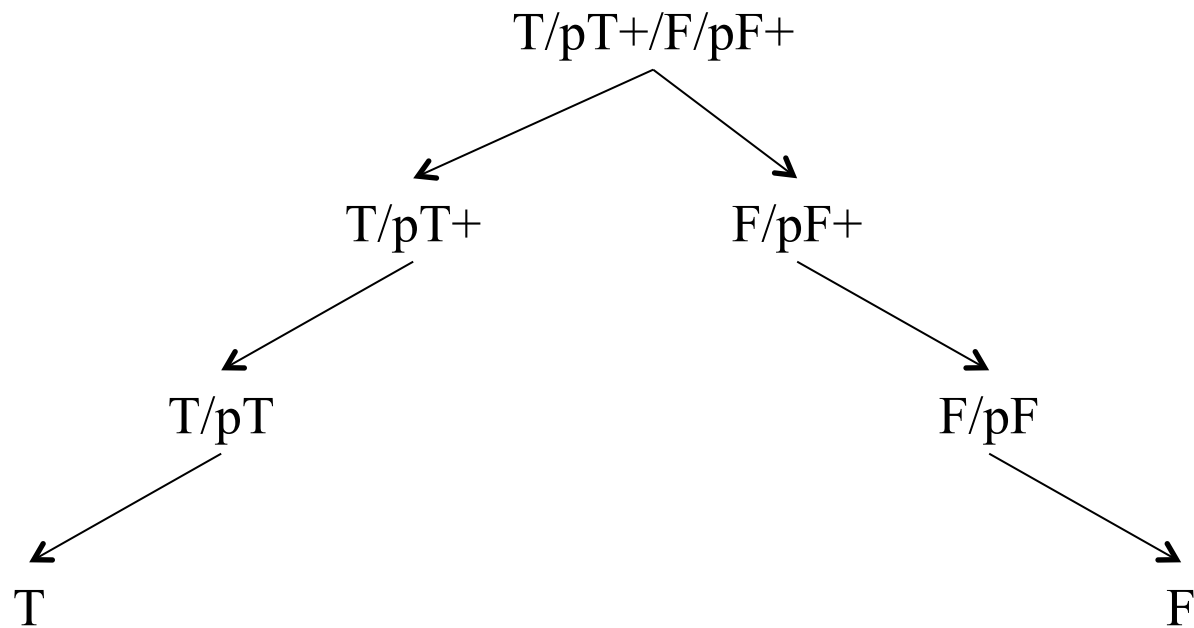
➤ Quelle branche de l'arbre choisir? Sont-elles équivalentes en termes de détection de fautes ? Besoin de couvrir quelques contraintes pT pour couvrir certaines contraintes de T?

□ Sélectionner un autre critère que T.

➤ Critère F: Une représentation autre que celle de l'arbre peut être nécessaire.

□ Comparaison empirique de différents critères (*voir transparent suivant*).

Subsumption (hierarchie de critères)



Étude empirique

- ❑ De Daniels et Tai.
- ❑ 3 programmes C++.
- ❑ Outil de mutation Proteum pour C.
- ❑ 71 opérations de mutation.
- ❑ Les critères ne sont pas exactement ceux définis ici pour T/pT+ et F/pF+.
- ❑ La plupart des mutants ont été éliminés avec le critère de type T et tous avec le type T/pT.

Test orienté objet des classes

- ❑ Introduction
- ❑ Considérations sur l'héritage
- ❑ Test des séquences de méthodes
 - Tranches de données
 - Méthodes de pré-conditions et post-conditions
- ❑ Test basé sur les états
 - Testabilité pour le Test basé sur les états
- ❑ Driver, Oracles et Stubs

Les méthodes de Chow pour le test du modèle d'état

- ❑ Offutt définit le critère de couverture, mais il ne propose pas de méthodes pour *automatiser* le test des séquences d'état.
- ❑ Un des premiers articles sur le sujet est celui de Chow (1978).
- ❑ Il n'offre pas de mise en garde sur les conditions de transitions (les opérations et actions seulement).
- ❑ La première étape est de générer une *transition* ou un *test arborescent* du diagramme d'état.
- ❑ Les chemins arborescents incluent tous les chemins *aller et retour* d'état-transition (comme défini par Binder) : **les séquences de transition qui commencent et se terminent avec le même état** (sans répétition de l'état autre que la séquence du commencement/fin de l'état) et *chemins simples de l'état initial à l'état final du modèle d'état*.
 - Un *chemin* ici est une séquence de transitions : état, state_p, event_i, state_q, event_j, state_r, ...
 - Un *chemin simple* ne contient pas de boucle d'itération.
- ❑ Ajouter chaque séquence avec l'ensemble de caractérisations (W) ou un appel à un 'état' / méthode get_state (assertion).

Procédure pour dériver l'arbre

- ❑ État initial comme le nœud de la racine de l'arbre.
- ❑ Une branche est tiré pour chaque transition en-dehors du nœud initial, avec des nœuds ajoutés comme des états résultats.
- ❑ Un nœud feuille est marqué comme terminal si l'état qu'il représente a déjà été tiré ou c'est l'état final.
- ❑ Pas d'autre transitions sortant d'un nœud terminal sont tracées.
- ❑ Cette procédure est répétée jusqu'à ce que tous les nœuds feuilles soient terminaux.
- ❑ La structure arborescente dépend de l'ordre dans lequel les transitions sont tirées (largeur d'abord ou profondeur d'abord).
- ❑ Une recherche en profondeur d'abord produit moins nombreuses mais plus longues séquences de test.
- ❑ L'ordre dans lequel les états sont examinés est supposé être non relevant

Diagramme d'état de CCoinBox

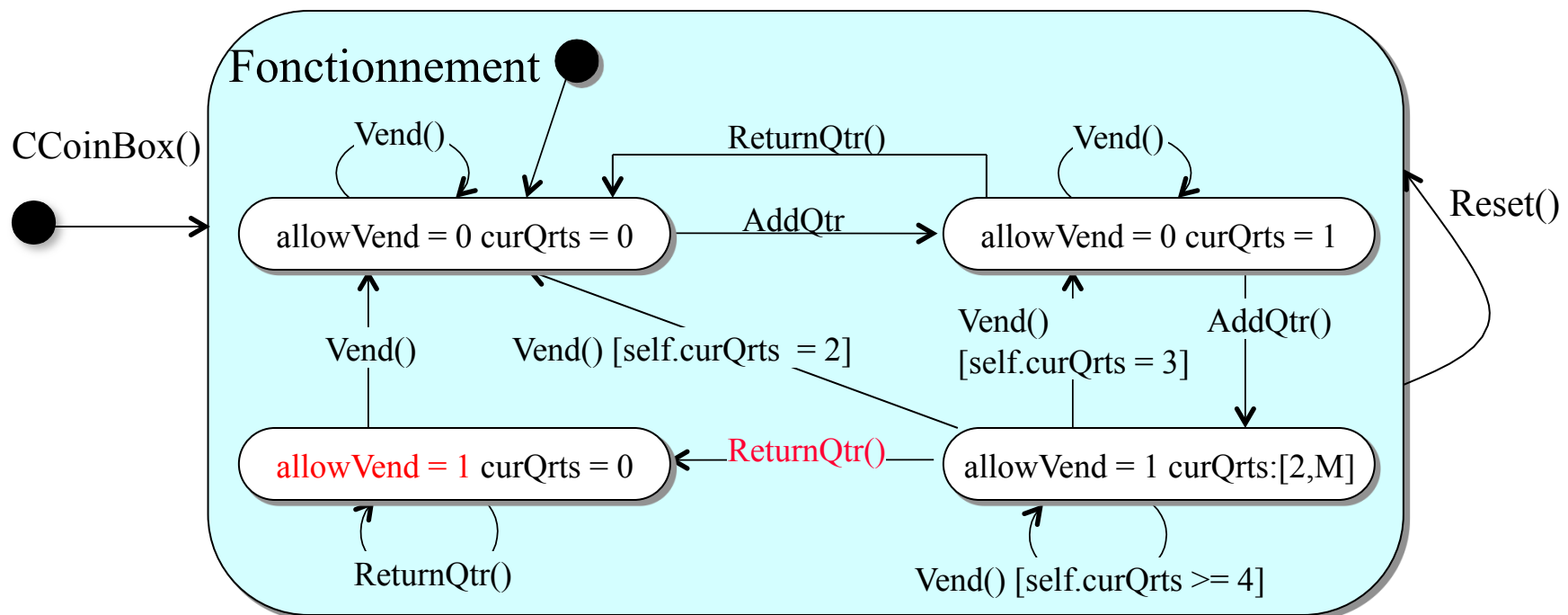


Diagramme d'état correspondant **au code**, pas complètement spécifié

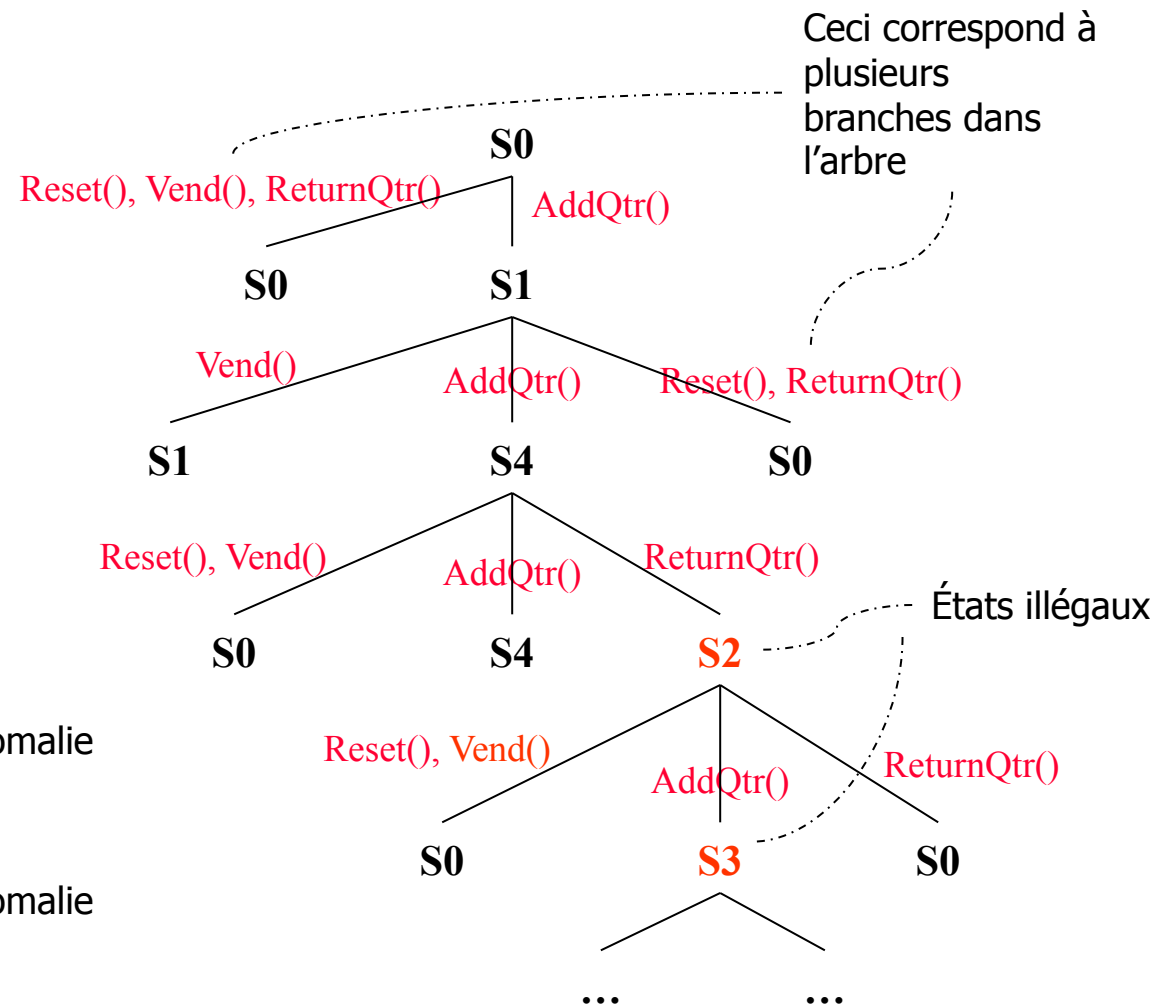
- Un état corrompu manquant : allowVend = 1 curQrts = 1
- Une transition manquante dans l'état allowVend = 1 curQrts = 0 : AddQrts()
- allowVend = 0 curQrts = 0: ReturnQtr() allowVend = 1 curQrts: [2,M]: AddQrt()

Arbre de test pour CCoinBox

- Voir 41

- Basé sur le diagramme d'état (défectueux) présenté précédemment.
- Nœud de la racine = état initial
- S3 n'était pas dans le diagramme d'état (état corrompu manquant) et n'est pas un nœud terminal.

- S0: allowVend = 0, curQtrs = 0
- S1: allowVend = 0, curQtrs = 1
- S2: allowVend = 1, curQtrs = 0
(état illégal rendu possible par une anomalie dans le code)
- S3: allowVend = 1, curQtrs = 1
(état illégal rendu possible par une anomalie dans le code)
- S4: allowVend = 1, curQtrs > 1



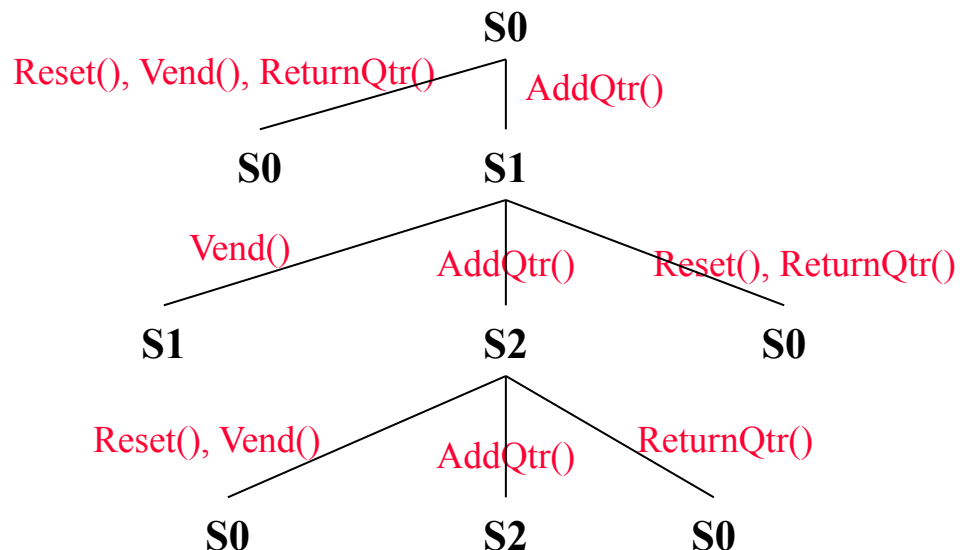
Test arborescent de la CCoinBox (correct)

Basé sur le diagramme d'état correct, modélisé comment le code devrait se comporter.

S0: allowVend = 0, curQtrs = 0

S1: allowVend = 0, curQtrs = 1

S2: allowVend = 1, curQtrs > 1



Utilisant cette transition arborescente, avec le programme `CCoinBox` défectueux, le `ReturnQtr()` en S2 ne devrait pas mener au S0 mais à un état corrompu – mais il devrait être observable sauf si nous avons un moyen pour accéder directement à l'état de `CCoinBox`, ou si nous essayons de vendre.

De l'arbre de test aux cas de test

- ❑ Chaque cas de test commence au nœud racine et finit au nœud feuille.
- ❑ Le **résultat attendu** (Oracle) **est la séquence des états et actions** (sorties, changement d'état d'autres objets).
- ❑ Les cas de test sont complétés en identifiant les valeurs **des paramètres de la méthode** et des **conditions requises** pour traverser un chemin.
- ❑ Nous exécutons les cas de test en plaçant l'objet sous test, à l'état *initial*, appliquer la séquence, et après vérifier les états *intermédiaires*, l'état *final* et les sorties (e.g., logged).

Conditions de garde

- ❑ La garde est une expression booléenne ou contient seulement des opérateurs logique *and* : une combinaison *vraie*.
- ❑ La garde est une expression booléenne composée contenant au moins un opérateur logique *or*. Une transition est requise pour chaque combinaison *vraie*.
- ❑ La garde spécifie une association qui se produit seulement après quelques événements plusieurs fois : arc simple annoté avec * pour la transition.
- ❑ Au moins une combinaison fausse dans tous les cas (voyez les chemins furtifs).

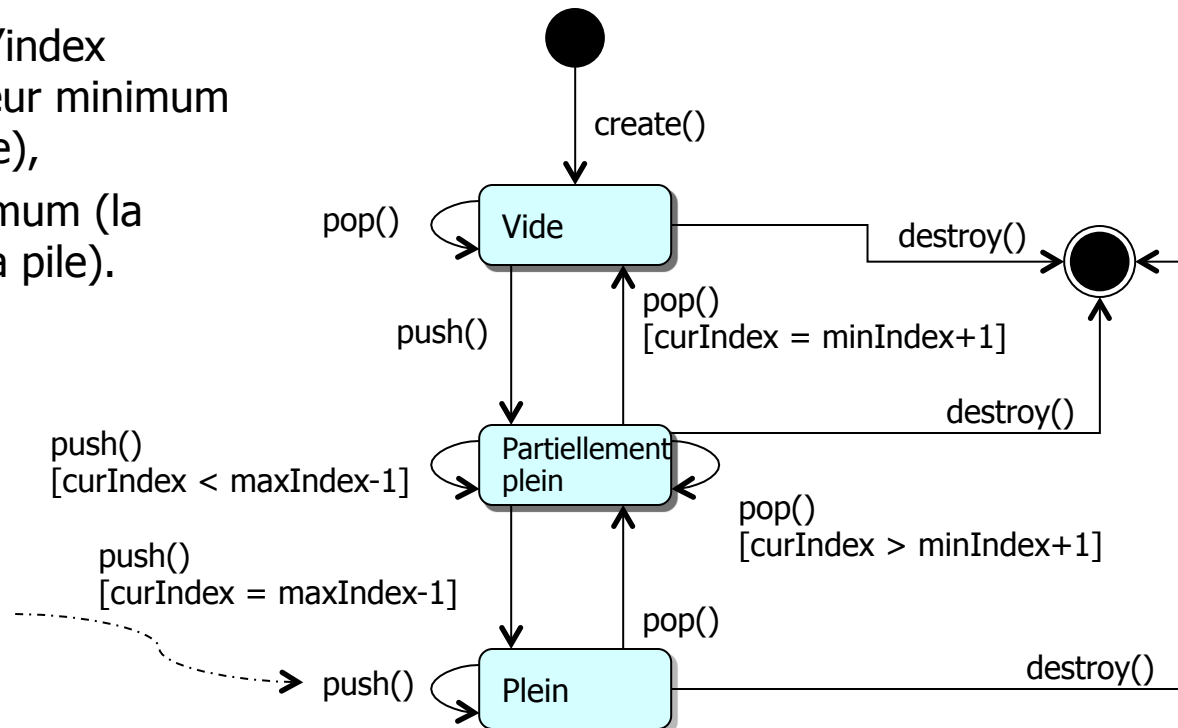
Exemple: BoundedStack

Supposons que trois membres de données sont définis dans la classe :

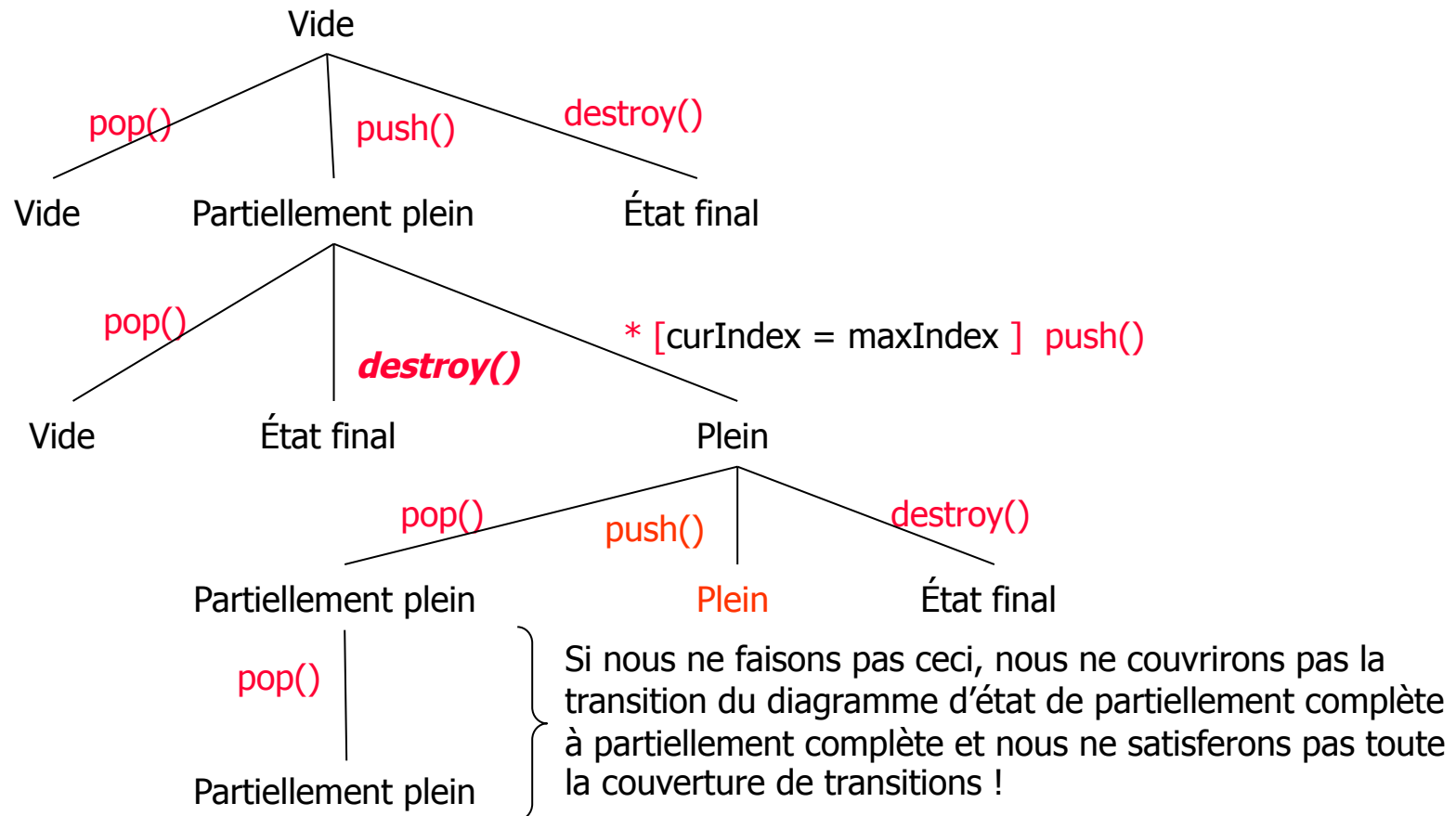
- `curIndex` : l'index courant du dernier élément introduit dans la pile,
- `minIndex` : la valeur de l'index minimum (e.g., 0 grandeur minimum des éléments dans la pile),
- `maxIndex` : l'index maximum (la grandeur maximum de la pile).

Comment manipulons-nous des transitions multiples du même état source, avec le même événement, menant à différents états cibles, basé sur les conditions de garde ?

Supposons que, par exemple, qu'il y a un bogue dans la manière avec laquelle *push()* manipule une pile pleine



Arbre de transitions (BoundedStack)



Driver de test pour BoundedStack

```
int boundedStack_test_driver() {
    BoundedStack stack(2);
    stack.push(3);           // push() when empty
    stack.push(1);           // push() when partially-full
    try {Stack.push(9);} // push() when full
    catch (Overflow ex) {} // expected to throw
    stack.pop();             // pop() when full
    stack.pop();             // pop() when partially-full
    try {stack.pop();}       // pop() when empty
    catch (Underflow ex) {} // expected to throw
    BoundedStack stack2(3);
    stack2.push(6);           // stack2 is partially-full
    stack2.push(5);           // stack2 is still partially-full
    BoundedStack stack3(1);
    stack3.push(6);           // stack3 is full
    // destructors called implicitly at end of block for
    // stack (empty), stack2 (partially-full) and stack3 (full)
};
```

Test orienté objet des classes

- ❑ Introduction
- ❑ Considérations sur l'héritage
- ❑ Test des séquences de méthodes
- ❑ Test basé sur les états
- ❑ Driver, Oracles et Stubs

Drivers

- ❑ Supposons que vous ayiez une opération `diff` qui calcule la différence entre deux ensembles ordonnés (i.e., des éléments qui sont dans le premier ensemble mais pas dans le second).

```
// Java code chunk
```

```
OrdSet s1, s2, s3;
```

```
...
```

```
s3 = s1.diff(s2); // s3=s1-s2, e.g., {1,5}={1,4,5}-{4}
```

```
System.out.println(s3);
```

- ❑ Comment exécuter les cas de test en utilisant la technique de la boîte noire ou celle de la boîte blanche ?
 - Comment construirez-vous le driver de test qui exécute vos cas de test ?

Drivers – Première solution

```
public class Driver {  
    public static void main(String argv[]) {  
        // The test case consists in s1={1,4,5} and s2={4}  
        OrdSet s1 = new OrdSet();  
        OrdSet s2 = new OrdSet();  
        OrdSet s3 = new OrdSet();  
        s1.add(1); s1.add(5); s1.add(4); // adding elements to sets  
        s2.add(4);  
        s3 = s1.diff(s2);  
        System.out.println(s3);  
    }  
}
```

- ❑ Tester l'opération requiert plusieurs cas de test.
- ❑ Solution 1 : écrire tous les cas de test dans une fonction principale du Driver :
 - Problèmes: On ne peut pas exécuter les cas de test sélectivement (par ex pour des tests de régression). On ne peut pas avoir plusieurs ensembles différents de cas de test pour une classe.

Drivers - Seconde solution

- Une méthode statique par ensemble de tests (c'est-à-dire TS1, TS2, ...).
- Une méthode statique par cas de test (c'est-à-dire TC1, TC2, ...).
- Les ensembles de tests peuvent partager les cas de tests.

```
public class Driver {  
    public static void main(String argv[]) {  
        TS1(); // Test set 1  
    }  
    public static void TS1() {  
        TC1();  
        TC2();  
        ...  
    }  
    public static void TS2() {  
        TC5();  
        TC2();  
        ...  
    }  
}
```

```
public static void TC1() {  
    OrdSet s1 = new OrdSet();  
    OrdSet s2 = new OrdSet();  
    OrdSet s3 = new OrdSet();  
    s1.add(1); s1.add(5);  
    s1.add(4);  
    s2.add(4);  
    s3 = s1.diff(s2);  
    System.out.println(s3);  
}  
public static void TC2() {...}  
...  
}
```

Discussion

- ❑ Toutes les méthodes statiques qui exécutent les cas de test ont (normalement) la même structure.
 - Ici : créer les ensembles s1 et s2, et appeler le `diff`.
- ❑ Une technique de test peut produire des centaines de cas de tests.
- ❑ Qu'arrive-t-il si nous voulons ajouter des cas de test ?
 1. Ajouter les méthodes statiques TCxxx et TSxxx.
 2. Ajouter les appels de ces méthodes dans le main.
 3. Compiler le Driver.
 4. Exécuter le Driver.
- Créer différents tests pour différents buts, e.q., ensembles de tests de régression successive.
- Peut changer un énoncé dans le `main()` pour exécuter les différents ensembles de tests.

Drivers – Troisième solution

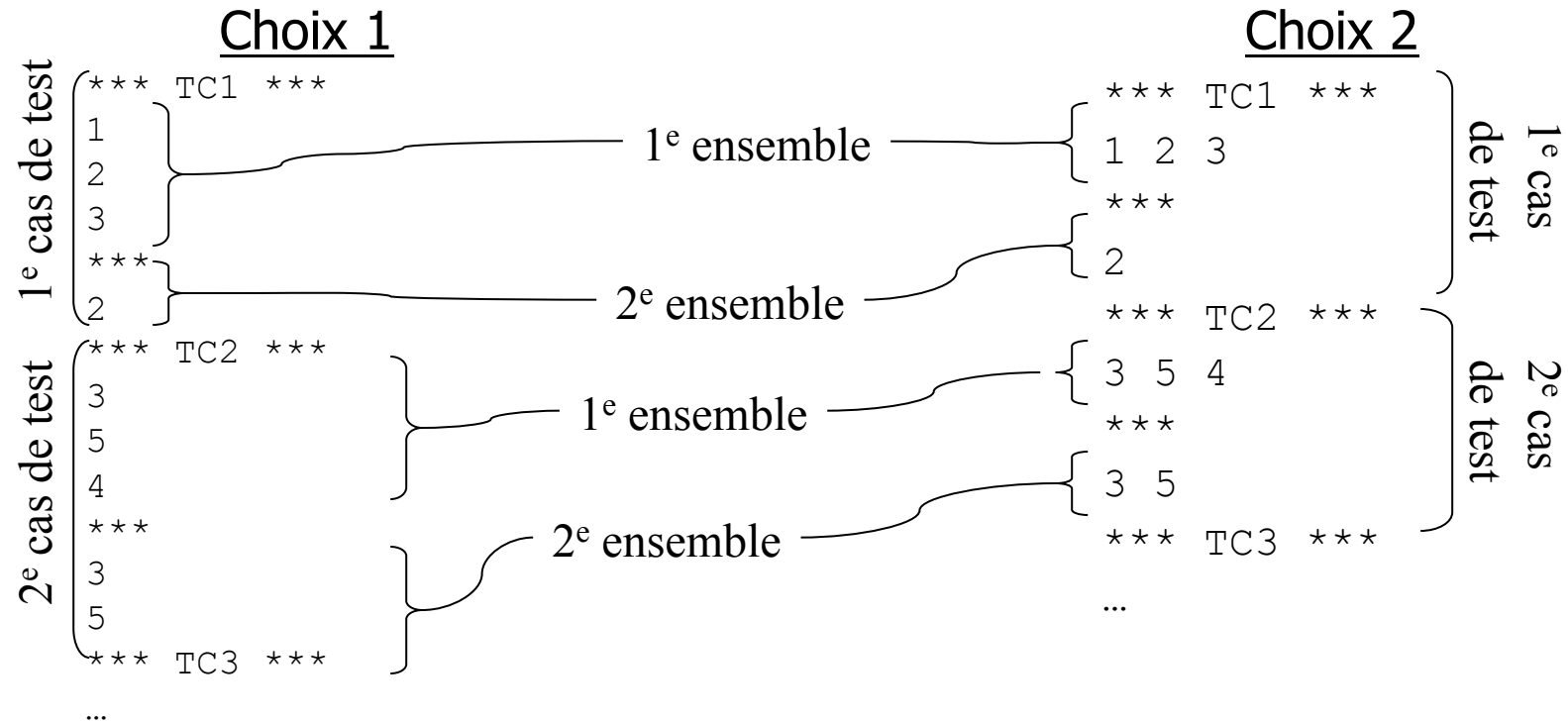
- La fonction principale dans le Driver lit un fichier texte :
 - 1 ensemble de tests par fichier de texte.
 - Pas besoin de recompiler quand on change de cas de test.
 - Le Driver doit lire le fichier texte, créer les objets en accord avec les contenus et exécuter le diff.

```
public class Driver {  
    public static void main(String argv[]) {  
        OrdSet s1, s2, s3  
        // open file and read test set name ...  
        while (not at the end of file) {  
            // read a test case (elements in sets)  
            // instantiate s1 and s2  
            // execute the diff, assign result to s3  
            // output the test case number and the result  
        }  
    }  
}
```

Discussion

- ❑ Quel est le format du fichier texte (nous devons identifier le cas de test) ?
 - Plusieurs cas de test dans le fichier texte.
- ❑ Le Driver (le main) a besoin :
 - d'identifier les cas de test :
 - ✓ le commencement et la fin d'un cas de test dans le fichier,
 - ✓ nombre/nom du cas de test.
 - de savoir comment lire les données dans les cas de test
 - ✓ la séparation entre les différents champs dans le cas de test (e.g., les deux ensembles).
- Définir un standard.

Ensemble de tests format fichier



Des Drivers plus complexes

E.g., les cas de test sont différents (différents types d'exécutions/données)

Solutions :

1. Plusieurs Drivers différents (fonctions principales) avec différents flux d'exécution (i.e., avec différents formats de fichiers).
 - Driver (principal) un : format fichier 1.
 - Driver (principal) deux : format fichier 2.
 - L'implémentation dépend du langage de programmation.
2. Un Driver avec plusieurs données possibles (e.g., arguments de ligne de commande, ou dans le fichier texte).
 - Driver (principal) avec donnée 1 (argument en ligne de commande) : format fichier 1.
 - Driver (principal) avec donnée 2 (argument en ligne de commande) : format fichier 2.
 - Implémentation indépendante du langage (nous passons l'information sur la ligne de commande).

Des Drivers plus complexes (en Java)

- ❑ En Java, nous pouvons avoir différentes classes de Drivers, chacune ayant une fonction principale.
 - Dans le même répertoire, nous devrions avoir les fichiers OrdSet.java, Driver1.java, Driver2.java, ...
 - et Driver1.java, Driver2.java, ..., tous ont une fonction principale.
- ❑ Nous choisissons seulement quel main (classe de Driver) nous voulons à exécuter quand nous exécutons la Machine Virtuelle Java.
 - java Driver1 [...]
 - java Driver2 [...]
 - ...
- ❑ La solution peut être utilisée même si nous testons un programme (pas une classe), qui a déjà une fonction principale.
 - Les Drivers peuvent être considérés comme « points d'entrée » additionnels dans le programme.

Des Drivers plus complexes (en C++)

- ❑ Compilation conditionnelle en C/C++ :
 - Rendre le programmeur capable de contrôler l'exécution des directives de pré-processeur et la compilation du code du programme.
 - Utiliser les directives de pré-processeur `#define`, `#ifdef`, `#ifndef` et `#endif`
- ❑ Deux solutions :
 - Enlever/Ajouter les directives de pré-processeur.
 - ✓ Requiert la modification des fichiers de source et de compilation.
 - Utiliser les options de compilateur.
 - ✓ Requiert seulement la compilation.
- ❑ Les solutions peuvent être utilisées même si nous testons un programme (pas une classe), qui a déjà une fonction principale.
 - Les Drivers peuvent être considérés comme des « points d'entrée » additionnels dans le programme.

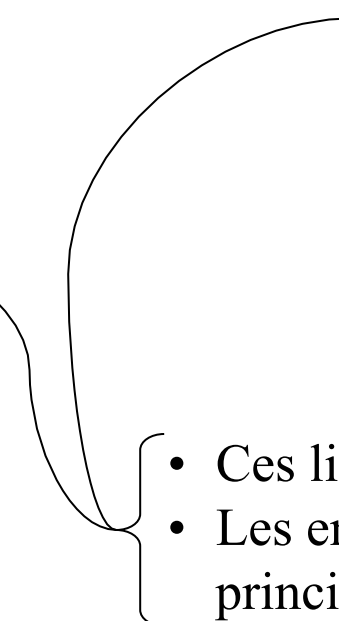
Des Drivers plus complexes (en C++)

```
// File: ClassTester.h
#include "ClassTested.h"
class ClassTester {
public:
    ClassTester();
    void runTestSuite();
private: ...
};
```

```
// File: MainProg.cpp
#define DRIVER
#ifndef DRIVER
#include "ClassTested.h"
class MainProg {
public: ...
private: ...
};
#endif
```

```
// File: ClassTester.cpp
#include "ClassTester.h"
ClassTester::ClassTester() {...}
void ClassTester::runTestSuite() {...}
// code to run and report test cases
```

```
// File: ClassTesterMain.cpp
#define DRIVER
#ifdef DRIVER
#include "ClassTester.h"
int main() {
    ClassTester tester;
    tester.runTestSuite();
}
#endif
```

- 
- Ces lignes assurent que le Driver s'exécute.
 - Les enlever fait que le programme principal s'exécute.

Des Drivers plus complexes (en C++)

```
// File: MainProg.cpp
#ifndef DRIVER
#include "ClassTested.h"
class MainProg {
public: ...
private: ...
};
#endif
```

```
// File: ClassTesterMain.cpp
#ifdef DRIVER
#include "ClassTester.h"
int main() {
    ClassTester tester;
    tester.runTestSuite();
}
#endif
```

```
// File: ClassTester.cpp
#ifdef DRIVER
#include "ClassTester.h"
ClassTester::ClassTester() {...}
void ClassTester::runTestSuite() {...}
// code to run and report test cases
#endif
```

- ❑ Exécuter le main du programme :

➤ le main en fichier MainProg.cpp

```
cc -c MainProg.cpp
```

```
cc -c ClassTester.cpp
```

```
cc -c ClassTesterMain.cpp
```

```
cc MainProg.o ClassTester.o ClassTesterMain.o -
o Exe
```

- ❑ Exécuter le main du Driver :

➤ le main en fichier ClassTesterMain.cpp

(option compilateur -D définit les constantes du Driver)

```
cc -DDRIVER -c MainProg.cpp
```

```
cc -DDRIVER -c ClassTester.cpp
```

```
cc -DDRIVER -c ClassTesterMain.cpp
```

```
cc MainProg.o ClassTester.o ClassTesterMain.o -
o Exe
```

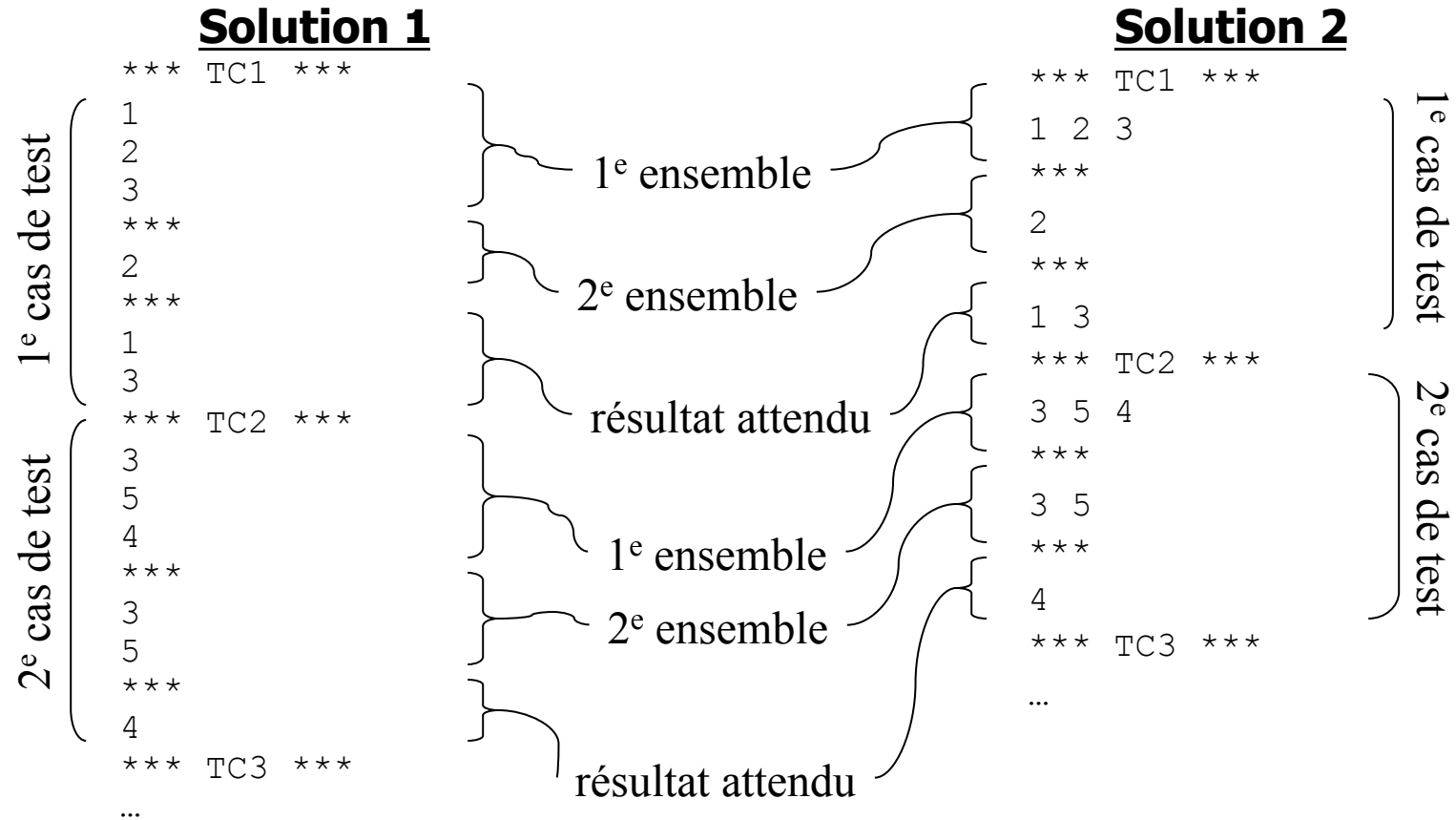
L'Oracle

- ❑ Dans les approches précédentes pour l'implémentation du Driver, le Driver signale les sorties (« print output »).
- ❑ Décider de quels cas de test ont échoué (une anomalie a été révélée) se fait manuellement.
 - Comparer la sortie attendue avec celle observée, pour chacun des cas de test.
- ❑ Automatiser l'oracle, i.e., la comparaison requiert que :
 - Nous connaissons exactement la sortie attendue (qui est le cas dans l'exemple du `diff`).
 - ✓ Cependant, quelques fois, nous ne connaissons pas exactement le résultat, comme (par exemple) faire (i.e., implémenter) la comparaison qui correspondrait à construire la fonction que nous testons.
 - Puis que le Driver fasse sortir le numéro de cas de test quand un échec a été détecté.
 - ✓ Nous avons seulement besoin de connaître quels cas de test ont échoué.

L'Oracle

- ❑ L'oracle est inclus dans le Driver.
 - La sortie attendue doit être trouvée dans le fichier texte de l'entrée.
 - ✓ Format ?
- ❑ L'oracle:
 - Obtient la sortie produite par le système sous test.
 - ✓ Dans notre exemple, la méthode `toString` de la classe `OrdSet` est appelée (format du lien de retour ?)
 - Obtient la sortie attendue du fichier texte de l'entrée.
 - Effectue la comparaison.
 - ✓ Dans notre exemple, une simple comparaison de lien.

L'Oracle



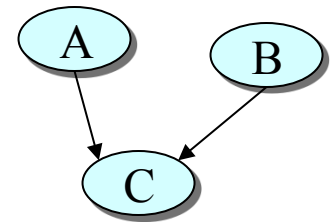
Les Stubs

❑ Besoin pour les Stubs :

- Des parties du systèmes qui n'ont pas encore effectué les test unitaires ou ne sont même pas disponibles (i.e., le code n'est pas prêt).
- La simulation des dispositifs matériels.

❑ Exemple 1:

- Les modules A et B utilisent des services fournis par le module C.
 - ✓ Ici, les modules peuvent être des fonctions, des classes, des sous-systèmes.
- Mais : A utilise seulement une partie des services de C, disons C_A.
B utilise seulement une partie des services de C, disons C_B.
- Alors :
 - ✓ Quand A est testé, nous créons une Stub simulant le comportement de C_A.
 - ✓ Quand B est testé, nous créons une Stub simulant le comportement de C_B.



❑ Commentaire : "Si le Stub f est réaliste dans tous les sens, ce n'est plus un Stub mais la routine elle-même" [Beizer]

Les Stubs

Exemple 2 :

- Un guichet automatique (*automated Teller Machine* (ATM)) a un clavier (i.e., le dispositif matériel) et une classe *Keyboard*.
- Durant le développement et le test, un Stub est développé pour la classe *Keyboard* qui n'est "connectée" à aucun dispositif matériel.
 - ✓ De préférence, c'est le Driver qui doit alimenter le système ATM avec un numéro NIP, un solde de compte, ...
- Cas de test typique :
 1. Entrer le NIP, 2. Sélectionner le compte, 3. Entrer un montant.
 - ✓ Chaque fois, le système ATM provoque les méthodes du Stub du *Keyboard*.
 - ✓ De telle manière que le Stub du *Keyboard* doit retourner la valeur correcte selon le cas de test.
 - ✓ Le Driver doit mettre en place différentes valeurs durant l'exécution d'un cas de test.
 - ◆ La classe *Keyboard* sait où (e.g., un fichier) lire les différentes entrées.

Autres considérations

❑ **Non-déterministe**

- Que faire si le comportement du système sous test n'est pas déterministe ?
- Systèmes concurrents
- Impact sur les cas de test, Drivers, Stubs, oracles.

❑ **Boucle infinie à cause d'une anomalie**

- Comment décidons-nous (quand l'exécution des cas de test est automatisée) qu'un cas de test a échoué ?

❑ **Le système sous test a un GUI**

- Le GUI est enlevé pour faciliter l'automatisation durant le test (unitaire/intégration/système), i.e., le GUI a besoin d'être remplacé par le Driver au moment de la compilation.
- Tester un GUI est une tâche séparée.
 - ✓ Ce n'est pas un test fonctionnel.