Programmer pour la fiabilité

Voir: Ian Sommerville, Software Engineering, 5th edition

Réalisation fiable

- ***Évitement de fautes***
  - Le logiciel est développé de telle manière qu'il ne contient aucune faute.
- ***Détection de fautes***
  - Le processus de développement est organisé de telle manière que les fautes dans le logiciel sont détectées et réparées avant la distribution aux consommateurs.
- ***Tolérance aux fautes***
  - Le logiciel est créé de telle manière que les fautes dans le logiciel livré ne résultent pas en une défaillance complète du système.

Tolérance aux fautes : Motivations

- Nous ne pouvons pas réaliser un logiciel avec une fiabilité complète.
- Démontrer une haute fiabilité pour des applications à sécurité critique est difficile.
- Comment pouvons-nous nous assurer un comportement acceptable du système quand une défaillance se produit ?
- E.g., les ordinateurs des systèmes de contrôle de trafic aérien doivent être continuellement disponibles.

Aspects de tolérance aux fautes

- ***Détection de défaillance***: Le système doit détecter qu'une combinaison d'état particulier a résulté ou résultera en une défaillance du système.
- ***Évaluation du dommage*** : Les parties de l'état du système qui ont été affectées par la défaillance doivent être détectées.
- ***Récupération de faute*** : Le système doit restaurer son état à un état connu « sécuritaire ».
- ***Réparation de faute*** : Ceci implique de modifier le système de telle manière que la faute ne se reproduira plus. Pour des systèmes qui ont besoin d'être continuellement disponibles, remplacer le composant ayant la faute est plus complexe.

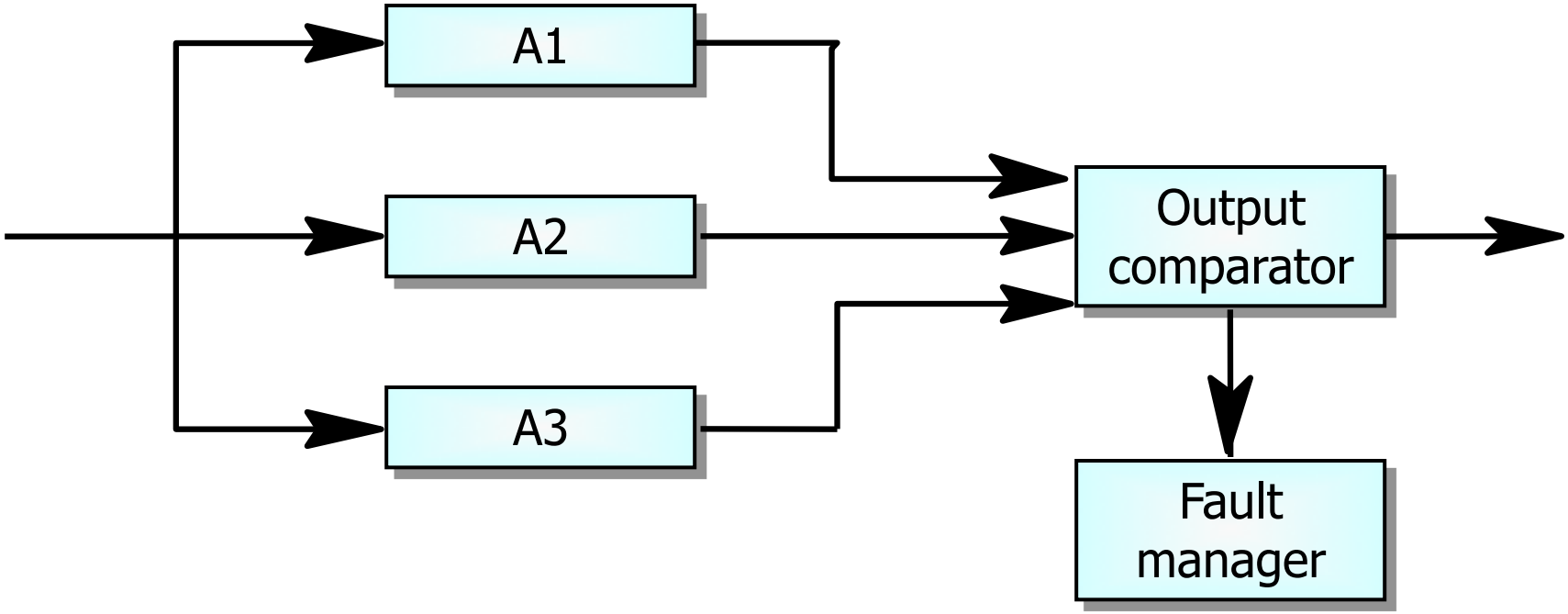
Deux approches principales

- ***Architectures tolérantes aux fautes*** : Support explicite pour la tolérance aux fautes (détection de problème, rétablissement).
- ***Programmation défensive*** : Pas d'architecture spécifique. Mais du codes redondant pour vérifier l'état du système après la modification. Si des incohérences sont détectées, l'état est restauré à un état correct connu.

Tolérance aux fautes matérielles

- ***Redondance modulaire triple*** *(Triple-modular Redundancy (TMR))* : L'unité matérielle est repliquée trois fois (ou plus) et leurs sorties sont comparées.
- Si une unité montre une sortie incohérente, elle est ignorée.
- Cette approche suppose le problème provient à partir des composants en défaillance plutôt que des fautes de conception.
- Faible probabilité de défaillances simultanées des composants dans toutes les unités matérielles.
- Les unités peuvent provenir de différents manufacturiers.

Fiabilité matérielle avec TMR



Sommerville, 1995

Architectures de logiciel tolérant aux défaillances

- Le succès de la TMR à permettre la tolérance aux défaillances est basé sur deux hypothèses fondamentales.
  - Les composants matériels n'incluent pas les fautes communes de la conception.
  - Les composants échouent aléatoirement et il y a peu de probabilité de défaillances de composants simultanées.
- Aucune de ces hypothèses sont vraies pour le logiciel.
  - Ce n'est pas possible de simplement reproduire le même composant parce qu'ils auraient les même fautes communes de conception.
  - Les défaillances simultanées des composants est alors virtuellement inévitables.
- Les systèmes logiciels doivent donc être divers.

Diversité de conception

- Des versions différentes du systèmes sont créées et implémentées de différentes manières. Elles doivent donc avoir des modes de défaillances différentes.
- Des approches différentes de conception (e.g., objet-orientée et orientée fonction).
  - Implémentation dans différents langages de programmation.
  - Utilisation de différents outils et environnements de développement.
  - Utilisation de différents algorithmes dans l'implémentation.

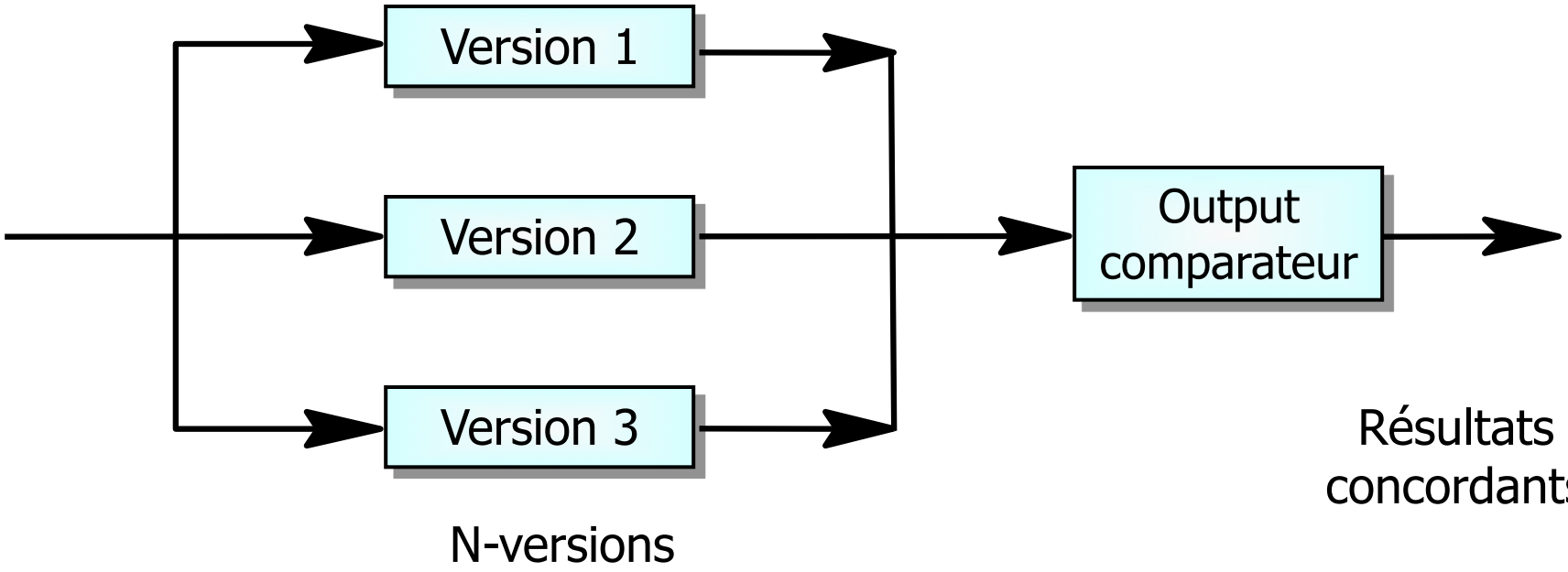
Les analogies du logiciel à la TMR

- Programmation de N-versions.
  - La même spécification est implémentée dans un nombre de versions différentes par des équipes différentes. Toutes les versions calculent simultanément et la sortie majoritaire est sélectionnée en utilisant un système de vote.
  - C'est l'approche la plus communément utilisée, e.g., dans Airbus 320.
- Blocs de récupération.
  - Un nombre de versions **explicitement** différentes de la même spécification sont écrites et exécutées en séquence.
  - Un test d'acceptation est utilisé pour sélectionné la sortie à être transmise.

Programmation de N-versions

- Utilisant la spécification commune, le système logiciel est implémenté dans un nombre de *versions différentes par des équipes différentes.*
- Des versions sont exécutées en parallèle.
- Les sorties sont comparées utilisant un *système de vote* et les sorties incohérentes sont rejetées.
- Au moins trois versions devraient être accessibles.
- ***Hypothèse*** : Il est improbable que différentes équipes fassent les mêmes erreurs de conception et de programmation.
- Toutefois, il y a quelques évidences empiriques que des équipes interprètent mal des spécifications de la même manière et utilisent des algorithmes identiques/similaires dans leurs systèmes.

Programmation de N-versions

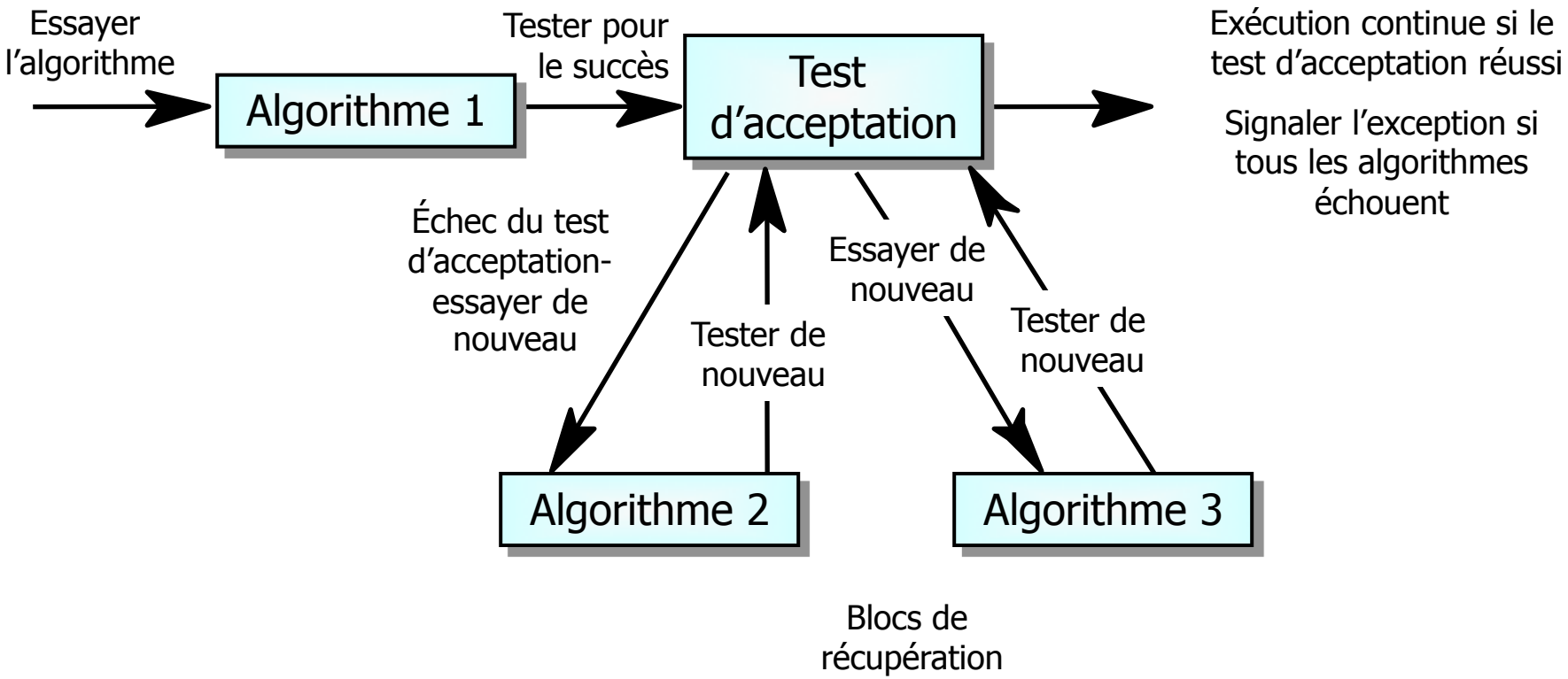


Sommerville, 1995

Blocs de récupération

- L'approche de graine la plus fine pour la tolérance aux fautes.
- Chaque composant du programme inclut un test pour vérifier si le composant est exécuté avec succès.
- Cela inclut un code alternatif pour sauvegarder et répéter le calcul avec un autre algorithme (version) si le test détecte une défaillance.
- Les versions sont exécutées en séquence.
- La sortie qui se conforme à un «test d'acceptation » est sélectionnée.
- Cela réduit la probabilité d'erreurs communes car différents algorithmes DOIVENT être utilisés pour chaque *bloc de récupération.*
- La faiblesse de ce système est d'écrire le test d'acceptation approprié.

Blocs de récupération



Sommerville, 1995

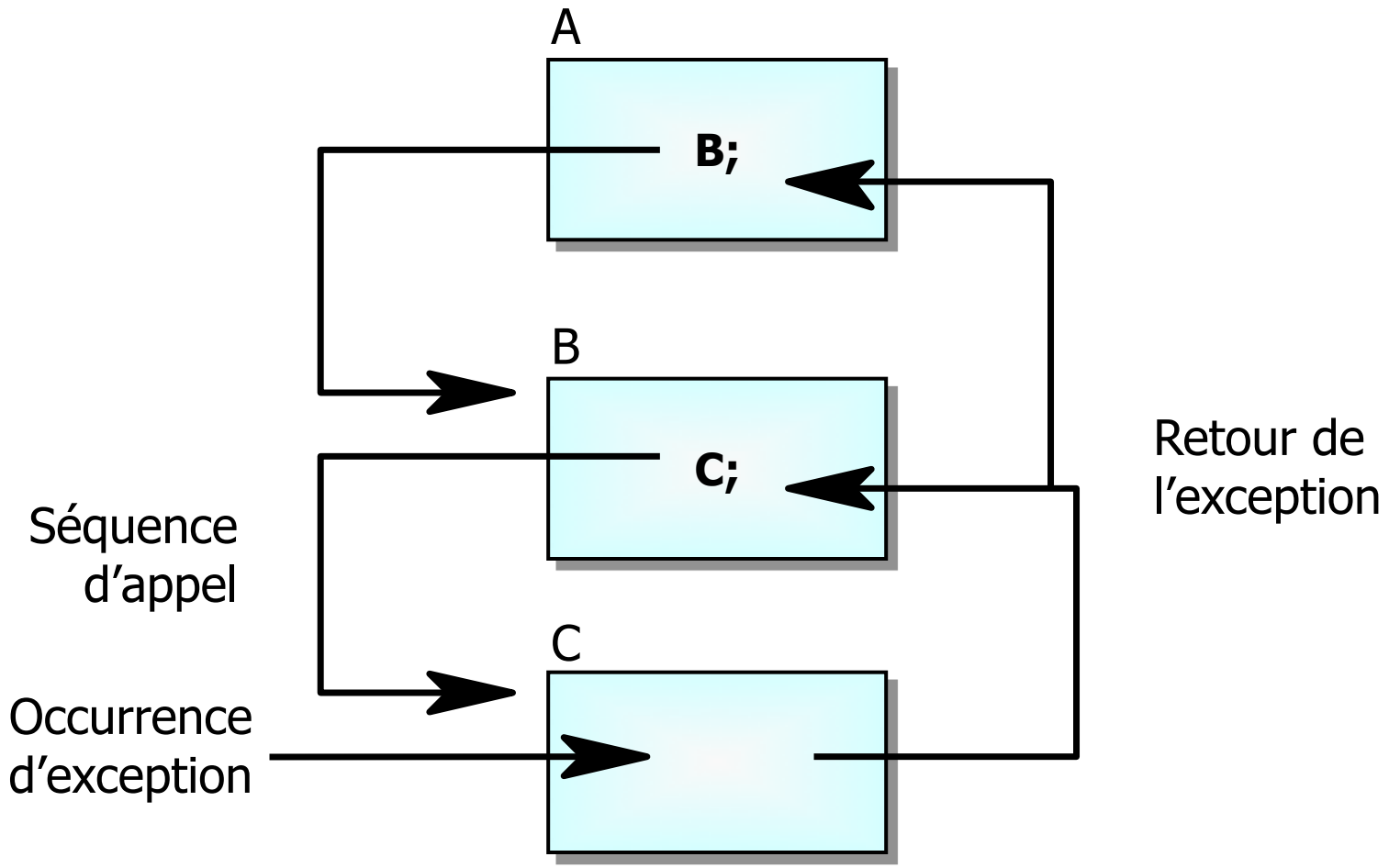
Discussion

- Différentes équipes commettent les même erreurs. Quelques parties de l'implémentation sont plus difficiles que d'autres, c'est pourquoi toutes les équipes ont tendance à faire les mêmes erreurs aux mêmes endroits.
- Programmer N-versions augmente la confiance pourtant, mais pas une confiance absolue.
- Les deux approches à la tolérance de défaillances présentées assument que les *spécifications sont correctes.*
- Les deux requièrent un *contrôleur de tolérance aux fautes* qui assurera que les étapes impliquant la tolérance aux fautes sont exécutées.
- Le contrôleur de tolérance aux fautes peut échouer …

Manipulation des exception

- ***Exception*** : Erreur de l'utilisateur, défaillance matérielle, défaillance logicielle.
- ***Manipulation d'exception*** : Le mécanisme par lequel un système traite une exception.
  - Erreur d'utilisateur : Message d'erreur significatif.
- **Dans les systèmes OO** : Exceptions normalement associées avec des violations de pré-conditions, post-conditions et/ou des invariants de classe.
- L'utilisation normale d'une structure de contrôle (If statement) pour détecter des exceptions dans une séquence d'appels de procédure imbriquée a besoin d'ajouter plusieurs instructions additionnelles au programme et ajoute un surcharge de temps significatif.
- Quelques langages ont des mécanismes incorporés pour les exceptions (e.g., Java, C++).

Exceptions dans des structures imbriquées



Implémentation classique

Rappel

- Des exceptions peuvent être signalées en associant une variable booléenne partagée (e.g., valeur de retour) avec chaque exception.
- Les conditions qui peuvent causer une exception sont testées avant que l'exception survienne.
- Le processus normal est arrêté si l'effet d'une instruction est de causer une exception.
- Dans une séquence d'appels imbriqués, le même test peut être répété plusieurs fois.

Manipulateurs d'Exception (Exception Handlers)

- Quelques langages de programmation inclut de moyens pour détecter et manipuler des exception (Ada, C++, Java).
- Une exception est signalée et contrôlée dans le programme est transférée à un manipulateur d'exception, I.e., un segment de code qui traite la situation exceptionnelle (e.g., *catch block* en Java).
- Les exceptions sont souvent manipulées par un bloc catch (catch block) dans une unité appelante plus élevée que la séquence d'appel, étant donné que les unités appelées ne savent souvent pas quoi faire quand une exception est détectée.

Rappel

Manipuler les exception en Java

- Le mot clé **`throw`** signifie relever une exception. Il ne peut être utilisé que dans un bloc **`try`** ou une fonction (indirectement) appelé par lui. Le manipulateur est indiqué par le mot clé **`catch`.**
- Le bloc **`try`** enveloppe le code qui peut lancer une exception et le code qui ne pourraient pas être exécutés dans ce cas.
- Les exceptions sont définies comme des classes donc peuvent héritées des propriétés d'autres classes d'exceptions. Il y a une classe ***Exception*** pré-définie en Java. Toutes les exceptions sont définies comme des sous-classes d'*Exception.*
- Quand c'est possible, les exceptions sont manipulées complètement dans un bloc où elles se produisent plutôt que de se propager pour être manipulées. Mais ce n'est pas souvent le cas.

Exemple : SensorFailureException

```
class SensorFailureException extends Exception {
   SensorFailureException (String msg) {
      super (msg) ;
      Alarm.activate (msg) ;
   }
} // SensorFailureException

class Sensor {
   int readVal () throws SensorFailureException {
   try {
      int theValue = DeviceIO.readInteger () ;
      if (theValue < 0)
         throw new SensorFailureException ("Sensor failure") ;
      return theValue ;
   }
   catch (deviceIOException e)
      { throw new SensorFailureException (" Sensor read error ") ; }
   } // readVal
} // Sensor
```

Autre exemple

- Le système qui contrôle un congélateur et garde la température à l'intérieur d'un intervalle donné.
- Commuter une pompe réfrigérante on et off.
- Poser une alarme si la température maximum permise est dépassée.
- Utilisation d'objets externe de type `Pump, TempDial, TempSensor, Alarm.`

Exemple : FreezerController (Java)

```
class FreezerController  extends Thread {
        Sensor tempSensor = new Sensor () ;
        Dial tempDial = new Dial () ;//what we need as temperature

        float freezerTemp = tempSensor.readVal () ; // we read the value
        final float dangerTemp = (float) -18.0 ;
        final long coolingTime = (long) 200000.0 ;
        public void run ( )  throws FreezerTooHotException,
InterruptedException {
        try {
                Pump.switchIt (Pump.on) ;
                do { if (freezerTemp > tempDial.setting ())
                                if (Pump.status == Pump.off)
                                {    Pump.switchIt (Pump.on) ;
                                     Thread.sleep (coolingTime) ;
                                }
                  else
                                if (Pump.status == Pump.on)
                                      Pump.switchIt (Pump.off) ;
                        if (freezerTemp > dangerTemp) // check and throw
                                 throw new FreezerTooHotException () ;
                        freezerTemp = tempSensor.readVal () ;
                } while (true) ;
        }  // try block
        catch (FreezerTooHotException f)
        {       Alarm.activate ( ) ;   }
        catch (InterruptedException e)
        {       System.out.println ("Thread exception") ;
                throw  new InterruptedException ( ) ;
        }
        } //run
} // FreezerController
```

Programmation défensive

- Pas besoin de contrôleur de tolérance aux fautes.
- Ne suppose pas de spécifications correctes.
- Un *code redondant* est incorporé pour prévenir un changement d'état incorrect et vérifier l'état du système après une modification.
- Si incohérent, le changement d'état est rétracté ou restauré à un état connu.
- Une approche commune de la tolérance aux fautes

Prévention de défaillances

- Une approche est d'utiliser l'*assertion d'état* pour vérifier si certaines contraintes sont remplies.
- Les prédicats logiques sur des variables d'état (invariant d'état en termes UML).
- Ce prédicat est vérifié avant qu'une assignation soit faite à une variable d'état.
- Si une *valeur anormale* pour la variable résulterait d'une assignation, alors une erreur s'est produite.
- Dans la plupart des langages de programmation, c'est au programmeur d'inclure des *vérifications explicites d'assertion.*
- Peut être simplifié si toutes les assignations aux variables d'état sont toujours implémentées comme les opérations (méthodes) dans les objets – le code d'assertion fait partie de l'opération.

Exemple : Even Number Class

```
class PositiveEvenInteger {
        int val = 0 ;

        PositiveEvenInteger (int n) throws NumericException
        {
            if (n < 0 l n%2 == 1)
                throw new NumericException () ;
            else
                val = n ;
        } // PositiveEvenInteger

        public void assign (int n) throws NumericException
        {
            if (n < 0 l n%2 == 1)
                throw new NumericException ();
            else
                val = n ;
        } // assign

       int toInteger ()
        {
            return val ;
        } //to Integer

        boolean equals (PositiveEvenInteger n)
        {
            return (val == n.val) ;
        } // equals
} //PositiveEven
```

Discussion

- *La prévention de défaillance* évite les problèmes reliés à l'évaluation de dommage et la récupération (suivant).
- Mais il entraîne des *surcharges* significatives (copies des variables d'état) et pour les systèmes où la performance est importante cela peut ne pas être applicable.
- La *détection de fautes rétrospectives* peut être une alternative plus adéquate dans quelques cas : *Évaluation de dommage et récupération*

Évaluation de dommage

- Analyser l'état du système, après un changement d'état, pour juger l'*'ampleur de la corruption.*
- Doit établir quelles parties de l'espace d'état ont été affectées par la défaillance.
- Généralement basée sur les « fonctions de validité » qui peuvent être appliquées aux éléments d'état pour établir si leur valeur est à l'intérieur d'un intervalle permis.
- Si un dommage est identifié, une *exception* est signalée et un mécanisme de *réparation* est utilisé pour récupérer l'état avant le dommage.

Implémentation Java

- Les objets à être vérifiés sont les instanciations d'une classe qui implémentent l'interface :

```
Interface CheckableObject {
   Public boolean check();
 }
```

- Chaque classe implémente sa propre méthode de vérification.
- Quand l'état comme un tout est vérifié, dynamic binding est utilisée pour assurer que la fonction de vérification appropriée soit exécutée.

Exemple : Évaluation du dommage (Java)

```
class RobustArray {
        // Checks that all the objects in an array of objects
        // conform to some defined constraint
        private boolean [] checkState ;
        private CheckableObject [] theRobustArray ;

        RobustArray (CheckableObject [] theArray)
        {
                checkState = new boolean [theArray.length] ;
                theRobustArray = theArray ;
        } //RobustArray
        public void assessDamage () throws ArrayDamagedException
        {
                boolean hasBeenDamaged = false ;

                for (int i= 0; i <this.theRobustArray.length ; i ++)
                {
                        if (! theRobustArray [i].check ())
                                {
                                        checkState [i] = true ;
                                        hasBeenDamaged = true ;
                                }
                        else
                                checkState [i] = false ;
                }
                if (hasBeenDamaged)
                        throw new ArrayDamagedException (checkState) ;
        } //assessDamage
} // RobustArray
```

Autres technique d'évaluation de dommage

- *Checksums* sont utilisées pour évaluer un dommage dans la transmission des données.
- *Des pointeurs redondants* peuvent être utilisés pour vérifier l'intégrité des structures de données.
- *Les minuteries du chien de garde (Watch dog timers)* peuvent vérifier pour les processus non-terminés dans des systèmes concurrent. S'il n'y a pas de réponse après un certain temps, on suppose qu'il y a un problème.

Récupération de faute

- Forward recovery
  - Appliquer les « réparations » à un état de système corrompu.
- Backward recovery
  - Restauration de l'état du système vers une état précédent connu sécuritaire.
- La récupération par saut avant (Forward recovery) est normalement spécifique pour chaque application.
  - La connaissance du domaine est requise pour calculer les corrections possibles d'état.
- La récupération par retour en arrière (retraitement) (Backward recovery)

est plus simple. Les détails d'un état sécuritaire sont maintenus et cela remplace l'état du système corrompu.

Récupération par saut avant (Forward Recovery)

- Corruption dans le codage de données :
  - Les techniques de codage d'erreurs, qui ajoute de la redondance à aux données codées, peuvent être utilisée pour réparer une donnée corrompu durant la transmission.
- Pointeurs redondants :
  - Quand des pointeurs redondants sont inclus dans les structures de données (e.g., listes bidirectionnelles (two-way lists)), une liste ou un fichier corrompu peut être reconstruit si un nombre suffisant de pointeurs ne sont pas corrompus.
  - Souvent utilisé pour réparer une base de données et un système de fichiers.
- Quelquefois, une simple approche est possible :
  - Réinitialiser un système, acquérir un nouveau contexte opérationnel (e.g., relire les senseurs), ramener à un état *sécuritaire.*

Récupération par retour en arrière (retraitement) (*Backward Recovery*)

- Les *transactions* utilisent fréquemment la méthode de récupération par **retour en arrière (retraitement)** . Les changements ne sont pas appliqués jusqu'à ce que le calcul soit complété. Si une erreur se produit, le système est remis dans l'état qui précède la transaction.
- E.g., système de base de données, les changements durant les transactions ne sont pas immédiatement incorporés dans la base de données (committed), la base de données fait une mise à jour après que la transaction soit complétée.
- Des *points de contrôles (checkpoints)* périodiques permet au système un « retour à l'arrière (roll-back) » à un état correct – restaure à un état correct à partir d'une *copie*.

Exemple : Procédure de tri sécuritaire

- L'opération de tri contrôle sa propre exécution et évalue si le tri a été correctement exécuté.
- Maintient une *copie* de ses données de telle manière que si une erreur se produit, les données ne seront pas corrompus.
- Basée sur l'identification et la manipulation des *exceptions.*
- Possible dans ce cas, car tri « valide » est connu. Toutefois dans plusieurs cas, il est difficile d'écrire des *contrôle de validité*.

Code de Récupération par retour en arrière (retraitement) Backward Recovery code (Java)

```
class SafeSort {
        static void sort ( int [] intarray, int order ) throws SortError
        {
                int [] copy = new int [intarray.length];

                // copy the input array

                for (int i = 0; i < intarray.length ; i++)
                        copy [i] = intarray [i] ;
                try {
                        Sort.bubblesort (intarray, intarray.length, order) ;
                        if (order == Sort.ascending)
                                for (int i = 0; i <= intarray.length-2 ; i++)
                                    if (intarray [i] > intarray [i+1])
                                                throw new SortError () ;
                        else
                                for (int i = 0; i <= intarray.length-2 ; i++)
                                    if (intarray [i+1] > intarray [i])
                                                throw new SortError () ;
                } // try block
                catch (SortError e )
                {
                        for (int i = 0; i < intarray.length ; i++)
                                intarray [i] = copy [i] ;
                        throw new SortError ("Array not sorted") ;
                } //catch
        } // sort
} // SafeSort
```

Conclusions

- Plusieurs techniques de programmation pour rendre le code plus fiable et plus robuste.
- Toutes ces techniques ont un coût, en termes d'effort de développement et de performance de système.
- Devrait être utilisé avec discrétion.
- Quelques problèmes techniques :
  - La récupération par retour en arrière (retraitement) est difficile à implémenter dans des systèmes concurrents, distribués, incompatible avec les systèmes ayant des délais temps réel