Informatique embarquée

Qualité et fiabilité des logiciels embarqués

Philippe.Plasson@obspm.fr

v1.1

Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

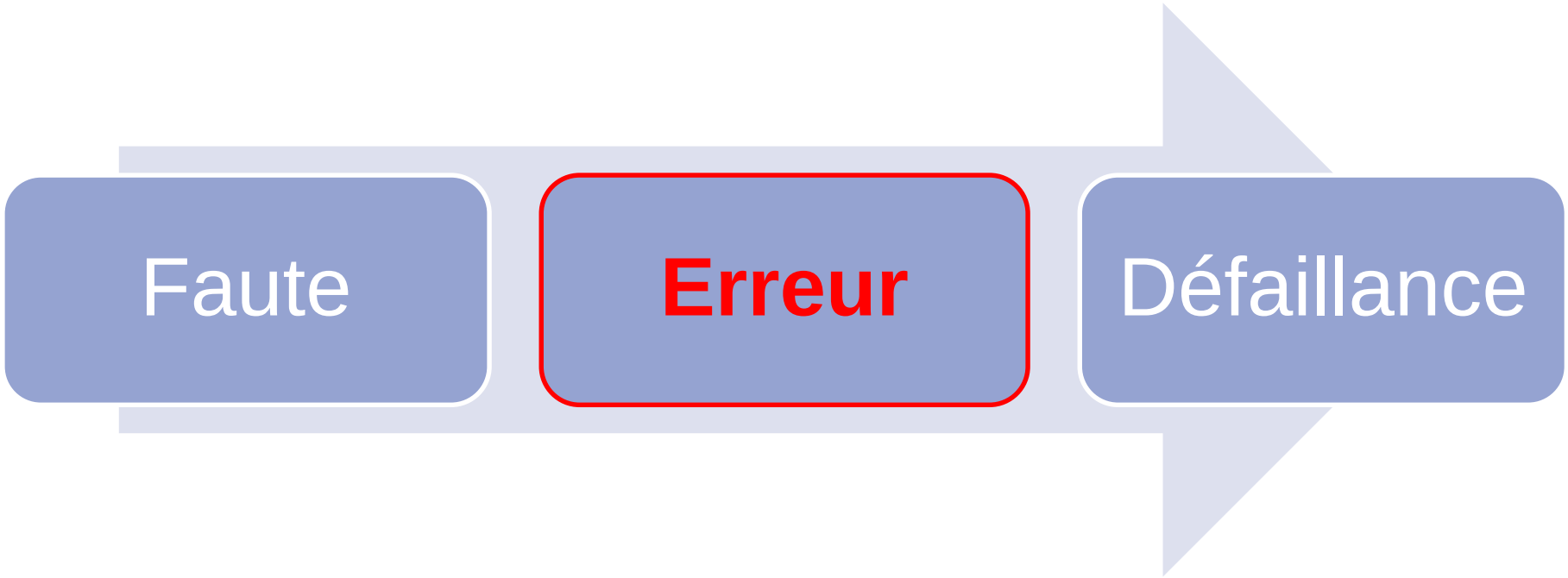
Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

De la faute à la défaillance 1/3

- Une erreur (négligence, approximation, oubli) humaine faite durant la spécification des exigences, durant la conception ou durant le codage peut entraîner la présence (latente) d'une **faute** (défaut) dans un élément du logiciel embarqué.

De la faute à la défaillance 2/3



- Ce défaut caché, dans des circonstances particulières, peut se manifester sous la forme d'une **erreur** = écart entre une valeur ou une action attendue et celle obtenue.

De la faute à la défaillance 3/3

- Une erreur peut mener à une **défaillance**, c'est-à-dire à un comportement inattendu ou non voulu du système.
- Les conditions qui conduisent les fautes à devenir des défaillances sont parfois extrêmement difficiles à prévoir.
- Souvent, les défaillances logicielles semblent apparaître de façon aléatoire.
- **Elles peuvent parfois conduire à des conséquences catastrophiques (crash d'un avion, explosion d'un lanceur, accident d'un véhicule autonome, ...).**

Fiabilité des logiciels embarqués

- La fiabilité d'un logiciel est la propriété d'un logiciel à être exempt de fautes pouvant entraîner son indisponibilité partielle ou complète.
- Question clé pour les systèmes embarqués :
  - La fiabilité et la sûreté de fonctionnement sont des questions d'une importance primordiale dans le développement et l'exploitation des systèmes embarqués.
  - La contribution des logiciels à la fiabilité des systèmes et à leur sûreté de fonctionnement est un facteur clé, surtout compte tenu de la complexité croissante des logiciels utilisés dans les systèmes embarqués critiques (spatial, aéronautique, ferroviaire, automobile, …), de l'augmentation des coûts et des contraintes de planning.

Fiabilité des logiciels embarqués

- Les défaillances causées par le logiciel peuvent entraîner une dégradation critique des performances du système jusqu'à une perte de complète de la mission.
- Les activités visant à accroître la fiabilité des logiciels sont menées tout au long du cycle de vie du logiciel.
- Le rôle de l'assurance qualité logicielle est de définir ces activités et de s'assurer, tout au long du projet, qu'elles sont correctement réalisées.

Maintenabilité

- La maintenabilité d’un logiciel est sa capacité à être conservé ou restauré dans un état dans lequel il peut accomplir une fonction requise, lorsque la maintenance est effectuée.
- En d'autres termes, la maintenabilité du logiciel a trait à la facilité avec laquelle le logiciel peut être modifié et remis en service.
- La maintenabilité est une question cruciale pour tous les logiciels embarqués et encore plus pour ceux ayant un rôle critique.

Maintenabilité

- La maintenabilité des logiciels embarqués doit être envisagée au tout début de la conception des logiciels.
- Les architectures généralement adoptées (séparation entre un logiciel de boot minimaliste et un logiciel applicatif) reflètent cette problématique (voir chapitre sur les architectures).

Disponibilité

- La disponibilité d’un logiciel est sa capacité à être dans un état lui permettant d'accomplir une fonction requise à un instant donné ou sur un intervalle de temps donné.
- La disponibilité d’un logiciel est fonction de sa fiabilité et de sa maintenabilité:
  - Des défaillances logicielles fréquentes et des périodes de maintenance longues réduisent la probabilité qu’un logiciel soit disponible à un instant donné ou sur un intervalle de temps donné.

Le concept de « dependability »

- Terminologie anglaise :
  - Fiabilité = Reliability
  - Maintenabilité = Maintainability
  - Disponibilité = Availability
- En anglais, il existe un terme couramment utilisé qui englobe les concepts de fiabilité, maintenabilité et disponibilité : le terme de « dependability »

Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

Méthodes et techniques pour l'évaluation de la fiabilité des logiciels embarqués

- Il existe un certain nombre de méthodes et techniques pour l'évaluation de la fiabilité et de la sûreté de fonctionnement :
  - SFMECA = Software failure modes, effects and criticality analysis
    - En français, AMDEC logicielle
  - SFTA = Software Fault Tree Analysis
  - HSIA = Hardware-Software Interaction Analysis
    - Est-ce que le logiciel réagit de façon acceptable à des pannes ou erreurs matérielles ?
  - SCCFA = Software Common Cause Failure Analysis
  - FDIR = Failure Detection Isolation and Recovery

Méthodes et techniques pour l'évaluation de la fiabilité
SFMECA – AMDEC logicielle

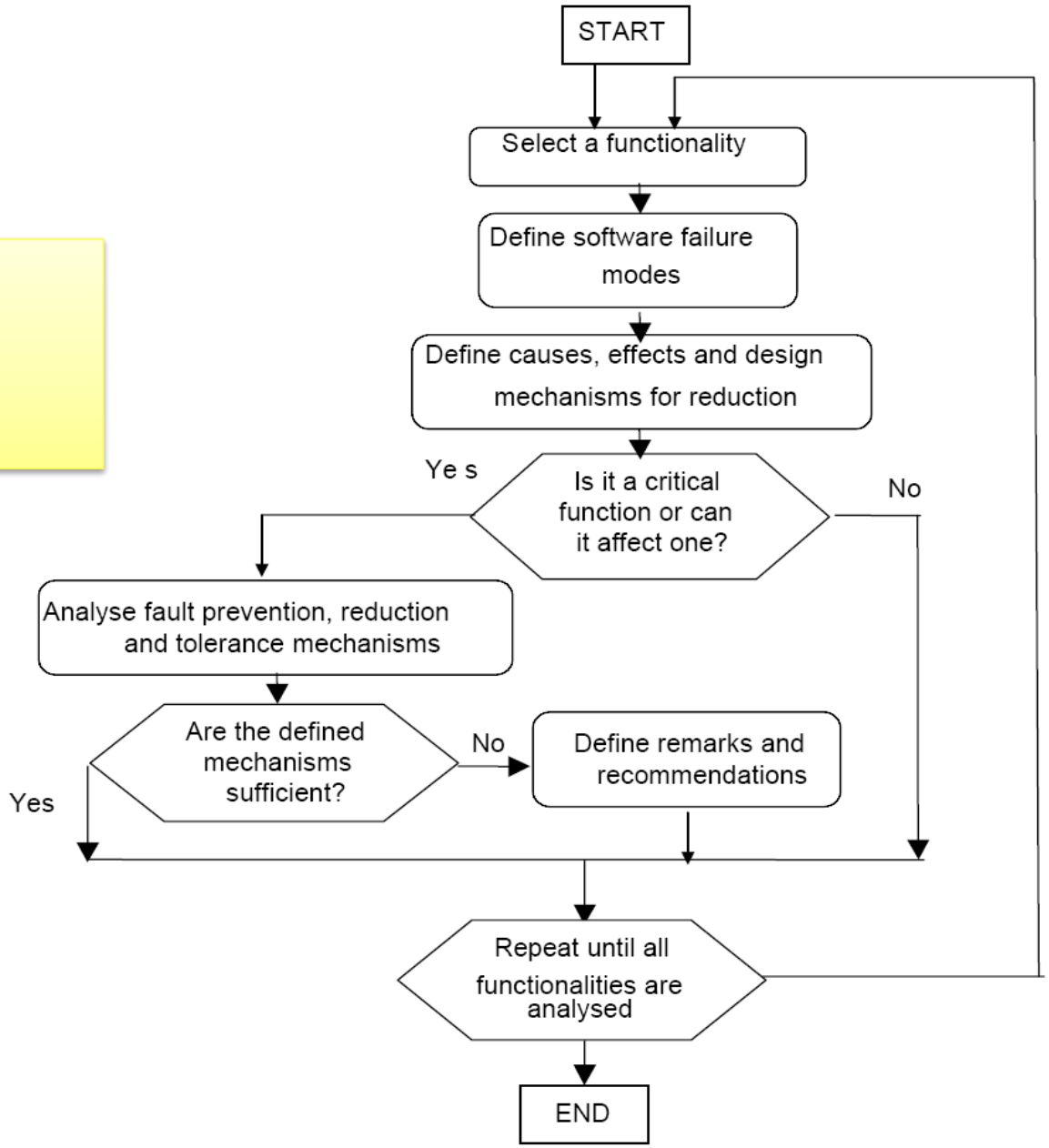
- L'objectif principale d'une analyse SFMECA est d'identifier les défaillances potentielles causées par un logiciel :
  - Analyse systématique et documentée des différentes possibilités qu'un composant logiciel a de défaillir (modes de défaillance), des causes de chaque mode de défaillance et des effets de chaque défaillance.
- L'analyse SFMECA va aussi permettre d'attribuer aux logiciels ou composants logiciels analysés un niveau de criticité basé sur la sévérité des conséquences engendrées par les modes de défaillance potentiels.
- L'analyse SFMECA est menée à partir d'un description structurelle du système ou du composant (le système doit être décomposé en blocs).

Méthodes et techniques pour l'évaluation de la fiabilité
SFMECA – AMDEC logicielle

- Après avoir déterminé la portée de l'analyse (quels composants et quel niveau de détail approprié), l'analyste identifie les modes de défaillance de chaque bloc.
- Après avoir déterminé les modes de défaillance, l'analyste détermine les effets de chacun des modes de défaillance sur le composant lui-même (effet local) et sur le système opérant dans son environnement (effet global), ainsi que les moyens d'y remédier (en anglais, « mitigation »).

Méthodes et techniques pour l'évaluation de la fiabilité

Procédure SFMECA (AMDEC logicielle)



Méthodes et techniques pour l'évaluation de la fiabilité
SFMECA – AMDEC logicielle

- Les résultats d'une analyse SFMECA sont consignés dans un tableau :
  - Les colonnes de gauche décrivent les modes de défaillance et les critères pour les identifier.
  - Les colonnes de droite permettent de tracer les effets et classer leur criticité.
- Certaines SFMECA sont quantitatives et peuvent définir des probabilités attachées à l'occurrence des modes de défaillance et de leurs effets.
- La colonne la plus à droite peut contenir des recommandations sur comment éviter / prévenir / récupérer le mode de défaillance.

Méthodes et techniques pour l'évaluation de la fiabilité
SFMECA – AMDEC logicielle

- Voir exemple page suivante (analyse d'un logiciel pilotant le déploiement d'antennes), inspiré du document "Software Fault Analysis Handbook (Software Fault Tree Analysis (SFTA) & Software Failure Modes, Effects and Criticality Analysis (SFMECA))", JPL D-28444, Rev. # 0

Failure	Function	Failure Mode	Effect	Criticality	Prob.	Impact	Action	Mitigation
Erroneous command	Command	Antenna does not move when commanded	Mission degraded severely	Complete loss of mission (6)	50%	5	Software checks all incoming commands	Modify the software module to validate commands
Corrupted command	Command	Antenna not behaving correctly	Mission degraded	Significant loss (4)	30%	4	Run checksum and reject corrupted commands	Add a checksum routine and check incoming commands for corruptions
Commanded while in an inappropriate mode	Command	Antenna does not behave correctly	Minor problem	Minor (1)	70%	2	Check the operating mode and if invalid ignore the command	Only execute the commands in valid operating modes
S/W module not responding	Command/ Telemetry	Antenna position unknown	S/W may damage the hardware	Subsystem loss (3)	10%	4	Awaken or restart the module	Provide a soft reset capability or a watchdog timer
Loss of Communica-tion	Command/ Telemetry	Unknown	Mission degraded severely	Significant degradation (2)	10%	3	Unless a hard failure restart the bus	Provide a reset capability
Incorrect command sequence	Command	Antenna does not move	Minor problem	Minor (1)	50%	1	Train operators and check the commands	Establish flight rules and command constraints

Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

Qualité du code logiciel
Introduction

- Tous les standards modernes de développement logiciels (RTCA/DO-178B, IEC 61508, MIL-STD 498, MISRA, ECSS, CMMI, ISO/IEC 12207) sont unanimes sur un point :
  - le rôle prépondérant de la qualité du code logiciel,
  - et en particulier le rôle des processus de vérification et de revue du code logiciel.
- C'est particulièrement vrai dans le domaine des logiciels embarqués critiques = préoccupation centrale dans les activités de conception et d'implémentation

Qualité du code logiciel
Objectifs

- Améliorer la stabilité des produits développés
- Accroître la réactivité des développements
- Réduire les coûts de développement, de test et de maintenance
- Créer des composants partageables et réutilisables en se focalisant sur les composants critiques demandant beaucoup d'effort pour la validation et la maintenance
- Valider les développements externalisés
- Préserver la réputation de l'organisation qui produit le logiciel (entreprise, laboratoire de recherche, etc.)

Qualité du code logiciel
Caractéristiques recherchées

Fiabilité

Portabilité

Maintenabilité

Testabilité

Réutilisation

Extensibilité

Lisibilité

Qualité du code logiciel
Caractéristiques recherchées

Fiabilité

Portabilité

Maintenabilité

Testabilité

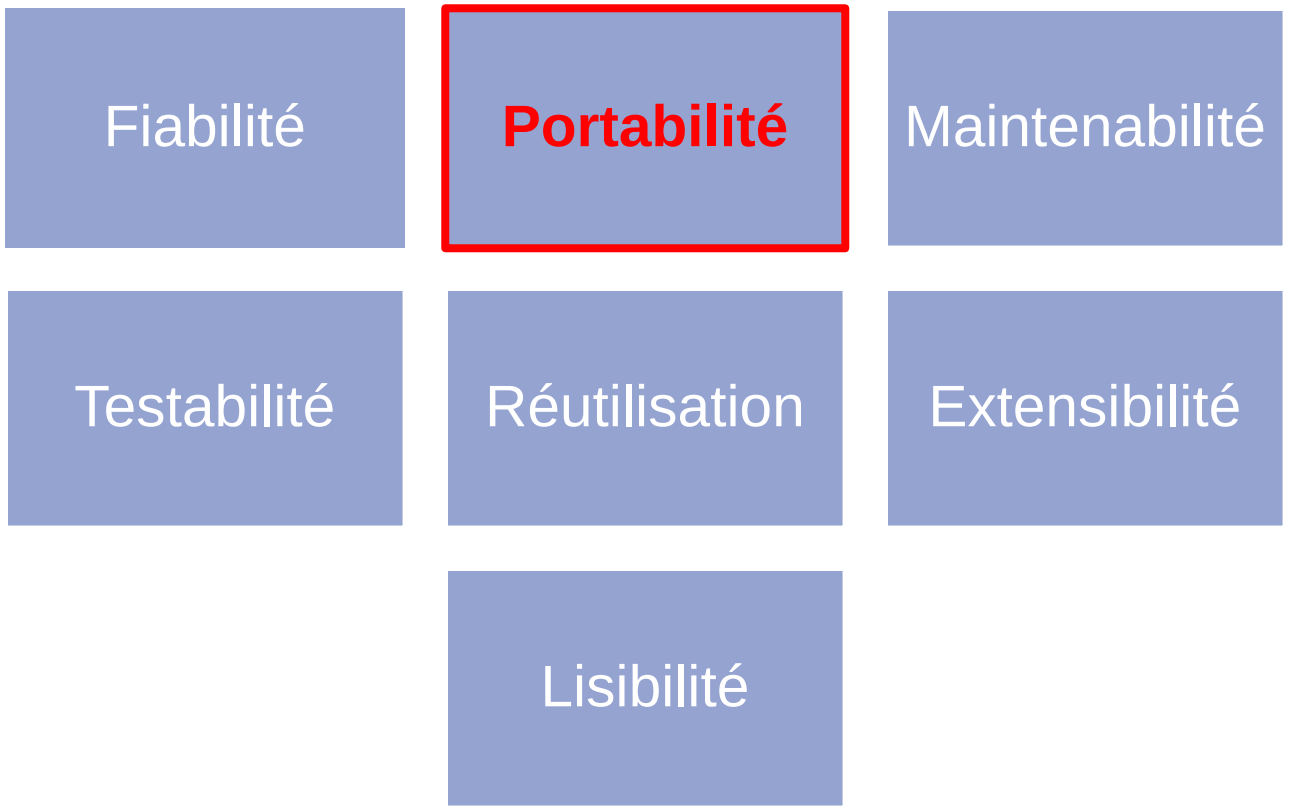
Réutilisation

Extensibilité

Lisibilité

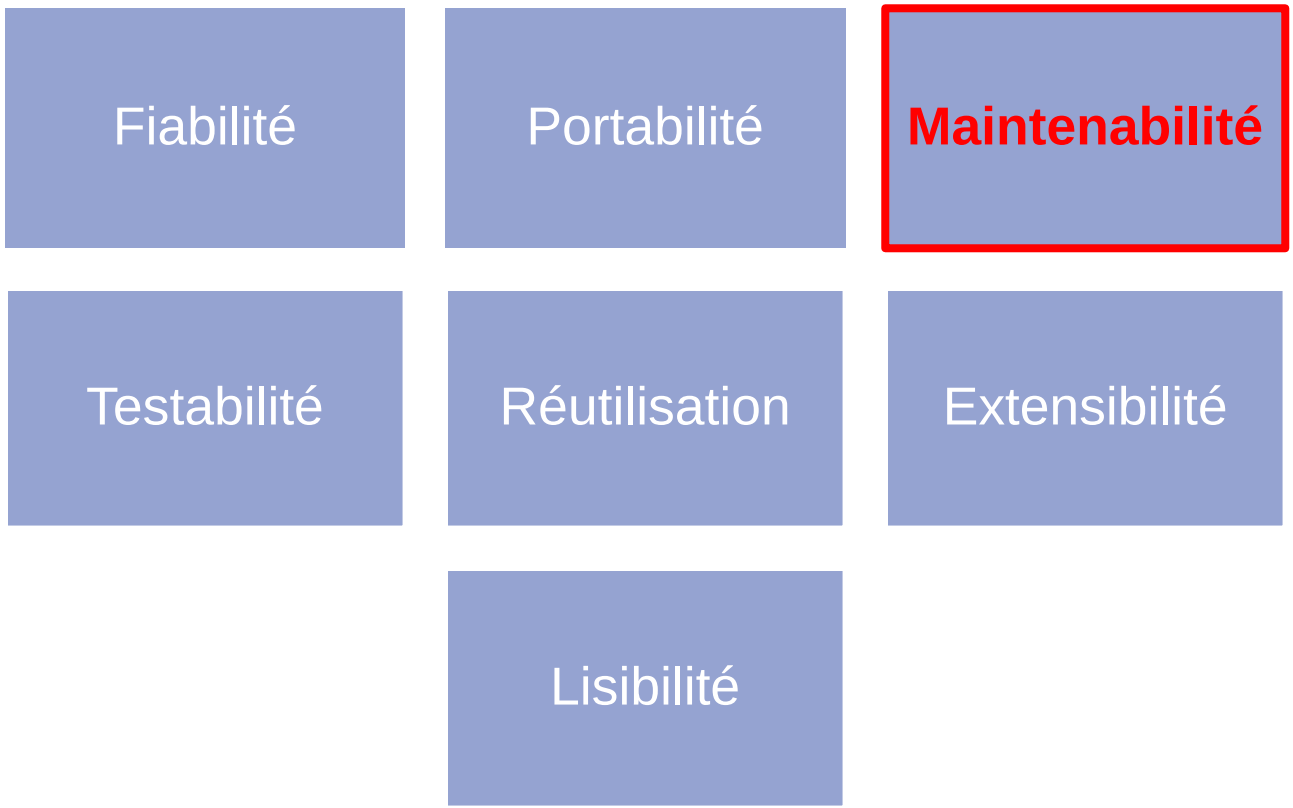
Le code exécutable doit toujours remplir toutes les exigences requises d'une manière prévisible.

Qualité du code logiciel
Caractéristiques recherchées



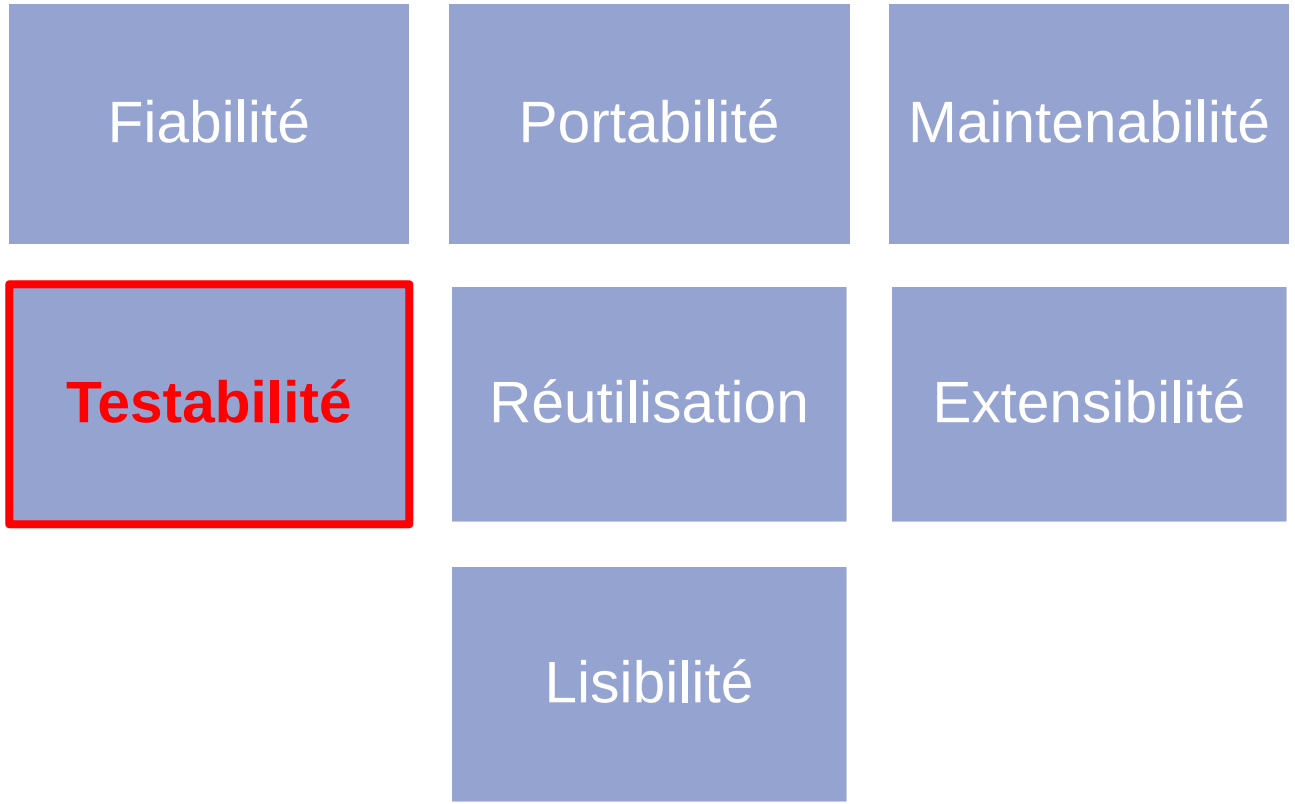
Le code source doit être portable, c'est-à-dire indépendant du compilateur ou de l'éditeur de liens.

Qualité du code logiciel
Caractéristiques recherchées



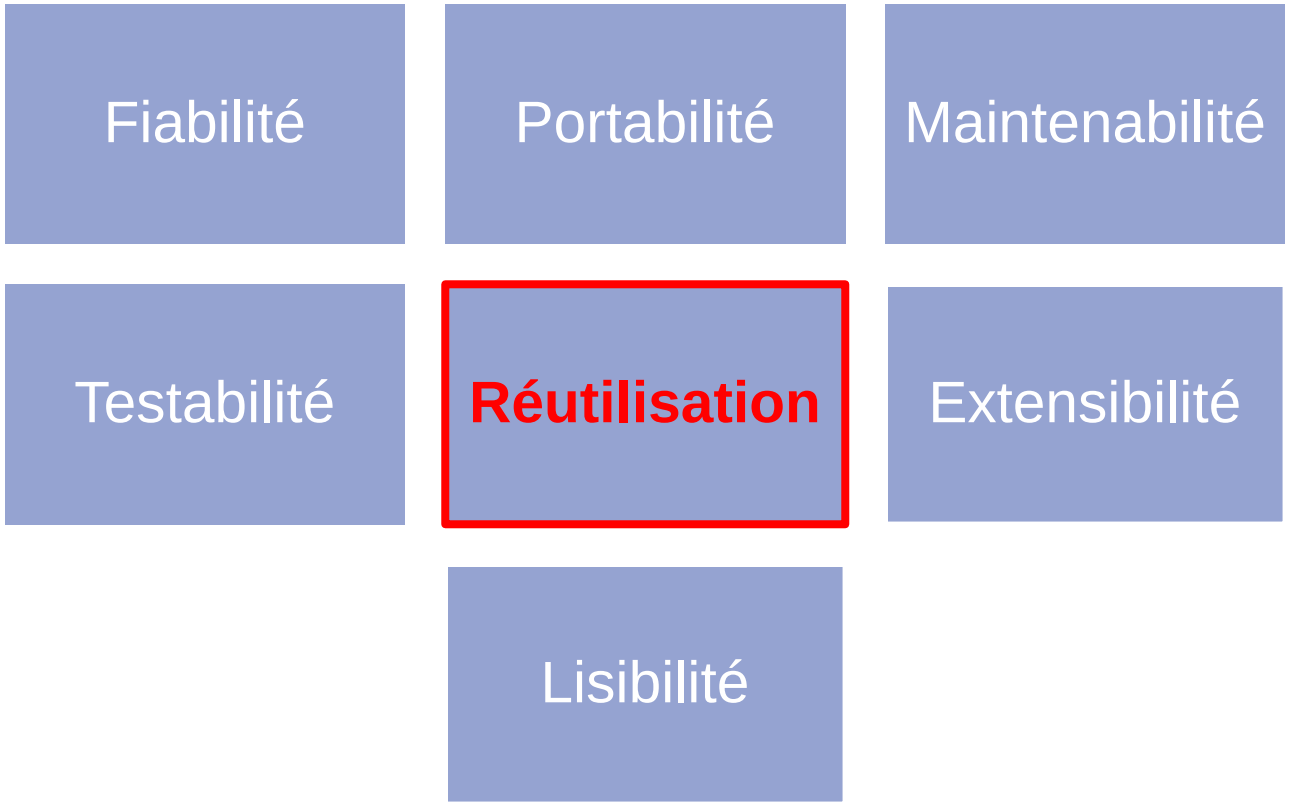
Le code source doit être rédigé d'une manière consistante, lisible, simple dans sa conception, facile à analyser et à déboguer.

Qualité du code logiciel
Caractéristiques recherchées



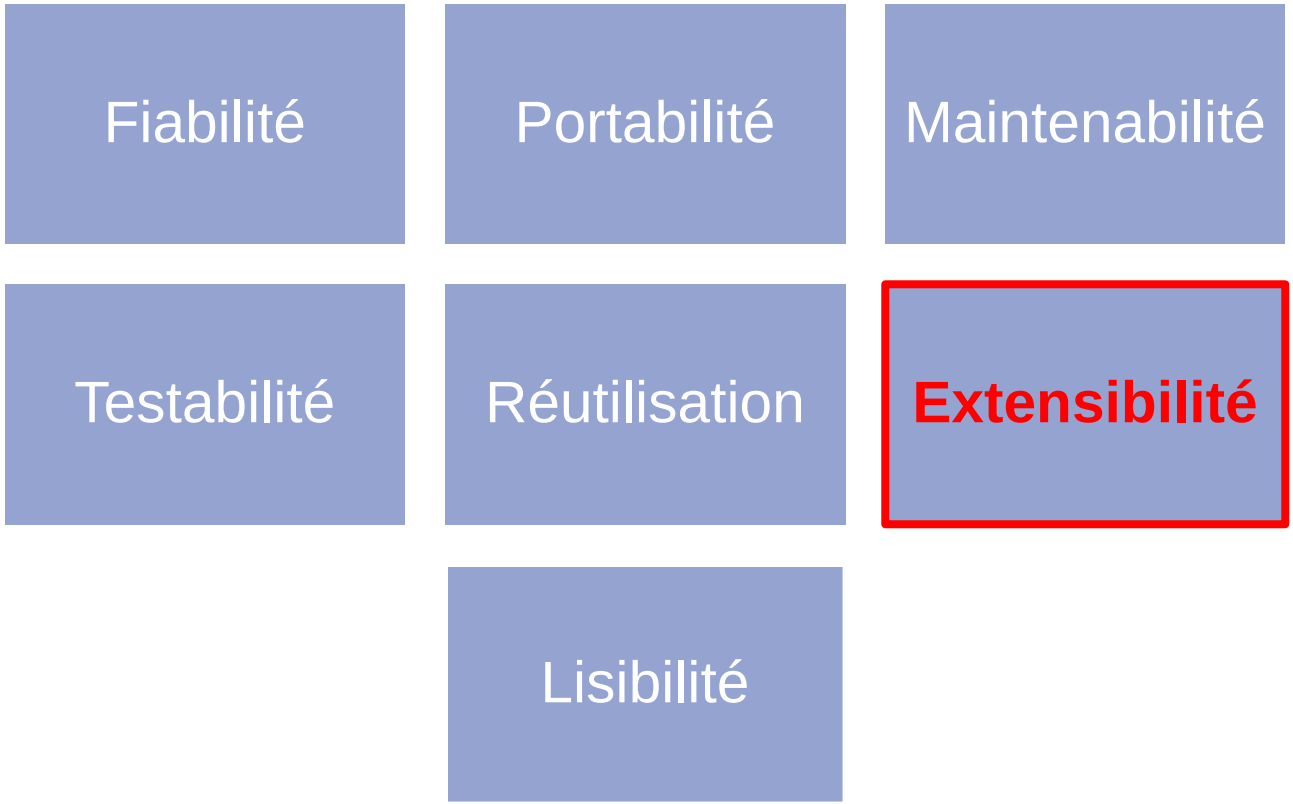
Le code source doit être écrit afin de faciliter la testabilité.
Réduire au minimum les caractéristiques suivantes pour chaque module logiciel le rendra plus testable et maintenable : taille du code, complexité, compte de trajets statiques (nombre de chemins à travers un morceau de code)

Qualité du code logiciel
Caractéristiques recherchées



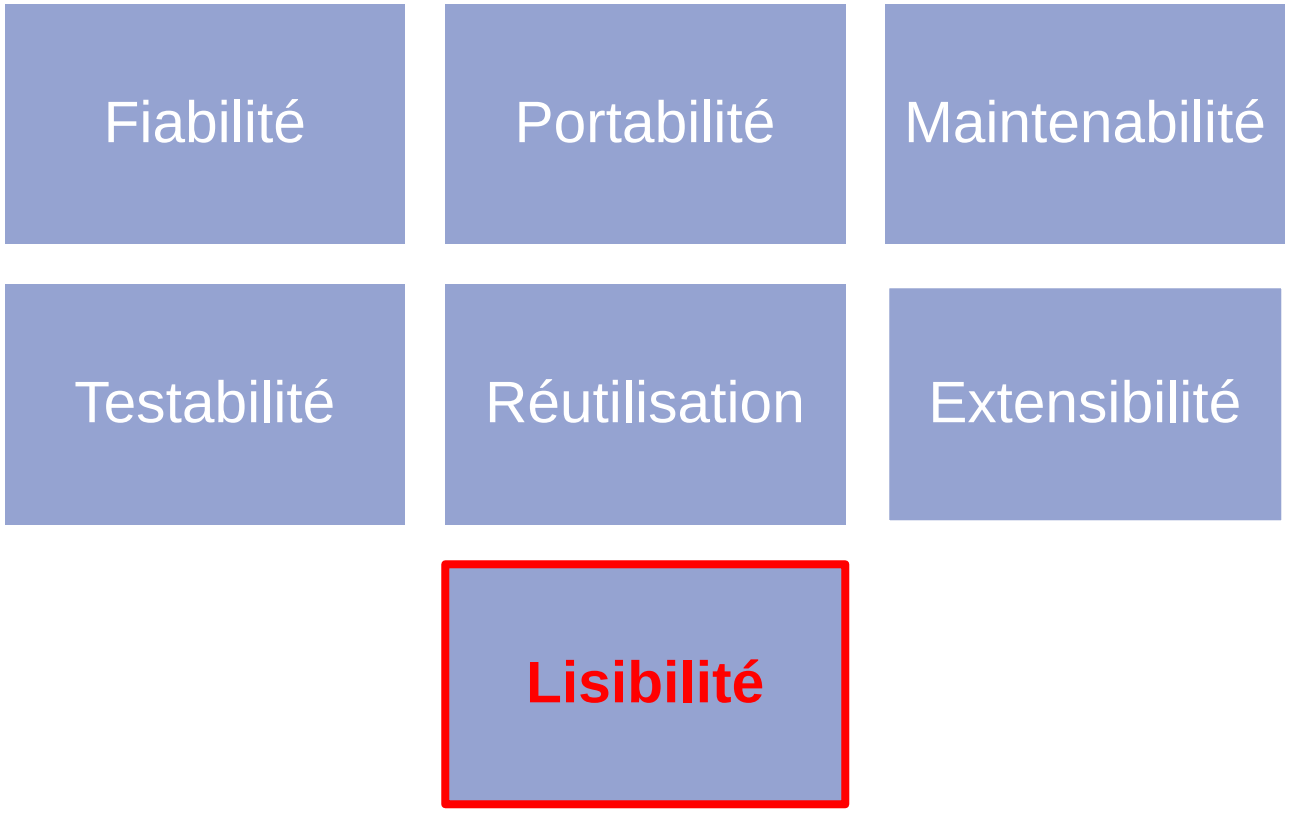
La conception de composants réutilisables est encouragée.
La réutilisation de composants permet d'éliminer les développements redondants et les activités de test (c'est-à-dire réduire les coûts).

Qualité du code logiciel
Caractéristiques recherchées



Les exigences sont appelées à évoluer au cours de la vie d'un produit.
C'est pourquoi, un système devrait toujours être développé d'une manière extensible : les modifications dans les exigences doivent être gérées grâce à des extensions locales plutôt que par des modifications impactant l'ensemble du logiciel.

Qualité du code logiciel
Caractéristiques recherchées



Le code source doit être rédigé d'une manière qui est facile à lire, à comprendre et à appréhender.

Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

Métriques logicielles

- Les métriques sont des indicateurs de la qualité du code pouvant être mesurés via des outils d'analyse statique du code (Polyspace, Logiscope quality checker, SonarQube, …)
- Dans un projet de logiciel embarqué, il faut :
  - Définir quelles métriques doivent être produites.
  - Définir pour chaque métrique des objectifs à atteindre.
  - Les objectifs dépendent de la criticité du logiciel.
  - Ils sont souvent définis dans des standards (ECSS pour le spatial par exemple).

Métriques logicielles

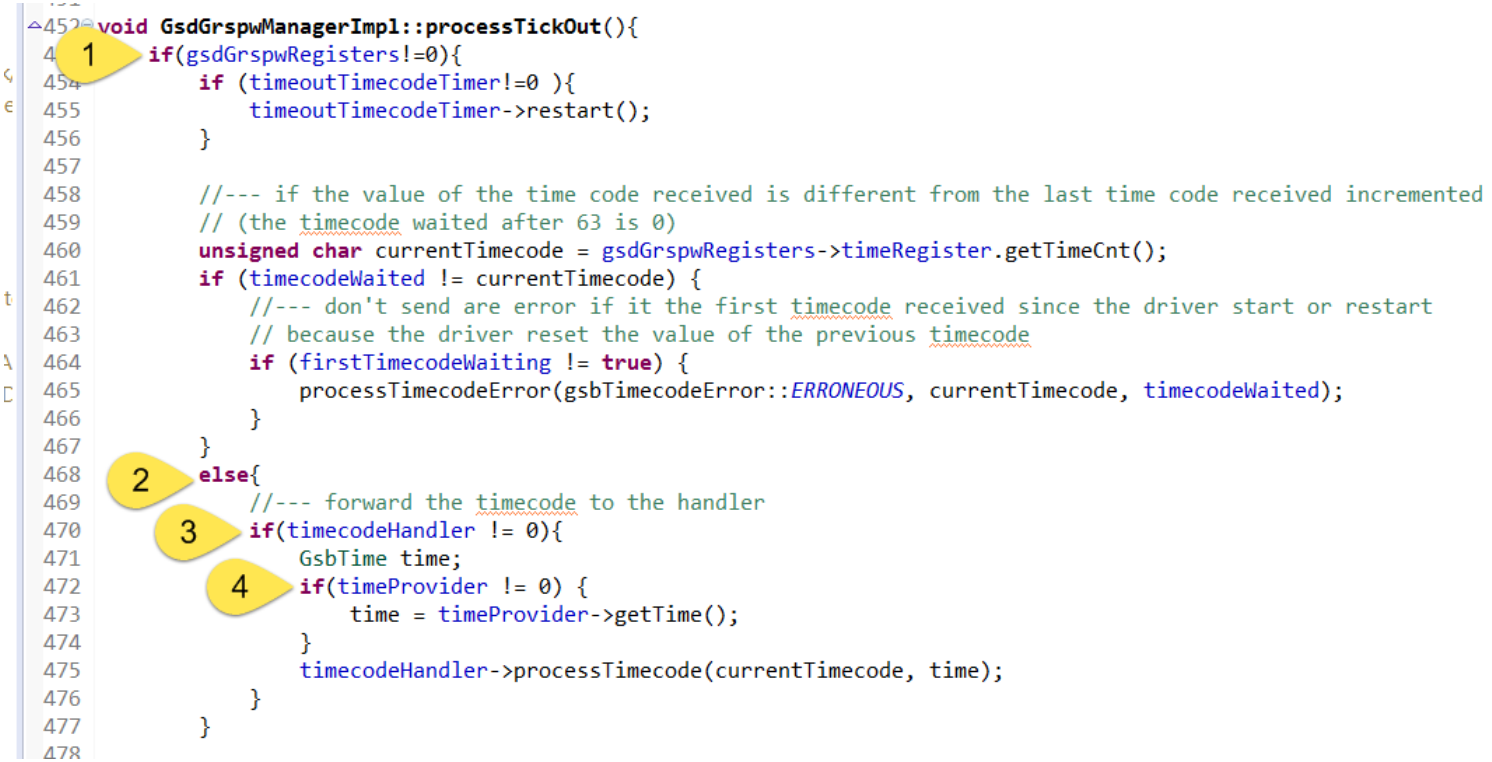
- Les développeurs doivent avant chaque « commit » du code s’assurer que les objectifs définis pour chaque métriques sont bien atteints.
- Les rapports de métriques sont fournis dans le « dossier de justification » du logiciel.

Principales métriques

- Nombre de lignes de code effectives par fonction (LOC)
  - Les lignes vides, les commentaires, les lignes contenant seulement des { ou } ne sont pas comptées
  - Objectif : ≤50 à ≤100
- Fréquence des commentaires
  - Rapport entre le nombre de lignes de commentaires (en excluant les headers) dans un fonctions et le nombre de lignes de code effectives
  - Objectif : ≥0.2 à ≥0.3

Principales métriques

- Niveau d'imbrication (nesting level)
  - Nombre maximum de niveaux de structures de contrôle imbriquées dans une fonction
  - Indicateur de la complexité du code
  - Objectif : ≤5 à ≤7




```
452 void GsdGrspwManagerImpl::processTickOut(){
453     if(gsdGrspwRegisters!=0){
454         if (timeoutTimecodeTimer!=0 ){
455             timeoutTimecodeTimer->restart();
456         }
457
458         //--- if the value of the time code received is different from the last time code received incremented
459         // (the timecode waited after 63 is 0)
460         unsigned char currentTimecode = gsdGrspwRegisters->timeRegister.getTimeCnt();
461         if (timecodeWaited != currentTimecode) {
462             //--- don't send are error if it the first timecode received since the driver start or restart
463             // because the driver reset the value of the previous timecode
464             if (firstTimecodeWaiting != true) {
465                 processTimecodeError(gsbTimecodeError::ERRONEOUS, currentTimecode, timecodeWaited);
466             }
467         }
468         else{
469             //--- forward the timecode to the handler
470             if(timecodeHandler != 0){
471                 GsbTime time;
472                 if(timeProvider != 0) {
473                     time = timeProvider->getTime();
474                 }
475                 timecodeHandler->processTimecode(currentTimecode, time);
476             }
477         }
478
```

Principales métriques

- Complexité cyclomatique
  - Mesure de la quantité de logique décisionnelle d'un module logiciel
  - Indicateur de la complexité du code
  - Objectif : ≤10 à ≤20

Principales métriques

- Couverture de code
  - Taux de couverture des lignes contenues dans le code source (statement coverage)
    - Objectif : 100%
  - Taux de couverture des branches contenues dans le code source (decision coverage, condition coverage)
    - Objectif : non mesuré à 100% selon la criticité
  - Taux de couverture des chemins d'exécution (Modified Condition/Decision Coverage - MC/D)
    - Objectif : non mesuré à 100% selon la criticité

Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

Mesure de la complexité du code
Introduction

- Le pourcentage d'erreurs et la robustesse du code dépendent de sa complexité.
- Un code complexe est difficile à tester ➔ plus d´erreurs dans l´application finale.
- Un code complexe entraîne une maintenance difficile et coûteuse.
- D'où l'importance d'avoir des métriques permettant d'évaluer de façon objective la complexité du code.

Mesure de la complexité du code
Complexité cyclomatique

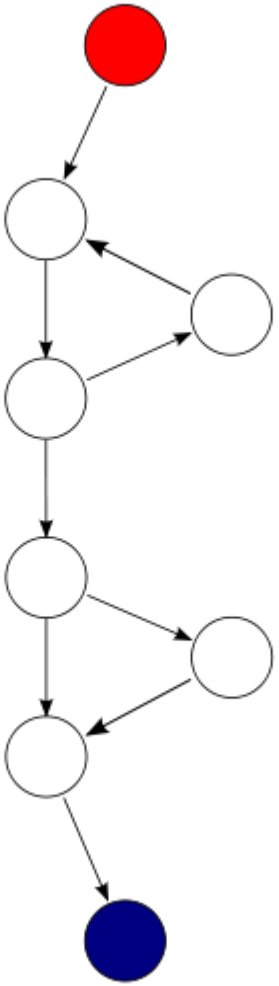
- La complexité cyclomatique mesure la quantité de logique décisionnelle d'un module logiciel (une application, une fonction, une méthode ou une classe), en d'autres termes, le nombre de chemins possibles d'exécution.
  - McCabe's Cyclomatic Complexity(MVG)

Mesure de la complexité du code
Complexité cyclomatique

- La complexité cyclomatique est une métrique couramment utilisée afin d'évaluer la complexité d'un module donné et afin d'aider à la mise au point des tests.
  - La complexité cyclomatique peut être utilisée dans toutes les phases du cycle de vie du logiciel embarqué, à commencer par la conception, afin d'améliorer la fiabilité des logiciels embarqués, leur testabilité et leur maintenabilité.
  - La complexité cyclomatique facilite le processus de conception des tests unitaires en permettant d'évaluer le nombre de tests requis pour un module donné (stratégie de test « Basic Path Testing »)
    - Tester chaque chemin linéairement indépendant à travers le programme.
    - Le nombre de cas de test sera égal à la complexité cyclomatique du programme.

Mesure de la complexité du code
Complexité cyclomatique

- La complexité cyclomatique est une métrique structurelle entièrement basée sur le flux de contrôle à travers une portion de code.
- La complexité cyclomatique comptabilise le nombre de « chemins » linéairement indépendants au travers d'un programme représenté sous la forme d'un graphe.
- Le complexité cyclomatique (v(G)) est définie pour chaque module par la formule suivante :
  - v(G) = e - n + 2 où
  - ‘n’ représente les noeuds (‘nodes’) du graphe, c’est-à-dire des instructions
  - ‘e’ représente les arêtes (‘edges’) du graphe, c’est-à-dire le transfert du contrôle entre des noeuds.



e = 9 ; n = 8 ➔ v(G) = 3

Mesure de la complexité du code
Complexité cyclomatique

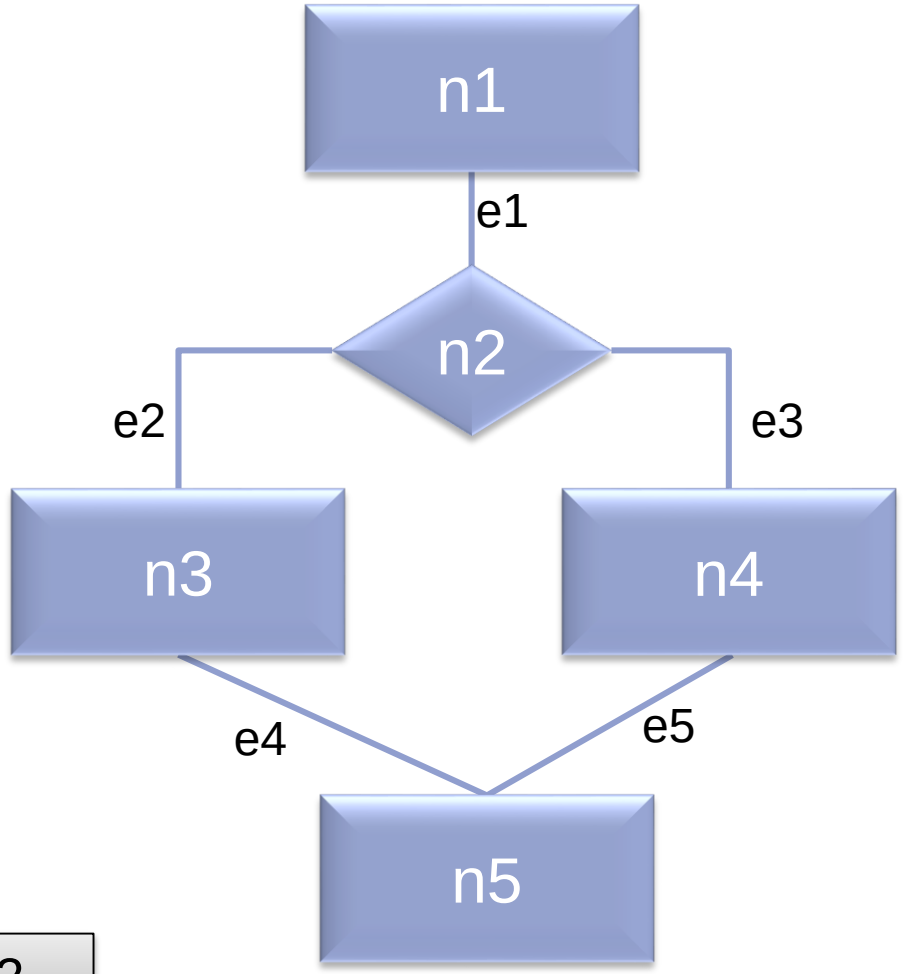
- On peut calculer approximativement la complexité cyclomatique d'une portion de code en comptant les mots-clés du langage et les opérateurs qui introduisent des décisions dans le flot d'exécution.
  - Les opérations booléennes ajoutent des chemins supplémentaires dans le code parce que le second opérande peut ou ne peut pas être évalué en fonction de la valeur du premier opérande.
- Cette approche pragmatique est assez précise dans la plupart des cas.
- Dans le cas du C/C++, il faut compter dans la portion de code que l'on analyse les éléments syntaxiques suivants :
  - 'if','while','for','switch','break','&&','||'

Mesure de la complexité du code
Complexité cyclomatique

```
1       int x = 3;

2       if (x>0)
        {
3           x++;
        }
        else
        {
4           x--;
        }

5       x = x * 2;
```



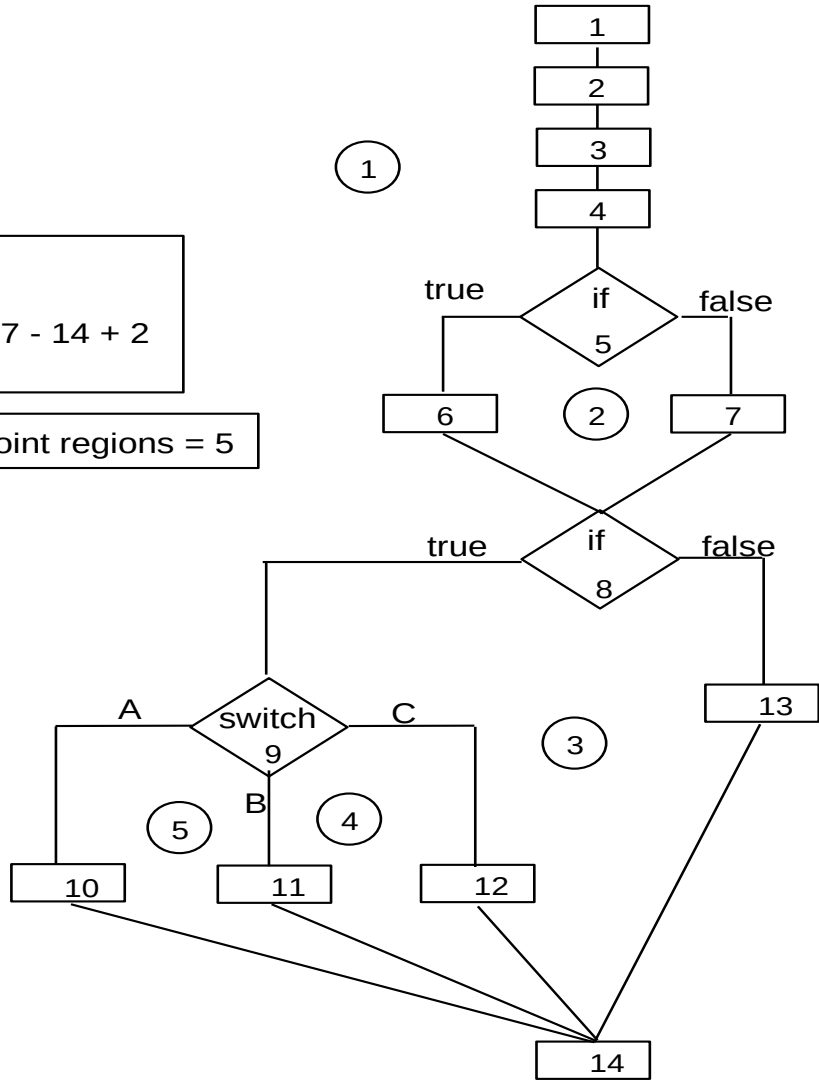
Complexité = e – n + 2 = 5 – 5 + 2 = 2

```
void compute_pay_check ( employee_ptr_type employee_ptr_IP, check_ptr_type chk_ptr_OP )
{
        //Calculate the employee's federal, fica and state tax withholdings
   1.   chk_ptr_OP->gross_pay  = employee_ptr_IP->base_pay;
   2.   chk_ptr_OP->ged_tax     = federal_tax ( employee_ptr_IP->base_pay );
   3.   chk_ptr_OP->fica         = fica ( employee_ptr_IP->base_pay );
   4.   chk_ptr_OP->state_tax   = state_tax ( employee_ptr_IP->base_pay );

        //Determine medical expense based on the employee's HMO selection
   5.   if ( employee_ptr_IP->participate_HMO == true )
        {
   6.     chk_ptr_OP->medical = med_expense_HMO;
        }
        else
        {
   7.     chk_ptr_OP->medical = med_expense_non_HMO;
        }
        // Calc a profit share deduction based on % of employee's gross pay
   8.   if (employee_ptr_IP->participate_profit_share == true )
        {
   9.     switch( employee_ptr_IP->profit_share_plan )
          {
            case plan_a:
  10.         chk_ptr_OP->profit_share = two_percent * chk_ptr_OP->gross_pay;
              break;
            case plan_b:
  11.         chk_ptr_OP->profit_share = four_percent * chk_ptr_OP->gross_pay;
              break;
            case plan_c:
  12.         chk_ptr_OP->profit_share = six_percent * chk_ptr_OP->gross_pay;
              break;
            default:
              break;
          }
        }
        else
        {
  13.     chk_ptr_OP->profit_share = zero;
        }
  14.   chk_ptr_OP->net_pay = (chk_ptr_OP->gross_pay –  chk_ptr_OP->fed_tax –  chk_ptr_OP->fica –  chk_ptr_OP->state_tax –  chk_ptr_OP->medical – chk_ptr_OP->profit_share );
}
```

Nodes = 14
Edges = 17
Complexity = 17 - 14 + 2
= 5

Number of disjoint regions = 5



Mesure de la complexité du code
Complexité cyclomatique

- Relation entre complexité cyclomatique et évaluation des risques, d'après le S.E.I. (Software Engineering Institute, http://www.sei.cmu.edu/), institut à l'origine de la norme CMMI

Valeur de la complexité cylomatique	Evaluation des risques
1-10	Programme simple, sans trop de risque
11-20	Complexité et risque modéré
21-50	Complexe, risque élevé
Supérieure à 50	Non testable, risque très élevé

Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

Qualité du code logiciel
Règles et standards de codage

- Un code qui « fonctionne » et qui a passé tous les tests n'est pas forcément un code pleinement acceptable.
- Le code doit être écrit selon des règles de codage clairement spécifiées.
- Plusieurs standards définissent des règles de codage.
- Exemples :
  - MISRA C, MISRA C++ (automobile / avionique / spatial)
  - JSF (avionique / spatial)
- Au sein de tout projet logiciel, on doit définir et documenter les règles de codage applicables.

Qualité du code logiciel
Règles et standards de codage

- Il faut écrire le code en s'appuyant sur des règles de codage clairement documentées.
- Objectifs :
  - Prévenir l'utilisation de construction dangereuses.
  - Renforcer la consistance du style de programmation entre les différents développeurs.

Qualité du code logiciel
Règles et standards de codage

- Il faut vérifier que le logiciel est écrit en respectant les règles de codage applicables au projet.
- La vérification des règles de codage peut et doit être largement automatisée grâce à des outils du type Logiscope ou Polyspace.
- Ces outils implémentent des règles générales.
- Ils peuvent être personnalisés et adaptés à un standard donné : personnalisation de règles existantes, ajout de nouvelles règles.
- Ils peuvent s'intégrer aux environnements de développement pour un usage transparent.

Qualité du code logiciel
Règles et standards de codage

- Exemple d'outils logiciels pour l'évaluation de la qualité du code :
  - Vérification des règles de codage automatisée :
    - Logiscope RuleChecker, Polyspace, etc.
  - Evaluation de la qualité du code
    - Logiscope QualityChecker, Polyspace, etc.
  - Analyse de la couverture des tests
    - Logiscope TestChecker, etc.
  - Amélioration du code (suppression du code dupliqué, etc.)
    - Logiscope Code Reducer

Qualité du code logiciel
Règles et standards de codage

- Quel est le comportement de ce code ?


```
if (type_error) {
    switch (val_buf->type) {
    case TYPE_CHAR:
        strcpy (str_buf, "char");
    case TYPE_REAL:
        strcpy (str_buf, "real");
    case TYPE_LONG:
        strcpy (str_buf, "long");
    default:
        strcpy (str_buf, "unknown");
    }
    sprintf(buf, "Wrong arg type %sfor attr %s",     str_buf, attr);
    warning_dispatch(func, buf);
    return false;
}
```

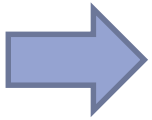
Règle : Chaque case dans une structure switch doit être terminée par un break.

Qualité du code logiciel
Règles et standards de codage

- Quel est le comportement de ce code ?


```
for (i=0; i<(int)num_pairs; i++) {
        if (SIG[i] == 0.0) /* if ANY is 0 then set ALL to 1 */
                for(j=0;j<(int)num_pairs;j++) SIG[j]=1.0;
                break;
}
```

Règle : Chaque bloc logique doit être limité par des accolades, même si le bloc ne contient qu'une instruction ou aucune instruction



```
for (i=0; i<(int)num_pairs; i++)
{
        /* if ANY is 0 then set ALL to 1 */
        if (SIG[i] == 0.0)
        {
                for(j=0;j<(int)num_pairs;j++)
                {
                        SIG[j]=1.0;
                }
                break;
        }
}
```

Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

Règles et standards de codage
Le standard JSF AV C++

- Standard JSF AV C++ = Joint Strike Fighter Air Vehicle – C++ coding standard = standard établi par Lockheed Martin sur la base du standard MISRA C.
- Le standard JSF C++ est un standard de codage qui définit un sous-ensemble sécurisé du langage C++ spécialement pensé pour le développement de logiciels embarqués dans des véhicules aériens.
- Utilisé dans l'avionique et dans le spatial.

Règles et standards de codage
Le standard JSF AV C++

- Standard dont le but est d'aider les programmeurs à développer du code conforme aux principes des logiciels critiques en terme de sécurité
  - c'est-à-dire du code ne contenant pas de défauts pouvant mener à une défaillance catastrophique entrainant des dommages importants aux personnes ou aux équipements.
- Le document définissant les règles peut être téléchargé ici :
  - http://www2.research.att.com/~bs/**JSF-**AV-rules.pdf
- Les pages suivantes donnent un aperçu d'un certain nombre de règles issues de ce standard.
- Se reporter au document complet, pour avoir un accès exhaustif à l'ensemble des règles et justifications de ces règles.

Règles et standards de codage
Le standard JSF AV C++

Règle 1

- Toute fonction (ou méthode) ne doit pas contenir plus de 200 lignes de code (métrique L-SLOCs).
- Justification : les fonctions longues ont tendance à être complexes et donc difficiles à appréhender et à tester.

Règle 3

- Toute fonction doit avoir une complexité cyclomatique inférieure ou égale à 20.
- Justification : limiter la complexité des fonctions.
- Exception : une fonction contenant une clause « switch » avec plusieurs « case » peut excéder cette limite.

Règles et standards de codage
Le standard JSF AV C++

Règle 41

- Les lignes de code doivent avoir une taille maximale de 120 caractères.
- Justification : lisibilité et style. Les lignes de code très longues peuvent être difficiles à lire et à comprendre.

Règles et standards de codage
Le standard JSF AV C++

Règle 42

- Chaque expression, déclaration ou instruction doit être positionnée sur une ligne séparée.
- Justification : simplicité, lisibilité et style

```
x = 7; y=3;              // Incorrect: multiple expression statements on the same line.
a[i] = j[k]; i++; j++;   // Incorrect: multiple expression statements on the same line.
a[i] = k[j];             // Correct.
i++;
j++;
```

```
for( i = 0 ; i < max ; ++i) fun();   // Incorrect: multiple expression statements on the same line.
for(i = 0 ; i < max ; ++i)           // Correct
{
   foo();
}
```

Règles et standards de codage
Le standard JSF AV C++

Règle 45

- Tous les mots dans un identificateur doivent être séparés par le caractère ‘_’.
- Justification : lisibilité et style.

Règle 47

- Les identificateurs ne doivent pas commencer par le caractère ‘_’.
- Justification: ‘_’ est souvent utilisé comme premier caractère dans le nom des fonctions de bibliothèque (e.g. _main, _exit, etc.). Afin d’éviter des collisions de nom, les identificateurs ne devraient jamais commencer par ‘_’.

Règles et standards de codage
Le standard JSF AV C++

Règle 49

- Tous les acronymes dans un identificateur doivent être composés de lettres majuscules.
- Note : un acronyme doit toujours être écrit en majuscule, même si l'acronyme est situé dans une partie d'un identificateur qui devrait être en minuscule selon d'autres règles.
- Justification : lisibilité

Règles et standards de codage
Le standard JSF AV C++

Règle 50

- Le premier mot d'un nom de classe, de structure, d'espace de nom, d'énumération ou de type créé à l'aide de typedef doit commencer par une lettre majuscule. Toutes les autres lettres doivent être minuscules.
- Justification: Style.
- Exemple:

```
class Diagonal_matrix { … };                // Only first letter is capitalized;
enum RGB_colors {red, green, blue};         // RGB is an acronym so all letters are un upper case
```

Règles et standards de codage
Le standard JSF AV C++

Règle 51

- Tous les noms de fonction ou de variable doivent être constitués exclusivement de lettres minuscules.
- Justification: Style.

```
class   Example_class_name
{
   public:
            uint16     example_function_name (void);
   private:
             uint16    example_variable_name;
};
```

Règles et standards de codage
Le standard JSF AV C++

Règle 59

- Les instructions formant le corps d'un if, else if, else, while, do…while ou for doivent toujours être entourées d'accolades, même si les accolades forment un bloc vide.
- Justification : lisibilité. Il peut être difficile de voir le terminateur ";" quand il apparaît seul.

```
if (flag == 1)
{
    success ();
}
else                                // Incorrect: log_error() was added at a later time
    clean_up_resources();           //            but is not part of the block (even though
    log_error();                    //            it is at the proper indentation level).
```

Règles et standards de codage
Le standard JSF AV C++

Règle 60

- Les accolades ("{}") qui entourent un bloc doivent être placées dans la même colonne, sur des lignes séparées directement avant et après le bloc.

```
if (var_name == true)
{
}
else
{
}
```

Règles et standards de codage
Le standard JSF AV C++

Règle 62

- L'opérateur de déréférence '*' et l'opérateur d'adressage '&' doivent être directement connectés au spécificateur du type.

```
int32_t*  p;        // Correct
int32_t   *p;       // Incorrect
int32_t*  p, q;     // Probably error.
```

Règles et standards de codage
Le standard JSF AV C++

Règle 67

- Les données “public” et “protected” doivent seulement être utilisées dans des structures et pas dans des classes.
- Justification: une classe est capable de maintenir la cohérence de son état interne en contrôlant l’accès à ses données. Cependant, une classe ne peut pas contrôler l’accès à ses membres si ceux-ci sont non-privés. C’est pourquoi, toutes les données dans une classe doivent être privées.

Règles et standards de codage
Le standard JSF AV C++

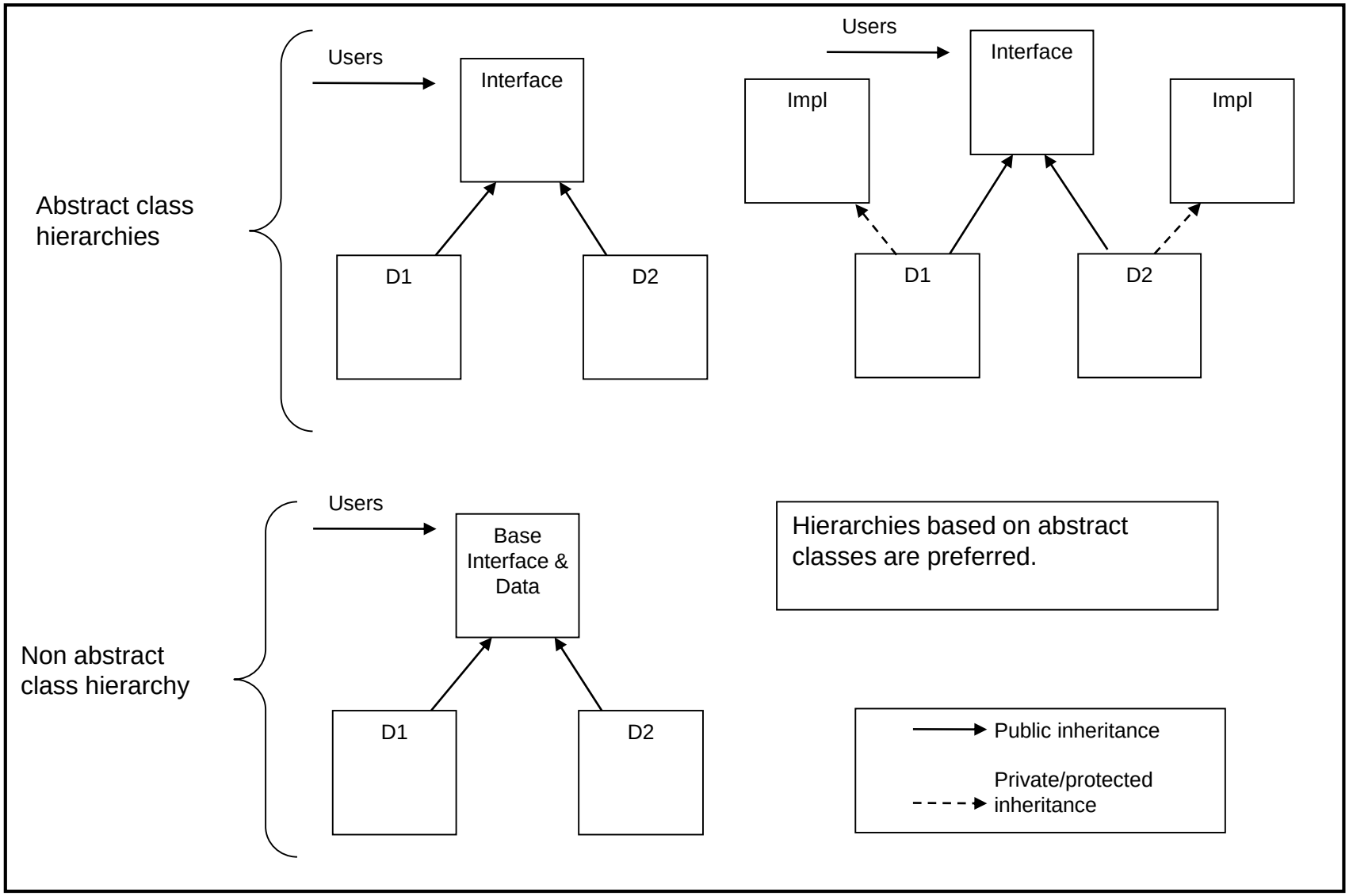
Règle 87

- Les hiérarchies de classes doivent être basées sur des classes abstraites.
- Justification: Les hiérarchies basées sur des classes abstraites tendent à concentrer la conception sur la production d'interfaces propres, à garder les détails d'implémentation en dehors des interfaces, et à minimiser les dépendances de compilation tout en permettant la coexistence d'implémentations alternatives.

Règle 88

- L'héritage multiple doit seulement être autorisé dans les cas suivants : n interfaces plus m implémentations privées, et au plus une implémentation protégée.
- Justification : L'héritage multiple peut mener à des hiérarchies d'héritage qui sont difficiles à comprendre et à maintenir.

Règles et standards de codage
Le standard JSF AV C++



Règles et standards de codage
Le standard JSF AV C++

Règle 113

- Les fonctions doivent avoir un seul point de sortie.
- Justification: De nombreux points de sortie dans une fonction ont tendance à rendre la fonction à la fois difficile à comprendre et à analyser.

Règles et standards de codage
Le standard JSF AV C++

Règle 116

- Les paramètres de petite taille et de type concret (taille de deux ou trois mots) doivent être passés par valeur, sauf si des changements doivent être apportés aux paramètres dans la fonction appelante.
- Justification: Passer des arguments par valeur est la méthode la plus simple et la plus sûre pour des petits objets de type concret.

Règle 117

- Les paramètres de fonction doivent être passées par référence si des valeurs NULL ne sont pas possibles.

Règle 118

- Les paramètres de fonction doivent être passées par pointeur si des valeurs NULL sont possibles.

Règles et standards de codage
Le standard JSF AV C++

Règle 143

- Les variables ne devraient pas être introduites tant qu'elles ne sont pas initialisées avec des valeurs pertinentes.
- Justification : Prévenir l'utilisation de variables initialisées avec des valeurs non pertinentes.

Règles et standards de codage
Le standard JSF AV C++

```
void fun_1()                          // Poor implementation
{
   int32_t  i;                        // Bad: i is prematurely declared (the intent is to use i in the for
                                      //        loop only)
   int32_t  max=0;                    // Bad: max initialized with a dummy value.
   …                                  // Bad: i and max have meaningless values in this
                                      //        region of the code.
   max = f(x);
   for (i=0 ; i<max ; ++i)
   {
      …
   }
   ….                                 // Bad: i should not be used here, but could be used anyway
}

void fun_1()                          // Good implementation
{
   ….
   int32_t max = f(x);                // Good: max not introduced until meaningful value is
                                      //          available
   for (int32_t i=0 ; i<max ; ++i)    // Good: i is not declared or initialized until needed
   {                                  // Good: i is only known within the for loop's scope
      …
   }
}
```

Règles et standards de codage
Le standard JSF AV C++

Règle 152

- Les déclarations de variable multiples ne doivent pas être autorisées sur la même ligne.
- Justification : Augmenter la lisibilité et prévenir la confusion.

```
int32_t* p, q;                                          // Probably error.
int32_t  first button_on_top_of_the_left_box, i;        // Bad: Easy to overlook i
```

Règles et standards de codage
Le standard JSF AV C++

Règle 180

- Les conversions de type implicites qui peuvent aboutir à une perte d'information ne doivent pas être utilisées.
- Justification : Le programmeur peut ne pas être conscient de la perte d'information.

```
int32_t     i =1024;
char        c = i;          // Bad: (integer-to-char) implicit loss of
information.
float32_t   f = 7.3;
int32_t     j= f;           // Bad: (float-to-int) implicit loss of information.
int32_t     k = 1234567890;
float32_t   g = k;          // Bad: (int-to-float) implicit loss of information
                            //      (g will be 1234567936)
```

Règles et standards de codage
Le standard JSF AV C++

Règle 193

- Toute clause “case” dans une structure “switch” doit être terminée par un “break”.

Règle 206

- L’allocation et la désallocation dynamique de mémoire (sur le tas) est interdite après la phase d’initialisation.
- Justification: des allocations (new/malloc) et des désallocations (delete/free) répétées peuvent aboutir à la fragmentation de la mémoire et donc à des délais non-déterministes durant les accès au tas (heap).

Règles et standards de codage
Le standard JSF AV C++

Règle 207

- Les données globales non encapsulées doivent être évitées.
- Justification : Les données globales sont dangereuses puisqu'aucune protection d'accès est fourni à l'égard des données.

```
int32_t x=0;          // Bad: Unencapsulated global object.

class Y {
    int32_t x;
    public:
        Y(int32_t y_);
        int32_t get_x();
        void set_x();
};

Y y (0);              // Bad: Unencapsulated global object.
```

Règles et standards de codage
Le standard JSF AV C++

Règle 208

- Les exceptions C++ ne doivent pas être utilisées (i.e. les instructions throw, catch et try ne doivent pas être utilisées.)
- Justification: Actuellement, les outils n’apportent pas une prise en charge satisfaisante des exceptions.

Qualité et fiabilité des logiciels embarqués

1. Fiabilité, maintenabilité, disponibilité
2. Méthodes et techniques pour l'évaluation de la fiabilité
3. Qualité du code logiciel
4. Métriques
5. Mesure de la complexité du code
6. Règles et standards de codage
7. Exemple : le standard JSF AV C++
8. Exercice

Exercice

- Soit la méthode « buildPacketReceptionHeader » donnée dans la page suivante.
  1. Calculer la complexité cyclomatique de la méthode en comptant les mots-clés et opérateurs qui introduisent des décisions dans le flot d'exécution.
  2. Représenter la méthode sous la forme d'un graphe.
  3. En comptant le nombre de nœuds et d'arcs, calculer à nouveau la complexité cyclomatique de la méthode.
  4. Combien de tests unitaires faut-il prévoir pour tester la méthode si l'on veut une couverture de 100% des branches ?
  5. Que peut-on dire du niveau de complexité de cette méthode ?
  6. Le code de cette méthode est-il conforme au standard de codage JSF ? Si non, quelles sont les règles non respectées ?

```
GsbPacketHeader GsdGrspwManagerImpl::buildPacketReceptionHeader(bool isRmap) {
     GsbCucTime cucTime;
     cucTime.setCoarseTime(GsbCucTime::DEFAULT_COARSE_TIME);
     cucTime.setFineTime(GsbCucTime::DEFAULT_FINE_TIME);
     GsbPacketHeader header;
     header.setGsbCucTime(cucTime);
     header.setPacketLenght(gsdGrspwRxDescriptor->getPacketLength());
     header.setStatus(SUCCESS);
     header.setError(NO_ERROR);
     if (gsdGrspwRxDescriptor->getTruncated()) {
          header.setError(TRUNCATED);
          header.setStatus(FAILURE);
     }
     else {
          if (gsdGrspwRxDescriptor->getEepTermination()) {
               header.setError(EEP);
               header.setStatus(FAILURE);
          }
          else {
               if (isRmap) {
                    if (gsdGrspwRxDescriptor->getHeaderCrc()) {
                         header.setError(CRC_HEADER_ERROR);
                         header.setStatus(FAILURE);
                    }
                    else {
                         if (gsdGrspwRxDescriptor->getDataCrc()) {
                              header.setError(CRC_DATA_ERROR);
                              header.setStatus(FAILURE);
                         }
                    }
               }
          }
     }
     return header;
}
```

Correction de l'exercice

1. Calculer la complexité cyclomatique de la méthode en comptant les mots-clés et opérateurs qui introduisent des décisions dans le flot d'exécution.
   - On compte 5 « if », ce qui donne une complexité de 5 + 1 = 6
2. Représenter la méthode sous la forme d'un graphe.
   - Voir page suivante
3. En comptant le nombre de nœuds et d'arcs, calculer à nouveau la complexité cyclomatique de la méthode.
   - Nombre de nœuds = 11
   - Nombre d'arcs = 15
   - Complexité = 15-11+2 = 6
4. Combien de tests unitaires faut-il prévoir pour tester la méthode si l'on veut une couverture de 100% des branches ?
   - 6
5. Que peut-on dire du niveau de complexité de cette méthode ?
   - Méthode simple, sans trop de risque.
   - Compatible avec le standard JSF.

Correction de l'exercice

6. Le code de cette méthode est-il conforme au standard de codage JSF ? Si non, quelles sont les règles non respectées ?
   - Non.
   - Règle 45 non respectée : Tous les mots dans un identificateur doivent être séparés par le caractère '_'.
   - Règle 49 non respectée : Tous les acronymes dans un identificateur doivent être composés de lettres majuscules.
   - Règle 51 non respectée : Tous les noms de fonction ou de variable doivent être constitués exclusivement de lettres minuscules.
   - Règle 60 non respectée : Les accolades ("{}") qui entourent un bloc doivent être placées dans la même colonne, sur des lignes séparées directement avant et après le bloc.

```
GsbPacketHeader GsdGrspwManagerImpl::buildPacketReceptionHeader(bool isRmap) {
    GsbCucTime cucTime;
    cucTime.setCoarseTime(GsbCucTime::DEFAULT_COARSE_TIME);
    cucTime.setFineTime(GsbCucTime::DEFAULT_FINE_TIME);
    GsbPacketHeader header;
    header.setGsbCucTime(cucTime);
    header.setPacketLenght(gsdGrspwRxDescriptor->getPacketLength());
    header.setStatus(SUCCESS);
    header.setError(NO_ERROR);
    if (gsdGrspwRxDescriptor->getTruncated()) {
        header.setError(TRUNCATED);
        header.setStatus(FAILURE);
    }
    else {
        if (gsdGrspwRxDescriptor->getEepTermination()) {
            header.setError(EEP);
            header.setStatus(FAILURE);
        }
        else {
            if (isRmap) {
                if (gsdGrspwRxDescriptor->getHeaderCrc()) {
                    header.setError(CRC_HEADER_ERROR);
                    header.setStatus(FAILURE);
                }
                else {
                    if (gsdGrspwRxDescriptor->getDataCrc()) {
                        header.setError(CRC_DATA_ERROR);
                        header.setStatus(FAILURE);
                    }
                }
            }
        }
    }
    return header;
}
```

1
2
3
4
5
6
7
8
9
10
11

Correction de l'exercice

Nombre de nœuds = 11
Nombre d'arcs = 15
Complexité = 15-11+2 = 6