La qualité du logiciel embarqué

EISTI

Octobre 2016

Ali Baktash

Objectifs

- Comprendre les **enjeux** et **fondamentaux** liés au domaine du logiciel embarqué

- Identifier les **principes, actions et outils** qui vous aideront à mettre en place une **démarche méthodique de progrès** dans la **production du logiciel embarqué de qualité**.

- Connaitre les **règles, référentiels, modèles et éléments normatifs** sur lesquels vous pourrez vous appuyer

Déroulement de la formation

4h30
- La problématique du Logiciel embarqué
- Les différents Cycles de développement logiciels (Cascade, V, Spirales, Agiles)
- La qualité au rendez-vous de chaque étape
- Rentrons dans les détails, MISRA

1h30h
- Règles de codages C-ANSI

1h30h
- Modèles de processus SPICE et CMMI

3h
- AUTOSAR

4h30h
- Sûreté de fonctionnement

Déroulement de la formation

La problématique du Logiciel embarqué

- Les différents Cycles de développement logiciels
- La qualité au rendez-vous de chaque étape
- Rentrons dans les détails, MISRA
- Règles de codages C-ANSI
- Modèles de processus SPICE et CMMI
- AUTOSAR

Selon vous?

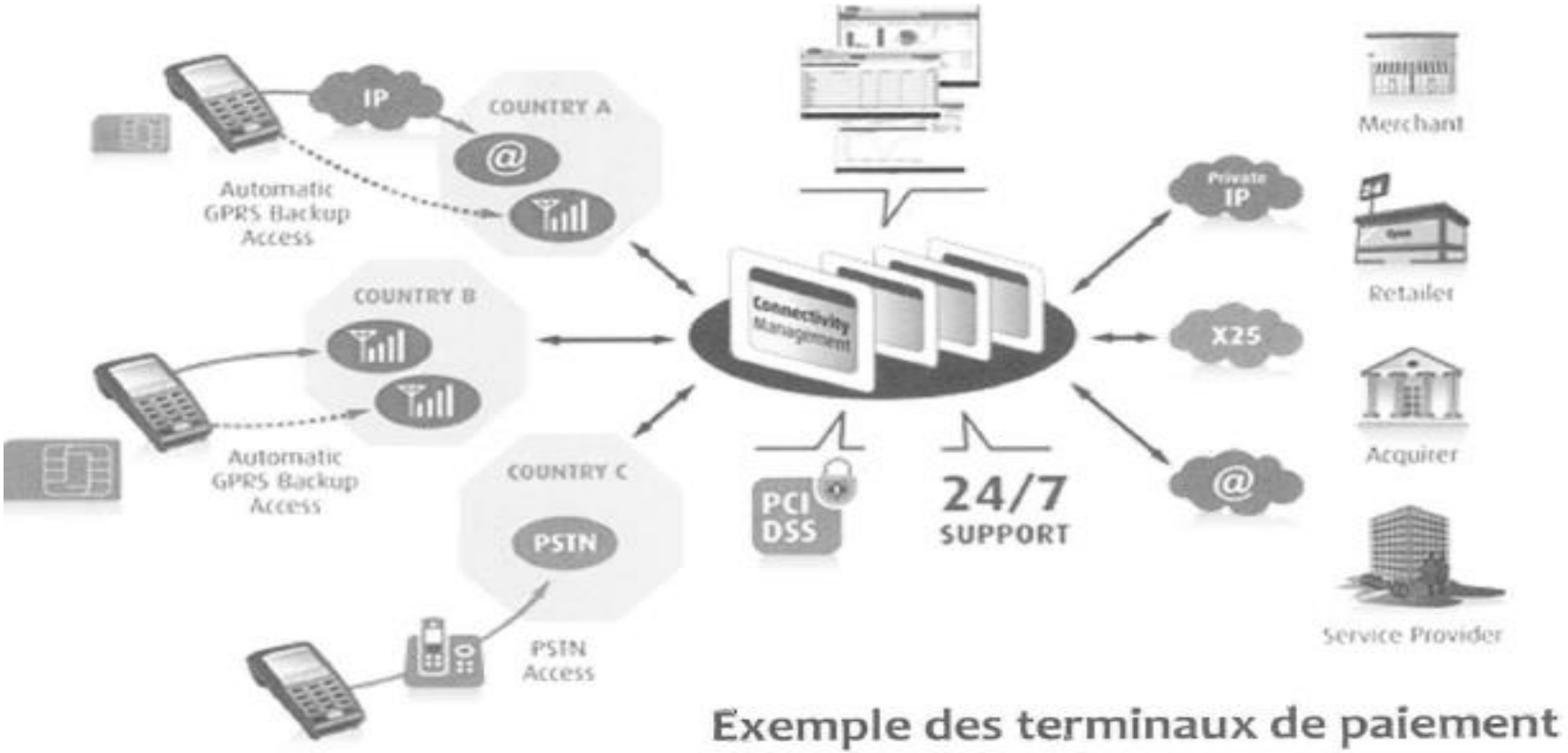
Qu’évoque pour vous la notion de logiciel ?

Logiciel embarqué est partout

- 90% des nouvelles fonctionnalités des automobiles sont apportées par l'électronique et l'informatique embarquée

- « Il y a plus d'informatique dans la Volvo S80 que dans un chasseur F15 » (Président d'Audi, Janvier 2000)

Environnement complexe



Exemple des terminaux de paiement

Fiabilité aléatoire

- Le logiciel n'est **pas fiable**,
- Il est incroyablement difficile de réaliser dans les délais prévus des logiciels satisfaisant leur cahier des charges.

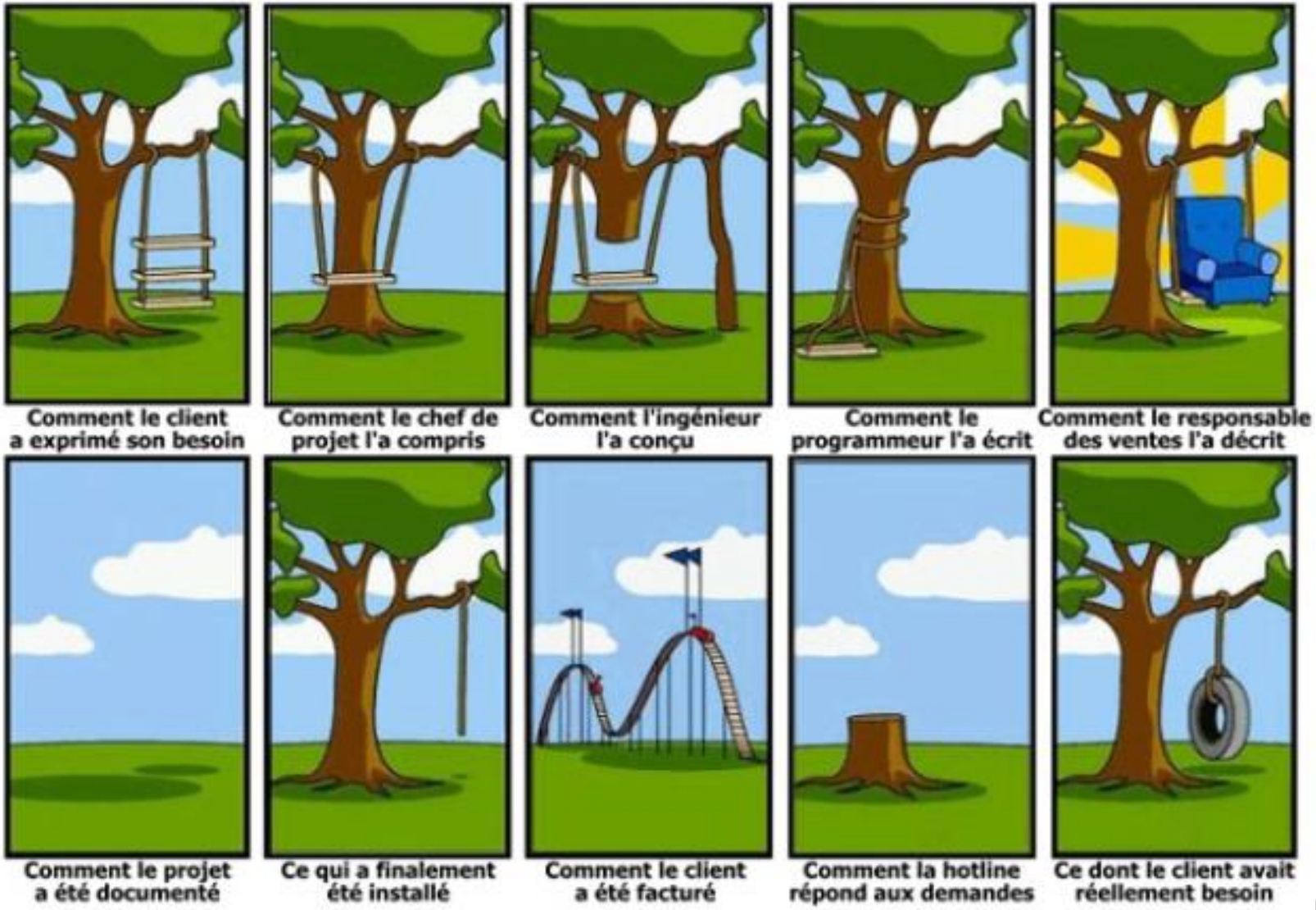
- Tout système comporte des bugs
  - Personne ne sait créer de logiciel sans défaut

Raisons d'échec des projets informatiques

• Exigences incomplètes	13,1%
• Manque d'implication des utilisateurs	12,4%
• Manque de ressources	10,6%
• Espérances déraisonnables	9,9%
• Manque de support de la direction	9,3%
• Changement de spécifications	8,7%
• Manque de planning	8,1%
• N'en veulent plus	7,5%

Source Standish group & Scientifics American

Explication possibles…



Caractéristiques d'un logiciel

- Le logiciel est facile à produire

  →Tout le coup se trouve dans le développement
- Le logiciel est immatériel et invisible

  →On ne peut l'observer qu'en l'utilisant et la qualité n'est pas vraiment apparente
- Le développement d'un logiciel est difficile à automatiser

  →Beaucoup de main d'œuvre
- Le logiciel ne s'use pas, mais il vieillit

  →Détérioration suite aux changements
  - Complexification, Duplication de code

  →Mauvaise conception au départ
  - Inflexibilité, Manque de modularité, Documentation insuffisante

  →Évolution du matériel

En résumé le Logiciel c'est…

- **Produit unique et atypique que l'on trouve partout**
- **Niveau d'abstraction + complexité = maîtrise difficile**
- **Activité centrée sur le développement = contrôle difficile**
- **Produit concret une fois terminé**
- **Bugs informatiques => dysfonctionnements majeurs des systèmes et services**
- **Corrections non maîtrisées = effets de bords + régression de l'existant**

Définition d'un logiciel

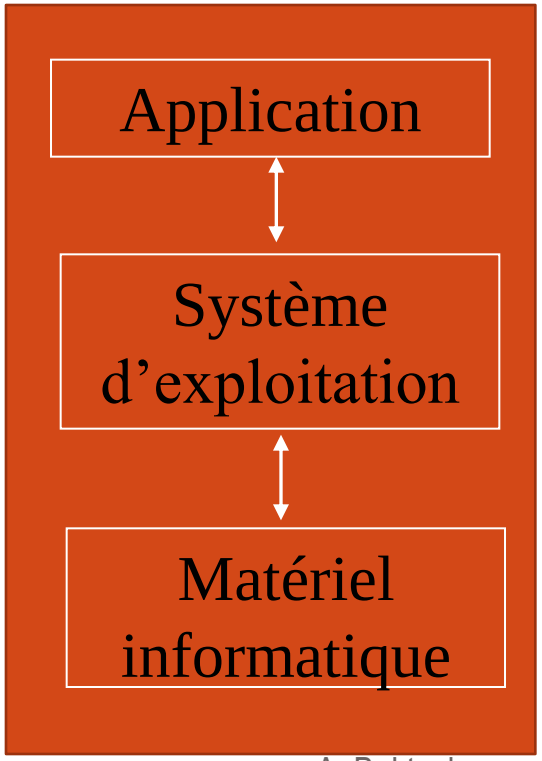
- Un logiciel est l'ensemble des programmes, des procédures et des documentations nécessaires au fonctionnement d'un système informatique

Programmes sources
Ensemble de textes décrivant formellement étape par étape, les opérations à effectuer

Code non exécutable

Compilateur

Code exécutable



Les activités logicielles

Répartition dans un projet logiciel Bien conduit

	Activités	définition
45%	Analyse des besoins	• Déterminer ce que le logiciel doit faire (et ne pas faire) • Déterminer les **ressources**, les **contraintes**…
	Spécification	• Établir une 1ere description du futur système • Consigner dans un document qui fait **référence**
	Conception architecture	• Définir et concevoir les composants logiciels et leur interactions
	Conception détaillé	• Elaborer les éléments internes de chaque composant • Décrire les structures de données, les algorithmes
10%	Programmation	• Réaliser les programmes, bibliothèques, modules
45%	Intégration	• Assembler les composants pour obtenir un exécutable
	Vérification et validation	• Vérifier la satisfaction des spécifications • Valider par rapport aux utilisateurs
	Maintenance	• Corriger les défauts • Améliorer certaines caractéristiques • Suivre les évolutions (besoin, environnement)

Déroulement de la formation

- La problématique du Logiciel embarqué
- Les différents Cycles de développement logiciels (Cascade, V, Spirales, Agiles)
- La qualité au rendez-vous de chaque étape
- Rentrons dans les détails, MISRA
- Règles de codages C-ANSI
- Modèles de processus SPICE et CMMI
- AUTOSAR

Qu’est-ce qu’un cycle de développement

- Définition:
  - Enchainent et interaction entre activités logicielles

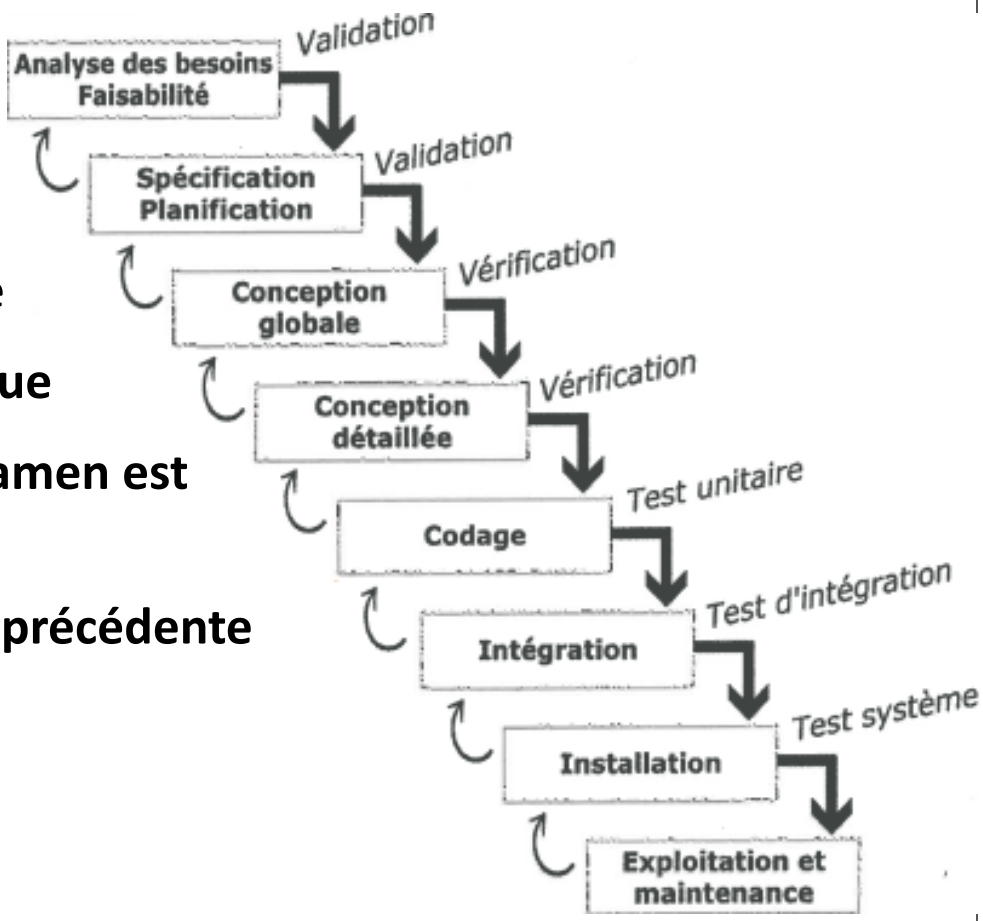
- But
  - Ne pas s’apercevoir des problèmes qu’à la fin
    - Contrôler l’avancement des activités en cours
    - Vérifier/valider les résultats intermédiaires

Qu’est-ce qu’un cycle de développement

- Objectif
  - Fiabiliser les processus de développement
    - Rationnels (visible)
    - Contrôlables et pilotable
    - Reproductibles et améliorations continues
- Types
  - Modèle en « cascade » (fin des années 1960)
  - Modèle en « V » (années 1980)
  - Modèle en spirale (Boehm, 1988)
  - Méthodes Agiles (Manifeste agile, 2001)

Modèle de développement en « Cascade »

- **Principe:**
  - Développement se divise en **étapes**
  - Une étape se termine à **une date**
  - Des **livrables** à la fin de chaque étape
  - Les résultats d'étapes soumis à la **revue**
  - On passe à l'étape d'après que si **l'examen est satisfaisant**
  - Une étape **ne remet en cause que la précédente**

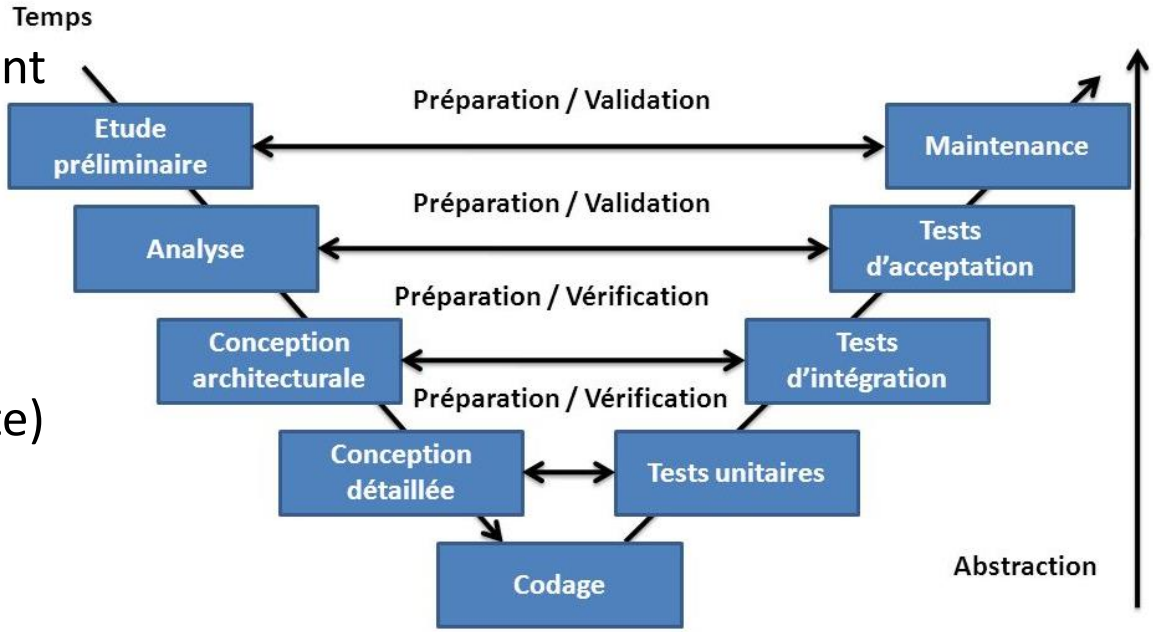


- Séduisant
  → Simple
- Moyennement réaliste
  → Trop séquentiel

Modèle de développement en « V »

Principes:

- Les premières étapes préparent les dernières
- Les étapes s'enchainent séquentiellement sur des branches descendantes (à gauche) et montantes (à droite) du « V »
- Les résultats des étapes de départ (à gauche)sont utilisés par les étapes d'arrivées (à droite)



- Réaliste
  →Modèle plus élaboré que le précédent
- Eprouvé
  →Le plus utilisé dans l'industrie

Modèle de développement en « Spirale »

- Principe
  - Développement **itératif** (prototypes)
  - Chaque mini-cycle se déroule en **4 phases**

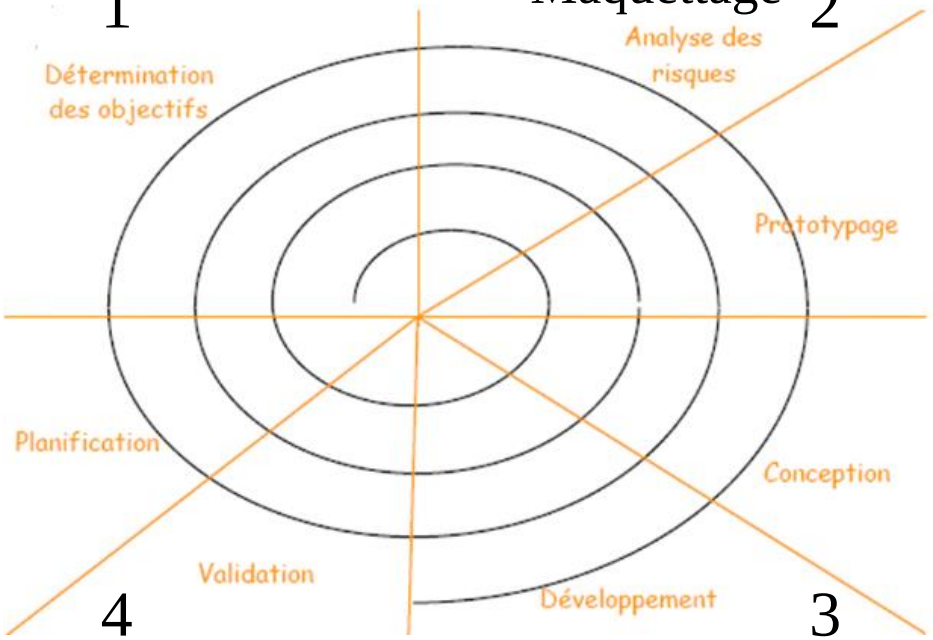
- Nouveau
  - →Analyse des risques
  - →Maquettes et prototypage
- Complet
  - →Complexe et général
- Lourd
  - →Requiert effort important
- Ciblé
  - →Utilisé pour des projets à risques ou innovants

Analyse des besoins / spécification

1

Analyse des risques et Maquettage

2



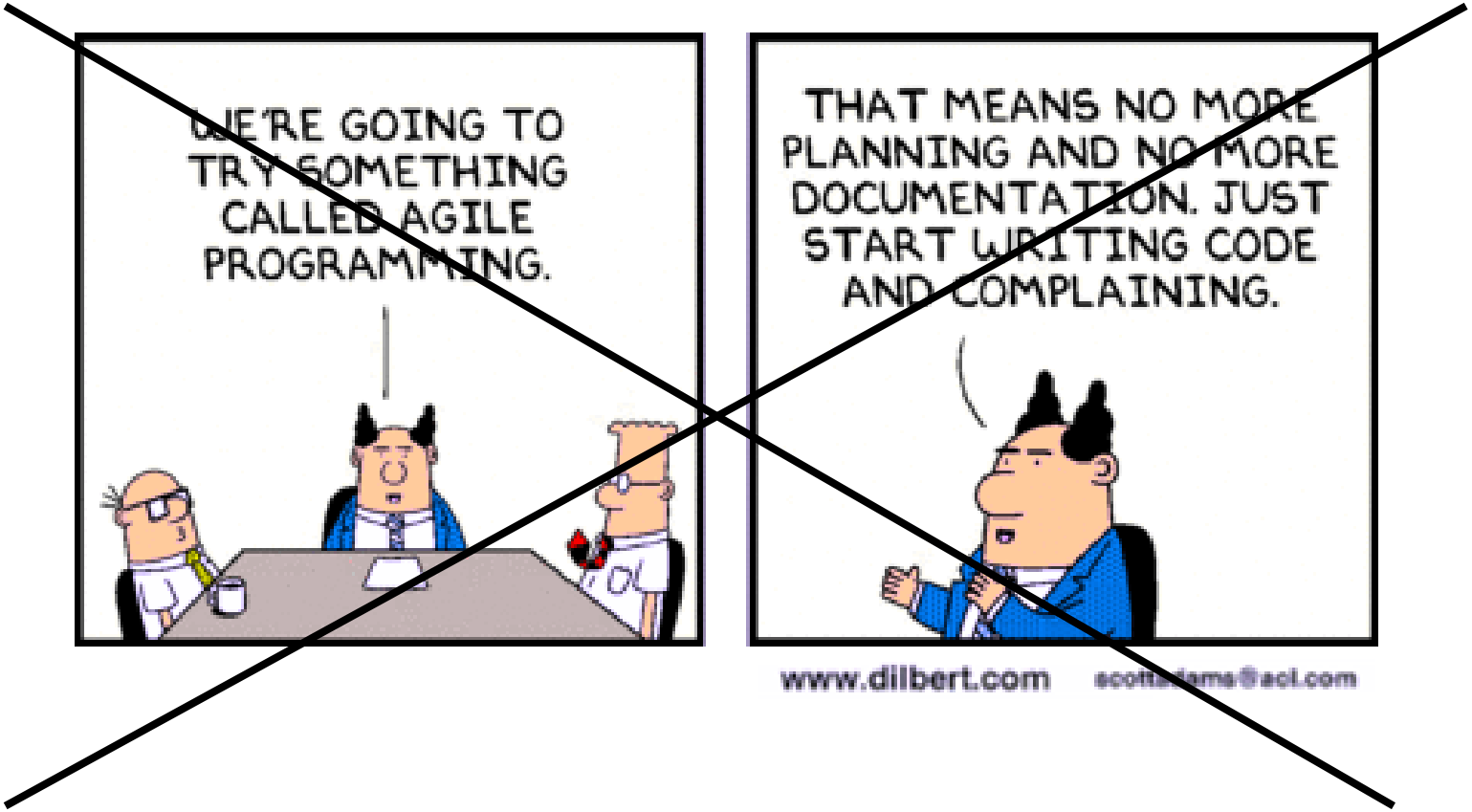
4

Vérification/ Validation/ planification

3

Conception et implémentation

Méthodes « Agiles » (ou le lean appliqué au logiciel



Méthodes « Agiles » (ou le lean appliqué au logiciel

- Années 1990:
  - Réaction contre les grosses méthodes
  - Prise en compte des spécificités du logiciel
- 2001
  - Accord sur le texte unificateur « Manifeste Agile »

  Émis par 17 figures éminentes du développement logiciel
- Depuis
  - Projets « Agiles » mixent des éléments de principales méthodes

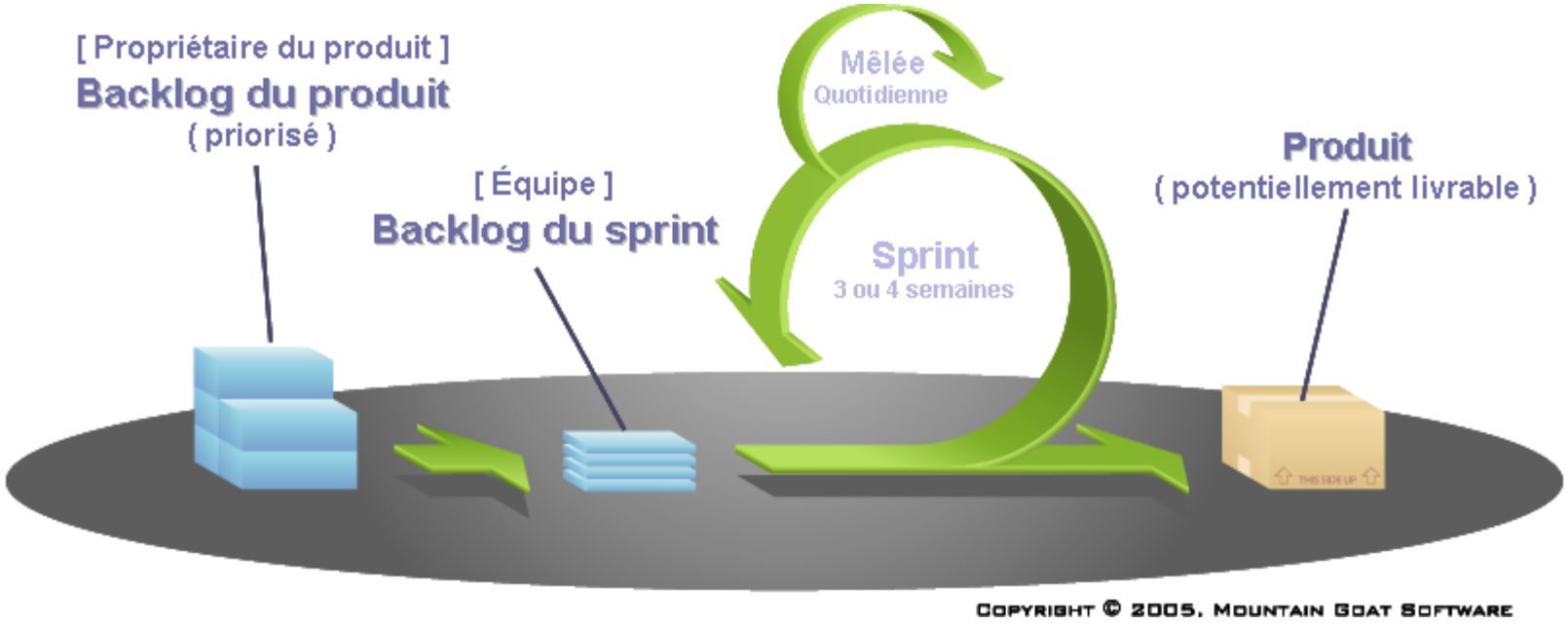
Les valeurs de l'agilité

Personnes et interaction	Plutôt que	Processus et outils
Un produit opérationnel	Plutôt que	Documentation exhaustive
Collaboration avec le client	Plutôt que	Négociation d'un contrat
Adaptation au changement	Plutôt que	Suivi d'un plan

Les 12 principes du manifeste « Agile »

1. « Notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée. »
2. « Accueillez positivement les changements de besoins, même tard dans le projet. Les processus Agiles exploitent le changement pour donner un avantage compétitif au client. »
3. « Livrez fréquemment un logiciel opérationnel avec des cycles de quelques semaines à quelques mois et une préférence pour les plus courts. »
4. « Les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet. »
5. « Réalisez les projets avec des personnes motivées. Fournissez-leur l'environnement et le soutien dont ils ont besoin et faites-leur confiance pour atteindre les objectifs fixés. »
6. « La méthode la plus simple et la plus efficace pour transmettre de l'information à l'équipe de développement et à l'intérieur de celle-ci est le dialogue en face à face. »
7. « Un logiciel opérationnel est la principale mesure d'avancement. »
8. « Les processus Agiles encouragent un rythme de développement soutenable. Ensemble, les commanditaires, les développeurs et les utilisateurs devraient être capables de maintenir indéfiniment un rythme constant. »
9. « Une attention continue à l'excellence technique et à une bonne conception renforce l'Agilité. »
10. « La simplicité – c'est-à-dire l'art de minimiser la quantité de travail inutile – est essentielle. »
11. « Les meilleures architectures, spécifications et conceptions émergent d'équipes autoorganisées. »
12. « À intervalles réguliers, l'équipe réfléchit aux moyens de devenir plus efficace, puis règle et modifie son comportement en conséquence. »

La méthode Scrum, le cycle de développement



Déroulement de la présentation

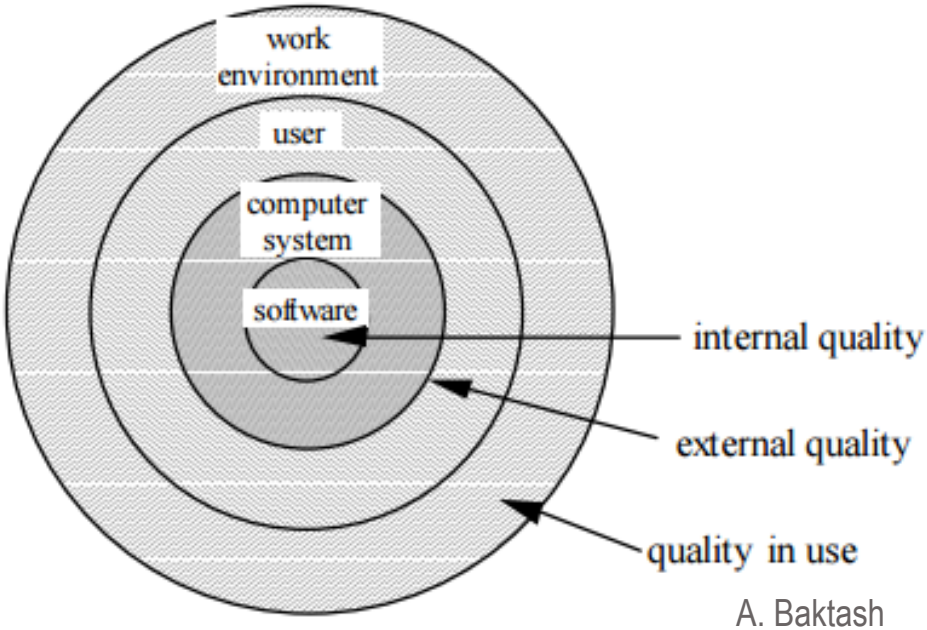
- La problématique du Logiciel embarqué
- Les différents Cycles de développement logiciels
- La qualité au rendez-vous de chaque étape : dans le bas du cycle en V
- Rentrons dans les détails, MISRA
- Règles de codages C-ANSI
- Modèles de processus SPICE et CMMI
- AUTOSAR

Selon vous?

Qu’est-ce qu’un logiciel de bonne qualité?

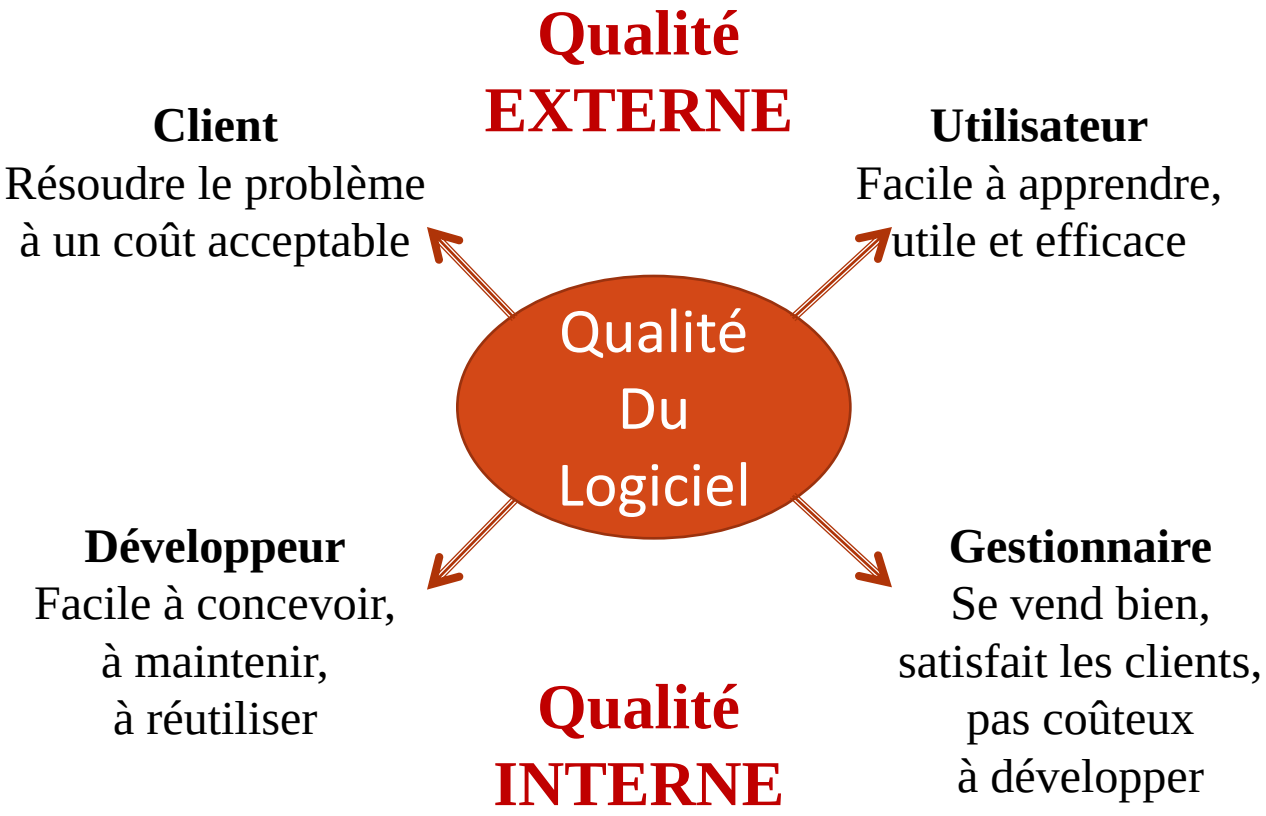
La qualité d'un logiciel selon ISO / IEC 9126

- La norme **ISO/CEI 9126** définit un langage commun pour modéliser les **qualités d'un logiciel** et organise leur **classification** afin d'en faciliter son utilisation lors de processus d'évaluation logiciel :
  1. Caractéristique et sous caractéristiques
  2. Métriques externes
  3. Métriques internes
  4. Qualité à l'utilisation



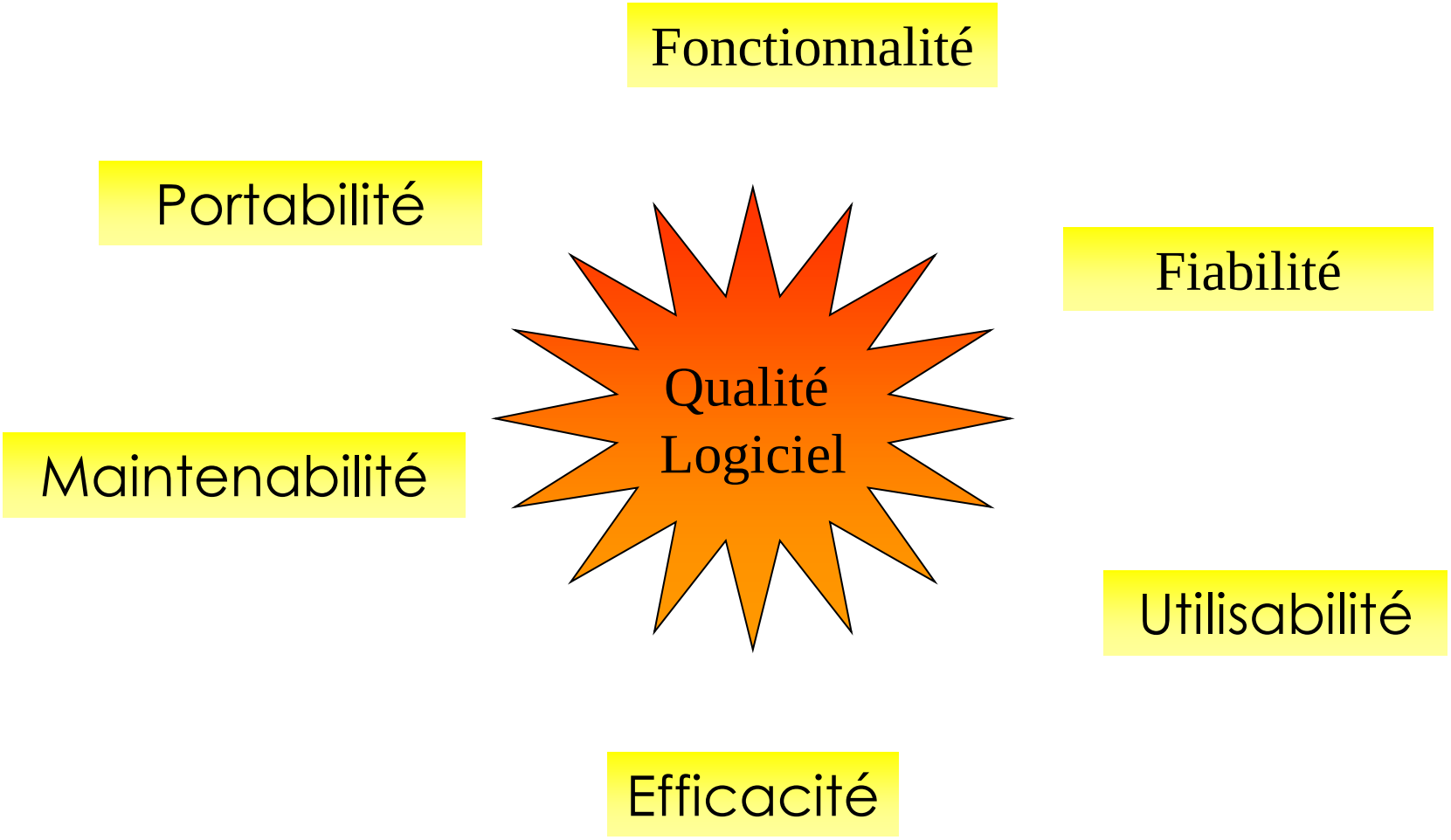
Métriques externes et internes

Mesures applicables au logiciel exécutable durant les tests ou après le développement

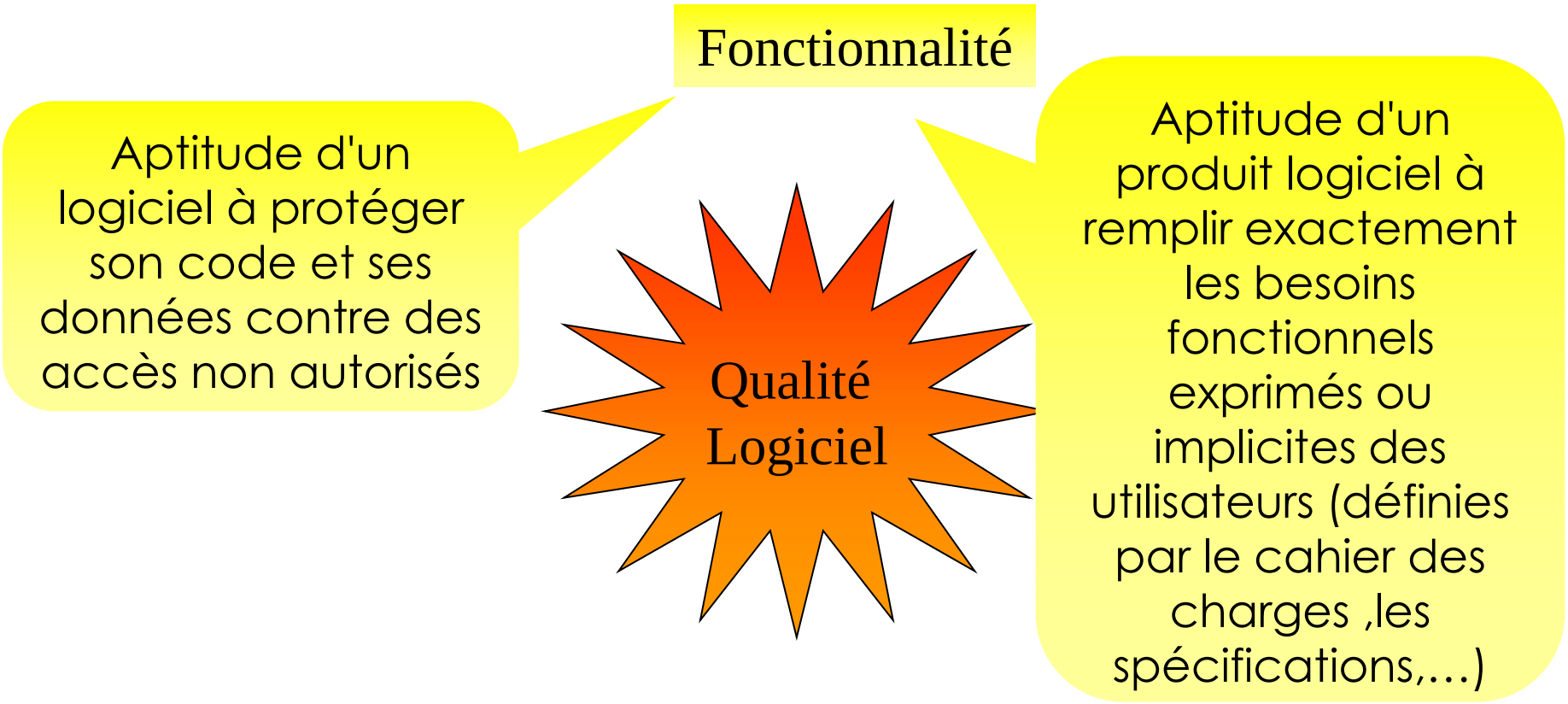


Mesures intrinsèques applicables au logiciel non exécutable durant la conception et le codage

Caractéristiques qualité du logiciel



Caractéristiques qualité du logiciel



Caractéristiques qualité du logiciel

Fiabilité

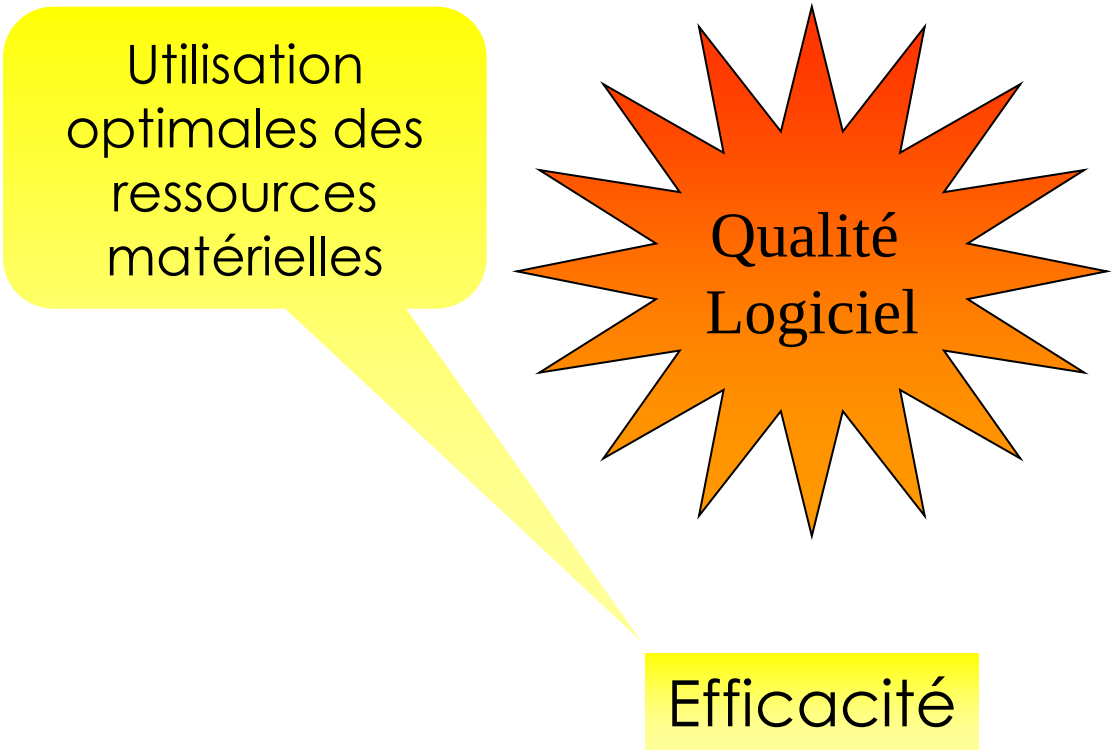
Aptitude d'un produit logiciel à fonctionner dans des conditions précises et pendant une période déterminé

Caractéristiques qualité du logiciel

Facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation

Utilisabilité

Caractéristiques qualité du logiciel



Caractéristiques qualité du logiciel

Maintenabilité

Facilité avec laquelle un logiciel se prête à une modification ou à une extension des fonctions qui lui sont demandées

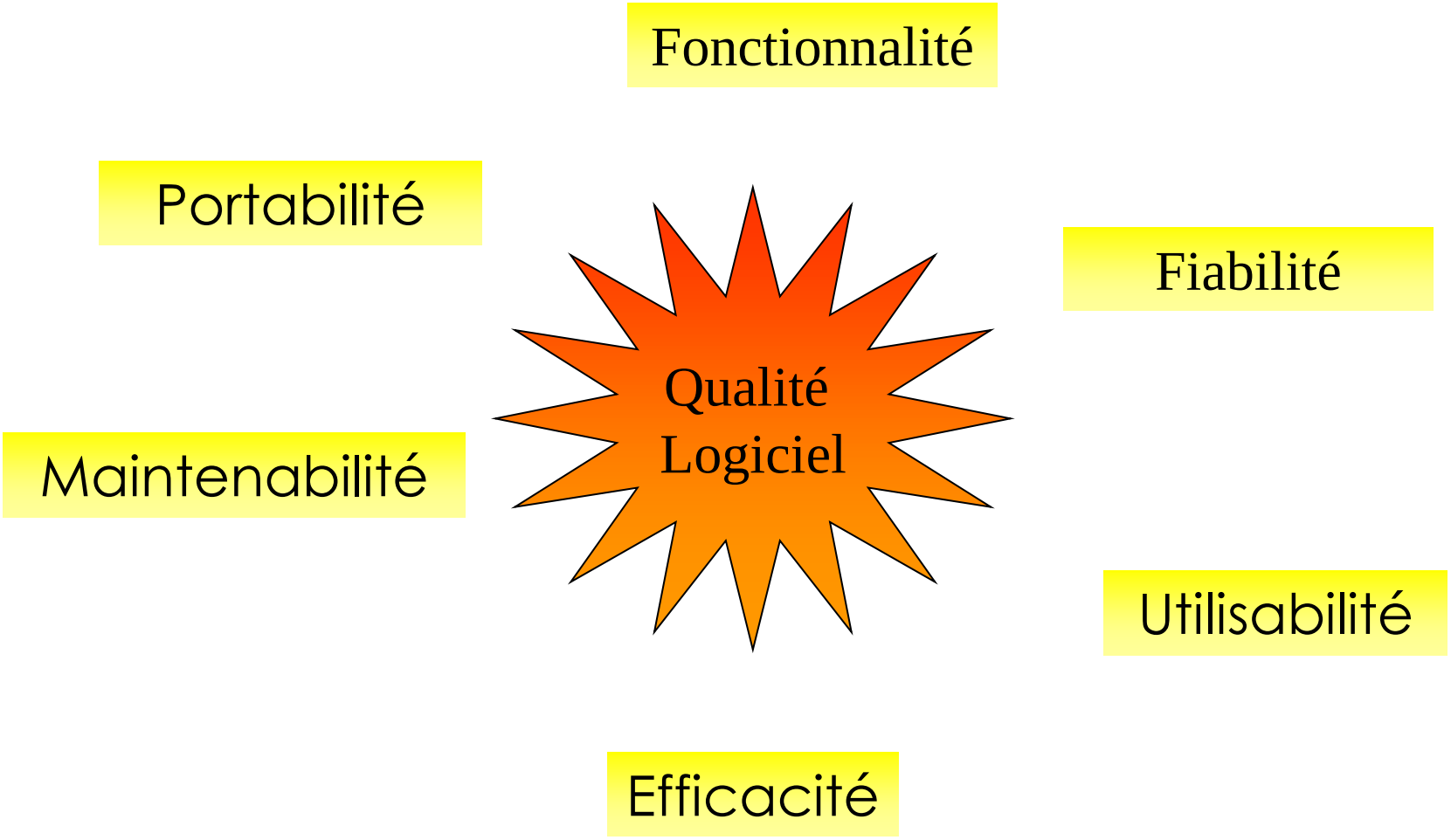
Caractéristiques qualité du logiciel

Aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications

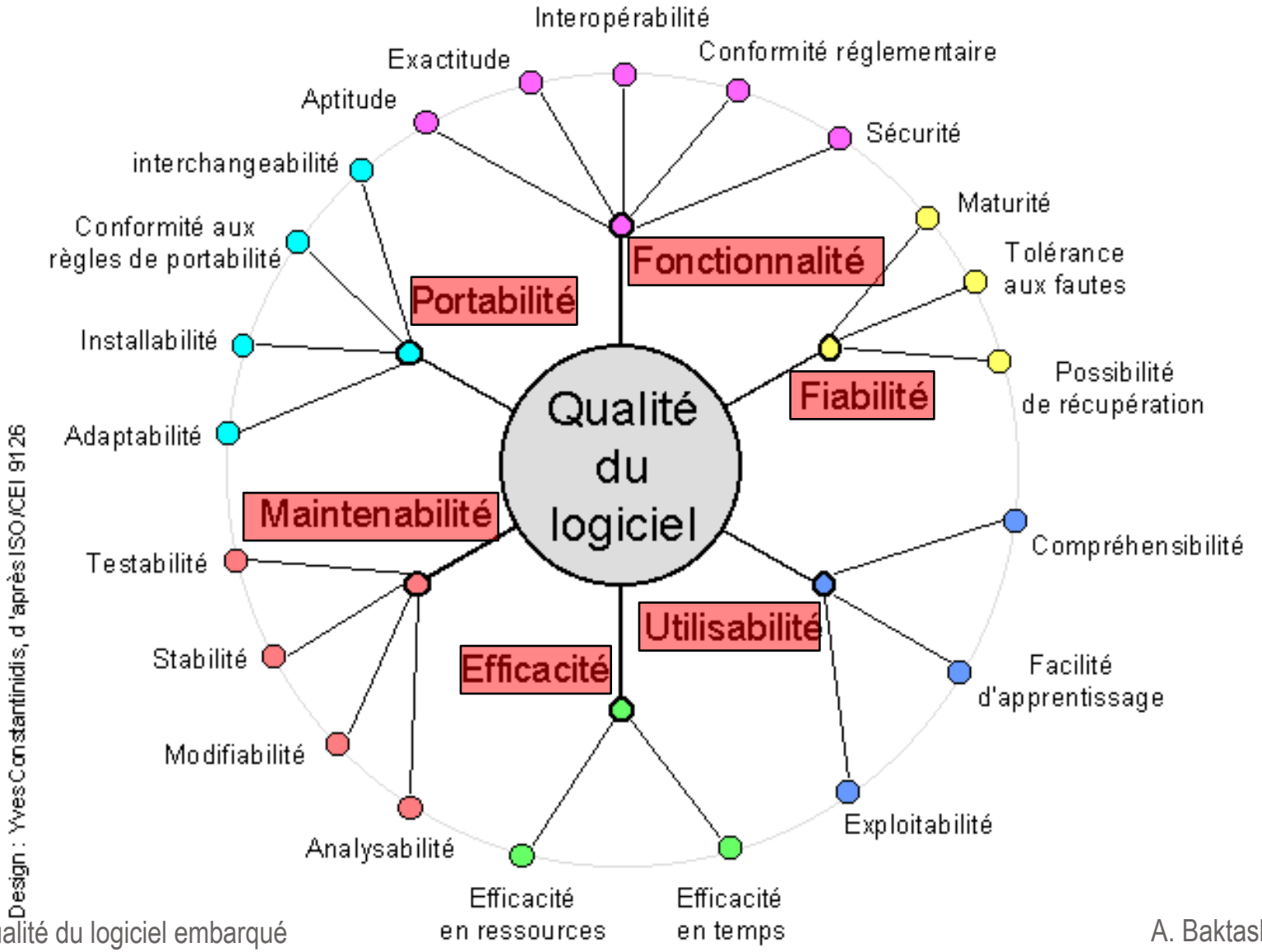
Portabilité

Facilité avec laquelle un logiciel peut être transférée sous différents environnements matériels et logiciels

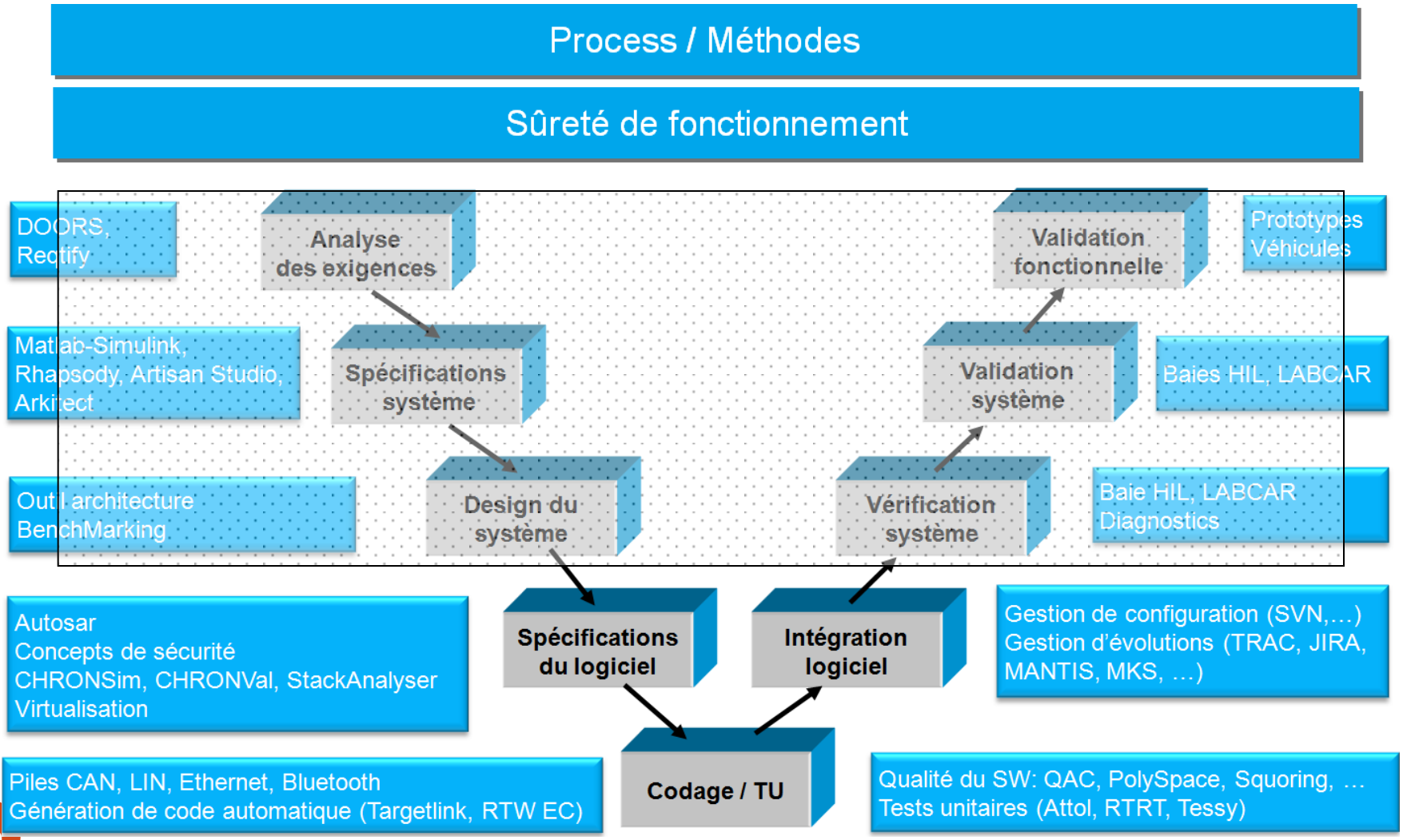
Caractéristiques qualité du logiciel



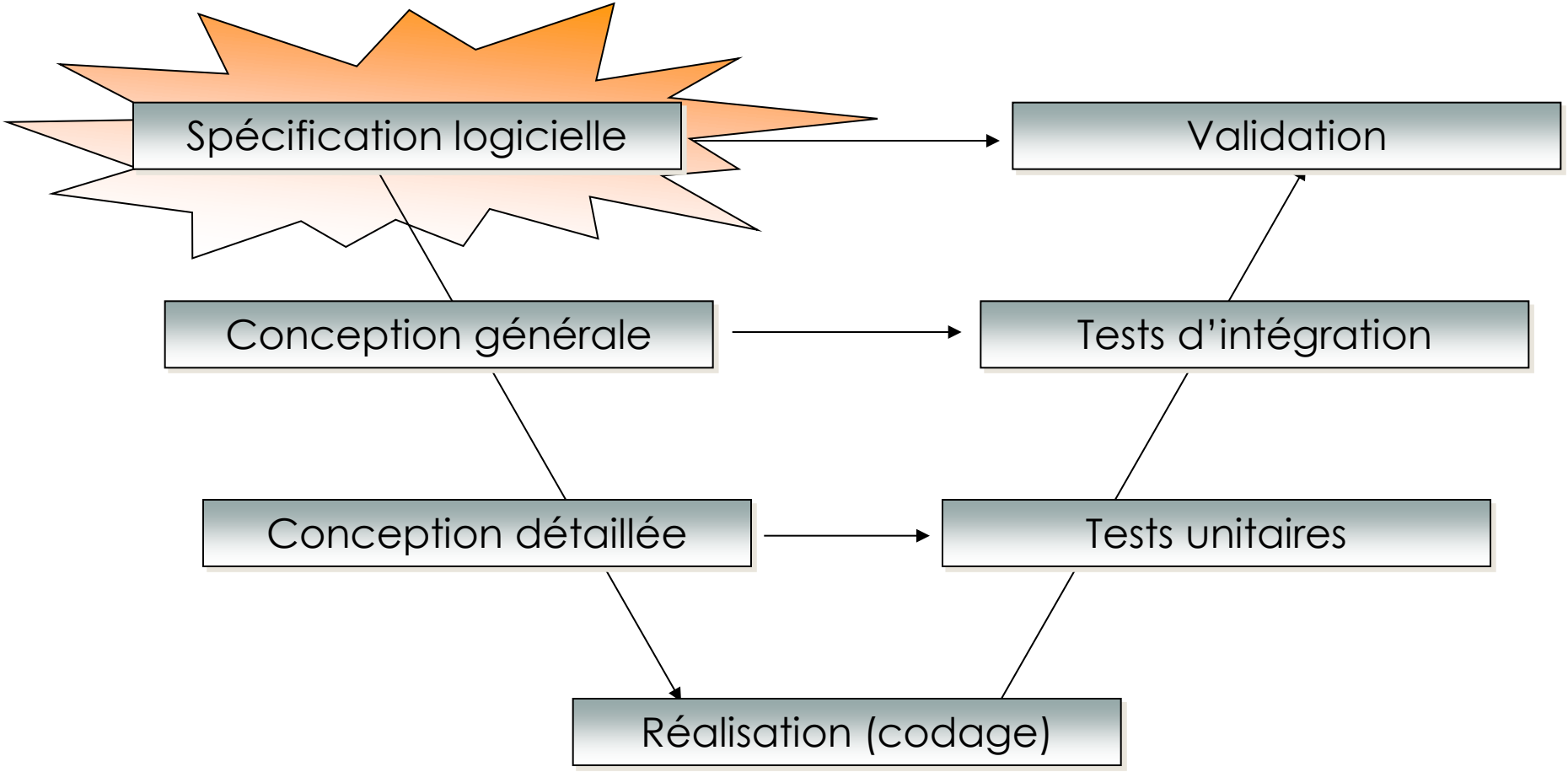
Caractéristiques et sous-caractéristiques qualité du logiciel



La qualité au rendez-vous de chaque étape dans le bas du Cycle en V



Spécification logicielle



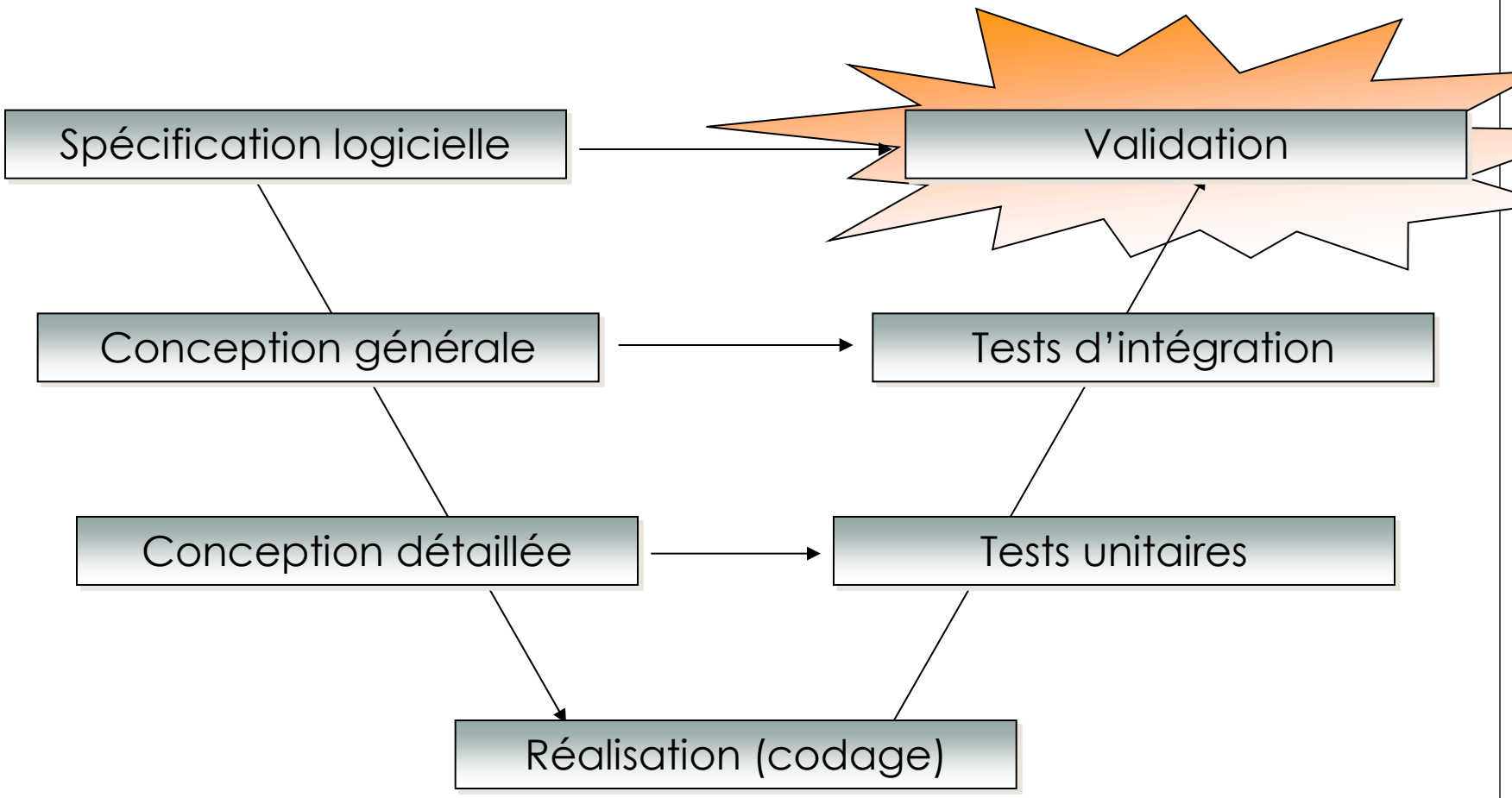
Spécification logicielle

- But
  - Description « haut niveau » des traitements et contraintes

Spécification logicielle

- Principes
  - Doit lister:
    - **Tous les traitements** pris en charge par le logiciel
    - Les contraintes **temps réel** (performances, temps de réponse, …)
    - Les contraintes de **sécurité**
    - Les contraintes de **réalisation** (utilisation de couches logicielles existantes, limitation en taille de code / RAM, format des données manipulées, …)
    - Les **modes d'utilisation** du logiciel (nominal, dégradé, …)

Validation


Spécification logicielle
Validation
Conception générale
Tests d'intégration
Conception détaillée
Tests unitaires
Réalisation (codage)

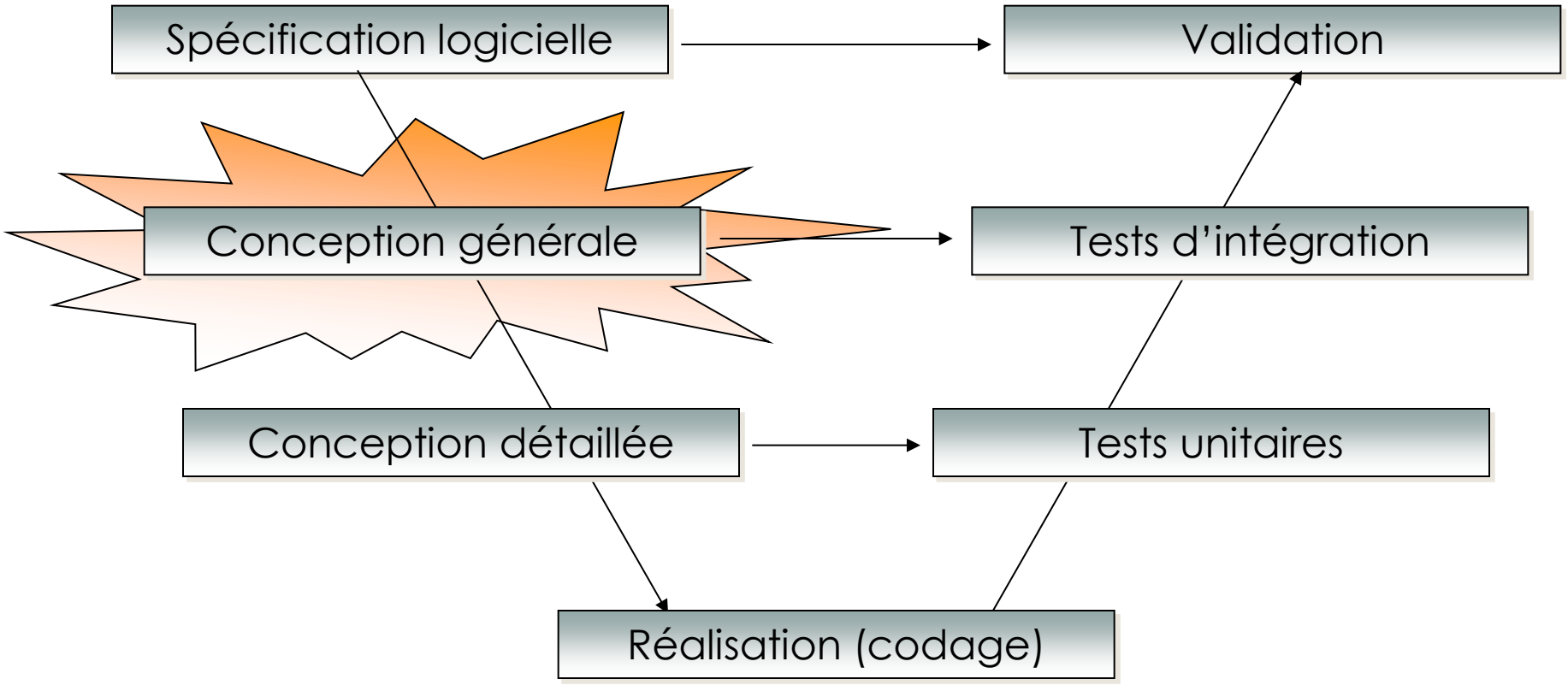
Validation

- But
  - Vérification de la conformité du logiciel développé aux exigences de la spécification.

Validation

- Principes (généraux)
  - Les exigences **fonctionnelles** et **temporelles** de la spécification doivent être vérifiées.
  - Tests effectués en « **boîte noire** ».
  - Peu de méthode formelle fiable.
  - Liste de tests à couvrir:
    - Tests en **mode nominal**
    - Tests aux **limites** & **réactions aux anomalies**
    - Tests de **performance**
    - Tests **fonctionnels combinés**

Conception générale



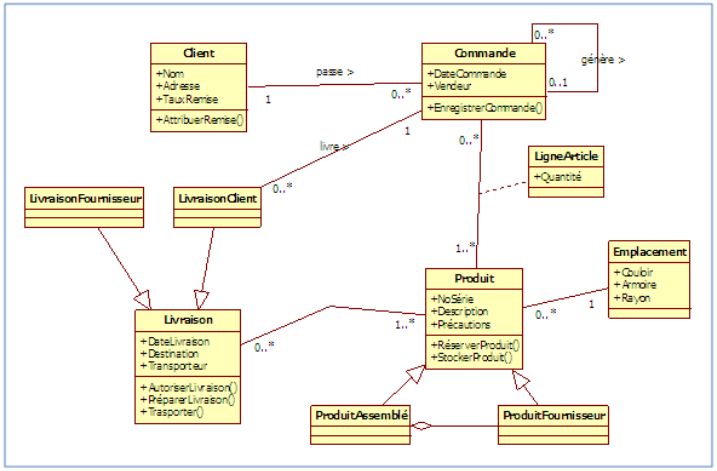
Conception générale

- But
  - Définition des architectures **statique** et **dynamique** du logiciel

Conception générale

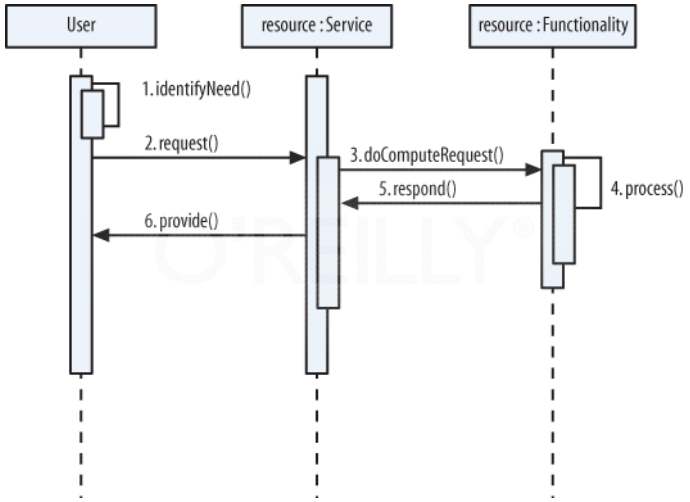
- Principes (1/2)

  - **<u>Architecture statique</u>**:
    - Découpage des traitements en **modules fonctionnels**.
    - Identification des **flux de données**.
    - Respects de **règles de hiérarchisation** (architecture en couches)



Conception générale

- Principes (2/2)
  - **Architecture dynamique**:
    - Prise en compte des **contraintes temps réel**.
    - Identification des **traitements immédiats et différés**.
    - **Répartition temporelle** des traitements.
    - Répartition des traitements faits **sous interruption**



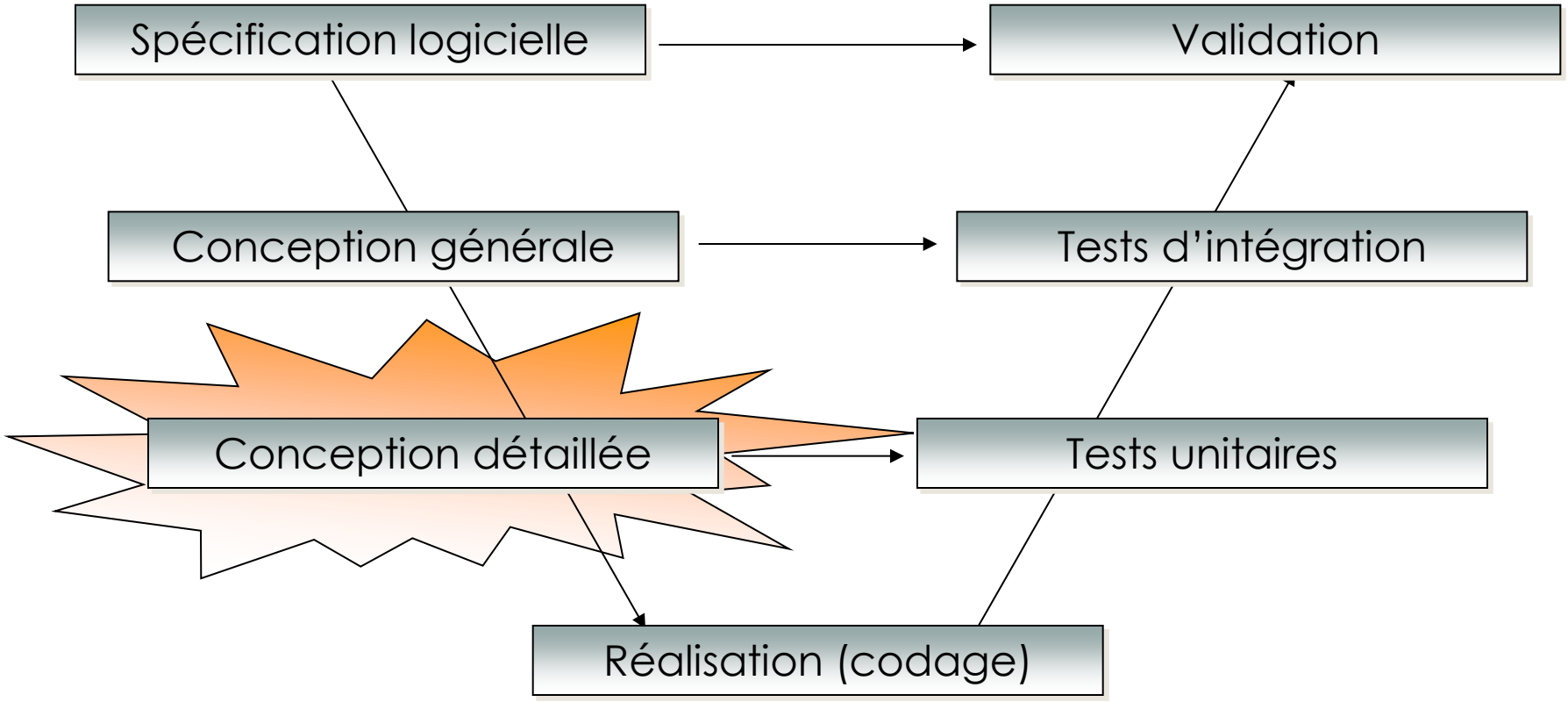
Conception générale

- Conseils
  - Les traitements sont des actions: les identifier à l'aide de verbes.
  - L'étape de conception doit permettre de lever toute ambiguïté de spécification.

Conception générale

- Pièges à éviter
  - Oubli d'exigence: toutes les exigences de la spécification doivent être prises en compte au niveau de la conception.

Conception détaillée



Conception détaillée

- But
  - Description des **algorithmes**, **structures de données** et **structures de contrôle** du logiciel.

Conception détaillée

- Principes
  - **Algorithmes**: pseudo-code, représentations graphiques
  - **Structures de données**: précision, représentation numérique, visibilité.
  - **Structures de contrôle**: Définition des automates / sous-automates

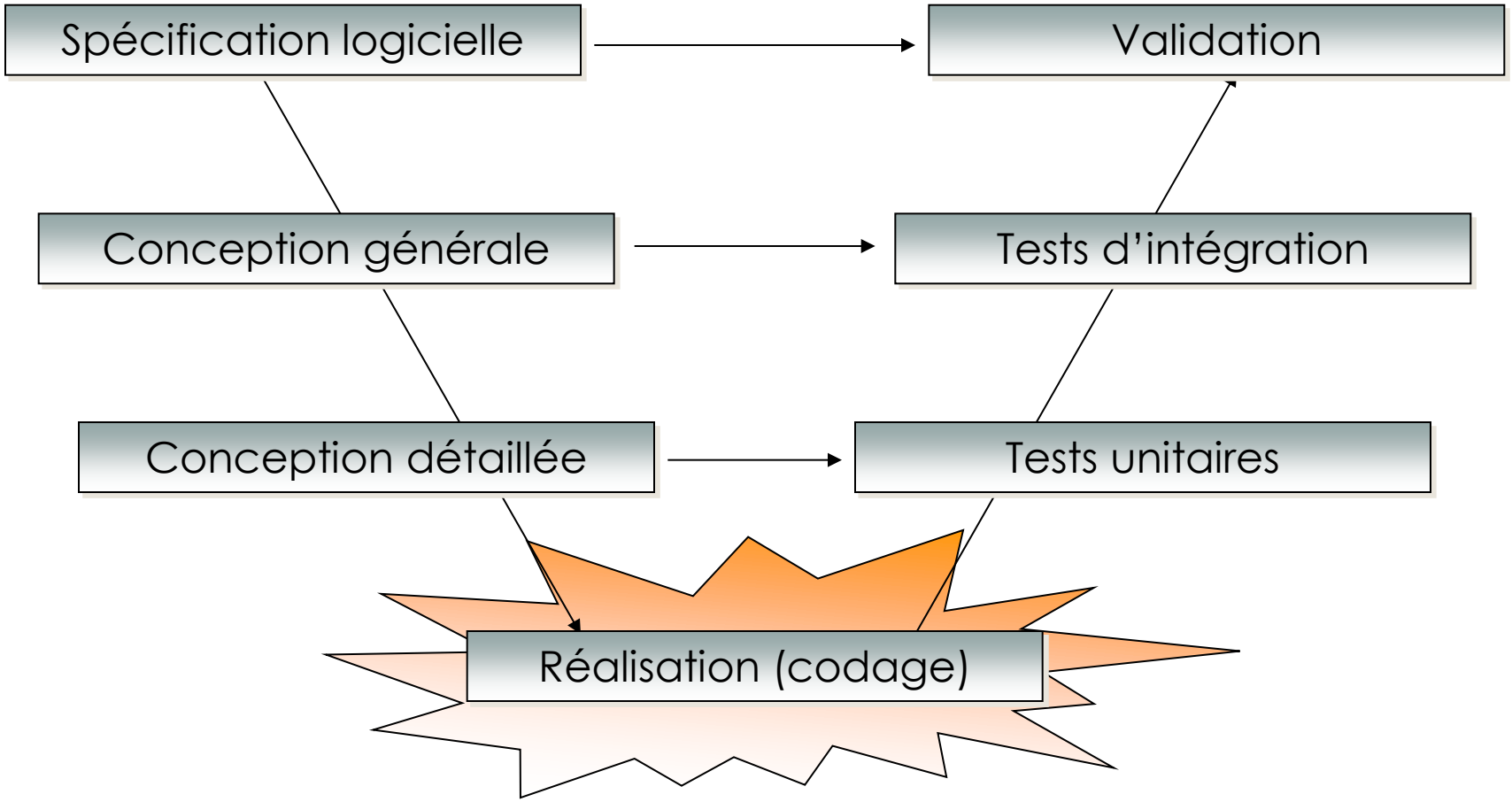
Conception détaillée

- Conseils
  - Savoir **adapter la représentation**: le choix du mode de représentation (pseudo-code, organigramme, machine d'états).
  - **Identifier et anticiper** les « problèmes » à venir:
    - **Taille** de code
    - **Utilisation mémoire**
    - **Charge CPU**

Conception détaillée

- Pièges à éviter
  - Dimensionnement des variables.
  - Oublier la gestion des cas d'erreur (→ modes dégradés).

Codage


Spécification logicielle
Validation
Conception générale
Tests d'intégration
Conception détaillée
Tests unitaires
Réalisation (codage)

Codage

- But
  - Traduction de la conception détaillée en langage logiciel (C, C++, assembleur…)

Codage

- Principes
  - Généralement regroupés dans un « guide »
    - Règles de nommage
    - Restrictions

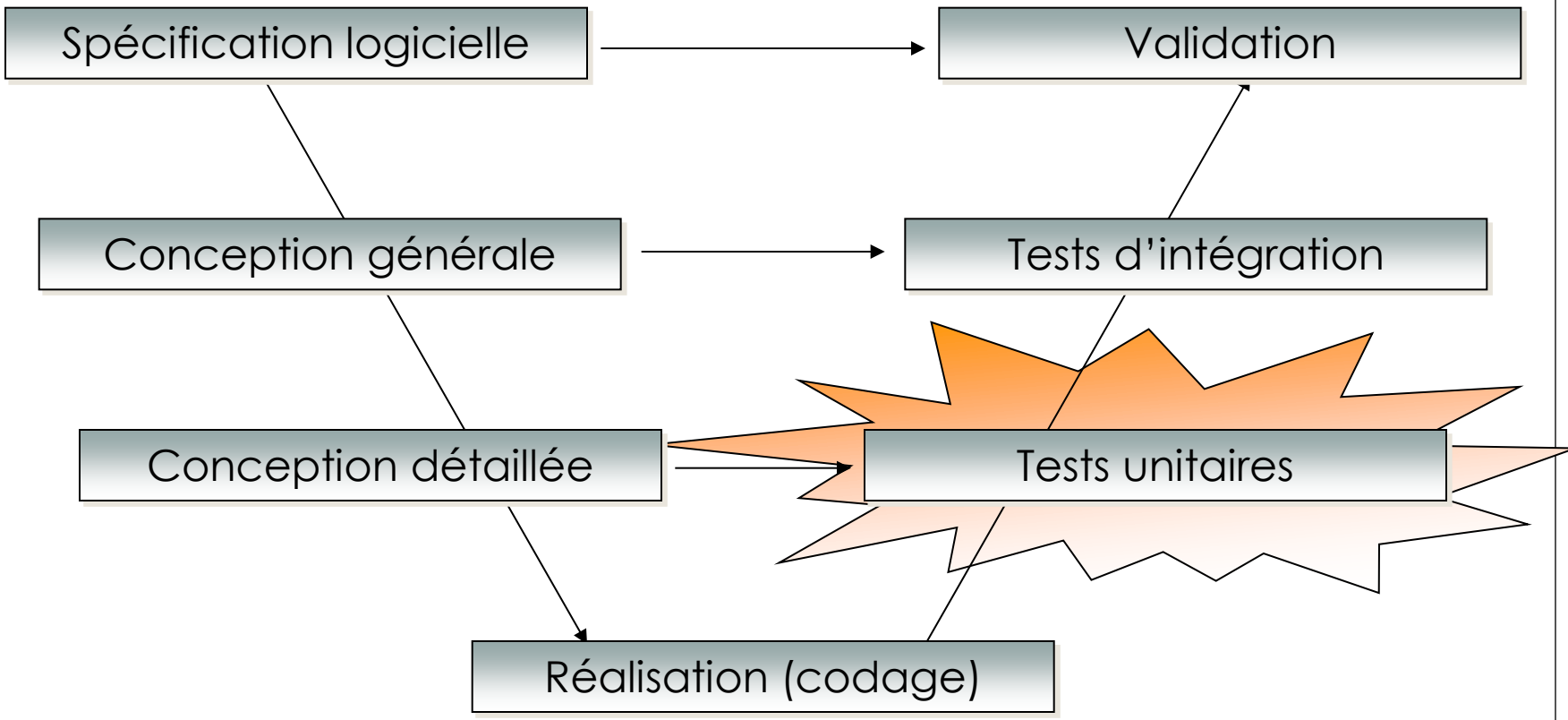
Codage

- Conseils
  - Partir de la conception détaillée et procéder « par partie » en soignant la gestion des fichiers .c et .h (fonctions et variables exportées ou non).

Codage

- Pièges à éviter
  - Se lancer dans le codage sans avoir fait de conception.
  - Négliger la **qualité** du codage.

Tests unitaires

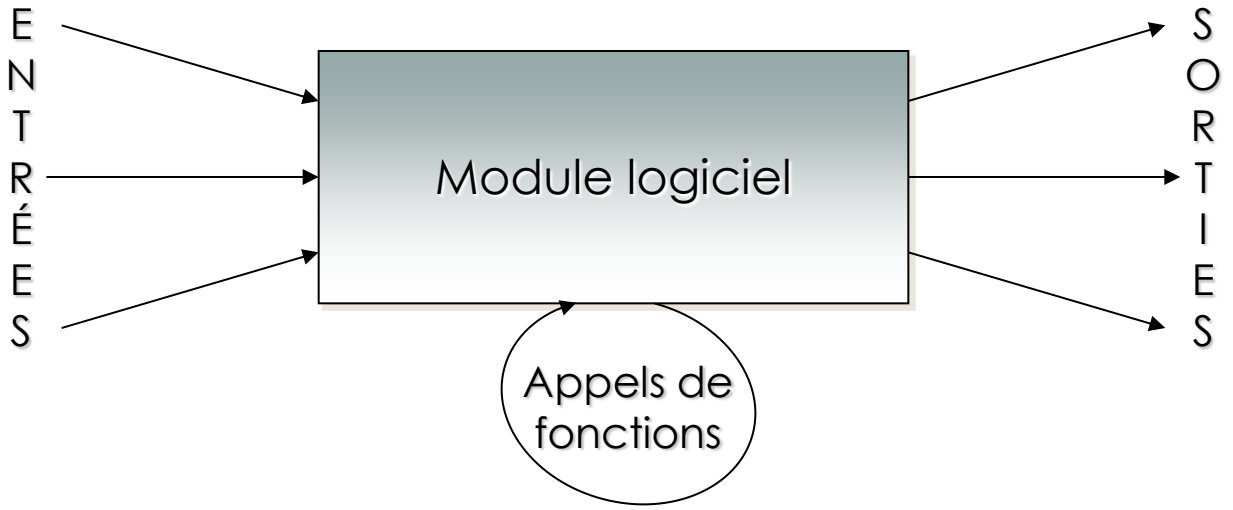


Tests unitaires

- But
  - Vérification de la conformité du code implémenté par rapport à la conception détaillée.

Tests unitaires

- Principes
  - S'appliquent à **chaque module** logiciel.
  - Permettent de **fiabiliser le processus d'intégration**.
  - S'assurer que le comportement de chaque module logiciel correspond à la **description qui en est faite dans la conception détaillée**.



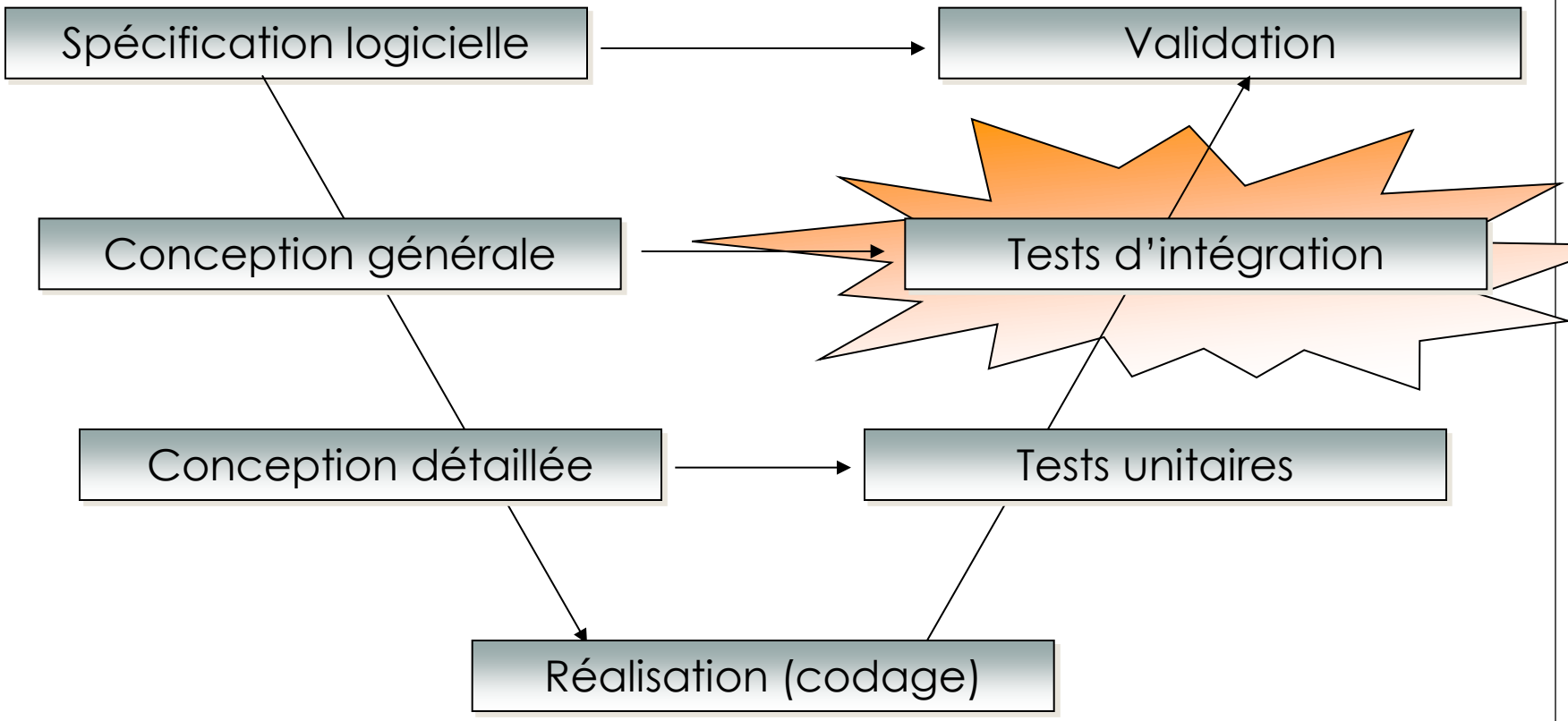
Tests unitaires

- Conseils
  - **Automatiser** les tests afin qu'ils puissent facilement être reproduits lors des évolutions.
  - Tester les modules avec un **effort proportionnel à la complexité**.
  - Valider **intégralement** les **interfaces**.
  - Tester les **cas nominaux, aux limites et hors limites**, ainsi que la **robustesse**.
  - Existence:
    - Normes définissant le niveau des tests à effectuer (ex: DO178C dans l'aéronautique).
    - Outils permettant de formaliser les tests.

Tests unitaires

- Pièges à éviter
  - Omettre les tests « aux limites » et en bordure des limites.
  - Omettre certaines branches d'exécution (notamment les cas d'erreur).
  - Faire les tests unitaires par rapport au code (et pas par rapport à la conception)

Intégration



Intégration

- But
  - Maîtrise de la « **construction** » du logiciel global par **intégration successive de modules**.

Intégration

- Principes
  - Intégrer **progressivement** les modules **testés unitairement**, suivant **un ordre pré-établi** (vérifier le bon comportement après assemblage).
  - Vérifier le **bon fonctionnement des interfaces**.
  - Vérifier les **exigences temporelles** du module intégré et des modules déjà intégrés.
  - **Surveiller les métriques** (empreinte mémoire et charge CPU).

Intégration

- Conseils
  - Procéder progressivement.
  - S'assurer régulièrement de l'impact de l'intégration du nouveau module sur l'ensemble du logiciel existant.
  - Optimiser l'utilisation des ressources.

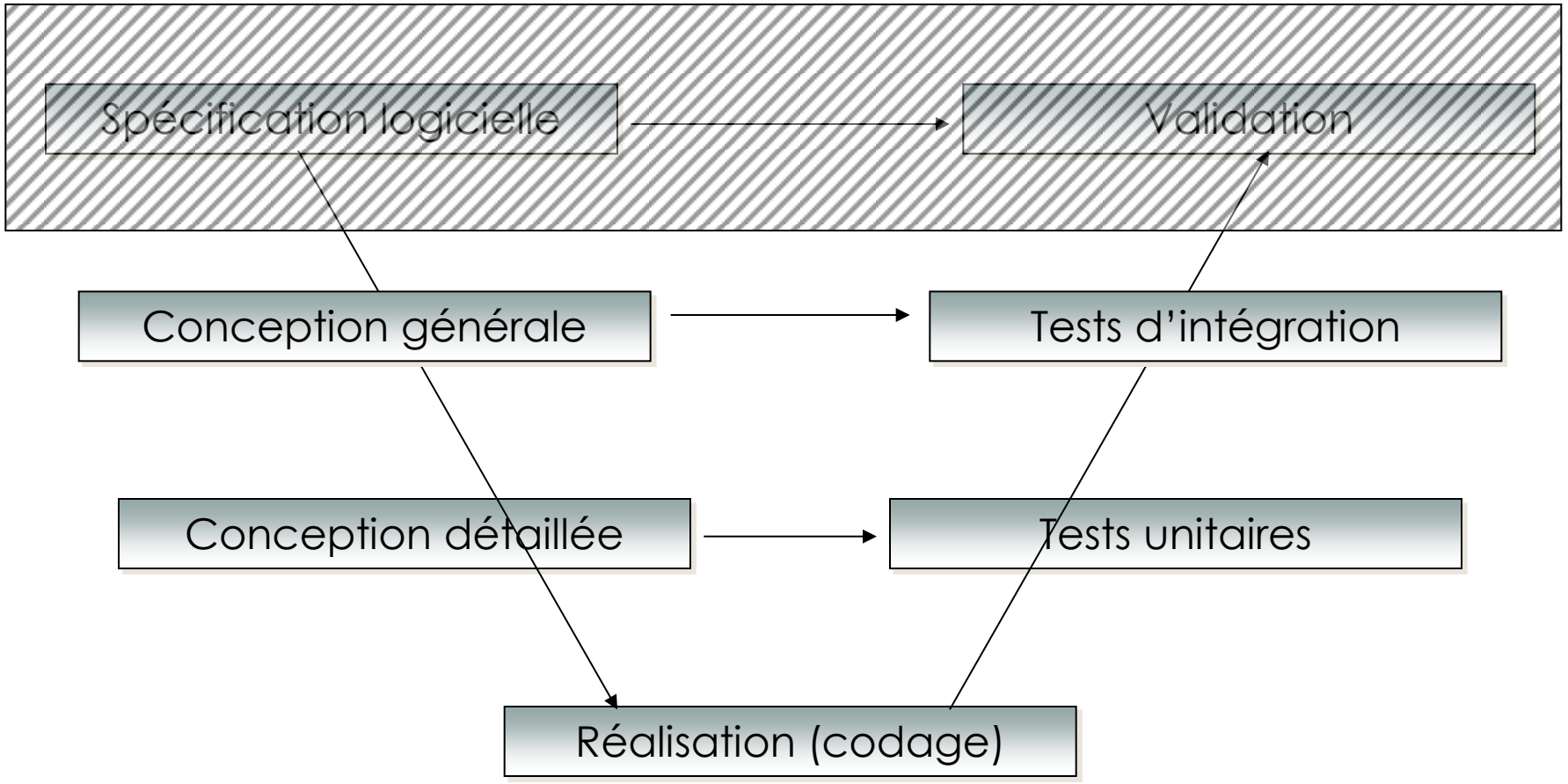
Intégration

- Pièges à éviter
  - Attention à la régression.
  - Ne pas faire suffisamment de tests entre chaque intégration.

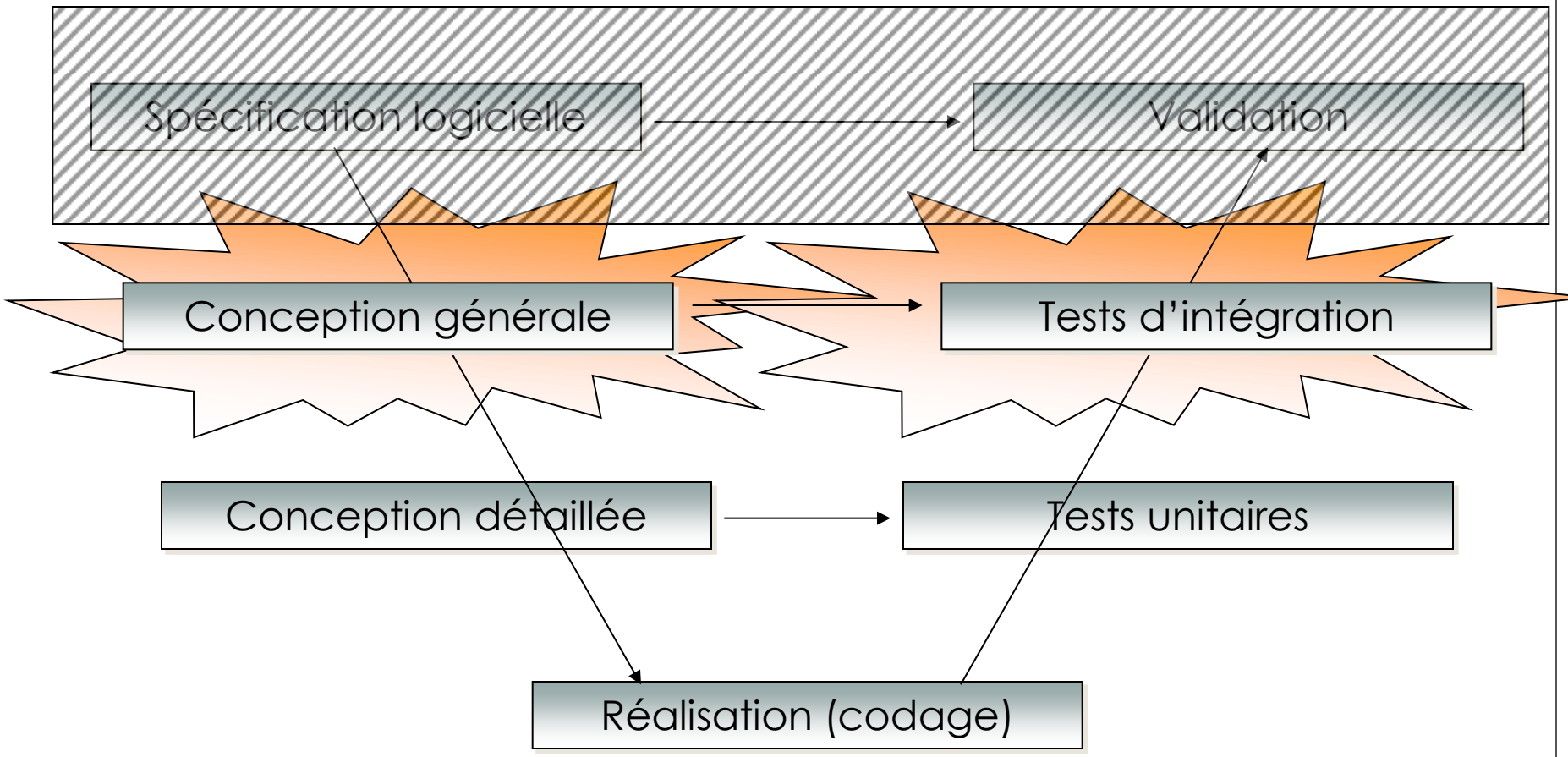
Déroulement de la présentation

- La problématique du Logiciel embarqué
- Les différents Cycles de développement logiciels
- La qualité au rendez-vous de chaque étape : dans le bas du cycle en V
- Rentrons dans les détails, MISRA
- Règles de codages C-ANSI
- Modèles de processus SPICE et CMMI
- AUTOSAR

Le bas du cycle en V



Conception générale – tests d'intégrations



Conception général – tests d'intégrations

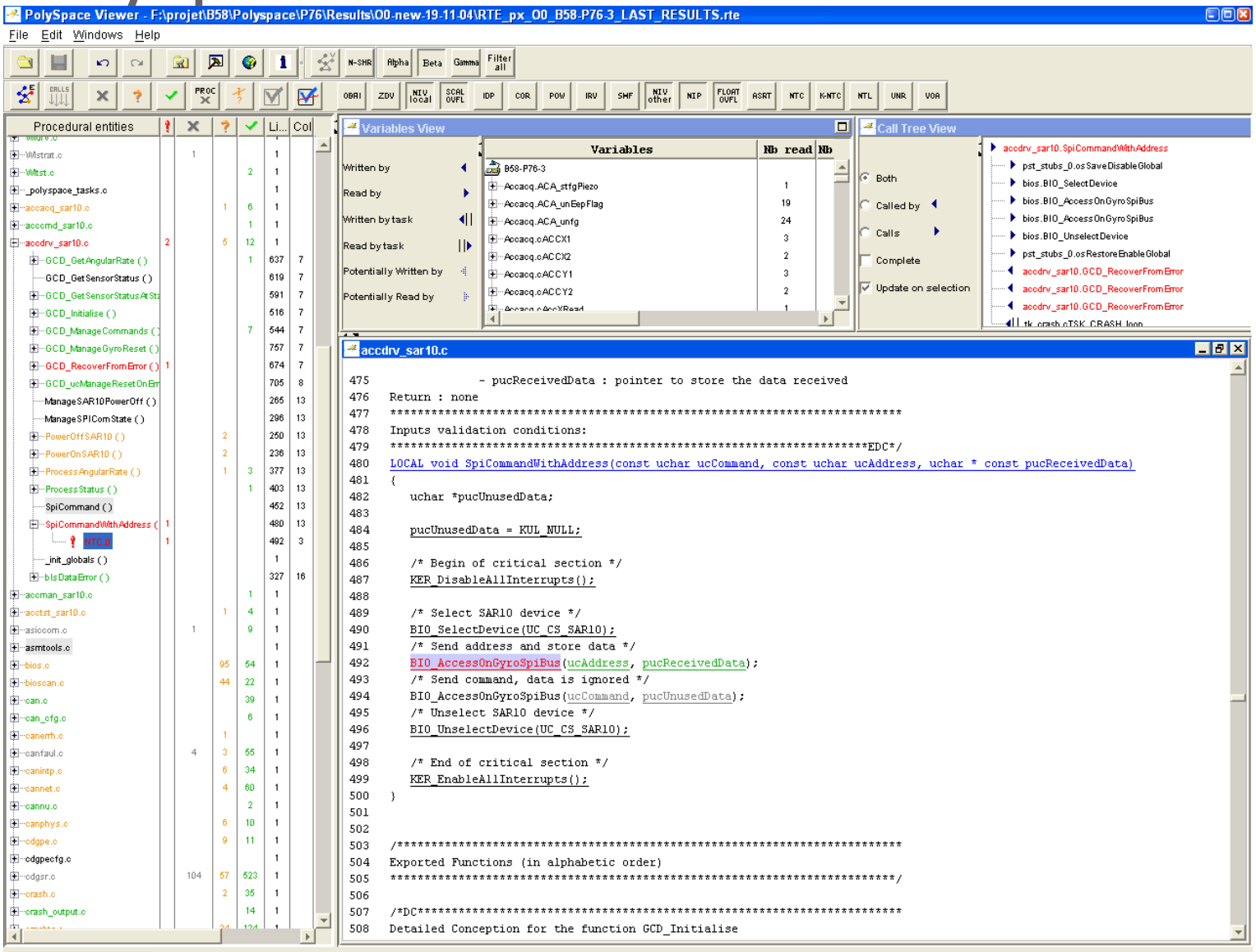
- **Règles de la qualité d'ingénierie logiciel** :
  - Stratégie de **test complète** (unitaire, intégration, validation)
  - **Revues** de spécification, codage et de validation, **relecture** croisé de code…
  - Procédure de **gestion de configuration**
  - Procédures de **gestion d'anomalies** détectées
- Attol de rational, Tessy de Hitex
- Polyspace
  - Analyseur de code détectant des erreurs lors de l'exécution (run time) :

Polyspace

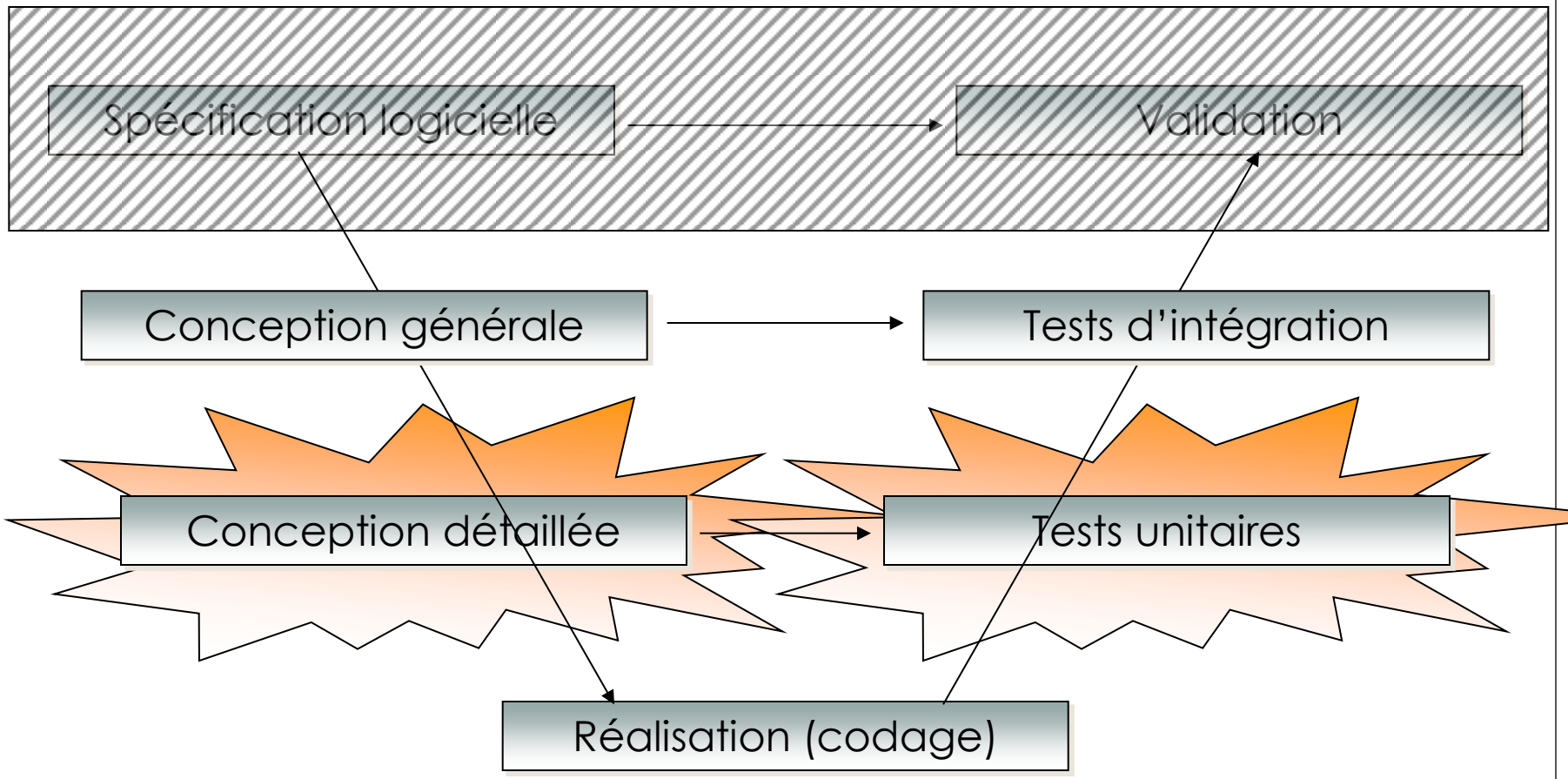
- Exemple d'erreurs run Times :

  - Division par zéro,
  - débordement,
  - racine carre négative,
  - accès en dehors de la taille d'un tableaux,
  - variables non initialisés,
  - Pointeur non initialisé
  - Pointeur déréférencé
  - conversion de type illégale,
  - Boucle sans fin…

Polyspace


PolySpace Viewer - F:\projet\B58\Polyspace\P76\Results\00-new-19-11-04\RTE_px_00_B58-P76-3_LAST_RESULTS.rte
File Edit Windows Help
Procedural entities
Variables View
Written by
Read by
Written by task
Read by task
Potentially Written by
Potentially Read by
Variables
Nb read
Call Tree View
Both
Called by
Calls
Complete
Update on selection
accdrv_sar10.c
B58-P76-3
Source file: cdgsr.c
cdgsr
Line: 1
Column: unknown

Conception détaillé – tests unitaires



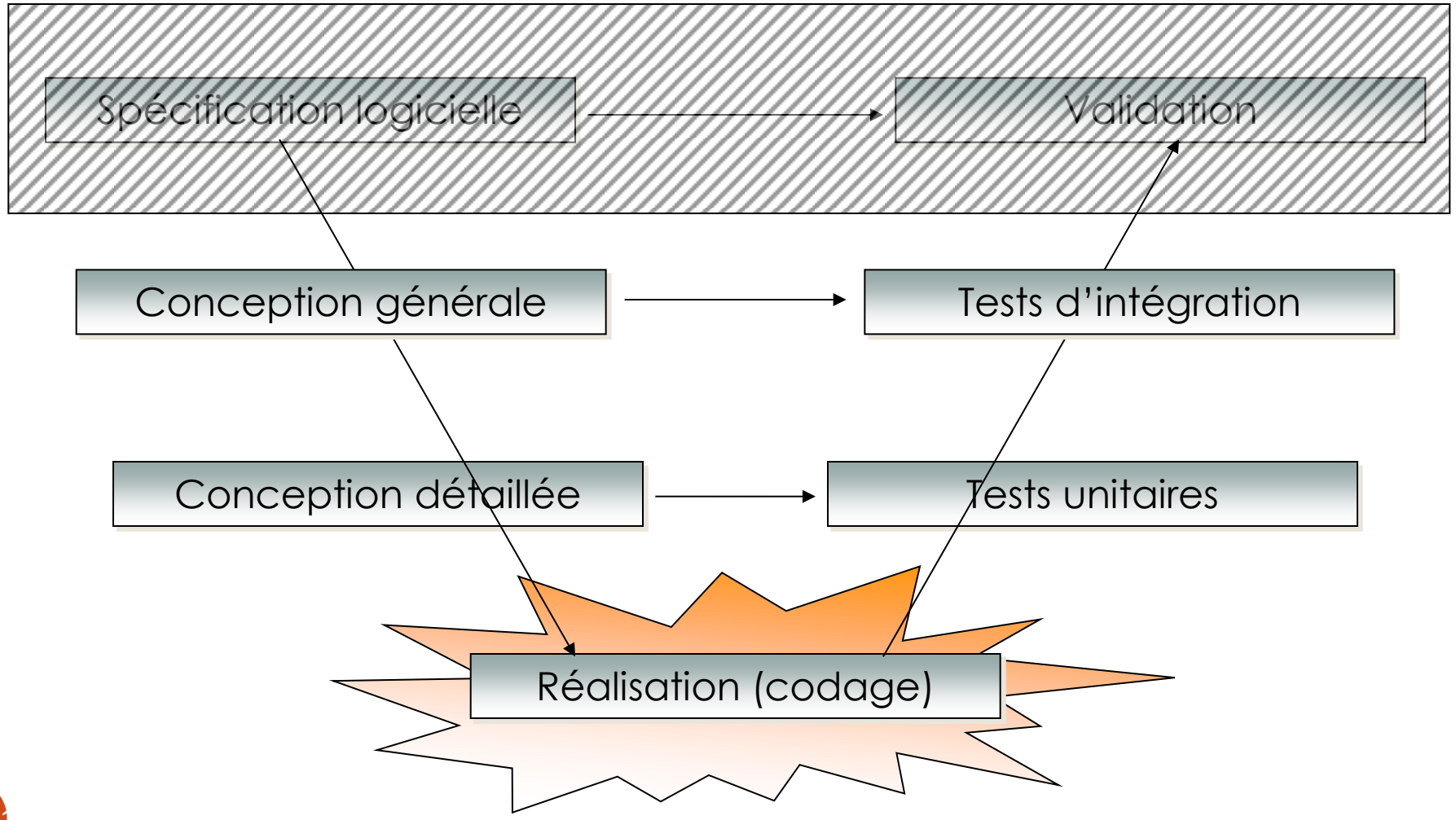
Conception détaillé – tests unitaires

- **Règles de conception du code** :
  - **Découpage modulaire** en fonction du logiciel (ex méthode SA-RT)
  - **limiter les interfaces au minimum entre les modules**, au niveau des appels et au niveau des données
    - limiter les variables globales
    - Des règles de manipulation sécuritaires de celle-ci (utilisation de sémaphores, recopies intermédiaires)
    - Une seule entrée et une seule sortie par fonction

Conception détaillé – tests unitaires

- **Règles de conception du code** :
  - **Fixer des règles concernant la gestion des erreurs** (les piéger par catégories, par endroit dans le programme…)
  - **Toute fonction doit rendre un code d'erreur en envisagent tous les cas** (débordement, valeurs non prévues…)

Codage


Spécification logicielle
Validation
Conception générale
Tests d'intégration
Conception détaillée
Tests unitaires
Réalisation (codage)

Codage

- Classification des erreurs de programmation
  - **Faute de programmation**
    - Mauvais **typage de variable**
    - **incompréhension de l'algorithme** par le développeur…
  - **Mauvaise compréhension du langage** c par le programmateur / Écart entre l'interprétation du compilateur et celle du programmeur :
    - L'ordre d'évaluation des opérateurs d'une expression, problème de portabilité d'un copilo à un autre
  - **Bugs du compilateur** :
    - Erreur résiduelle similaire à celle de tout logiciel
    - Utilisation d'un C non standard
  - **Erreurs lors de l'exécution** (run time)

Codage

- règles de codage
  - C ANSI, ISO 9899
  - MISRA (Motor industry software reliability association)
  - instructions locales…

Langage c

- Historique :
  - Brian Kernighan, Denis Ritchie et Ken Thompson :
    - mis au point par Thompson du langage B (pour *Unix*, 1969-1972)
    - Amélioration de ce dernier par Ritchie pour devenir le langage C (*Unix*, 1973)
    - En 1978, publication de *The C Programming Language* par Brian W. Kernighan et Denis M. Ritchie

Langage c

- C ANSI et iso 9899 :
  - décembre 1989 : le standard ANSI X3.159-1989 (ou C89) (l'*American National Standard Institut*)
  - 1990 : adoption par L'*International organization for standardization* (iso) de ce standard sous le nom de ISO/IEC 9899:1990 (ou C90).
  - décembre 1999 : ISO/IEC 9899:1999, (C99) . Revision tous les 5 ans

MISRA

- MISRA (Motor industry software reliability association) :
  - Règles de programmation de logiciels embarqués pour des systèmes sensibles à la sécurité
  - l'utilisation cohérente de types et méthodes
  - un contrôle des limites
  - un comportement reproductible du code
  -
  - portabilité

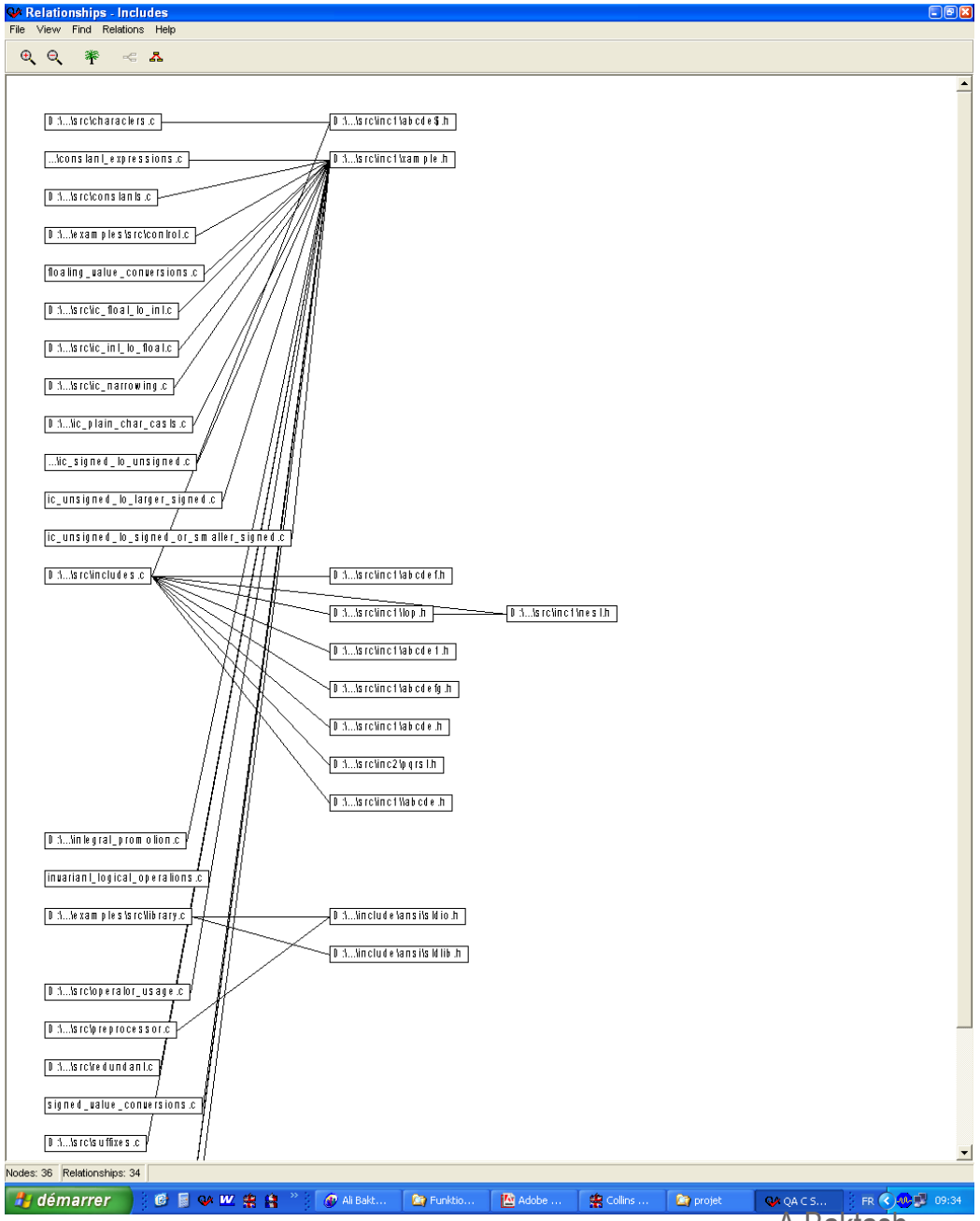
QAC

- Analyseur de code statique
  - Vérifications des règles MISRA et prés de 1000 autre règles
    - Erreurs d'implémentation
    - Pratiques de programmation dangereuse
    - Programmes non maintenable et/ou portable
  - Mesure de metrics
    - Complexité cyclomatique,
    - nombre de ligne de code,
    - nombre d'appel des fonctions,…
  - Visualisation de structure de code :
    - Liens entre modules
    - Entre les fonctions

QAC

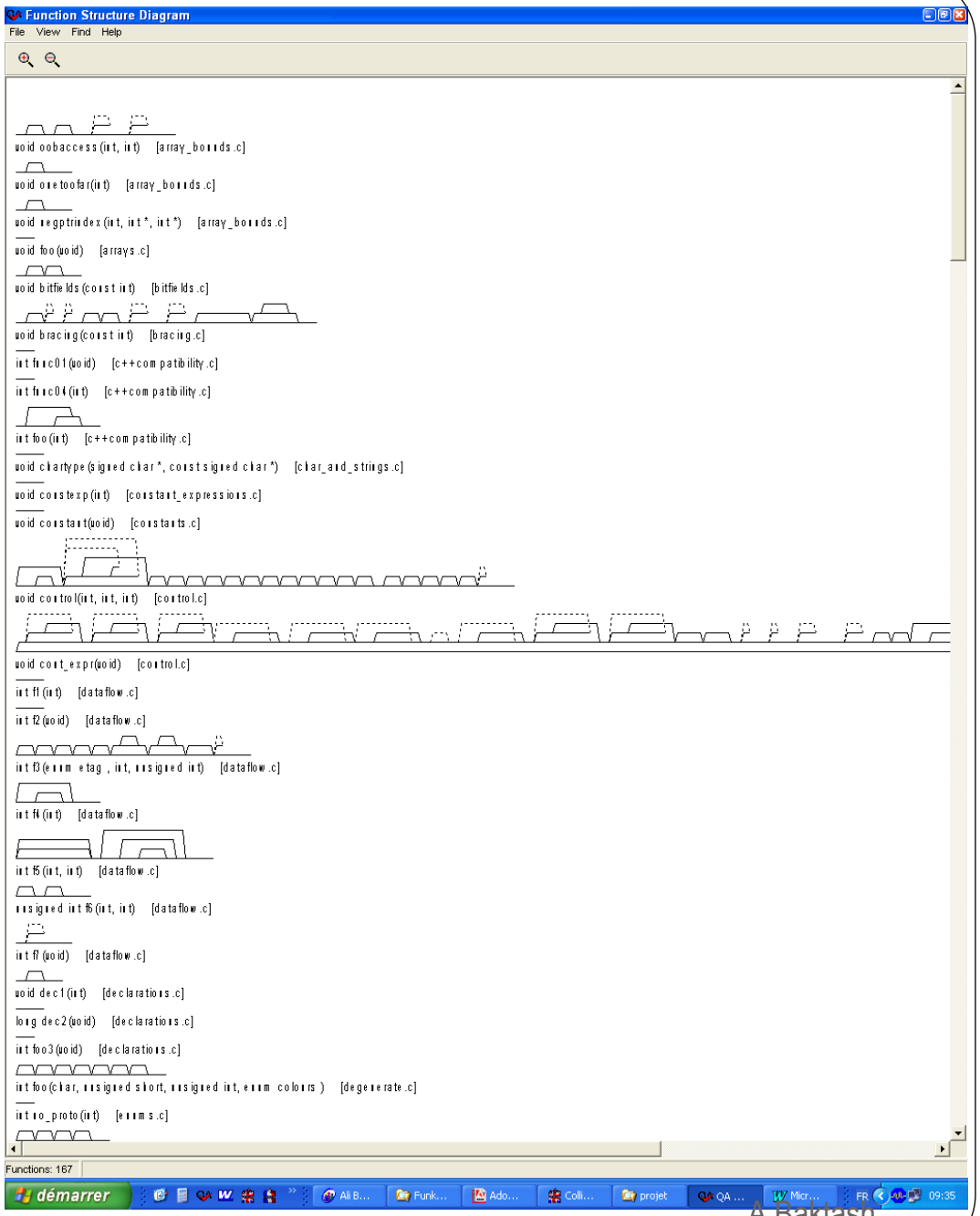
➔ Visualisation de structure de code :

- Liens entre modules
- Entre les fonctions



QAC

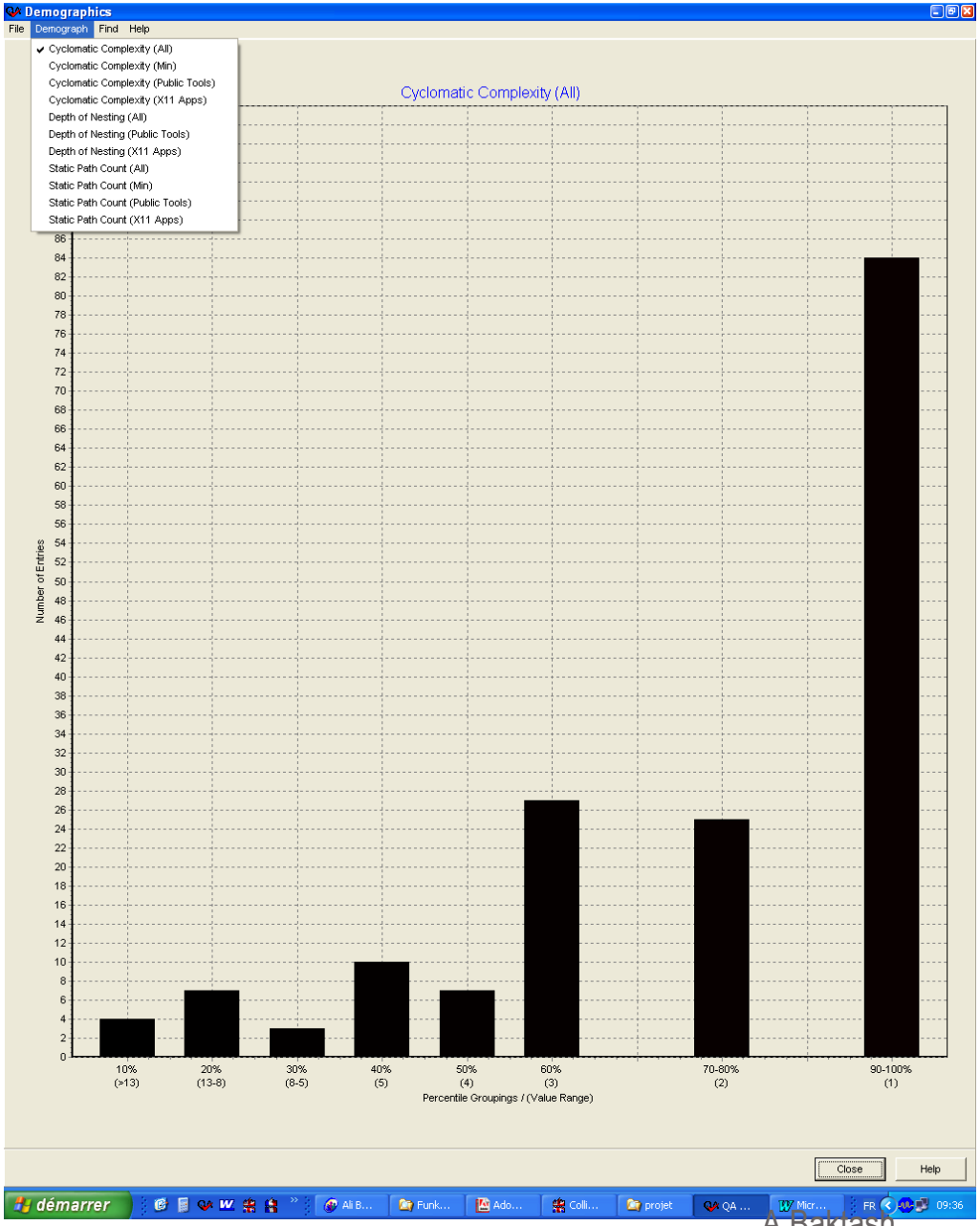
➔ Visualisation de structure de code :



QAC

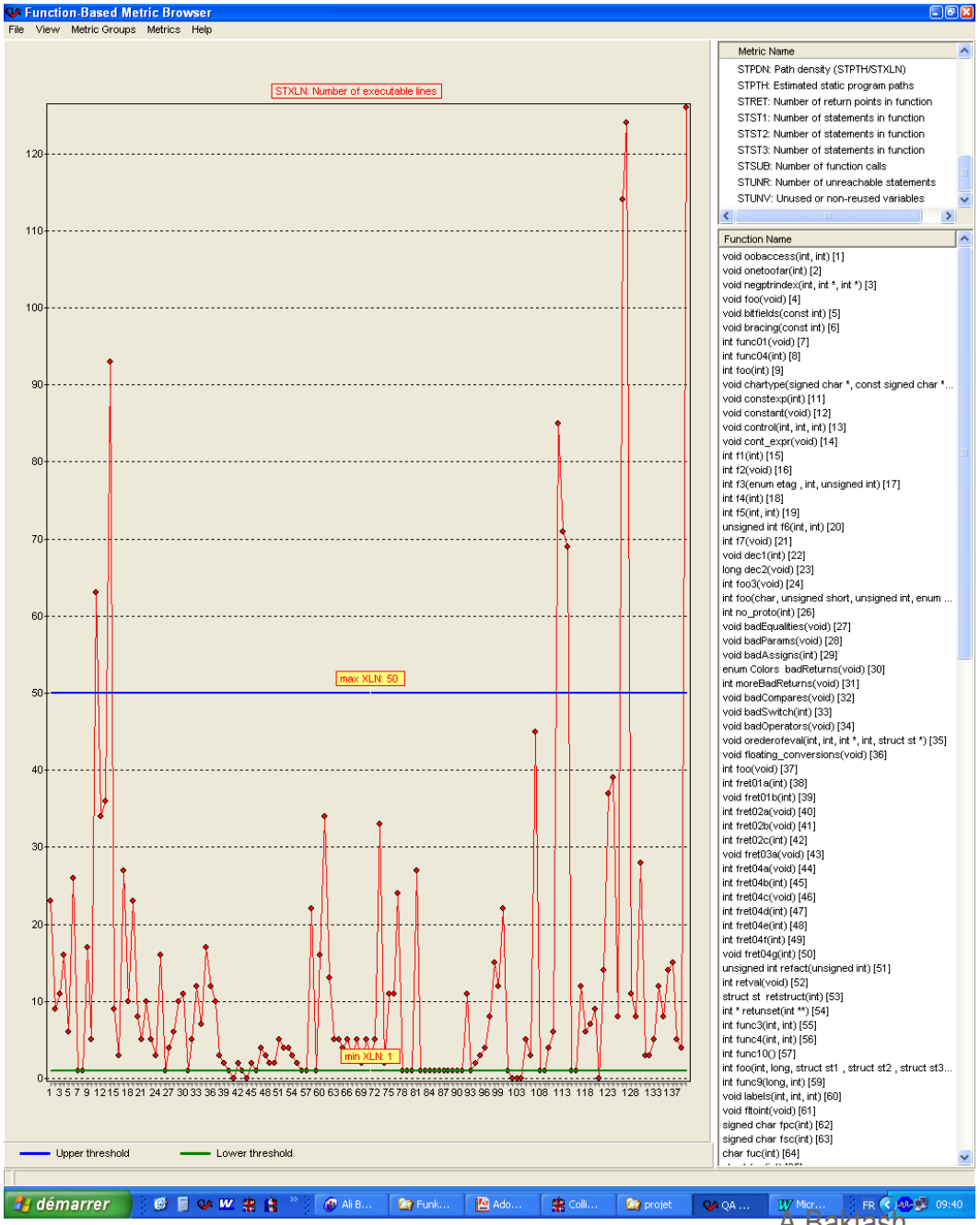
➔ Mesure de metrics

- Complexité cyclomatique,
- nombre de ligne de code,
- nombre d'appel des fonctions,…



QAC

➔ Mesure de metrics



Déroulement de la présentation

- La problématique du Logiciel embarqué
- Les différents Cycles de développement logiciels
- La qualité au rendez-vous de chaque étape : dans le bas du cycle en V
- Rentrons dans les détails, MISRA
- Règles de codages C-ANSI
- Modèles de processus SPICE et CMMI
- AUTOSAR

Programmation défensive

- Prévenir et se protéger des défaillance **avant qu'ils surviennent**,

- Prévoir tout cas de figure de défaillance même (et surtout)des **cas paraissant impossible**

Programmation défensive

- Origine des défaillances logicielles
  - Dégradation du comportement logiciel dues aux **perturbations électriques** externes
  - Dysfonctionnement du **temps réel**
  - Problèmes liées au **microcontrôleur**
  - **Ignorance de bonnes règles** concernant la production du logiciel

Programmation défensive

- Dégradation du comportement logiciel dû aux perturbations électriques externes
  - Perturbations :
    - Interférences électromagnétiques,
    - influences électrostatiques
    - perturbations électriques …
  - Effets typiques :
    - Le déroutement de code
    - altération du contenue de l'EEPROM
    - l'altération de la RAM et des registres
    - l'altération de la ROM/flash

Programmation défensive

- Solutions Matériel ou logiciel pour minimiser ces effets :
  - Déroutement de code :
    - **Compléter l'espace mémoire non utilisé** en ROM par des instructions redirigeant vers une instruction rétablissant un état défini du micro
    - Prévoir un **WatchDog**
    - Pour un micro offrant plusieurs modes de fonctionnement (normal, faible conso, veille) s'assurer que le **WD est présent pour chacun des modes**

Programmation défensive

- Dysfonctionnement du temps réel
  - Problème de conception :
    - La charge du microcontrôleur dans le **pire cas**
    - Prévoir Un **temps max** pendant laquelle une tache garde la main
    - Vérifier **l'appel** et **l'ordre d'appel des taches** (spécialement important pour des taches critiques)

Programmation défensive

- information sur le réseau (ex vitesse de véhicule):
  - Le **producteur doit garantir** la justesse et la pertinence de l'information
  - Les **filtrage** de l'information sont à la charge des **consommateur**, fonction du besoin de chacun

Programmation défensive

- Problèmes liées au microcontrôleur
  - gestion des entrées/sorties :
    - Vérifier toutes les **E/S programmable** par registre sont **initialisés** (y compris celles non utilisés)
    - Vérifier que les **E sont filtrées** (en valeur, en cohérence …)
  - Spécification des interfaces du micro :
    - Ne pas concevoir de stratégie s'appuyant sur des **comportement non spécifié du micro** (prudence aux spécifications préliminaires

Déroulement de la présentation

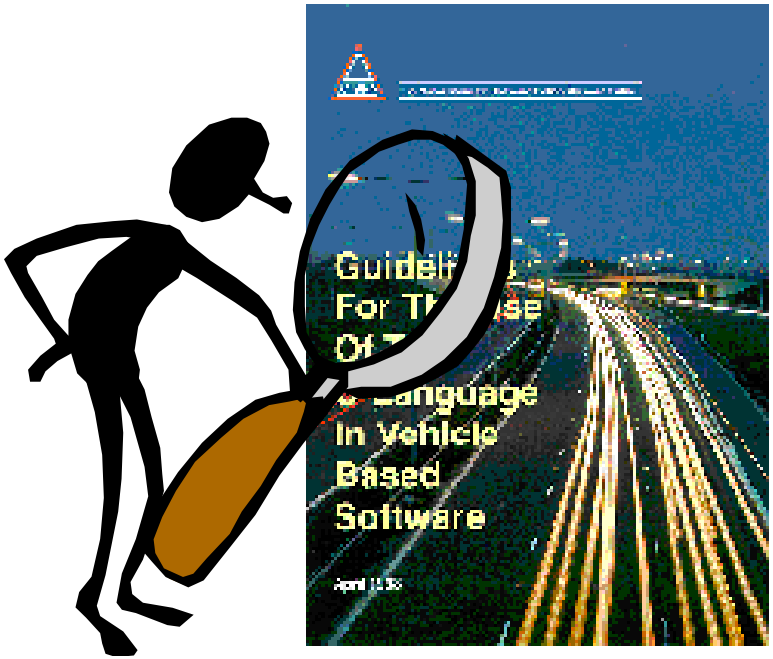
- La problématique du Logiciel embarqué
- Les différents Cycles de développement logiciels
- La qualité au rendez-vous de chaque étape

Rentrons dans les détails, MISRA

- Règles de codages C-ANSI
- Modèles de processus SPICE et CMMI
- AUTOSAR

Rentrons dans les détails…

- …des règles MISRA
  - 127 règles réparties dans 17 catégories



Rentrons dans les détails…

Rule 10 (comments)	Rule 13 (types)	Rule 14 (types)	Rule 21 (declarations)	Rule 32 (initialisations)
Rule 33 (operators)	Rule 36 (operators)	Rule 43 (conversions)	Rule 46 (expressions)	Rule 47 (expressions)
Rule 49 (expressions)	Rule 52 (control flow)	Rule 59 (control flow)	Rule 60 (control flow)	Rule 62 (control flow)
Rule 67 (control flow)	Rule 69 (functions)	Rule 70 (functions)	Rule 81 (functions)	Rule 82 (functions)
Rule 86 (functions)	Rule 92 (preprocessor)	Rule 96 (preprocessor)	Rule 101 (pointers & arrays)	Rule 106 (pointers & arrays)

Rule 10

Sections of code should not be 'commented out'.

- Where it is required for sections of source code to not be compiled then this should be achieved by use of conditional compilation (e.g. #if or #ifdef constructs).

Rule 13

The basic types of char, int, long, float and double should not be used, but specific-length equivalents should be typedef'd for the specific compiler, and these type names used in the code.

```
int main( int argc, char **argv )       /* MISRA Violation */
{
   S32   I = 1;
   int   j;                         /* MISRA Violation */
   SC    c;
   char  d;                          /* MISRA Violation */
   UCHAR f;
   float g;                         /* MISRA Violation */
   g = (float)i;                     /* MISRA Violation */
   return(0);
}
```

Rule 14

The type char shall always be declared as unsigned char or signed char.

```
typedef char  CH;                /* MISRA Violation */
static S32 test( void )
{
   CH          *a = "AAAA";
   char        *b = "Test";      /* MISRA Violation */
   unsigned char *c;
   UC          *d;
   signed char   *e;
   SC          *f;
   return(0);
}
```

Rule 21

Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

```
static S32 var;
static S32 test1( void )
{
  return(var);
}
static S32 test2( void )
{
  S32 var = 0;          /*  Misra violation  */
  return(var);
}
static S32 test3( void )
{
  S32 ret = var;
  return(ret);
}
```

Rule 32

In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.

```
#define VAL 4294
static S32 test( void )
{
   enum game    { tennis = 2, cricket, golf, hurling };       /* OK */
   enum town    { London, Paris, New_York };                  /* OK */
   enum country {Germany = 1, Italy = 2, Australia = 3};      /* OK */
   enum color   { red, blue = VAL, green };                   /* MISRA Violation */
   enum cars    { BMW, Mercedes, Ford = VAL };                /* MISRA Violation  */
   return(0);
}
```

Rule 33

The right hand operand of a && or || operator shall not contain side effects

```
extern S16 foo(S16 x);
static S16 test( S16 a, S16 b )
{
   S16 r    = 0;
   if ( (a > 1) && ( b++ < 0) )        /* MISRA Violation */
   {
      r = 1;
   }
   if ( (b > 0) || ( foo(a) != 0 ) )    /* MISRA Violation */
   {
      r = 1;
   }
   r = (( a > 0 ) ? (b) : (b++));       /* MISRA Violation */
   return r;
}
```

Rule 36

Logical operators should not be confused with bitwise operators.

- Logical operators:

  &&
  ||
  !

- Bitwise operators

  &
  |
  ~

Rule 43

Implicit casts which may result in a loss of information shall not be used.

```
static S32 test( S16 x, S32 y )
{
   x = y;            /* MISRA Violation */
   x = (S16) y;      /* OK              */
   y = y + x;        /* OK              */
   x = x + y;        /* MISRA Violation */
   return(0);
}
```

Rule 46

The value of any expression shall be independent of the order of its evaluation.

```
extern S32 f( S32, S32 );
extern S32 g( S32 );
static S32 test( void )
{
   S32 i = 0;
   S32 a[2]={1,1};
   S32 x;
   volatile S32 k = 0;
   x = a[i] + i++;          /* MISRA Violation */
   x = f( i++, i++ );       /* MISRA Violation */
   x = g( i++ ) + g( i++ );  /* MISRA Violation */
   k = k;                   /* MISRA Violation */
   return(0);
}
```

Rule 47

No dependence should be placed on C's operator precedence rules in expressions.

```
static U32 test( U32 i, U32 j )
{
    U32 k = 0UL;
    k = 3UL * i + j * 2UL;
    k = 10UL + j & 1UL;    /* MISRA Violation */
    return(k);
}
```

Rule 49

Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

```
static S32 testf( S32 i )
{
   S32 r = 0;
   S32 j;
   if ( i )       /* MISRA Violation */
   {
      r = 1;
   }
   j = i ? 2 : 3;  /* MISRA Violation */
   if ( i + j )    /* MISRA Violation */
   {
      r = 2;
   }
   if ( !i )      /* MISRA Violation */
   {
      r = 3;
   }
   if ( i > j )    /* OK for relational operator (<, > <= or >=) */
   {
      r = 4;
   }
   if ( i = 1 )    /* MISRA Violation */
   {
     r = 5;
   }
   return(r);
}
```

Rule 52

There shall be no unreachable code.

```
static S16 test( void )
{
   S16 x = 0;
   S16 y = 0;
   if ( x == 0 )
   {
   }
   else
   {
      x = 0;
   }
   switch( x )
   {
      S16 z = 1;          /* MISRA Violation */
      y = 0;              /* MISRA Violation */
   case 0:
      y = 1;
      break;
   default:
      y = 0;
   }
   while (x > 10)
   {
      --x;
      break;
      ++y;                /* MISRA Violation */
   }
}
```

Rule 59

The statements forming the body of an if, else if, else, while, do ... while or for statement shall always be enclosed in braces.

```
static S16 test( S16 x )
{
   S16 n = 0;
   for (n = 0; n < 5; n++)
      x = 2;                                  /* MISRA Violation */
   if (n == 2)
      n = 3;                                  /* MISRA Violation */
   else
      n = 2;                                  /* MISRA Violation */
   while (n < 5)
      n++;                                    /* MISRA Violation */
   do
      n++;                                    /* MISRA Violation */
   while (n<9);
   return(x);
}
```

Rule 60

All if, else if constructs should contain a final else clause.

```
static S16 test( S16 n )
{
   S16 r = 0;
   if (n == 2)
   {
      r = 3;
   }
   else if (n == 3)        /* MISRA Violation */
   {
      r = 2;
   }
   return(r);
}
```

Rule 62

All switch statements should contain a final default clause.

```
static S32 test( S32 n )
{
   S32 x = 0;
   switch (n)
   {
   case 0:
      x = 0;
      break;
   case 1:
      x = 1;
      break;
   }          /* MISRA Violation */
   return(x);
}
```

Rule 67

Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop

```
static S16 test( S16 v )
{
   S16 n  = 0;
   static S16 si;
   for ( n = 0; n < 10; n++)
   {
      n++;                  /* MISRA Violation */
   }
   for ( si = 0; si < v; si++)
   {
     si += 2;               /* MISRA Violation */
     v++;                   /* MISRA Violation? */
   }
   return(0);
}
```

Rule 69

Functions with variable numbers of arguments shall not be used.

```
extern void myprintf(S32 n, ...);                    /* MISRA
   Violation */
static void fred(S32 x, ...);                /* MISRA Violation */
static S32 test( void )
{
    myprintf(2, "tom", "dick");
   fred(1, "harry");
    return(0);
}
static void fred(S32 x, ...)
{
}
```

Rule 70

Functions shall not call themselves, either directly or indirectly

```
static void f1(void);
static void f2(void);
static void f3(void);
static void f4(void);
static void f5(void);
static S32 test( void )
{
   f1();
   f5();
   return(0);
}
static void f1(void)
{
   f2();          /* MISRA Violation */
}
static void f2(void)
{
   f3();          /* MISRA Violation */
}
static void f3(void)
{
   f4();          /* MISRA Violation */
}
static void f4(void)
{
   f1();          /* MISRA Violation */
}
static void f5(void)
{
   f5();          /* MISRA Violation */
}
```

Rule 81

const qualification should be used on function parameters which are passed by reference, where it is intended that the function will not modify the parameter.

```
void myfunc( const SI_16 * myparam )
{
   *myparam = 2;   /* Attempt to modify value pointed to by myparam.
                      The const in the parameter type prevents this */
   return;
}
```

Rule 82

A function should have a single point of exit.

```
static S16 f1(S16 i)
{
    if (i > 0)
    {
        return(0);
    }
    else
    {
        return(1);                /* MISRA Violation */
    }
}
```

Rule 86

If a function returns error information, then that error information should be tested.

```
extern S32 f1(void);          /* MISRA Violation - from usage below */
static S32 test( void )
{
   S32 r;
   f1();                  /* MISRA Violation - flagged on prototype above */
   r = f1();
   return(r);
}
```

Rule 92

#undef should not be used.

```
#include "misra.h"
#define             L           0
#undef   L                      /* MISRA Violation */
static S32 test( void )
{
   return(0);
}
```

Rule 96

In the definition of a function-like macro the whole definition, and each instance of a parameter, shall be enclosed in parentheses.

```
#define  MAXCONST   5                    /* OK */
#define  MAX1(A,B)  (A)>(B)?(A):(B)       /* MISRA Violation */
#define  MAX2(A,B)  (A>B?A:B)             /* MISRA Violation */
#define  RMAX(A,B)  (((A)>(B))?(A):(B))    /* OK */
static void test( S32 i, S32 j )
{
   S32 k = MAXCONST;
   k = MAX1(i,j);
   k = RMAX(i,j);
}
```

Rule 101

Pointer arithmetic should not be used.

```
#define INDEX 3
static S32 test( SC buf[] )
{
   S32 n = 0;
   SC  *p;
   SC  *q;
   buf[INDEX]     = 1;                  /* OK             */
   *(buf + INDEX) = 1;                    /* MISRA VIOLATION */
   p = buf;
   while (*p != '\0')
   {
      n += *p;
      p++;                            /* OK             */
      p += 1;                          /* OK             */
      p += 2;                          /* MISRA VIOLATION */
      p = p + 1;                        /* OK             */
      p = p + 2;                        /* MISRA VIOLATION */
      p++;                            /* OK             */
   }
   q = buf;
   n = p - q;                          /* MISRA VIOLATION */
   return ( n );
}
```

Rule 106

The address of an object with automatic storage shall not be assigned to an object which may persist after the object has ceased to exist.

```
static S32 * test( void )
{
   static S32 * pc;
   S32 *p;
   S32 c = 0;
   {
      S32 i = 1;
      p = &i;              /* MISRA Violation */
   }
   pc = &c;                /* MISRA Violation */
   return(&c);             /* MISRA Violation */
}
```

Merci de votre attention…

Avez-vous des questions?

Contact:
ali.baktash@smile.fr
ali.baktash@autoliv.com

Présentation inspirée de documents rédigés par:
Christophe Brunschweiler, Smile
Alex Veret, Ingenico