UMR IRISA

Eléments de **Génie Logiciel**
Cours 6
Qualité logicielle

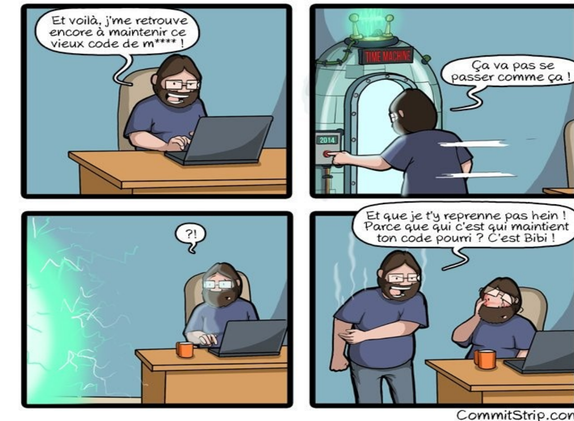
Pr. Jean-Marc Jézéquel
IRISA – Université de Rennes
Campus de Beaulieu
F-35042 Rennes Cedex
e-mail : jezequel@irisa.fr
http://people.irisa.fr/Jean-Marc.Jezequel
X @jmjezequel

istic Informatique Électronique

Université de Rennes

Université de Rennes

1 - Introduction



Université de Rennes

Qualité du process vs qualité du produit

- Qualité du process de fabrication
  - Ensemble de normes ISO 9001
  - Capability Maturity Model (CMM)
  - Etc.
- Qualité du produit = qualité du logiciel
  - Facteurs contribuant à la qualité du logiciel

Université de Rennes

Facteurs de qualité externes d'un logiciel (1/3)

- Validité
  - Aptitude à réaliser les tâches définies par sa spécification
    Spécifications informelles => objectif difficile à atteindre
- Robustesse
  - Aptitude à fonctionner même dans des conditions anormales
    Réaction non catastrophique à des situations non prévues par la spécification, forcément incomplète
- Extensibilité
  - Facilité d'adaptation d'un logiciel aux changements (correctifs ou évolutifs) de spécification
    éviter les logiciels château de cartes

Université de Rennes

Facteurs de qualité externes d'un logiciel (2/3)

- Réutilisabilité
  - Aptitude à être réutilisé en partie pour de nouvelles applications
    Cesser de réinventer, re-coder, et surtout, re-tester
- Compatibilité
  - Aptitude des logiciels à pouvoir être combinés entre eux
    Assemblage de briques de base
- Efficacité
  - Bonne utilisation des ressources du matériel: processeurs, mémoires, communications, énergie, etc.
    A ne pas confondre avec temps réel…

Université de Rennes

Facteurs de qualité externes d'un logiciel (3/3)

- Portabilité
  - Facilitée avec laquelle un produit logiciel peut-être adapté à différents environnements matériels et logiciels
    - *souvent incompatible avec efficacité optimale…*
- Vérifiabilité
  - Facilité de préparation des procédures de validation (tests), de recette et de certification
- Ergonomie
  - Facilité avec laquelle les utilisateurs d'un composant peuvent apprendre à l'utiliser et à en tirer le meilleur part
    documentation à jour (=> extraite du code : Javadoc)

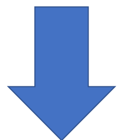
Université de Rennes

2 – Conception par Contrat

Université de Rennes

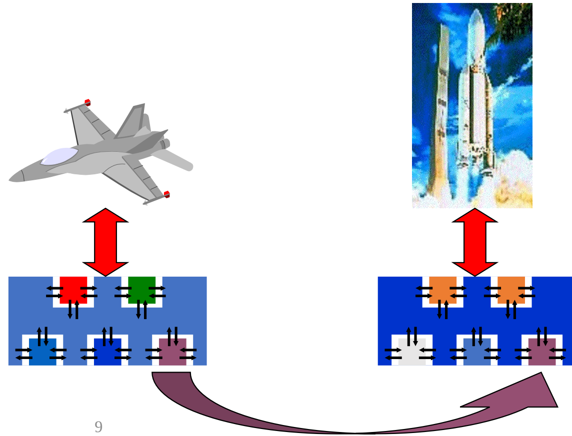
Principe de l'approche objet

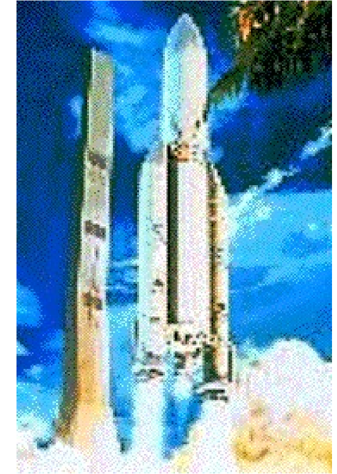
- Structurer les systèmes autour des objets
  - Plutôt qu'autour des fonctions
- Obtenir des systèmes modulaires et maintenables
  - Par assemblage de briques de base
  - Composition d'éléments simples pour obtenir des systèmes sophistiqués
- Favorise :
  - masquage d'information (abstraction)
  - encapsulation (facilite les modifications à portée locale)
  - dissociation interface/implantation=> composant réutilisab



Université de Rennes

Validité inter-composants : Peut-on (ré)-utiliser un composant?





Université de Rennes

Ariane 501 Vol de qualification
Kourou, ELA3 -- 4 Juin 1996,12:34 UT

- H0 -> H0+37s : nominal
- Dans SRI 2:
  - BH (Bias Horizontal) > 2^15
  - convert_double_to_int(BH) fails!
  - exception SRI -> crash SRI2 & 1
- OBC disoriented
  - Angle attaque > 20°,
  - charges aérodynamiques élevées
  - Séparation des boosters

Université de Rennes

Ariane 501 : Vol de qualification
Kourou, ELA3 -- 4 Juin 1996,12:34 UT

- H0 + 39s: auto-destruction (coût: 500M€)

Université de Rennes

Pourquoi ? (cf. IEEE Comp. 01/97)

- Pas une erreur de programmation
  - Non-protection de la conversion = décision de conception ~1980
- Pas une erreur de conception
  - Décision justifiée vs. trajectoire Ariane 4 et contraintes TR
- Problème au niveau du test d'intégration
  - Comme toujours, aurait pu être détecté…
  - Mais gigantesque espace de test vs. ressources limitées
  - En plus, SRI inutile à cette étape du vol!

Université de Rennes

Pourquoi? (cf. IEEE Computer 01/97)

- Réutilisation dans Ariane 5 d'un composant de Ariane 4 ayant une contrainte « cachée » !
  - Restriction du domaine de définition
    - Précondition : abs(BH) < 32768.0
  - Valide pour Ariane 4, mais plus pour Ariane 5

Université de Rennes

Spécification = contrat *entre un composant et ses clients*

- Dans la vie réelle, différents types de contrats
  - *Du « Contrat social » de Jean-Jacques Rousseau au "cash & carry"*
- De même, plusieurs types de contrats dans un monde réparti

Université de Rennes

Quatre niveaux de contrats logiciels

- Elémentaire (syntaxique)
  - le programme compile…
- Comportemental (fonctionnel)
  - pré et post conditions
- Synchronisations
  - e.g. *path expressions*, etc. [McHale]
- Qualité de service (quantitative)
  - Négociation dynamique possible

Cf. IEEE Computer July 1999

Université de Rennes

Représentation des contrats en UML avec OCL (Object Constraint Language)

- Typage par signature des méthodes insuffisant
  - besoin de pouvoir exprimer des restrictions
    - valeurs d 'entrées *et* de sorties
  - besoin de préciser la sémantique
    - ce que fait une méthode (le quoi) sans entrer dans le détail comment
      - Préserve le masquage d'information : Indépendance vis-à-vis de l'implantation
- Inspirée par la notion de Type Abstrait de Données: Spécification = Signature +
  - Préconditions (conditions sous lesquelles une méthode peut être appelée)
  - Postconditions (propriétés garanties par une méthode)
  - Invariants de classe (vrai à l'entrée et à la sortie des méthodes)

- Directement dans le modèle
  - notation entre **{ } accrochée à un élément de modèle**

- Dans un document séparé, en précisant le **contexte**
  - Invariants = Propriétés vraies pour l'ensemble des instances de la classe
    - dans un état stable, chaque instance doit vérifier les invariants de sa classe

context Compte inv:
solde >= plancher

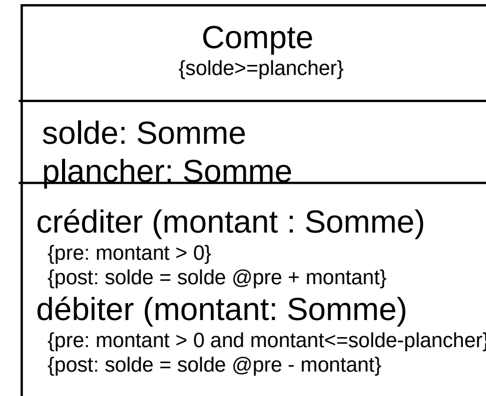
Précondition: *ce qui doit être respecté par le client*

- Spécification des conditions nécessaires pour qu'un client soit autorisé à appeler une méthode
  - exemple: montant > 0
- Notation en UML
  - {«precondition» OCL boolean expression}
  - Abbreviation: {pre: OCL boolean expression}

Postcondition: *Ce qui doit être assuré par l'implantation*

- Spécification de ce qui sera vrai à la complétion d'un appel valide à une méthode
  - exemple: solde = solde @pre + montant
- Notation en UML
  - {«postcondition» OCL boolean expression}
  - Abbreviation: {post: OCL boolean expression}
  - Opérator pour accéder à la valeur « d 'avant »:
    - OCL expression @pre

Etre abstrait et précis avec UML



Université de Rennes

Conception par Contrat

- Assertions des contrats :
  - jouent un rôle crucial dans la séparation nette des responsabilités dans un système modulaire
  - contrat entre l’appelant d’une méthode (le client) et l’implantation de la méthode (le contractant) :

Pourvu que le client appelle la méthode dans des conditions où l’invariant de classe du contractant et la précondition de la méthode sont respectés,
alors le contractant promet que lorsque la méthode terminera, le travail spécifié dans la postcondition sera effectué, et l’invariant de classe sera respecté.

Université de Rennes

Contrats en Java

- Pas de support direct en Java
  - Support direct en Eiffel, Scala, C#/Spec#
  - Support par annotations en Python (PyContracts), Kotlin, JavaScript
- Pour la **spécification** : dans le JavaDoc
- Pour **l’implantation** : utilisation de *assert* et *exceptions*
- Pour aller plus loin : Java Modeling Language (JML)
  - Cf cours MEF de Sandrine Blazy

Université de Rennes

Contrats en Java : Spécification en Javadoc

Compte
{solde>=plancher}
solde: Somme
plancher: Somme
créditer (montant : Somme)
{pre: montant > 0}
{post: solde = solde @pre + montant}
débiter (montant: Somme)
{pre: montant > 0 and montant<=solde-plancher}
{post: solde = solde @pre - montant}

```
/**
* ...
* @param montant : quantité à créditer sur le compte
* @requires
* montant > 0
* @ensures
* solde = solde@pre + montant
*/
public void créditer(int montant) {
}
```

précondition

postcondition

Université de Rennes

Contrats en Java : Implantation

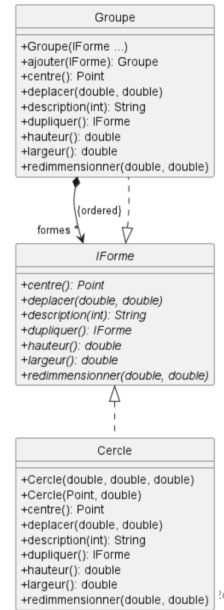
- Invariant
  - Faire une méthode : `boolean invariant(){}`
    - Augmentable dans les sous-classes par redéfinition avec appel à `super.invariant();`
- Précondition
  - Lancer une exception. Par ex. pour `créditer(int montant)`:
  - `if (montant<0) throw new IllegalArgumentException("montant<0");`
- Post-condition
  - assert invariant() && solde == oldSolde + montant : *"Postcondition : solde incorrect"*;
    - *Simulation du @pre en utilisant une variable locale*
- **Limitation** : heritage *à la main*
  - Si on redéfini une méthode, il faut ré-écrire son contrat

Intérêt pratique des contrats

- Spécification, documentation
  - *Pas un outil pour la tolerance aux fautes*
- Aide au Test
  - Les post-conditions jouent le role des **Oracles** dans les tests unitaires
  - Usage systématique => plus jamais de debugging
- Permet d'attribuer facilement la responsabilité d'un bug lors de l'integration
  - Précondition violée : problème dans le code du client
  - Postcondition violée : problème dans le code du fournisseur

Exercice

- Contrat de
  - IForme.deplacer()?
  - IForme.redimentionner(dx,dy)?
    - Quid si dx<=0?
    - Quid des figures symétriques comme Cercle ou Losange si dx!=dy?
  - IForme.dupliquer()



3 – Qualité des tests

Quand s'arrêter d'écrire des tests ?

Quels sont les différents types de tests unitaires ?

Quelles données utilisées ?

De la difficulté de la validation intra-composant

```
Acquérir une valeur positive n
Tant que n > 1 faire
    si n est pair
    alors n := n / 2
    sinon n := 3n+1
Sonner alarme;
```

- Prouver que l'alarme est sonnée pour tout n?
  - Revient à prouver que la suite $n_{i+1} = n_i$ pair ? $n_i/2$: $3n_i+1$ converge
  - C'est la conjecture de Syracuse…
- Indécidabilité de certaines propriétés
  - problème de l'arrêt de la machine de Turing…

- **Recours au test**
  - ici, si machine 32 bits, 2^31 = 10^10 cas de tests
  - **5 lignes de code => *10 milliards de tests* !**

Université de Rennes

Couverture de code

- Quand s'arrêter d'écrire des tests ?
- Couverture de code par les tests :
  - méthode d'aide pour savoir quels tests écrire (quels scénarios)
  - Critère de qualité des tests
  - Plusieurs stratégies : les critères de couverture
    - Couverture des lignes de code
    - Couverture des instructions
    - Couverture de branches
    - Couverture de conditions

Université de Rennes

Couverture de code

- Critères de couverture : couverture des lignes de code
  - Proportion des lignes de code exécutées lors d'un test
  - n = 0 => 40%
  - n = 4 => 80%
  - n = 7 => 100%

```
Acquérir une valeur positive n
Tant que n > 1 faire
    si n est pair
    alors n := n / 2
    sinon n := 3n+1
```

- Objectif
  - toutes les lignes de code sont couvertes par au moins un test
- Problème de ce critère : ni nécessaire, ni suffisant

Université de Rennes

Limite de la couverture des instructions

- L'unique test couvre le code de la méthode *multipli*
  - ... mais ne vérifie presque rien (sauf d'éventuelles exceptions)
  - => les assertions sont nécessaires pour vérifier des propriétés
- **=> couverture de 100 % ne veut pas dire code sûr**
- Il existe des critères de couverture plus sophistiqués, mais toujours sans garantie sur le code
  - Couverture des instructions
  - Couverture de branches
  - Couverture de conditions

```
public double multiplier(double val) {
  return valeur*val;
}
```

```
public class TestNombre {
  Nombre nb;

  @BeforeEach
  void setUp() {
    nb = new Nombre(2);
  }

  @Test
  void testToutPourri() {
    nb.multiplier(0);
  }
```


Université de Rennes

Données de test : les classes d'équivalence

- Partitionner les données d'entrée en classes d'équivalence
  - telles que pour chaque classe, le programme a le même comportement
    - n <= 0
    - n = 2^k
    - n = 2*k + 1
    - Valeurs extrêmes :
      - Integer.MAX_VALUE, Integer.MIN_VALUE

```
Acquérir une valeur positive n
Tant que n > 1 faire
    si n est pair
    alors n := n / 2
    sinon n := 3n+1
```

- Donc pas nécessaire d'écrire **tous** les tests
  - Pour chaque classe, prendre une valeur représentative et écrire un test avec
    - n = 0
    - n = 4
    - n = 7
    - ...

Université de Rennes

4 – Problème de l'oracle

Metamorphic testing & Snapshot testing

Université de Rennes

Tester un diagramme Camembert

```
public class CamembertTest {
    static final double delta = 0.000001;
    Camembert c;

    @Before
    public void setUp() {
        c = new Camembert(110,110, 100);
    }

    @Test
    public void testAjouterSecteur() {
        c.ajouterSecteur("red", 0.15);
        assertEquals(0.15*360, c.getArc(),delta);
        c.ajouterSecteur("blue", 0.2);
        assertEquals(0.35*360, c.getArc(),delta);
        c.ajouterSecteur("green", 0.65);
        assertEquals(360.0, c.getArc(),delta);
    }

    @Test
    public void testEnSVG() { // Oracle ????
    }
```


Université de Rennes

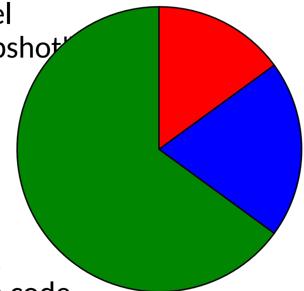
Problème de l'oracle

- Parfois l'oracle est difficile à spécifier
  - Données complexes
  - Etat de l'objet testé « opaque » (ie non facilement accessible)
- 2 possibilités
  - Snapshot testing
  - Metamorphic testing

Université de Rennes

Snapshot testing

- Technique de test logiciel où le comportement actuel d'une application est enregistré sous forme de "snapshot" et comparé aux futures exécutions
  - Executer testAjouterSecteur(), sauver le résultat
  - Vérifier visuellement que c'est OK
  - Copier manuellement le svg dans
    - Public static final String camembertSVG = ...;
- Les variations inattendues dans le snapshot peuvent indiquer des changements non intentionnels dans le code source, aidant ainsi à détecter les régressions potentielles au fil du développement.



```
@Test
    public void testEnSVG() {
        testAjouterSecteur();
        assertEquals(camembertSVG, c.svg());
    }
```

Université de Rennes

Test Métamorphique

- **Approche de Test :** Évaluation du comportement du système en fonction de transformations spécifiques appliquées à ses entrées.
- **Évaluation des Transformations :** Examine comment les sorties évoluent avec des transformations définies telles que modifications, perturbations, ou permutations des données d'entrée.
- **Objectif :** Identifier des comportements indésirables dans des situations où les résultats sont difficiles à prédire, mais où les relations entre les données sont mieux comprises.
- Exemple
  - forme.deplacer(30,20).deplacer(-30,-20) doit remettre l'objet forme dans son état initial