Tests de régression dans le contexte des systèmes orientés aspect : une approche basée sur les modèles

Patrick Chaput, Baccalauréat en informatique
Département de Mathématiques et Informatique
Université du Québec à Trois-Rivières, hiver 2014

Université du Québec à Trois-Rivières
Savoir. Surprendre.

Problématique

La gestion de l'évolution des systèmes orientés aspects est difficile en raison de la complexité et des dépendances entre ses composants. Par conséquent, il est difficile d'évaluer l'impact d'une modification sur l'ensemble du système et d'identifier les parties précises qui doivent être testées.

Objectifs

- Définir, implémenter et expérimenter une approche pour les tests de régression des systèmes orientés aspect basée sur les cas d'utilisation et diagrammes d'interaction.
- Adapter la notation UML pour intégrer les aspects.
- Identifier les cas d'utilisation modifiés et les parties du système qui doivent être testées.

Définitions

Aspect : Module d'une préoccupation qui s'étend sur plusieurs objets.
Préoccupation : Partie de la conception encapsulée pour la rendre réutilisable, et faciliter sa maintenance et son remplacement.
Transversale : Qui recoupe plusieurs disciplines ou secteurs.
Préoccupation transversale (*cross-cutting concern*) : Fonctionnalité dont on ne peut assigner la responsabilité à un objet en particulier.
Point de jointure (*join point*) : Endroit **OÙ** est inséré un greffon.
Point de coupure (*pointcut*): Spécifie **QUAND** le greffon sera exécuté.
Introduction : Nouvelles méthodes et attributs à greffer à la classe.
Greffon (*advice*) : Précise **QUOI** faire à quel endroit et à quel moment.
Tissage (*weaving*) : Insertion statique des greffons, points de jointures et points de coupure dans les classes existantes.
Symbiose : Résultat du tissage d'une classe et d'un aspect.
Hôte : Classe à laquelle se greffent des aspects.

Le paradigme orienté aspect

Discipline qui abstrait et encapsule les préoccupations transverses dans des modules séparés appelés aspects. Une préoccupation transversale peut mener à une duplication du code et une dépendance entre les composants. Les aspects viennent enrichir ou modifier les classes pour répondre à des besoins circonstanciels.

Un aspect définit une série d'opérations appliquées de manière dynamique à des endroits spécifiques. Une classe peut être enrichie d'un nouvel attribut et d'une nouvelle fonctionnalité qu'on appelle greffon. Le greffon permet d'altérer la fonctionnalité d'une classe existante en ajoutant de nouvelles instructions à des points de jointure qui sont déclenchés à des points de coupure spécifiques.

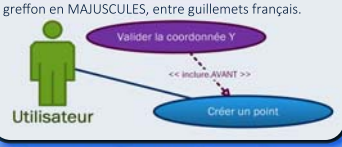
Le tissage

Le tissage s'effectue au moment de la compilation ou de l'exécution
Les aspects se greffent aux classes hôtes pour former des symbioses.
Les introductions sont faites à l'insu de la classe hôte et ne sont exploitables que par les aspects. Les aspects peuvent exploiter les attributs et méthodes de la classe hôte et des autres classes.

Notation UML des aspects

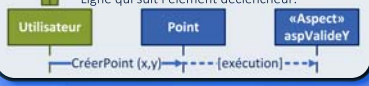
Cas d'utilisation (*use case*)

Extension de la relation **Inclure** (*uses*) suivie du type de greffon en MAJUSCULES, entre guillemets français.



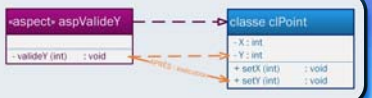
Diagrammes d'interaction

Point de coupure entre crochets sur une flèche pointillée.
Greffon d'interruption (*before* ou *around*)
Même ligne que l'événement déclencheur.
Greffon de réaction (*after*)
Ligne qui suit l'élément déclencheur.



Diagrammes de classe

Greffon : Pointe sur la classe hôte.
Points de jointure : Contient points de coupure et type de greffon en MAJUSCULE.
Intrusion : Pointe sur l'attribut ou la méthode d'une autre classe.



Expérimentation - cas de base

Machine distributrice de soda

La machine est initialement constituée de deux modules : *Distribution* et *Paiement*. Ces composants sont indépendants. Un utilisateur est libre de l'ordre dans lequel il les utilise.

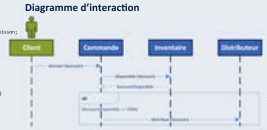
Composant Distribution

Diagramme de classes

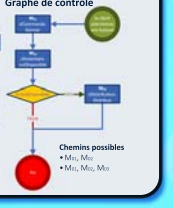
Le module *distributeur* fait tomber la boisson.
Le module *commande* sert d'interface avec l'utilisateur.
Le module *inventaire* vérifie la disponibilité d'une boisson.

```
CLASSE clDistributeur {
  PUBLIQUE VOID Distribue (INT iBoisson) {
    ECRIRE "Distribution de la boisson #" + iBoisson;
  }
}

CLASSE clCommande {
  PUBLIQUE VOID donner (INT iBoisson) {
    SI (objInventaire.estDisponible (iBoisson))
      distributeur.Distribue (iBoisson);
    SINON
      ECRIRE "Boisson non disponible.";
    FIN SI
  }
}

CLASSE clInventaire {
  PRIVE BOOL[] produit;
  PUBLIQUE BOOL estDisponible (INT iBoisson) {
    RETOURNE produit[iBoisson];
  }
}
```

Diagramme d'interaction



Graphe de contrôle



Chemins possibles
- M_{01}, M_{04}
- M_{01}, M_{06}, M_{09}

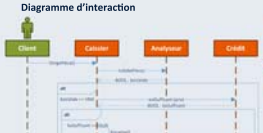
Composant Paiement

Diagramme de classes

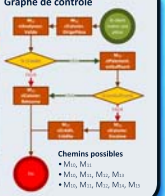
Le module *crédit* fait le suivi du paiement du client.
Le module *caissier* remet ou encaisse le jeton.
Le module *analyseur* vérifie la validité du jeton.

```
CLASSE clCredit {
  PRIVE REEL credit;
  PUBLIQUE VOID crediter (REEL rMontant) {
    credit = credit + rMontant;
  }
  PUBLIQUE VOID debiter (REEL rMontant) {
    credit = credit - rMontant;
  }
  PUBLIQUE BOOL estSuffisant (REEL rPrix) {
    RETOURNE (credit >= rPrix);
  }
}

CLASSE clCaissier {
  PUBLIQUE void DirigePiece (clPiece objPiece) {
    SI (objAnalyseur.Valide(objJeton))
      ALORS SI (objPaiement.estSuffisant(1))
        Retourne();
      SINON
        Encaisse();
        objCredit.credite(objJeton.getValeur());
      FIN SI
    SINON
      Retourne();
    FIN SI
  }
  PRIVEE void Retourne() { ECRIRE "Retour piece.";}
  PRIVEE void Encaisse() { ECRIRE "Garde piece.";}
}

CLASSE clAnalyseur {
  PRIVE clPiece[] tabPieceValides;
  PUBLIC BOOL Valide (clPiece objPiece) {
    RETOURNE (tabPieceValides.CONTIENT (objPiece));
  }
}
```

Diagramme d'interaction



Graphe de contrôle



Chemins possibles
- M_{05}, M_{10}
- M_{05}, M_{06}, M_{07}, M_{14}
- M_{05}, M_{06}, M_{07}, M_{08}, M_{13}, M_{15}

Tests de régression

Technique des cas d'utilisation

Méthode en trois étapes pour la programmation orientée objet basée sur les diagrammes d'états et diagrammes d'interaction des cas d'utilisation, développée par Linda Badri, Mourad Badri et Pierre-Luc Vincent.

Étape 1 : Analyse statique du code
Comparer la version originale du code et la nouvelle version. Dresser la liste des méthodes modifiées, supprimées et ajoutées.

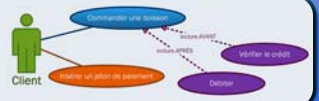
Étape 2 : Identifier les méthodes rattachées aux cas d'utilisation
En se basant sur les diagrammes d'interaction, identifier les cas d'utilisation impactés par la modification.

Étape 3 : Générer les séquences de tests
À partir des graphes de contrôle, identifier parmi les séquences existantes celles qui sont réutilisables, celles qu'on doit créer, celles qu'on doit modifier et celles qui sont désuètes.

Expérimentation - ajout d'aspects

Ajout d'aspects à des méthodes sans aspects

Le composant *Distribution* vérifie le crédit. Cette responsabilité n'appartient à aucune des classes. L'aspect *aspCrédit* se greffe au module de *commande* et vérifie auprès du module *crédit* si le client a rempli les conditions nécessaires pour l'obtention de sa boisson. Ensuite, l'aspect *aspDébit* se charge d'encaisser le jeton.



Étape 1 : Analyse statique du code

La première modification consiste à greffer des aspects à des classes qui n'en ont pas. À partir du code de ces aspects, on peut repérer les classes et les méthodes modifiées par ceux-ci si leur signature ne contient pas de métacaractères.

```
ASPECT aspCrédit {
  private bool clCommande.estSuffisant () {
    return clCrédit.estSuffisant(1);}
  AROUND () execution (clCommande.Donner) {
    SI (clCommande.estSuffisant())
      proceed;
    SINON
      ECRIRE "Crédit insuffisant.";
    FIN SI
  }
}

ASPECT aspDébit {
  private void clDistributeur.Débite() {
    clCrédit.Débiter (1);}
  AFTER () execution (clDistributeur.Distribue) {
    clDistributeur.Débite();}
}
```

- Ajout d'une méthode **estSuffisant** à la classe *clCommande*
- Modification à la méthode *clCommande.**Donner***
- Appel de la méthode ajoutée **estSuffisant**
- Reprise de l'exécution
- Ajout d'une méthode **Débite** à la classe *clDistributeur*
- Modification à la méthode *clDistributeur.**Distribue***
- Appel de la méthode ajoutée **Débite**

Diagramme de classes

Méthodes originales modifiées
- M_{01} : clCommande.Donner
- M_{03} : clDistributeur.Distribue

Méthodes ajoutées
- M_{04} : clCommande.estSuffisant
- M_{05} : clDistributeur.Débite

Étape 2 : Identifier les méthodes rattachées aux cas d'utilisation

Cas d'utilisation impactés
Les méthodes M_{01}, M_{03}, M_{04} et M_{05} appartiennent toutes au cas d'utilisation *Commander une boisson*.

Étape 3 : Générer les séquences de tests

Chemins possibles
- M_{01}, M_{04}, M_{12}
- M_{01}, M_{04}, M_{13}, M_{02}
- M_{01}, M_{04}, M_{13}, M_{02}, M_{03}, M_{05}, M_{16}

Conclusion et recherches futures

D'autres études de cas plus étoffées ont été étudiées, dont la modification d'une classe dotée d'un aspect et d'un aspect déjà intégré. On doit encore valider cette approche et identifier les avantages et les inconvénients qu'elle apporte.

Le tisseur d'aspect et l'analyse statique
À cause du tissage, l'analyse statique du code de manière classique peut s'avérer difficile. La méthode exige des points de jointure définis de manière stricte sans métacaractères. Il faudra développer de nouveaux outils en utilisant l'environnement de travail JUnit pour supporter cette analyse.

La notation UML et la programmation orientée aspect
La notation pour représenter les aspects et leur influence dans un diagramme UML est encore au stade expérimental et a besoin d'être raffinée pour être efficace.

Les patrons d'intrusion des aspects
Rinard, Salcianu et Bugrara classent les aspects selon leurs interactions directes et leur degré d'intrusion. Cette classification est une piste intéressante pour analyser l'impact d'un aspect sur un programme. Le défi sera de pouvoir bien les représenter grâce à une notation UML légère et intuitive.

Références

Livres
- Robinson, David. Aspect-oriented programming with the e verification language: a pragmatic guide for testbench developers / David Robinson.

Articles
- Linda Bardi, Mourad Badri & Pierre-Luc Vincent : Regression Testing of Object-Oriented Software: Towards a Hybrid Technique
- Linda Bardi, Mourad Badri & Daniel St-Yves : Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique
- Farida Mostefaoui and Julie Vachon, Formalization of an Aspect-Oriented Modeling Approach.
- Romain Delamare. Analyses automatiques pour le test de programmes orientés aspect
- Freddy Munoz, Benoit Baudry, Olivier Barais : Improving Maintenance in AOP Through an Interaction Specification Framework.
- Martin Rinard, Alexandru Salcianu, & Suhabe Bugrara : A Classification System and Analysis for Aspect-Oriented Programs

Sites
- Spring Framework : http://static.springsource.org/spring/docs/2.0.8/reference/aop.html
- AspectJ : Pointcuts - Appendix B. Language Semantics : http://www.eclipse.org/aspectj/doc/next/progguide/semantics-pointcuts.html