Fiche écrite par Timothée Blanchy, Antoine Girard, Priscilia Gonthier, Yu-Chi Lee et Romain Peyremorte

Sirius : Permet d'afficher et de modifier graphiquement un modèle

Contraintes OCL :
- Permettent d'imposer des contraintes sur les paramètres/méthodes d'un élément. (ex : employé)

1) Spécifier le contexte : **context Employee**
2) Invariant pour une classe : **inv: self.age>18**
3) Pre et Post pour les méthodes :

pre : increment > 0
post : self.wage = self.wage@pre + increment
(@pre signifie que l'on prend la valeur du salaire au début de la méthode)

4) Renseigner le corp d'une fonction :

body : self.wage

5) Valeur initiale d'une variable : **init : 900**
6) Définir une variable utilisée par OCL :

Def : annualIncome : Integer = wage*12

collect(*expr*) -> retourne collection résultante d'*expr* pour chaque *elt* de *self*
select(*expr*) -> retourne collection d'*elt* de *self* où *expr* est vrai
reject(*expr*) -> inverse de select (où expr *faux*)
forAll(*expr*) -> *expr* vrai pour tous ?
exists(*expr*) -> *expr* vrai pour au moins un ?
collection.size() -> retourne taille de *collection*
p.oclIsKindOf(t) -> vrai si p conforme au type t
p.oclAsType(t) -> p est du type t
self.name.matches('[A-Za-z_][A-Za-z0-9_]*') : nom ✓
forAll(p1, p2 | p1 = p2 or p1.name <> p2.name) :nom dif
nom.notEmpty() (isEmpty()) : nom non null (null)

Syntaxe basique XText (T2M) :
Process : 'process' ID '{' ProcessElement* '}' ;
ProcessElement : WorkDefinition | WorkSequence | Guidance ;
WorkDefinition : 'wd' ID ;
WorkSequence : 'ws' WorkSequenceType 'from' [WorkDefinition] 'to' [WorkDefinition] ;
enum WorkSequenceType : 's2s' | 'f2s' | 's2f' | 'f2f' ;

Exemple de syntaxe textuelle concrète :

```
process ExempleProcessus {
    wd RedactionDoc
    wd Conception
    wd Developpement
    wd RedactionTests
    ws Conception f2f RedactionDoc
    ws Conception s2s RedactionDoc
    ws Conception f2s Developpement
    ws Conception s2s RedactionTests
    ws Developpement f2f RedactionTests
}
```

Transformation modèle à texte (Acceleo) :
Engendrer du code, de la doc ou un texte d'entrée d'un outil (représentation Tina d'un réseau de Pétri)

Acceleo : - langage à balise (à la XML),
- utilise OCL pour requêter le modèle,
- commentaire [comment] peut contenir directives,
- requête (query) : méthode qui renvoie une donnée
- Template : contient le texte à produire qui peut inclure des structures de contrôle (for, if...) ou des expressions Acceleo ([expression/])
- Pas de liaison dynamique lors d'appel requêtes et templates

Transformation modèle à modèle(M2M) :

Moyens :
- Langage généraliste (e.g. Java) : tout est à la charge du programmeur !
- Langage de transformation déclaratif :
  - ensemble de règles
  - une règle transforme un motif sur M_A en éléments de M_B
  - l'outil de transformation identifie les motifs, applique les règles dans le bon ordre, collecte les résultats et construit M_B
- Dans tous les cas : importance d'avoir un langage de requête efficace (type OCL)
- Un standard publié par l'OMG : QVT (Query/View/Transformation)

Propriétés d'une transformation M2M :
- Transformations :
  - endogènes : mêmes méta-modèles source et cible,
  - exogènes : méta-modèles source et cible différents
- Transformations unidirectionnelles ou bidirectionnelles
- Traçabilité : garder un lien (une trace) entre les éléments cibles et les éléments sources correspondants.
- Incrémentalité : une modification du modèle source sera répercutée immédiatement sur le modèle cible.
- Réutilisabilité : mécanismes de structuration, capitalisation et réutilisation de transformation.
- Rq : Il peut y avoir plusieurs modèles d'entrée et/ou plusieurs modèles de sortie.

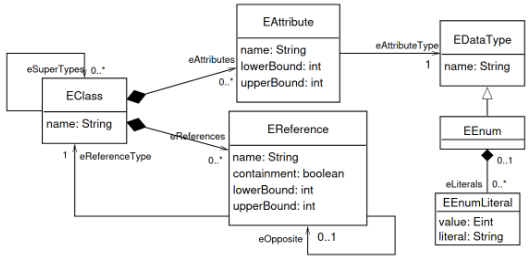
Modèle : simplification, abstraction d'un système.
But : Mieux comprendre le système
Méta-modèle : modèle du modèle
Modèle peut être conforme à un méta-modèle

Pyramide : Instance -> Modèle -> Méta-modèle -> Méta-méta-modèle

Intérêts :
- Définir des propriétés structurelles
- Décider de la conformité
- Transformer/rendre interopérable
- Raisonner/travailler sur les modèles

LTL : Logique Temporelle Linéaire
Ex : [] <> Ete; # Toujours il y aura un été
= <> Ete; # Il n'y aura pas d'été

Langages de méta-modélisation :
- MOF (Meta-Object Facility de OMG
- Ecore (implantation de MOF)
- KM3, Kermeta, …



Extrait du méta-modèle Ecore

EMF : Ecore Modeling Framework (Eclipse).
Permet d'engendrer :
- Le modèle java
- Un schéma XML
- Un éditeur arborescent pour saisir un modèle

ATL :

```
module nomModule;
create OUT: modelSortie from IN: modelEntree;
--def methode en plus
helpers context appliquéSurQuoi
def: nomMethode() : typeRetour =
        Code;
--règle de transformation de E en S avec transfert attr1 et attr2
rule nomRegle {
   from nomVarE : typeVarE
   to nomVarS : typeVarS (attr1VarS <- attr1VarE,
                          attr2VarS <- attr2VarE)
}
```

Type constitué de model!ElementModel, code en ocl

Tests

JUnit :
Permet de rédiger des tests et de les exécuter

JaCoCo : Java Code Coverage
Permet de voir si les tests rédigés permettent de couvrir l'intégralité du code et donne les zones non-couvertes

Mockito :
Outil qui va avec JUnit qui permet de lancer des tests dans des cas précis.

PITest :
Permet de rédiger des tests mutants (léger changement des paramètres qui invalident normalement les tests).

Introspection : Interroger dynamiquement les objets d'une application pour retrouver la structure de l'application : les classes, les attributs, les méthodes, les constructeurs, …

En Java :
Class<?> cls obtenable avec *Point.class* ou *Class.forname("Point")* ou *p1.getClass()*
Méthodes d'un objet *Class* (*Method* peut être remplacé par *Constructor* ou *Field*) :
- *getMethods()*: méthodes publiques et héritées
- *getDeclaredMethods()*: toutes méthodes déclarées (pas héritées)
- *java.lang.reflect.Method getMethod(name)*
- *String getName()*
- *Class<?> getSuperclass()*
- *Class<?> getInterfaces()*
- *java.lang.reflect.Modifier getModifiers()* : droits d'accès

Intercession : Modifier l'état d'une application en s'appuyant sur les informations obtenues par introspection.

En java :
- Créer un objet à partir d'un constructeur (ou classe mais *deprecated*) avec *newInstance()*
- *set()* et *get()* un field
- *invoke()* une méthode

Exemple d'utilisation : Tests de méthodes privés
Attention : pénalité en temps d'exécution

Couverture de test :
(I) Instruction : exécuter chq instruction (ligne) du code
(D) Décision : dans sélection, choix, conditionnelle : passer dans toutes les branches
(DC) Décision/Condition : chq condition élémentaire prend au moins une fois la valeur vrai et la valeur fausse
(MC) conditions multiples : D et C en même temps
(MC/DC) Modified Condition/Decision Condition : on enlève un certain nb de cas, tq en changeant 1 condition élémentaire on change la condition composée

Patrons fondamentaux

Délégation : Attribut d'une classe
- Plus lourd, plus large qu'héritage
- Permet de cacher des éléments de la super classe
- Classe « Métier » utilise classe « Utilitaire »

Interface/Classe Abstraite
- On fait souvent les 2 pour laisser le choix

Immuable : ex : string l'est mais pas string buffer

Interface de Marquage : Interface vide, Définir une propriété sémantique booléenne sur une classe

Patrons créateurs

Monteur : Dissocier la construction d'un Objet complexe de representa° de sorte que le même processus de construc° permette des représenta° différentes.

Fabrique : Définir une interface pour la créa° d'un objet mais en laissant à des sous-classes le choix des classes à instancier.

Fabrique Abstraite : Fournir une interface pour la créa° de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes.

Prototype : Spécifie le type des objets à créer par une instance (le prototype) et crée de nouveaux objets en copiant ce prototype (clonage).

Singleton : Garantir qu'une classe n'a qu'une seule instance et fournir un point d'accès global à cette instance.

Patrons structurels

Adaptation : Convertit l'interface d'une classe en une autre conforme à l'attente d'un client. Permet de faire collaborer des classes aux interfaces incompatibles.

Pont : Découple une abstrac° de sa réalisa° afin que les deux puissent être modifiées indépendamment de l'autre.

Composite : Compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Permet au client de traiter uniformément un objet individuel ou les combinaisons de ceux-ci

Décorateur : Attache dynamiquement des responsabilités supplémentaires à un objet. Alternative souple à l'héritage pour ajouter de nouvelles fonctionnalités

Façade : Fournit une interface unifiée pour un sous-système, interface de haut niveau rendant le système plus facile à utiliser (couplage réduit).

Poids Mouche : Utilise une technique de partage permettant la mise en œuvre efficace d'un grand nombre d'objets de granularité fine.

Procuration (Proxy) : Fournit à un client un mandataire ou un remplaçant pour contrôler l'accès à un objet fournisseur

Patrons comportement

Chaîne de responsabilités : Évite le couplage entre l'émetteur d'une requête et ses récepteurs. Chaîne les récepteurs qui se passent la requête jusqu'à ce qu'un la traite (le gestionnaire)

Commande : encapsuler une requête (ou un traitement) comme un objet (version objet des callbacks). Paramétrer le contexte client. Gestion possible des requêtes/traitements en FIFO, annulation possible.

Interpréteur : Définit une représentation de la grammaire d'un langage simple ainsi qu'un interpréteur.

Itérateur : Fournit un moyen pour accéder séquentiellement à chacun des éléments d'un agrégat d'objets sans révéler la représentation interne de l'agrégat.

Médiateur : Définit un objet encapsulant les modalités d'interaction d'autres objets. Limite le couplage car évite une référence explicite entre ces objets

Mémento : Transmettre à l'extérieur d'un objet son état interne sans violation de l'encapsulation dans le but de restaurer ultérieurement sont état.

Observateur : Définit une interdépendance de type un à plusieurs de sorte que quand un objet change d'état, tous ceux qui en dépendent en soient avertis et automatiquement mis à jour

État : Permet à un objet de modifier son comportement quand son état interne change. Donne l'impression que l'objet change de classe.

Stratégie : Définit une famille d'algorithme, encapsule chacun d'entre eux et les rend interchangeables. Permet aux algorithmes d'évoluer indépendamment de leurs clients

Patron de méthode : Définit, dans une opération, le squelette d'un algorithme en en déléguant certaines étapes à des sous-classes.

Visiteur : Modélise par une classe une opération applicable aux éléments d'une structure d'objets et permet de définir de nouvelles opérations sans modifier les classes de la structure

Patron de conception (def patron)

Définition : (design pattern) décrit une structure commune et répétitive de composants en interaction (la solution) qui résout un problème de conception dans un contexte particulier.

Description :

Nom : nom de référence du patron -> étend le vocabulaire du concepteur (décrit le pb de conception)

Intention : courte description de :
- ce que fait le patron de conception ;
- sa raison d'être ou son but ;
- cas ou problème particulier de conception concerné.

Alias : Autres noms connus pour le patron

Motivation :
- scénario qui illustre un cas de conception
- montre l'architecture en classes et objets de la solution
- aide à comprendre les descriptions plus abstraites du modèle

Indications d'utilisation :
- Quels sont les cas qui justifient l'utilisation du patron?
- Quelles situations de conception peuvent tirer avantage du patron ?
- Comment reconnaître ces situations ?

Structure : description de la solution sous forme de :
- un diagramme de classe pour l'architecture ;
- des diagrammes d'interaction pour la dynamique.

Constituants : classes/objets de la solution avec leurs responsabilités

Collaborations entre les constituants pour assumer leurs responsabilités

Conséquences :
- compromis induits par l'utilisation du patron
- impacts sur l'architecture de conception
- gains en terme de diminution du couplage dans la solution

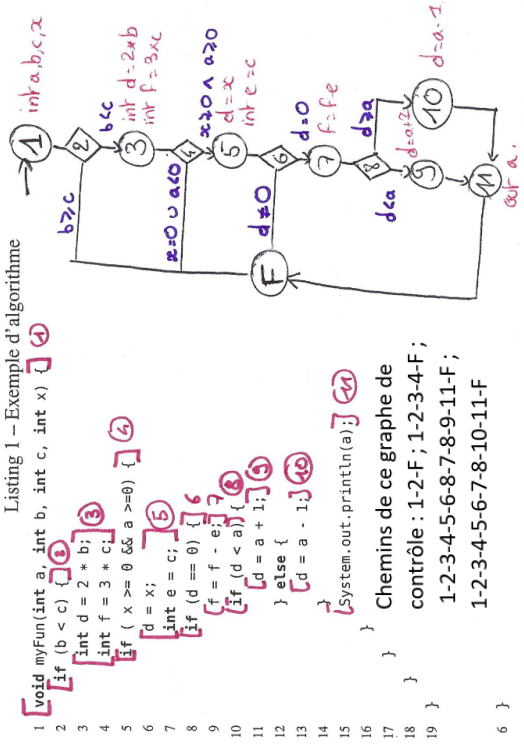
Implantation : Solutions types, techniques, pièges et astuces.

Exemples de code : extraits de code illustrant la mise en œuvre du patron

Utilisations remarquables : exemples issus de systèmes existants

Patrons apparentés :
- patrons similaires et différences essentielles utilisation conjointe avec d'autres patrons

Couverture Control Flow Graph (CFG)



Noeuds du graphe de contrôle

Ensemble d'utilisations des variables (groupe de ligne) :
- **c-use** : var utiliser pour calcul à droite
- **p-use** : var utiliser com predicat (cond)
- **A-def** : var utilizer à gauche de "="
- **def-use** (DU-path) : paire d'où var est def pour où var est used

all-def : pour chaque def D, avoir un test qui relie D à une de ses utilisations

all-uses : couvrir au moins un chemin pour chaque paire def-use (variantes : ***all-p-uses*** ou ***all-c-uses***)

all-du-path : couvrir tous chemins de paire def-use.

Cas de tests (CT)

Donnée de tests (DT)

Oracle de test : Source utilisée pour déterminer les résultats attendus à comparer avec les résultats obtenus de l'application en cours de test.

Test Boite Noire : à partir de spécifications

Test Boite Blanche : à partir du code

Test Probabiliste : domaines des entrées + args stats

Test BN combinatoire = test exhaustif sur une sous-partie (bien identifiée) des combinaisons possibles des valeurs d'entrée -> select les DT pour couvrir ∀ paires de val

Annotation :

@Retention dit comment l'annotation est stockée :
- `RetentionPolicy.SOURCE` : conservé uniquement par la source, ignorée par compilateur
- `RetentionPolicy.CLASS` : conservé par le compilateur, ignorée par la JVM
- `RetentionPolicy.RUNTIME` : conservé par la JVM → utilisé par runtime environment.

@Documented doit être documenté (avec JavaDoc)

@Target restreint le type d'élément où l'appliquer :
- `ElementType.ANNOTATION_TYPE` sur annotation.
- `ElementType.CONSTRUCTOR` sur constructeur.
- `ElementType.FIELD` sur attribut ou property.
- `ElementType.LOCAL_VARIABLE` sur variable locale.
- `ElementType.METHOD` sur method-level
- `ElementType.PACKAGE` sur la déclaration de package.
- `ElementType.PARAMETER` sur parametres de methode.
- `ElementType.TYPE` sur tous les éléments d'une classe.

@Inherited types d'annotations peuvent être hérités d'une superclasse.

@Repeatable annotation peut être appliquée + d'1 fois

@Interface pour définir une nouvelle annotation