Conception et Programmation à Objets Avancée 3

Mikal Ziane

- Refactoring et cycle agile
- Raisonner sur le graphe de dépendances
- Extract Method et Move Method
- Extract Interface / Extract Superclass
- Replace conditional with polymorphism
- Déplacer les new interdits

Refactoring (verbe)

- Transformer une partie d'un logiciel
  - **pour améliorer sa qualité,**
  - **sans changer sa fonctionnalité**.
- Fonctionnalité: telle donnée donne tel résultat.
- Sans changer son comportement **externe**
- Mais des aspects "non fonctionnels" du comportement peuvent changer : vitesse d'exécution …
- But pour nous : réduire l'impact de certains changements

Refactoring (nom)

- Petite **transformation** « non fonctionnelle »
- **Attention** : certains "refactorings" peuvent parfois casser des fonctionnalités !
- Il peut donc y avoir des **préconditions** à respecter.
- Exemple : *rename* peut casser du code client s'il n'est pas accessible !
- Un catalogue en ligne : http://sourcemaking.com/refactoring

Cycle agile

- Un nouveau besoin arrive
- Si c'est une surprise
  - **cacher** ce qui n'était pas censé changer (restreindre le couplage)
  - tant qu'il reste des dépendances interdites
    - **remanier pour ôter des dépendances interdites**
- Tant que le besoin n'est pas suffisamment vérifié
  - Ajouter un test
  - Le faire compiler
  - Tant qu'il échoue
    - modifier le programme
- Remanier pour **simplifier** le programme

Quand une dépendance est-elle nuisible ?

- A utilise B est nuisible quand
- A et B varient à des **rythmes différents**
  - Surtout si **B change plus vite que A**
- **A est difficile ou impossible à changer**
  - Si A est gros, mal documenté …
- **B n'est qu'un cas particulier**
  - A doit traiter tous les cas similaires !

Comment détecter les dépendances nuisibles ?

- En exprimant un contrainte de couplage
  - en français : détection par l'humain
  - ou mieux dans un langage utilisable par un outil comme puck
- Quand on pense que A -> B est nuisible on cache B de A

Hide B from A

- Il vaut mieux utiliser des ensembles généraux
  - ne pas cacher chaque classe ou méthode une à une

As = [A1, A2]

Bs = [B1, B2, B3]

Hide Bs **except** B3 **from** As **but-not-from** A2

Supprimer les dépendances nuisibles

- Dépendances nuisibles : ne respectent pas une contrainte de couplage
- Chaque lien interdit est une **violation** de la contrainte
- On s'attaque souvent à chaque violation une à une
- Mais on doit grouper certaines violations si elles ont une **dépendance principale**
  - A a; // **dépendance principale** vers A
  - a.m(); // dépendance secondaire vers A.m
  - si on change la principale on change aussi la seconde

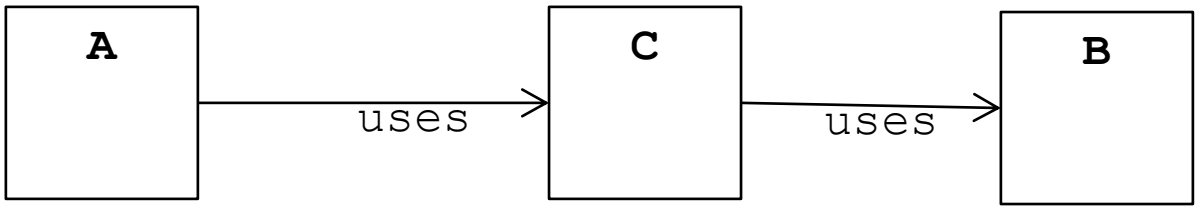
Raisonner sur le graphe de dépendance

- On veut **minimiser** le nombre de **violations**
- **Planifier** les étapes de refactoring sur le graphe ou sur l'architecture
- Chaque refactoring doit
  - ôter une ou plusieurs violations
  - ou préparer un autre refactoring qui lui ôtera une violation
- Exemple
  - Ajouter une abstraction
- **Répercuter** sur le code (avec ou sans outil)

Casser une violation

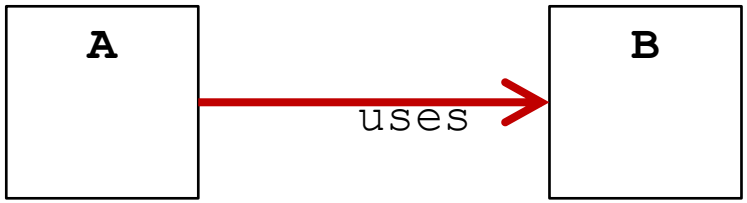
- Déplacer les entités **mal placées** (soulignées, rouges)
  - move method, move field, (move class)
  - peut nécessiter de créer un conteneur
  - dans le code : mettre à jour leurs utilisations
- Oter les arcs **uses** interdits (gras, rouges)
  - extract method, extract interface, …
  - nécessite en général d'introduire une abstraction
  - dans le code : mettre à jour les utilisations
- Très peu de sortes de refactoring à considérer

Pliages (*extracts*) simples

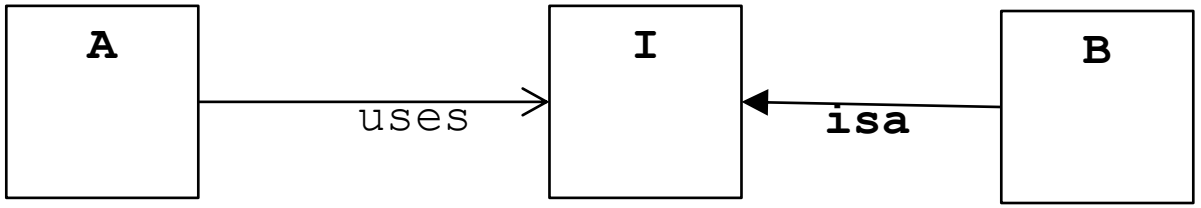


- On introduit ou on réutilise une indirection : C
- C doit être une abstraction de l'utilisation de B par A

Inversion de dépendance



B **est utilisé** par A



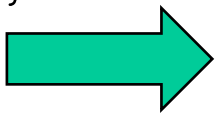
B **utilise** I

- Plus personne n'utilise B directement
- A manipule des instances de B typées I via des méthodes polymorphes
- Attention avec les **new** B() : `new` n'est **pas polymorphe** !

Extract Method (pliage)

- Remplace un fragment de code par un appel de méthode (de la même classe)
- Le fragment est déplacé dans le code de la méthode
  - Il peut y avoir besoin de paramètres
  - Le fragment est alors modifié pour utiliser les paramètres
- Parfois la méthode existe déjà : ne pas la dupliquer
- Utilité : casse la dépendance sur le code d'origine surtout **combiné à un Move Method**

```
public class Personne {
   private String nom;
   public String toString()
      { return nom; }
}
```



```
public class Personne {
   private String nom;
   public String toString()
      { return getNom(); }
   public String getNom()
      { return nom; }
}
```

Move Method

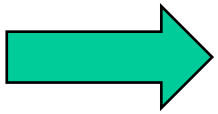
- On déplace une méthode m d'une classe A à une classe B.
  - il faut réécrire tous les appels A.m() en B.m()
- Variante : on copie m dans B et on la garde dans A
  - A.m appelle alors B.m (plus besoin de réécrire les appels de A.m)
- Attention si m est une méthode publique :
  - le code client inaccessible au compilateur peut être cassé
  - préférer alors la variante
- Utilité: renforcer la cohésion de A
  - une entité doit avoir une seule raison de changer (d'après R.C. Martin)
- Autre usage : combiné à extract method

Extract Method puis Move Method

```
public class Personne {
  public int âge;    // pas bien !
}

public class Election {
  public boolean peutVoter(Personne p) {
        return p.âge > 18; // pas bien !
    }
}
```

Extract Method puis Move Method

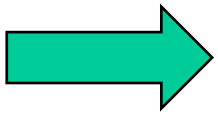


```
public class Personne {
  private int âge;   // bien !
  public int getAge()
    { return âge; }
}
class Election {
  public boolean peutVoter(Personne p)
    { return p.getAge() > 18; } // bien !
}
```

Extract Interface / Extract Superclass

- A partir d'une ou plusieurs classes Ci
- Au moins une méthode doit être en commun
  - renommer au besoin les méthodes similaires
- Choisir quelles méthodes extraire ?
  - union => ajouter exception où elles manquent
  - intersection : parfois trop restrictif
- Introduire une interface I avec ces méthodes
- Ajouter "implements I" dans les classes
- Utilité : casse les dépendances vers les classes Ci grâce aux méthodes **polymorphes** !
- Attention : **ne résout pas les dépendances `new Ci()`**

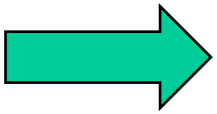
```
public class Livre {
  private String titre;
  private boolean estEmprunté = false;
  private GregorianCalendar dateEmprunt;
  public Livre (String titre) {this.titre = titre;}
  public void emprunter () {
    assert ! estEmprunté;
    estEmprunté = true;
    dateEmprunt = new GregorianCalendar();
  }
  public String toString () {
    String s = titre;
    if (! estEmprunté)
       s += " (disponible)";
    else
       s += " (empunté le "
         + dateEmprunt.get(Calendar.DATE) + "/"
         + dateEmprunt.get(Calendar.MONTH) +")";
    return s;
  }
}
```



```
public interface IEmpruntable {
     void emprunter();
}
public class Livre
                implements IEmpruntable {
// . . .
};
```

- "Inversion" (en fait suppression) des dépendances sur Livre
- Sauf les new !

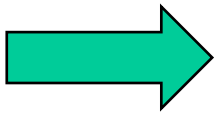
```
public static void main(String[] args) {
  Livre livre = new Livre("One Piece");
  System.out.println(livre);
  livre.emprunter();
  System.out.println(livre);
}
```



```
public static void main(String[] args) {
  IEmpruntable livre = new Livre("One Piece");
  System.out.println(livre);
  livre.emprunter();
  System.out.println(livre);
}
```

Replace conditional with polymorphism

```
String langue = "français";
// ...
if (langue.equals("français"))
  System.out.println("Bonjour");
else
  System.out.println("Hello");
```



```
Langue la = new Français();
System.out.println(la.salutation());

public class Langue {
  public String salutation()
     {return "Hello";}
}

class Français extends Langue {
  public String salutation()
     {return "Bonjour";}
}
```

Déplacer les new interdits

- Souvent les dernières dépendances qui restent
- Une méthode polymorphe dépend du **type dynamique** de la **cible**.
- Création d'instance : **pas polymorphe** car **pas de cible** (l'objet n'existe pas encore) !
- Solutions
  - injection : simple mais un injecteur est requis
  - factory statique : simple, fragile, qui dépend d'elle ?
  - interface factory : qui crée la factory concrète ?
  - pattern prototype : assez complexe, boostrap délicat
  - factory method : complexe (hiérarchie parallèle)

Injection de dépendance

- Problème
  - Un client a besoin d'instances d'une ou plusieurs classes
  - Mais ne veut pas en dépendre structurellement
- Solution
  - Un **injecteur** lui fourni les instances
  - en les typant par une **interface**
- Variantes
  - L'injecteur appelle un **constructeur** du client
  - ou un **setteur** du client

Exemple sans injection

adpaté de https://en.wikipedia.org/wiki/Dependency_injection

```
public class Client {
        private IService service;
        Client() {
                service = new Impression();
        }
        public String nomService() {
                return service.getNom();
        }
}
```

Avec Injection

```
// Variante avec constructeur
public Client (IService service) {
        this.service = service;
}

// Variante avec setteur
public void setService(IService service) {
        this.service = service;
}
```