Conception et Programmation à Objets Avancée 2

Mikal Ziane

- Qu'est-ce que la conception du logiciel ?
- Réduire le coût des changements
- Dépendances statiques
- Architecture, graphe de dépendances
- Contraintes de couplage
- Puck

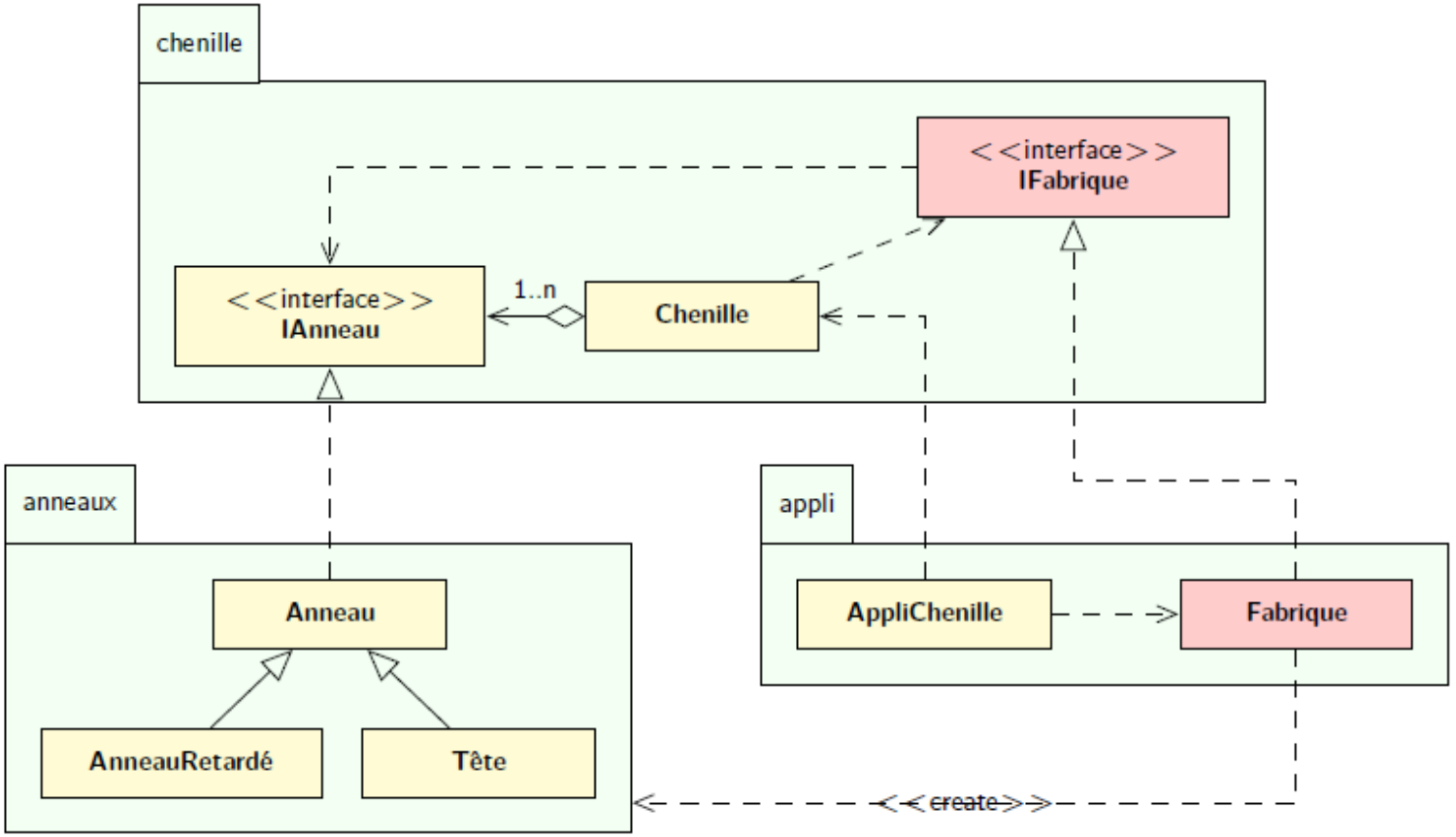
Développer c'est faire des choix

- Quelles **entités** (paquetages, classes, méthodes) définir ?
- Comment les nommer, spécifier, vérifier, implémenter, documenter ?
- Où les placer ?
- Chaque choix influe certains critères
- On ne peut pas gagner sur tous les critères => **compromis**
- Quels critères sont importants dans **mon application** ?
- Les critères et leur importance peuvent **varier** dans le temps.
- On doit donc pouvoir **modifier ses choix** pour s'adapter.

La conception du logiciel

- La **conception** concerne l'architecture logicielle
  - Quels paquetages, interfaces, services ?
  - Quels liens et dépendances entre eux ?
  - On ne s'occupe pas ici du code des méthodes.
- Son résultat est notamment un **diagramme d'architecture**
  - **paquetages** et leurs dépendances
  - **classes** et leurs relations (sous-typage, dépendances, …)
  - on ne montre jamais les attributs des classes
  - souvent on ne montre pas non plus les méthodes
- On peut **améliorer** une application sans changer sa fonctionnalité : **refactoring**

Exemple d'architecture (D. Poitrenaud)



Critères de qualité du logiciel

- Critères externes (visibles par les utilisateurs) :
  - correction externe (tests de validation), facilité d'utilisation, prix, interopérabilité, rapidité, encombrement mémoire, **agilité externe** …
- Critères internes (visibles par les développeurs)
  - correction interne (tests unitaires), clarté, **agilité interne**, **complexité**, …
- Les critères difficiles (mais essentiels) :
  - complexité
  - agilité : facilité de prise en compte des changements (externes et internes)

La complexité d'une application

- Taille (trop de classes, de méthodes, de lignes de code)
- Hétérogénéité (pas de convention de nommage …)
- Manque de cohésion
  - des entités font plusieurs choses à la fois
- Duplication de code
- Manque de clarté (nommage, commentaires …)
- Architecture spaghetti

Enjeux de la conception du logiciel

- Gérer la **complexité**
  - il n'y a pas que la taille (nombre de classes) qui importe
  - les dépendances comptent au moins autant
  - Les dépendances gênent la réutilisation et les changements
- Gérer les **changements**
  - les besoins et les contraintes changent
  - le coût de certains changements peut être prohibitif
  - une bonne architecture peut réduire ce coût en éliminant certaines dépendances

Réduire le coût des changements

- Coût = coût intrinsèque + coût marginal
- Le coût intrinsèque est inévitable
- Le coût marginal dépend de l'architecture logicielle
- Si une entité E change :

  les entités qui **dépendent** de E risquent de changer

$\Rightarrow$ **Réduire le coût marginal** des changements **attendus**

$\Rightarrow$ En éliminant certaines dépendances (via une indirection)

- Attention : éliminer une dépendance peut impliquer l'ajout de nouvelles entités (classes …)
- Donc n'éliminer que les dépendances vraiment nuisibles

L'essentiel à maîtriser

- Critères clés : **complexité** et **agilité**
- Relation clé : **dépendance** entre entités
- Diagramme clé : **architecture** avec dépendances
- Démarche :
  - identifier les **changements**
  - **cacher** les entités qui changent derrière des **abstractions**
  - pour **casser** les **dépendances** nuisibles ou les **inverser**
  - grâce à des **refactorings**
  - sous couvert de **tests**

Dépendances structurelles

- Ce sont des dépendances **statiques**.
- Concernent des **textes**
- Un texte T1 dépend du texte T2 si dans T1 apparait un mot m défini dans T2 : T1 dépend de T2.m.
- Plusieurs niveaux de granularité :
  - quelle partie de T1 dépend de T2.m ?
- En java on se concentre sur les **entités nommées**.
- Un entité nommée e1 dépend de l'entité nommée e2 si le **nom** de e2 **apparait** dans e1.

Deux sortes de dépendances

- **Contenir** une entité
  - une classe ou interface contient méthodes, variables
  - un package contient classes, interfaces
  - une méthode contient variables, instructions
- **Utiliser** une entité via son nom
  - hériter d'une classe, implémenter une interface
  - appeler une méthode
  - utiliser un attribut
  - toute **occurrence d'un nom** est une dépendance (y compris dans un commentaire ou une chaine de caractères)

Impact des changements

- Coût intrinsèque
  - ajouter une classe ou une méthode coûte un certain effort (conception, codage, test, documentation …)
- Coût marginal s'il implique d'autres changements
  - pour que le code compile toujours
  - pour maintenir certaines fonctionnalités
- Le coût marginal dépend du nombre de **dépendances entrantes** et de celui des changements induits
- Le coût marginal peut être énorme !

Propriétés des dépendances

- **Asymétriques** : la direction est très importante

  A dépend de B ≠ B dépend de A

- Concept plus précis que celui de **couplage**
- **Non transitives**
  - Si A dépend de B et B dépend de C
  - A ne dépend pas forcément de C

⇒ **Une indirection (détour) isole de certains types de changements**

- Attention certains changement radicaux peuvent traverser plusieurs dépendances !
  - Ex: Si vous changer le type d'un attribut un accesseur ne peut pas forcément le masquer

Trouvez les dépendances nuisibles

```
static void badDependencies() {
  String[] animaux = new String[3];
                          // chiens et chats
  Scanner in = new Scanner(System.in);
  for (int i =0; i <3; ++i) {
     System.out.println("Chien ou Chat ?");
     animaux[i] = in.nextLine();
  }
  for (int i =0; i <3; ++i)
     badCrier(animaux[i]);
  in.close();
}
```

Trouvez les dépendances nuisibles

```
static void badCrier(String espèce) {
  if (espèce.equals("Chien"))
    System.out.println("aboie");
  else if (espèce.equals("Chat"))
    System.out.println("miaule");
  else System.out.println("crie");
}
```

Dépendances nuisibles en rouge

```
static void badDependencies() {
  String[] animaux = new String[3];
                          // chiens et chats
  Scanner in = new Scanner(System.in);
  for (int i =0; i <3; ++i) {
     System.out.println("Chien ou Chat ?");
     animaux[i] = in.nextLine();
  }
  for (int i =0; i <3; ++i)
     badCrier(animaux[i]);
  in.close();
}
```

Dépendances nuisibles en rouge

```
static void badCrier(String espèce) {
  if (espèce.equals("Chien"))
    System.out.println("aboie");
  else if (espèce.equals("Chat"))
    System.out.println("miaule");
  else System.out.println("crie");
}
```

Refactoring (correction)

```
static void better() {
  final int nbAnimaux = 3;
  Animal animaux[] = new Animal[nbAnimaux];
  String espèce; // espece choisie
  Scanner in = new Scanner(System.in);
  for (int i =0; i <animaux.length; ++i) {
      System.out.println("Quelle espèce d'animal ?");
      espèce = in.nextLine();
      animaux[i] = Animal.créer(espèce);
  }
  for (int i =0; i <animaux.length; ++i)
      System.out.println(
            animaux[i].crier());
  in.close();
}
```

Refactoring (correction)

```
// solution qu'on peut améliorer cf. cours 3
class Animal {
public String crier() { // méthode polymorphe
     return "crie";
}
public static Animal créer (String espèce) {
  if (espèce.equals("Chien"))
     return new Chien();
  else if (espèce.equals("Chat"))
     return new Chat();
  else return new Animal();
  }
}
```

Refactoring (correction)

```
class Chien extends Animal {
public String crier() {
     return "aboie";
     }
}
class Chat extends Animal {
public String crier() {
     return "miaule";
     }
}
```

Quelles dépendances sont nuisibles ?

- Si un entité E ne change pas
  on peut dépendre de E
- Si entité cliente C change au même rythme que E
  C peut dépendre de E
- Si C et E ne changent pas au même rythme : danger !
  C et E ne doivent **pas** dépendre l'une de l'autre
- Si C change peu et E change fréquemment : gros danger !
  - C ne doit **surtout pas** dépendre de E
- Prendre aussi en compte la **difficulté à changer** (taille, disponibilité des sources…) :
- Si C est difficile à changer et E change : danger !

Exemple plus complet

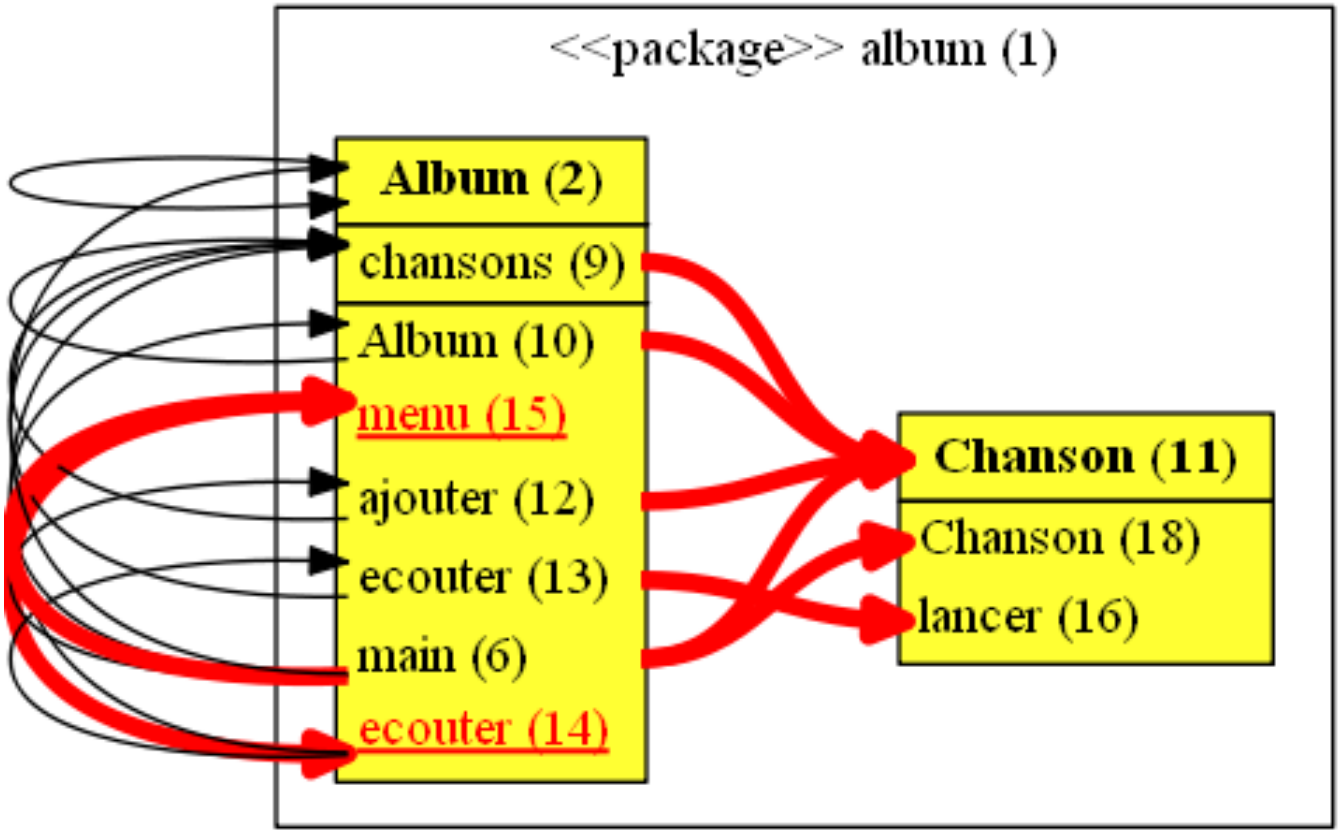
- Embryon de gestionnaire de documents multimédia
- Pour le moment uniquement des chansons
- D'autres types de médias sont prévus
- D'autres interfaces utilisateurs sont prévues
- Deux classes : Album et Chanson
- Album ne doit pas dépendre de Chanson
- Album ne doit pas dépendre de l'interface utilisateur

```
package album;
import java.util.*;

public class Album {
   private ArrayList<Chanson> chansons;

   public Album() {
     chansons = new ArrayList<Chanson>();
   }

   public void ajouter( Chanson c) {
       chansons.add(c);
   }

   public void ecouter (int i) {
       assert (i <0 || i>= chansons.size());
       chansons.get(i).lancer();
   }
```

```
public void menu() {
    String choix;
    while (true) {
        System.out.println("e)couter ou q)uitter");
        Scanner in = new Scanner(System.in);
        choix = in.nextLine();
        if (choix.equals("e"))
                ecouter();
        else if (choix.equals("q"))
                break;
        else System.out.println("Choix non valide");
    }
}
public static void main (String[] args) {
    Album monAlbum= new Album();
    monAlbum.ajouter(new Chanson());
    monAlbum.menu();
}
```

```
 public void ecouter() {
   if (chansons.size() ==0) {
       System.out.println("Il n'y a pas de chansons");
       return;
   }
   System.out.println("Numéro de la chanson de 0 à "
                      + (chansons.size()-1));
   Scanner in = new Scanner(System.in);
   int i = in.nextInt();
   if (i <0 || i>=chansons.size())
       System.out.println("Choix non valide");
   else ecouter(i);
 }
}
package album;
public class Chanson {
   public void lancer() {
       System.out.println("Tra la la");
       }
}
```

Graphe de dépendances



Cacher des entités

- On veut interdire certaines dépendances
- On veut donc cacher des entités
  - soit de certains clients
  - soit de tout le monde
- Si E est cachée de C alors
  - E n'a pas le droit de **contenir** C
    - le nom de C est souligné et en rouge sur le graphe
  - E n'a pas le droit **d'utiliser** C
    - les arcs uses de E vers C sont **épais** et en rouge

Contraintes de couplage

- Le langage **Weland** permet de définir des contraintes de couplages : https://pages.lip6.fr/puck/weland.php
- Pour cacher (au choix)
  - un élément (une entité mais pas ce qu'elle contient)
  - un scope (une entité et ce qu'elle contient récursivement)
- Cacher de qui ?
  - de tout le monde
  - d'un ou plusieurs clients seulement
- En général on cache tout un scope (classe ou paquetage)

Définir des ensembles d'entités

- On veut cacher Chanson d'Album et du package album
- Mais si on mentionne Chanson dans la contrainte
  - elle dépendra elle aussi de Chanson !
- On définit d'abord un ensemble : les documents
- Puis on cache les documents de la classe Album
- Idem pour les méthodes de l'interface : gui

Exemple de fichier decouple.wld

```
import [album]
documents = [Chanson]
gui = [‘Album.ecouter()’,’Album.menu()’]
hide documents from Album
hide gui from Album
```

La grammaire de Weland est définie

https://pages.lip6.fr/puck/weland.php

Puck

- Prototype de recherche Ziane, Besse, Girault (LIP6)
- Disponible sur puck.lip6.fr
- Produit un graphe de dépendance à partir
  - d'un ensemble de fichiers .java ou .jar
  - d'un fichier decouple.wld
- 3 vues sont disponibles
  - Tree- view : la plus complète et la plus stable
  - Dot-svg view : nécessite que dot.exe soit dans le PATH ou qu'un chemin pointe dessus
  - Piccolo2 view : expérimental

Utilisation de Puck

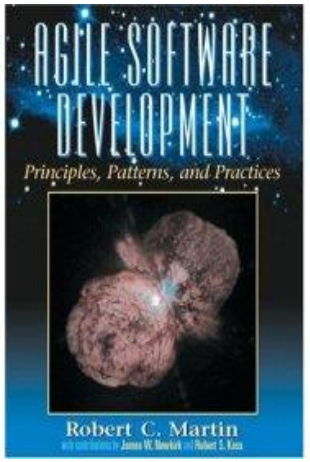
- Attention à l'encodage de vos fichiers (UTF8 recommandé) sinon supprimez les accents
- Organisez vos sources
  - Toutes les sources dans un répertoire src
  - Tous les .jar comme juni4.jar dans un répertoire lib
- Créez un projet ou chargez un projet existant (puck.xml)
  - Le chemin à donner est celui du répertoire qui contient src
  - Attention : d'une machine à l'autre les répertoires changent donc un fichier puck.xml est très peu portable tel quel
- A la création d'un projet Puck va créer un fichier puck.xml
  - Nécessite qu'aucun fichier puck.xml n'existe déjà

Réglage de Puck

- Si les sources ne sont pas dans un répertoire src
  - indiquez les répertoires concernés dans settings/src
- Si une librairie est utilisée (comme junit4.jar)
  - indiquez le chemin sur le fichier .jar dans settings/classpath
  - ou mieux mettre le fichier .jar dans un répertoire lib à coté de src
- Si dot.exe n'est pas dans le PATH
  - Donnez un chemin dessus dans settings/dot-path
- Si le fichier de contrainte ne s'appelle pas decouple.wld ou bien n'est pas au même niveau que src
  - Donnez un chemin dessus dans settings/decouple

Pour aller plus loin

- ***Agile Software development***
  Principles, Patterns, and Practices
  Robert C. Martin
  Prentice Hall
  2003
  www.objectmentor.com/resources/omi_reports_index.html



- **Refactoring: Improving the Design of Existing Code**
  Martin Fowler
  Addison-Wesley
  2002
  refactoring.com/catalog