Génie logiciel pour l'entreprise: Dépendance cyclique et structuration par composants.

Jean-Baptiste Apoung Kamga

Cours M2 IM
Université Paris-sud XI

Plan

1. Notion de dépendances cyclique
   - Définitions et conséquences
   - Unités de mesures
2. Techniques et idées pour réduire les dépendances cycliques
   - Réduction de temps de compilation
   - Réduction de temps d'édition de liens
3. Extracteur de dépendance et Analyseur de *Package*
   - Outils idep, récupération et installation
   - Utilisation pratique de idep

Outline

Définition de la dépendance cyclique

C'est ce processus caractérisée par une inclusion réciproque de deux fichiers entre eux, au travers de potentiels fichiers intermédiaires.

Les fichiers participant à cette inclusion réciproque sont dit **être dans un cycle**.

Conséquence immédiate

- Dans un cycle, la modification d'un fichier induit une nouvelle compilation de tous les fichiers de ce cycle.
- Si le nombre de fichiers dans un cycle est important, la compilation peut requérir un temps considérable.

La dépendance cyclique peut être préjudiciable dans un processus XP.

En effet les tests unitaires qui soutiennent le processus de développement basé sur l'XP pourraient en être sérieusement affectées.

En Stage, en Thèse ou ailleurs, la non prise en compte de cette notion de dépendance cyclique peut compromettre les échéances de la mission confiée !!!

Autre forme de dépendance cyclique

- La dépendance cyclique est liée plutôt à l'aspect physique du projet: Organisation des fichiers et des répertoires etc.
- Cependant l'édition des liens peut aussi quelque fois conduire à des temps de compilation importants.

Cette notion de temps d'édition de liens affine la notion de dépendance cyclique:

- La dépendance cyclique est liée à l'estimation du nombre de fichiers du projet requis pour tester une composante.
- Le temps d'édition de lien quant-à lui quantifie le temps nécessaire pour mettre à jour l'outil de test de non régression.

Ces deux entités sont si liés qu'on ne les dissocie généralement pas lors de la recherche des solutions aux problèmes de structuration efficace, de tests incrémentaux et hiérarchiques et du temps excessif de compilation de tests unitaires.

Outline

Unités de mesure

CCD: *Cummulative Component Dependency*

La dépendance cummulative des composantes d'un sous-système est la somme sur toutes les composants C_i d'un sous-système, du nombre de composantes nécessaires afin de tester C_i.

Exemples

- Dans un système en dépendance cyclique de N composantes, $CCD(N) = N^2$.
- Dans un système de N composantes disposées en arbre **binaire équilibré**, $CCD(N) = (N+1)(\log_2(N+1) - 1) + 1$.

AVD: *Average Component Dependency*

C'est le ratio du CCD d'un système sur le nombre de composantes du sytème: $ACD(S) = CCD(S)/N(S)$

NCCD: Normalized Cumulative Component Dependency

C'est le ratio du CCD d'un système sur le CCD d'un système en arbre équilibré formé du même nombre d'éléments: $NCCD(S) = CCD(S)/CCD_{arbre}(N_S)$

- Une bonne architecture vise à minimiser le CCD.
- Dans une bonne architecture, NCCD $= 1.00$.
- Lorsque $NCCD \leq 1$ l'architecture est dite **plus horizontale** ce qui est souhaité.
- Lorsque $NCCD > 1$ l'architecture est dite **plus verticale** ce qui est éviter tant que possible.

Techniques de réduction des dépendances cycliques

Deux directions possibles

contrairement à ce qu'indique le titre, le problème à résoudre adresse aussi bien le point de vue **compilation** que celui de **l'édition des liens**. Il faut donc identifier des méthodes de résolution dans chaque cas, pour un meilleur résultat global.

1. Techniques de réduction du temps de compilation:
   Escalation(promotion), Démolition, Pointeur opaque, Dumb Data, Rédondance, *Callbacks*, classe manageur, *Factoring*, Escalation de l'encapsulation
2. Techniques de réduction du temps d'édition de liens: Suppression de : héritage privée, données membres locales, fonctions membres privées, fonctions membres protégées, données membres privées, fonctions automatiques générées par le compilateur, certaines directives d'inclusion, arguments par défaut, énumérations.

Outline

Technique de réduction le temps de compilation en C++

La technique est basée sur la suppression de certaines constructions courantes. (*consulter le code fournis en TP pour les identifier*)

- *Escalation (promotion)*: Déplacer les fonctionnalités mutuellement dépendantes plus **haut** dans la hiérarchie
- *Démolition*: Déplacer les fonctionnalités mutuelles dépendantes plus **bas** dans la hiérarchie
- Pointeurs opaques: Amener un objet à dépendre d'un autre uniquement par son nom et non par sa classe.
- *Redondance*: Choisir de remplacer un objet par une quantité négligeable de code (ex. string par char*)
- *Callbacks*: Autoriser au client de fournir une fonction permettant au systèmes de bas-niveau d'agir dans un contexte global
- *Classe manager*: établir une classe qui possède et coordonne des objets de bas niveau.
- *Escalation de l'encapsulation*: Déplacer le point où les détails d'implémentation sont cachées à l'utilisateur à un niveau supérieur dans la hiérarchie.

Outline

Technique de réduction le temps d'édition de liens en C++

La technique est basée sur la suppression de certaines constructions courantes. (*consulter le code fournis en TP pour les identifier*)

On y parvient par la suppression

- De l'héritage privée: en convertissant les **Est-Un** en **A-Un**.
- Des données (Classes) membres privées: en convertissant les **A-Un** en **Reférence**
- Des fonctions membres privées: en les convertissant en fonctions statiques dans les fichiers d'implémentation (*.cpp)
- Des fonctions membres protégées: en créant une composante utilitaire ou en extrayant une classe **protocol** (Pimpl).
- Des fonctions générées automatiquement par le compilateur: en les définissant explicitement ou en prévenant leur définition.
- De certaines directive #include: en utilisant des déclarations (**forward**)
- Des arguments par défaut: en fournissant différentes variantes de la même fonction grâce au polymorphisme
- Des énumérations: en les plaçant dans les fichiers .cpp ou en les remplaçant par des données membres **static const**, ou en les redistribuant dans les classes qui les utilisent.

Outline

Motivation

Motivation

La capacité de modifier un projet de sorte à le rendre rapide à la compilation et à l'édition les liens ainsi qu'au test des composantes dépendra de la vision quantitative que l'on a de l'ensemble du projet.
Il est donc impératif d'identifier un outil capable non seulement de mesurer les problèmes potentiels de dépendance cyclique dans le projet, mais aussi qui soit à même de fournir des pistes éventuelles pour la correction des dépendances cycliques observées.

Exemple d'outil d'analyse dû à John Lakos (1996)

- adep: crée **aliases** pour grouper les fichiers en de composantes cohérentes
- cdep: extrait les dépendances entre plusieurs fichiers lors de la compilation
- ldep: analyse les dépendances entre les fichiers des composants lors de l'édition des liens

Récupération et compilation des outils idep

Récupération de idep

- Récupérer sur la toile l'outil **idep** (attention aux versions anciennes!)
- Alternativement, récupérer le mien dans le répertoire du présent cours.

Compilation de idep

- Entrer dans le répertoire **idep** et créer un sous répertoire **build**.
- Entrer dans le répertoire **build**, exécuter **cmake ..** pour obtenir le `Makefile`
- Taper ensuite **make**, ce qui génèrera **cdep, adep, ldep**

Vous pouvez maintenant placer ces trois exécutables à une endroit approprié et les rendre accessible dans tout terminal (en modifiant la variable PATH).

Commandes de base de idep

Commandes et options de base

1. Extraire des potentielles dépendances lors de la compilation et placer le résultat dans le fichier **deps**:
   cdep -isearchpath *.cpp *.hpp > deps
2. Liter les noms de fichiers non convenablement appariés:
   adep *.cpp *.hpp
3. Mettre les fichiers non convenablement appariés dans un fichier **aliases** pour un traitement ultérieur :
   adep -s *.cpp *.hpp > aliases
4. Vérifier si la première instruction `#include` du fichier .cpp correspond au bon fichier en-tête:
   adep -v -aaliases *.cpp
5. Reprendre en extrayant les composantes de manière automatique:
   adep -e *.cpp > aliases
6. Lister les composantes d'un projet dans un ordre hiérarchique et afficher les statistiques: ldep -ddeps -aaliases

Outline

Utilisation pratique de idep

Démarche pratique

1. On commence par créer le fichier **searchpath** dans lequel on met sur chaque ligne le chemin vers les fichiers de dépendances. On peut mettre sur une ligne un '.' si les fichiers se trouvent dans le répertoire courant. On peut aussi mettre des chemins du type **/usr/.../include/**
2. On génère ensuite le fichier **deps**: cdep -isearchpath *.cpp *.hpp > deps
   Ce fichier contient des dépendances potentielles pouvant influer sur la compilation. Le fichier **deps** n'est pas conçu pour être lu par l'humain, son contenu pouvant être assez dense.
3. On essaie ensuite de créer les composantes locales en groupant les fichiers en-têtes et les différents fichiers d'implémentation, au moyen de la commande adep *.cpp *.hpp. La liste obtenue rend a priori difficile l'appariement des fichiers qui s'y trouvent. On utilise pour cela l'option **-s** en dirigeant la sortie dans le fichier **aliases**: adep -s *.cpp *.hpp > aliases.
4. On évalue enfin le temps d'édition des liens, et les cycles potentiels: ldep -ddeps -aaliases

Utilisation pratique de idep suite

Quelques observations

Loption -s supprime les suffixes. Dans **aliases**, on regroupe chaque fichier d'implémentation avec le fichier en-tête correspondant (*Il faut donc faire attention à la première instruction include du fichier implémentation*). On peut vérifier cette mise en garde avec l'instruction:adep -v *.cpp. Ceci nous informe si certains fichiers *.cpp n'ont pas le même nom que le fichier *.hpp inclus en premier. Si ce choix est volontaire, il faut le renseigner dans le fichier *aliases* de sorte que l'instruction adep -v -aalises *.cpp n'affiche aucune plainte.

Sortie de ldep -ddeps -aaliases

- Les fichiers qui sont en dépendance cycliques.
- Les niveaux de hiérarchie pour tests unitaires incrémentaux des composants.
- Un résumé quantitatif du projet:
  - le nombre de cycles, de composants, de packages, de fichiers etc.
  - les valeurs des entités CDD, ACD, NCCD.

Pour une bonne architecture, NCCD ≤ 1.00