MAINTENANCE

2012-10-01

© Emmanuel Chieze
Département d'informatique, UQAM
INF3135

Plan

- ***Maintenance de logiciels***
- Débogage en C

Maintenance de logiciels

3

- Modification d'un logiciel déjà livré
  - correction d'erreurs
  - améliorations de fonctionnement (performance, changements dans la représentation des données, traitement de cas spéciaux nouveaux …)
  - modification/ajout de fonctionnalités plus ou moins importantes
  - amélioration de la qualité du logiciel (refonte du code)

Maintenance de logiciels

4

- 30 % des activités autour du logiciel concernent le développement de nouveaux logiciels
- 70 % concernent la maintenance
  - 20 % corrections de bugs
  - 25 % adaptations de logiciels
  - 55 % améliorations de logiciels

Maintenance de logiciels

5

- Chiffres probablement différents dans le cas de départements informatiques d'entreprises non informatiques
  - peu de développement de logiciels
  - intégration de logiciels existants
    - développement d'interfaces entre logiciels
    - développement de rapports et autres outils de consultation des données
- Activité très formatrice

Maintenance de logiciels

6

- Coût élevé de la maintenance
  - plus difficile d'ajouter un élément à un système en production qu'à un système en développement :
    - mesurer les impacts sur les opérations
    - limiter ces impacts

Maintenance de logiciels

7

- Coût élevé de la maintenance
  - les responsables de la maintenance n'ont pas nécessairement participé au développement. Temps d'apprentissage du logiciel aux niveaux
    - fonctionnel : comprendre le domaine d'application
    - organique : comprendre la structure du logiciel
    - technique : maîtriser le(s) langage(s) de programmation utilisé(s) et les styles de programmation
      - Exemple : un programmeur utilise systématiquement l'expression p & 1 pour vérifier qu'un nombre est impair (au lieu de p % 2)

Maintenance de logiciels

8

- Coût élevé de la maintenance
  - plus un programme est vieux, plus il a subi des activités de maintenance, plus il est complexe à modifier
  - néanmoins parfois préférable de garder un vieux système que de le changer
    - performance connue et satisfaisante du système
    - coût d'investissement trop élevé
    - risque trop élevé de changer de système

Maintenance de logiciels

9

- Procédure de maintenance
  - comprendre le logiciel (aux niveaux fonctionnel et organique)
  - comprendre les modifications demandées (au niveau fonctionnel)
  - proposer une ou plusieurs approches de mise en œuvre des modifications demandées
  - évaluer les impacts de la réalisation de ces modifications
    - les structures de données sont-elles affectées ? Comment ?
      - les impacts sur les BD sont plus complexes que ceux sur des fichiers appartenant exclusivement à l'application modifiée
    - quelle sections de code sont touchées ?
      - quels fichiers/quelles fonctions/quelles lignes de code ?

Maintenance de logiciels

10

- Procédure de maintenance
  - choisir la moins coûteuse/complexe à réaliser et celle qui sera la plus facile à maintenir à long terme
  - la réaliser
    - la programmer
      - respecter si possible le style de programmation et de documentation (commentaires) utilisé
      - sauf si code spaghetti
    - la mise à jour des plans de tests (unitaires/intégrés)
    - la réalisation des tests (non-régression + tests des nouvelles fonctionnalités)
    - la mise à jour de la documentation (documents d'analyse, documentation utilisateur)
    - la formation des utilisateurs aux nouvelles fonctionnalités
    - la coordination de la mise en production

Documentation de la maintenance

11

- En en-tête de chaque fichier modifié
  - Indiquer la date, l'auteur, la référence et le libellé de la modif
  - À la fin de la section en commentaires définissant le fichier
- Avant chaque section de code ajoutée (ligne, bloc ou fonction)
  - Ajouter un commentaire devant la modif spécifiant la date et l'auteur
  - Préciser le cas échéant les détails de la modif
- En cas de suppression ou de modification de sections de code
  - Il peut être pertinent de garder les anciennes versions en commentaires (en spécifiant la date et l'auteur)
- Et suivre les procédures spécifiques à l'organisation

Plan

12

- Maintenance de logiciels
- ***Débogage en C***

Débogage en C

13

- Problème :
  - reproduire le bug
  - identifier sa source
  - identifier la correction à apporter
  - la soumettre au processus d'approbation des demandes de modification de logiciels
    - il est possible que la modification ne soit pas réalisée
    - mais que l'on spécifie dans la documentation une limite du logiciel concernant le "bogue"
- Le débogage s'apparente à un travail de détective

Reproduire le bug

14

- Difficile lorsque le bug est mal décrit
- Surtout lors de l'utilisation interactive d'applications
  - utilisateurs décrivant de façon imprécise ou erronée le contexte d'apparition du bug
    - par manque de connaissance/de compréhension
    - par manque de rigueur
- Aide possible : prévoir un mécanisme de journalisation des entrées de l'utilisateur
  - avec mécanisme de mise à zéro du journal
  - permet de reproduire le comportement de l'utilisateur à distance

Identifier la source du bogue

15

- Débogage par affichage de messages, activé au moment de la compilation (1)
  - afficher le contenu de variables
  - afficher une trace permettant de vérifier l'ordre d'exécution des lignes du programme
  - maintenir une seule version du programme : utiliser les macro-variables et/ou macro-fonctions du préprocesseur à ces fins

Identifier la source du bogue

16

- Débogage par affichage de messages, activé au moment de la compilation (2)

```
#ifdef DEBUG_ON
   printf("…",var1,var2,var3);
#endif
```

  - ou

```
#ifdef DEBUG_ON
   #define DEBUG(x) x
#else
   #define DEBUG(x)
#endif
DEBUG(printf("…",var1,var2,var3));
```

  - et compiler avec l'option -DDEBUG_ON
  - on peut afficher le nom du fichier et le numéro de ligne (variables spéciales du préprocesseur __FILE__ et __LINE__)

```
printf ("%s : %d : i = %d\n", __FILE__, __LINE__, i);
```

Identifier la source du bogue

17

- Débogage par affichage de messages, activé au moment de la compilation (3)
  - et compiler avec l'option -DDEBUG_ON
  - on peut afficher le nom du fichier et le numéro de ligne (variables spéciales du préprocesseur __FILE__ et __LINE__)

```
printf ("%s : %d : i = %d\n", __FILE__,
  __LINE__, i);
```


Identifier la source du bogue

18

- Débogage par affichage de messages à l'utilisation d'une option de débogage sur la ligne de commande
  - une seule compilation nécessaire
  - on peut demander au client d'exécuter le programme avec l'option de débogage et d'envoyer la sortie obtenue pour analyse
  - on peut avoir plusieurs options complémentaires de débogage en cas de programme complexe

Identifier la source du bogue

19

- Autres approches
  - utiliser un débogueur interactif : plus complexe, mais parfois inévitable
    - à utiliser après avoir essayé l'approche précédente
    - combiner alors les deux approches
      - identifier la zone concernée par affichage de messages
        - exemple : identifier la fonction qui plante
      - utiliser le débogueur interactif sur cette zone
- Dans tous les cas, s'assurer de connaître la version de logiciel de l'utilisateur et de tester la bonne version !

Identifier la source du bogue

20

- Selon le type de comportement du programme
  - segmentation fault : erreur sur les pointeurs, généralement liée à la non-allocation d'une zone mémoire référencée par un pointeur
  - bus error : problème d'E/S (typiquement avec scanf)
  - stack overflow : peut être lié à une récursion infinie

Identifier la source du bogue

21

- Attention : un programme C qui plante ne vide pas ses buffers
  - certains printf peuvent avoir été exécutés sans que les résultats ne soient visibles
  - rajouter au besoin des DEBUG(fflush(stdout);) pour forcer le vidage de la sortie standard à intervalles réguliers dans le code
  - idem lorsque l'on écrit dans un journal

Approche systématique

22

- Prendre le temps de réfléchir au problème, de le reproduire et d'être certain de l'avoir compris avant de modifier le programme
  - On n'a pas nécessairement écrit soi-même le code que l'on débogue
  - Inspection manuelle du code nécessaire pour le comprendre
  - grep peut aider à comprendre le code
    - permet de trouver toutes les occurrences d'une variable ou d'une fonction dans un ou plusieurs fichiers à travers plusieurs répertoires

Approche systématique

23

- Lorsque vous corrigez une erreur, vérifier si cette erreur se répète dans le code, et corrigez-en chaque instance
  - ne pas attendre qu'un test (ou un utilisateur) repère chaque instance de la même erreur
  - utilisation d'outils comme grep, find

```
grep –n toto *.c
find . –name *.c –print –exec grep –n toto {} \;
```


Approche systématique

24

- Lors du développement, corrigez chaque erreur trouvée au fur et à mesure
  - ne pas ignorer de comportement anormal même si ce dernier est rare
  - une correction de bogue en cours de développement est moins coûteuse qu'en mode maintenance

Approche systématique

25

- Il est parfois utile d'expliquer son code à quelqu'un d'autre (pas nécessairement un programmeur !)
  - ce faisant, il n'est pas rare que l'on trouve son erreur
- Il peut aussi arriver que le bogue soit dans le programme de tests, pas dans le code de l'application !

Bibliographie

26

- Maintenance :
  - G. Tremblay, 2004. *Notions de base de la maintenance de logiciels.*