

CNAM - CRA Nancy
2003

Génie Logiciel

Jacques Lonchamp

HUITIEME PARTIE

**Les ressources :
outils, aspects organisationnels et humains.**

Nous passons en revue ci après les grandes catégories d'outils de base.

a) Les outils d'édition

- Editeurs de texte : classiques ou syntaxiques

Exemple d'édition syntaxique : deux raffinements d'un programme

```
read(a, b, c) ;
d := b ** 2 - 4 * a * c ;
if <cond> then
  <instr>
else
  <instr>
endif ;
<instr> ;

read(a, b, c) ;
d := b ** 2 - 4 * a * c ;
if d >= 0 then
  -- racines réelles
  s := sqrt(d) ;
  rac1 := (-b + s)/(2*a) ;
  rac2 := (-b - s)/(2*a) ;
else
  <instr>
endif ;
<instr> ;
```

- Editeurs graphiques : dessin libre (Draw), dessins prédéfinis (Visio), dessins typés avec vérifications (Merise, UML, ...), méta éditeurs (permettant de définir de nouveaux symboles, de nouveaux contrôles, ...).

b) Les outils de programmation

- Assembleurs, compilateurs, interprètes, assembleurs ou compilateurs croisés, éditeurs de liens, pré-processeurs, ...
- Débugueurs.
- Générateurs de code, restructureurs de code (COBOL à COBOL structuré), 'pretty-printers' (mise en page selon syntaxe), générateurs de compilateurs (Lex pour analyse lexicale, Yacc pour analyse syntaxique).

c) Les outils de vérification

- Analyseurs statiques (lint, purify, ...).

Exemple de sortie de l'analyseur lint : alors que le compilateur ne trouve pas d'erreurs, l'analyseur statique propose des avertissements.

```
138% more prog.c
#include <stdio.h>
printarray (Anarray)
  int Anarray;
{
  printf("%d",Anarray);
}
main()
{
  int Anarray[5]; int i; char c;
  printarray(Anarray, i, c);
  printarray(Anarray);
}
139% cc prog.c
140 lint Prog.c
prog.c(10): warning: c may be used before set
prog.c(10): warning: i may be used before set
printarray: variable # of args. prog.c(4) :: prog.c(10)
printarray, arg 1 used inconsistently prog.c(4) :: prog.c(10)
printarray, arg 1 used inconsistently prog.c(4) :: prog.c(11)
printf returns value which is always ignored
```

- Analyseurs dynamiques, outils de mesure (métriques sur le code ou la conception), outils de monitoring (métriques en cours de fonctionnement).

- Comparateurs de fichiers (diff).

Exemple de sortie de diff :sur 2 fichiers (fich1.aa et fich2.aa)

```
X : integer = 20           X : integer = 20
Y : integer = 17           Y : integer = 18
Z : integer = 34           Z : integer = 34
name :charray = 'J Smith'  name :charray = 'F Jones'
```

```
diff fich1.aa fich2.aa
2c2
< Y : integer = 17
---
> Y : integer = 18
4c4
< name :charray = 'J Smith'
---
> name :charray = 'F Jones'
```

- Emulateurs, simulateurs.
- Prouveurs de programmes (assistants de preuve).
- Générateurs de tests (pour un code donné), gestionnaires de test (gestion des jeux de données et des résultats).
- Outils de 'reverse engineering' (code à conception, code C++ à UML).

d) Les outils de *gestion de version et de gestion de configurations*

- Gestionnaires de configuration (SCCS, RCS, CVS, ClearCase, Continuous, ...) : ce sont peut être *les outils les plus importants pour le développement des gros logiciels*. Ils permettent à plusieurs développeurs de stocker et de partager les mêmes objets en gardant la trace de leurs évolutions : programmes sources, documentations, makefiles, etc. Les outils de base comme SCCS et RCS s'appuient sur le modèle du 'check in-check out'. Le modèle du 'check in-check out' exploite un *référentiel partagé* (centralisé, éventuellement répliqué et/ou partitionné) qui stocke des fichiers *multi versionnés*, et des *espaces de travail privés* qui sont des sous ensembles mono versionnés du référentiel comme par exemple des répertoires peuplés de fichiers (cf. Figure 2).

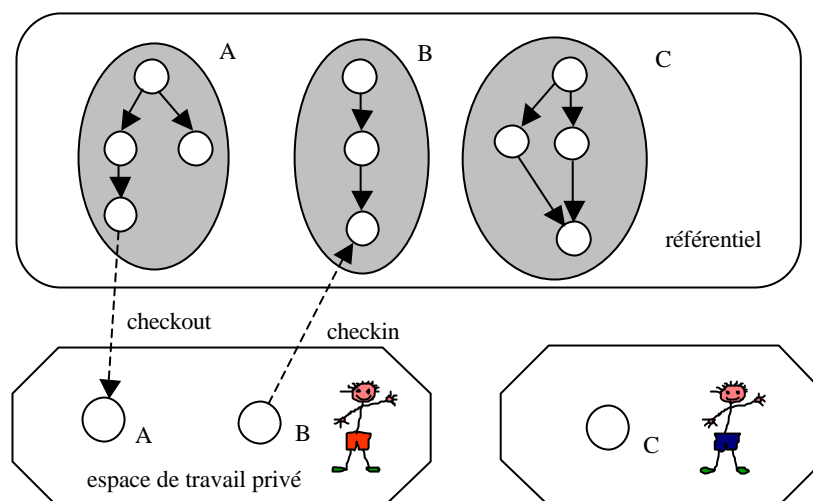


Figure 2 le modèle du check in – check out.

Les versions sont organisées selon un graphe direct acyclique, avec une branche principale et des branches annexes (cf. Figure 3).

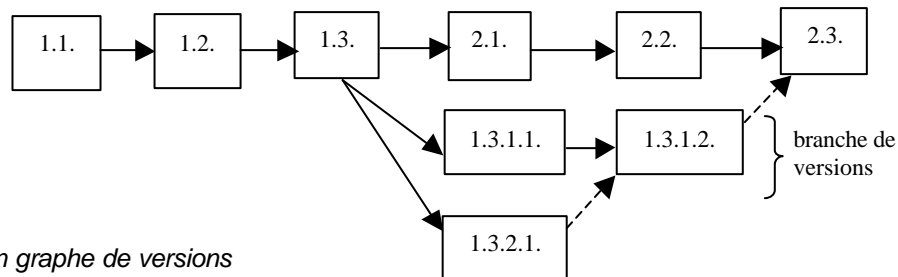


Figure 3 Un graphe de versions

L'utilisateur dispose d'opérateurs de transfert entre le référentiel et son espace de travail :

- *checkout* : création d'un fichier dans l'espace de travail à partir d'un nom de fichier et d'un numéro de version,
- *checkin* : création d'une nouvelle version dans le référentiel à partir d'un nom de fichier dans l'espace de travail.

Les accès concurrents sont gérés par des *verrous en lecture et en écriture*. Lorsqu'un fichier est extrait en écriture (*checkout* en mode WRITE), l'utilisateur est assuré de pouvoir créer la prochaine version dans la même branche du référentiel.

Ce mode de fonctionnement permet :

- le *développement parallèle* : un même fichier peut être développé de plusieurs manières différentes dans des branches de versions séparées. Une branche de versions peut être ultérieurement fusionnée avec une autre ou abandonnée.
- la *gestion des mises à jour conflictuelles* : si un fichier est en cours de modification, il est verrouillé. Un second utilisateur peut effectuer une mise à jour en créant une nouvelle version dans une autre branche. Elle sera fusionnée à la version principale ultérieurement.

Ce mode de fonctionnement permet toujours la lecture des objets. Par contre, *il n'assure pas la correction des exécutions concurrentes*. Un utilisateur peut lire des objets liés incohérents si l'un a été modifié et remis dans le référentiel et l'autre est encore en cours de modification. L'utilisateur risque alors d'accéder à la version mise à jour du premier objet et à l'ancienne version du second. Par ailleurs, les *opérations de fusion de version peuvent être très complexes* même si certains outils d'assistance existent pour fusionner des textes (inspiré de l'outil 'diff' sous Unix).

Les outils plus sophistiqués que SCCS ou RCS (comme CVS) permettent le transfert atomique de *configurations cohérentes de fichiers* au lieu du transfert de fichiers isolés.

- Constructeur d'application (Make)
 - Il permet de (re)construire automatiquement un ou des fichiers cibles à partir de fichiers origines en exécutant des commandes lorsque l'ancienneté des fichiers l'exige. Le fichier makefile décrit les dépendances.
 - Exemple : le système sys est reconstruit par une édition de liens de 2 fichiers objet, chaque fichier objet étant construit par compilation d'un fichier C avec une librairie. La commande `make sys` reconstruit sys en exécutant les commandes nécessaires pour mettre à jour les .o et sys, si nécessaire.

```

sys : mod1.o mod2.o
      ld mod1.o mod2.o -o sys
mod1.o : mod1.c incl.h
      cc -c mod1.c
mod2.o : mod2.c incl.h
      cc -c mod2.c

```

- Gestionnaires de changement : gère la prise en compte des demandes de modification ou 'change requests', leur études, les décisions qui en résultent et les réalisations éventuelles des changements.

e) Les outils de *gestion de projet et de productivité individuelle ou collective*

- Estimation des coûts (Cocomo),
- Planification (PERT, Gantt, ...),
- Travail collectif ('groupware' ou collecticiel: courrier, forums, agendas, 'notebooks', éditeurs partagés, téléconférence, ... ; 'workflow')
- Productivité individuelle (tableurs, documentation hypertexte, ...).

2. Aspects organisationnels et humains

La gestion de projet inclut de nombreuses activités telles que :

- L'écriture des propositions de projets.
- L'estimation des coûts des projets.
- La planification et l'ordonnancement des projets.
- Le suivi et l'évaluation des projets.
- La sélection et l'évaluation des personnels ; l'organisation des équipes.
- La rédaction des rapports de gestion.

a) L'organisation des équipes

Les structures varient beaucoup selon la taille des organisations. On a souvent un *découpage par projets* (applications).

Exemple :

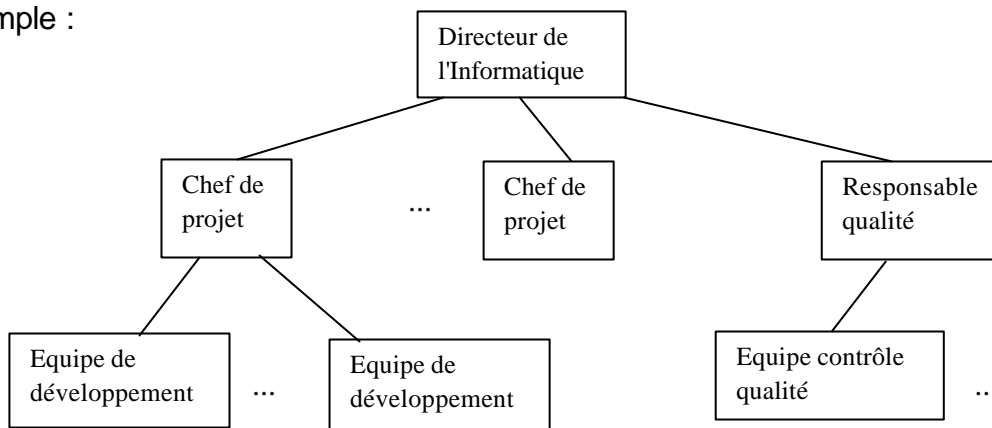


Figure 4 Découpage par projets

On peut aussi par exemple avoir un *découpage par centres de compétences* (analyse, conception, programmation, test et intégration, maintenance).

Les équipes de développement peuvent être organisées de manière informelle sans chef ('démocratique' ou 'egoless programming team') ou de manière très structurée (autour d'un 'chief programmer', d'un bibliothécaire, d'un testeur, ...).

Les mesures de productivité individuelle des développeurs en termes de ligne de code produites sont à prendre avec beaucoup de précautions (influence du langage, de la difficulté du code, non prise en compte de la qualité, etc).

b) La planification

La planification implique la *décomposition* organisationnelle du projet ('work breakdown structure'). Puis la *planification* proprement dite, par construction de diagrammes (Gantt, PERT) permettant de prévoir les dates des évaluations d'avancement ('milestones'), et de trouver les tâches critiques.

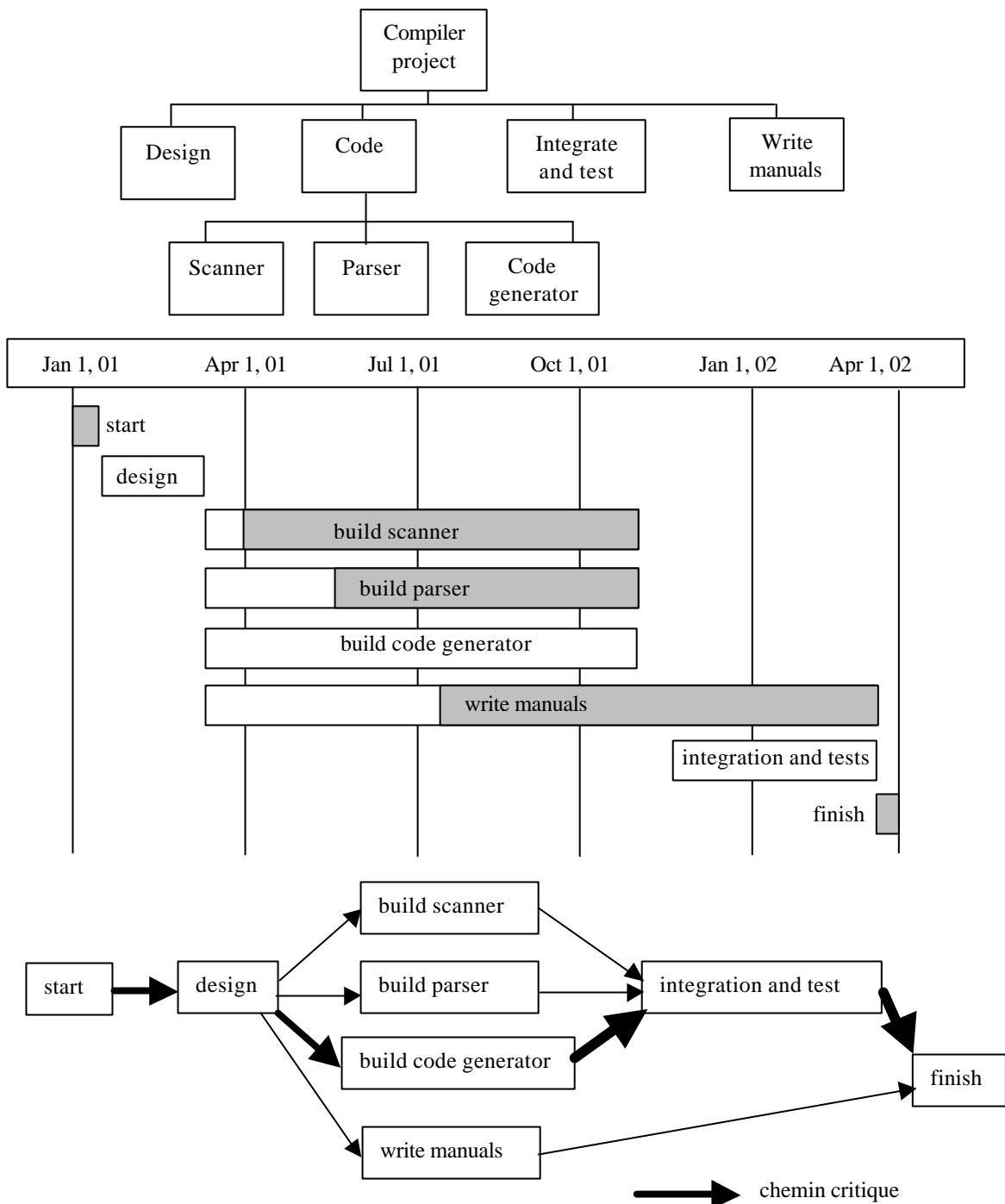


Figure 5. Décomposition organisationnelle, diagramme de Gantt et PERT

c) L'estimation des coûts

Le modèle COCOMO de B. Boehm est le plus connu. A un premier niveau, il fournit des équations estimant l'effort (ou charge) en hommes/mois en fonction de la nature du projet ((1) projet classique pour l'équipe de développement, (2) projet avec des aspects innovants, (3) projet complexe et complètement innovant).

Ces équations ont été obtenues empiriquement à partir de statistiques sur des projets réels. Elles sont de la forme : effort = A (KIL)^b, où KIL est le nombre d'instructions livrées en milliers.

Effort en HM (hommes mois)	Effort en mois de développement T
(1) HM = 2,4 KIL ^{1,05}	(1) T = 2,5 HM ^{0,38}
(2) HM = 3 KIL ^{1,12}	(2) T = 2,5 HM ^{0,35}
(3) HM = 3,6 KIL ^{1,20}	(3) T = 2,5 HM ^{0,32}

Le nombre de personnes nécessaires

$$N = HM / T.$$

Exemple : pour un projet complexe (3) de 128000 lignes : HM = 1216, T = 24 mois, et N = 51 personnes.

Ces résultats bruts peuvent ensuite être affinés en fonctions de 14 paramètres qui influencent la productivité (coefficients pour les valeurs très bas, bas, normal, haut, très haut, extrêmement haut) : expérience du langage de programmation, contraintes de délai, volume des données, expérience du système d'exploitation, outils de développement, contraintes de temps de réponse, contraintes de sûreté de fonctionnement, complexité, compétence des individus, etc.

Le résultat doit être pris comme une base de réflexion.

Une version révisée (COCOMO II) date de 97. Elle part d'une estimation du nombre d'écrans, d'éditions et de modules. Chaque élément est classé en simple, moyen ou complexe et les points suivants sont donnés ('points d'objets') :

	Simple	Moyen	Complexe
Ecran	1	2	3
Edition	2	5	8
Module	0	0	10

Le total des points (CPO) est recalculé pour tenir compte du % de réutilisation :

$$NPO = CPO * (100 - \%réutilisation)/100$$

Le taux de productivité PROD est calculé via le tableau suivant (addition des 2 valeurs) :

Expérience et capacité des concepteurs	très faible	faible	moyenne	grande	très grande
Maturité et capacité des outils	très faible	faible	moyenne	grande	très grande
PROD	4	7	13	25	50

Le coût en HM (Hommes/Mois) HM = NPO / PROD

COCOMO II possède également des versions plus détaillées.

Il faut noter en conclusion que la notion d'homme/mois a été critiquée de manière pertinente par Brooks dans son livre 'The mythical man-month'. L'idée sous jacente est que *les hommes et les mois sont interchangeables* (2 hommes pendant un mois équivaut à un homme pendant deux mois). Ceci n'est vrai que lorsque le problème est partitionnable en sous problèmes *indépendants*. Dès qu'il y a nécessité de *communication* l'effort pour assurer cette communication doit être pris en compte et il croît *non linéairement* (pour N personnes l'effort de communication e est proportionnel à $N(N-1)/2$ – ex : N=2 e=1, N=3 e=3, N=4 e=6, N=5 e=10, ...).

3. Conclusions du cours de GL

Le résultat de 30 années d'évolution du génie logiciel est contrasté :

- Comme nous l'avons vu dans l'introduction, beaucoup d'imperfections subsistent et le développement d'un gros projet logiciel reste une aventure *risquée et coûteuse*.
- Par ailleurs, beaucoup de *révolutions* annoncées se sont avérées de *peu d'influence pratique* (généralisation des méthodes formelles, intelligence artificielle, environnements intégrés guidant activement les concepteurs, langages de programmation universels -ADA- ou permettant l'industrialisation des composants logiciels -objets- ...).

Cependant, les informaticiens *font malgré tout face à la complexité en croissance continue des applications*, qu'elles soient classiques en gestion, ou qu'il s'agisse d'applications avancées (systèmes temps réel, embarqués, distribués sur des réseaux). Ces applications avancées deviennent monnaie courante.

Cette évolution est rendue possible par des progrès lents mais réels dans tous les domaines : principes, concepts et méthodes, techniques (langages), outils. On peut parler de *lente maturation* (cf. le temps mis par unix et C/C++ pour se diffuser !). Cette évolution ne peut aller plus vite, car l'évolution des *connaissances des informaticiens du terrain* est un important *facteur limitant*.