heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'Ingénieurs de l'état de Vaud
ERIC LEFRANÇOIS
1er Février 2018

Génie Logiciel

Chap 1&2 - Problématique

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Sommaire

1 En préliminaire

LES OBJECTIFS DU COURS

Les objectifs du cours sont principalement les suivants :

- Acquérir la capacité de créer des logiciels correctement conçus, robustes et maintenables en utilisant des technologies objets avec des langages tels que C#, Ruby, Java ou encore C++.
- Introduire à l'analyse et à la conception objet, en basant le discours sur la notation **UML**, le processus de développement unifié (**UP** - Unified Process) et les méthodes agiles, SCRUM en particulier. L'accent sera placé sur l'apprentissage des concepts fondamentaux. Cela va donc bien au-delà d'un simple apprentissage du langage UML, qui n'est après tout qu'une simple notation.

En particulier:

- *Du point de vue d'analyse* : acquérir les connaissances permettant de mener à bien l'étape **d'analyse des besoins** avec la spécification des cas d'utilisation.
- *Du point de vue de conception :* connaître et savoir appliquer des modèles de conception (**design patterns**) qui aideront à établir quelles responsabilités confier à tel ou tel objet, à établir de quelle manière les objets vont interagir.
- *Du point de vue de méthode :* introduire l'étudiant à une méthode, une marche à suivre, pouvant être mise en œuvre par une équipe de développeurs dans le cadre de la réalisation d'un logiciel. Il s'agira notamment de la méthode dite **processus unifié** (UP - Unified Process), l'archétype de la démarche incrémentale itérative.

Cette méthode, bien loin d'être reconnue comme «**la** méthode », constitue néanmoins une référence et un cadre intéressant pour introduire les concepts fondamentaux qui - une fois maîtrisés - peuvent être appliqués dans d'autres démarches.

D'autres approches seront présentées : les méthodes dites « agiles », et en particulier celle qui a le mérite d'être le plus en vogue : la méthode **SCRUM**. Cette dernière sera mise en pratique dans le cadre du laboratoire (mini-projet).

 Remarquons que le fait d'être passé maître dans la manipulation d'un marteau ne fait pas de nous un bon architecte : la connaissance et la pratique d'un langage de programmation ne suffit pas.

L'expérience fera le reste ! 

En bref, les objectifs du cours peuvent être résumés comme suit :

- Appliquer des principes et des modèles de conception pour créer de meilleures conceptions objet
- Mener à bien un certain nombre d'activités fondamentales comme l'analyse et la conception, dans le cadre d'une démarche basée sur le processus unifié, pris à titre d'exemple.
- Mettre en œuvre les diagrammes de modélisation les plus courants en utilisant la notation UML

---

2 La problématique

Quelques devinettes...

☺ Pour vous mettre sur la voie, voici une première série de devinettes.

- Y a-t-il une différence entre :
  - Construire une cabane au fond du jardin ;
  - Construire le bâtiment de l'HEIG-VD ?
- Quel projet coûte le plus cher ?
- Quel projet est le plus long à réaliser ?
- Quel projet nécessite le plus de personnes pour sa réalisation ?
- Dans quel projet la phase de construction est-elle proportionnellement la plus longue (par rapport à la durée complète du projet) ?

Une deuxième série...

Le bâtiment de l'HEIG-VD a été construit aux environs de 1975. À cette époque, il correspondait aux besoins de l'école.

Peut-on dire que ce bâtiment correspondrait encore à nos besoins d'aujourd'hui ?

- Le bâtiment n'est pas modulaire ;
- Les normes de l'époque ne conviennent plus.

La taille des bâtiments a une grande importance sur leur réalisation.
Est-ce la même chose en informatique ?

✍ Cette petite introduction nous a permis de mettre en évidence l'importance considérable que peut prendre la **taille** même du logiciel à développer. Aussi, nous allons examiner de plus près ce qui distingue un « petit logiciel » d'un «gros logiciel»..

Les petits logiciels

- Le cahier des charges est petit, facile à comprendre et même parfois transmis oralement.
- La conception du programme se fait souvent mentalement sans documentation.
- Le coût du logiciel est faible ou négligeable.
- Le choix des technologies (système d'exploitation, environnement de développement, choix des machines, etc.) est opéré par le développeur.
- Le développement est réalisé personne et dure peu de temps, parfois quelques semaines, au plus quelques mois, d'ailleurs aucune estimation préalable du temps de développement n'est faite le cas échéant.
- Le logiciel est rarement documenté, son usage est facile.
- Le logiciel sera déployé une ou deux fois, par le développeur lui-même.
- La maintenance du logiciel cesse de facto lorsque le développeur cesse de le supporter (départ de la société, changement d'affectation, etc.). Cela entraîne souvent la mort du logiciel.

☺ Toutes les conditions pour le succès du développement sont donc réunies !

Les gros logiciels

- Le cahier des charges est volumineux. Il est même parfois:
  - Difficile à comprendre ;
  - Mal écrit, le client n'a pas une idée très claire du résultat final, il y manque beaucoup d'informations ;
  - Écrit par des personnes qui ne connaissent pas grand-chose à l'informatique.
- En général les développeurs ne connaissent pas ou peu le métier où s'applique le logiciel.
- La conception du programme est un vrai casse-tête, pleine d'embûches, à cause des incertitudes sur le cahier des charges et de la taille du programme à réaliser.
- Le coût du logiciel est énorme.
- Le développement se fait par une équipe, parfois assez grande, il peut durer des années.
- Une estimation préalable du temps de développement est importante pour pouvoir estimer les coûts du logiciel.
- L'équipe de développement devrait pouvoir respecter les délais de développements estimés, c'est un défi.
- Le logiciel doit être parfaitement documenté. Cette documentation doit être livrée en même temps que le logiciel, il faut donc développer simultanément le logiciel et sa documentation.
- Le logiciel n'est pas déployé par l'équipe de développement :
- Il est peut-être livré sur CD et diffusé en grand nombre ;
  - Il est peut-être livré à très petite échelle, mais son installation est compliquée. Des personnes spécialement formées sont chargées du déploiement.
  - La maintenance du logiciel est gérée par un contrat, renouvelé souvent d'année en année. Le logiciel doit survivre à l'équipe de développement

☹Toutes les conditions pour l'échec du développement sont réunies.

✋Posez-vous seulement la question.

Dans quelle catégorie vous situez-vous lorsque vous développez un projet informatique :

- Dans le cadre scolaire ?
- Dans le cadre professionnel ?

✍ Maintenant, un petit exercice...

Combien coûte une ligne de code, sachant que :

- Un bon développeur coûte Frs 200'000.- par année à son entreprise tous frais com-pris ;
- Il code en moyenne 20 lignes par jour ;
- Il est absent en moyenne 1/5 du temps (vacances, armée, maladie, etc.);
- Il y a environ 250 jours ouvrables par année ;

Quelques définitions clés

Voici quelques définitions clés du génie logiciel, sur lesquelles nous reviendrons, mais qu'il est nécessaire d'appréhender d'ores et déjà afin de lire plus aisément ce qui va suivre dans le cadre de cette introduction sur le génie logiciel.

- **Spécification**
  Activité consistant à décrire formellement ou semi-formellement le fonctionnement d'un système. On doit répondre aux questions : que doit faire le système ? Pourquoi ?
- **Conception**
  Activité consistant à réfléchir, modéliser et décider comment le système va être implémenté. On doit répondre à la question : comment réaliser le système ?
- **Implémentation**
  Activité consistant à coder le système
- **Test**
  Activité consistant à exécuter le système ou ses parties pour vérifier qu'il fonctionne correctement ou pour découvrir ses anomalies
- **Maintenance**
  Le logiciel est en service chez le client. Cette activité comprend la correction d'erreurs détectées par le client, mais aussi et surtout l'ajout de nouvelles fonctionnalités ou encore l'adaptation, la modification de fonctionnalités existantes.

2.1 LA CRISE DU LOGICIEL DES ANNÉES 70

Avec l'accroissement de la complexité des logiciels (notamment de leur taille), le développement ne suit pas : il faudra 10 ans pour développer l 'OS 360 des IBM 360

Figure 1. Rapport du Congrès Américain en 1979 sur 487 projets

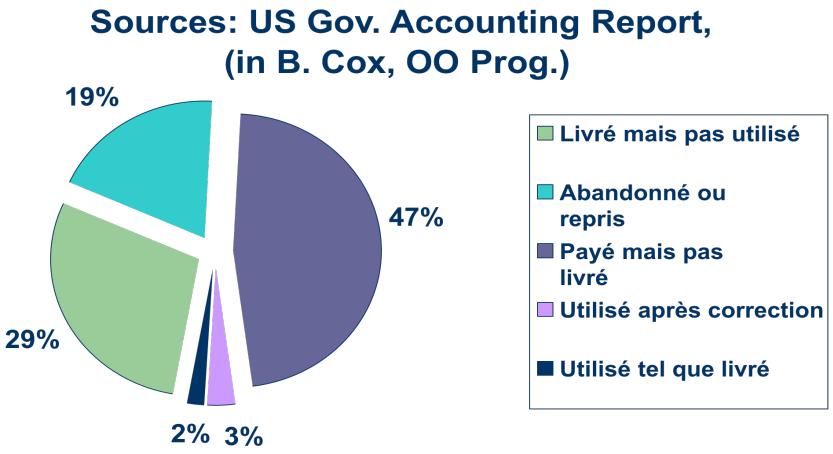
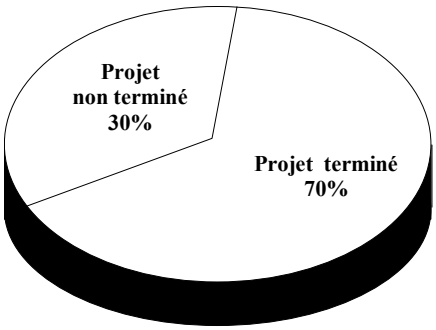


Figure 2. Un rapport américain de 1994 - Standish Group International



- Les projets analysés ont tous utilisé le modèle en cascade (waterfall) que l'on présentera un peu plus loin dans le cours.
- 53% des projets ont dépassé le budget prévu initialement de 200% !
- Environ $81 milliards dépensés pour des projets abandonnés aux USA en 1995

Panique à bord du bateau logiciel

La qualité du développement laisse franchement à désirer...

Du point de vue utilisateur

- Les logiciels réalisés ne correspondent pas souvent aux besoins des utilisateurs ;
- Les changements de besoin du client sont difficiles à intégrer dans le développement ;

Du point de vue de l'équipe de développement

- La maintenance des logiciels est complexe et coûteuse
- Trop d'erreurs constatées
- La performance du système est souvent inacceptable
- Les logiciels développés sont rarement portables d'un système à l'autre

Du point de vue gestion et planification

- Le système est difficilement réutilisable pour de futurs développements
- Coûts imprévisibles et généralement prohibitif
- Délais généralement dépassés

Et pourquoi donc ?

Car le logiciel est une tâche complexe...

Reconnaissons tout d'abord que ça n'est pas une tâche facile...

Une quantité de facteurs influent de manière négative.

- Taille et complexité du système à informatiser.
- Complexité attachée à l'environnement du système.
- Complexité de communication entre les futurs utilisateurs (diversité des points de vue)
- Complexité de communication entre les futurs utilisateurs d'une part et les développeurs d'autre part (vocabulaire technique et non-technique)

Car l'évolution est trop rapide...

Figure 3. Evolution du matériel et du logiciel

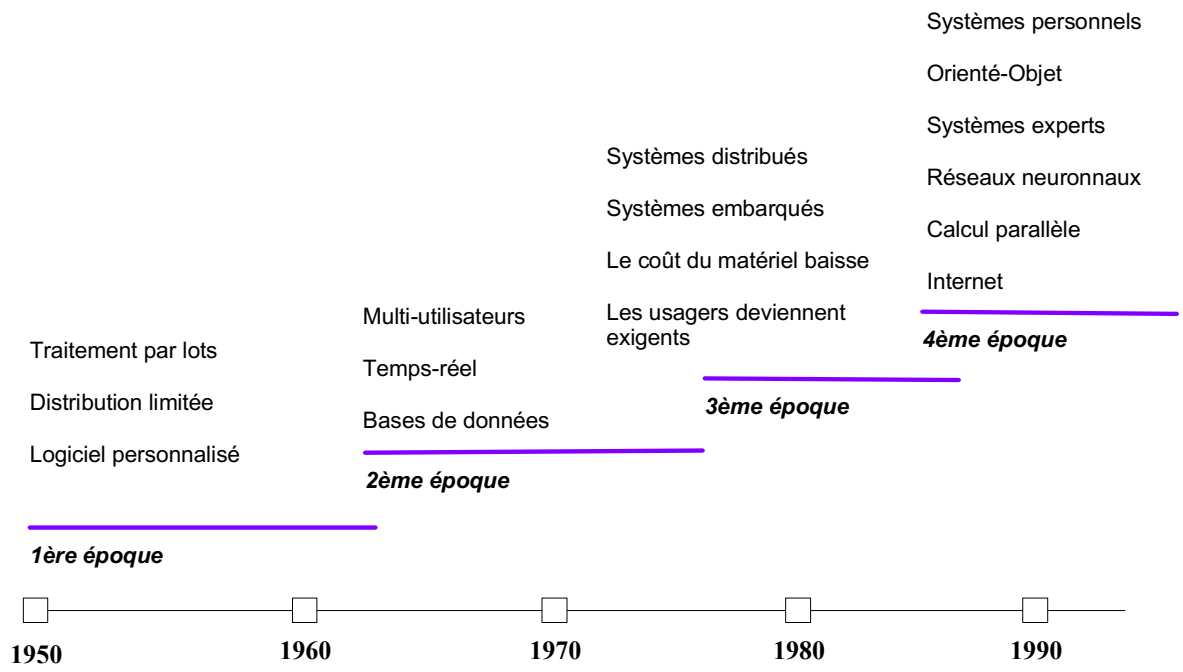
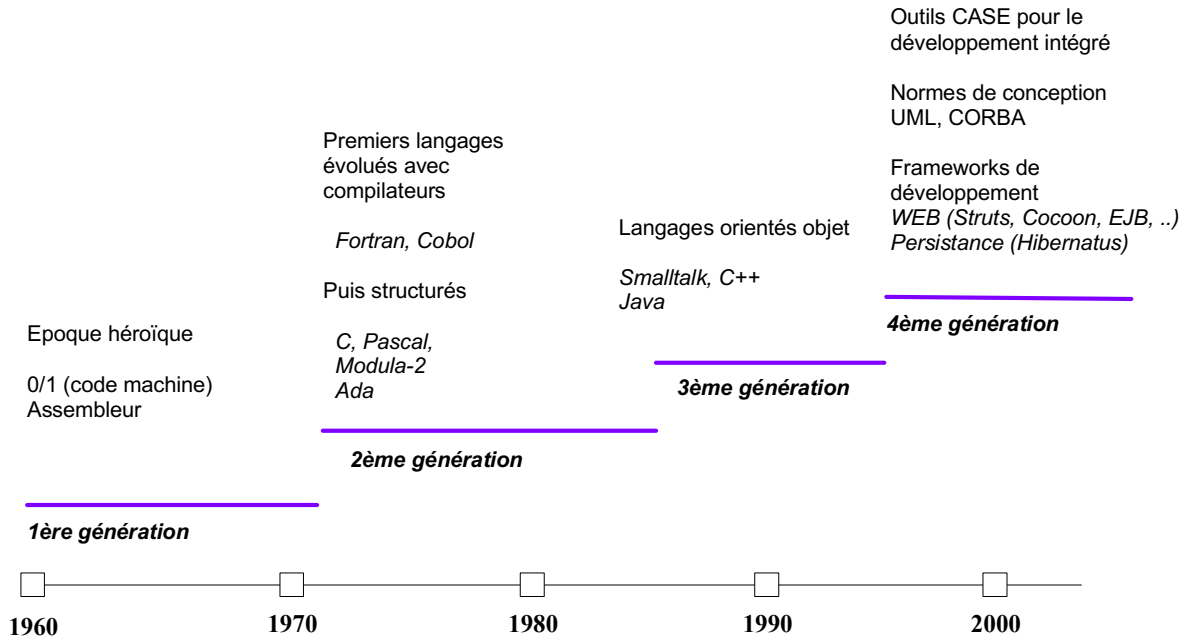


Figure 4. Evolution des langages et des outils



Car au-delà des mythes, la réalité...

Les mythes du côté usager

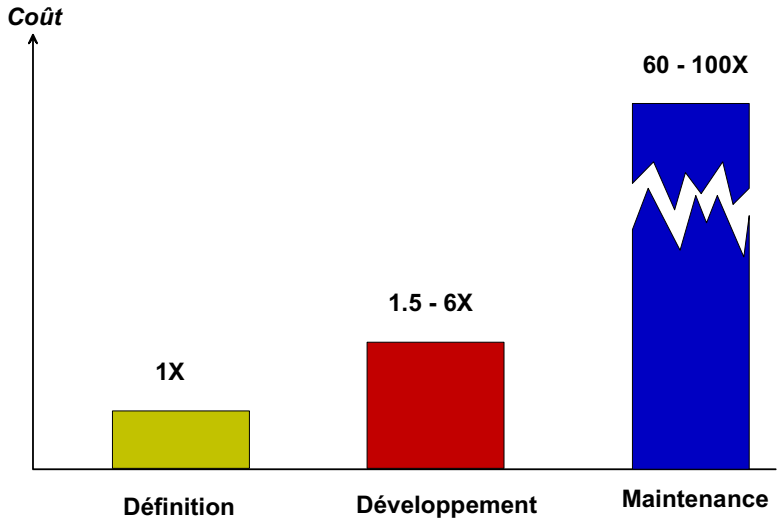
✍Un énoncé général des objectifs est suffisant. On verra pour les détails plus tard !

☹***La réalité :*** Une définition insuffisante des besoins des usagers est une cause majeure de production d'un logiciel de mauvaise qualité

✍Les besoins du projet changent, mais on incorporera les modifications facilement parce que le logiciel est flexible !

☹***La réalité :*** Les coûts pour un changement du logiciel augmentent de manière dramatique dans les dernières phases du développement.

Figure 5. Le coût du changement



Les mythes du côté développeur

✍Une fois que le programme est écrit et qu'il fonctionne, le travail du développeur est terminé !

☹***La réalité :*** 50% à 70% de l'effort consacré à un programme se produit après la livraison à l'usager.

✍Tant qu'un programme ne fonctionne pas, il n'y a pas moyen d'en mesurer la qualité !

☹***La réalité :*** les revues de logiciel peuvent être plus efficaces pour détecter les erreurs que les jeux de tests.

✍Le succès d'un projet tient essentiellement de la livraison d'un programme fonctionnel !

☹***La réalité:*** Une configuration logicielle inclut toute la documentation, des données d'entrée pour les tests, etc..

Les mythes du côté gestionnaire

✍L'entreprise possède des normes, le logiciel développé devrait être satisfaisant !

☹***La réalité:*** les standards sont-ils utilisés, appropriés et complets?

✍Les ordinateurs et les outils logiciels que l'entreprise possède sont suffisants

☹***La réalité:*** il faut plus que des outils pour réaliser du logiciel de qualité. Il faut aussi une bonne pratique.

✍Si le projet prend du retard, il suffira d'ajouter quelques programmeurs.

☹***La réalité :*** Le développement du logiciel n'est pas une activité mécanique. Ajouter des programmeurs peut empirer la situation

Naissance du génie logiciel

- 7 au 11 octobre 1968, conférence Garmish-Partenkirchen (sponsorisée par l'OTAN)
  Le terme de « Software engineering » est utilisé pour la première fois par les professeurs Bauer et Bolliet, dans un rapport de la conférence scientifique.
- La traduction française « Génie logiciel » est plus tardive.
- Le langage Ada est une conséquence directe de cette crise. C'est la réponse de Département de la Défense américaine à ce problème.
- Puis arrivent les premières méthodes : VDM (Vienna Development Method) date du début des années 70, le langage Z (Spécification formelle - Abrial) date de 1978.

Points marquants de l'histoire du Génie Logiciel

Années 1960 ⇨ balbutiements

- Mise en évidence du problème
- Crise du logiciel
- Nato Meetings (Conférences Otan) : Garmish-Partenkirchen & Rome
  Rome ⇨ Guerre au « Goto »
  Garmish-Partenkirchen ⇨ « Software Engineering » (1968)

Années 1970 ⇨ Recherche d'un modèle (Outils, Gestion)

Début 70 :	« Structured Programming » , inspiré par Pascal
Milieu 70 :	Notion de cycle de vie ⇨ Cycle en cascade
Fin 70 :	Méthodologies de spécification ⇨ « Structured Analysis »
	Méthodologies de conception ⇨ « Structured Design »

Années 1980 ⇨ Rassemblement, cohésion

- Méthodes de développement globales : OMT, Booch, Fusion… puis UP
- Amélioration des outils : POO notamment
- On constate des gains de productivité

Années 2000 ⇨ Retour à la souplesse

- Méthodes de développement XP, SCRUM

Définition du génie logiciel

Définition 1: Le génie logiciel

Discipline d'ingénierie concernée par le problème pratique du développement de **grands systèmes** logiciels.
Sommerville Ian « Le génie logiciel », Addison-Wesley France, 1992

Cette première définition met en avant la problématique liée à la taille du projet. La deuxième est un peu plus précise quant aux objectifs de cette discipline.

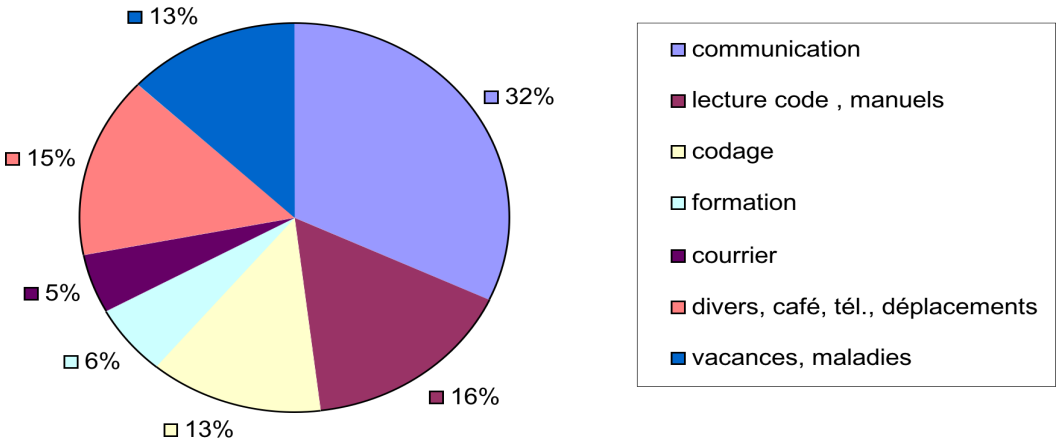
Définition 2: Le génie logiciel

Discipline pour spécifier, construire, distribuer et maintenir des logiciels, en assurant :

- La qualité (fiables, adaptés, conviviaux, évolutifs)
- Des coûts contrôlés
- Des délais garantis

La productivité du développement logiciel

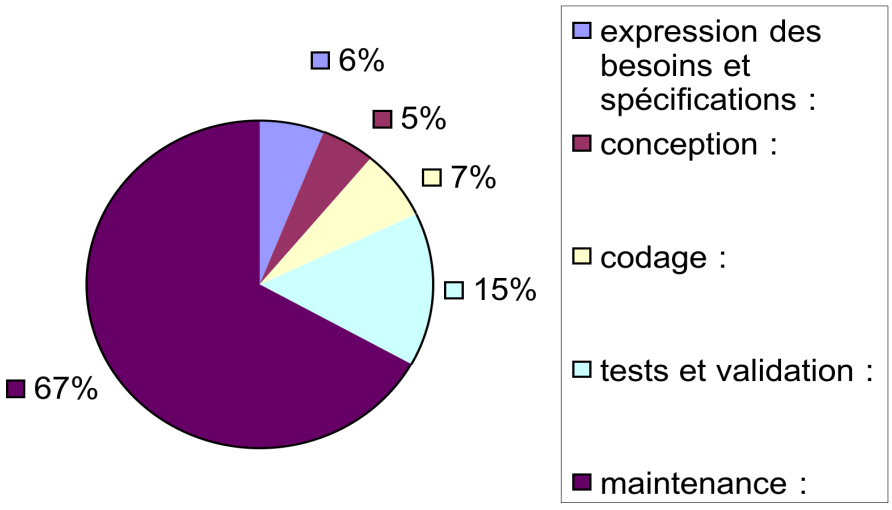
L'activité même du programmeur est assez réduite du strict point de vue de sa productivité, comme le montre cette statistique réalisée en 1980.



En tenant compte de l'ensemble des activités entreprises par un programmeur, une instruction coûte en moyenne 40$ à 200$ (USA 1980).

Répartition des coûts de développement

Et si on découpe le développement du projet en grandes phases, on se rend compte que la partie dévolue strictement au « codage », la partie dite « productive », est très faible.



Cette statistique date des années 80. Elle concerne des projets importants (plus de 5 personnes travaillant pendant plus de 12 mois), et des logiciels à longue durée de vie (plusieurs années).

Remarquons au passage qu'un logiciel coûte plus cher à maintenir qu'à développer.

2.2 LES QUALITÉS REQUISES D'UN LOGICIEL

La crise des années 70, telle que nous l'avons décrite, nous permet de préciser dans ses grandes lignes les objectifs du génie logiciel (reprise de la deuxième définition du génie logiciel).

Définition 3: Objectifs du génie logiciel

Donner les moyens nécessaires à l'équipe de développement pour que le système réalisé :

- Respecte les coûts et les délais de développement ;
- Soit un **produit de qualité.**

Dans cette définition, le mot-clé « qualité » - mot magique -, mérite à lui seul tout un développement...

On peut aborder le critère « qualité » selon 3 points de vue :

- La **qualité externe** (point de vue de l'utilisateur)
- La **qualité interne** (point de vue du développement)
- La **qualité du processus de développement**

Remarquons d'emblée qu'il y a une grande interdépendance entre ces trois points de vue.

2.2.1 QUALITÉS EXTERNES SIGNIFICATIVES

*La **correction**, la **fiabilité** et la **robustesse** s'adressent à l'aspect fonctionnel du logiciel fourni au client.*

Correction

Un logiciel « correct » satisfait à sa spécification (fonction attendue) de manière absolue.

Fiabilité

La fiabilité est une notion relative à l'idée que s'en fait l'utilisateur même du logiciel.

La fiabilité mesure le degré de confiance que l'on peut avoir dans le fonctionnement du logiciel. Il marche aujourd'hui, marchera-t-il demain ? Peut-on compter sur ce produit ?

Robustesse

La « robustesse » dénote une aptitude à fonctionner correctement sous des conditions anormales, comme par exemple :

- Dans les cas non spécifiés par l'analyse des besoins :
  - Ressource non disponible « fichier non-existant », etc.
  - Valeur inattendue, p.ex. : division par zéro, date= « `29/2/2002` »
- A l'occasion de pannes du réseau ou de machines
  - « Serveur ne répond pas »,
  - etc.

Le système doit alors pouvoir s'adapter et continuer de fonctionner.

Performance

- Qualité liée à l'algorithmique (complexité polynomiale, exponentielle) .
- Evaluation par mesure, analyse et simulation.

Ergonomie

- Qualité liée à la facilité d'utilisation.
- Se rattache principalement à l'interface utilisateur mais dépend aussi de la performance et de la correction.

2.2.2 QUALITÉS INTERNES SIGNIFICATIVES

- **Maintenabilité (~60% du coût)**
  - Maintenance corrective : pour éliminer les erreurs
  - Maintenance adaptative : changement dans l'environnement
  - Maintenance perfective : changement pour améliorer ses qualités
- **Réparabilité**
  Facilité avec laquelle le logiciel peut s'adapter à des corrections.
  Nécessite une bonne structure modulaire avec de bons interfaces, un langage adéquat.
- **Evolutivité**
  Facilité avec laquelle le logiciel peut s'adapter à des changements de spécification.
- **Réutilisation**
  A priori ou à posteriori.
  L'idée étant de développer une étagère de composants.
- **Portabilité**
  Le produit est indépendant du genre d'environnement

- **Interopérabilité**
  Capacité à interagir avec d'autres systèmes. Autrefois problématique, la venue des outils de sérialisation (XML, Json) et d'Internet a grandement facilité les choses.

2.2.3 QUALITÉS DU PROCESSUS DE DÉVELOPPEMENT

- **Productivité et réutilisation**
- **Respect des délais**
- **Echange d'informations**
  Bonne documentation, accessible à tous

2.2.4 AUTOPSIE DE QUELQUES CATASTROPHES

1962 : La perte de Mariner I

Pour quelques lignes de FORTRAN..

```
DO 5 K = 1. 3
      T(K) = W0
      Z = 1.0/(X**2)*B1**2+3.0977E-4*B0**2
      D(K) = 3.076E-2*2.0*(1.0/X*B0*B1+3.0977E-4*
                  *(B0**2-X*B0*B1))/Z
      E(K) = H**2*93.2943*W0/SIN(W0)*Z
      H = D(K)-E(K)
5 CONTINUE
```

La première ligne est fausse, elle aurait dû s'écrire

```
DO 5 K = 1, 3          Boucle avec K variant de 1 à 3
```

Au lieu de cela, le compilateur FORTRAN a compris

```
DO 5 K = 1.3           Boucle parcourue 1 fois avec K valant 1.3
```

- Ainsi, la perte de Mariner est due à une faute de frappe.
- Cette erreur a mis en avant la faiblesse de la syntaxe du langage FORTRAN
  - Trop grande proximité entre l'affectation est l'instruction de boucle ;
  - Difficulté de lecture du programme.

Notons par ailleurs que le code fautif n'avait pas été testé correctement...

Il s'avère aujourd'hui qu'il s'agissait de désinformation : La faute à FORTRAN est un mythe, véhiculé par on ne sait qui... (un anti-fortran??), un mythe qui a subsisté très longtemps... Car en fait aucun code FORTRAN n'était utilisé dans les programmes de guidage..

Mars Climate Orbiter ($125M)

Le 23 septembre 1999 à 11h27, la NASA perd tout contact avec la sonde spatiale 'Mars Climate Orbiter'. La sonde s'est désintégrée dans l'atmosphère de Mars en essayant de se mettre en orbite à une hauteur de 53 km au lieu des 193 km qui étaient planifiés.

L'origine de la panne est une erreur parfaitement stupide:

- Lockheed Martin Astronautics a utilisé les mesures anglo-saxonnes;
- Jet Propulsion Laboratory a utilisé le système métrique;
- Personne n'a converti les données échangées (1 livre = 4.48 Newton).

Erreur dans les processus de validation de la NASA

Rapport de la NASA (2000) :

- L'erreur de navigation n'a pas été détectée par la simulation informatique de la propulsion.
- L'équipe opérationnelle de navigation n'était pas assez informée de la manière dont la sonde était orientée dans l'espace.
- Le processus de validation des tâches interconnectées n'était pas assez robuste à cause d'un changement de stratégie de la NASA dans sa manière de réaliser les nouveaux projets.
- Certains canaux d'information entre groupes d'ingénieurs étaient trop informels.
- Le personnel n'était pas assez entraîné à remplir les formulaires formels d'anomalies.
- Etc..

Autres échecs retentissants

- En 1972, lors d'une expérience météorologique en France, 72 ballons contenant des instruments de mesure furent détruits à cause d'un défaut dans le logiciel.
- En 1981, le premier lancement de la navette spatiale a été retardé de deux jours à cause d'un problème logiciel. La navette a d'ailleurs été lancée sans que l'on ait localisé exactement le problème (mais les symptômes étaient bien délimités).
- Le développement du compilateur PL1 de Control Data n'a jamais abouti.
- L'explosion de la fusée Ariane 5, le 4 juin 1996, qui a coûté un demi-milliard de dollars (non assuré!), est due à une faute logicielle d'un composant dont le fonctionnement n'était pas indispensable durant le vol.