

# Evolution et restructuration

Nicolas Hlad

## Contenu de L'UE

### L'objectif de l'UE

- Comprendre les processus de maintenances et d'évolutions des logiciels
- Découvrir des techniques de réingénierie logicielle
  - Vers des Lignes de Produits logiciel
  - Vers des Micro-Services
- Améliorer la maintenabilité grâce au le Refactoring
- Prise en main d'outil pour la maintenabilité et la réingénierie (Moose, Gumtree, RCA-Explorer)

### Contenu des Séances du Cours (CM+TD/TP)

- Introduction à la compréhension et maintenance logiciel
- Analyse statique & dynamique (2 séances)
- Extraction d'Architecture et réingénierie vers micro-services
- Réingénierie vers ligne de produits logiciels (3 séances)
- Introduction de Moose
- Découverte du Refactoring avancé

**Code Moodle : HMIN306\_2020**

## Planning du cours

- ❖ - 08 / 09 : Compréhension et Maintenance Logiciel
- ❖ - 15 / 09 : Analyse statique
- ❖ - 22 / 09 : Analyse dynamique
- ❖ - 29 / 09 : Introduction au Refactoring
- ❖ - 06 / 10 : Extraction de Ligne de Produits Logiciels - 1
- ❖ - 13 / 10 : Extraction de Ligne de Produits Logiciels - 2 FCA et CLEF
- ❖ - 20 / 10 : Conférence SPLC2020
- ❖ - 03 / 11 : Extraction d'architecture Variable avec Gumtree et FCA
- ❖ - 10 / 11 : Extraction d'architecture et Réingénierie vers Micro-service
- ❖ - 17 / 11 : Introduction à Moose
- ❖ - 24 / 11 : Introduction Multi-Ligne De Produits Logiciel avec RCA



**TOUS LES TPs SONT A RENDRE !!**



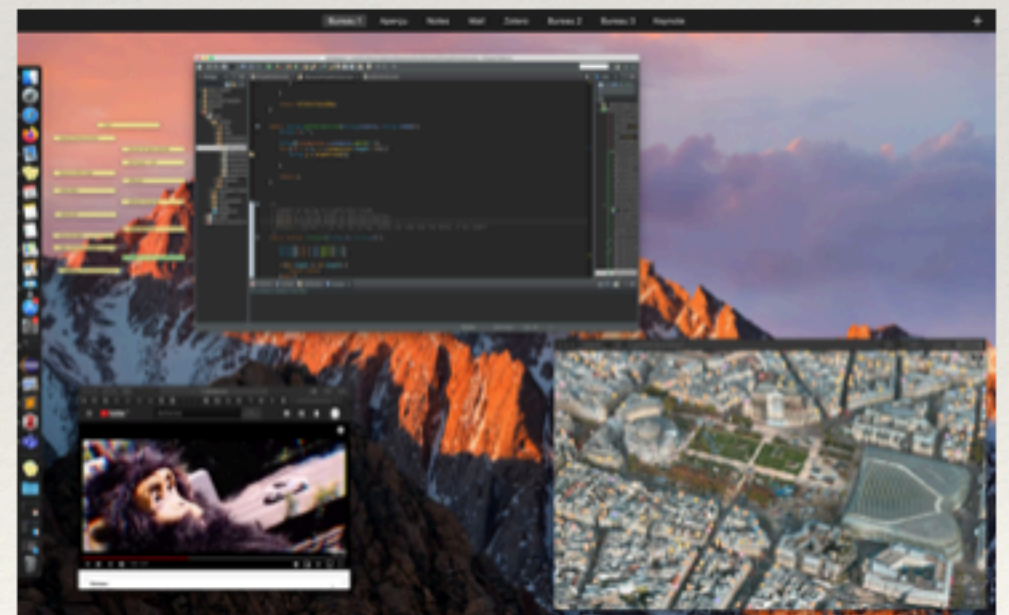
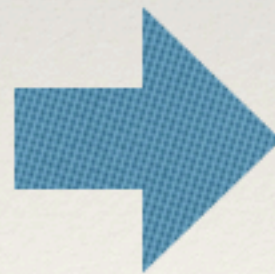
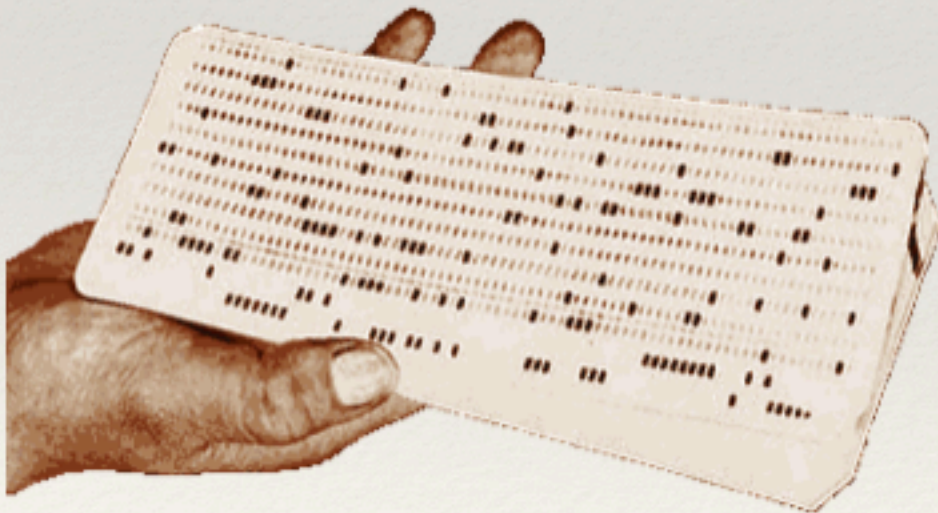
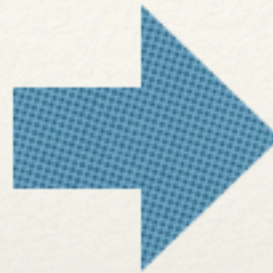
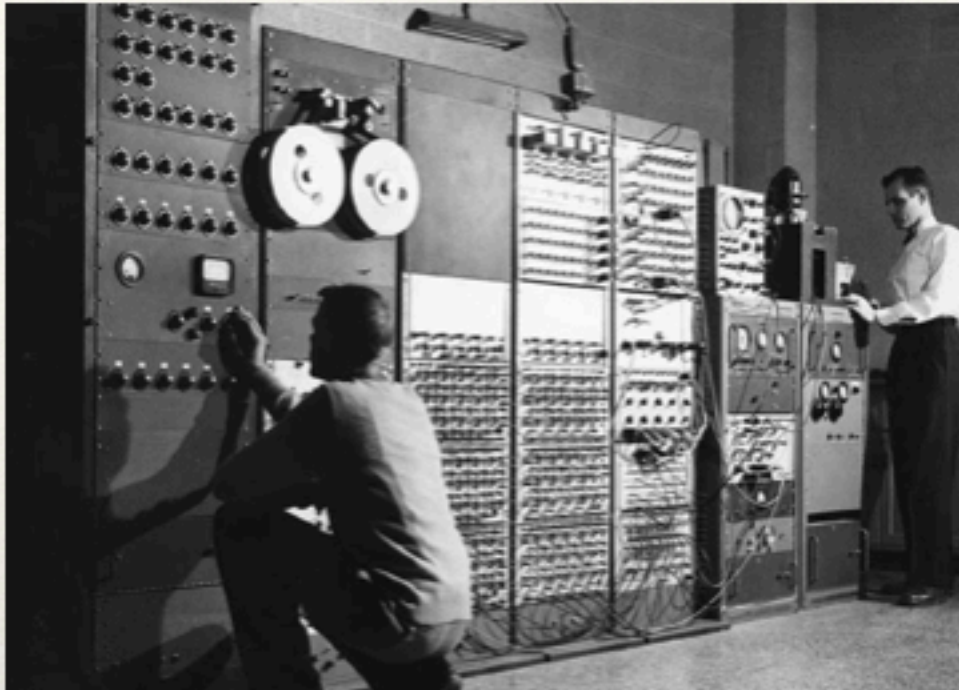
## Cours 1

---

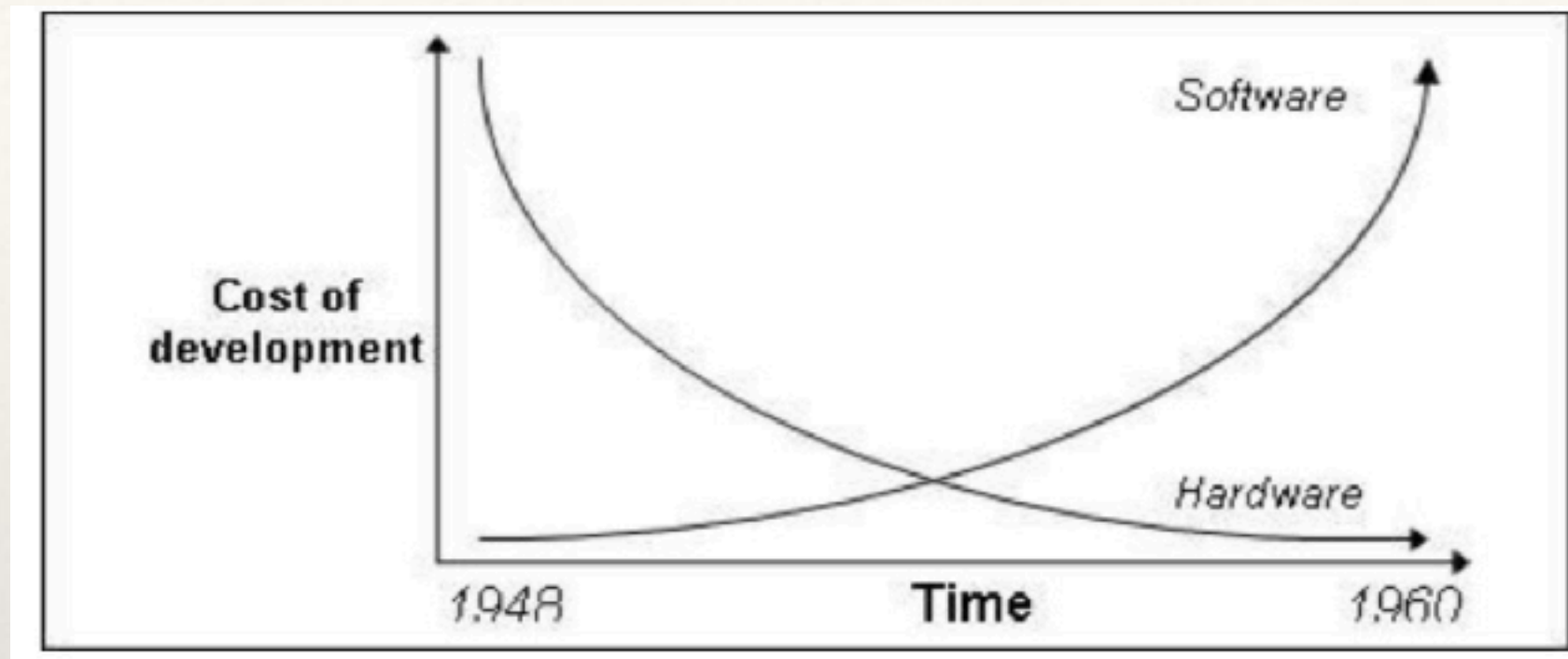
# Introduction à la Maintenance Logiciel



# La crise du Logiciel



# La crise du Logiciel



Evolution du coût du logiciel par rapport au matériel entre les années 1948 et 1960.

- Très vite les *softwares* sont devenu de plus en plus complexe, augmentant leurs coûts
- Le *hardware* est devenu de abstrait et la complexité (spécificité) a été transféré au *software*.
- On parle de « crise » car nous n'avons toujours pas trouvé de solution pour diminuer ou stabiliser les coûts du *software*, qui augmentent d'années en années.



# La crise des logiciels


Emergence de grands systèmes logiciels composés de milliers de lignes de code:

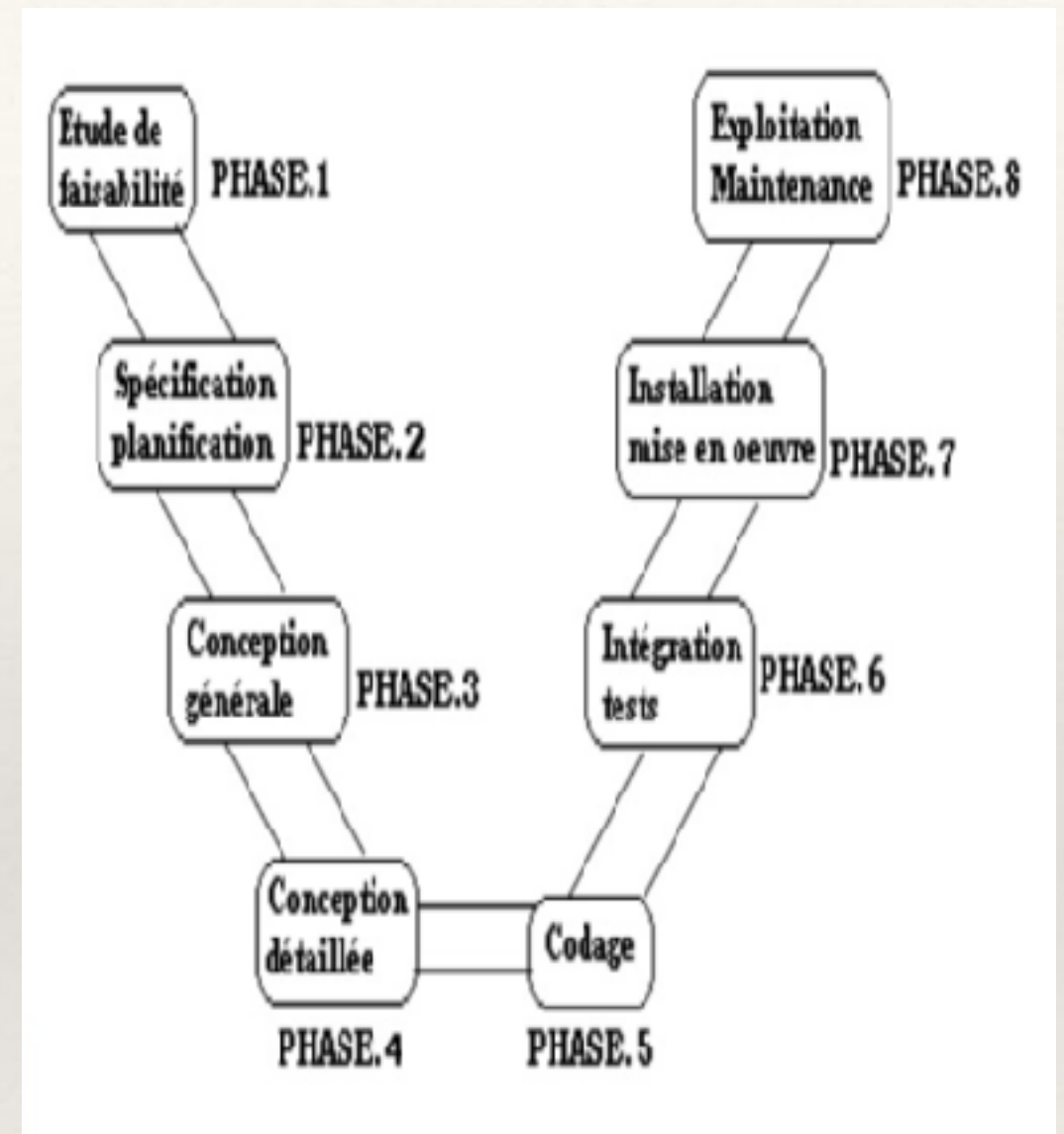
Système	langage	NOP	NOC	kLOC
ArgoUML	Java	141	2442	143
Azureus	Java	457	4734	274
IText	Java	149	1250	80
Jmol	Java	105	1032	85
JDK 1.5	Java	137	4715	160
Moose	Smalltalk	278	994	35
ScrumVM	C++	17	1331	80

[Richard Wettel] NOP = Number of Package; NOC = Number of Class; kLOC = Line of Code (in the thousands)

Les développeurs ont de plus en plus de mal à avoir la compréhension nécessaire pour réaliser efficacement leurs tâches de développement

# Le cycle de Production

- ❖ Les coûts de production augmentent indépendamment du processus suivie.
- ❖ Exemple avec le **cycle en V**, un processus de conception en 8 phases.
- ❖ Le cycle boucle sur la phase d'Exploitation & Maintenance
- ❖ Environ 67% des coûts du cycle sont consacrés à la maintenance  
[ Digital Aggregates]



Le cycle en V

Bien que différente, les méthodes dites *agiles* produisent le même résultat que le cycle en V. Elles comprennent également une phase de **maintenance**.

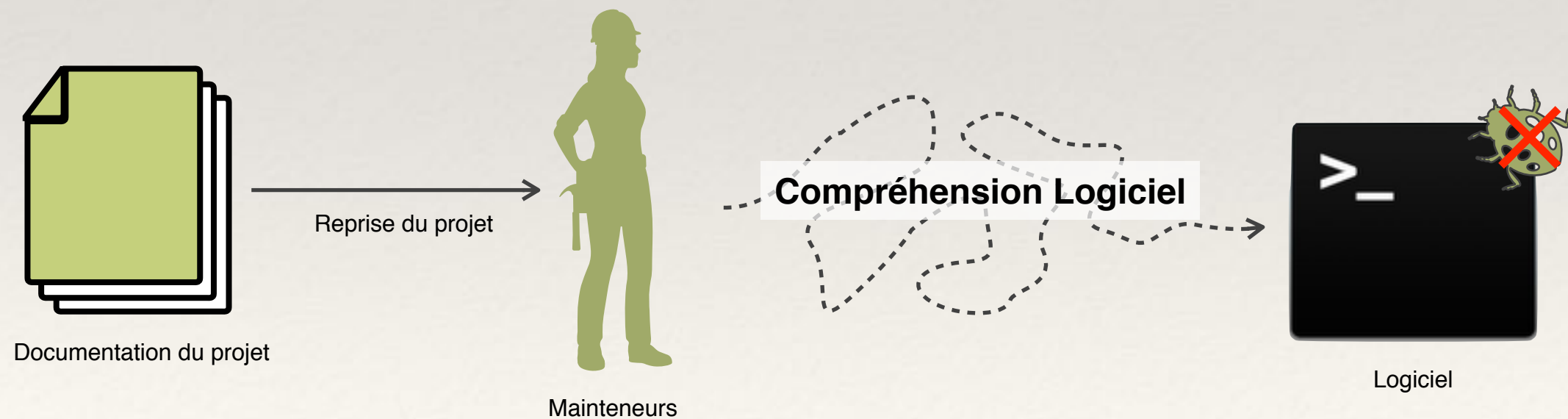
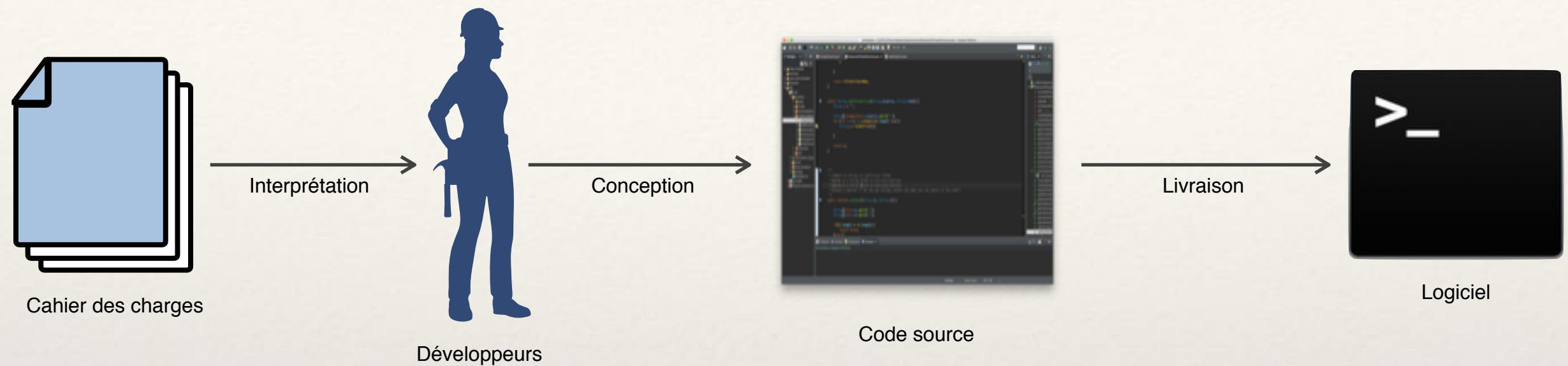


# Les déclencheurs de la phase de maintenance

## Qu'Elles Sont les Éléments Déclencheurs de la Phase de Maintenance ?

- De nouvelles exigences apparaissent lorsque le logiciel est utilisé ou même quand le logiciel est en développement
- Les changements de l'environnement des affaires
- Nouveaux clients ou les clients existants avec de nouvelles exigences
- Les erreurs doivent être réparées
- Nouveaux ordinateurs et équipements sont ajoutés au système
- La performance ou la fiabilité du système peuvent être améliorées
- De nouvelles technologies sont utilisées (nouvelles normes, de nouveaux OS, nouvelles versions du logiciel, ...)
- De nouvelles législations (ex: RGPD)
- etc.

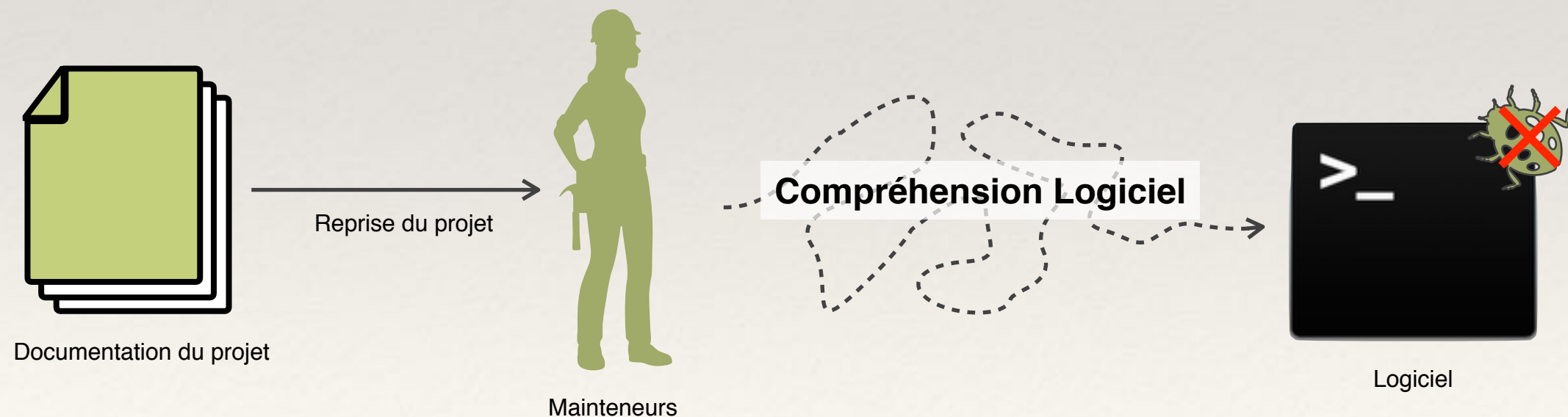
# La compréhension logiciel



# La compréhension logiciel

## L'Intérêt de Comprendre un Logiciel

- La compréhension est nécessaire afin de combler les **écarts** qui peuvent exister encore
  - entre le **domaine d'ingénierie** et le **domaine d'application**
  - entre le domaine **matériel** et celui du **logiciel**
  - entre le **cahier charge** et l'**implémentation finale**
  - entre les **modèles qui décrivent l'implémentation** (architecture, doc, etc) et l'**implémentation résultante** qui peut se dégrader au fil du temps.





## Objectif et contenu du cours

### Objectifs

- Comprendre les problématiques de la conception logiciel
- Introduire le concept “compréhension logiciel”
- Expliquer les fondamentaux essentiels de la maintenance des logiciels
- Convaincre du caractère essentiel de la maintenance, inhérente à tout développement
- Présenter les aspects techniques de la maintenance
- Présenter les aspects organisationnels et à la gestion des activités de maintenance



Partie 1

# La compréhension logiciel

## CODE PROGRESSION

ARCHITECTURE



PROTOTYPE



PILOT



BETA



RELEASE



LEGACY



DOCUMENTATION



# Définition

- ❖ La compréhension logiciel est la capacité de **s'approprier** et de **comprendre** le code source d'un logiciel
- ❖ cela revient à
  - ❖ **examiner et comprendre le système**
  - ❖ **acquérir des connaissances sur le programme**
  - ❖ **développer des modèles mentaux** de l'architecture logiciel, du comportement et de la sémantique du code source.
- ❖ Souvent des difficultés de compréhension apparaissent lorsqu'un écart apparaît entre la conception papier (cahier des charges, modèles UML, etc) et le code source implémenté
  - ❖ celle apparaît car :
    - ❖ le code du *système hérité* est trop spécifique
    - ❖ le code contient des erreurs (non-respect de la conception papier)
    - ❖ l'étape d'appropriation à introduite de nouvelles erreurs.



# Intérêt

- ❖ La compréhension logiciel est essentiel si l'on veut :
  - ❖ **maintenir un code**
  - ❖ **le faire évoluer**
    - ❖ ajouter de nouvelles fonctionnalités ou changer d'architecture
  - ❖ **faire une migration logicielle**
    - ❖ objets vers composants
    - ❖ native vers multi-plateforme
    - ❖ monolithe vers micro-service
    - ❖ etc

La compréhension logiciel représente 40% à 60% de l'activité de maintenance ! [Pfleeger 1998]

# Modèle de représentation

- ❖ On cherche à créer des modèles de représentation du code source.
  - ❖ ces modèles se portent sur :
    - ❖ la **structure** : les composants logiciels et leurs interrelations
    - ❖ les **fonctionnalités** : quelles opérations sont effectuées et sur quels composants
    - ❖ le **comportement dynamique** (à l'exécution)
    - ❖ la **raison d'être** (pourquoi certaines décisions d'implémentation ont été prise)
    - ❖ la **conception des différents éléments du code** : les modules, la documentation, les tests, etc.
  - ❖ Il existe deux types de modèles :
    - ❖ Modèle de maintenance **statique** : regroupe toutes les informations relatives à l'analyse statique d'un programme
    - ❖ Modèle de maintenance **dynamique** : regroupe toutes les informations relatives à l'analyse dynamique obtenu sur une exécution d'un programme.

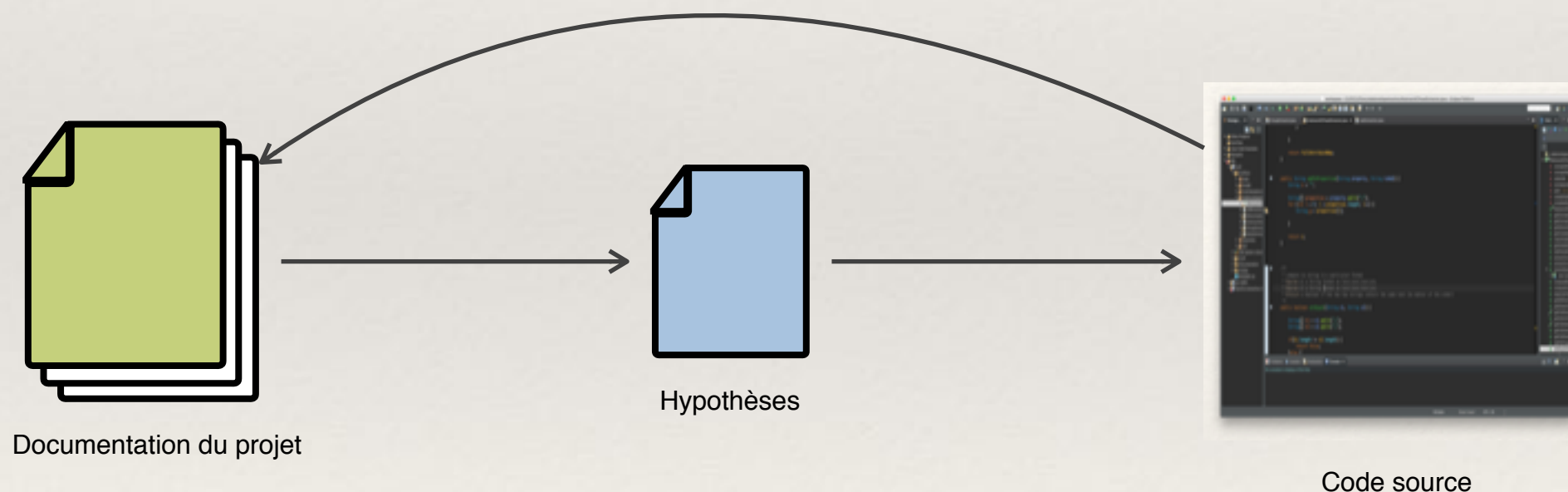
# Approche de compréhension

- ❖ La difficulté de compréhension d'un logiciel apparait lorsque apparait un écart entre ce que l'on comprend du logiciel et ce que l'on observe dans le code source.
  - ❖ il est alors important de combler cet écart, ce qui peut être fait par 3 approches de compréhension
    - ❖ **Approche descendante**
      - ❖ associer les conceptions les plus abstraites du domaine d'application sur le code source
    - ❖ **Approche ascendante**
      - ❖ se concentrer sur la compréhension de petits morceaux de code pour les associer à des concepts plus abstraits du domaine d'application
    - ❖ **Approche opportuniste**
      - ❖ un mixe des deux autres qui s'effectue en basculant du statut de développeurs à mainteneur.



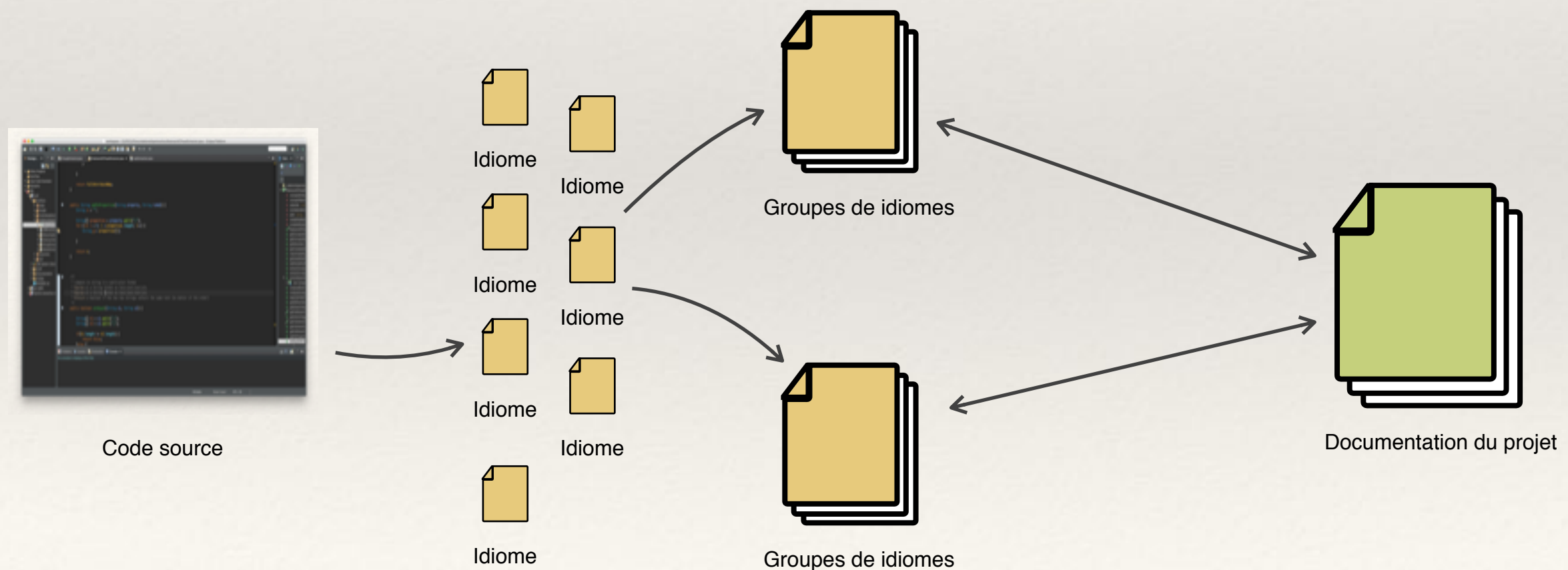
# Approche descendante

- ❖ Le mainteneur effectue des hypothèses à la suite de sa compréhension du domaine d'application.
- ❖ il essaie ensuite de vérifier ses hypothèses dans le code sources.
- ❖ Le mainteneur définit des balises : endroits du code qui servent à affirmer ou réfuter une hypothèse.
- ❖ l'approche simple si on dispose de la documentation



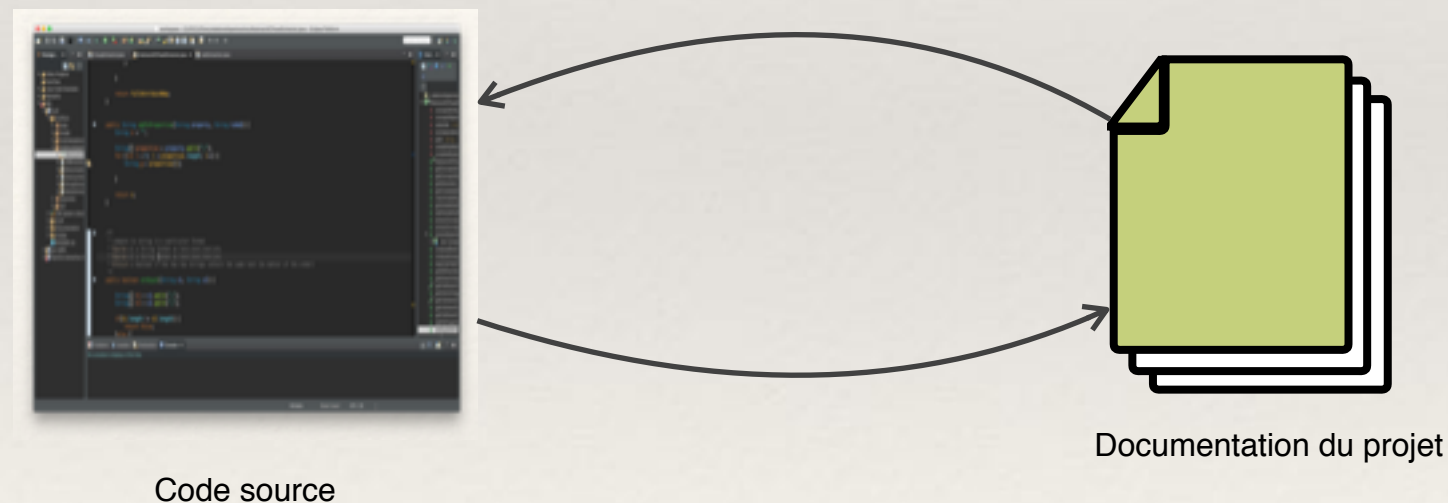
# Approche ascendante

- ❖ utiliser lorsque l'on est peu familier avec le code source du programme.
- ❖ le mainteneur recherche dans le code ce qu'on appelle des idiomes : c'est-à-dire des endroits facilement reconnaissable (patrons de conception, motif, etc).
- ❖ l'objectif est de combiner les idiomes de plus en plus grands pour les relier au domaine d'application.
- ❖ l'approche la plus difficile



# Approche opportuniste

- ❖ le programmeur bascule fréquemment entre ascendante et descendante.
- ❖ les informations obtenues lors de l'analyse ascendante vérifient les hypothèses de l'analyse descendante.
- ❖ elle se destine plus aux développeurs.
- ❖ Il s'agit sans doute de l'approche la plus réaliste (dans le contexte industriel actuel)





# Comprendre le code source

- ❖ On dispose souvent peu de document pour comprendre le code source.
- ❖ **La principale source d'information est donc le code source.**
  - ❖ A partir du code source, on peut :
    - ❖ collecter des informations sur les parseurs, débogueurs, registre d'événement, etc
    - ❖ information sur les modèles abstraits ( de haut niveaux) comme l'UML
    - ❖ obtenir des informations sur la structure et la navigation dans le code avec :
      - ❖ Des représentations graphiques (diagramme d'architecture, objets, etc).
      - ❖ **slicing** : isoler et trancher les parties du code qui indépendante

# Exemple Slicing

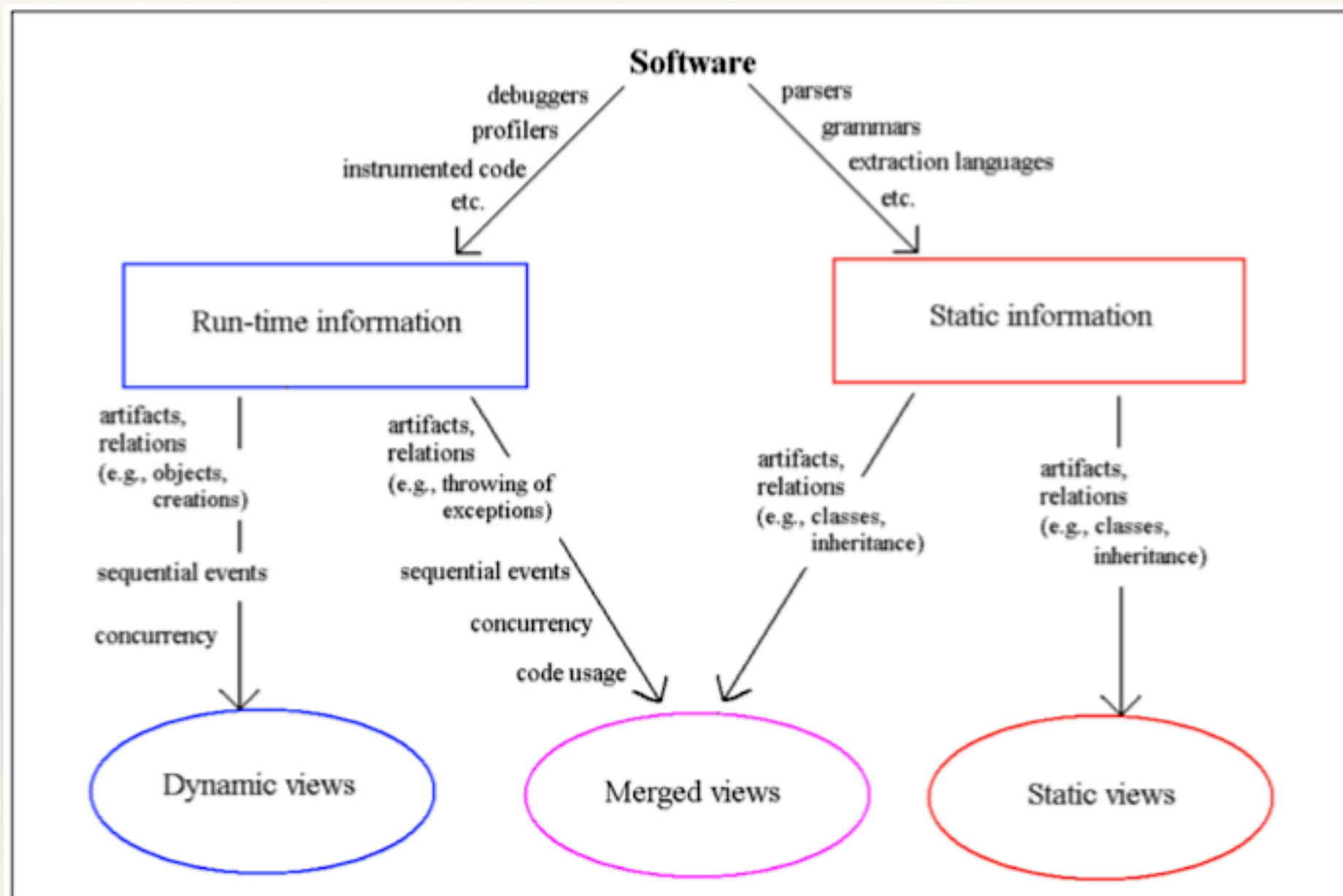
```
1  public Class File{
2      private String name ; // file's name
3      private String path ; // full path of a file
4      private String extension ; // file's extention (.java, .png, etc)
5
6      public File(String filePath){
7          this.name = filePath.substring(filePath.lastIndexOf("/"));
8          this.path = filePath;
9          this.extension = name.substring(name.lastIndexOf("."));
10     }
11 }
```

Avant

```
1  public Class File{
2      private String name ; // file's name
3      private String path ; // full path of a file
4      private String extension ; // file's extention (.java, .png, etc)
5
6      public File(String filePath){
7          this.path = filePath;
8
9          // these 2 instruction are dependent
10         this.name = filePath.substring(filePath.lastIndexOf("/"));
11         this.extension = name.substring(name.lastIndexOf("."));
12     }
13 }
```

Après

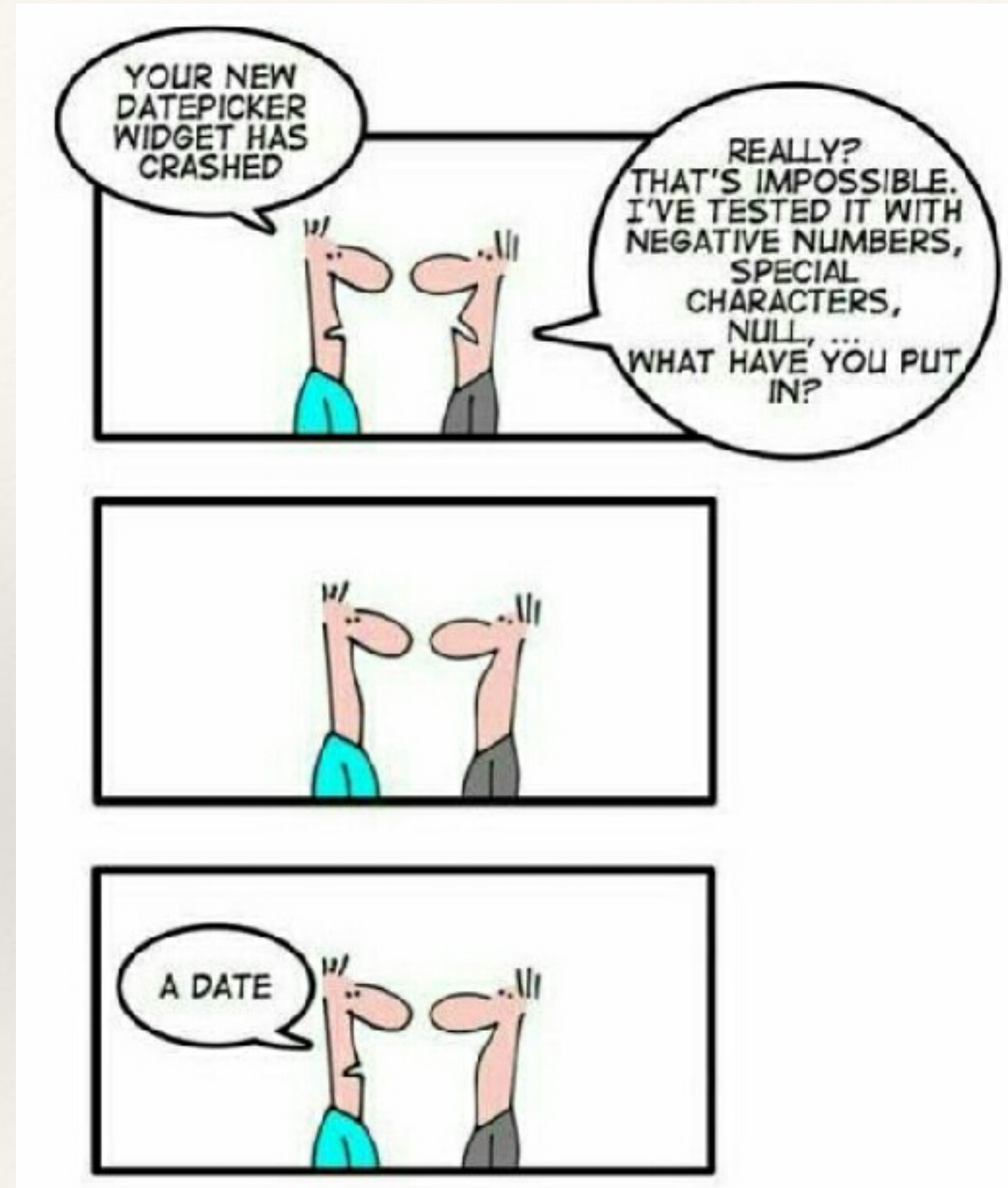
# Récolter les informations



[M. Ward]

## Partie 2

# La maintenance logiciel





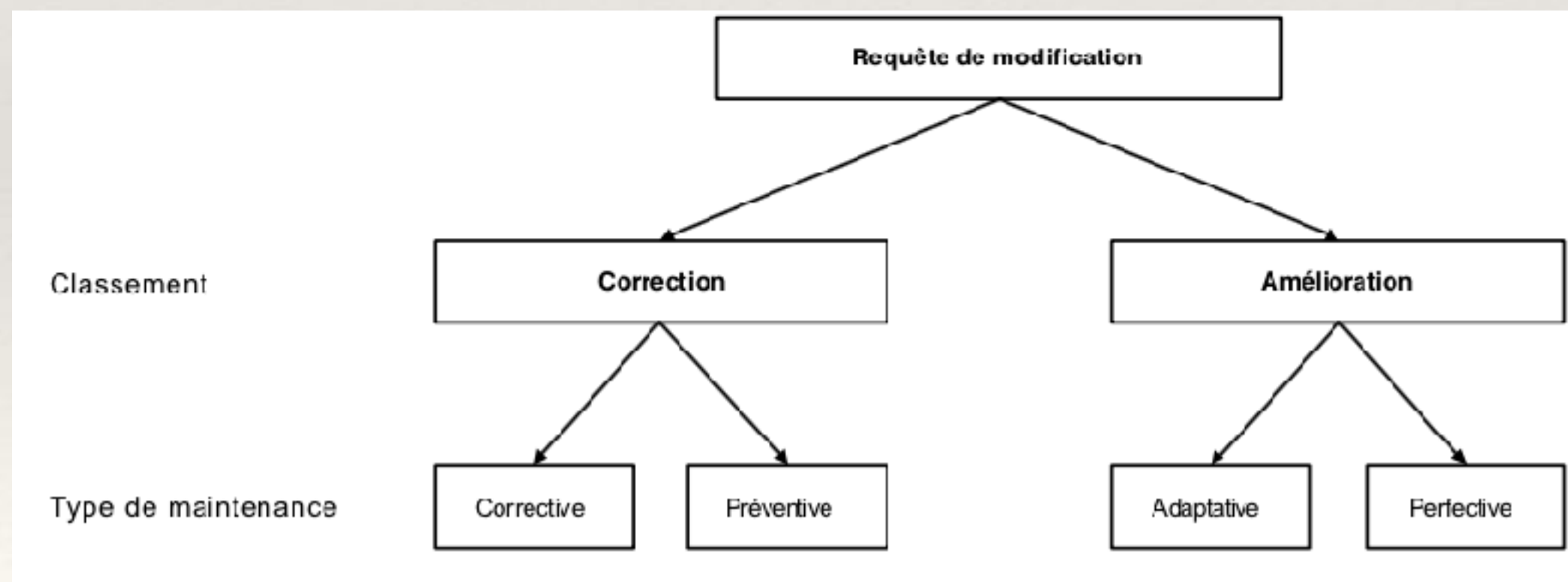
# Quelques définitions

- ❖ Qu'est-ce que la maintenance / évolution ?
  - ❖ Les changements apportaient au logiciel et à sa documentation, causés par un problème ou le besoin d'amélioration  
— ISO 12207, 1995
  - ❖ La totalité des activités qui sont requises afin de procurer un support, au meilleur cout possible, d'un logiciel. Certaines activités débutent avant la livraison du logiciel, donc pendant la conception initiale, mais la majorité des activités ont lieu après sa livraison finale (quand l'équipe de développement considère qu'elle a terminé son travail)  
— SWEBOK, 2005
  - ❖ L'ensemble des activités requises afin de garder le logiciel en état d'opération suite à sa livraison opérationnelle  
— FIPS, 1984
- ❖ Maintenance ou évolution Logiciel ?
  - ❖ La maintenance est une opération de modification du logiciel existant, elle entraine forcément une évolution (plus ou moins importante).
  - ❖ (définition qui peut dépendre de la communauté à laquelle on s'adresse).

# Les différents types de maintenance selon ISO/IEC

	Why?	Correction	Enhancement
When?			
Proactive		Preventive	Perfective
Reactive		Corrective	Adaptive

Classification de ISO/IEC 14674



Classification d'une requête de modification [SELLAMI 2000, ISO/IEC 14764]

# Les 4 catégories de maintenance

## Proactive (pendant le développement)

- ❖ Maintenance **Perfective** [Swanson 1989]
  - ❖ La maintenance perfective ou d'amélioration comprend tous les changements faits sur un système afin de satisfaire aux besoins de l'utilisateur.
- ❖ Maintenance **Préventive** [IEEE 1219]
  - ❖ La maintenance exécutée afin d'empêcher des problèmes avant qu'ils surviennent.
  - ❖ Ce type de maintenance peut être considéré comme très important dans le cas de systèmes critiques.

# Les 4 catégories de maintenance

## Réactives (après le développement)

- ❖ Maintenance **Corrective** [Swanson 1989]
  - ❖ Fixe les différents bogues d'un système.
  - ❖ Le système peut ne pas fonctionner comme il devrait le faire selon ses plans de conception. L'objectif de la maintenance corrective est donc de localiser les spécifications originelles afin de déterminer ce que le système était initialement sensé faire.
- ❖ Maintenance **Adaptative** [Swanson 1989]
  - ❖ Fixe les différents changements qui doivent être faits afin de suivre l'environnement du système en perpétuelle évolution.
  - ❖ Par exemple, le système d'exploitation peut être mis à jour et des modifications peuvent être nécessaires afin de s'accommoder à ce nouveau système d'exploitation.



# Les caractéristiques de la maintenance

## ❖ difficile car :

- ❖ résultat d'une mauvaise conception
- ❖ code source mal documenté résultant sur des problèmes compréhensions logiciels
- ❖ activité de maintenant laissé aux second plan (attribué à des développeurs débutants)
  - ❖ "Programmers [...] **tend to think of program development as a form of puzzle solving and it is reassuring to their ego when they manage to successfully complete a difficult section of code.** Software Maintenance on the other hand **entails very little new creation and is therefore categorized as dull, unexciting detective work**".  
[ D. A. Higgins, Data Structured maintenance: The warnier / Orr Approach. New York: Dorset house Publishing Co. Inc. ]
- ❖ amateurisme de toutes les étapes du processus de maintenance
- ❖ manque d'outils d'aide à la maintenance

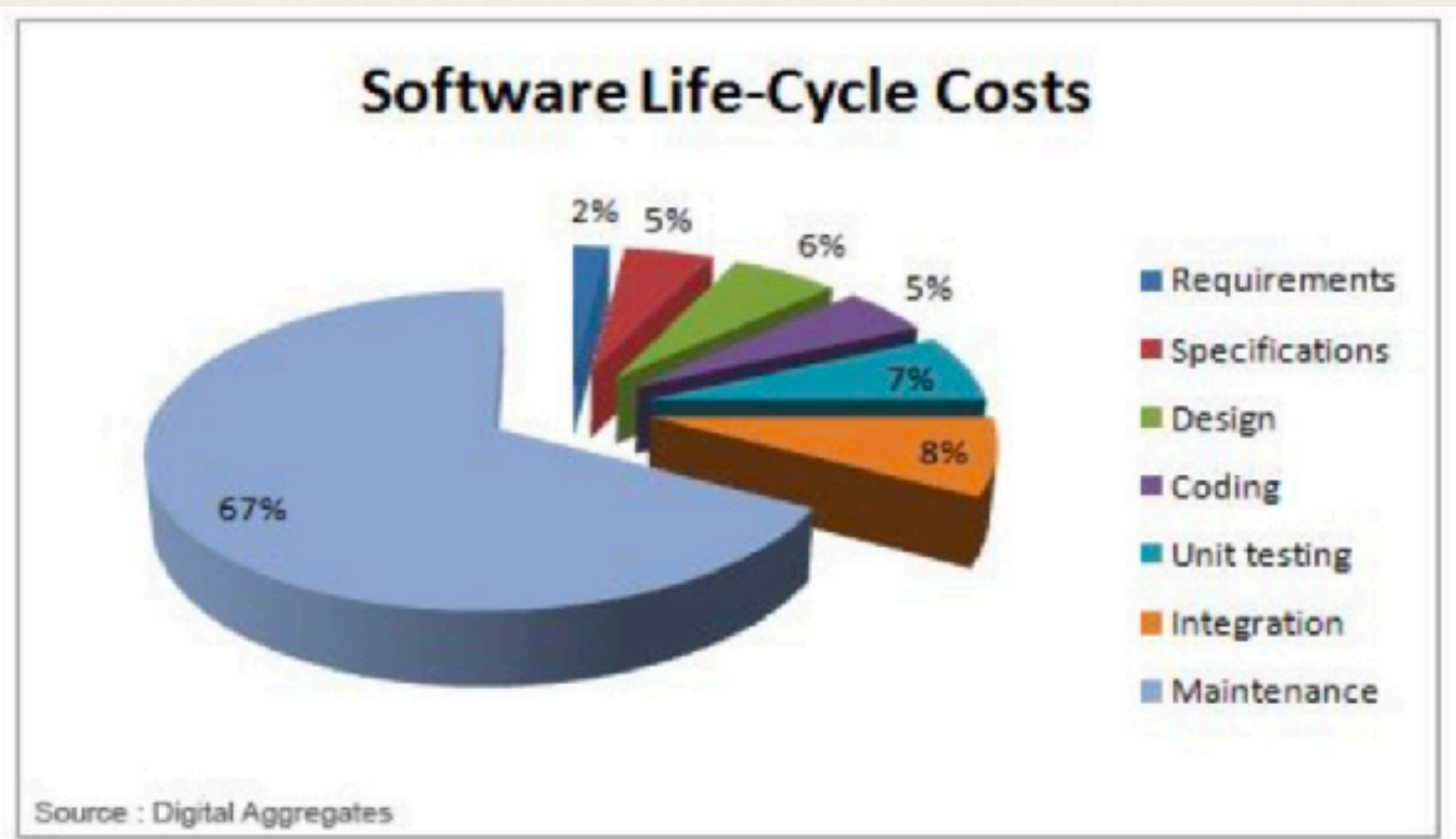
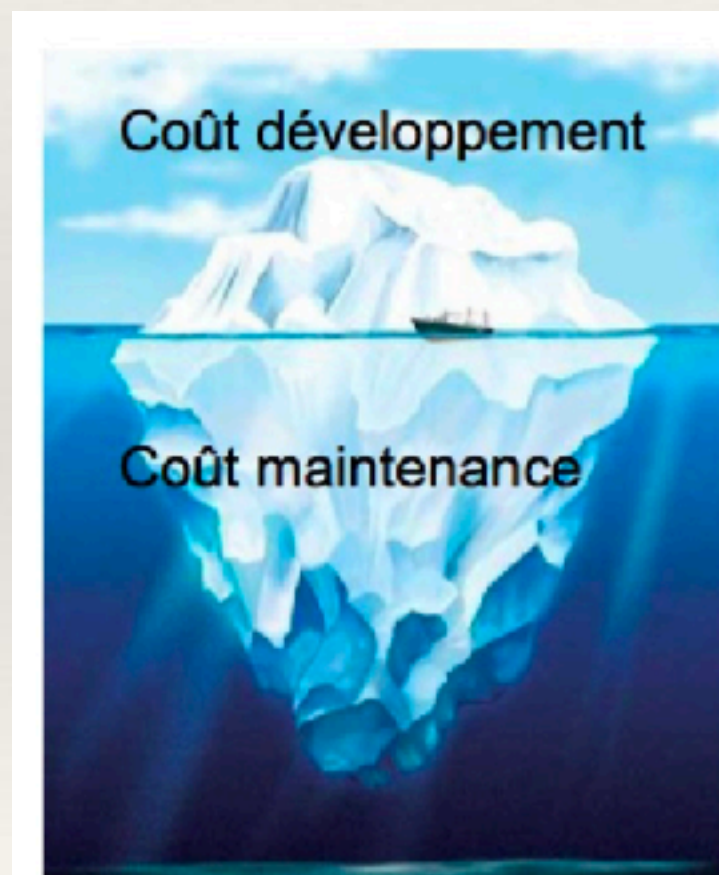
# Les caractéristiques de la maintenance

## ❖ des changements imprévisibles

- ❖ certains changements étaient imprévus par les concepteurs d'origines et posent donc problèmes lors de leur implémentation (architecture incompatible, technologies trop vieille, etc)
- ❖ "Le problème fondamental, soutenue par 40 ans d'expérience difficile, est que de nombreux changements réellement requis sont ceux que les **concepteurs originaux ne pouvaient même pas imaginer.**" [Bennett & Rajlich 2000].

# Les caractéristiques de la maintenance

- ❖ **plus coûteuse** que le développement de l'application (environ 67% du coût total de l'application)
- ❖ Les coûts de maintenance sont supérieurs aux coûts estimés avec un facteur compris entre 2 et 100 (selon l'application)






# Les Lois de Lehman

- ❖ Ces *Lois* sont basées sur des **études empiriques** par rapport à l'évolution sur une période de **30 ans**
- ❖ Elles décrivent les conséquences et les scénarios de l'évolution du logiciel sur le long terme.

Lehman M.M. and Belady L.A. (Eds.), 1985  
*Software Evolution – Processes of Software Change*, Academic Press, London  
(Free download from  
[wiki.ercim.eu/wg/SoftwareEvolution](http://wiki.ercim.eu/wg/SoftwareEvolution))



[ M. M. Lehman *Laws of Software Evolution Revisited*. Lecture Notes in computer science 1149, pp.108-124, Springer Verlag, 1997]



# Les lois de Lehman

## ❖ La modification continue

- ❖ Un programme utilise dans un environnement du monde réel doit nécessairement changer sinon il deviendra progressivement de moins en moins utile dans cet environnement.

## ❖ La complexité croissante

- ❖ Comme le logiciel est modifié, il devient plus en plus complexe à moins que le travail soit effectué pour réduire la complexité

## ❖ L'autorégulation

- ❖ L'évolution des grands programmes est un processus auto-régulateur. Les attributs comme la taille, le temps entre versions et le nombre d'erreurs signalées sont approximativement invariants pour chaque version du programme

## ❖ La stabilité organisationnelle

- ❖ Pendant la vie d'un programme, son **taux de développement (à relire dans la source)** est approximativement constant et indépendant des ressources qui y sont consacrées

## ❖ La conservation de la familiarité

- ❖ Pendant la vie d'un programme, l'incrément de changement dans chaque version est approximativement constant.

# Les système hérités (Legacy Systems)

- ❖ Il s'agit de logiciel hérité de projets anciens (souvent plusieurs décennies en arrière).
- ❖ Ces logiciels sont gardés car toujours **rentable**, mais deviennent de **plus en plus complexe** à maintenir (loi de Lehman)
  - ❖ pas de style de programmation cohérent
  - ❖ utilisation d'ancien langage ou d'ancienne version de langage
  - ❖ documentation et support indisponible ou obsolète
  - ❖ structure incohérente à force des ad hoc successifs <sup>= ajouts</sup>.
  - ❖ programme optimisé mais code source déstructuré
- ❖ plusieurs contributeurs actifs sur le projet, mais personne ne connaît le système en entier (causé par le *turn over*)



Logo Emacs

*Emacs* : 1970 à aujourd'hui

Le système a connu des évolutions pendant 50 ans !

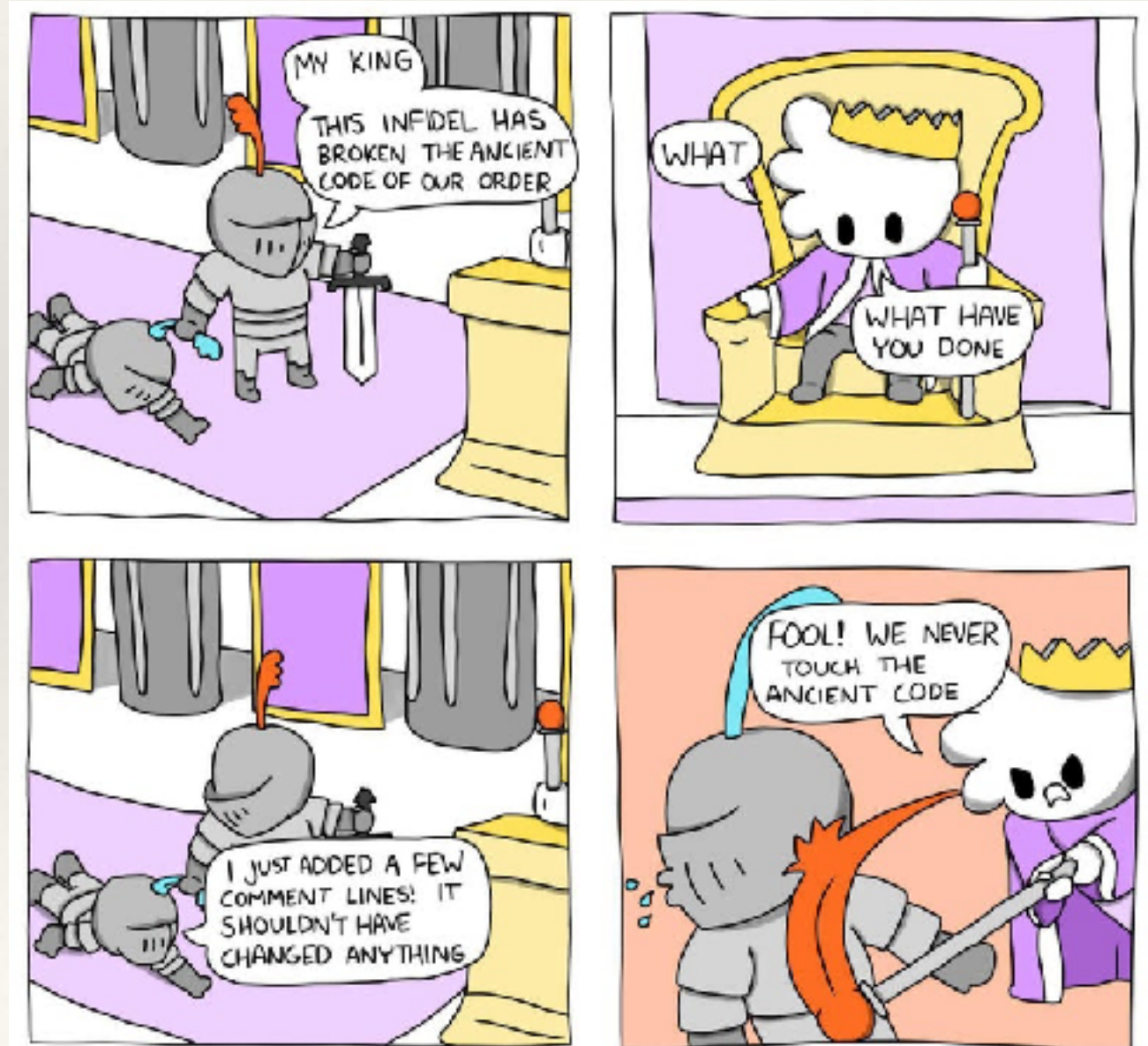
# Les système hérités (Legacy Systems)

## ❖ Le dilemme des systèmes hérités

- ❖ continuer à les entretenir accroît inévitablement leurs couts
- ❖ les remplacer coute cher et ne garantisse pas une rentabilité sur le court terme
  - ❖ les entreprises misant sur le long terme cherchent à allonger la durée de vie des systèmes qui ont fait leur preuve plutôt que les remplacer

## ❖ Les solutions (ou options)

- ❖ abandonner le système (*à la Apple*)
  - ❖ remplacer le système par un nouveau
- ❖ continuer de maintenir le système (*à la Microsoft*)
  - ❖ transformer le système pour faciliter la maintenance





# Les problèmes de la maintenance

- ❖ La compréhension de logiciel : limité et difficile
  - ❖ les études affirment que 40 à 60% de l'effort de la maintenance est consacré à la compréhension logiciel
  - ❖ limité par la rapidité avec laquelle on peut comprendre un logiciel (logiciel inconnu et nouveau)
  - ❖ la compréhension est plus difficile si elle est faite uniquement par l'analyse du source code

la documentation est donc essentielle !



# Les problèmes techniques de la maintenance

## ❖ Les tests de régression

- ❖ il s'agit de (re-)tester un programme après l'introduction d'une modification
- ❖ l'ajout de nouveaux tests rend fastidieux la compréhension des tests existants. Leurs intégrations sont donc plus difficiles à mesure que le programme est modifié (loi de Lehman)
- ❖ l'enjeu est d'identifier les **tests pertinents** pour minimiser l'effort de test tout en maximisant la couverture des risques de régression
  - ❖ un risque de régression apparaît lorsqu'une modification menace les développeurs de faire revenir le programme vers ancienne version, si jamais la nouvelle version introduit un nouveau bug.

La réalité....

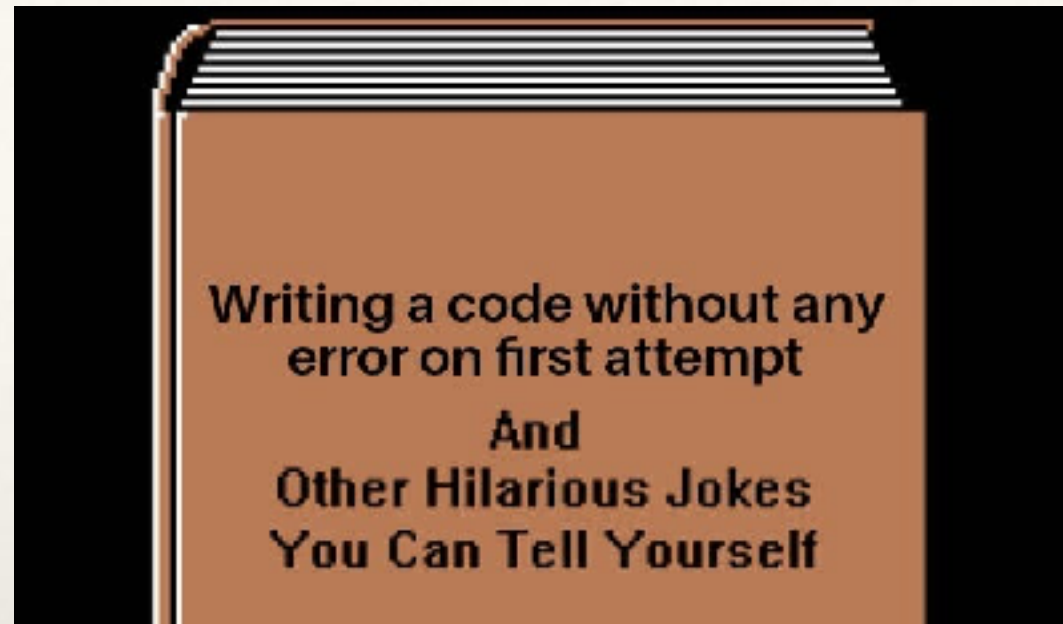


# Les problèmes techniques de la maintenance

## ❖ L'analyse de l'impact

- ❖ Décrit comment mener avec un coût pertinent l'analyse des impacts d'une modification du logiciel
- ❖ nécessite d'analyser la structure et le contenu d'un logiciel.
- ❖ objectif de l'analyse :
  - ❖ déterminer la portée d'une modification et établir / estimer un plan d'action
  - ❖ analyser les coûts / bénéfices de la modification
  - ❖ communiquer aux autres la complexité d'une modification

# Les problèmes techniques de la maintenance



## ❖ Maintenabilité

- ❖ Caractéristique d'un logiciel
- ❖ mesure l'aisance avec laquelle un logiciel peut être modifier, amélioré, adapté, corrigé pour satisfaire de nouvelles exigences

# Les sous-caractéristiques de la maintenabilité

---

- ❖ **analysabilité**

- ❖ mesure de l'effort à fournir pour diagnostiquer les déficiences et les causes de problème dans un logiciel

- ❖ **interchangeabilité**

- ❖ mesure de l'effort à fournir pour implémenter une modification spécifique

- ❖ **stabilité**

- ❖ mesure des comportements inattendus du logiciels incluant ceux rencontré lors des tests

- ❖ **testabilité**

- ❖ mesure de l'effort à fournir pour réaliser les tests d'un logiciel modifié



# Mesurer la maintenabilité

Exemple de métriques utilisé pour déterminer le *score* de maintenabilité d'un projet.  
Voici les critères utilisé à la NASA [Stark et al., 1994] :

- ❖ **taille du logiciel**
- ❖ **effectif personnel affecté au logiciel**
- ❖ **procédés des requêtes de maintenance**
- ❖ **procédés d'amélioration du logiciel**
- ❖ **programme de ressources des ordinateurs**
- ❖ **densité d'une erreur (mesure à définir)**
- ❖ **volatilité d'un logiciel**
- ❖ **durée d'un rapport de désaccords (entre spécification et logiciel)**
- ❖ **ratio break/fix**
- ❖ **fiabilité d'un logiciel**
- ❖ **complexité de conception**
- ❖ **distribution du type erreur.**

# Les problèmes de la gestion de la maintenance

- ❖ **s'aligner avec les objectifs de l'entreprise (RoI)**
  - ❖ objectif de livraison et de longévité du logiciel (satisfaire le client).
  - ❖ la maintenance est perçue par les investisseurs comme une activité à forte consommation des ressources, sans bénéfices clairs ou quantifiables
- ❖ **constituer une équipe de maintenance**
  - ❖ souvent perçu par les collaborateurs comme une tâche ingrate et insatisfaisante
    - ❖ manque de maîtrise et manque de techniques et outils
  - ❖ la mise en commun du développement et de la maintenance dans un même processus est souvent mal justifiée pour les investisseurs.
- ❖ **l'organisation de la maintenance**
  - ❖ décrivent comment identifier quelle organisation et/ou service sera responsable de la maintenance du logiciel
- ❖ **externaliser la maintenance**
  - ❖ Les logiciels les moins critiques sont le plus souvent sujets à l'externalisation par crainte de perdre le contrôle sur les logiciels qui sont au cœur du métier de l'entreprise.
  - ❖ un des défis majeurs pour ceux qui fournissent le service externalisé est de déterminer la portée des services de maintenance requis et les détails contractuels.

# Maintenance Interne vs. Externe

Pro vs Con de réaliser la maintenance d'un logiciel par ses développeurs (**interne**)

## ❖ Avantages

- ❖ Les développeurs ont une meilleure connaissance du système
- ❖ la compréhension logiciel est facilitée
- ❖ pas besoin de système de communication (formel ou informel) entre développeurs et mainteneurs
- ❖ le client ne traite qu'avec *une* seule entreprise
- ❖ apportent une diversité de tâche à l'entreprise

## ❖ Désavantages

- ❖ représente une tâche à risque et beaucoup d'effort (peut isoler les développeurs et donc les forcer à quitter l'entreprise)
- ❖ peut repousser les nouveaux employés
- ❖ en cas de tourne over, il faut assurer la reprise de la maintenance par les prochains développeurs
- ❖ les développeurs ne sont pas forcément de bon mainteneurs (*puzzle solving issu*)

# Maintenance Interne vs. Externe

Pro vs. Con d'externaliser la maintenance d'un logiciel (**externe**)

## ❖ Avantages

- ❖ travailler effectué par des experts (potentiellement un meilleur résultat)
- ❖ pas de problème de frustration des développeurs interne
- ❖ les mainteneurs connaissent les forces et faiblesses du projet
- ❖ procédures d'implémentation des modifications sont pris en compte et établie.

## ❖ Désavantages

- ❖ la compréhension logiciel prend du temps
- ❖ des problèmes financiers peuvent être rencontré par l'entreprise des développeurs
- ❖ peut déboucher vers un mauvais support utilisateur

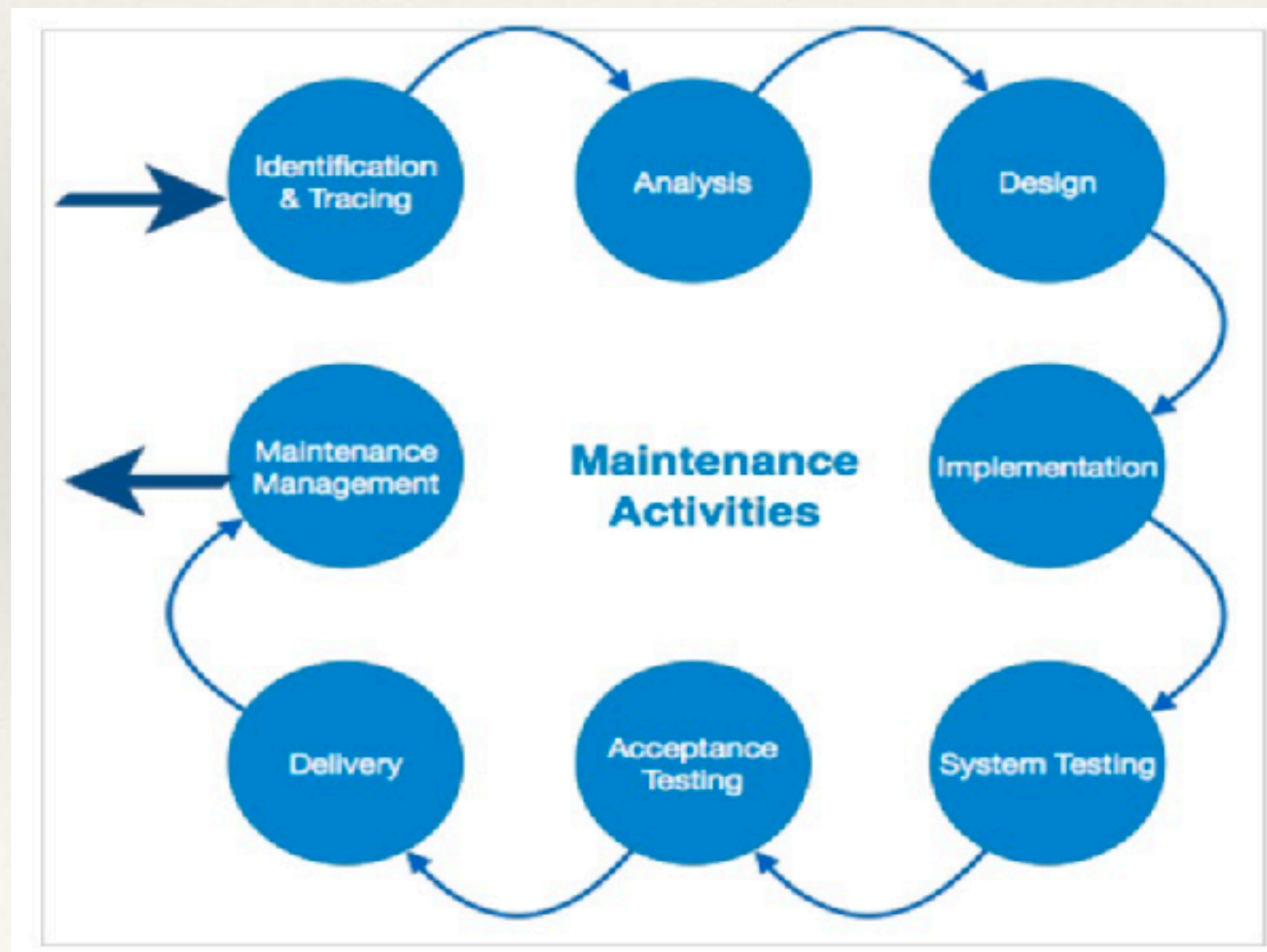


# Estimer les coûts de la maintenance

- ❖ définis sur les **critères techniques et non technique**
- ❖ D'après ISO/IEC 14764
  - ❖ les deux approches les plus populaires pour l'estimation des ressources pour la maintenance de logiciels sont :
    - ❖ l'utilisation de **modèles paramètres**
      - ❖ ex. COCOMO pour estimer les coûts d'un logiciel [Boehm 1981]
    - ❖ l'utilisation de **l'expérience** (expérience des mainteneurs)

# Exemple d'un processus de maintenance

- ❖ Processus de maintenance
- ❖ IEEE 1999 publie un processus standard de la maintenance
- ❖ IEEE Standard for Software Maintenance
- ❖ elle prend en entrée une demande de modification à effectuer sur le logiciel



# Le processus de maintenance

## 1. Identification & Tracing

- ❖ identification du type de modification (corrective, perfective, adaptative)
- ❖ détermine la priorité de la modification

## 2. Analysis

- ❖ étude de faisabilité, analysé détaillé (plan d'action) et analyse des risques
- ❖ évaluation des impacts
- ❖ proposer plusieurs plans d'actions possible et les chiffrer
- ❖ planification des actions à faire pour la modification

## 3. Design

- ❖ conception de la modification s'appuyant sur l'analyse,
- ❖ en fin de conception, la documentation doit être mise à jour (test, plans, exigences, etc)

# Le processus de maintenance

## 4. Implémentation

- ❖ comprend tout le développement nécessaire pour l'ajout de modification dans le produit
- ❖ tous les documents doivent être mise à jour (commentaire, documentation, UML, doc utilisateur, etc)

## 5. Test du Système

- ❖ s'assurent que les exigences d'origine sont toujours présentes
- ❖ test la modification

## 6. Test d'acceptation

- ❖ s'assure que les modifications sont OK pour le client.
- ❖ les testeurs doivent faire remonter tous problèmes aux mainteneurs

## 7. Delivery

- ❖ Livraison du logiciel modifié au clients. il doit être opérationnelle

## 8. Maintenance et management

- ❖ redémarre un cycle de maintenance



# Quelles pratiques pour améliorer la maintenabilité ?

- ❖ Produire une **documentation claire et précise** (UML, Diagramme use-case, cahier des charges à jour, etc)
- ❖ **Appliquer des *design patterns***
  - ❖ Il existe une corrélation forte entre l'application des *design pattern* et une bonne maintenabilité logicielle [Hegedüs 2013]
  - ❖ Le *design pattern* doit être approprié (sinon on parle d'*anti-pattern*)
- ❖ Pratiquer une **maintenabilité proactive**
  - ❖ ⚠ Ne pas confondre optimisation et maintenabilité !
- ❖ **Appliquer un refactoring** qui facilite la maintenabilité
  - ❖ Cela permet de changer la structure interne sans modifier le comportement externe

# Quelles pratiques pour améliorer la maintenabilité ?

- ❖ Concevoir une **architecture la plus modulaire possible**
  - ❖ Essayer tant que possible de d'isoler physiquement les caractéristiques (*features*) dans des briques logicielles distinctes
  - ❖ Les architectures orientées composants sont plus simple à maintenir que les architectures orientées objets.
- ❖ **Privilégier des technologies viables sur le long terme**
  - ❖ Éviter les Frameworks « révolutionnaire qui viennent de sortir »
  - ❖ Éviter les nouveaux langages

# Conclusion

- ❖ La compréhension Logiciel est fondamental dans le processus de maintenance
  - ❖ Malheureusement, elle n'est pas formalisée et son application se fait souvent par l'expérience du mainteneur
  - ❖ Elle est facilitée par la disponibilité de tous documents lié à la conception du produit à maintenir.
- ❖ La maintenance logiciel est fondamental et inhérente à tout développement de logiciel
  - ❖ Elle représente une part très coûteuse dans le cycle de vie d'un logiciel.
  - ❖ Malgré les tentatives de formalisation (Norme ISO/IEC 14764 et autres) elle reste encore trop informelle

Il est donc essentiel de fournir une documentation détaillée et de prévoir dès le début une architecture logiciel favorable à la maintenance.

# Aller Plus loin

- [1] U. Akhlaq et M. U. Yousaf, « Impact of Software Comprehension in Software Maintenance and Evolution », Blekinge Institute of Technology, Sweden.
- [2] M. Riaz, E. Mendes, et E. D. Tempero, « A systematic review of software maintainability prediction and metrics », in Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, October 15-16, 2009, Lake Buena Vista, Florida, USA, 2009, p. 367–377, doi: [10.1109/ESEM.2009.5314233](https://doi.org/10.1109/ESEM.2009.5314233).
- [3] A. SELLAMI, « ANALYSE COMPARATIVE DES MODÈLES DE MAINTENANCE DU LOGICIEL ENTRE SWEBOK, ISO/IEC 14764 ET LA LITTÉRATURE », Mémoire, Quebec, Montréal, 2000.
- [4] M.-A. D. Storey, « Theories, Methods and Tools in Program Comprehension: Past, Present and Future », in *13th International Workshop on Program Comprehension (IWPC 2005)*, 15-16 May 2005, St. Louis, MO, USA, 2005, p. 181–191, doi: [10.1109/WPC.2005.38](https://doi.org/10.1109/WPC.2005.38).