

Génie Logiciel

(d'après A.-M. Hugues, D. Wells)

Cycle de vie

Renaud Marlet

LaBRI / INRIA

<http://www.labri.fr/~marlet>

Quand commence la construction d'un logiciel ?

- quand on écrit la première ligne de code ?
- quand on a planifié son développement ?
- quand on a écrit la spécification ?
- quand on a écrit le cahier des charges ?
- quand on a terminé l'étude de marché ?
- ...

Votez

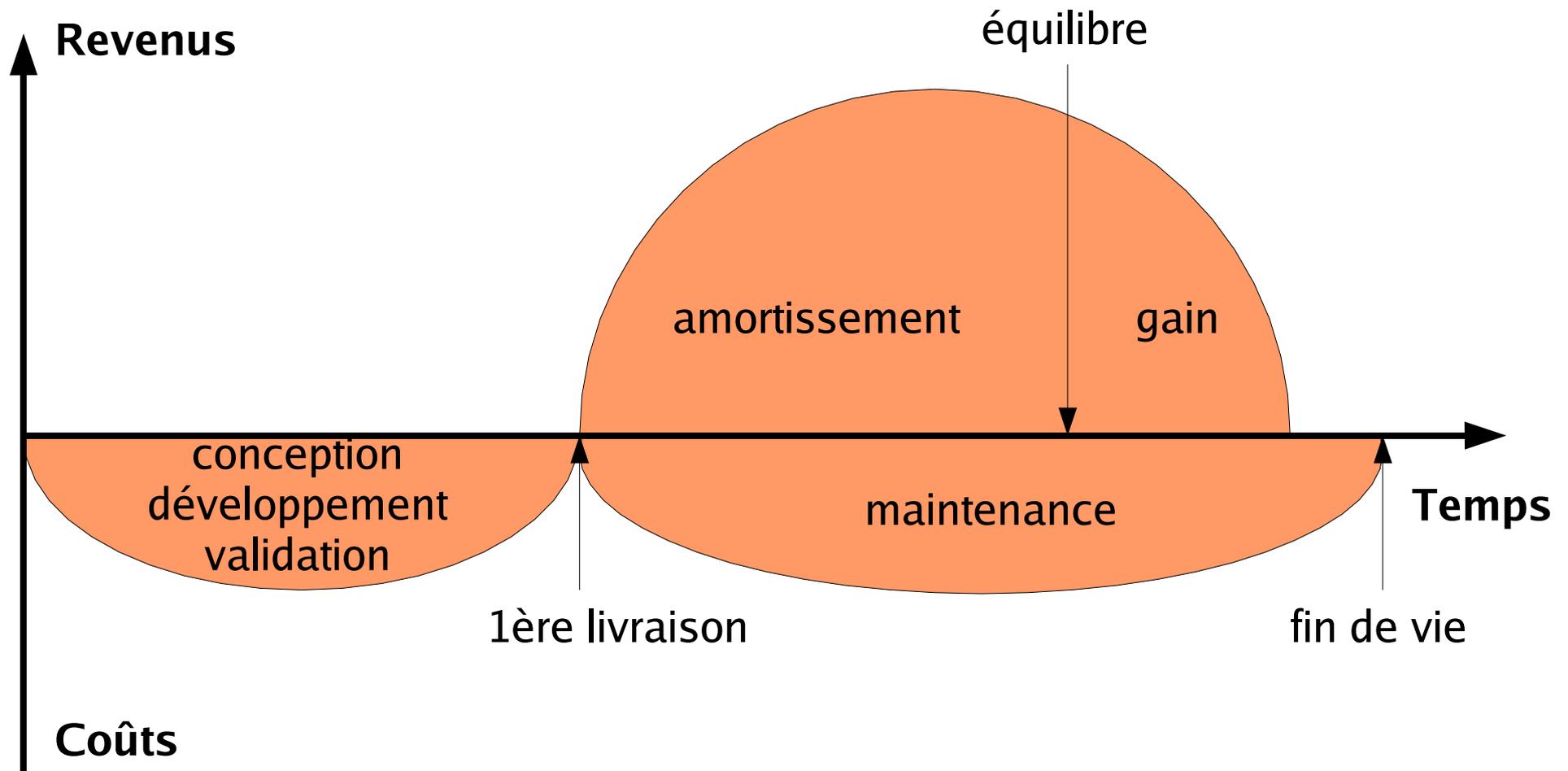
Quand un logiciel est-il terminé ?

- quand on a fini de le programmer ?
- quand on l'a compilé ?
- quand il s'exécute sans se planter ?
- quand on l'a testé ?
- quand on l'a documenté ?
- quand il est livré au premier client ?
- quand il n'évolue plus ?
- quand il n'est plus maintenu ?

Votez

Vie du logiciel

(d'après J. Printz)



Pourquoi se préoccuper d'un « cycle de vie » ?

- C'est un processus
 - phases : création, distribution, disparition
- But du découpage
 - maîtrise des risques
 - maîtrise des délais et des coûts
 - contrôle que la qualité est conforme aux exigences (→)
- En fait, problématique plus générale
 - mais spécificités relatives aux logiciels

Quelques chiffres

(d'après B. Boehm)

- Logiciel de réservation aérienne d'United Airlines
 - 56 millions de dollars
- Faille : nombre d'instructions par transaction
 - 146.000 au lieu de 9.000 prévues
- Inutilisable par manque d'analyse
 - besoins, étude de faisabilité
- Aurait pu être évité
 - par des inspections et revues intermédiaires

Pourquoi se préoccuper d'un « cycle de vie » ?

- C'est un processus
 - phases : création, distribution, disparition
- But du découpage
 - maîtrise des risques
 - maîtrise des délais et des coûts
 - contrôle que la qualité est conforme aux exigences (←)
- En fait, problématique plus générale
 - mais spécificités relatives aux logiciels

Cycle de vie du logiciel

- Définition des besoins (cahier des charges)
- Analyse des besoins (spécification)
- Planification (gestion de projet)
- Conception
- Développement (codage, test, intégration)
- Validation
- Qualification (mise en situation)
- Distribution
- Support

- Définition des besoins
- Spécification
- Planification
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase - Définition des besoins

(on dit aussi « expression des besoins »)

- **Activité (du client, externe ou interne) :**
 - consultation d'experts et d'utilisateurs potentiels
 - questionnaire d'observation de l'utilisateur dans ses tâches
- **Production :**
 - cahier des charges (les « exigences »)
 - fonctionnalités attendues
 - contraintes non fonctionnelles
 - qualité, précision, optimalité, ...
 - temps de réponse, contraintes de taille, de volume de traitement, ...

- Définition des besoins
- **Spécification**
- Planification
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase -

Analyse des besoins / Spécification

(on dit aussi « définition du produit »)

- **Productions :**
 - dossier d'analyse
 - spécifications fonctionnelles
 - spécifications non-fonctionnelles
 - ébauche du manuel utilisateur (interface attendue)
 - première version du glossaire (être bien compris)
- **À l'issue :**
 - client et fournisseur OK sur produit et contraintes

- Définition des besoins
- Spécification
- **Planification**
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase -

Planification / Gestion de projet

- **Activités :**
 - tâches : découpage, enchaînement, durée, effort (→)
 - choix : normes qualité, méthodes de conception
- **Productions :**
 - plan qualité
 - plan projet destiné aux développeurs
 - estimation des coûts réels, devis
 - liste de dépendances extérieures (risques)

Unité de mesure de l'effort

Travail d'un homme pendant 1 mois (ou 1 an)

- homme(s)-mois [anglais : man-month]
- homme(s)-an(s) [anglais : man-year]

☛ Attention : \neq durée de développement

Quiz : combien d'hommes-ans pour la navette spatiale (logiciels) ?

?

Quiz : combien d'hommes-ans pour la navette spatiale (logiciels) ?

> 1000 hommes-ans

Unité de mesure de l'effort

Attention! (☹)

- 1 homme \neq 1 homme (compétences)
- 1 mois \neq 1 mois (congés)
- tendance naturelle à la sous-estimation

Exercice

- Hypothèse : on a évalué qu'un logiciel nécessitait 18 hommes-mois de développement
- Question : combien de personnes faut-il pour le réaliser en 6 mois?
 - $N < 3$
 - $N = 3$
 - $N > 3$
- Pourquoi ?

- Définition des besoins
- Spécification
- **Planification**
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase -

Planification / Gestion de projet

- **Activités :**
 - tâches : découpage, enchaînement, durée, effort (↔)
 - choix : normes qualité, méthodes de conception
- **Productions :**
 - plan qualité
 - plan projet destiné aux développeurs
 - estimation des coûts réels, devis
 - liste de dépendances extérieures (risques)

- Définition des besoins
- Spécification
- Planification
- **Conception**
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase -

Conception (globale, détaillée)

- **Activités :**
 - architecture du logiciel
 - interface entre les différents modules
- **Productions :**
 - dossier de conception
 - plan d'intégration
 - plans de tests
 - planning mis à jour

- Définition des besoins
- Spécification
- Planification
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase -

Codage et tests unitaires

- Activités :
 - chaque module codé et testé indépendamment
- Productions :
 - modules codés et testés (→)
 - documentation de chaque module
 - résultats des tests unitaires
 - planning mis à jour

Unité de mesure de la taille (volume)

Nombre de lignes de code source

- y compris commentaires et lignes vides
- anglais : source line of code (LOC ou SLOC)

Quiz : combien de LOC pour la navette spatiale ?

?

Quiz : combien de LOC pour la navette spatiale ?

2,2 millions de LOC

(~ 50.000 pages)

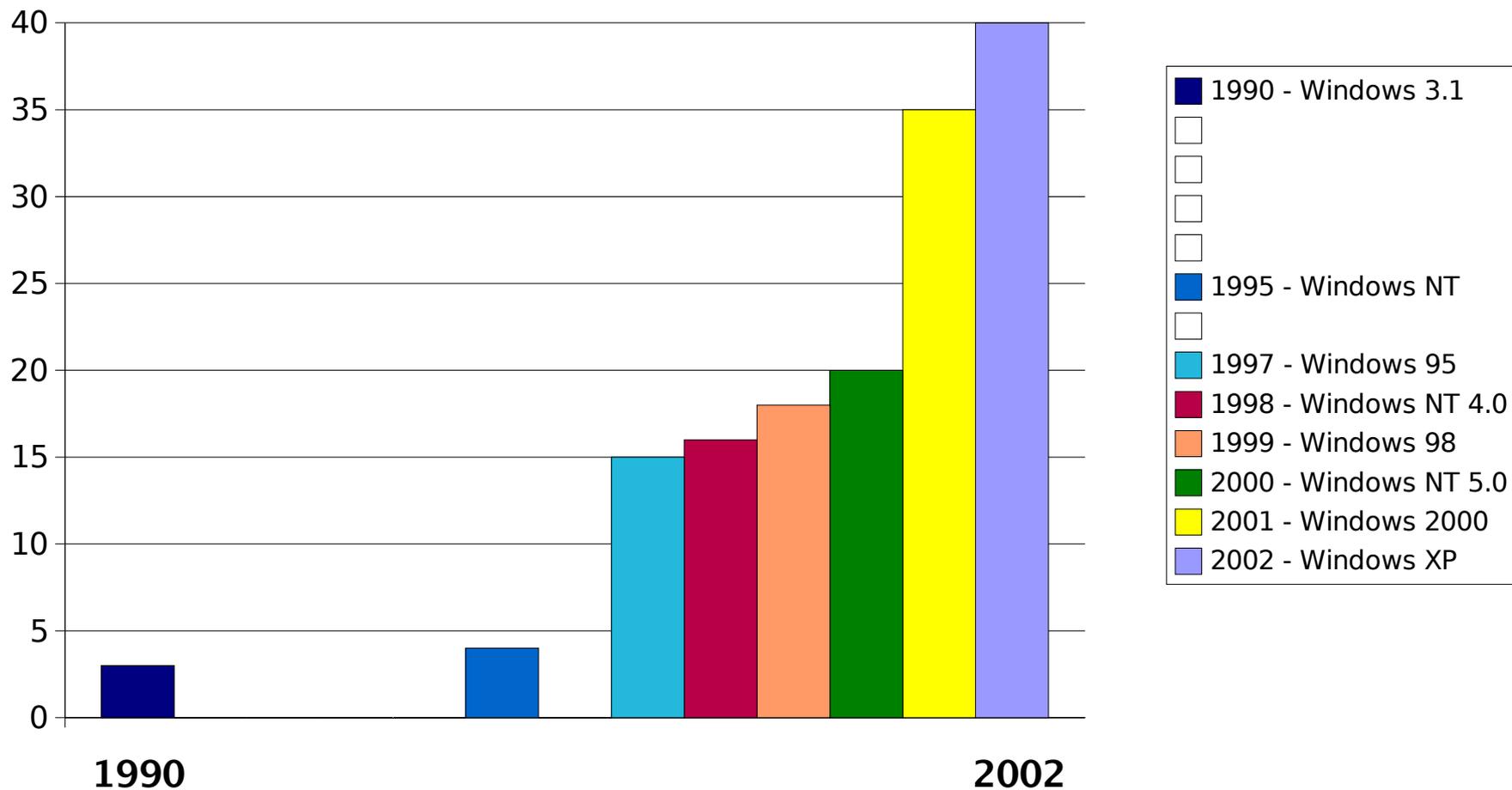
Unité de mesure de la taille (volume)

Attention! (☹)

- facile à calculer
- mais approximatif (différences de mise en page)
- en pratique, pas d'écarts énormes entre codeurs
- standard de fait (mais il existe de meilleures mesures)

Quelques chiffres : évolution de la taille de Windows

millions de lignes de code source (LOC)



Quelques chiffres

(d'après D. Wheeler)

<i>OS</i>	<i>Solaris</i>	<i>Red Hat 6.2</i>	<i>Red Hat 7.1</i>	<i>Debian 2.2</i>
Année		2000	2001	2000
LOC (million)	7,5	17	30	56
Hommes-ans		5000	8000	14000
Millions de \$		620	1000	1900

Red Hat, en un an : +60%

Coût d'une ligne de code

Dans les années 1990, pour un projet de TI moyen

- 100 \$US / ligne

Pour un projet complexe comme la refonte du système de gestion du trafic aérien de la FAA (Federal Aviation Administration)

- budgété : 500 \$US / ligne
- réalisé : 800 \$US / ligne

Quiz Unix : comment compter (le total de) toutes les LOC de C ?

Dans un répertoire :

-
-

Dans un répertoire et tous ses sous-répertoires :

(petit nombre de fichiers)

-
-

(grand nombre de fichiers)

-

Quiz Unix : comment compter (le total de) toutes les LOC de C ?

Dans un répertoire :

```
wc -l *. [ch]
```

```
cat *. [ch] | wc -l
```

cf. « Manipulations de fichiers
et de données sous Unix »

Dans un répertoire et tous ses sous-répertoires :

(petit nombre de fichiers)

```
wc -l `find . -name '*. [ch] '`
```

```
cat `find . -name '*. [ch] '` | wc -l
```

(grand nombre de fichiers)

```
find . -name '*. [ch] ' | xargs cat | wc -l
```

- Définition des besoins
- Spécification
- Planification
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase -

Codage et tests unitaires

- Activités :
 - chaque module codé et testé indépendamment
- Productions :
 - modules codés et testés (←)
 - documentation de chaque module
 - résultats des tests unitaires
 - planning mis à jour

- Définition des besoins
- Spécification
- Planification
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase - Intégration

- Activités :
 - modules testés intégrés suivant le plan d'intégration
 - test de l'ensemble conformément au plan de tests
- Productions :
 - logiciel testé
 - jeu de tests de non-régression
 - manuel d'installation
 - version finale du manuel utilisateur (→)

À propos d'informations sur le logiciel pour l'utilisateur

- Manuel d'installation cf. « Documentation »
 - guide dans les choix d'installation
- Tutoriel :
 - introduction pédagogique, exemples d'utilisation
- Manuel de l'utilisateur (user's guide)
 - comment faire « ça », non exhaustif
- Manuel de référence (reference manual)
 - liste exhaustive de chaque fonctionnalité

À propos d'informations sur le logiciel pour l'utilisateur

- Dans le logiciel :
 - aide en ligne
 - thématique (proche du manuel de l'utilisateur)
 - par index (proche du manuel de référence)
 - recherche
 - messages d'erreur
 - bulles d'aide
 - trucs et astuces (tips)

- Définition des besoins
- Spécification
- Planification
- Conception
- Développement
- Validation
- **Qualification**
- Distribution
- Support

- Phase - Qualification

- **Activités :**
 - test en vraie grandeur dans les conditions normales d'utilisation
- **Production :**
 - diagnostic OK / KO

- Définition des besoins
- Spécification
- Planification
- Conception
- Développement
- Validation
- Qualification
- Distribution
- **Support**

- Phase - Support

- **Activités :**
 - formation (documents, cours, vidéos, ...)
 - maintenance
- **Productions :**
 - utilisateurs formés
 - logiciel maintenu

- Définition des besoins
- Spécification
- Planification
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase - Maintenance

- Activités :
 - maintenance corrective : correction des bugs
 - maintenance évolutive : adaptative ou perfective
- Productions :
 - logiciel corrigé
 - mises à jour, patch
 - documents corrigés

- Définition des besoins
- Spécification
- Planification
- Conception
- Développement
- Validation
- Qualification
- Distribution
- Support

- Phase - Maintenance

Capitalisation sur :

- la qualité de l'auto-diagnostic
- les efforts de documentation
- la qualité de l'architecture
- la qualité du codage
- la gestion de l'historique
- la facilité à rejouer les tests
- ...

Quiz : combien coûte la maintenance (en proportion) ?

?

Quiz : combien coûte la maintenance (en proportion) ?

maintenance

=

jusqu'à $2/3$ du coût total !

Ne jamais sous-estimer la maintenance

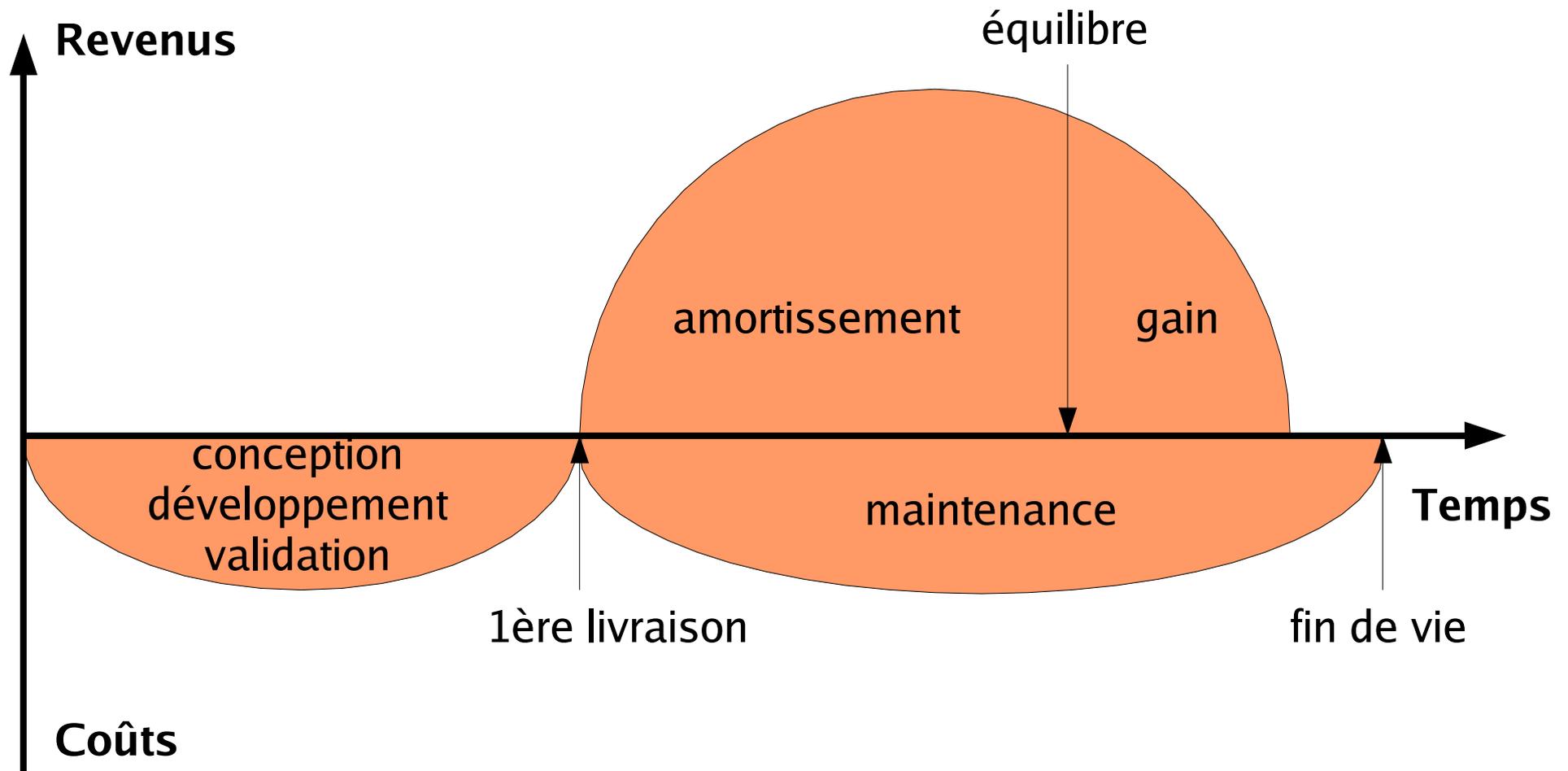
(d'après A.-M. Hugues)

Coûts

- maintenance = jusqu'à 67% du coût total
 - dont 48 % consacrés à réparer des défauts
 - 60% des défauts correspondent à des erreurs de spécification et de conception
- ☛ Ce sont des moyennes, ça peut varier d'un contexte à un autre...

Vie du logiciel

(d'après J. Printz)



Cycle de vie

Différents modèles :

- en cascade
- en V
- en spirale
- Extreme Programming (XP)
- ...

Des cycles de vie

S'il y a plusieurs modèles, c'est que :

- pas un parfait, ni même meilleur que les autres
- des qualités et des défauts, suivant les contextes
- suivre les grandes lignes, sans règle dogmatique
- garder un regard critique

Attention! (☹)

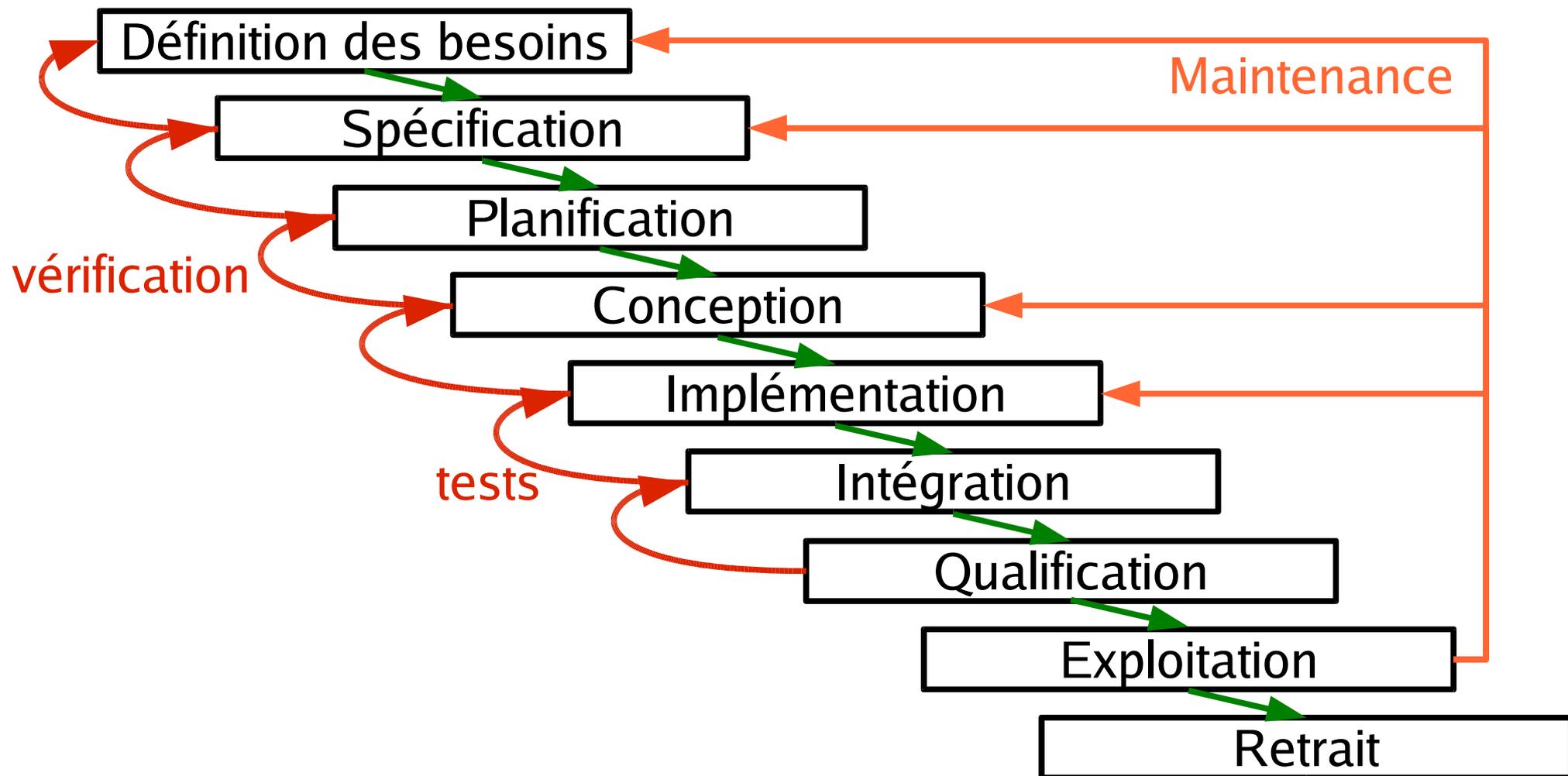
- certaines recommandations sont contre-intuitives
- et pourtant peuvent être justifiées

Cycle de vie

Différents modèles :

- en cascade
- en V
- en spirale
- Extreme Programming (XP)
- ...

- Cycle de vie - Modèle en cascade



Caractéristiques du modèle en cascade

(date des années 70)
(mais reste pertinent)

- Séquentiel
- Importance du contrôle du processus
 - rétro-actions
 - validation, vérification, tests

Cycle de vie et assurance qualité

- Validation :
 - Sommes-nous en train de faire le bon produit ?
- Vérification :
 - Est-ce que nous faisons le produit correctement ?
- ☞ En pratique
 - souvent confondus, ou pris l'un pour l'autre
 - on parle de « V&V » (validation et vérification)

Cycle de vie et assurance qualité

- Inspections
 - lecture critique : spécification, conception, code, ...
(critique constructive : ne pas tout jeter)
 - faite par équipes indépendantes
 - rédaction de fiches de défaut
 - affectation de responsabilités pour la correction des défauts
- Revues
 - validation successive des phases du cycle de vie

Cycle de vie et assurance qualité

Attention! (☹)

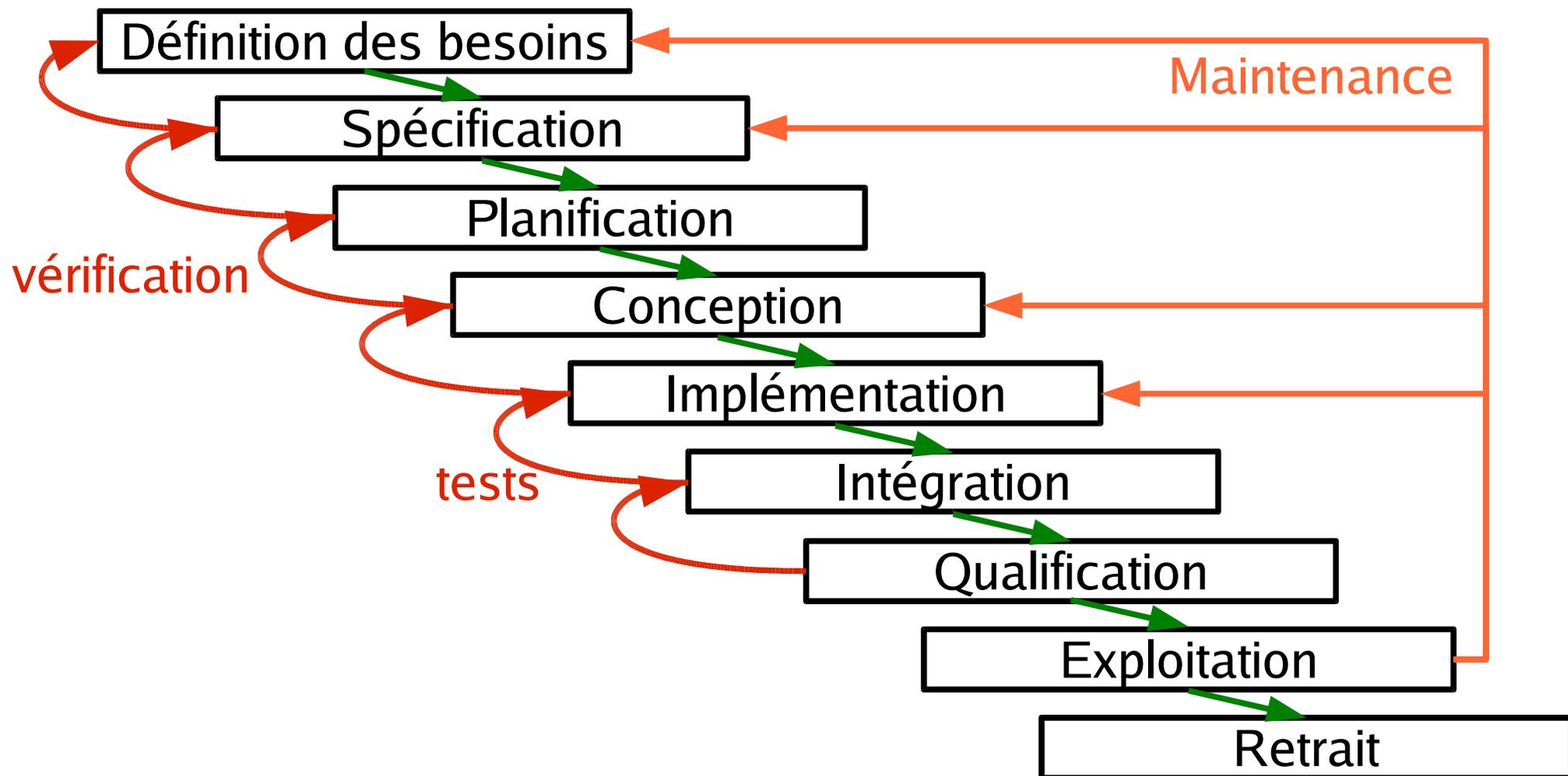
- Plus une erreur est découverte tard dans cycle de vie, plus la réparation est coûteuse

Erreur de spécification trouvée en maintenance

- ☛ coûte + de 100 fois + cher que si trouvée lors des spécifications

Critique du modèle en cascade : que peut-on lui reprocher ?

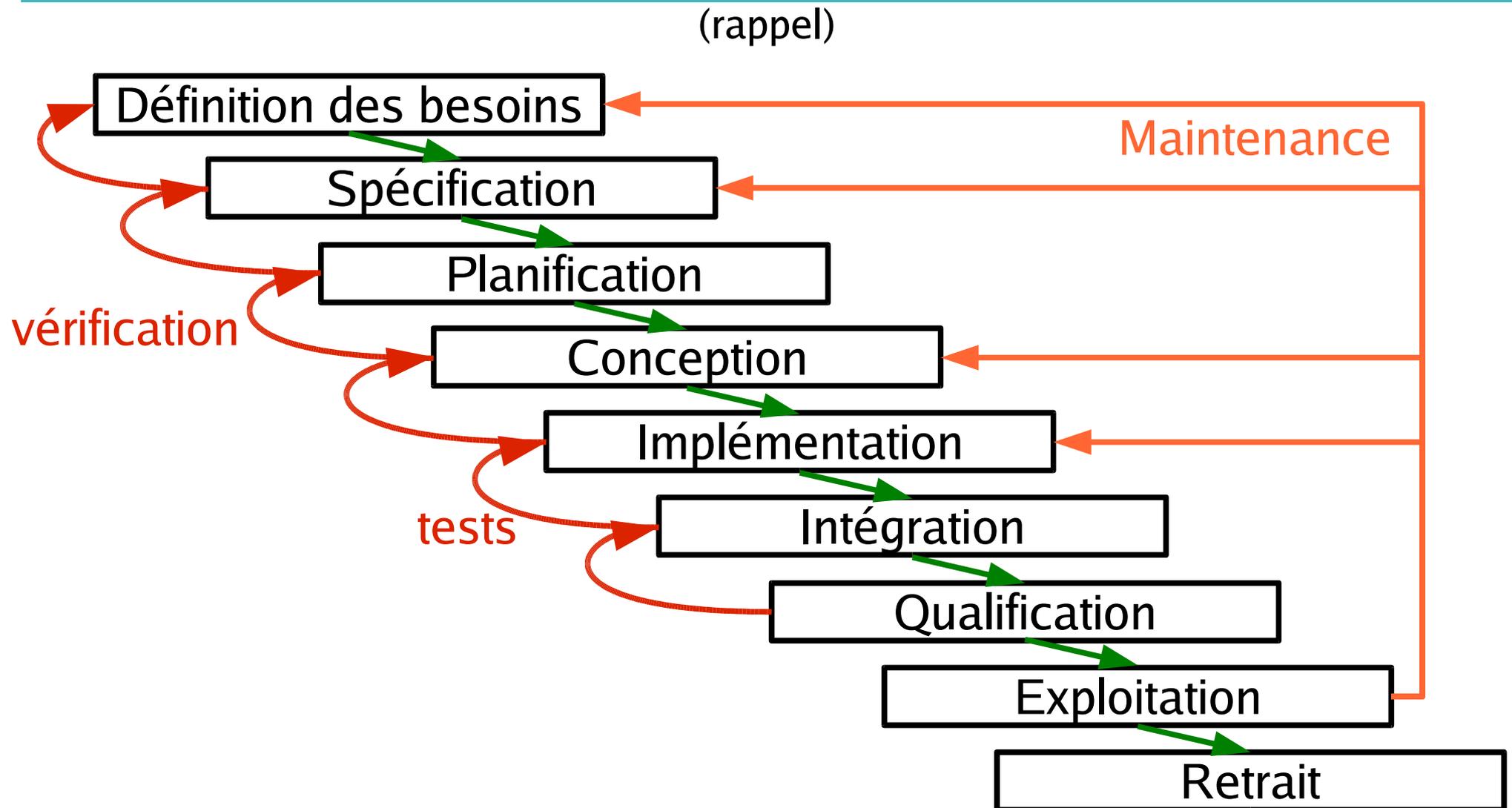
(rappel)



Critique du modèle en cascade

- Modèle trop séquentiel
 - dure trop longtemps
- Validation trop tardive
 - et remise en question coûteuse des phases précédentes
- Sensibilité à l'arrivée de nouvelles exigences
 - refaire toutes les étapes

Comment améliorer le modèle en cascade ?

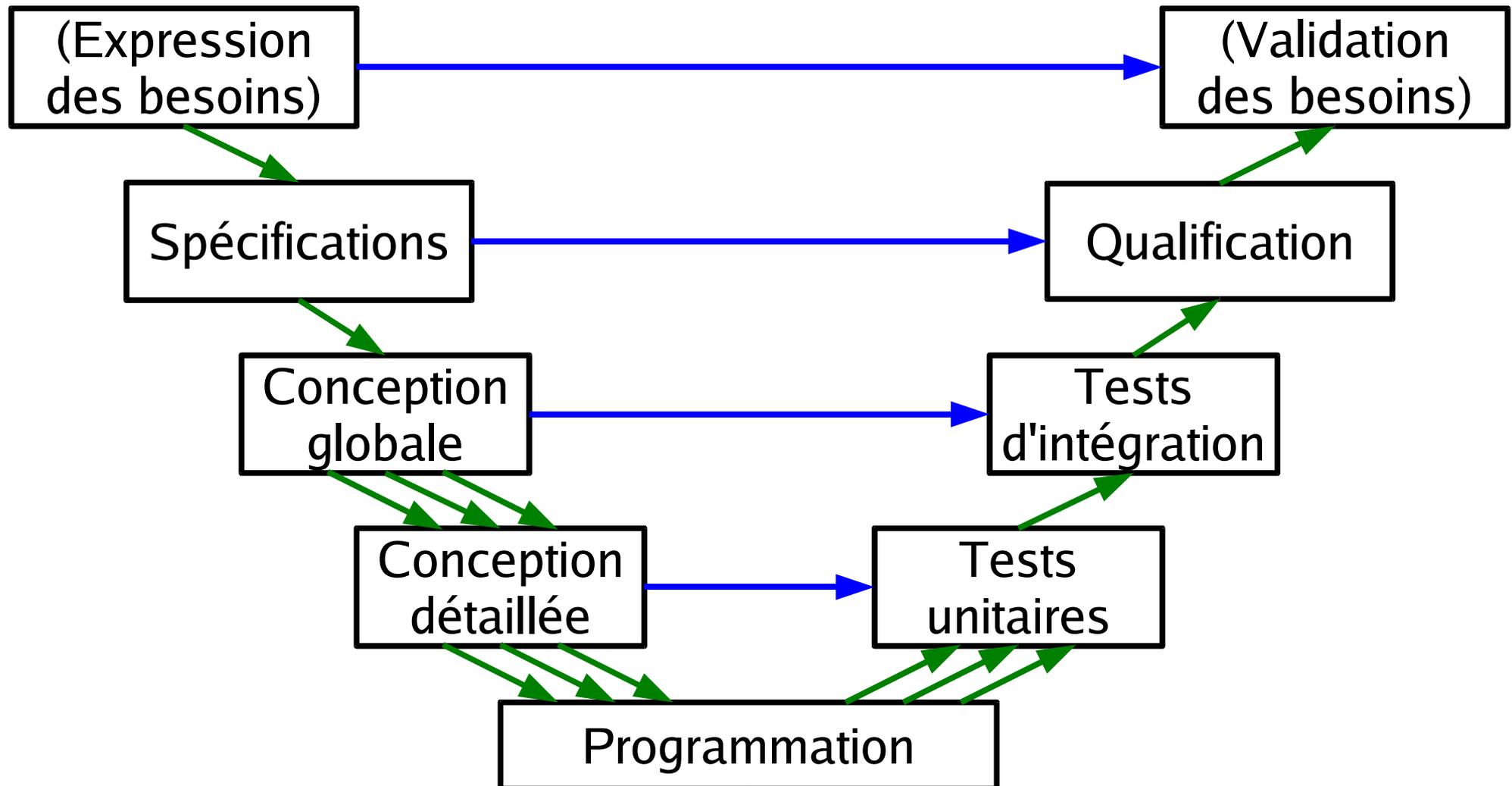


Cycle de vie

Différents modèles :

- en cascade
- en V
- en spirale
- Extreme Programming (XP)
- ...

Cycle de vie : modèle en V



Caractéristiques du modèle en V

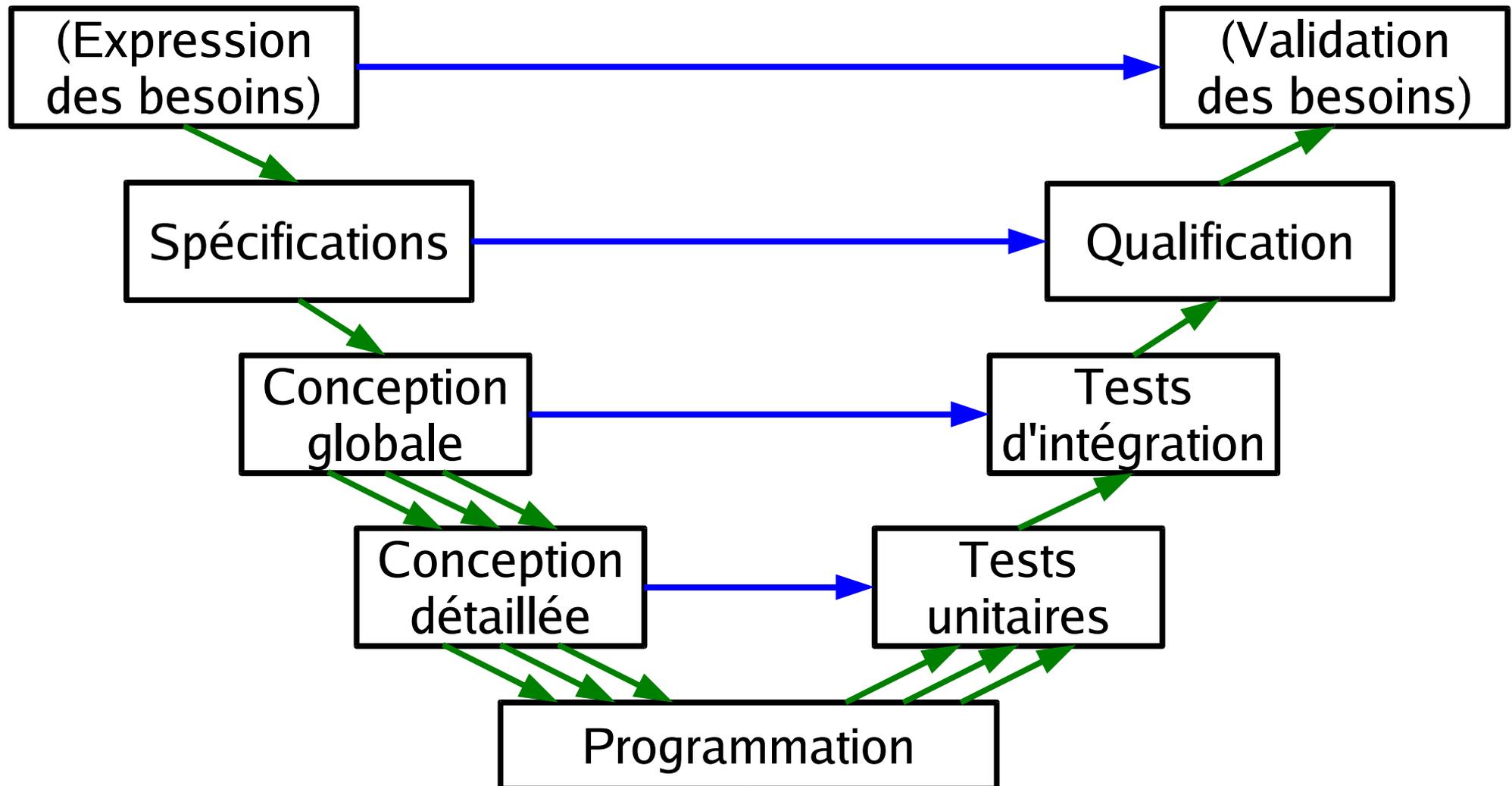
- Tâches effectuées en parallèle
 - horizontalement : préparation de la vérification
 - Ex. : dès que la spécification fonctionnelle est faite : (↑)
 - plan de tests de qualification
 - plan d'évaluation des performances
 - documentation utilisateur
 - verticalement : développement des modules
 - Ex. : dès que la conception globale est validée : (↑)
 - conception détaillée des modules
 - programmation et tests unitaires

Tâches effectuées en parallèle

- Importance du parallélisme
 - Rappel : navette spatiale >1000 hommes-ans
- Augmenter / favoriser le parallélisme
 - tâches aussi indépendantes que possible
 - développement des modules, tests unitaires, ...
- Mais ne pas oublier
 - le contrôle
 - le plan d'intégration

Critique des modèles cascade et en V: que peut-on leur reprocher ?

(rappel)

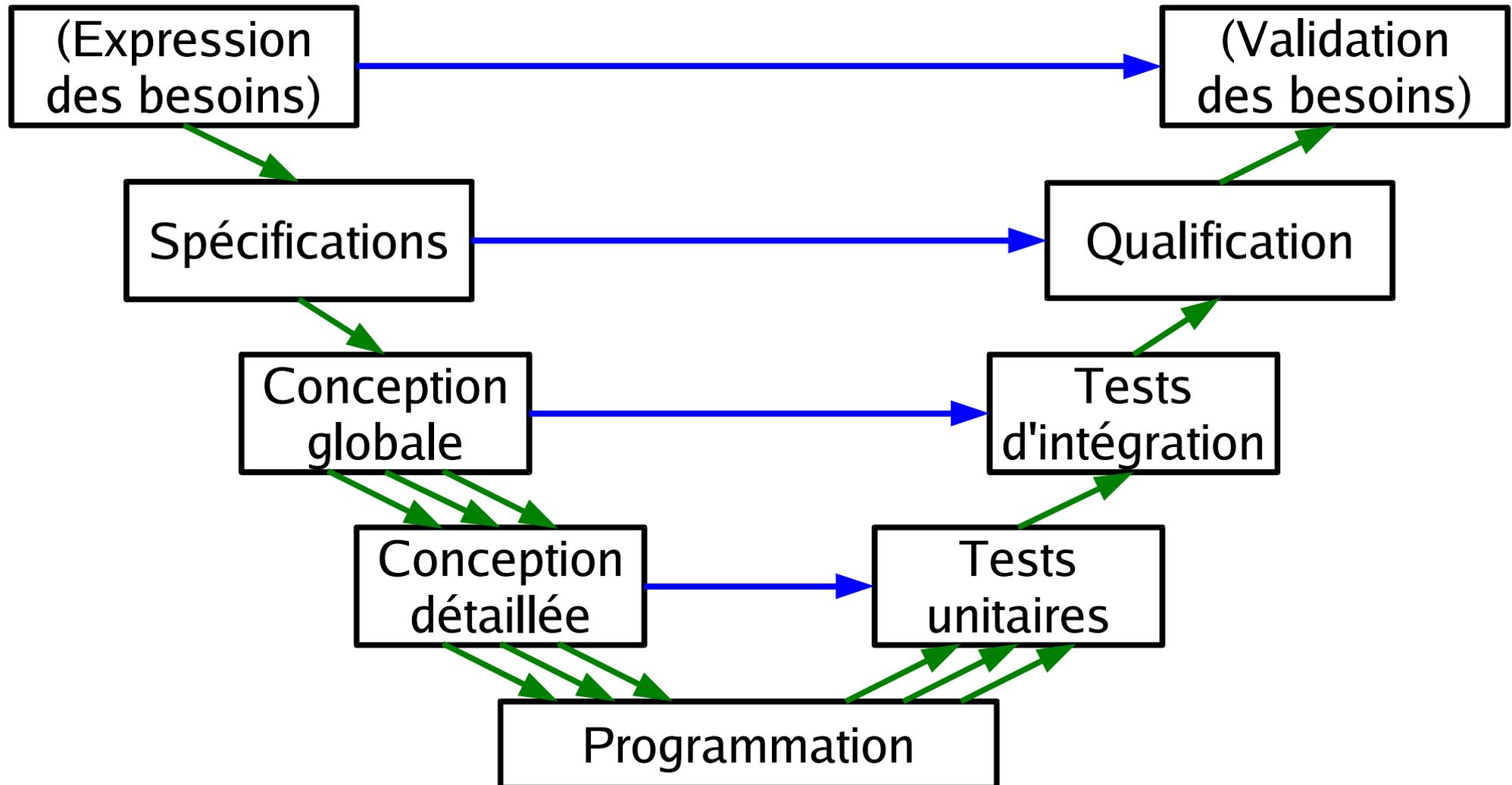


Critique des modèles en cascade et en V

- Modèles parfois difficiles à appliquer :
 - difficile de prendre en compte des changements importants dans les spécifications dans une phase avancée du projet
 - durée parfois trop longue pour produits compétitifs
- Gestion du risque :
 - trop de choses reportées à l'étape de programmation (par ex. l'interface utilisateur)
 - pas assez de résultats intermédiaires pour valider la version finale du produit

Comment améliorer ces modèles ?

(rappel)



Améliorations à apporter

- Retours d'expérience avant version finale
 - Dans l'industrie :
 - prototype : premier d'une série
 - maquette : modèle réduit (pas censé être opérationnel)
 - En développement de logiciel :
 - prototype rapide (ou « maquette »), opérationnel (mais restreint), expérimental, évolutif
- Développement incrémental
 - ajouts successifs de nouvelles fonctionnalités sur une base opérationnelle

Prototypage rapide (ou maquettage)

- Construit et utilisé lors des phases suivantes :
 - analyse des besoins
 - spécifications fonctionnelles
- Validation des spécifications par l'expérimentation
« Je saurai ce que je veux quand je le verrai ! »
- Client et développeur d'accord sur le produit
 - nature, interface, fonctionnalités
 - rapidité : réduction des allers-retours

[cf. « Génie Logiel - Introduction » (facteur de succès ou d'échec)]

Prototypage expérimental

- Construit et utilisé en phase de conception
 - s'assurer de la faisabilité de parties critiques
 - valider des options de conception
- Statut
 - En général, jeté après développement
 - Si gardé, alors s'appelle « prototype évolutif »

Prototypage évolutif

- Principe :
 - Première version du proto = embryon du produit final
 - On itère jusqu'au produit final
- Problèmes :
 - difficulté à mettre en œuvre des procédures de validation et de vérification
- ☞ Voir
 - modèle en spirale, développement incrémental

Cycle de vie

Différents modèles :

- en cascade
- en V
- **en spirale**
- Extreme Programming (XP)
- ...

- Cycle de vie - Modèle en spirale

Principe :

- Identifier les risques, leur affecter une priorité
- Développer des prototypes pour réduire les risques, en commençant par le plus grand risque
- Utiliser un modèle en V ou en cascade pour implémenter chaque cycle de développement
- Contrôler :
 - si un cycle concernant un risque est achevé avec succès, évaluer le résultat du cycle, planifier le cycle suivant
 - si le risque est non résolu, interrompre le projet

Caractéristiques du modèle en spirale

(Boehm, 1988)

- Utilisation du prototypage
- Analyse (progressive) des risques

Analyse des risques (1)

Risques technologiques :

- exigences démesurées par rapport à la technologie
- incompréhension des fondements de la technologie
- problèmes de performance
- dépendance en un produit soi-disant miracle
- changement de technologie en cours de route
- ...

Analyse des risques (2)

Risques liés au processus :

- gestion projet mauvaise ou absente
- calendrier et budget irréalistes
- calendrier abandonné sous la pression des clients
- composants externes manquants
- tâches externes défailtantes
- insuffisance de données
- invalidité des besoins
- développement de fonctions inappropriées
- développement d'interfaces utilisateur inappropriées
- ...

Analyse des risques (3)

Risques humains

- défaillance du personnel
- surestimation des compétences
- travailleur solitaire
- héroïsme
- manque de motivation
- ...

Cycle de vie

- Différents modèles :
 - en cascade
 - en V
 - en spirale
 - Extreme Programming (XP)
 - ...

Extreme Programming (XP)

(d'après D. Wells)

(Modèle récent : 1996)

(= peu à l'échelle du GL, qui évolue relativement lentement)

- **Emphase sur la satisfaction du client**
 - le produit dont il a besoin
 - des livraisons aux moments où il en a besoin
 - un processus robuste si les exigences sont modifiées
- **Emphase sur le travail d'équipe**
 - managers, clients, développeurs : ensemble

Extreme Programming

Caractéristiques :

- Communication
 - programmeurs avec clients et autres programmeurs
- Simplicité
 - design propre et simple
- Feedback
 - test du code dès *le premier jour*
- Livraison
 - aussi tôt que possible
 - adaptabilité à des changements d'exigences ultérieurs

Succès de Extreme Programming (« pur » ou adapté)

- Applications :
 - comptabilité de Chrysler (gestion de 10.000 employés)
 - banque, assurance vie, automobile...
- Conférences, livres, sites, forums, outils, ...
- Sociétés spécialisées dans la formation à XP
- Associations dans le monde (cf. XP-France, ...)
- Utilisateurs de XP (entre autres modèles) :
 - IBM, Xerox, Nortel, Cisco, Symantec, Sun, ...

Extreme Programming

Attention (!)

- beaucoup de choses contre-intuitives
- certaines sont vraies dans beaucoup de contextes... pas dans tous les contextes
- une méthodologie, une philosophie, une éthique...
 - garder un regard à la fois ouvert et critique (au bon sens du terme)
 - source de réflexion, d'inspiration

Extreme Programming

Quatre types d'activité :

- Planification
- Design
- Codage
- Test

Extreme Programming : Planification

- Scénarios d'utilisation (« user's stories »)
 - écrits par les utilisateurs : ce que ça doit faire pour eux
 - utilisés à la place d'un cahier des charges (!)
 - base pour estimer les dates de livraison (priorités)
 - base pour les tests de recette
 - quantité (ordre de grandeur) : $\sim 80 \pm 20$ scénarios

Extreme Programming : Planification

- Division du projet en « itérations » successives
 - itération ~ tâche unitaire
 - durée : 1 à 3 semaines
- Planning de l'itération
 - en tout début de l'itération
 - suit mieux des besoins évolutifs
- Principe :
 - ne pas implémenter ce qui n'est pas dans l'itération
 - si semble déborder, créer une nouvelle itération

Extreme Programming : Planification

- Nombreuses livraisons intermédiaires (!)
 - retour (feedback) des clients au plus tôt
 - planifier les fonctionnalités importantes en premier
- Planning des livraisons
 - selon le nombre de scénarios à implémenter avant une date donnée
 - pour l'itérations à venir

Extreme Programming : Planification

- Mesure de la vitesse du projet
 - temps effectivement passé / temps prévu
- Déplacer les gens (!)
 - diffusion des connaissances
- Réunion « debout » quotidienne (!)
 - forme : tous les matins, courte (pas de longs meetings)
 - contenu : problèmes, solutions, focalisation
- Changer les règles quand ça ne marche pas (!)

Extreme Programming : Design

- Faire simple (!)
 - c'est plus rapide
 - c'est moins cher
- Pas de nouvelle fonctionnalité si pas utile
 - a priori, n'encourage pas la généricité et la réutilisabilité

Extreme Programming : Design

- Explorer des solutions potentielles
 - « spike » = petit programme dédié (sorte de prototype)
 - quand une difficulté technique risque de figer le développement
 - un binôme pendant 1-2 semaines pour l'étudier
 - spike généralement jeté après usage
- (Choisir une métaphore de système)
- (Design orienté objet : cartes CRC)

Extreme Programming : Design

- Restructurer sans pitié (!)
 - partout et dès que possible
 - opérations : « refactoring »
 - nettoyage
 - simplification
 - élimination des redondances et du code inutilisé
 - rajeunissement des designs obsolètes
 - gagne du temps et améliore la qualité

Extreme Programming : Codage

- Avoir le client toujours disponible (!)
 - ajout de détails au scénario pour définir le codage
 - feedback sur les développements
 - participation aux tests fonctionnels

En théorie :

- acceptable pour le client car il fait des économies (pas de cahier des charges...)

En pratique :

- disponibilité fréquente difficile à organiser
- pas facile de faire tenir ses promesses au client

Extreme Programming : Codage

- Coder les tests (unitaires) en premier (!)
 - programmation ultérieure plus facile et plus rapide
 - fixe la spécification (ambiguïtés) avant le codage
 - feedback immédiat lors du développement
 - processus :
 - créer un test pour un petit aspect du problème
 - créer le bout de code le plus simple qui passe le test
 - itérer jusqu'à ce qu'il n'y ait plus rien à tester
- (☛ temps d'écriture des tests ~ temps de programmation)

Extreme Programming : Codage

- Programmation en binôme (!)
 - deux personnes, un seul PC
 - quand l'un crée le code d'une fonction
 - l'autre se demande comment elle s'insère dans le reste
 - contre-intuitif, pourtant :
 - même délai
 - meilleure qualité
- Suivre des standards de codage
 - cohérence globale
 - code lisible / modifiable par toute l'équipe

Extreme Programming : Codage

- Intégration séquentielle (!)
 - un binôme à la fois
 - préparée en local et avant diffusion
 - réduit les pb d'intégration
 - combinatoire des problèmes unitaires
 - chasse aux bugs inexistantes (vieilles versions)

Extreme Programming : Codage

- Intégration fréquente (!)
 - > 1 fois / jour
 - évite : divergences, efforts fragmentés
 - permet : diffusion rapide pour la réutilisation
 - N.B. il faut que les tests unitaires courants soient OK

Extreme Programming : Codage

- Possession du code collective
 - tout le monde peut contribuer à tout (!)
 - nouvelles fonctionnalités, correction de bogues (bug fix), restructurations (refactoring)
 - pas de personnes dépositaires de droits exclusifs
 - OK parce que :
 - chacun teste ce qu'il développe
 - les tests unitaires sont accessibles à tous
 - intégrations fréquentes
 - extensions et réparations passent inaperçues

Extreme Programming : Codage

- N'optimiser qu'à la fin (!)
 - ne pas chercher à *deviner* les goulots d'étranglement
 - mais les *mesurer*
 - « make it work, make it right, then make it fast »
- Pas d'heures supplémentaires (!)
 - affaiblit la motivation
 - en cas de problème, changer l'objectif ou le timing
 - ne pas ajouter de nouvelles personnes pour combler le retard

Extreme Programming : Test

- Tout bout de code doit avoir ses tests unitaires
 - besoin d'un environnement (framework) de test :
 - automatisation, gestion
 - créer les tests avant de coder
 - tests archivés avec le source correspondant
 - semble coûteux mais très rentable au final
 - rend possible : restructuration, intégration fréquente

Extreme Programming : Test

- Tests de recette (acceptance tests)
 - créés à partir des scénarios des utilisateurs
 - pour tout scénario qui apparaît dans l'itération
 - un scénario → un ou plusieurs tests de recette
 - tests « boîte noire » (sans connaître le contenu)
 - teste ce que ça fait sans savoir comment ça le fait
 - aussi utilisés pour non-régression avant distribution
 - exécutés aussi souvent que possible
 - client = responsable de la validité et priorité de ces tests

Extreme Programming : Test

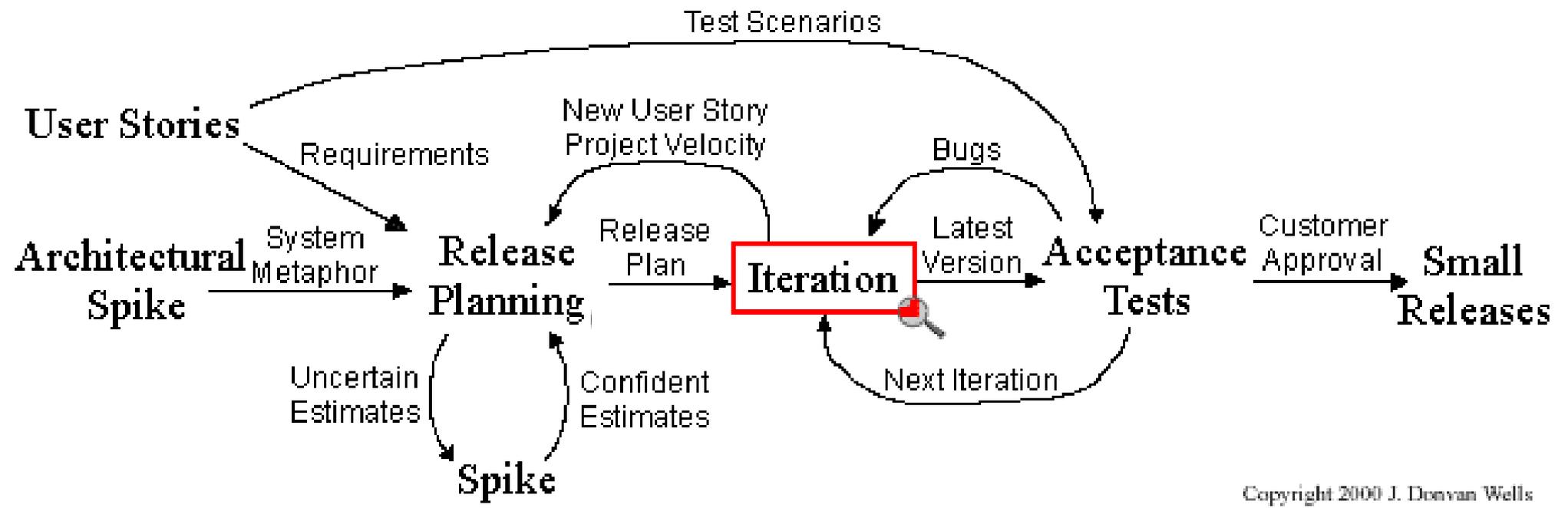
- Pas de distribution sans 100% des tests OK
- Quand un bug est trouvé, ajouter un test
 - éviter qu'il ne revienne

ExtremeProgramming.org

(d'après D. Wells)



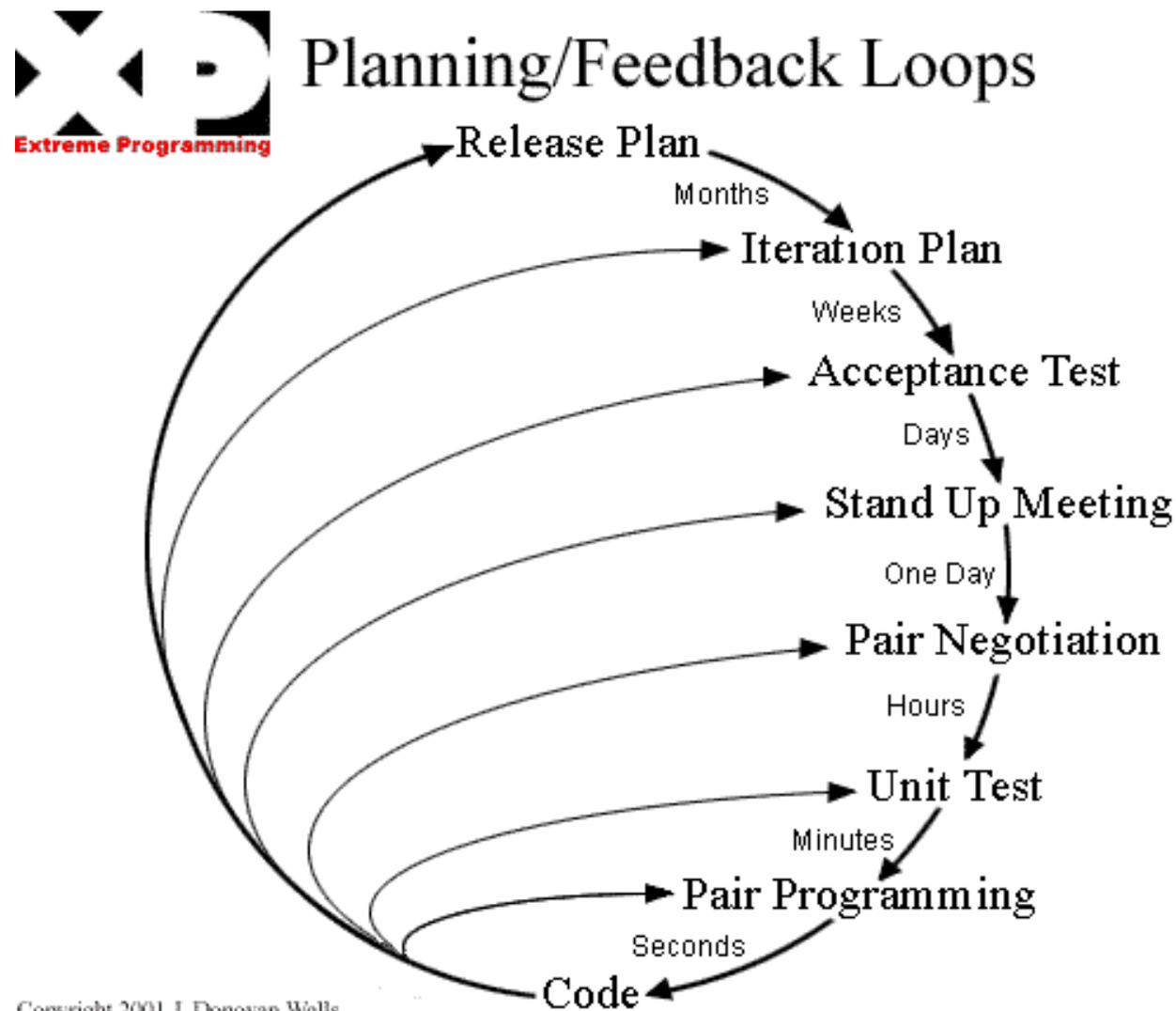
Extreme Programming Project



Copyright 2000 J. Donovan Wells

ExtremeProgramming.org

(d'après D. Wells)

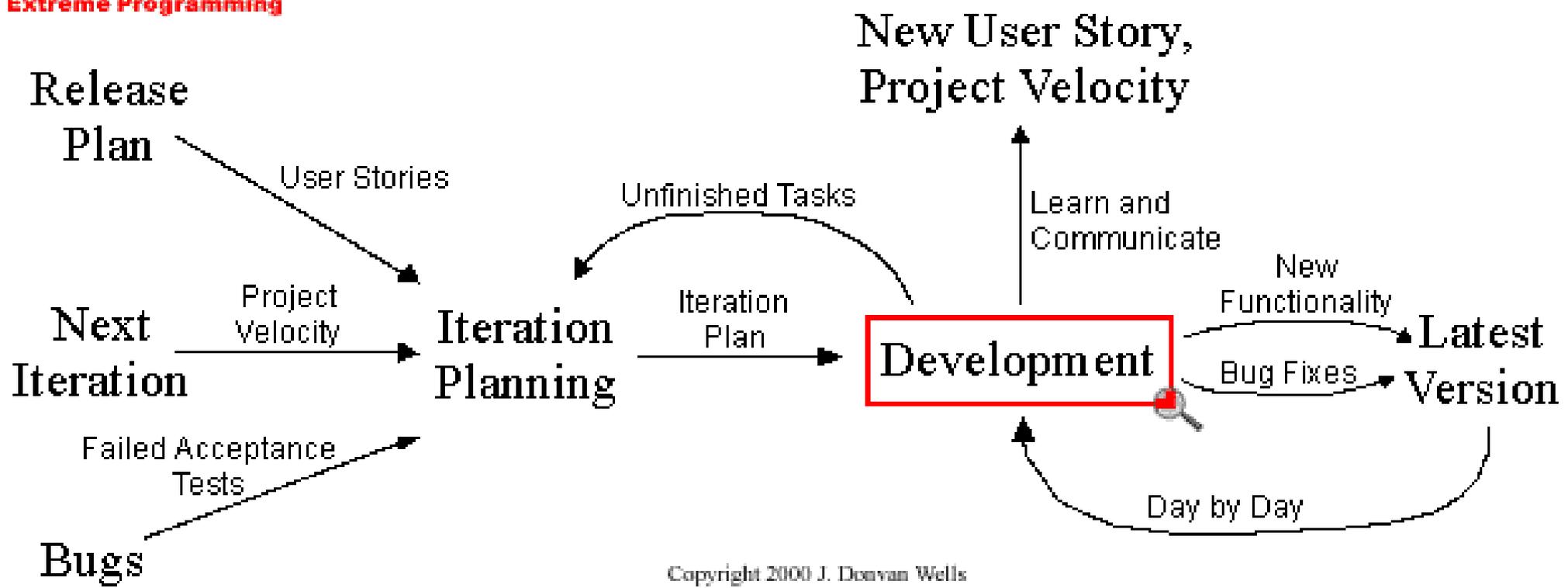


ExtremeProgramming.org

(d'après D. Wells)



Iteration



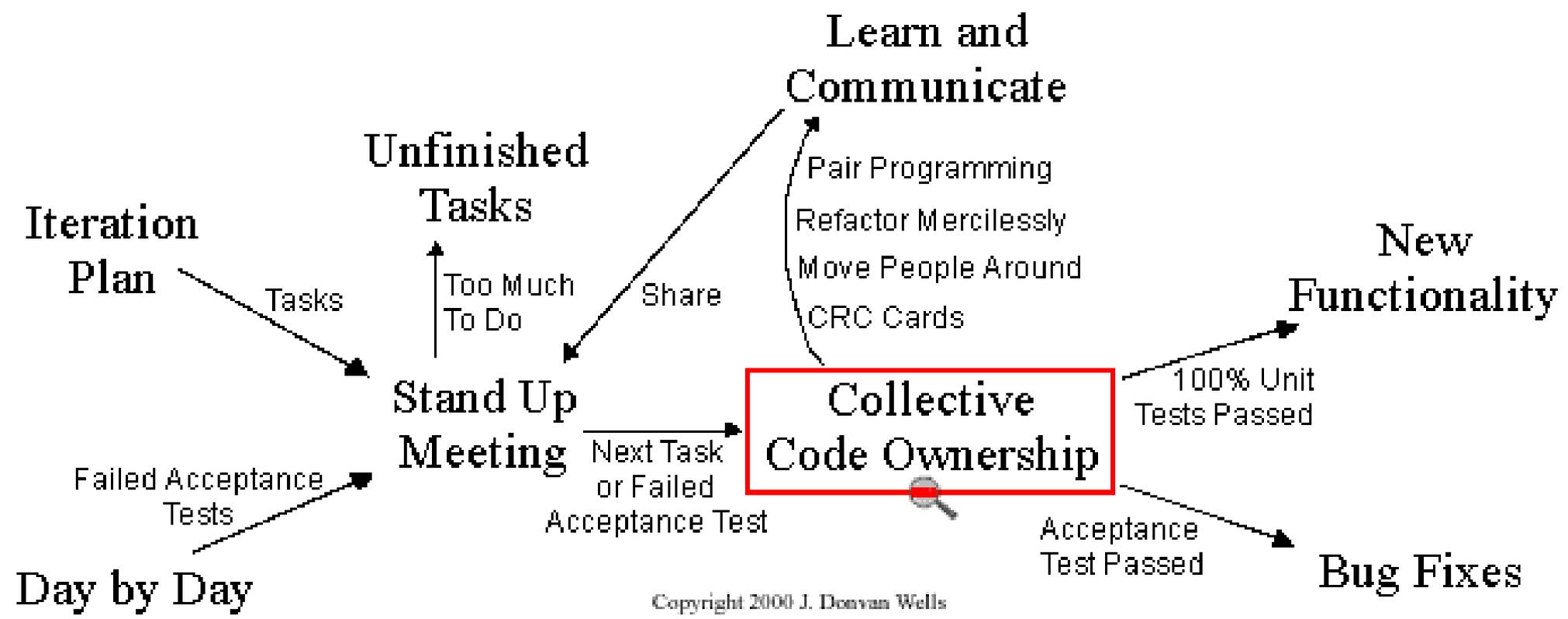
Copyright 2000 J. Donovan Wells

ExtremeProgramming.org

(d'après D. Wells)



Development

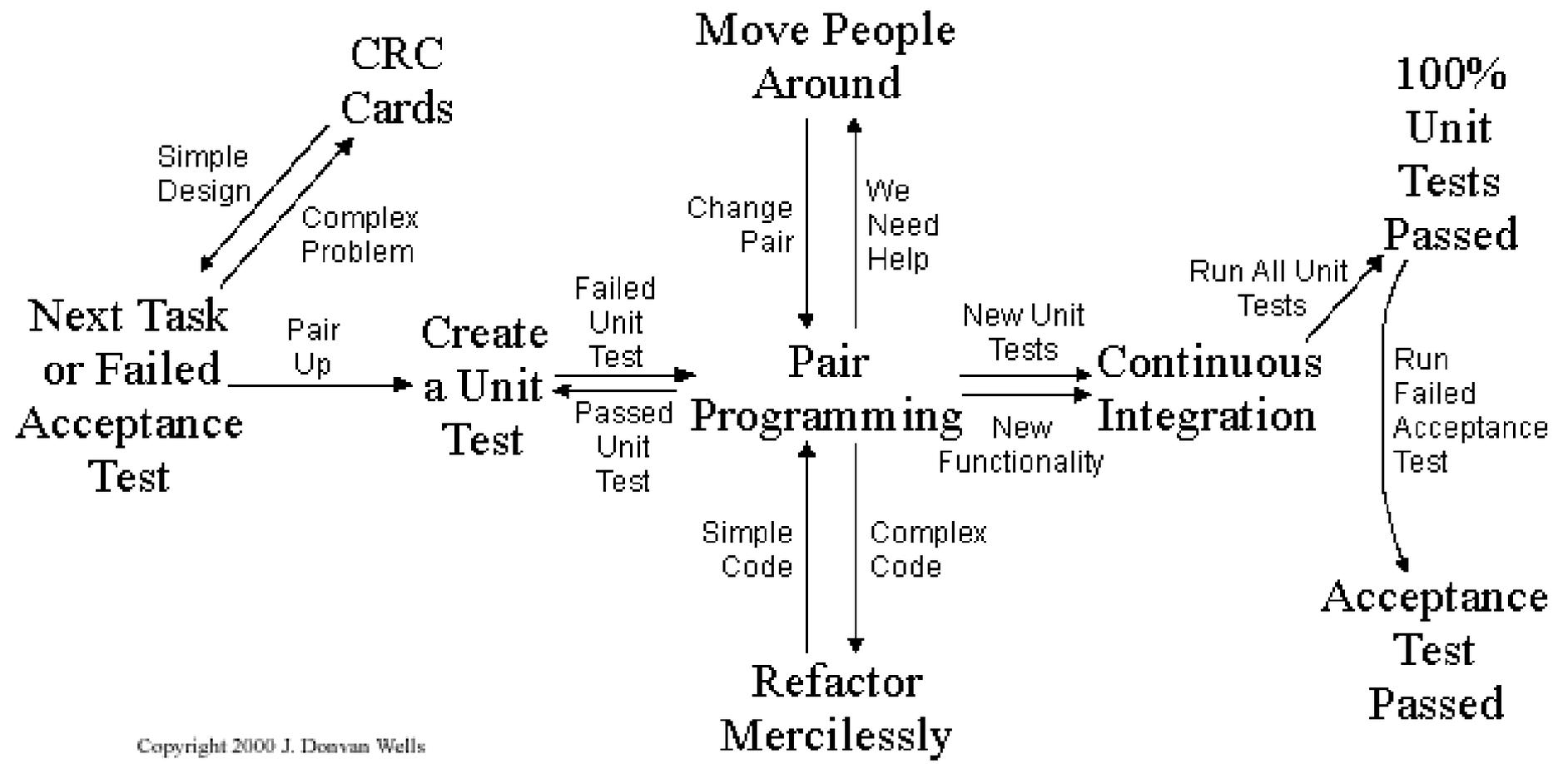


ExtremeProgramming.org

(d'après D. Wells)



Collective Code Ownership



Cycle de vie

Autres modèles :

- MERISE (1978)
- RAD (rapid application development, 90's), DSDM
 - prototypage avec le client, génération de squelettes, ...
- développement incrémental
- en fontaine (1993)
 - recouvrement des phases d'analyse et de conception
 - adapté à une approche orientée objets
- orienté réutilisation de composants
- ...

Dans la vraie vie : ☹️

Contraintes de délai (client ou time-to-market)

- documents de spécification et de conception
 - peu précis, ambigus, absents, parfois faits plus tard
- codage
 - bâclé, peu documenté, verrues, code refait (peu réutilisé), ...
- tests
 - unitaires : rares ou absents (faits « au vol » et perdus)
 - intégration : peu de tests de non-régression
- qualification
 - seule vraie épreuve (mais bien trop tard !)

Être pragmatique : 😊

- Organisation :
 - **indispensable** pour de gros projets
 - mais lourde pour de petits projets
 - ☛ suivre l'esprit plutôt que la lettre ; être souple
- Ne pas se tromper de bataille
 - aller d'abord au plus important, au plus risqué, ...
 - mais ne jamais sous-estimer l'impact à long terme

Peut-on différer ?

- Quelque chose qui n'est pas fait au moment où on devrait le faire... est rarement fait plus tard car il y a toujours autre chose à faire
 - sclérose du logiciel : plus le temps passe, plus il y a de dépendances, d'habitudes, de choses à modifier, ...
- ☛ Le faire quand il est temps
 - au vol, quand c'est chaud

Exemple :

- écrire les commentaires au cours de la programmation

Court ou long terme ?

- L'investissement sur le long terme
 - gagne très souvent (plus que ce qu'on croit)
 - sauf pour projet jetable, fonctionnalité à utilité douteuse, ...
 - ne coûte pas nécessairement beaucoup plus
 - mais est très difficile à expliquer / justifier
 - mais on n'en a pas toujours les moyens
- (N.B. « long terme » = à toutes les échelles de temps)

À retenir

- La création logicielle suit un cycle de vie
- Ne pas rester figé dans un modèle : l'/s'adapter
- Importance de la planification
 - découpage en tâches (parallèles, indépendantes)
 - ne pas oublier : tâche d'intégration
 - surtout ne pas oublier : contrôle (revues, tests, ...)
 - mais uniquement en fonction des besoins identifiés
- « Si vous ne le faites par aujourd'hui, vous ne le ferez jamais »