Cycle de vie d'un logiciel

Ilhem Boussaïd
ilhem_boussaid@yahoo.fr

Université des Sciences et de la Technologie Houari Boumediene
Licence 3 Académique
http://sites.google.com/site/ilhemboussaid

20 novembre 2010

Plan

Processus de développement : c'est quoi ?

Le développement et l'évolution d'un logiciel doit être vu comme un processus.

Processus :

*Ensemble d'**activités** coordonnées et régulées, en partie ordonnées, dont le but est de créer un produit.*

Processus de développement

L'ordre et la manière d'enchaîner les étapes d'un développement

- Un processus décrit 2 choses importantes :
  - Les activités (étapes) (**QUOI ?**)
  - L'enchainement des activités (**QUAND ?**)

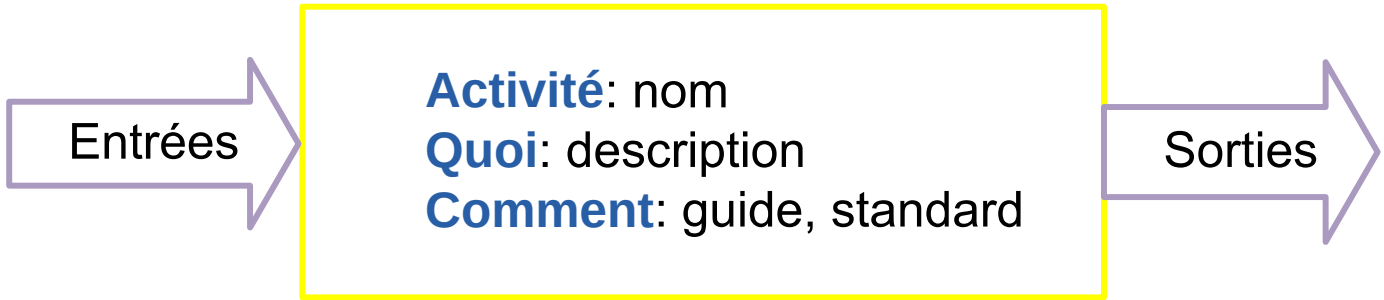
Processus de développement : Pourquoi ?

- Un logiciel à développer est un **problème très complexe à résoudre**.
- Pour appréhender la complexité de développement du logiciel, il faut arriver :
  - à rendre les choses **simples**.
  - **séparer les problèmes** qui peuvent l'être afin de les **résoudre de manière presque indépendante.** « *Diviser pour mieux régner.* »

Activité

Les activités d'un processus sont souvent décrites en termes de :

- **Entrées** de l'activité (matière première)
- **Sorties** de l'activité (résultat)
- **Intervenants** et rôles (qui ?)
- **Description** de l'activité (quoi - quel est le problème à traiter ?)
- Standards, guides, « best practices »à appliquer (comment ?)



Activités : les classiques

Les activités courantes et problèmes traités :

Spécifier *Qu'est-ce que le logiciel doit faire?* *Comment s'assurer qu'il le fait?* *Comment s'assurer qu'on développe le bon logiciel?*	**Valider** *Le logiciel fait-il ce qu'il doit faire?* *Développe t on le bon logiciel?*
Concevoir *Comment organiser le logiciel pour qu'il fasse ce qu'il doit faire?* *Quelles choix techniques faut-il faire pour que le logiciel fasse ce qu'il doit faire?* *Comment s'assurer que le logiciel est organisé et construit de manière à faire ce qu'il doit faire?*	**Intégrer** *Le logiciel est-il organisé et construit de manière à faire ce qu'il doit faire?*
Coder *Comment traduit-on cette organisation en code source?*	**Tester unitairement** *Le code source est-il bien écrit?*

Faisabilité (pourquoi ?)

Questions

- Pourquoi développer le logiciel ?
- Y a-t-il de meilleures alternatives ?
- Comment procéder pour faire ce développement ?
- Y a-t-il un marché pour le logiciel ?
- Quels moyens faut-il mettre en oeuvre ? A-t-on le budget, le personnel, le matériel nécessaires ?

Faisabilité (pourquoi ?)

Activité	**Étude préalable**
Description	Etudier la faisabilité du projet, ses contraintes techniques (coût, temps, qualité) et les alternatives possibles
Entrées	Problème à résoudre, objectifs à atteindre
Sorties	**OUI**(la décision est prise de réaliser le logiciel) ou **NON**(le projet est abandonné)

Spécification (quoi ?)

Activité	**Spécifier**
Description	Décrire ce que doit faire le logiciel (comportement en boite-noire). Décrire comment vérifier en boite-noire que le logiciel fait bien ce qui est exigé.
Entrées	Client qui a une idée de ce qu'il veut. Exigences, désir, besoins. . . concernant le système permettant de résoudre le problème.
Sorties	Cahier des charges du logiciel (ou spécification du logiciel). Des procédures de validation. Version provisoire des manuels d'utilisation et d'exploitation du logiciel

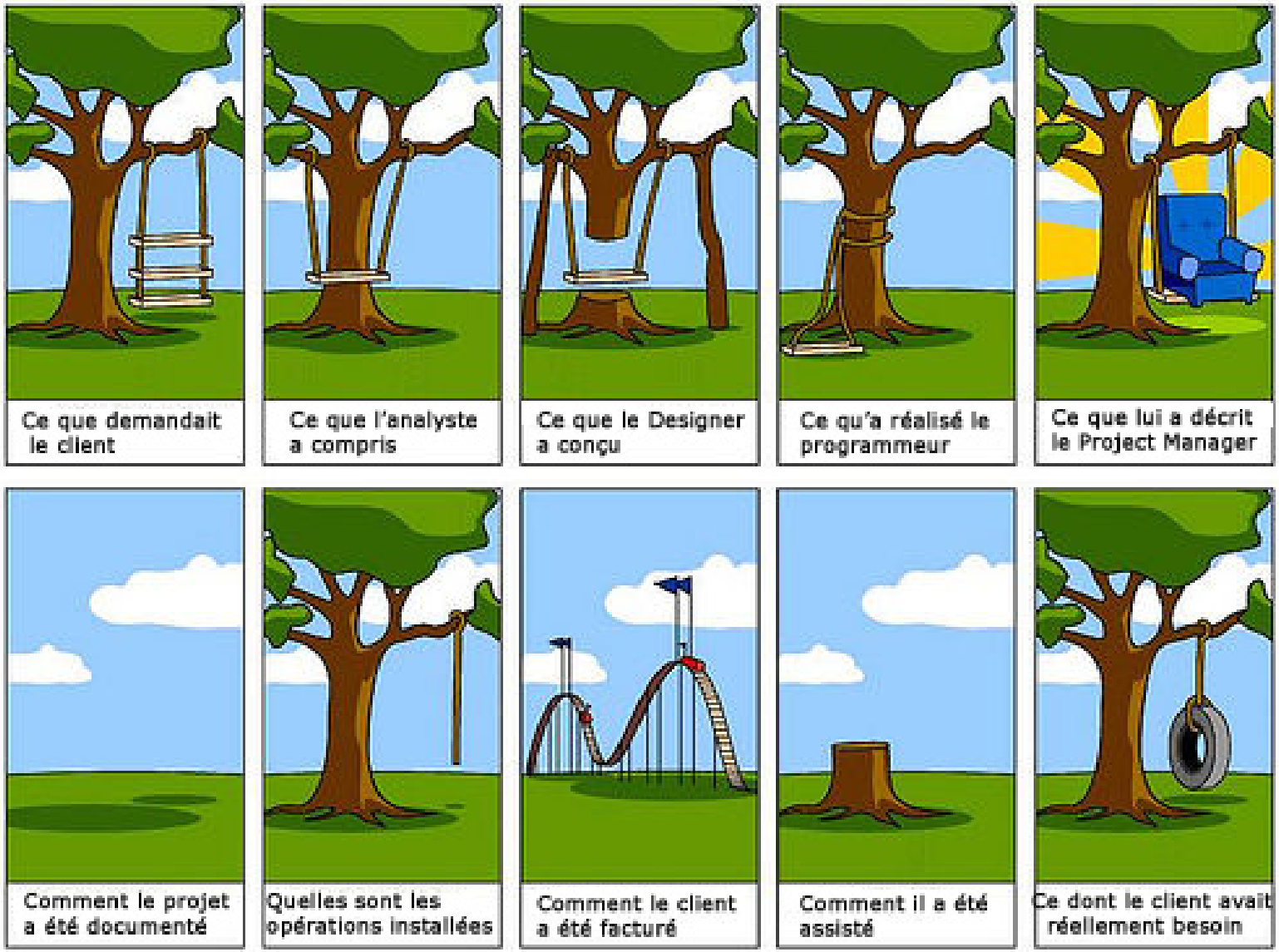
- Une spécification peut suivre de **nombreux** formalismes : cas d'utilisation, modèles UML, exigences, ...
- Les procédures de validation peuvent être des procédures manuelles ou des tests.

Spécification - Bonnes pratiques

Une bonne définition des exigences est capitale et contribue à la réussite du projet en :

- permettant d'obtenir un accord explicite entre les développeurs, les clients et les utilisateurs sur le travail à réaliser et les critères d'acceptation du système final,
- fournissant une base solide pour l'estimation des ressources nécessaires (temps, coût, équipement, qualifications des développeurs,...),
- évitant les incompréhensions et les omissions.

Spécification - Difficultés



Spécification - Difficultés

- Les besoins du client sont imprécis et changeants,
- 2 pratiques différentes du génie logiciel s'opposent sur la manière de traiter ce problème :
  - Faut-il faire un effort pour préciser et figer le besoin du client en début de projet ?
  - Faut-il développer de manière à être tolérant aux imprécisions et aux changements de besoins ?

Spécification - Difficultés

- Les erreurs dans les exigences sont plus fréquentes que celles de conception et d'implémentation.
- Elles sont également les plus difficiles et les plus coûteuses à corriger a posteriori.
  - **Il est donc important de les repérer le plus rapidement possible.**

Spécification - Cahier de charges

Plan type :

1. introduction : présentation générale, motivations, définitions des termes
2. contexte : environnement matériel et humain, acteurs et utilisateurs, interaction avec d'autres systèmes et logiciels, existant
3. spécifications fonctionnelles : grandes fonctionnalités du système, acteurs et autres systèmes qu'elles impliquent
4. spécifications non fonctionnelles, contraintes :
   - charte graphique
   - matériel : marques, RAM, débit de connexion Internet...
   - interfaçage : protocoles de communication, formats de fichiers, etc., pour l'interaction avec des matériels, logiciels, systèmes d'exploitation...
   - performances : temps réel...
   - sécurité : sauvegardes, confidentialité, ...
   - charge à supporter : volume des données, nombre d'utilisateurs simultanés, ...
   - comportement en cas de panne...
5. priorités relatives des spécifications, versions à prévoir, délais
6. évolutions à prévoir
7. annexes

(norme IEEE 830 :1993)

Spécification - Cahier de charges

Qualités requises dans un cahier de charges :

- Bon niveau de généralité
- Formulation adéquate des besoins ? Problème bien décrit
- Être précis, **non ambigu** malgré l'usage d'un langage naturel ($\neq$ mathématique)
- Être **complet** (pas d'omission involontaire)
- Être **cohérent** (pas d'inférence de fonctionnalités)
- Être **vérifiable** : critères de validation définis. Évaluer la faisabilité des besoins ? faire éventuellement une maquette, une simulation
- Être **modifiable** : facilité à exprimer un changement ou ajout de besoins

Spécification - Cahier de charges

Rédaction d'un cahier des charges :

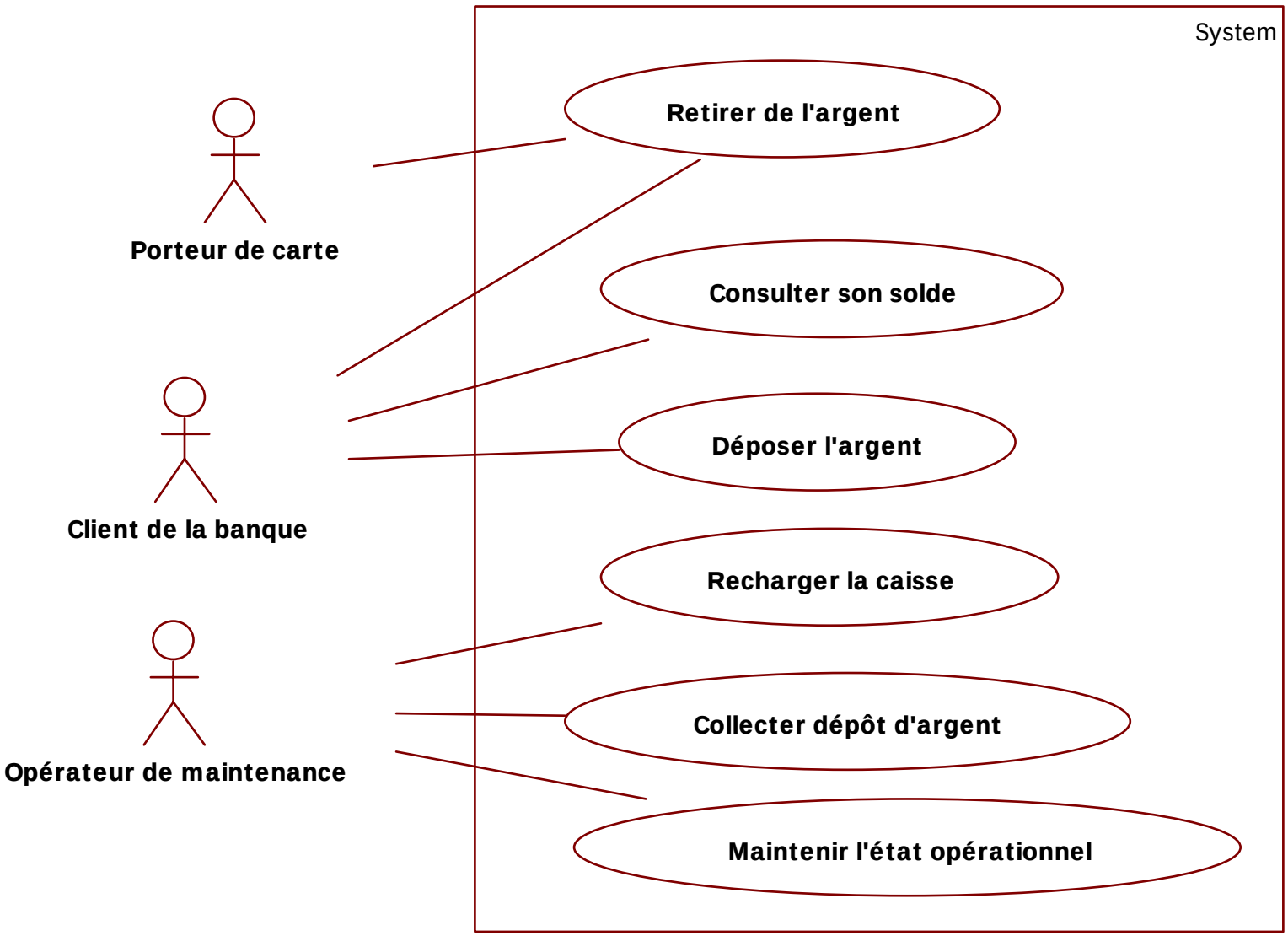
- norme IEEE 830 (1998) pour la rédaction des spécifications ;
- organisation : numérotation, tableaux, schémas, exemples. . . (ce n'est pas un roman !) ;
- diagrammes UML : en particulier des cas d'utilisation pour présenter les fonctionnalités et les acteurs qu'elles impliquent ;
- autres schémas : écrans types, diagramme de déploiement UML...
- scénarios (d'interaction) : illustration des cas d'utilisation complexes par un exemple d'interaction le réalisant ; spécifier les acteurs impliqués et l'enchaînement des interactions sur un exemple (éventuellement par un diagramme de séquence UML), ainsi que le traitement des erreurs et les éventuelles préconditions et postconditions ;

Spécification - Exemple

Un Guichet Automatique de Banque (GAB) offre les services suivants :

- Distribution d'argent à tout Porteur de carte de crédit, via un lecteur de carte et un distributeur de billets.
- Consultation de solde de compte, dépôt en numéraire et dépôt de chèques pour les clients porteurs d'une carte de crédit de la banque.
- Un opérateur de maintenance se charge de la collecte des dépôts d'argent et de la recharge du distributeur.

Spécification - Exemple



Spécification - Exemple

Besoins d'interface Homme/Machine (IHM)

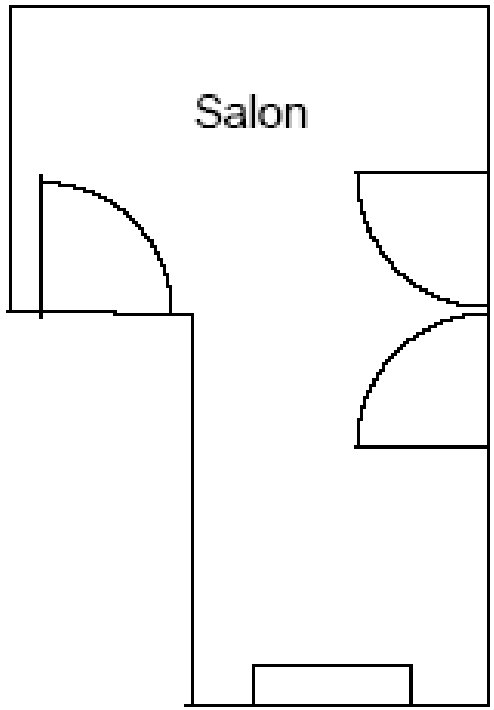
- Les dispositifs d 'entrée/sortie à la disposition du Porteur de carte doivent être :
  - Un lecteur de carte bancaire.
  - Un clavier numérique (pour saisir son code), avec des touches "validation", "correction" et "annulation".
  - Un écran pour l'affichage des messages du GAB.
  - Des touches autour de l'écran pour sélectionner un montant de retrait parmi ceux qui sont proposés.
  - Un distributeur de billets.
  - Un distributeur de tickets.

Conception (comment ?)

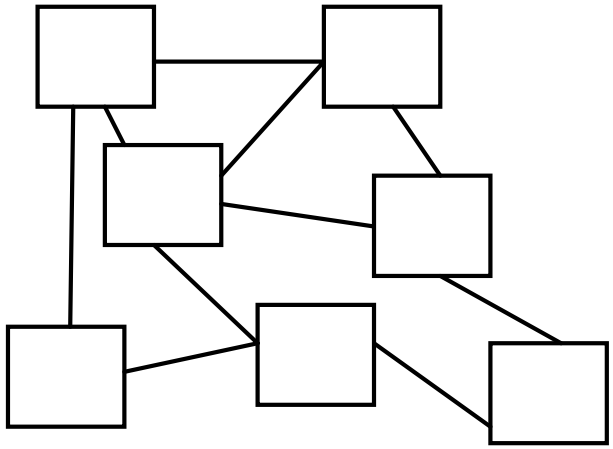
Activité	**Concevoir**
Description	**Organiser** le logiciel afin qu'il puisse satisfaire les exigences de la spécification. Faire les principaux **choix techniques** pour satisfaire les exigences de la spécification.
Entrées	Une spécification.
Sorties	Une description des décisions de conception. Des procédures de tests qui permettent de vérifier que les décisions de conception sont correctement implémentées en code source et qu'elles contribuent à satisfaire les exigences de la spécification.

Conception architecturale

- Décomposer le système en sous-systèmes
- Définir les interfaces, les liens entre les composants
- *Résultat* : Une description de l'architecture du logiciel.

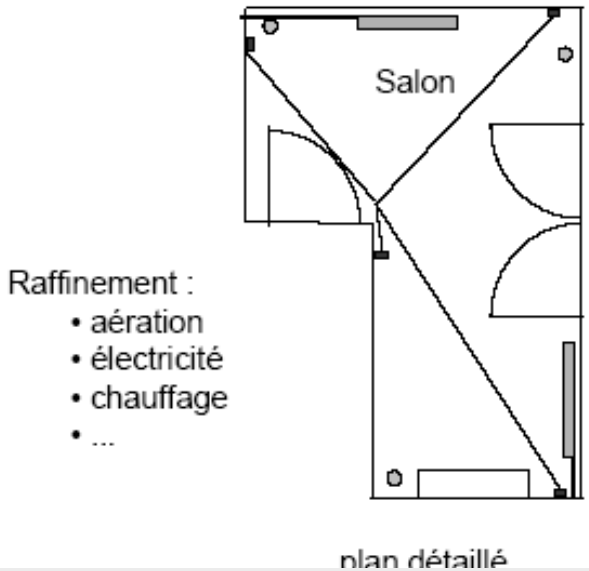


plan général



Conception détaillée

- Détailler le fonctionnement des composants
- Définir quelques algorithmes, la représentation des données, ...
- des tests unitaires sont définis pour s'assurer que les composants réalisés sont bien conformes à leurs descriptions.
- *Résultat* : pour chaque composant, le résultat consiste en
  - Un dossier de conception détaillée.
  - Un dossier de tests unitaires.
  - Un dossier de définition d'intégration logiciel.



plan détaillé

Qualité d'une conception

- La composante la plus importante de la qualité d'une conception est la **maintenabilité**.
  - maximiser la **cohésion** à l'intérieur des composants
  - minimiser le **couplage** entre ces composants
- Les dégradations de la conception sont liées aux modifications des spécifications,
- Ces modifications sont *à faire vite* et par d'autres que les concepteurs originaux
  - *ça marche* mais sans respect de la conception initiale
  - corruption de la conception (avec effet amplifié au fur et à mesure)

Couplage/Cohésion

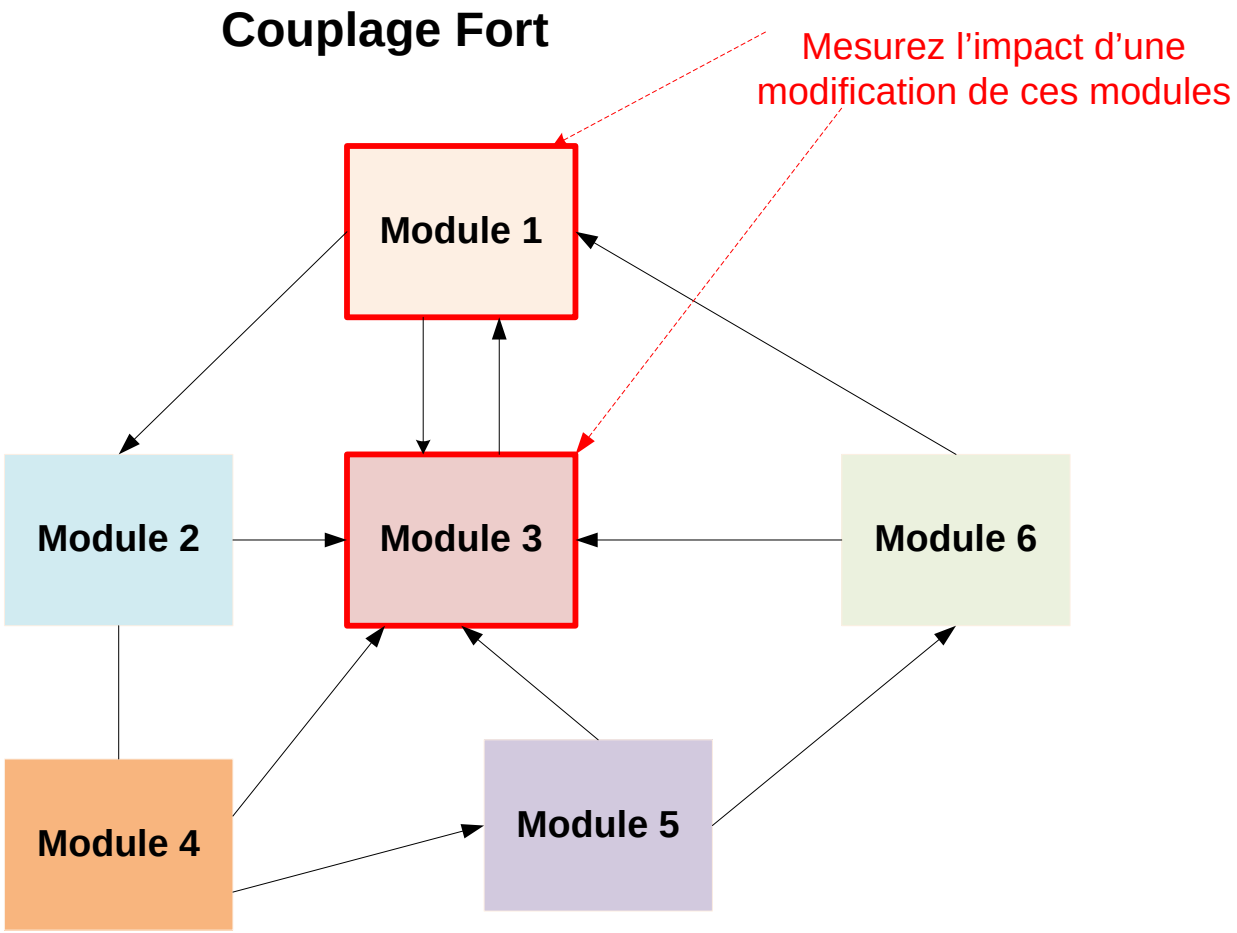
A RETENIR ABSOLUMENT

- Pour qu'un logiciel soit **extensible** et **réutilisable**, il faut qu'il soit découpé en modules
  - faiblement couplés et
  - à forte cohésion.

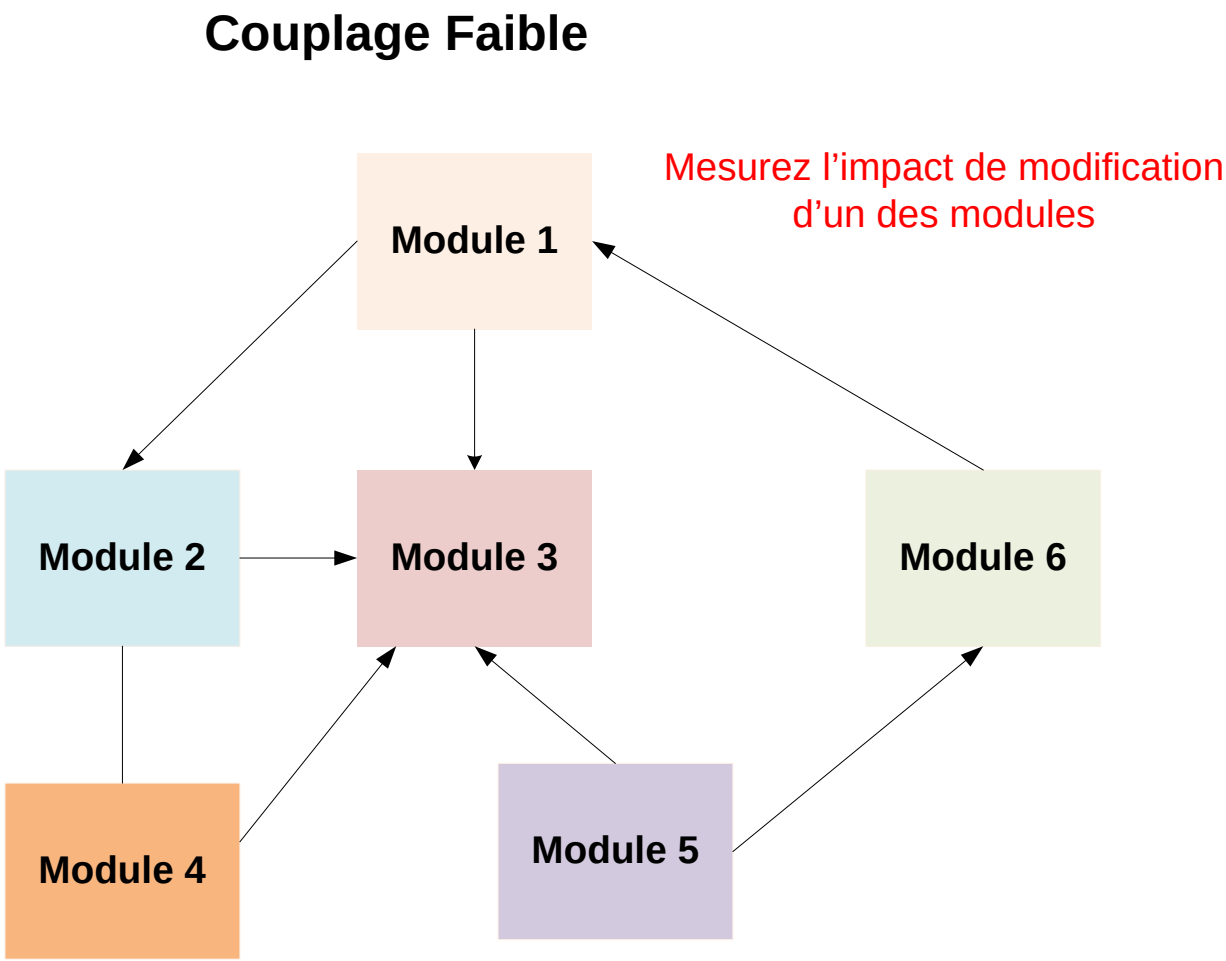
Couplage

- **Couplage**
  - *Une entité (fonction, module, classe, package, composant) est couplée à une autre si elle **dépend** d'elle.*
- **Couplage faible**
  - *Désigne une relation faible entre plusieurs entités (classes, composants), permettant une grande souplesse de programmation, de mise à jour.*
  - Ainsi, chaque entité peut être modifiée en **limitant l'impact du changement** au reste de l'application.
- **Couplage fort**
  - *au contraire, tisse un lien puissant qui rend l'application plus rigide à toute modification de code.*

Couplage fort



Couplage faible



Forte cohésion

- Cohésion entre modules dans un composant, entre services dans un module :"***qui se ressemble s'assemble*** "
- L'idée est de vérifier que nous rassemblons bien dans une classe des méthodes cohérentes, qui visent à réaliser des objectifs similaires.

Cohésion - Mauvais exemple

- **Mauvais exemple** : il serait mal venu d'implémenter une méthode "Afficher()" ou "Tracer()" puisque l'objectif de cette classe est de représenter le modèle d'un compte en banque et non celui d'une fenêtre graphique ou d'un service de journalisation.

CompteEnBanque
-solde : double
+debiter(entrée montant : double) +crediter(entrée montant : double) +afficher(entrée message : string) +alerter(entrée message : string) +tracer(entrée message : string)

Implantation (comment ?)

Activité	**Coder et tester**
Description	**Écrire le code** source du logiciel. **Tester** le comportement du code source afin de vérifier qu'il réalise les responsabilités qui lui sont allouées.
Entrées	Spécification, conception.
Sorties	Code source. Tests unitaires. Documentation

Intégration

Activité	**Intégrer**
Description	Assembler le code source du logiciel (éventuellement partiellement) et dérouler les tests d'intégration.
Entrées	Conception, code source, tests d'intégration (tests).
Sorties	Un rapport de test d'intégration.

Validation

Activité	**Valider**
Description	Construire le logiciel complet exécutable. Dérouler les tests de validation sur le logiciel complet exécutable.
Entrées	Logiciel complet exécutable à valider. Tests de validation.
Sorties	Rapport de tests de validation.

- **Qualification** (ou **recette**), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.

Maintenance

Activités :

- maintenance **corrective** : correction des bugs
- maintenance **adaptative** : ajuster le logiciel
- Maintenance **perfective**, d'extension : augmenter/améliorer les possibilités du logiciel

Productions :

- logiciel corrigé
- mises à jour
- documents corrigés

Maintenance corrective

- Corriger les erreurs : défauts d'utilité, d'utilisabilité, de fiabilité...
  - Identifier la défaillance, le fonctionnement
  - Localiser la partie du code responsable
- Attention
  - La plupart des corrections introduisent de nouvelles erreurs
  - Les coûts de correction augmentent exponentiellement avec le délai de détection
  - Corriger et estimer l'impact d'une modification
- La maintenance corrective donne lieu à de nouvelles livraisons (release)

Maintenance adaptative

Maintenance adaptative

- Ajuster le logiciel pour qu'il continue à remplir son rôle compte tenu du l'évolution des
  - Environnements d'exécution
  - Fonctions à satisfaire
  - Conditions d'utilisation
- Ex : changement de SGBD, de machine, de taux de TVA, an 2000, euro...

Maintenance perfective

Maintenance perfective, d'extension

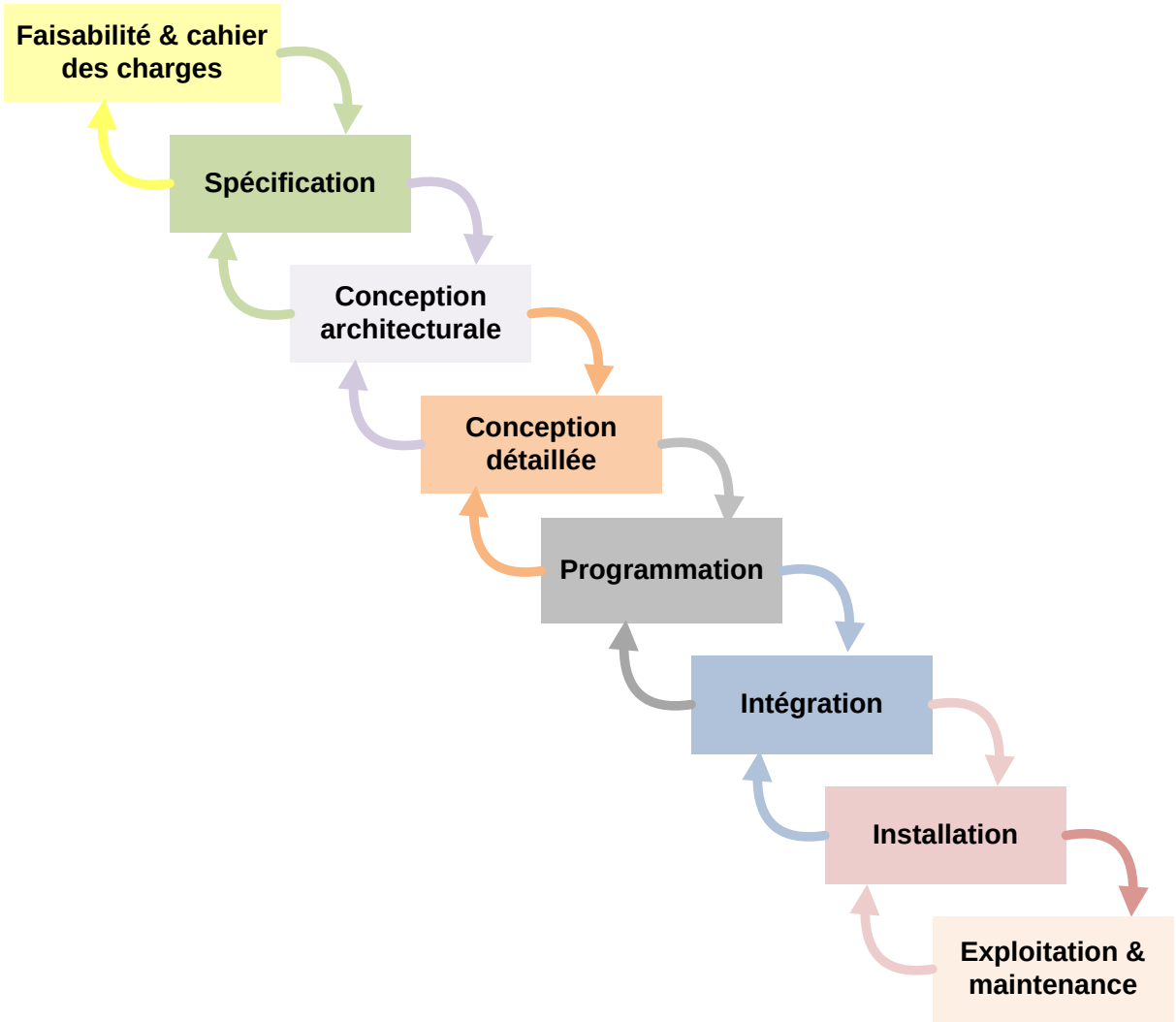
- Accroître/améliorer les possibilités du logiciel
- Ex : les services offerts, l'interface utilisateur, les performances...
- Donne lieu à de nouvelles versions

Cycle de vie en cascade

Le modèle en cascade (Waterfall model) est le plus classique des cycles de vie.

- Cycle de vie linéaire sans aucune évaluation entre le début du projet et la validation
- Le projet est découpé en phases successives dans le temps
- A chaque phase correspond une activité principale bien précise produisant un certain nombre de livrables
- Chaque phase ne peut remettre en cause que la phase précédente

Cycle de vie en cascade

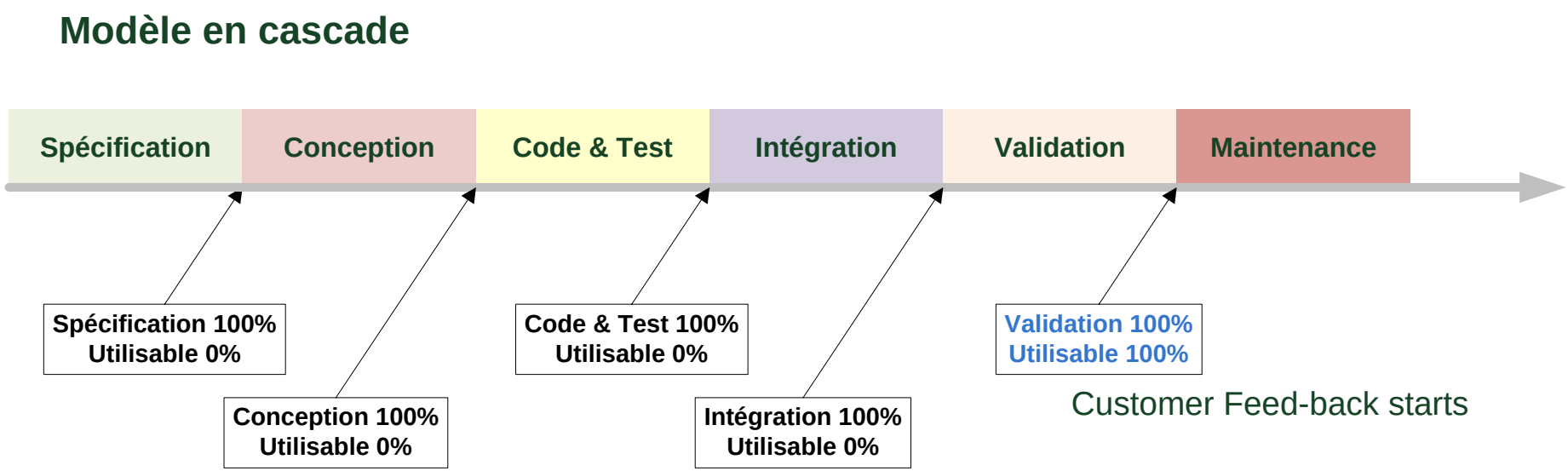


Cycle de vie en cascade - Inconvénients

Inconvénients

- Les vrais projets suivent rarement un développement séquentiel
- Établir tous les besoins au début d'un projet est difficile
- **Aucune validation intermédiaire** (Aucune préparation des phases de vérification)
  - Obligation de définir la totalité des besoins au départ
  - augmentation des risques car validation tardive : remise en question **coûteuse** des phases précédentes
- Sensibilité à l'arrivée de nouvelles exigences : refaire toutes les étapes
- Bien adapté lorsque les besoins sont **clairement identifiés** et **stables**

Cycle de vie en cascade - Inconvénients

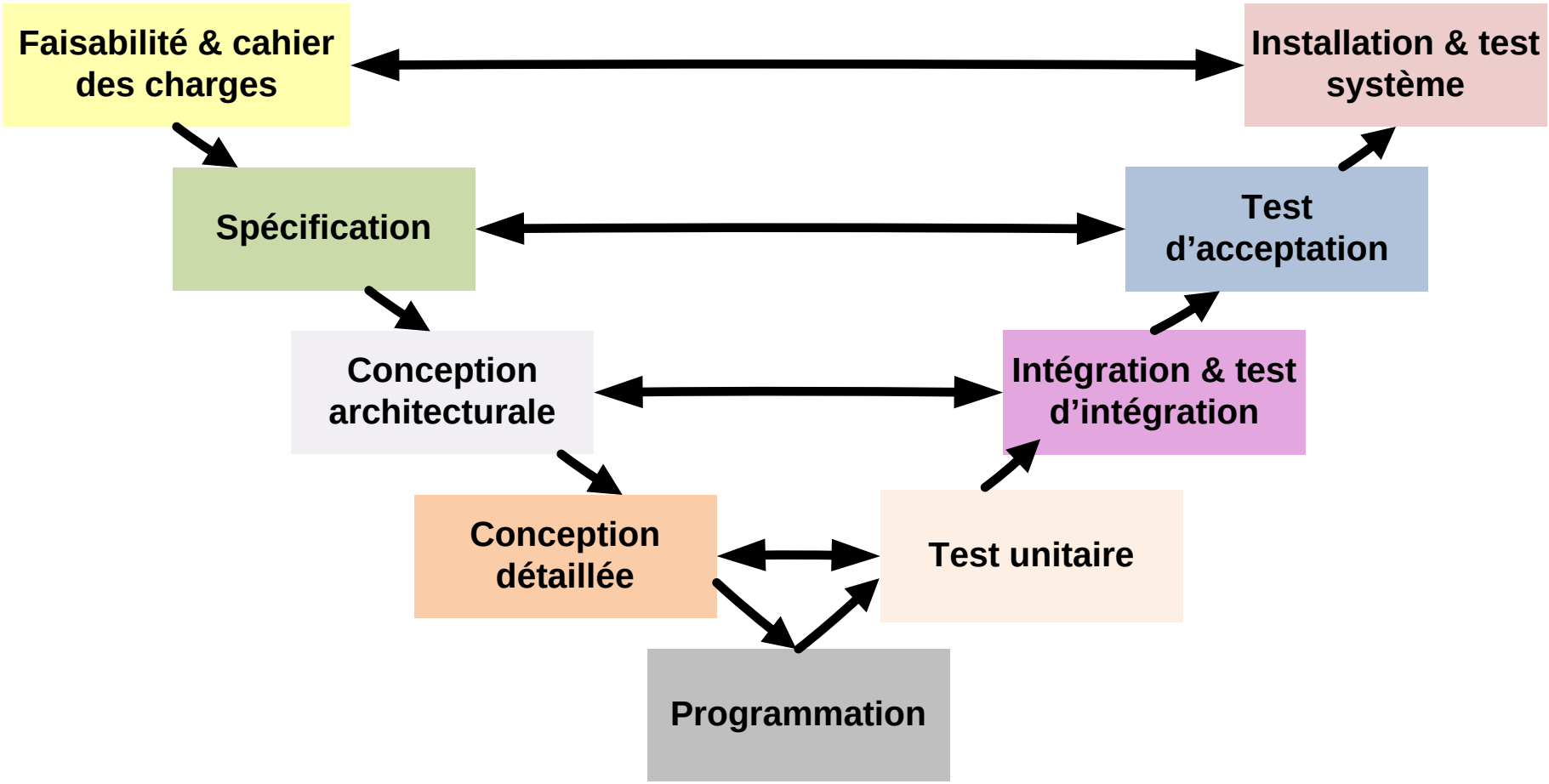


Cycle de vie en V

A ce jour, le cycle en V reste le cycle de vie **le plus utilisé**. C'est un cycle de vie **orienté test** :

- A chaque **activité créative** (spécification, conception et codage) correspond une **activité de vérification** (validation, intégration, tests unitaires).
- Chaque phase en amont prépare la phase correspondante de vérification (la vérification est prise en compte au moment même de la création).

Cycle de vie en V



Avantages et inconvénients

Avantages

- La préparation des dernières phases (validation-vérification) par les premières (construction du logiciel), permet d'éviter d'énoncer une propriété qu'il est impossible de vérifier objectivement après la réalisation.

Inconvénients

- Construit-on le bon logiciel ? Le logiciel est **utilisé très (trop) tard**.
  - Il faut attendre longtemps pour savoir si on a construit le bon logiciel.
  - Difficile d'impliquer les utilisateurs lorsque qu'un logiciel utilisable n'est disponible qu'à la dernière phase
- **Idéal** quand les besoins sont bien connus, quand **l'analyse et la conception sont claires**

Documents et phases

phases / documents	avant-projet	analyse	conception générale	conception détaillée	implémentation	tests unitaires	intégration et test d'intégration	installation et test de réception
cahier des charges du projet	■	→						
spécification		■	→					→
conception générale			■	→			→	
conception détaillée				■	→	→		
listages					■	→		
tests unitaires				▒	?	■		
test d'intégration			▒	?			■	
test de réception		▒	?					■
manuels d'utilisation et d'exploitation		▒	→ ?				■	→ ?

■ document élaboré entièrement pendant la phase

▒ document partiellement élaboré pendant la phase

→ document en entrée de la phase

? document éventuellement complété pendant la phase

Prototypage

- Construit et utilisé en phase d'analyse (spécification)
  - Discuter et interagir avec l'utilisateur - Pour montrer aux clients les fonctions que l'on veut mettre en oeuvre
  - identifier les besoins : **Je saurai ce que je veux quand je le verrai !**
  - Vérifier des choix spécifiques d'IHM
- Construit et utilisé en phase de conception
  - s'assurer de la faisabilité de parties critiques
  - valider des options de conception
  - Vérifier l'efficacité réelle d'un algorithme

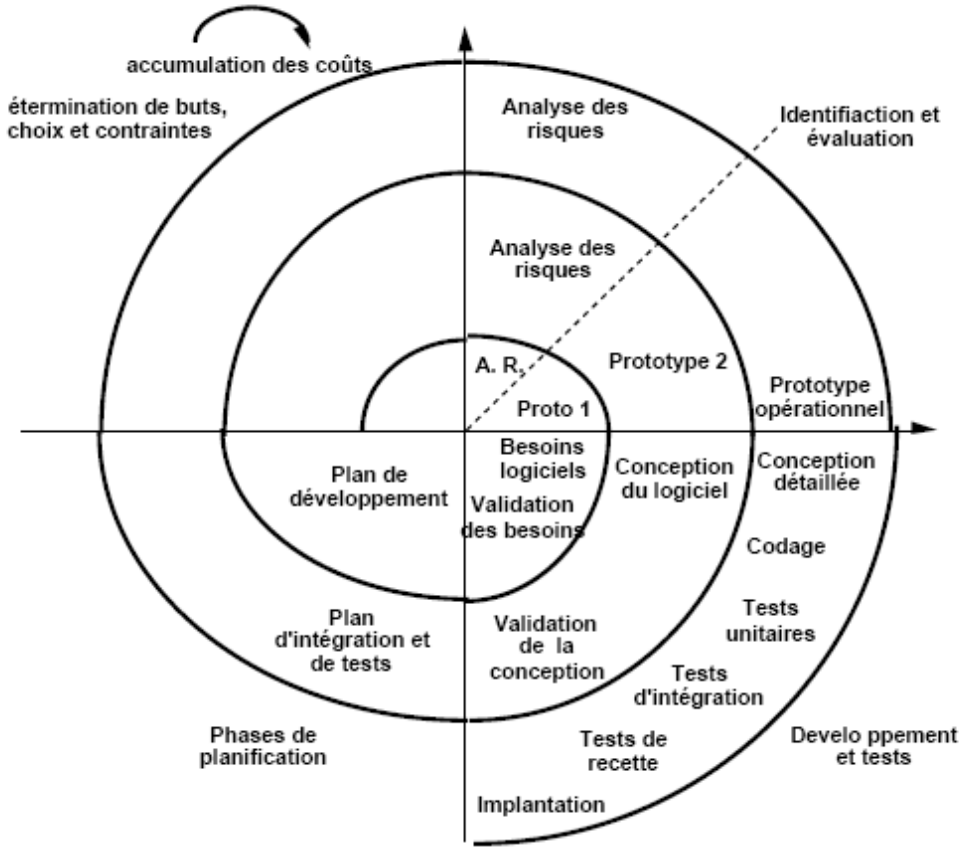
Prototypage évolutif/jetable

- (**Prototype jetable**) : squelette du logiciel qui n'est crée que dans un but et dans une phase particulière du développement
- Si gardé, alors s'appelle (**prototype évolutif**)
  - Première version du prototype = embryon du produit final
  - On itère jusqu'au produit final

Prototypage : Avantages/Inconvénients

- **Avantage :**
  - Les efforts consacrés au développement d'un prototype sont le plus souvent compensés par ceux gagnés à ne pas développer de fonctions inutiles
- **Mais :**
  - Les décisions rapides sont rarement de bonnes décisions
  - Le prototype évolutif donne-t-il le produit demandé ?

Cycle de vie en spirale



Cycle de vie en spirale

Chaque cycle de la spirale se déroule en quatre phases :

1. détermination, à partir des résultats des cycles précédents, ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
2. analyse des risques, évaluation des alternatives et, éventuellement maquettage ;
3. développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
4. revue des résultats et vérification du cycle suivant.

Spirale : Analyse des risques

Risques technologiques

- exigences démesurées par rapport à la technologie
- l'incomprehension des fondements de la technologie
- changement de technologie en cours de route
- ...

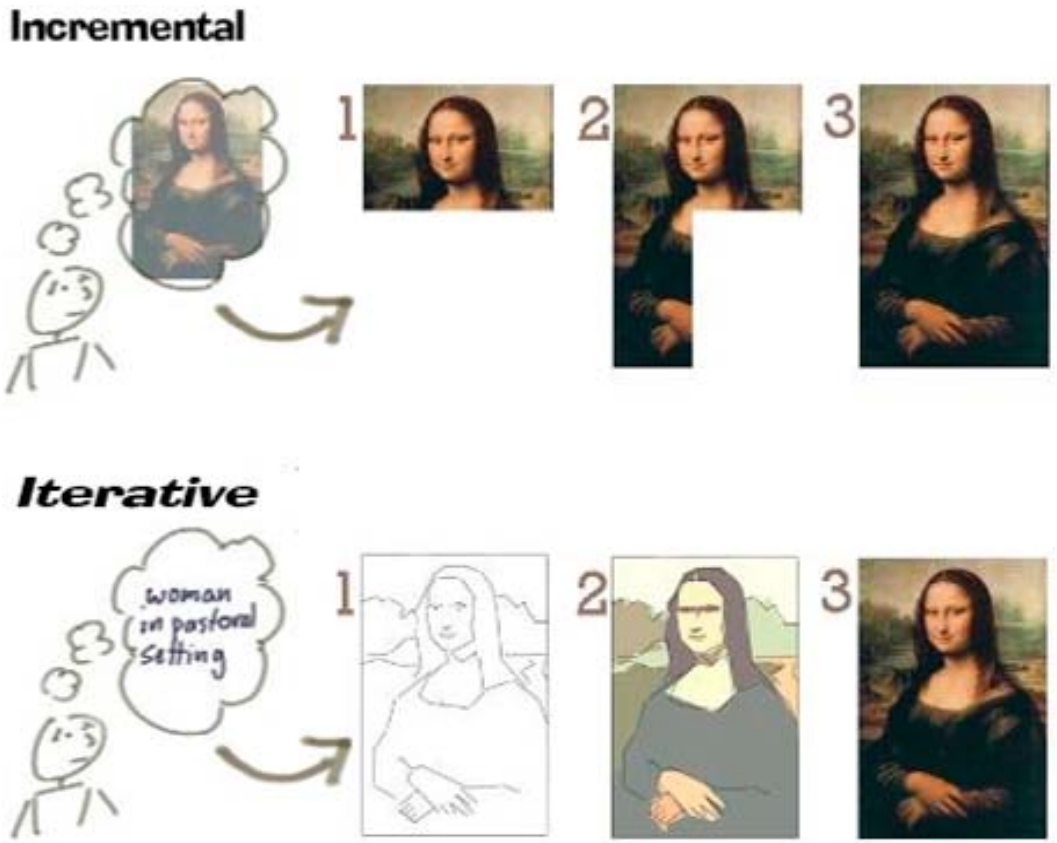
Risques liés au processus

- gestion projet mauvaise ou absente
- calendrier et budget irréalistes
- calendrier abandonné sous la pression des clients
- développement de fonctions inappropriées
- ...

Risques humains

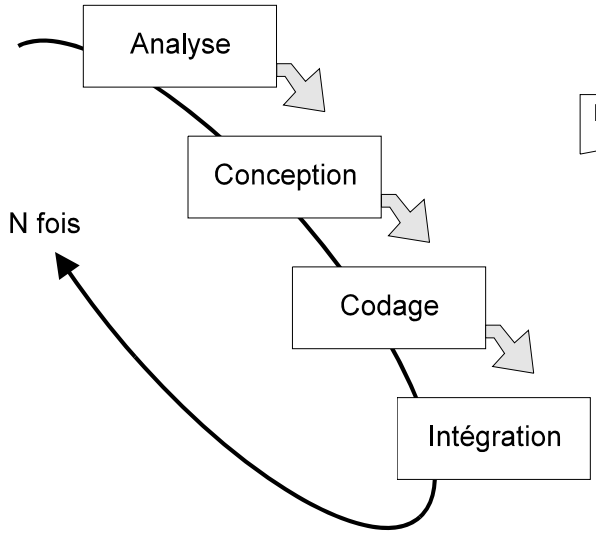
- défaillance du personnel
- surestimation des compétences
- travailleur solitaire
- manque de motivation

Cycle itératif et incrémental



Cycle itératif et incrémental

- Segmentation du travail
- Concentration sur les besoins et les risques
- Les itérations sont des prototypes
  - Expérimentation et validation des technologies
  - Planification et évaluation
- Les prototypes « s'enroulent » autour du noyau de l'architecture



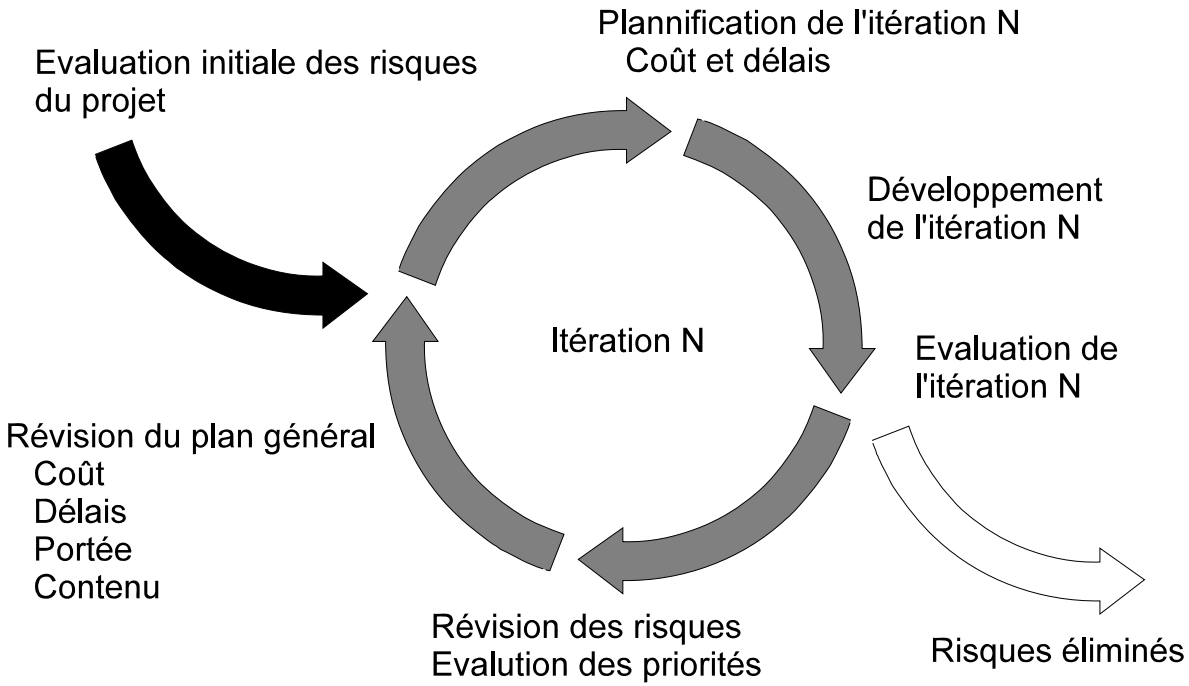
Principes

- Développez de petits incréments fonctionnels livrés en courtes itérations
  " ***Un tiens vaut mieux que deux tu l'auras***. "
- Un incrément fonctionnel est un besoin du client.
- Pour chaque version à développer après la 1ère version livrée, il faut arbitrer entre :
  - les demandes de **correction**,
  - les demandes de **modification** et
  - les **nouvelles** fonctionnalités à développer.

Dans quel ordre faut-il réaliser les itérations ?

- **Pilotez le développement par les priorités.**
  - Commencez par développer les fonctionnalités les plus importantes pour le client et les plus risquées.
- **Pilotez le développement par les tests d'acceptation.**
  - Écrivez des tests automatisés d'acceptation.
  - Utilisez ces tests pour mesurer l'avancement du développement
  - Il n'y a pas de meilleure mesure de l'avancement que les fonctionnalités s'**exécutant** conformément aux besoins du client.
  - Une fonctionnalité s'exécute conformément au besoin du client si elle réussit ses tests d'acceptation.

Pilotage par les risques



Avantages

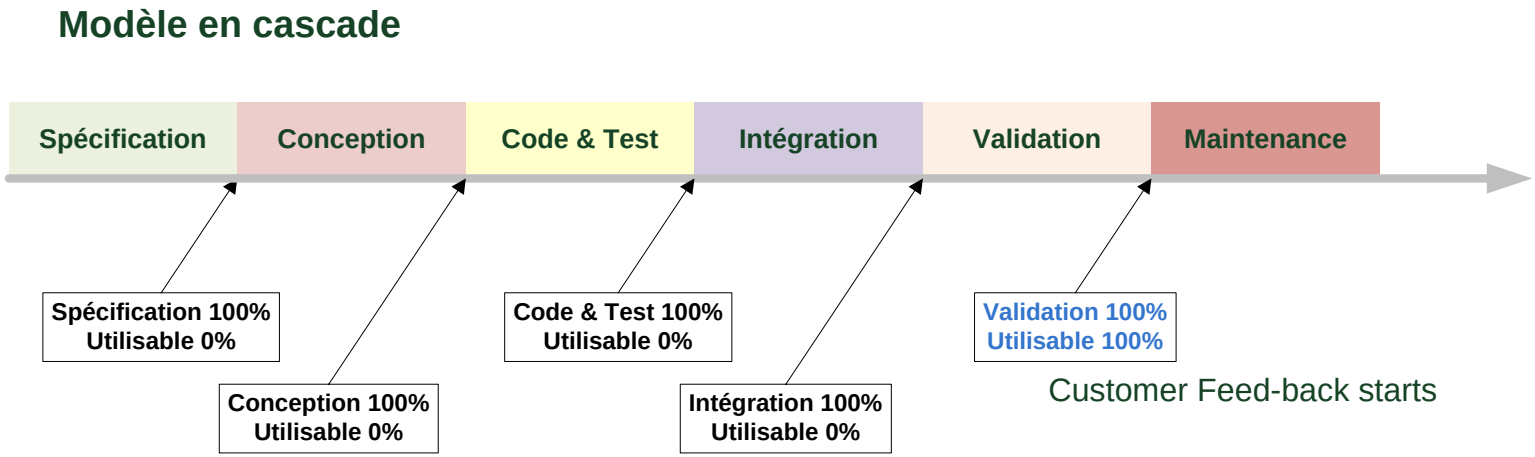
Le cycle de vie itératif

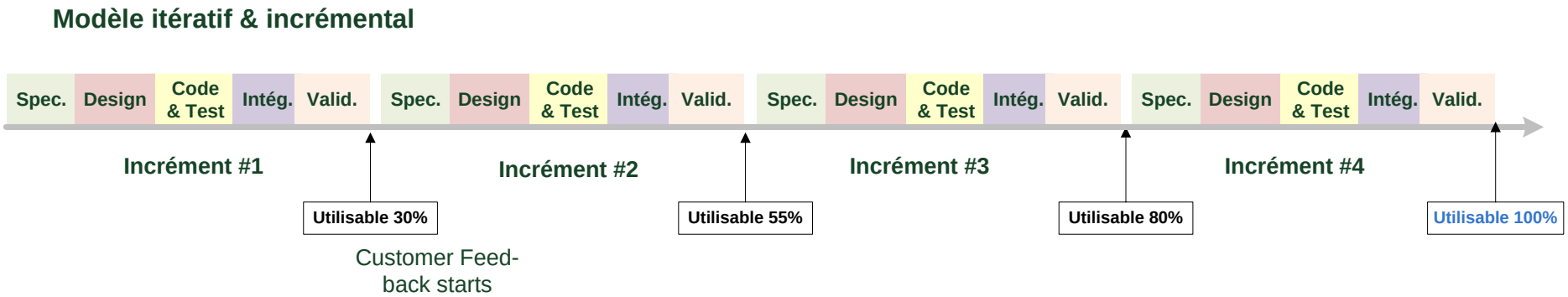
- est en phase avec la réalité
- permet la prise en compte de l'évolution
- demande un pilotage continu
- bien adapté à l'approche objet (et inversement)
- Les choix techniques (spécification, conception, codage) sont **validés très tôt par test**.
  - L'enchainement des itérations permet de régulièrement vérifier que l'on construit bien le logiciel.
- Le logiciel est **utilisé très tôt**.
  - L'enchainement des itérations permet aux utilisateurs de vérifier régulièrement que l'on construit le **bon** logiciel.

Principaux risques récurrents

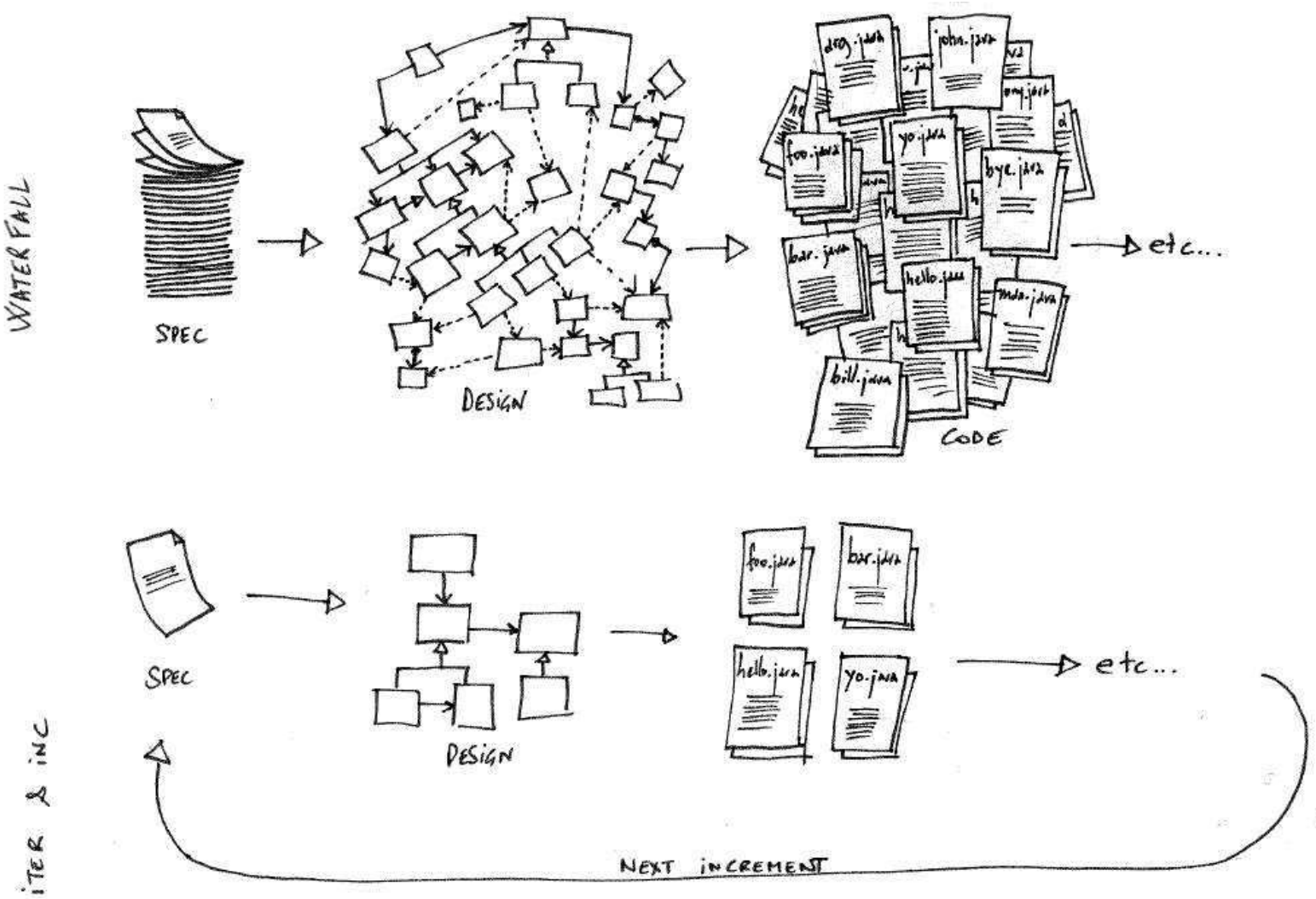
- Intégration trop complexe
- Environnement non adapté
- Utilisateurs défavorables
- Technologie complexe
- Lourdeur des activités manuelles
- Composants réutilisables inadaptés

Cycle de vie en cascade vs. Itératif

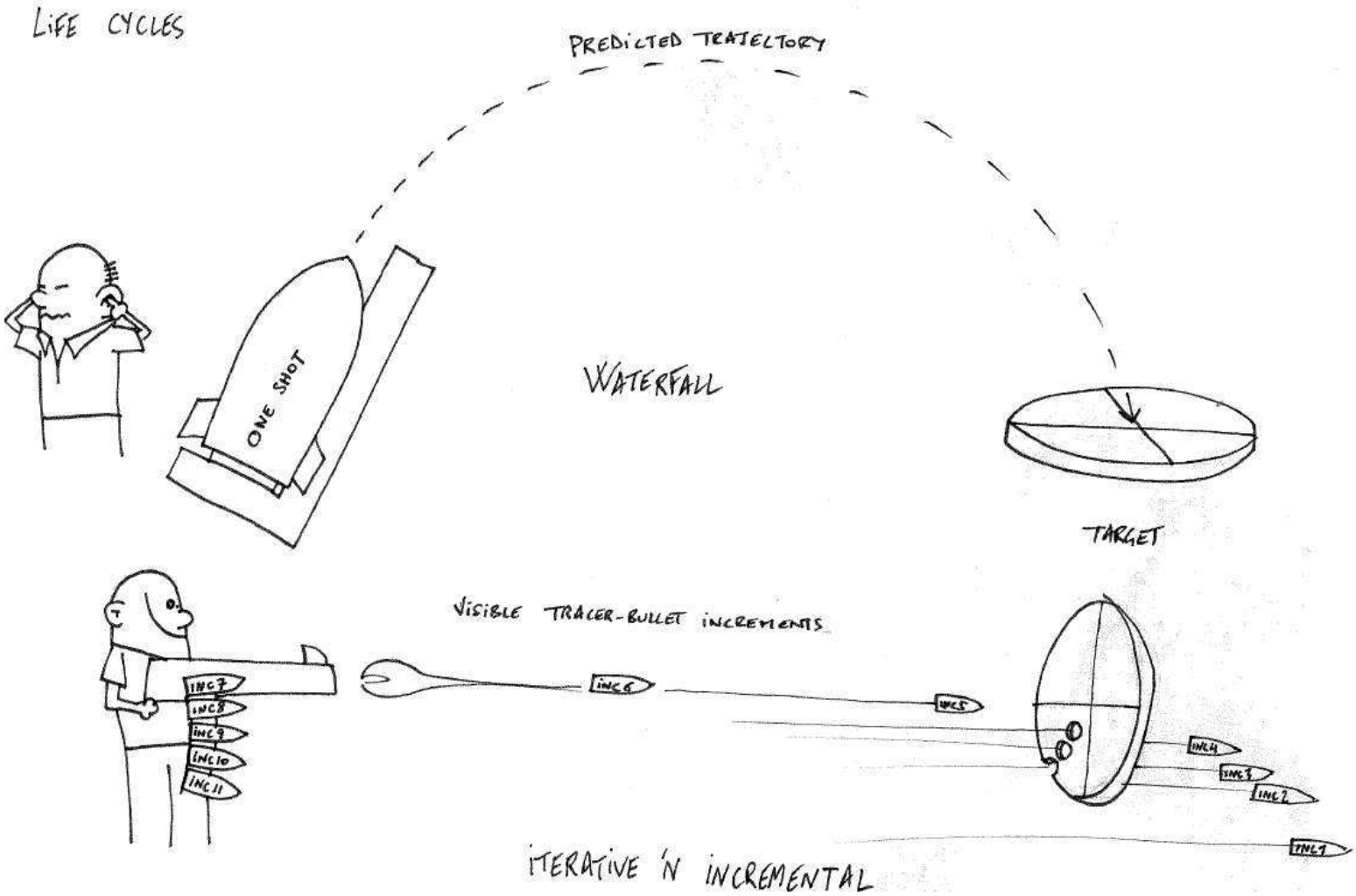




Cycle de vie en cascade vs. Itératif



Cycle de vie en cascade vs. Itératif



- Chenu, Emmanuel *Cours - Génie Logiciel orienté Objet*, ESISAR
- Chenu, Emmanuel *Cours - Génie Logiciel pragmatique*, ESISAR
- Marie-Claude Gaudel *Précis de génie logiciel* Editions Dunod
- Gérard, Pierre *Génie Logiciel - Principes et Techniques*,
- Bertrand Meyer *Conception et programmation orientées objet* Editions Eyrolles, 2000
- Pierre-Alain Muller *Démarche itérative et incrémentale*
- I. Sommerville et Franck Vallée *Software Engineering* 6th edition, Addison-Wesley, 2001
- Strohmeier, Alfred *Cycle de vie du Logiciel*, Laboratoire de Génie Logiciel - Département d'Informatique. Ecole Polytechnique Fédérale de Lausanne