Chapitre1 : Introduction

La gestion et le contrôle des **coûts** constituent souvent la plus grande contrainte de l'ingénierie logicielle : dans un système, le logiciel est souvent plus coûteux que le matériel. De plus, le maintien du logiciel est souvent plus coûteux que son développement.

Petit Projet	Grand projet
•Cahier des charges court + facile •coûts faible + calcul •conception mentalement •communication •développeur a le choix techno •pas de planif	•Cahier charges volumineux •Conception casse tête •Coût + choix difficile à évaluer •Délais diff. à évaluer et respecter •Documentation indispensable

Loi de **Brooks** : ajouter de la main d'œuvre à un projet logiciel n'est pas sans risque, cela prend du temps aux gens pour devenir productifs et les frais de communication ont tendance à augmenter à mesure que le nombre de personnes augmente.

Utilisateurs	Développeurs	Managers
logiciels ne correspondent pas aux besoins, changements de besoin difficile à intégrer dans le développement.	maintenance trop complexe et chère, trop d'erreurs, performance inacceptable, logiciels peu portables	système difficilement réutilisable, coûts imprévisibles et excessifs, délais dépassés

La **complexité** est tout ce qui rend un logiciel difficile à comprendre. Elle peut prendre plusieurs formes et on la combat de deux manières :

1. En rendant le code plus simple et évident, par exemple en éliminant des cas particuliers ou en utilisant des identifiants consistants
2. En l'encapsulant de manière que les programmeurs puissent travailler sur le système sans être exposé à sa complexité totale

QualitéInterne	Qualité Externes
maintenabilité (corrective, adaptative et perfective), réparabilité, évolutivité, réutilisation, portabilité et interopérabilité	correction, fiabilité, robustesse, performance et ergonomie

Chapitre2 : Le processus logiciel

Un **processus logiciel** est une décomposition en activités structurées qui mènent à la production d'un logiciel.

Spécification : formulation précise des besoins et des exigences (Elicitation du besoin, spécification et enfin validation du besoin)

Conception : Description de la structure du système, souvent entrelacée avec l'implémentation. Il y a architectural design, database design, interface design, component selection and design.

Implémentation : programmation *tactique* (aller le plus vite possible) vs programmation *stratégique* (penser au long terme)

Vérification et validation (V & V) : tests de composants (tests unitaires, tests d'intégration), tests de système et tests utilisateurs.

Evolution : Contrairement au hardware, le software est flexible et peut changer après avoir été développé. On distingue donc le développement de la maintenance, mais cette distinction tend aujourd'hui à disparaître.

Modèle Cascade (planifiés) : waterfall, Cycle en V, Cycle en spirale	Modèle Incrémental (agile): Rapid application development, Rational Unified Process, Extreme programming
Le processus est piloté et le progrès est mesuré à l'aide d'une planification faite à l'avance. Les activités sont considérées comme des phases distinctes et indépendantes et chaque phase doit être complétée pour passer à la suivante. Il n'est pas prévu d'adapter le processus en cours d'exécution. C'est un modèle approprié pour les grands projets d'ingénierie (lorsque les besoins sont clairs et ne changent pas, le système est développé par plusieurs équipes sur différents sites ou le projet requiert une bonne coordination du travail par exemple) mais pas lorsque les besoins du client évoluent (manque de flexibilité en cas de changement)	Le processus est semi-piloté et la planification incrémentale. Le processus est adapté aux besoins et contraintes du client. Les activités sont entrelacées plutôt que séparées et les échanges entre activités sont permis. Le logiciel est développé en une série de versions spécifiées et validées avec toutes les parties prenantes et l'automatisation joue un rôle clé (tests unitaires, intégration et livraison continue, …). **Avantages** : le client teste et valide des versions intermédiaires (applications mobiles), les couts d'adaptation aux besoins du client sont réduits et le client peut tirer profit du logiciel avant la fin du projet (applications web). **Inconvénients** : bonne communication entre les parties impliquées requise, avancement du projet dur à mesurer, dégradation de l'architecture avec les incréments et changements de plus en plus difficiles et couteux.

Modèle d'intégration et de configuration, les activités consistent à réutiliser et configurer des composants existants. Utilisé dans la majorité des projets qui adaptent les composants aux besoins. Livraison et déploiement rapide, réduction des coûts et des risques, mais une perte de contrôle de l'élément réutilisé et on doit trouver un compromis entre les besoins formulés et la fonctionnalité proposée.

Un processus peut *anticiper* le **changement** et l'éviter (prototype), ou alors le *tolérer* et s'en accommoder (développement par incrément) :

Un **prototype** est une version initiale d'un système permettant de prouver sa faisabilité ou tester des variantes en se focalisant sur les inconnues du système. Il ne devrait jamais être mis en production, le système devrait être développé en tirant des leçons du prototype et non pas à partir de celui-ci. Il permet une meilleure compréhension des besoins de l'utilisateur et un effort de développement réduit.

Lors d'un **développement et livraison par incrément**, les incréments sont déployés et utilisés en production. Le logiciel est utilisable et disponible après chaque incrément. Les premiers incréments peuvent servir de prototypes. On a une réduction des risques d'échecs.

Afin d'**améliorer le processus logiciel**, on peut utiliser les bonnes pratiques (niveau de maturité) ou les cycles d'amélioration :

Les **niveaux de maturité** ont 5 étapes : initial (description des processus), managed (mise en œuvre des processus), defined (collecte des données de mise en œuvre), quantitatively managed (mesure de la performance) et optimizing (amélioration des processus).

Les **cycles d'amélioration** sont composés de 3 étapes qui se répètent en boucle : measure (collecte des données quantitatives comme le nombre de problèmes), analyze (analyse des données pour identifier les faiblesses du processus) et change (amélioration du processus).

La **rapidité** des cycles de développement et de livraison est aujourd'hui l'un des besoins les plus fondamentaux. Les autres besoins évoluent tellement vite qu'il est difficile de les spécifier sans qu'ils changent, le logiciel doit donc s'adapter aux besoins et non l'inverse.

Dans les approches **pilotées**, les itérations apparaissent au sein des activités mais parfois entre les activités dans les approches agiles.

Chapitre3 : Méthode agile

Il s'agit d'un développement sur mesure pour une organisation qui s'engage à prendre part au processus de développement et qui n'est pas sujette à une régulation qui affecte le logiciel.

L'**agile manifesto**

- valoriser les individus et leurs interactions plus que les processus et les outils
- des logiciels opérationnels plus qu'une documentation exhaustive
- la collaboration avec les clients plus que la négociation contractuelle
- l'adaptation au changement plus que le suivi d'un plan. Attention à ne pas tomber dans la programmation tactique et l'accumulation de complexité.

L'**extreme programming**	valeurs	principes
• cycles de développement court pour obtenir du feedback rapide, concret et continu • une planification progressive avec un plan global qui évolue tout au long du projet • une planification flexible en réponse aux besoins changeants de l'entreprise. • tests automatisés rédigés par programmeurs, clients et testeurs pour surveiller l'avancement, permettre au système d'évoluer et détecter tôt les erreurs.	simplicité, communication, feedback, respect, courage	sont l'humanité (sécurité matérielle, accomplissement, appartenance, croissance, intimité), l'économie (quelqu'un paie pour le projet), le bénéfice mutuel, l'auto-similarité, l'amélioration, la réflexion, le flux (vs grandes étapes), l'opportunité, la redondance, l'échec, la qualité, le petit pas et la responsabilité acceptée.

Enfin ses *pratiques* sont s'asseoir ensemble, l'équipe complète (toutes les compétences nécessaires doivent y être représentées), l'espace de travail informatif (mur de post-its), le travail énergisé (limiter les heures), la programmation en paire, les stories (unité de fonctionnalité visible par l'utilisateur), le cycle hebdomadaire (planifier une semaine à la fois et avoir un logiciel déployable à la fin de la semaine), le cycle trimestriel (réfléchir sur l'équipe le projet et ses progrès une fois par trimestre), builder en 10 minutes (tester toutes les 10 minutes), l'intégration continue (tests automatiques), le test-first programming et la conception incrémentale (un petit peu tous les jours). Très concentré sur le développeur, peu ou pas de manager qui doit seulement gérer les délais et le budget.

Le **SCRUM** est une méthode de gestion de projet agile qui se focalise sur la gestion d'un processus de développement itératif sans spécifier les pratiques qui l'accompagne. Il est composé de 3 phases : la phase *initiale* est une phase de planification générale pour définir les objectifs généraux du projet et concevoir l'architecture logicielle, suivie d'une série de *sprints* correspondant aux incréments du système et d'une phase de *clôture* permet de compléter la documentation, les manuels d'utilisation et de tirer des leçons du projet. Un sprint dure 2 à 4 semaines. Le point de départ de la planification est le backlog (liste des tâches à effectuer sur le projet). Durant la phase de sélection, l'équipe du projet et le client déterminent quelles tâches du backlog seront développées .L'équipe s'organise ensuite pour développer le logiciel et s'isole du client grâce au Scrum master. A la fin du sprint, une revue de projet est faite avec le client, puis un nouveau sprint démarre. La *vélocité* est une mesure de performance basée sur le nombre de tâches du backlog qu'une équipe peut exécuter en un sprint.

Chapitre4 :Requirement engineering

L'**ingénierie des exigences** est le processus permettant d'établir les services que le client exige du système, et les contraintes sous lesquels il opère et est développé. Les exigences d'un système sont les descriptions des services de ce système et les contraintes identifiées durant ce processus. Les requirements *business* (pourquoi) doivent être compréhensibles par les preneurs de décisions, les req. *utilisateurs* (quoi, appel d'offre) doivent être compréhensible par des personnes sans connaissances en informatique et les requirements *systèmes* (quoi, plus détaillé) peuvent être plus techniques, le tout dans un document clair, sans ambiguïté, facile à comprendre, complet et cohérent.

Les requirements **fonctionnels** décrivent les services fournis par le système et comment il réagit aux entrées utilisateurs et aux situations, éventuellement ce que le système ne doit pas faire. Les requirements **non-fonctionnels** sont les contraintes sur les fonctions du système, parfois liées au domaine (vitesse d'exécution, standards à respecter, technologies à utiliser, taille, facilité d'utilisation, fiabilité, robustesse, portabilité). Attention à ne pas avoir un trop haut niveau d'abstraction et donc à laisser place à des ambiguïtés.

Les **histoires** (texte narratif de haut niveau) et les **scénarios** (document détaillé et structuré) permettent de mieux comprendre le problème Un scénario contient une description de l'état initial, du flux d'événements, des cas d'échecs, des activités concurrentes et de l'état final.

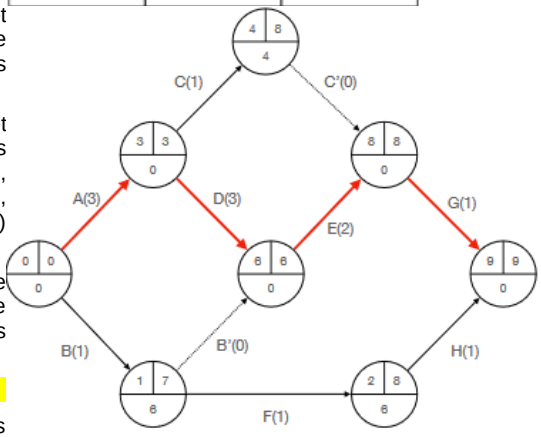
Une spécification de type **UML use case** identifie l'ensemble des acteurs et interactions du système dans une représentation duale, sous forme de graphiques et de textes. Une bonne spécification est valide, cohérente, complète, réaliste, vérifiable, compréhensible et traçable. Il place les acteurs principaux à gauche, secondaires à droite et les acteurs externes (par exemple un système d'audit) généralement en bas

Une **user story** est une description courte d'une fonctionnalité du point de vue de l'utilisateur/client (En tant que <un type d'utilisateur>, je veux <un but> pour que <une raison>). Une persona est un personnage de fiction représentant les utilisateurs utilisé dans les user stories.

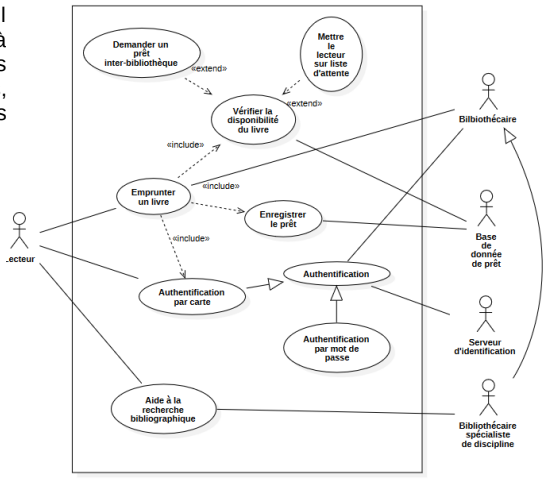
Chapitre 5 : Program Evaluation and Review Technique

PERT est un outil statistique utilisé pour représenter et analyser les tâches impliquées dans l'exécution d'un projet. Il permet d'estimer le temps même lorsqu'il y a de l'incertitude dans les estimations, d'identifier le temps minimum requis à l'exécution du projet et d'identifier et estimer la durée du chemin critique des grands projets. Il est composé de 5 phases : établir la liste des tâches, antécédents et durées, construire le réseau avec les dates au plus tôt, calculer les dates au plus tard, calculer les marges et enfin identifier le chemin critique.

Tâche	Antécédent(s)	Durée
A	-	3
B	-	1
C	A	1
D	A	3
E	D, B	2
F	B	1
G	C, E	1
H	F	1

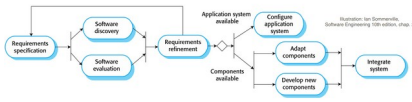


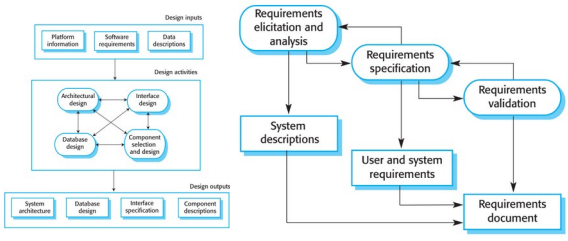
- Hypothèse d'une distribution normale
- La probabilité que la durée du projet soit $E(project) \pm SD(project)$ est de 68.26%
- **La probabilité que la durée du projet soit $E(project) \pm 2 \times SD(project)$ est de 95.44%**
- La probabilité que la durée du projet soit $E(project) \pm 3 \times SD(project)$ est de 99.72%

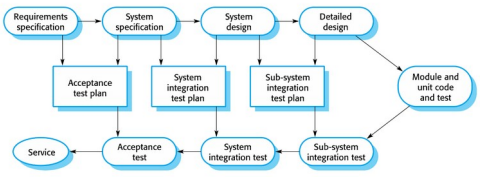


Distribution de PERT: $E(task) = \mu = \frac{o + 4a + p}{6}$ Distribution triangulaire: $E(task) = \mu = \frac{o + a + p}{3}$

$$E(project) = \sum E(task)$$







Déviation standard de PERT: $SD(task) = \sigma = \frac{p - o}{6}$ $SD(project) = \sqrt{\sum SD(task)^2}$