
Le projet **DELIRE**

**D**éveloppement par **E**quipe

de **L**ivrables **I**nformatiques

et **R**éalisation **E**ncadrée

# GP2 – Spécificité des projets logiciels





## LA PARTICULARITE DU DEVELOPPEMENT LOGICIEL

Le développement d'un système logiciel est avant tout le développement d'un système, et de ce fait il est piloté par les mêmes méthodes standards de développement de système que la conception d'un avion ou d'une voiture, nous le verrons plus tard.

Mais le développement logiciel présente 3 spécificités majeures :

1. Le coût de la fabrication et de la distribution d'un logiciel est quasi nul.
2. Le processus de développement est un processus incrémental
3. Il n'y a pas de produit virtuel pour simuler ce que donnera le produit logiciel

Le cout de la fabrication et de la distribution d'un logiciel est quasi nul.
En particulier, avec le réseau Internet, le client peut télécharger son produit, une nouvelle version de son produit ou un correctif sans avoir à le commander et le recevoir sous forme d'une bande ou d'un CD par la poste.
D'autre part, une fois le produit, la nouvelle version ou le correctif mis en ligne, que celui-ci soit téléchargé par un, mille ou un million de clients ne change pas le coût de l'opération pour l'éditeur de logiciel
Mais d'un autre coté, un logiciel est bourré d'erreurs : c'est normal, on considère de façon standard qu'un logiciel c'est 50 erreurs par kloc (kilo lines of code, millier de lignes de code). Bien entendu on inclut dans ces 50 erreurs les erreurs de design et les erreurs de coding trouvées avant la mise sur le marché. De ce point, on tire 2 conclusions :

1. L'investissement de qualité réalisé sous forme de tests avant la mise sur le marché est fondamental
2. L'investissement fait durant la phase de coding pour rendre le code clair et facile à débuguer est lui aussi fondamental

Il faut également savoir que :

1. Une erreur d'architecture est facile à trouver et difficile à corriger
2. Une erreur de coding est difficile à trouver et facile à corriger

En conception de logiciel, c'est donc le coût de conception et le coût de la maintenabilité qui doivent être optimisé
C'est une différence forte par rapport à un produit mécanique comme un avion ou une voiture : si le fait d'augmenter le cout de la phase de développement vous permet de diminuer le prix de fabrication, alors c'est une approche qui peut se révéler gagnante tant pour le client que pour le fabricant.

Le processus de développement est un processus incrémental
En d'autre terme, un produit peut être mis sur le marché même s'il est incomplet, dès lors qu'il apporte une véritable valeur ajoutée au client. Et on considère qu'un produit apporte de la valeur ajoutée lorsque le client est capable d'exécuter dans sa totalité (end to end) un processus industriel. Ainsi, dans un logiciel d'ERP (Entreprise Resource Planning) vous pouvez disposer de la gestion des commandes mais ne pas avoir encore la gestion des stocks.
Les fonctionnalités attendues par le client seront livrées au fil des releases. Le plan de développement est en général négocié avec le client, c'est un équilibre subtil entre :

1. Les fonctionnalités prioritaires pour le client
2. Le coût de développement de chaque fonctionnalité



3. Les possibilités offertes par la maturité de la plateforme.

C'est une différence forte par rapport à un produit mécanique : on imagine mal livrer un avion par tronçons, ou une demi-voiture

Il n'y a pas de produit virtuel pour simuler ce que donnera le produit logiciel

Le coding est lui-même le produit virtuel

Dans la conception d'un produit mécanique, la conception se fait dans le cadre d'un produit virtuel, une maquette numérique. On peut ainsi, alors que pas une seule pièce n'a encore été fabriquée, détecter des interférences et les corriger plutôt qu'avoir la désagréable surprise de les découvrir durant le premier assemblage.

Les méthodes dites de « Relational Design » vous permettent de capturer le réseau de savoir faire de la conception du produit, et de pouvoir ensuite converger à faible coût vers le jeu de paramètres optimal (optimal au sens du TCO, Total Cost of Ownership)

Dans la conception du logiciel, il n'existe malheureusement pas encore de langage universel qui permettent de capturer les intentions du concepteur, et de faire ensuite générer le code optimal. Il existe certes des langages qui ont globalement cette caractéristique, mais ils sont très spécifiques d'une zone de compétence donnée :

1. C'est le cas de générateur d'interface user
2. Ou de structuration des tables d'une base de données.

Ces langages sont en général difficiles à apprendre et à utiliser, et ne se justifient que pour des personnes dont le développement de logiciel spécifique est le métier.

Si on se pose une question dans le développement du logiciel, il est nécessaire de faire un prototype : c'est un logiciel avec des critères de qualité bien moindres, et de ce fait nettement moins couteux, mais sa caractéristique est également de produire du code jetable.

On tente en général d'éviter le recours au prototype par :

1. Un suivi assez rigoureux des méthodes de conception de système, en particulier lors de l'architecture du système et dans l'élaboration des critères qualité.
2. L'appui sur l'expérience des développeurs. Expérience qui se construit par des années de pratique.

Malheureusement, dans le projet DELIRE, votre expérience est encore limitée.




## LES METHODES DE DEVELOPPEMENT

Lorsque l'informatique est apparue, dans les années 40, développer était à la fois très compliqué et très simple.

1. Très compliqué car les capacités du langage et la taille des données manipulables étaient plus que limitées
2. Très simple parce que les problèmes qu'on cherchait à adresser étaient basiques.

Les progrès aidant, tant dans la puissance des ordinateurs que dans l'apparition de nouveaux langages plus évolués (le premier langage structuré, FORTRAN, FORmula TRANslator, date de 1956) ont conduit les programmeurs à traiter des problèmes de plus en plus complexes.
Mais la méthode pour programmer était celle du joyeux bordel : dès qu'on avait une idée, on passait une demi-heure à poser les principes de ce qu'on allait faire, et on se jetait sur le clavier pour pisser le programme qui allait résoudre ledit problème. Et c'était une époque merveilleuse, où on était capable de traiter enfin des problèmes qui dépassaient jusque-là les capacités humaines.
L'offre créant le besoin, les demandes des services informatiques se sont faites de plus en plus gourmandes. Dans les années 60 ce sont surtout la gestion (les commandes, les feuilles de payes) et les banques (la gestion des comptes, les bilans) qui sont les gros consommateurs d'informatiques. L'industrie ne se met à l'ordinateur que dans les années 70 : calcul de RDM, commande numérique…

Au point que l'offre n'a plus été capable de suivre la demande. Dans les années 70 on a parlé de crise du logiciel, engendrée par 2 phénomènes concourants :

1. La population des programmeurs augmentait de 4% par an, la productivité augmentait de de 4% par an, mais la demande de logiciel progressait de 12% par an.
2. En parallèle le cout de la maintenance devenait prohibitif

C'est à l'armée (essentiellement le DOD, Departement of Defense, et à NATO ou OTAN en français) que revient le privilège d'avoir fédéré dès la fin des années 60 la réflexion qui allait donner naissance aux méthodes de développement et au génie logiciel.

Les réflexions vont déboucher sur un certain nombre de propositions :

1. Il faut suivre un cycle formel de développement qui permette de développer à coût minimum un programme qui satisfasse le besoin du client. C'est l'origine du modèle du cycle en V
2. Il faut introduire la notion de composant, avec des interfaces clairement identifiées, pour favoriser la collaboration du développement. Et cette notion engendre de facto la nécessité et l'importance de l'architecture.
3. Il faut mettre en place des bibliothèques de composants réutilisables, pour ne pas avoir à réécrire systématiquement les mêmes programmes. Ceci conduira aux concepts de la Conception et de la Programmation Orientée Objets.
4. La qualité n'est pas une notion qu'on rajoute en fin de cycle mais quelque chose qui doit être intégrée dans tout le cycle de développement.

Et toutes ces propositions seront à l'origine du génie logiciel



Pour être tout à fait honnête, le premier cycle de développement qui avait été proposé était le **cycle en cascade**, hérité de l'industrie du BTP (Bâtiment et Travaux Publics).
Le principe de base étant de construire le logiciel par couches successives, en partant des couches d'infrastructure (Connexion à l'Operating System, gestion de la persistance des données…) pour terminer par les services de présentation (graphique, dialogue…).
De la même façon que dans l'industrie du bâtiment on construit les murs après les fondations, murs qui seront suivis la charpente, laquelle précède la couverture pour mise hors d'eau.
Comme dans l'industrie du BTP, l'organisation était :

1. Des livrables clairement définis au préalable ;
2. Une livraison à une date précise
3. Une étape de validation-vérification lors de la recette d'un ensemble de livrables.

Le problème majeur de cette approche du développement était qu'on pouvait découvrir un gros problème lors de l'assemblage des derniers livrables, ce qui imposait une modification dans les composants amonts, et donc un impact majeur sur le Délai et le Coût du projet.
Mais on a néanmoins gardé dans les méthodes suivantes les bonnes notions de ce cycle de développement

Le modèle du **cycle en V** a été imaginé pour contourner le problème de réactivité du modèle en cascade. Il permet en cas d'anomalie, de limiter un retour aux étapes précédentes. Les phases de la partie montante doivent renvoyer de l'information sur les phases en vis-à-vis lorsque des défauts sont détectés afin d'améliorer le logiciel.

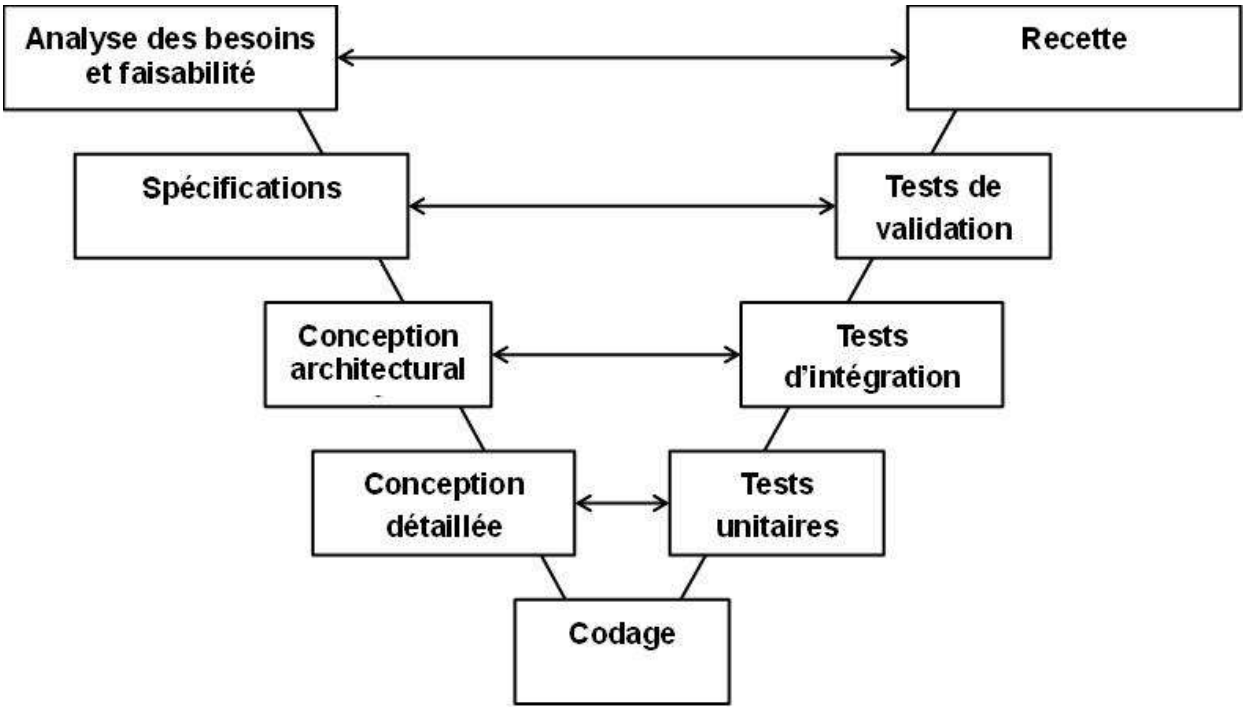
Le cycle en V est décrit sous forme :

1. D'une partie descendante, décomposée en 4 phases
    a. Analyse des besoins et faisabilité
    b. Spécification logicielle
    c. Conception architecturale : une décomposition récursive des composants en composants élémentaires, et la façon dont lesdits composants s'échangent les informations
    d. Conception détaillée : l'architecture interne de chaque composant élémentaire sous forme d'unité de codage, et la structure logique de chacune de ces unités
2. D'une phase de codage : transformer l'architecture interne de chaque composant élémentaire en code exécutable
3. D'une partie ascendante, décomposée en 4 phases
    a. Test unitaires : validation « boite blanche » de chaque unité de codage de chaque composant élémentaires
    b. Tests d'intégration : validation de la connexion, de la cohérence et de la complétude des composants élémentaires
    c. Tests de Vérification, de Validation : validation que le logiciel répond à l'ensemble des spécifications techniques
        i. La validation est la preuve tangible que les exigences initiales ont été satisfaites pour une utilisation spécifique donnée. Valider, c'est répondre à la question : « Faisons- nous le bon produit ? ». C'est une approche orienté produit



ii. La vérification est la preuve tangible que les exigences initiales ont été satisfaites dans l'ensemble du domaine d'utilisation. Vérifier, c'est répondre à la question : « Faisons- nous le produit de la bonne façon? ». C'est une approche orientée process.

d. Recette ou Qualification : validation par le client que le produit qui lui est livré est conforme au contrat initial.




Selon la phase, c'est le maitre d'ouvrage (MOA) ou le maître d'œuvre (MOE) qui porte la responsabilité du projet.

| Répartition des rôles en fonction des étapes | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Niveau de détail | | Besoins et Faisabilité | Spécifications | Conception Architecturale | Conception Détaillée | Codage | Tests unitaire | Tests d'intégration | Tests de vérification, de Validation | Recette |
| Système | MOA | X | | | | | | | | X |
| Fonctionnel | MOE | | X | | | | | | X | |
| Technique et Metier | MOE + Equipe architecturale | | | X | | | | X | | |
| Composant | MOE + Equipe de Dévelop | | | | X | X | X | | | |

On retrouve dans ce découpage le V, d'où le nom de ce modèle. V peut être également pris pour Vérification, Validation
La bonne communication entre les différents acteurs du projet passe par la formalisation de documents de référence, les livrables.

Le cycle en V met en évidence la nécessité d'anticiper et de préparer dans les étapes descendantes les attendus des futures étapes montantes. Dans la pratique :

1. les tests de recette sont décrits durant la phase de l'analyse des besoins et faisabilité
2. les tests de vérification et de validation sont décrits durant la phase de spécification fonctionnelle
3. les tests d'intégration sont décrits durant la phase de conception architecturale
4. les tests unitaires sont décrits durant la phase de conception détaillée

Il est même conseillé d'écrire les tests en début de phase : cette écriture anticipée donne de précieux conseil sur la façon de réaliser la phase.

| Documents en fonction des étapes | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Besoins et Faisabilité | Spécifications | Conception Architecturale | Conception Détaillée | Codage | Tests unitaires | Tests d'intégration | Tests de vérification, de Validation | Recette |
| Spécification des Besoins Utilisateur<br>Cahier des charges | | | | | | | | Rapport de Recette |
| | Spécifications Générales<br>Spécif Tech des Besoins | | | | | | Procès Verbal de Validation | |
| | | Dossier de Déf du Logiciel<br>Dossier Archi Technique<br>Plan de tests | | | | Rapport de Tests d'intégration | | |
| | | | Rapport de Conception Détaillée | | Rapport de Tests Unitaires | | | |
| | | | | Codage | | | | |

On peut donc considérer qu'il existe deux axes de temps dans le cycle en V

1. Un axe de temps vertical qui correspond à la conception, de plus en plus détaillée
2. Un axe de temps horizontal, qui correspond à la réalisation, c'est à dire l'enchainement des opérations.

Le cycle en V a pour origine l'industrie lourde. Une des particularités est que chaque étape de la phase descendante nécessite beaucoup plus de ressources que la précédente.
Le cycle en V a permis au développement du logiciel de devenir une véritable industrie. Actuellement des éditeurs de logiciels emploient plusieurs milliers de développeurs (ORACLE : 100000 employés, Microsoft : 90000 employés, SAP : 50000 employés) et la plus grosse équipe de développement travaillant simultanément sur un même produit est celle de Dassault Systèmes, 4500 ingénieurs.
Le cycle en V est devenu le standard de la gestion de projet logiciel dans les années 80, et il a depuis prouvé sa robustesse en devenant dans les années 90 le standard conceptuel dans tous les domaines de l'Industrie pour le management des systèmes complexes.

Deux reproches majeurs ont été formulés à l'égard du cycle en V

1. Il arrive que, lors de la phase d'implémentation (celle où on « pisse » le code) on découvre que les spécifications sont incomplètes, incohérentes, fausses ou à tout le moins irréalisables. Ce problème a été en partie adressé par la notion de traçabilité des exigences
2. Une évolution des besoins du client apparait durant la réalisation du projet. Et ce phénomène est généralement accentué par le fait que le cycle en V est un cycle assez long. Ce problème a été traité par le cycle en spirale.

Le **cycle en spirale** a pour objectif de construire un produit non pas d'un seul jet mais par enrichissement successif dans le déroulement des différentes versions du produit. Sur le principe il reprend les différentes étapes du cycle en V et par itération conduit à un produit de plus en plus complet et robuste.



Au début de chaque itération, on commence par une phase d'analyse des risques, d'analyse des défauts de l'itération précédente et des évolutions des besoins du client.
Le cycle en spirale permet d'avoir des temps de cycle courts. Le cycle en spirale est utilisé par les éditeurs de logiciel (dans la réalité le développement d'un produit comme Word n'est jamais fini) plus que par les SSII qui construisent des logiciels à façon.

Le **cycle semi-itératif** est apparu au début des années 90 aux USA sous le nom de RAD (Rapid Application Developpement). Il a été consacré par son adoption, dans le cadre de Unified Process, par Rational, filiale d'IBM et propriétaire du langage UML (Unified Modeling Language)
Le cycle semi itératif reste relativement proche du cycle en V dans sa phase descendante (de l'analyse des besoins jusqu'à la conception détaillée). C'est durant le codage et dans la phase ascendante de tests que la notion d'itération courte intervient : plutôt que développer l'ensembles des composants en parallèle et de poursuivre par une phase d'intégration, on préfère livrer et tester régulièrement dans le cadre d'un mécanisme essai/erreur les composants au fur et à mesure de leur avancement pour identifier au plus vite les problèmes d'incohérence en particulier en termes d'architecture ou de performance. Cette itération se fait tant dans la constitution des composants (on reprend ici certaines idées du cycle en spirale, on commence par les composants d'infrastructure) que dans la progression de chaque composant (on commence par les fonctions prioritaires et à fort impact)

Les **méthodes agiles**, poussant au paroxysme la notion d'itération, sont des méthodes itératives, incrémentales et adaptatives qui sont apparues au début des années 2000, en partie en réaction face à la lourdeur du cycle en V. Elles visent la satisfaction réelle du besoin du client et non les termes d'un contrat de développement : l'idée est de livrer au plus tôt quelque chose qui puisse être testé par le client. On est dans un processus d'amélioration continue de la qualité

Chaque cycle itératif ne dépasse pas deux mois. Il comporte 4 étapes :

1. La faisabilité :
    - La spécification
    - C'est l'acceptation d'un nouveau besoin
2. L'élaboration
    - L'architecture
    - On imagine comment on va le réaliser
3. La fabrication :
    - La construction du prototype
4. La transition :
    - Les tests
    - Tout est mis en œuvre pour livrer au client
    - Le client est fortement impliqué dans la validation du prototype et l'analyse des manques, qui permettront de spécifier les objectifs de l'itération suivante.

Les méthodes agiles défendent 4 valeurs fondamentales tournant autour de la communication :



1. L'équipe : personnes et interactions plutôt que processus et outils
    - Il vaut mieux avoir une équipe soudée de développeurs qui communiquent qu'une équipe composée d'experts autistes. .
2. L'application : logiciel fonctionnel plutôt que documentation complète
    - Le but est d'avoir l'application qui fonctionne.
    - La documentation technique est certes une aide précieuse mais non un but en soi. Elle représente une charge de travail importante, et peut n'être pas à jour.
    - Il est préférable de commenter abondamment le code, et surtout de diffuser les compétences au sein de l'équipe
3. La collaboration : collaboration avec le client plutôt que négociation de contrat
    - Le client doit être impliqué dans le développement.
    - On ne peut se contenter de négocier un contrat au début du projet, puis de négliger les demandes du client.
    - Le client doit collaborer avec l'équipe et fournir un feed-back continu sur l'adaptation du logiciel à ses attentes.
4. La réactivité : réaction au changement plutôt que suivie stricte d'un plan
    - La planification initiale et la structure du logiciel doivent être flexibles afin de permettre l'évolution de la demande du client tout au long du projet.
    - Les premières releases du logiciel vont souvent provoquer des demandes d'évolution.

Les méthodes agiles ont été largement popularisées depuis 10 ans, en particulier par les startups de quelques personnes qui peuvent ainsi mettre au point rapidement un produit répondant parfaitement aux besoins de leur premier client, et leur permettant d'accumuler simultanément du savoir-faire.
Elles permettent d'éviter la lourdeur d'une méthode comme le cycle en V et permettent de maintenir une pression permanente. Elles sont donc très populaires chez les jeunes développeurs qui y voient un espace de liberté.

Il est intéressant de retrouver en partie dans cette approche la façon dont on développait il y a 40 ans, avec les limitations afférentes. Les principales limites des méthodes agiles sont :

1. Risques de dérive de la finalité du produit, pouvant amener à la fin à un système mal positionné
2. Documentation insuffisante pouvant faire perdre une partie du savoir faire
3. Inadaptation à des projets de grande envergure.



Si on tente de comparer l'utilisation d'une méthode agile plutôt qu'une méthode plus traditionnelle de cycle en V, on mettra en évidence :

1. des facteurs favorisant la méthode agile :
    - Besoin rapide de mise à disposition du produit au client pour le rassurer d'une part, et pour lui permettre de mieux formaliser son besoin
    - Imprévisibilité du besoin du client, nécessité de changements fréquents
    - Nécessité de feedback de la part de l'utilisateur, par exemple pour l'IHM (Interfaces-Homme-Machine) ou les APIs (Applications Programming Interface)
2. Des facteurs la défavorisant :
    - Indisponibilité chronique du client
    - Gouvernance complexe, répartition distribuée des responsabilités, nombre d'acteurs important
    - Nécessité d'une importante connaissance du métier de l'utilisateur.

Les deux approches ne sont néanmoins pas totalement incompatibles : on peut utiliser une méthode agile pour mettre au point un prototype, même dans le cadre d'un projet de grande taille.  La méthode agile la plus populaire en France est XP (eXtrem Programming) apparue à la fin des années 90

Aussi désolant que cela puisse vous paraître, je vais vous imposer de suivre un cycle en V ou un cycle semi itératif pour le projet DELIRE. Mon objectif est de vous faire découvrir, au travers de ce projet, certaines des méthodes de management des systèmes complexes.

La seule phase où vous aurez un peu de liberté est celle de la réalisation (la phase de codage). Je vous laisse libre de travailler par itération sur le codage de vos composants.



**ET POUR DELIRE ?**

Pour DELIRE, nous allons essentiellement nous polariser sur la partie sous la responsabilité du MOE (maître d'œuvre).

Ainsi, le cahier des charges devra être rapidement défini, de façon à ce que l'équipe puisse se consacrer à la définition des spécifications (SFG) du produit.

De même, en fin de projet, nous nous arrêterons au stress tests (si vous en avez le temps). La partie tests de recette ne sera pas réalisée. Dit autrement, il vous faudra considérer la soutenance de fin mars comme la formalisation de la remise par le MOE (maître d'œuvre) du produit au MOA (maître d'ouvrage) à des fins de recette.

Dans votre carrière, vous serez amenés à être maîtres d'œuvre ou maîtres d'ouvrage de projets plus ou moins importants. Les deux casquettes sont passionnantes

Le MOE a un cadre de travail (le cahier des charges) et une méthode de gestion de projet pour réaliser sa tâche. Contrairement à ce qu'on peut intuitivement penser, le travail de MOA est le plus difficile

Le MOA, lui, va devoir

1. Imaginer tous les besoins du client pour rédiger le cahier des charges
2. Répondre à toutes les demandes de choix de solution qui lui seront présentés par le MOE
3. Prendre la décision finale d'accepter le produit, au risque de découvrir beaucoup plus tard un problème qui se révèlera critique pour l'utilisation opérationnelle du produit.

Ne l'oublions pas, c'est le client qui paye et qui prend donc les plus gros risques.

Ce sont les ingénieurs qui ont acquis un véritables savoir-faire technique en réalisant des projets en tant que MOE qui se révèlent plus tard les meilleurs MOA.

Dans le projet DELIRE, apprenez en tant que MOE à ne pas rouler le MOA dans la farine, mais à développer des relations de confiance. L'objectif est que chacun se sente gagnant à la fin du projet.

Si au contraire vous cherchez à piéger le MOA, vous aurez ensuite du mal à assumer un rôle de MOA dans votre carrière : vous serez persuadés que le MOE cherche à vous avoir, vous deviendrez de ce fait pénible car tatillon et dubitatif sur ce qu'on vous proposera, ce qui créera une ambiance délétère dans le projet.

Vous allez passer 42 ans dans l'industrie, 45 semaines par an et facilement 40 heures par semaine au travail. 67200 heures. Autant que cela se passe le mieux possible.