Université Libre de Bruxelles

ECOLE POLYTECHNIQUE DE BRUXELLES

Bloc 3 - Génie logiciel et gestion de projet

Résumé du cours Génie et logiciel et gestion de projet

Auteurs :
Billy **Tran**

Professeur :
Frédérique **Pluquet**

30 mai 2017

Table des matières

Table des figures

Liste des tableaux

1 Rappels et approfondissement d'Analyse et méthodes

1.1 Etapes de développement

1.1.1 Appréhender le problème

Il s'agira de bien comprendre le problème afin de mieux la gérer par la suite.

1.1.2 Analyser le problème

Analyser consiste à comprendre le problème pour pouvoir en définir une solution ainsi que sa modélisation.

Comprendre finement le problème

Il faut rechercher dans le SRD(Software Requirements Documents) les requis fonctionnels et non fonctionnels que le développeur et le client se sont mis d'accord. Un requis fonctionnel c'est ce qui définit le système et un requis non fonctionnel c'est ce que le système devrait supposer être.

Définir et modéliser la solution

L'UML (Unified Modeling Language) est un moyen d'interpréter un programme correspondant à un domaine du monde réel afin de faciliter sa compréhension . Plusieurs diagrammes sont possibles :

1. Diagrammes de cas d'utilisation (Use Cases) : utilisés pour donner une vision globale du comportement fonctionnel d'un système logiciel. Ils sont utiles pour des présentations auprès de la direction ou des acteurs d'un projet, mais pour le développement, les cas d'utilisation sont plus appropriés
2. Diagrammes de classes : Le diagramme de classes est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que les différentes relations entre celles-ci. Ce diagramme fait partie de la partie statique d'UML car il fait abstraction des aspects temporels et dynamiques.
3. Diagramme de séquence : Les diagrammes de séquences sont la représentation graphique des interactions entre les acteurs et le système selon un ordre chronologique dans la formulation Unified Modeling Language.
4. Diagramme d'états : schéma utilisé en génie logiciel pour représenter des automates déterministes. Il fait partie du modèle UML et s'inspire principalement du formalisme des statecharts et rappelle les grafcets des automates. S'ils ne permettent pas de comprendre globalement le fonctionnement du système, ils sont directement transposables en algorithme. Tous les automates d'un système s'exécutent parallèlement et peuvent donc changer d'état de façon indépendante.
5. Diagramme de composants : décrit l'organisation du système du point de vue des éléments logiciels comme les modules (paquetages, fichiers sources, bibliothèques, exécutables), des données (fichiers, bases de données) ou encore d'éléments de configuration (paramètres, scripts, fichiers de commandes). Ce diagramme permet de mettre en évidence les dépendances entre les composants (qui utilise quoi).
6. Diagramme de déploiement : une vue statique qui sert à représenter l'utilisation de l'infrastructure physique par le système et la manière dont les composants du système

sont répartis ainsi que leurs relations entre eux. Les éléments utilisés par un diagramme de déploiement sont principalement les nœuds, les composants, les associations et les artefacts. Les caractéristiques des ressources matérielles physiques et des supports de communication peuvent être précisées par stéréotype.

1.1.3 Réaliser la solution

Les solutions sont réalisées avec la programmation en utilisant les notions suivantes :
- Objets/Instances : message
- Classe :attribut et méthodes
- Abstraite/Interface
- Héritage
- Method lookup(Encapsulation)
- Polymorphisme

Mais aussi en réalisant une certaine architecture comme le MVC voir figure1

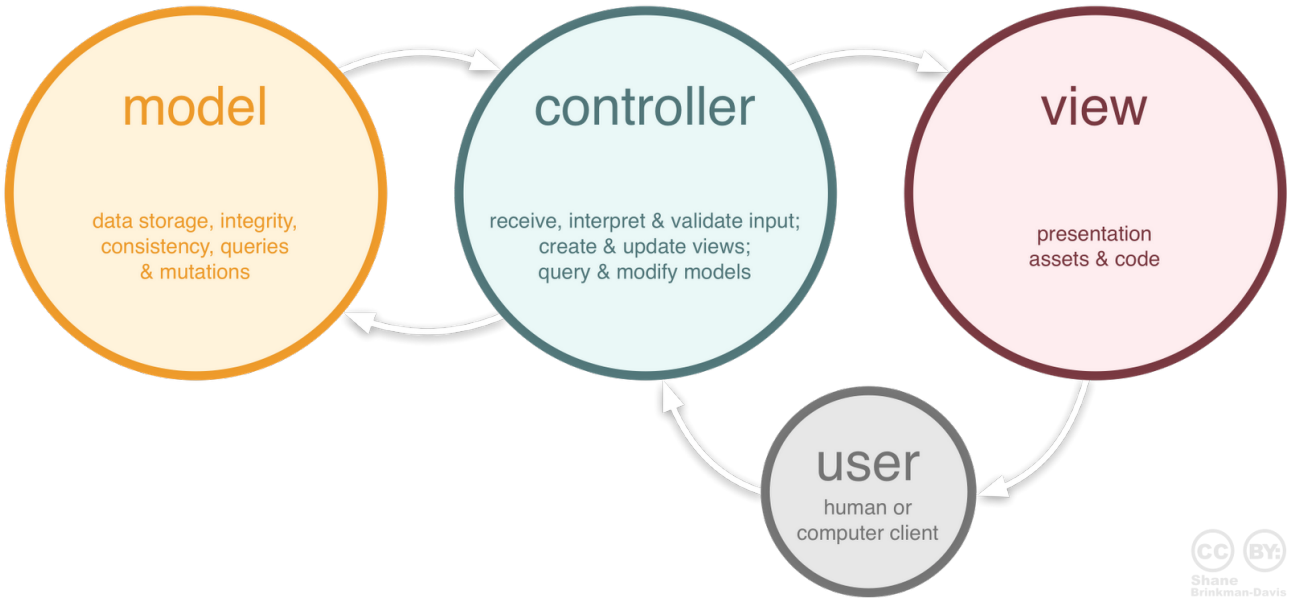

FIGURE 1 – Architecture

Le MVC est un design pattern mais ils en existent d'autres comme les composites , l'observer , strategy , factory , bridge etc...
Comment fonctionne le MVC ?

1. L'utilisateur met quelques choses (appuie sur un bouton, clique,...)
2. Le Contrôleur va, à l'aide de ses méthodes , envoyer à la Vue les modifications et va modifier dans le modèle.
3. le modèle et la vue s'envoient des requêtes et des mises à jours entre-eux.

Réaliser une solution c'est facile mais est-ce une bonne solution dans le sens est-ce que c'est bien codé ? Pour cela il faut :

1. **Une convention standard de nommage** (par exemple un nom d'attributs ou méthodes : objetAttribut ou objet_attribut).
2. **La longueur des méthodes** ne doivent pas dépasser 10 lignes de préférences.
3. **Le code dupliqué**doit être évité.

4. **Refactoring** permet d'éviter de faire de bêtes erreurs en renommant ou en reproduisant du code dans une autre classe...
5. **Les nombres magiques** doivent être évités. (Les nombres provenant d'allocation par exemple)
6. **GRASP** (General responsibility assignment software patterns) reprend deux principes :
   (a) La haute cohésion , c'est-à-dire, qu'il faut que les liens entre les classes soient cohérentes.
   (b) Couplage faible, c'est-à-dire, qu'il faut que les composants de la fonction communiquent un minimum
7. **La loi de Demeter** dit que toute méthode M d'un objet O peut simplement appeler les méthodes de l'objet lui-même, des paramètres de M , des objets membre de O ou des objets instanciés dans M. Afin d'éviter un couplage fort .

1.1.4 Tester

Tester consiste à d'abord créer un contexte , envoyer un stimulus et vérifier le résultat.

1.1.5 Logiciel de qualité

On dit que le logiciel est de qualité si l'ensemble des caractéristiques d'une entité qui lui confèrent l'aptitude à satisfaire des besoins exprimés ou implicites. C'est à dire que le code doit être :

1. Pertinent
2. Apport de bénéfice
3. Fonctionnement correct (pas de bugs, pas de pertes de données , sécurité)
4. Une bonne performance
5. Facile de comprendre , d'apprendre
6. évolutivité , on peut encore l'améliorer
7. Réutilisable , on peut l'utiliser pour un autre domaine
8. Portabilité , facilement portable dans un autre os.

1.1.6 Méthodologie

Par définition , il s'agit d'un procédé qui a pour objectif de permettre de formaliser les étapes préliminaires du développement d'un système afin de rendre ce développement plus fidèle aux besoin du client.
Il existe deux grandes méthodes :

Waterfall :

Elle consiste à suivre un ordre d'étapes dont une étape nécessite l'accomplissement de l'étape précédente et part sur une philosophie " Définir très précisément ce que l'on souhaite construire avant de le construire. Cela peut se résumer à ces étapes suivantes : Pré-étude (Définition du cahier des charges) , analyse(document de spécification), conception (document de conception générale), construction (document de conception détaillée) , tests (plan de tests), déploiement(faire l'application).
Cette méthode a bien des limites et a un bon pourcentage d'échouer car c'est très difficile d'avoir dès le début une spécification et une conception poussée , il va y avoir constamment des

idées qui vont venir durant le projet et il faudra donc modifier mais avec la méthode Waterfall il faudra alors refaire toute la procédure depuis le début.

+ Découpe en phase donc parallélisation possible

  vision simplifiée de la gestion du temps pour les managers

+ Utile si besoin de tout spécifier avant de construire

  dans le cas où chaque étape peut coûter beaucoup alors on doit bien faire attention à ce qu'on doit faire.

- compartimentation des équipes

  Séparation en développeurs et analystes où chaque développeur a son code à faire et est coordonné par un chef d'équipe .

- qualité faible du code car chacun développe sa partie dans son coin.(aucune forme d'unité)

-Temps de développement plus long car il faut que le groupe chargé de réunir les parties comprennent ce qu'on fait les autres.

Agile :

Cette méthode reprend les principes où les requis et les solutions évoluent au fur et à mesure à travers des efforts collaboratives avec une organisation des membres par eux-mêmes. La méthode que nous allons voir ici est une méthode parmi tant d'autres d'Agile. Il s'agit de l'eXtreme Programming.
Remarque : les sections Gestion de projet , gestion d'équipe et gestion de la qualité du code sont bien sûr une section dédiée au XP.

2 XP(eXtreme Programming)

L'extreme programming possède une plus grande agilité au niveau du code ,de l'équipe de développement et au niveau de la gestion du projet. Dans une équipe XP, il y a le clien/testeur qui crée des histoires,scénarios de tests pour le programmeur qui lui fournit au client un logiciel ou un prototype. Entre les deux, il y a le coach qui sera présent pour aider les deux partis.

1. Pratique de gestion de projet
   (a) Livraison fréquente (après x temps il faut monter au client le travail)
   (b) planification itérative(à chaque fois il faut planifier avec le client le prochain travail à faire)
   (c) Client doit être sur le terrain(le client doit essayer d'être le plus présent possible avec les programmeurs pour que lui voit mieux le programme et que les développeurs puissent poser des questions et avoir une réponse rapidement).
   (d) Rythme durable
2. Pratiques de collaboration
   (a) Programmation en binôme (pair programming)
   (b) responsabilité collective du code
   (c) règles de codage
   (d) métaphore
   (e) intégration continue

3. Pratiques de programmation
   (a) Conception simple
   (b) Refactoring
   (c) Développement piloté par les tests unitaires
   (d) Tests de recette

XP valorise 4 points :

1. La communication : Le développement est équivalent à un effort collectif des membre de l'équipe ainsi que du client. Une bonne communication permettra une meilleure coordination. En effet, pouvoir parler directement est plus facile pour exprimer ses idées personnelles. Le seul souci, c'est qu'il n'y a pas d'historique ou de structures de l'information.
2. Simplicité : Il faut rester simple. Il ne faut pas commencer à chercher loin, il faut penser à faire la chose la plus simple qui puisse marcher.
3. Feedback : Très important, il faut savoir à tout moment où en est le projet que ce soit pour le client ou pour le développeur. Il faut rectifier les erreurs de conception si besoin est et réduire les risques. Il faut utiliser une itération pour planifier la suivante et le feedback des changements grâce aux tests unitaires et du binôme. Voici comment se passer une boucle feedback (voir Figure2.
4. Courage : Il faut du cran pour se lancer dans un projet sans savoir la destination finale , il faut vouloir se borner à réaliser des choses simples , **se focaliser uniquement sur les besoins du moment** , pouvoir **jeter une partie de code devenue inutile** , **réécrire du code devenu trop complexe** , **communiquer tout le temps** , chercher le feedback , dévoiler **ses faiblesses à son binôme**

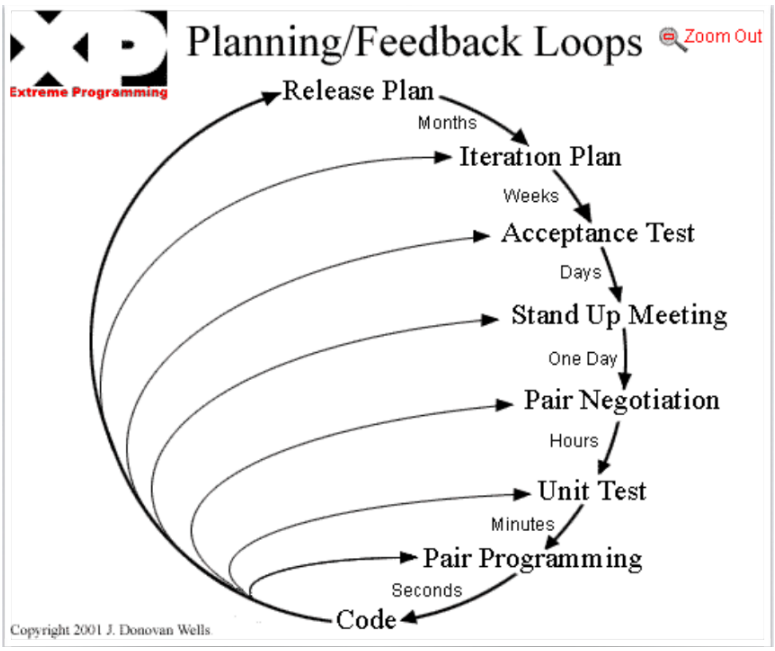

FIGURE 2 – Boucle du feedback

3 Gestion de projet

La partie gestion de projet peut se décomposer en ces différents principes :

1. Rechercher le rythme optimal
2. (Re)définir régulièrement le projet
3. Maintenir l'équilibre entre développeurs et client
4. Définir les spécifications tout au long du projet

3.0.1 Rythme optimal

Pour avoir du bon code , il faut pas se presser mais il ne faut pas être trop relaxe car du code de mauvaise qualité est plus dur à maintenir, prend plus de temps pour chaque changement effectué.
Il faut trouver son rythme optimal malgré les contraintes entre celui-ci et les contraintes de l'entreprise.
Un projet est dimensionné par 4 variables :

1. **le coût :** Engager ou changer d'environnement de travail peut poser des risques sur le court terme.
2. **les délais de livraison :** Les coûts indirects de synchronisation et éroder la confiance du client.
3. **la qualité du produit :** Moins de qualité donc plus de temps nécessaire pour le développement.
4. **le contenu du produit :** Le seul critère sur lequel on peut jouer.

3.0.2 Redéfinition régulière

On reprend le dicton du waterfall " Visez , Feu , raté" et celui de l'XP "Feu , Visez visez visez" qui explique bien la démarche de redéfinir de l'XP.

3.0.3 équilibre entre développeurs et client

Il est très important que le client et les développeurs jouent chacun leur rôle. Le client ne doit pas avoir trop de pouvoir car il pourrait demander l'impossible mais l'inverse non plus car ils pourraient faire un projet trop générique qui ne convient pas au client.
XP impose à chacun des responsabilités pour régler ça. Le client se doit de définir les fonctionnalités et de décider de l'ordre d'implémentation et les développeurs fournissent les estimations de coûts et se chargent de la réalisation des fonctionnalités.(voir Figure3)

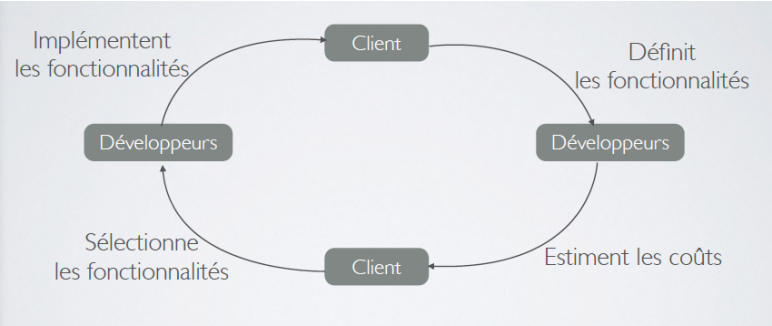

FIGURE 3 – Equilibre client-développeur

3.0.4 Spécifications

En reprenant les points de l'introduction de l'XP au niveau pratique, nous avons :

1. Client sur site
2. Rythme durable
3. Livraisons fréquentes
4. Planification itérative

Client sur site :

La communication à distance est une mauvaise idée , il est préférable que le client se trouve dans le même bureau que les développeurs afin de favoriser les questions directes et surtout des réponses directes. En effet le client peut apporter ses compétences métier et comme il n'y a pas de spécifications il faut constamment communiquer avec le client. Si il y a une question de fonctionnalité , c'est au client de répondre et pas aux développeurs car c'est le client qui décide ce qu'il veut.
Les tests unitaires doivent être fait (valide la mécanique interne de l'application) selon ce que veut tester le client ainsi que les tests automatiques.
Le soucis c'est que le client ne peut pas toujours être disponible. Ce n'est pas grave, plusieurs personnes peuvent jouer le rôle du client mais il faut limiter au maximum le nombre de personnes éloignées du cadre final pour éviter les divergences de points de vue des clients.

Rythme Durable :

Il faut pas surcharger son travail et penser à se reposer. Trop travailler pour réduire la qualité du code.

Livraisons fréquentes :

Deux idées principales, l'un est de rendre une 1ère livraison assez rapidement pour éviter les malentendus et donner de la consistance au projet et l'autre doit être une livraison aussi rapprochée que possible (pilotage précis et preuves fréquentes de l'avancement).
Il faut faire des livraisons quotidiennes (à l'aide de l'intégration continue et des tests automatiques) mais dans le cas d'un processus trop lourd c'est mieux de limiter les fréquences.
Ceci permet d'avoir de meilleurs feedback du client , de mieux cerner les besoins pour les livraisons suivantes , d'avoir un feedback pour l'équipe, d'avoir un sentiment régulier de " travail fini" et surtout entretient la motivation et diminue la pression.

Planification itérative :

La planification itérative permet de redéfinir le contenu des livraisons à l'aide du planning game qui consiste en un jeu (moins de tension) et d'un but (tirer le meilleur produit du projet). Le jeu doit se diviser en plusieurs d'itérations où une itération dure 1 à 3 semaines. Une livraison est un ensemble d'itérations. (voir Figure4)

Dans la partie exploration, le client définit ses scénarios (à la main et dans son propre langage) et décrivent **les interactions** entre l'utilisateur et le système pour une fonctionnalité donnée le plus simple possible. Chaque scénario est noté sur une petite fiche cartonnée(A5). Sur cette fiche on y marque la priorité fonctionnelle (1 :indispensable à 3 :utile), le risque technique (1 : Fort à 3 :Faible) , la description de la fonctionnalité et son estimation en "points". Le client définit les priorités et à chaque itération il répartira ses scénarios dans les itérations pour avoir les plus importantes d'abord. Comment ça se passe ?

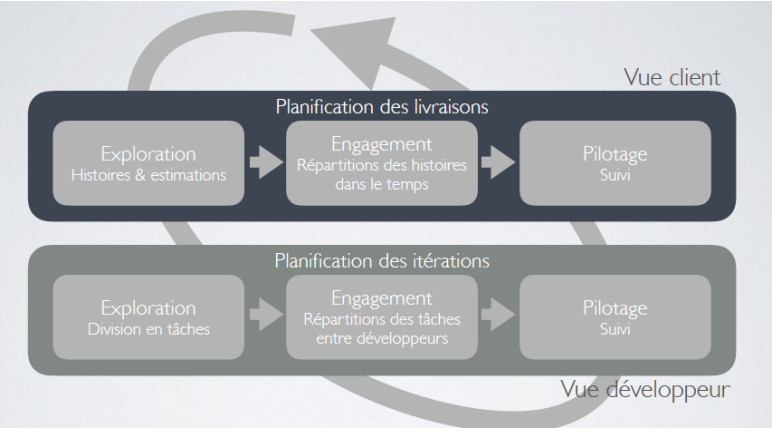

FIGURE 4 – Planification itérative des livraisons

Le client n'écrit que les titres de ces scénarios et racontent oralement chaque histoire pour que les programmeurs se fassent une idée générale de l'application. Les développeurs peuvent poser toutes les questions nécessaires à leur compréhension. Comme ça le client apprend son métier de client et le corrige selon le feedback et pourra définir ces scénarios les plus importants.
Du coté des développeurs, ils doivent estimer l'envergure et la difficulté. Cette estimation comprend la production du code , la documentation , tests unitaires et de recette. Généralement on parle pas en temps mais en point où 1 point= 1 semaine de temps idéal). Il faut penser à lever les inconnues techniques , c'est-à-dire que si il y a trop d'inconnus techniques pour estimer alors on arrête temporairement la séance d'estimation et on fait vite un prototype (spikes) ou que si il y a des points techniques à discuter , il faut le faire entre développeurs.
Si des scénarios sont trop gros pour un certain de travail, le client peut scinder. A l'inverse, si il y a des scénarios trop petits, le client peut les fusionner.
Cette phase se termine lorsque le client aura fini de présenter tous les besoins et que tous les scénarios sont estimés. Plus les besoins sont bien définis mieux l'estimation sera durant le projet.
Dans la partie Engagement, on répartit les scénarios parmi les livraisons à venir pour établir le plan de livraison. Les développeurs annoncent leur vélocité donc le total des estimations des points des scénarios qui ont été implémentés au cours de l'itération précédente. De cette manière le client peut savoir dans quels scénarios, il peut dépenser à chaque itération. Le coach fournit sa propre estimation de la vélocité en faisant le calcul suivant :
#binômes*#semaines/itérations*% réel du temps idéel* #pts par semaine de temps idéal
Donc le client doit choisir ces scénarios de façon à combler les points à une valeur égale à la vélocité de l'équipe en s'assurant que les scénarios les plus importants à ces yeux sont traités au plus tôt.
Dans la partie PILOTAGE, il faut faire des suivis de tests, un burndown chart(montre les efforts , le travail accompli en fonction du temps). En cas d'un défaut , il faut soit écrire un scénario avec une estimation du coût de la réparation (toujours avec une importance maximale) ou soit arrêter les tâches en cours et corriger le problème.
A la fin du cycle, il y a la livraison marquant la fin de la phase de pilotage. Pour rappel, on joue avec le contenu et non le délais. Qui dit fin de travail dit célébration !
Et donc lors d'un nouveau cycle il faut redéfinir les durées effectives de réalisation des scénarios , mettre à jour les durées des scénarios restants si nécessaire et déterminer avec plus de précision la vélocité. Quand est-ce la fin du projet ? Quand le client décidera que les scénarios restants ne valent pas l'investissement.

4 Gestion d'équipe

Une bonne équipe a plus de chance de réussir. Par bonne équipe, on parle pas d'avoir des personnes avec de bonnes compétences individuelles mais d'avoir un bon esprit d'équipe. XP renforce cet esprit d'équipe tout en installant au moins un rôle pour chaque personne.

4.1 Rôles

Il existe plusieurs rôles :

1. Le programmeur ou développeur
2. le client
3. le testeur
4. le tracker
5. le manager
6. le coach

Programmeur

Au sens large , il s'occupe du code, des tests, d'écouter le client et il conçoit...Il fait quasi tout mais au sein d'une équipe les programmeurs expérimentés doivent avoir envie de partager et les moins expérimentés l'envie d'apprendre. Cet idée d'apprentissage est favorisé par XP grâce à la communication. XP donne aussi des responsabilités à chaque programmeur afin de les motiver. Voici la charte des droits du programmeur (voir Figure6).

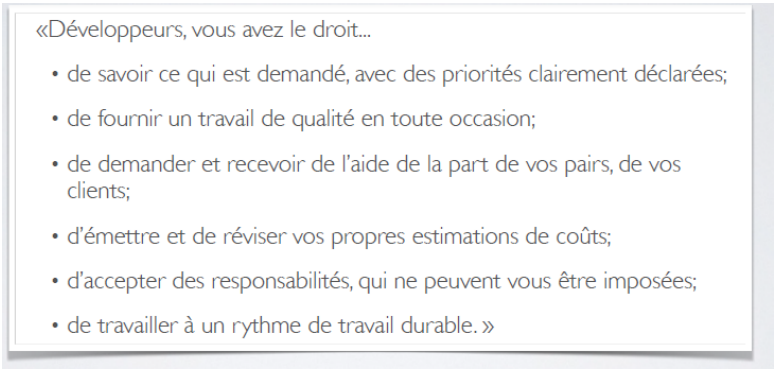

FIGURE 5 – La charte des droits du programmeur

Client

Le client a pour responsabilité de définir ce que doit faire le logiciel , de définir de quelle façon et de communiquer ces informations aux programmeurs. Il a doit définir ses besoins par ses scénarios (voir chapitre précédent). Il faut désigner un client (idéalement 1 personne pour le représenter) mais parfois en cas d'absence le client peut être un client artificiel qui jouera ce rôle mais il faut que la communication entre l'équipe et le client passe par une seule personne ! Voici la charte des droits du client(voir Figure**??**).

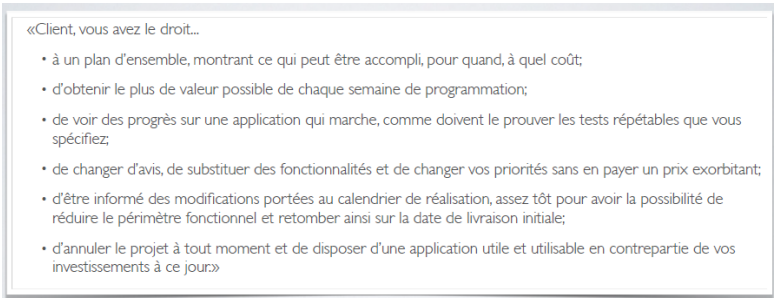

FIGURE 6 – La charte des droits du client

Testeur

Le testeur est chargé d'aider le client à définir les tests de recette(chaque scénario= au moins un test de recette). Au plus tard au milieu d'une itération , les tests sont fournis à l'équipe. Une fois ces tests passés , l'itération est terminée. Cela donne une motivation et un sentiment d'avancement au client et aux programmeurs. Ces tests définissent le besoin du client et vérifient toutes les demandes. Il peut conseilleur ou déconseiller le client/programmeur. Il assure donc la communication avec les programmeurs pour assurer la testabilité automatique du logiciel notamment pour les interfaces graphiques.
On définit un bon testeur, un testeur qui est un capable de combiner des outils , rigoureux , intègre et bon d'esprit d'équipe. (doit chercher les problèmes dans le travail fourni et annoncez les mauvaises nouvelles)

Tracker

Il doit suivre l'évolution des tâches selon le temps qui avait été annoncé et repéré les difficultés. Il faut essayer de pas trop mettre la pression et il faut savoir délier les langues. Si il y a un soucis, il doit en parler au coach.
Un bon tracker est une personne qui ne donne pas l'impression de contrôler , à qui on se confie facilement et de préférence pas un programmeur, pour éviter les discutions techniques.

Manager

Ce n'est pas un programmeur et il fait pas partie de l'équipe. Il est là pour superviser ceux-ci et s'occupe du matériel , recrute et demande des comptes à l'équipe (résultat concrets et visibles). Il doit laisser le processus se dérouler , ne pas exiger des fonctionnalités si les programmeurs émettent un contre avis et doit veiller au respect des engagements et des avantages promis par la méthode. Dans le cas d'un projet inclus dans un plus grand projet , il coordonne le tout.
Un bon manager est souvent le chef de service de l'équipe.

Coach

Il fait partie intégrante de l'équipe. Il vérifie que chacun joue son rôle, que les pratiques XP sont respectées et que l'équipe fait ce qu'elle a dit qu'elle ferait. Il doit donc bien connaître XP. Le but du coach est d'être présent au début pour les séances de planification, les standup meetings, il aide le client à rédiger ses scénarios, travaille avec les programmeurs pour inculquer la méthodologie XP et rassure le manager en expliquant pourquoi la méthode fonctionne. Mais il faut que petit à petit le projet se déroule sans lui permettant ainsi de favoriser la créativité du groupe.
Un bon coach est une personne connaissant bien XP et doit en être absolument convaincu

de l'utilité d'XP. C'est un expert technique, un programmeur chevronné ou un architecte de système, fédérateur et une référence méthodologique et technique ainsi qu'un communicateur , pédagogue et sensible. Il faut qu'il ait du sang froid et qu'il reste calme à tout moment surtout quand les autres paniquent.

4.2 Répartition des rôles

Il est possible qu'une personne puisse avoir plusieurs rôles (voir Figure7).

FIGURE 7 – Une personne pour plusieurs rôles

Dans une équipe , il faut minimum 2 programmeurs , 1 tracker à un moment donné , 1 coach, 1 manager et le client peut être une équipe. Lorsqu'on compare une équipe XP à une équipe classique, on voit que dans le classique c'est généralement le chef de projet qui définit les spécifications , le planning et le budget tandis qu'en XP il n'y a aucune hiérarchie ni aucune séparation des tâches au sein des programmeurs. Personne a la responsabilité d'une partie du projet mais tout le groupe. On consulte un consultant dans le cas d'un problème précis et celui-ci travaille toujours avec 2 programmeurs pour qu'après l'équipe essaie seule de recommencer le travail.(c'est pour respecter XP, c'est à dire qu'on fait le code avec le consultant puis on supprime le code et on le refait à notre sauce mais le but est d'avoir compris la solution)

4.3 Collaboration

XP est fondée sur le travail d'équipe où collaboration et interactions sont permanentes et donc travailler seul devient une exception.
Le code est la propriété de toute l'équipe , chacun doit pouvoir travailler dessus (tester, corriger, optimiser) . Cela permet d'avoir une meilleure qualité et fiabilité et de faciliter la gestion du projet. XP fixe la cohésion de l'équipe comme objectif via les pratiques suivantes :

1. La métaphore
2. La programmation en binôme(pair programming)
3. La responsabilité collective du code(collective code ownership)
4. Les règles de codage(coding standards)
5. L'intégration continue(continuous integration)

4.3.1 Métaphore

Si c'est possible il faut remplacer des aspects compliquées par une image visuelle qui parlera mieux au groupe pour éviter les notions techniques. Les métaphores connus sont la toile d'araignée(net), le bureau avec ses documents, des dossiers(ou classeurs , fichiers), arbres, la souris, la corbeille, la mémoire vive/morte, une passerelle(coupe-feu), client-serveur.

4.3.2 Pair programming

Le pair programming consiste à être 2 sur une machine. Une personne prend les commandes du pc, écrit le code et s'occupe des aspects tactiques de la tâche. Le copilote lui aide le pilote, s'occupe des aspects stratégiques de la tâche et relis immédiatement et continuellement le code pour proposer des solutions d'implémentations ou de design, imaginer de nouveaux tests, penser aux impacts des modifications ...
Il faut établir des règles pour les formations de binômes.

1. Changement fréquent(plusieurs fois par jour) afin de toucher à toutes les parties du projet.
2. Choix volontaire des binômes (choix selon la connaissance du code à travailler, être intéressé par la tâche, choix libre).

Au niveau de la répartition individuelles des tâches

Il n'y a pas de responsabilités partagées des tâches (pas de " Tu es responsable de " tout le groupe est responsable). A chaque instant, un développeur est soit en train de réaliser l'une de ses tâches,soit aider un développeur à réaliser l'une des siennes. On peut répartir un binôme sur plusieurs tâches ou plusieurs binômes sur une tâche , les développeurs sont assez libre dans la manière pour réaliser leur tâche.

(A vérifier)Est-ce que le pair programming est rentable ? A court terme les tâches sont 40% plus rapide mais on perd du temps (A et B font des tâches en parallèle chacun en 1h et dans le cas xP A et B sur une tâche ils le font en 36 min par tâches donc 12 min en +) mais il y a en plus une relecture immédiate des erreurs de syntaxes ou de logique, plus de compétences techniques ou fonctionnelles et stimulation réciproque. Cela donne une qualité supérieure du code car il y a moins de code (plus réfléchi et plus rapide à implémenter). A long terme, un binôme sera plus rapide qu'un programmeur seul mais pas deux fois plus rapide (coût salarial limité) , la conception et le code sont meilleurs via un binôme (réduction des coûts de modification, d'évaluation et de maintenance), la connaissance de l'application est mieux répartie dans l'équipe(accélération du développement et facilitation de l'allocation des tâches) , l'équipe est plus cohésive et motivée et les tâches les plus critiques peuvent être réalisées plus rapidement (livraisons plus rapides).

Au niveau du partage des connaissances

Travailler à deux permet d'avoir une meilleure vision globale du projet , une solution plus propre, si un binôme est coincé l'équipe connaît un minimum le contexte et échange des connaissances techniques et fonctionnelles.

Au niveau de la qualité du code

Il y a de l'amélioration. On a une plus grande clarté(respect des règles de codages, meilleurs noms dans le code), une meilleure conception(plus de connaissance du code donc meilleur refactorisation), moins d'erreurs et meilleur respect des règles de codage(parfait pour le partage).

Au niveau de la satisfaction et cohésion de l'équipe

Tout le code est à toute l'équipe car chaque développeur joue un rôle actif dans sa construction. On est une équipe donc entraide si il y a un binôme en difficulté. Il y a plus de dialogues

et de communications(professionnelle et personnelle).
Travailler en binôme n'est pas facile (habitude de travailler seul, séniors ne veulent pas travailler avec des juniors, personne qui ne s'apprécient pas), il faut apprendre à le faire (présence du coach pour aider). Pour que le travail d'un binôme soit efficace, il faut faire des efforts personnels d'ouverture, faire esprit d'ouverture et d'honnêteté (prend beaucoup plus en considération les qualités humaines). Il faut oser dire les problèmes et pouvoir les accepter (ne pas être susceptible). Il faut savoir accepter de discuter, entendre et oser proposer. En gros il faut savoir accepter ses limites et lacunes des autres.

Que faut-il faire ?

Il faut maintenir un rythme. Les 2 développeurs doivent essayer de rester concentrés (surtout le copilote). Si il n'y a aucune discussion durant plusieurs minutes alors il faut ramener le copilote dans la tâche ou il peut penser tout haut. Si ça ne fonctionne pas, il faut peut-être échanger les rôles.
Il faut penser à gérer les différences de niveaux. c'est mieux de laisser le junior commencer pour que le sénior puisse lui montrer les astuces et raccourcis pour aller plus vite (le guider en permanence de manière à tirer le junior vers le haut).
Il faut aussi exploiter les différents point de vue. Il est utile de confronter les idées pour extraire les avantages et inconvénients et s'en remettre au reste de l'équipe pour trancher en cas d'indécision. Ces débats permettent une cohésion d'équipe(si bonnes qualités humaines). L'efficacité dépend de l'équilibre entre **discussion**(débat contradictoire où les idées de chacun se confrontent. Il y a un gagnant-perdant-ex aequo. C'est une sélection d'idées) et dialogue(conversation pour trouver un terrain d'entente, visant à améliorer les idées des autres. Processus de génération d'idées). Le principe étant de générer des idées et d'en sélectionner qu'une idée. Faire très attention aux débats trop long ou au blocage sur un désaccord. Le coach doit maintenir l'esprit d'équipe en modérant le tout.
Remarque : Utiliser des cartes CRC(classe,responsabilité,collaboration) est une bonne chose. Elles permettent de voir les responsabilités de chaque classe et les collaborations qu'elles ont avec les autres.
Pour l'embauche, il faut bien sûr évaluer la capacité d'intégration des candidats (voir si le courant passe).
Comment bien faire le travail par binôme ?

1. D'abord il faut s'associer occasionnellement (découvrir la pratique et le code) et assigner deux développeurs sur une nouvelle tâche.
2. travailler de plus en plus en binômes en respectant un rythme d'apprentissage.

Au niveau de l'aménagement de l'environnement de travail

Les outils de développement doivent être rapides(modification et exploration du code, compilation et exécution) sinon le copilote décroche et les dépendances minimales entre les modules de l'application(pour une compilation plus rapide après chaque modification).
Dans un espace de travail classique, les développeurs se tournent le dos et chacun sur sa machine alors qu'en XP (voir Figure8, il y a un espace central dédié au développement (1 bureau=2 chaises) , un écran opposé au milieu pour se passer les commandes facilement, les bureaux se font face pour une comminication entre binômes et un bureau pour les réunions, proche des développeurs(pour des interventions plus naturelles des développeurs). Il faut aussi des bureaux pour placer les affaires et pouvoir lire les mails ou téléphoner librement ainsi que des affichages liés à l'avancement du projet tout autour (voir Figure9)

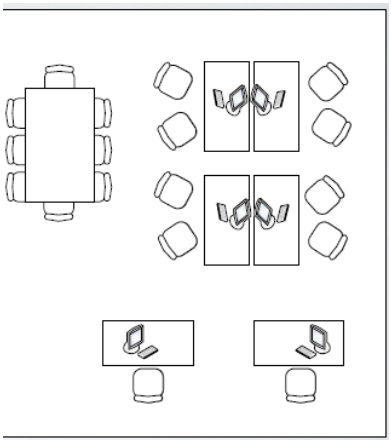

FIGURE 8 – Aménagement de l'environnement de travail xP

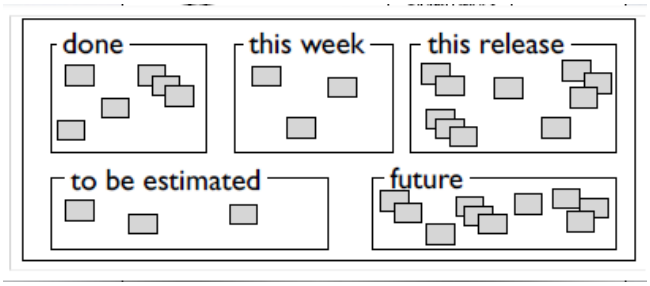

FIGURE 9 – Aménagement de l'environnement de travail :affichage

Bien sûr, ceci n'est qu'à titre indicatif. C'est à l'équipe de voir comment faire ça mais de préférence éviter les obstacles à l'espace de travail XP.

4.3.3 Responsabilité du code

Les modèles actuels : en classique, chaque développeur a sa partie à développer, il faut les accords sur les interfaces à utiliser entre parties ce qui est pratique pour la gestion de la configuration du logiciel et ça permet une autonomie dans la conception et d'implémentation pour les développeurs. Mais responsabilité se change parfois en faute, ce qui est contre-constructif, de plus le qualité est moins bien (duplication du code, fragilité, peu performant, maintenance difficile).

Responsabilité collective du code : En XP, chaque binôme peut intervenir sur n'importe quelle partie de l'application. Chacun est responsable de l'ensemble de l'application et donc si un code n'est pas clair c'est à dire qu'il faut le remanier. Contrairement aux modèles courants, il n'y a personne à blâmer sauf l'équipe. De plus il y a un partage des connaissances et une relecture pas seulement du copilote mais aussi des autres binômes.
Le soucis c'est que chaque développeur ne connaît parfaitement que le code qu'il a écrit. Il faut du temps pour comprendre le reste et il est possible que le code régresse. Mais la conception sera plus simple , le remaniement, les règles de codage et les tests unitaire aussi. Ainsi que mettre plusieurs binômes sur une même partie de l'application peut poser un surcoût de temps de gestion de configuration mais ça permet une intégration continue.
Si il y a des spécialistes, ils doivent essayer de se fondre dans le simple rôle de " développeur" . En cas de question, il peut aider mais il doit pas se prendre comme quelqu'un qui ne fait que des tâches nobles (dans le cas contraire le coach doit intervenir).

4.3.4 Règles de codage

Il est important d'avoir une homogénéisation des styles de codage dans l'équipe. Pour cela, il faut se mettre d'accord dès le début sur un ensemble de règles que chaque développeur devra respecter absolument. Cela peut être difficile pour certains mais c'est assez rapide d'adopter de nouvelles règles et le sentiment d'équipe est renforcé.

Choix d'équipe

L'objectif étant de faciliter la lecture du code, chaque développeur doit intervenir pour définir les règles de codage (permet l'intégration d'équipe). De plus, les règles sont le résultat d'un consensus. Il faut voter avec intervention du coach si nécessaire.

Mise en oeuvre

Ce n'est pas nécessaire d'écrire les règles car chaque membre participe à l'élaboration des règles qu'on verra bien dans le code déjà produit mais il faut expliquer ces règles pour les nouveaux arrivants. Si un binôme s'écarte du style, un autre binôme le verra bien et le corrigera. Ces règles se verront sur les 1ers fichiers mais ces règles pourront encore changer au fil du projet.

Eléments de choix des règles

XP conseille de réduire au maximum le nombre et la complexité des règles tout en garantissant une homogénéité suffisante du code. Les règles se portent principalement sur :

1. la présentation du code(indentation...)
2. Les commentaires (courts et peu nombreux)
3. les règles de nommage(NomDeClass,nomDeMéthode,NOM_DE_CONSTANTE)
4. Système de noms (vocabulaire commun exemple domaine ou métaphores)
5. Idiomes de codage(structures récurrentes exemple for..)

Il est possible de configurer l'IDE pour un formatage automatique.

4.3.5 Intégration continue

L'idée d'une intégration continue est qu'il faut connaître la version opérationnelle à tout moment. Cela permet d'éviter des périodes pour essayer d'intégrer quelque chose de complexe. On peut pour cela utiliser deux outils (outil de gestion et serveur d'intégration).

Les problèmes d'intégration

Si la fusion des modifications fait individuellement ne fonctionnent pas mais séparément oui, que faire ? L'intégration est un tâche compliquée et devrait être une tâche régulière(toutes les 2 semaines par exemple). Plus on met du temps avant d'intégrer plus cela devient difficile d'intégrer car on fusionnera beaucoup plus de codes.
XP dit il faut intégrer très souvent au moins 1 fois par jour voire plusieurs fois par jour ! Si les pratiques d'XP sont respectées, il n'y aura aucun problème.

Rôle des tests unitaires

Les tests unitaires automatisés sont un outil fondamental pour l'intégration continue (feedback direct après chaque intégration et 100% des tests doit passer avant l'intégration et 100% après, avant d'envoyer la release). La compilation et l'exécution des tests doivent être donc très rapides et la couverture des tests doit être la plus grande possible.

Organisation des copies de travail

Il faut faire une branche privée de travail du logiciel par développeur et une branche commune pour toute l'équipe(l'un étant pour le développement et l'autre pour la release). Cette branche est comme un tampon de synchronisation (chaque version sur cette branche aura tous ses tests qui fonctionnent). (voir Figure10)

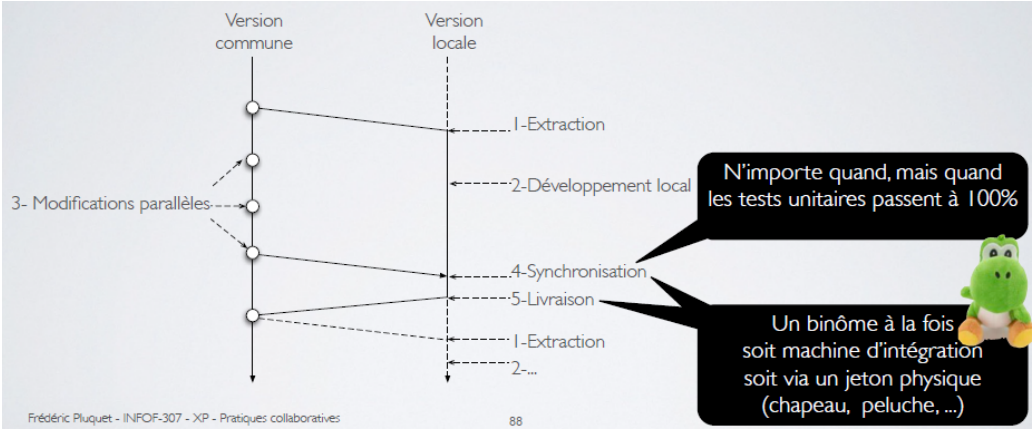

FIGURE 10 – Processus de modification et d'intégration

On peut s'aider d'un serveur d'intégration. Il s'agit d'un programme qui aide à l'intégration continue :

1. Il veille sur un gestionnaire de version de fichiers (svn ,**git**).
2. Le code est compilé et les tests sont exécutés automatiquement.(lors de changements ou à une date/heure précise, le résultat, email...).

4.4 Quelle taille ?

Jusqu'à 30 personnes mais si le projet est trop grand , on peut diviser en sous-projets où XP est appliqué dans chaque projet . Il faut pour que ça fonctionne que tout le monde soit dans le même bureau et les programmeurs en binôme et avoir une responsabilité collective du code(surtout éviter les sous-groupes, se formant assez naturellement).
Pour faire une équipe X il faut au minimum 3 personne : un programmeur , un client et un coach/programmeur.

5 GRASP et Design patterns

5.1 GRASP

GRASP veut dire General Responsibility Assignment Software Patterns.Qu'est-ce que ça implique ?

5.1.1 Haute cohésion

Pour concevoir des objets clairs, gérables et bien concentrés sur leur tâche, il faut maximiser la cohésion.
La cohésion exprime combien les opérations de la classe sont liées entre elles.

5.1.2 Couplage faible

Pour réduire l'impact du changement dans le code, il faut minimiser le couplage.
Le couplage exprime combien les objets sont dépendants les uns des autres.

5.1.3 Expert

Pour décider l'affectation des responsabilités aux classes, il faut choisir la classe qui possède l'information nécessaire pour réaliser la tâche.

5.1.4 Créateur

. Pour décider l'affectation des responsabilités de la création des objets d'une classe A, il faut idéalement une classe qui est composée de la classe A, contient des A, utilise des A et/ou possède les infos d'initialisation pour les objets de A.

5.1.5 Contrôleur

Le premier objet qui reçoit et coordonne une opération système(après le GUI) est soit un objet représentant tout le système soit un objet représentant un use case.

5.1.6 Polymorphisme

Pour gérer les traitements alternatifs basés sur le type, il suffit d'assigner les responsabilités aux classes pour lesquelles le traitement varie.(**Ne pas faire de "if" sur le type de l'objet**)

5.1.7 Fabrique

Pour décider l'affectation des responsabilités lorsque toute solution semble enfreindre la haute cohésion ou le couplage faible, il faut créer une classe artificielle(pas directement liée à un concept-métier) et contenant des responsabilités bien cohésives.

5.2 Design Patterns

Un design pattern décrit une structure récurrente de design.

- Extraction des noms et de l'abstraction de designs concrets.
- Identification des classes, des collaborations et des responsabilités.
- Définition du domaine d'application, les compromis, les conséquences et les problèmes liés aux langages.

Cela permet de

- Montrer des solutions efficientes à des problèmes récurrents dans le développement.
- D'améliorer la qualité du code produit.
- Réutiliser les architectures logicielles.
- Appliquer facilement les connaissances d'experts.

Un design pattern est constitué de 4 éléments.

- **Le nom du pattern** : Définit un vocabulaire de design.
- **La description du problème** : Quand appliquer le pattern et dans quel contexte l'utiliser.

- **La description de la solution** : Les éléments qui font le design(classes,..) et leurs relations, responsabilités et collaborations.
- **Conséquences** : Résultats et trade-offs de la solution.

Il y a 3 types de design patterns.

- **Création** : Il se focalise sur le processus de créations d'objets
- **Structure** : Il se focalise sur la composition de classes pour former de plus larges structures
- **Comportement** : Il se focalise sur les algorithmes et l'attribution des responsabilités entre les objets

Les plus importants qu'on verra dans ce cours sont **Singleton,Observer,Composite et Visitor**.

5.2.1 Singleton(Création)

But

Comment garder une seule instance pour une classe ? Utiliser le singleton pour être sûr qu'une classe n'aura qu'une seule instance et fournir un point d'accès global.

Motivation

La motivation pour laquelle on utilise le singleton est quand il ne peut y avoir qu'une instance et la classe doit contrôler sa seule instance.
Exemple : beaucoup de requêtes à une base de données mais une seule connexion active.

Application

Le singleton est utilisé lorsqu'il ne doit exister qu'une seule instance d'une classe et elle doit être accessible par un point d'accès connu de tous. La seule instance doit pouvoir être extensible par sous-classe et les clients doivent être capables d'utiliser une instance étendue sans changer leur propre code.

Structure

La structure est dans la figure11

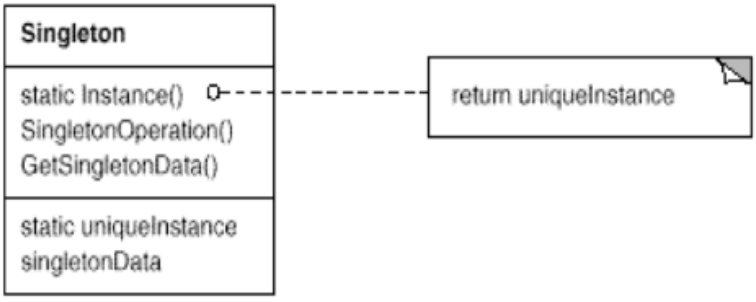

FIGURE 11 – Structure d'un singleton

Participants et collaborations

Le singleton définit une méthode d'instance qui laisse le client accéder à son unique instance. Instance est une méthode de classe qui soit retourne soit crée et retourne la seule instance.

conséquences

- Contrôle d'accès à l'unique instance : comme la classe singleton encapsule sa seule instance, elle peut avoir un contrôle stricte sur comment et quand les clients y accèdent.
- Un namespace réduit : le singleton est une amélioration par rapport aux variables globales qui gardent des instances uniques.
- Raffinement des opérations : comme la classe Singleton peut-être sous classée, une application peut alors être configurée avec une instance d'une classe dont vous avez besoin à l'exécution.
- Un nombre variable d'instances : la même approche peut être utilisée pour contrôler un nombre donné d'instances.
- Plus flexible que des opérations de classes

Exemple

```
public class MazeFactory {
        private static MazeFactory instance = null;
            public static MazeFactory getInstance(){
            if(instance == null){
                    instance = new MazeFactory();
              }
              return instance;
              private MazeFactory(){}
 }
 public class Main {
        public static void main(String[] args) {
        MazeGame gmg = new MazeGame();
        //MazeFactory factory = new MazeFactory();
        MazeFactory factory = MazeFactory.getInstance();
        Maze mz = gmg.createMaze(factory);
        }
}
```

Utilisations connues

A chaque fois que vous voulez une limite de création d'instances supplémentaires après la création de la première. C'est utile pour limiter l'usage de la mémoire lorsque plusieurs instances ne sont pas nécessaires.

5.2.2 Observer(comportement)

But

Comment écouter les changements d'un objet en gardant un couplage faible ? Utiliser un observer pour définir une relation de dépendance entre un et plusieurs objets de sorte que, quand l'objet change d'état, tous ses dépendants sont notifiés et mis à jour automatiquement.

Motivation

Très utile pour différents types d'éléments graphiques affichant les mêmes données d'une application et différentes fenêtres montrant différentes vues sur le même modèle d'une application.

Application

L'observer est utilisé lorsque l'abstraction a deux aspects, un dépendant d'un autre. Encapsuler ces aspects dans des objets séparés vous permet de les modifier et les réutiliser de manière indépendante. Il est aussi utilisé lorsqu'un changement sur un objet demande d'en changer d'autres et que vous ne savez pas combien d'objets ont besoin d'être changés. Il est aussi utilisé lorsqu'un objet doit notifier d'autres objets sans devoir faire d'hypothèse sur ces objets. En d'autres mots, vous ne voulez pas que ces objets aient un couplage fort.

Structure

La structure est dans la figure 12

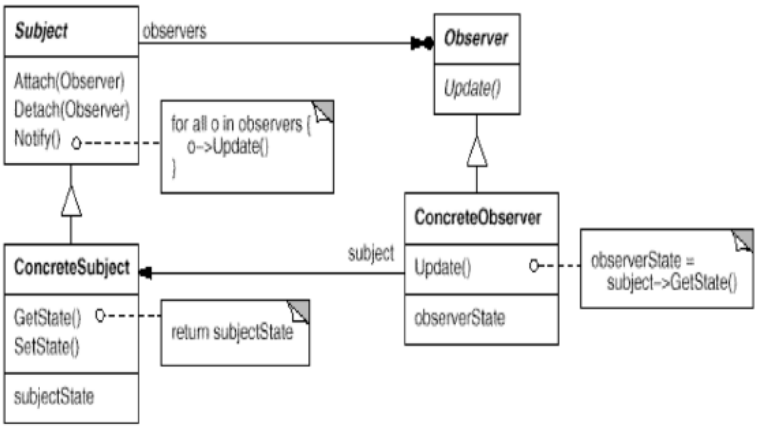

FIGURE 12 – Structure d'un observer

Participants et collaborations

Le subject connaît ses observateurs. N'importe quel nombre d'observateurs peuvent observer un sujet et il fournit une interface pour attacher et détacher les observateurs.
L'observer définit l'interface de mise à jour des objets qui peuvent être notifiés des changements du sujet.
Le ConcreteSubject garde l'état qui intéresse les objets ConcreteObserver et envoie une notification à ses observateurs lorsque son état change.
Le ConcreteObserver maintient une référence vers un ConcreteSubject, garde l'état qui doit être consistent avec celui du sujet et implémente l'interface de mise à jour d'observer.

Conséquences

- Abstraction et couplage faible entre Subject et Observer : le sujet ne doit pas connaître la classe concrète de ses observateurs. Un sujet concret et un observateur concret peuvent être réutilisés indépendamment.
- Support pour une communication "broadcast" : la notification qu'envoie un sujet n'a pas besoin de spécifier un receveur, il sera envoyé à toutes les parties intéressées.

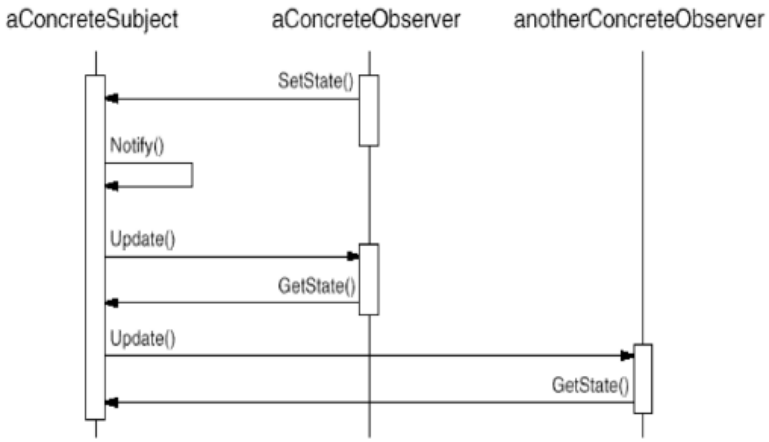

FIGURE 13 – Diagramme séquentiel observer

- Mises à jour inattendues : les observateurs ne sont pas conscients de la présence des autres observateurs. Une petite opération peut entraîner une cascade de mise à jour.

Exemple

```
public class Subject {
        private List observers = new LinkedList();
        protected Subject();
        void attach(Observer o){
                observers.add(o);
        }
        void detach(Observer o){
                observers.remove(o);
        }
        void notifyObservers(){
                ListIterator i = observers.listIterator(0);
                while(i.hasNext()){
                        ((Observer) i.next()).update(this);
                }
        }
}
public abstract class Observer {
        abstract void update(Subject changedSubject);
        protected Observer();
}
public class ClockTimer extends Subject {
        int hour = 0;
        int minutes = 0;
        int seconds = 0;
        int getHour(){
                return hour;
        }
        int getMinutes(){
                return minutes;
        }
        int getSeconds(){
```

```
                return seconds;
        }


void tick(){
//updating the time
        if(seconds == 59){
                if (minutes == 59){
                        if (hour == 23){
                                hour = 0;
                                minutes = 0;
                                seconds = 0;
                        }
                        else {
                                seconds = 0;
                                minutes = 0;
                                hour = hour + 1;
                        }
                }
                else {
                        seconds = 0;
                        minutes = minutes + 1;
                }
        }
        else {
                seconds = seconds + 1;
                notifyObservers();
        }
}
public class ClockDisplay extends Observer{
        private ClockTimer subject;
        private String clocktype;
        ClockDisplay(ClockTimer c, String type){
                subject = c;
                subject.attach(this);
                clocktype = type;
        }
        void update(Subject changedSubject){
                if (changedSubject == subject){
                        displayTime();
                }
        }
        void displayTime(){
                System.out.println(clocktype + "-->" + subject.getHour()+ ":" + subject.
                subject.getSeconds());
        }
}


//if not interrupted, will continue for 24 hours
```

```
public class Main {
        public static void main(String[] args) {
                ClockTimer c = new ClockTimer();
                ClockDisplay d1 = new ClockDisplay(c, "Digital");
                ClockDisplay d2 = new ClockDisplay(c, "Analog");
                int count = 0;
                while (count < (60*60*24)){
                        c.tick();
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                        e.printStackTrace();
                        }
                        count = count + 1;
                }
        }
}
Digital-->0:0:1
Analog-->0:0:1
Digital-->0:0:2
Analog-->0:0:2
```

Utilisations connues

Le design pattern d'observer est utilisé dans le MVC et dès que vous voulez dissocier des classes modèles d'autres types de classes.

5.2.3 Composite(Structure)

But

Comment gérer le tout et les parties de façon transparente ? Utiliser un composite pour composer des objets dans une structure d'arbre pour représenter des relations tout- parties. Ce design pattern permet au client de traiter des objets individuels et les compositions d'objets de manière uniforme.

Motivation

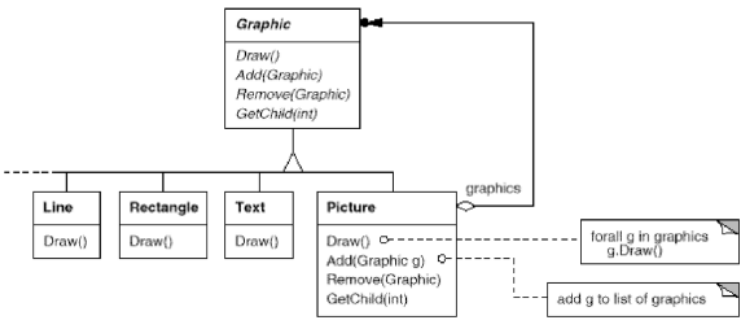

FIGURE 14 – Diagramme composite

Application

Le composite est utilisé lorsqu'on veut représenter des hiérarchies tout-parties ou que les clients soient capables d'ignorer la différence entre des compositions d'objets et des objets individuels. Les clients vont traiter tous les objets dans la structure composite de manière uniforme.

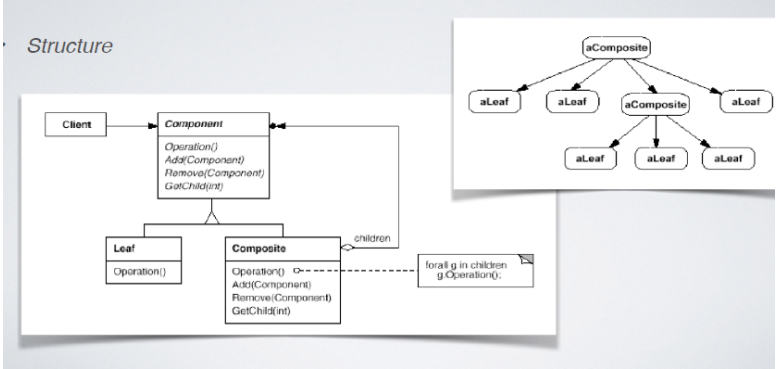

FIGURE 15 – Structure composite

Participants et collaborations

- Composant : il déclare l'interface des objets dans la composition, implémente le comportement par défaut de l'interface commune à toutes les classes et déclare une interface pour accéder et gérer ses "enfants".
- Feuille : il représente les feuilles dans la composition. Une feuille n'a pas d'enfant et définit le comportement pour les objets primitifs de la composition.
- Composite : définit le comportement pour les composants ayant des enfants, stocke les enfants et implémente les opérations sur les enfants déclarées dans composant.
- Client : manipule les objets de la composition à travers l'interface de composant.

Conséquences

- Il définit une hiérarchie de classes avec des objets primitifs et des objets composés
- Permet de rendre le client plus simple : il manipule de manière uniforme les objets primitifs et composés.
- Création simplifiée de nouveaux composants
- Cela peut rendre votre design trop général , ATTENTION !

Exemple

```
abstract class Equipment {
        String name;
        public String name(){
                return name;
        }
        abstract int power();
        abstract int netPrice();
        abstract void add(Equipment e);
        abstract void remove(Equipment e);
```

```
        public Iterator createIterator(){
                return new NullIterator();
        }
        protected Equipment(String n){
        name = n;
        }

abstract class CompositeEquipment extends Equipment {
        ArrayList l;
        public CompositeEquipment(String name){
                super(name);
                l = new ArrayList();
        }
        abstract int power();
        public int netPrice(){
                Iterator i = createIterator();
                int total = 0;
                while(i.hasNext()){
                        Equipment e = (Equipment)i.next();
                        total += e.netPrice();
                }
                return total;
        }
        public void add(Equipment e){
                l.add(e);
        }
        public void remove(Equipment e){
                l.remove(e);
        }
        public Iterator createIterator(){
                return l.listIterator();
        }
}

public class NullIterator implements Iterator {
        public boolean hasNext(){
                return false;
        }
        public Object next() throws NoSuchElementException{
                throw new NoSuchElementException();
        }
        public void remove() throws IllegalStateException{
                throw new IllegalStateException();
        }
}

public class Cabinet extends CompositeEquipment {
        public Cabinet(String name){
                super(name);
        }
```

```
        public int power(){
                return 0;
        }
        public int netPrice(){
                int total = 60 + super.netPrice();
                return total;
        }
}

public class Chassis extends CompositeEquipment {
        public Chassis(String name){
                super(name);
        }
        public int power(){
                return 0;
        }
        public int netPrice(){
                int total = 40 + super.netPrice();
                return total;
        }
}

public class Bus extends CompositeEquipment {
        public Bus (String name){
                super(name);
        }
        public int power(){
                return 40;
        }
        public int netPrice(){
                int total = 100 + super.netPrice();
                return total;
        }
}

public class Card extends Equipment{
        public Card(String name){
                super(name);
        }
        public int power(){
                return 60;
        }
        public int netPrice(){
                return 100;
        }
        public void add(Equipment e){}
        public void remove(Equipment e){}
}

public class BluRayDisk extends Equipment {
```

```
        public BluRayDisk(String name){
                super(name);
        }
        public int power(){
                return 60;
        }
        public int netPrice(){
                return 50;
        }
        public void add(Equipment e){}
        public void remove(Equipment e){}
}

public class Main {
        public static void main(String[] args) {
                Cabinet cabinet = new Cabinet("PC Cabinet");
                Chassis chassis = new Chassis("PC Chassis");
                cabinet.add(chassis);
                Bus bus = new Bus("MCA bus");
                bus.add(new Card("NetworkCard"));
                chassis.add(bus);
                chassis.add(new BluRayDisk("3.5 Floppy"));
                System.out.println("The price is " + cabinet.netPrice());
        }
}
Console:
The price is 350
```

Utilisations connues

Il est très répandu dans les systèmes OO.

5.2.4 Visitor(Comportement)

But

Comment appliquer plusieurs fonctionnalités sur une architecture sans se disperser ? Utiliser le visitor pour représenter une opération à effectuer sur des éléments d'une structure d'objets. Le visitor vous permet de définir une opération sans changer les classes des éléments sur lesquelles elle opère.

Motivation

Application

Le visitor est utilisé lorsqu'il y a une structure d'objets qui contient beaucoup de classes avec différentes interfaces et que vous voulez créer des opérations sur ces objets qui dépendent de leurs classes concrètes.
Il est aussi utilisé lorsqu'il y a beaucoup d'opérations distinctes et non liées qui ont besoin d'être effectuées sur des objets d'une structure et vous ne voulez pas "polluer" leurs classes avec ces opérations.

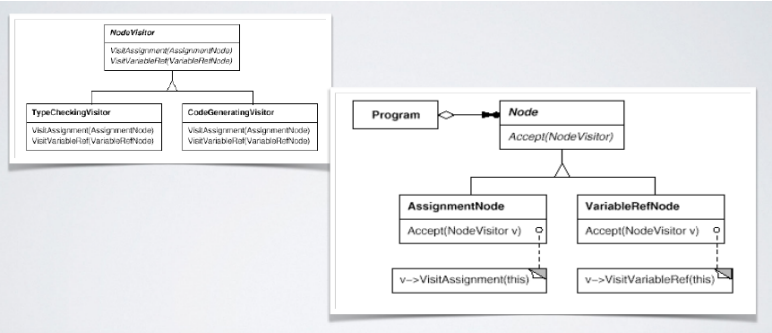

FIGURE 16 – But et Motivation : Visitor

Il est utilisé quand les classes définissant la structure d'objets changent rarement mais que vous voulez définir souvent des nouvelles opérations sur cette structure.

Structure

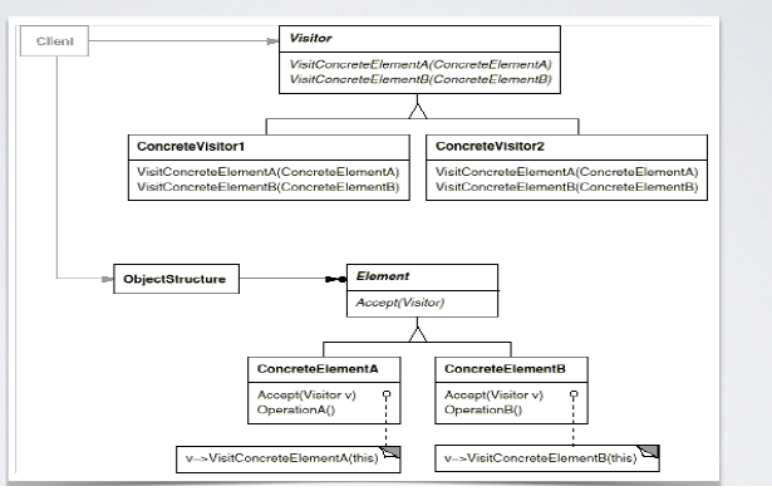

FIGURE 17 – Structure :visitor

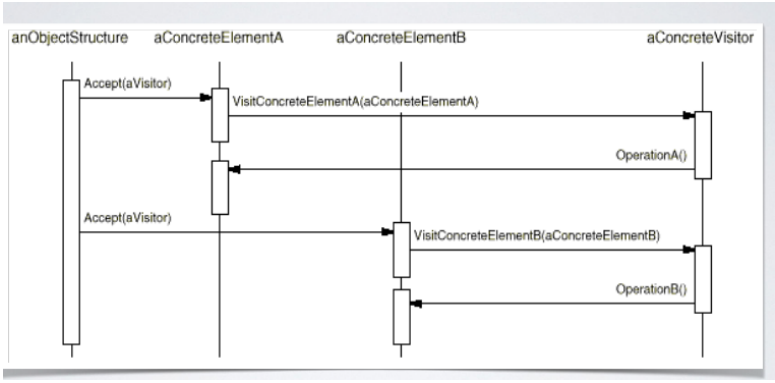

FIGURE 18 – Diagramme séquentiel visitor

Participants et collaborations

- Visitor : Déclare une opération Visit pour chaque classe de ConcreteElement dans la structure et les noms d'opérations et leur signature identifient la classe qui envoie la requête de visite.

- ConcreteVisitor : Implémente chaque opération déclarée par Visitor. Chaque opération implémente une partie de l'algorithme pour la classe correspondante. Il fournit le contexte pour l'algorithme et stocke son état(souvent pour accumuler les résultats).
- Element : définit une opération accept qui prend un Visitor comme argument.
- ConcreteElement : implémente une opération accept qui prend un Visitor comme argument.
- ObjectStructure : il peut énumérer ses éléments. Il peut fournir une interface de haut niveau pour permettre au visiteur de visiter ses éléments et peut soit être un composite ou une collection telle qu'une liste ou un ensemble.

En ce qui concerne les collaborations, un client qui utilise le visitor pattern doit créer un ConcreteVisitor et traverser la structure en visitant chaque élément avec le Visitor et quand un élément est visité il appelle l'opération qui correspond à sa classe sur le visiteur. L'élément est passé lui-même comme argument à cette opération.

Conséquences

— Permet d'ajouter des opérations facilement : une nouvelle opération est définie en ajoutant un nouveau visiteur.
— Rassemble les opérations corrélées et sépare les opérations indépendantes : le comportement lié est localisé dans le visiteur et pas éparpillé parmi les classes définissant la structure.
— Accumulation de l'état : le visitor peut accumuler l'état pendant qu'il parcourt la structure. Sans un visiteur cet état devrait être passé comme un paramètre supplémentaire ou dans des variables globales.
— Encapsulation cassée : l'approche du visitor suppose que l'interface de ConcreteElement est assez fournie pour permettre aux visiteurs de faire leur travail. Le pattern force souvent alors à fournir des opérations publiques qui accèdent à un état interne d'un élément, ce qui peut compromettre son encapsulation.

Exemple

```
abstract class Equipment {
        String name;
        public String name(){
                return name;
        }
        abstract int power();
        abstract int netPrice();
        abstract void add(Equipment e);
        abstract void remove(Equipment e);
        abstract void accept(EquipmentVisitor v);
        protected Equipment(String n){
                name = n;
        }
}

abstract class CompositeEquipment extends Equipment {
        ArrayList l;
        public CompositeEquipment(String name){
                super(name);
```

```
                l = new ArrayList();
        }
        abstract int power();
        //public int netPrice(){
                // Iterator i = createIterator();
                // int total = 0;
                // while(i.hasNext()){
                        // Equipment e = (Equipment)i.next();
                        // total += e.netPrice();
                // }
                // return total;
        //}
        public void add(Equipment e){
                l.add(e);
        }
}

abstract class EquipmentVisitor {
        public abstract void visitBluRayDisk(BluRayDisk b);
        public abstract void visitCard(Card c);
        public abstract void visitChassis(Chassis c);
        public abstract void visitBus(Bus b);
        public abstract void visitCabinet(Cabinet c);
}

class InventoryVisitor extends EquipmentVisitor {
        String inventory = "";
        public void collect(Equipment e) { inventory = e.getName() + "\n" + inventory;}
        public String getInventory(){ return inventory; }
        public void visitBluRayDisk(BluRayDisk b){ collect(b); }
        public void visitCard(Card c) {collect(c);}
        public void visitChassis(Chassis c){collect(b);}
        public void visitBus(Bus b){collect(b);}
        public void visitCabinet(Cabinet c){collect(c);}
}

class PricingVisitor extends EquipmentVisitor {
        int price = 0;
        public int getTotal() { return price;}
        public void collect(Equipment e) { price += e.netPrice();}
        public void visitBluRayDisk(BluRayDisk b){ collect(b); }
        public void visitCard(Card c) {collect(c);}
        public void visitChassis(Chassis c){}
        public void visitBus(Bus b){}
        public void visitCabinet(Cabinet c){}
}

public class BluRayDisk extends Equipment {
        public BluRayDisk(String name){
                super(name);
```

```
        }
        public int power(){
                return 60;
        }
        public int netPrice(){
                return 50;
        }
        public void accept(EquipmentVisitor v){
                v.visitBluRayDisk(this);
        }
        public void add(Equipment e){}
        public void remove(Equipment e){}
}

public class Cabinet extends CompositeEquipment {
        public Cabinet(String name){
                super(name);
        }
        public int power(){
                return 0;
        }
        public int netPrice(){
                return 60;
        }
        public void accept(EquipmentVisitor v){
                Iterator i = createIterator();
                while(i.hasNext()){
                        Equipment e = (Equipment) i.next();
                        e.accept(v);
                }
                v.visitCabinet(this);
        }
}

public class Main {
        public static void main(String[] args) {
                Cabinet cabinet = new Cabinet("PC Cabinet");
                Chassis chassis = new Chassis("PC Chassis");
                cabinet.add(chassis);
                Bus bus = new Bus("MCA bus");
                bus.add(new Card("NetworkCard"));
                chassis.add(bus);
                chassis.add(new BluRayDisk("3.5 Floppy"));
                PricingVisitor p = new PricingVisitor();
                cabinet.accept(p);
                InventoryVisitor i = new InventoryVisitor();
                cabinet.accept(i);
                System.out.println("The computer contains: ");
                i.getInventory();
                System.out.println("The price is " + p.getTotal());
```

```
        }
}

The computer contains:
NetworkCard
MCA bus
3.5 Floppy
PC Chassis
PC Cabinet
The price is 350
```

Utilisations connues

Le visitor est utilisé dans le compilateur de Smalltalk-80 et dans la 3D quand des scènes tridimensionnelles sont représentées comme une hiérarchie de noeuds, le visitor pattern peut être utilisé pour effectuer différentes actions sur ces noeuds.