eXtreme Programming

Stéphane Frénot

INSA Lyon - Département télécommunications Services et Usages

eXtreme programming Explored

William C. Wake
Addison Wesley, 2002

eXtreme programming : les concepts

- Séparation des rôles : client, programmeur, ...
- Couches clairement définies :
  - Programmation : comment écrire du code eXtreme
  - Travail d'équipe : comment organiser les équipes de développement
  - Processus : organisation du processus de développement

eXtreme prog. : développement

- Programmation
  - Conception simple
  - Approche par les tests
  - Refactorisation immédiate
  - Règles de codage

eXtreme programming : l’équipe

- Travail d’équipe
  - Propriété collective du code
  - Intégration continue du code
  - La métaphore
  - Règles de codage
  - 40h / semaines
  - Programmation en tandem
  - Petites livraisons

eXtreme programming : le procesus

- Processus d’organisation
  - Client sur site
  - Approche par les tests
  - Petites livraisons
  - Jeu de prévision

Le développement

le Développement : tester en premier

- Programmation incrémentale
  - petits codes
- Ecrire d’abord les tests
  - Le code est testable car il a été fait pour
  - Les tests sont testés : le test échoue si le code qui implante le test n’existe pas
  - Les tests sont répétables (car il s’agit de code)
  - Les tests aident à la documentation du code
  - Les tests forcent à définir le support minimal
- Existence de bibliothèque de test : JUnit (www.junit.org)
- http://www.xprogramming.com

Utilisation de JUnit

```
import junit.framework.*;
public class TestVector extends TestCase {
  public TestVector(String name){super(name);}
  public void testAddElement(){
    //Mise en place
    Vector v=new Vector();
    //Appels des méthodes
    v.addElement("Une chaine");
    v.addElement("Une autre chaine");
    //Assertion
    assertEquals(2, v.size());
  }
}
```


Cycle de codage eXtreme

- Ecrire un test
- Compiler le test il doit échouer
- Implanter le strict nécessaire
- Lancer le test et voir s'il échoue
- Implanter le minimum pour réussir le test
- Refactoriser pour clareté et duplication
- Recommencer au début
- Maintenir une TODO liste

le Développement : test ?

Une question : Tout code peut il être testé ?
Exemple : difficulté d'automatiser le test d'une IHM

- Simulation de l'interface avec
  `getText, setText, doClick`
- Fabrication d'une version light du modèle

Q&A

- *Combien de temps dure un test ?*
  - De 1 à 5 minutes (10 au max)
- *Que faire si c'est plus long ?*
  - Réduire la taille du test
- *5 minutes ?*
  - Oui
- *L'A/R entre test et code fait une surcharge cognitive*
  - Non.. Vite
- *Ecrire un test ralentit le codage ?*
  - Non pas à terme

Réorganisation

Améliorer la conception du code sans modifier son comportement externe.

- Prérequis
  - code initial
  - tests unitaires
- Il faut :
  - une technique pour améliorer le code
  - un ensemble de réusinage que l'on sait comment appliquer
  - un processus pour organiser le réunisage

Réorganisation : sentir le code

En regardant un classe, on peut "sentir" les réorganisations possibles.http://www.refactoring.com

- Classes trop grandes
- Méthodes trop grandes
- Switch (à la place du polymorphisme)
- Classe de structure (get/set sans méthodes)
- Code dupliqué
- Code dupliqué à un béta près
- Surdépendance de types de base
- Commentaires inutiles

Quand arrêter la refactorisation ?

On arrête la refactorisation quand le programme :

- 1) Réussit les tests
- 2) Communique tout ce qu'il doit
- 3) N'a pas de duplication
- 4) Possède un minimum de classes et de méthodes

==> Once and Only Once

==> DRY : Don't Repeat Yourself (The pragmatic programmmer)

Travail d'équipe

Travail d'équipe

- Propriété du code
- Intégration du code
- Surcharge de travail
- Espace de travail
- Plannification des livraisons
- Standard de codage

Propriété du code

A qui appartient le code ?

- Personne : perdu dans les méandres du temps
- Dernier à l'avoir touché / Le nouveau (marque, chaises musicale / test par le feu)
- Un propriétaire par classe/package (Monogamie) : on ne touche pas à MON code
- Sequential owner (Propriété locative / Serial Monogamy) : problème du long terme
- Propriété par couche (tribalisme) (client/serveur) Réduit la "Bus number" law
- Propriété collective

Propriété collective

Quels sont les problèmes potentiels

- Fierté du code
- Tragedie des communs (tout le monde est responsable, donc personne)
- Expertise
- Mic/Mac de code (mélange de style)
- Marcher sur les "pieds" de l'autre (attend la suite)

Propriété collective : Xp solutions

Quels sont les problèmes potentiels

- Fierté du code : équipe Xp
- Tragedie des communs : tandem + test
- Expertise : deep expertise
- Mic/Mac de code : standard de code
- Marcher sur les "pieds" de l'autre : Intégration continue

Quelques règles de fonctionnement 1

Quand intégrer le code ?

- Juste avant la livraison : Spécial WE difficiles
- 1 fois par jour : le soir (et si ca ne marche pas...)
- Intégration en continue : Approche XP

Comment travailler ?

- Finir coûte que coûte (death march)
- 40 heures / semaines (+5 heures de veille techno)

Quelques règles de fonctionnement 2

Comment organiser l'espace de travail ?

- Séparation géographique (Réunion trimestrielle)
- En office (1 à 2 personnes)
- Cubicle
- Open Workspace

Quelques règles de fonctionnement 3

Quels codes écrire ?

```
for(;;){}                    | while(true){}
-----------------------------+-----------------------------
if(i==j)                     | if(i==j){
{                            |   System.out.println();
  System.out.println();      | }
}                            |
-----------------------------+-----------------------------
if(i==j)                     | if(i==j){
  System.out.println();      |   System.out.println();
                             | }
```


Quelques règles de fonctionnement 3

Quels standard choisir

- Aucun : ARRRGH
- Choix du programmeur (création) et s'impose à l'ajout
- Choix du programmeur (création) et se plie à l'ajout
- Choix d'équipe : exemple microsoft, javabeans ...

JavaBeans :

- 4 espaces d'indentation
- Ouverture des accolades sur la même ligne
- Convention de nommage des javabeans

Développement en Tandem

Comment organiser le développement

- Solo : Codage tout seul dans un coin
- Equipe entière : Tout le monde participe
- Désengagement : Le partenaire regarde l'air distrait
- Programmation par paire
  - changement régulier de clavier
  - changement régulier de partenaire
  - propriétaire de la tâche (un des deux)
  - difficulté de trouver des paires
  - junior/senior
  - 1/2 productivité

Tandem : une analyse

- Le partenaire demande de l'aide au moment de la formation de l'équipe
- Le partenaire aide aussi bien globalement que techniquement
- Il existe un protocole d'échange de clavier
- La paire apprend ensemble; l'apprentissage est dans le processus
- Le partenaire offre une vision externe de qualité
- Le partenaire fournit l'autorisation (l'excuse) de faire une chose
- Si un partenaire oubli quelque chose, l'autre le remet à l'ordre

Y a t'il une "architecture" ?

- Approche orientée architecture
  - *"Certaines choses sont difficiles à changer, donc il faut concevoir un squelette avant"*
- Approche orientée Xp
  - *"Accepter d'adapter au changement (merci l'objet)"*

L'Architecture Xp

- Principe d'exploration
- Principe de la métaphore
- La première itération (squelette)
  - Faire la surface du programme (hello, world)
  - Tout configurer
  - Supprimer les fonctionnalités
  - Interface minimaliste
- Petites livraisons
- Refactorisation
- Pratique d'équipe

Q&A

- Qui définit l'architecture de déploiement ?
  - L'équipe au moment de l'exploration
- Comment est documentée l'architecture ?
  - Test, plus ce que veut le client... (coût)
- La version de surface (ou résoudre la persistance ?)
  - Permet au client de commencer à tester
- Si tout le monde est architecte alors personne ?
  - Notion de coach
- Donc on utilise un architecte ?
  - Non, il est plus important de l'intégrer dans Xp
- Où ca va planter ?
  - MCD et MultiThread

Principe de la métaphore

Pourquoi chercher une métaphore ?

- Vision commune
- Partage du vocabulaire
- Créativité
- Identifie une architecture

Quelques exemples de métaphores

- Le bureau pour les interfaces utilisateurs
- Ligne de montage pour la gestion de flux
- Caddie électronique pour le e-commerce
- L'annuaire papier pour l'annuaire
- Le courrier papier pour la messagerie

Utilisation de la métaphore

- Choisie pendant la phase d’exploration
- Remise en question au fil de l’eau
- Elle guide la solution
- Utiliser ses concepts pour les classes principales

Limites de la métaphore

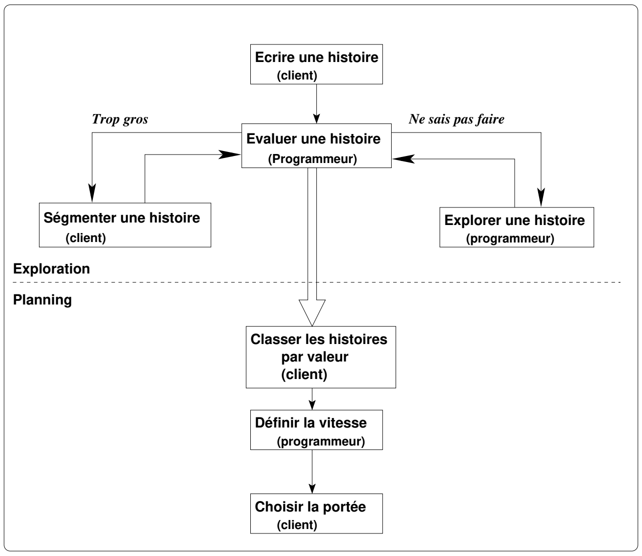
- Non familière à tout le monde
- La métaphore peut être trop faible
- La métaphore peut être trop forte
- La métaphore peut borner la conception
- Attention au mot "magique"

Les processus

Processus

- Plannification de release (sorties)
- Organisation d'une itération
- Les acteurs

Plannification d'une release



Release : Exploration

- But :
  - comprendre le but du système afin de pouvoir estimer la charge
- Méthode :
  - le client écrit une histoire, le programmeur l'évalue
- Résultat :
  - ensemble d'histoires évaluées
- Durée :
  - jour ==> Semaines

Les rôles

- Client écrit une histoire : testable, 1 à 3 "semaines"
- Programmeur évalue l'histoire
- Programmeur fait une exploration (étayer)
- Client découpe une histoire pour simplifier
- A la fin : chaque histoire à un coût

==> Notion de semaine "idéale"

Release : Planning

- But :
  - Plannifier la prochaine livraison
- Entrées :
  - Des histoires
- Méthode :
  - Réaliser les 3 actions ci-dessous
- Résultat :
  - Une liste triée par priorité des histoires livrées la prochaine fois
- Durée :
  - Quelques heures

Planning : comment

- Client trie les histoires par valeur (haute, moyen, basse)(doit, peut,pourrait)
- Programmeur déclare la vitesse
  - nombre de story point / itération
  - 1ère itération = 1 à 3 semaines
  - estimation : 1/3 d'histoire / programmeur / s.
  - après : la règle du "temps d'hier"
- Client sélection la portée
  - Le client sélectionne les histoires
  - La durée = nb points / vitesse
  - La première : L'ensemble du système end<->end

Q&A

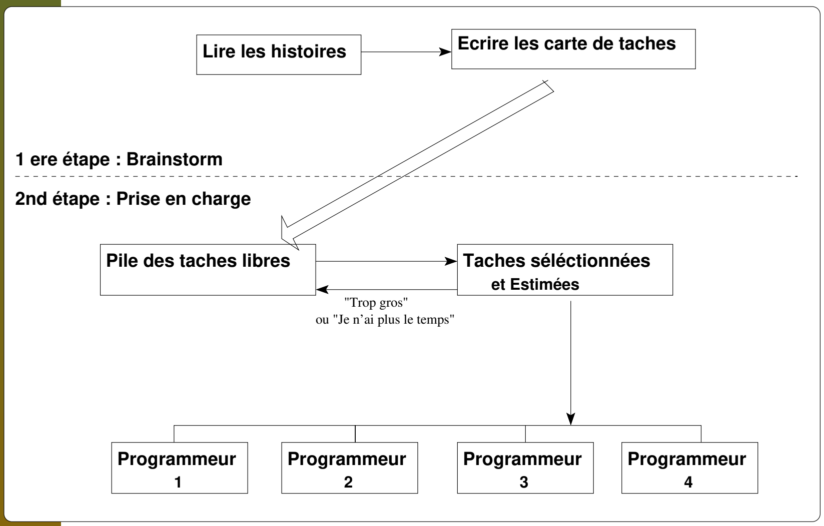
- Si le client n'aime pas le résultat, il peut :
  - changer les histoires, livrer moins d'histoires, accepter une nouvelle date, changer une partie de l'équipe, quitter le projet
  - il ne peut pas changer les estimations
- Le client est il lié aux priorités fixées ?
  - Non, c'est un point de départ
- Peut on trier par risque ?
  - Why not, mais on peut réduire les risques en augmentant la taille des histoires

"Software developpement as Cooperative Game"
http://members.aol.com/humansandt/papers/asgame/

Exemple : Vitesse 5 (5 / 3 semaines)

high	Z39.50 (3)	3	Z39.50
	Query (2)	2	Query
	MARC (2)	===	===
	GUI (2)	2	MARC
	Performance (1)	2	GUI
Medium	Boolean (3)	1	Performance
	Sorting (2)	===	===
	Drill-Down (2)	1	SUTRS
	SUTRS (2)	2	Config
	Config (2)	1	Portable
	Portable (1)		
low	Print (2)		
	No-Drill (1)		
	Save Result (1)		
	Save Query (1)		

Plannification d’une release



Mise en place du jeu

- Le suiveur (trackeur) indique le nombre d’histoires réalisée la fois précédente
- Le client sélectionne les tâches inacomplies < limite des points
- Le tracker donne pour chaque programmeur ses points de programmation (1ère : 1/2 /program/jours de l’iteration)
- Chaque programmeur reçoit son nombre de points (yesterday’s rule)

Préparation d'une itération

Phase 1 : Brainstorm d'équipe

- Le client sélectionne une des histoires et la lit
- Les programmeurs définissent les tâches
- Le processus est répété pour chaque histoire

Phase 2 : Les programmeurs prennent en charge

- Pour chaque tâche un programmeur :
  - la séléctionne
  - Estime le temps qu'il passe dessus (en points programme)
  - Ecrit ce temps sur la carte
- Si la tâche est trop grosse (> 3 jours)==>split de la tâche

Fin d'itération

- Pas de solution
  - Le client choisit une histoire à décomposer ou à reporter
- L'équipe a besoin de plus d'histoires
- L'équipe gagne : Tout le monde est à peu prèt à 0

A la fin l'équipe à un plan d'itération des histoires

- Le gestionnaire enregistre le plan
- Le client écrit les tests de validation
- Les programmeurs bossent !

Q& A

- Durée : Quelques heures
- Pourquoi des tâches et pas des histoires ? plus simple à estimer
- Chaque programmeur choisi sa tâche ==> pas de guru
- Qui possède la tâche programmeur / tandem ?
- Pourquoi ce n'est pas l'équipe qui évalue les tâches ?
  Implication du programmeur
- Somme des tâches == histoire ? Les tests valident l'histoire
- Estimation ? le passé + en faisant des tests de 5 - 60 minutes
- Estimation en fonction du partenaire ? oui

Les rôles

- Le client
- Le programmeur
- Les gestionnaires
  - Le coach
  - Le trackeur
  - Le mentor

Le client

- Client : 4 rôles pendant l'itération + 1 pendant la recette
  - 1) Répondre aux questions
  - 2) Ecrire les tests
  - 3) Lancer les tests
  - 4) Piloter l'itération
    - Sélectionne le contenu d'une release
    - Plannifie les itérations
    - Prend des décisions au vol
  - 5) Valide la livraison (c'est la fête)

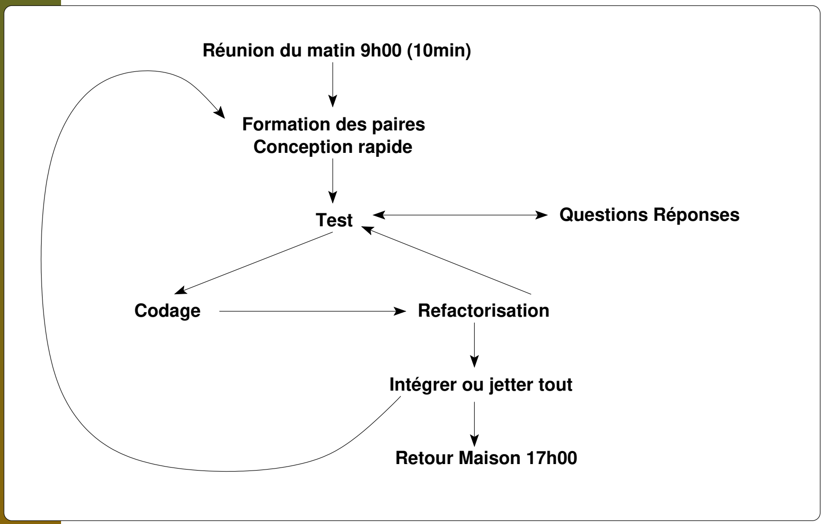
Q& A

Les problèmes du client

- J'ai un boulot à côté :
  - il faut dégager du temps
- Je ne peux pas indiquer aux analystes... :
  - non, il n'y en a pas
- Que faire s'il y a plein de clients :
  - il faut sérialiser les clients
  - Le système est fondé sur un "flux" client
- Mais vraiment plein, plein de clients :
  - il faut s'adapter

Les programmeurs

Teste, code, refactorise (!= conception, code, test)



Le manager

- Représentant du projet au monde extérieur
- Forme l'équipe
- Obtient les ressources
- Pilote l'équipe
  - Présente les avancements
  - Convoque les réunions
  - Organise les célébrations
- Gère les problèmes (client, membres...)

Le manager ne donne

- Pas de priorités (le client)
- Pas d'assignation de tâches (le programmeur)
- Pas d'estimations (programmeur)
- Pas de calendriers (négociation client/programmeur)

==> Masque aux programmeurs les problèmes

Le tracker et le coach

Le tracker

- Suit le planning de livraison (histoires)
- Suit le planning d'iteration (tâches)
- Suit l'état de validation des tests

Le coach

- Attributs : droit, respecté et respecte, sur-site, souvent excellent programmeur
- Activités : surveille les processus (réunions qui débordent, etc...), maintient les règles, modifie le processus, mentor, fournit les distractions

conclusions

Les problèmes à surveiller

- Ralentissement de la vitesse (refactoring, test ?)
- Mauvais code ou usine à gaz (refactoring)
- Baisse de la qualité (tests)

conclusion

Quels sont les points forts d'XP ?

- Communication rapide et efficace (client, programmeur)
- Validation primordiale (réduction du temps, car tout est validé)
- Le principe d'itération fonctionne (itération 1 à 3 semaines, release qq mois)

Quels sont les points à améliorer ?

- Simplification et éclaircissement (processus de plannification)
- Limites d'Xp : Grosse équipe ? Valide dans d'autres domaines ? Quand faire du Xp ?
- Trouver des concepts fondateurs
- Tests d'acceptation : valide dans d'autres contextes

Bibliographie

1. Hunt, Andrew and D. Thomas. 2000 – The pragmatic programmeur : From Journeyman to Master. Boston : Addison-Wesley
2. . 2000 – Programming Pearls, Second Edition. Boston : Addison-Wesley
3. 2000-2001 – xp series – Addison-Wesley

Sites web

1. http://www.egroups.com/group/extremeprogramming
2. http://www.extremeprogramming.org
3. http://www.junit.org
4. http://www.pairprogramming.com
5. http://www.refactoring.com
6. http://www.xpdevelopper.com
7. http://www.xprogramming.com
8. http://www.xp123.com