

# Les 12 pratiques de l'Extreme Programming

## Pratiques de feedback fin

Ces pratiques visent à obtenir un retour d'information **très rapidement** (de quelques secondes à quelques jours) plutôt que d'attendre des semaines ou des mois.

### 1. Programmation en binôme (Pair Programming)

Deux développeurs travaillent ensemble sur le même ordinateur. L'un écrit le code (le "driver"), l'autre observe et réfléchit à la stratégie (le "navigator"). Les rôles s'échangent régulièrement.

#### Bénéfices :

- Meilleure qualité du code
- Partage des connaissances
- Réduction des erreurs

Le Pair Programming contribue aussi fortement à la **compréhension partagée** du système. </ ==> 2. Jeu de planification (Planning Game) ==> Processus de planification collaboratif entre l'équipe technique et le client. Se divise en deux parties : \* **Release Planning** : planification à long terme \* **Iteration Planning** : planification détaillée pour l'itération en cours **Principes** : \* Le client définit les priorités \* Les développeurs estiment la complexité \* Décisions basées sur la valeur métier ==> 3. Développement piloté par les tests (Test-Driven Development - TDD) ==> Les tests unitaires sont écrits **avant** le code de production. **Cycle TDD** : - Écrire un test qui échoue (Red) - Écrire le code minimal pour passer le test (Green) - Améliorer le code (Refactor) ==> 4. Client sur site (On-Site Customer) ==> Un représentant du client est présent à temps plein avec l'équipe de développement. **Rôle du client** : \* Répondre aux questions immédiatement \* Écrire les user stories \* Définir les tests d'acceptation \* Prioriser les fonctionnalités ==> Pratiques de processus continu ==> ==> 5. Intégration continue (Continuous Integration) ==> Le code est intégré et testé plusieurs fois par jour dans le dépôt principal. **Règles** : \* Intégrer au minimum une fois par jour \* Tous les tests doivent passer à 100% \* Si les tests échouent, corriger immédiatement ==> 6. Refactoring ==> Amélioration continue de la structure du code sans modifier son comportement. **Objectifs** : \* Éliminer la duplication \* Améliorer la lisibilité \* Simplifier la conception \* Faciliter les évolutions futures **Principe** : Le refactoring est une activité continue, pas une phase distincte. ==> 7. Petites releases (Small Releases) ==> Livrer fréquemment des versions fonctionnelles du logiciel en production. **Avantages** : \* Feedback rapide du client \* Réduction des risques \* Valeur délivrée rapidement \* Apprentissage continu **Fréquence typique** : De quelques semaines à quelques mois maximum ==> Pratiques de compréhension partagée ==> ==> 8. Conception simple (Simple Design) ==> Le code doit être aussi simple que possible, tout en rempliesant les besoins actuels. **Critères d'une conception simple (par ordre de priorité)** : - Passe tous les tests - Révèle l'intention - Pas de duplication - Minimum d'éléments (classes, méthodes, etc.) Ne pas anticiper les besoins futurs (YAGNI - You Aren't Gonna Need It) ==> 9. Métaphore système (System Metaphor) ==> Une histoire simple et partagée qui décrit comment fonctionne le système. **Objectif** : \* Vocabulaire commun entre tous les acteurs \* Guide pour l'architecture \* Facilite la communication **Exemple** : "Le système fonctionne comme une chaîne de montage" ou "C'est comme un bureau de poste" ==> 10. Propriété collective du code (Collective Code Ownership) ==> Tout le monde est responsable de tout le code. N'importe quel développeur peut modifier n'importe quelle partie du système. **Conditions nécessaires** : \* Standards de codage respectés \* Tests unitaires exhaustifs \* Intégration continue **Bénéfices** : \* Pas de goulot d'étranglement \* Meilleure connaissance du système \* Flexibilité des affectations ==> 11. Standards de codage (Coding Standards) ==> L'équipe suit des conventions de codage communes et cohérentes. **Éléments couverts** : \* Formatage du code \* Conventions de nommage \* Structures de fichiers \* Commentaires **Objectif** : Le code doit sembler écrit par une seule personne. ==> Pratiques de bien-être du programmeur ==> ==> 12. Rythme soutenable (Sustainable Pace) ==> L'équipe travaille à un rythme qui peut être maintenu indéfiniment, typiquement 40 heures par semaine. **Règle** : Pas plus d'une semaine

d'heures supplémentaires consécutive. **Raisons :** \* La fatigue génère des erreurs \* La créativité diminue \* Le turnover augmente \* La productivité à long terme baisse Cette pratique était initialement appelée "40-hour week" mais a été renommée pour être plus universelle. ===== Relations entre les pratiques ===== Les pratiques XP se soutiennent mutuellement : \* **TDD** rend possible la **propriété collective** (confiance dans le code) \* **Programmation en binôme** facilite le **refactoring** (deux cerveaux pour améliorer) \* **Intégration continue** s'appuie sur les **tests** (validation automatique) \* **Client sur site** nourrit le **jeu de planification** (feedback immédiat) \* **Rythme soutenable** améliore la **conception simple** (esprit clair) XP recommande d'adopter **toutes** les pratiques ensemble. En retirer une affaiblit l'ensemble du système. ===== Sources ===== \* Kent Beck - "Extreme Programming Explained: Embrace Change" (1999, 2nd edition 2004) \* [extremeprogramming.org](http://extremeprogramming.org) - Site de référence un peu pourri

From:

<http://slamwiki2.kobject.net/> - **SlamWiki 2.1**

Permanent link:

<http://slamwiki2.kobject.net/eadl/bloc3/xp/chap2?rev=1764056064>

Last update: **2025/11/25 08:34**

