Présentation XP

Introduction

Les piliers de l'eXtreme Programming (XP) reposent sur des mécanismes clés pour :

- Optimiser l'efficacité des équipes (pair programming, intégration continue, etc.),
- Raccourcir les boucles de feedback (livraisons fréquentes, tests automatisés, rétrospectives),
- Fluidifier les releases (déploiement continu, petites itérations incrémentales),
- Renforcer la collaboration client (user stories, planification adaptative, démonstrations régulières).

La maîtrise de XP passe par une compréhension hiérarchisée :

- Ses Valeurs (communication, simplicité, feedback, courage, respect) → fondations culturelles.
- Ses Principes (ex : "Embrasser le changement", "Livrer de la valeur rapidement") → cadrage stratégique pour choisir/adapter les pratiques.
- Ses Pratiques (TDD, refactoring, CI/CD, velocity tracking, etc.) → outils concrets pour implémenter les principes.

Adaptation contextuelle :

XP n'est pas un framework rigide. Certaines pratiques (ex : pair programming) peuvent être ajustées ou combinées (avec Scrum/Kanban) selon :

- La taille de l'équipe (startup vs. scale-up),
- La criticité du projet (MVP vs. système embarqué),
- La maturité technique (legacy code vs. greenfield).

XP Agile DevOps TDD Clean code

Valeurs XP

Les **valeurs XP (eXtreme Programming)** définissent la culture et les comportements clés pour des équipes **agiles et techniques**.

Tableau des Valeurs XP

Valeur	Définition Technique
Communication	**Échanges structurés** pour :
	- *Knowledge sharing* : **mob programming**, docs collaboratives (ex : Confluence/Notion).
	- *Résolution de problèmes* : **stand-ups techniques**, canaux dédiés (Slack/Teams).
	- *Coordination* : **planning poker**, raffinement de backlog.
Simplicité	**Approche minimaliste** :
	- *YAGNI* : **"You Aren't Gonna Need It"** → pas de sur-ingénierie.
	- *DTSTTCPW* : **"Do The Simplest Thing That Could Possibly Work"**.
	- *Refactoring* : **incrémental** (ex : *boy scout rule*).
Feedback	**Boucles courtes et automatisées** :
	- *Tests* : **TDD/BDD** (feedback en ms).
	- *Démos* : **Sprint reviews** (validation client en jours).
	- *Rétros* : **Amélioration continue** des processus.

Valeur	Définition Technique
Courage	**Décisions techniques audacieuses** :
	- Remettre en question les choix (ex : “Ce framework est-il adapté ?”).
	- *Fail fast* : **Spikes** pour valider des hypothèses avant le dev.
	- *Blameless culture* : **Postmortems** sans jugement.
Respect	**Collaboration bienveillante** :
	- *Pair programming* : **Montée en compétence** (juniors/seniors).
	- *Code reviews* : **Focus sur l’amélioration**, pas la critique.
	- *Retros actionables* : **Écoute active** des feedbacks.

Principes XP

Les **principes XP** servent de **pont entre les valeurs et les pratiques**. Ils expliquent **pourquoi** certaines pratiques (comme le TDD ou le pair programming) sont efficaces, et comment les adapter à différents contextes techniques.

Principes XP

1. Humanity

Pourquoi ? Des développeurs épanouis = code de meilleure qualité.

Aspect	Définition	Impact Concret	Outils/Pratiques
Sécurité psychologique	Environnement où on ose poser des questions sans jugement.	➔ -30% de *knowledge loss* (turnover réduit).	- **1:1s réguliers** (template structuré).
Autonomie	Liberté de choisir comment résoudre un problème technique.	➔ +25% de productivité (moins de micro-management).	- **Objectifs SMART** (OKRs techniques).
Reconnaissance	Visibilité des contributions techniques (ex : refactor, tests).	➔ +40% d'engagement (enquêtes internes).	- **Kudos Slack** / **Bonus techniques**.

2. Economics

Pourquoi ? Toute décision technique doit se justifier par sa **valeur business**.

Critère	Exemple Technique	Bénéfice Business	Outils
Coût vs. Valeur	Migration vers Kubernetes : 3 sprints vs. gain de scalabilité.	➔ ROI calculé : économie de 20% sur les coûts cloud.	- **Business Case Template**.
Dette Technique	Refactor d'un module legacy (5j) vs. risque de *firefighting*.	➔ Évite 10j de hotfixes/an.	- **SonarQube** (coût de la dette).
Priorisation	Feature A (valeur client élevée) vs. Feature B (technically cool).	➔ Focus sur ce qui génère du revenue.	- **User Story Mapping** (Miro).

3. Mutual Benefit

Pourquoi ? Les pratiques XP doivent avantage **à la fois les devs et le business**.

Pratique	Bénéfice Devs	Bénéfice Business	Exemple
TDD	Code plus simple à maintenir.	➔ -40% de bugs en production.	- **JUnit** / **pytest**.
CI/CD	Feedback immédiat sur les changements.	➔ Livraisons 5x plus fréquentes.	- **GitHub Actions**.
Clean Code	Onboarding des nouveaux plus rapide.	➔ -30% de temps passé en reviews.	- **ESLint** / **Prettier**.

4. Self-Similarity

Pourquoi ? Réutiliser des solutions éprouvées, mais **adapter au contexte**.

Contexte	Solution Réutilisable	Adaptation Nécessaire	Outil
Microservices	Pattern *Strangler Fig* pour découper un monolithe.	Vérifier la compatibilité avec le legacy.	- **ADR (Architecture Decision Records)**.
Tests	Structure *Given/When/Then* pour les tests.	Adapter aux spécificités métiers.	- **Cucumber** / **SpecFlow**.
Code Reviews	Checklist standardisée.	Ajouter des critères projet-spécifiques.	- **GitHub PR Templates**.

5. Improvement (Kaizen)

Pourquoi ? L'excellence vient de **petites améliorations continues**.

Niveau	Action	Impact	Metric
Code	+1% de couverture de tests par sprint.	➔ -15% de régressions.	- **SonarQube**.
Processus	Réduire le *lead time* de 10%.	➔ Livraisons plus prévisibles.	- **DORA Metrics**.
Équipe	Rétrospective hebdomadaire.	➔ +20% de satisfaction d'équipe.	- **Officevibe**.

6. Diversity

Pourquoi ? Des équipes diversifiées = **meilleures solutions techniques**.

Type de Diversité	Exemple	Bénéfice Technique	Pratique
Compétences	Backend + DevOps dans la même équipe.	➔ Solutions full-stack optimisées.	- **Pair Programming rotatif**.
Expérience	Juniors + Seniors sur un spike technique.	➔ Innovation + rigueur.	- **Mentorat inverse**.
Perspectives	Brainstorming avec des non-techniques.	➔ UX/UI plus réalistes.	- **Workshops cross-fonctionnels**.

7. Reflection

Pourquoi ? Apprendre de chaque action pour **s'améliorer**.

Type	Format	Exemple	Outil
Rétrospective	Mad/Sad/Glad.	“Pourquoi ce bug a passé les tests ?”	- **Retrium**.
Postmortem	Blameless (sans culpabilité).	Analyse d'un incident de prod.	- **Google Docs Template**.
Code Review	Feedback structuré.	“Pourquoi ce PR a pris 3 jours ?”	- **GitHub PR Comments**.

8. Flow

Pourquoi ? Un flux de travail fluide = **livraisons plus rapides**.

Obstacle	Solution	Impact	Outil
Blocages	WIP Limits (max 2 tâches en cours).	➔ -50% de *context switching*.	- **Kanban Board**.
Dépendances	User stories plus petites.	➔ Livraisons incrémentales.	- **Story Splitting Techniques**.
Feedback Lent	CI/CD avec tests automatisés.	➔ Feedback en <10 min.	- **GitHub Actions**.

9. Opportunity

Pourquoi ? Transformer les problèmes en **opportunités d'apprentissage**.

Problème	Opportunité	Action	Exemple
Bug Critique	Améliorer la suite de tests.	Ajouter des tests E2E.	- **Cypress**.
Dette Technique	Moderniser le code.	Spike pour évaluer les options.	- **Tech Radar**.
Conflit d'équipe	Renforcer la collaboration.	Atelier de team building technique.	- **Mob Programming**.

10. Redundancy

Pourquoi ? Certaines redondances sont **utiles**, pas du *waste*.

Type	Exemple	Bénéfice	Outil
Tests	Tests unitaires + intégration pour un module critique.	➔ Couverture à 95%.	- **Jest** / **Pytest**.
Documentation	README + docs dans le code + wiki.	➔ Onboarding en 1j vs 1 semaine.	- **Markdown** / **Confluence**.
Reviews	2 approvers pour les PRs critiques.	➔ 0 bug en prod sur 6 mois.	- **GitHub Protected Branches**.

11. Failure

Pourquoi ? L'échec est une **source d'apprentissage**, pas une honte.

Type d'Échec	Leçon Apprise	Action Corrective	Outil
Déploiement Raté	Pipeline CI/CD trop lent.	Optimiser les étapes de build.	- **GitHub Actions Cache**.
Bug en Prod	Tests E2E manquants.	Ajouter des tests de non-régression.	- **Selenium**.
Estimation Fausse	User story mal découpée.	Utiliser le *Story Splitting*.	- **Planning Poker**.

12. Quality

Pourquoi ? La qualité n'est **pas négociable** - c'est un multiplicateur.

Pratique	Critère	Impact	Outil
Clean Code	Fonctions <20 lignes, noms explicites.	➔ -40% de temps en reviews.	- **ESLint** / **SonarQube**.
TDD	100% de couverture pour le code critique.	➔ 0 régression sur les features core.	- **JUnit** / **pytest**.
Definition of Done	Checklist stricte (tests, docs, reviews).	➔ Livraisons prévisibles.	- **Jira DoD**.

13. Baby Steps

Pourquoi ? Des petites étapes = **moins de risques**, plus de succès.

Contexte	Action	Bénéfice	Exemple
Legacy Code	Ajouter des tests sur 1 module à la fois.	➔ Refactor sécurisé.	- **Approach: Strangler Fig**.
Nouvelle Feature	Découper en sous-tâches <1j.	➔ Livraison en 3 sprints vs 1.	- **User Story Splitting**.
Apprentissage	Spike de 2h pour évaluer une techno.	➔ Décision éclairée.	- **Timeboxed Research**.

14. Accepted Responsibility

Pourquoi ? La responsabilité **se prend**, ne s'assigne pas.

Pratique	Responsabilité	Impact	Outil
TDD	Le dev écrit **aussi** les tests.	➔ Code testé à 100%.	- **JUnit** / **pytest**.
On-Call	Rotation volontaire.	➔ Résolution plus rapide des incidents.	- **PagerDuty**.
Code Ownership	Un dev “possède” une feature de bout en bout.	➔ Moins de *handovers* problématiques.	- **Feature Flags**.

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
http://slamwiki2.kobject.net/eadl/bloc3/xp/chap1?rev=1763905141

Last update: **2025/11/23 14:39**