

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

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

### 1. Humanity

**Pourquoi ?** Des développeurs épanouis = code de meilleure qualité.

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

### 2. Economics

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

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

### 3. Mutual Benefit

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

Pratique	Bénéfice Devs	Bénéfice Business	Exemple
<b>TDD</b>	Code plus simple à maintenir.	→ -40% de bugs en production.	- <b>JUnit / pytest</b> .
<b>CI/CD</b>	Feedback immédiat sur les changements.	→ Livraisons 5x plus fréquentes.	- <b>GitHub Actions</b> .

Pratique	Bénéfice Devs	Bénéfice Business	Exemple
<b>Clean Code</b>	Onboarding des nouveaux plus rapide.	→ -30% de temps passé en reviews.	- <b>ESLint / Prettier.</b>

#### 4. Self-Similarity

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

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

#### 5. Improvement (Kaizen)

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

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

#### 6. Diversity

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

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

#### 7. Reflection

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

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

#### 8. Flow

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

Obstacle	Solution	Impact	Outil
<b>Blocages</b>	WIP Limits (max 2 tâches en cours).	→ -50% de *context switching*.	- <b>Kanban Board.</b>

Obstacle	Solution	Impact	Outil
<b>Dépendances</b>	User stories plus petites.	→ Livraisons incrémentales.	- <b>Story Splitting Techniques.</b>
<b>Feedback Lent</b>	CI/CD avec tests automatisés.	→ Feedback en <10 min.	- <b>GitHub Actions.</b>

### 9. Opportunity

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

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

### 10. Redundancy

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

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

### 11. Failure

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

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

### 12. Quality

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

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

### 13. Baby Steps

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

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

#### 14. Accepted Responsibility

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

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

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

Permanent link:  
<http://slamwiki2.kobject.net/eadi/bloc3/xp/chap1?rev=1763905293>

Last update: **2025/11/23 14:41**

