CM10
Développement piloté par les tests

Gerson Sunyé – JM Mottu

2010/2011 Master Alma

Plan

- Introduction.
- TDD et le test unitaire.
- Stratégies.
- TDD et le test de conformité.
- Conclusion.

Introduction

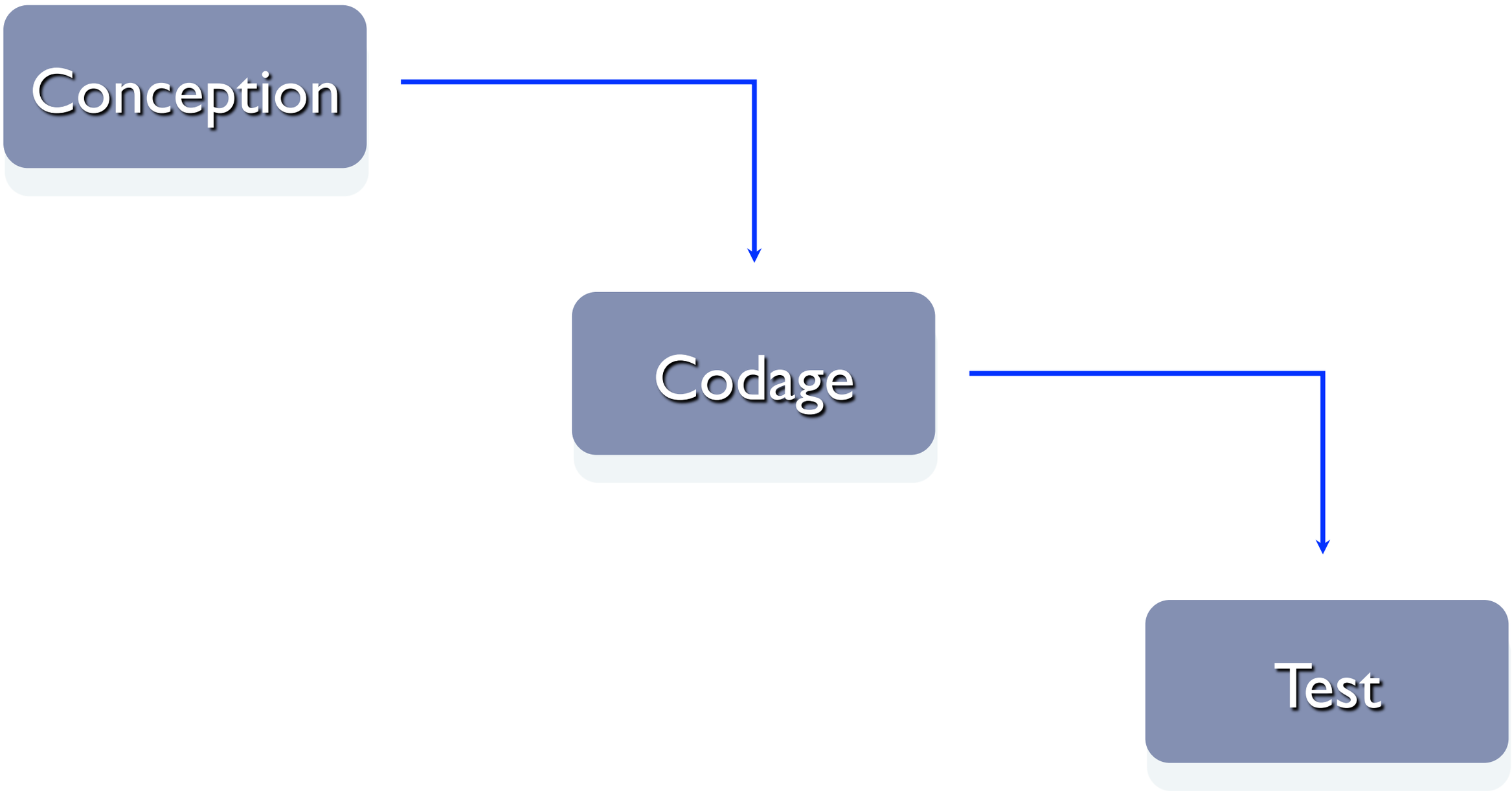
Méthodes Agile

- Méthode de développement pragmatique
  - On observe rapidement si le développement est correct
  - Cycle de développement court
- Adaptation
  - Retour sur développement rapide
  - Ajustement régulier
  - Souplesse de la planification
  - Développement incrémental
  - Pas de prédiction lourde dont les erreurs seraient remarquées tard
- Forte implication humaine

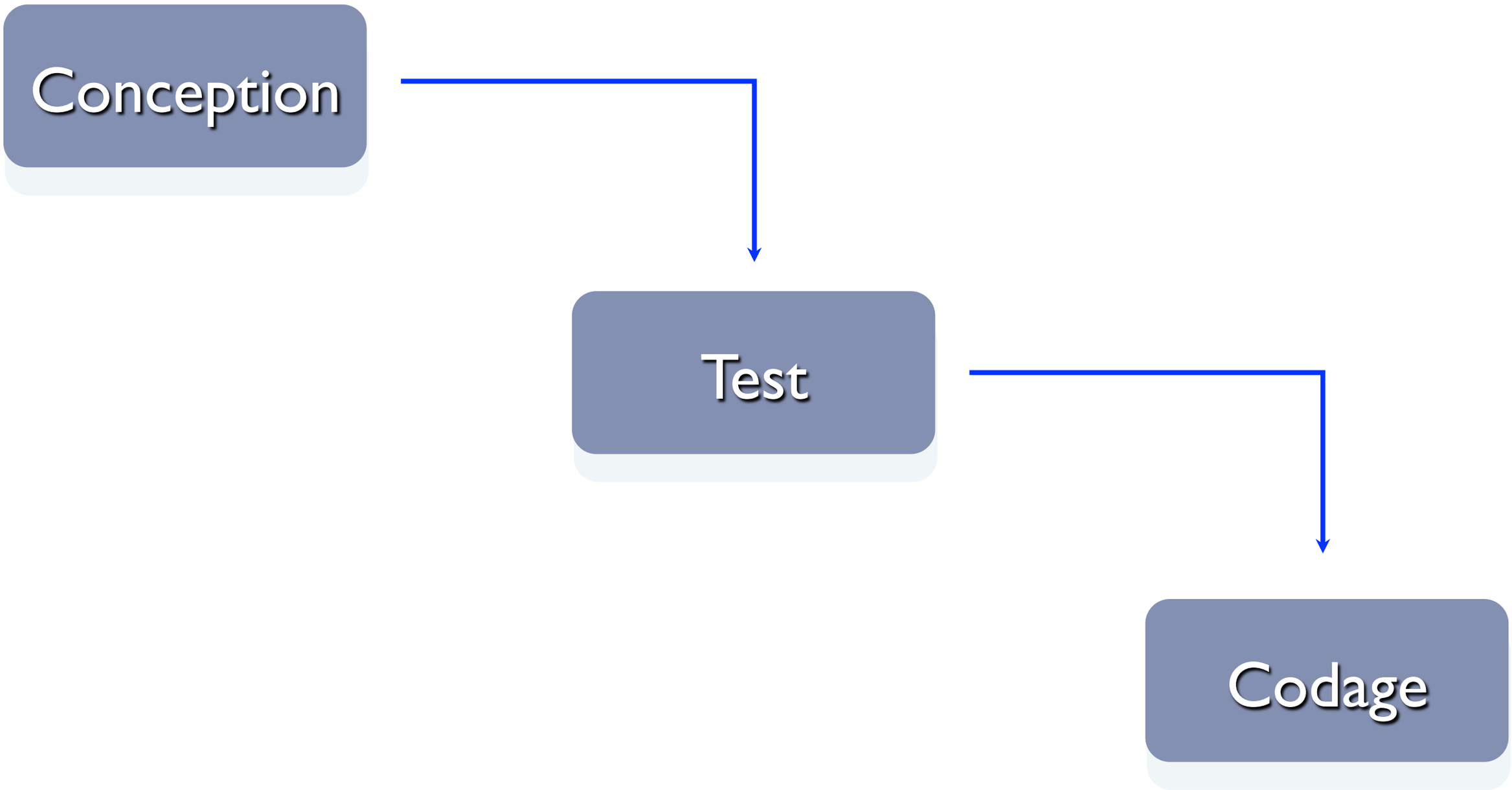
Méthodes Agile

- RAD (Rapid Application Development)

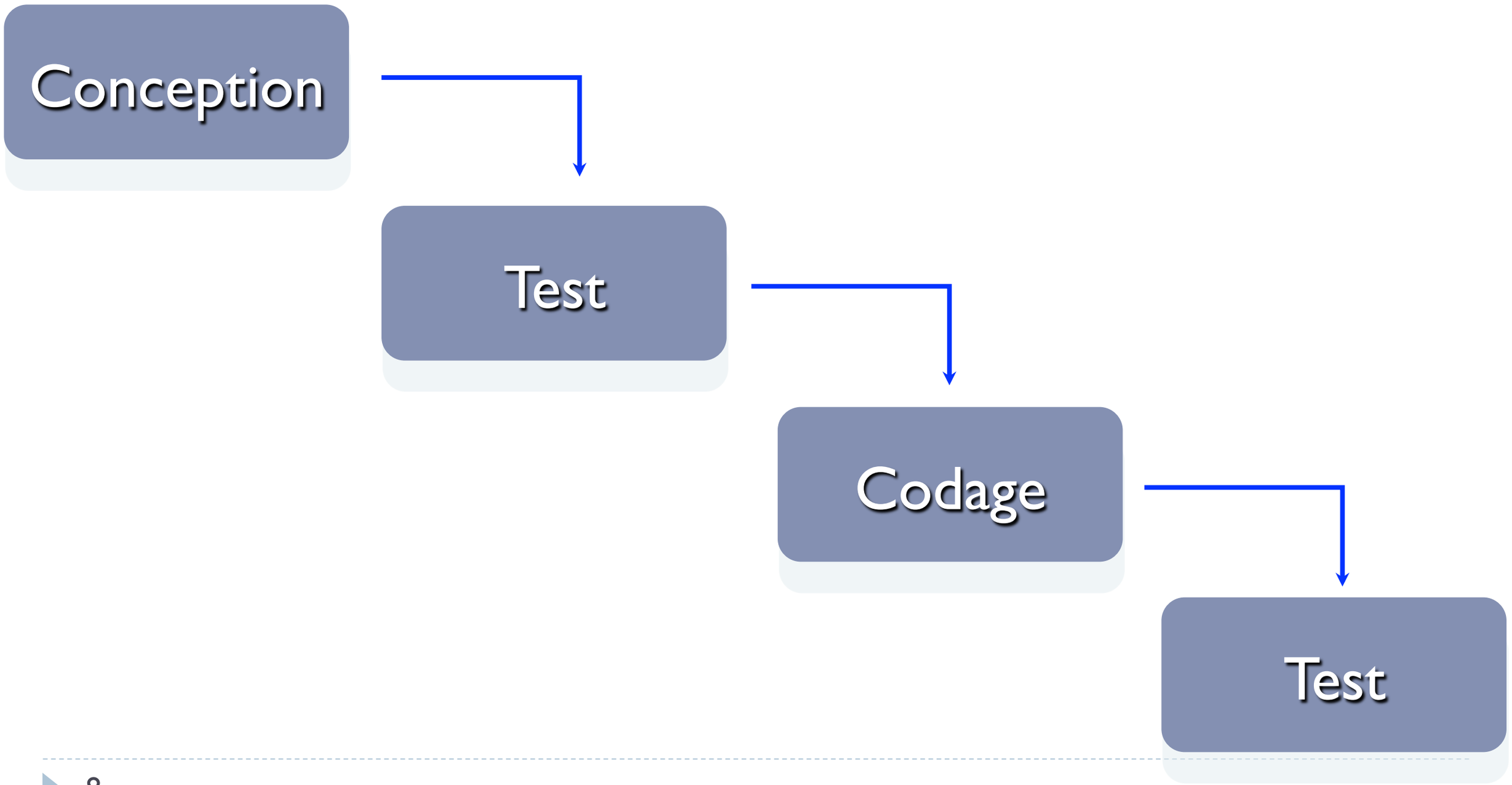
Développement traditionnel



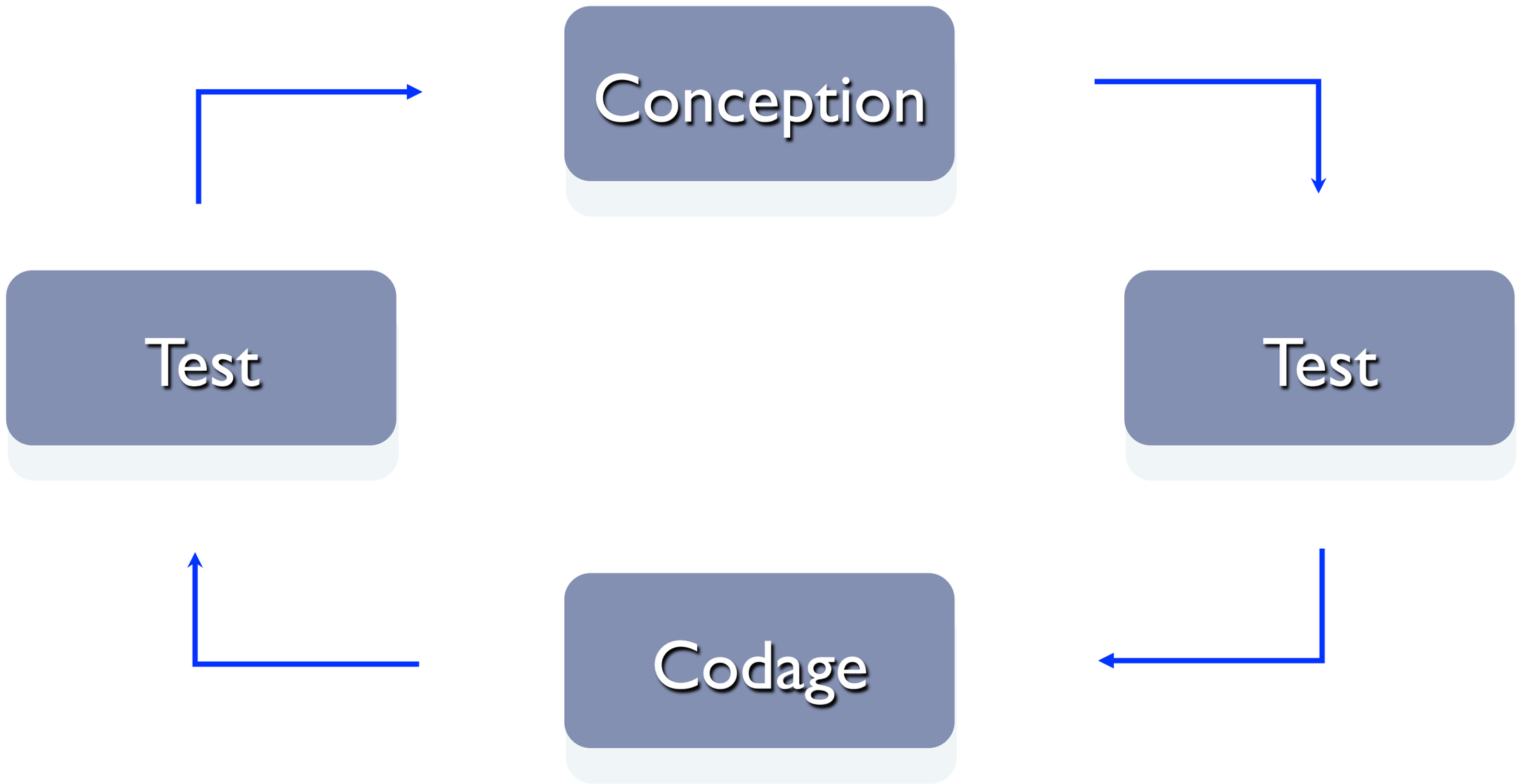
Développement piloté par les tests



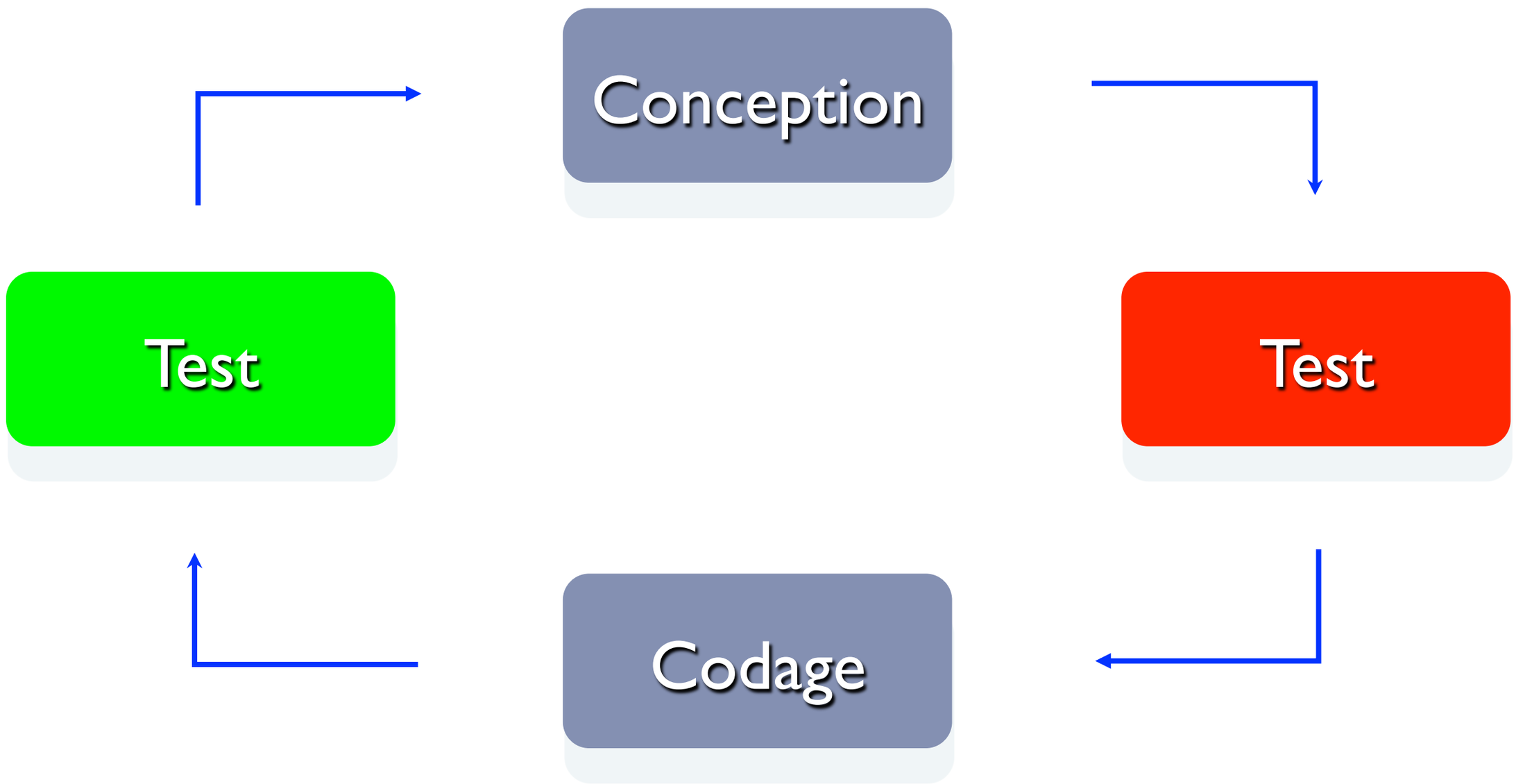
Développement piloté par les tests



Développement piloté par les tests



Développement piloté par les tests



Démarche

- XP: Extreme programming
- Écrire les cas de test avant le composant
  - les cas de test décrivent ce que le composant doit faire
  - avant d'écrire le code, les cas de test échouent
- Ensuite, on implémente la partie du composant qui fait passer le cas de test
  - après l'implémentation les cas de test doivent passer

Style de développement

- TDD Mantra : red/green/refactor
  1. Écrire les tests et les faire échouer
  2. Écrire le code qui les fait passer.
  3. Restructurer le code.

Conséquences

- Les cas de test spécifient ce que les composants doivent faire, mais pas comment

Conséquences

- Le développeur doit d'abord penser à la manière d'utiliser un composant.
- Ensuite, il peut penser à son implémentation.

Conséquences

- Une technique de conception qui engendre des composants:
  - simples à tester.
  - évolutifs.

Conséquences

- A tout moment, les développeurs connaissent:
  - ce qui marche correctement.
  - ce qui ne marche plus.

Motivations

- Les développeurs n'ont jamais le temps d'écrire les tests.
- Plus simple de maîtriser la complexité: retour rapide de ce qui marche et ce qui ne marche pas.

TDD et le test unitaire

Cycle de développement

- Cycle en cinq étapes :
  1. écrire un test ;
  2. le faire échouer;
  3. écrire le code suffisant pour passer le test,
  4. vérifier que le test passe,
  5. éliminer le code dupliqué, améliorer la conception.

Avantages

- Les cas de test servent de support à la documentation:
  - ils spécifient le comportement attendu des composants.
  - ils montrent des exemples d'utilisation du code.

Avantages

- YAGNI (You Ain't Gonna Need It).
  - seulement ce qui est indispensable est développé: on ne code que quand un test a échoué.
  - il n'y a pas de code sans test et pas de test sans besoin des utilisateurs.

Avantages

- Retours rapides sur la qualité du code
  - itérations courtes dans le cycle de développement.
  - on exécute le code tout de suite (avant même de l'avoir écrit).

Avantages

- Les tests unitaires sont reproductibles:
  - ils peuvent être utilisés comme test de non-régression lors de la restructuration du code.
  - ils réduisent la peur des modifications.

Stratégies

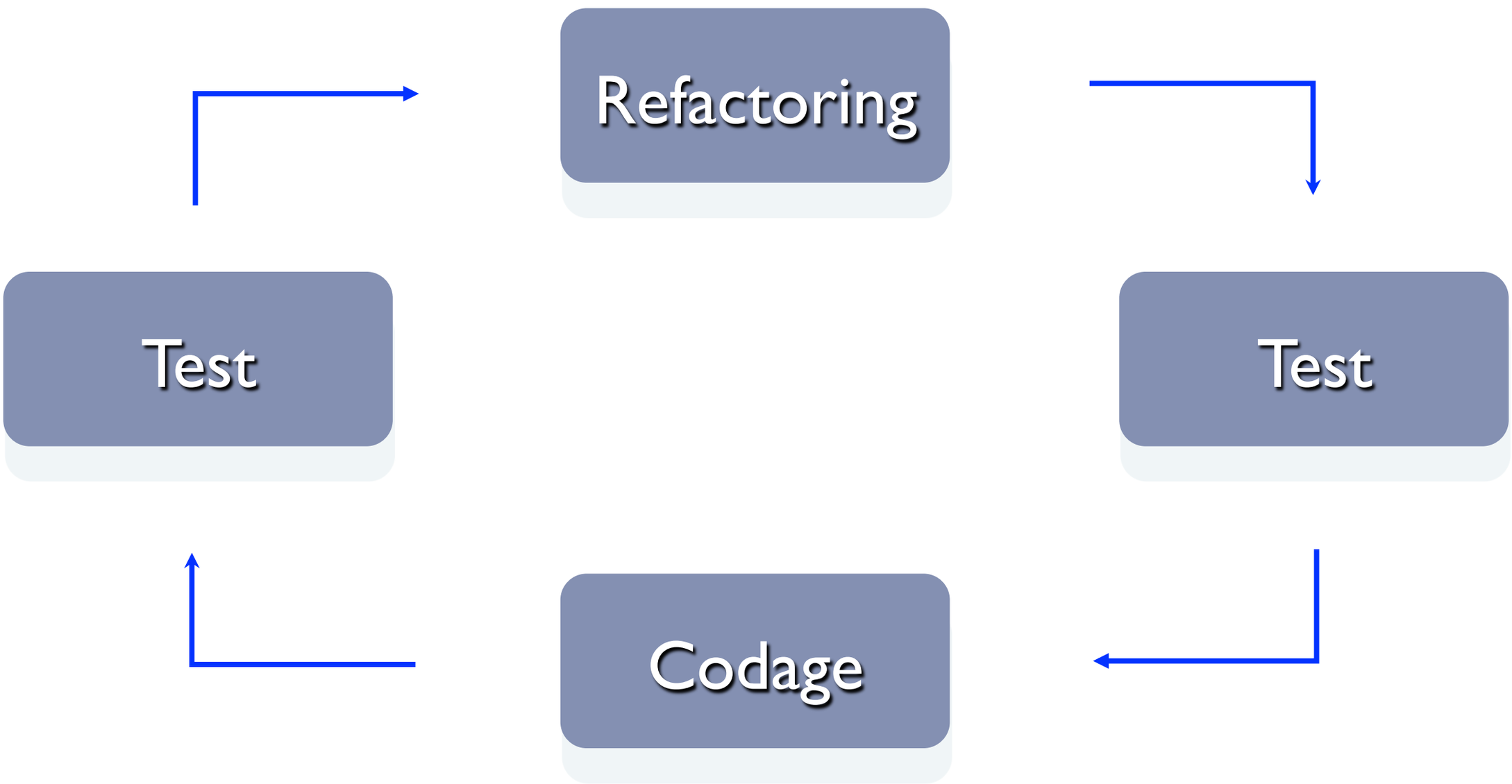
Stratégies

- Développement.
- Débogage
- Legacy code

Stratégie de développement

- Tester souvent:
  - trouver rapidement les erreurs.
  - savoir quand le but est atteint.
- Avancer par des petits pas:
  - programmation et restructuration.
  - Utilisation d'outils d'automatisation.

Stratégie de développement



Outils d'automatisation

- Test Unitaire:
  - Codage facile des tests unitaires, exécution rapide, possibilité de réexécution.
  - JUnit, TestNG
- Environnement de développement intégré (IDE)
  - Codage: auto-complétion, génération de classes de test.
  - Exécution des tests.
  - Restructuration.
  - NetBeans, Eclipse.

Outils d'automatisation

- Environnement de production de logiciels (build tool)
  - Exécution automatique des tests.
  - Génération de rapports.
  - Ant, Maven, Cobertura, PMD, Checkstyle, etc.

Ant / Maven

- Automatisation de tâches (répétitives)
- Enchaînement de différentes étapes de développement

PMD

- PMD est un outil d'analyse et de vérification de code source Java.
- Il permet de détecter :
  - code mort : variables, paramètres ou méthodes inutilisés,
  - Objets instanciés inutilement
  - blocks try / catch / finally / switch vides
  - code dupliqué/redondant : attention aux copier/coller
  - Expressions trop compliquées : if non-nécessaires, boucles for qui devraient être des boucles while
  - . . .

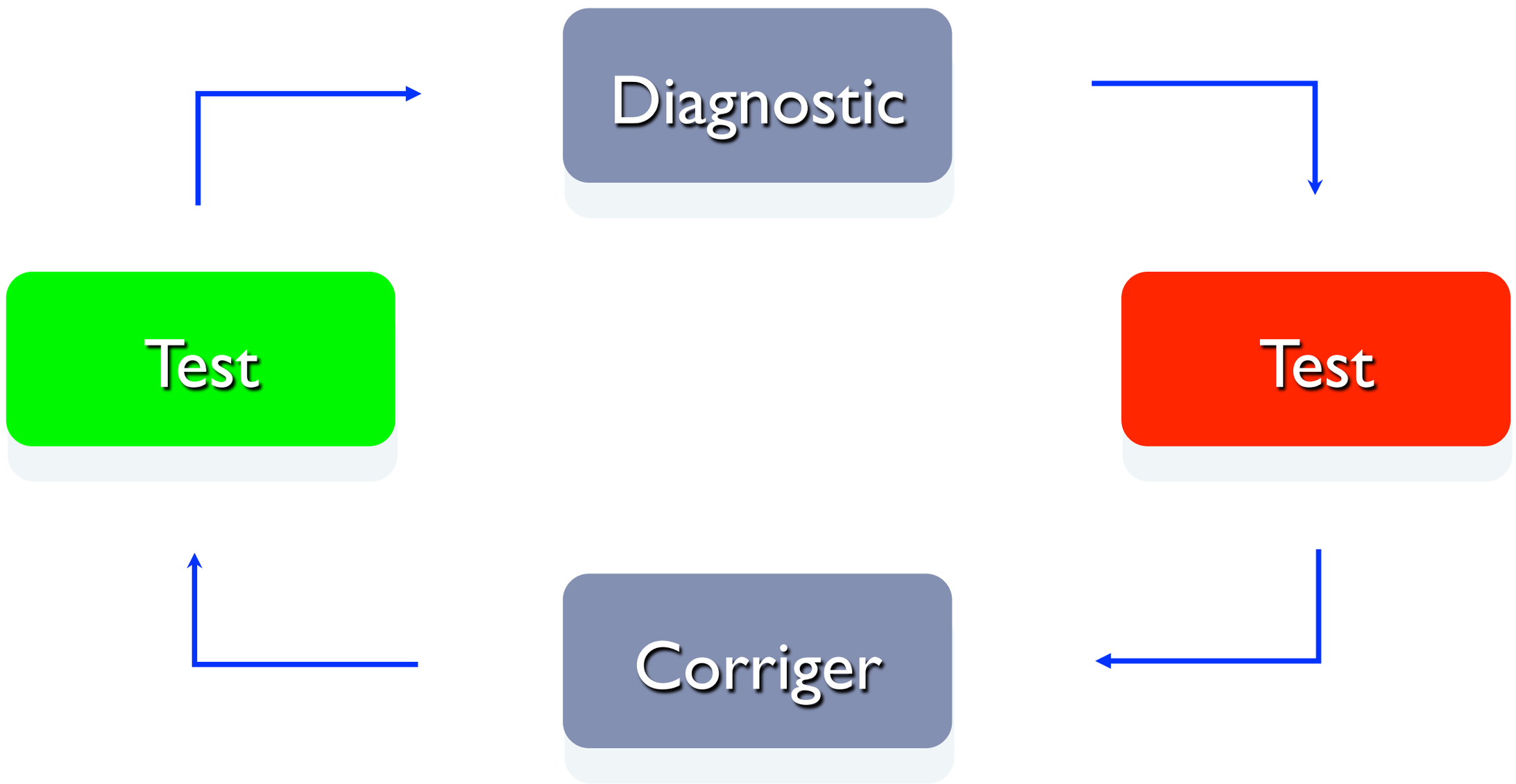
Checkstyle

- Outil de vérification du formatage et de présentation d'un code source Java
  - Open-source
  - Les vérifications produisent, selon le cas, une notification, un avertissement ou une erreur
  - Supporte les conventions de code de Sun (entre autre)
  - Plugins pour intégration dans des IDEs (Eclipse, Netbeans, JEdit, . . .)
  - peut être exécuté comme une tâche ANT
- http://checkstyle.sourceforge.net

Stratégie de débogage

- Repeat
  1. Chercher le défaut le plus gênant.
  2. Écrire un test qui le met en évidence.
  3. Corriger le défaut.
  4. Restructurer.
  5. Retester.

Stratégie de débogage



Legacy code

«— Êtes-vous sûrs que RPG est un langage de programmation ?»

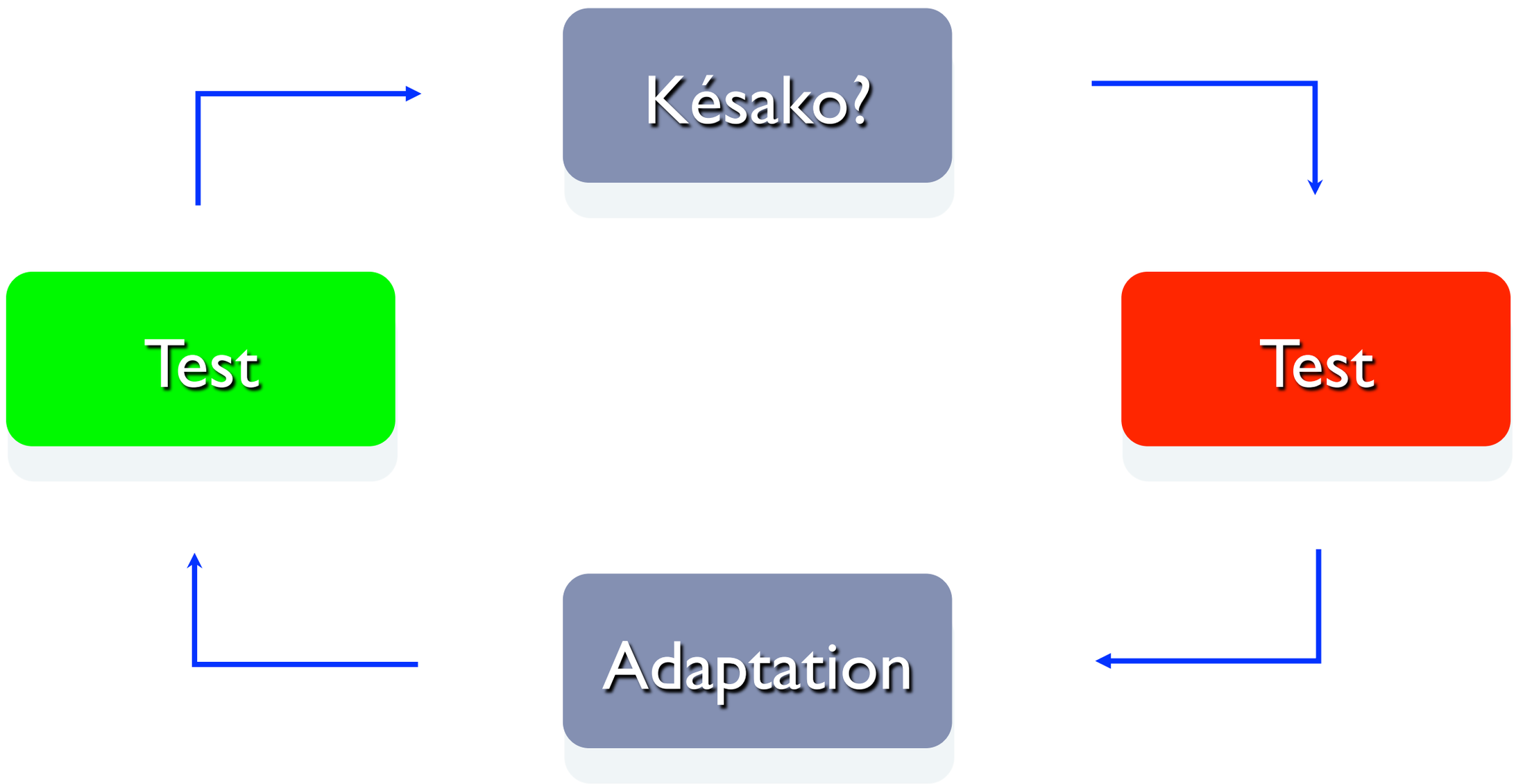
Legacy code

- «— Quelqu'un se rappelle de qui a écrit ce programme ?»
- «— La dernière fois que quelqu'un a touché à ça, il a fallut 3 semaines pour le remettre en état.»
- «— Il y a une doc, mais elle ne correspond pas au code.»
- Legacy code = code patrimonial

Legacy code

- Avancer doucement, par des petits pas, comme un funambule.
- Sans oublier le filet de sécurité !
- Penser à sauvegarder.

Legacy code



Legacy code

1. Essayer de comprendre le code. Et ne pas réussir.
2. Écrire un test basé sur ce qui a été compris. Le test échoue.
3. Adapter le test, jusqu'à ce qu'il passe.
4. Recommencer avec la méthode, la classe ou le composant suivants.

TDD et le test d'acceptation

Test d'acceptation

Test formel en rapport avec les besoins, exigences et processus métier, conduit pour déterminer si un système satisfait ou non aux critères d'acceptation et permettre aux utilisateurs, clients ou autres entités autorisées de déterminer l'acceptation ou non du système. [IEEE 610]

Développement itératif

1. Le client explique la prochaine fonctionnalité attendue, selon ses pritorités.
2. L'équipe divise la fonctionnalité en tâches.
3. Deux itérations différentes:
    1. interne à l'équipe
    2. avec le client

Itérations internes

- Itération très courte (24 ou 48h).
- Dédiées à la réalisation des tâches.
- A la fin de chaque itération, l'équipe se réunit pour faire le point:
  - tâches réalisées.
  - problèmes et solutions.

Itérations avec le client

- Itérations un peu plus longues (entre 1 semaine et 1 mois).
- Dédiées à la réalisation des fonctionnalités.
- A la fin de chaque itération, l'équipe présente au client la nouvelle fonctionnalité (test d'acceptation).

TDD et le test d'acceptation

- Écrire les tests d'acceptation le plus tôt possible.
- Les utiliser pour:
  - mieux comprendre chaque fonctionnalité.
  - mesurer l'évolution du développement.
- Les exécuter à la fin de l'itération.

Motivation

- Définition claire des responsabilités du client:
  1. spécification des besoins.
  2. validation des besoins.
- Meilleure collaboration entre clients et développeurs.

Bonus

- Les tests d'acceptation sont complémentaires aux tests unitaires.
- Ils détectent des défauts différents:
  - Besoins mal interprétés/exprimés.
  - IHM mal adaptée.

Approche traditionnelle

- Test de recette: le client écrit différents scénarios, où chaque scénario est une suite d'étapes.
- Après chaque livraison:
  - exécution à la main de chaque scénario.
  - exécution grâce un outil du type «capture & replay»: Marathon, SeleniumHQ

Test de recette manuel

- On ne voit que ce que l'on souhaite voir.
- La validation est difficile, voir confuse. Est-ce que le test a réussi ou pas ?
- Les développeurs ne connaissent pas les attentes des clients (anti tdd).
- Cher.

Test de recette automatique

- Très peu d'outils disponibles.
- Le système (et son IHM) doivent exister pour tester.
- Les tests sont fragiles.

Approche TDD

- Utilisation d'outils d'automatisation (e.g. FIT, FitNess, FitRunner):
  1. Le client décrit les besoins sous la forme d'un texte (et avant le codage).
  2. Le développeur lie les besoins aux composants-métier.
  3. Les objetifs sont clairs.

Framework for Integrated Test (FIT)

- Cadre d'applications écrit par Ward Cunningham.
- http://fit.c2.com
- Accepte plusieurs langages: Java, Python, etc.
- Adapté aux tests orientés données.

FIT : Utilisation

- Le client écrit les tests sous la forme de tableaux HTML.
- FIT interprète les données et la «glue» passe les valeurs aux tests.
- Les résultats sont présentés sous la forme de tableaux HTML.

Exemple: un document FIT

result.htm - Microsoft Word

File Edit View Insert Format Tools Table Window Documents To Go Help

77% Recount

Basic Employee Compensation

For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and holidays.

Here are some typical examples of this:

Payroll.Fixtures.WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360 *expected* $1040 *actual*

Page 1 Sec 1 1/1 At 1" Ln 1 Col 1 REC TRK EXT OVR E

Exemple: la glue


```
public class WeeklyCompensation : ColumnFixture
{
    public int StandardHours;
    public int HolidayHours;
    public Currency Wage;

    public Currency Pay()
    {
        WeeklyTimesheet timesheet = new WeeklyTimesheet(StandardHours, HolidayHours);
        return timesheet.CalculatePay(Wage);
    }
}
```

Conclusion

En résumé

- Les besoins pilotent les tests.
- Les tests pilotent le développement du code
- Aucun code n'est écrit avant l'écriture d'un test.
- Les tests sont exécutés fréquemment.
- Les tests et le code sont écrits par incréments.
- Les opérations de restructuration sont réalisées en continu et sont assurées par les tests.

Bénéfices

- Moins de temps de débogage (et les bugs).
- Preuve que le code satisfait les besoins.
- Des cycles de développement plus courts.
- Réduction du temps de développement.
- Les tests deviennent partie du produit.
- Documentation.
- Amélioration de la qualité du code.
- Diminution des coûts de maintenance

Outils

- FIT: http://fit.c2.com/
- FitNess: http://www.fitnesse.org/
- FitRunner: http://fitrunner.sourceforge.net/

Références

- «Extreme programming explained: embrace change. Kent Beck. Addison-Wesley, 1999.
- «Test-Driven Development: By Example». Kent Beck. Addison-Wesley, 2002.
- « Test-Driven Development: A Practical Guide». David Astels. Prentice Hall, 2003.

Références

- «Test-Driven Development».
- Christoph Steindl.
- http://www.agilealliance.org/system/article/file/1423/file.pdf
- «Test Driven Development Tutorial».
- Kirrily Robert.
- http://www.slideshare.net/Skud/test-driven-development-tutorial