ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

# 04 Chapitre

# Méthode agile

**1I2AC1 : Génie logiciel et Conception orientée objet**

Régis Clouard, ENSICAEN - GREYC

« Je ne suis pas un grand programmeur.
Je suis juste un bon programmeur avec de bonnes habitudes. »
**Kent Beck (créateur de la méthode X-Programming)**

# Objectif du chapitre



- Une introduction à l'agilité pour le développement logiciel.
  - *Les méthodes de développement seront vues en détail au semestre 8.*
  - Ici, nous présentons une approche pragmatique utilisable pour les TP et les projets.
- À l'issue de ce chapitre, vous serez sensibilisé à :
  - l'importance d'un cycle de développement itératif et incrémental ;
  - la nécessité d'un dialogue permanent avec les futurs utilisateurs sur la base d'une version opérationnelle du futur logiciel ;
  - le rôle central du code ;
  - l'obligation de tester son code.
- Cela doit vous permettre de changer votre façon de développer en TP et en projet pour aborder plus sereinement le travail de conception et de programmation.

# Plan du chapitre



1

Mauvaises pratiques du développement

# Développement en TP

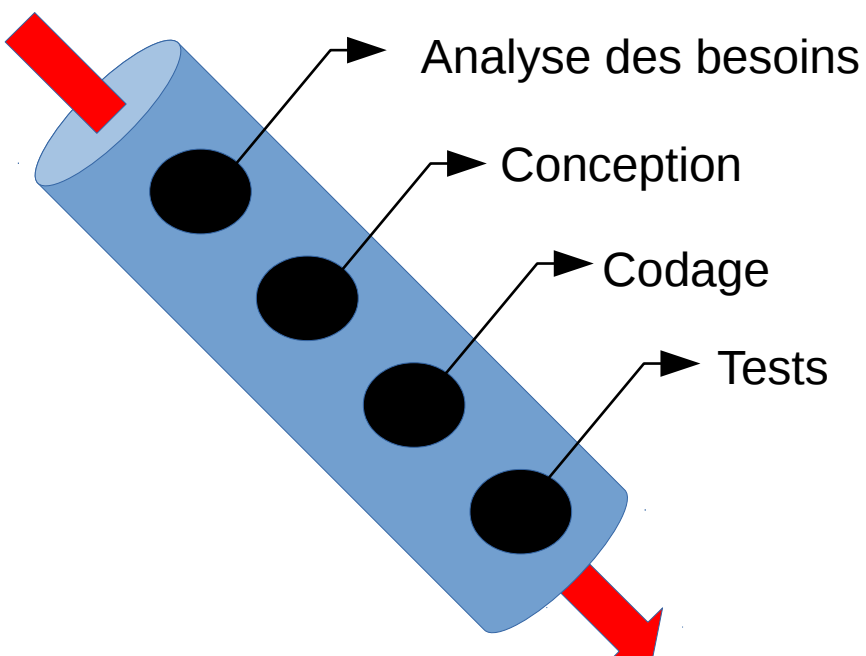


- Pratique recherchée et parfois enseignée :
  1. Je code toutes les fonctions demandées dans le sujet.
  2. Je compile.
  3. J'exécute pour tester le programme.
- Aie, ça ne marche pas.
  - Ça ne compile pas !
  - Le compilateur génère un flot de messages d'erreur que j'ai du mal à comprendre.
  - Plus d'erreur de compilation, je teste le programme en vérifiant visuellement que les sorties semblent correctes.
  - Si le programme ne fonctionne pas correctement, je truffe le code d'affichages intermédiaires pour identifier la cause du dysfonctionnement.
- Cette pratique porte un nom en génie logiciel : le cycle de développement en cascade.

# Développement en cascade



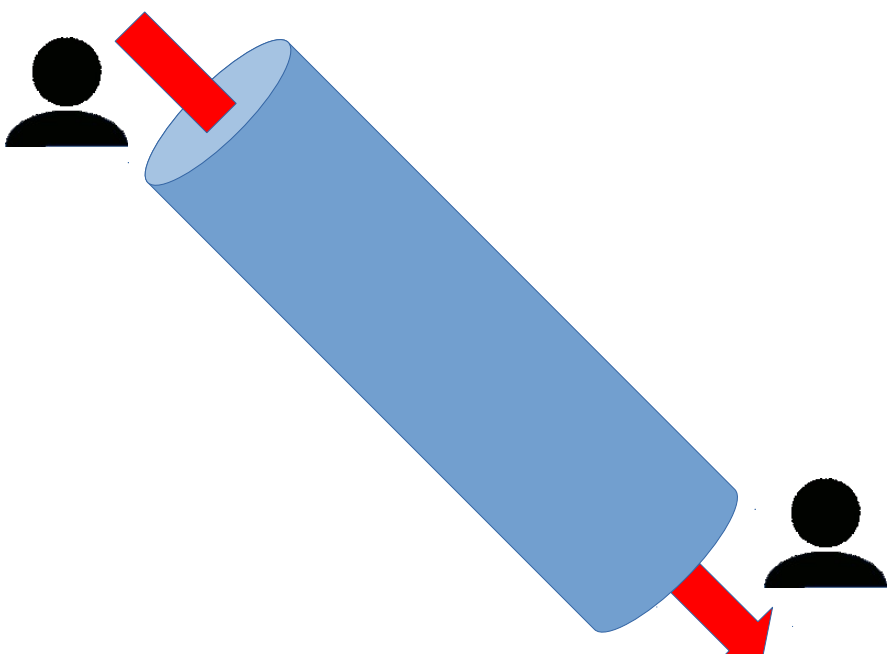
- Le cycle de développement en cascade a longtemps été enseigné et utilisé en entreprise.
  - Il fait partie des méthodes prédictives.
  - Il est la cause de l'échec de nombreux projets professionnels.



# Analyse des besoins au début et effet tunnel



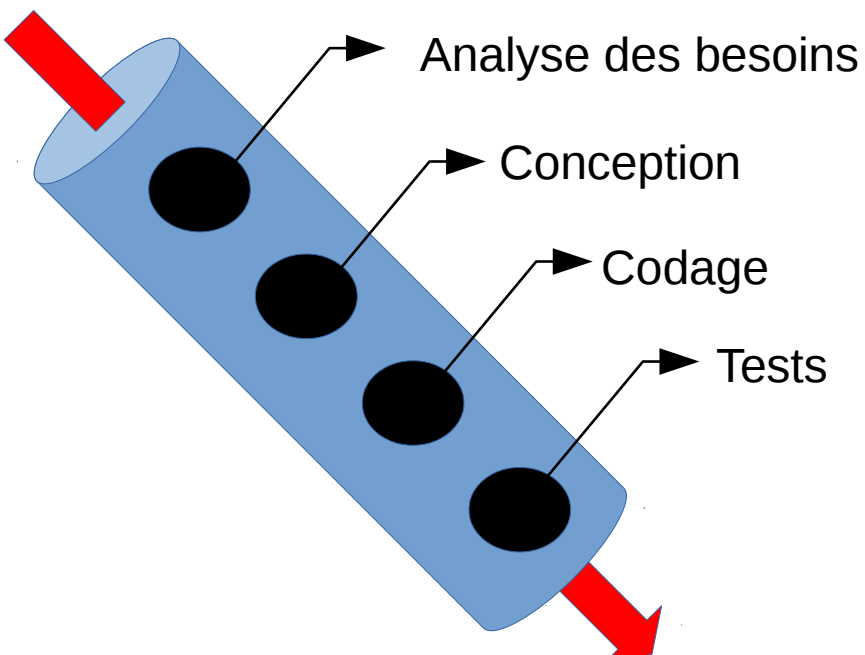
- Ne marche pas en pratique :
  - On se met d'accord avec le client à l'entrée du tunnel sur la liste des fonctionnalités à développer (étape qui peut durer 2 mois).
  - Et quand on ressort du tunnel, les besoins du client ont changé !



# Conception puis codage



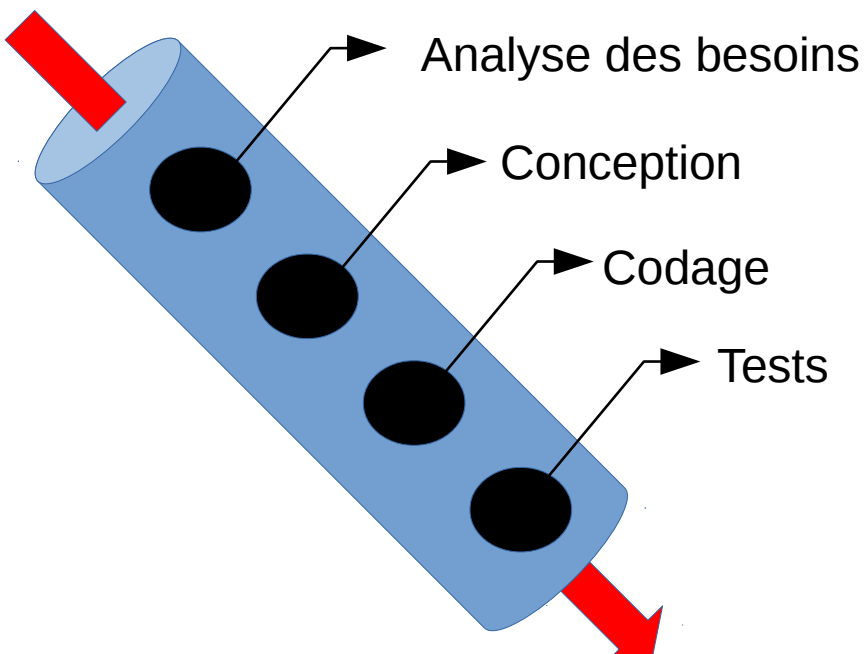
- Ne marche pas en pratique :
  - On peut très bien découvrir au moment du codage un problème qui remet en cause la conception voire l'analyse des besoins.
  - Il faut recommencer, ce qui peut mettre le projet en péril.



# Test après le codage



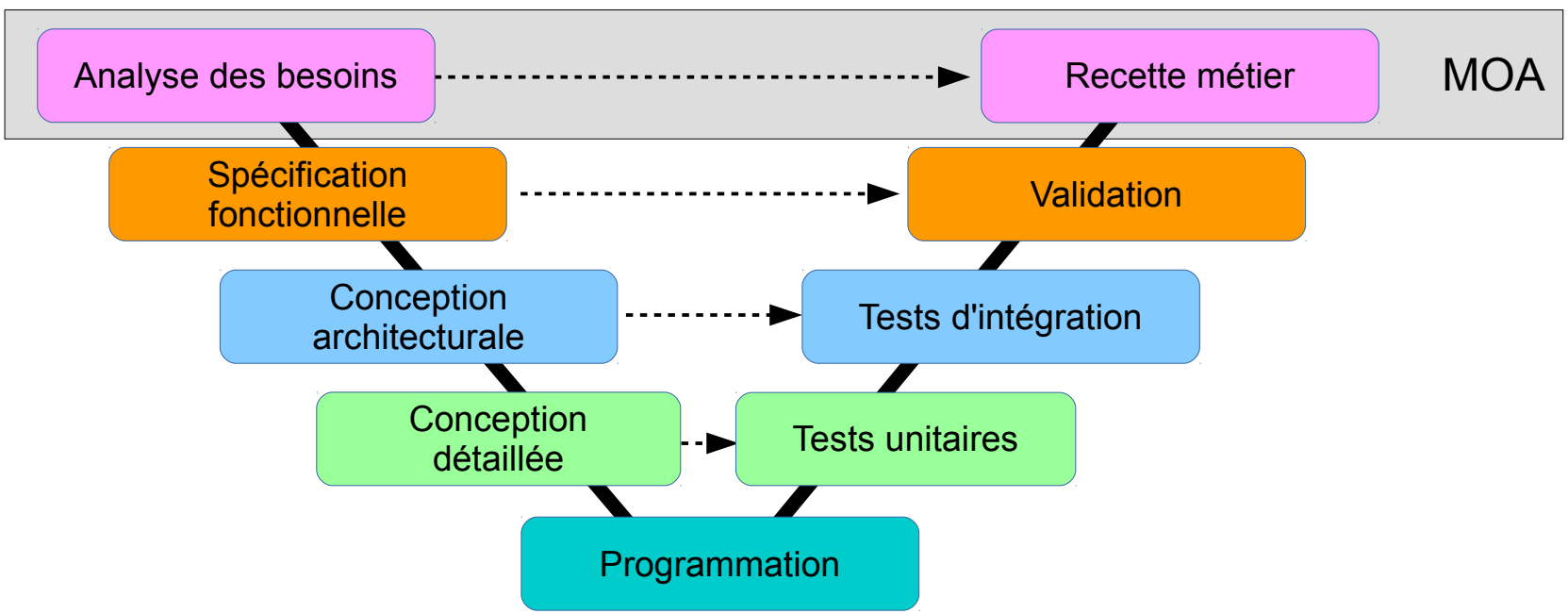
- Très inefficace
  - Les bugs et erreurs doivent être détectés le plus tôt possible.
  - La définition des tests en fin est difficile parce que l'on a perdu les objectifs de ce que l'on doit tester.
  - Les tests sont en général court-circuités pour tenir les délais.



# Une amélioration : le cycle en V



- L'apport du cycle en V se situe au niveau des tests.
  - Le meilleur moment pour définir les tests, c'est lors de la spécification des éléments à tester.
- Mais globalement, l'amélioration est modeste.
  - p. ex. du point de vue MOA, le cycle en V est toujours vu comme un tunnel.



# Plan du chapitre



1

Mauvaises pratiques du développement

2

Bonnes pratiques du développement : agilité

# Les 4 valeurs de l’agilité



- Méthode prédictive
  - Processus et outils. → Individus et interactions.
  - Documentation exhaustive. → Logiciel qui fonctionne.
  - Négociation contractuelle. → Collaboration avec les clients.
  - Suivi d’un plan. → Adaptation au changement.

- Méthode agile

- Remarque
  - « Bien qu'il y ait de la valeur dans les éléments situés à gauche, notre préférence se porte sur les éléments qui se trouvent sur la droite. »

# Mise en pratique



- Cycle de développement itératif
- Cycle de développement incrémental
- Collaboration entre le client et l'équipe de développement
- Petite équipe (< 10 personnes) auto-organisée et rythme constant
- L'intégration continue du travail des développeurs (quasi-quotidienne).

- Il s'agit de produire rapidement des versions opérationnelles successives et de plus en plus complètes du logiciel : des prototypes.

« Un intellectuel assis ira toujours moins loin qu'un con qui marche » - Michel Audiard.

# Exemples de méthodes agiles

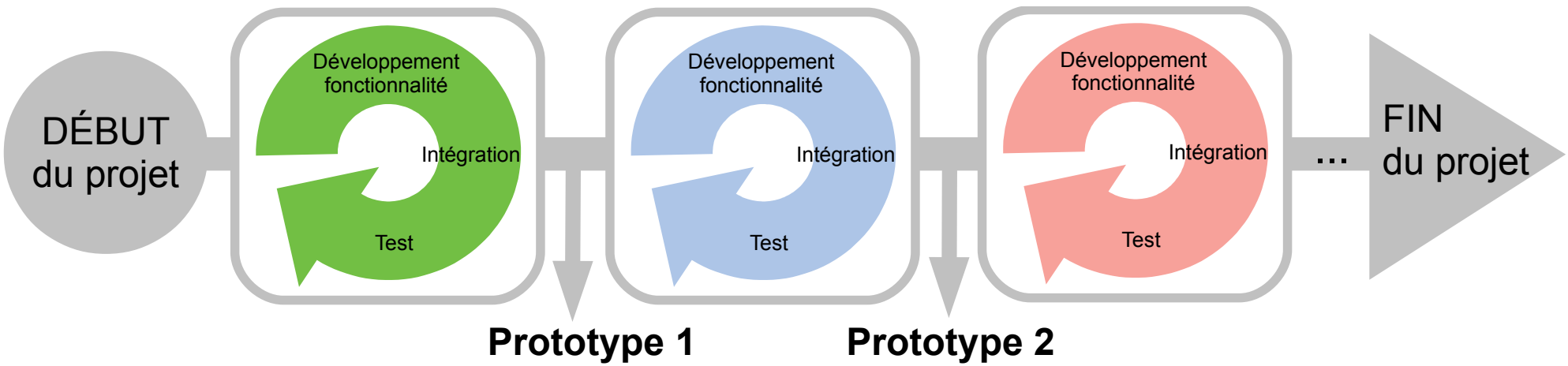


- Il n'existe pas une mais plusieurs méthodes agiles :
  - eXtreme-Programming (aussi nommée XP)
    - La méthode pionnière dont la plupart des principes sont repris par les suivantes (et dans ce cours).
  - Kanban
    - La plus simple et connue par son fameux tableau (TODO/DOING/DONE).
  - Scrum
    - La plus utilisée en France.
- La mise en place concrète de l'agilité au sein d'une organisation mixe plusieurs des méthodes ci-dessus.
- L'application de l'agilité doit elle-même être agile : il faut s'approprier l'agilité.

# a. Cycle de développement itératif



- Le temps de développement est divisé en **itérations**.




- Itération
  - Courte durée (environ 2 semaines)
  - Correspond au développement / intégration / test de fonctionnalités.
  - Se termine par la livraison d'un prototype opérationnel livré au client.
  - Une fin en soi (comme si le projet s'arrêtait à l'issue).
    - Pas de tâche s'étalant sur plusieurs itérations ; si nécessaire la découper.
    - Pas d'ajout de nouvelle tâche au cours d'une itération.

# b. Cycle de développement incrémental

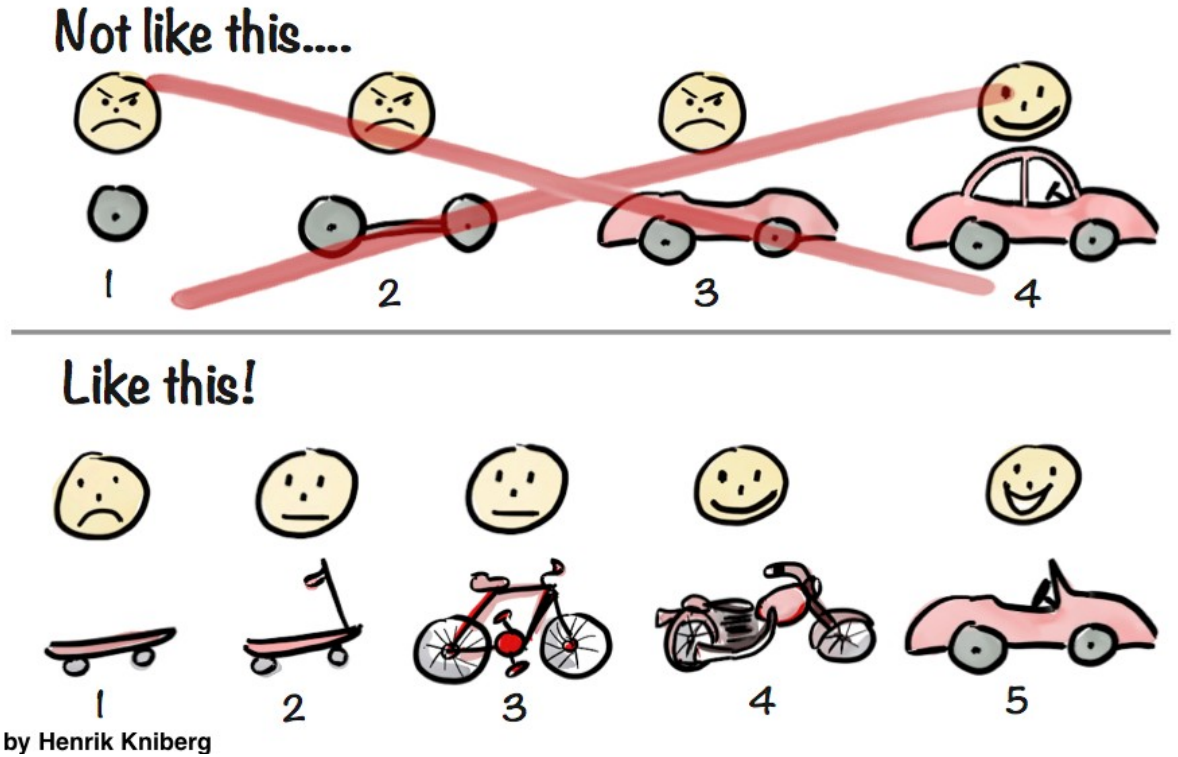


- Le logiciel est construit par **incrémentation**.
  - Le prototype n'est pas jetable.
  - Au contraire, il sera amélioré au cours des itérations suivantes s'il est jugé intéressant par le client.
  - Chaque prototype suivant ajoute de nouvelles fonctionnalités au prototype précédent.
- La règle des 80 – 20 (*loi de Pareto*)
  - 20 % des fonctionnalités couvrent 80 % des cas d'utilisation du futur logiciel et les 80 % des fonctionnalités restantes ne couvrent que 20 % des cas d'utilisation.
  - Il y a évidemment tout intérêt à commencer par développer les fonctionnalités des 20 %. Ce sont celles qui apportent le plus de valeur au client immédiatement.

# Incrément : prototype



- MVP : Minimum Viable Product
  - Un prototype avec juste assez de fonctionnalités pour satisfaire les premiers clients et fournir une rétroaction pour poursuivre le développement.




https://blog.crisp.se/2016/01/25/henrikkniberg/making-sense-of-mvp

# c. Collaboration client - développeur



- Client (utilisateur)
  - C'est un partenaire. Il est fortement impliqué dans le cycle de développement (quotidiennement dans l'idéal).
  - C'est lui qui définit les priorités.

# d. Équipe



- Équipe d’ingénieurs développeurs
  - Auto-organisée.
  - Rythme soutenable. Ne prendre qu’une liste de tâches réalisables dans l’itération.
  - Dialogue face à face.
  - Motivation des équipes.
  - Propriété collective du code.

# e. Intégration continue

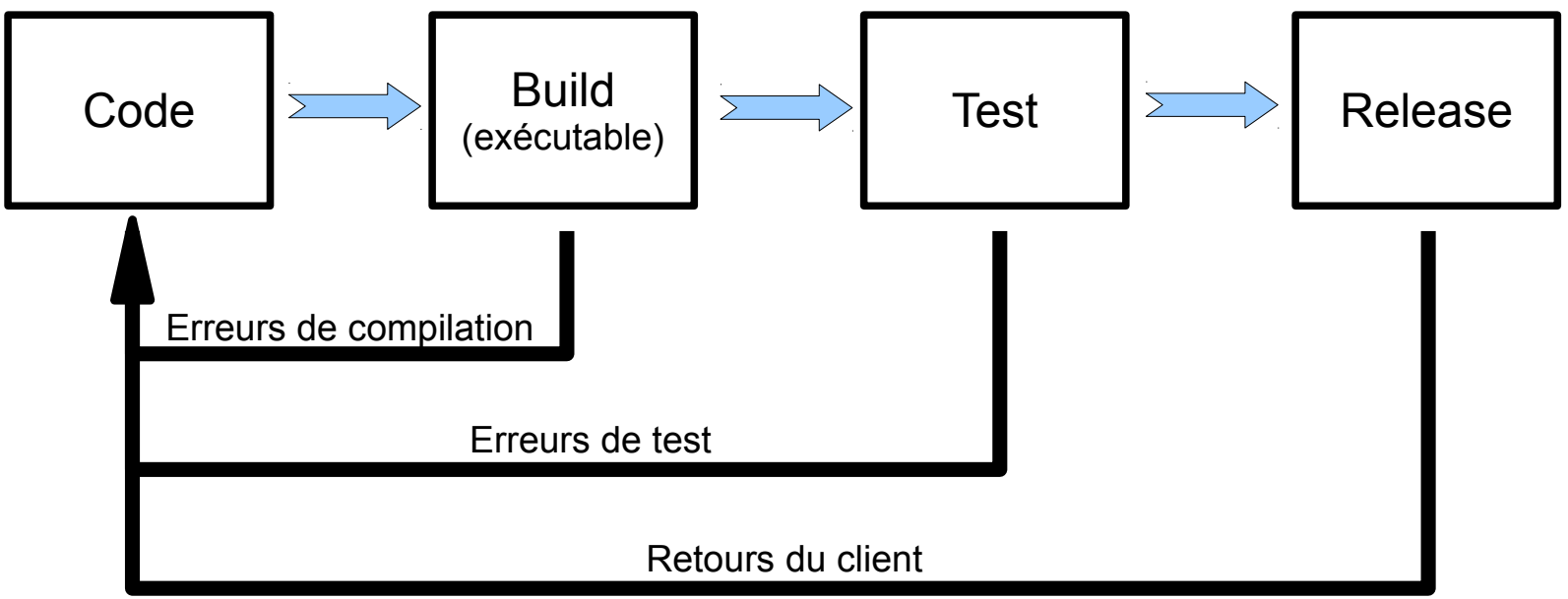


- Intégration quasi-quotidienne du travail des développeurs.
  - Le but est d'éviter l'effet big bang de l'intégration en fin d'itération (ne fonctionne jamais).
  - Il n'existe qu'une version courante du logiciel partagée par tous les développeurs.
    - Version opérationnelle et testée
  - Chaque développeur en possède une copie de travail.
    - Le développeur ajoute ses modifications à la version commune et cela déclenche des mécanisme automatiques de vérification de la non régression de la version commune.

# Cycle d'intégration continue



- Vérification automatique de la non régression de la version courante.
  - La version doit être compilable et testée sans intervention humaine.
  - Peut aller jusqu'au déploiement automatique chez le client (déploiement continue).
  - On parle aussi de DevOps.



# Avantages



- Le cycle itératif
  - On avance à petits pas testés.
  - Si on rate un pas, on n'a raté qu'un pas, sans grosses conséquences.
  - Chaque pas est validé avec le client.
  - Le manque de temps conduit à un déficit de fonctionnalités et pas à un échec total du projet ni à un logiciel non testé.
  - On a toujours une version intermédiaire mais opérationnelle du logiciel à donner au client.
  - Réduit le temps de mise sur le marché (*time to market*).
- Incrémentation
  - Donne rapidement de la valeur au produit.
  - Le client voit une évolution rassurante et constante du produit.

# Application en TPs et projets



- Démarche qui s'inspire du développement agile :
  - Prendre une seule fonctionnalité à implémenter.
  - Programmer la fonctionnalité.
  - Tester la fonctionnalité pourquoi pas en ajoutant un « `main()` » pour tester ce code (ou mieux en utilisant une bibliothèque de test).
  - Si la fonction est bien programmée, passer à la suivante.
- Permet de se rassurer et voir son projet avancer

# Plan du chapitre



1

Mauvaises pratiques du développement

2

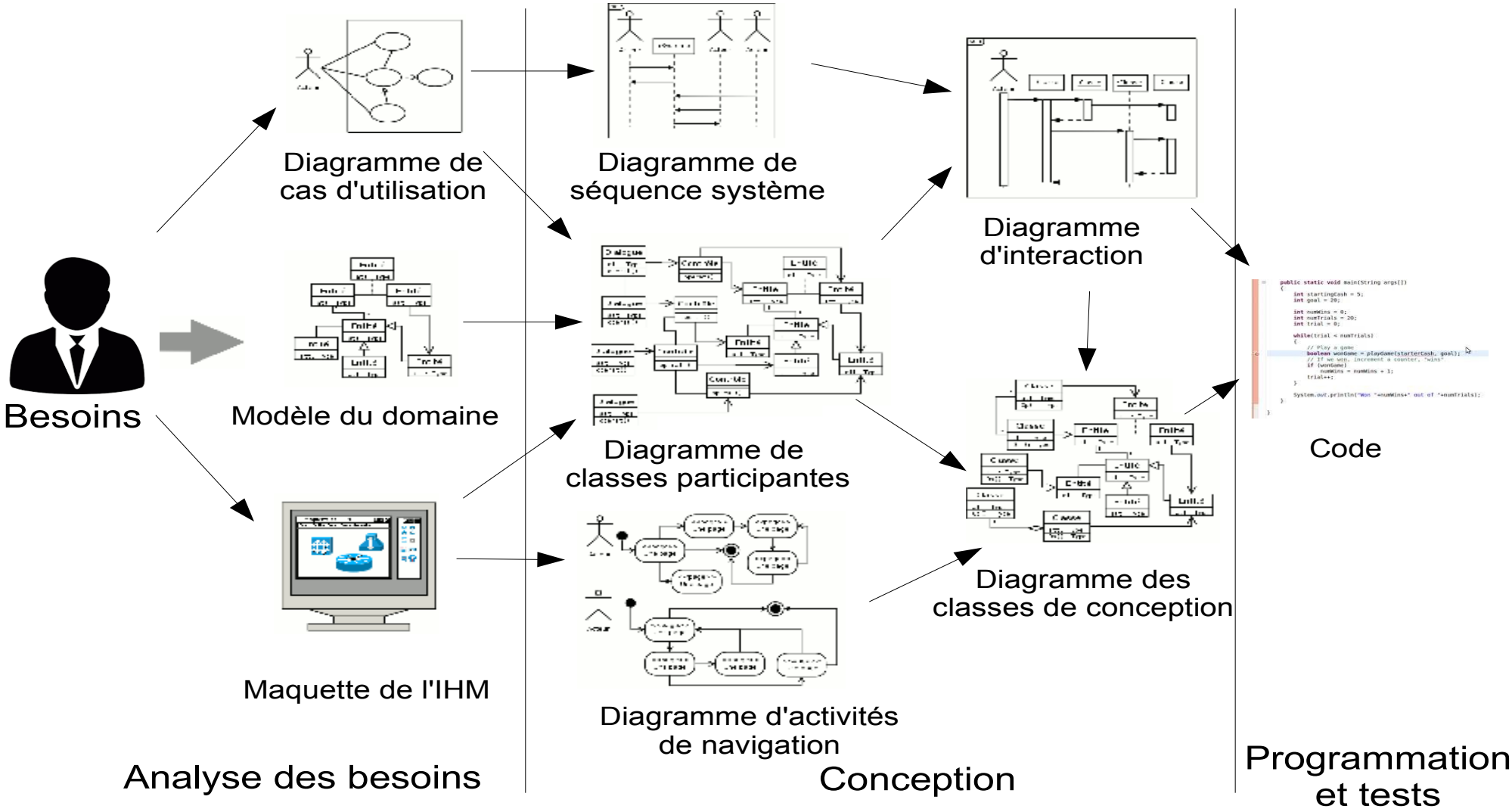
Bonnes pratiques du développement

3

Développement et diagrammes UML

# Où utiliser les diagrammes UML





# Analyse des besoins



- Diagramme des cas d'utilisation.
  - Poser le problème en termes d'interactions entre les utilisateurs et le système.
    - Identifier les fonctionnalités à réaliser.
- Maquettes des écrans de l'IHM.
  - Les écrans des futures interactions.
    - Papier ou graphique (avec un créateur d'interface eg. *SceneBuilder*).
- Modèle du domaine.
  - Une première analyse du domaine à partir des concepts manipulés dans le domaine.

# Conception



- Diagramme de classes
  - Parce qu'il est difficile de trouver les classes du futur logiciel :
    1) S'appuyer sur le « modèle du domaine ».
       - Il fournit les premières classes.
    2) Modéliser les aspects fonctionnels avec les diagrammes dynamiques (séquence, activité, état-transition).
       - Ils permettent de comprendre comment cela peut fonctionner.
    3) Ajouter les « classes participantes » dans le diagramme de classes.
       - On obtient le diagramme de « classes de conception ».

# Conception modulaire



- « Diviser pour régner »
  - Face à des problèmes complexes, rechercher la modularité afin d'obtenir un ensemble de modules plus simples et plus facilement gérables.
- En conception orientée objet, la notion de module se décline en :
  - **Méthode** : implémentation de services.
  - **Classe** : abstraction de données et de services.
  - **Paquet** : groupement de classes dans une seule collection.

# Conception modulaire



- **La décomposition d'un système en modules est une tâche difficile.**
  - Pas de théorie.
- Or ces modules jouent un rôle important pour la flexibilité, la fiabilité et la robustesse du système.
- Plusieurs méthodologies à partir du cahier des charges.
  - Débat sur la meilleure approche.

# Méthodologie pour déterminer les classes : exemple 1



- Méthode 1 : Analyse de texte (Dennis, 2002)
  - Un nom commun → une **classe**
  - Un nom propre ou une référence directe→ un **objet**
  - Un nom collectif → une **classe**
  - Un adjectif → un **attribut**
  - Un verbe “faire” → une **méthode**
  - Un verbe “être”→ un **héritage** ou une **instanciation**
  - Un verbe “avoir” → une **association**
  - Un verbe transitif → une **méthode**
  - Un verbe intransitif → une **exception**
  - Une phrase verbale prédicative → une **méthode**
  - Un adverbe → un attribut d'une **méthode**

# Méthodologie pour déterminer les classes : exemple 2



- Méthode 2 : les « cartes CRC » **C**lasses, **R**esponsabilités et **C**ollaborations
  - 1 post-it.
  - Nom de la classe en haut.
  - Les responsabilités de la classe avec en face les collaborations pour assurer cette responsabilité.
    - On assigne à un objet *o* les responsabilités pour lesquelles *o* possède toutes les informations nécessaires pour remplir ces responsabilités (dans certains cas, il collabore avec d'autres objets pour cela).
  - Les cartes sont collées sur un tableau et forment ainsi l'architecture du logiciel.
- Attention !
  - Attributs : ne pas s'intéresser aux attributs (encapsulation).
  - Méthodes : ne s'intéresser qu'aux services (ie méthodes publiques).

# UML et Agilité



- En agilité : seul compte le code
  - Pas de production de documentation (modèles)
    - Les modèles mentent parce qu'ils ne suivent pas les modifications qui sont apportées au code.
    - Le code ne ment pas.
  - Les modélisations UML ne sont utilisées que pour échanger entre développeurs et avancer dans la compréhension du domaine et des besoins.
    - Utilisation d'une version allégée d'UML.
    - Ce ne sont que des modélisations temporaires
    - Mais :
      - Elles doivent être exactes (syntaxe) pour éviter l'ambiguïté.
      - Elles doivent rester simples pour apporter une valeur claire.
  - Et si on a besoin de documentation, par exemple lors de la reprise de code ?
    - Rétro-ingénierie du code (reverse engineering).

# Que retenir de ce chapitre ?



- Les méthodes agiles proposent de nouvelles façons d'aborder le développement logiciel.
  - Le développement suit un cycle itératif et incrémental.
    - Chaque itération correspond au développement de plusieurs petites fonctionnalités qui sont intégrées aussitôt au logiciel et testées.
    - Chaque itération doit se terminer par la livraison d'une version opérationnelle et testée du logiciel en cours.
    - Chaque incrément doit ajouter une valeur pour le client au prototype : MVP.
    - Chaque itération est une fin en soi : un mini-projet.
- Attention
  - Les méthodes agiles ne renient pas les méthodes prédictives telles le cycle en V, mais dans la mesure du possible il est préférable d'être agile.