ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Chapitre 4
Processus de développement agile

1I2AC1 : Génie logiciel et Conception par 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 première introduction aux principes agiles du développement logiciel.
  - *(Les méthodes de développement seront vues en détail au semestre 8.)*
  - Ici, nous utilisons une approche pragmatique inspirée des méthodes agiles.
- À l'issue de ce chapitre, vous serez sensibilisé :
  - à l'importance d'un cycle de développement itératif et incrémental.
  - à la nécessité d'un dialogue permanent avec les futurs utilisateurs.
  - à l'obligation de tester son code.
- Cela doit vous permettre de changer votre façon de développer en TP pour aborder sereinement le travail de conception et de programmation.

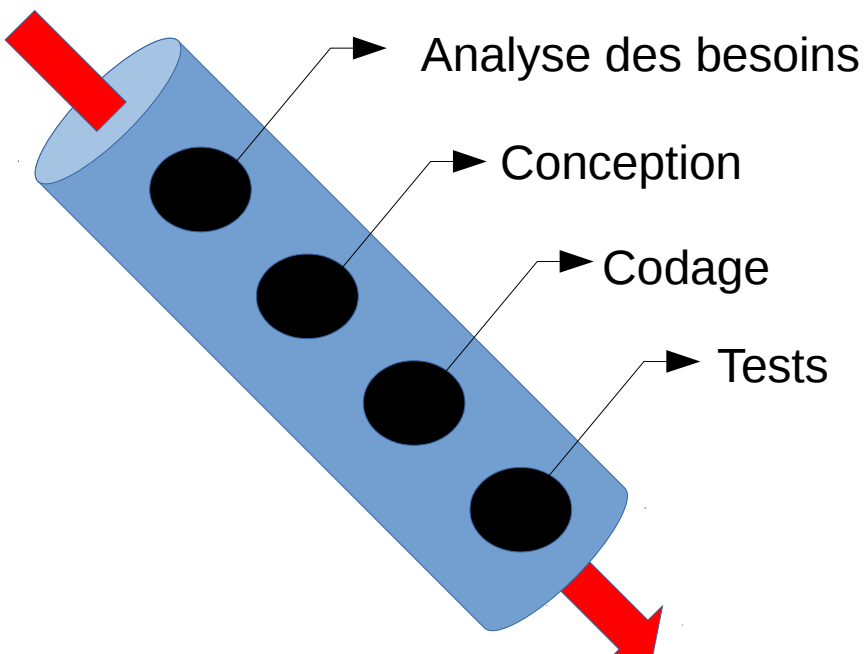
Plan du chapitre

Développement en TP

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

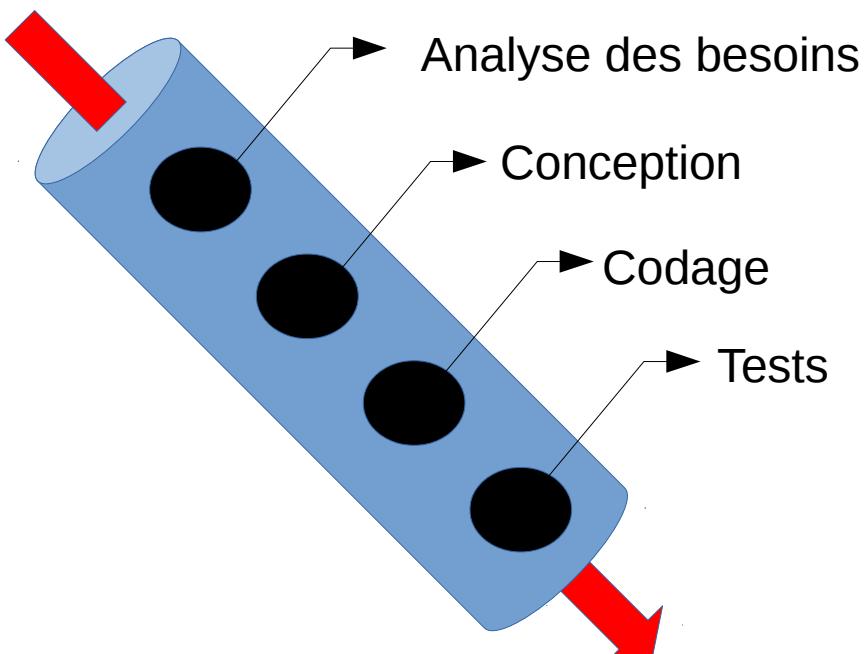
Développement en cascade

- Le cycle de développement en cascade a longtemps été enseigné et utilisé.
  - Il est la cause de l'échec de nombreux projets.
  - Toutefois, il a une vertu pédagogique : il identifie les 4 étapes du développement.



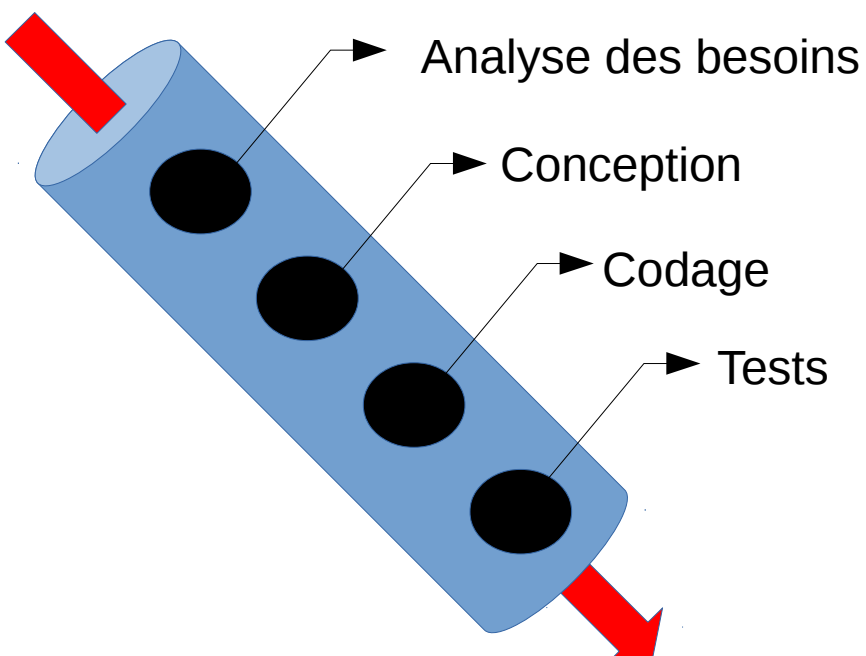
Cascade et effet tunnel

- 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.
  - Et quand on ressort du tunnel, les besoins du client ont changé !



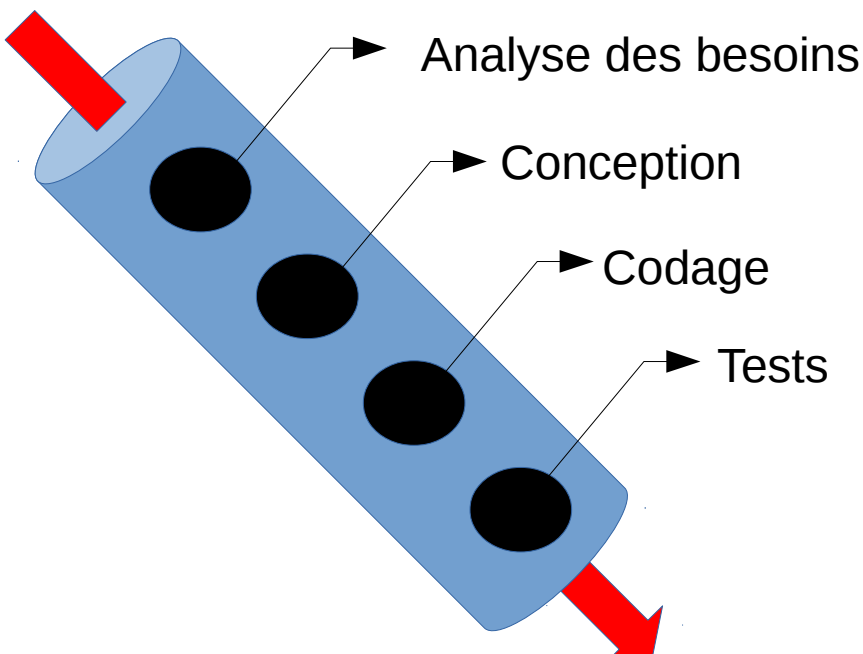
Conception puis codage

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



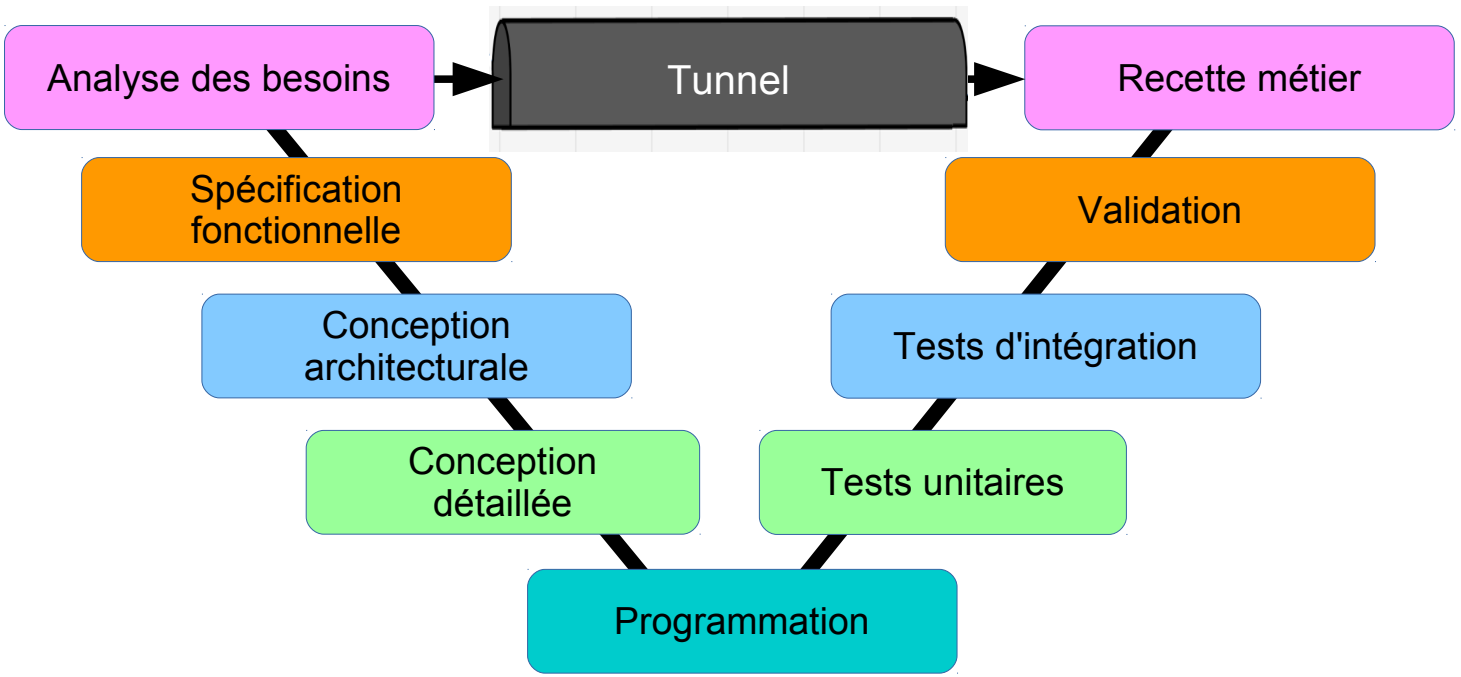
Test après le codage

- 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 perdu les objectifs de ce que l'on veut tester.
  - Les tests sont en général court-circuités pour tenir les délais.



Cycle en V

- L’apport du cycle en V porte sur le moment de la définition des tests.
  - Le meilleur moment pour définir les tests, c’est au moment de la spécification des solutions et de la conception.
- Mais globalement, il n'y a pas de réelle amélioration.
  - P. ex. du point de vue de la MOA, le cycle en V ressemble à un tunnel.



Plan du chapitre

1 Mauvaises pratiques du développement

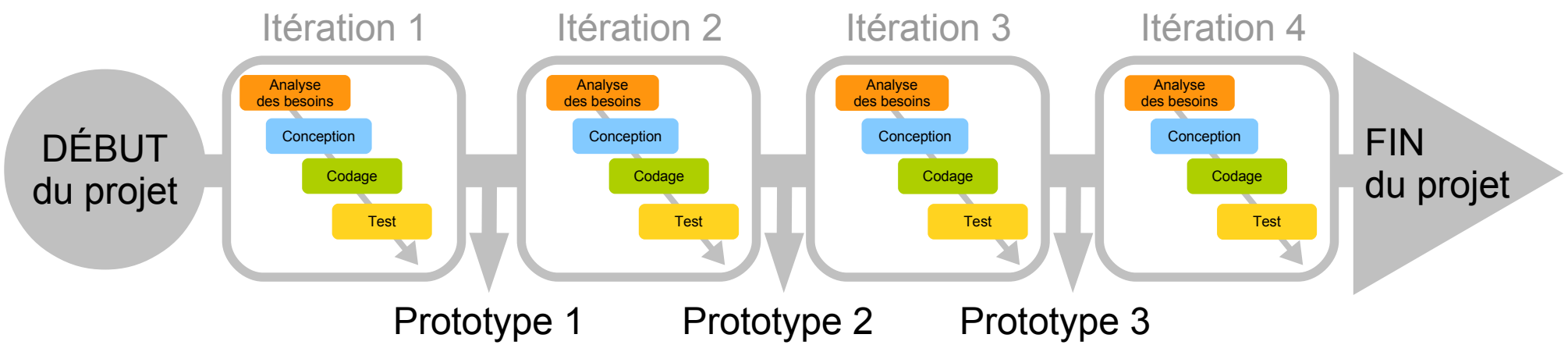
2 Bonnes pratiques du développement

Pratiques actuelles : l'agilité

- Méthodes agiles (principalement *Extreme programming*).
  - Le développement est centré sur le code.
  - Le logiciel est développé **itérativement** par **incréments**.
  - Il s'agit de produire rapidement des versions exécutables minimales du logiciel : des prototypes.
    - « Un intellectuel assis ira toujours moins loin qu'un con qui marche » - Michel Audiard.
  - Il est donné un rôle fondamental aux tests.
  - Intégration continue (quasi-quotidienne).
    - Les productions des développeurs sont rassemblées dans une version du logiciel,
      - opérationnelle dans le futur environnement d'exécution,
      - qui passe tous les tests unitaires.
    - Cf **DevOps**

Cycle de développement itératif

- Itération :
  - Reproduit un cycle en cascade complet,
    - pour un petit sous-ensemble de fonctionnalités seulement.
    - pendant un temps très court (2 semaines).
  - Produire une version du logiciel opérationnel, testé et évalué par le client.
    - Un prototype est une version intermédiaire du logiciel exécuté pour démontrer des concepts ou faire des essais de choix.
  - Réviser les objectifs initiaux à chaque début d'itération avec le client.



Cycle de développement incrémental

- Le logiciel est construit par **incrémentation**.
  - Prototype 0
    - Au moins un ersatz du futur logiciel axé principalement sur les futures interfaces d'interaction avec l'utilisateur.
  - Chaque prototype suivant, ajoute de nouvelles fonctionnalités au prototype précédent.
    - La règle 80 - 20
      - 20 % des cas d'utilisation couvrent 80 % des fonctionnalités du futur logiciel et les 80 % restants ne couvrent que 20 % des fonctionnalités.
      - Il y a évidemment intérêt à commencer par les 20 %.
      - L'erreur de débutant est de se perdre dans les détails.

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 projet non testé.
  - On a toujours une version intermédiaire mais opérationnelle du logiciel à donner au client.
  - Aujourd'hui, même le cycle en V est itératif.
- Incrémentation
  - Donne rapidement de la valeur au produit.
  - Le client voit une évolution rassurante et constante du produit.
  - Réduit le temps de mise sur le marché (*time to market*).

Mise en pratique en TP de C

- Démarche qui s'inspire du développement agile :
  1. Construire l'environnement de développement.
     - Makefile, organisation des dossiers, etc.
  2. Construire l'environnement de test.
     - fichiers de test.
  3. Coder la fonction `main()` vide
     - Compiler, et exécuter pour vérifier l'environnement de test.
  4. Coder une première fonction appelée par le `main()`.
     - Compiler, exécuter et tester.
  5. On poursuit le développement fonction par fonction :
     1. Codage d'une fonction.
     2. Test.

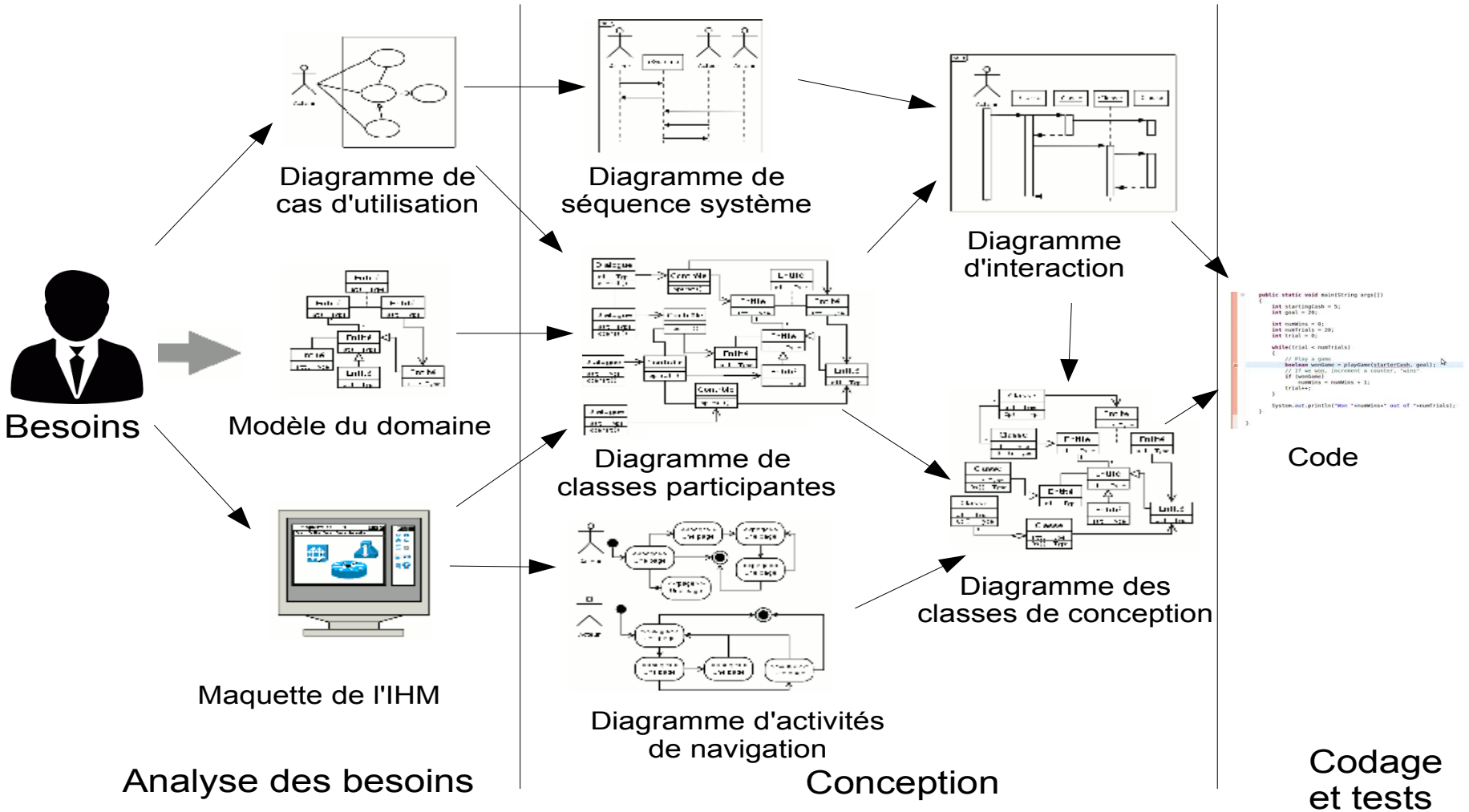
Plan du chapitre

1 Mauvaises pratiques du développement

2 Bonnes pratiques du développement

3 Développement et diagrammes UML

Utilisation des diagrammes UML lors d'une itération



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).
- Modèle du domaine.
  - Une première analyse du domaine à partir des concepts manipulés dans le domaine.

- 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, activités, état).
       - 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.
  - Plusieurs méthodologies à partir du cahier des charges.
    - Débat sur la meilleure approche.
- Or ces modules jouent un rôle important pour la flexibilité, la fiabilité et la robustesse du système.

Exemple : déterminer les classes

- 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**

Exemple : déterminer les classes

- 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 !
  - Attribut : ne pas s'intéresser aux attributs (encapsulation).
  - Méthodes : ne s'intéresser qu'aux services.

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 est une mini-cascade qui enchaîner les étapes d'analyse des besoins, conception, codage et test.
    - Chaque itération doit se terminer par une version opérationnelle et testée du logiciel en cours.
    - Chaque incrément doit ajouter une valeur au prototype en cours.