Conception de classes

Cycle de vie

Cycle de vie d'un logiciel : phases

- ⋆ Analyse de tâches, spécification
- ⋆ Conception
- ⋆ Implémentation
- ⋆ Tests
- ⋆ Déploiement

Phases

Analyse. Quels sont les besoins du client ? On doit définir les tâches, mais non pas la manière dont les faire. Sortie : document spécifiant les fonctionnalités et les critères de performance.

Conception. Analyse de structure du problème. En OO : classes et méthodes principales. Sortie : spécification de classes et méthodes.

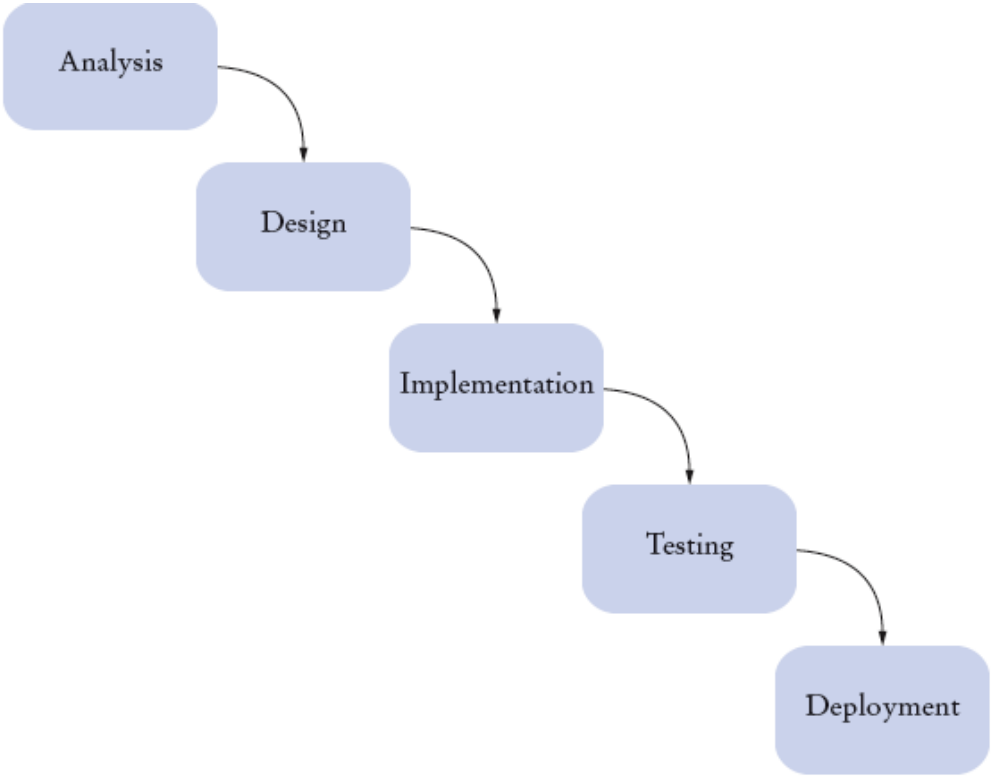
Implémentation. Codage.

Tests. Définis par le client et les développeurs. Sortie : déscription de tests et les résultats.

Déploiment. *Packaging*, installation, distribution.

Cascade

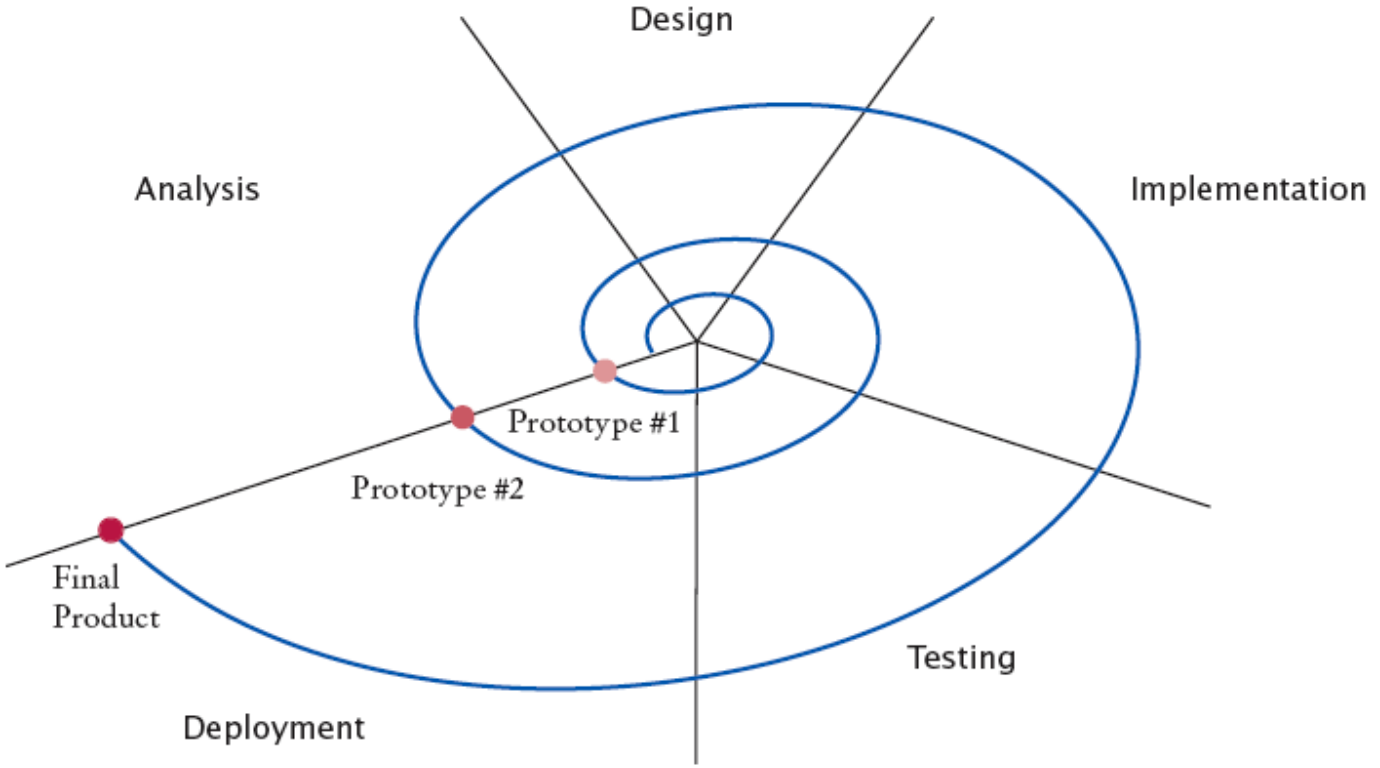
Modèle de cascade (*waterfall*) : séparation idéale de phases



En vérité, il y a beaucoup d'itérations lors du developpement d'un logiciel

Spirale

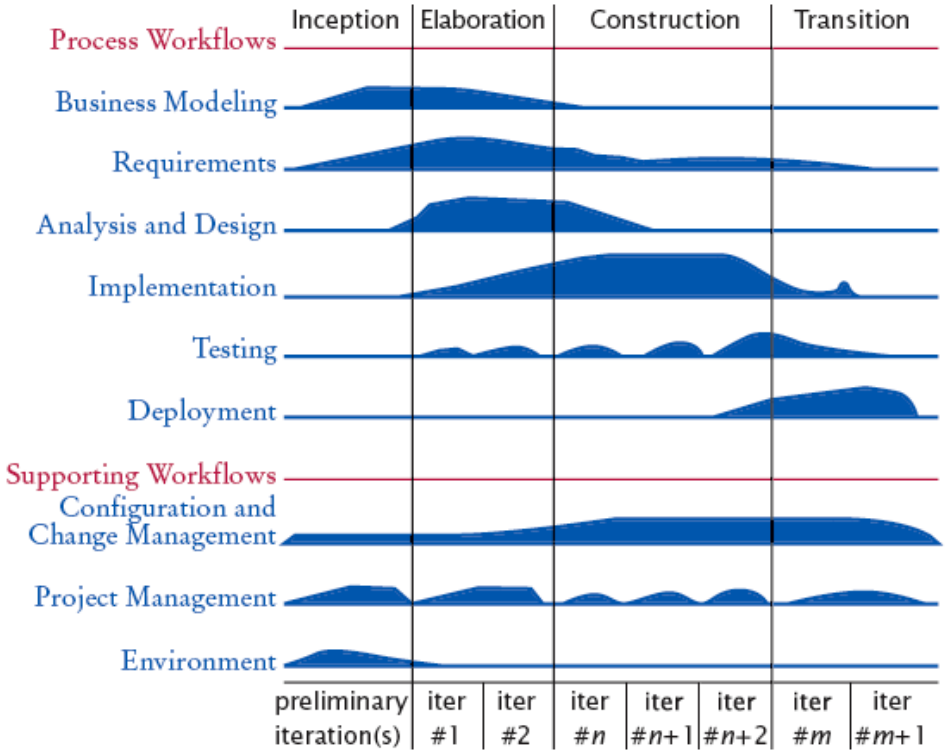
Modèle de spirale : prototypage rapide (p.e., GUI)



OK, mais n'est pas un bon plan à suivre rigidement : quand est-ce qu'on finit ?

En pratique

Le métaphore d'un cascade ou d'une spirale est beau mais c'est mieux d'être plus pragmatique : *Rational Unified Process Methodology*



Conception : découverte de classes

Une classe devrait représenter un concept en soi : cohésion

- classes représentant des entités multiples avec un comportement identique (règle guidant : chercher les noms propres dans la déscription de tâches) : pour commencer, on peut juste écrire une grande liste de candidats. . . Il y aura des classes qu'on découvert seulement plus tard.
- Il n'est pas toujours nécessaire de définir une nouvelle classe pour la représentation (est-ce qu'on peut juste utiliser `String` ?)
- classes utilitaires (p.e., `Math`) : aucune instanciation ou très peu d'instances

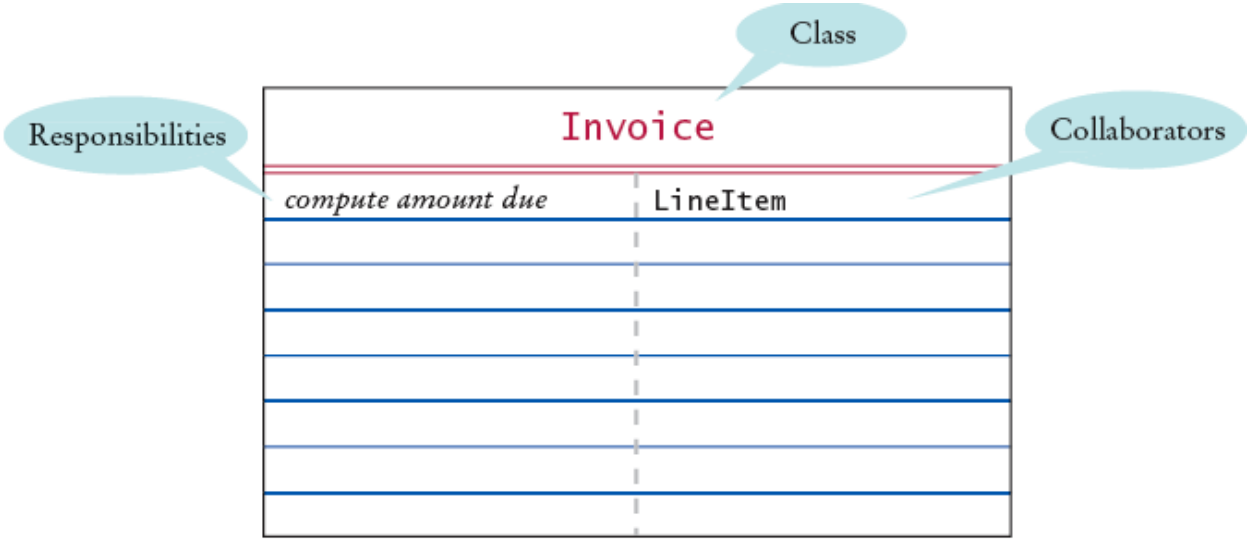
Conception : comportement de classes

Après avoir découvert les classes, il faut définir leur comportement.

1. Quelles sont les tâches ? (chercher les verbes dans la déscription du problème)

2. Assigner les tâches à des classes : responsabilités et collaborateurs

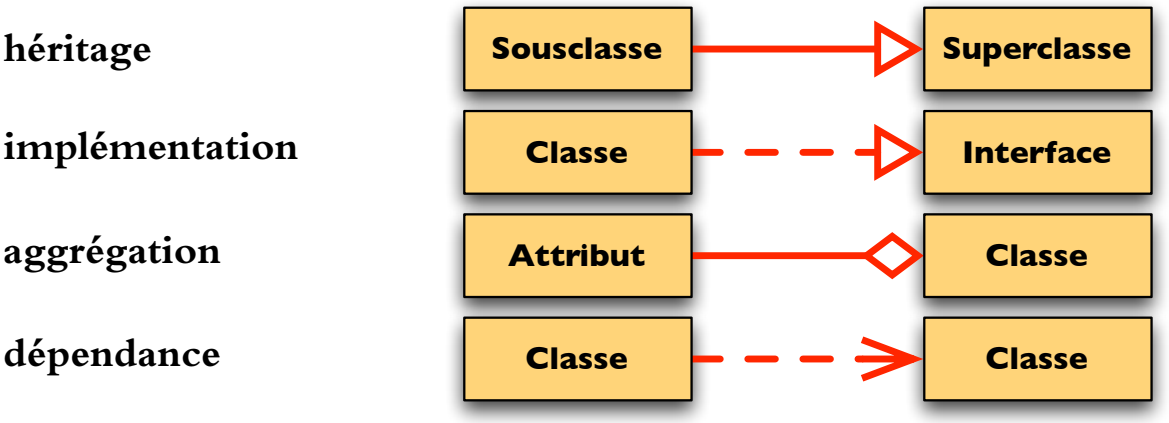
Méthode informelle : cartes CRC — aide à organiser ses idées



Relations entre classes

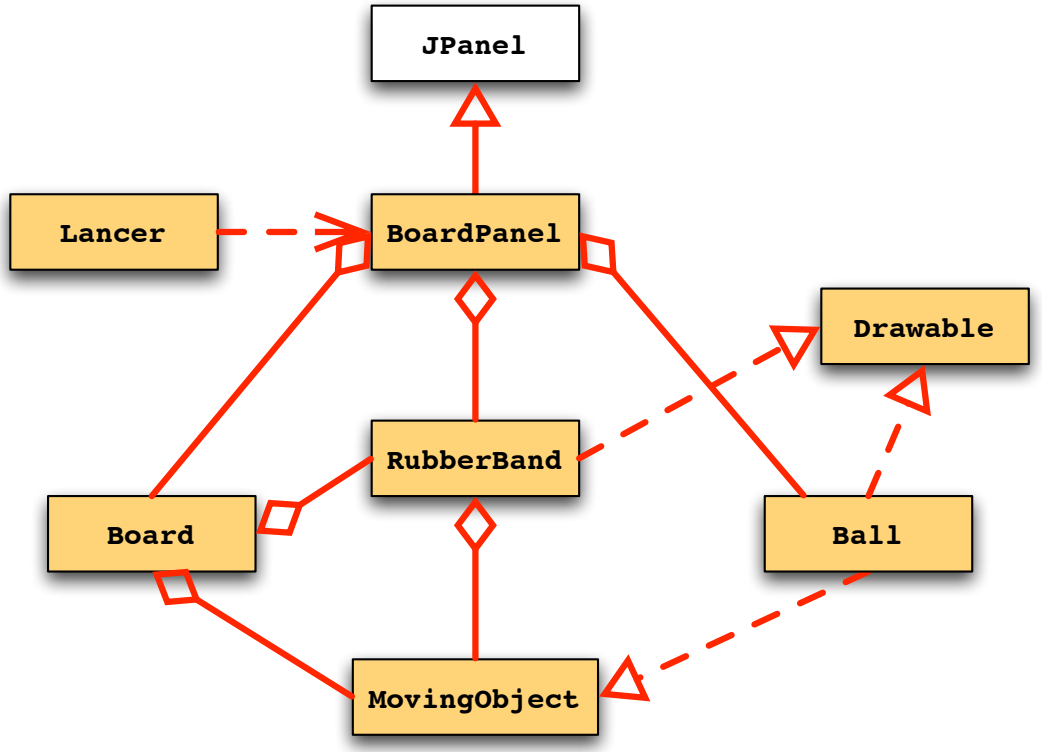
Types de relations : héritage, aggrégation, implémentation, dépendance

Diagramme UML :



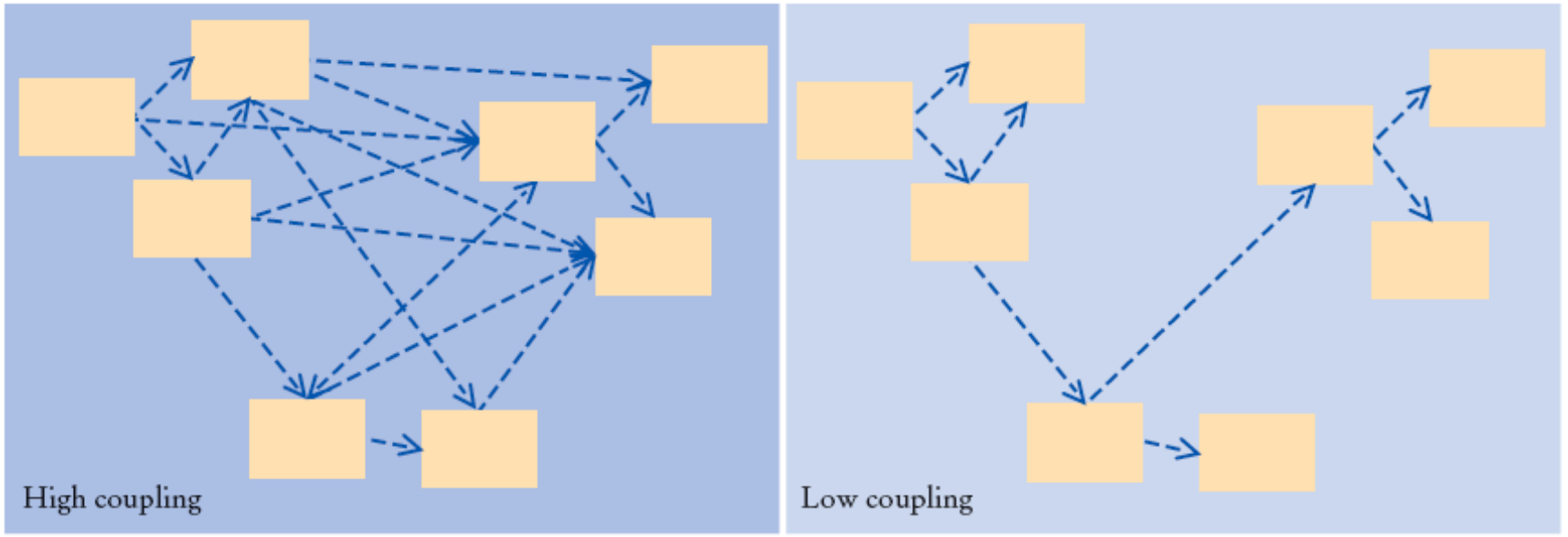
Relations entre classes II

Exemple de Devoir 2



Couplage

Couplage : connectivité du réseau de relations entre les classes



Avec peu de couplage il est plus facile de maintenir le code. . .

Désirable : cohésion forte et couplage faible (devoir 2 n'est pas un bon exemple)

Documentation

Utiliser `javadoc` : génération de documentation en HTML
⇒ syntaxe HTML pour écrire le texte (p.e., `<i>italiques</i>`)

Un commentaire pour javadoc doit commencer par `/**`

On doit les placer avant la ligne de déclaration (pour classes, variables et des méthodes)

Pour documenter le comportement de vos classes, on finit la conception par écrire des fichiers de source vides : commentaires avec la déclaration de méthodes.

Documentation 2

```
/**
   Description de la classe
 */
public class MyClass
{
    /**
        Description du constructeur
      */
    public MyClass(){...}

    /**
        Description de la variable
      */
    public int myvar;

    /**
        Description de la méthode
      */
    public void mymethod(){...}
}
```

Javadoc

Documentation structurée : tags (syntaxe `@nom-de-tag`)

`@param` paramètre d'une méthode
`@return` valeur retournée de la méthode
`@exception` ou **`@throws`** exception lancée par la méthode
`@author` nom du développeur (pour la classe)
`@version` version (classe ou méthode)
`@deprecated` marque la méthode comme dépreciée (avertissement lors de compilation)
`@see` association avec une autre méthode ou classe

Pour toutes les méthodes non-privées : `@param` (chaque paramètre), `@return` (sauf si `void`), `@exception` (sauf `RuntimeException`)

Javadoc 2

```
/**
 * Valide un mouvement de jeu d'Echecs.
 * @param leDepuisFile   File de la pièce à déplacer
 * @param leDepuisRangée Rangée de la pièce à déplacer
 * @param leVersFile     File de la case de destination
 * @param leVersRangée   Rangée de la case de destination
 * @return vrai(true) si le mouvement d'échec est valide
 */
 boolean estUnDéplacementValide(int leDepuisFile,
  int leDepuisRangée, int leVersFile, int leVersRangée)
 {
     ...
 }
```