IFT 1015 - Art de programmer III

Professeur:

Stefan Monnier

B. Kégl, S. Roy, F. Duranleau, S. Monnier

Département d'informatique et de recherche opérationnelle

Université de Montréal

hiver 2006

Au programme

[Niño: 2.3.3, 7-9, 11.3]

Choix de conception

Flot d'un programme orienté objets

Conception de classes

Responsabilités

Relations

Modularité

Tester une classe

Documenter une classe

Appronfondissment: [Niño: 4, **7**-**11**]

Choix de conception

- Il n'y a jamais de solution unique pour résoudre un problème.

Choix de conception

- Il n'y a jamais de solution unique pour résoudre un problème.

$\longrightarrow$ Chacune réalise le travail demandé...

Choix de conception

- Il n'y a jamais de solution unique pour résoudre un problème.

$\longrightarrow$ Chacune réalise le travail demandé...

$\longrightarrow$ *Cependant*, les solutions ne sont pas toutes de même qualité.

Choix de conception

- Il n'y a jamais de solution unique pour résoudre un problème.

$\longrightarrow$ Chacune réalise le travail demandé...

$\longrightarrow$ *Cependant*, les solutions ne sont pas toutes de même qualité.

$\Rightarrow$ On choisit selon plusieurs critères: ***efficacité*** (i.e. rapidité d'exécution et/ou consommation de mémoire), ***extensibilité*** (i.e. pouvoir facilement modifier ou ajouter des fonctionnalités), ***compréhensibilité***, ...

Choix de conception

- Il n'y a jamais de solution unique pour résoudre un problème.

$\longrightarrow$ Chacune réalise le travail demandé...

$\longrightarrow$ *Cependant*, les solutions ne sont pas toutes de même qualité.

$\Rightarrow$ On choisit selon plusieurs critères: ***efficacité*** (i.e. rapidité d'exécution et/ou consommation de mémoire), ***extensibilité*** (i.e. pouvoir facilement modifier ou ajouter des fonctionnalités), ***compréhensibilité***, ...

- Bien concevoir un programme n'est pas une tâche facile (par où commencer, l'évaluation de la qualité est ambiguë, ...).

Choix de conception

- Il n'y a jamais de solution unique pour résoudre un problème.

$\longrightarrow$ Chacune réalise le travail demandé...

$\longrightarrow$ *Cependant*, les solutions ne sont pas toutes de même qualité.

$\Rightarrow$ On choisit selon plusieurs critères: ***efficacité*** (i.e. rapidité d'exécution et/ou consommation de mémoire), ***extensibilité*** (i.e. pouvoir facilement modifier ou ajouter des fonctionnalités), ***compréhensibilité***, ...

- Bien concevoir un programme n'est pas une tâche facile (par où commencer, l'évaluation de la qualité est ambiguë, ...).

$\longrightarrow$ C'est un art en soi, et une grande partie du génie logiciel.

Procédures vs Objets

Programmation procédurale

- Le programme est fractionné en un ensemble de fonctions.
- Chacune réalise une action.
- Un programme est fait de fonctions qui s'appellent et passent les infos (pour une bonne conception) en paramètres.

Programmation orientée objets

- Le programme est fractionnés en un ensemble de classes.
- Chaque méthode des classes réalise une action.
- Encapsulation des infos dans les instances des classes.
- Une programme est fait d'objets qui communiquent entre eux.

Flot d'un programme orienté objets

Quand crée-t-on les objets? À quoi ressemble le programme principal?

Flot d'un programme orienté objets

Quand crée-t-on les objets? À quoi ressemble le programme principal?

$\longrightarrow$ (**Rappel**) Tout commence dans une fonction `main`.

$\Rightarrow$ Au moins un objet doit être créé ici.

$\longrightarrow$ (**Rappel**) Une méthode d'une classe réalise une action.

$\Rightarrow$ Les actions du programme principal sont exécutées par des appels à des méthodes des objets créés dans `main`.

En bref, c'est la même chose qu'en programmation procédurale, sauf:

- les informations initiales se trouvent dans des objets au lieu de variables locales à `main`;
- les fonctions sont des méthodes appelées par les objets.

Exemple simple

Calcul de la circonférence d'un cercle.

```
public class CalculCirc {
  public static void main (String[] arg) {
    Cercle c = new Cercle(); // Instanciation.
    c.lire();          // Action: lecture au clavier des donnr
    c.afficheCirc(); // Action: calcul et affichage de la ci
} }
```

Approche procédurale équivalente:

```
public class CalculCirc {
  public static void main (String[] arg) {
    double rayon;           // "Instanciation" des données.
    rayon = Cercle.lire(); // Action: lecture au clavier des
    Cercle.afficheCirc(rayon); // Action: calcul + aff. de c
} }
```

Cas général

Doit-on instancier tous les objets dans `main`?

Cas général

Doit-on instancier tous les objets dans `main`?

$\longrightarrow$ **NON!!!** Seulement dans les programmes très simple.

Dans le cas général, on crée typiquement une classe qui gère l'ensemble du programme.

$\Rightarrow$ Dans `main`, on crée une instance de cette classe et on appelle une ou des méthodes pour exécuter le programme.

On peut appeler un tel objet un objet d'application ou de contrôle.

Exemple

Voici un programme principal simple, très générique et typique pour un programme orienté objets quelconque:

```
public class Main
{
  public static void main (String[] arg)
  {
    // Création et initialisation des données.  Le construct
    // s'occupera de créer les autres objets initiaux.
    Application app = new Application(arg);

    // Méthode qui exécute le programme principal.
    app.exec();
} }
```

On choisit habituellement un nom plus significatif que **`Application`**.

Conception de classes

Qu'elles sont les classes qu'on devrait créer?

Qu'elles seront leurs propriétés? Leurs commandes?

Conception de classes

Qu'elles sont les classes qu'on devrait créer?

Qu'elles seront leurs propriétés? Leurs commandes?

$\longrightarrow$ Il n'y a pas de recette miracle pour les identifier.

Conception de classes

Qu'elles sont les classes qu'on devrait créer?

Qu'elles seront leurs propriétés? Leurs commandes?

$\longrightarrow$ Il n'y a pas de recette miracle pour les identifier.

Généralement, les classes principales ressortent intuitivement.

Conception de classes

Qu'elles sont les classes qu'on devrait créer?

Qu'elles seront leurs propriétés? Leurs commandes?

$\longrightarrow$ Il n'y a pas de recette miracle pour les identifier.

Généralement, les classes principales ressortent intuitivement.

E.g.: Dans un jeu d'échec, on peut clairement identifier les classes suivantes: l'échiquier, les pièces et les joueurs.

Conception de classes

Qu'elles sont les classes qu'on devrait créer?

Qu'elles seront leurs propriétés? Leurs commandes?

$\longrightarrow$ Il n'y a pas de recette miracle pour les identifier.

Généralement, les classes principales ressortent intuitivement.

E.g.: Dans un jeu d'échec, on peut clairement identifier les classes suivantes: l'échiquier, les pièces et les joueurs.

Il faut aussi penser aux classes de gestion, i.e. ce qui gère le flot général du programme.

Conception de classes

Qu'elles sont les classes qu'on devrait créer?

Qu'elles seront leurs propriétés? Leurs commandes?

$\longrightarrow$ Il n'y a pas de recette miracle pour les identifier.

Généralement, les classes principales ressortent intuitivement.

E.g.: Dans un jeu d'échec, on peut clairement identifier les classes suivantes: l'échiquier, les pièces et les joueurs.

Il faut aussi penser aux classes de gestion, i.e. ce qui gère le flot général du programme.

E.g.: Pour le jeu d'échec, on aurait aussi besoin d'une classe pour le jeu même, i.e. création des autres objets, exécution du jeu, ...

Conception de classes

Puis on réfléchit à l'interaction des objets, au système en général, et souvent d'autres concepts de classes vont ressortir, souvent pour des utilitaires simples.

E.g.: Pour le jeu d'échec, il pourrait y avoir une classe pour une position dans l'échiquier.

En bref, une approche *top-down* (haut en bas) peut souvent aider.

Responsabilités

Lors de la conception d'une classe, on doit réfléchir à trois types de responsabilités des objets:

De connaissance

Quelles informations l'objet doit-il connaître?

$\longrightarrow$ Ceci définit les attributs de la classe.

D'action

Quelles sont les tâches que l'objet doit accomplir?

$\longrightarrow$ Ceci définit les méthodes de la classe.

De création

L'objet doit-il créer d'autre objets?

$\longrightarrow$ Ceci définit si un attribut étant un objet doit être instancié dans la classe ou créé ailleurs et passé en paramètre.

Responsabilités: exemple (1)

Pour le jeu d'échec, voici quelques exemples de responsabilités pour les classes identifiées précédemment:

Pièce

Connaissance:	type, couleur
Requêtes:	type et couleur, peut-elle faire un mouvement donné
Commandes:	aucune
Création:	aucune.

Responsabilités: exemple (2)

Joueur

Connaissance:	couleur, échiquier
Requêtes:	couleur
Commandes:	jouer un coup
Création:	aucune

Échiquier

Connaissance:	grille de l'échiquier, les pièces
Requêtes:	pièce à une position donnée
Commandes:	bouger une pièce
Création:	grille, les pièces

Responsabilités: exemple (3)

Jeu

Connaissance:	échiquier, joueurs
Requêtes:	aucune *a priori*
Commandes:	exécution du jeu
Création:	échiquier, les joueurs

On pourrait ajouter un utilitaire: une classe pour une position, ainsi une position est manipulées comme une entité plutôt que comme 2 entités implicitement liées. (laissé en exercice).

Concevoir avant d'écrire

Certains choix de l'exemple peuvent paraître arbitraires. C'est un premier jet.

En développant davantage sur la conception du programme, on peut se rendre compte de certains mauvais choix et adapter la solution.

$\Rightarrow$ ***D'où l'importance de l'étape de la conception-même avant de se lancer dans l'écriture du programme!!***

$\Rightarrow$ Évite de modifier constamment le programme et risquer d'y insérer des bogues souvent subtils.

Relations

Dans la conception d'un programme, il est aussi important d'établir les relations entre les objets. Pourquoi?

Relations

Dans la conception d'un programme, il est aussi important d'établir les relations entre les objets. Pourquoi?

- $\longrightarrow$ Aide à clarifier le rôle de chaque objet;
- $\longrightarrow$ met en évidence les dépendances entre les objets;
- $\longrightarrow$ peut aider à mieux comprendre le flot des informations et du programme.
- $\longrightarrow$ peut aider à déceler les dangers d'effets de bord;
- $\longrightarrow$ ...

Les responsabilités de chaque objet déterminent plusieurs des relations entre les objets.

Types de relations

Voici les plus fréquents:

Utilisation: Un objet fait usage de méthodes d'un autre objet.

Appelée aussi relation *client-serveur*

Aggrégation: Un objet possède une référence sur un autre

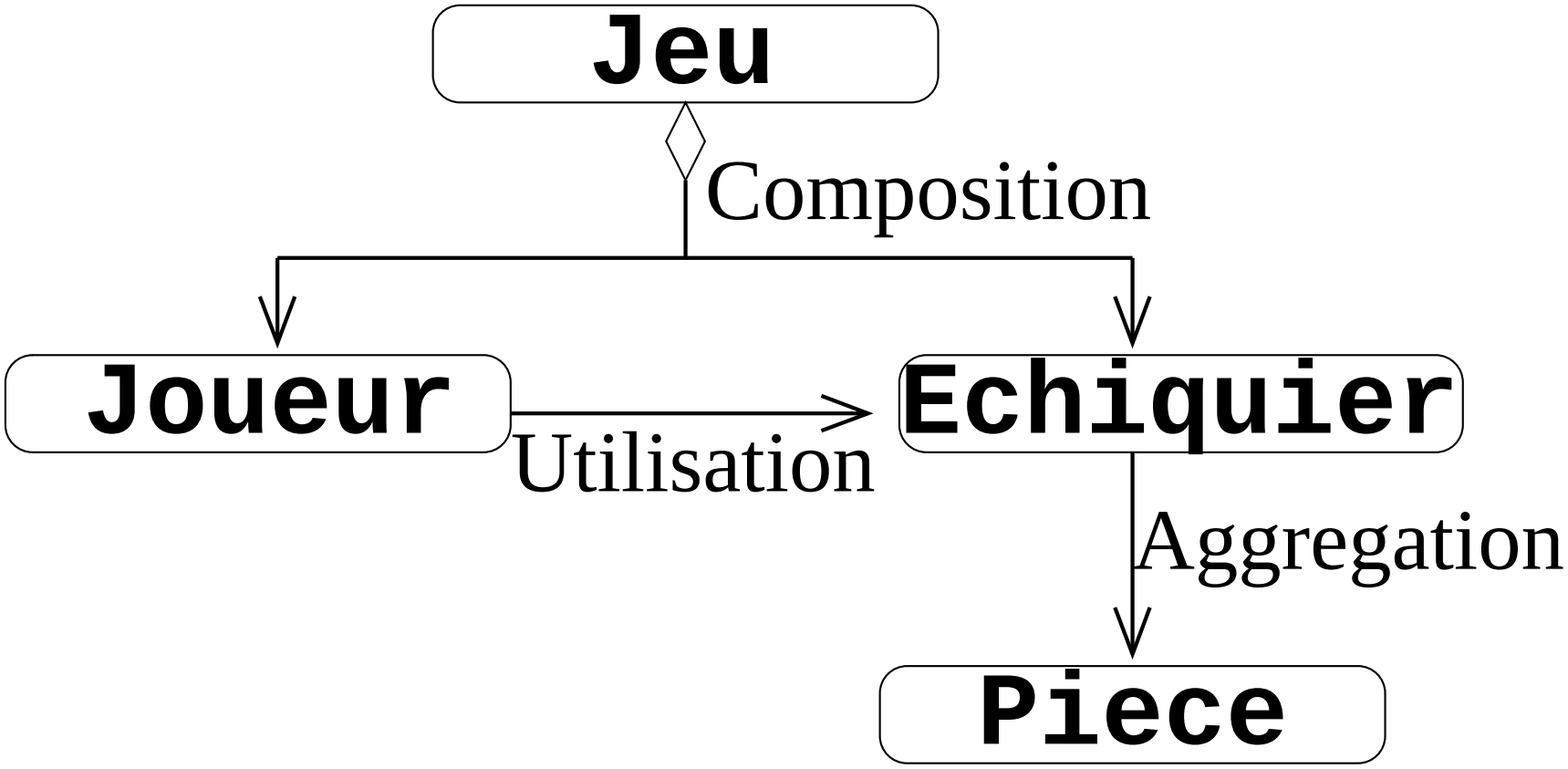
Composition: Un objet fait partie intégrante d'un autre objet

Abstraction: Relation hiérarchique entre classes. Inverse de *spécialisation*.

Pas sujet à ce cours.

Lors de la conception, on représente habituellement ces relations schématiquement (il existe des formalismes pour ça, tel UML).

Quelques relations pour notre jeu d'échec



Explications

Le jeu est composé de son échiquier et ses deux joueurs

$\Rightarrow$ *composition*.

L'échiquier a un ensemble de pièces sur sa grille

$\Rightarrow$ *aggrégation*. On pourrait considérer ça aussi de la *composition* si on considère que l'échiquier et ses pièces forment un tout. L'échiquier sera sûrement aussi en relation d'*utilisation* avec une pièce via les méthodes de requêtes.

Le joueur utilise l'échiquier pour jouer ses coups

$\Rightarrow$ *utilisation*. En fait, le joueur est aussi probablement en *aggrégation* avec l'échiquier car il doit connaître l'échiquier sur lequel il joue.

Relations

Le choix des relations peut être ambigü.

$\Rightarrow$ Le choix de conception aussi.

$\longrightarrow$ Grande partie du génie logiciel.

Pour l'instant, l'important est de voir les liens essentiels.

Remarques

aggrégation $\not\Rightarrow$ *utilisation*

Ex.: Un attribut `String` dont on ne fait que consulter la valeur (requête sur l'objet) ou l'afficher.

aggrégation $\not\Rightarrow$ responsabilité de création

Ex.: Presque tout usage de `String`.

utilisation $\not\Rightarrow$ *aggrégation*

Ex.: Utilisation de méthodes d'un objet passé en paramètre à une méthode mais sans conserver la référence dans l'objet.

Et évidemment:

composition $\Rightarrow$ *aggrégation*

composition $\Rightarrow$ responsabilité de création (généralement)

Modularité

Les concepts de *cohésion* et *couplage* s'appliquent tout autant aux classes qu'aux fonctions.

Cohésion d'une classe:

Une classe est *cohésive* si elle encapsule une *seule* notion.

Ex.: `Cercle`, `Piece`, `Joueur`, `Echiquier`

Couplage d'une classe:

Le *couplage* d'une classe est son degré d'interaction avec d'autres classes, autrement dit la quantité de relations.

$\Rightarrow$ Une classe devrait être le plus modulaire possible, i.e. avoir une grande *cohésion* et un faible *couplage* ($\Rightarrow$ peu de *relations*).

Parce que...

Plus ***réutilisable***

Plus compréhensible

Localité des modifications au code

[$\Rightarrow$] réduit le potentiel d'insertions de bogues

Localité des changements à l'éxécution (i.e. impact d'une variable qui change de valeur)

[$\Rightarrow$] réduit les dangers d'effets de bord

Exemple échiquier: code original (1)

```
public class Echiquier
{
  public static final int LIBRE = -1; // Indique une case vi

  // Valeur des pièces dans les cases de la grille
  public static final int PION = 0;
  public static final int TOUR = 1;
  ... // etc

  // Valeur à (i,j) indique le type de la pièce.
  private int[][] grilleType = new int[8][8];

  // Valeur à (i,j) indique la couleur de la pièce.
  private boolean[][] grilleCouleur = new boolean[8][8];
```

Exemple échiquier: code original (2)

```
public Echiquier () { ... // Initialise les grilles  }

public int litTypePiece (int i, int j)
{ return grilleType[i][j]; }

public boolean litCouleurPiece (int i, int j)
{ return grilleCouleur[i][j]; }

public boolean bougerPiece (int i0, int j0, int i1, int j1

...
```

Elle renferme deux notions: échiquier et pièce. Mieux vaut séparer les deux et alléger le code de chaque classe.

Exemple échiquier: 2e essai (échiquier)

```
public class Echiquier
{
  // Un null indiquera un trou.
  private Piece[][] grille = new Piece[8][8];

  public Echiquier () { // Initialise grille.  }

  public Piece litPiece (int i, int j)
  { return grille[i][j]; }

  public boolean bougerPiece (int i0, int j0, int i1, int j1

  ...
}
```

Exemple échiquier: 2^{e} essai (pièce)

```
public class Piece
{
  // Type des pièces.
  public static final int PION = 0;
  public static final int TOUR = 1;
  ... // etc

  private int type;          // Type de la pièce.
  private boolean couleur; // Couleur de la pièce.

  public Piece (int type, boolean couleur) { ... // Init. }
  public int litType () { return type; }
  public boolean litCouleur () { return couleur; }
  ...
}
```

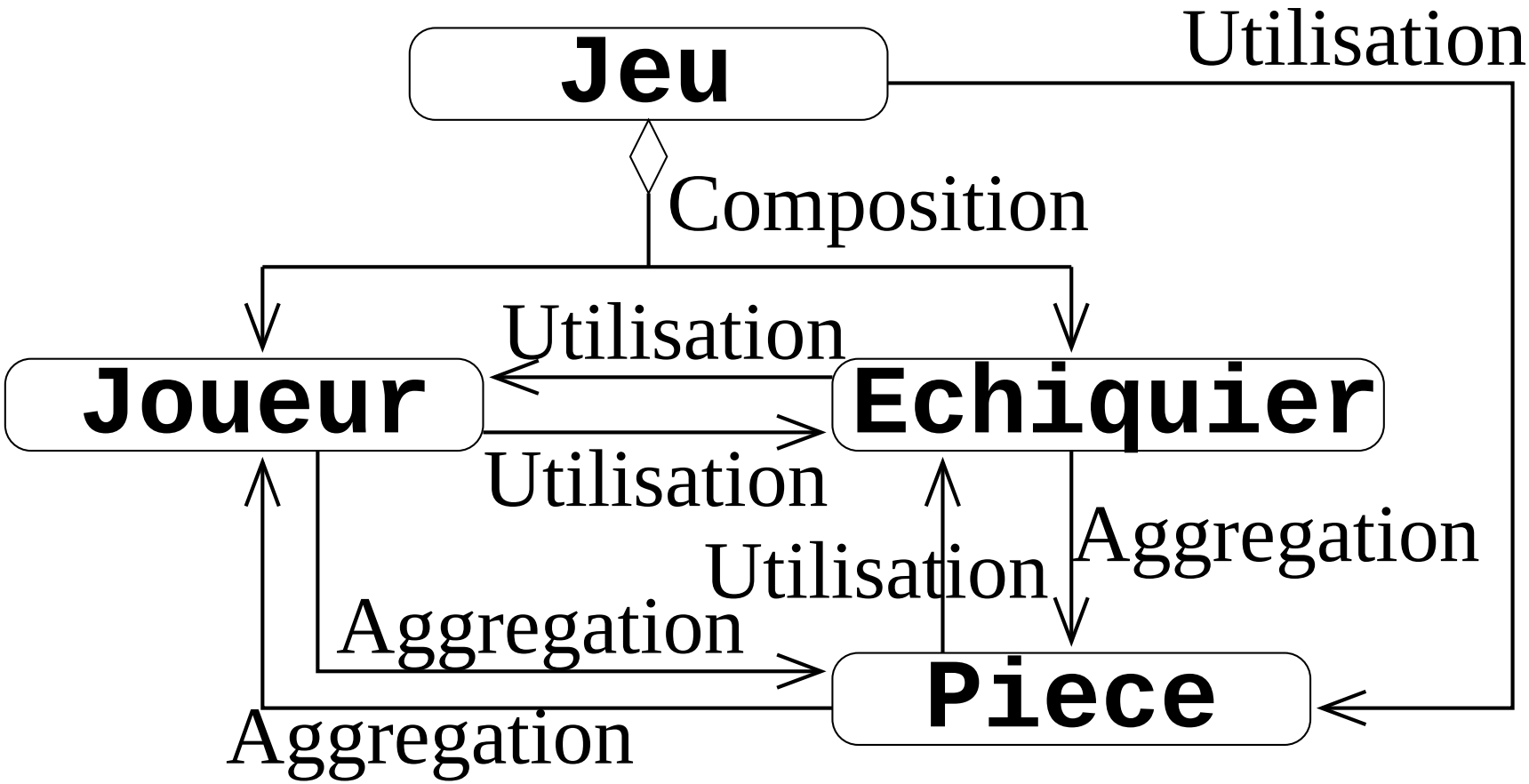
Exemple échecs: alternatives de conception

- au lieu d'avoir un attribut couleur pour une pièce, elle pourrait avoir une référence sur le joueur à qui elle appartient (*aggrégation*);

$\Rightarrow$ l'échiquier doit connaître les joueurs (à la création) (*utilisation*);

- l'entièreté du déplacement des pièces peut être fait dans la classe `Piece`, auquel cas elle a besoin d'*utiliser* l'échiquier;
- la validation du choix d'une pièce à déplacer pourrait être faite au moment où le joueur fait son coup $\Rightarrow$ il doit connaître les pièces qui lui reste (*aggrégation*);
- enfin, pour terminer le jeu, le roi d'un joueur doit-être échec et mat; si ce test est dans la classe du jeu, elle *utilise* les pièces.

Exemple échecs: relations alternatives



Exemple échec: moins de couplage

Toutes ces relations additionnelles peuvent être éliminées avec un meilleur choix de conception, et en considérant la *transitivité* dans les relations.

Exemple:

- Le jeu connaît l'échiquier qui connaît ses pièces $\Rightarrow$ via l'échiquier le jeu peut savoir l'état d'échec et mat.
- De même pour le joueur et ses pièces, le joueur peut consulter l'échiquier pour valider un coup au lieu de tout faire.

$\Rightarrow$ si on représente la couleur par un type à part (un booléen, par exemple), plus besoin du lien entre pièce et joueur, et échiquier et joueur.

Tester une classe

- On teste une classe presque exactement comme on teste une fonction, i.e. on se crée une classe à part avec une fonction `main` qui crée une instance de la classe et appelle ses méthodes.
- Cependant, un objet a généralement plus d'une méthode $\Rightarrow$ le teste sera nécessairement plus complexe.
- Dépendant de la nature des relations de la classe, il faut peut-être faire usage d'autres classes.

Remarque: Plus un test doit être complexe ou utiliser d'autres classes, plus on doit se poser des questions sur la modularité de la classe testée.

Documenter une classe

Devoir consulter le code d'une classe pour savoir comment l'utiliser n'est pas une façon efficace de travailler.

$\Rightarrow$ Il faut documenter son code.

Une méthode de plus en plus populaire: employer un formalisme dans la façon d'écrire ses commentaires et utiliser un programme qui génère automatique une documentation (en format `pdf`, `html`, `...`) en analysant le code (e.g. Lisp *docstrings*, `WEB`, `javadoc`, `doxygen`).

Attention: l'automatisme est seulement dans l'extraction et l'organisation des commentaires spéciaux. Il faut soi-même écrire ce que chaque classe représente, ce chaque méthode fait, le rôle de chaque attribut.

Quoi écrire ?

Le *rôle* d'une classe, son fonctionnement *général*.

La *tâche* accomplie par une méthode, le *rôle* de chaque paramètre.

Le *rôle* d'un attribut.

Que *représente* la constante.

Comment un constructeur initialise un objet.

On ne parle jamais des détails caché de la classe (le code, ce qui est privé), sauf peut-être pour documenter certaines méthodes *internes*.

$\Rightarrow$ Toujours rester d'un point de vue *conceptuel*.

Préconditions

Une bonne pratique est de documenter, s'il y a lieu, les responsabilités de l'utilisateur (*préconditions*) avant d'appeler une méthode.

Exemples:

- Indiquer une contrainte sur la valeur d'un paramètre.
- Indiquer une contrainte sur l'état d'un objet, i.e. contrainte sur l'ordre d'appel de certaines méthodes.

Postconditions

Une autre bonne pratique est de documenter les garanties (*postconditions*) offertes par une méthode.

Exemples:

- Indiquer la plage de valeurs pour un retour d'une méthode de requête.
- Indiquer les répercursions sur l'état d'un objet (si on change le rayon d'un cercle, la méthode peut assurer que le rayon sera toujours positif).

Ceci est très important pour éviter les effets de bord produits par le fait que l'utilisateur ne connait pas tous les impacts de l'appel d'une méthode.

Exemple échec: esquisse des pièces (1)

```
/**
 * Représente une pièce dans un jeu d'échec. Une pièce possède une
 * couleur et un type, qui gouverne la légalité d'un coup.
 */
public class Piece
{
  /** Couleurs possibles.  */
  public static final boolean BLANC = true;
  public static final boolean NOIR = false;

  /** Valeur du type pour un pion.  */
  public static final int PION = 0;

  /** Valeur du type pour une tour.  */
  public static final int TOUR = 1;

  ... // etc.
```

Exemple échec: esquisse des pièces (2)

```
/** Type de la pièce.  */
private boolean couleur;

/** Type de la pièce (la valeur doit etre l'une des constantes).
private int type;

/**
 * Construit une pièce d'une couleur et de type donnés.
 * @param couleur  Piece.BLANC ou Piece.NOIR.
 * @param type     La valeur doit être l'une des constantes
 *                 entières de cette classe.
 */
public Piece (boolean couleur, int type)
{
  this.couleur = couleur;
  this.type = type;
  ... // Validation du type.
}
```

Exemple échec: esquisse des pièces (3)

```
  /** Retourne la couleur de la pièce (Piece.BLANC ou Piece.NOIR).
  public boolean litCouleur () { return couleur; }

  /** Retourne le type de la pièce.  */
  public int litType () { return type; }


  /**
   * Indique si le déplacement de la position pos0 à pos1 est
   * légal pour la pièce selon les règles.
   */
  public boolean coupLegal (Position pos0, Position pos1)
  {
    ... // si type == UN_TYPE alors... sinon si... etc.
  }

  ...
}
```

Exemple échec: esquisse du joueur (1)

```
/**
 * Classe représentant un joueur.
 * Un joueur déplace ses pièces sur l'échiquier.
 */
public class Joueur
{
  /** Couleur du joueur (i.e. de quelle couleur sont ses pièces).  *
  boolean couleur;

  /** Échiquier sur lequel le joueur joue.  */
  Echiquier echiquier;
```

Exemple échec: esquisse du joueur (2)

```
/**
 * Construit un joueur avec sa couleur et l'échiquer sur
 * lequel il joue.
 * @param couleur  Couleur des pièces: Piece.BLANC ou Piece.NOIR.
 * @param e        L'échiquier sur le quel le joueur joue.
 *                 Sa valeur ne doit pas être null.
 */
public Joueur (boolean couleur, Echiquier e)
{
  this.couleur = couleur;
  echiquier = e;
  ... // valider que e != null
}

/** Retourne la couleur des pièces du joueur.  */
public boolean litCouleur () { return couleur; }
```

Exemple échec: esquisse du joueur (3)

```
  /**
   * Joue un coup pour le joueur.  Devra demander le coup à jouer
   * interactivement.
   * @return True si le coup est légal, false sinon.
   */
  public boolean jouerCoup ()
  {
    ... // demander et jouer le coup déplacer la pièce sur l'échiqui
  }

  ...
}
```

Exemple échec: esquisse de l'échiquier (1)

```
/**
 * Représente un échiquier avec ses pièces.  La rangée 0 est du côté
 * des blancs et la colonne 0 correspond à la droite des blancs.
 */
public class Echiquier {
  /**
   * Grille de l'échiquier.  Une valeur null à une position (i,j)
   * indique qu'il n'y a pas de pièce à cette position.
   */
  private Piece[][] grille;

  /** Construit un échiquier avec ses pièces en position initiale.
  public Echiquier ()
  {
    grille = new Piece[8][8];
    grille[0][0] = new Piece(Piece.BLANC, Piece.TOUR);
    ... // etc.
  }
```

Exemple échec: esquisse de l'échiquier (2)

```
  /**
   * Retourne la pièce à la position donnée,
   * ou null s'il n'y en a pas.
   */
  public Piece litPiece (Position pos)
  { return grille[pos.litI()][pos.litJ()]; }

  /**
   * Déplace la pièce se trouvant à pos0 vers pos1.
   * @return true si le coup a été fait, false sinon (coup illégal).
   */
  public boolean deplacePiece (Position pos0, Position pos1)
  {
    ... // tester légalité du coup, déplacer
  }
  ...
}
```

Exemple échec: esquisse du jeu (1)

```
/**
 * Gère et exécute le jeu d'échec.
 */
public class Jeu
{
  /** Échiquier du jeu.  */
  private Echiquier echiquier;

  /** Joueur blanc.  */
  private Joueur joueurBlanc;

  /** Joueur noir.  */
  private Joueur joueurNoir;
```

Exemple échec: esquisse du jeu (2)

```
  /** Crée un jeu avec son échiquier et ses deux joueurs.  */
  public Jeu ()
  {
    echiquier = new Echiquier();

    // Remarque: Piece.XX crée une dépendance supplémentaire entre
    // Jeu et Piece, mais assure l'unicité de la représentation.
    joueurBlanc = new Joueur(Piece.BLANC, echiquier);
    joueurNoir = new Joueur(Piece.NOIR, echiquier);
  }

  /** Exécute une partie.  */
  public void exec ()
  {
    ... // Le jeu: chaque joueur joue à tour de rôle tant que l'un n
  }
  ...
}
```