

# IFT 1010 - Programmation 1

## Art de programmer 3

Sébastien Roy & François Duranleau



Département d'informatique et de recherche opérationnelle  
Université de Montréal  
automne 2004

## 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

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

1

## Choix de conception

- Il n'y a jamais de solution unique pour résoudre un problème.
- Chacune réalise le travail demandé...
- *Cependant*, les solutions ne sont pas tous de même qualité.
- ⇒ 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**, *etc.*
- Bien concevoir un programme n'est pas une tâche facile (par où commencer, l'évaluation de la qualité est ambiguë, *etc.*).
- C'est un art en soi, et une grande partie du génie logiciel.

2

## Rappel

### Programmation procédurale

- Le programme est fractionnés en un ensemble de fonctions.
- Chacune réalise une action.
- L'ensemble du programme est une suite logique d'appels de fonctions où les informations (pour une bonne conception) sont passées en paramètres.

### Programmation orientée objets

- Le programme est fractionnés en un ensemble de classes.
- Chaque méthode des classes réalise une action.
- Les informations sont encapsulées dans les instances des classes (objets).
- L'ensemble du programme correspond à des objets qui communiquent entre eux.

3

## Flot d'un programme orienté objets

- Quand crée-t-on les objets ? À quoi ressemble le programme principal ?
- **(Rappel)** Tout commence dans une fonction `main`.
- ⇒ Au moins un objet doit être créé ici.
- **(Rappel)** Une méthode d'une classe réalise une action.
- ⇒ 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 que :
  - les informations initiales se trouvent dans des objets au lieu de variables locales à `main` ;
  - les fonctions sont des méthodes appelées par les objets (contexte d'appel).

4

## Cas général

- Doit-on instancier tous les objets dans `main` ?
- **NON!!!** Seulement dans les programmes très simple.
- Dans le cas général, on crée typiquement une classe qui gère l'ensemble du programme.
- ⇒ 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.

6

## 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 données
        c.afficheCirc();        // Action: calcul et affichage de la circonférence
    }
}
```

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 données
        Cercle.afficheCirc( rayon ); // Action: calcul + aff. de circ.
    }
}
```

5

## 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 constructeur
        // 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`.

7

## Conception de classes

- Qu'elles sont les classes qu'on devrait créer ?
- Qu'elles seront leurs propriétés ? Leurs commandes ?
- Il n'y a pas de recette miracle pour les identifier.
- Généralement, les classes principales ressortent intuitivement (souvent reliées à des objets réels).

**Exemple :** 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.

**Exemple :** 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, *etc.*

8

## 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.

**Exemple :** 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.

9

## 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 ?

→ Ceci définit les attributs de la classe.

**D'action:** Quelles sont les tâches que l'objet doit accomplir ?

→ Ceci définit les méthodes de la classe. On les divise généralement en méthodes de requêtes et de commandes.

**De création:** L'objet doit-il être créé d'autres objets ?

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

10

## Exemple

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

### Pièce

*Connaissance :* type, couleur.

*Action :*

Requêtes : type et couleur, peut-elle faire un mouvement donné

Commandes : aucune.

*Création :* aucune.

11

## Exemple

### Joueur

*Connaissance* : couleur, échiquier.

*Action* :

Requêtes : couleur.

Commandes : jouer un coup.

*Création* : aucune.

### Échiquier

*Connaissance* : grille de l'échiquier, les pièces.

*Action* :

Requêtes : pièce à une position donnée

Commandes : bouger une pièce.

*Création* : grille, les pièces.

12

## 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.
- ⇒ ***D'où l'importance de l'étape de la conception-même avant de se lancer dans l'écriture du programme !!***
- ⇒ Évite de modifier constamment le programme et risquer d'y insérer des bogues souvent subtils.

14

## Exemple

### Jeu

*Connaissance* : échiquier, joueurs

*Action* :

Requêtes : aucune *a priori*.

Commandes : exécution du jeu.

*Création* : échiquier, les joueurs.

On pourrait ajouter un utilitaire : une classe pour une position, ce qui permet souvent de sauver le l'écrite en passant un seul paramètre au lieu de deux. (laissé en exercice).

13

## Relations

- Dans la conception d'un programme, il est aussi important d'établir les relations entre les objets. Pourquoi ?
  - Aide à clarifier le rôle de chaque objet ;
  - met en évidence les dépendances entre les objets ;
  - peut aider à mieux comprendre le flot des informations et du programme.
  - peut aider à déceler les dangers d'effets de bord ;
  - ...
- Les responsabilités de chaque objet déterminent plusieurs des relations entre les objets.

15

## Types de relations

Voici les plus fréquents :

*Utilisation* : Un objet fait usage de méthodes d'un autre objet. Appelée aussi relation *client-serveur*.

*Possession* : Un objet possède une référence sur un autre via une variable d'instance.

*Aggrégation* : Un objet fait partie intégrante d'un autre objet.

*Abstraction* : Relation entre classe. Une classe est une abstraction d'une autre si la première est une spécialisation de la seconde. Appelé aussi relation hiérarchique. *Pas sujet à ce cours.*

Lors de la conception, on représente habituellement ses relations schématiquement (il existe des formalismes pour ça, tel UML, pour *Unified Modelin Language*).

16

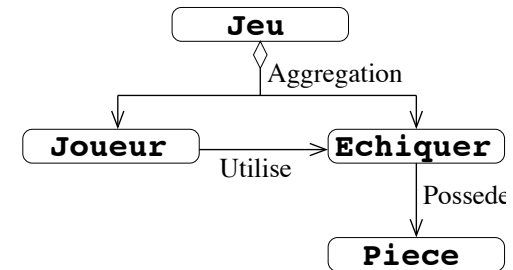
## Exemple

- Le jeu est composé de son échiquier et ses deux joueurs  
⇒ *aggrégation*.
- L'échiquier a un ensemble de pièces sur sa grille  
⇒ *possession*. À la limite, on pourrait considérer ça aussi de l'*aggrégation* 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  
⇒ *utilisation*. En fait, le joueur *possède* aussi un échiquier car il doit connaître l'échiquier sur lequel il joue.

18

## Exemple

Voici quelques relations pour notre jeu d'échec :



17

## Relations

- Le choix des relations peut être ambigu.  
⇒ Le choix de conception aussi.  
→ Grande partie du génie logiciel.
- Pour l'instant, l'important est de voir les liens essentiels.

19

## Remarques

– *possession*  $\neq$  *utilisation*

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

– *possession*  $\neq$  responsabilité de création

**Ex.** : Presque tout usage de `String`.

– *utilisation*  $\neq$  *possession*

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

Et évidemment :

– *aggrégation*  $\Rightarrow$  *possession*

– *aggrégation*  $\Rightarrow$  responsabilité de création (généralement)

20

## Parce que...

– Plus grande *réutilisabilité*

– Meilleure compréhension

– Localité des modifications au code

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

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

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

22

## 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 *relation*).

Pourquoi ?

21

## Exemple 1

La classe suivante pourrait être plus cohésive :

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

    // 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 int[ 8 ][ 8 ];

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

23

```

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.

24

## Exemple 1

Et Une classe pour la notion de 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; }

    ...
}

```

26

## Exemple 1

Une classe pour la notion d'échiquier :

```

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

    public Echiquier() { // initialise grille }

    public boolean bougerPiece( int i0, int j0, int i1, int j1 ) { ... }
    public Piece litPiece( int i, int j ) { return grille[ i ][ j ]; }

    ...
}

```

25

## Exemple 2

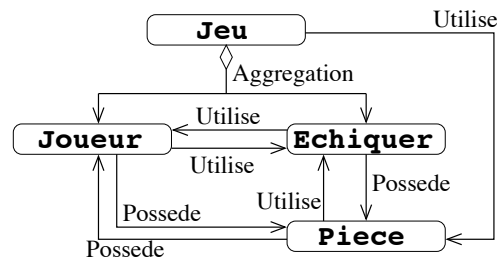
Dans le jeu d'échec, on pourrait ajouter les relations suivantes :

- au lieu d'avoir un attribut couleur pour une pièce, elle pourrait avoir une référence sur le joueur à qui elle appartient (*possession*);
- ⇒ comme l'échiquier instancie les pièces, il doit au moins connaître les joueurs à la création (*utilisation*);
- l'entière du déplacement des pièces pourrait être fait dans la classe `Piece`, mais les déplacements ont besoin d'information globale ⇒ la pièce doit consulter l'échiquier (*utilisation*);
- la validation du choix d'une pièce à déplacer pourrait être fait au moment où le joueur fait son coup ⇒ il doit connaître les pièces qui lui reste (*possession*);

27

- enfin, pour terminer la boucle du jeu, le roi d'un joueur doit-être mis en échec et mat ; si ce test est dans la classe du jeu, le jeu *utilise* les pièces.

Ce qui nous donne le schéma suivant :



Il y a beaucoup de couplage !

28

## 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 créer une instance de la classe et appelle ses méthodes.
- Cependant, un objet a généralement plus d'une méthode ⇒ le teste sera nécessairement plus complexe.
- Tout dépendant de la nature des relations de la classe, il faudrait peut-être faire usager 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.

30

## Exemple 2

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 ⇒ 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.
- ⇒ 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.

29

## Documenter une classe

- Devoir consulter le code d'une classe pour savoir comment l'utiliser n'est pas une façon efficace de travailler.
  - D'autant plus que le code n'est pas toujours disponible.
- ⇒ 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, *etc.*) en analysant le code.
- Ex. :** 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.

31



## Quoi écrire ?

- Le *rôle* d'une classe, son fonctionnement *général*.
  - La *tâche* accomplie par une méthode, mais *PAS* une description du code. On écrit aussi 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 cachés de la classe (le code, ce qui est privés), sauf peut-être pour documenter certaines méthodes *internes*.
- ⇒ Toujours rester d'un point de vue *conceptuel*.

32

## 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.

33

## Postconditions

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

### Exemples :

- Indiquer la plage de valeur pour un retour d'une méthode de requête.
- Indiquer les répercussions 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 connaît pas tous les impacts de l'appel d'une méthode.

34

## Exemple (esquisse du code des classes du jeu d'échec)

[Piece.java]

```
/**
 * 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
{
    /** Couleur blanche. */
    public static final boolean BLANC = true;

    /** Couleur noire. */
    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.
```

35

```

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

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

/**
 * Construit une pièce d'une couleur et de type donnés.
 * @param couleur Couleur de la pièce: Piece.BLANC ou Piece.NOIR.
 * @param type La valeur du type 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
}

/** 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; }

```

36

## [Joueur.java]

```

/**
 * 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;

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

    /**
     * Construit un joueur avec sa couleur et l'échiquier sur lequel il joue.
     * @param couleur Couleur du des pièces du joueur: Piece.BLANC ou Piece.NOIR.
     * @param e L'échiquier sur lequel 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
    }
}

```

38

```

/**
 * 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.
}

...
}

```

37

```

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

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

...
}

```

39

## [Echiquier.java]

```
/**
 * Représente un échiquier avec ses pièces. La rangé 0 est du coté 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 à leur position initiale.
     */
    public Echiquier()
    {
        grille = new Piece[ 8 ][ 8 ];
        grille[ 0 ][ 0 ] = new Piece( Piece.BLANC, Piece.TOUR );
        ... // etc.
    }
}
```

40

## [Jeu.java]

```
/**
 * 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;

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

        // Remarque: on cree une relation supplémentaire entre Jeu et Piece en
        // utilisant ces constantes, mais assure l'unicité de la représentation.
        joueurBlanc = new Joueur( Piece.BLANC, echiquier);
    }
}
```

42

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

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

...
}
```

41

```
joueurNoir = new Joueur( Piece.NOIR, echiquier);
}

/** Exécute une partie. */
public void exec()
{
    ... // Le jeu: chaque joueur joue à son tour tant que l'un n'a pas perdu
}

...
}
```

43