IFT3912 - Développement et maintenance de logiciels

Organisation du code

Bruno Dufour
dufour@iro.umontreal.ca

Lisibilité

- Du bon code doit être suffisamment facile à lire et à comprendre pour
  - déterminer ses interactions avec le reste du système
  - y apporter des changements
  - identifier des bogues
- Du bon code vise à minimiser le temps nécessaire à la compréhension par un autre développeur
  - Souvent, un autre développeur = vous dans quelques mois
  - facilite la réutilisation

Taille et lisibilité


```
for(Node* node = list->head; node != NULL; node = node->next)
    print(node->data);
```

OU

```
Node* node = list->head;
if (node == NULL) return;
while (node->next != NULL) {
    print(node->data);
    node = node->next;
}
if (node != NULL) print(node->data);
```


Taille et lisibilité


```
return exponent >= 0 ? mantissa * (1 << exponent)
                     : mantissa / (1 << -exponent);
```

OU

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

Taille et lisibilité


```
assert ((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

OU

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```


Taille et lisibilité


```
hash = (hash << 6) + (hash << 16) - hash + c;
```

OU

```
// Fast version of "hash = (65599 * hash) + c"
hash = (hash << 6) + (hash << 16) - hash + c;
```


Nommer les éléments

L'importance des noms

- Les noms sont omniprésents en programmation
  - Variables, arguments, fonctions, classes, paquetages, fichiers, dossiers, etc.
- Les noms véhiculent de l'information
  - Utiliser des nom approprié améliore la compréhension du code et peut aider à éviter les bogues
  - Utiliser un nom mal choisi peut introduire de la confusion et causer des erreurs
- Un nom doit être choisi en considérant si un collègue comprendrait sa signification

L'importance des noms


```
int d; // aucune information utile
int days;
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```


L'importance des noms


```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```


L'importance des noms


```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```


L'importance des noms


```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Recommendations de nommage

- Choisir des termes spécifiques
- Éviter les noms génériques
- Attacher de l'information importante aux noms
- Choisir la longueur d'un nom en fonction de sa portée
- Utiliser la mise en forme pour encoder de l'information
- Considérer les attentes des autres développeurs

Choisir des termes spécifiques

- Éviter les termes trop génériques et imprécis
  - ex: get, make, tmp, retval, ...
- Un mot plus précis donne plus d'information
  - make → create, set up, build, generate, compose, add, new, ...

Choisir des termes spécifiques


```
def GetPage(url):
    ...
```

OU

```
def FetchPage(url):
    ...

def DownloadPage(url):
    ...
```


Choisir des termes spécifiques


```
class Graph {
    public int size() {...}
}
```

OU

```
class Graph {
    public int numberOfNodes() {...}
    public int numberOfEdges() {...}
}
```

Éviter l'ambiguité


```
results = Database.all_objects.filter("year <= 2011")
```

- Que contient "results"?
  - Les objets dont l'année est ≤ 2011?
  - Les objets dont l'année est > 2011?
- Un meilleur choix de nom aurait été
  - `select` si ≤ 2011
  - `exclude` si > 2011

Éviter les noms génériques


```
def swap(data, i, j):
    tmp = data[i]
    data[i] = data[j]
    data[j] = tmp
```

✔ utilisation justifiée de "tmp", puisque la nature temporaire de la variable est le fait le plus important à évoquer

Éviter les noms génériques


```
String tmp = user.name();
tmp += " " + user.phone_number();
tmp += " " + user.email();
...
template.set("user_info", tmp);
```

✘ utilisation injustifiée de "tmp"

Éviter les noms génériques


```
var euclidean_norm = function (v) {
    var retval = 0.0;
    for (var i = 0; i < v.length; i += 1)
        retval += v[i] * v[i];
    return Math.sqrt(retval);
};
```

Éviter les noms génériques


```
var euclidean_norm = function (v) {
    var sum_squares = 0.0;
    for (var i = 0; i < v.length; i += 1)
        sum_squares += v[i] * v[i];
    return Math.sqrt(sum_squares);
};
```


Éviter les noms génériques


```
public static void copyChars(char a1[], char a2[]) {
    for (int i = 0; i < a1.length; i++) {
        a2[i] = a1[i];
    }
}
```


Éviter les noms génériques


```
public static void copyChars(char source[], char dest[]) {
    for (int i = 0; i < source.length; i++) {
        dest[i] = source[i];
    }
}
```


Éviter les noms génériques


```
for (int i = 0; i < clubs.size(); i++)
  for (int j = 0; j < clubs[i].members.size(); j++)
    for (int k = 0; k < users.size(); k++)
      if (clubs[i].members[k] == users[j])
        print("user[" + j + "] is in club[" + i + "]\n");
```

Éviter les noms génériques


```
for (int c = 0; c < clubs.size(); c++)
  for (int m = 0; m < clubs[c].members.size(); m++)
    for (int u = 0; u < users.size(); u++)
      if (clubs[c].members[m] == users[u])
        print("user[" + u + "] is in club[" + c + "]\n");
```


Attacher des propriétés aux noms


```
def start(delay):
    ...
```

OU

```
def start(delayInMillis):
    ...
```


Attacher des propriétés aux noms


```
password = ...
comment = ...
data = ...
```

OU

```
plaintextPassword = ...
unescapedComment = ...
urlEncodedData = ...
```


Choisir la longueur des noms


```
if (debug) {
    Map<String,Integer> m = lookUpNamesNumbers();
    display(m);
}
```

✔ Portée locale: le nom “m” ne cause pas de problème puisque toute l’information nécessaire à la compréhension est disponible

Choisir la longueur des noms


```
if (debug) {
    this.m = lookUpNamesNumbers();
    display(m);
}
```

✘ Portée étendue: le nom “m” est insuffisant pour communiquer l’information nécessaire à la compréhension

Utiliser la mise en forme

- Une convention qui inclut la mise en forme peut transmettre de l’information importante
- Par exemple
  - Majuscule initiale pour les classes: `StringBuffer`
  - Minuscule initiale les méthodes: `toString`
  - Majuscules pour les constantes: `MAX_CHARS`

Considérer les attentes

- Par exemple
  - `get*` devrait être réservé pour des méthodes qui n’effectuent pas de calculs
  - `getMean()` devrait être nommée `computeMean()` si la méthode effectue un calcul sur une quantité importante de données
  - De même, un développeur s’attend à ce que la méthode `size()` d’une collection s’exécute rapidement

Commentaires

L'importance des commentaires

- Les commentaires visent à transmettre au lecteur du code les connaissances de son auteur
- Attention! Les commentaires doivent être modifiés en même temps que le code
  - Il faut donc choisir judicieusement quoi commenter
- Les commentaires ne compensent pas du mauvais code
  - bon code > mauvais code + commentaires

Quoi commenter ?

- En général, n'inclure que des faits qui ne peuvent pas être inférés rapidement à partir du code lui-même

```
// Class definition for Account
class Account {
    /** Default constructor */
    public Account() {
        this.balance = 0; // Initialize balance to 0
    }
}
```

- "Rapidement" peut être relatif

```
# remove everything after the second '*'
name = '*'.join(line.split('*')[:2])
```


Quoi commenter ?

- Ajout d'information

```
// Surprisingly, a binary tree was 40% faster than a
// hash table for this data.
```

- Explication de l'intention du programmeur

```
// This is our best attempt to get a race condition
// by creating large number of threads.
for (int i = 0; i < 25000; i++) { ... }
```

- Vue d'ensemble

```
/* This is the glue code between our business logic and
the database. None of the application code should use
this directly. */
```


Quoi commenter ?

- Mise en garde

```
// Calls an external service to deliver email. (Times
// out after 1 minute.)
void SendEmail(string to, string subject, string body);
```

- Clarification

```
// Force vector to relinquish its memory (look up "STL
// swap trick")
vector<float>().swap(data);
```

Quoi commenter ?

- Résumé

```
def GenerateUserReport():
        # Acquire a lock for this user
        ...

        # Read user's info from the database
        ...

        # Write info to a file
        ...

        # Release the lock for this user
        ...
```


Quoi commenter ?

- Problèmes & tâches

```
// TODO(alice): handle other image formats besides JPEG
```

Marqueur	Signification usuelle
TODO	Tâches restantes
FIXME	Code erroné connu
HACK	Solution inélégante à un problème
XXX	Problème majeur

Commentaires et noms

- Code commenté

```
// Check to see if the employee is eligible
// for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65)) ...
```

- Code réécrit élimine le besoin d'un commentaire

```
if (employee.isEligibleForFullBenefits())
```


Mauvais commentaires

- Logs
  - Les systèmes de révision de contrôle sont mieux adaptés à cette tâche

```
*  Changes (from 11-Oct-2001)
*  -------------------------
*  11-Oct-2001 : * Re-organised the class and moved it to
*                  new package com.jrefinery.date (DG);
*  12-Nov-2001 : * IBD requires setDescription() method,
*                  now that NotableDate class is gone (DG);
*                * Changed getPreviousDayOfWeek(),
*                  getFollowingDayOfWeek() and
*                  getNearestDayOfWeek() to correct bugs
*                  (DG);
```

Mauvais commentaires

- Commentaires de fin de blocs

```
while ((line = in.readLine()) != null) {
    lineCount++;
    charCount += line.length();
    String words[] = line.split("\\W");
    wordCount += words.length;
} //while
```

  - Si ces commentaires deviennent nécessaires, il est généralement préférable de raccourcir la fonction.

Esthétique

Formatter le code

- Formatter le code est important pour sa lisibilité
- L'utilisation judicieuse des espaces peut encoder de l'information utile
  - Formattage vertical : espaces entre les lignes
  - Formattage horizontal : espaces entre les termes d'une même ligne

Formattage vertical

- Les éléments qui apparaissent près l'un de l'autre dans un fichier sont habituellement perçus comme appartenant à un même concept
  - Les espaces entre les lignes permettent de délimiter les concepts
- Les éléments reliés devraient être placés près l'un de l'autre dans le fichier
  - ex: placer une méthode appelée tout de suite après sa méthode appelante
- Les éléments les plus importants devraient être placés avant ceux de moindre importance
  - analogie: article de journal

Formattage horizontal

- Les espaces peuvent améliorer la lisibilité d'une ligne de code
  - Rendre la priorité des opérations plus explicite:

```
return b*b - 4*a*c;
```

  - Rendre plus visible des éléments difficiles à voir :

```
while (read(buf, 0, readBufferSize) != -1)
    ;
```


Variables

Importance des variables

- Les variables peuvent contribuer à documenter le code
- Variable d'explication / de résumé
  - Nomme (explique) une sous-expression

```
username = line.split(':')[0].strip()
if username == "root":
```

  - Nomme un concept utilisé

```
userOwnsDocument = (request.user.id ==
                    document.owner_id)
if userOwnsDocument:
```


Variables temporaires inutiles

- Les variables temporaires peuvent aider à la compréhension, mais aussi nuire lorsqu'elles sont utilisées incorrectement
- Par exemple:

```
now = datetime.datetime.now()
root_message.last_view_time = now
```

  - L'utilisation de la variable now ne réduit pas la taille du code et n'ajoute rien à la compréhension

Variables intermédiaires inutiles


```
var remove_one = function (array, value_to_remove) {
    var index_to_remove = null;
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            index_to_remove = i;
            break;
        }
    }
    if (index_to_remove !== null) {
        array.splice(index_to_remove, 1);
    }
};
```


Variables intermédiaires inutiles


```
var remove_one = function (array, value_to_remove) {
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            array.splice(i, 1);
            return;
        }
    }
};
```


Portée des variables

- La portée des variables devrait être réduite le plus possible
  - Préférer des variables locales à des variables globales
  - Préférer les déclarations à l'intérieur du bloc le près possible de l'utilisation de variables locales
  - Préférer les définitions le plus près possible de l'utilisation plutôt qu'au début d'un bloc

Flot de contrôle

Ordre des comparaisons

- Gauche: expression “interrogée”
- Droite: expression de référence
- Par exemple :
  - `if (length >= 10)` est plus lisible que
    `if (10 < length)`
  - `while (bytes_received < bytes_expected)`
    est plus lisible que
    `while (bytes_expected >= bytes_received)`

Ordre des blocs blocs if/else

- En général
  - préférer le cas attendu / positif en premier
  - préférer le cas le plus simple en premier (peut permettre de voir les deux cas simultanément à l’écran)

Retour précoce

- Il est souvent utile de retourner rapidement d’une fonction pour simplifier le flot de contrôle
- Par exemple

```
public boolean Contains(String str, String substr) {
    if (str == null || substr == null) return false;
    if (substr.equals("")) return true;
    ...
}
```


Imbrication

- L’imbrication de blocs rend la compréhension du code plus difficile
  - Le lecteur doit mentalement maintenir une pile de conditions pour suivre la logique du code
    - { → empiler la condition
    - } → dépiler une condition
  - Il peut être difficile de se rappeler l’état de la pile au moment de dépiler

Limiter l'imbrication


```
if (user_result == SUCCESS) {
    if (permission_result != SUCCESS) {
        reply.WriteErrors("error reading permissions");
        reply.Done();
        return;
    }
    reply.WriteErrors("");
} else {
    reply.WriteErrors(user_result);
}
reply.Done();
```


Limiter l'imbrication


```
if (user_result != SUCCESS) {
    reply.WriteErrors(user_result);
    reply.Done();
    return;
}

if (permission_result != SUCCESS) {
    reply.WriteErrors(permission_result);
    reply.Done();
    return;
}

reply.WriteErrors("");
reply.Done();
```


Limiter l'imbrication dans les boucles


```
for (int i = 0; i < results.size(); i++) {
    if (results[i] != NULL) {
        non_null_count++;
        if (results[i]->name != "") {
            ...
        }
    }
}
```


Limiter l'imbrication dans les boucles


```
for (int i = 0; i < results.size(); i++) {
    if (results[i] == NULL) continue;
    non_null_count++;
    if (results[i]->name == "") continue;
    ...
}
```

Gestion des exceptions

Exceptions vs codes de retour

- L'utilisation des exceptions permet de simplifier le code appelant
  - Les tests de valeur ajoutent du code qui n'est pas directement relié à la tâche qui doit être accomplie
  - Moins de tests = moins d'imbrication, flot de contrôle plus simple
  - Il est facile d'oublier de vérifier certaines valeurs retournées

Exceptions vs codes de retour


```
public class DeviceController {
    ...
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        if (handle != DeviceHandle.INVALID) {
            // Save the device status to the record field
            retrieveDeviceRecord(handle);
            // If not suspended, shut down
            if (record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                ...
```


Exceptions vs codes de retour


```
public class DeviceController {
    ...
    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e);
        }
    }

    private void tryToShutDown() throws DeviceShutDownError{
        DeviceHandle handle = getHandle(DEV1);
        DeviceRecord record = retrieveDeviceRecord(handle);
        pauseDevice(handle);
        ...
}
```

Retourner `null`

- Retourner la valeur `null` force l'appelant à vérifier la valeur retournée
  - L'appelant peut oublier de vérifier la valeur, et donc introduire un bogue
- Alternatives :
  - Objets spéciaux : par exemple, retourner Collections.emptyList() pour indiquer l'absence d'éléments
  - Exceptions spécialisées : indiquent qu'un événement anormal s'est produit
    - Ne pas utiliser NullPointerException puisque l'appelant ne saura pas comment l'interpréter
- Pour des raisons similaires, il est rarement conseillé de passer `null` en paramètre à une fonction

Méthodes et classes

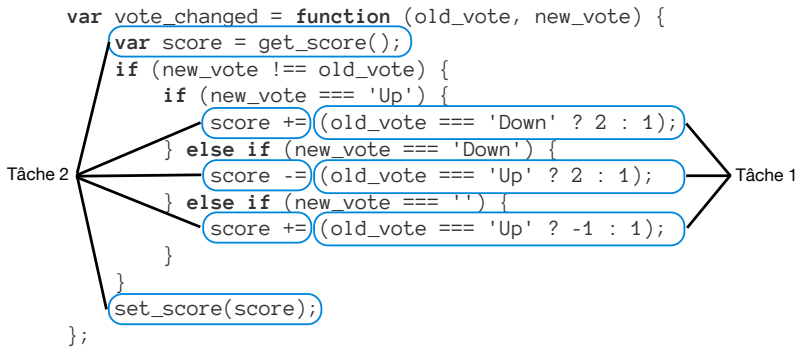
Taille

- Les fonctions devraient être le plus courtes possibles
  - En général, quelques lignes (< 20)
- Une fonction courte est
  - plus facile à lire / comprendre
  - moins propice à d'autres sources de complexité (ex: imbrication)
- Les compilateurs et machines virtuelles modernes optimisent les appels à des fonctions courtes
  - Très peu ou pas d'impact sur le temps d'exécution

Une tâche à la fois

- Une fonction devrait être écrite de façon à n'effectuer qu'une seule tâche
  - "*Do one thing and do it well*"
- En général, une fonction qui n'effectue qu'une tâche ne devrait opérer qu'à un seul niveau d'abstraction
  - ex: File vs String (chemin) vs StringBuffer (contenu)
- Comment reconnaître une fonction minimale ?
  - La fonction n'effectue que des tâches qui sont directement sous le niveau associé à son nom
  - On ne peut extraire de code qui causera un changement du niveau d'abstraction

Une tâche à la fois



Une tâche à la fois


```
var vote_value = function (vote) {
    if (vote === 'Up') {
        return +1;
    } else if (vote === 'Down') {
        return -1;
    }
    return 0;
};

var vote_changed = function (old_vote, new_vote) {
    var score = get_score();
    score -= vote_value(old_vote);
    score += vote_value(new_vote);
    set_score(score);
};
```


Arguments

- Une fonction devrait nécessiter le moins d'arguments possible
  - Idéalement, 0 ou 1, mais rarement plus de 2
- La majorité des programmeurs s'attendent à ce que les arguments constitue les entrées d'une fonction et non la sortie
  - Argument d'entrée

    `renderedPage = renderer.render(data)`
  - Argument de sortie

    `data.renderInto(page)`

Éviter les effets de bord

- Les effets de bord qui ne sont pas annoncés par le nom d'une fonction peuvent causer des surprises
- En général, il est conseillé d'éviter le plus possible les effets de bord
  - Souvent requis pour respecter le principe d'une tâche par fonction

Effets de bord


```
public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
        String codedPhrase = user.getPhraseEncodedByPassword();
        String phrase = cryptographer.decrypt(codedPhrase,
                                              password);
        if ("Valid Password".equals(phrase)) {
            Session.initialize();
            return true;
        }
    }
    return false;
}
```

Peut effacer les données de la session!

Principe de responsibilité unique

- Principe de responsabilité unique : une classe devrait avoir une seule raison de changer (une seule responsabilité)
  - Réduit la taille des classes
  - Facilite la lecture du code et la compréhension
  - Facilite l'écriture de tests
  - Facilite l'évolution et la maintenance (meilleure structure)

Références

- Robert C. Martin, "Clean Code: A Handbook of Agile Software Craftsmanship", Prentice-Hall 2008.
- Dustin Boswell & Trevor Foucher, "The Art of Readable Code", O'Reilly 2011.