Clean Code
Clément D.
Redouane B.
Sébastien L.
ubik
INGÉNIERIE

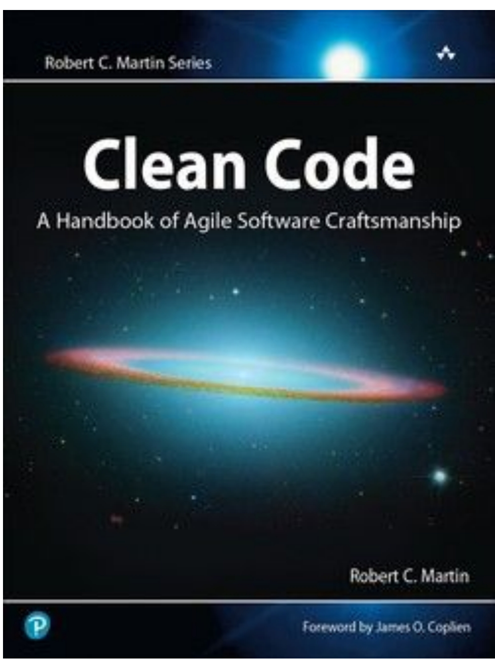
Qui sommes-nous ?

Clément D.
Développeur Java & Spring Boot (+ de la CI et du front)
Chez Ubik Ingénierie depuis 3 ans

Redouane B.
Développeur full-stack depuis un peu plus d'un an
Back-end Java et Node.js / front-end Angular, JSP et EJS
Chez Ubik Ingénierie depuis moins d'un an

Sébastien L.
Développeur full-stack (Java / TypeScript) depuis 13 ans
Principalement sur de l'e-commerce
Chez Ubik Ingénierie depuis 9 ans

Clean Code: A Handbook of Agile Software Craftsmanship



Auteur : Robert C. Martin
Date de publication : Août 2008

Définitions multiples :
Certains choix sont subjectifs.

Plan

1. Objectifs
2. Exemple en fil rouge
3. Principes & méthode
4. Ⓐ Nommage
5. Ⓑ Paradigmes & classes
6. Ⓒ Fonctions
7. Refactoring

Des questions ?
Prenez-note ✏ :

Exemple : avant

```java
@Value
public class HistoryData {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String lbl;
    LocalDate beginDt;
    LocalDate endDt;
    boolean deleted;
    boolean active;

    public HistoryData(Map<String, Object> rs) {
        lbl = (String) rs.get("LABEL");

        try {
            beginDt = rs.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) rs.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + rs.get(START_DATE), e);
        }

        try {
            endDt = rs.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) rs.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + rs.get(STOP_DATE), e);
        }

        deleted = rs.get("DELETED") == Boolean.TRUE;

        LocalDate date = LocalDate.now();
        active = !deleted &&
                (beginDt == null || (beginDt.isBefore(date) || beginDt.equals(date))) &&
                (endDt == null || (endDt.isAfter(date) || endDt.equals(date)));
    }

}
```

Exemple : après

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

    public DateInterval getDateInterval(String startDateColumnName, String stopDateColumnName) {
        LocalDate startDate = getDateOrNull(startDateColumnName);
        LocalDate stopDate = getDateOrNull(stopDateColumnName);
        return new DateInterval(startDate, stopDate);
    }

    public boolean getBooleanOrFalse(String columnName) {
        return resultSet.get(columnName) == Boolean.TRUE;
    }

    private LocalDate getDateOrNull(String columnName) {
        String value = (String) resultSet.get(columnName);
        return value == null ? null : parseDate(columnName, value);
    }

    private LocalDate parseDate(String columnName, String value) {
        try {
            return tryParseDate(value);
        } catch (DateTimeParseException e) {
            throw new FormatException(columnName + " has wrong format: " + value, e);
        }
    }

    private LocalDate tryParseDate(String value) {
        return LocalDate.parse(value, DateTimeFormatter.ISO_LOCAL_DATE);
    }
}
```

Objectifs

Objectifs

- Compréhensible par tous
- Facilement maintenable
- Réduit la complexité
- Indépendant du développeur initial

Exemple

Spécifications fonctionnelles

Mapper une ligne de base de données vers un objet Java

Une ligne a :

- un libellé
- un intervalle de validité dans le temps

Une ligne peut :

- être supprimée (historisée) : grisée
- être active : en surbrillance

Ligne	Début	Fin
Valeur 1	-	31/05/2021
Valeur 2	01/06/2021	02/06/2021
Valeur 3	01/06/2021	02/06/2021
Valeur 4	03/06/2021	-

```java
@Value
public class HistoryData {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String lbl;
    LocalDate beginDt;
    LocalDate endDt;
    boolean deleted;
    boolean active;

    public HistoryData(Map<String, Object> rs) {
        lbl = (String) rs.get("LABEL");
```

```java
try {
    beginDt = rs.get(START_DATE) == null
            ? null
            : LocalDate.parse((String) rs.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
} catch (DateTimeParseException e) {
    throw new FormatException(START_DATE + " has wrong format: " + rs.get(START_DATE), e);
}

try {
    endDt = rs.get(STOP_DATE) == null
            ? null
            : LocalDate.parse((String) rs.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
} catch (DateTimeParseException e) {
    throw new FormatException(STOP_DATE + " has wrong format: " + rs.get(STOP_DATE), e);
}
```

```java
        deleted = rs.get("DELETED") == Boolean.TRUE;

        LocalDate date = LocalDate.now();
        active = !deleted &&
                (beginDt == null || (beginDt.isBefore(date) || beginDt.equals(date))) &&
                (endDt == null || (endDt.isAfter(date) || endDt.equals(date)));
    }

}
```

Principes
&
méthode

Principes

- KISS : Keep It Simple Stupid
- DRY : Don’t Repeat Yourself
- YAGNI : You Aren’t Gonna Need It
- La règle du boy scout

Nous sommes des traducteurs Français ↔ Informatique

Réduction de 50 % si le produit périme dans les 4 jours

↕

```
discount = daysUntilExpiration <= 4 ? percent(50) : NO_DISCOUNT;
```

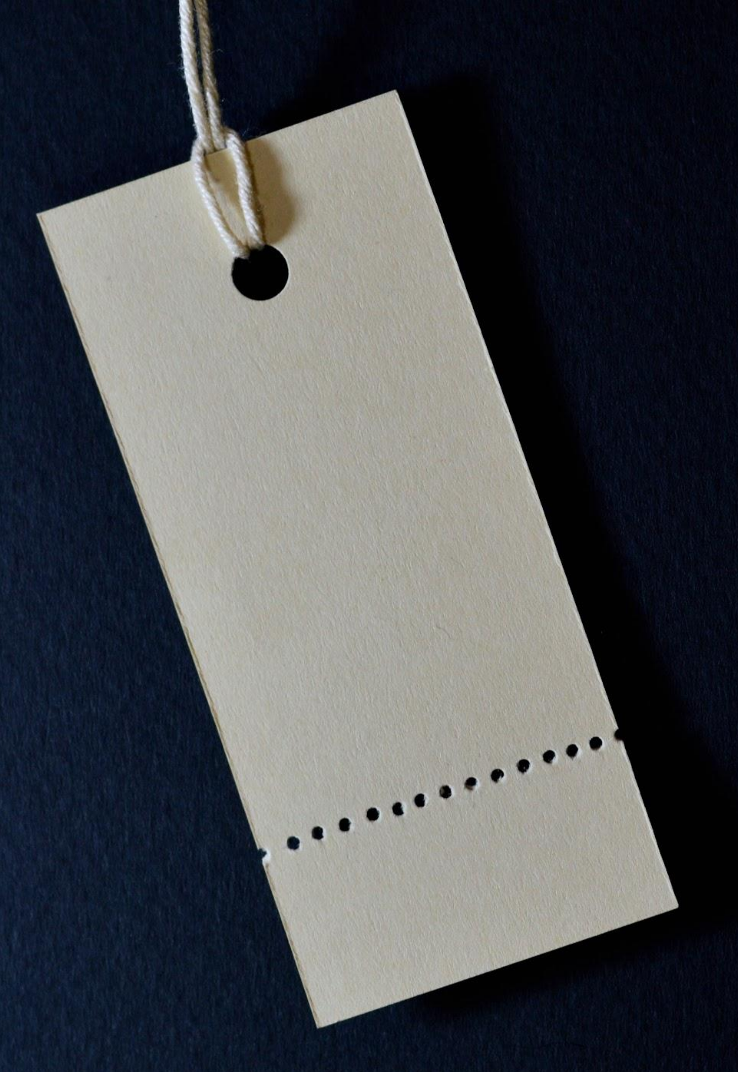
Méthode

Les règles métier doivent être limpide dans le code

1. Via le nommage

2. Puis via le découpage en petites fonctions :
   Chaque fonction fait une seule chose et bien
   Chaque fonction a un seul niveau d'abstraction

3. Puis via l'organisation des fonctions en classes
   Chaque classe a sa propre responsabilité
   Chaque classe encapsule ses détails techniques



Nommage

Pas d’abréviations ni de noms imprononçables

⛔ `config + conf + cnf + cfg` 🔎
✅ `configuration`

⛔ `pdt` ❓❗
✅ `pediatry ? pedometer ?`
`periodicity ? ponderation ?`
`postdate ?`
`product ? production ?`
`pendant ? pommeDeTerre ?`

⛔ `generationYmdhms` 📣
✅ `generationTimestamp`
🥇 `generationDateTime`

Pas d'abréviations : dans l'exemple

```
String lbl;                                      »17    17«  String label;
LocalDate beginDt;                                18    18   LocalDate beginDate;
LocalDate endDt;                                  19    19   LocalDate endDate;
boolean deleted;                                  20    20   boolean deleted;
boolean active;                                   21    21   boolean active;
                                                  22    22
public HistoryData(Map<String, Object> rs) {   @ »23    23« @ public HistoryData(Map<String, Object> resultSet) {
```

Sans parasites

```
⛔ Variable, Value, Data, Info, Details, Manager, Processor

⛔ Order orderObject;       ✅ order
⛔ class PriceInfo {}       ✅ Price
⛔ class PriceDetails {}    ✅ PriceTimeline / PricingPlan
⛔ String eanString;        ✅ String ean 🥇 Ean13 ean // Not EuropeanArticleNumbering13
⛔ String strName;          ✅ name
⛔ List<Long> idList;       ✅ ids

⛔ void fStart(int pSpeed) { _player.fRun(pSpeed); }
✅ void start(speed) { player.run(speed); }

⛔ interface ICar {}          ✅ interface Car {}
   ICar iCar = new Car();        Car car = new CitroenCar() // PeugeotCar... or CarImpl
                                 List<String> names = new LinkedList<String>()
```

Sans parasites : dans l'exemple

```
@Value                                 11    11     @Value
public class HistoryData {          »  12    12  «  public class HistoryRow {
```

Révéler l'intention

À quoi sert la variable / méthodes / classe ?

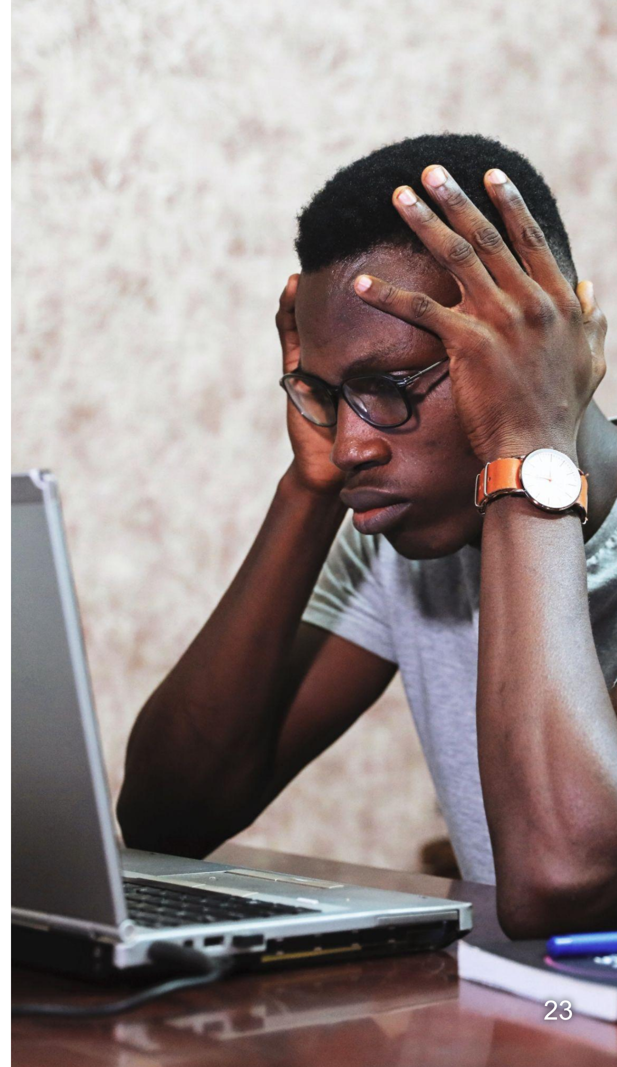
⛔ `List<Integer> list = …`
✅ `List<Integer> flags = …`
🥇 `EnumSet<Flag> flags = …`

Comment peut-on l’utiliser ?

⛔ `overridePrice(Product productA, Product productB)`
✅ `overridePrice(Product source, Product destination)`

Expliquer “pourquoi” au lieu de “comment”

⛔ `String getBarcodeReplacingNumbersUsingSmurfAlgorythm()`
✅ `String getBarcodeTextForFont()`



Révéler l'intention : dans l’exemple

```java
LocalDate date = LocalDate.now();
active = !deleted &&
        (beginDate == null || (beginDate.isBefore(date) || beginDate.equals(date))) &&
        (endDate == null || (endDate.isAfter(date) || endDate.equals(date)));
```

44 45 46 47

```java
LocalDate today = LocalDate.now();
active = !deleted &&
        (beginDate == null || (beginDate.isBefore(today) || beginDate.equals(today))) &&
        (endDate == null || (endDate.isAfter(today) || endDate.equals(today)));
```

Expliquer les concepts : cohérent & cherchable

Mots **communs** pour ce qui se **ressemble**

✅ `Product (pas Article)`
✅ `Cart    (pas Shopping Cart, ni Shopping Basket)`
✅ `Voucher (pas Coupon, ni Token)`

Mots **différents** pour ce qui **ne doit pas être confondu**

✅ `Site = Magasin (“site d’implantation” dans tout le SI)`
✅ `Store = Le store Vue.js (ne pas utiliser pour décrire un magasin)`

Glossaire commun au projet **et au métier**

Cohérent & cherchable : dans l’exemple

```
try {                                                          26  26      try {
    beginDate = resultSet.get(START_DATE) == null           » 27  27 «        startDate = resultSet.get(START_DATE) == null
            ? null                                             28  28                ? null
            : LocalDate.parse((String) resultSet.get(START_DATE),  29  29         : LocalDate.parse((String) resultSet.get(START_DATE),
} catch (DateTimeParseException e) {                          30  30      } catch (DateTimeParseException e) {
    throw new FormatException(START_DATE + " has wrong format: "  31  31      throw new FormatException(START_DATE + " has wrong format: "
}                                                              32  32      }
```

Déplacer les commentaires dans les noms et / ou le typage

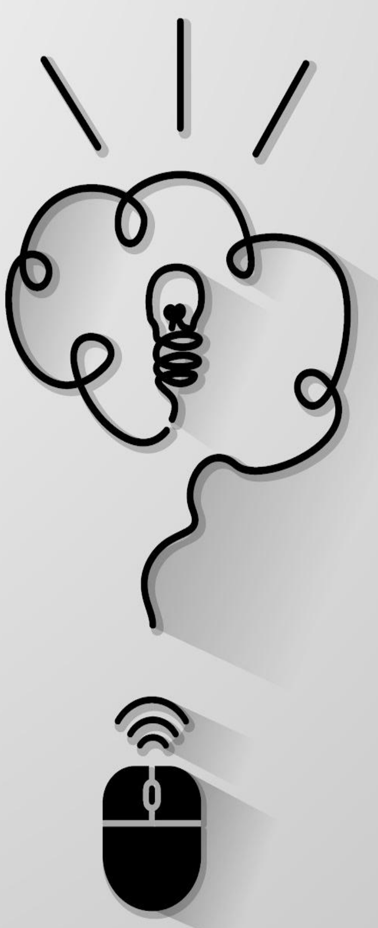
⛔
```
int DD = 3; // Default duration (in minutes)
```

⚠️
```
int DURATION = 3; // Default (in minutes)
```

🚸
```
int DEFAULT_DURATION = 3; // In minutes
```

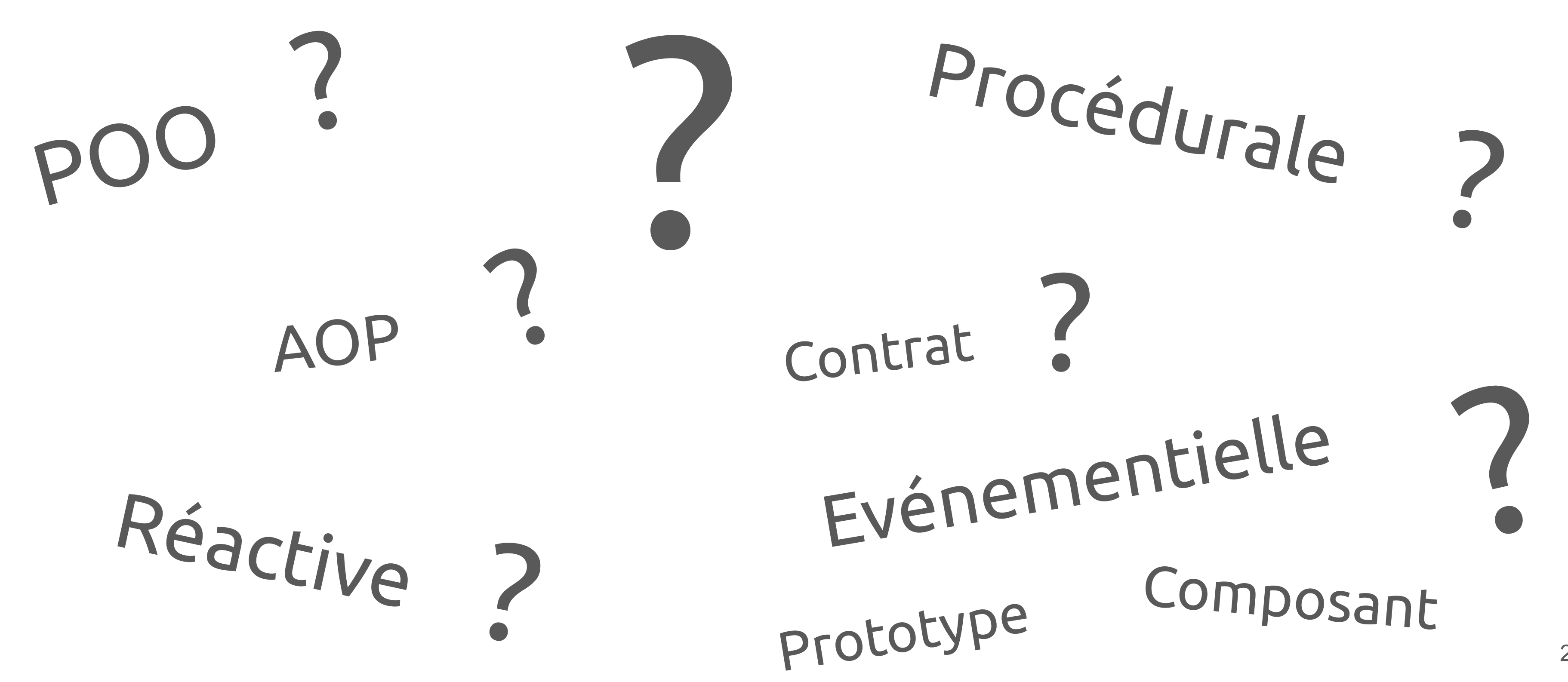
✅
```
int DEFAULT_DURATION_IN_MINUTES = 3;
```

🥇
```
Duration DEFAULT = Duration.ofMinutes(3);
```



Paradigmes

Les paradigmes de programmation



Au fait…

Java == JavaScript ?

Au fait…

Structure de données == Objet ?

Les structures hybrides ?

Représentent une conception confuse

Assemblent le pire des deux mondes

Types ou Fonctions ?

Création d’une structure de données

Fonctions

Création d’objets

Ce qu'il faut retenir ?

Structure de donnée

Facilite l'ajout de nouvelles fonctions
sans modifier les structures de données existantes

Objet

Facilite l'ajout de nouveaux types
(Classes, Interfaces) sans modifier les fonctions existantes

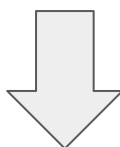
Classes

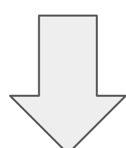
```
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

    public DateInterval getDateInterval(String startDateColumnName, String stopDateColumnName) {
        LocalDate startDate = getDateOrNull(startDateColumnName);
        LocalDate stopDate = getDateOrNull(stopDateColumnName);
        return new DateInterval(startDate, stopDate);
    }

    public boolean getBooleanOrFalse(String columnName) {
        return resultSet.get(columnName) == Boolean.TRUE;
    }

    private LocalDate getDateOrNull(String columnName) {
        String value = (String) resultSet.get(columnName);
        return value == null ? null : parseDate(columnName, value);
    }

    private LocalDate parseDate(String columnName, String value) {
        try {
            return tryParseDate(value);
        } catch (DateTimeParseException e) {
            throw new FormatException(columnName + " has wrong format: " + value, e);
        }
    }

    private LocalDate tryParseDate(String value) {
        return LocalDate.parse(value, DateTimeFormatter.ISO_LOCAL_DATE);
    }

}
```

Principe de responsabilité unique

Une classe ne doit changer que pour une seule raison





Faire de petites classes

Peu de variables

Maintenir la cohésion

Fonctions

Heuristique des fonctions

Une fonction ne doit faire qu’une seule chose, n’avoir qu’un seul but

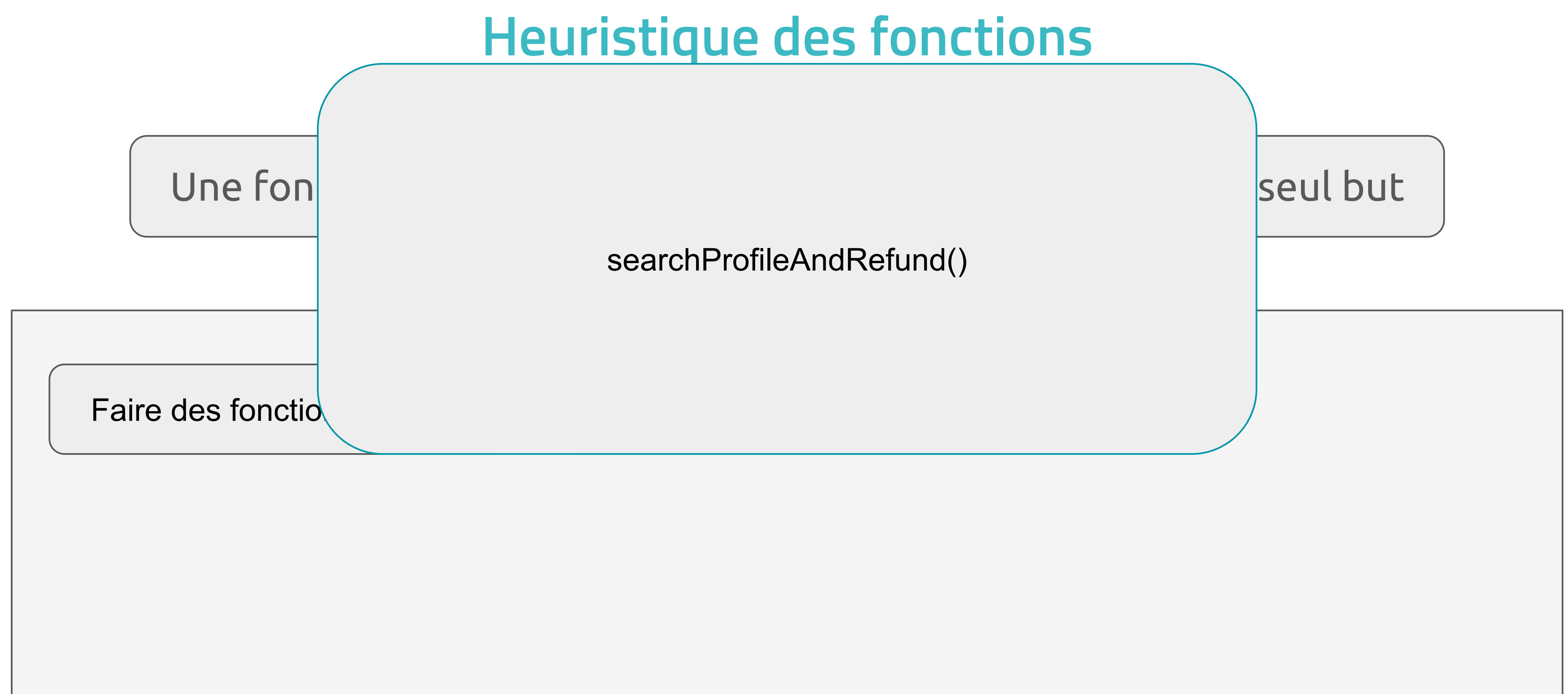
Faire des fonctions courtes

Heuristique des fonctions

Une fonction ne doit faire qu’une seule chose, n’avoir qu’un seul but

- Faire des fonctions courtes
- Nommage de fonction avec un seul verbe

Heuristique des fonctions



Heuristique des fonctions

Une fon seul but

Faire des fonctio

```
private Profil searchProfil(int id){
        ...
}
private void refund(Profil profil){
        ...
}
```

Heuristique des fonctions

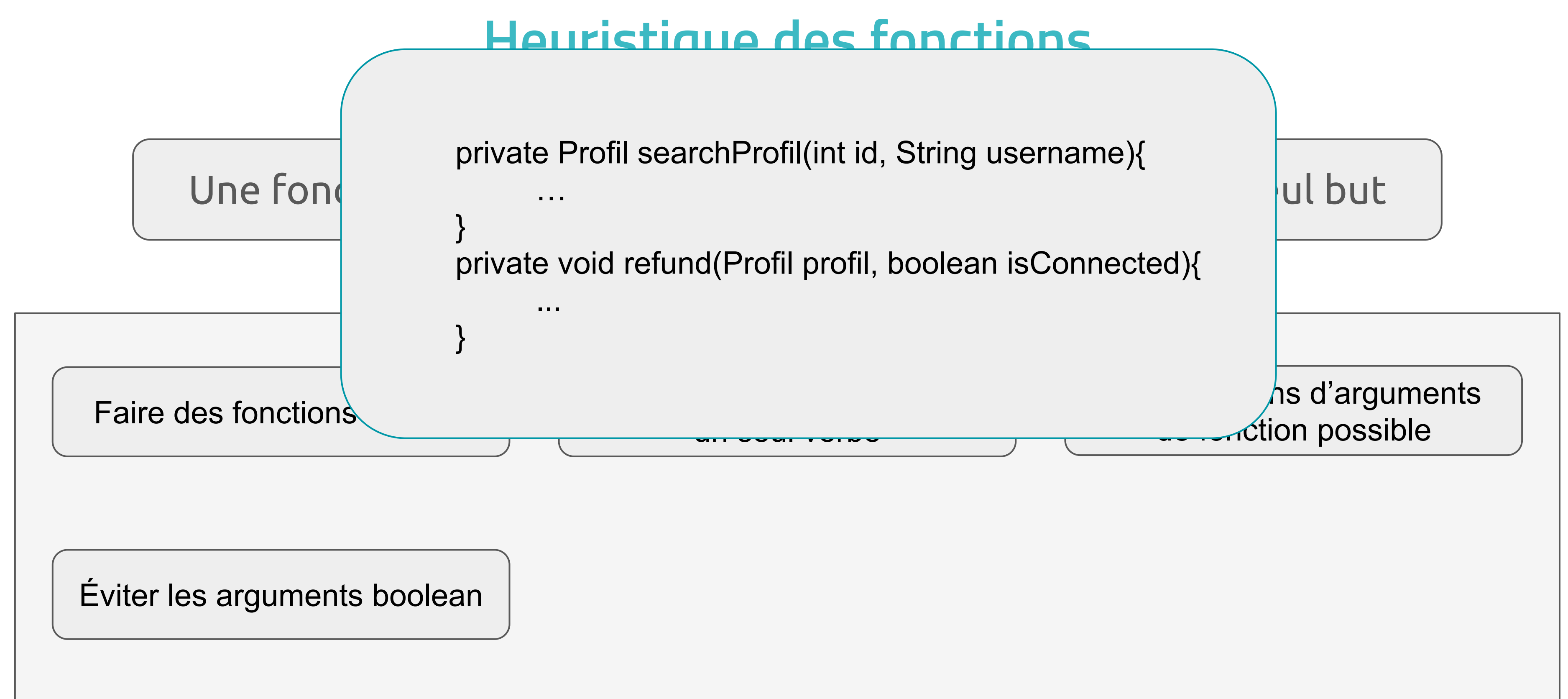
Une fonction ne doit faire qu'une seule chose, avoir qu'un seul but

- Faire des fonctions courtes
- Nommage de fonction avec un seul verbe
- Avoir le moins d'arguments de fonction possible

Heuristique des fonctions

Une fonction ne doit faire qu'une seule chose, avoir qu'un seul but

- Faire des fonctions courtes
- Nommage de fonction avec un seul verbe
- Avoir le moins d'arguments de fonction possible
- Éviter les arguments boolean


Heuristique des fonctions
Une fonc
ul but
private Profil searchProfil(int id, String username){
…
}
private void refund(Profil profil, boolean isConnected){
...
}
Faire des fonctions
ns d’arguments
ction possible
Éviter les arguments boolean

Heuristique des fonctions

Une fonction ne doit faire qu'une seule chose, avoir qu'un seul but

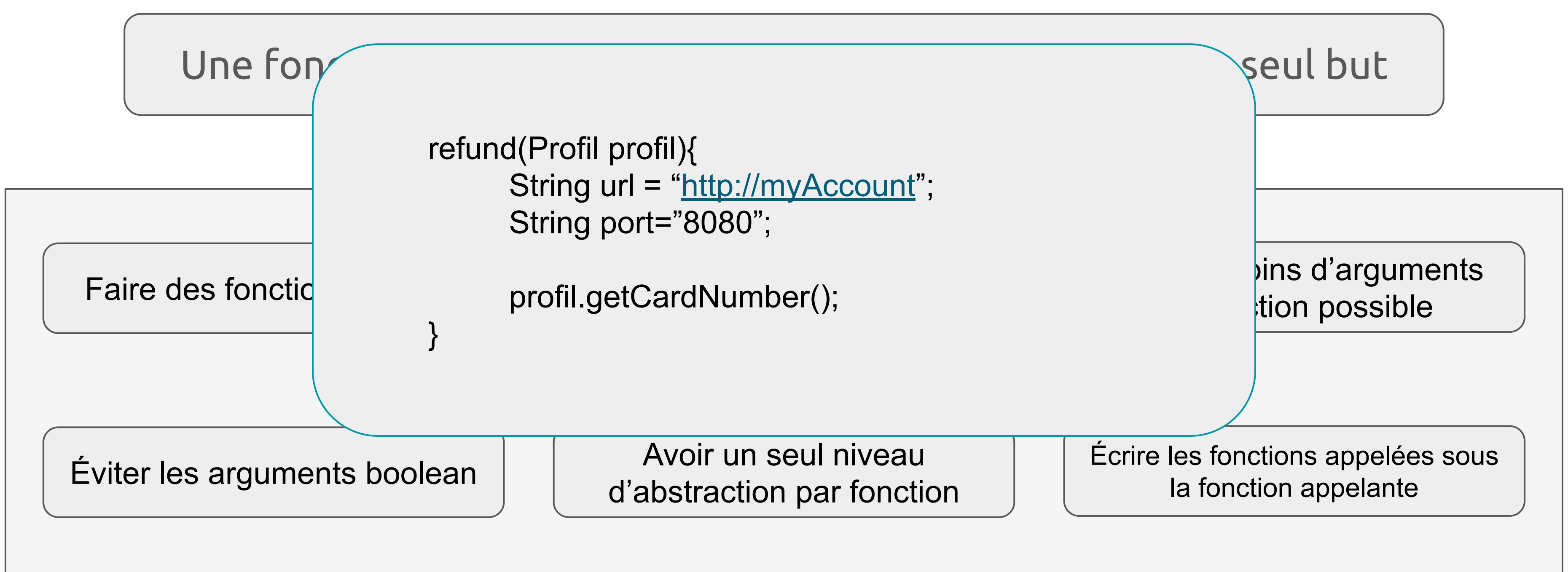
- Faire des fonctions courtes
- Nommage de fonction avec un seul verbe
- Avoir le moins d'arguments de fonction possible
- Éviter les arguments boolean
- Avoir un seul niveau d'abstraction par fonction

Heuristique des fonctions

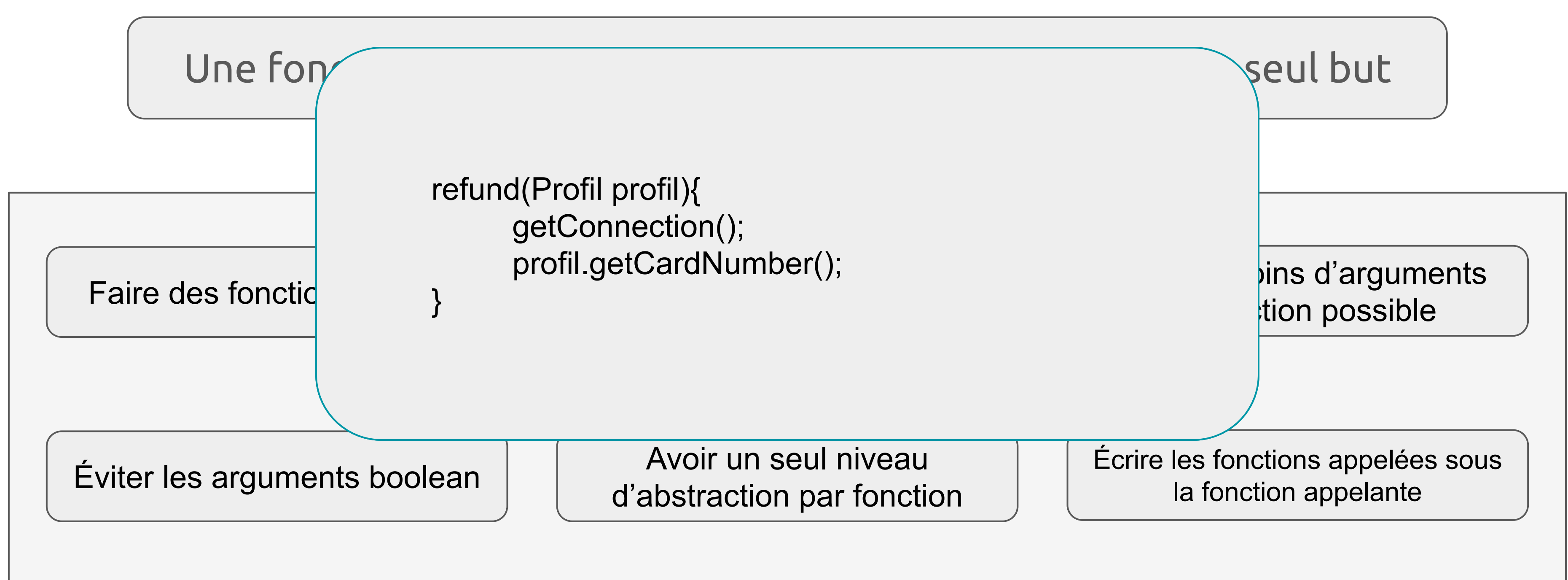
Une fonction ne doit faire qu'une seule chose, avoir qu'un seul but

- Faire des fonctions courtes
- Nommage de fonction avec un seul verbe
- Avoir le moins d'arguments de fonction possible
- Éviter les arguments boolean
- Avoir un seul niveau d'abstraction par fonction
- Écrire les fonctions appelées sous la fonction appelante

Heuristique des fonctions



Heuristique des fonctions




```
refund(Profil profil){
        getConnection();
        profil.getCardNumber();
}
```

Refactoring

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");
```
Noms et types des colonnes en base
```java
        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

Transformation des valeurs en base vers des valeurs Java

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

Parsing des dates au format de l'application (= ISO)

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

Gestion des exceptions

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

Connaissance de la façon
d'intepréter un intervalle de dates

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

Règle métier : quand une ligne est-elle considérée comme active ?

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";
```

Constantes pour ne pas dupliquer de code... Vraiment ?

```java
    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
```

Pas dans une variable, car elle s'appellerait aussi "startDate"

```java
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now(); Testabilité
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

Constantes pour ne pas dupliquer de code... Vraiment ?

Noms et types des colonnes en base

Transformation des valeurs en base vers des valeurs Java

Pas dans une variable, car elle s'appellerait aussi "startDate"

Parsing des dates au format de l'application (= ISO)

Gestion des exceptions

Testabilité

Règle métier : quand une ligne est-elle considérée comme active ?

Connaissance de la façon d'intepréter un intervalle de dates

Suivons les principes du Clean Code pour refactorer

Le nommage est déjà bon

Extraire chaque responsabilité dans une méthode courte

Réorganiser toutes ces méthodes dans des classes cohérentes

La métaphore de l’article de presse

Avoir une vue globale
Plonger dans les détails uniquement si besoin
Hiérarchiser pour faciliter la décision

Titre 1
Sous-titre 1
Paragraphe d'accroche 1
Corps de texte
en pyramide inversée

Titre 2
Sous-titre 2
Paragraphe d'accroche 2
Corps de texte
en pyramide inversée

Titre 3
Sous-titre 3
Paragraphe d'accroche 3
Corps de texte
en pyramide inversée

La métaphore de l'article de presse

Avoir une vue globale

Plonger dans les détails uniquement si besoin

Hiérarchiser pour faciliter la décision

```
entryPoint() {
  stage1();
  stage2();
}
```

```
stage1() {
  actionA();
  actionB();
}

stage2() {
  actionC()
  actionD()
}
```

```
actionA() {
  // Fonction courte
}
actionB() {
  // Fonction courte
}
actionC() {
  // Fonction courte
}
actionD() {
  // Fonction courte
}
```

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                 stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }




}
```

Noms et types des colonnes en base

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}
```

Règle métier : quand une ligne est-elle considérée comme active ?

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

}
```

Testabilité

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

Connaissance de la façon d'intepréter un intervalle de dates

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

}
```

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

}
```

Transformation des valeurs en base vers des valeurs Java

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                 stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

    public DateInterval getDateInterval(String startDateColumnName, String stopDateColumnName) {
        LocalDate startDate = getDateOrNull(startDateColumnName);
        LocalDate stopDate = getDateOrNull(stopDateColumnName);
        return new DateInterval(startDate, stopDate);
    }

}
```

Transformation des valeurs en base vers des valeurs Java

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

    public DateInterval getDateInterval(String startDateColumnName, String stopDateColumnName) {
        LocalDate startDate = getDateOrNull(startDateColumnName);
        LocalDate stopDate = getDateOrNull(stopDateColumnName);
        return new DateInterval(startDate, stopDate);
    }

    public boolean getBooleanOrFalse(String columnName) {
        return resultSet.get(columnName) == Boolean.TRUE;
    }

}
```

Transformation des valeurs en base vers des valeurs Java

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

    public DateInterval getDateInterval(String startDateColumnName, String stopDateColumnName) {
        LocalDate startDate = getDateOrNull(startDateColumnName);
        LocalDate stopDate = getDateOrNull(stopDateColumnName);
        return new DateInterval(startDate, stopDate);
    }

    public boolean getBooleanOrFalse(String columnName) {
        return resultSet.get(columnName) == Boolean.TRUE;
    }

    private LocalDate getDateOrNull(String columnName) {
        String value = (String) resultSet.get(columnName);
        return value == null ? null : parseDate(columnName, value);
    }
}
```

Transformation des valeurs en base vers des valeurs Java

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

    public DateInterval getDateInterval(String startDateColumnName, String stopDateColumnName) {
        LocalDate startDate = getDateOrNull(startDateColumnName);
        LocalDate stopDate = getDateOrNull(stopDateColumnName);
        return new DateInterval(startDate, stopDate);
    }

    public boolean getBooleanOrFalse(String columnName) {
        return resultSet.get(columnName) == Boolean.TRUE;
    }

    private LocalDate getDateOrNull(String columnName) {
        String value = (String) resultSet.get(columnName);
        return value == null ? null : parseDate(columnName, value);
    }

    private LocalDate parseDate(String columnName, String value) {
        try {
            return tryParseDate(value);
        } catch (DateTimeParseException e) {
            throw new FormatException(columnName + " has wrong format: " + value, e);
        }
    }

}
```

Gestion des exceptions

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}

@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

    public DateInterval getDateInterval(String startDateColumnName, String stopDateColumnName) {
        LocalDate startDate = getDateOrNull(startDateColumnName);
        LocalDate stopDate = getDateOrNull(stopDateColumnName);
        return new DateInterval(startDate, stopDate);
    }

    public boolean getBooleanOrFalse(String columnName) {
        return resultSet.get(columnName) == Boolean.TRUE;
    }

    private LocalDate getDateOrNull(String columnName) {
        String value = (String) resultSet.get(columnName);
        return value == null ? null : parseDate(columnName, value);
    }

    private LocalDate parseDate(String columnName, String value) {
        try {
            return tryParseDate(value);
        } catch (DateTimeParseException e) {
            throw new FormatException(columnName + " has wrong format: " + value, e);
        }
    }

    private LocalDate tryParseDate(String value) {
        return LocalDate.parse(value, DateTimeFormatter.ISO_LOCAL_DATE);
    }

}
```

Parsing des dates au format de l'application (= ISO)

```java
@Value
public class HistoryRow {

    String label;
    DateInterval validityInterval;
    boolean deleted;

    public HistoryRow(Row row) {
        label = row.getStringOrNull( columnName: "LABEL");
        validityInterval = row.getDateInterval( startDateColumnName: "START_DATE",
                                                stopDateColumnName: "STOP_DATE");
        deleted = row.getBooleanOrFalse( columnName: "DELETED");
    }

    public boolean isActive() {
        return !deleted && validityInterval.containsToday();
    }

}
```

Noms et types des colonnes en base

Règle métier : quand une ligne est-elle considérée comme active ?

```java
@Value
class DateInterval {

    LocalDate startDate;
    LocalDate stopDate;

    public boolean containsToday() {
        LocalDate today = LocalDate.now();
        return contains(today);
    }

    private boolean contains(LocalDate date) {
        return isStartDateBeforeOrEquals(date) && isStopDateAfterOrEquals(date);
    }

    private boolean isStopDateAfterOrEquals(LocalDate date) {
        return stopDate == null || stopDate.isAfter(date) || stopDate.equals(date);
    }

    private boolean isStartDateBeforeOrEquals(LocalDate date) {
        return startDate == null || startDate.isBefore(date) || startDate.equals(date);
    }

}
```

Testabilité
Mockable

Bonus : méthodes qui peuvent être rendues publiques si l'application en a besoin en grandissant

Connaissance de la façon d'intepréter un intervalle de dates

```java
@RequiredArgsConstructor
class Row {

    private final Map<String, Object> resultSet;

    public String getStringOrNull(String columnName) {
        return (String) resultSet.get(columnName);
    }

    public DateInterval getDateInterval(String startDateColumnName, String stopDateColumnName) {
        LocalDate startDate = getDateOrNull(startDateColumnName);
        LocalDate stopDate = getDateOrNull(stopDateColumnName);
        return new DateInterval(startDate, stopDate);
    }

    public boolean getBooleanOrFalse(String columnName) {
        return resultSet.get(columnName) == Boolean.TRUE;
    }

    private LocalDate getDateOrNull(String columnName) {
        String value = (String) resultSet.get(columnName);
        return value == null ? null : parseDate(columnName, value);
    }

    private LocalDate parseDate(String columnName, String value) {
        try {
            return tryParseDate(value);
        } catch (DateTimeParseException e) {
            throw new FormatException(columnName + " has wrong format: " + value, e);
        }
    }

    private LocalDate tryParseDate(String value) {
        return LocalDate.parse(value, DateTimeFormatter.ISO_LOCAL_DATE);
    }

}
```

Transformation des valeurs en base vers des valeurs Java

Bonus : on n'est plus lié à la Map fournie par Spring : on a notre propre abstraction entre deux, et on peut la faire évoluer pour répondre aux besoins uniques de l'application, comme le getDateInterval(), ou rajouter des validations ou des logs…

Gestion des exceptions

Parsing des dates au format de l'application (= ISO)

Bonus : cette connaissance peut être déplacée dans une classe qui connaît les formats spécifiques à l'application : format de prix, etc.

```java
@Value
public class HistoryRow {

    private static final String START_DATE = "START_DATE";
    private static final String STOP_DATE = "STOP_DATE";

    String label;
    LocalDate startDate;
    LocalDate stopDate;
    boolean deleted;
    boolean active;

    public HistoryRow(Map<String, Object> resultSet) {
        label = (String) resultSet.get("LABEL");

        try {
            startDate = resultSet.get(START_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(START_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(START_DATE + " has wrong format: " + resultSet.get(START_DATE), e);
        }

        try {
            stopDate = resultSet.get(STOP_DATE) == null
                    ? null
                    : LocalDate.parse((String) resultSet.get(STOP_DATE), DateTimeFormatter.ISO_LOCAL_DATE);
        } catch (DateTimeParseException e) {
            throw new FormatException(STOP_DATE + " has wrong format: " + resultSet.get(STOP_DATE), e);
        }

        deleted = resultSet.get("DELETED") == Boolean.TRUE;

        LocalDate today = LocalDate.now();
        active = !deleted &&
                (startDate == null || (startDate.isBefore(today) || startDate.equals(today))) &&
                (stopDate == null || (stopDate.isAfter(today) || stopDate.equals(today)));
    }

}
```

Constantes pour ne pas dupliquer de code… Vraiment ?

Noms et types des colonnes en base

Transformation des valeurs en base vers des valeurs Java

Pas dans une variable, car elle s'appellerait aussi "startDate"

Parsing des dates au format de l'application (= ISO)

Gestion des exceptions

Testabilité

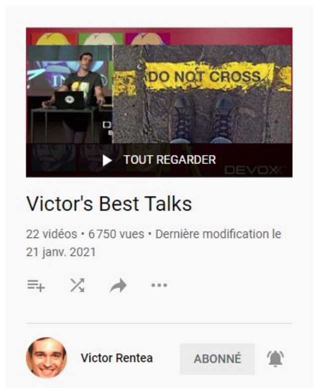
Règle métier : quand une ligne est-elle considérée comme active ?

Connaissance de la façon d'intepréter un intervalle de dates

Liens

Vidéos

Clean Code - Uncle Bob
(leçons 1 à 6 résumant le livre de manière vivante)
https://www.youtube.com/watch?v=7EmboKQH8lM&list=PLmmYSbUCWJ4x1GO839azG_BBw8rkh-zOj



Exercices de nettoyage de codes de Victor Rentea
(excellent pédagogue)
https://www.youtube.com/playlist?list=PLggcOULvfLL_MfFS_O0MKQ5W_6oWWbIw5

Les Code Smells du livre

https://moderatemisbehaviour.github.io/clean-code-smells-and-heuristics/

G12: Clutter

Of what use is a default constructor with no implementation? All it serves to do is clutter up the code with meaningless artifacts. Variables that aren't used, functions that are never called, comments that add no information, and so forth. All these things are clutter and should be removed. Keep your source files clean, well organized, and free of clutter.

G13: Artificial Coupling

Things that don't depend upon each other should not be artificially coupled. For example, general `enums` should not be contained within more specific classes because this forces the whole application to know about these more specific classes. The same goes for general purpose `static` functions being declared in specific classes.

In general an artificial coupling is a coupling between two modules that serves no direct purpose. It is a result of putting a variable, constant, or function in a temporarily convenient, though inappropriate, location. This is lazy and careless.

Take the time to figure out where functions, constants, and variables ought to be declared. Don't just toss them in the most convenient place at hand and then leave them there.

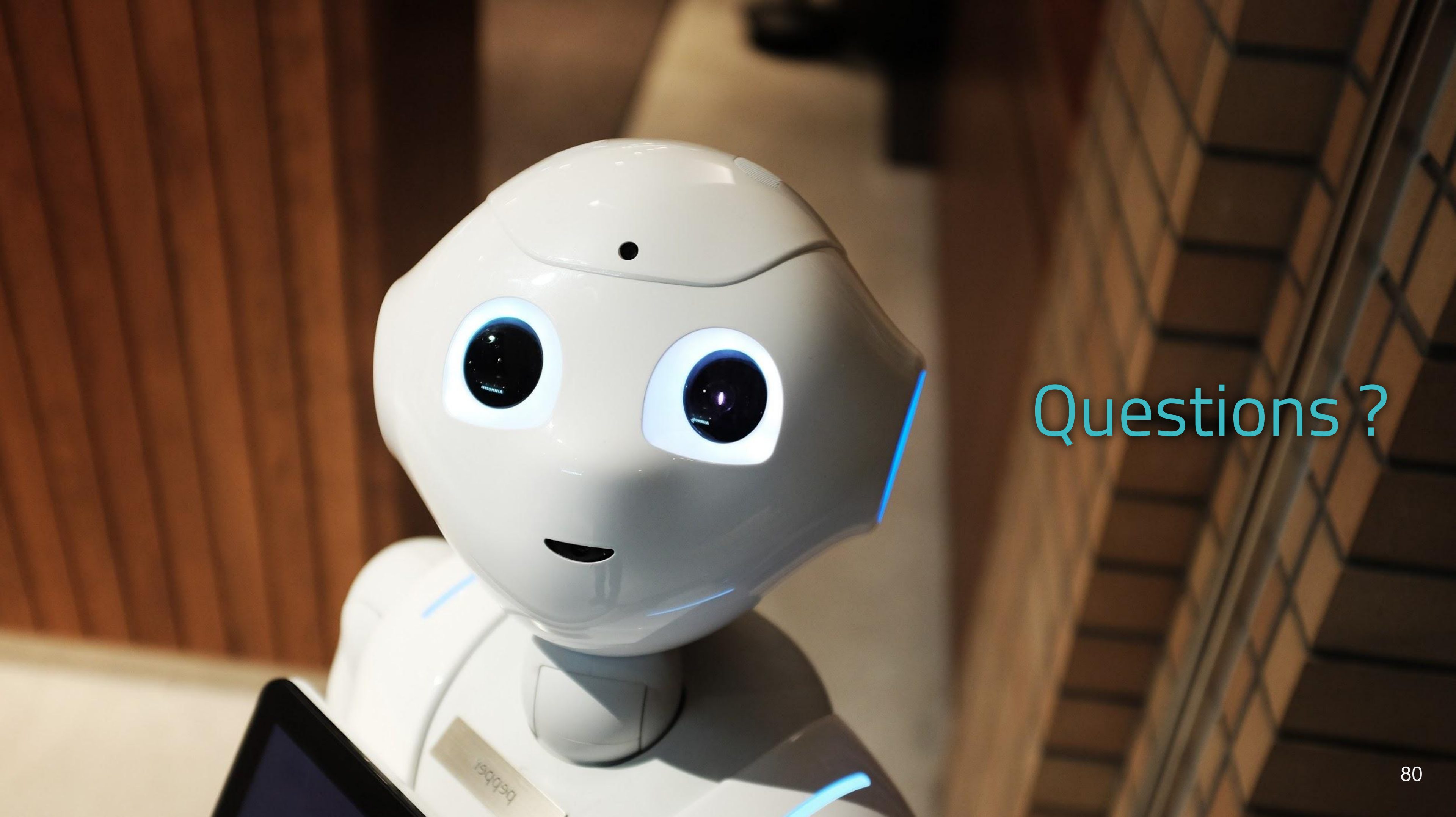
G14: Feature Envy

This is one of Martin Fowler's code smells[1]. The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. When a method uses accessors and mutators of some other object to manipulate the data within that object, then it *envies* the scope of the class of that other object. It wishes that it were inside that other class so that it could have direct access to the variables it is manipulating. For example:

```
public class HourlyPayCalculator {
  public Money calculateWeeklyPay(HourlyEmployee e) {
    int tenthRate = e.getTenthRate().getPennies();
```

Aides mémoire de refactoring

https://drive.google.com/drive/folders/1hvLAWeAgT753Mkb8zaHxNyBYwCewuWbb (par Victor Rentea)


Questions ?

Crédits des images

Titre :	https://www.pexels.com/fr-fr/photo/vue-de-l-exterieur-du-batiment-327483/
Plan :	Internet...
Exemple :	https://www.pexels.com/fr-fr/photo/texte-6140480/
Objectifs :	https://www.pexels.com/fr-fr/photo/gros-plan-de-main-humaine-327533/
Principes & méthode :	https://www.pexels.com/fr-fr/photo/bibliotheque-interieure-lumineuse-590493/
Nommage :	https://www.pexels.com/fr-fr/photo/etiquette-de-produit-blanche-1111319/
Révéler l'intention :	https://www.pexels.com/fr-fr/photo/homme-travaillant-ordinateur-2696299/
Cherchable :	https://www.pexels.com/fr-fr/photo/personne-tenant-une-boussole-1125272/
Au fait... :	https://www.pexels.com/fr-fr/photo/illustration-point-d-interrogation-356079/
Classes :	https://www.pexels.com/fr-fr/photo/bois-art-peinture-maison-6692956/
Fonctions :	https://www.pexels.com/fr-fr/photo/ordinateur-portable-noir-et-gris-546819/
Refactoring :	https://www.pexels.com/fr-fr/photo/homme-portant-une-pipe-grise-585419/
Liens :	https://www.pexels.com/fr-fr/photo/corde-noire-de-machine-d-exercice-moderne-accrochee-a-un-mousqueton-en-metal-3839053/
Questions ? :	https://www.pexels.com/fr-fr/photo/photo-en-grand-angle-du-robot-2599244/