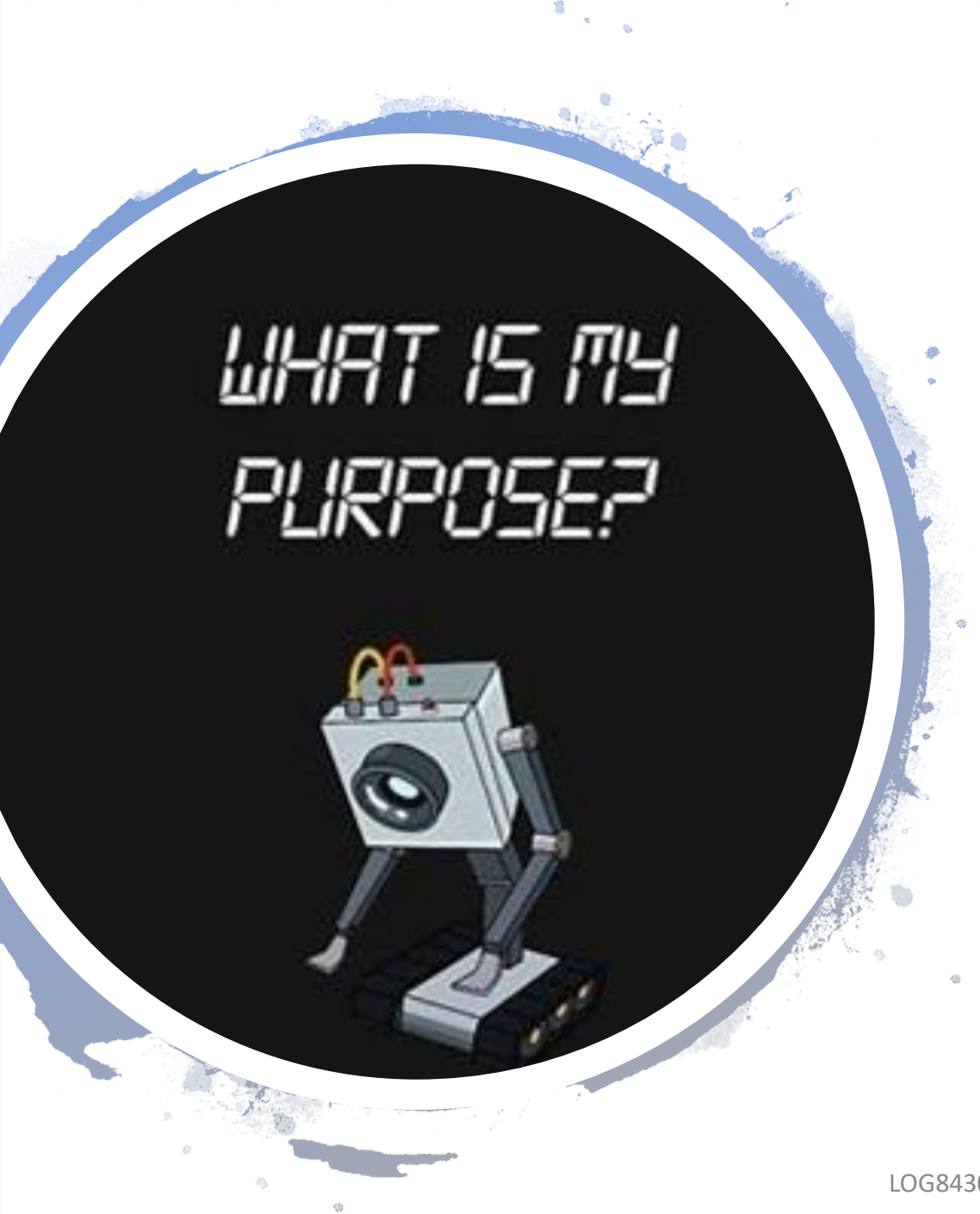
POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

LOG8430: Principes (SOLID) et Patrons de Conception

Aujourd'hui

LOG2410

Conception
OO (SOLID)



Patrons, Principes et Bonnes Pratiques

- Pour livrer plus vite
- Pour gérer le changement
- Pour gérer la complexité


SOLID
Software development is not a Jenga game.

« Le développement de logiciel n'est pas un jeu de Jenga. »

- Au début de la phase de la conception, le système semble beau, élégant, pur…
- …mais ensuite, pendant le développement, on a des *hacks*, des implémentations vites faites et négligentes pour respecter les délais.
- Le logiciel commence à pourrir!
- Quels sont les symptômes d'une conception en train de pourrir?
  - Dans les prochains cours!
- Pour s'assurer de maintenir la qualité de la conception, on doit se baser sur des principes SOLID!
  - Introduits par Oncle Bob (Robert C. Martin)
- Quels sont ces principes?

Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

« Parce que tu peux ne signifie pas que tu dois. »

- Une classe doit avoir une *seule responsabilité*.

SRP violé?

```
class Invoice {
    private String account;
    private String name;
    private HashMap<String,double> productsBought;

    public String getAccount() {…};
    public void setAccount() {…};
    …
    public String formatInvoice(String formatType) {
        switch(formatType) {
            case "JSON":
                \\implement JSON formatting here.
                return jsonInvoice;
                break;
            default:
                \\implement default formatting here.
                return defaultInvoice;
                break;
        }
    }
}
```

SRP violé?

```
class Invoice {
    private String account;
    private String name;
    private HashMap<String,double> productsBought;

    public String getAccount() {…};
    public void setAccount() {…};
    …
    public String formatInvoice(String formatType) {
        switch(formatType) {
            case "JSON":
                \\implement JSON formatting here.
                return jsonInvoice;
                break;
            default:
                \\implement default formatting here.
                return defaultInvoice;
                break;
        }
    }
}
```

- La class Invoice contient:
  1. Les données spécifiques à un objet Invoice
  2. La fonctionnalité pour produire des factures des différents formats
- Peut-être on doit diviser la classe…?

SRP garanti!

```
class Invoice {

    private String account;

    private String name;

    private HashMap<String,double>
productsBought;


    public String getAccount() {…};
    public void setAccount() {…};

    …

}
```

```
class InvoiceFormatter {
 public String formatInvoice(String formatType) {
        switch(formatType) {
            case "JSON":
                \\implement JSON formatting here.
                return jsonInvoice;
                break;
            default:
                \\implement default formatting here.
                return defaultInvoice;
                break;
        }
    }}

}
```

« Parce que tu peux ne signifie pas que tu dois. »

- Une classe ne devrait avoir qu'une *seule responsabilité*.
- …mais qu'est-ce que ça veut dire?
- Une classe ne devrait avoir qu'une *seule raison de changer.*
- Est-il possible de prévoir les changements d'une classe?

SRP violé?

```
class Invoice {
    private String account;
    private String name;
    private HashMap<String,double> productsBought;

    public String getAccount() {…};
    public void setAccount() {…};
    …
    public String formatInvoice(String formatType) {
        switch(formatType) {
            case "JSON":
                \\implement JSON formatting here.
                return jsonInvoice;
                break;
            default:
                \\implement default formatting here.
                return defaultInvoice;
                break;
        }
    }
}
```

- La class Invoice contient:
  1. Les données spécifiques à un objet Invoice
  2. La fonctionnalité pour produire des factures des différents formats
- Que va-t-il se passer si
  1. on veut ajouter un nouveau format?
  2. on change le contenu de la facture?


Intermission

(Quelques) principes GRASP

- GRASP: General Responsibility Assignment Software Principles
- Haute cohésion
  - Les méthodes d'une classe utilisent beaucoup d'autres membres de la classe elle-même

```
class HelloWorld {
    private String firstName;
    private String lastName;

    public String sayHelloWorld() {
        return "Hello, "+firstName+" "+lastName+"!";
    }
}
```

(Quelques) principes GRASP

- Faible couplage
  - Les méthodes d'une classe utilisent peu de membres d'autres classes.

```
class HelloWorld {
    private String greeting;
    private String question;

    public String sayHelloWorld(Person person) {
        return greeting+ “, ”+person.getName()+“. ”+question;
    }
}

class Person {
    private String name;

    public String getName() {
        return name;
    }
}
```

« Parce que tu peux ne signifie pas que tu dois. »

- Une classe ne devrait avoir qu'une *seule responsabilité*.
- …mais qu'est-ce que ça veut dire?
- Une classe ne devrait avoir qu'une *seule raison de changer.*
- Est-il possible de prévoir les changements d'une classe?
- Haute cohésion + Faible couplage ➔ Moins de responsabilités, moins de raisons de changer.
- **Alors,** maintenez la cohésion et le couplage de la classe!


Open-Closed Principle
Open-chest surgery isn't needed when putting on a coat.

« La chirurgie à cœur ouvert n'est pas nécessaire pour mettre un manteau. »

- Les entités logicielles doivent être ouvertes aux extensions, mais fermées aux modifications.
- Pensez toujours : « qu’est-ce qui va se passer si l’entité change? »

OCP violé?

Bcp de else if ajoutés, la complexité sera affectée! Elle augmente si on augmente les branches dans le code. Testabilité réduite. Problématique, car on doit ajouter des cycles et la complexité sera augmentée

```
14 +     public void handleEvent(Event event) {
15 +         if (event instanceof PreLoad) {
16 +             ei.preLoad(entity);
17 +         } else if (event instanceof PostLoad) {
18 +             ei.postLoad(entity);
19 +         } else if (event instanceof PrePersist) {
20 +             ei.prePersist(entity);
21 +         } else if (event instanceof PreSave) {
22 +             ei.preSave(entity);
23 +         } else if (event instanceof PostPersist) {
24 +             ei.postPersist(entity);
25 +         }
26 +     }
```

OCP violé?

```
14 +    public void handleEvent(Event event) {
15 +        if (event instanceof PreLoad) {
16 +            ei.preLoad(entity);
17 +        } else if (event instanceof PostLoad) {
18 +            ei.postLoad(entity);
19 +        } else if (event instanceof PrePersist) {
20 +            ei.prePersist(entity);
21 +        } else if (event instanceof PreSave) {
22 +            ei.preSave(entity);
23 +        } else if (event instanceof PostPersist) {
24 +            ei.postPersist(entity);
25 +        }
26 +    }
```

- Que va-t-il se passer si on a besoin d'un nouveau évènement?
- Que va-t-il se passer si on a trop d'évènements à gérer?
- Le code semble étrangement répétitif, n'est-ce pas ?

EventHandler.java @ 91c499d

```java
package com.mechanitis.blog.upsource.infrastructure;

import com.mechanitis.blog.upsource.infrastructure.event.Event;
import com.mechanitis.blog.upsource.infrastructure.event.PostLoad;
import com.mechanitis.blog.upsource.infrastructure.event.PostPersist;
import com.mechanitis.blog.upsource.infrastructure.event.PreLoad;
import com.mechanitis.blog.upsource.infrastructure.event.PrePersist;
import com.mechanitis.blog.upsource.infrastructure.event.PreSave;

public class EventHandler {
    private Object entity;
    private EventInterceptor ei;

    public void handleEvent(Event event) {
        if (event instanceof PreLoad) {
            ei.preLoad(entity);
        } else if (event instanceof PostLoad) {
            ei.postLoad(entity);
        } else if (event instanceof PrePersist) {
            ei.prePersist(entity);
        } else if (event instanceof PreSave) {
            ei.preSave(entity);
        } else if (event instanceof PostPersist) {
            ei.postPersist(entity);
        }
    }
```

EventHandler.java @ d0e6048

```java
package com.mechanitis.blog.upsource.infrastructure;

import com.mechanitis.blog.upsource.infrastructure.event.Event;

public class EventHandler {
    private Object entity;
    private EventInterceptor ei;

    public void handleEvent(Event event) {

        ei.interceptEvent(event, entity);

    }
```

EventInterceptor.java @ 91c499d

```java
public class EventInterceptor {
    public void preLoad(Object entity) {

    }

    public void postLoad(Object entity) {

    }

    public void prePersist(Object entity) {

    }

    public void preSave(Object entity) {

    }

    public void postPersist(Object entity) {

    }
}
```

EventInterceptor.java @ d0e6048

```java
public class EventInterceptor {
    public void interceptEvent(PreLoad event, Object entity) {
        //do pre-load stuff here
    }

    public void interceptEvent(PostLoad event, Object entity) {
        //do post-load stuff here
    }

    public void interceptEvent(PrePersist event, Object entity) {
        //do pre-persist stuff here
    }

    public void interceptEvent(PreSave event, Object entity) {
        //do pre-save stuff here
    }

    public void interceptEvent(PostPersist event, Object entity) {
        //do post-persist stuff here
    }

    public void interceptEvent(Event event, Object entity) {
        //do some general handling here for other events
    }
}
```

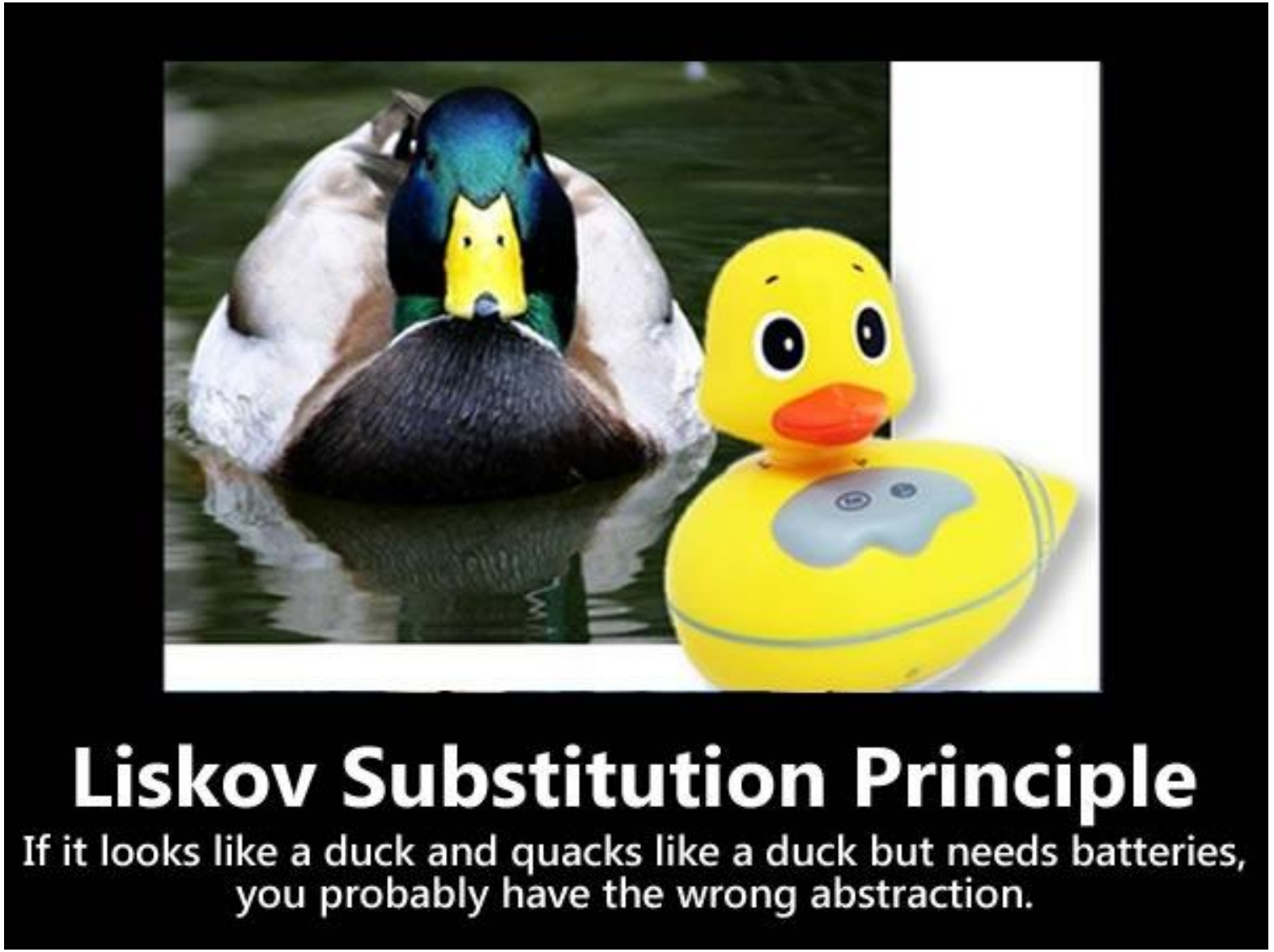
POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

OCP garanti!

- Polymorphisme est votre ami!
- Contrôle des types (« Type Checking ») est toujours une indication pour une hiérarchie.

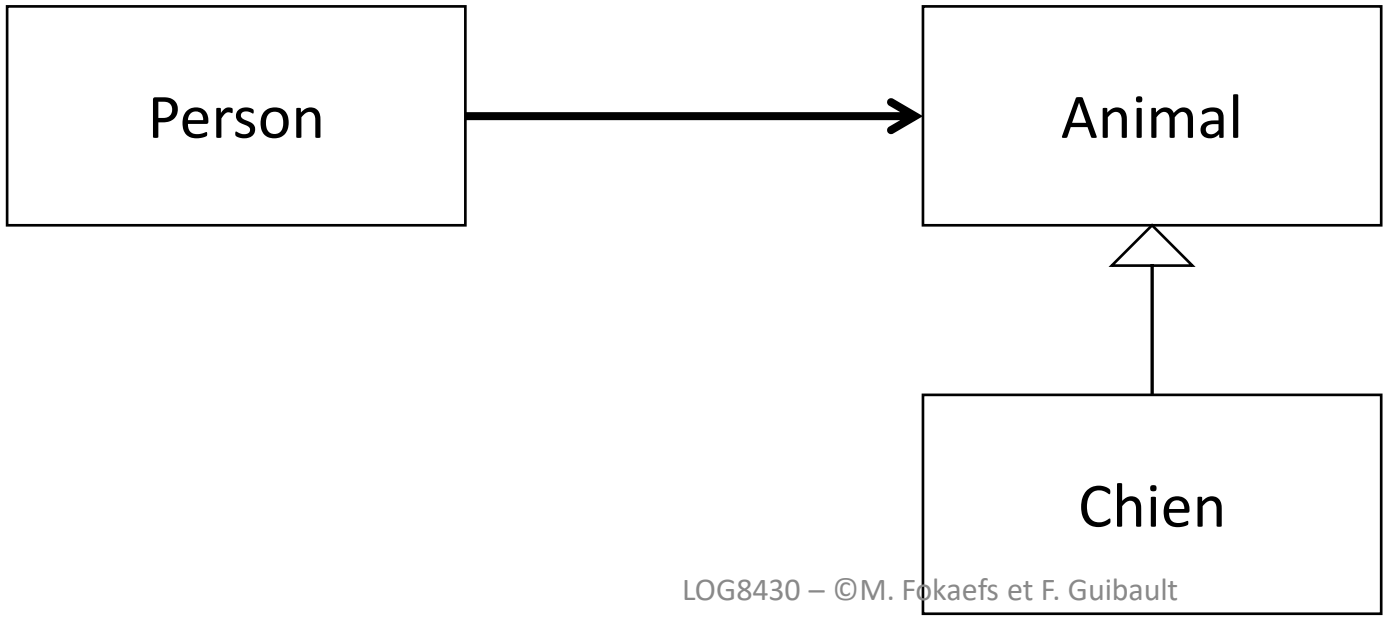
« La chirurgie à cœur ouvert n'est pas nécessaire pour mettre un manteau. »

- Les entités logicielles doivent être ouvertes aux extensions, mais fermées aux modifications.
- Pensez toujours : « qu'est-ce qui va se passer si l'entité change? »
- L'héritage et le polymorphisme favorisent l'extension.
- Refactoring : Remplacer la vérification de type par du polymorphisme (si la hiérarchie existe déjà).
- Refactoring : Remplacer la vérification de type par le patron « Stratégie ». (si la hiérarchie n'existe pas encore).
  - On va voir les deux dans les prochains cours.


Liskov Substitution Principle
If it looks like a duck and quacks like a duck but needs batteries,
you probably have the wrong abstraction.

« Si cela ressemble à un canard, sonne comme un canard, mais a besoin de piles, vous avez probablement la mauvaise abstraction. »

- Les fonctions qui utilisent des références à des classes de base doivent pouvoir utiliser des objets de classes dérivées sans le savoir.
- Toutes les méthodes qui prennent un paramètre de type « Animal », peuvent accepter un argument de type « Chien ».



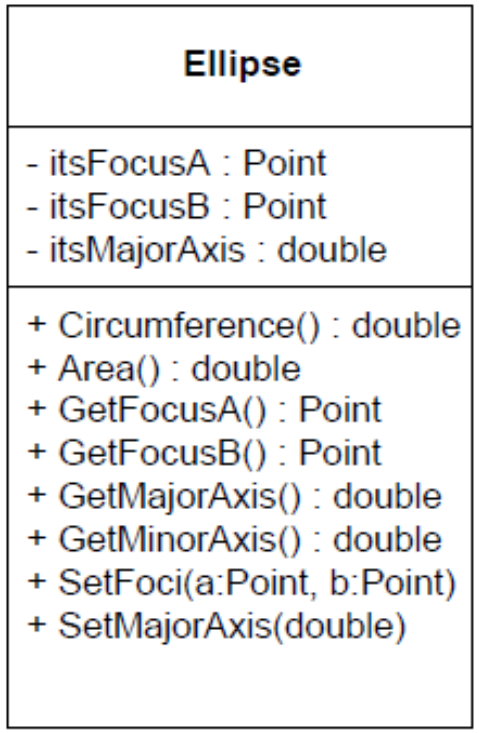
« Si cela ressemble à un canard, sonne comme un canard, mais a besoin de piles, vous avez probablement la mauvaise abstraction. »

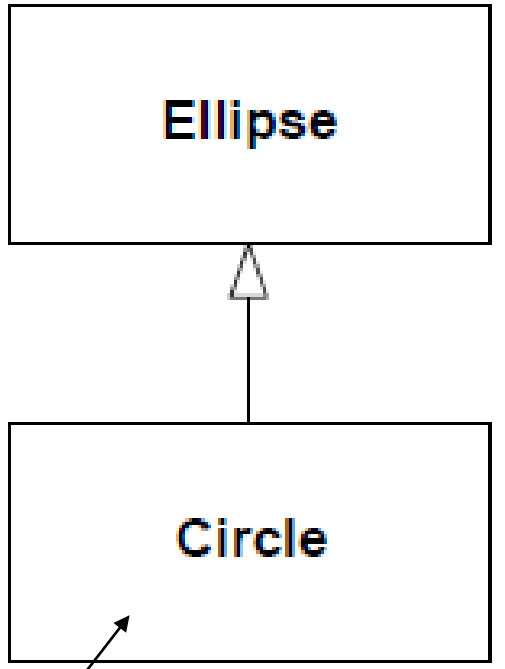
- Les fonctions qui utilisent des références à des classes de base doivent pouvoir utiliser des objets de classes dérivées sans le savoir.
- Les préconditions ne peuvent pas être renforcées dans le type dérivé.
  - Les méthodes dérivées ne peuvent pas attendre plus que les méthodes de base.
- Les postconditions ne peuvent pas être plus faibles dans le type dérivé.
  - Les méthodes dérivées ne peuvent pas fournir moins que les méthodes de base.

On ne peut retourner un string vide ou un NULL. PAs accepté par le client

LSP violé

- Un cercle est une ellipse dont les deux foyers coïncident.






```
void f(Ellipse e) {
    Point a = new Point(-1,0);
    Point b = new Point(1,0);
    e.setFoci(a,b);
    e.setMajorAxis(3);
    assert(e.getFocusA() == a);
    assert(e.getFocusB() == b);
    assert(e.getMajorAxis() == 3);
}
```

```
void setFoci(Point a, Point b) {
    itsFocusA = a;
    itsFocusB = a;
}
```

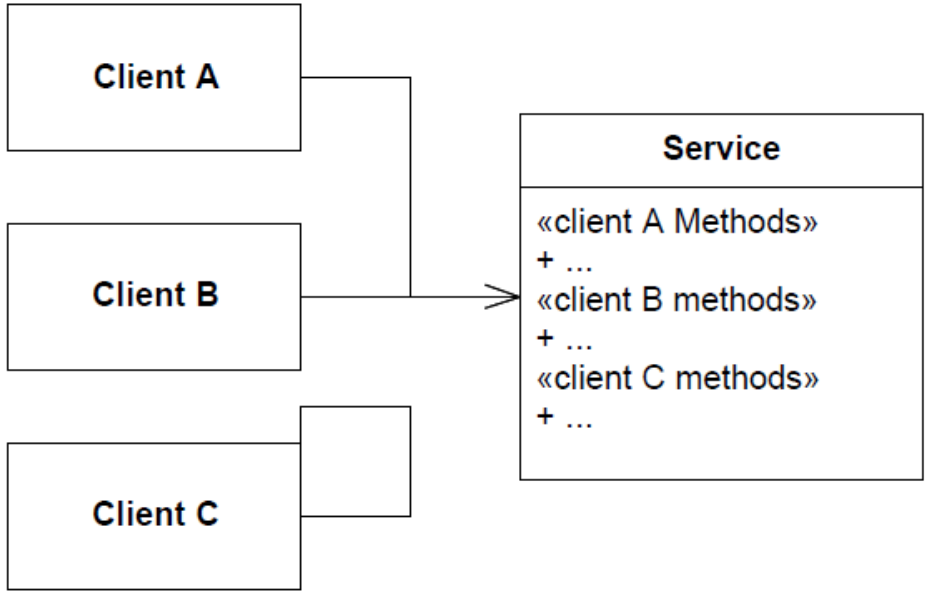
- OCP violé!!

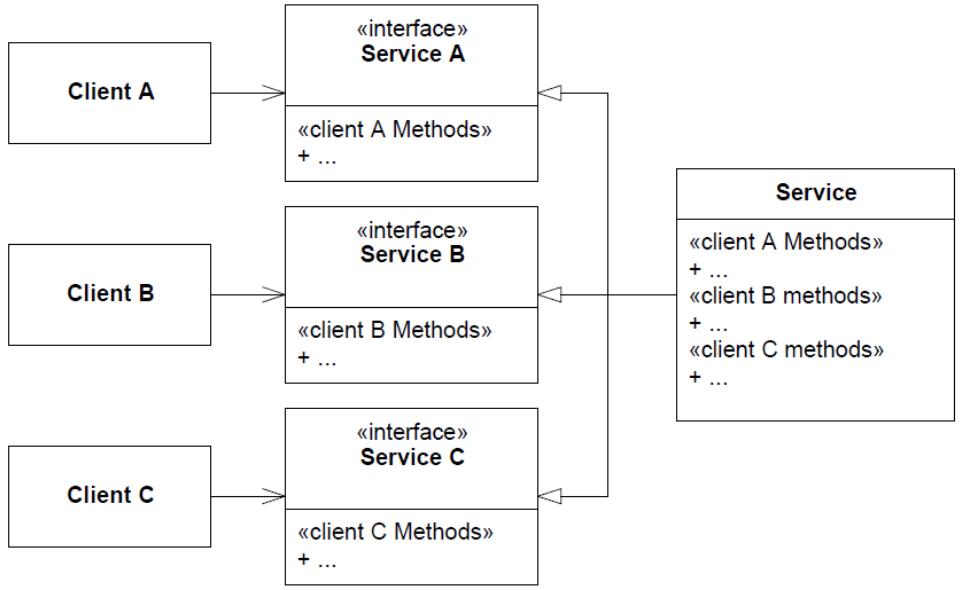
```
if(e instanceof Ellipse) {
```


Interface Segregation Principle
You want me to plug this in *where?*

« Tu veux que je mette ça où? »

- De nombreuses interfaces spécifiques aux clients valent mieux qu'une seule interface.





ISP violé?

```
+ public class StringCodec implements SimpleCodec<String> {
+
+     @Override
+     public String decode(Reader reader) {
+         throw new UnsupportedOperationException("Should never have to decode a String object");
+     }
+
+     @Override
+     public void encode(final String value, Writer writer) {
+         writer.writeString(value);
+     }
+ }
```

- SimpleCodec est une interface qui spécifie deux méthodes (decode et encode).
  - Mais StringCodec a besoin seulement d'encoder.

StringCodecpeut décoder/encoder un string

ISP violé?

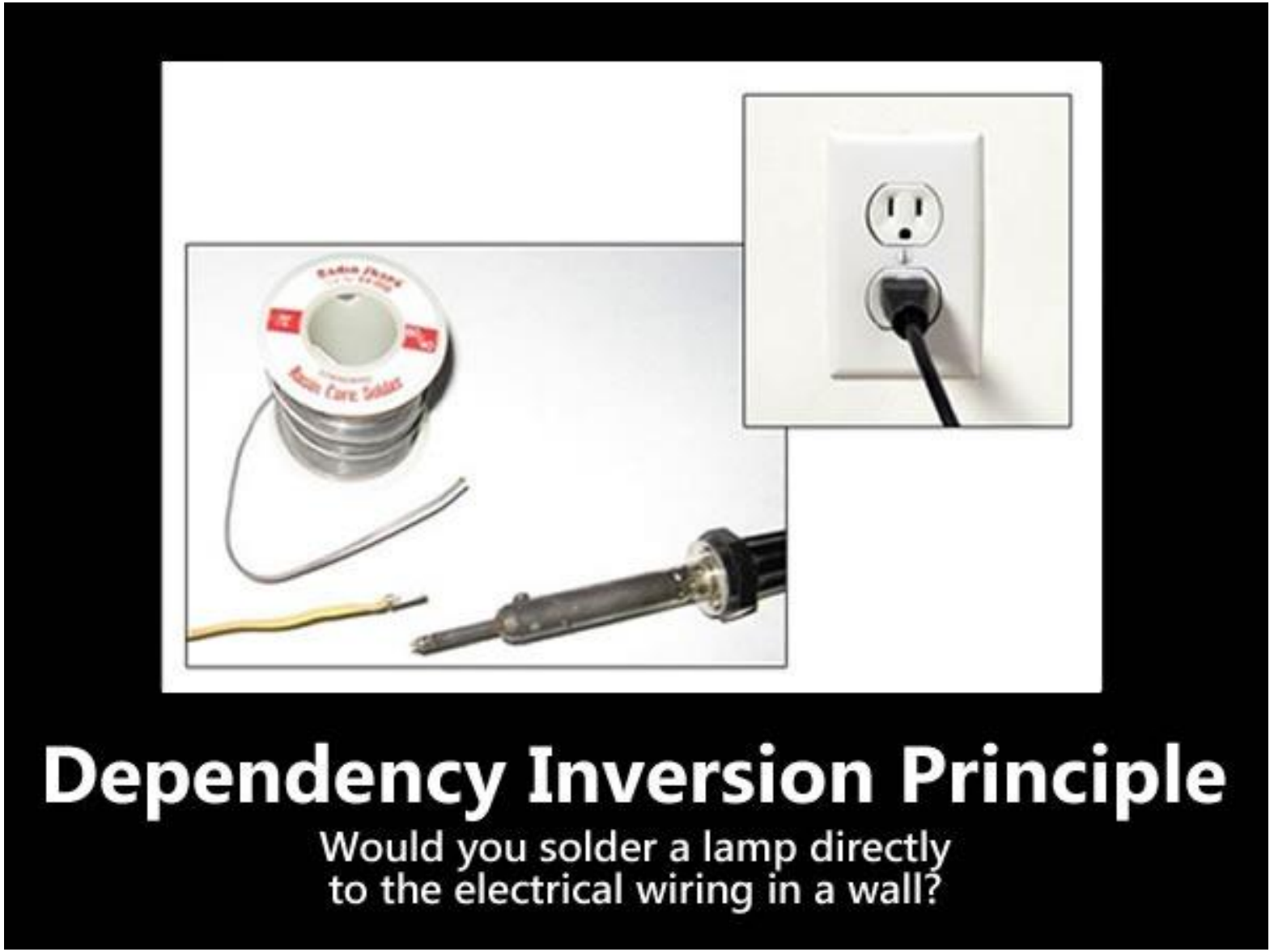
```
6  + public class StringCodec implements SimpleCodec<String> {
7  +
8  +     @Override
9  +     public String decode(Reader reader) {
10 +         throw new UnsupportedOperationException("Should never have to decode a String object");
11 +     }
12 +
13 +     @Override
14 +     public void encode(final String value, Writer writer) {
15 +         writer.writeString(value);
16 +     }
17 + }
```

- SimpleCodec spécifie une interface complexe qui ne convient pas à tous les clients.
- Diviser `SimpleCodec` en un `Encoder` et un `Decoder`.

« Tu veux que je mette ça où? »

- De nombreuses interfaces spécifiques aux clients valent mieux qu'une seule interface.
- Le cas d'Amazon EC2 : Au travers de 16 versions, les opérations dans l'interface du service ont augmenté de 15 à 90.
  - Il y a des considérations spéciales pour les systèmes SOA.
  - On va les voir aux prochains cours!

Fokaefs, M., & Stroulia, E. (2014). Wsdarwin: Studying the evolution of web service systems. In Advanced Web Services (pp. 199-223). Springer, New York, NY.


Dependency Inversion Principle
Would you solder a lamp directly
to the electrical wiring in a wall?

« Souderiez-vous une lampe directement sur le câblage électrique dans le mur? »

- Dépend des abstractions. Ne dépend pas de classes concrètes.

DEPENCY INVERSION PRINCIPLE

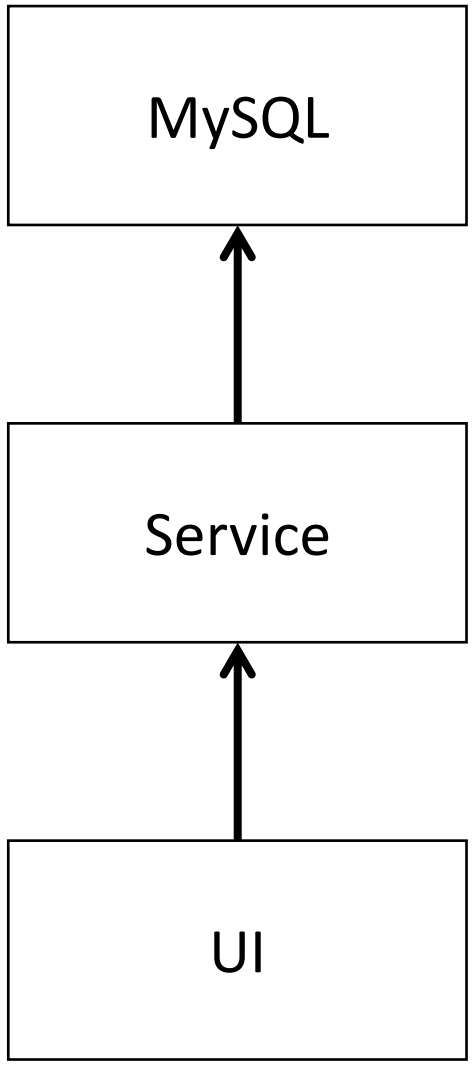
DIP violé?

```
6 12  public class CustomerService {
   13 +    private static final String INSERT_CUSTOMER_STATEMENT = "INSERT INTO Customers"
   14 +                                                          + "(id, first_name, last_name)"
   15 +                                                          + "VALUES"
   16 +                                                          + "( ?, ?, ? )";
   17 +    private Database database;
   18 +
   19 +    public CustomerService(Database database) {
   20 +        this.database = database;
   21 +    }
   22 +
   23 +    public void addCustomer(int id, String firstName, String lastName) {
   24 +        try (Connection connection = database.getConnection();
   25 +             PreparedStatement statement = connection.prepareStatement(INSERT_CUSTOMER_STATEMENT)) {
   26 +            statement.setInt(1, id);
   27 +            statement.setString(2, firstName);
   28 +            statement.setString(3, lastName);
   29 +
   30 +            statement.executeUpdate();
   31 +        } catch (SQLException e) {
   32 +            doDatabaseErrorHandling(e);
   33 +        }
   34 +    }
   35 +
7  36       public void placeOrder(Customer customer, Order order) {
8  37           customer.incrementOrders();
9  38           order.placeOrder(getWarehouse());
10 39       }
```

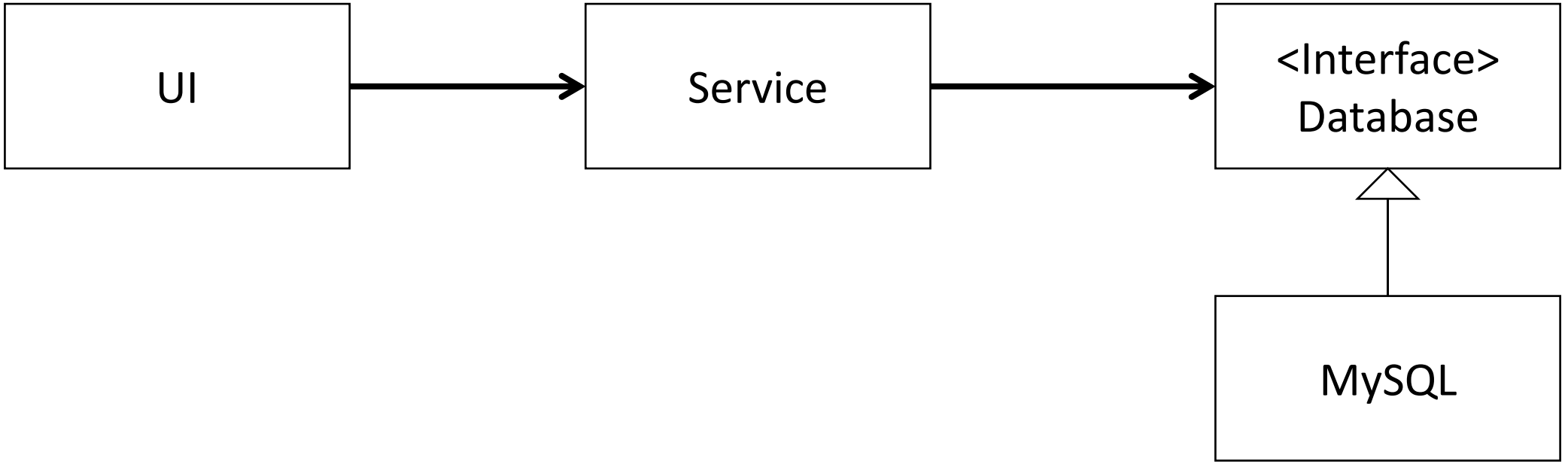
DIP violé?

Code SQL. Trés spécifique…

```
6 12  public class CustomerService {
13  +     private static final String INSERT_CUSTOMER_STATEMENT = "INSERT INTO Customers"
14  +                                                           + "(id, first_name, last_name)"
15  +                                                           + "VALUES"
16  +                                                           + "( ?, ?, ? )";
17  +     private Database database;
18  +
19  +     public CustomerService(Database database) {
20  +         this.database = database;
21  +     }
22  +
23  +     public void addCustomer(int id, String firstName, String lastName) {
24  +         try (Connection connection = database.getConnection();
25  +              PreparedStatement statement = connection.prepareStatement(INSERT_CUSTOMER_STATEMENT)) {
26  +             statement.setInt(1, id);
27  +             statement.setString(2, firstName);
28  +             statement.setString(3, lastName);
29  +
30  +             statement.executeUpdate();
31  +         } catch (SQLException e) {
32  +             doDatabaseErrorHandling(e);
33  +         }
34  +     }
35  +
7 36      public void placeOrder(Customer customer, Order order) {
8 37          customer.incrementOrders();
9 38          order.placeOrder(getWarehouse());
10 39     }
```



DIP garanti!



« Souderiez-vous une lampe directement sur le câblage électrique dans le mur? »

- Dépend des abstractions. Ne dépend pas des classes concrètes.
- Le Saint Graal de l'architecture logicielle!
- Si on veut changer la base de données, combien de classes doit-on changer?
- Si on dépend de bibliothèques logicielles, notre système est enfermé dans un langage spécifique. (DIP violé)
- Si on dépend des APIs REST, notre système reste indépendant des langage. (DIP garanti)


SOLID
Software development is not a Jenga game.

La prochaine fois

LOG2410

Conception OO (SOLID)

Patrons de Conception

Styles et Patrons d'Architectures Distribuées