Trip Service Kata

Isabelle BLASQUEZ
@iblasquez

Un Kata sur du code legacy

Contexte métier : **Réseau Social pour voyageurs**

Règles métier:

→ **Il faut être connecté(e) pour accéder au contenu du site.**

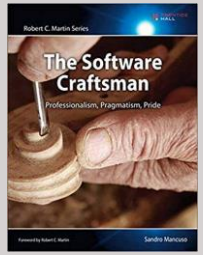
→ **Il faut faire partie des ami(e)s d'un utilisateur pour voir ses voyages.**

Objectif : **Tester** *puis* **refactorer** le code legacy

Idée originale de Sandro Mancuso :

https://github.com/sandromancuso/trip-service-kata

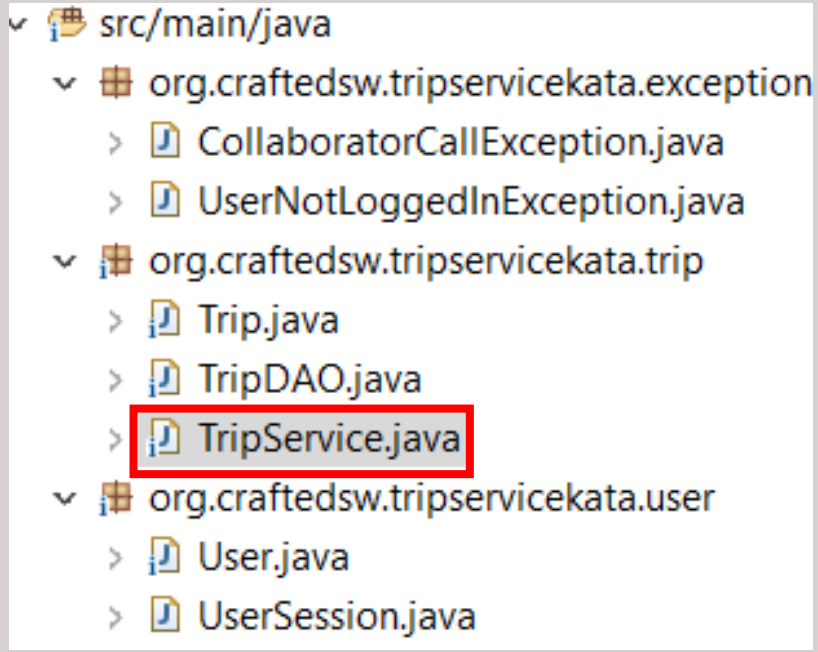
(existe pour différents langages)



Code Legacy

Pour la phase de couverture par les tests
Se focaliser uniquement sur la classe `TripService`

```
public class TripService {

    public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
        List<Trip> tripList = new ArrayList<Trip>();
        User loggedUser = UserSession.getInstance().getLoggedUser();
        boolean isFriend = false;
        if (loggedUser != null) {
            for (User friend : user.getFriends()) {
                if (friend.equals(loggedUser)) {
                    isFriend = true;
                    break;
                }
            }
            if (isFriend) {
                tripList = TripDAO.findTripsByUser(user);
            }
            return tripList;
        } else {
            throw new UserNotLoggedInException();
        }
    }
}
```



Première Partie

(Couverture du code legacy par des tests)

Mettre en place
un harnais de tests
pour garantir
le *bon* comportement
du produit

Pas de refactoring, sans test !

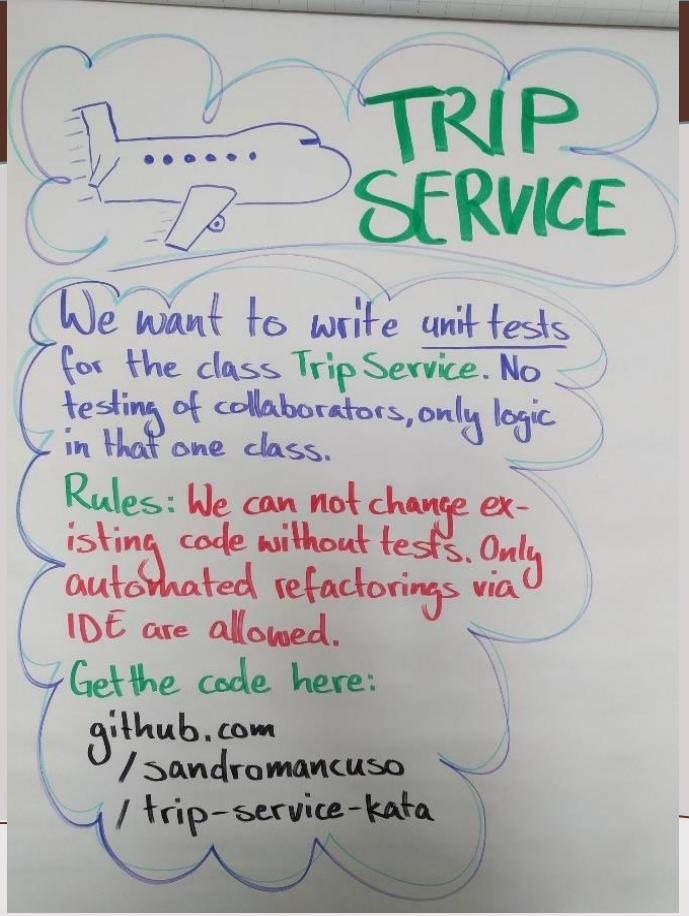
Mettre en place un harnais de tests

100% du code legacy actuel couvert

c'est-à-dire

100% de couverture de code par les tests

sur la méthode `getTripsByUser` de la classe `TripService`



Règles

Pas de refactoring sur un code de production non couvert par un test !

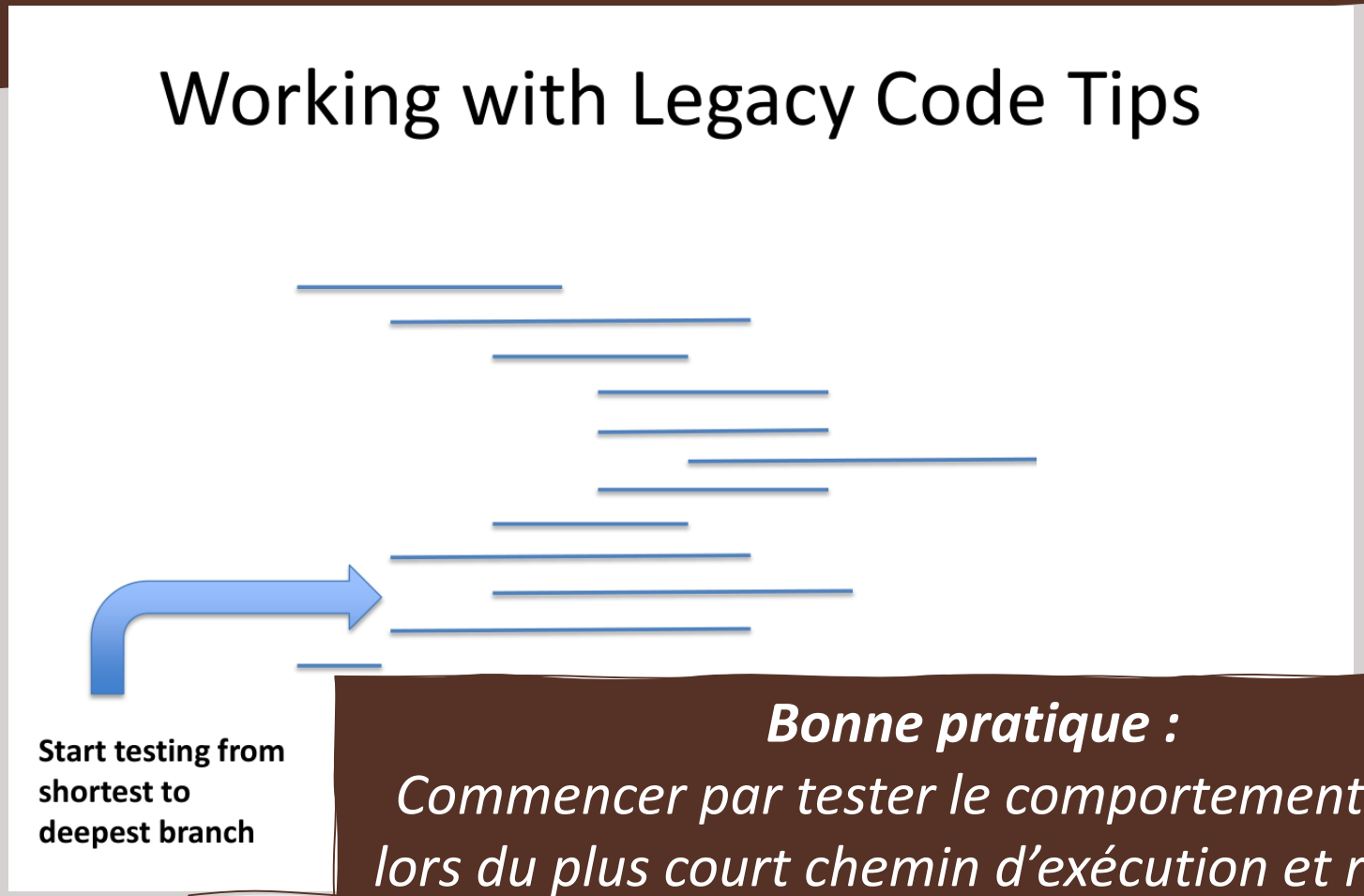
***Exception :* refactoring automatisés proposés par votre IDE**

possible sur le code de production pendant la phase de couverture par les tests

Image extraite de : https://kata-log.rocks/trip-service-kata - https://icon-icons.com/fr/icone/cible-objectif-la-reussite-le-but-le-tir-a-arc/55987

Par où commencer ? …

Petit conseil de Sandro pour écrire le premier test



Bonne pratique :

Commencer par tester le comportement obtenu lors du plus court chemin d'exécution et remo

⇒ branche de code la moins profonde

c-a-d branche la plus à gauche qui nécessite moins de préparation pour y accéder

Extrait slides de Sandro Mancuso disponible via de : https://kata-log.rocks/trip-service-kata

Par où commencer ? … Sur quelle partie du code focaliser le premier test ?

```
public class TripService {

    public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
        List<Trip> tripList = new ArrayList<Trip>();
        User loggedUser = UserSession.getInstance().getLoggedUser();
        boolean isFriend = false;
        if (loggedUser != null) {
            for (User friend : user.getFriends()) {
                if (friend.equals(loggedUser)) {
                    isFriend = true;
                    break;
                }
            }
            if (isFriend) {
                tripList = TripDAO.findTripsByUser(user);
            }
            return tripList;
        } else {
            throw new UserNotLoggedInException();
        }
    }
}
```

⇒ *branche de code la moins profonde*
chemin d'exécution le plus court

⇒ **Exception levée**
si l'utilisateur N'est PAS connecté

Test n°1

Comportement à couvrir :

Exception levée quand l'utilisateur n'est connecté

```java
public class TripServiceTest {


    @Test (expected = UserNotLoggedInException.class)
    public void should_throw_an_exception_when_user_is_not_loggedd_in(){
        TripService tripService = new TripService();
        tripService.getTripsByUser(null);
    }

}
```

null car peu importe la valeur du **User** passée en paramètre de cette méthode,
elle ne sera jamais utilisée dans le code qui permet de déterminer le comportement du test …

Pour ce test, seul importe que la valeur de **userLogged** dans **getTripService** soit bien à **null**
pour simuler un utilisateur non connecté 😉

Attention : **getTripsByUser** manipule bien deux utilisateurs distincts : **userLogged ≠ user**
(local) *(paramètre d'entrée)*

→ **Mais à l'exécution …**

Utilisateur « *curieux* »
connecté ou non

Utilisateur potentiellement un *ami*
passé en paramètre

Failure Trace

java.lang.Exception: Unexpected exception, expected<org.craftedsw.tripservicekata.exception.UserNotLoggedInException> but was<org.craftedsw.tripservicekata.exception.CollaboratorCallException>

Caused by: org.craftedsw.tripservicekata.exception.CollaboratorCallException: UserSession.getLoggedUser() should not be called in an unit test

at org.craftedsw.tripservicekata.user.UserSession.getLoggedUser(UserSession.java:17)

at org.craftedsw.tripservicekata.trip.TripService.getTripsByUser(TripService.java:14)

at org.craftedsw.tripservicekata.trip.TripServiceTest.should_show_an_exception_when_user_is_not_loggedd_in(TripServiceTest.java:11)

... 15 more

Une idée pour que l'exception **CollaboratorCallException** **ne soit pas levée** lors de l'exécution de ce test?

```
public class TripService {

    public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
        List<Trip> tripList = new ArrayList<Trip>();
        User loggedUser = UserSession.getInstance().getLoggedUser();
        boolean isFriend = false;
        if (loggedUser != null) {
            for (User friend : user.getFriends()) {
                if (friend.equals(loggedUser)) {
                    isFriend = true;
                    break;
                }
            }
            if (isFriend) {
                tripList = TripDAO.findTripsByUser(user);
            }
            return tripList;
        } else {
            throw new UserNotLoggedInException();
        }
    }
}
```

```
public class UserSession {

//…

public User getLoggedUser() {
    throw new
CollaboratorCallException("UserSession.
getLoggedUser() should not be called in
an unit test");
   }
}
```

On voudrait avoir `null` pour `loggedUser` mais une exception est levée
(métaphore d'accès à un service tiers sur lequel on n'a pas la main…)

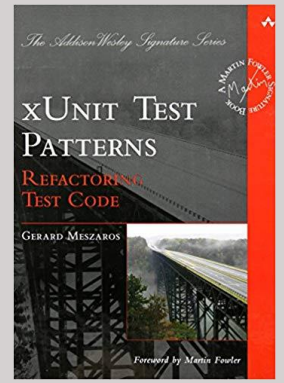
Rappel règle du kata : on ne doit pas toucher au code de production tant que ce dernier n'est pas couvert par mes tests

Des idées pour isoler la dépendance ?

☒ Mettre en place une doublure de tests (*mocker*)

→ **Stub** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

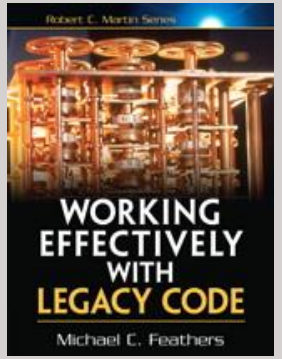
→ Doublure implémentée via mockito



Mais Mockito ne permet de mocker ni méthode statique, ni singleton ;-)

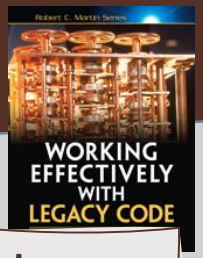
☑ Mettre en place une couture (Seam)

A **seam** is a place where you can alter behavior in your program without editing in that place.



Extrait : http://www.informit.com/articles/article.aspx?p=359417&seqNum=3

Zoom sur la couture (**Seam**)



A **seam** is a place where you can alter behavior in your program without editing in that place.
Every seam has an **enabling point**, a place where you can make the decision to use one behavior or another.

La couture, en ajoutant une fine couche (*thin layer*),
va permettre d'**isoler une dépendance** et de contrôler son comportement
pendant la phase de **couverture de tests** pour avoir le contrôle total du SUT

La mise en place d'un Seam va permettre au code legacy
de rester compatible pour la production tout en l'ouvrant pour les tests…
(c-a-d que le code de production ne sera ni modifié, ni *cassé* 😉)

Mais la couture n'est pas une fin en soi…
Sa présence n'a pour but que de vous permettre de tester votre code.
Pendant la phase de ***refactoring***, il faudra faire en sorte de ***casser cette dépendance***
(et donc **surement revisiter les tests** pour supprimer la couture à ce moment là)

Extrait : http://www.informit.com/articles/article.aspx?p=359417&seqNum=3 https://icon-icons.com/fr/icone/couture/16073
https://pixabay.com/fr/illustrations/thumbs-up-visage-souriant-emoji-4007573/

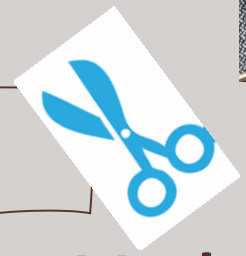
Isabelle BLASQUEZ

Mettre en place la couture (Seam)

pour isoler la dépendance au Singleton

Etapes pour mettre en place le Seam
Isoler la dépendance et forcer son comportement

1. Dans le code de production (`TripService`)



✓ **Isoler le code qui appelle la dépendance (**à l'aide de l'**Extract Method** de l'IDE)

⇒ L'extraire dans une méthode **`getLoggedInUser`** de type **`protected`**

2. Dans le code test, procéder à la couture avec l'aide d'une polymorphisme

✓ Créer une « *thin layer*» avec une nouvelle **classe privée `TestableTripService`** qui hérite de **`TripService`**

✓ Redéfinir la méthode **`getLoggedInUser`** dans **`TestableTripService`** et bouchonner son comportement (**`null`**)

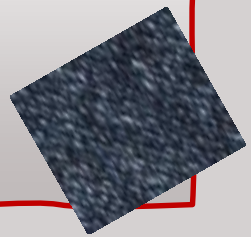
✓ Instancier une classe **`TestableTripService`** pour que le code passe désormais par la *thin layer*

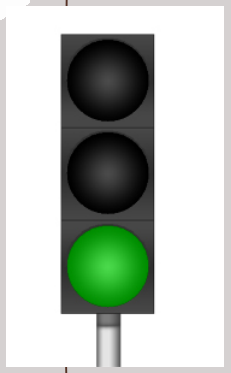
Isabelle BLASQUEZ

Isoler la dépendancance au Singleton
Et faire passer le test !

```java
public class TripService {

  public List<Trip> getTripsByUser(User user)
            throws UserNotLoggedInException {
    //…
    User loggedUser = getLoggedInUser();
    //…
    }

protected User getLoggedInUser() {
  return
    UserSession.getInstance().getLoggedUser();
  }
}
```

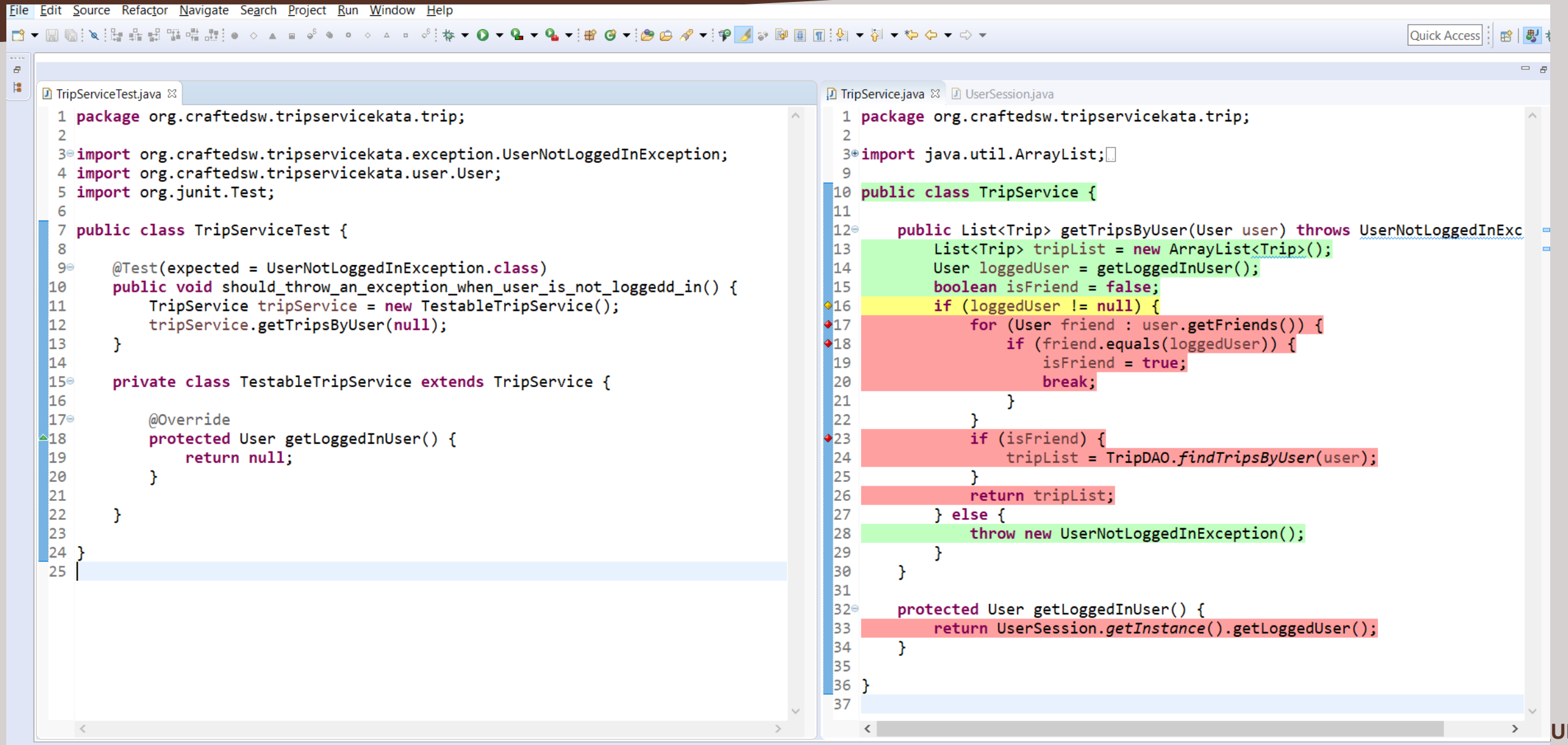
```java
public class TripServiceTest {

  @Test(expected = UserNotLoggedInException.class)
  public void should_throw_an_exception_when_user_is_not_loggedd_in(){
    TripService tripService = new TestableTripService();
    tripService.getTripsByUser(null);
  }

  private class TestableTripService extends TripService {
    @Override
    protected User getLoggedInUser() {
       return null;
    }
  }
}
```





Isabelle BLASQUEZ

Organisation de l'espace de travail et petit coup d'œil sur la couverture de code




```java
package org.craftedsw.tripservicekata.trip;

import org.craftedsw.tripservicekata.exception.UserNotLoggedInException;
import org.craftedsw.tripservicekata.user.User;
import org.junit.Test;

public class TripServiceTest {

    @Test(expected = UserNotLoggedInException.class)
    public void should_throw_an_exception_when_user_is_not_loggedd_in() {
        TripService tripService = new TestableTripService();
        tripService.getTripsByUser(null);
    }

    private class TestableTripService extends TripService {

        @Override
        protected User getLoggedInUser() {
            return null;
        }

    }

}
```

```java
package org.craftedsw.tripservicekata.trip;

import java.util.ArrayList;

public class TripService {

    public List<Trip> getTripsByUser(User user) throws UserNotLoggedInExc
        List<Trip> tripList = new ArrayList<Trip>();
        User loggedUser = getLoggedInUser();
        boolean isFriend = false;
        if (loggedUser != null) {
            for (User friend : user.getFriends()) {
                if (friend.equals(loggedUser)) {
                    isFriend = true;
                    break;
                }
            }
            if (isFriend) {
                tripList = TripDAO.findTripsByUser(user);
            }
            return tripList;
        } else {
            throw new UserNotLoggedInException();
        }
    }

    protected User getLoggedInUser() {
        return UserSession.getInstance().getLoggedUser();
    }

}
```

⇒ Un peu de refactoring sur le code de test est permis et même fortement conseillé

Sentez-vous une mauvaise odeur dans ce code ?

⇒ **Améliorer la lisibilité des tests**

Faire apparaître
des **termes métiers** relatif au
concept d'utilisateur :
pour **rendre explicite le rôle**
que joue chaque utilisateur
dans le comportement
du programme

```
public class TripServiceTest {

    @Test(expected = UserNotLoggedInException.class)
    public void should_throw_an_exception_when_user_is_not_loggedd_in(){
        TripService tripService = new TestableTripService();
        tripService.getTripsByUser(null);
    }


    private class TestableTripService extends TripService {
        @Override
        protected User getLoggedInUser() {
            return null;
        }
    }
}
```

Rappel :
2 utilisateurs peuvent influencer le comportement du programme

Utilisateur « *curieux* »
connecté ou non

utilisateur potentiellement un *ami*
de l'utilisateur curieux

Refactoring : Améliorer la lisibilité des tests
Zoom sur **null** de **getLoggedInUser()**

```
public class TripServiceTest {



    @Test(expected = UserNotLoggedInException.class)
    public void should_throw_an_exception_when_user_i

        TripService tripService = new TestableTripService();
        tripService.getTripsByUser(null);
    }


    private class TestableTripService extends TripSe
        @Override
        protected User getLoggedInUser() {
            return null;
        }
    }
}
```

Dans ce test, l'utilisateur **n'est pas connecté**.

Cet état de l'utilisateur a un impact sur le comportement du programme
donc sur le *verdict du test* (le ***Assert*** du test).

Cet état devrait donc être explicitement spécifier dans le test, et plus précisément au moment où on définit le contexte du test
c-a-d dans la partie ***Arrange***

Rappel Bonne Pratique :
Un test bien écrit respecte le pattern AAA !

Se rapporte à

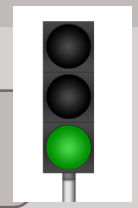
Utilisateur « *curieux* »
non connecté
(pour ce test)

Etapes de refactoring pour faire apparaître Explicitement que l'utilisateur n'est pas connecté


```
public class TripServiceTest {
    private User loggedInUser;

    @Test(expected = UserNotLoggedInException.class)
    public void should_throw_an_exception_when_user_is_not_loggedd_in(){
        loggedInUser = null;
        TripService tripService = new TestableTripService();
        tripService.getTripsByUser(null);
    }


    private class TestableTripService extends TripService
        @Override
        protected User getLoggedInUser() {
            return loggedInUser
        }
    }
```

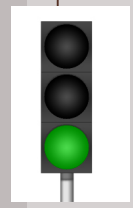
1.b Déclarer **`loggedInUser`** pour qu'il soit accessible à ces deux méthodes

Utilisateur « *curieux* »
non connecté



1.a Dans l'étape ***Arrange***,
spécifier explicitement
que l'utilisateur n'est pas connecté
`loggedInUser= null`
(en s'inspirant de la terminologie métier existante utilisée par la méthode d'où vient le **`null`**)

2. renvoyer un **`loggedInUser`**
(c-a-d interroger cette méthode pour savoir
si l'utilisateur est connecté (un vrai objet **`User`**) ou pas (**`null`**)



Isabelle BLASQUEZ

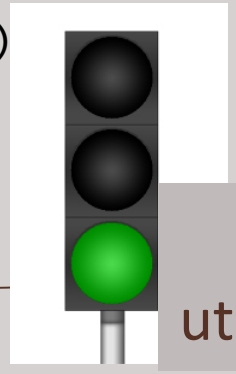
Reste plus qu'à expliciter le null en « s'imprégnant » du langage du métier …


```
public class TripServiceTest {
   private static final User GUEST = null;
   private User loggedInUser;

   @Test(expected = UserNotLoggedInException.class)
   public void should_throw_an_exception_when_user_is_not_loggedd_in(){
      loggedInUser = GUEST;
      TripService tripService = new TestableTripService();
      tripService.getTripsByUser(null);
   }


   private class TestableTripService extends TripService {
      @Override
      protected User getLoggedInUser() {
          return loggedInUser;
      }
    }
}
```

null reste mystérieux : un *Magic Number ?*

Remplaçons le par une constante,
mais comment la nommer ?

Dans une appli web, un utilisateur qui navigue sans être connecté est habituellement appelé : ***invité***



Qu'à cela ne tienne,
utilisons **GUEST** dans le code

Bonne Pratique : Terme métier repris dans le code

Isabelle BLASQUEZ

Refactoring : Améliorer la lisibilité des tests
Zoom sur **`null`** de **`getTripsByUser()`**

```
public class TripServiceTest {

  private static final User GUEST = null;
  private User loggedInUser;

  @Test(expected = UserNotLoggedInException.class)
  public void should_throw_an_exception_when_user_is_not_loggedd_in() {
     TripService tripService = new TestableTripService();
     loggedInUser = GUEST;

      tripService.getTripsByUser(null);
   }

  private class TestableTripService extends TripService {

    @Override
    protected User getLoggedInUser() {
      return loggedInUser;
    }
  }
}
```

Dans ce test, l'utilisateur « *curieux* » **n'est pas connecté**.

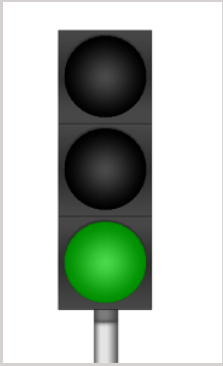
Peu importe **qui** est l'utilisateur *ami*
car dans ce test, il ne sera jamais consulté
et donc n'impactera en aucun cas le *verdict* du test

C'est pour cela qu'un **`null`** suffisait :
pas la peine de perdre son temps à créer un
utilisateur non utilisé

utilisateur potentiellement un *ami*
de l'utilisateur curieux

Reste plus qu'à expliciter le null en montrant l'intention métier ...


```
public class TripServiceTest {
  private static final User UNUSED_USER = null;
  private static final User GUEST = null;
  private User loggedInUser;

  @Test(expected = UserNotLoggedInException.class)
  public void should_throw_an_exception_when_user_is_not_loggedd_in() {
     TripService tripService = new TestableTripService();
     loggedInUser = GUEST;

     tripService.getTripsByUser (UNUSED_USER);
   }

  private class TestableTripService extends TripService {

    @Override
    protected User getLoggedInUser() {
      return loggedInUser;
    }
  }
}
```



Qu'à cela ne tienne,
utilisons **UNUSED_USER** dans le code !

null suffit :
pas la peine de perdre son temps à créer un
utilisateur non utilisé

Petit coup d'œil sur la couverture de code

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access

TripServiceTest.java

```
1  package org.craftedsw.tripservicekata.trip;
2
3  import org.craftedsw.tripservicekata.exception.UserNotLoggedInException;
4  import org.craftedsw.tripservicekata.user.User;
5
6  import org.junit.Test;
7
8  public class TripServiceTest {
9
10     private static final User GUEST = null;
11     private static final User UNUSED_USER = null;
12
13     private User loggedInUser;
14
15     @Test(expected = UserNotLoggedInException.class)
16     public void should_throw_an_exception_when_user_is_not_loggedd_in() {
17         TripService tripService = new TestableTripService();
18
19         loggedInUser = GUEST;
20
21         tripService.getTripsByUser(UNUSED_USER);
22     }
23
24     private class TestableTripService extends TripService {
25         @Override
26         protected User getLoggedInUser() {
27             return loggedInUser;
28         }
29
30     }
31 }
32
```

TripService.java

```
1  package org.craftedsw.tripservicekata.trip;
2
3  import java.util.ArrayList;
9
10 public class TripService {
11
12     public List<Trip> getTripsByUser(User user) throws UserNotLoggedInExcepti
13         List<Trip> tripList = new ArrayList<Trip>();
14         User loggedUser = getLoggedInUser();
15         boolean isFriend = false;
16         if (loggedUser != null) {
17             for (User friend : user.getFriends()) {
18                 if (friend.equals(loggedUser)) {
19                     isFriend = true;
20                     break;
21                 }
22             }
23             if (isFriend) {
24                 tripList = TripDAO.findTripsByUser(user);
25             }
26             return tripList;
27         } else {
28             throw new UserNotLoggedInException();
29         }
30     }
31
32     protected User getLoggedInUser() {
33         return UserSession.getInstance().getLoggedUser();
34     }
35
36 }
37
```

Par où continuer ? … Sur quelle partie du code focaliser le deuxième test ?

```
public class TripService {

    public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
        List<Trip> tripList = new ArrayList<Trip>();
        User loggedUser = UserSession.getInstance().getLoggedUser();
        boolean isFriend = false;
        if (loggedUser != null) {
            for (User friend : user.getFriends()) {
                if (friend.equals(loggedUser)) {
                    isFriend = true;
                    break;
                }
            }
            if (isFriend) {
                tripList = TripDAO.f
            }
            return tripList;
        } else {
            throw new UserNotLoggedInException();
        }
    }
}
```

2ème branche la moins profonde (2ème chemin d'execution le plus court)

Quand l'utilisateur curieux et l'utilisateur ami NE sont PAS amis

⇒ AUCUN voyage

⇒ 1er test : DONE !!!

Test n°2

Comportement à couvrir :

Aucun voyage quand les utilisateurs NE sont PAS amis

```
public class TripServiceTest {

    private static final User GUEST = null;
    private static final User UNUSED_USER = null;
    private static final User LOGGED_IN_USER = new User();
    private static final User ANOTHER_USER = new User();
    private static final Trip TO_ITALY = new Trip();

    private User loggedInUser;

    @Test(expected = UserNotLoggedInException.class)
    public void should_throw_an_exception_when_user_is_not_l

    @Test
    public void should_not_return_any_trips_when_users_are_not_friends() {
        TripService tripService = new TestableTripService();
        loggedInUser = LOGGED_IN_USER;
        User friend = new User();
        friend.addFriend(ANOTHER_USER);
        friend.addTrip(TO_ITALY);
        List<Trip> friendTrips=tripService.getTripsByUser(friend);
        assertThat(friendTrips).hasSize(0);
        }

        private class TestableTripService extends TripService { //… }
}
```

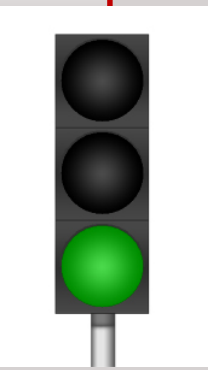
*Cette fois-ci, l'**utilisateur** curieux* **est connecté** (*LOGGED_IN_USER*);

Yagni! (Simple) : contexte minimum
1 seul ami et 1 seul voyage
suffisent pour écrire ce test

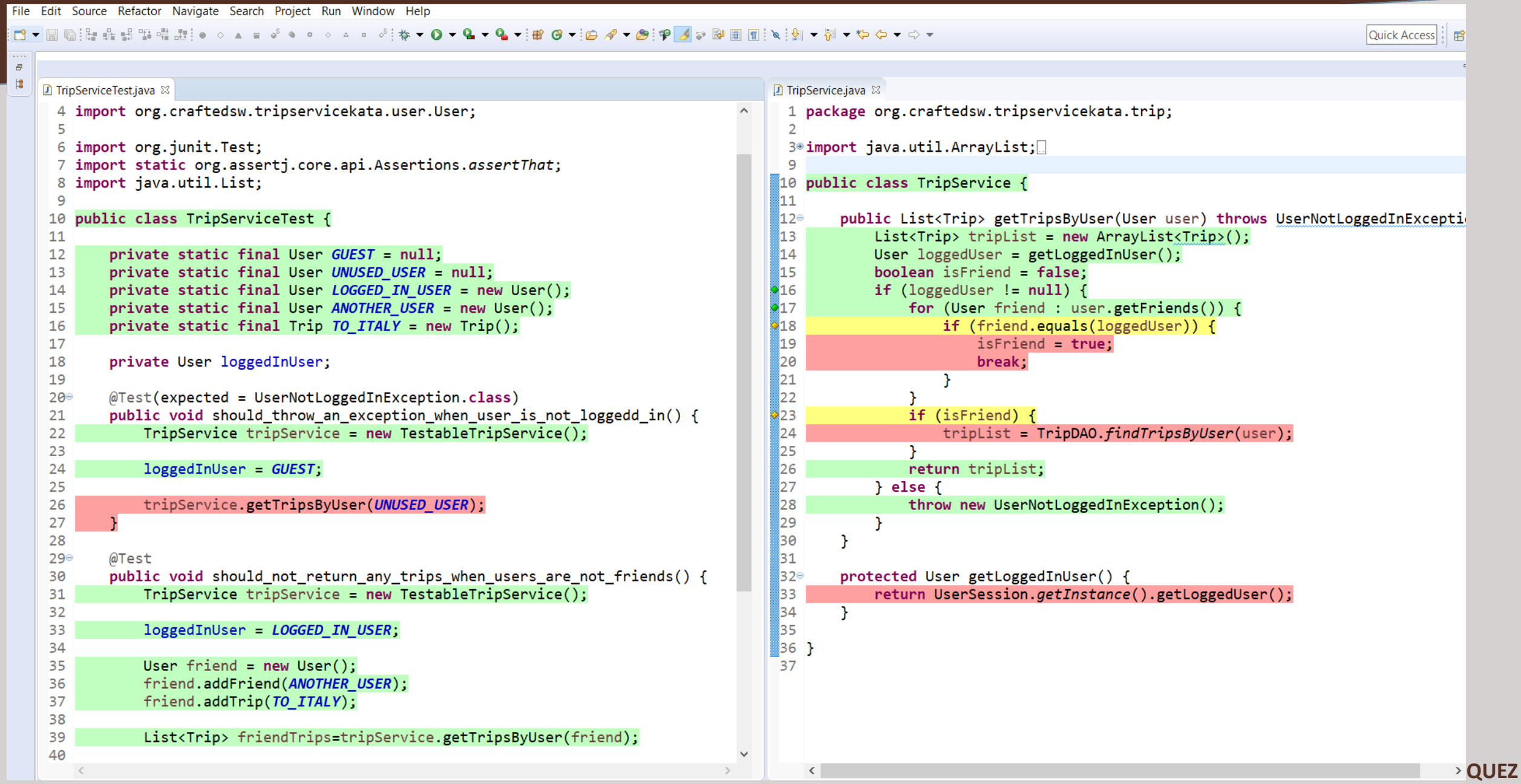
*Nécessaire de **créer un** **vrai utilisateur** **ami***
avec une "histoire" (d'ami(s) et de voyage(s))

pas ami avec l'utilisateur connecté,
*mais ami avec un **autre utilisateur*** (*ANOTHER_USER*)

***Assertion :* aucun voyage !**



Petit coup d'œil sur la couverture de code



Par où continuer ? … Sur quelle partie du code focaliser le deuxième test ?

```
public class TripService {

    public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
        List<Trip> tripList = new ArrayList<Trip>();
        User loggedUser = UserSession.getInstance().getLoggedUser();
        boolean isFriend = false;
        if (loggedUser != null) {
            for (User friend : user.getFriends()) {
                if (friend.equals(loggedUser)) {
                    isFriend = true;
                    break;
                }
            }
            if (isFriend) {
                tripList = TripDAO.findTripsByUser(user);
            }
            return tripList;
        } else {
            throw new UserNotLoggedInException();
        }
    }
}
```

⇒ *2ème test : DONE !!!*

⇒ *1er test : DONE !!!*

branche la moins profonde restante à couvrir

Liste de voyages Quand l'utilisateur curieux et l'utilisateur ami SONT amis

Test n°3

Comportement à couvrir :

Aucun voyage quand les utilisateurs NE sont PAS amis

En s'inspirant des tests précédents …

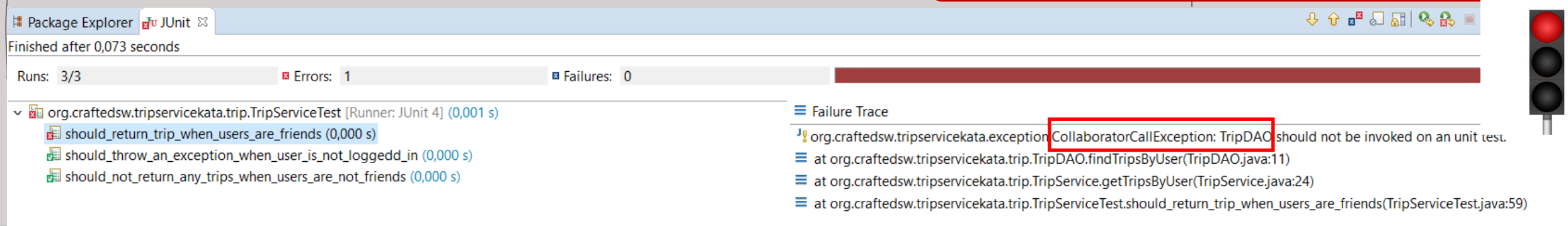
```
@Test
public void should_return_trip_when_users_are_friends() {
  TripService tripService = new TestableTripService();

  loggedInUser = LOGGED_IN_USER;

  User friend = new User();
  friend.addFriend(ANOTHER_USER);
  friend.addFriend(loggedInUser);
  friend.addTrip(TO_ITALY);
  friend.addTrip(TO_SPAIN);

  List<Trip> friendTrips = tripService.getTripsByUser(friend);

  assertThat(friendTrips).hasSize(2);
  assertThat(friendTrips).contains(TO_ITALY, TO_SPAIN);
```

*l'utilisateur curieux **est connecté***

*L'ami **est ami** avec l'utilisateur connecté*

*Une liste de **2 voyages** est récupérée*

Qui contient bien les voyages de l'ami

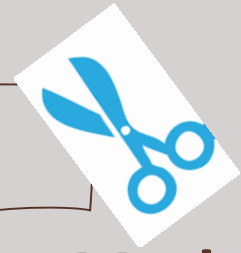


Mettre en place la couture (Seam)

pour isoler la dépendance à la couche d'accès aux données (`TripDAO`)

Etapes pour mettre en place le Seam et Isoler la dépendance à la couche d'accès aux données

1. Dans le code de production (`TripService`)



✓ **Isoler le code qui appelle la dépendance (**à l'aide de l'**Extract Method** de l'IDE)

⇒ dans une méthode **`tripsBy`** de type **`protected`**

2. Dans le code test, procéder à la couture avec l'aide d'une polymorphisme

✓ La « *thin layer*» **`TestableTripService`** existe déjà …

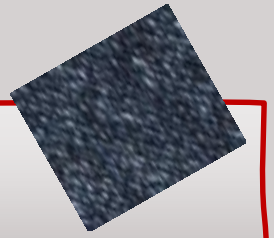
✓ Redéfinir la méthode **`tripsBy`**
et bouchonner son comportement (utiliser le **`trips`** de **`User`**)

✓ puisque c'est bien la « *thin layer*»
qui est déjà instanciée dans l'étape Arrange du test.

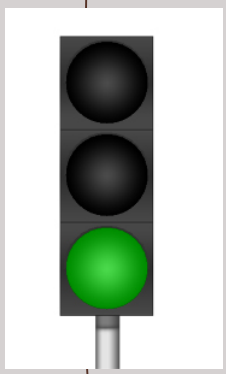
Isabelle BLASQUEZ

Isoler la dépendancance au Singleton
Et faire passer le test !

```
public class TripServiceTest {

  //…

  public void should_return_trip_when_users_are_friends(){
    // idem précédemment,
    // TestableTripService étant déjà instancié
  }

  private class TestableTripService extends TripService {
    @Override
    protected User getLoggedInUser() {
       return null;
    }

    @Override
    protected List<Trip> tripsBy(User user) {
       return user.trips();
    }

  }
}
```



```
public class TripService {

  public List<Trip> getTripsByUser(User user)
              throws UserNotLoggedInException {
      //…
      tripList = tripsBy(user);
      //…
  }

  protected List<Trip> tripsBy(User user) {
     return TripDAO.findTripsByUser(user);
  }
}
```



Isabelle BLASQUEZ

Petit coup d'œil sur la couverture de code

File Edit Source Refactor Navigate Search Project Run Window Help

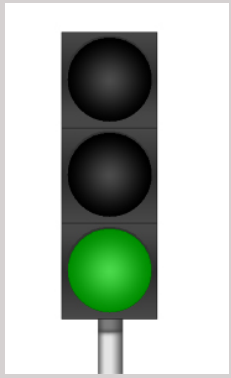
Quick Access

TripServiceTest.java

```
44
45      @Test
46      public void should_return_trip_when_users_are_friends() {
47          TripService tripService = new TestableTripService();
48
49          loggedInUser = LOGGED_IN_USER;
50
51          User friend = new User();
52          friend.addFriend(ANOTHER_USER);
53          friend.addFriend(loggedInUser);
54          friend.addTrip(TO_ITALY);
55          friend.addTrip(TO_SPAIN);
56
57          List<Trip> friendTrips = tripService.getTripsByUser(friend);
58
59          assertThat(friendTrips).hasSize(2);
60          assertThat(friendTrips).contains(TO_ITALY, TO_SPAIN);
61      }
62
63      private class TestableTripService extends TripService {
64          @Override
65          protected User getLoggedInUser() {
66              return loggedInUser;
67          }
68
69          @Override
70          protected List<Trip> tripsBy(User user) {
71              return user.trips();
72          }
73
74      }
75  }
```

TripService.java

```
3   import java.util.ArrayList;
9
10  public class TripService {
11
12      public List<Trip> getTripsByUser(User user) throws UserNotLoggedInExceptio
13          List<Trip> tripList = new ArrayList<Trip>();
14          User loggedUser = getLoggedInUser();
15          boolean isFriend = false;
16          if (loggedUser != null) {
17              for (User friend : user.getFriends()) {
18                  if (friend.equals(loggedUser)) {
19                      isFriend = true;
20                      break;
21                  }
22              }
23              if (isFriend) {
24                  tripList = tripsBy(user);
25              }
26              return tripList;
27          } else {
28              throw new UserNotLoggedInException();
29          }
30      }
31
32      protected User getLoggedInUser() {
33          return UserSession.getInstance().getLoggedUser();
34      }
35
36      protected List<Trip> tripsBy(User user) {
37          return TripDAO.findTripsByUser(user);
38      }
39  }
```



Legacy couvert à 100%

Refactoring du code de test

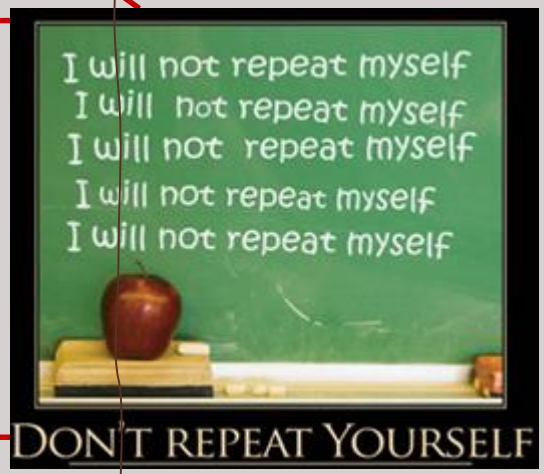
fortement conseillé

avant de passer

au refactoring du code de production

```java
public class TripServiceTest {
  //…

  @Test(expected = UserNotLoggedInException.class)
  public void should_throw_an_exception_when_user_is_not_loggedd_in() {
    TripService tripService = new TestableTripService();
    loggedInUser = GUEST;
    //…
  }

  @Test
  public void should_not_return_any_trips_when_users_are_not_friends() {
    TripService tripService = new TestableTripService();
    loggedInUser = LOGGED_IN_USER;
    User friend = new User();
    friend.addFriend(ANOTHER_USER);
    friend.addTrip(TO_ITALY);
    //…
  }

  @Test
  public void should_return_trip_when_users_are_friends() {
    TripService tripService = new TestableTripService();
    loggedInUser = LOGGED_IN_USER;
    User friend = new User();
    friend.addFriend(ANOTHER_USER);
    friend.addFriend(loggedInUser);
    friend.addTrip(TO_ITALY);
    friend.addTrip(TO_SPAIN);
    //…
  }

  private class TestableTripService extends TripService {//…}

}
```

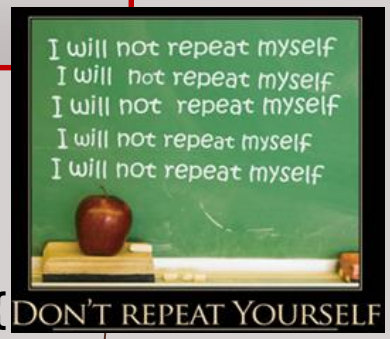
Ne sentez-vous pas comme une mauvaise odeur dans les étapes *Arrange* des tests ?



Bonne pratique :
A partir de 3 fois, on a de la duplication, il est temps d'agir …

```
public class TripServiceTest {
  //…

  @Before
  public void setUp() {
    tripService = new TestableTripService();
    loggedInUser = LOGGED_IN_USER;
  }

  @Test(expected = UserNotLoggedInException.class)
  public void should_throw_an_exception_when_user_is_not_loggedd_in() {
    loggedInUser = GUEST;
    //…
  }

  @Test
  public void should_not_return_any_trips_when_users_are_not_friends() {
    User friend = new User();
    friend.addFriend(ANOTHER_USER);
    friend.addTrip(TO_ITALY);
    //…
  }

  @Test
  public void should_return_trip_when_users_are_friends() {
    User friend = new User();
    friend.addFriend(ANOTHER_USER);
    friend.addFriend(loggedInUser);
    friend.addTrip(TO_ITALY);
    friend.addTrip(TO_SPAIN);
    //
  }

  p                                    rvice {//…}
}
```

Factorisation dans un **setUp**
et choix sur l'état de connection le plus frequent…



Encore un dernier petit effort ?


```
User friend = UserBuilder.aUser()
                         .friendsWith(ANOTHER_USER, loggedInUser)
                         .withTrips(TO_ITALY, TO_SPAIN)
                         .build();
```

Améliorer la lisibilité :
plus simple et plus rapide à comprendre

Implémentation du `UserBuilder`

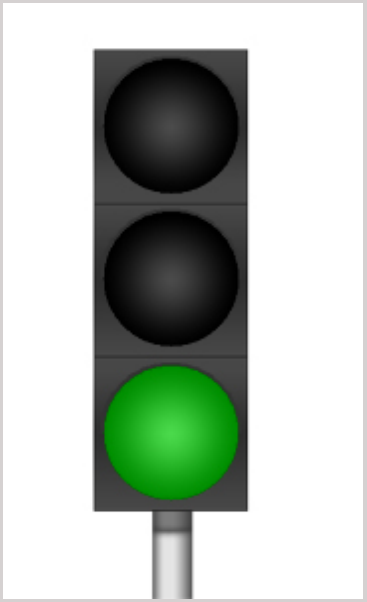
```
public class UserBuilder {
  private User[] friends;
  private Trip[] trips;

  public static UserBuilder aUser() {
     return new UserBuilder();
  }

   public UserBuilder friendsWith(User...friends) {
      this.friends = friends;
      return this;
   }

   public UserBuilder withTrips(Trip...trips) {
      this.trips = trips;
      return this;
   }

   public User build() {
      User user = new User();
      addFriendsTo(user);
      addTripsTo(user);
      return user;
   }
// suite ci-contre …
```

Bonne Pratique :
Améliorer la lisibilité de vos tests avec un builder
(⇒ construction de vos objets plus simple et plus sûre)

```
   private void addFriendsTo(User user) {
      for (User friend : friends) {
         user.addFriend(friend);
      }
   }

   private void addTripsTo(User user) {
      for (Trip trip : trips) {
          user.addTrip(trip);
       }
   }
}
```

Etapes *Arrange* du code de test refactorées …

```
public class TripServiceTest {
  //…

  @Before
  public void setUp() {
    tripService = new TestableTripService();
    loggedInUser = LOGGED_IN_USER;
  }

  @Test(expected = UserNotLoggedInException.class)
  public void should_throw_an_exception_when_user_is_not_loggedd_in() {
    loggedInUser = GUEST;
    //…
  }

  @Test
  public void should_not_return_any_trips_when_users_are_not_friends() {
    User friend = aUser()
                  .friendsWith(ANOTHER_USER)
                  .withTrips(TO_ITALY)
                  .build();
    //…
  }

  @Test
  public void should_return_trip_when_users_are_friends() {
    User friend = aUser()
                  .friendsWith(ANOTHER_USER, loggedInUser)
                  .withTrips(TO_ITALY, TO_SPAIN)
                  .build();
    //…
  }
 private class TestableTripService extends TripService {//…}
}
```



Seconde Partie

Refactorer !!!

Isabelle BLASQUEZ

Que cherche-t-on en refactorant ?

… Façonner un code plus propre en essayant de …

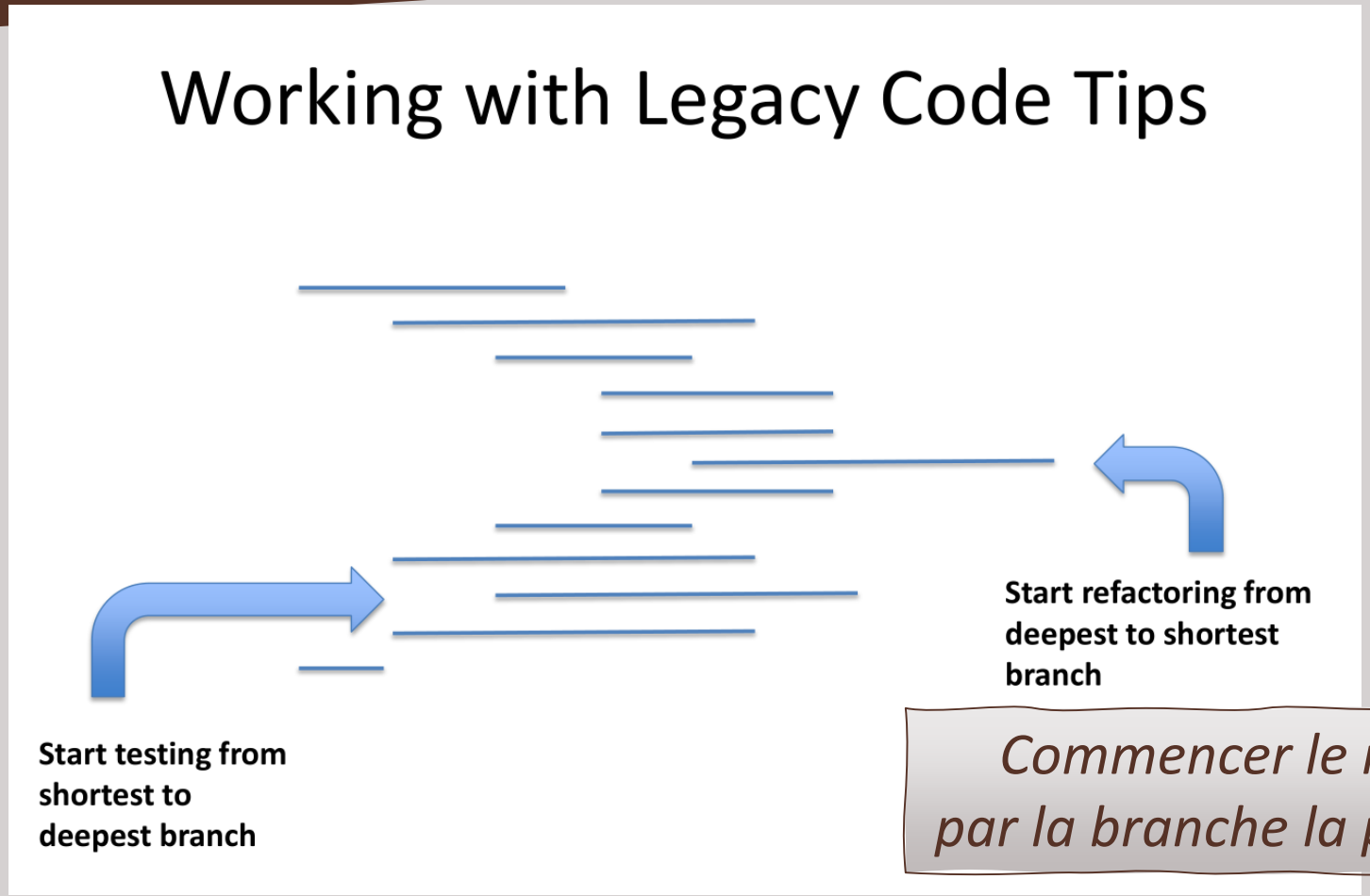
Supprimer un(des) **code smell**(s)

Rendre le code plus SOLID et rompre les dépendances

Améliorer la lisibilité du test
en portant une attention particulière au **nommage**
(pour **rendre explicite l'intention métier**)

Par où commencer ? …

Petit conseil de Sandro pour se lancer dans la phase de refactoring



Extrait slides de Sandro Mancuso disponible via de : https://kata-log.rocks/trip-service-kata

Par où commencer ? … Sur quelle partie du code démarrer le refactoring?

```
public class TripService {

    public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
        List<Trip> tripList = new ArrayList<Trip>();
        User loggedUser = getLoggedInUser();
        boolean isFriend = false;
        if (loggedUser != null) {
            for (User friend : user.getFriends()) {
                if (friend.equals(loggedUser)) {
                    isFriend = true;
                    break;
                }
            }
            if (isFriend) {
                tripList = tripsBy(user);
            }
            return tripList;
        } else {
            throw new UserNotLoggedInException();
        }
    }
  //…
}
```

2ème branche la plus profonde
⇒ Le refactoring pourrait commencer avec ce bout de code …

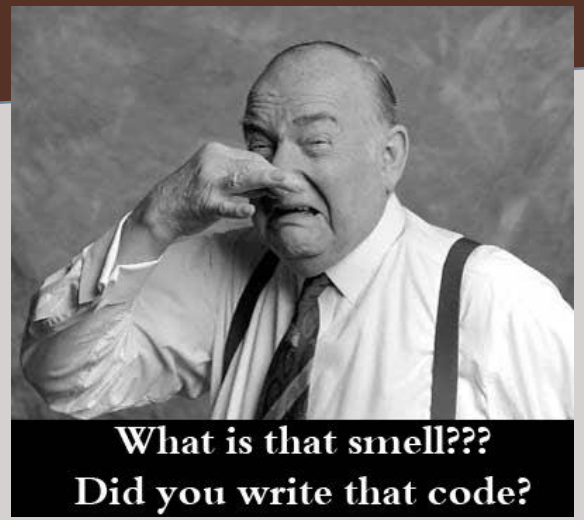
1ère branche la plus profonde mais rien à refactorer….

Revue de code

à la recherche des mauvaises odeurs

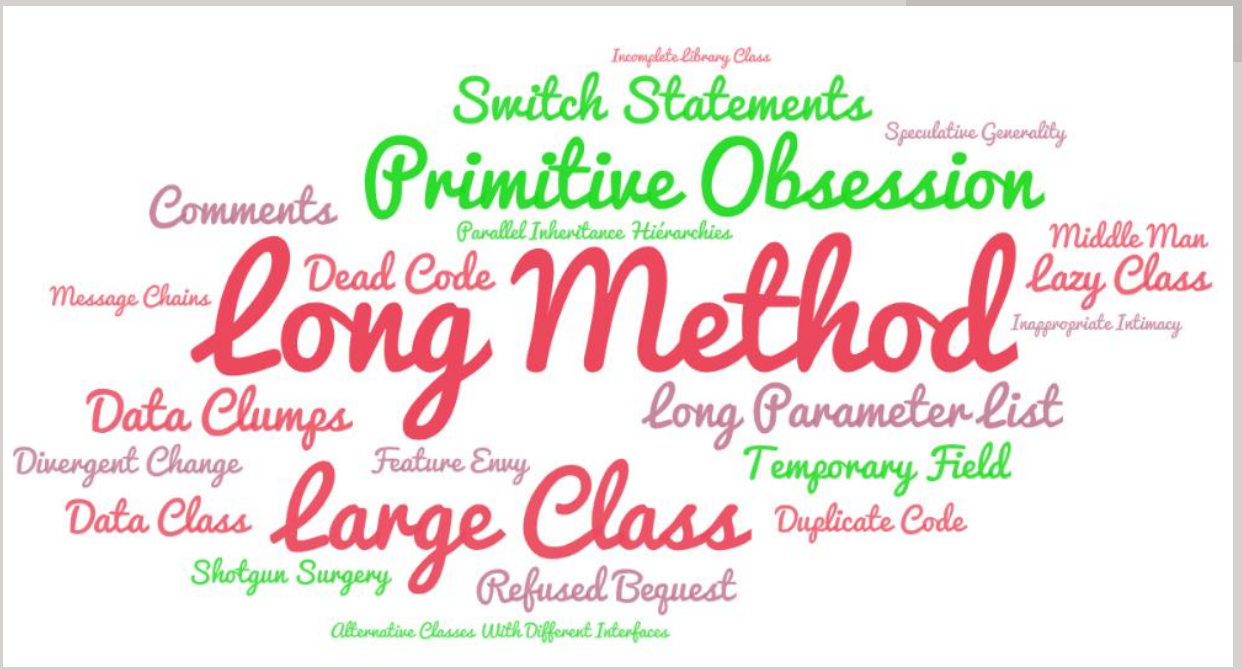
(Code smell)

Qu’est-ce qui sent mauvais dans ce code ?




```
public class TripService {

   public List<Trip> getTripsByUser(User user)
                                        throws UserNotLoggedInException {
      //…
         for (User friend : user.getFriends()) {
            if (friend.equals(loggedUser)) {
               isFriend = true;
               break;
            }
      //…
```

Quid des responsabilités ?

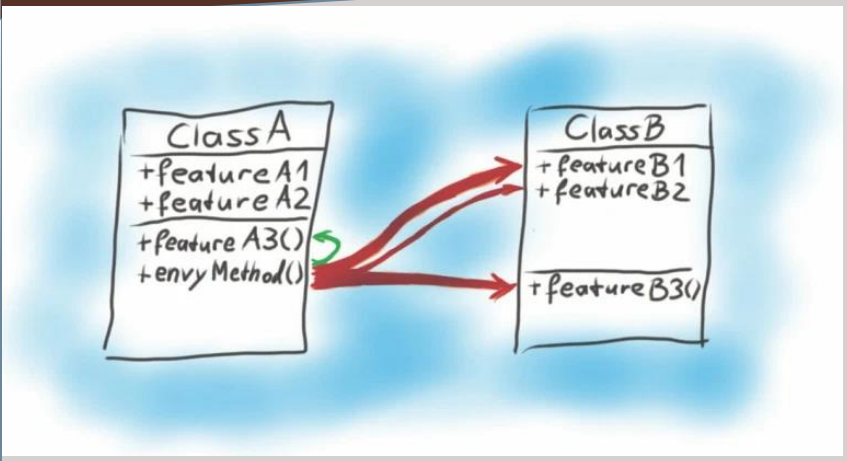


Avez-vous reconnu le code smell correspondant à cette mauvaise odeur ?

En savoir plus sur les code smells :
https://sourcemaking.com/refactoring/smells

Nuage de mots généré via https://wordart.com

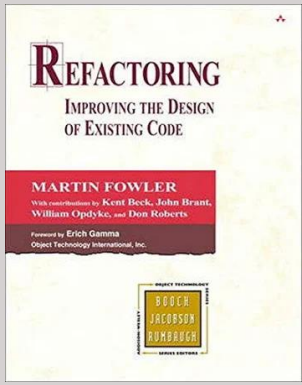
Détecter un code Smell **Feature Envy**



Methods suffer from Feature Envy,
if they use other classes more than their own.

Martin Fowler, the inventor of Code Smells and Feature Envy, puts it like this:

The whole point of objects is that they are a technique to package data with the processes used on that data. ***A classic smell is a method that seems more interested in a class other than the one it is in. The most common focus of the envy is the data.***



We 've lost count of the times we've seen a method ***invokes half-dozen getting*** *methods on another object to calculate some value*

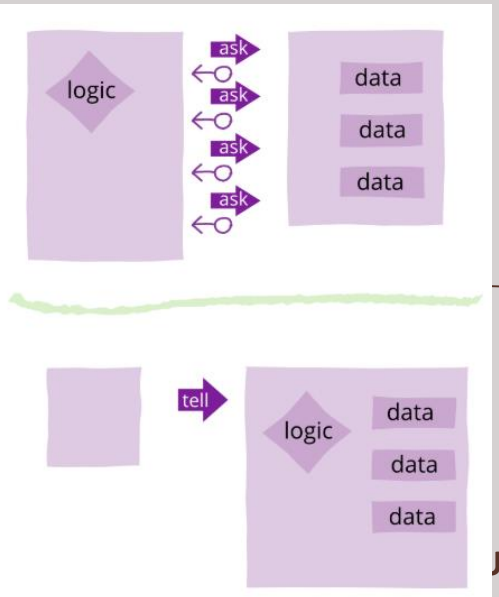
Images extraites : https://waog.wordpress.com/2014/08/25/code-smell-feature-envy
Textes extraits : https://martinfowler.com/books/refactoring.html (1st edition)

Isabelle BLASQUEZ

Le **Feature Envy** de notre legacy

```
public class TripService {

   public List<Trip> getTripsByUser(User user)
                        throws UserNotLoggedInException
{
      //…
         for (User friend : user.getFriends()) {
            if (friend.equals(loggedUser)) {
               isFriend = true;
               break;
            }
      //…
}
```

```
public class User {

//…

public List<User> getFriends() {
return friends;
}

//…

}
```



La solution :

Tell don't ask !

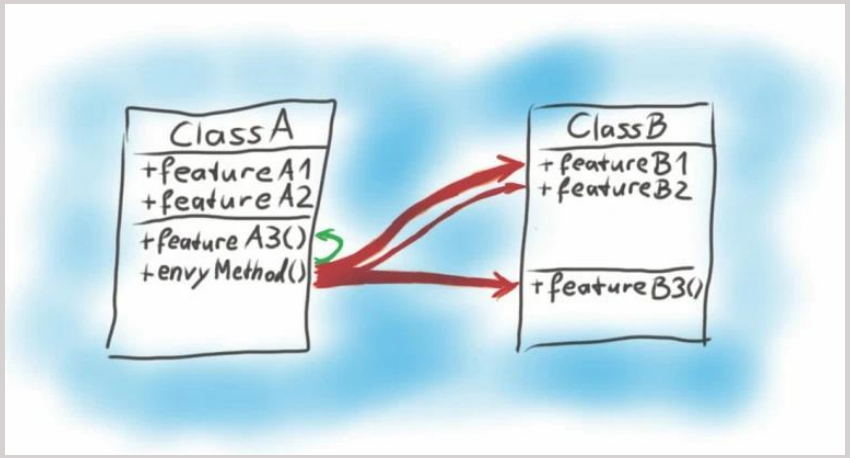
En savoir plus sur le Tell don't ask : https://martinfowler.com/bliki/TellDontAsk.html

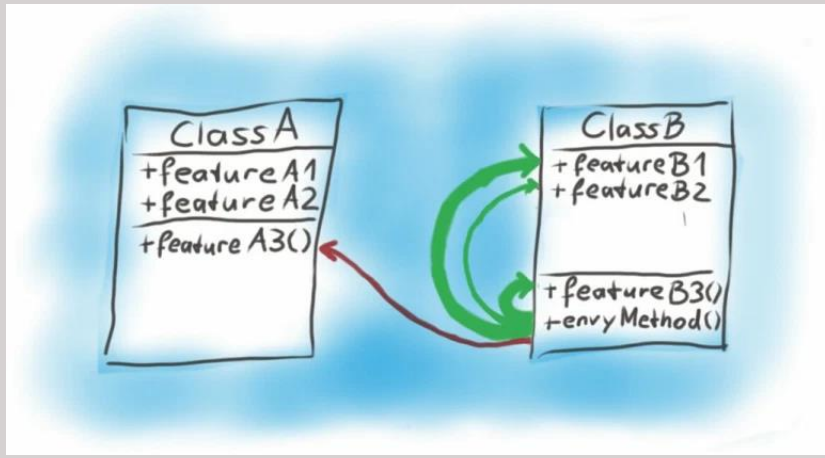
Refactorer un code smell Feature Envy

Tell don't ask !

Give the method what it desires – the other class!

*The method clearly wants to be elsewhere, so you use **Move Method** to get it there.*





After moving the envy method to the desired class,
the green arrows total thickness exceed the others.
We have ***no more Feature Envy***.

We **reduced the coupling** (red arrows) between our classes
and **raised the cohesion** (green arrows) inside our classes.

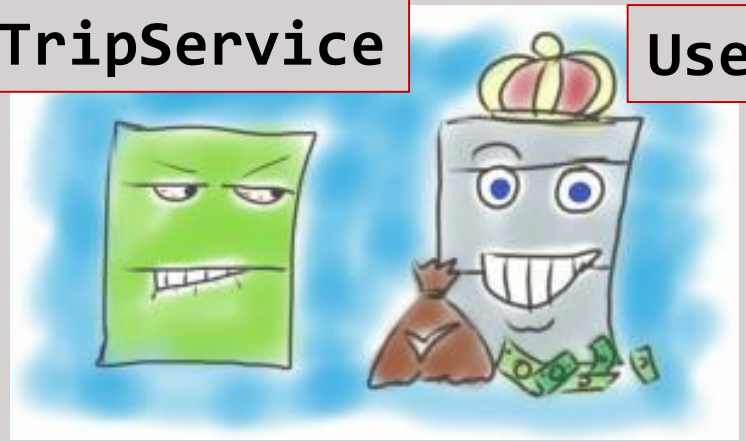
Extrait : https://waog.wordpress.com/2014/08/25/code-smell-feature-envy

Isabelle BLASQUEZ

Refactorer notre Feature Envy : comment faire ?

Tell don't ask !

`TripService` `User`



Ne vous contentez pas d'extraire le code **`TripService`** *et de déplacer ce code dans* **`User`** *…*

… mais pour avoir une implémentation aussi simple que possible et bien adapter au besoin…

Implémenter plutôt en TDD *une nouvelle méthode dans* ***`User`***
qui rendra le ***service attendu par*** **`TripService`**

Etapes pour refactorer notre Feature Envy

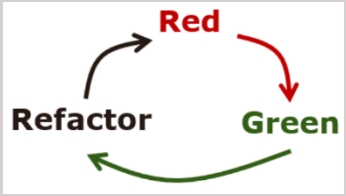
Tell don't ask !

1. **Bien nommer la nouvelle méthode** qui doit refléter l'intention métier

D'un point de vue métier, il semblerait qu'on cherche à savoir
si un utilisateur est ami avec un autre utilisateur

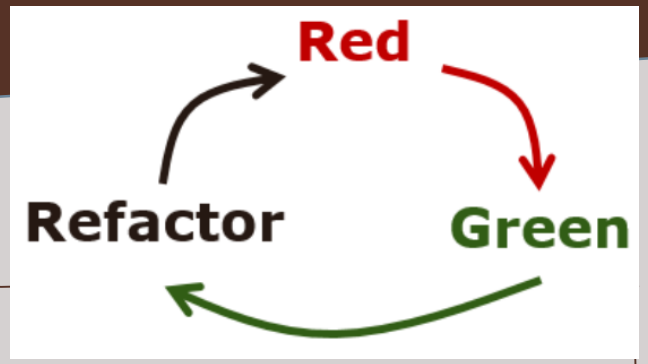
⇒ Une nouvelle méthode **`isFriendWith(anotherUser)`** dans la classe **`User`**

2. **Développer en TDD**
la nouvelle méthode **`isFriendWith`** de la classe **`User`**



3. **Appeler** la méthode **`isFriendWith`** **depuis** **`TripService`**

Développer en TDD de `isFriendWith` de la classe `User`




```java
public class UserTest {

    private static final User ALICE = new User();
    private static final User BOB = new User();

    @Test
    public void should_inform_when_users_are_not_fiends() {
        User user = aUser().friendsWith(ALICE).build();

        assertThat(user.isFriendWith(BOB)).isFalse();
    }

    @Test
    public void should_inform_when_users_are_fiends() {
        User user = aUser().friendsWith(ALICE, BOB).build();

        assertThat(user.isFriendWith(BOB)).isTrue();
    }
}
```

```java
public class User {

//…

    public boolean isFriendWith(User anotherUser) {
        return friends.contains(anotherUser);
    }
}
```

*Pour éviter le **NullPointerException**, verifier que les tableaux aient bien été initialisés dans le **`UserBuilder`***

```java
public class UserBuilder {
    private User[] friends = new User[]{};
    private Trip[] trips   = new Trip[]{};

    //…
}
```

Aviez-vous bien prévu dans le monteur qu'un utilisateur pourrait n'avoir aucun ami et/ou aucun voyage ?

Appeler la méthode `isFriendWith` depuis `TripService`

Tell don't ask !

```java
public class TripService {

    public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
        List<Trip> tripList = new ArrayList<Trip>();
        User loggedUser = getLoggedInUser();
        boolean isFriend = false;
        if (loggedUser != null) {
            isFriend = user.isFriendWith(loggedUser);
            if (isFriend) {
                tripList = tripsBy(user);
            }
            return tripList;
        } else {
            throw new UserNotLoggedInException();
        }
    }
  //…
}
```

```java
public class User {

//…

    public boolean isFriendWith(User anotherUser) {
        return friends.contains(anotherUser);
    }
}
```

Refactoring :
Un peu de nettoyage autour de `isFriend`

```java
public class TripService {

   public List<Trip> getTripsByUser(User user)
                        throws UserNotLoggedInExceptior
      //…
      boolean isFriend = false;
      if (loggedUser != null) {
         isFriend = user.isFriendWith(loggedUser);
         if (isFriend) {
            tripList = tripsBy(user);
         }
         return tripList;
      }
      //…
   }
 //…
}
```

Bonne Pratique :
Rapprocher la déclaration de la variable `isFriend`
au plus près du code où elle est utilisée

et simplifier en une ligne

```java
if (loggedUser != null) {
    boolean isFriend = user.isFriendWith(loggedUser);
    if (isFriend) {
         tripList = tripsBy(user);
    }
    return tripList;
}
```

```java
if (loggedUser != null) {
   if (user.isFriendWith(loggedUser)) {
         tripList = tripsBy(user);
    }
    return tripList;
}
```

Refactoring Inline de l'IDE
(suppression de la variable `isFriend`)

Après le petit nettoyage autour de `isFriend` …

```
public class TripService {

   public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
      List<Trip> tripList = new ArrayList<Trip>();
      User loggedUser = getLoggedInUser();
      if (loggedUser != null) {
         if (user.isFriendWith(loggedUser)) {
            tripList = tripsBy(user);
         }
         return tripList;
      } else {
        throw new UserNotLoggedInException();
      }
   }
 //… méthodes protected tripsBy et getLoggedInUser…
}
```

Bonne pratique :

Améliorer la **lisibilité** et réduire la **complexité** grâce aux clauses de garde (***guard clauses)***

Clause de garde: Définition

Guard Clause

A GuardClause (one of the SmalltalkBestPracticePatterns, and equally applicable in a whole bunch of languages) is a chunk of code at the top of a function (or block) that serves a similar purpose to a Precondition.

It typically does one (or any or all) of the following:

- checks the passed-in parameters, and returns with an error if they're not suitable.
- checks the state of the object, and bails out if the function call is inappropriate.
- checks for trivial cases, and gets rid of them quickly.

For example ...

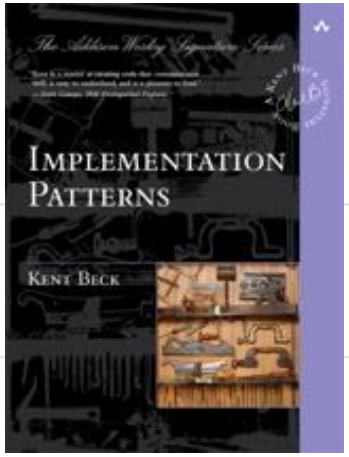
```
draw() {
  if (! isVisible()) return;
  ...
}
```

Extrait : https://wiki.c2.com/?GuardClause

Guard Clause

While programs have a main flow, some situations require deviations from the main flow. The guard clause is a way to express simple and local exceptional situations with purely local consequences. Compare the following:

```
void initialize() {
  if (!isInitialized()) {
    ...
  }
}
```



with:

```
void initialize() {
  if (isInitialized())
    return;
  ...
}
```

When I read the first version, I make a note to look for an else clause while I am reading the then clause. I mentally put the condition on a stack. All of this is a distraction while I am reading the body of the then clause. The first two lines of the second version simply give me a fact to note: the receiver hasn't been initialized.

If-then-else expresses alternative, equally important control flows. A guard clause is appropriate for expressing a different situation, one in which one of the control flows is more important than the other. In the initialization example above, the important control flow is what happens when the object is initialized. Other than that, there is just a simple fact to notice, that even if an object is asked to initialize multiple times it will only execute the initialization code once.

Extrait : http://www.informit.com/articles/article.aspx?p=1149121&seqNum=4

Clause de garde: Exemples & Avantages

Améliorer la lisibilité

Réduire la complexité

```
function calculateInsurance(userID: number){
    const user = myDB.findOne(userID);
    if(user){
      if(user.insurance === 'Allianz' or user.insurance === 'AXA'){
         if(user.nationality === 'Spain'){
            const value = /***
              Complex Algorithm
            */
            return value;
         }else{
            throw new UserIsNotSpanishException(user);
         }
      }else{
       throw new UserInsuranceNotFoundException(user);
      }
    }else{
     throw new UserNotFoundException('User NotFound!');
    }
}
```

```
function calculateInsurance(userID: number){
    const user = myDB.findOne(userID);
    if(!user){
      throw new UserNotFoundException('User NotFound!');
    }
    if(!isValidInsurante(user)){
       throw new UserInsuranceNotFoundException(user);
    }
    if(!isSpanish(user)){
       throw new UserIsNotSpanishException(user);
    }

    const value = /***
          Complex Algorithm
        */
    return value;
}
```

Extrait : https://medium.com/swlh/refactoring-guard-clauses-8f8e45fbc41e

Mise en place de la clause de garde …

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                   throws UserNotLoggedInException {
    List<Trip> tripList = new ArrayList<Trip>();
    User loggedUser = getLoggedInUser();
    if (loggedUser != null) {
       if (user.isFriendWith(loggedUser)) {
          tripList = tripsBy(user);
       }
       return tripList;
    } else {
      throw new UserNotLoggedInException();
    }
 }
 //… méthodes protected tripsBy et getLoggedInUser…
}
```

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                   throws UserNotLoggedInException {
    List<Trip> tripList = new ArrayList<Trip>();
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
        throw new UserNotLoggedInException();
    } else {
         if (user.isFriendWith(loggedUser)) {
         tripList = tripsBy(user);
         }
         return tripList;
    }
 }
 //… méthodes protected tripsBy et getLoggedInUser…
}
```

Inversion du if
À l'aide de l'IDE …

Refactoring :
Un peu de nettoyage autour des *trips*

Rapprocher la déclaration au plus près de son utilisation …

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                 throws UserNotLoggedInException {
    List<Trip> tripList = new ArrayList<Trip>();
    User loggedUser = getLoggedInUser();
    if (loggedUser != null) {
       if (user.isFriendWith(loggedUser)) {
          tripList = tripsBy(user);
       }
       return tripList;
    } else {
      throw new UserNotLoggedInException();
    }
 }
 //… méthodes protected tripsBy et getLoggedInUser
}
```

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                 throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
       throw new UserNotLoggedInException();
    } else {
       List<Trip> tripList = new ArrayList<Trip>();
       if (user.isFriendWith(loggedUser)) {
       tripList = tripsBy(user);
       }
       return tripList;
    }
 }
 //… méthodes protected tripsBy et getLoggedInUser
}
```

Rapprocher la déclaration au plus près de son utilisation

Supprimer la variable `tripList` …

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                    throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
          List<Trip> tripList = new ArrayList<>();
          if (user.isFriendWith(loggedUser)) {
              return tripsBy(user);
          }
          return tripList;
    }
 }
 //… méthodes protected tripsBy et getLoggedInUser
}
```

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                    throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
          if (user.isFriendWith(loggedUser)) {
              return tripsBy(user);
          }
          return new ArrayList<>();
    }
 }
 //… méthodes protected tripsBy et getLoggedInUser
}
```

Remplacer le premier `tripList` par **`return`**

Inliner via l'IDE le deuxième `tripList`

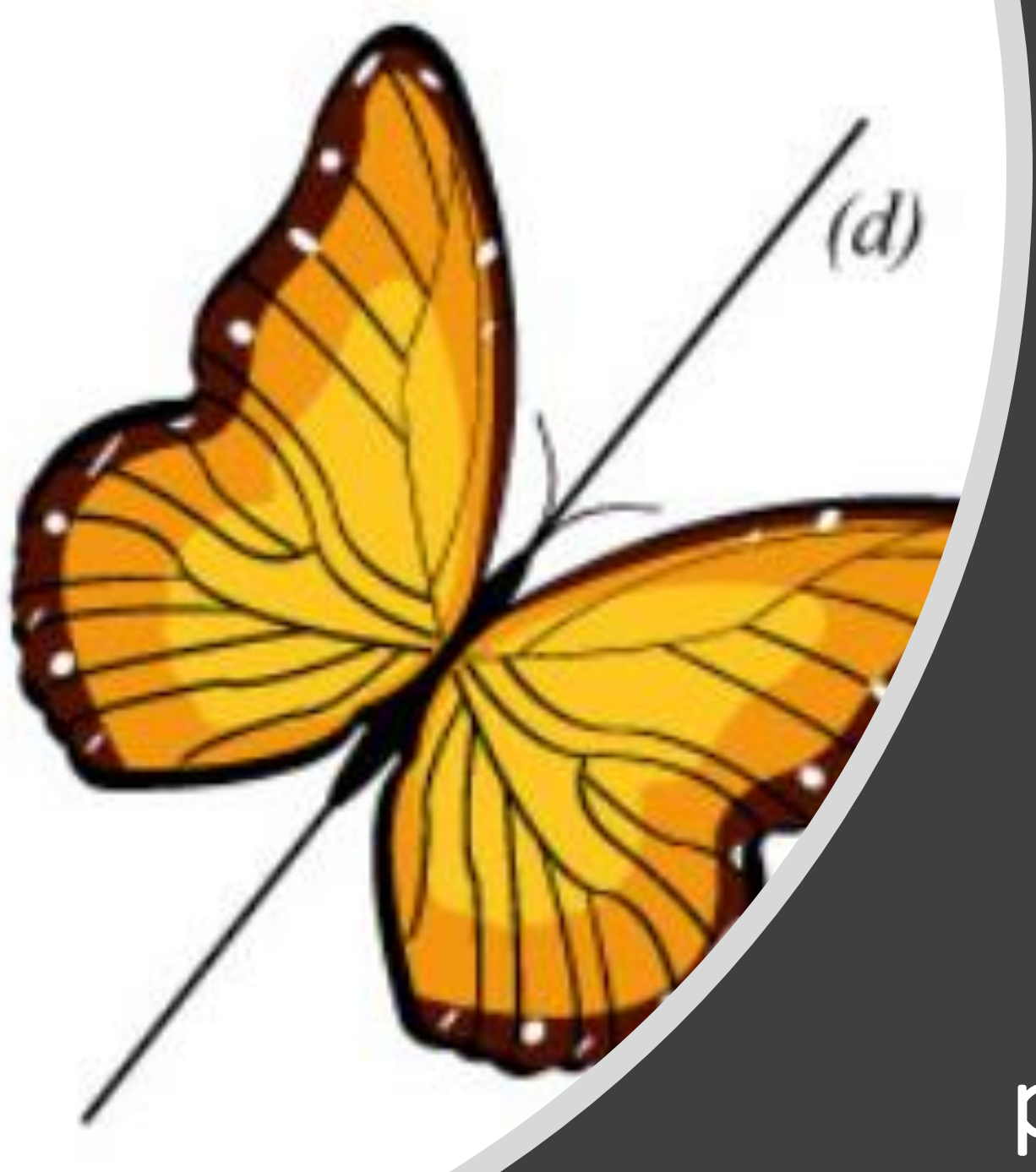
Isabelle BLASQUEZ

Améliorer la lisibilité en explicitant l'intention métier

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                     throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
         if (user.isFriendWith(loggedUser)) {
              return tripsBy(user);
          }
          return new ArrayList<>();
    }
 }
 //… méthodes protected tripsBy et getLoggedInUser
}
```

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                     throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
         if (user.isFriendWith(loggedUser)) {
              return tripsBy(user);
          }
          return noTrips();
    }
 }

  private ArrayList<Trip> noTrips() {
    return new ArrayList<>();
  }
 //… méthodes protected tripsBy et getLoggedInUser
}
```

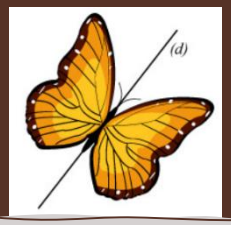
Extraire dans une méthode bien nommée
Qui rendre explicite l'intention métier : aucun voyage



Bonne pratique :

Ajouter du code
pour faire ressortir
la symétrie

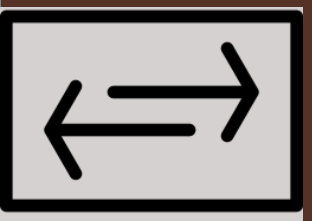
permet parfois
de simplifier
plus facilement le code



Rendre le `if` symétrique avec un `else` …

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                   throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
         if (user.isFriendWith(loggedUser)) {
              return tripsBy(user);
          }
          return noTrips();
    }
  }

  private ArrayList<Trip> noTrips() {
    return new ArrayList<>();
   }
//… méthodes protected tripsBy et getLoggedInUser
}
```

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                   throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
         if (user.isFriendWith(loggedUser)) {
              return tripsBy(user);
          }
         else {
          return noTrips();
         }
    }
  }

  private ArrayList<Trip> noTrips() {
    return new ArrayList<>();
   }
//… méthodes protected tripsBy et getLoggedInUser
}
```

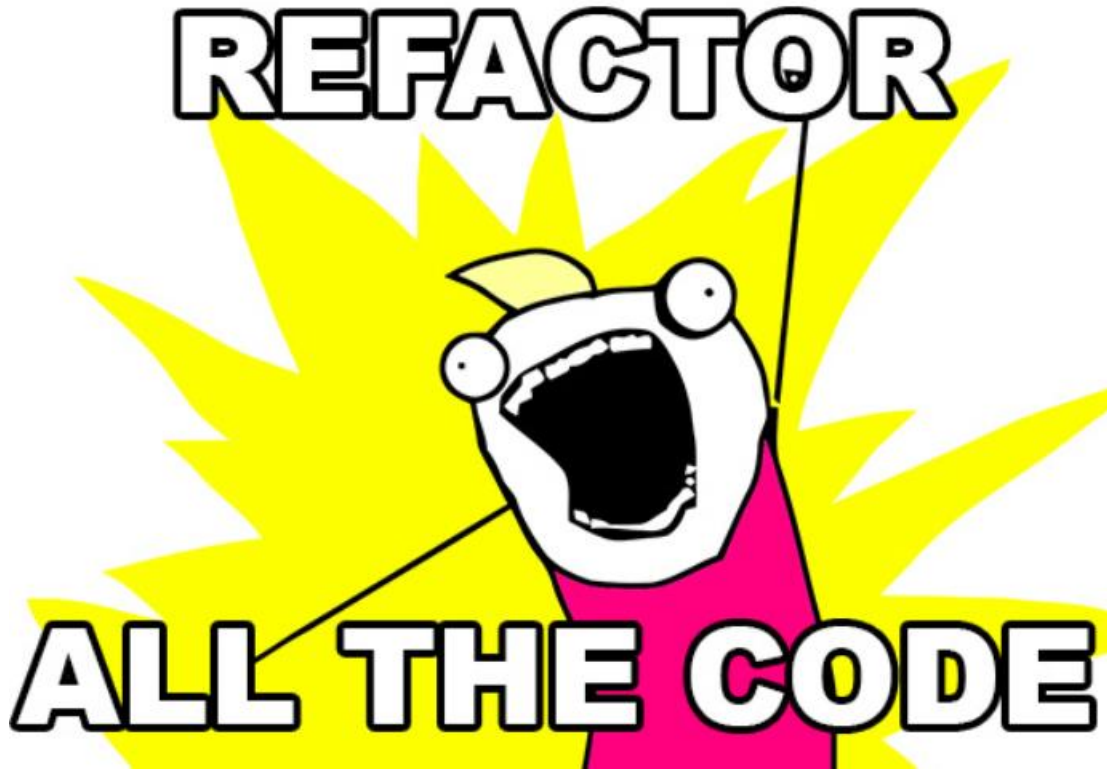


`if` classique ou opérateur ternaire selon sa préférence …

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                   throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
         if (user.isFriendWith(loggedUser)) {
               return tripsBy(user);
          }
          else {
           return noTrips();
          }
    }
  }

  private ArrayList<Trip> noTrips() {
    return new ArrayList<>();
   }
//… méthodes protected tripsBy et getLoggedInUser
}
```

If symétrique
Facilite écriture ternaire
(conversion auto. via IDE)

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                   throws UserNotLoggedInException {
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
         return user.isFriendWith(loggedUser)
                ? tripsBy(user)
                : noTrips();
    }
  }

  private ArrayList<Trip> noTrips() {
    return new ArrayList<>();
   }
//… méthodes protected tripsBy et getLoggedInUser
}
```


REFACTOR
ALL THE CODE

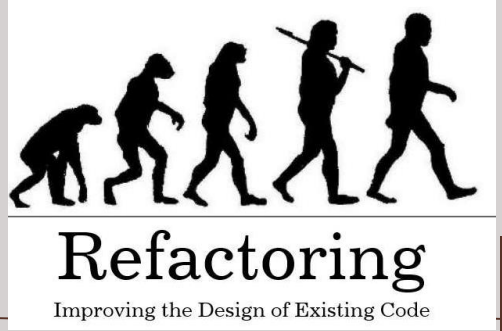
Les tests et le code parle le même langage …

```
public class TripServiceTest {

//…
@Before
public void setUp() {//…}

@Test(expected = UserNotLoggedInException.class)
public void
should_throw_an_exception_when_user_is_not_loggedd_in()
{//…}

public void
should_not_return_any_trips_when_users_are_not_friends(
) {//…}

@Test
public void should_return_trip_when_users_are_friends()
{//…}

private class TestableTripService extends TripService
{//…}

}
```

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                    throws UserNotLoggedInException{
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
      throw new UserNotLoggedInException();
    } else {
        return user.isFriendWith(loggedUser)
                    ? tripsBy(user)
                    : noTrips();
    }
  }

  private ArrayList<Trip> noTrips() {
    return new ArrayList<>();
   }
//… méthodes protected tripsBy
// et getLoggedInUser
}
```

Avant

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                     throws UserNotLoggedInException{
   List<Trip> tripList = new ArrayList<Trip>();
   User loggedUser =
         UserSession.getInstance().getLoggedUser();
   boolean isFriend = false;
   if (loggedUser != null) {
      for (User friend : user.getFriends()) {
         if (friend.equals(loggedUser)) {
            isFriend = true;
            break;
         }
      }
      if (isFriend) {
         tripList = TripDAO.findTripsByUser(user);
      }
      return tripList;
   } else {
     throw new UserNotLoggedInException();
   }
}
}
```



Après

```
public class TripService {

 public List<Trip> getTripsByUser(User user)
                     throws UserNotLoggedInException{
    User loggedUser = getLoggedInUser();
    if (loggedUser !== null) {
         throw new UserNotLoggedInException();
    } else {
         return user.isFriendWith(loggedUser)
                     ? tripsBy(user)
                     : noTrips();
   }
  }

  private ArrayList<Trip> noTrips() {
    return new ArrayList<>();
   }
//… méthodes protected tripsBy
// et getLoggedInUser
}
```

Rendre le code plus SOLID

et rompre les dépendances

… A suivre ….

Crédits Photos

https://fr.seaicons.com/verifiez-utilisateur-icone
https://fr.seaicons.com/icone-des-utilisateurs-65/
https://publicdomainvectors.org/fr/gratuitement-des-vecteurs/Image-d%E2%80%99ic%C3%B4ne-utilisateur-f%C3%A9minin/71148.html
https://www.aeroportlimoges.com/fileadmin/_processed_/7/2/csm_voyages-2_d1856d84b8.jpg
https://blog.malwarebytes.com/cybercrime/exploits/2018/01/meltdown-and-spectre-fallout-patching-problems-persist/
https://icon-icons.com/fr/icone/cible-objectif-la-reussite-le-but-le-tir-a-arc/55987

https://www.classe-numerique.fr/types-dactivites/flashcards/vois-tu-un-axe-de-symetrie
https://medium.com/swlh/refactoring-guard-clauses-8f8e45fbc41e
https://www.classe-numerique.fr/types-dactivites/flashcards/vois-tu-un-axe-de-symetrie
https://fr.seaicons.com/switch-icone-3/
https://stapp.space/content/images/2016/02/refactoring_poster-r723da0bee0b0423799654d8a59350032_i5v9b_8byvr_1024.jpg