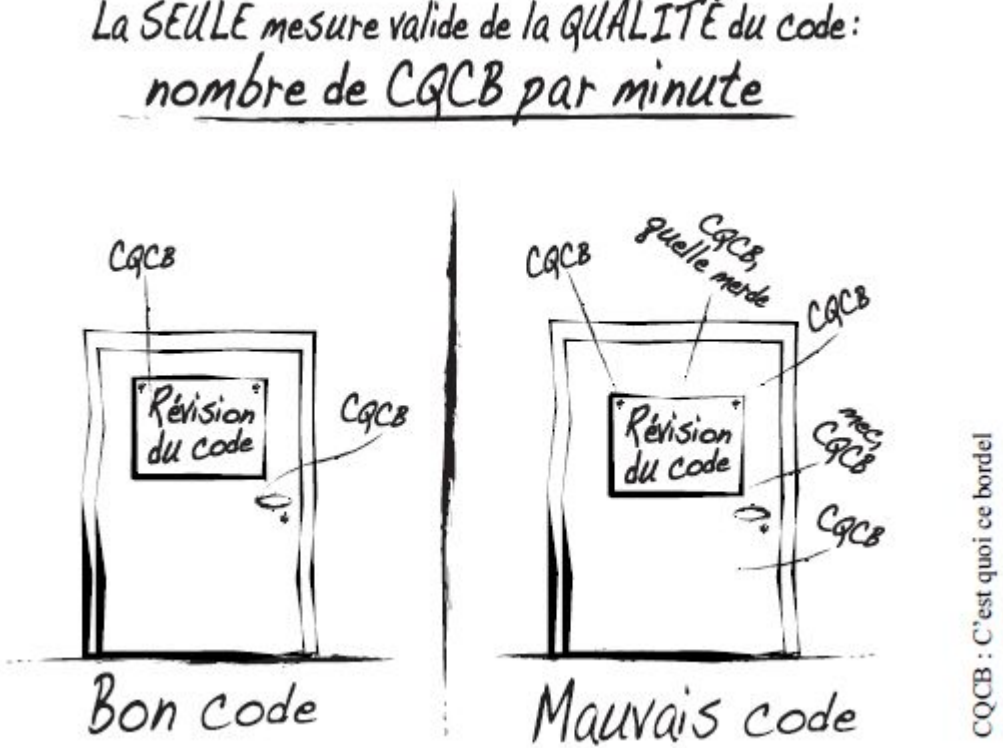
Programmation Java : Fin Projet du cours, Partie BackOffice JVA_260

Coder proprement, tests automatisés, Kata Bowling Game

CQCB


La SEULE mesure valide de la QUALITÉ du code:
nombre de CQCB par minute
CQCB
Révision du code
CQCB
Bon code
CQCB
CQCB, quelle merde
CQCB
Révision du code
mec, CQCB
CQCB
Mauvais code
CQCB : C'est quoi ce bordel

Bjarne Stroustrup, inventeur du C++ et auteur du livre Le Langage C++

J’aime que mon code soit élégant et efficace. La logique doit être simple pour que les bogues aient du mal à se cacher. Les dépendances doivent être minimes afin de faciliter la maintenance. La gestion des erreurs doit être totale, conformément à une stratégie articulée. Les performances doivent être proches de l’idéal afin que personne ne soit tenté d’apporter des optimisations éhontées qui dégraderaient le code. Un code propre fait une chose et la fait bien.

Grady Booch, auteur du livre Object Oriented Analysis and Design with Applications

Un code propre est un code simple et direct. Il se lit comme une prose parfaitement écrite. Un code propre ne cache jamais les intentions du concepteur, mais est au contraire constitué d'abstractions nettes et de lignes de contrôle franches.

"Big" Dave Thomas, fondateur d'OTI, parrain de la stratégie d'Eclipse

Un code propre peut être lu et amélioré par un développeur autre que l'auteur d'origine. Il dispose de tests unitaires et de tests de recette. Il utilise des noms significatifs. Il propose une manière, non plusieurs, de réaliser une chose. Ses dépendances sont minimales et explicitement définies. Il fournit une API claire et minimale. Un code doit être littéraire puisque, selon le langage, les informations nécessaires ne peuvent pas toutes être exprimées clairement par le seul code

Ward Cunningham, inventeur des wikis

Inventeur de Fit, co-inventeur de l'eXtreme Programming. Force motrice derrière les motifs de conception. Leader des réflexions sur Smalltalk et l'orienté objet. Parrain de tous ceux qui prennent soin du code.

Vous savez que vous travaillez avec du code propre lorsque chaque méthode que vous lisez correspond presque parfaitement à ce que vous attendiez. Vous pouvez le qualifier de beau code lorsqu'il fait penser que le langage était adapté au problème.

La règle du boy-scout

Laissez le campement plus propre que vous ne l’avez trouvé en arrivant.

Noms significatifs

Choisir des noms révélateurs des intentions

Éviter la désinformation

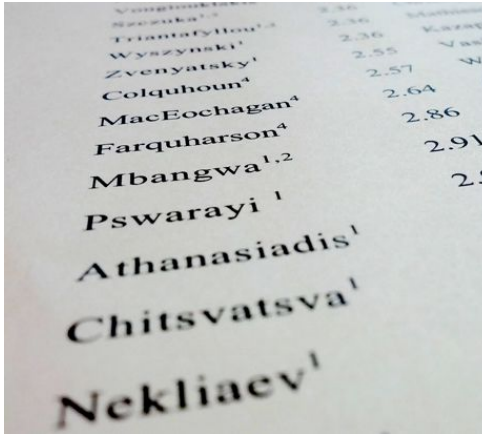
Faire des distinctions significatives

Choisir des noms prononçables

Choisir des noms compatibles avec une recherche

Éviter la codification

Choisir un mot par concept



Fonctions

Faire court / Faire une seule chose

Un niveau d'abstraction par fonction

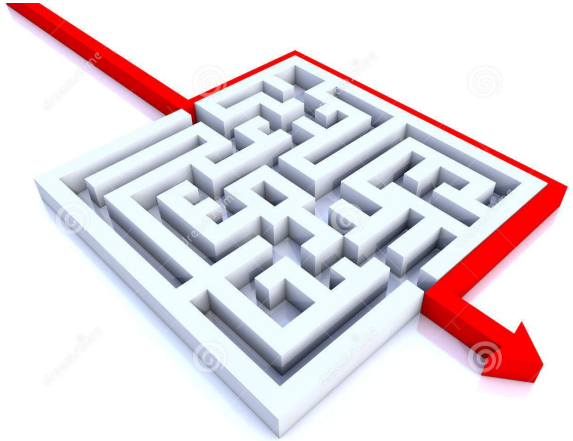
Arguments d'une fonction

Éviter les effets secondaires

Séparer commandes et demandes

Préférer les exceptions au retour de codes d'erreur

Ne vous répétez pas DRY



Commentaire

Ne pas compenser le mauvais code par des commentaires

S’expliquer dans le code

Bons commentaires?

Mise en forme

Objectif de la mise en forme

Mise en forme verticale

Mise en forme horizontale

Objets et structures de données

Antisymétrie données/objet

Objets de transfert de données

Gestion des erreurs

Utiliser des exceptions à la place des codes de retour

Commencer par écrire l'instruction try-catch-finally

Employer des exceptions non vérifiées

Ne pas retourner null

Ne pas passer null

Tests unitaires

F.I.R.S.T

FAST / INDEPENDANT / REPEATABLE / SELF VALIDATING / TIMELING

Garder des tests propres

TDD

Première loi. Vous ne devez pas écrire un code de production tant que vous n’avez pas écrit un test unitaire d’échec.

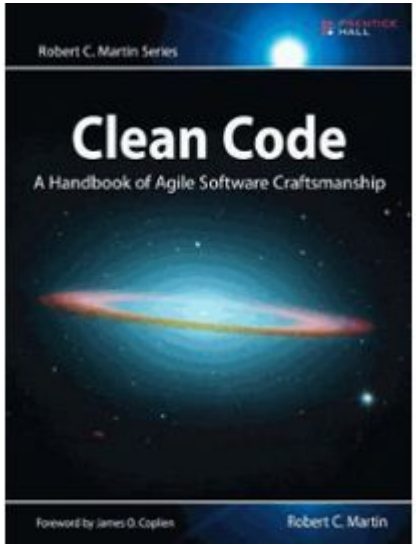
Deuxième loi. Vous devez uniquement écrire le test unitaire suffisant pour échouer ; l’impossibilité de compiler est un échec.

Troisième loi. Vous devez uniquement écrire le code de production suffisant pour réussir le test d’échec courant

Pour aller plus loin

Clean Code

Robert C. Martin



Focus sur les différents type de TESTS AUTOMATISES

Tests unitaires (Objet)

```
@Test
public void shouldNotValidateWhenFirstNameEmpty() {
    Person person = new Person();
    person.setFirstName("");

    Validator validator = createValidator();
    Set<ConstraintViolation<Person>> constraintViolations = validator.validate(person);

    assertThat(constraintViolations.size()).isEqualTo(1);
    ConstraintViolation<Person> violation = constraintViolations.iterator().next();
    assertThat(violation.getPropertyPath().toString()).isEqualTo("firstName");
    assertThat(violation.getMessage()).isEqualTo("must not be empty");
}
```

Vérifier règles métier

Pur JAVA
Pas de base de données
Pas de Spring
Pas (souvent) de mock
Pas de problème

Test unitaire DAO

```
@Test
  public void testFindByType() throws Exception {

      List<Pokemon> pokemons = dao.findByType(Type.ELECTRIC);
      assertThat(pokemons.get(0).getName()).isEqualTo("Pikachu");
  }
```

Si règles métier dans DAO
⇒ tests compliqués

Vérification des requêtes

Besoin d'une base de données
⇒ Ce n'est plus un test unitaire

Utilisation d'une base de données en mémoire ? Distante ?

Comment insérer des données ?
Comment supprimer les données ?

Test unitaire Service

Vérification règles métier

Besoin de mocker les DAO
Besoin de vérifier les appels aux DAO

```
@Test(expected = NumeroSecuAlreadyExists.class)
public void testAddCustomer_alreadyExists() {
    when(daoMock.findByNumeroSecu("1234")).thenReturn(new Customer());

    Customer customer = new Customer();
    customer.setNumeroSecu("1234");
    customerService.createNewCustomer(customer);
}
```

Tests unitaires de Controleur (SpringMVC)

```
@Test
public void testSaveOrder() throws Exception {
    String payload = "{ \"products\": [{ \"name\": \"Mon produit\" }]}";
    MockHttpServletRequestBuilder req = post(SERVICE_URI)
            .contentType(APPLICATION_JSON)
            .accept(APPLICATION_JSON_UTF8)
            .content(payload);
    verify(mockService).saveOrder(argThat(productWithName("Mon produit")));
    this.mockMvc.perform(req).andExpect(status().isOk());
}
```

Vérification paramètres (format) et appels aux service

Besoin de mocker les DAO
Besoin de vérifier les appels aux DAO

Tests d'intégration

```
@Test
public void testRequest() {
    HttpHeaders headers = this.template.getForEntity("/example", String.class)
                                .getHeaders();
    assertThat(headers.getLocation()).hasHost("other.example.com");
}
```

Tests unitaires vs test intégrations

Tests unitaires : je teste tous (ou presque) les cas. Vérifie que les règles métiers sont respectés

Tests d'intégration : je teste le cas nominal (ou plus mais nombre limité). Vérifie que toutes couches communiquent bien entre elles.

Tests de performance (gatling)


```
val scn = scenario("View 5 random products")
  .exec(http("Home page").get("/").check(status.is(200)))
  .exec(http("View the list of products").get("/product").check(status.is(200)))
  .repeat(5) {
    feed(products).exec(http("View a random product").get("/product/${productId}").check(status.is(200)))
  }
```

```
val httpConf = httpConfig.baseURL("http://localhost:8085/bees-shop")
setUp(
  scn.scn
    .inject(rampRate(10 usersPerSec) to(100 usersPerSec) during(5 minutes))
    .protocolConfig(httpConf)
)
```

Bowling game kata

Tennis game kata