



Chapitre 14

Validation et sécurité

Tests, validations et sécurité dans les applications

1. Clean code

Clean code

- Code propre
 - ⇒ Permet d'éviter des erreurs
 - ⇒ Plus facile de s'assurer que le code est correct
- Réutilisation plus aisée
 - Car plus facile à lire et à comprendre

Validation et sécurité

1. Clean code
2. Jeux de tests

Jeux de tests

Tester pour valider le code

⇒ Attention à la qualité des jeux de tests !

Ex: Intervalles de valeurs



Jeu de test : 5 valeurs minimum

1. Valeur < A
2. A
3. $A < \text{Valeur} < B$
4. B
5. Valeur > B

Validation et sécurité

1. Clean code
2. Jeux de tests
3. Testing

Test unitaires

- Pour vérifier le bon fonctionnement d'une portion de code
 - Lors d'une modification d'un programme, les tests unitaires peuvent signaler les éventuelles erreurs
- Test Driven Development (TDD)
 - Spécifications déclinées en tests cases
- Extreme programming (XP)
 - Type de développement Agile
 - Petits cycles de développement ⇒ fréquents releases
 - Programmation "in pairs" (par deux programmeurs)

Test d'intégration

- Chacun des modules indépendants du logiciel est assemblé et testé dans l'ensemble

Test de validation

- **Validation fonctionnelle**
 - Assurer que les différents modules implémentent correctement les exigences client
- **Validation de solution**
 - Validation des use cases
 - Chaque cas d'utilisation est validé isolément
 - Puis tous les cas d'utilisation sont validés ensemble
- **Validation de performance**
 - Vérifier la conformité de la solution par rapport à ses exigences de performance
- **Validation de robustesse**
 - Mettre en évidence des éventuels problèmes de stabilité et de fiabilité dans le temps

Validation et sécurité

1. Clean code
2. Jeux de tests
3. Testing
4. Découpe en couches

Découpe en couches

- Une couche = une boîte noire
 - Indépendante des autres couches
 - Sécurisée au maximum
- Les valeurs en entrée doivent être testées!
- Chaque couche doit effectuer ses propres tests
 - Sur les valeurs en entrée

Découpe en couches

Exemple: quantité commandée introduite par l'utilisateur

Couche vue

- *Composants swing*
Tester la valeur introduite et afficher une boîte de dialogue si pas OK
- *Page HTML*
Tester en Javascript

Couche business

Tester la quantité commandée en Java et remonter une *exception* si pas OK

Base de données

Prévoir des *checks* dans le script SQL de création de la base de données
ex: *check (quantiteCommande > 0)*

Découpe en couches

Intérêts de placer les mêmes tests dans différentes couches?

Intérêt de retester les valeurs en entrée dans la couche Business :

Le Javascript peut être désactivé par les internautes!

La couche Business peut être réutilisée dans d'autres applications

Intérêt de placer des checks SQL dans la base de données :

Une base de données n'est pas liée à une application,
elle peut être **réutilisée** dans le futur par **d'autres applications**

Validation et sécurité

- 1. Clean code**
- 2. Jeux de tests**
- 3. Testing**
- 4. Découpe en couches**
- 5. Validation des formulaires**

Validation des formulaires

Couche vue

Validation du formulaire avant de créer des objets et de les envoyer aux autres couches

Validation des formulaires

Erreurs possibles sur une valeur introduite par un utilisateur

- Champ obligatoire non rempli
- Valeur numérique contenant des caractères
- Nombre négatif ou nul (si valeur positive attendue)
- Valeur non comprise dans la liste des valeurs permises
- Nouveau login déjà existant
- Format non respecté (ex: email)

Voire **malveillance** volontaire

(ex: tentative d'injection SQL)



Validation et sécurité

- 1. Clean code**
- 2. Jeux de tests**
- 3. Testing**
- 4. Découpe en couches**
- 5. Validation des formulaires**
- 6. Tester les paramètres**

Tester les paramètres

Couche Business

= Librairie de fonctions métiers réutilisables

⇒ Tester les valeurs des arguments (= entrées) des méthodes

⇒ Si pas valables ⇒ Exception remontée

Validation et sécurité

1. Clean code
2. Jeux de tests
3. Testing
4. Découpe en couches
5. Validation des formulaires
6. Tester les paramètres
7. Information Hiding

Information Hiding

Couche Model

Empêcher de créer des "mauvais" objets

(c'est-à-dire avec des mauvaises valeurs dans leurs variables d'instance)

- ⇒ Variables d'instance **privées**
- ⇒ **Settors** publiques avec rôle de **filtres**
- ⇒ OK seulement si settors **appelés dans le constructeur**

Information Hiding

Couche Model

Contre-exemple

```
public class Division
{ private int denominator; ...

  public int getDenominator() { return denominator; }

  public void setDenominator (int denominator) throws DenominatorException
  { if (denominator == 0) throw new DenominatorException();
    else this.denominator = denominator; }

  public Division (int denominator, ...) throws DenominatorException
  { this.denominator = denominator; } ⇒ { setDenominator(denominator); }
}
```

Validation et sécurité

- 1. Clean code**
- 2. Jeux de tests**
- 3. Testing**
- 4. Découpe en couches**
- 5. Validation des formulaires**
- 6. Tester les paramètres**
- 7. Information Hiding**
- 8. Administration de base de données**

Bons réflexes d'administrateur BD

- Création d'utilisateurs : **login** et **mot de passe**
 - Pour empêcher les accès non autorisés aux données

Administration de base de données

Maintenir la base de données dans un état cohérent via :

- Bons types de colonnes
 - Assure l'introduction de bons types de données
- Not null éventuels
 - Empêche les données incomplètes
- Primary key
 - Evite la duplication des données
- Foreign Key
 - Oblige d'insérer dans une colonne uniquement des valeurs qui existent dans une autre table de la BD
 - Evite des liens incohérents entre données
- Checks
 - Empêche l'introduction de valeurs invalides

Validation et sécurité

1. Clean code
2. Jeux de tests
3. Testing
4. Découpe en couches
5. Validation des formulaires
6. Tester les paramètres
7. Information Hiding
8. Administration de base de données
9. Injection de code

Injection de code

- OWASP
 - Communauté libre
- But
 - Permettre aux organisations de concevoir des applications auxquelles on peut faire confiance
- Top ten Owasp en 2017
 - Most Critical Web Application Security Risks
 - **1. Injection de code** (SQL...)
 - *"Untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization."*
 - https://www.owasp.org/index.php/Top_10_2017-Top_10

Injection SQL

- Rappel
 - La classe **Statement** ne permet pas d'éviter les injections SQL
- Eviter les injections SQL
 - Via **PreparedStatement** utilisés à bon escient
 - ⇒ En utilisant les ?

Injection SQL

Contre-exemple

Login :

Password :

```
TextField loginField, passwordField;
```

```
String requeteSQL =
```

```
"select * from user where login = ' " + loginField.getText() + " ' and password =  
' " + passwordField.getText() + " ' " ;
```

```
PreparedStatement statement = connexion.prepareStatement(requeteSQL );
```

Injection SQL

Exemple OK

Login :

Password :

```
TextField loginField, passwordField;
```

```
String requeteSQL = "select * from user where login = ? and password = ? " ;
```

```
PreparedStatement statement = connexion.prepareStatement(requeteSQL );
```

```
statement.setString(1, loginField.getText() );
```

```
statement.setString(2, passwordField.getText() );
```

Validation et sécurité

1. Clean code
2. Jeux de tests
3. Testing
4. Découpe en couches
5. Validation des formulaires
6. Tester les paramètres
7. Information Hiding
8. Administration de base de données
9. Injection de code
10. Cryptage des données

Cryptage des données

- Protection des données sensibles
 - Crypting, Hashing...
- Attention : utiliser un algorithme récent !
- *Pas de mot de passe en clair dans les bases de données !*