



UNIVERSITÉ
LAVAL

Atelier: Les tests en pratique

GLO-2003

Tests: leurs propriétés

➤ Automatiques

- Facile et rapide à exécuter
- Répétable (pas de test *flaky*)
- Rapide à exécuter

➤ Résultat binaire

- Passe ou échec

➤ Les tests sont indépendants

- On ne reteste pas plusieurs fois la même chose
- Assume que les autres tests font leur travail
- Évite qu'un bogue cause plusieurs tests à échouer

➤ Ce qui signifie... que le *Single-responsibility principle* s'applique aussi aux tests

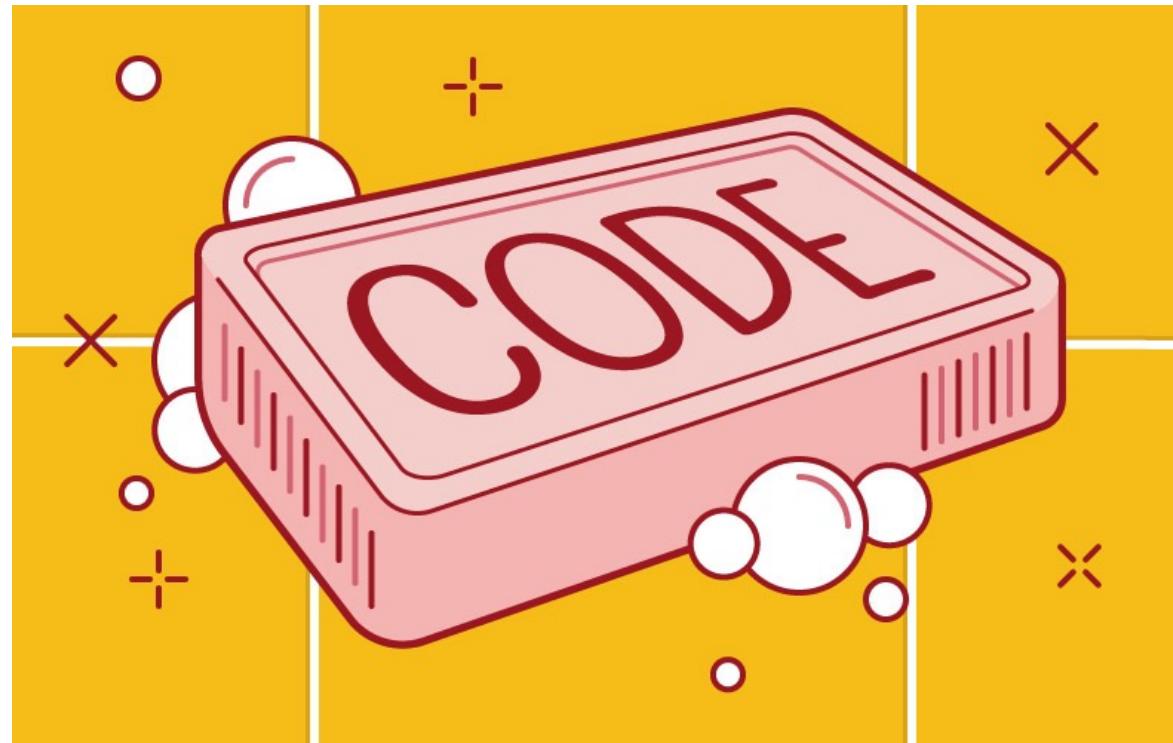
- Un test ne teste qu'une chose
- Un seul concept / comportement par test
- Idéalement une seule assertion logique par test
 - Une assertion logique peut correspondre à plus d'un assert dans le code

Robert C. Martin. Clean Code : A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, USA, 1 edition, 2008. ISBN 0132350882. Chapitre 9

Tests: leurs propriétés

➤ Les tests sont aussi du code

- On doit les **maintenir** aussi
- Ils doivent être aussi propres que le “vrai” code



Structure d'un test

➤ 3A: Arrange-Act-Assert

- Arrange: Prépare les données de test
- Act: Appelle la méthode sous test
- Assert: Valide le résultat

➤ Facile de repérer les différentes parties du test

➤ Permet de facilement comprendre le test

```
@Test
void test() {
    // Arrange
    Calculator calculator = new Calculator();

    // Act
    boolean result = calculator.isOdd(SOME_ODD_NUMBER);

    // Assert
    assertThat(result).isEqualTo(expected: true);
}
```

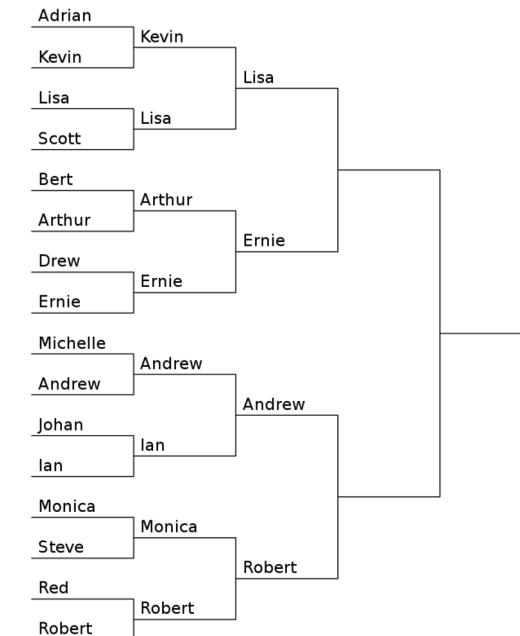
Étude de cas — UTournament

➤ API de gestion de tournois

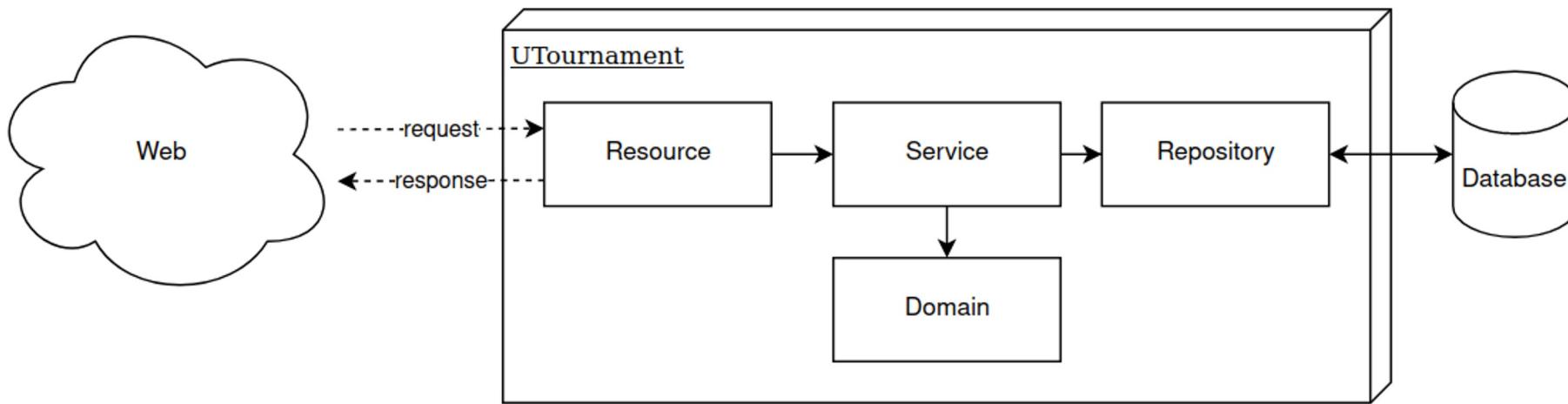
- Création de tournois à partir d'une liste de participants
- Gestion de la progression du tournoi
- Suppression de tournois

➤ API REST avec Spark en Java

➤ Code source: [glo2003/UTournament](#)



Étude de cas — UTournament



Types de tests

1. Tests unitaires

- Testent une unité en isolation des autres

2. Tests d'intégration

- Testent l'intégration (la collaboration) entre des composants

3. Tests de système

- Testent le système complet
- Aussi connus sous le nom de tests de bout en bout (*end-to-end testing*)

4. Tests d'acceptation

- Testent si le logiciel fait ce qui est demandé par le client
- Centrés sur un scénario d'utilisation
- Aussi connus sous les noms: tests fonctionnels, tests clients ou *story tests*

<https://www.guru99.com/levels-of-testing.html>

<https://www.agilealliance.org/glossary/unit-test/>

<https://www.agilealliance.org/glossary/acceptance>

1. Tests unitaires

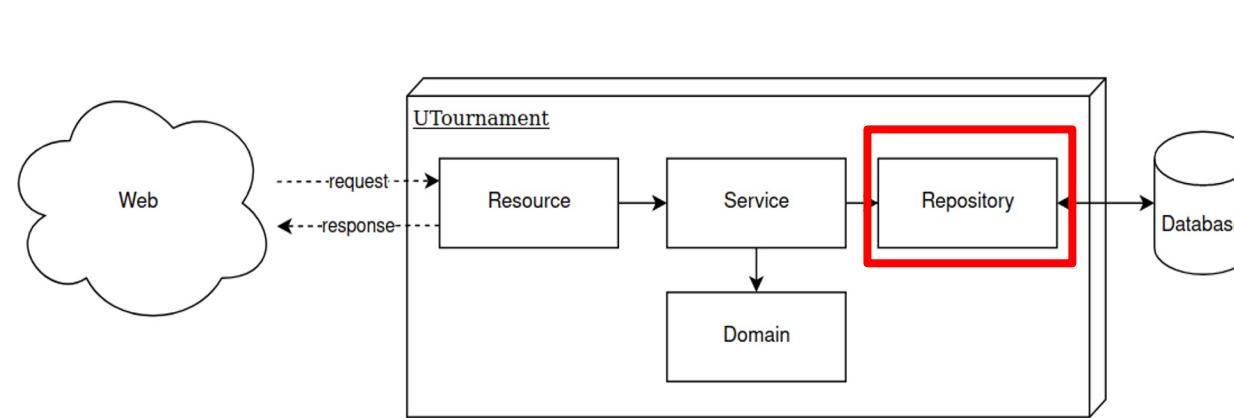
- Testent une unité
 - Une classe, une méthode, un module, une fonction
- Isolation de l'unité avec des *Mocks*
 - Remplace les autres unités avec lesquelles l'unité testée interagit
 - Permet de ne tester que l'unité sous test
 - Possibilité de piloter les mocks
 - Facilite la mise en place des tests
- On teste des comportements ou des actions, pas les détails d'implémentation
 - Évite de créer des tests fragiles qui brisent souvent
- Outils
 - JUnit 5 (Framework de tests)
 - Google Truth (Permet de mieux exprimer des assertions)
 - Mockito (Bibliothèque de *Mocks*)

<https://www.agilealliance.org/glossary/unit-test/>

<https://martinfowler.com/articles/practical-test-pyramid.html#UnitTests>

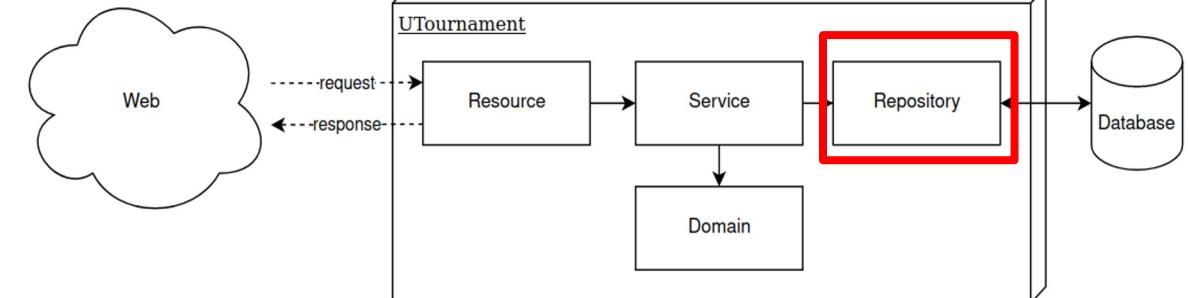
Exemple de test unitaire — TournamentRepository

- Classe permettant de stocker, d'obtenir et de supprimer des Tournaments
- On a une implémentation qui stocke les Tournaments dans une table de hachage (HashMap)
- [Code source](#)
- [Code des tests](#)



Exemple de test unitaire: Code à tester

```
public class InMemoryTournamentRepository implements TournamentRepository {  
    private final Map<TournamentId, Tournament> database;  
  
    public InMemoryTournamentRepository() {  
        database = new HashMap<>();  
    }  
  
    @Override  
    public Optional<Tournament> get(TournamentId id) {  
        return Optional.ofNullable(database.get(id));  
    }  
  
    @Override  
    public void save(Tournament tournament) {  
        TournamentId tournamentId = tournament.getTournamentId();  
        database.put(tournamentId, tournament);  
    }  
  
    @Override  
    public void remove(TournamentId tournamentId) {  
        database.remove(tournamentId);  
    }  
}
```



Exemple de test unitaire: Classe de test et setup

```
class InMemoryTournamentRepositoryTest {  
    private InMemoryTournamentRepository repository;  
    private Tournament tournament;  
    private TournamentId tournamentId;  
  
    @BeforeEach  
    void setUp() {  
        repository = new InMemoryTournamentRepository();  
        List<Participant> participants = List.of();  
        String tournamentName = "smash";  
        tournament = new Tournament(new TournamentId(),  
            tournamentName,  
            participants,  
            new ByeBracket(new BracketId(), new Participant( name: "Alice")));  
        tournamentId = tournament.getTournamentId();  
    }  
}
```

- `setUp` est exécuté avant chaque test
- *Arrange* commun à plusieurs tests
- Permet de réinitialiser les objets entre les tests

Exemple de test unitaire: Méthodes de test

- Arrange-Act-Assert
- Noms descriptifs
- Assertions avec Google Truth

```
@Test
void returnTournamentWhenInRepository() {
    repository.save(tournament);

    Optional<Tournament> gottenTournament = repository.get(tournamentId);

    assertThat(gottenTournament.isPresent()).isTrue();
    assertThat(gottenTournament.get()).isEqualTo(tournament);
}

@Test
void returnEmptyWhenTournamentNotInRepository() {
    Optional<Tournament> tournament = repository.get(tournamentId);

    assertThat(tournament.isEmpty()).isTrue();
}
```

Exemple de test unitaire: Méthodes de test

```
@Test
void canDeleteTournamentFromRepository() {
    repository.save(tournament);

    repository.remove(tournament.getTournamentId());

    Optional<Tournament> tournament = repository.get(tournamentId);
    assertThat(tournament.isPresent()).isFalse();
}

@Test
void canDeleteTournamentNotInRepositoryWithoutError() {
    repository.remove(tournament.getTournamentId());

    Optional<Tournament> tournament = repository.get(tournamentId);
    assertThat(tournament.isPresent()).isFalse();
}
```

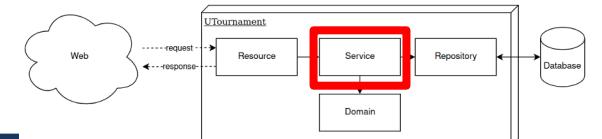
- Arrange-Act-Assert
- Noms descriptifs
- Assertions avec Google Truth

Exemple de test unitaire avec des Mocks

- On désire tester la méthode `deleteTournament` du service `TournamentService` qui supprime un tournoi du `TournamentRepository`.
- On utilise un mock pour plusieurs raisons
 - Comme `TournamentRepository` est déjà testé unitairement, on mock pour ne pas tester à nouveau ses comportements. On ne veut tester que le `TournamentService`
 - `TournamentRepository` est une interface, et l'on ne veut pas devoir créer une vraie instance seulement pour les tests
 - `TournamentRepository` pourrait communiquer avec une vraie base de données, on mock alors pour éviter cela et garder les tests rapides à executer

- Code source
- Code des tests

```
public void deleteTournament(String tournamentIdString) {  
    TournamentId tournamentId = TournamentId.fromString(tournamentIdString);  
    tournamentRepository.remove(tournamentId);  
}
```



Exemple de test unitaire avec des Mocks

```
@ExtendWith(MockitoExtension.class)
class TournamentServiceTest {

    @Mock
    TournamentRepository tournamentRepository;

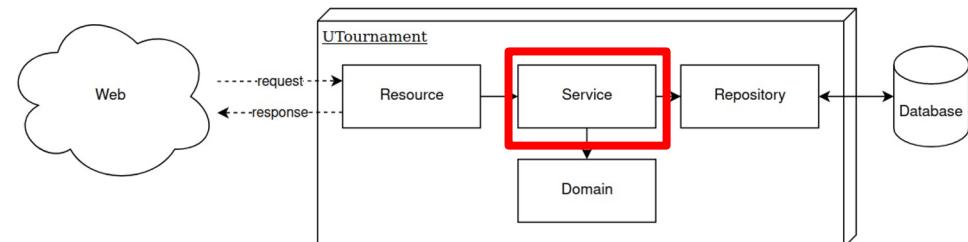
    @BeforeEach
    void setUp() {
        tournamentService = new TournamentService(tournamentFactory,
            tournamentRepository,
            findPlayableBracketsVisitor,
            winBracketVisitor);
    }

    @Test
    void canDeleteTournament() {
        tournamentService.deleteTournament(TOURNAMENT_ID_STRING);

        verify(tournamentRepository).remove(TOURNAMENT_ID);
    }
}
```

- Le mock crée un faux objet
- On l'injecte dans le constructeur du service
- Permet de vérifier que certains appels de méthode ont été faits

Le `@ExtendWith` permet d'utiliser des mocks avec JUnit5



Exemple de test unitaire avec des Mocks

- Il est aussi possible de piloter les mocks
- `when().thenReturn()` permet de retourner un objet en particulier lorsque certains arguments sont passés à une méthode
- Facilite la mise en place des tests

```
@Test
void throwsWhenTournamentNotFound() {
    when(tournamentRepository.get(TOURNAMENT_ID)).thenReturn(Optional.empty());

    assertThrows(TournamentNotFoundException.class,
        () -> tournamentService.getTournament(TOURNAMENT_ID_STRING));
}

@Test
void canGetTournamentFromId() {
    when(tournamentRepository.get(TOURNAMENT_ID)).thenReturn(Optional.of(tournament));

    TournamentDto gottenTournament = tournamentService.getTournament(TOURNAMENT_ID_STRING);

    assertThat(gottenTournament).isEqualTo(tournamentDto);
}
```

2. Tests d'intégration

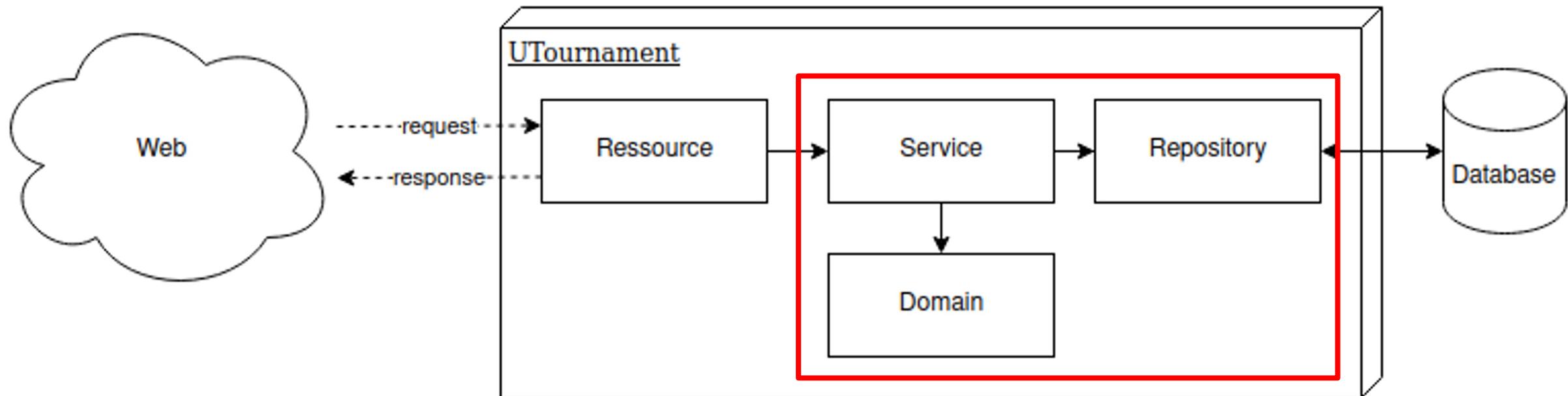
- Testent la collaboration entre des composantes (sans mock)
- Permettent de tester le flux de données entre les composantes
- Supposent que les composantes fonctionnent
 - Elles ont été testées unitairement, on ne teste pas en double
 - On vise surtout à tester que les unités interagissent correctement
- Outil
 - [JUnit 5](#) (Framework de tests)
 - [Google Truth](#) (Permet de mieux exprimer des assertions)



Même si A et B sont corrects (testés unitairement), un bogue peut se glisser dans leur interaction.

Exemple de test d'intégration — TournamentService

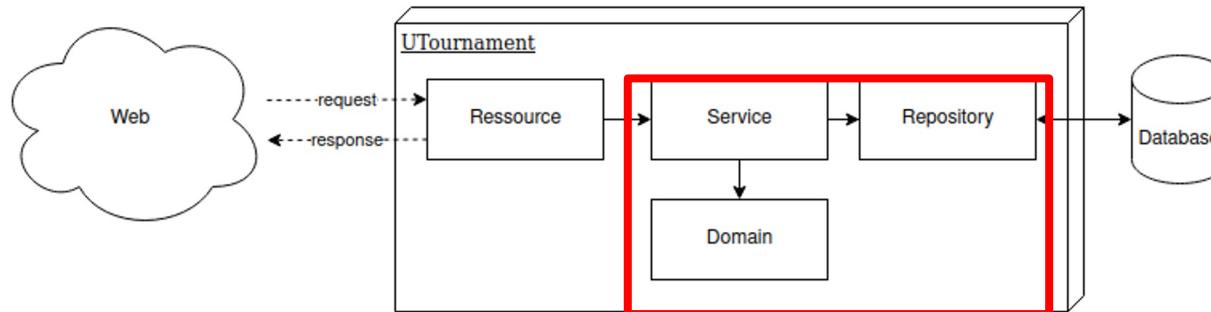
- [Code source](#)
- [Code des tests](#)



Exemple de test d'intégration

- Tests sans mock
- On crée de vraies instances des dépendances

```
@Test  
void canGetCreatedTournament() {  
    TournamentId tournamentId = tournamentService.createTournament(TOURNAMENT_NAME, participantDtos);  
    String tournamentIdString = tournamentId.toString();  
  
    TournamentDto tournamentDto = tournamentService.getTournament(tournamentIdString);  
  
    assertThat(tournamentDto.tournamentId).isEqualTo(tournamentIdString);  
}
```



3. Tests de système

- Testent le système complet
 - Testent toutes les couches du système de l'interface utilisateur aux bases de données
- Permettent aussi de tester la configuration du programme
- Dans le cas d'une API
 - Démarre le serveur et simule des requêtes
 - Outils
 - [REST-assured](#)
- Dans le cas d'une application avec interface utilisateur
 - Démarre l'application et simule des entrées utilisateurs
 - Outils pour app web (JavaScript / ECMA Script)
 - [Cypress](#)
 - [Selenium](#)

<https://www.guru99.com/system-testing.html>

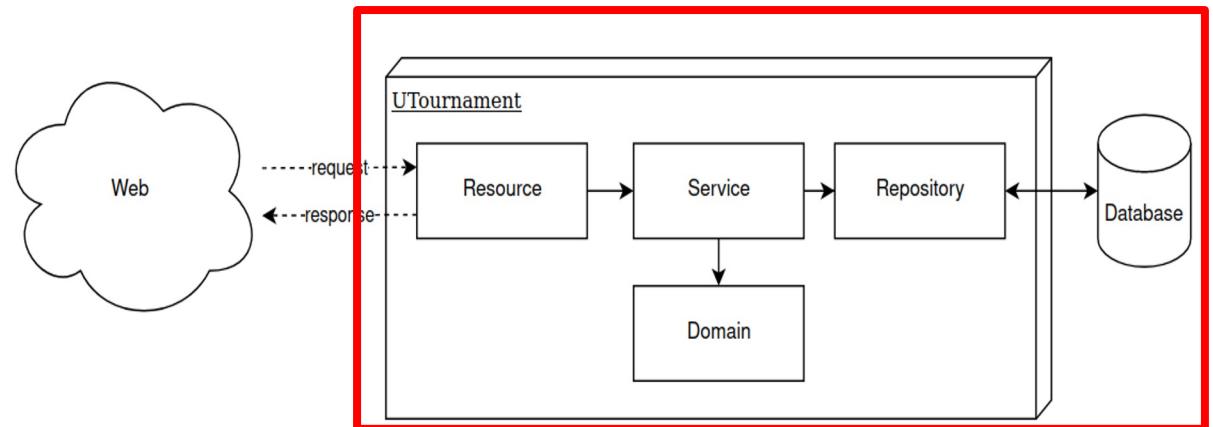
Tests de système: Quoi et comment tester

- On évite les tests trop précis qui *assert* trop de choses sur le système
 - Ces comportements sont déjà testés avec des tests unitaires et intégrés.
 - On évite ainsi des tests fragiles qui seront toujours brisés à la moindre modification.
 - Il faut que ces tests puissent bien évoluer avec le changement du projet sans demander des modifications constantes.
- En général, on évite
 - Une pléthore d'assertions sur la réponse du système
 - Qu'une route retourne exactement cette valeur
- On préfère
 - Valider qu'un code HTTP ne retourne pas d'erreur
 - Valider qu'un header HTTP n'est pas vide
 - etc.

Exemple de test de bout en bout

```
@BeforeAll  
static void setUp() {  
    UTournament.main(new String[0]);  
}  
  
@AfterAll  
static void tearDown() {  
    stop();  
}
```

- Code des tests
- On démarre le serveur avant tous les tests
- On ferme après l'exécution des tests



Exemple de test de bout en bout

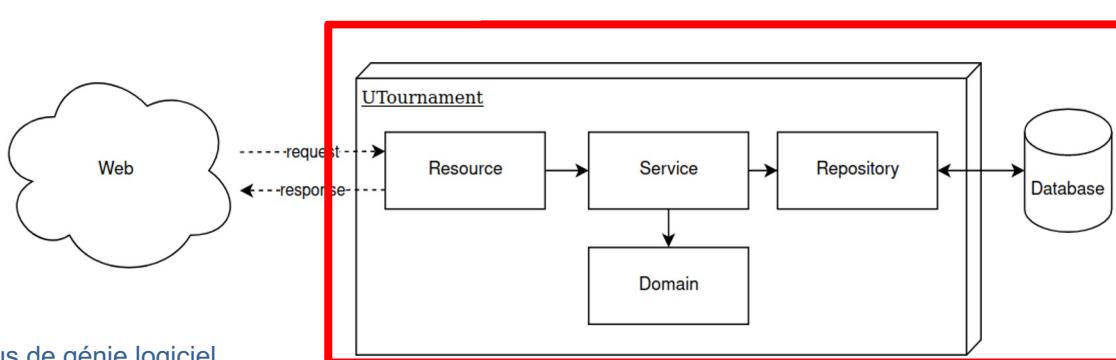
```
@Test
void healthReturnsOk() {
    Response response = when().get( $: BASE_URL + "/health");

    response.then().statusCode(200);
}

@Test
void canCreateTournamentHasLocationHeader() {
    TournamentCreation tournamentCreation = new TournamentCreation();
    tournamentCreation.name = "Smash";
    tournamentCreation.participants = ParticipantTestUtils.createParticipantDtos( num: 16);

    Response response = given().body(tournamentCreation).
        when().post( $: BASE_URL + "/");
    response.then().statusCode(200).header( $: "Location", not(emptyOrNullString()));
}
```

- On fait des requêtes HTTP sur le serveur
- Validations haut niveau
 - Status code
 - Headers
 - etc.



Comparaison entre les tests unitaires, intégrés et systèmes

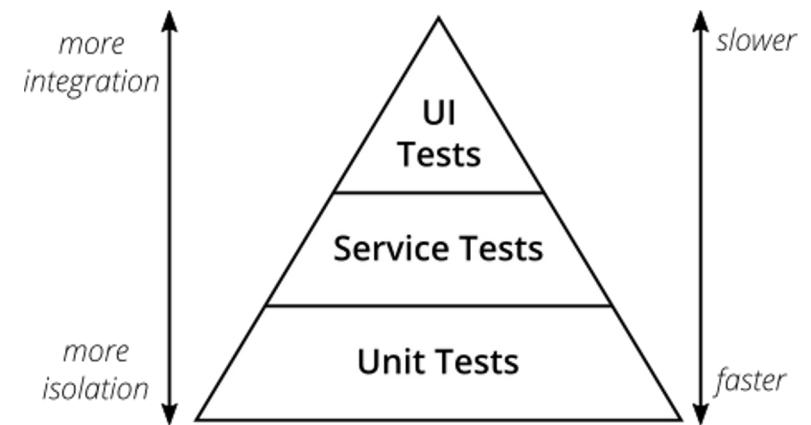
➤ Portée des tests

- Unitaire = une classe
- Intégration = quelques classes
- Système = tout le système

➤ Pyramide des tests

- Il y aura plus de tests unitaires que de tests intégrés, et plus de tests intégrés que de tests systèmes
- #unitaires > #intégrés > #systèmes
- Comme une pyramide!

<https://martinfowler.com/articles/practical-test-pyramid.html>



Pourquoi la pyramide de tests?

- On a quatre fonctions
 - A possède $a = 5$ cas possibles
 - B possède $b = 7$ cas possibles
 - C possède $c = 3$ cas possibles
 - D qui appelle A, B et C. Ainsi, D possède $a * b * c$ cas possibles
- Si on veut tester tous les cas de D de façon intégrée
 - il faudra $a * b * c = 5 * 7 * 3 = 105$ tests
- Si on veut tester tous les cas de A, B, C de façon unitaire
 - il faudra $a + b + c = 5 + 7 + 3 = 15$ tests
- Il est donc avantageux de tester unitairement pour couvrir le plus de cas possible

Integrated Tests Are a Scam, J.B. Rainsberger
<https://www.youtube.com/watch?v=VDfx44fzoMc>

Pourquoi la pyramide de tests?

- Les tests intégrés et systèmes ratissent large
 - S'ils échouent, le problème peut être n'importe où dans le code appelé
 - Plus difficile à déboguer qu'un test unitaire
 - Un test unitaire est plus précis et facile à déboguer
- Abus de tests intégrés
 - Fausse impression de sécurité
 - Tests fragiles
- **MAIS**, les tests intégrés permettent de trouver des problèmes qu'il serait impossible de trouver avec seulement des tests unitaires
- Les tests intégrés et de système peuvent apporter de la valeur

Integrated Tests Are a Scam, J.B. Rainsberger
<https://www.youtube.com/watch?v=VDfx44fzoMc>

4. Tests d'acceptation

- Testent le logiciel depuis la perspective de l'utilisateur
- Valident que le logiciel fait ce que le client demande
- Scénarios d'utilisation
 - Structure given-when-then
- Outil
 - Cucumber (IntelliJ offre une intégration avec Cucumber)

<https://www.agilealliance.org/glossary/acceptance>

<https://martinfowler.com/articles/practical-test-pyramid.html#acceptance>

Structure d'un test d'acceptation

- Fichier de feature qui décrit les scénarios de tests avec gherking, un langage dédié aux tests (*domain-specific language* de test)
- Scénario de test
 - Given: énonce les préconditions
 - When: l'action testée
 - Then: les assertions
- Code source

```
Feature: tournament feature

Scenario: Create a tournament
  Given the following participants
    | name   |
    | Alice  |
    | Bob    |
    | Caroline |

  When the user creates the tournament named Smash
  Then the tournament exists

Scenario: Play a bracket
  Given the following participants
    | name   |
    | Dylan  |
    | Elliott |
    | Frank  |
    | Xavier |

  Given a tournament named Rocket League
  When the user plays the first playable bracket
  Then that bracket is played

Scenario: Can delete tournament
  Given the following participants
    | name   |
    | Tommy  |
    | William |
    | Xavier |
    | Yan    |

  Given a tournament named Smash
  When the user deletes the tournament
  Then the tournament does not exist
```

Code d'intégration: *glue code*

```
@Given("^the following participants$")
public void givenFollowingParticipants(final List<ParticipantDto> participantDtos) {
    this.participantDtos = participantDtos;
}

@Given("^a tournament named (.+)$")
public void givenTournament(String tournamentName) {
    tournamentId = createTournamentGetId(tournamentName, participantDtos);
}

@When("^the user deletes the tournament$")
public void whenDeleteTournament() {
    deleteTournament(tournamentId);
}

@Then("^the tournament does not exist$")
public void thenTournamentDoesNotExist() {
    getTournament(tournamentId).then().statusCode(404);
}
```

- Code qui vient adapter les énoncés gherking à votre application
- Utilisation d'expressions régulières pour le nom des étapes
- Possibilité de captures (.+)
- [Code source](#)

Intégration des tests d'acceptation avec JUnit5

➤ Code source

```
@RunWith(Cucumber.class)
@CucumberOptions(plugin = {"pretty"}, features = "src/test/resources/features")
public class TournamentCucumberTest {
}
```

Bonus: Tests par propriétés (*Property-based Testing*)

- Pose des propriétés que le code doit respecter
 - Ex. Pour toute entrée, la sortie doit être positive
- Fonctionnement
 - La bibliothèque génère des cas de test aléatoirement pour tenter de faire échouer le test
 - Valide que la propriété est respectée
 - Si la propriété n'est pas respectée, retourne un cas simple pour aider le débogage
- Popularisé par QuickCheck, une bibliothèque de tests écrite en Haskell
- Il existe aussi des implémentations de QuickCheck en Java:
 - <https://jwwik.net/>
 - <https://github.com/pholser/junit-quickcheck>
 - etc.

[QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs](#), K. Claessen, J. Hughes

Exemple de *Property-based Testing*

- Dans le premier test, on valide que la valeur absolue d'un nombre est positive
- Dans le deuxième, on valide que la concaténation de deux Strings est de longueur plus grande que chacune des Strings d'entrée
- Il ne semble pas y avoir de problème avec le code
- Que dit jqwik?

```
import net.jqwik.api.*;
import org.assertj.core.api.*;

class PropertyBasedTests {

    @Property
    boolean absoluteValueOfAllNumbersIsPositive(@ForAll int anInteger) {
        return Math.abs(anInteger) >= 0;
    }

    @Property
    void lengthOfConcatenatedStringIsGreaterThanLengthOfEach(
        @ForAll String string1, @ForAll String string2
    ) {
        String conc = string1 + string2;
        Assertions.assertThat(conc.length()).isGreaterThan(string1.length());
        Assertions.assertThat(conc.length()).isGreaterThan(string2.length());
    }
}
```

Exemple tiré de <https://jqwik.net/docs/current/user-guide.html#writing-properties>

Résultats — absoluteValueOfAllNumbersIsPositive

- $\text{Math.abs}(-2147483648) = 2147483648$
- Trop grand pour un int, alors overflow à -2147483648, donne un nombre négatif

```
|-----jqwik-----  
tries = 10           | # of calls to property  
checks = 10          | # of not rejected calls  
generation = RANDOMIZED | parameters are randomly generated  
after-failure = PREVIOUS_SEED | use the previous seed  
when-fixed-seed = ALLOW      | fixing the random seed is allowed  
edge-cases#mode = MIXIN       | edge cases are mixed in  
edge-cases#total = 9          | # of all combined edge cases  
edge-cases#tried = 1           | # of edge cases tried in current run  
seed = 2348343322668266667 | random seed to reproduce generated values  
  
Sample  
-----  
arg0: -2147483648
```

Résultats — lengthOfConcatenatedStringIsGreaterThanLengthOfEach

- ## ➤ On a oublié le cas de Strings vides

Tests par propriétés

- Permet de tester des cas que l'on aurait oublié de tester
 - Limites des types numériques
 - Chaînes de caractères vides
 - Gestion de l'UTF-8
 - Listes vides
 - Etc.
- Permet de rendre le code plus robuste
- Similaire au Fuzzing
- Peut être difficile de poser de bonnes propriétés
 - Il faut éviter de refaire l'algorithme dans la propriété