JUnit Patterns

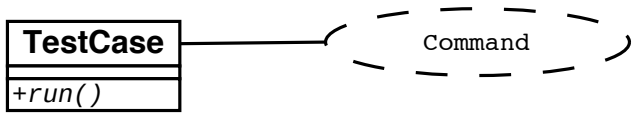
Le cadre d'applications JUnit en quelques patrons de conception

Gerson Sunyé
gerson.sunye@univ-nantes.fr

LS2N – Université de Nantes

21 novembre 2018

TestCase implémente «Command»

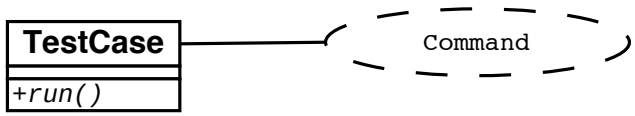



```
public abstract class TestCase implements Test {
  public abstract void run();
}
```

Avantages:

- Le concept de base est représenté par un objet (TestCase).
- Il est plus simple de créer des tests qui gardent leur résultat.
- Simplification de l'écriture de tests.

TestCase implémente «Command»

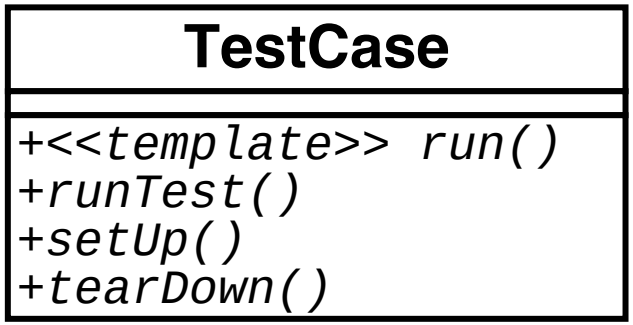



```
public abstract class TestCase implements Test {
  public abstract void run();
}
```

Le patron Commande encapsule un message en un objet et permet que les messages soient mis en enfilade ou enregistrés.

- Il créé un objet par opération et
- implémente la méthode /execute() pour chaque objet.

TestCase.run() implémente «Template Method»



Avantages:

- Fournit un lieu unique pour implémenter la partie fixe et la partie variable des tests.
- Fournit une structure commune à tous les tests:
  - Initialisation.
  - Exécution des tests.
  - Vérification des résultats.
  - Nettoyage.

Le patron «Template Method»

Une méthode Template définit le squelette d'un algorithme en laissant quelques étapes aux sous-classes. Elle laisse les sous-classes redéfinir certaines étapes d'un algorithme sans changer sa structure.

```
public void run() {
  setUp();
  runTest();
  tearDown();
}
```

Les crochets

Le comportement par défaut de ces méthodes est nul:

```
protected void runTest() { }
protected void setUp() { }
protected void tearDown() { }
```

Echecs et erreurs (1/3)

- Échec
  - Anticipe et vérifie les assertions (assert).
  - Signalé par une erreur *AssertionFailedError*
- Erreur
  - Problèmes non anticipés, comme *ArrayIndexOutOfBoundsException*

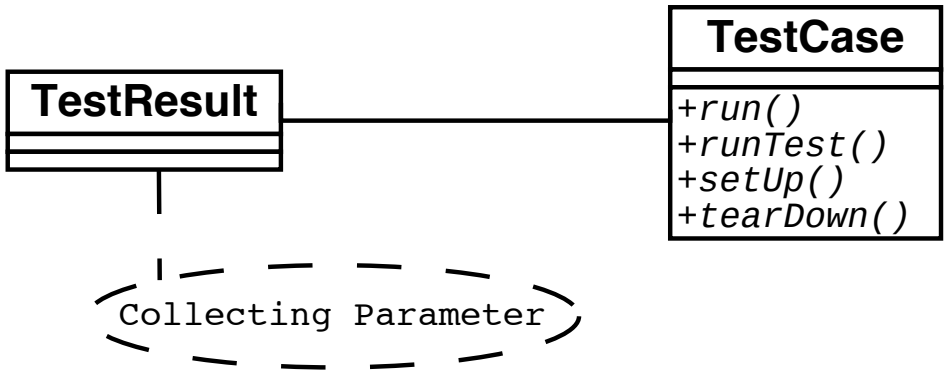
Echec et erreurs (2/3)

```
public void run(TestResult result) {
   result.startTest(this);
   setUp();
   try {
      runTest();
   } catch (AssertionFailedError e) {
      result.addFailure(this, e);
   } catch (Throwable e) {
      result.addError(this, e);
   } finally {
      tearDown();
   }
}
```

Collecte de résultats

- Après l'exécution d'un test, on a besoin d'un résumé de ce qui a et ce qui n'a pas marché.
- La forme canonique de la collecte de paramètres demande qu'un paramètre de collecte soit passé à chaque méthode.
  - Chaque méthode de test demanderait un paramètre pour la classe TestResult.
  - Une *pollution* de signatures de ces méthodes.
  - Il est possible d'utiliser le mécanisme d'exceptions pour éviter cette poluttion.

Le patron «Collecting Parameter»

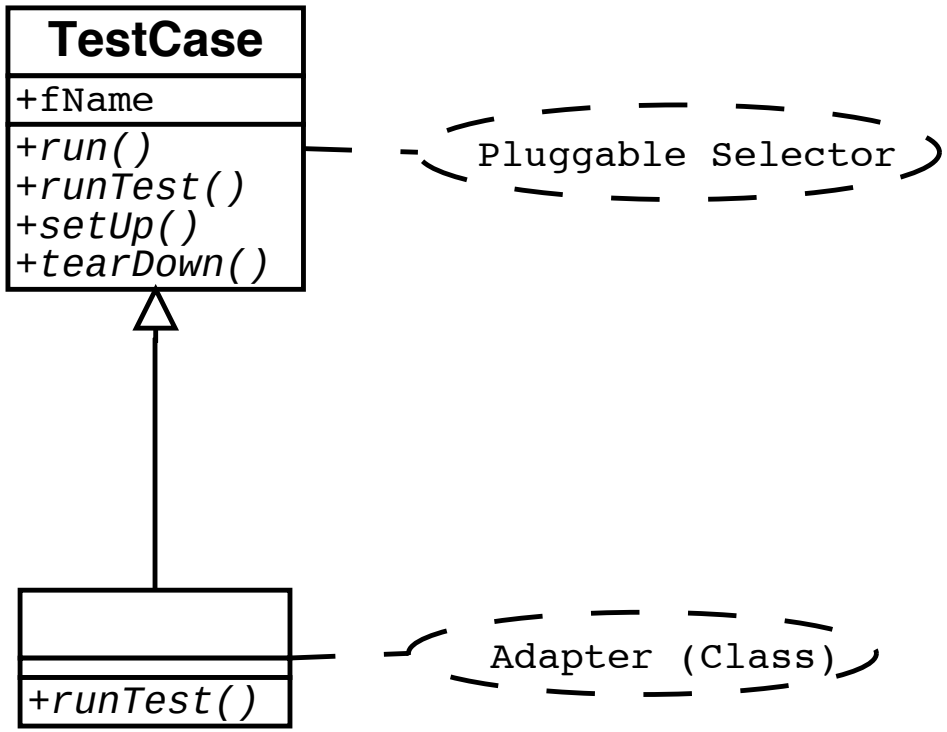


Patron «Collecting Parameter»

- Quand on a besoin de collecter les résultats de plusieurs méthodes, on peut ajouter un paramètre à la méthode et lui passer un objet qui collecte les résultats.
- TestResult : résultats des tests.

TestCase implémente ≪Adapter≫

TestCase implémente le patron ≪Adapter≫ en utilisant des classes anonymes.



Un cas de test == une méthode

- Pour éviter une prolifération inutile de classes
  - Les tests sont implémentés comme des différentes méthodes dans la même classe.
  - Une classe de test peut implémenter différentes méthodes, chacune définissant un cas de test.
  - Chaque cas de test a un nom descriptif, comme testMoneyEquals() ou testMoneyAdd().

Les méthodes de test

- Les cas de test ne se conforment pas à une interface unique du patron commande.
  - Les différentes instances de la même classe *Commande* doivent être appelées par des méthodes différentes.
- Le problème est de faire que tous les cas de test paraîssent similaires, du point de vue des clients.

Le patron «Adapter»

L'adaptateur convertit l'interface d'une classe en une interface attendue par les classes clientes.

La classe TestCase, utilise la spécification (i.e. héritage) pour adapter l'interface et implémente une sous-classe pour chaque cas de test.

Adaptation par spécification

```
// sous-classe explicite
public class TestMoneyEquals extends MoneyTest {
    public TestMoneyEquals() {
      super("testMoneyEquals"); }
    protected void runTest () { testMoneyEquals(); }}

//classe anonyme
  TestCase test= new MoneyTest("testMoneyEquals ") {
    protected void runTest() { testMoneyEquals(); }};
```

TestCase implémente «Pluggable Selector»

Un PluggableSelector utilise une seule classe, qui peut être personnalisée pour exécuter des logiques différentes, sans la création de sous classes.

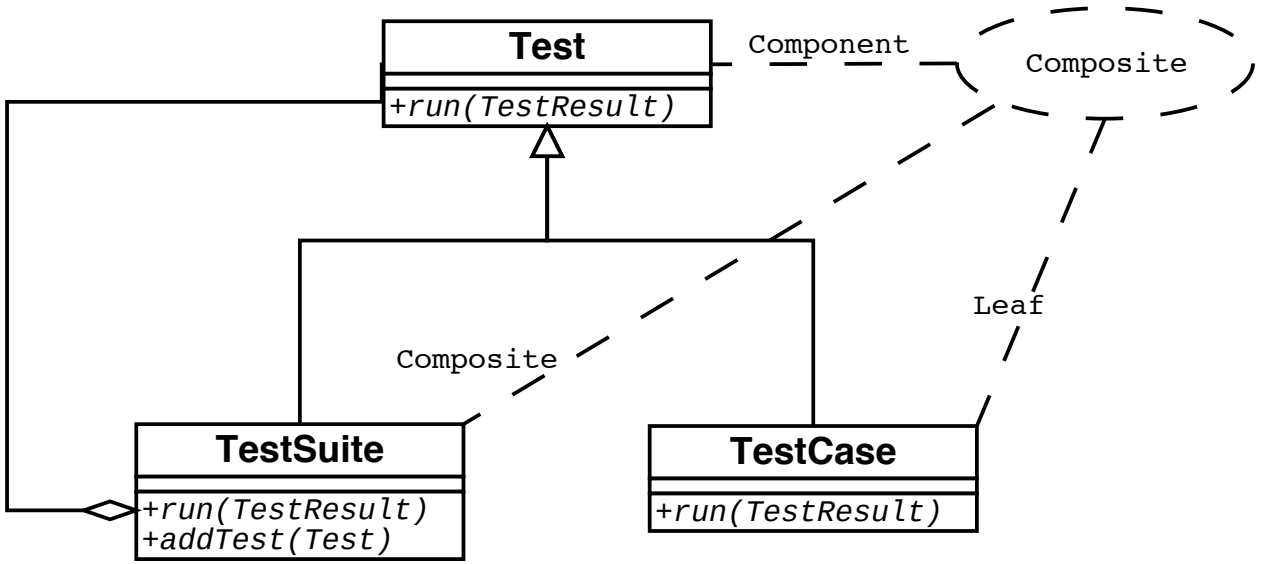
- Il stocke un sélecteur de méthodes dans une variable d'instance.
- Le mécanisme de réflexion de Java permet l'appel d'une méthode à partir d'une chaîne contenant son nom.
- Le Pluggable Selector est l'implémentation de la méthode runTest().

«Pluggable Selector»

```
protected void runTest() throws Throwable {
  Method runMethod= null;
  try {
    runMethod= getClass().getMethod(fName, new Class[0]);
  } catch (NoSuchMethodException e) {
    assert("Method \""+fName+"\" not found", false);
    }
  try {
    runMethod.invoke(this, new Class[0]);
  } catch (InvocationTargetException and IllegalAccessException ) {}
}
```

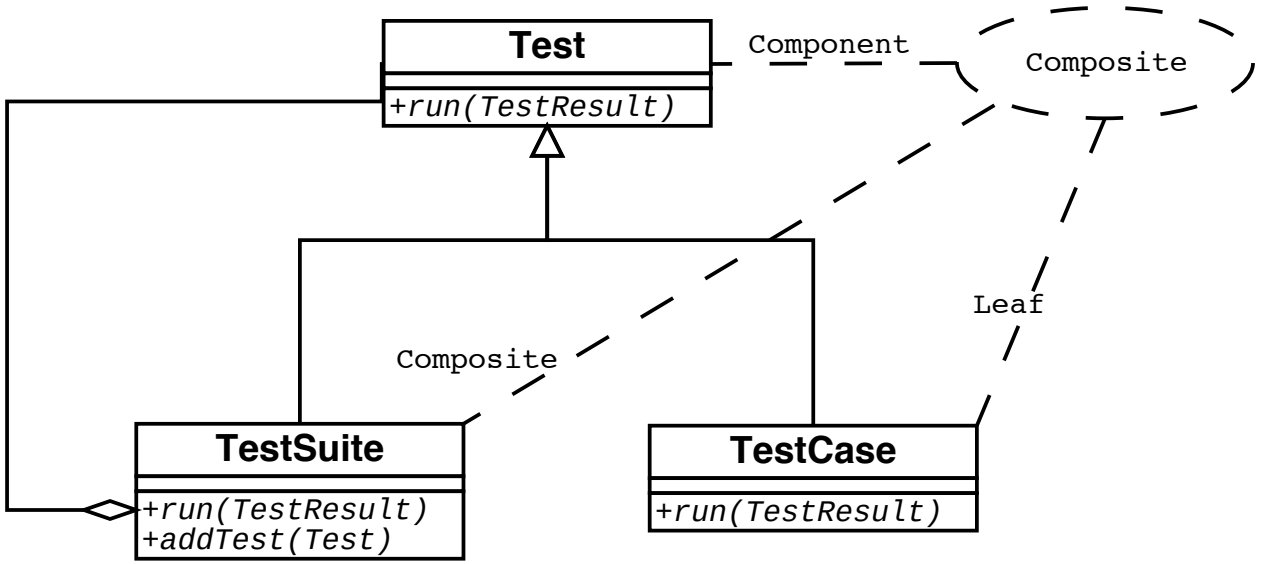
TestSuite implémente «Composite»

- Pour être certain de l'état du système, il faut exécuter plusieurs tests.
- Il faut implémenter des suites de cas de tests.



Le patron «Composite»

Le composite permet que les clients traitent des objets simples et composés de manière uniforme.



Composants, composites et feuilles

Participants:

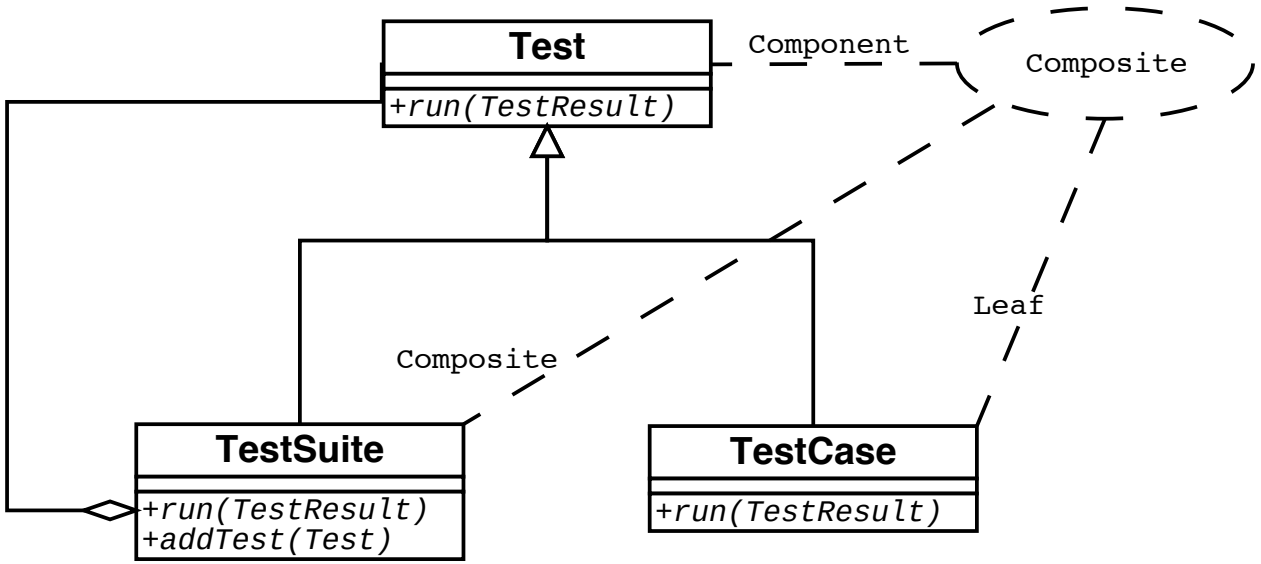
Composant : déclare l'interface qui sera utilisée pour interagir avec les tests: Test.

Composite : implémente cette interface et stocke une collection de tests: TestSuite.

Feuille : représente un cas de test, dans une composition conforme à l'interface Composant: TestCase.

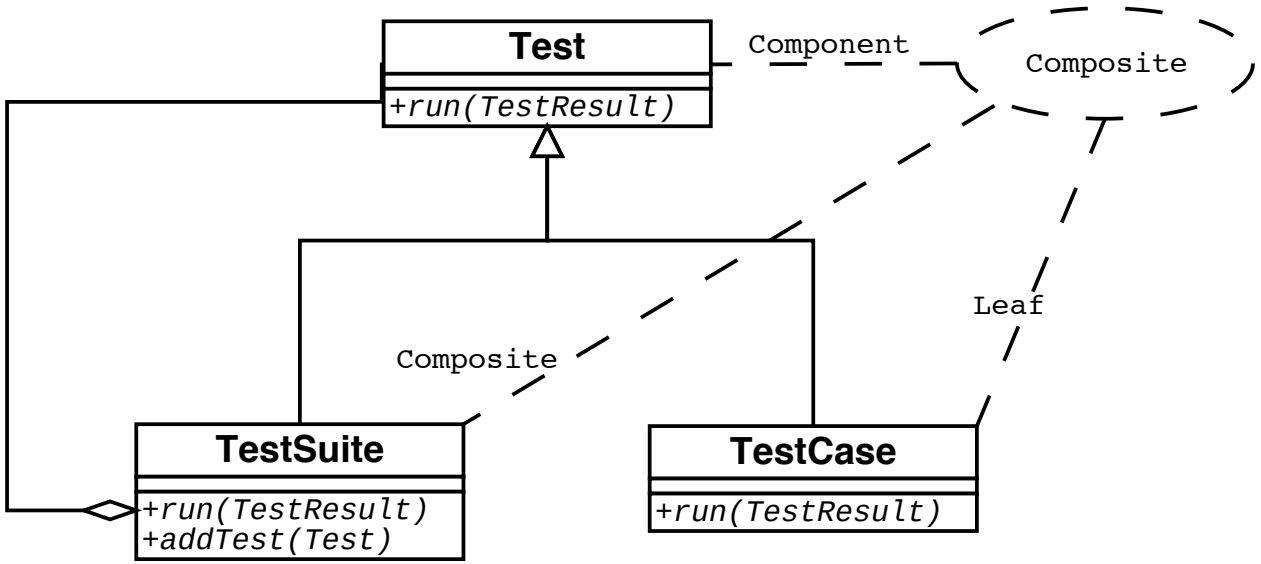
Composant

- Composant: déclare l'interface qui sera utilisée pour interagir avec les tests:
  public interface Test **public abstract void** run(TestResult result);



Composite

- Composite: implémente cette interface et stocke une collection de tests.



Composite

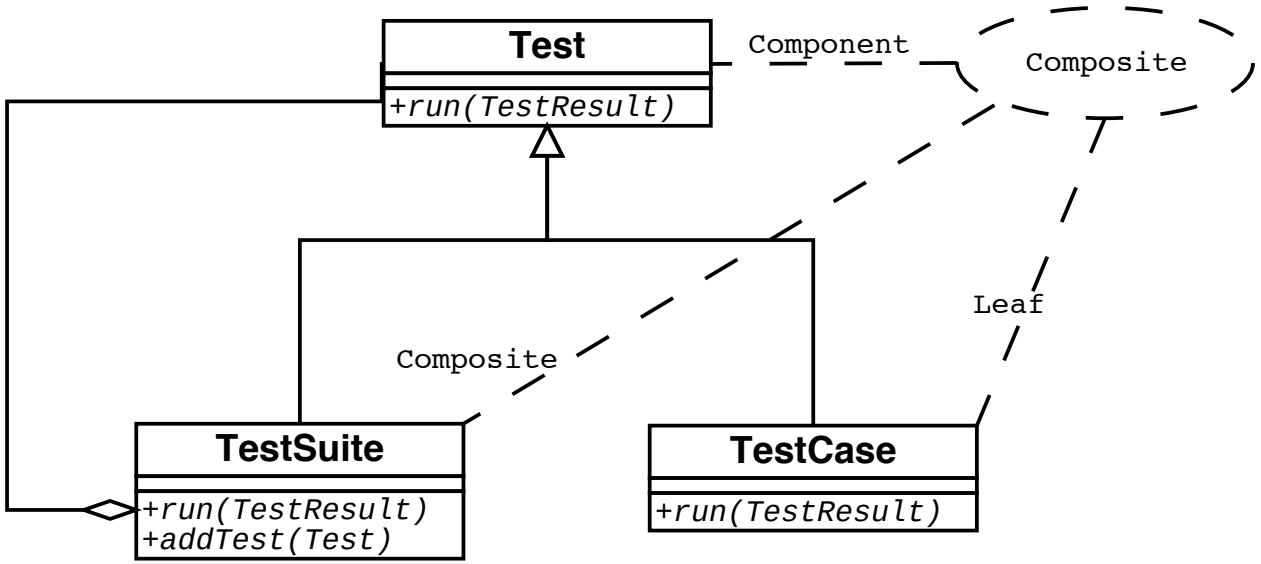
```
public class TestSuite implements Test {
  private Vector fTests= new Vector(); //children

  // delegate to children
  public void run(TestResult result ) {
    for (Enumeration e= fTests.elements(); e.hasMoreElements(); ){
      Test test= (Test)e.nextElement();
      test.run( result ); }}

  // add test to a test suite
    public void addTest(Test test ) {
      fTests.addElement(test);}}
```

Feuille

- Feuille: représente un cas de test, dans une composition conforme à l'interface Composant.



Feuille

```
public abstract class TestCase implements Test {
   public abstract void run();
}
```

Création statique d'une suite de tests

Spécification d'une suite de tests:

```
public static Test suite () {
  TestSuite suite = new TestSuite();
  suite.addTest(new MoneyTest("testMoneyEquals"));
  suite.addTest(new MoneyTest("testSimpleAdd"));}
```

Création dynamique d'une suite de tests

Extraction dynamique des méthodes de test et création d'une suite qui les contient:

- Par convention, les méthodes commençant par "test" et ne contenant pas de paramètres.
- Construction des objets de tests utilisant la réflexion.

```
public static Test suite () {
  return new TestSuite(MoneyTest.class);}
```