Let's play

Université Nice Sophia Antipolis
POLYTECH NICE-SOPHIA
Membre de UNIVERSITÉ CÔTE D'AZUR

Etude de cas : un Junit sans annotations, Junit 3

- Des cas de test dans une classe
  - Des méthodes spécifiques (setup, testXXX, teardown)
  - Création de suite de tests, faites de tests ou d'autres suites
  - Exécution des tests selon la sémantique setup/test/teardown + récupération d'exceptions
  - Regroupement des résultats

Boite à outils

- Singleton
- Factory method
- Abstract Factory
- Builder
- Decorator
- Proxy
- Observer
- Command
- Template Method
- State
- Strategy
- Adapter
- Facade
- Composite

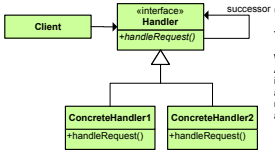
- **Composition**
- **Héritage**

- **Par ou commencer ?**

https://lms.univ-cotedazur.fr/pluginfile.php/5244/course/section/35573/designpatternscard.pdf

C Abstract Factory
S Adapter
S Bridge
C Builder
B Chain of Responsibility
B Command
S Composite
S Decorator
S Facade
C Factory Method
S Flyweight
B Interpreter
B Iterator
B Mediator
B Memento
C Prototype
S Proxy
B Observer
C Singleton
B State
B Strategy
B Template Method
B Visitor

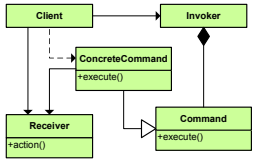
Chain of Responsibility



Type: Behavioral

What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

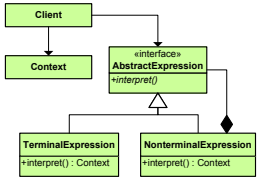
Command



Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

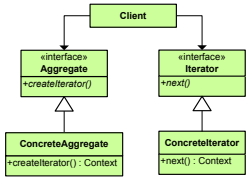
Interpreter



Type: Behavioral

What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

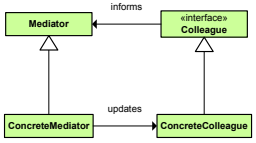
Iterator



Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator

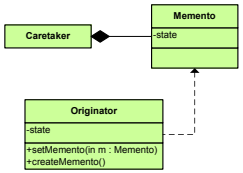


Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.

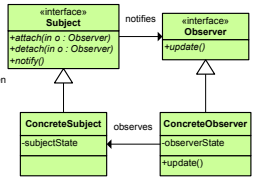
Memento



Type: Behavioral

What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

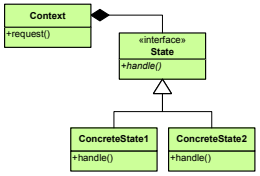
Observer



Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

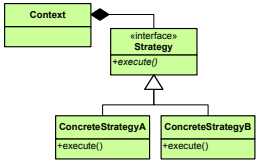
State



Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

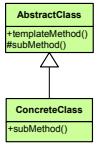
Strategy



Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.

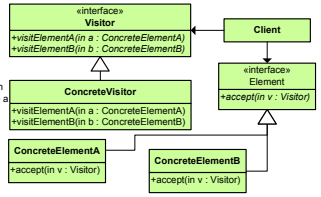
Template Method



Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

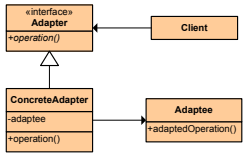
Visitor



Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

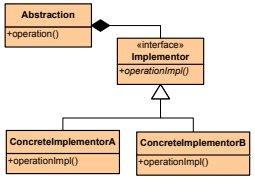
Adapter



Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

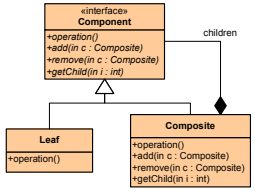
Bridge



Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

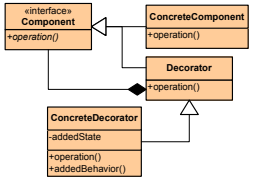
Composite



Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

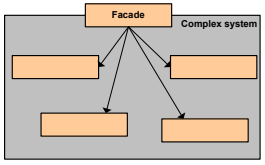
Decorator



Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

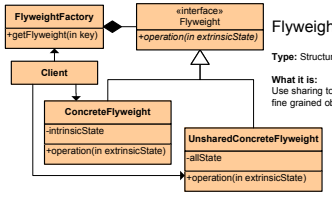
Facade



Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

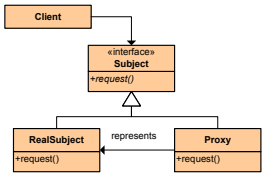
Flyweight



Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

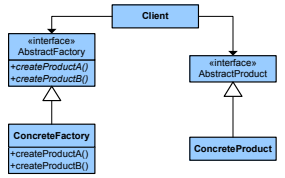
Proxy



Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.

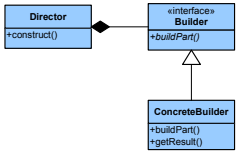
Abstract Factory



Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.

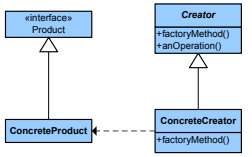
Builder



Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.

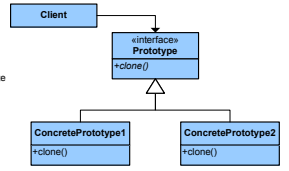
Factory Method



Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

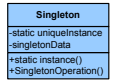
Prototype



Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton



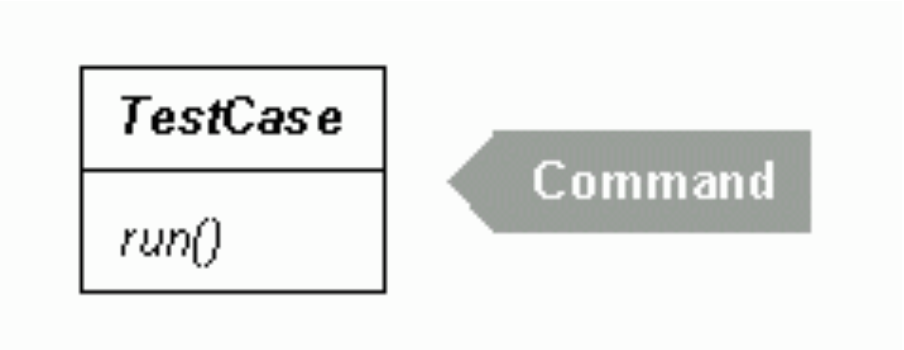
Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.

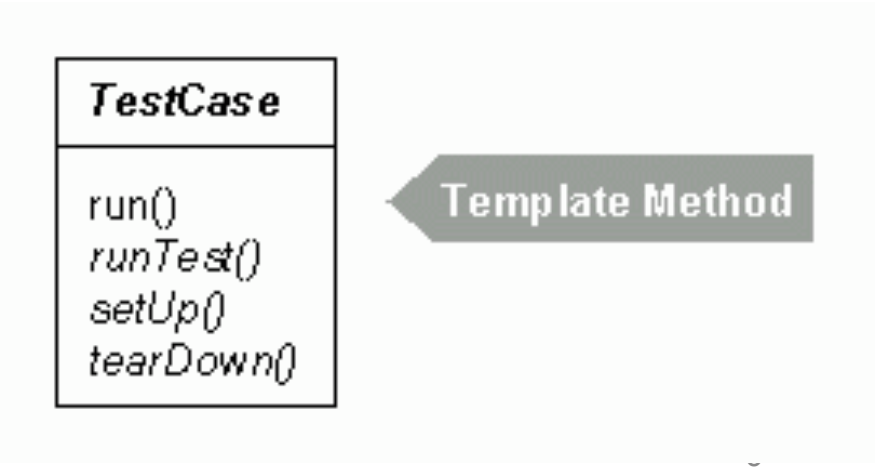
Concept de base : un cas de test

```
public abstract class TestCase  {
   …

   private final String fName;    public TestCase(String name) {
      fName= name;
   }


   public abstract void run();
      …
}
```



Ou placer le code du développeur ?

```
 void run();

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


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



Récupérer les résultats ?

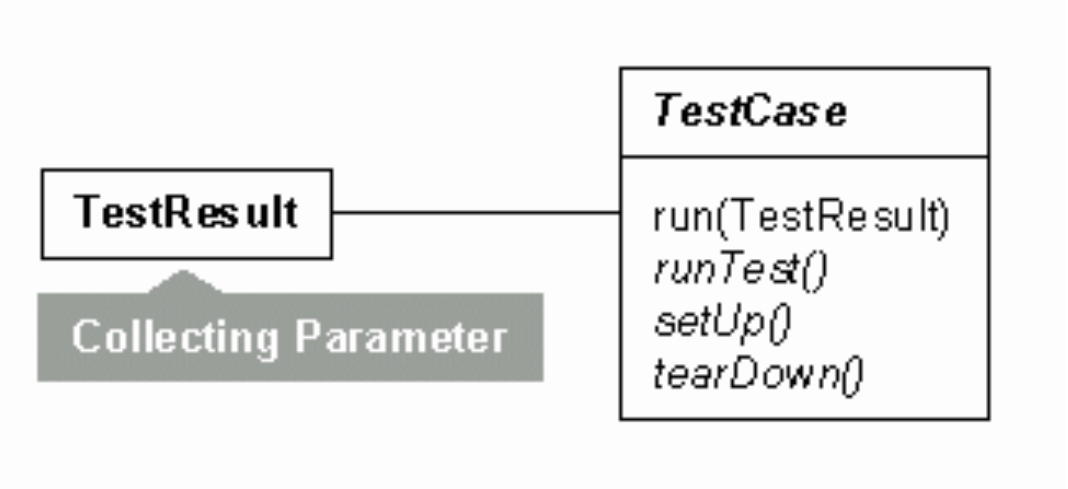
- Collecting Parameter (du langage Smalltalk)
  - *It suggests that when you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you.*

```
public class TestResult  {

protected int fRunTests;

public TestResult() {
      fRunTests= 0;
   }
}
```

Résultats (suite)

TestCase

```
public void run(TestResult result) {
   result.startTest(this);
   setUp();
   runTest();
   tearDown();
}
```



TestResult

```
public synchronized void startTest(Test test) {
   fRunTests++;
}
```

Et une méthode run() qui crée son prope TestResult

Failure / Errors ?

- assertTrue ?

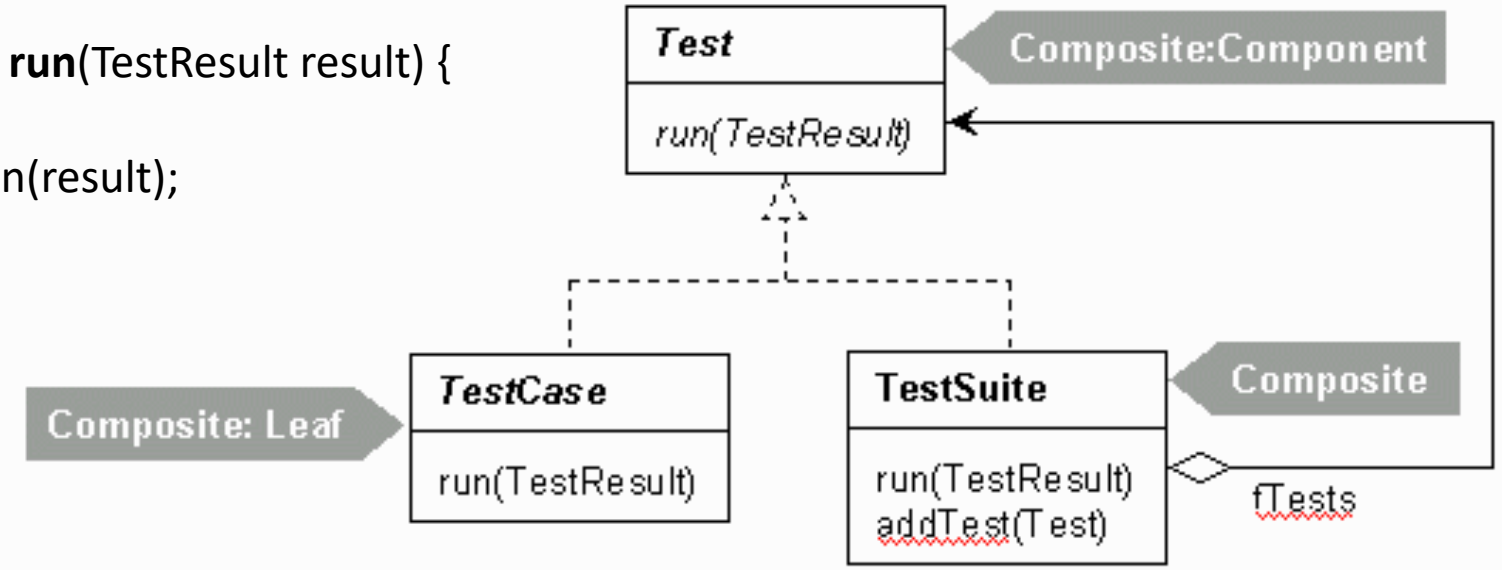
```
protected void assertTrue(boolean condition) {
   if (!condition)
      throw new AssertionFailedError();
}
```

```
public void run(TestResult result) {
   result.startTest(this);
   setUp();
   try {
      runTest();
   }
   catch (AssertionFailedError e) { //1
      result.addFailure(this, e);
   }   catch (Throwable e) { // 2
      result.addError(this, e);
   }
   finally {
      tearDown();
   }
}
```

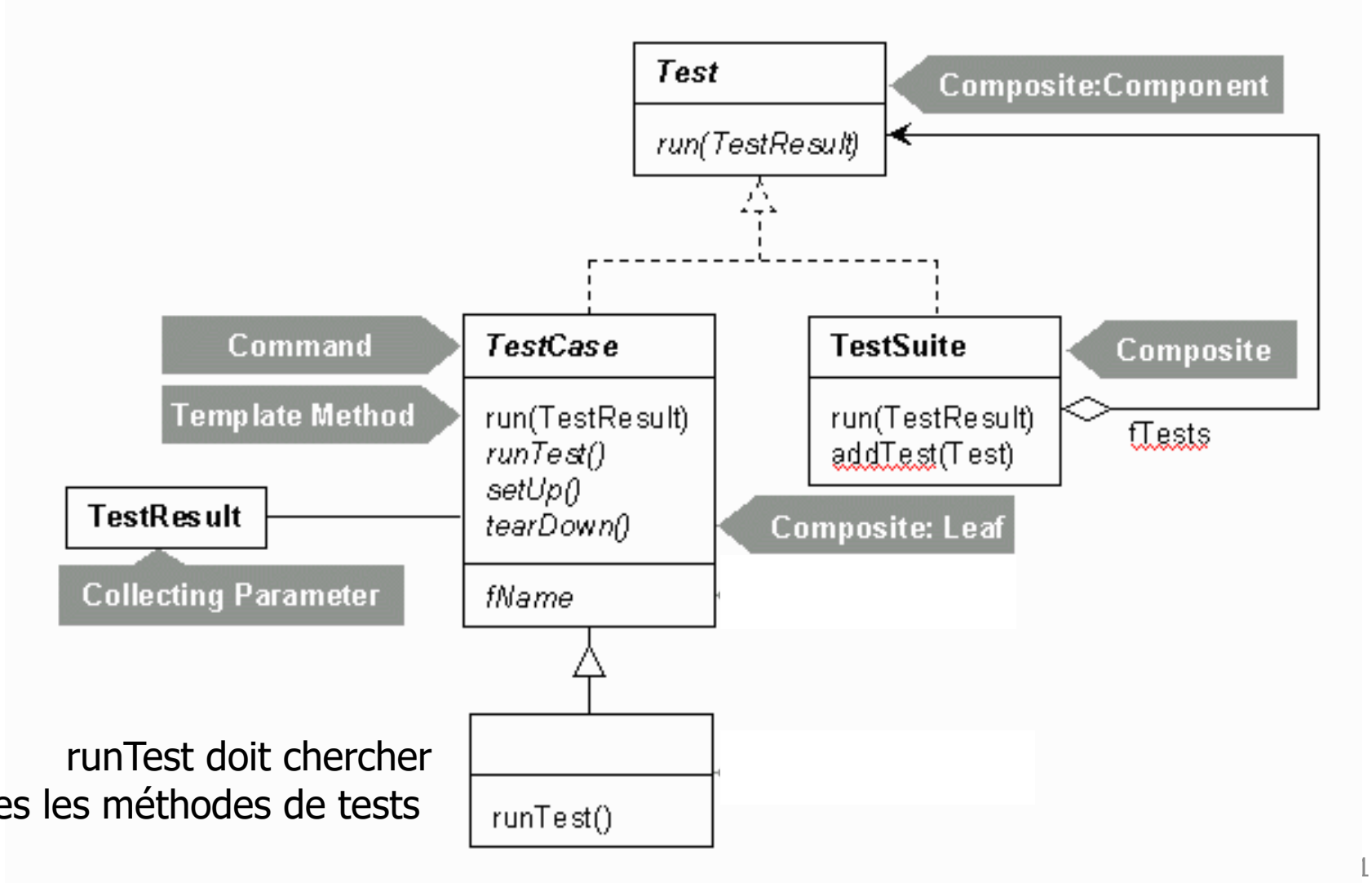
TestResult est un point d'extensions

Suite de tests ?

```
public interface Test {
   public abstract void run(TestResult result);
}

public class TestSuite implements Test {
   // list of Test
}

public void run(TestResult result) {
   for (…) {
      test.run(result);
   }
}
```



Architecture Junit v3



runTest doit chercher toutes les méthodes de tests