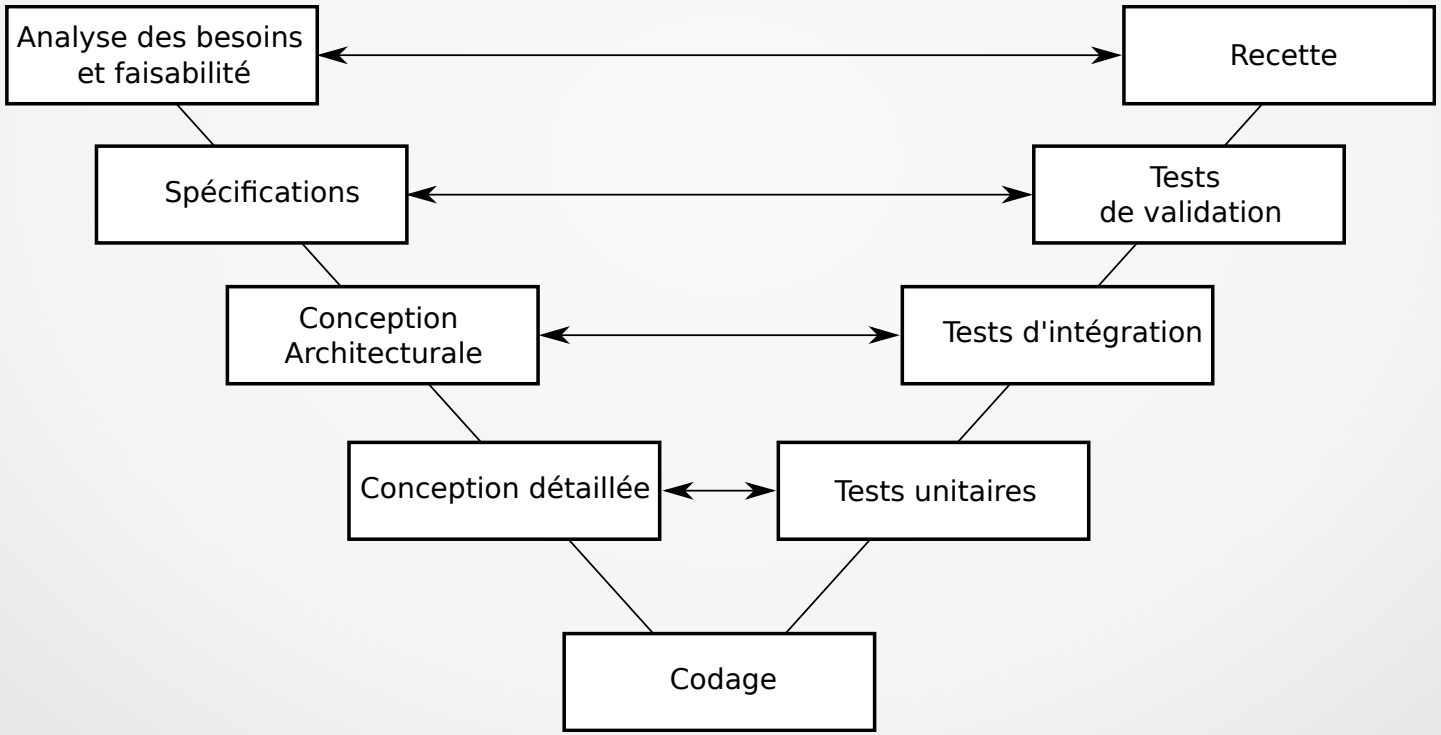
Méthodes et bonnes pratiques

Conception : Cycle en V

- Méthode définie et utilisée depuis les annés 1980
- On définit les besoins, et on valide selon le schéma suivant



« ?Cycle de developpement en v ?» par Original by fr:Christophe.moustier. — Wikipédia

Lourdeurs

- Rédaction du cahier des charges !
- Rédaction de la recette
- Rédactions rédactions..
- Pas de flexibilité
- Résultats conformes mais utilité ?
-

Méthodes AGILE

- Idées
  - racourcir le cycle,
  - supprimer les intermédiaires
  - Être beaucoup réactif
  - Adapter le résultat aux besoins
  - Évolution de la demande
- Manifeste AGILE (Agile manifesto 2001)
- Principes mis en œuvre dans :
  - SCRUM
  - Extrem programming (XP)
  - Test Driven Development (TDD)

Agilemanifesto.org

Manifeste pour le développement Agile de logiciels

Nous découvrons comment mieux développer des logiciels
par la pratique et en aidant les autres à le faire.
Ces expériences nous ont amenés à valoriser :

Les individus et leurs interactions plus que les processus et les outils
Des logiciels opérationnels plus qu'une documentation exhaustive
La collaboration avec les clients plus que la négociation contractuelle
L'adaptation au changement plus que le suivi d'un plan

Nous reconnaissons la valeur des seconds éléments,
mais privilégions les premiers.

1er exemple : méthode SCRUM

- Méthode agile
- Pas de cahier des charges, car pas de projection réaliste
- Trop complexe, besoin de réactivité, d'interaction avec le « client »
- Obtenir un produit final le plus adapté aux besoins…
- Qui peuvent évoluer en cours de réalisation

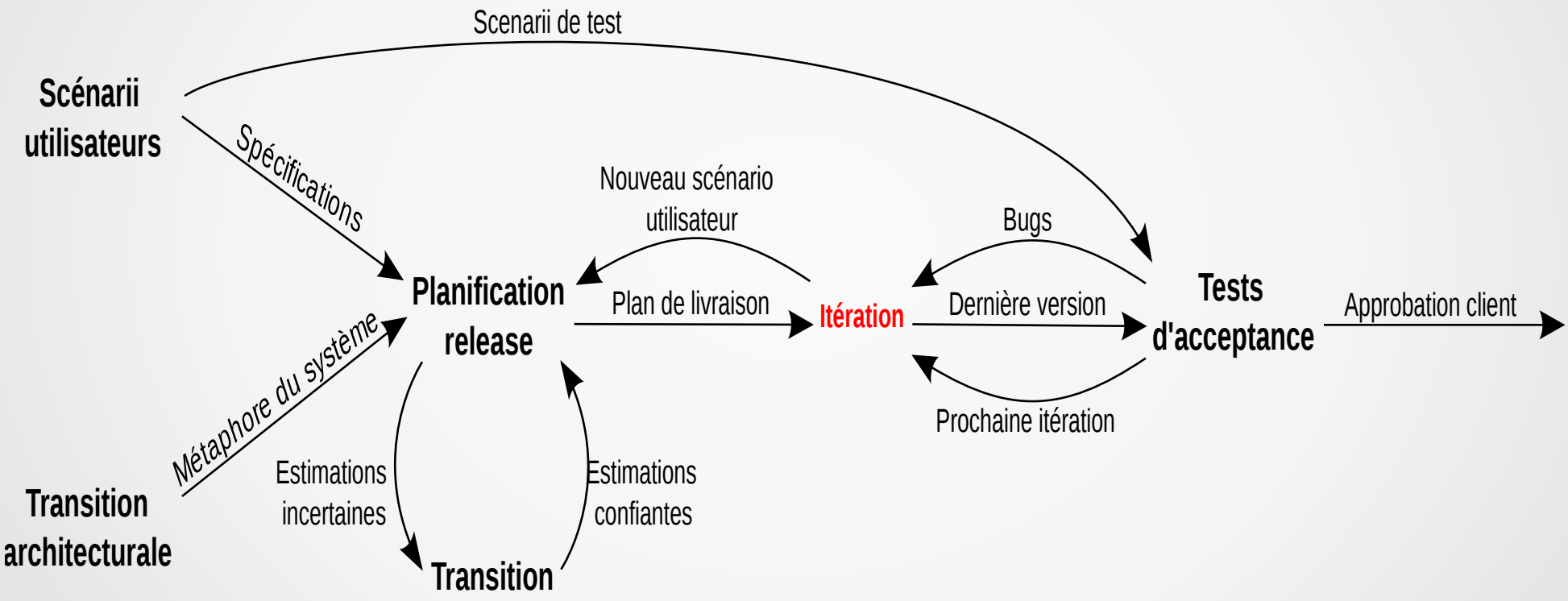
SCRUM

- **Idée** : boucler très rapidement entre des spécifications de besoins et l'implémentation de la solution
- **Sprint** : Le client exprime un besoin. Les développeurs entrent en « **Mêlée** » pour le résoudre au plus vite. Au final du sprint, le client évalue le résultat. Il affine la demande, la mêlée sprinte pour le résoudre, et ainsi de suite
- **Itérations très rapides** : cycle d'une semaine, avec réunion/validation/expression du nouveau besoin
- Engagement sur une durée, et pas automatiquement sur un résultat (obligation de moyens / obligation de résultats ?)

eXtreme Programming

- Programmation en binome
  - Un driver qui code
  - Un partner qui contrôle, conseille, teste, critique
  - On change régulièrement l'équipe (pas de *gouroutisation* du code)
- créer les tests directement lors de la création de chacune des classes.
- Client sur site, qui regarde, lance les tests, spécificie, valide
- Intégration continue
  - toute avancée validée est immédiatement intégrée à la version courante)

Extreme Programming XP



« Extreme programming » par KrapFr based on original work of Traroth - Wikipédia

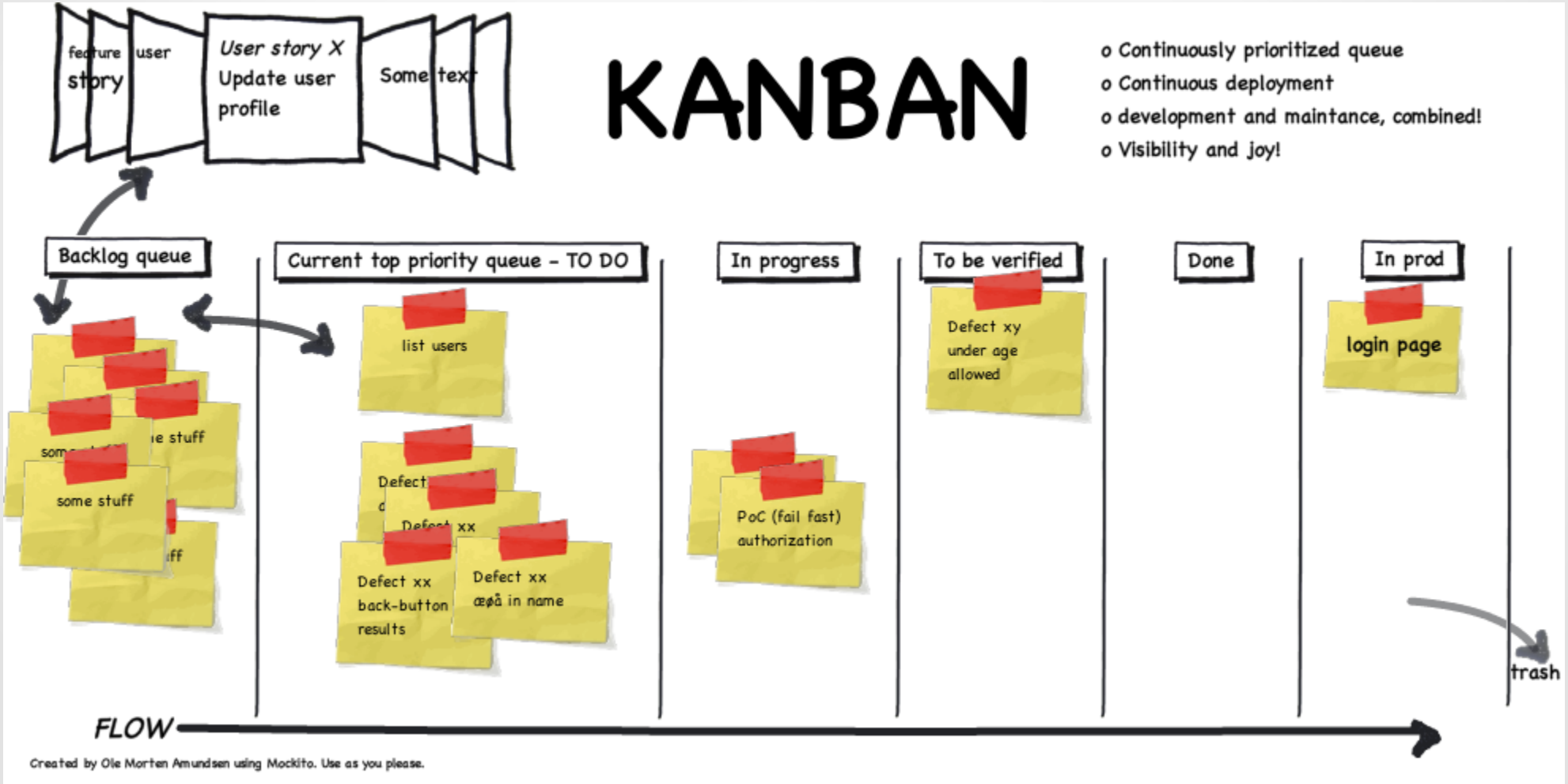
TDD

Le TDD (*Test Driven Development,* développement dirige par les tests) :

- Consiste a découvrir les fonctionnalités que doivent implémenter chacune des classes lors de l'écriture des tests
- Résoudre les problèmes au fur et à mesure de la découverte de la complexité du problème
- Être capable d'éviter les régressions
- Garantir la qualité et la constance du résultat

KANBAN

- Inspirée des pratiques chez Toyota



Les TESTS

Besoins de tests fiables

- L'ensemble des approches utilise des tests pour valider le travail effectué
- C'est la fréquence et l'ampleur du test qui varie selon les approches
- Même la recette est une forme de test de l'application à l'échelle réelle
- L'expérience montre qu'il y a des bugs qui passent à travers ces filtres.
- Des régressions apparaissent : lors de l'évolution des logiciels, la mise à jour entrainent des bugs sur d'anciennes fonctionnalités qui marchaient bien jusqu'àlors.

Junit et la programmation par tests

Idée directrice : Seuls les tests permettent de valider une application.

- Tests conçus trop tard dans le processus de création
- Ne couvrent pas l'ensemble des points de l'application
- Pas suffisamment automatisés pour être lancés fréquemment.

Problème des tests

- Les tests « *sauvages* » ne sont pas assez modulaires, couvrent peu de caractéristiques et ciblent toute l'application.

- plus un bogue est découvert tôt, plus rapidement il sera corrigé
- un bogue découvert en fin de développement avant la mise en production coûtera 10 fois plus en temps à corriger
- 100 fois plus si il est découvert lors de l'utilisation de l'applicatif)

Comment réaliser ces tests ?

- En quel langage écrire les tests ?
- Sous quel format seront les résultats ?

Enfin, des tests « *sauvages* » n'offrent pas les garanties suffisantes pour :

- Garantir une bonne couverture du code (*bug dans une fonction)*
- Garantir un bonne couverture des cas rencontrés (*tout est-il traité?*)
- Permettre une refonte complète de l'application (*refactoring*)

Tests

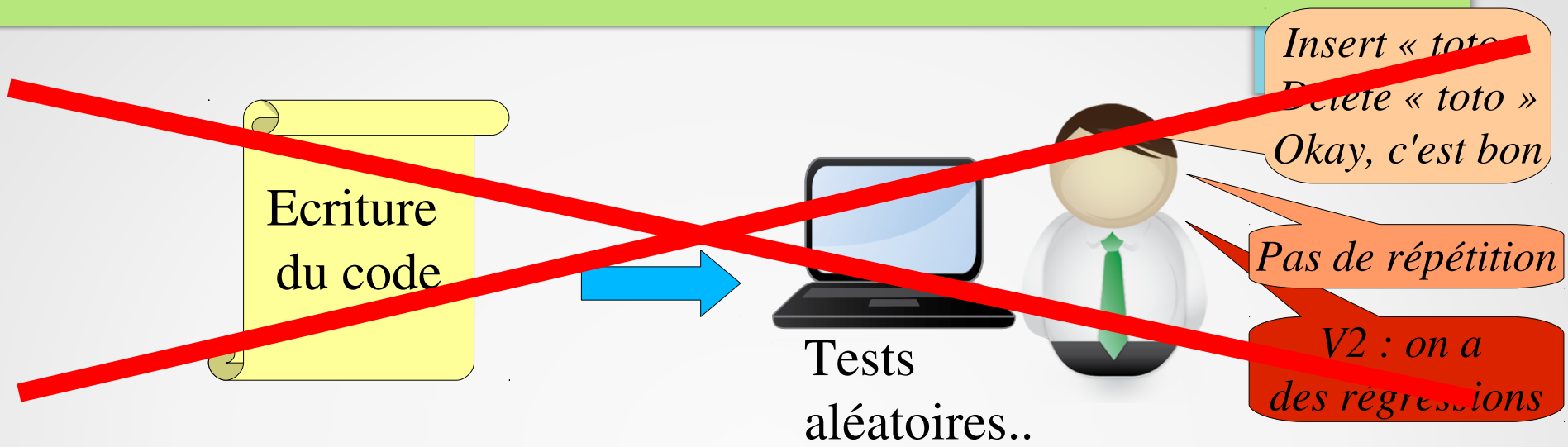
Dans un souci de qualité et d'efficacité :

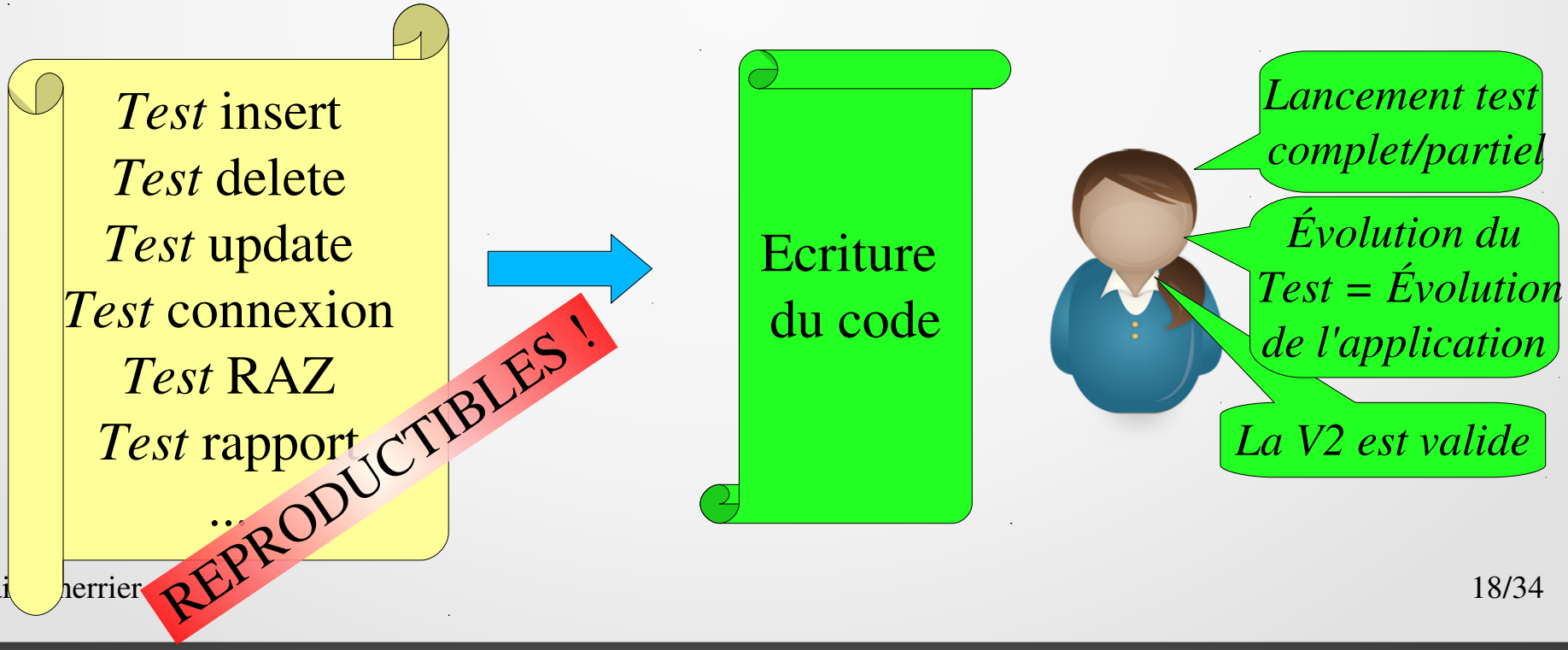
•créer les tests directement lors de la création de chacune des classes.

Technique du test-first

•conception des classes de tests en premier
•exécution pour bien vérifier qu'elles détectent une erreur (les méthodes ne font rien)
•Résolution des erreurs en écrivant les méthodes cibles
•Vérification que le test passe bien

Démarche Test Driven





Réalisme de la démarche

L'apparente perte de temps (*premier a-priori)* est très rapidement compensée par la qualité du travail réalisé, et par la liberté qu'offre cette assurance de pouvoir (re)valider son travail, même en cas de refonte complète.

Comme toute méthode, on pourra la prendre en défaut en ne jouant tout simplement pas le jeu (*ou en détournant son objectif, un peu comme ce que devenu le 'pas de goto', ou UDP mode non fiable*), mais pourtant :

Caractéristiques des Tests

TESTS EXHAUSTIFS :

- Toutes les méthodes sont elles testées ?
- les différents embranchements ?

Il existe des outils qui testent vos classes de tests (couverture, pertinence)

QUALITE DES TESTS :

Vos tests sont ils fiables ? Un outil prend vos classes testées, les modifie aléatoirement (*de très peu à beaucoup*) , et relance vos procédures de tests.

La moindre altération du comportement de vos classes doit être détectée par vos tests.

JUnit

Junit est un framework (un framework est un canevas applicatif, une structure prête a l'emploi dans laquelle on vient ajouter ses fonctionnalités. (*créé par E. Gamma, du GOF*)

A la différence d'un **IDE** qui aide a la réalisation d'application, ou d'un toolkit (*boite a outils*) qui offre un simple ensemble d'outils...). Il est écrit en Java, pour tester du Java (*cohérence du langage de test*)

Ce framework permet d'écrire facilement des classes de test, d'en stocker les résultats, et de capitaliser le plus possible votre travail.

JUnit

Ce Framework fonctionne directement (*si ce n'est qu'il ne teste rien*). Vous devez étendre ses capacités pour tester vos classes (héritage). Il peut être intégré dans votre IDE.
Junit est écrit en **Java**, et permet de tester vos classes **Java**. Il existe des framework pour d'autres langages (PHPUnit, cppunit, unit.js)

Junit est un logiciel libre : http://www.junit.org
Des informations sur l'**Extreme Programming** http://www.extremeprogramming.org/

Première classe de test

La première démarche consiste à tester vos classes... Il faut éviter de mélanger les classes soumises à une même batterie de tests (*du moins dans cette première partie*), et donc une classe de test permettra de tester une classe réelle

Ainsi, vos tests doivent être discriminants : vous devez pouvoir affirmer :

Telle méthode de telle classe fonctionne ou ne fonctionne pas (*et non pas : il y a une erreur quelque part*)

Première classe de test

Pour tester une classe, nous allons utiliser et spécialiser une classe offert par **Junit** : TestCase.

Il faudra implémenter les méthodes de test (**public void testXXX()** - le XXX est libre, le préfixe test obligatoire).
La super-classe vous offre des méthodes **assert()**, c'est à dire '*assurez vous que...*'...

C'est ce que le framework doit vérifier, ce qui doit être valide.

Première classe de test

Imaginons que nous voulons tester une classe **Employe**, qui offre les méthodes **getNom()** et **setNom(String)**.

Cette classe sera considérée comme *valide* si elle est capable de bien nous renvoyer un nom arbitrairement choisi.

TestEmploye

```
import junit.framework.*;

public class TestEmploye extends TestCase {
    /**Test de la classe Employe
     * premiere version */
    public void testNom() {
        /** test du getter et du setter du nom de l'employe */
        Employe e1=new Employe();
        e1.setNom("Vincent Massol");
        assertEquals("Vincent Massol",e1.getNom());
    }
}
```

Pour lancer la compilation, il faut indiquer le chemin du framework (et aussi le chemin local des classes, soit .)

javac -classpath /opt/junit/junit.jar:. TestEmploye.java

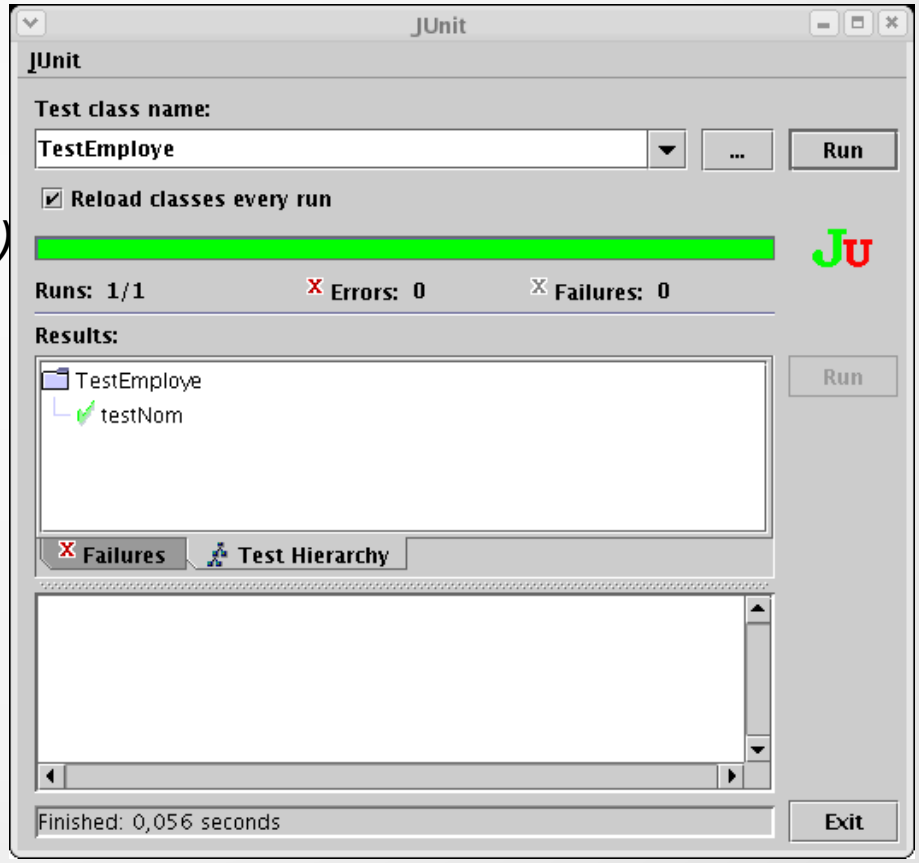
Pour l'exécution, on va appeler le framework (il y en a 3 : 1 en mode texte, et 2 en mode graphique (AWT et Swing)... Prenons Swing. On lance le programme TestRunner à qui on passe en argument notre classe de Test. Toutes ses methodes seront exécutées...

java -cp /opt/junit/junit.jar:. junit.swingui.TestRunner TestEmploye

Employe, la classe sous test

Bien sur, ici, nous obtenons une barre rouge : Échec ! La classe Employe n'existe pas (l'accroche de Junit est : *Keep the bar green to get the code clean***)**
Écrivons maintenant la classe Employe

```
public class Employe {
    String nom_employe;
    public void setNom(String nom)
        nom_employe=nom;
    }
    public String getNom() {
        return nom_employe;
    }
}
```



Après compilation, on relance le test....
Barre verte...

Suite du développement

On ajoute à notre Classe la notion Cadre/non-cadre : Une méthode *isCadre***() retourne un booléen selon le cas...**
Première proposition de code :

```
public void testCadre() {
    Employe e2=new Employe();
    e2.setCadre(true);
    assertTrue(e2.isCadre());
    e2.setCadre(false);
    assertFalse(e2.isCadre());
}
```

assertTrue(*boolean*) et **assertFalse**(*boolean*) sont deux des 8 méthodes offertes (avec **assertEquals**(), **assertNull**(), **assertNotNull**(), **assertSame**(), **assertNotSame**(), et **fail**())

Ici. on remarque du code dupliqué (*Déclaration et instanciation de* ***e2****, doublette avec le* ***e1*** *du test précédent*)
Pour **TestCase**, il existe une méthode **setUp()** qui permet de créer les conditions de tests avant le lancement des méthodes de tests (même d'une seule) (il existe aussi un finisseur, **tearDown()**)

Classe TestEmploye v2

```
import junit.framework.*;
public class TestEmploye extends TestCase {
    private Employe e1i,e2; //Définis ici, pour la visibilite

    public TestEmploye(String name) {    super(name);}//Juste pour nommer
    public void setUp() {
        e1=new Employe(“Vincent Massol”,true);
        e2=new Employe('Linux Torvalds',false);
    }
    public void testNom() {
        assertEquals(“Vincent Massol”,e1.getNom());
        assertEquals('Linus Torvalds',e2.getNom());
    }
    public void testCadre() {
        assertTrue(e1.isCadre());
        assertFalse(e2.isCadre());
    }
}
```

Après la mise à jour de la classe Employe (avec un constructeur qui accepte le nom et le booléen cadre, et la méthode isCadre()), la barre devrait être verte.

Un peu plus sur les Tests

Lorsque nous appelons la classe **junit.swingui.TestRunner** et que nous lui passons une classe de **Test** en argument, elle lance l'ensemble des méthodes de tests de cette classe…

Ainsi, on valide toutes les méthodes testées.

Il est en suite possible de définir des **SUITES**, c'est à dire un ensemble de tests qui portent sur certaines méthodes d'un ensemble de classes.

Un peu plus sur les Tests

Cela permettra de tester l'application a un plus haut niveau, plutôt qu'une analyse exhaustive d'une classe.
Si l'objet contient une méthode **public static Test suite**(), alors celle-ci sera invoquée (ATTENTION, C'EST DU STATIQUE)

```
public static Test suite() {
    TestSuite maSuite=new TestSuite("Test de l'appli");
    maSuite.addTest(new TestEmploye("testCadre"));
    return maSuite;
}
```

Junit et ANT

Un autre reproche possible : la multiplicité de classes, et la complexité de la compilation...
ANT va nous permettre de bien ranger nos classes, et d'automatiser les appels...
On pourra organiser de la façon suivante :
un répertoire src et un répertoire bin... et dans chacun d'eux, un sous répertoire java et un test
Ce rangement certes cohérent entraîne des commandes de compilation et de lancement de plus en plus complexe :
D ou le fichier build.xml suivant

```
<project name="Employes" default="test">
    <property name="src.dir" location="src" />
    <property name="src.java.dir" location="${src.dir}/java" />
    <property name="src.test.dir" location="${src.dir}/test" />

    <property name="target.dir" location="bin" />
    <property name="target.java.dir" location="${target.dir}/java" />
    <property name="target.test.dir" location="${target.dir}/test" />
```

[...] les variables étant définies, sur la page suivante, on définit les commandes (target)

Suite du build.xml

```
<target name="compile.java">
    <mkdir dir="${target.java.dir}" /> //fabrique le rep en cas de besoin
    <javac destdir="${target.java.dir}"> //ou range t on les binaires ?
        <src path="${src.java.dir}" /> //ou sont les .java ?
    </javac>
</target>

<target name="compile.test" depends="compile.java">
    <mkdir dir="${target.test.dir}" /> //fabrique le rep en cas de besoin
    <javac destdir="${target.test.dir}"> //ou range t on les binaires de test ?
        <src path="${src.test.dir}" /> //ou sont les .java de test ?
        <classpath>
            <pathelement location="${target.java.dir}" /> //ou sont les classes a tester ?
        </classpath>
    </javac>
</target>
```

[...}les cibles qui fabriquent les tests et l'application sont maintenant définies

Fin du build.xml

```
<target name="test" depends="compile.java">
    <junit printsummary="yes" fork="yes"> //fork differencie bien les tests
        <formatter type="plain" usefile="false" />  //sortie en mode texte
        <test name="TestEmploye" /> //la classe de test
        <classpath>
            <pathelement location="${target.java.dir}" /> //ou sont les classes a tester ?
            <pathelement location="${target.test.dir}" /> //ou sont les classes de tests ?
        </classpath>
    </junit>
</target>
</project>  //fin du build.xml
```

[...}et enfin, la cible qui lance les tests sur vos classes....