

A la chasse aux bugs : comment rendre l'informatique plus sûre (Extraits)

Gérard Berry

Collège de France

Chaire Algorithmes, machines et langages

Académie des sciences, Académie des technologies

gerard.berry@college-de-france.fr

<http://www.college-de-france.fr/site/gerard-berry/index.htm>

Montpellier, GDR

GPL, 14 juin

2017

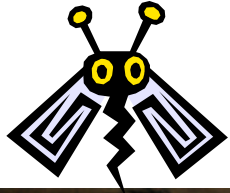
As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought.

Debugging had to be discovered.

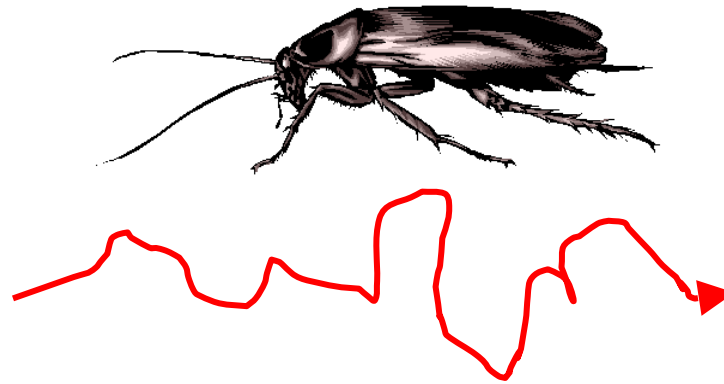
I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

Maurice Wilkes, 1949

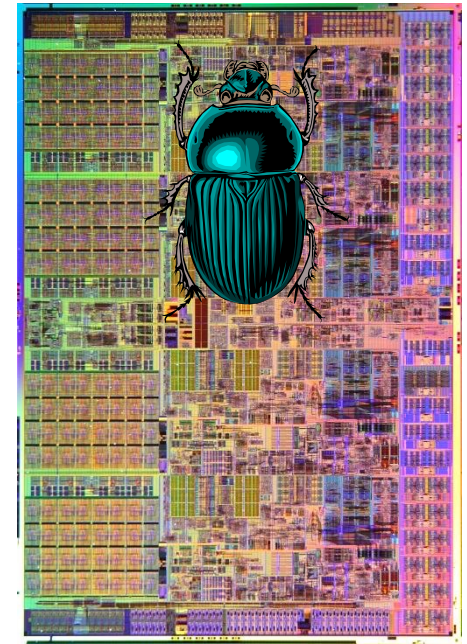
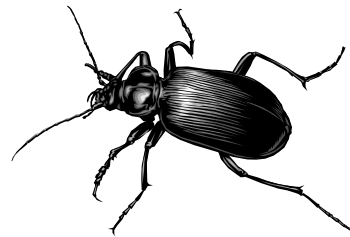
Homme / ordinateur, un gouffre à combler



Intuition
Rigueur
Lenteur



Maîtrise ?



Stupidité
Exactitude
Rapidité

L'ordinateur est le plus extraordinaire
amplificateur d'erreurs
jamais construit !

Comment nourrir les bugs

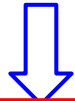
- Ne pas comprendre les spécificités de l'informatique
 - bugs vus comme des « pannes de logiciels » - FAUX !
 - logiciel vu comme « mécanique plus légère et plus souple »
- Employer des raisonnements valables ailleurs mais pas en informatique
 - redondance par simple duplication du système
 - calculs de « probabilités de bugs » - MEFIANCE
- Mal spécifier, mal documenter, mal maintenir
 - mauvaises méthodes de travail, mauvais outils
- Limiter les tests à la « marche normale »
 - les problèmes sont dans les coins et dans le non prévu
 - les trous de sécurité sont dans le non-fonctionnel

Les bugs sont des **pannes des hommes**
pas des pannes des machines !

De la conception à l'implémentation



analyse



spécification



programmation

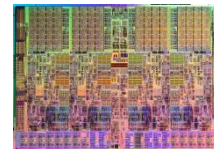
Nid à bugs



intégration



implémentation



Comment réduire les bugs

- Utiliser de bonnes méthodes de design
 - spécifications, langages et débogueurs à base formelle
 - caractérisant l'environnement autant que de l'application
- Rendre tout visible et explorable (→ traçable)
 - procédures de développement, spécification, code, documentation, tests et résultats de test, etc.
- Faire des revues indépendantes
 - processus de certification
 - communautés open source
- Tester systématiquement
 - les composants, leur intégration, la non-régression
 - l'application globale sur la cible réelle ou par simulation

21^e siècle : **vérifier formellement**
avec des outils automatiques ou semi-automatiques

Tests logiciels

Kent Beck « Une fonctionnalité sans test automatique n'existe tout simplement pas »

Penser que les tests [et le refactoring] ralentissent le développement
c'est comme penser que prendre des voyageurs ralentit le bus
David Evans

Merci à tous ceux qui ont rendu leurs cours et exposés
disponibles sur le web & dans les livres,
voir Biblio.

M. Blay-Fornarino
blay@unice.fr,
IUT Département Informatique

Votre vision du test ?

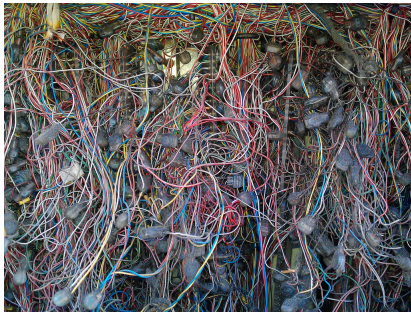




Fascinant



Pour les autres



Complexe, multi points de vue

Industrial Robotics Evolves Very Fast!

Industrial robots are now complex cyber-physical systems (motion control and perception systems, multi-robots sync., remote control, Inter-connected for predictive maintenance, ...)



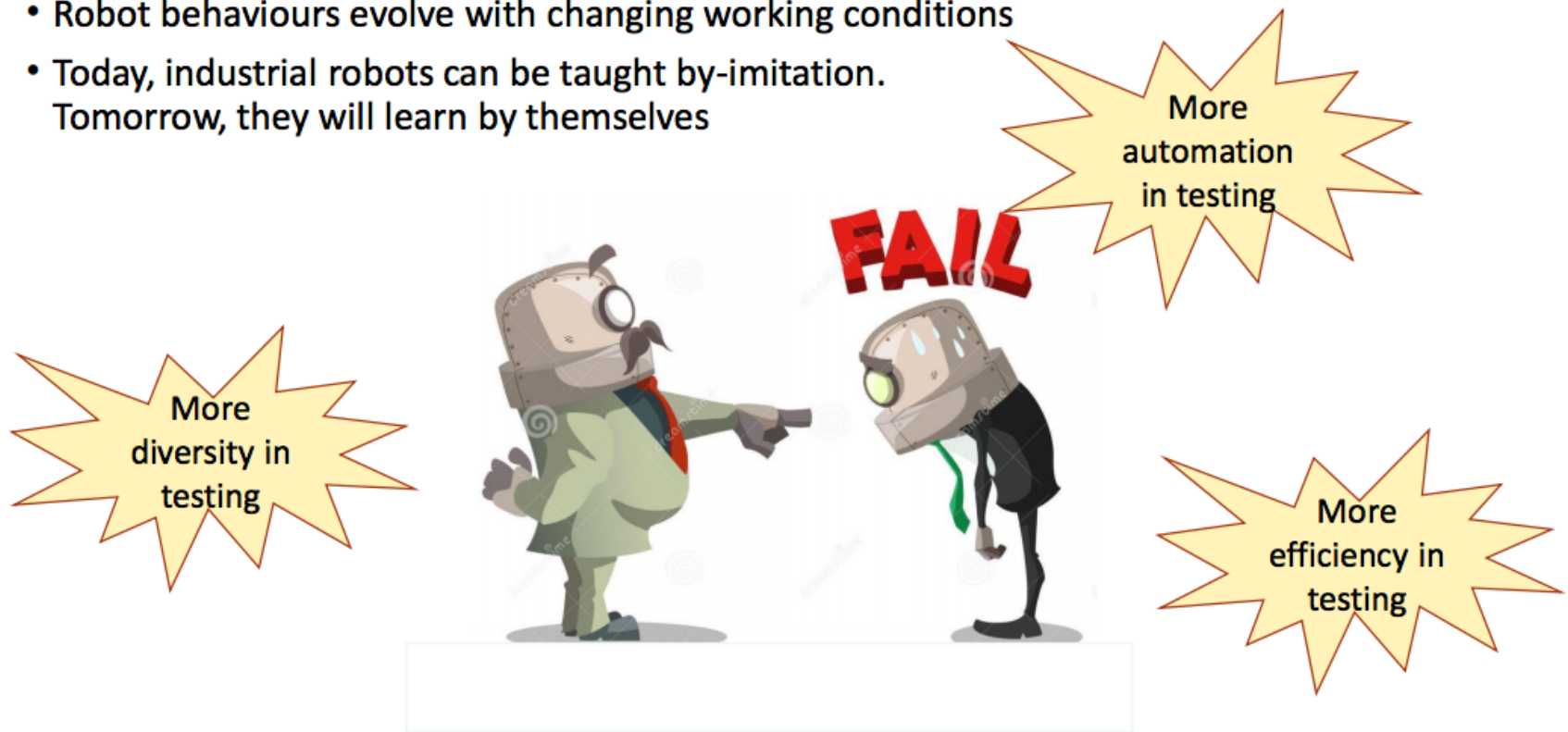
They are used to perform safety-critical tasks in complete autonomy (high-voltage component, on-demand painting with color/brush change, ..)

And to collaborate with human co-workers



Testing Robotic Systems is Crucial and Challenging

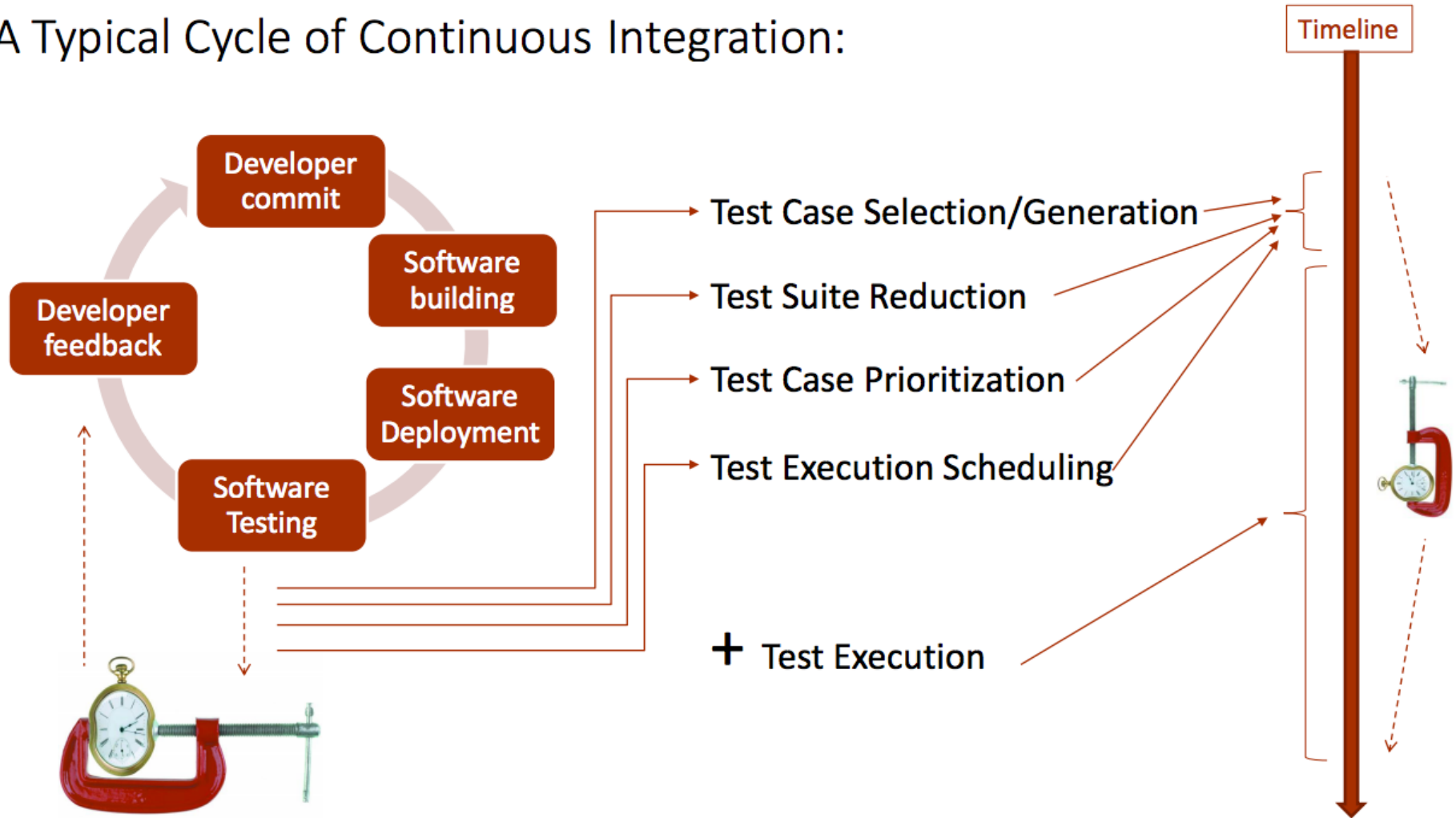
- The validation of industrial robots still involve too much human labour
- *“Hurry-up, the robots are uncaged!”*: Failures are not anymore handled using fences
- Robot behaviours evolve with changing working conditions
- Today, industrial robots can be taught by-imitation. Tomorrow, they will learn by themselves



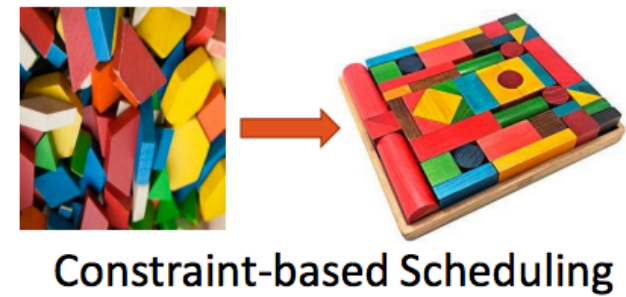
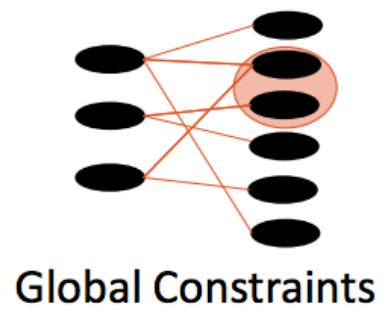
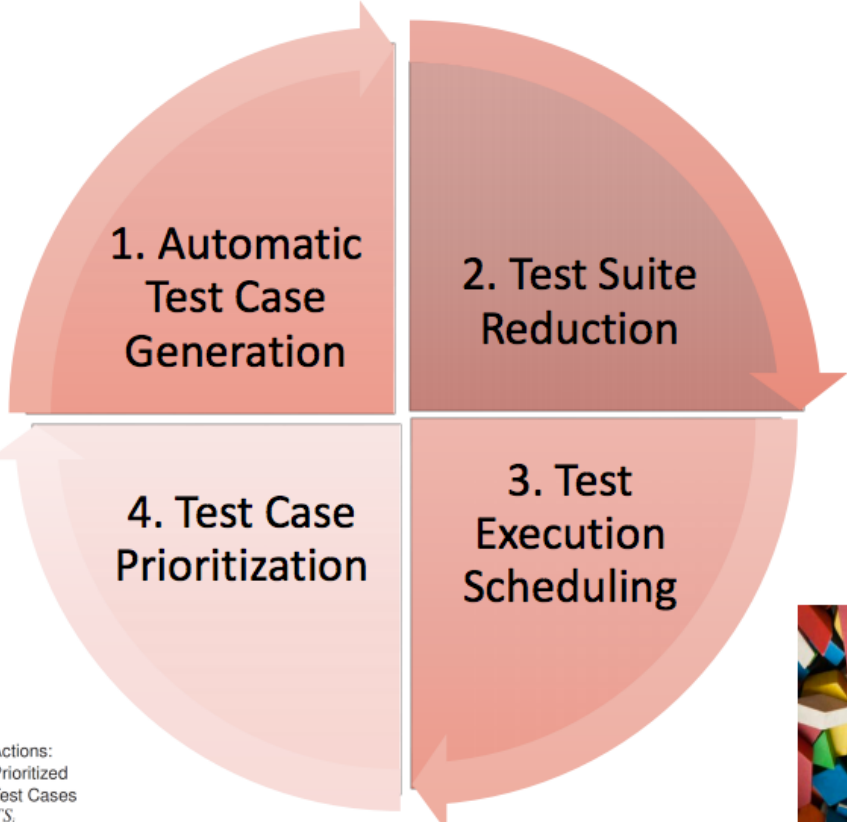
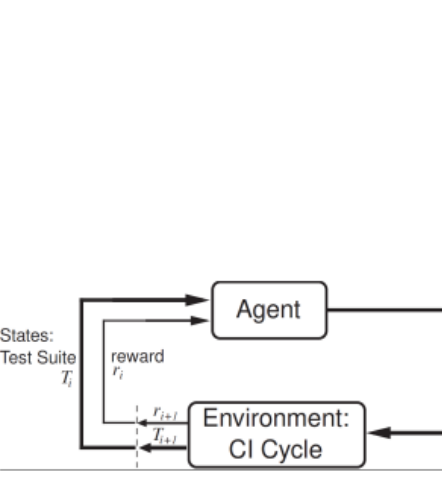
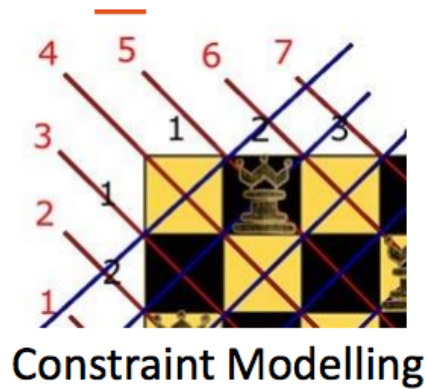
How Software Development of Industrial Robots Has Evolved...

From....	To...
Single-core, single application system	Multi-core, complex distributed system
All source code maintained by a small team located at the same place	Subsystems developed by distinct teams located at distinct places in the world
Manual system testing only handled in a single place, on actual robots	Automated software testing handled in a continuous integration process

A Typical Cycle of Continuous Integration:



Our Focus : Artificial Intelligence for Testing of Robotic Systems



Bibliographie

- ➔ Programmation par les tests, ESIREM, Céline ROUDET
- ➔ Comment écrire du code testable, Conférence Agile France 2010, Florence CHABANOIS
- ➔ Reflexion on Software Quality and Maintenance, Alexandre Bergel, Chili
- ➔ An Introduction to Test-Driven Development (TDD), Craig Murphy
- ➔ Tests et Validation du logiciel, <http://home.nordnet.fr/~eric/leu>
- ➔ Test à partir de modèles : pistes pour le test unitaire de composant, le test d'intégration et le test système, Yves Letraon
- ➔ Les tests en orienté objet, J. Paul Gibson <http://www-inf.int-evry.fr/cours/CSC4002/Documents>
- ➔ Mocks and Stubs, Martin Fowler
- ➔ Introduction au test du logiciel, Premiers pas avec JUnit, Mirabelle Nebut
- ➔ Écrire du code testable Par Aurélien Bompard

Des bugs et des tests



→ Objectifs

-Savoir où sont les bugs, pour

-Les corriger ou

-les documenter et donner des contournements :

-de version en version on voit les corrections apparaître (même dans le hard il y a des versions);

-Eviter la mauvaise image de la découverte du bug par le client

-Eviter le update qui régresse, c'est pire qu'un bug présent, ...

-Oser améliorer son code sans avoir peur d'introduire de «nouveaux» bugs

-Retour sur les exigences(bugs show product usage)

→ En conclusion : 80% du code sert à tester les cas d'erreur (principe de Pareto)

Tests...

- unitaire
- integration
- GUI
- non regression
- coverage
- load
- stress
- performance
- scalability
- volume
- usability/utilisateurs
- security
- recovery
- L10N/I18N (langue?, symbole)
- accessibility
- Installation/configuration
- Documentation
- Platform
- samples/tutorial
- code inspections

Qui teste ?

- ➔ L'utilisateur
- ➔ Les collègues en charge du test (s'il y en a)
- ➔ Le **développeur** : il a le devoir de fournir un code le plus clair et le mieux testé possible....
- ➔ La **machine**

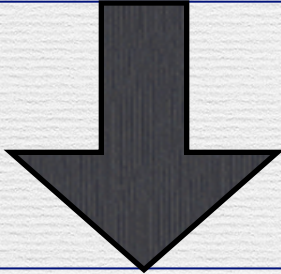
Qu'est-ce qu'on teste et comment ?

Quoi ?

- Fonctionnalité
- Sécurité / intégrité
- Utilisabilité
- Cohérence
- Maintenabilité
- Efficacité
- Robustesse
- Sûreté de

Comment ?

- Test statique :**
 - relecture / revue de code
 - analyse automatique (vérification de propriétés, règles de codage ...)
- Test dynamique :**
 - exécution du programme, et observation du comportement en fonction des valeurs en entrée.



Une spécification exprime ce qu'on attend du système, elle prend différentes formes en fonction de ce qui est ciblé : cahier des charges, use cases, données numériques, ...

S'organiser pour tester

- ➔ Module contenant les tests, indépendant du code « réel »
- ➔ Faire des **tests indépendants** les uns des autres
 - Pas de tests imbriqués
 - Un test qui échoue ne fait pas échouer les tests suivants
- ➔ **Automatiser les tests**
 - Gitlab CI, *Maven* : compiler et lancer automatiquement tous les tests
 - Utiliser un Environnement de tests : framework JUnit
- ➔ Réutiliser les tests et leurs résultats comme documentation

Et aussi ... Construire des suites de tests, des tests dynamiques

- ◆ A field can accept integer values between 20 and 50.
- ◆ What tests should you try?

A Test Ideas List for Integer-Input Tests

- ◆ Common answers to the exercise would include:

Test	Why it's interesting	Expected result
20	Smallest valid value	Accepts it
19	Smallest -1	Reject, error msg
0	0 is always interesting	Reject, error msg
Blank	Empty field, what's it do?	Reject? Ignore?
49	Valid value	Accepts it
50	Largest valid value	Accepts it
51	Largest +1	Reject, error msg
-1	Negative number	Reject, error msg
4294967296	2^{32} , overflow integer?	Reject, error msg

Sortes de Tests

Sortes de tests

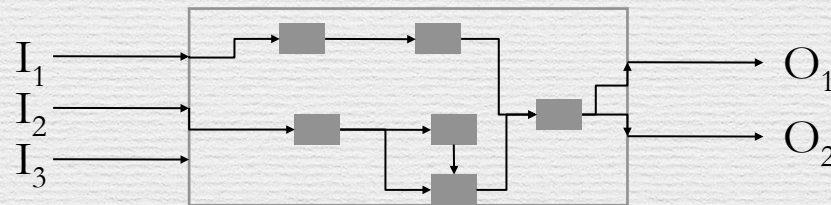
→ Tests en boîte noire: par ex. Tests système/fonctionnels

- Utilise la description des fonctionnalités du programme
- Provenant des spécifications (scenarii, uses cases)



→ Tests en boîte blanche: par ex Tests structurels

- Utilise la structure interne du programme

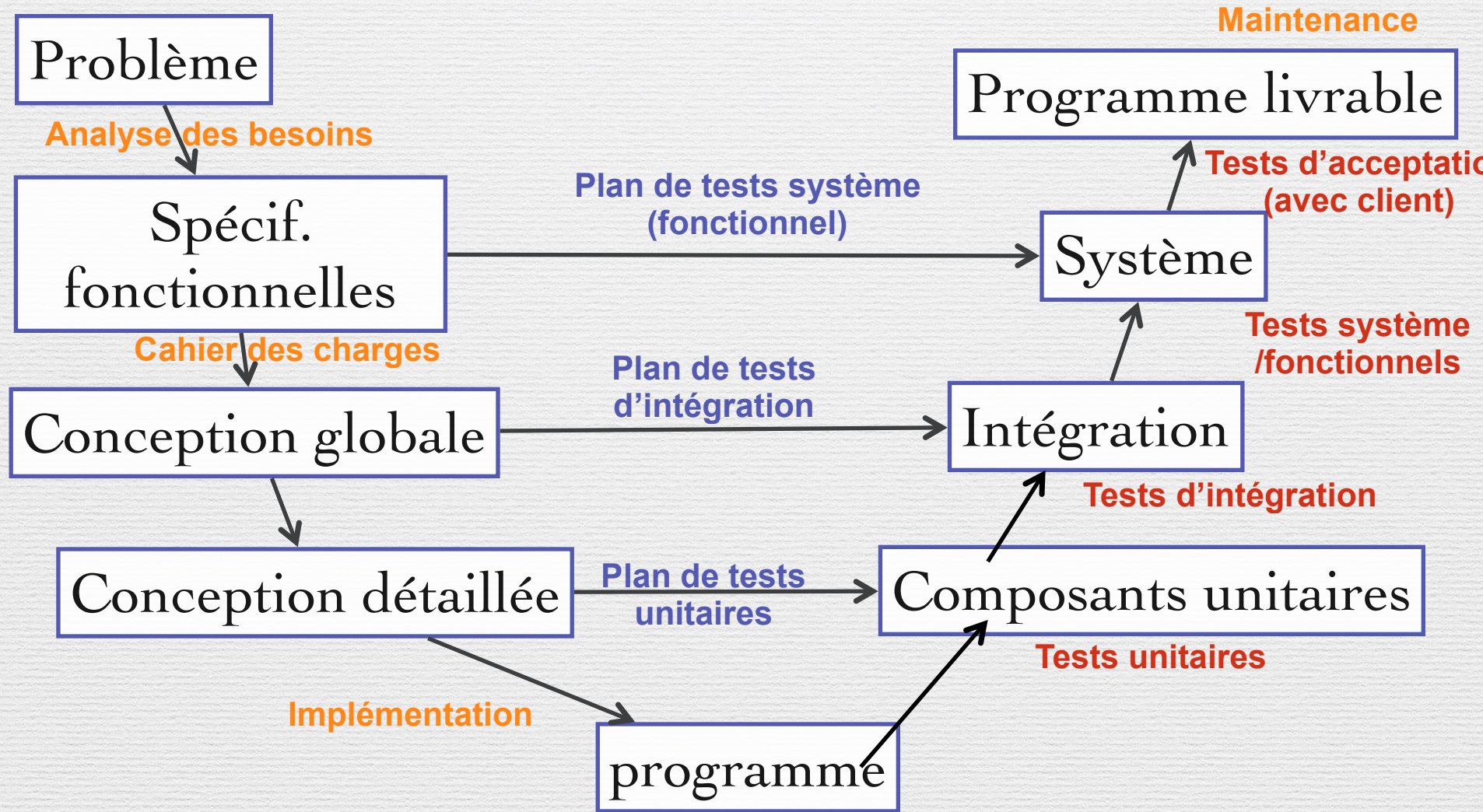


Sortes de tests

- ➔ Tests de (non) régression (après modifications)
 - Correction et évolution ne créent pas d'anomalies nouvelles
- ➔ Tests de robustesse
 - Cas de tests correspondant à des entrées non valides
- ➔ Tests de performance (application intégrée dans son environnement)
 - load testing : résistance à la montée en charge
 - stress testing : résistance aux demandes de ressources anormales
- ➔ ...

Hiérarchisation des tests dans le cycle en V

Déjà vu



Diagrammes UML et Tests

→ Niveau Application (spécification)

-Diagramme des cas d'utilisation Tests Système/Fonctionnel

-Diagramme de classes : test des associations, des agrégations

‣multiplicité,

‣création, destruction

Tests d'intégration

-Diagramme de séquence : test de séquences

‣ construction d'un graphe de flot

→ Niveau Classes (conception détaillée)

-Classes détaillées

Tests d'intégration

-Diagrammes de machine à états

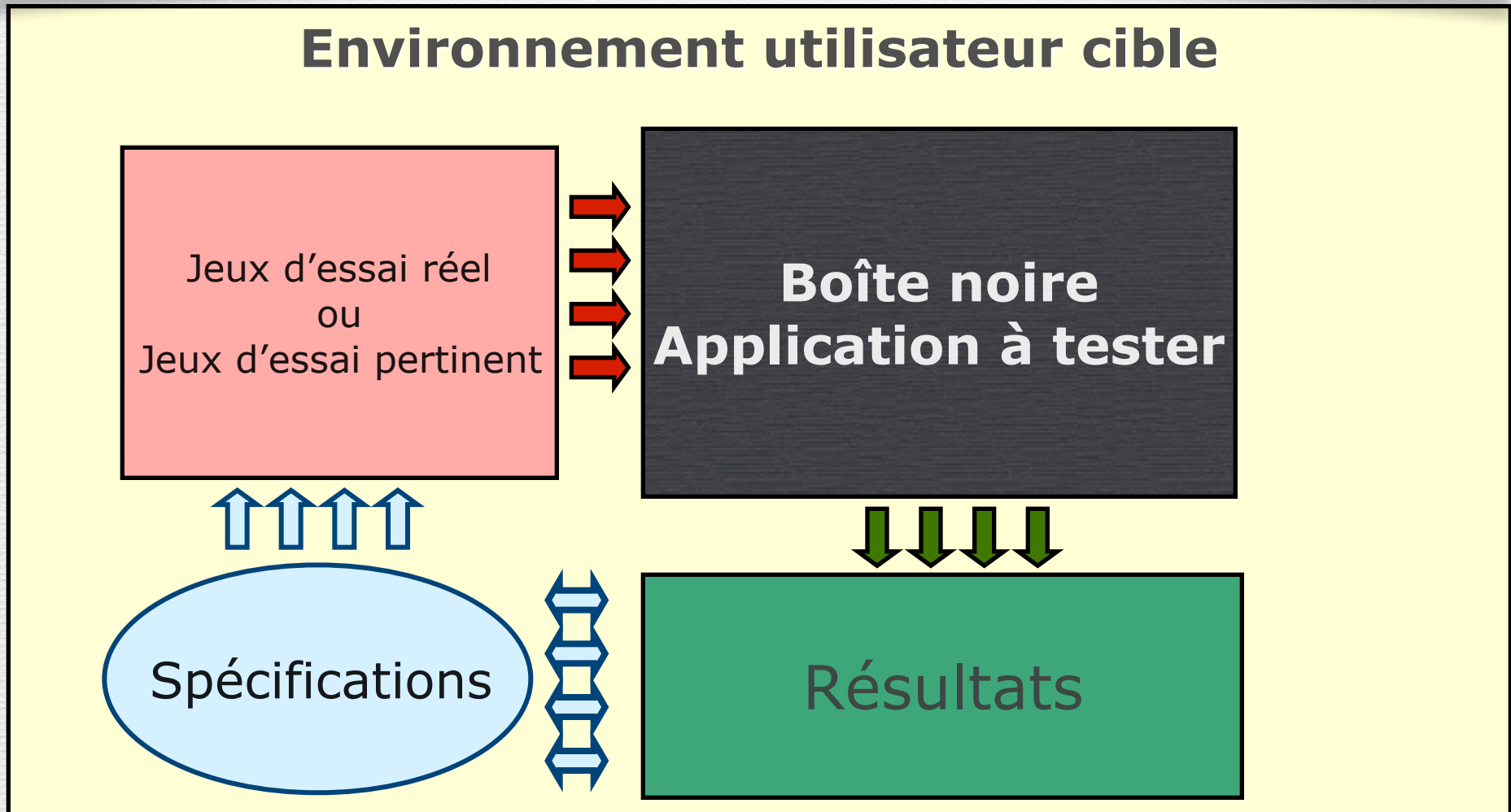
Tests unitaires

Tests Fonctionnels



Test Système (fonctionnel)

Vérifier que les fonctions correspondant aux attentes sont *bien atteintes*



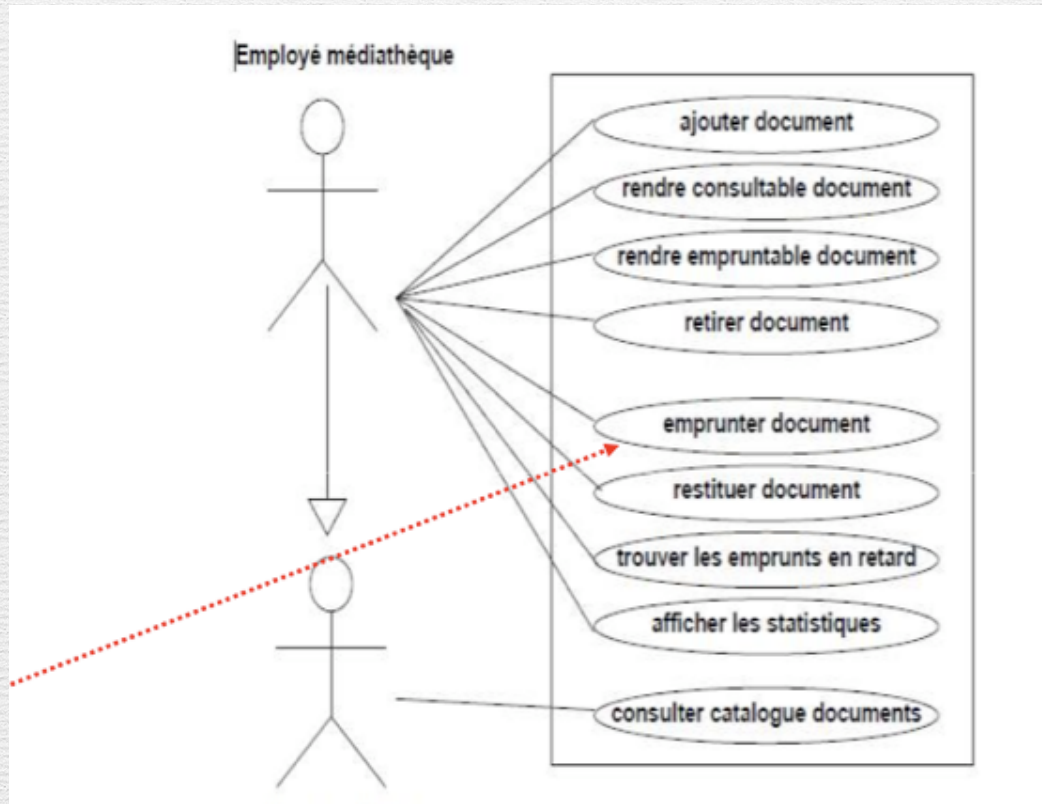
Tests Fonctionnels

→ Quand les spécifier ?

→ Comment les spécifier ?

Tests fonctionnels & UML

Quand sont-ils définis?



Tests fonctionnels & UML

Données d'entrée

client: peut être inscrit ou non;

emprunts: déjà effectués par le client

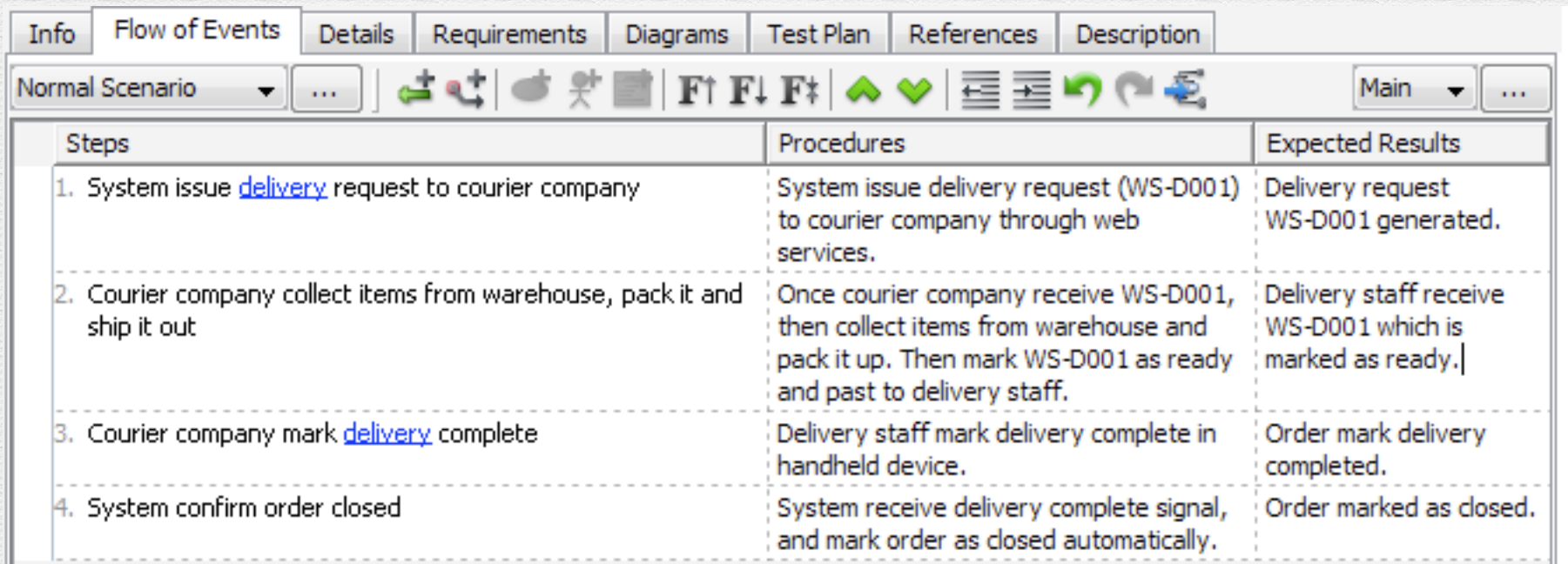
- existe-t-il un emprunt en retard ?
- le nombre d'emprunts déjà effectués correspond-il au nombre maximum de ce client ?

document:

- existe?
- empruntable ou consultable?,
- déjà emprunté ou disponible?

Use cases et Tests

- Pour chaque cas d'utilisation, sélectionner les scénarios et définir les tests correspondants.




The screenshot shows a software interface with a menu bar (Info, Flow of Events, Details, Requirements, Diagrams, Test Plan, References, Description) and a toolbar with various icons. Below the toolbar is a table with three columns: Steps, Procedures, and Expected Results. The table contains four rows of test data.

Steps	Procedures	Expected Results
1. System issue delivery request to courier company	System issue delivery request (WS-D001) to courier company through web services.	Delivery request WS-D001 generated.
2. Courier company collect items from warehouse, pack it and ship it out	Once courier company receive WS-D001, then collect items from warehouse and pack it up. Then mark WS-D001 as ready and past to delivery staff.	Delivery staff receive WS-D001 which is marked as ready.
3. Courier company mark delivery complete	Delivery staff mark delivery complete in handheld device.	Order mark delivery completed.
4. System confirm order closed	System receive delivery complete signal, and mark order as closed automatically.	Order marked as closed.

<http://www.visual-paradigm.com/product/vpuml/tutorials/testingprocedure.jsp>


D'une manière générale, la préparation des jeux de tests permet de lever les ambiguïtés et réparer des oublis.

Tests fonctionnels & US



En tant que passager
PRIMO, **je veux**
annuler ma
réservation,...

Seules les taxes «
Autres » me sont
remboursées moins
les frais administratifs



En tant que passager
BUSINESS FLEX,, **je**
veux annuler ma
réservation, ...

L'ensemble du billet
m'est remboursé à
100% moins les frais
administratifs

Tests fonctionnels... vers l'automatisation

<http://seleniumhq.org/docs/>

<https://www.youtube.com/watch?v=gsHyDlyA3dg>

<http://watir.com>



Feature: Features of my service

Scenario: Calling my service
Given my service exists
When I call my service
Then it should have been a success

Executer le test.

Cucumber vous demande alors d'implémenter les phrases présentes dans le scenario.
Il faut les mettre dans une nouvelle classe :

```
public class MyServiceSteps {  
    private MyService service;  
    private boolean success;  
  
    @Given("My service exists$")  
    public void my_service_exists() throws Throwable {  
        service = new MyService();  
    }  
  
    @When("^I call my service$")  
    public void i_call_my_service() throws Throwable {  
        success = service.doSomething();  
    }  
  
    @Then("^it should have been a success$")  
    public void it_should_have_been_a_success() throws Throwable {  
        Assert.assertTrue(success);  
    }  
}
```

<http://www.opensourcetesting.org>

Prise en compte de la variabilité

« Au sein de la société, nous utilisons donc les principaux navigateurs utilisés : Google Chrome (49,8% des utilisateurs) , Safari (14,9%) , Internet Explorer/ Edge (14,8%) et Mozilla Firefox (7,7%) . »

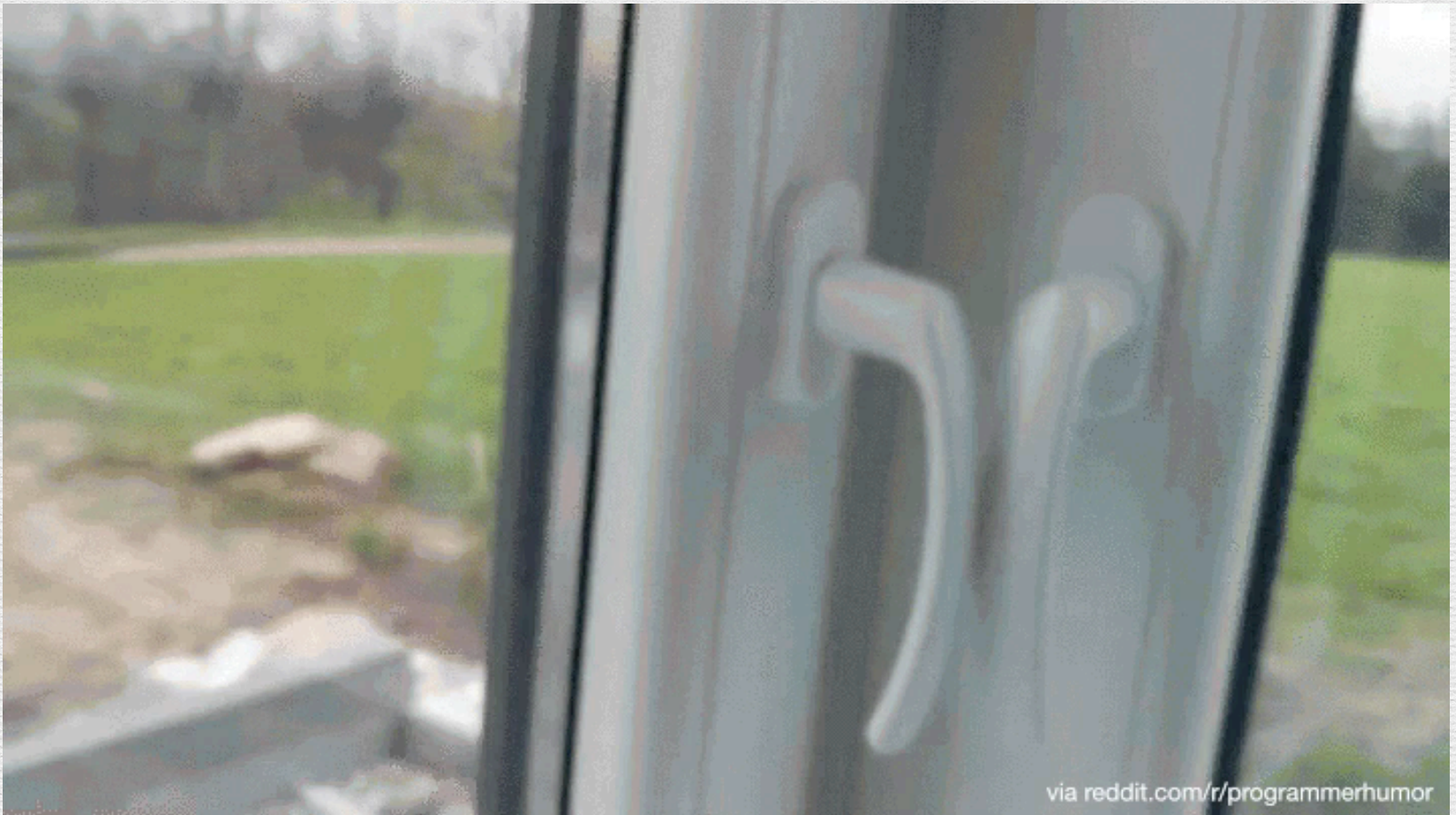
Tests d'intégration (voir cours dédié)

- ✓ Différents modules d'une application peuvent fonctionner unitairement, leur intégration, entre eux ou avec des services tiers, peut engendrer des dysfonctionnements.
- ✓ Il est souvent impossible de réaliser les tests unitaires dans l'environnement cible avec la totalité des modules à disposition.
- ➡ Les tests d'intégration ont pour objectif de créer une version complète et cohérente du logiciel (avec l'intégralité des modules testés unitairement) et de garantir sa bonne exécution dans l'environnement cible.

 The Practical Dev a retweeté

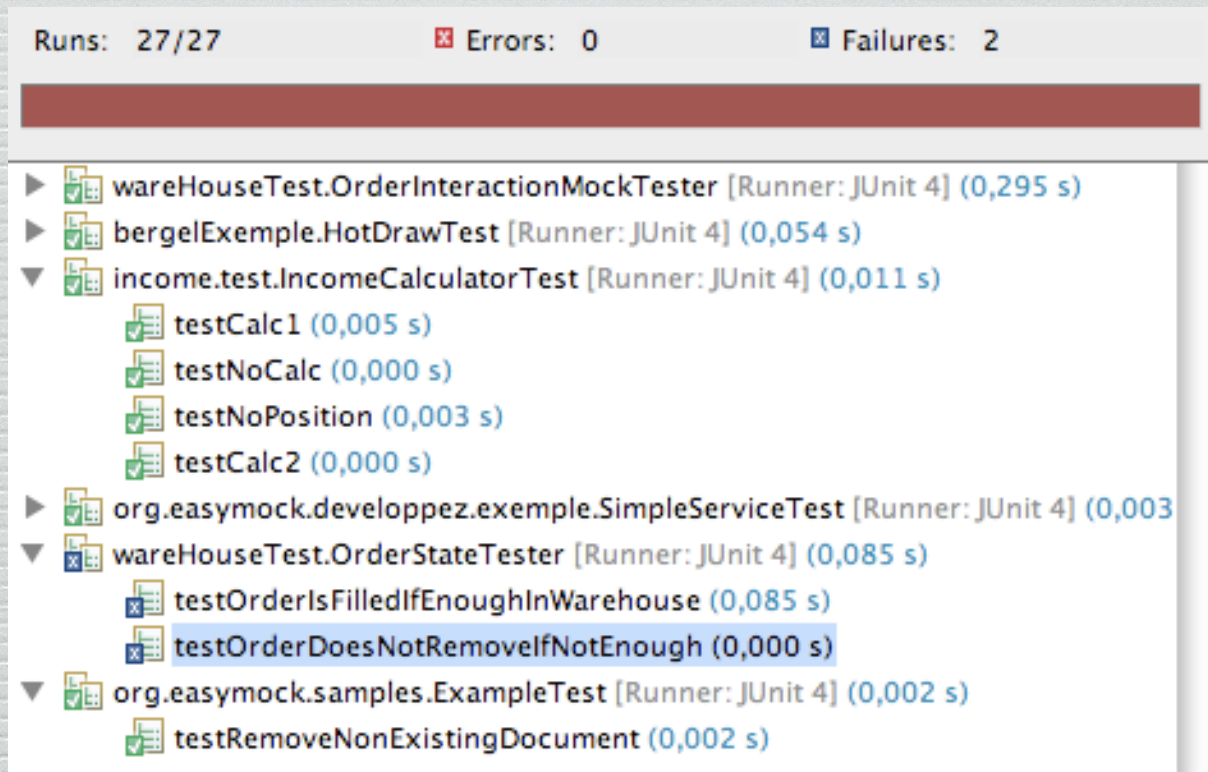
DEV

The Practical Dev @ThePracticalDev · 14 janv.
2 unit tests. 0 integration tests



Tests Unitaires

exemple
JUnit



The screenshot displays the JUnit test runner interface. At the top, it shows the overall test status: "Runs: 27/27", "Errors: 0", and "Failures: 2". Below this, a list of test classes and methods is shown, each with a corresponding icon indicating its status (green checkmark for success, red X for failure). The test classes and their methods are:

- wareHouseTest.OrderInteractionMockTester [Runner: JUnit 4] (0,295 s)
- bergelExemple.HotDrawTest [Runner: JUnit 4] (0,054 s)
- income.test.IncomeCalculatorTest [Runner: JUnit 4] (0,011 s)
 - testCalc1 (0,005 s)
 - testNoCalc (0,000 s)
 - testNoPosition (0,003 s)
 - testCalc2 (0,000 s)
- org.easymock.developpez.exemple.SimpleServiceTest [Runner: JUnit 4] (0,003 s)
- wareHouseTest.OrderStateTester [Runner: JUnit 4] (0,085 s)
 - testOrderIsFilledIfEnoughInWarehouse (0,085 s)
 - testOrderDoesNotRemovelfNotEnough (0,000 s)
- org.easymock.samples.ExampleTest [Runner: JUnit 4] (0,002 s)
 - testRemoveNonExistingDocument (0,002 s)

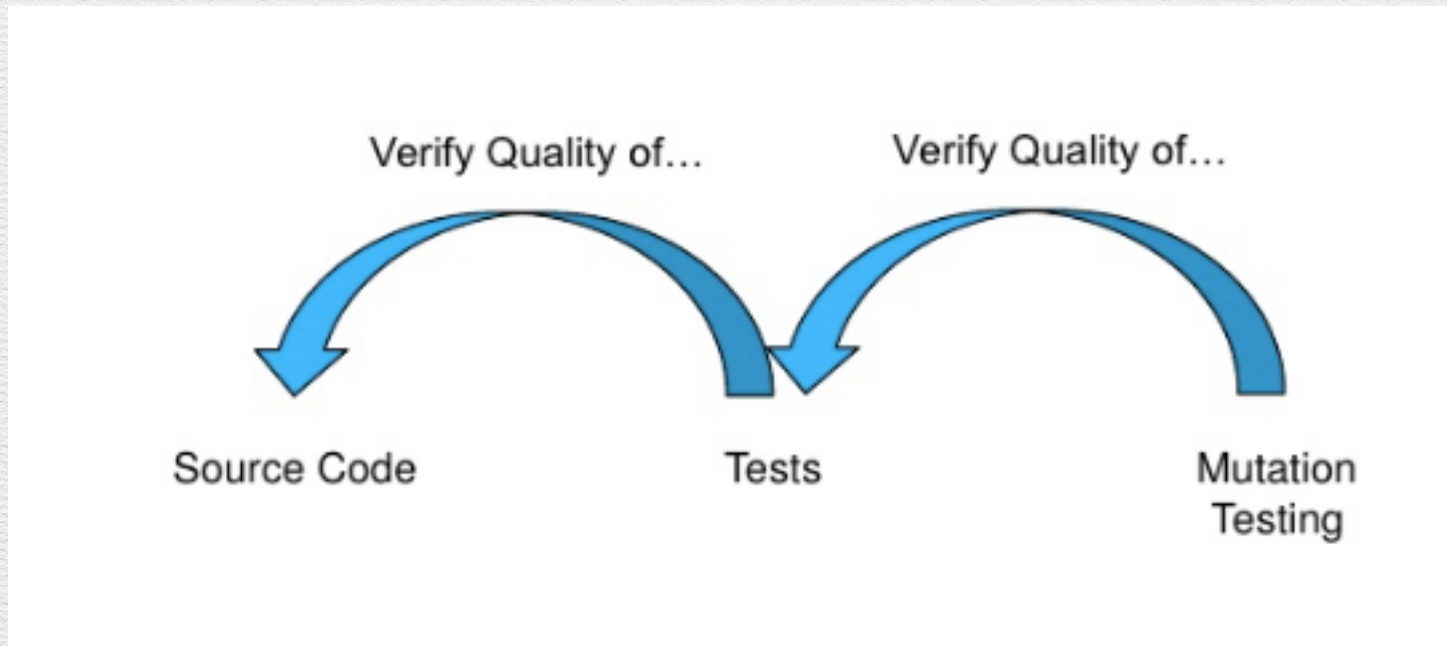
Automatisation des tests ...

- 1- tests par mutation
2. Tests d'utilisation

- tests dynamiques, génération de tests en fonction des données,

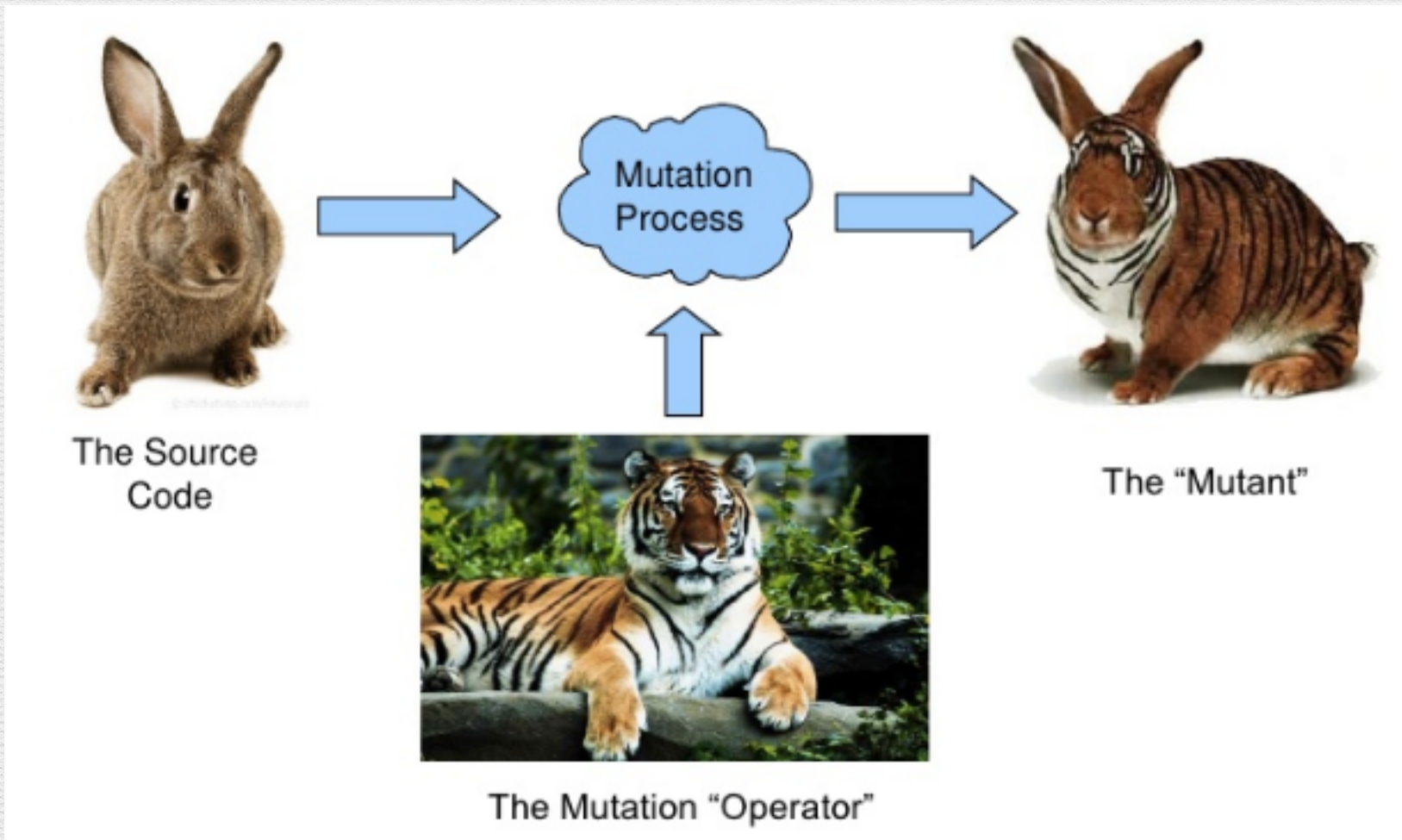
Tests par mutation

→ Technique pour vérifier la qualité des tests



<http://www.slideshare.net/esug/mutation-testing>

Comment cela fonctionne ?



Transformations?

```
DebitCard >>= anotherDebitCard  
^(type = anotherDebitCard type)  
and: [number = anotherDebitCard number]
```

Operator: Change #and: by #or:

```
CreditCard >>= anotherDebitCard  
^(type = anotherDebitCard type)  
or: [number = anotherDebitCard number]
```

Tuer les mutants ?



The "Mutant"



A Killer
tries to kill the Mutant!



The Test Suite

All tests run → The Mutant Survives!!!

A test fails or errors → The Mutant Dies

Usability Testing

→ Purpose: Find out if the system is really usable by its intended audience

→ Why?

- System is built by developers ... but used by Business Users
- Even minimal UI changes can confuse business users with years of experience of “doing it this way”
- System has to face real-life usage

Usability Testing

- ➔ How: Almost impossible to automate
- ➔ Tips:
 - Involve ergonomic specialists early in the project
 - Use reusable, standardized UI components
 - Take performance into account: a slow responding system won't be accepted easily
 - Have Business Users test early on UI mock

Usability Testing

Un cours dédié à la préparation des tests

► Tools: Eye tracking

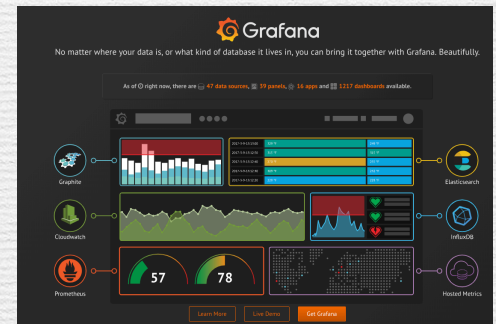
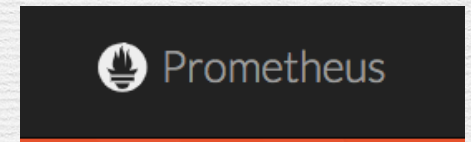
The image shows a screenshot of the Heuga website with several blue circular markers and lines overlaid, indicating eye-tracking data. The website layout includes:

- Header:** Heuga logo, language selection (UK, Germany, France), and a shopping trolley icon.
- Navigation:** Catalogue, Advice and information, Why Heuga?, Product search.
- Main Content:** A large image of a living room with a modular floor. Text reads: "Design led flooring solutions for your home - for people who think differently." Below this is a call to action: "View and buy our products online >>".
- Quick links:** A dropdown menu with options: "Living room", "Bedroom", "Bathroom", "Kitchen", "Office", "Kitchen", "Utility room".
- Footer/Bottom Section:** Several smaller text blocks and images, including "Combine SmartSteps" and "Get inspired...".

The eye-tracking markers are concentrated on the main product image, the navigation menu, and the quick links dropdown, suggesting these are key areas of user interest.

Test de charge & performances

- ➔ Volume des données traitées
- ➔ Nombre d'utilisateurs simultanés



Nécessité d'une bonne estimation des besoins dès le début et prise en compte de leur évolution

Prenons un exemple

*« XX ayant des problèmes de ralentissements importants voir de crash assez régulièrement il a été décidé de lancer une campagne de **tests de performance**, ... Trois principales étapes composent la campagne de tests : la création de scénarios fonctionnels, le scripting de ces scénarios et les tirs de performance.*

Création de tests fonctionnels

La première chose à faire pour débuter la campagne est d'écrire des scénarios fonctionnels qui seront joués par l'outil de tirs. Il faut donc définir les scénarios les plus fidèles à une utilisation classique de l'outil, mais aussi les plus consommateurs afin de comprendre pourquoi l'application subit des ralentissements. »

A quoi cela correspond-il d'après vous?

Suite à cela, j'ai pu commencer la rédaction de quatre scénarios, selon le plan indiqué par l'équipe de tests, c'est-à-dire une suite d'instructions pas à pas avec une capture d'écran à chaque étape, le temps d'attente moyen d'un utilisateur entre chaque page, le nombre d'utilisateurs effectuant ce scénario en période normale et en période de pic d'activité, etc.

Ensuite, il faut constituer un jeu de données : des comptes utilisateur de tests, des données qui seront consommées pendant les tests, etc.

Et ce n'est pas toujours simple !

Un des problèmes rencontrés a été l'insuffisance du jeu de données pour un des scénarios. Nous avons environ 140 données, mais selon nos calculs pour l'ensemble de la campagne de test, 7000 données auraient été nécessaires. Créer des données à la main étant bien trop long et fastidieux, nous avons dû trouver une solution rapide et efficace... **nous avons convenu de créer un nouveau scénario qui serait joué en masse et qui permettrait de créer les données nécessaires.**

Tests de performance en entreprise

La phase des tirs se déroule en 3 étapes :

- ✎ Un **tir nominal**, soit environ 1h30 en utilisation classique, puis une montée en charge pour simuler un pic d'activité,
- ✎ Un **tir de stress** soit environ 1h30 en simulation de pic d'activité, puis une montée en charge jusqu'au crash de l'application,
- ✎ Un **tir d'endurance**, soit 8h d'utilisation classique et plusieurs montées en charge pour simuler des pics d'activité au cours de la journée.

DevOps... Tester
en continu.



C'est aujourd'hui.