

VÉRIFICATION & VALIDATION

Eduardo COUTO MONTENEGRO – EDF 2023-2024



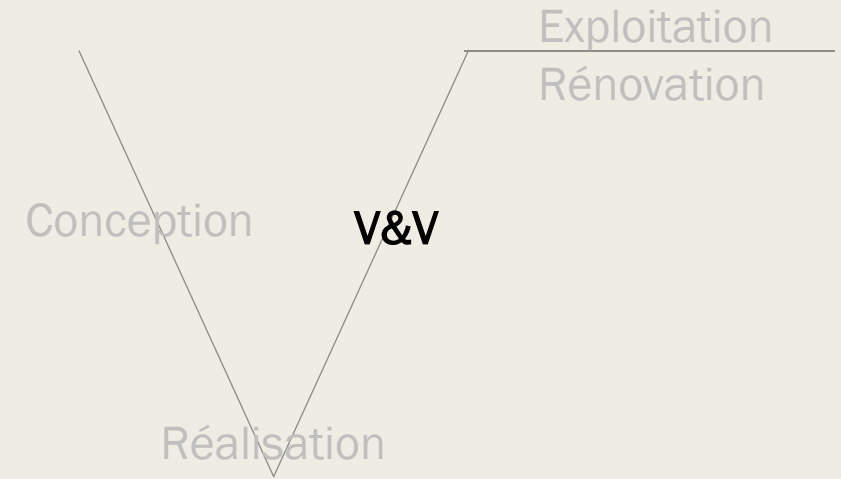
Eduardo.couto-montenegro@edf.fr

Vérification & Validation

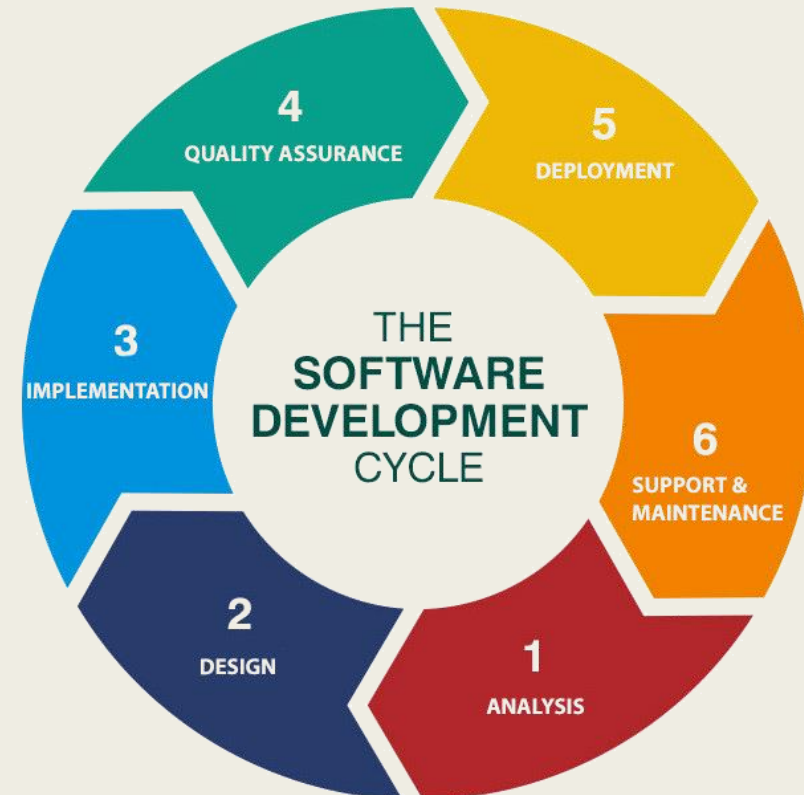
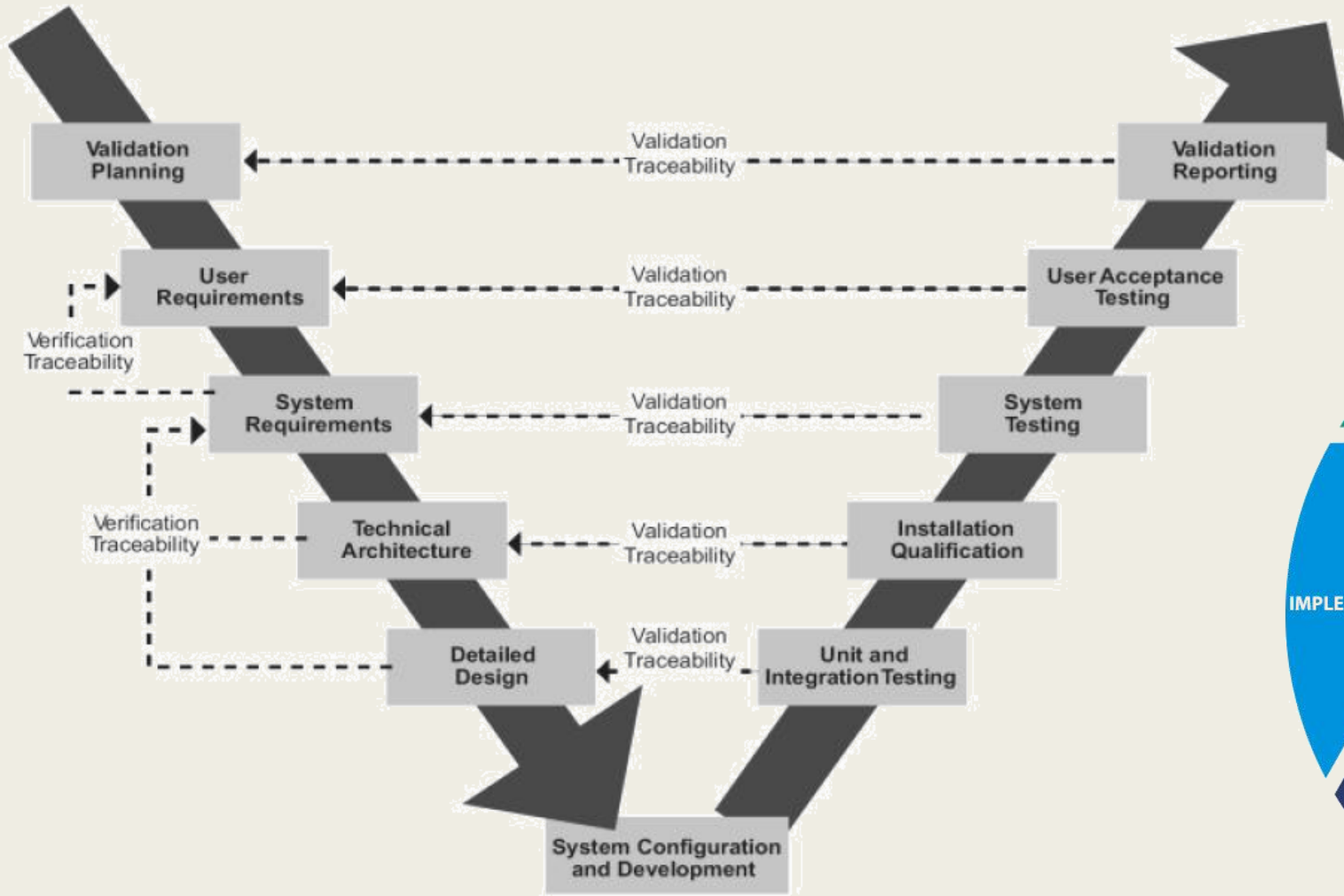
- Notions abordées
 - *Techniques de test*
 - *Validation des exigences*
 - *Test HIL (Hardware In The Loop)*
 - *Plateformes de Intégration Continue (devOps)*

Intervenant : Eduardo Couto Montenegro (Dina Irofti)

- 4h cours + 3h TP



SDLC - Software Development Lifecycle



SDLC - Software Development Lifecycle

Le cycle de vie du développement logiciel (SDLC) est un processus complet de développement d'une solution logicielle avec différentes étapes et étapes pour amener le logiciel de l'idéation à la création, au déploiement et à la maintenance.

Divers professionnels sont impliqués dans les **tests** d'applications, tels que les testeurs, les gestionnaires, les développeurs et les utilisateurs finaux. De plus, un cycle de vie de test d'application implique des phases, notamment :

- Analyse des exigences de test
- Teste plannconception et conception
- Internationaux
- Analyse de texte
- Rapport de bogue



Ingénieur QA



QA, cela veut dire Quality Assurance, Assurance Qualité en français.

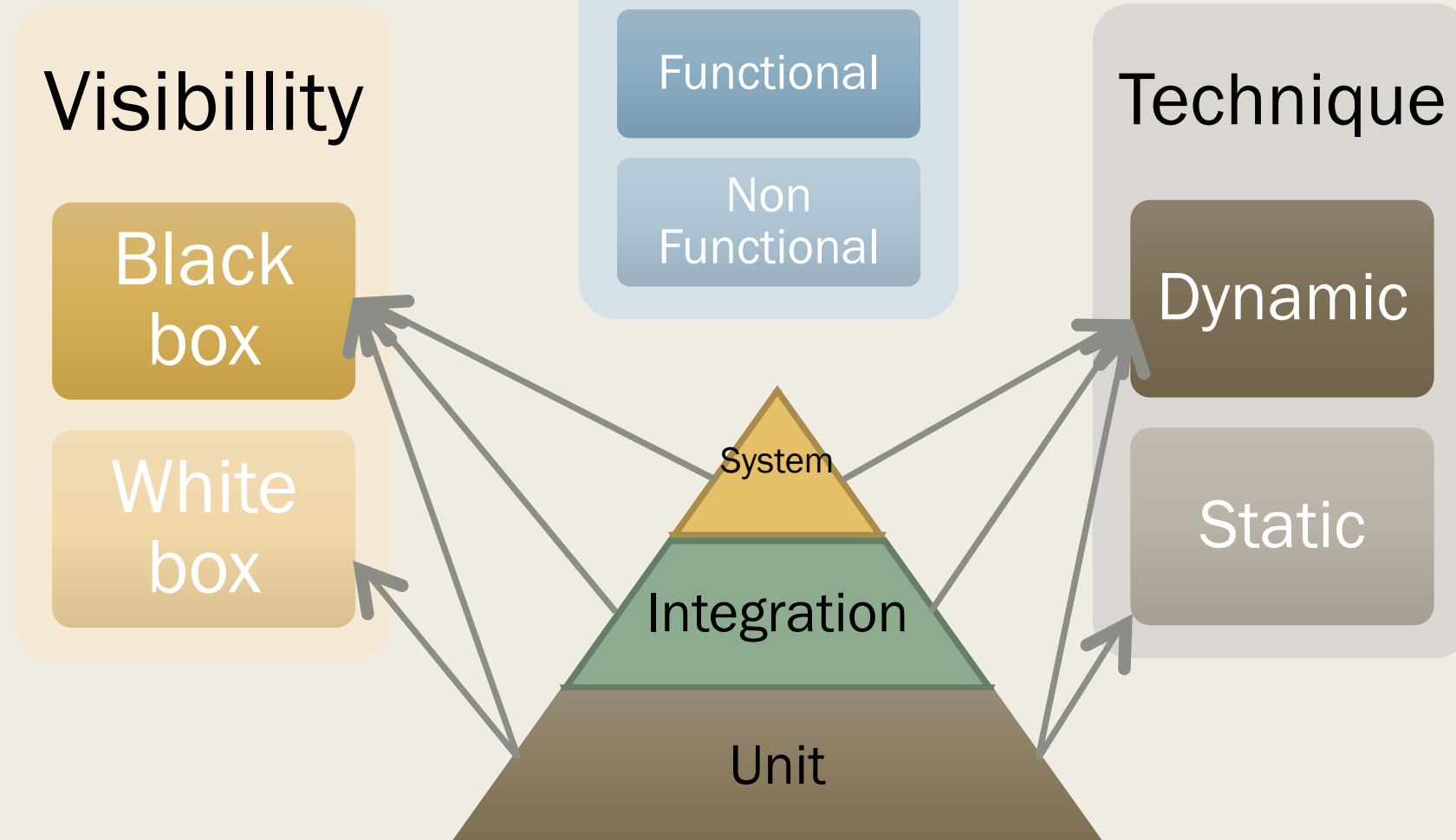
Dans le domaine informatique, il s'agit de **garantir la qualité des développements** réalisés ; autrement dit, qu'une fois mis en production, le logiciel va fonctionner comme attendu.

Pour garantir la qualité du code produit, l'ingénieur QA va concevoir et **mettre en œuvre** un ensemble de **tests** fonctionnels et techniques, détecter les bugs éventuels et les cas limites d'utilisation, et enfin présenter tous ces éléments aux développeurs afin qu'ils puissent corriger les problèmes.

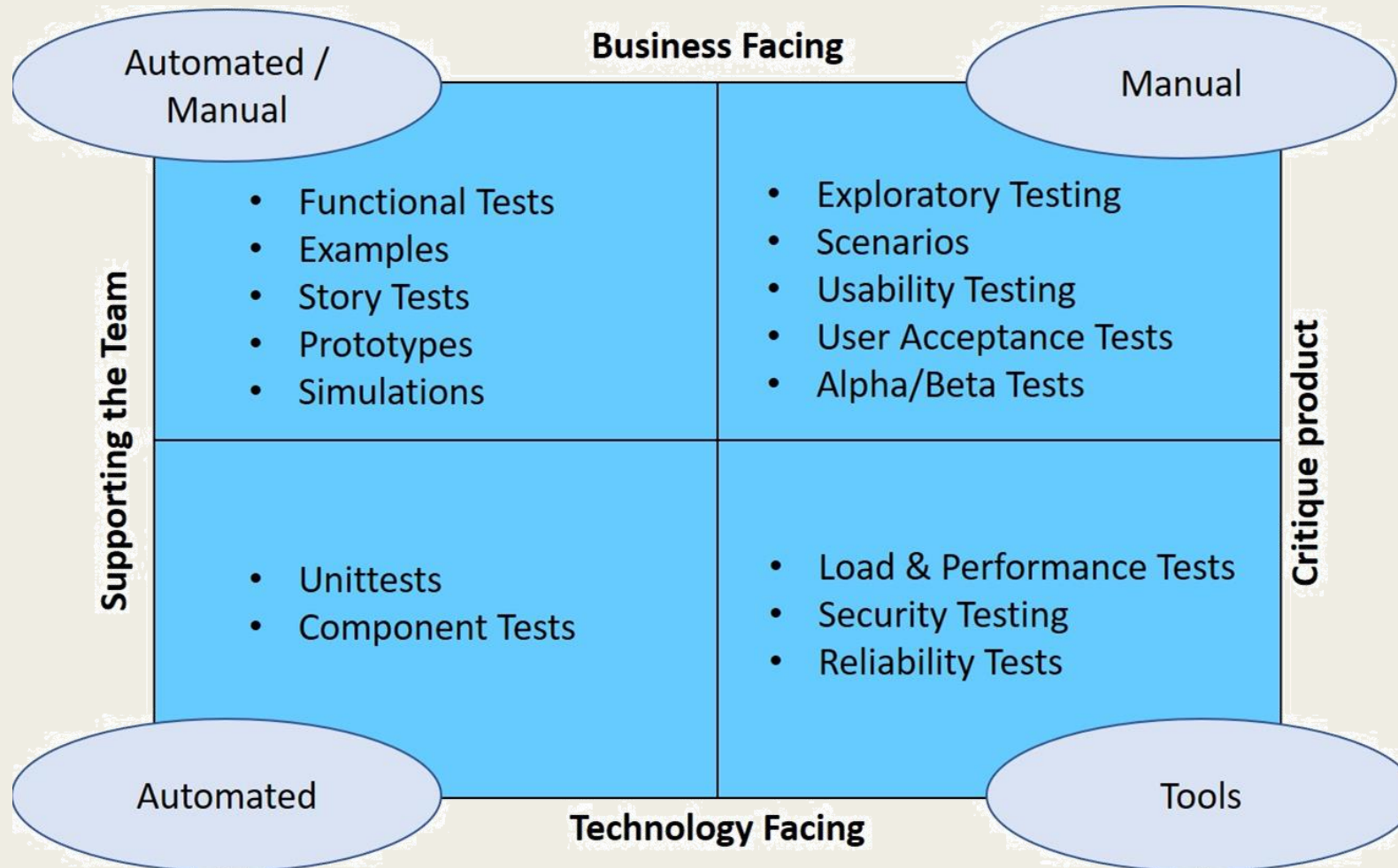
D'une manière générale, on peut dire qu'il faut savoir :

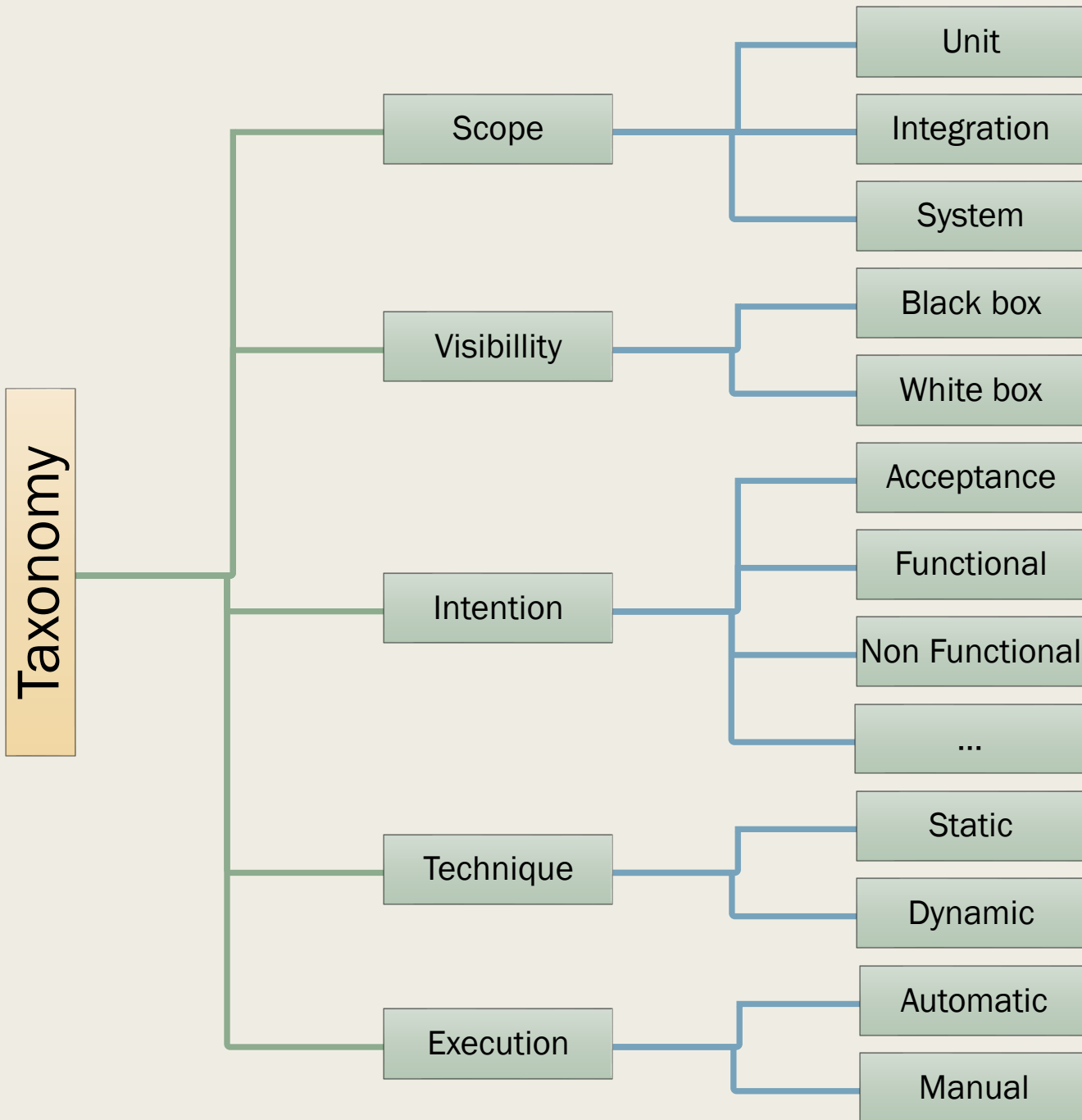
- Gérer des campagnes de test (Intégration, Validation, Non régression)
- Concevoir des plans de test
- Automatiser des scénarii de test (et donc utiliser de langages de scripting comme Shell ou Bash)
- Réaliser des rapports précis pour décrire les problèmes rencontrés

Tests



Agile testing quadrants





Évaluation statique

La notion d'analyse statique de programmes couvre une variété de méthodes utilisées pour obtenir des informations sur le comportement d'un programme lors de son exécution sans réellement l'exécuter.

- Méthodes informel
- Méthodes formelles (inspection)
 - *Listes de contrôle*
 - *Extractions successives*
- Procédure pas à pas

Évaluation dynamique

- White box
 - *Path Coverage*
 - *Statement Coverage*
 - *Condition Coverage*
 - *Function Coverage*
- Black box
 - *Equivalence partitioning*
 - *Boundary values analysis*

White box (*-coverage)

1. Obtenir un organigramme

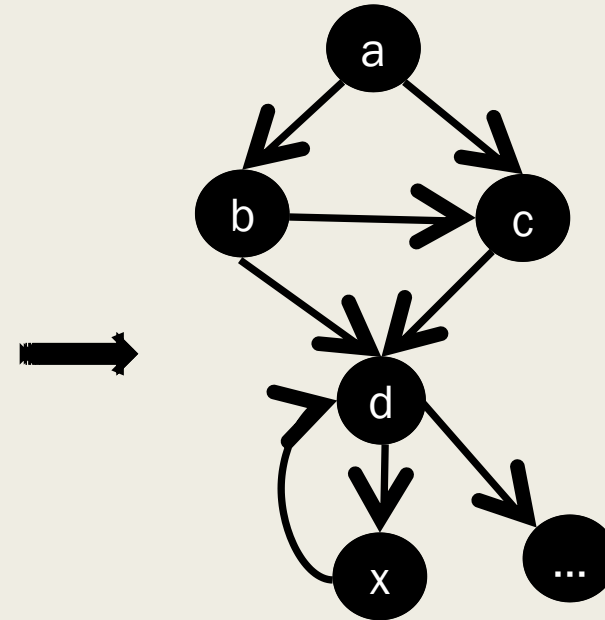
```
...
errors = []
if (user.name == null || user.email == null) {
    errors.push('mandatory fields not found');
}
//do the rest of whatever
for (var i=0; i < user.friends ; i++) {
    errors.push(checkFriendShipt(user.friends[i]))
}
...
```

2. Calculer la complexité cyclomatique
3. Déterminer un ensemble de données qui force à suivre un chemin ou un autre

White box (*-coverage)

1. Obtenir un organigramme

```
...  
errors = []  
if (user.name == null || user.email == null) {  
    errors.push('mandatory fields not found');  
}  
//do the rest of whatever  
for (var i=0; i < user.friends; i++) {  
    errors.push(checkFriendShipt(user.friends[i]))  
}  
...
```

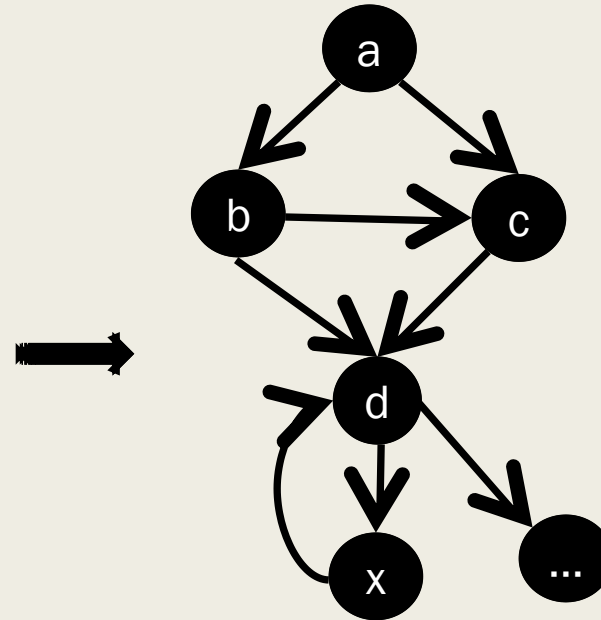


2. Calculer la complexité cyclomatique
3. Déterminer un ensemble de données qui force à suivre un chemin ou un autre

White box (*-coverage)

1. Obtenir un organigramme

```
...
errors = []
if (user.name == null || user.email == null) {
  errors.push('mandatory fields not found');
}
//do the rest of whatever
for (var i=0; i < user.friends; i++) {
  errors.push(checkFriendShipt(user.friends[i]))
}
...
```



2. Calculer la complexité cyclomatique

$edges - nodes + 2 = predicate\ nodes + 1 = number\ of\ regions$

3. Déterminer un ensemble de données qui force à suivre un chemin ou un autre

Testing workflow

1. Préparation
2. Exécuteur
3. Vérifier
4. Démolir

- X-Unit (collectif de plusieurs frameworks de tests unitaires)
 - `@Before(Class)`
 - `@Test`
 - `@After(Class)`

```
@Before
public void setUp() {
    this.userValidator = mock(UserValidator.class);
    this.userDao = mock(UserDao.class);
    this.userService = new UserService(userValidator, userDao);
}

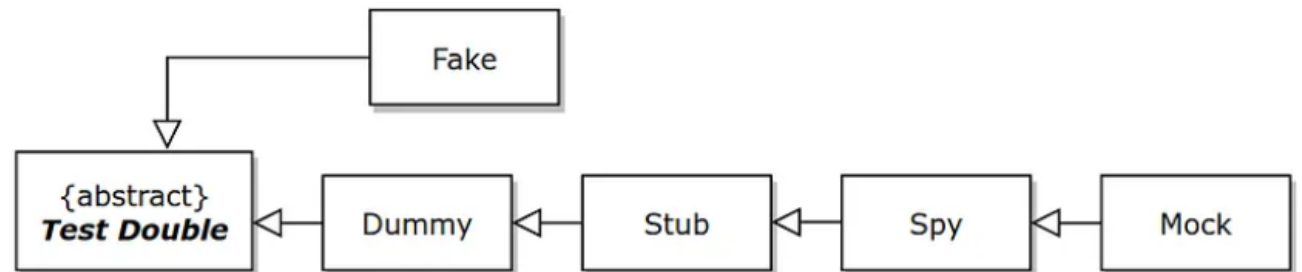
@Test
public void createValidUserShouldNotFail() {
    //Exercise
    User expectedCreatedUser = new User("irrelevantUser");
    when(userValidator.validate(any(User.class)));
    when(userValidator.validate(any(User.class))).thenReturn(createdUser);
    User createdUser = userService.create(new User());
    //Assertions
    assertThat(createdUser, equalTo(expectedCreatedUser));
}

@Test(expected=ValidationException)
public void createInvalidUserShouldFail() {
    when(userValidator.validate(any(User.class)))
        .thenReturn(new ValidationException());
    userService.create(new User("irrelevantUser"));
}

@After
public void tearDown() {
    //clean the state here
}
```

Test doubles (Doublures d'essai)

- **Fake:** used as a simpler implementation, e.g. using an in-memory database in the tests instead of doing real database access.
- **Dummy:** It is used as a placeholder when an argument needs to be filled in. when a parameter is needed for the tested method but without actually needing to use the parameter.
- **Stub:** It provides fake data to the SUT (System Under Test).
- ~~**Spy:** It records information about how the class is being used.~~
- **Mock:** It defines an expectation of how it will be used. It will cause failure if the expectation isn't met.



Fake Implémentation afin de faire réussir le test.

```
public UserDaoFake implements UserDao {  
    @Override  
    public User create(User user) {  
        return new User(...);  
    }  
}
```

Spy Sont des objets qui enregistrent également certaines informations en fonction de la façon dont ils ont été appelés

```
UserValidator validatorSpy = spy(new UserValidator());  
doThrow(new ValidationException()).when(validatorSpy).validate();  
verify(validatorMock).validate(any(User.class))
```

Stub Fournissent des réponses prédéfinies aux appels passés pendant le test, ne répondant généralement pas à quoi que ce soit en dehors de ce qui est programmé pour le test.

```
UserValidator validatorMock = mock(UserValidator.class);  
stub(validatorMock.validate(any(User.class))).toThrow(new ValidationException());
```

Mocks Il répond avec des valeurs par défaut aux méthodes non explicitement déclarées

```
UserValidator validatorMock = mock(UserValidator.class);  
when(validatorMock.validate(any(User.class))).thenReturn(new ValidationException());  
  
verify(validatorMock).validate(any(User.class))
```


gtest

[GitHub - google/googletest: GoogleTest - Google Testing and Mocking Framework](https://github.com/google/googletest)

Google Test (également connu sous le nom de **gtest**) est une bibliothèque de tests unitaires pour le langage de programmation C++, publiée sous la licence BSD 3 et basée sur l'architecture xUnit.

Google Test peut être compilé pour une variété de plates-formes POSIX et Microsoft Windows, ce qui permet des tests unitaires des sources C ainsi que C++ avec une modification minimale des sources.

```
-----] Running 1 test from 1 test case.  
-----] Global test environment set-up.  
[ RUN   ] 1 test from testCase  
[ OK    ] testCase.test0  
[       ] testCase.test0 (0 ms)  
[       ] 1 test from testCase (0 ms total)  
-----] Global test environment tear-down  
[ PASSED ] 1 test from 1 test case ran. (0 ms total)  
[ PASSED ] 1 test.
```



googletest
Google C++ Testing Framework

Mock and test

```
class Turtle {  
    ...  
    virtual ~Turtle() {}  
    virtual void PenUp() = 0;  
    virtual void PenDown() = 0;  
    virtual void Forward(int distance) = 0;  
    virtual void Turn(int degrees) = 0;  
    virtual void GoTo(int x, int y) = 0;  
    virtual int GetX() const = 0;  
    virtual int GetY() const = 0;  
};
```

mock

```
#include <gmock/gmock.h> // Brings in gMock.  
  
class MockTurtle : public Turtle {  
public:  
    ...  
    MOCK_METHOD(void, PenUp, (), (override));  
    MOCK_METHOD(void, PenDown, (), (override));  
    MOCK_METHOD(void, Forward, (int distance), (override));  
    MOCK_METHOD(void, Turn, (int degrees), (override));  
    MOCK_METHOD(void, GoTo, (int x, int y), (override));  
    MOCK_METHOD(int, GetX, (), (const, override));  
    MOCK_METHOD(int, GetY, (), (const, override));  
};
```

test

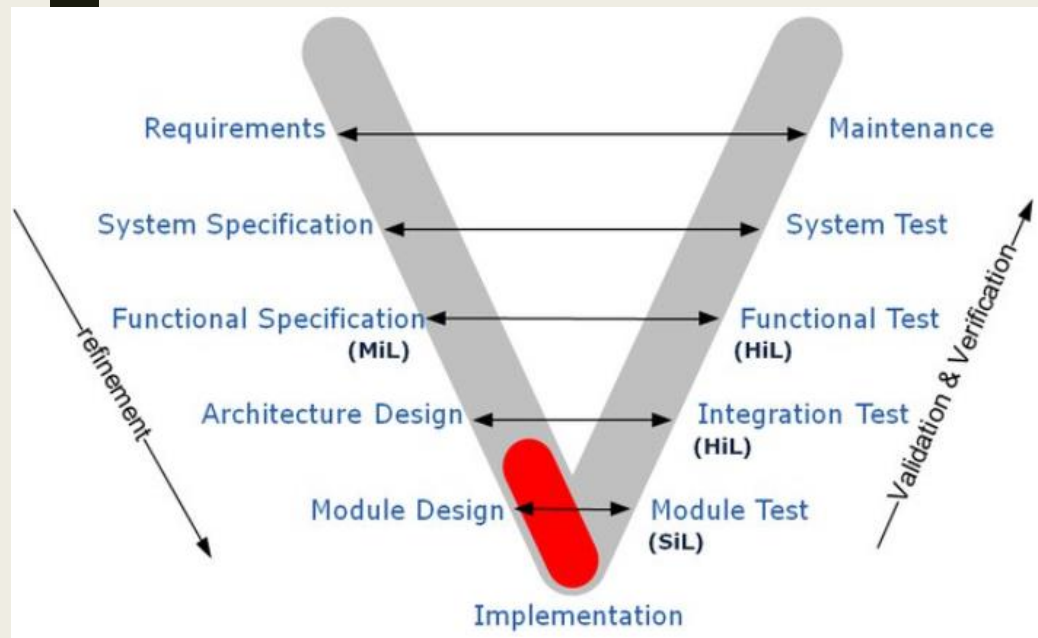
```
#include "path/to/mock-turtle.h"  
#include <gmock/gmock.h>  
#include <gtest/gtest.h>  
  
using ::testing::AtLeast; // #1  
  
TEST(PainterTest, CanDrawSomething) {  
    MockTurtle turtle; // #2  
    EXPECT_CALL(turtle, PenDown()) // #3  
        .Times(AtLeast(1));  
  
    Painter painter(&turtle); // #4  
  
    EXPECT_TRUE(painter.DrawCircle(0, 0, 10)); // #5  
}
```

As you might have guessed, this test checks that `PenDown()` is called at least once. If the `painter` object didn't call this method, your test will fail with a message like this:

```
path/to/my_test.cc:119: Failure  
Actual function call count doesn't match this expectation:  
Actually: never called;  
Expected: called at least once.  
Stack trace:  
...
```

Utilisent des simulations

Leur utilisation accélère la conception et améliore la maîtrise de la qualité, tout en réduisant le recours aux prototypes réels et aux essais physiques. Permettent de valider les solutions développées en situation réelle simulée



MIL model in the loop : L'ensemble du système est modélisé, ce qui permet de simuler un environnement complet (véhicule par exemple) afin de tester les lois de commande et de corriger les erreurs mécaniques, électroniques et logicielles avant la fabrication de prototypes (validation fonctionnelle) ;

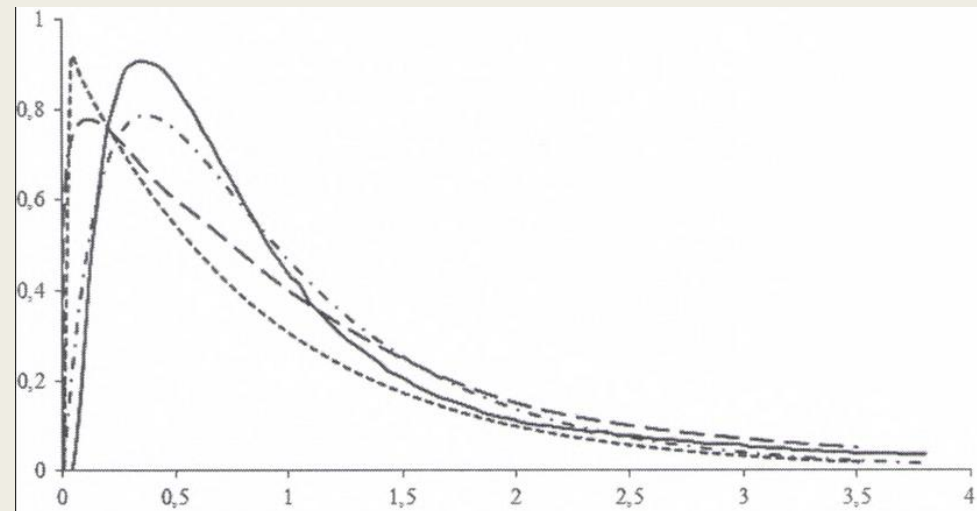
SIL software in the loop : Le code qui sera utilisé ensuite dans le calculateur est testé unitairement, afin de corriger les erreurs en simulant le fonctionnement du système réel (validation de la génération de code) ;

HiL hardware in the loop : Les lois de commande sont intégrées dans un calculateur physique relié à l'environnement de simulation, et parfois à une partie mécanique réelle, afin de tester l'implémentation matérielle du contrôle, les de temps de réponse, etc. (validation de l'intégration).

Architecture Model-In-the-Loop (MIL)

La co-simulation (MIL ou « Model-In-the-Loop ») permet de simuler et de faire interagir plusieurs modèles (souvent simulés avec des solveurs ou des pas de temps différents). Un des avantages est d'avoir une plus grande indépendance entre les outils et de développer des modèles avec des logiciels différents, plus adaptés à chaque métier.

Plusieurs interfaces entre les modèles sont possibles, par exemple, par réseau, par interface propriétaire dédiée ou par le standard de co-simulation FMI (« Functional Mock-up Interface »).



Test HIL (Hardware In The Loop)

Les tests HIL sont des simulations **temps réel** qui vous permettent de commencer à tester du code embarqué sans hardware.

Permettent de tester des scénarios anormaux et dysfonctionnels susceptibles d'endommager les composants hardware si le code en cours de développement est exécuté en dehors des spécifications.

La validation du code embarqué pour des systèmes de contrôle et système complexes en testant des prototypes constitue un défi important, car il y a le risque d'endommagement des composants hardware.

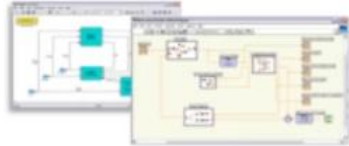
Les performances des simulations dépendent à la fois de la:

- complexité de la dynamique des systèmes que vous modélisez
- du hardware temps réel que vous utilisez.

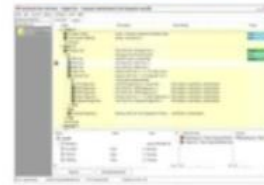


NI HIL Testing Platform

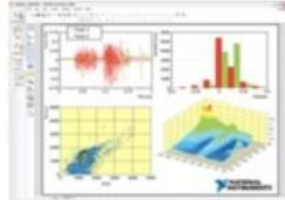
Modeling and Simulation



Test Automation



Analysis and Reporting



Real-Time Testing SW



Requirements Traceability



Real-Time
Processor



Analog/
Digital I/O



Fault
Insertion



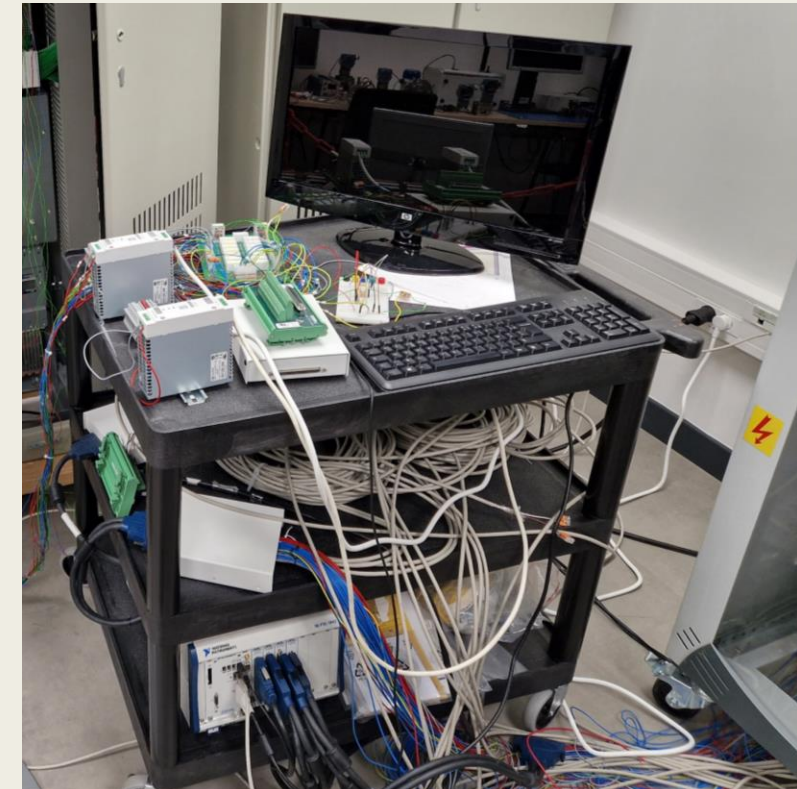
Bus
Interfaces



Instrument
Grade and RF I/O



Vision/
Motion



Intégration Continue (CI)

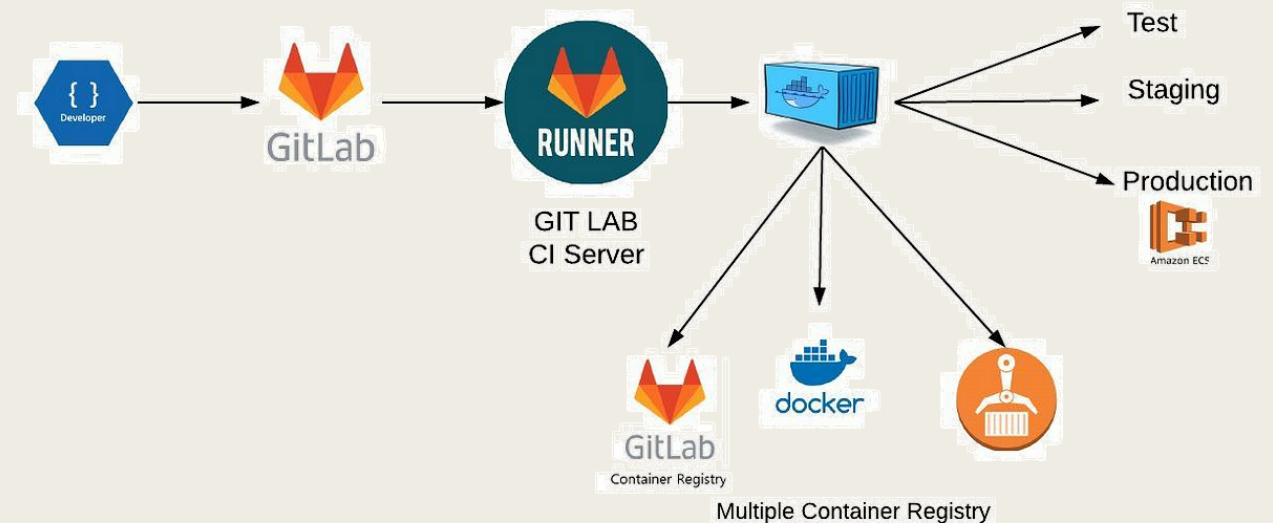
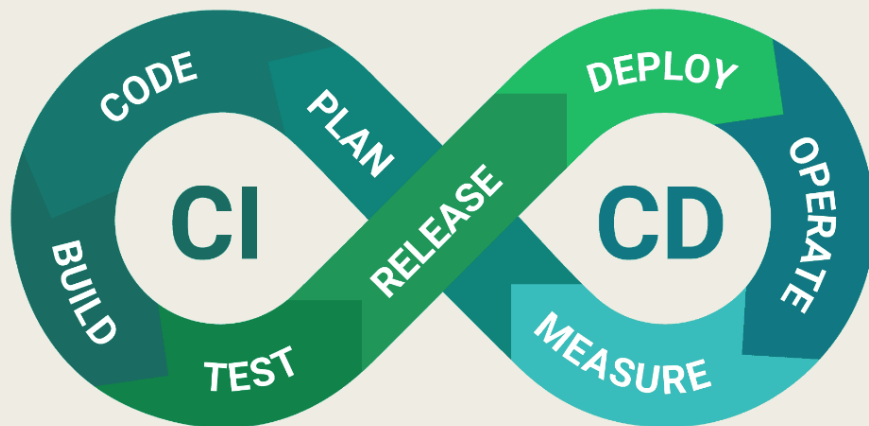
L'intégration continue (**Continuous integration**, CI), est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée.

L'intégration continue repose souvent sur la mise en place d'une **brique logicielle** permettant l'automatisation de tâches : compilation, tests unitaires et fonctionnels, validation produit, tests de performances... À chaque changement du code, cette brique logicielle va exécuter un ensemble de tâches et produire un ensemble de résultats, que le développeur peut par la suite consulter. Cette intégration permet ainsi de ne pas oublier d'éléments lors de la mise en production et donc ainsi améliorer la qualité du produit

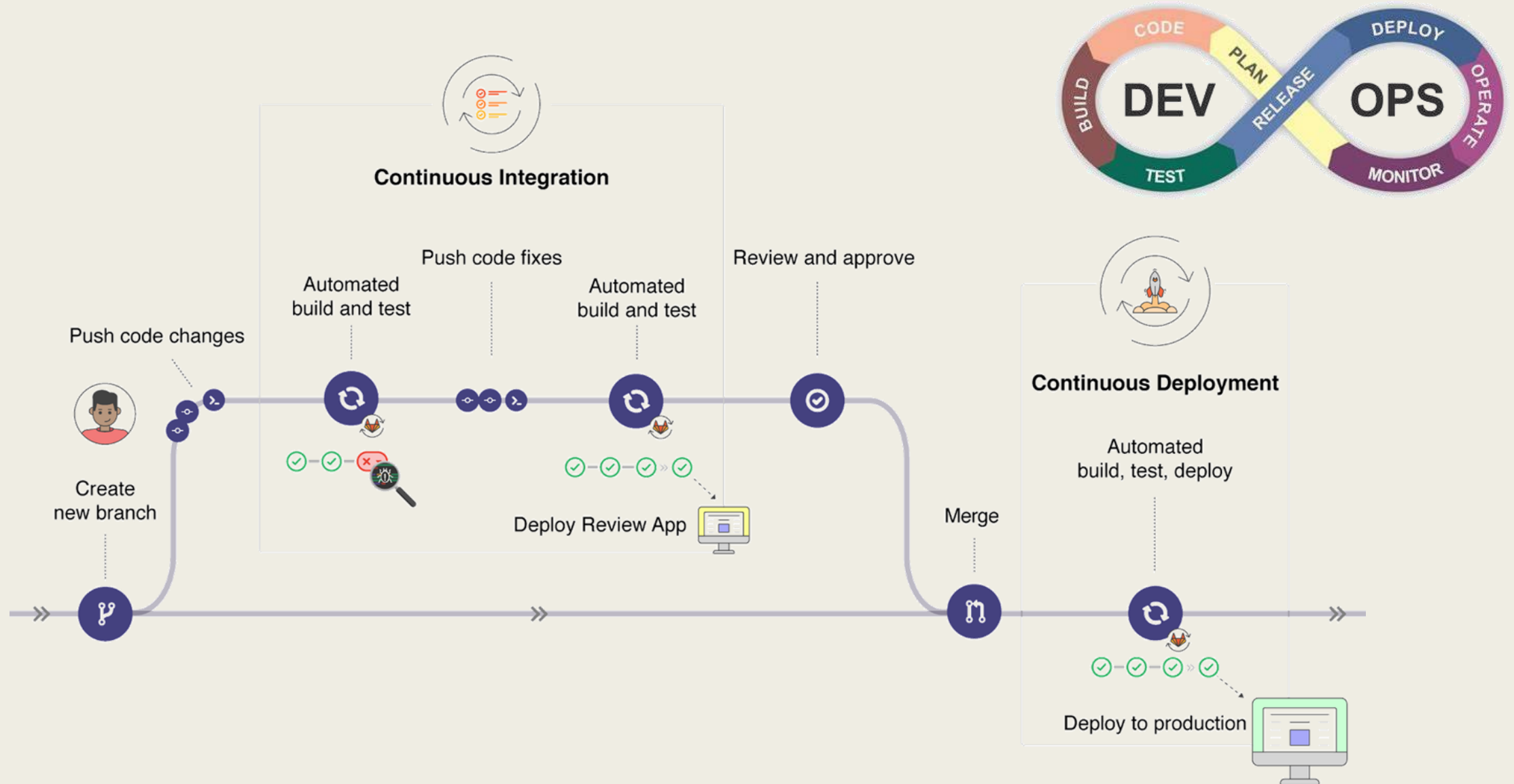
CI/CD

L'approche CI/CD permet d'augmenter la fréquence de distribution des applications grâce à l'introduction de l'automatisation au niveau des étapes de développement des applications.

Les principaux concepts liés à l'approche CI/CD sont l'intégration continue, la distribution continue et le déploiement continu. Il représente une solution aux problèmes posés par l'intégration de nouveaux segments de code pour les équipes de développement et d'exploitation



Combinant développement (**Dev**) et opérations (**Ops**), **DevOps** est l'union des personnes, des processus et des technologies destinés à fournir continuellement de la valeur aux clients



Jenkins



Jenkins est un outil open source de serveur d'automatisation.

Il aide à automatiser les parties du développement logiciel liées au build, aux tests et au déploiement, et facilite l'intégration continue et la livraison continue. Écrit en Java, Jenkins fonctionne dans un conteneur de servlets tel qu'Apache Tomcat, ou en mode autonome avec son propre serveur Web embarqué.

Il s'interface avec des systèmes de gestion de versions tels que Git et Subversion, et exécute des projets basés sur Apache Ant et Apache Maven aussi bien que des scripts arbitraires en shell Unix ou batch Windows. Ses fonctionnalités peuvent être étendues facilement à l'aide de plugins supplémentaires qui vous permettront, par exemple, de connecter l'outil à d'autres environnements de tes



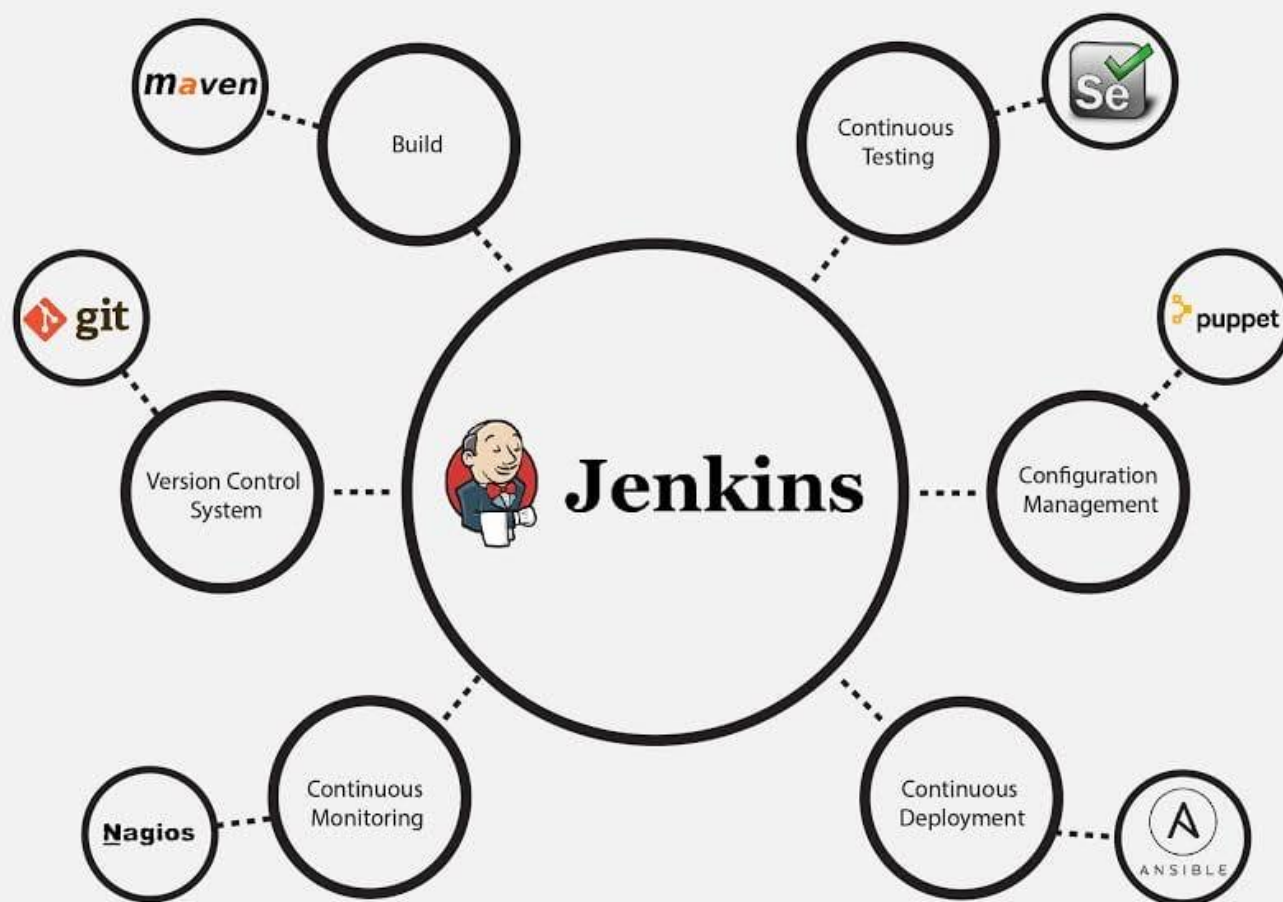


Jenkins

Maven™




Jenkins





Jenkins - tests

**Jenkins**

search

log in

Jenkins

People

Build History

Project Relationship

Check File Fingerprint

Disk usage

We Need Beer

Build Queue

No builds in the queue.

Build Executor Status

master

1 Idle

2 Idle

celery

1 Idle

2 Idle

remote-slave-3 (offline)

remote-slave-6

1 Idle

2 Idle

remote-slave-7 (offline)

remote-slave-8 (offline)

AllAll DisabledAll FailedAll UnstableInfrastructureJenkins coreLibrariesOther Projects

S	W	Name ↓	Last Success	Last Failure	Last Duration	LC
		core_selenium-test	N/A	2 yr 0 mo - #18	12 min	
		infra_backend-merge-all-repo	1 mo 8 days - #138	3 days 13 hr - #143	5 hr 8 min	
		infra_backend-war-size-tracker	1 mo 9 days - #731	12 hr - #770	1 min 11 sec	
		infra_commit_history_generation	4 mo 22 days - #421	22 hr - #562	4 min 8 sec	
		infra_extension-indexer	1 mo 10 days - #128	4 days 1 hr - #134	3 hr 23 min	
		infra_github_repository_list	1 mo 9 days - #1339	12 hr - #1378	4 min 39 sec	
		infra_plugin_changes_report	8 mo 0 days - #304	2 days 8 hr - #340	13 min	
		infra_plugins_svn_to_git	4 yr 0 mo - #593	3 yr 12 mo - #768	4 min 54 sec	
		infra_svnsync	3 yr 9 mo - #21199	3 yr 9 mo - #21243	1.5 sec	
		infra_sync_maven-hpi-plugin_www	N/A	2 hr 39 min - #376	2.7 sec	
		jenkins_pom	1 mo 14 days - #264	2 days 20 hr - #270	47 sec	
		jenkins_ui-changes_branch	2 yr 5 mo - #32	2 yr 1 mo - #33	4 min 55 sec	
		lib-jira-api	9 mo 25 days - #5354	6 mo 21 days - #5355	58 sec	
		libs_synkit	N/A	2 yr 9 mo - #11	19 sec	
		selenium-tests	N/A	2 yr 0 mo - #11	22 sec	

Icon: [S](#) [M](#) [L](#)

Legend

RSS for all

RSS for failures

RSS for just latest builds



Jenkins - tests

[Pipeline Examples \(jenkins.io\)](https://jenkins.io)



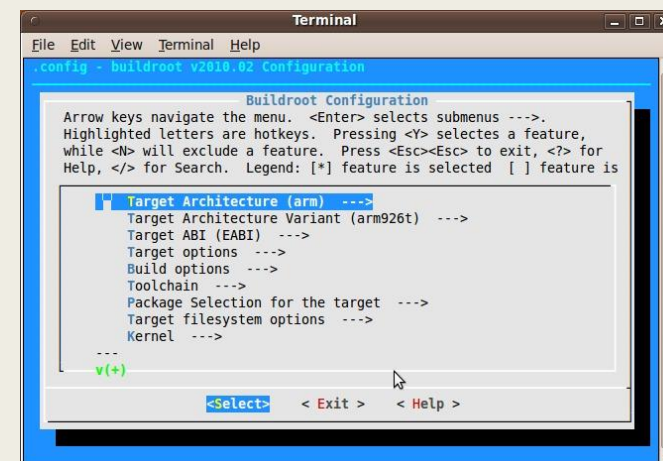


Buildroot



Buildroot is a set of Makefiles and patches that **simplifies and automates** the process of building a complete and bootable Linux environment for an embedded system, while using cross-compilation to allow building for multiple target platforms on a single Linux-based development system.

Buildroot peut créer automatiquement la chaîne d'outils de compilation croisée requise, créer un système de fichiers racine, compiler une image du noyau Linux et générer un chargeur de démarrage pour le système embarqué ciblé, ou il peut effectuer toute combinaison indépendante de ces étapes.

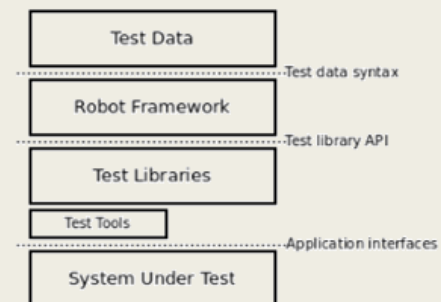


Robot Framework



Robot Framework est un framework d'automatisation open source générique. Il peut être utilisé pour l'automatisation des tests et l'automatisation robotisée des processus (RPA).

Robot Framework a une syntaxe simple, utilisant des mots-clés lisibles par l'homme. Ses capacités peuvent être étendues par des bibliothèques implémentées avec Python, Java ou bien d'autres langages de programmation. Robot Framework dispose d'un écosystème riche, composé de bibliothèques et d'outils développés en tant que projets distincts.

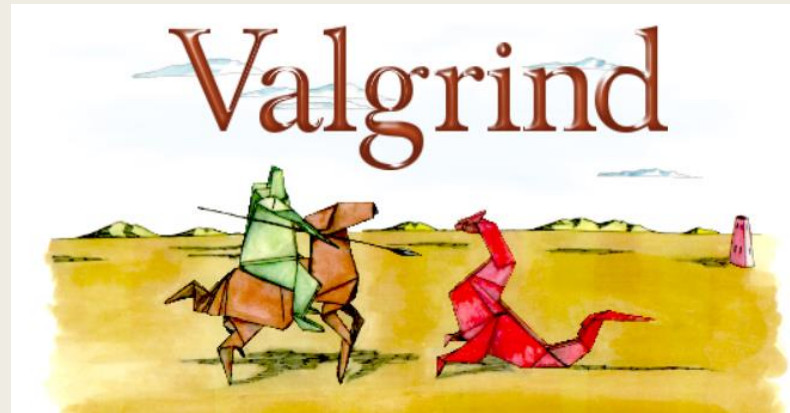


[Robot Framework](#)

```
TestSuite.robot  mots-clés.ressource  CustomLibrary.py  COURIR ►

Exécuter Test Suite
1 Paramètres***
2 Documentation Une suite de tests pour une connexion valide.
3 ...
4 ... Les mots-clés sont importés à partir du fichier de recherche
5 Mots-clés de la ressource.ressource
6 Balises par défaut positives
7
8 Cas de test ***
Exécuter le test
9 Utilisateur de connexion avec mot de passe
10 Se connecter au serveur
11 Login Utilisateur ironman 1234567890
12 Vérifier la validité de l'identifiant Tony Stark
13 [Démontage] Fermer la connexion au serveur
14
Exécuter le test
15 Connexion refusée avec un mot de passe erroné
16 [Balises] négatif
17 Se connecter au serveur
18 Exécuter le mot-clé et s'attendre à une erreur *Mot de passe invalide Utilisateur de connexion ironr
19 Vérifier l'accès non autorisé
20 [Démontage] Fermer la connexion au serveur
```

Valgrind

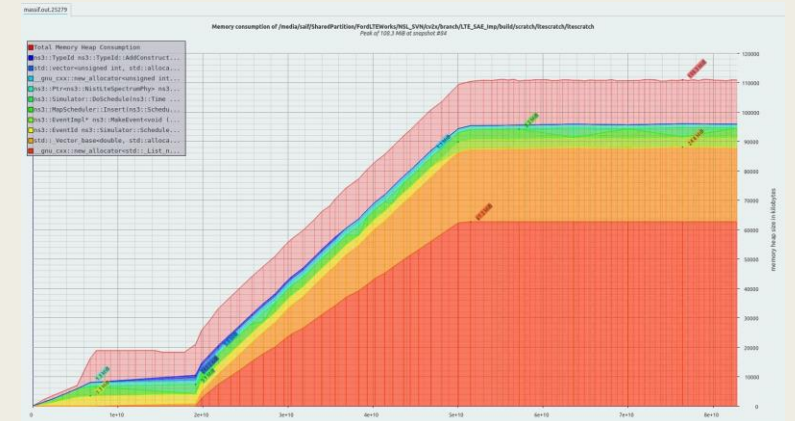


Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

La distribution de Valgrind comprend actuellement sept outils :

- un détecteur d'erreurs mémoire,
- deux détecteurs d'erreurs de thread,
- un cache et branch-prediction profiler,
- un graphe d'appel générant du cache et profileur de prédiction de branche,
- et deux profileurs de tas différents.
- Il comprend également : un générateur expérimental de vecteurs de blocs de base SimPoint.

Il fonctionne sur les éléments suivants plates-formes : X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x et versions ultérieures), ARM64/Android, X86/Android (4.0 et versions ultérieures), MIPS32/Android, X86/FreeBSD, AMD64/FreeBSD, X86/Darwin et AMD64/Darwin (Mac OS X 10.12).



Selenium (web)

Il s'agit principalement d'automatiser des applications Web à des fins de test, mais cela ne se limite certainement pas à cela.

Les tâches d'administration ennuyeuses basées sur le Web peuvent (et devraient) également être automatisées.



MERCI

- On passe au TP

