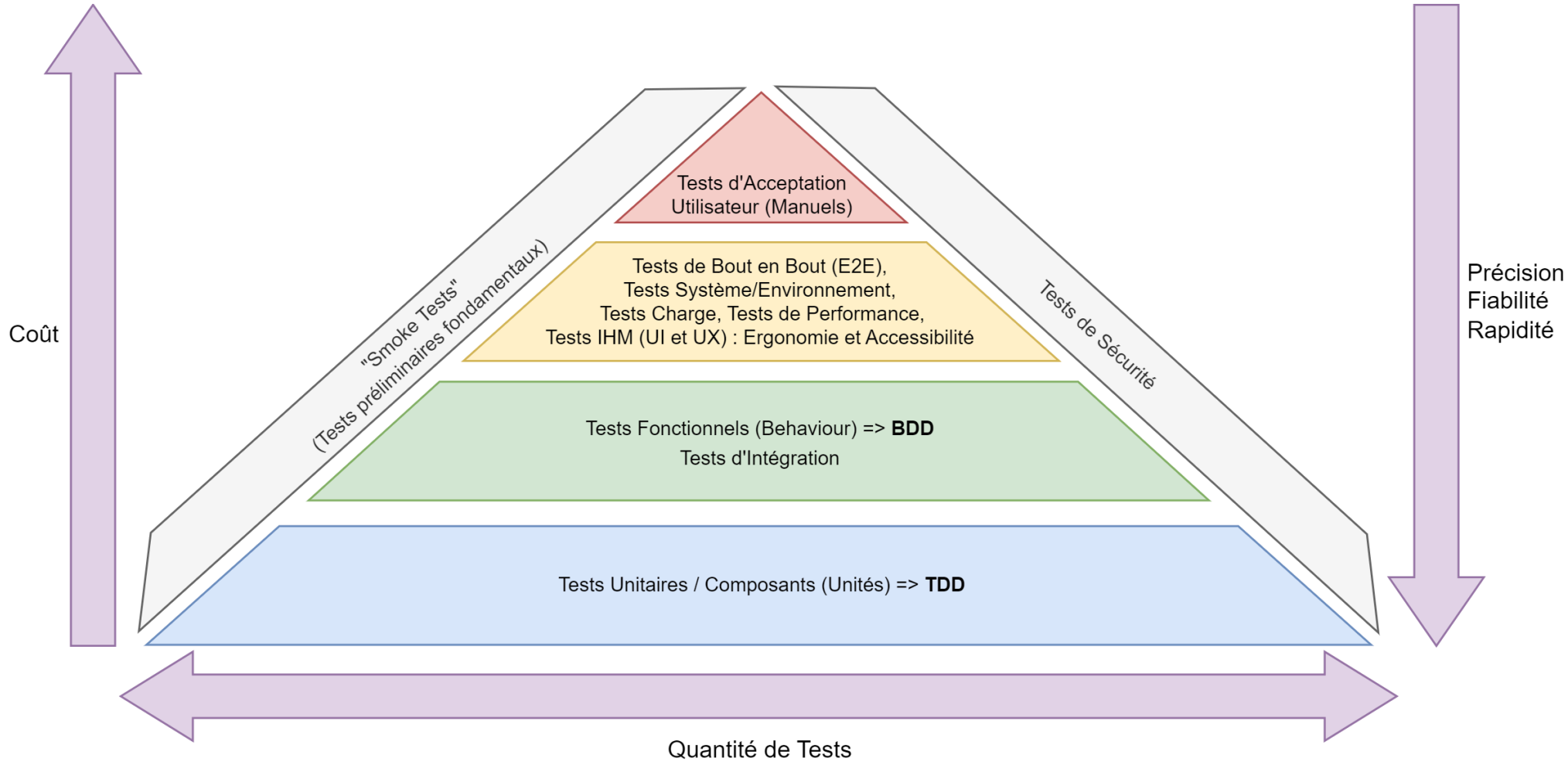
CSharp Perfectionnement

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Tests Unitaires en C#

Différents types de Tests / Pyramide des tests



Pourquoi les tests unitaires ?

- Garantissent la **fiabilité** et la **maintenabilité** du code.
- Détectent rapidement les **régressions** et **erreurs**.
- Favorisent une **meilleure compréhension** et **documentation** du code.

Frameworks de tests .NET

- Il existe en .NET 3 frameworks principaux pour les tests Unitaires :
  - **MSTest**
  - **NUnit**
  - **XUnit**
- Pour faire des tests unitaires, on **créera un projet** de type "**Bibliothèque de Tests**", il fera **référence au projet C#** que l'**on veut tester**.

MSTest

```
[TestClass]
public class MyTests {
    [TestMethod]
    public void TestMethod1() {
        Assert.AreEqual(4, 2 + 2);
    }
}
```

NUnit

```
[TestFixture]
public class MyTests {
    [Test]
    public void TestMethod1() {
        Assert.That(4, Is.EqualTo(2 + 2));
    }
}
```

XUnit

```
public class MyTests {
    [Fact]
    public void TestMethod1() {
        Assert.Equal(4, 2 + 2);
    }
}
```

Comparatif syntaxique

Aspect	xUnit	NUnit	MSTest
Test simple	`[Fact]`	`[Test]`	`[TestMethod]`
Test paramétré	`[InlineData]`	`[TestCase]`	`[DataRow]`
Initialisation globale	Non disponible	`[OneTimeSetUp]`	`[ClassInitialize]`
Nettoyage global	Non disponible	`[OneTimeTearDown]`	`[ClassCleanup]`
Initialisation avant chaque test	Non disponible	`[SetUp]`	`[TestInitialize]`
Nettoyage après chaque test	Non disponible	`[TearDown]`	`[TestCleanup]`

Comparatif syntaxique

Aspect	xUnit	NUnit	MSTest
Ignorer un test	`[Fact(Skip="raison")]`	`[Ignore("raison")]`	`[Ignore]`
Tests paramétrés avancés	`[MemberData]`	`[TestCaseSource]`	`[DynamicData]`
Ordre d'exécution	`[TestCaseOrderer]`	`[Order]`	Non disponible
Catégories de tests	Non disponible	`[Category("nom")]`	`[TestCategory("nom")]`
Timeout pour un test	Non disponible directement	`[Timeout(ms)]`	`[Timeout(ms)]`

Comment bien écrire un test unitaire ?

- **Caractéristiques d'un bon test unitaire**
  - **Indépendant** : Un test ne doit **pas dépendre des autres**.
  - **Précis** : Chaque test **vérifie un cas spécifique**.
  - **Automatisé** : **Exécuté facilement** dans une suite de tests.
  - **Rapide** : Pour permettre des **itérations rapides**.
  - **Nom explicite** : Le **nom** du test doit **indiquer clairement son objectif**. => Conventions de nommage
- Recommandations Microsoft

AAA (Arrange, Act, Assert)

Il convient de décomposer un test en 3 phases :

- **Arrange** : **Prépare** les données nécessaires au test.
- **Act** : **Exécute** la logique à tester.
- **Assert** : **Vérifie** le résultat attendu.

```
[TestMethod]
public void CalculateTotal_ShouldReturnCorrectValue() {
    // Arrange
    var calculator = new Calculator();
    int expected = 4;

    // Act
    int result = calculator.CalculateTotal(2, 2);

    // Assert
    Assert.AreEqual(expected, result);
}
```

Convaincre les développeurs de l'utilité des tests unitaires

Arguments en faveur des tests unitaires

- **Détection précoce des erreurs** : Moins de bugs en production, ce qui réduit les coûts, surtout avec l'utilisation de CI/CD.
- **Confiance dans les modifications** : Les tests assurent la non-régression.
- **Documentation vivante** : Les tests montrent des exemples d'utilisation du code.
- **Maintenance facilitée** : Les tests facilitent les refactorisations et améliorent la qualité du code.

Convaincre les développeurs de l'utilité des tests unitaires

Démontrer l'efficacité

- Partager des exemples concrets de bugs évités grâce aux tests.
- Montrer comment les tests peuvent accélérer le développement sur le long terme.
- Encourager une culture de tests via des révisions de code et des sessions de pair programming.

Mocking

- Permet de créer des **objets simulés** pour **tester sans les dépendances**.
- Facilite les tests en isolant **la logique métier des dépendances externes** (autre classe, base de données, API, etc.).
- **Essentiels** pour **se concentrer** sur **un aspect** (une unité) du code **à la fois** et ainsi **respecter** le principe de Test **Unitaire**.

Les Frameworks de Mocking

- **Moq** : Simple et flexible, souvent utilisé pour les interfaces et les services.
- **NSubstitute** : Syntaxe intuitive, permet de créer des substituts dynamiques.
- **FakeItEasy** : Design minimaliste et syntaxe expressive.

Exemple de Moq

```
var mockRepository = new Mock<IRepository>();
mockRepository.Setup(repo => repo.GetData()).Returns("Mocked Data");

var service = new MyService(mockRepository.Object);
string result = service.GetData();

Assert.AreEqual("Mocked Data", result);
```

Principes de Mocking

- **Stubbing** : Créer des comportements spécifiques pour des méthodes.
- **Verification** : Valider que certaines méthodes ont été appelées.
- **Isolation des tests** : Permet de tester une unité de code en se concentrant uniquement sur sa logique.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

TDD

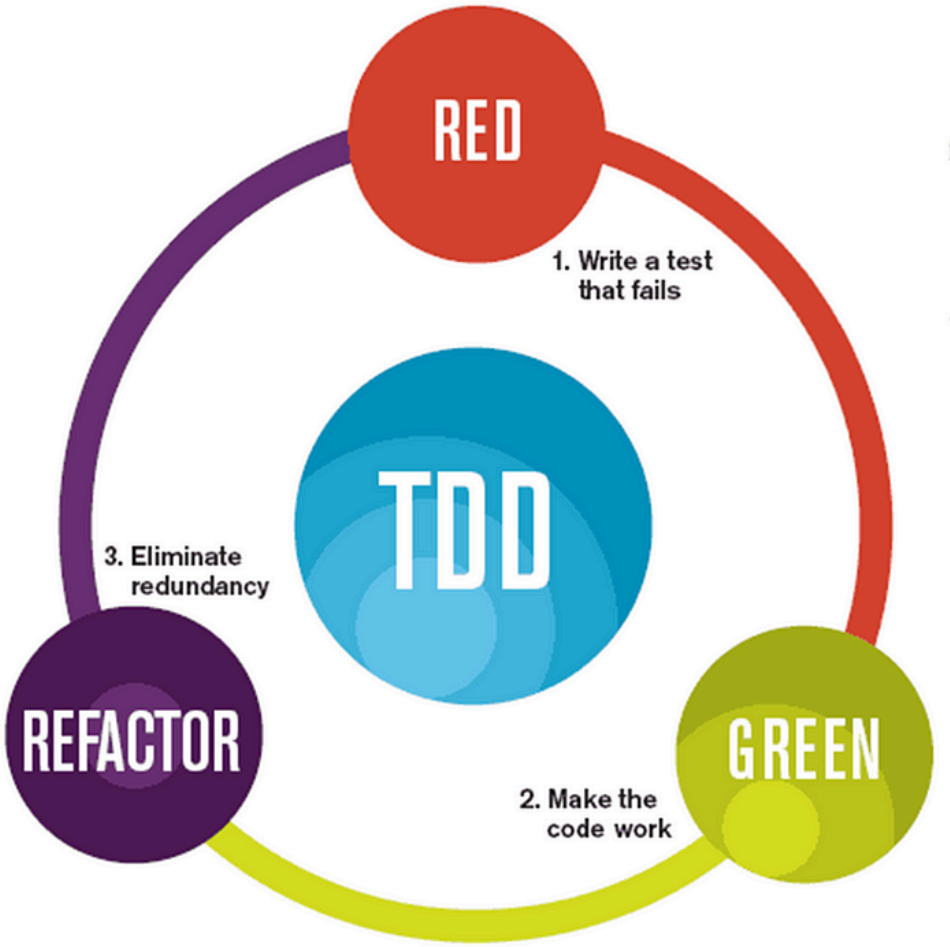
Les paradigmes du TDD

Le **Test Driven Development (TDD)**, ou Développement Dirigé par les Tests, consiste à écrire des tests avant le code de production.
En .NET, le processus suit généralement ces étapes :

- Red
- Green
- Refactor

Red Green Refactor

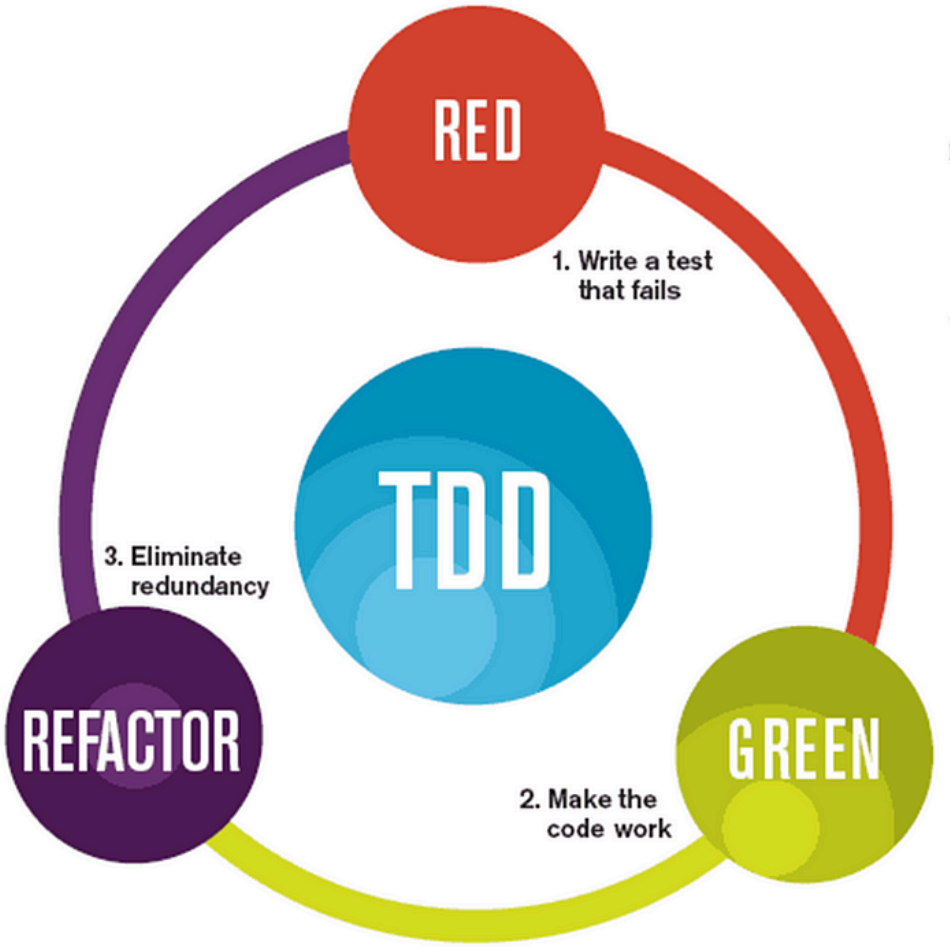
- **Écrire un test** : Rédigez un test pour la fonctionnalité cible. Ce test échoue initialement, car le code de production est inexistant.
- **Exécuter le test pour vérifier son échec** : Cela confirme que le test est valide et qu'il vérifie la bonne condition.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Red Green Refactor

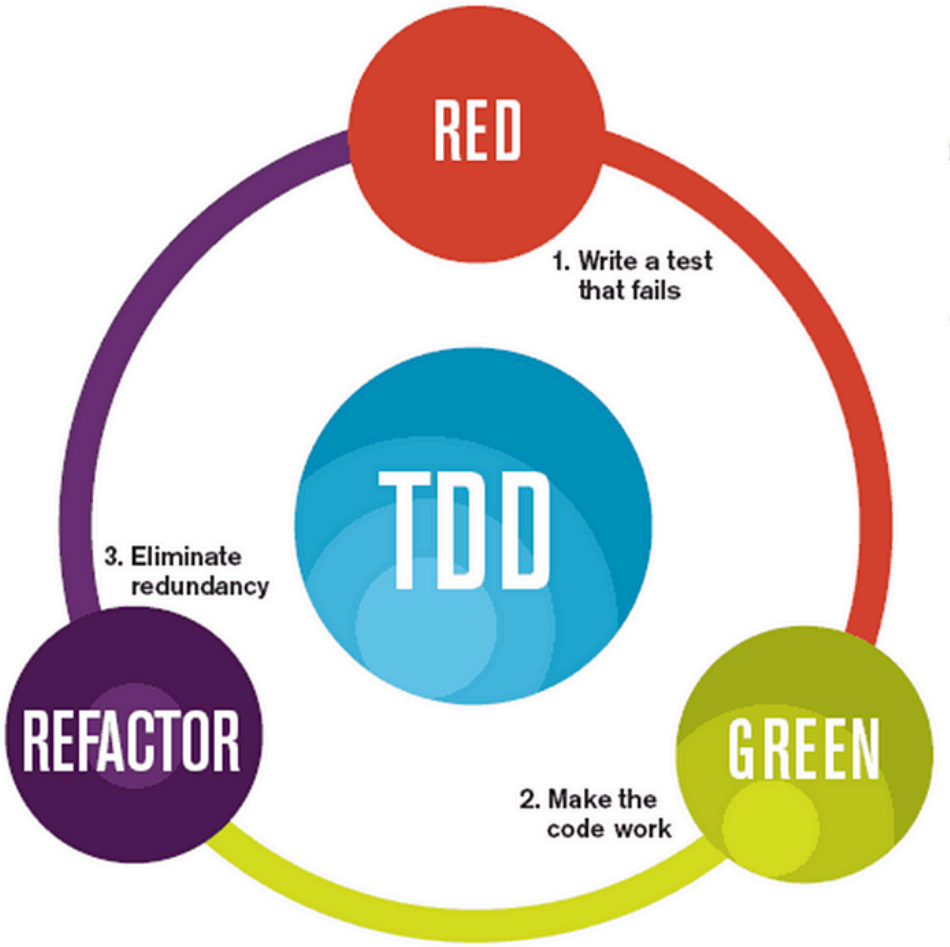
- **Écrire le code de production** : Écrivez juste assez de code pour que le test réussisse, sans chercher à optimiser.
- **Exécuter les tests** : Vérifiez que le nouveau test passe et que les anciens tests restent valides.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Red Green Refactor

- **Refactoriser le code** : Améliorez la structure du code tout en conservant ses fonctionnalités. Après chaque refactorisation, exécutez les tests.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Les bonnes pratiques du TDD

- **Comprendre les exigences** : Assurez-vous de bien comprendre ce que le code doit accomplir avant de commencer.
- **Tests simples** : Chaque test doit être indépendant et vérifier une seule fonctionnalité.
- **Éviter les anticipations** : Restez concentré sur les besoins actuels, sans coder pour des besoins futurs (principe **YAGNI**).
- **Code minimum** : Écrivez uniquement le code nécessaire pour que le test passe.

Les bonnes pratiques du TDD

- **Refactoriser régulièrement** : Simplifiez et améliorez le code une fois les tests validés, puis réexécutez-les.
- **Automatiser les tests** : Exécutez-les automatiquement à chaque modification pour détecter rapidement les problèmes.
- **Utiliser des doubles de test** : Simulez les dépendances avec des mocks, stubs ou objets factices pour isoler le code testé.
- **Exécution régulière** : Lancez les tests fréquemment pour maintenir leur efficacité.

Les bonnes pratiques du TDD

- **Bonne couverture** : Testez autant de parties du code que possible, sans viser systématiquement 100%.
- **Tester à différents niveaux** : Incluez tests unitaires, d'intégration, système, et d'acceptation.

Principe FIRST

F - **Fast (Rapide)** : Les tests doivent s'exécuter rapidement pour être utilisés fréquemment.

I - **Independent (Indépendant)** : Aucun test ne doit dépendre d'un autre.

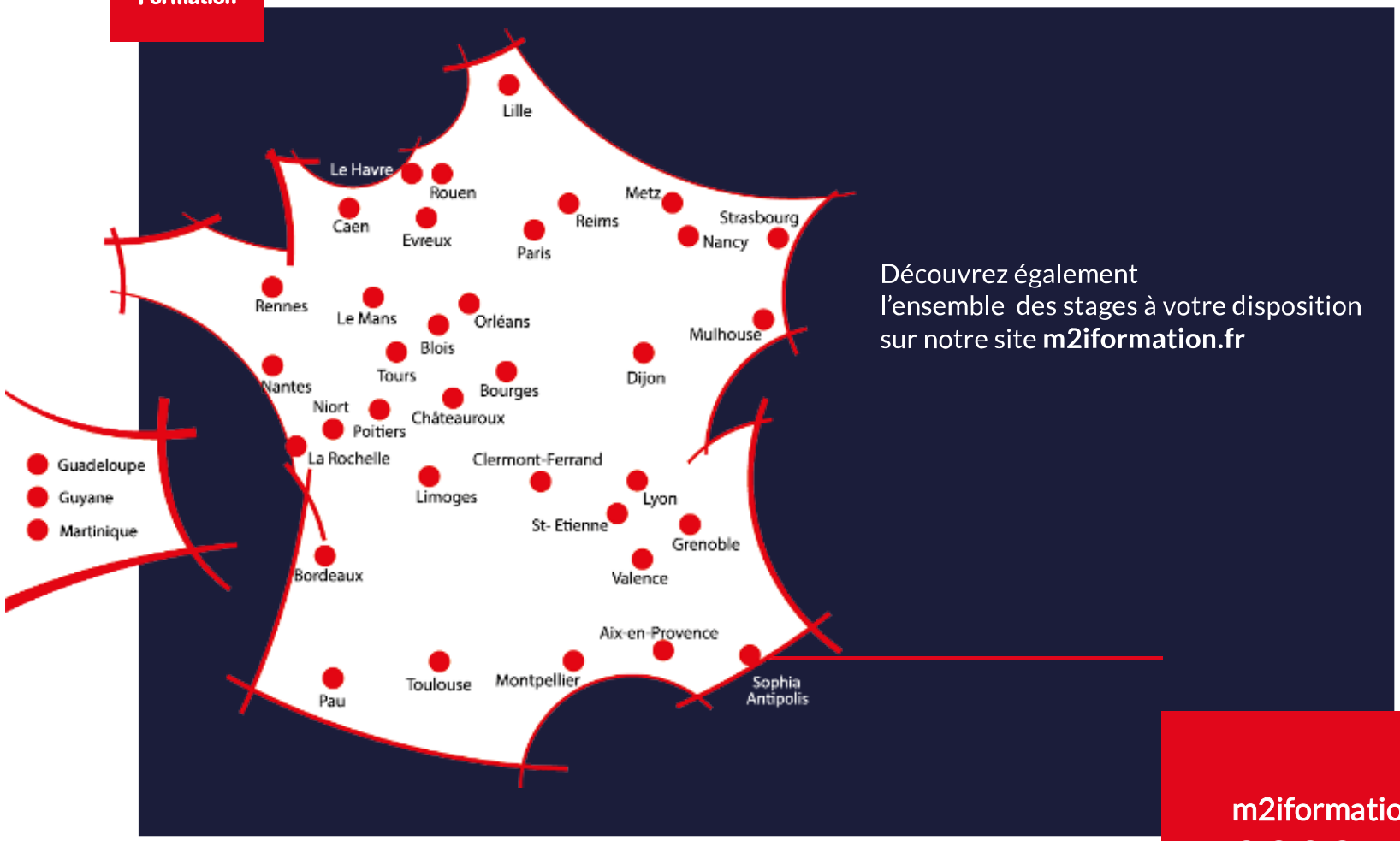
R - **Repeatable (Reproductible)** : Les tests doivent fournir les mêmes résultats, quel que soit l'environnement.

S - **Self-validating (Auto-vérifiable)** : Les résultats doivent être clairs (succès/échec), sans analyse manuelle.

T - **Timely (Opportun)** : Les tests doivent être écrits en parallèle ou avant le code de production.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr