KEYCE Informatique & Intelligence Artificielle ISKIIA

INTRODUCTION À LA VÉRIFICATION ET LA VALIDATION DES LOGICIELS

2

CHAP I

DÉFINITION DES CONCEPTS DE VÉRIFICATION ET DE VALIDATION.

DÉFINITION DES CONCEPTS DE VÉRIFICATION ET DE VALIDATION.

- **Vérification : Faire les choses correctement**

La vérification, c'est l'assurance que nous faisons les choses correctement. Elle consiste à s'assurer que notre logiciel est développé conformément aux spécifications, aux normes et aux meilleures pratiques. En d'autres termes, la vérification nous aide à garantir que notre logiciel est techniquement correct.

- **Validation : Faire les bonnes choses**

D'un autre côté, la validation se concentre sur le fait de faire les bonnes choses. Elle vise à s'assurer que notre logiciel répond aux besoins et aux attentes de ses utilisateurs. La validation est la garantie que nous construisons le bon produit, pas seulement un produit correct.

Ces deux concepts, vérification et validation, sont intégrés tout au long du cycle de vie du développement logiciel. Depuis la phase de planification jusqu'à l'exécution des tests et au-delà, la V&V joue un rôle crucial

OBJECTIFS DES CONCEPTS DE VÉRIFICATION ET DE VALIDATION.

- **La Vérification**

Les objectifs de la vérification incluent la garantie que notre logiciel est développé techniquement correct. Cela signifie s'assurer que le code est conforme aux spécifications, qu'il est dépourvu d'erreurs techniques et qu'il respecte les normes de l'industrie. Pour atteindre ces objectifs, nous utilisons des techniques telles que les revues de code et les tests unitaires

- **La Validation**

D'un autre côté, les objectifs de la validation sont axés sur les utilisateurs. La validation garantit que notre logiciel répond à leurs besoins. Cela signifie qu'il offre les fonctionnalités attendues, une expérience utilisateur satisfaisante, et qu'il résout réellement les problèmes pour lesquels il a été créé. Les tests d'acceptation, les évaluations de l'expérience utilisateur et d'autres techniques sont utilisés pour atteindre ces objectifs.

LA DIFFÉRENCE FONDAMENTALE ENTRE LA VÉRIFICATION ET LA VALIDATION.

- **Vérification** :
  1. La vérification se concentre sur "faire les choses correctement".
  2. Elle vise à s'assurer que le logiciel est développé conformément aux spécifications, aux normes, aux règles de codage, etc.
  3. La vérification porte sur des aspects techniques, tels que la structure du code, la syntaxe, l'absence d'erreurs de programmation, et la conformité aux normes de l'industrie.
  4. Les techniques de vérification incluent les revues de code, l'analyse statique, les tests unitaires, etc.
  5. L'objectif de la vérification est d'assurer que le logiciel est techniquement correct.

- **Validation** :
  1. La validation se concentre sur "faire les bonnes choses".
  2. Elle vise à s'assurer que le logiciel répond aux besoins, aux attentes et aux exigences des utilisateurs.
  3. La validation porte sur des aspects fonctionnels, tels que l'adéquation aux besoins, la convivialité, la satisfaction de l'utilisateur et l'efficacité dans la résolution des problèmes.
  4. Les techniques de validation incluent les tests d'acceptation, les évaluations de l'expérience utilisateur, les scénarios d'utilisation, etc.
  5. L'objectif de la validation est de garantir que le logiciel est le bon produit à construire, en répondant aux besoins réels des utilisateurs.

VÉRIFICATION ET DE VALIDATION.

- Ces deux concepts, vérification et validation, sont intégrés tout au long du cycle de vie du développement logiciel. Depuis la phase de planification jusqu'à l'exécution des tests et au-delà, la V&V joue un rôle crucial.
- En conclusion, la vérification et la validation sont des éléments essentiels du développement logiciel. La vérification garantit que nous construisons le logiciel correctement, tandis que la validation assure que nous construisons le bon logiciel. Ensemble, ces processus contribuent à la qualité et à la satisfaction des utilisateurs

IMPORTANCE DE LA V&V DANS LE DÉVELOPPEMENT.

IMPORTANCE DE LA V&V DANS LE DÉVELOPPEMENT.

- Pourquoi la Vérification et la Validation (V&V) sont-elles cruciales dans le développement logiciel ? Dans cette section, nous allons explorer l'importance fondamentale de la V&V et son rôle dans la création de logiciels de haute qualité.
- **Garantir la Fiabilité Technique**
- L'une des raisons essentielles de la V&V est de garantir la fiabilité technique de votre logiciel.
- La Vérification s'assure que le logiciel est construit correctement du point de vue technique : il respecte les spécifications, les normes et les meilleures pratiques.
- Cela réduit les risques d'erreurs de programmation, de bogues et de problèmes de compatibilité.

IMPORTANCE DE LA V&V DANS LE DÉVELOPPEMENT.

- **Prévenir les Erreurs et les Coûts**
- La V&V prévient les erreurs précocement, ce qui permet d'économiser du temps et de l'argent.
- En détectant et en corrigeant les problèmes dès le début du processus de développement, vous évitez des coûts plus importants à des étapes ultérieures.
- Cela contribue à la qualité, à la productivité et à la satisfaction du client.
- **Assurer la Satisfaction des Utilisateurs**
- La validation est essentielle pour s'assurer que votre logiciel répond aux besoins et aux attentes des utilisateurs.
- Lorsque le logiciel est validé, il offre une expérience utilisateur positive, ce qui est crucial pour la satisfaction des clients.
- La V&V conduit à des produits qui sont plus utiles, plus efficaces et qui résolvent réellement les problèmes des utilisateurs.

IMPORTANCE DE LA V&V DANS LE DÉVELOPPEMENT.

- **Réduire les Risques en Sécurité**
- La Vérification et la Validation sont essentielles pour garantir la sécurité du logiciel.
- Elles aident à identifier et à corriger les vulnérabilités de sécurité, réduisant ainsi les risques de piratage ou d'accès non autorisé.
- Cela est particulièrement crucial dans les applications sensibles, telles que la santé, les finances et la sécurité.
- **Respecter les Normes de l'Industrie**
- La conformité aux normes de l'industrie est souvent exigée, et la V&V est le moyen de s'assurer que votre logiciel les respecte.
- Cela est particulièrement important dans les secteurs réglementés, tels que la santé, où la conformité est obligatoire.

En résumé, la Vérification et la Validation sont les gardiennes de la qualité du logiciel. Elles garantissent la fiabilité technique, préviennent les erreurs, assurent la satisfaction des utilisateurs, réduisent les risques en sécurité et assurent la conformité aux normes. En conséquence, elles jouent un rôle essentiel dans la création de logiciels de haute qualité, tout en évitant des coûts et des problèmes potentiels à l'avenir.

MÉTHODOLOGIES ET APPROCHES DE LA V&V.

MÉTHODOLOGIES ET APPROCHES DE LA V&V.

Dans cette section, nous allons explorer les différentes méthodologies et approches de la Vérification et de la Validation (V&V). Ces approches sont essentielles pour mettre en œuvre la V&V de manière efficace et systématique.

- **Approches de la Vérification**

Commençons par les approches de la Vérification, qui se concentrent sur l'assurance que le logiciel est correct du point de vue technique.

- Les revues de code : Une méthode de vérification où le code est examiné par les pairs pour détecter des erreurs ou des non-conformités.
- Les tests unitaires : Des tests automatisés qui vérifient le bon fonctionnement de parties spécifiques du code.

- **Approches de la Validation**

- La Validation se concentre sur l'assurance que le logiciel répond aux besoins des utilisateurs.
- Les tests d'acceptation : Des tests qui évaluent si le logiciel satisfait les critères d'acceptation définis par les utilisateurs finaux.
- L'évaluation de l'expérience utilisateur : Une approche qui mesure la convivialité et la satisfaction de l'utilisateur lors de l'utilisation du logiciel.

MÉTHODOLOGIES ET APPROCHES DE LA V&V.

- **Intégration de la V&V dans le Cycle de Vie**
- La V&V doit être intégrée à chaque étape du cycle de vie du développement logiciel.
- Planification de la V&V : Établir une stratégie pour la V&V dès le début du projet.
- Exécution de la V&V : Appliquer les méthodologies de manière cohérente pendant le développement."
- Évolution continue : La V&V est un processus itératif qui s'étend tout au long du cycle de vie du logiciel.
- **Outils de V&V**
- De nombreux outils sont disponibles pour soutenir la V&V.
- Outils de revue de code : Facilitent l'examen et la détection des erreurs dans le code source."
- Outils de test automatisés : Permettent d'automatiser les tests unitaires, les tests d'intégration et d'autres types de tests.
- Outils de suivi des bogues : Aident à gérer et à suivre les problèmes identifiés lors de la V&V.

MÉTHODOLOGIES ET APPROCHES DE LA V&V.

- **Méthodologies Agiles et V&V**
- Les méthodologies Agiles, telles que Scrum et Kanban, encouragent la V&V tout au long du processus de développement.
- Des itérations fréquentes de développement, des tests continus et une collaboration étroite avec les utilisateurs sont des éléments clés des méthodologies Agiles.

En conclusion, les méthodologies et les approches de la Vérification et de la Validation sont essentielles pour garantir la qualité des logiciels. En combinant des méthodes de vérification solides avec des approches de validation orientées utilisateur, et en intégrant la V&V à chaque étape du cycle de vie, nous pouvons développer des logiciels de haute qualité qui répondent aux besoins des utilisateurs de manière efficace et fiable.

TECHNIQUES DE TEST LOGICIEL

16

CHAP II

INTRODUCTION TECHNIQUES DE TEST LOGICIEL

- Le test logiciel est une étape cruciale dans le développement de logiciels de haute qualité. Il s'agit du processus systématique qui vise à évaluer un logiciel pour garantir qu'il fonctionne correctement, qu'il répond aux exigences et qu'il est exempt d'erreurs.
- Les techniques de test logiciel sont les méthodes et les approches utilisées pour planifier, concevoir, exécuter et évaluer ces tests. Elles jouent un rôle essentiel dans l'assurance de la qualité du logiciel et dans la réduction des risques potentiels pour les utilisateurs finaux.
- L'importance du test logiciel réside dans la complexité croissante des logiciels que nous utilisons au quotidien. Les logiciels sont devenus omniprésents, alimentant nos smartphones, nos voitures, nos systèmes de santé et bien d'autres domaines de notre vie. Les conséquences des logiciels défectueux sont significatives, allant des pertes financières aux atteintes à la réputation de l'entreprise, voire aux risques pour la sécurité des utilisateurs.
- Les techniques de test logiciel offrent un moyen structuré d'identifier et de corriger les erreurs avant qu'elles ne deviennent des problèmes graves. Elles incluent diverses approches, des tests fonctionnels qui vérifient que le logiciel fait ce qu'il est censé faire, aux tests non fonctionnels qui évaluent des aspects tels que la performance, la sécurité et la compatibilité.

TYPES DE TEST LOGICIEL.

TYPES DE TEST LOGICIEL

Il existe de nombreux types de tests logiciels, chacun ayant des objectifs spécifiques pour évaluer différentes facettes du logiciel. Voici une liste des types de tests logiciels les plus courants :

1. **Tests Unitaires :** Ces tests se concentrent sur des parties spécifiques du code source, généralement des fonctions ou des méthodes individuelles. Ils visent à s'assurer que chaque unité de code fonctionne correctement.
2. **Tests d'Intégration :** Les tests d'intégration vérifient la manière dont différentes unités de code s'assemblent et interagissent. Ils garantissent que les composants du logiciel fonctionnent bien ensemble.
3. **Tests de Système :** Ces tests évaluent l'ensemble du système logiciel pour s'assurer qu'il répond aux exigences et qu'il fonctionne comme une entité cohérente.
4. **Tests de Récupération :** Ces tests évaluent la capacité du logiciel à récupérer après une panne, un plantage ou une perte de données.

TYPES DE TEST LOGICIEL

5. **Tests de Performance :** Les tests de performance mesurent les performances du logiciel en termes de vitesse, d'efficacité, de consommation de ressources et de capacité à gérer des charges de travail élevées.
6. **Tests de Stress** : Les tests de stress poussent le logiciel au-delà de ses limites pour évaluer son comportement en cas de charges excessives ou de conditions de fonctionnement inhabituelles.
7. **Tests de Charge :** Ils visent à déterminer le point au-delà duquel le logiciel commence à montrer des signes de sous-performance en raison d'une charge importante.
8. **Tests de Sécurité :** Ces tests évaluent la résistance du logiciel aux vulnérabilités et aux attaques, en identifiant les failles potentielles.
9. **Tests de Compatibilité :** Les tests de compatibilité vérifient que le logiciel fonctionne correctement sur différentes plates-formes, navigateurs, systèmes d'exploitation, etc.

TYPES DE TEST LOGICIEL

10. **Tests d'Acceptation :** Ces tests sont effectués pour vérifier que le logiciel répond aux attentes du client ou de l'utilisateur final. Ils peuvent inclure des tests d'acceptation utilisateur (UAT) et des tests d'acceptation fonctionnelle (FAT).
11. **Tests de Non-Régression :** Après des modifications ou des mises à jour du logiciel, les tests de non-régression s'assurent que les fonctionnalités existantes ne sont pas affectées négativement.
12. **Tests d'Interface Utilisateur :** Ces tests vérifient que l'interface utilisateur du logiciel est conviviale, cohérente et répond aux exigences en matière d'ergonomie.
13. **Tests de Montée en Charge :** Ils évaluent la capacité du logiciel à gérer une augmentation progressive de la charge de travail.
14. **Tests de Vérification et de Validation :** Ils valident que le logiciel répond aux spécifications et aux besoins du client, en mettant l'accent sur la conformité aux exigences.
15. **Tests de Cycle de Vie :** Ces tests évaluent le logiciel à différentes étapes de son cycle de vie, de la conception à la maintenance.

TYPES DE TEST LOGICIEL COURANTS

- Parmi les nombreux types de tests logiciels, les cinq les plus couramment utilisés sont les suivants :

1. **Tests Unitaires :** Les tests unitaires sont essentiels et courants. Ils se concentrent sur des parties spécifiques du code source, vérifiant si chaque unité de code (comme une fonction ou une méthode) fonctionne correctement de manière isolée.
2. **Tests d'Intégration :** Les tests d'intégration sont utilisés pour vérifier comment différentes unités de code s'assemblent et interagissent. Ils s'assurent que les composants du logiciel fonctionnent correctement ensemble.
3. **Tests de Système :** Les tests de système sont incontournables pour évaluer le logiciel dans son ensemble, garantissant qu'il répond aux exigences et qu'il fonctionne comme une entité cohérente.
4. **Tests de Non-Régression :** Ces tests sont fréquemment utilisés après des modifications ou des mises à jour du logiciel. Ils s'assurent que les fonctionnalités existantes ne sont pas affectées négativement.
5. **Tests d'Acceptation :** Les tests d'acceptation, qu'ils soient des tests d'acceptation utilisateur (UAT) ou des tests d'acceptation fonctionnelle (FAT), sont essentiels pour vérifier que le logiciel répond aux attentes du client ou de l'utilisateur final.

TESTS UNITAIRES

Un test unitaire est une pratique de test dans le développement logiciel qui vise à vérifier le bon fonctionnement d'une unité individuelle de code, telle qu'une fonction, une méthode ou une classe, de manière isolée. L'objectif est de s'assurer que chaque unité du logiciel remplit correctement sa fonction, produisant les résultats attendus pour un ensemble donné d'entrées.

- **Pourquoi les Tests Unitaires Sont Importants :**
  1. **Isolation des Problèmes :** Les tests unitaires permettent d'isoler les problèmes potentiels à un niveau très granulaire. Si un test échoue, il est plus facile de localiser et de corriger la source du problème.
  2. **Facilité de Maintenance :** Les tests unitaires aident à documenter le comportement attendu des unités de code. Ils servent de référence pour les développeurs qui peuvent ainsi comprendre rapidement comment une unité de code est censée fonctionner.
  3. **Détection Précoce des Erreurs :** Les tests unitaires peuvent être exécutés fréquemment, ce qui permet de détecter les erreurs tôt dans le processus de développement, ce qui réduit les coûts de correction.
  4. **Amélioration de la Conception :** L'écriture de tests unitaires oblige souvent à penser à la manière dont les unités de code sont conçues et interagissent, ce qui conduit à une meilleure architecture logicielle.

TESTS UNITAIRES

- **Caractéristiques des Tests Unitaires :**
  - **Indépendance :** Chaque test unitaire doit être indépendant des autres, ce qui signifie qu'il ne doit pas dépendre du résultat d'un autre test. Cela garantit que les tests sont reproductibles et isolés.
  - **Automatisation :** Les tests unitaires doivent être automatisés pour pouvoir être facilement exécutés à plusieurs reprises. Des outils tels que JUnit pour Java, pytest pour Python ou xUnit pour divers langages facilitent l'automatisation.
  - **Rapidité :** Les tests unitaires doivent être rapides à exécuter, ce qui les rend adaptés pour une utilisation fréquente pendant le développement.
  - **Répétabilité :** Un bon test unitaire doit produire des résultats cohérents à chaque exécution, quelle que soit la machine sur laquelle il est exécuté.
- **Comment Écrire un Test Unitaire :**
  1. **Configuration :** Préparez l'environnement de test en instanciant les objets nécessaires et en configurant les données.
  2. **Appel de la Fonction à Tester :** Appelez la fonction ou la méthode que vous souhaitez tester avec des données d'entrée spécifiques.
  3. **Vérification :** Utilisez des assertions pour vérifier que la sortie de la fonction est conforme à ce que vous attendez.
  4. **Nettoyage :** Rétablissez l'environnement de test à son état initial.

TESTS UNITAIRES

- Un exemple de test unitaire en Java utilisant le framework de test JUnit

```java
public class Calculatrice {

    public int additionner(int a, int b) {
        return a + b;
    }
}
```

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalculatriceTest {

    @Test
    public void testAddition() {
        // Créez une instance de la classe Calculatrice
        Calculatrice calculatrice = new Calculatrice();

        // Appelez la méthode à tester et stockez le résultat
        int resultat = calculatrice.additionner(3, 5);

        // Utilisez les méthodes d'assertion de JUnit pour vérifier le résul
        assertEquals(8, resultat); // Vérifiez si le résultat est égal à 8
    }
}
```

TESTS D'INTÉGRATION

Les tests d'intégration sont une composante essentielle de la stratégie de test logiciel. Ils visent à vérifier comment différentes unités de code interagissent lorsqu'elles sont combinées pour former des composants logiciels plus importants ou un système complet.

- **Objectifs des Tests d'Intégration :**

1. **Validation des Interactions :** Les tests d'intégration s'assurent que les différentes parties du logiciel fonctionnent ensemble harmonieusement. Ils valident que les composants interagissent correctement.
2. **Détection de Problèmes d'Intégration :** Ils identifient les problèmes qui peuvent surgir lorsque plusieurs unités de code sont combinées. Ces problèmes ne sont souvent pas détectables par des tests unitaires individuels.
3. **Amélioration de la Qualité :** Les tests d'intégration contribuent à améliorer la qualité globale du logiciel en s'assurant que toutes les parties du système fonctionnent bien ensemble.

TESTS D'INTÉGRATION

- **Niveaux d'Intégration :**

Il existe plusieurs niveaux d'intégration, allant de l'intégration des composants individuels au sein d'un module ou d'une bibliothèque, jusqu'à l'intégration de systèmes complets. Les niveaux d'intégration courants sont les suivants :

1. **Intégration des Composants :** À ce niveau, des composants individuels tels que des classes ou des fonctions sont combinés et testés pour s'assurer qu'ils fonctionnent bien ensemble.
2. **Intégration de Modules :** Des modules logiciels plus importants, généralement constitués de plusieurs composants, sont intégrés et testés.
3. **Intégration de Systèmes :** À ce niveau, le système complet, y compris toutes les applications et services, est intégré et testé.
4. **Intégration d'Entreprises :** Dans certains cas, des tests d'intégration sont effectués pour s'assurer que différents systèmes d'entreprises interagissent correctement.

TESTS D'INTÉGRATION

- **Stratégies de Tests d'Intégration :**

1. **Big Bang Integration :** Toutes les unités de code sont intégrées simultanément, puis le système est testé dans son ensemble. Cela peut être risqué, car les erreurs détectées sont difficiles à isoler.
2. **Top-Down Integration :** L'intégration commence par le composant le plus élevé (l'interface utilisateur ou le module principal) et progresse vers le bas. Des simulateurs ou des composants factices peuvent être utilisés pour remplacer les composants qui ne sont pas encore développés.
3. **Bottom-Up Integration :** L'intégration commence par les composants de bas niveau, puis monte vers le composant le plus élevé. Des composants factices peuvent être utilisés pour les parties du système qui ne sont pas encore développées.
4. **Tests d'Intégration Continue :** Les tests d'intégration sont effectués continuellement pendant le processus de développement, chaque fois qu'une nouvelle unité est intégrée. Cela réduit les risques d'erreurs d'intégration tardives.

TESTS D'INTÉGRATION

```java
public class Panier {
    private List<Produit> produits = new ArrayList<>();

    public void ajouterProduit(Produit produit) {
        produits.add(produit);
    }

    public double calculerTotal() {
        double total = 0;
        for (Produit produit : produits) {
            total += produit.getPrix();
        }
        return total;
    }
}
```

```java
public class Produit {
    private String nom;
    private double prix;

    public Produit(String nom, double prix) {
        this.nom = nom;
        this.prix = prix;
    }

    public double getPrix() {
        return prix;
    }
}
```

Dans ce cas, le test d'intégration vérifierait que l'ajout de produits au panier et le calcul du total fonctionnent correctement lorsque les classes **Panier** et **Produit** sont utilisées ensemble.

TESTS DE SYSTÈME

Les tests de système sont une composante cruciale du processus de test logiciel. Ils visent à évaluer le système logiciel dans son ensemble pour s'assurer qu'il fonctionne conformément aux exigences spécifiées. Voici une description plus détaillée des tests de système :

- **Objectifs des Tests de Système :**

1. **Validation des Exigences :** Les tests de système vérifient que le logiciel respecte toutes les exigences fonctionnelles et non fonctionnelles définies dans les spécifications du système. Cela inclut les fonctionnalités attendues, la performance, la sécurité, la convivialité, etc.
2. **Évaluation de l'Intégration :** Ils s'assurent que toutes les composantes du système s'intègrent correctement et interagissent de manière cohérente. Cela inclut la vérification des interfaces entre les modules.
3. **Validation des Scénarios d'Utilisation :** Les tests de système vérifient que le logiciel fonctionne correctement dans des scénarios d'utilisation réalistes. Ils couvrent les flux de travail typiques des utilisateurs finaux.
4. **Détection de Problèmes Globaux :** Ils visent à détecter des problèmes globaux qui peuvent survenir à l'échelle du système, tels que des goulots d'étranglement de performance, des problèmes de sécurité ou des conflits d'interopérabilité.

TESTS DE SYSTÈME

- **Types de Tests de Système :**

1. **Tests de Fonctionnalité :** Ces tests évaluent si les fonctionnalités du système fonctionnent comme prévu. Ils s'assurent que le logiciel effectue correctement ses tâches spécifiques.
2. **Tests de Performance :** Les tests de performance évaluent la vitesse, la réactivité et la capacité du système à gérer des charges de travail élevées. Ils identifient les problèmes de performance potentiels.
3. **Tests de Sécurité :** Les tests de sécurité évaluent la résistance du système aux vulnérabilités, aux attaques et aux menaces. Ils visent à garantir la sécurité des données et la protection contre les intrusions.
4. **Tests d'Interface Utilisateur :** Ils vérifient l'aspect et le comportement de l'interface utilisateur, en s'assurant qu'elle est conviviale et qu'elle respecte les normes de conception.
5. **Tests de Compatibilité :** Les tests de compatibilité vérifient que le logiciel fonctionne correctement sur différentes plates-formes, navigateurs, systèmes d'exploitation, etc.
6. **Tests de Conformité :** Ces tests s'assurent que le logiciel est conforme aux normes, réglementations et directives spécifiques de l'industrie ou de la région.

TESTS DE NON-REGRESSION

Les tests de système sont une composante cruciale du processus de test logiciel. Ils visent à évaluer le système logiciel dans son ensemble pour s'assurer qu'il fonctionne conformément aux exigences spécifiées.

- **Objectifs des Tests de Système :**

1. **Validation des Exigences :** Les tests de système vérifient que le logiciel respecte toutes les exigences fonctionnelles et non fonctionnelles définies dans les spécifications du système. Cela inclut les fonctionnalités attendues, la performance, la sécurité, la convivialité, etc.
2. **Évaluation de l'Intégration :** Ils s'assurent que toutes les composantes du système s'intègrent correctement et interagissent de manière cohérente. Cela inclut la vérification des interfaces entre les modules.
3. **Validation des Scénarios d'Utilisation :** Les tests de système vérifient que le logiciel fonctionne correctement dans des scénarios d'utilisation réalistes. Ils couvrent les flux de travail typiques des utilisateurs finaux.
4. **Détection de Problèmes Globaux :** Ils visent à détecter des problèmes globaux qui peuvent survenir à l'échelle du système, tels que des goulots d'étranglement de performance, des problèmes de sécurité ou des conflits d'interopérabilité.

TESTS DE NON-REGRESSION

Les tests de non-régression, également appelés tests de régression, sont un type de test logiciel conçu pour s'assurer que les modifications apportées à un logiciel n'ont pas introduit de nouveaux bugs ou affecté négativement les fonctionnalités existantes **Objectifs des Tests de Non-Régression :**

1. **Identifier les Régressions :** L'objectif principal des tests de non-régression est de détecter les régressions, c'est-à-dire les comportements indésirables introduits par des changements récents dans le code source, tels que des mises à jour, des correctifs ou de nouvelles fonctionnalités.
2. **Maintenir la Stabilité :** Ils visent à maintenir la stabilité globale du logiciel en garantissant que les fonctionnalités existantes continuent de fonctionner correctement après des modifications.
3. **Réduire les Risques :** En identifiant les problèmes rapidement, les tests de non-régression aident à réduire les risques liés à la livraison de nouvelles versions logicielles.

TESTS DE NON-REGRESSION

- **Principes de Base des Tests de Non-Régression :**

1. **Automatisation :** Les tests de non-régression sont souvent automatisés pour permettre des exécutions rapides et fréquentes. L'automatisation garantit également la reproductibilité des tests.
2. **Couverture Étendue :** Ils couvrent généralement un large éventail de fonctionnalités du logiciel, en mettant l'accent sur les fonctionnalités les plus critiques et les plus susceptibles d'être affectées par les modifications.
3. **Suivi des Versions :** Les tests de non-régression sont exécutés à chaque nouvelle version du logiciel. Cela peut être intégré dans un processus de livraison continue (CI/CD) pour garantir des tests réguliers.
4. **Gestion des Rapports :** Les résultats des tests de non-régression sont enregistrés dans des rapports détaillés, avec une distinction claire entre les tests réussis et les tests échoués.

TESTS DE NON-REGRESSION

- **Scénarios de Tests de Non-Régression :**

1. **Régression Fonctionnelle :** Ces tests vérifient que les fonctionnalités existantes ne sont pas affectées par les modifications du code. Par exemple, une fonction de recherche qui fonctionnait correctement avant une mise à jour doit continuer à fonctionner après.
2. **Régression de Performance :** Ils s'assurent que la performance du logiciel n'a pas été dégradée par les changements récents. Par exemple, un temps de réponse acceptable doit être maintenu.
3. **Régression d'Interface Utilisateur :** Ces tests vérifient que l'interface utilisateur n'a pas été altérée et que les éléments d'interface continuent de fonctionner correctement.
4. **Régression de Sécurité :** Ils visent à détecter les vulnérabilités de sécurité introduites par les modifications récentes. Cela garantit que les correctifs de sécurité ne compromettent pas d'autres parties du logiciel.

TESTS DE NON-REGRESSION

- **Flux de Travail des Tests de Non-Régression :**

1. **Identification des Zones à Risque :** Avant de commencer les tests de non-régression, il est essentiel d'identifier les parties du logiciel qui sont les plus susceptibles d'être affectées par les modifications récentes.
2. **Automatisation des Tests :** Les scénarios de test de non-régression sont automatisés à l'aide d'outils appropriés, tels que des frameworks de test ou des outils de test automatisé.
3. **Exécution Régulière :** Les tests de non-régression sont exécutés régulièrement, idéalement après chaque modification du code source. Les tests automatisés peuvent être intégrés dans un pipeline CI/CD.
4. **Gestion des Rapports :** Les résultats des tests sont enregistrés dans des rapports. En cas d'échec, des notifications sont envoyées à l'équipe de développement pour enquêter sur la régression.

Les tests de non-régression sont un élément essentiel de l'assurance qualité du logiciel, car ils garantissent que le logiciel continue de fonctionner correctement au fil du temps, malgré les évolutions constantes du code source.

TESTS D'ACCEPTATION

- Les tests d'acceptation sont une étape essentielle du processus de test logiciel. Ils visent à s'assurer que le logiciel répond aux besoins et aux attentes des utilisateurs et des parties prenantes, et qu'il est prêt à être mis en production.
- **Objectifs des Tests d'Acceptation :**

1. **Validation des Exigences :** Les tests d'acceptation sont conçus pour valider que le logiciel respecte toutes les exigences spécifiées, qu'elles soient fonctionnelles ou non fonctionnelles.
2. **Confirmation des Attentes :** Ils s'assurent que le logiciel répond aux attentes des utilisateurs finaux, des clients et des parties prenantes, en termes de fonctionnalités, de performance, de sécurité, etc.
3. **Préparation à la Mise en Production :** Les tests d'acceptation sont l'une des dernières étapes avant la mise en production. Ils doivent garantir que le logiciel est prêt à être déployé sans causer de problèmes majeurs.

TESTS D'ACCEPTATION

- **Types de Tests d'Acceptation :**

1. **Tests d'Acceptation Utilisateur (UAT) :** Ils sont effectués par les utilisateurs finaux du logiciel pour s'assurer qu'il répond à leurs besoins. Les UAT sont généralement les derniers tests avant la mise en production.
2. **Tests d'Acceptation Fonctionnelle :** Ils vérifient que toutes les fonctionnalités spécifiées dans les exigences fonctionnelles sont correctement mises en œuvre.
3. **Tests d'Acceptation Non Fonctionnelle :** Ils se concentrent sur les aspects non fonctionnels du logiciel, tels que la performance, la sécurité, la convivialité, la compatibilité, etc.
4. **Tests d'Acceptation de Compatibilité :** Ils vérifient que le logiciel fonctionne correctement sur différentes plates-formes, navigateurs, systèmes d'exploitation, etc.

TESTS D'ACCEPTATION

- **Tests d'Acceptation Automatisés :**
- Bien que les tests d'acceptation soient souvent effectués manuellement, il est possible de les automatiser dans certains cas, en particulier pour les aspects fonctionnels répétitifs. Cela peut accélérer le processus et améliorer la reproductibilité des tests.
- Les tests d'acceptation jouent un rôle crucial dans l'assurance de la qualité du logiciel, car ils permettent de garantir que le logiciel satisfait réellement les besoins et les attentes des utilisateurs. Ils sont une étape clé avant la mise en production et contribuent à minimiser les risques de problèmes majeurs une fois le logiciel en service.

```
Feature: Réservation de Vol

  Scenario: Réserver un Vol
    Given l'utilisateur ouvre l'application de réservation de vols
    When l'utilisateur recherche et sélectionne un vol pour une destination
    And l'utilisateur remplit les détails du voyage, y compris les noms des
    And l'utilisateur confirme la réservation
    Then l'utilisateur reçoit une confirmation de réservation avec un numéro
    And le paiement est traité correctement
    And l'utilisateur reçoit un e-mail de confirmation de réservation
```

TYPES DE TESTS RECAP

Type de Test	Portée	Objectif	Isolation des Dépendances	Exécution	Couverture	Rapidité
Test Unitaire	Unité de code individuelle	Vérifier le comportement d'une unité	Isolé	Rapide	Spécifique	Très rapide
Test d'Intégration	Plusieurs unités, modules	Vérifier l'intégration des composants	Isolé ou partiellement isolé	Moyenne	Partielle	Généralement rapide
Test de Système	Système logiciel en entier	Vérification du système dans son ensemble	Non isolé	Généralement plus lente	Large	Variable

TYPES DE TESTS RECAP

Type de Test	Portée	Objectif	Isolation des Dépendances	Exécution	Couverture	Rapidité
Test de Non-Régression	Partie du logiciel modifiée	Identifier les régressions	Variable selon la portée	Après chaque modification	Spécifique	Variable
Test d'Acceptation	Application complète, Scénarios	Validation des exigences	Variable selon le contexte	Avant la mise en production	Large	

CONCEPTION DE CAS DE TEST.

CONCEPTION CAS DE TEST

La conception de cas de test est une étape cruciale dans le processus de test logiciel. Elle consiste à créer des cas de test qui décrivent comment vérifier si le logiciel fonctionne correctement et répond aux exigences spécifiées. Voici une description plus détaillée de la conception de cas de test :

- **Étapes de Conception de Cas de Test :**

1. **Compréhension des Exigences :** La première étape consiste à comprendre les exigences du logiciel. Il s'agit de s'assurer que vous avez une connaissance claire et précise de ce que le logiciel est censé faire. Les exigences fonctionnelles, les exigences non fonctionnelles et les cas d'utilisation sont des sources clés d'informations.
2. **Identification des Scénarios de Test :** Une fois que vous avez compris les exigences, identifiez les scénarios de test. Les scénarios de test sont des situations ou des actions spécifiques que vous souhaitez tester. Par exemple, dans une application de réservation de vols, un scénario de test pourrait être "Réserver un vol".
3. **Définition des Entrées et des Données :** Pour chaque scénario de test, définissez les entrées nécessaires. Cela inclut les données d'entrée, les conditions initiales et les états du système. Par exemple, pour le scénario de réservation de vol, les entrées pourraient inclure la sélection d'un vol, les détails du voyage, etc.

CONCEPTION CAS DE TEST

4. **Définition des Étapes de Test :** Pour chaque scénario de test, écrivez les étapes spécifiques que l'utilisateur ou le testeur doit suivre pour exécuter le test. Les étapes doivent être claires et détaillées. Par exemple, les étapes pour le scénario de réservation de vol pourraient inclure l'ouverture de l'application, la sélection d'un vol, la saisie des détails du voyage, etc.

5. **Spécification des Critères de Validation :** Pour chaque scénario de test, spécifiez les critères qui indiquent si le test est réussi ou non. Les critères de validation sont basés sur les attentes définies dans les exigences. Par exemple, le critère de validation pour le scénario de réservation de vol pourrait être la réception d'une confirmation de réservation avec un numéro de réservation.

6. **Gestion des Cas de Test :** Les cas de test sont généralement gérés dans un système de gestion de tests ou une feuille de calcul. Chaque cas de test est associé à un identifiant unique, et des informations telles que la priorité, l'état et les résultats sont enregistrées.

CONCEPTION CAS DE TEST

- **Meilleures Pratiques pour la Conception de Cas de Test :**
- Assurez-vous que les cas de test sont clairs, complets et non ambigus.
- Évitez les cas de test redondants. Si un test couvre déjà un scénario similaire, il n'est pas nécessaire de le répéter.
- Tenez compte des cas limites, des valeurs atypiques et des conditions exceptionnelles.
- Utilisez des données de test réalistes qui reflètent les scénarios d'utilisation réels.
- Vérifiez que chaque cas de test est indépendant et ne dépend pas du résultat d'autres tests.
- Documentez les préconditions et les postconditions pour chaque cas de test.
- La conception de cas de test est une discipline essentielle pour garantir la qualité d'un logiciel. Des cas de test bien conçus contribuent à identifier les problèmes tôt dans le cycle de développement, ce qui réduit les coûts et les risques associés à la détection de bugs tardifs.

Cas pratiques sur 2 exemples

EXÉCUTION DES TESTS ET ANALYSE DES RÉSULTATS

L'exécution des tests et l'analyse des résultats sont des étapes essentielles du processus de test logiciel. Elles consistent à effectuer les tests planifiés, enregistrer les résultats et évaluer si le logiciel se comporte conformément aux attentes.

- **1. Exécution des Tests :**
- **Préparation de l'Environnement :** Avant d'exécuter les tests, assurez-vous que l'environnement de test est correctement configuré. Cela inclut l'installation du logiciel à tester, la configuration des données de test, la mise en place de l'environnement de test (matériel, réseau, etc.) et la préparation des outils de test.
- **Sélection des Cas de Test :** Les cas de test à exécuter sont sélectionnés en fonction des priorités, de la planification et de l'objectif du test. Les cas de test peuvent être exécutés manuellement par des testeurs ou automatisés à l'aide d'outils de test automatisés.
- **Exécution des Tests :** Les tests sont effectués conformément aux scénarios de test spécifiés. Chaque étape du cas de test est suivie, les entrées sont fournies, les actions sont exécutées et les résultats sont enregistrés. Tout comportement non conforme aux critères de validation est noté comme un problème ou un bogue.
- **Documentation des Résultats :** Les résultats de chaque cas de test sont documentés de manière détaillée. Cela inclut les données d'entrée, les étapes du test, les résultats observés, les problèmes identifiés, les captures d'écran (le cas échéant) et d'autres informations pertinentes.

EXÉCUTION DES TESTS ET ANALYSE DES RÉSULTATS

- **2. Analyse des Résultats :**
- **Évaluation des Résultats :** Une fois les tests exécutés, les résultats sont évalués pour déterminer si le logiciel se comporte conformément aux attentes. Chaque cas de test réussi est confirmé, tandis que les échecs sont examinés plus en détail.
- **Détection des Problèmes :** Les problèmes, les anomalies et les bogues identifiés sont documentés et signalés. Ils sont classés en fonction de leur gravité et de leur impact sur le logiciel.
- **Suivi des Progrès :** La progression globale du processus de test est suivie. Les cas de test réussis, échoués et non encore exécutés sont surveillés pour évaluer l'état d'avancement des tests.
- **Communication des Résultats :** Les résultats sont communiqués aux parties prenantes, y compris aux développeurs, aux responsables de projet et aux équipes de test. Des rapports de test sont générés pour résumer les résultats et les problèmes.
- **Réexécution des Tests :** Si des problèmes sont identifiés, ils sont résolus par les développeurs, puis les tests affectés sont réexécutés pour confirmer que les problèmes ont été résolus. Cette étape peut nécessiter plusieurs itérations jusqu'à ce que le logiciel soit conforme.
- **Validation des Critères d'Acceptation :** Les critères d'acceptation définis lors de la conception des cas de test sont validés. Si un cas de test ne remplit pas ces critères, il est marqué comme échoué.
- L'exécution des tests et l'analyse des résultats sont des activités itératives. Les tests sont exécutés, les résultats sont analysés, les problèmes sont signalés, résolus, puis les tests sont réexécutés jusqu'à ce que le logiciel réponde aux attentes et soit prêt à être mis en production. Ces étapes contribuent à garantir la qualité et la fiabilité du logiciel.

OUTILS DE GESTION DES TESTS.

Pour la gestion des tests dans le contexte Java, il existe plusieurs outils et frameworks spécifiques que vous pouvez utiliser pour planifier, exécuter et gérer vos tests. Voici quelques-uns des outils de gestion des tests couramment utilisés dans l'écosystème Java :

1. **JUnit** : JUnit est un framework de test unitaire pour Java. Il est largement utilisé pour écrire et exécuter des tests unitaires dans des projets Java. Bien qu'il se concentre principalement sur les tests unitaires, il peut être utilisé en conjonction avec d'autres outils pour la gestion globale des tests.

2. **TestNG** : TestNG est un framework de test en Java qui offre des fonctionnalités plus avancées que JUnit, notamment la prise en charge de tests parallèles, la gestion des suites de tests et la génération de rapports détaillés.

3. **Selenium** : Selenium est un outil d'automatisation de tests fonctionnels largement utilisé pour les tests d'applications Web. Il prend en charge plusieurs langages, dont Java, pour l'écriture de scripts de test automatisés.

4. **Cucumber** : Cucumber est un framework de tests automatisés basé sur le comportement (BDD - Behavior-Driven Development). Il permet de définir des scénarios de test en langage naturel (Gherkin) et de les automatiser en utilisant Java.

5. **TestLink** : TestLink est un outil de gestion de tests open source qui prend en charge l'intégration avec Java pour la gestion des cas de test, la planification des tests et la génération de rapports.

OUTILS DE GESTION DES TESTS.

6. **JIRA** : JIRA, bien que principalement un outil de gestion de projet, peut être utilisé pour gérer des tests en utilisant des plugins comme Zephyr for Jira, qui permet la création, l'exécution et la gestion de cas de test.

7. **Jenkins** : Jenkins est un outil d'intégration continue qui peut être utilisé pour automatiser l'exécution de tests Java. Vous pouvez configurer des projets Jenkins pour déclencher automatiquement des tests lors de chaque nouvelle version du code.

8. **Allure** : Allure est un outil de génération de rapports visuels pour les résultats des tests en Java. Il peut être utilisé avec JUnit, TestNG, Cucumber, etc., pour créer des rapports faciles à lire et à comprendre.

9. **Robot Framework** : Bien que principalement utilisé avec Python, Robot Framework offre également une bibliothèque pour Java. Il prend en charge l'automatisation des tests pour les applications Web, les API, et plus encore.

10. **Serenity BDD** : Serenity est un framework BDD pour Java qui intègre des rapports détaillés et visuels. Il peut être utilisé pour la gestion des tests automatisés et manuels.

Le choix de l'outil dépendra des besoins spécifiques de votre projet, de la manière dont vous concevez vos tests (unitaires, d'intégration, fonctionnels, etc.), de la facilité d'intégration avec d'autres outils, et de vos préférences en matière de développement et de gestion de tests. Chacun de ces outils a ses avantages et ses inconvénients, il est donc essentiel de les évaluer en fonction de vos besoins.

KEYCE Informatique & Intelligence Artificielle ISKIIA

AUTOMATISATION DES TESTS

CHAP III

INTRODUCTION À L'AUTOMATISATION DES TESTS

L'automatisation des tests est une pratique essentielle dans le domaine de l'assurance qualité logicielle. Elle consiste à utiliser des logiciels et des scripts pour exécuter automatiquement des tests sur un logiciel, au lieu de les effectuer manuellement. Cette approche présente de nombreux avantages, notamment l'efficacité, la reproductibilité, la couverture exhaustive des tests et la réduction des coûts. Voici un aperçu de l'automatisation des tests :

- **Pourquoi l'Automatisation des Tests est-elle Importante ?**

L'automatisation des tests offre plusieurs avantages majeurs :

- **Efficacité :** Les tests automatisés s'exécutent beaucoup plus rapidement que les tests manuels, ce qui permet d'obtenir des résultats plus rapidement.
- **Reproductibilité :** Les tests automatisés sont exécutés de manière cohérente, éliminant ainsi les erreurs humaines potentielles liées à la répétition des tests.
- **Couverture exhaustive :** Il est possible d'exécuter un grand nombre de cas de test en même temps, ce qui garantit une couverture exhaustive des tests.
- **Réduction des coûts :** Bien que la mise en place de l'automatisation des tests puisse nécessiter un investissement initial, elle permet d'économiser du temps et des ressources à long terme.

INTRODUCTION À L'AUTOMATISATION DES TESTS

- **Quels Types de Tests Peuvent être Automatisés ?**

Un large éventail de tests peut être automatisé, notamment :

- **Tests Unitaires :** Les tests de petites unités de code, tels que les fonctions ou les méthodes.
- **Tests d'Intégration :** Les tests visant à vérifier l'intégration entre les composants.
- **Tests Fonctionnels :** Les tests de fonctionnalités, y compris les scénarios d'utilisation.
- **Tests de Performance :** Les tests visant à évaluer les performances du logiciel.
- **Tests de Sécurité :** Les tests pour identifier les vulnérabilités de sécurité.
- **Tests de Régression :** Les tests pour s'assurer que les modifications ne cassent pas les fonctionnalités existantes.

- **3. Outils d'Automatisation des Tests :**
- Il existe une grande variété d'outils d'automatisation des tests, allant des frameworks open source comme Selenium et Appium aux solutions commerciales comme HP UFT (Unified Functional Testing) et TestComplete. Le choix de l'outil dépendra des besoins spécifiques du projet, du type d'application testée et du langage de programmation utilisé.

INTRODUCTION À L'AUTOMATISATION DES TESTS

- L'automatisation ne remplace pas complètement les tests manuels, mais les complète.
- Les tests automatisés nécessitent une maintenance continue.
- Les compétences en programmation sont souvent nécessaires pour automatiser efficacement les tests.
- La planification et la stratégie sont cruciales pour réussir l'automatisation des tests.
- L'automatisation des tests est devenue une pratique courante dans le développement logiciel moderne, offrant des avantages significatifs en termes de qualité, d'efficacité et de rapidité de livraison des logiciels. Elle permet aux équipes de test de se concentrer sur des tâches plus créatives et complexes tout en garantissant une couverture de test exhaustive.

SÉLECTION DES OUTILS D'AUTOMATISATION

SÉLECTION DES OUTILS D'AUTOMATISATION

La sélection des outils d'automatisation est une étape cruciale dans le processus d'automatisation des tests. Le choix de l'outil approprié dépend de divers facteurs, notamment les besoins du projet, le type d'application testée, les compétences de l'équipe et d'autres considérations. Voici un aperçu des étapes et des considérations pour sélectionner les outils d'automatisation des tests :

- **Évaluation des Besoins :**
- La première étape consiste à définir clairement les besoins de votre projet en matière d'automatisation des tests. Posez-vous les questions suivantes :
- Quels types de tests devez-vous automatiser (tests unitaires, tests d'intégration, tests fonctionnels, tests de performance, tests de sécurité, etc.) ?
- Quel est le langage de programmation principal de votre application ?
- Sur quelles plates-formes votre application doit-elle être testée (navigateurs, appareils mobiles, systèmes d'exploitation) ?
- Quel est le budget disponible pour l'achat d'outils ?
- Quels sont les délais pour la mise en place de l'automatisation des tests ?

SÉLECTION DES OUTILS D'AUTOMATISATION

Une fois que vous avez identifié vos besoins, recherchez les outils d'automatisation qui correspondent à ces besoins. Il existe de nombreuses options disponibles, à la fois open source et commerciales, pour une variété de types de tests. Voici quelques exemples d'outils populaires :

- **Selenium** : Pour l'automatisation des tests de sites Web.
- **Appium** : Pour l'automatisation des tests mobiles.
- **JUnit** et TestNG : Pour l'automatisation des tests unitaires et de tests fonctionnels en Java.
- JIRA et Zephyr : Pour la gestion et l'automatisation des tests dans un environnement agile.
- **LoadRunner** et **JMeter** : Pour l'automatisation des tests de performance.
- OWASP ZAP : Pour les tests de sécurité.

SÉLECTION DES OUTILS D'AUTOMATISATION

- **Évaluation et Comparaison :**

Une fois que vous avez identifié plusieurs options, évaluez-les et comparez-les en fonction de critères importants, tels que :

- Facilité d'utilisation : Est-ce que l'outil est convivial pour les membres de l'équipe ?
- Compatibilité : L'outil prend-il en charge les technologies et les plates-formes pertinentes pour votre projet ?
- Fonctionnalités : Dispose-t-il des fonctionnalités requises pour vos types de tests ?
- Coût : Le prix de l'outil est-il compatible avec votre budget ?
- Communauté et Support : L'outil dispose-t-il d'une communauté active et de support en cas de besoin ?

- **Essais et PoC (Proof of Concept) :**

Il est souvent recommandé de réaliser un essai ou un PoC avec l'outil sélectionné. Cela vous permettra de tester sa pertinence dans le contexte de votre projet avant de vous engager pleinement. Lors de l'essai, prenez en compte les aspects pratiques, tels que la création de scripts de test, l'exécution des tests, la génération de rapports, etc.

SÉLECTION DES OUTILS D'AUTOMATISATION

Langage	Framework Test Unitaire	Framework de Test End-to-End
JAVA	JUnit	Selenium
	TestNG	Appium
PYTHON	unittest (bibliothèque standard)	Selenium (avec Python)
	pytest	Pytest (pour les tests fonctionnels)
	nose	Robot Framework
JAVASCRIPT	Jasmine	Cypress
	Mocha	Puppeteer
	Jest	Protractor
C#	MSTest (Visual Studio)	Selenium (avec C#)
	NUnit	Appium (avec C#)
	xUnit	SpecFlow (pour le BDD)

SÉLECTION DES OUTILS D'AUTOMATISATION

PLATEFORME	Framework Test Unitaire	Framework de Test End-to-End
FLUTTER	flutter_test (bloc de test inclus dans Flutter)	Flutter Driver (pour les tests d'intégration)
	Mockito (pour les mocks)	
REACT NATIVE	Jest (intégré avec React Native)	Detox (pour les tests d'end-to-end)
	React Testing Library (pour les tests de composants)	
REACT	Jest (intégré avec React)	Cypress (pour les tests d'end-to-end)
	Enzyme (pour les tests de composants)	
LARAVEL	PHPUnit (intégré avec Laravel)	Laravel Dusk (pour les tests d'end-to-end)
		Behat (pour les tests BDD)

SÉLECTION DES OUTILS D'AUTOMATISATION

PLATEFORME	Framework Test Unitaire	Framework de Test End-to-End
SPRING	JUnit (intégré avec Spring)	Selenium (avec Spring)
	TestNG (si préféré)	Cucumber (pour les tests BDD)
ANDROID	JUnit (bibliothèque standard)	Espresso (pour les tests d'interface utilisateur)
	Robolectric (pour les tests unitaires)	UI Automator (pour les tests d'interface utilisateur)
	Mockito (pour les mocks)	Appium (pour les tests multi-plateformes)
IOS	XCTest (intégré avec Xcode)	XCUITest (pour les tests d'interface utilisateur)
	Quick (pour des tests plus expressifs)	Appium (pour les tests multi-plateformes)
	Nimble (pour les assertions)	Calabash (pour les tests d'interface multi-plateformes)

CRÉATION DE SCRIPTS DE TEST AUTOMATISÉS

CRÉATION DE SCRIPTS DE TEST AUTOMATISÉS

Voici les étapes de base pour créer des scripts de test automatisés en utilisant Java :

- **Configuration de l'Environnement de Développement :**

Avant de commencer à écrire des scripts de test, assurez-vous d'avoir une configuration de développement Java fonctionnelle. Vous aurez besoin de Java JDK (Java Development Kit) et d'un environnement de développement intégré (IDE) tel que Eclipse, IntelliJ IDEA ou NetBeans.

- **Choix d'un Framework de Test :**

Choisissez un framework de test adapté à vos besoins. En Java, **JUnit** et TestNG sont des frameworks de test unitaire populaires. Ces frameworks vous permettent de structurer vos tests et d'effectuer des assertions pour vérifier le comportement de votre application.

- **Création d'un Projet de Test :**

Créez un nouveau projet Java dans votre IDE et configurez-le pour utiliser le framework de test que vous avez choisi. Par exemple, si vous utilisez JUnit, vous devrez ajouter les dépendances JUnit à votre projet.

- **Écriture de Cas de Test**

CRÉATION DE SCRIPTS DE TEST AUTOMATISÉS

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class MonTest {

    @Test
    public void testExemple() {
        // Écrire votre test ici
        assertEquals(4, 2 + 2);
    }
}
```

EXÉCUTION ET RAPPORT DES TESTS AUTOMATISÉS.

RAPPORT DES TESTS AUTOMATISÉS.

L'exécution et la création de rapports pour les tests automatisés sont des étapes cruciales du processus de test automatisé. Elles impliquent l'exécution des scénarios de test automatisés que vous avez créés et la génération de rapports pour évaluer les résultats. Voici comment ces étapes se déroulent généralement :

- **1. Configuration de l'Environnement d'Exécution :** Avant d'exécuter des tests automatisés, assurez-vous que l'environnement d'exécution est correctement configuré. Cela inclut le bon environnement de développement, les dépendances, les configurations de serveur, etc. Assurez-vous également que l'application à tester est déployée et prête à être testée.

- **2. Exécution des Tests Automatisés :** Une fois l'environnement prêt, lancez l'exécution des tests automatisés. Selon le framework de test que vous utilisez, cela peut être effectué à partir de votre IDE, à l'aide de scripts de ligne de commande ou intégré dans un processus d'intégration continue (CI).

RAPPORT DES TESTS AUTOMATISÉS.

- **3. Surveillance de l'Exécution :** Pendant l'exécution des tests, surveillez les résultats. Cela inclut la vérification que les tests sont correctement exécutés, que les applications ne plantent pas, et que les assertions et vérifications fonctionnent conformément aux attentes.
- **4. Gestion des Problèmes :** Si des échecs de tests sont détectés pendant l'exécution, identifiez la source du problème. Les échecs peuvent être dus à des bogues dans l'application testée ou dans les scripts de test automatisés. Il est important de documenter et de signaler ces problèmes.
- **5. Génération de Rapports :** Une fois les tests terminés, générez des rapports. Ces rapports servent à documenter les résultats des tests, y compris les tests réussis, les tests échoués, les erreurs et les exceptions. Les rapports sont essentiels pour évaluer la qualité du logiciel testé.

RAPPORT DES TESTS AUTOMATISÉS.

- **6. Types de Rapports :** Il existe différents types de rapports que vous pouvez générer, notamment :
- **Rapports de résumé :** Ces rapports fournissent une vue d'ensemble des résultats des tests, montrant le nombre de tests réussis, échoués, et les taux de réussite.
- **Rapports détaillés :** Ces rapports incluent des informations spécifiques sur chaque test, y compris les données d'entrée, les résultats attendus, et les résultats réels.
- **Rapports de journalisation :** Les journaux détaillent l'exécution des tests et peuvent être utiles pour le débogage en cas d'échec.
- **7. Analyse des Rapports :** Examinez attentivement les rapports pour comprendre la qualité de l'application testée. Identifiez les tests qui ont échoué et les raisons de ces échecs. Cela peut vous aider à cibler les bogues et à améliorer le code de l'application.

RAPPORT DES TESTS AUTOMATISÉS.

- **8. Automatisation de la Génération de Rapports :** Pour faciliter le processus, l'automatisation de la génération de rapports est recommandée. Des outils comme Allure, TestNG, ou des plugins de CI (Jenkins, Travis CI) peuvent générer automatiquement des rapports à la fin de l'exécution des tests.
- **9. Intégration dans le Processus CI/CD :** Pour des processus d'intégration continue (CI) et de déploiement continu (CD), l'exécution des tests automatisés et la génération de rapports sont souvent intégrées à ces flux de travail pour automatiser les tests à chaque nouvelle version du logiciel.
- **10. Répétition :** Les tests automatisés sont généralement exécutés régulièrement pour s'assurer que le logiciel conserve sa qualité au fil du temps. Les rapports des tests précédents servent de référence pour comparer les nouvelles exécutions.

APPROCHE DE TEST.

PAIRWISE, UNIWISE TESTING

Le pairwise testing (aussi appelé combinaison deux à deux ou testing à deux facteurs) et le uniwise testing sont des techniques de test qui visent à réduire la complexité des scénarios de test en se concentrant sur les combinaisons les plus importantes de paramètres ou de variables. Ces approches sont souvent utilisées pour améliorer l'efficacité des tests en réduisant le nombre total de scénarios de test nécessaires tout en couvrant un large éventail de conditions.

Le choix entre le pairwise testing et le uniwise testing dépend des besoins spécifiques de votre projet, des ressources disponibles et de votre compréhension des interactions entre les paramètres. Si vous recherchez un équilibre entre la réduction du nombre de tests et la détection des bogues d'interaction, le pairwise testing est généralement un bon choix. Si vous devez minimiser le nombre de tests au maximum, le uniwise testing peut être approprié, mais il peut être moins robuste en termes de détection des bogues.

PAIRWISE, UNIWISE TESTING

- **Pairwise Testing :**

Le pairwise testing, ou testing à deux facteurs, repose sur le principe qu'un grand nombre de bogues ou d'erreurs de logiciel résultent d'interactions entre deux paramètres (facteurs) plutôt que de problèmes individuels. Au lieu de tester toutes les combinaisons possibles de paramètres, le pairwise testing consiste à tester uniquement toutes les combinaisons deux à deux de ces paramètres.

- Par exemple, si vous avez trois paramètres A, B et C, avec chacun deux valeurs possibles (par exemple, vrai ou faux), le testing complet testerait 2^3 = 8 combinaisons. En revanche, le testing deux à deux ne testerait que 2^2 = 4 combinaisons : AB, AC, BC, et ABC.
- Les avantages du pairwise testing sont les suivants :
- Réduction significative du nombre de tests nécessaires.
- Identification rapide des bogues résultant d'interactions entre deux paramètres.
- Amélioration de l'efficacité du processus de test.
- Les outils de génération de plans de test, tels que les tables de décision, sont souvent utilisés pour mettre en œuvre le pairwise testing.

PAIRWISE, UNIWISE TESTING

- **Uniwise Testing :**
- Le uniwise testing est une variante du pairwise testing qui se concentre sur la réduction du nombre de tests encore plus loin en testant uniquement une combinaison de paramètres à la fois. Cette approche est particulièrement utile lorsque les ressources de test sont limitées, et vous ne pouvez exécuter qu'un seul test à la fois.
- Contrairement au pairwise testing, le uniwise testing suppose que les interactions entre les paramètres ne sont pas courantes et que tester un seul paramètre à la fois est suffisant pour détecter la plupart des bogues.
- Le principal avantage du uniwise testing est la réduction drastique du nombre de tests nécessaires. Cependant, il présente également un risque plus élevé de manquer des interactions complexes entre les paramètres.

ELEMENTS DE TEST

- **Scénario de Cas (ou Cas de Test) (SC) :** Un scénario de cas est une description détaillée d'une situation ou d'une condition spécifique à tester dans une application ou un système. Chaque scénario de test est conçu pour tester un aspect particulier de la fonctionnalité ou du comportement de l'application. Il est composé de plusieurs éléments, dont les plus importants sont les suivants :
- **Fonction de Test (g):** La fonction de test décrit la fonctionnalité ou le comportement spécifique de l'application que vous souhaitez tester. Elle devrait expliquer en détail ce que l'application est censée faire dans le contexte du scénario de test. Par exemple, "Vérifier que l'utilisateur peut se connecter avec succès en utilisant un nom d'utilisateur et un mot de passe valides."
- **Données d'Entrée (DE) :** Les données d'entrée sont les informations, valeurs ou conditions que vous devez fournir à l'application pour exécuter le scénario de test. Elles sont utilisées pour simuler des conditions réelles. Par exemple, dans un scénario de connexion, les données d'entrée peuvent inclure un nom d'utilisateur valide et un mot de passe valide.

ELEMENTS DE TEST

- **Résultats Attendus (RA) :** Les résultats attendus décrivent le comportement attendu de l'application une fois que le scénario de test est exécuté avec les données d'entrée fournies. Ils spécifient ce que l'application devrait produire ou afficher en réponse aux données d'entrée. Par exemple, "L'application doit afficher un message de bienvenue et rediriger l'utilisateur vers son profil."
- **Tolérance (Tol) :** La tolérance (ou critère de succès) est une spécification des marges d'erreur ou des variations acceptables par rapport aux résultats attendus. Il indique dans quelle mesure l'application peut diverger des résultats attendus tout en étant considérée comme réussissant le test. Par exemple, la tolérance pourrait indiquer que la longueur d'une chaîne de caractères ne doit pas dépasser 10 caractères.
- **Mesures d'Erreur (N) :** Les mesures d'erreur sont des indicateurs ou des métriques qui permettent de quantifier les écarts entre les résultats obtenus lors de l'exécution du test et les résultats attendus. Elles servent à évaluer la performance de l'application par rapport aux critères de succès définis. Par exemple, dans un test de vitesse, la mesure d'erreur pourrait être le temps nécessaire pour effectuer une opération, qui ne doit pas dépasser un certain seuil.

ELEMENTS DE TEST

res=[]

For s in SC:

res[s]=N(g(DE),RA)<=Tol

le probleme de **l'oracle** determiner RA, g(X),N

LE "PROBLÈME DE L'ORACLE"

LE PROBLÈME DE L'ORACLE

Le problème de l'oracle est un concept important en matière de test logiciel et de vérification. Il se réfère au défi de déterminer si un logiciel fonctionne correctement en l'absence d'un "oracle", c'est-à-dire d'une source de vérité qui peut déclarer si les résultats produits par le logiciel sont corrects ou non.

- **L'Oracle :** Dans le contexte du test logiciel, l'oracle est une entité externe, une référence, ou une source de vérité qui sert de comparaison pour évaluer si le logiciel testé fonctionne correctement. Cet oracle peut prendre différentes formes :

- Une spécification ou une documentation détaillée qui décrit le comportement attendu du logiciel.
- Un système existant ou une version précédente du logiciel considéré comme correct.
- Une réponse ou un ensemble de réponses connues à partir desquelles vous pouvez vérifier les résultats du logiciel.
- Un expert humain qui évalue manuellement les résultats.

LE PROBLÈME DE L'ORACLE

- **Le Problème de l'Oracle :** Le problème de l'oracle survient lorsque vous devez tester un logiciel, mais que vous ne disposez pas d'un oracle fiable pour établir ce qui est "correct". Dans de nombreux cas, il peut être difficile ou coûteux de créer un oracle, surtout pour des logiciels complexes.
- Le problème de l'oracle se pose notamment dans les scénarios suivants :
- Lorsque le logiciel est en cours de développement et qu'il n'existe pas encore de spécifications détaillées ou de version de référence.
- Lorsque le logiciel effectue des calculs complexes pour lesquels il est difficile de déterminer la réponse correcte à l'avance.
- Lorsque le logiciel interagit avec des systèmes externes sur lesquels vous n'avez pas de contrôle, et dont le comportement peut varier.

LE PROBLÈME DE L'ORACLE

- **Solutions au Problème de l'Oracle :** Pour atténuer le problème de l'oracle, plusieurs approches sont possibles :
- Créer une spécification détaillée ou des cas de test qui définissent clairement le comportement attendu du logiciel.
- Utiliser des données de test "oracles" qui sont des données connues et valides pour lesquelles vous connaissez les réponses correctes.
- Faire appel à des experts humains pour évaluer manuellement les résultats.
- Utiliser des techniques de génération de test automatisées qui peuvent générer des cas de test en se basant sur le comportement observé du logiciel.
- Le problème de l'oracle souligne l'importance de la planification des tests et de la définition des critères de succès pour les tests. Il peut être particulièrement difficile à résoudre dans des cas où la définition de ce qui est "correct" est ambiguë ou complexe. Il est donc essentiel de développer des stratégies de test efficaces pour aborder ce problème.

KEYCE Informatique & Intelligence Artificielle ISKIIA

ANALYSE STATIQUE

INTRODUCTION.

- L'analyse statique est une méthode d'inspection du code source d'un programme sans l'exécuter. Elle est réalisée par des outils automatisés ou manuellement par des développeurs. L'objectif principal de cette analyse est d'identifier des problèmes potentiels dans le code qui pourraient entraîner des erreurs lors de l'exécution du programme.
- **Principaux Objectifs :**

1. **Détection d'Erreurs :** Identifier les erreurs de programmation, telles que les erreurs de syntaxe, les accès à des variables non initialisées, les dépassements de tableaux, etc.
2. **Conformité aux Normes :** Vérifier si le code source respecte les normes de codage spécifiées, telles que les conventions de nommage, les règles de qualité du code, etc.
3. **Analyse de la Sécurité :** Identifier les vulnérabilités de sécurité potentielles, comme les failles de sécurité liées à la manipulation incorrecte des données.
4. **Performance :** Identifier des pratiques de codage inefficaces qui pourraient affecter les performances du programme.

ANALYSE STATIQUE DU CODE SOURCE

- **Étapes de l'Analyse Statique :**

1. **Lexical Analysis :** C'est la première étape où le code source est analysé au niveau lexical pour identifier les éléments tels que les mots-clés, les identificateurs, les opérateurs, etc. Cela peut détecter des erreurs de syntaxe simples.
2. **Syntax Analysis :** Cette étape implique l'analyse syntaxique pour vérifier si le code suit la grammaire du langage de programmation. Les erreurs de syntaxe, telles que les parenthèses non fermées, sont détectées à cette étape.
3. **Semantic Analysis :** Il s'agit d'une analyse plus approfondie qui vise à comprendre la signification du code au-delà de sa syntaxe. Des erreurs sémantiques, telles que l'utilisation incorrecte des variables, peuvent être identifiées ici.
4. **Style Checking :** Vérification du respect des normes de codage et des conventions de style. Cela inclut des éléments tels que l'indentation, les commentaires, les noms de variables significatifs, etc.
5. **Data Flow Analysis :** Cette analyse suit le flux de données à travers le programme pour détecter des problèmes tels que les variables non initialisées ou les fuites de mémoire.
6. **Control Flow Analysis :** Étude de la séquence d'exécution des instructions pour identifier des problèmes potentiels tels que des boucles infinies ou des points de sortie manquants.
7. **Security Analysis :** Identification de vulnérabilités de sécurité potentielles, telles que les injections SQL, les débordements de tampon,

DÉTECTION DES ERREURS DE CODE : ANALYSE DES RÈGLES DE CODAGE

- **Exemples de Règles de Codage :**

1. **Nom des Variables :** Les variables doivent avoir des noms significatifs et suivre une convention de nommage (par exemple, camelCase, snake_case).
2. **Indentation :** Assurer une indentation cohérente pour améliorer la lisibilité du code.
3. **Longueur des Fonctions :** Limiter la longueur des fonctions pour maintenir la clarté et la facilité de maintenance.
4. **Utilisation des Commentaires :** Encourager l'utilisation de commentaires significatifs pour expliquer le code complexe ou les décisions de conception.
5. **Gestion des Erreurs :** S'assurer que les erreurs sont correctement gérées avec des mécanismes appropriés, tels que les blocs try-catch.
6. **Sécurité :** Appliquer des règles pour prévenir les vulnérabilités de sécurité, comme l'injection SQL ou les failles XSS.

Détection des erreurs de code : analyse des règles de codage

- **Avantages :**

1. **Consistance du Code :** Assure la cohérence dans la manière dont le code est écrit, facilitant la collaboration au sein de l'équipe.
2. **Prévention des Erreurs Courantes :** Aide à prévenir des erreurs potentielles dès les premières étapes du développement.
3. **Maintenabilité :** Facilite la maintenance en rendant le code plus lisible et compréhensible.

- **Limitations :**

1. **Subjectivité :** Certaines règles peuvent être subjectives, nécessitant parfois des ajustements pour s'adapter aux besoins spécifiques du projet.
2. **Faux Positifs/Négatifs :** Les outils peuvent parfois signaler des problèmes inexistants (faux positifs) ou manquer des problèmes réels (faux négatifs).

OUTILS D'ANALYSE STATIQUE

Aspect de l'Analyse Statique	Java	Python	JavaScript	PHP	C++
Linters Généraux	Checkstyle, PMD, FindBugs	Flake8, Pylint, Pyflakes	ESLint, JSHint, Standard	PHP_Code Sniffer, PHPLint, PHPStan	Clang Static Analyzer, Cppcheck, PVS-Studio
Analyse de la Sécurité	SonarQube, OWASP Dependenc y-Check	Bandit, Safety	ESLint (security plugins), Snyk	RIPS, PHPStan	PVS-Studio, Clang Static Analyzer
Analyse de la Qualité du Code	SonarQube, PMD, FindBugs, Checkstyle	Pylint, Flake8, Bandit	ESLint, JSHint, Prettier, Standard	PHP_Code Sniffer, PHPStan, Psalm	Clang Static Analyzer, Cppcheck, PVS-Studio
Analyse de la Cohérence du Code	Checkstyle, SonarQube	Pylint	ESLint, JSHint, Prettier, Standard	PHP_Code Sniffer, PHPStan	Clang Static Analyzer, Cppcheck, PVS-Studio

AVANTAGES/ LIMITATIONS.

- **Avantages :**

1. **Détection Précoce :** Les erreurs sont identifiées avant l'exécution du programme, ce qui permet une correction précoce.
2. **Amélioration de la Qualité du Code :** Contribue à améliorer la lisibilité, la maintenabilité et la qualité globale du code source.
3. **Économie de Coûts :** La correction d'erreurs tôt dans le cycle de développement est généralement moins coûteuse que la correction d'erreurs découvertes lors des phases ultérieures.

- **Limitations :**

4. **Limitations d'Inférence :** Certains problèmes, tels que les erreurs de logique, peuvent ne pas être détectés par l'analyse statique.
5. **Faux Positifs/Négatifs :** Les outils peuvent parfois signaler des problèmes inexistants (faux positifs) ou manquer des problèmes réels (faux négatifs).

- L'analyse statique est une composante essentielle des pratiques de développement logiciel robustes, contribuant à la création de logiciels fiables, sécurisés et de haute qualité.

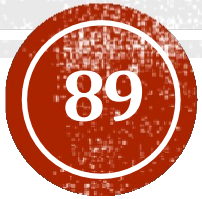
EXERCICE PRATIQUE

Intégration avec un Système de Build (Maven ou Gradle) sur le projet EQUATION SECOND DEGREE

1. Intégrez **Checkstyle, PMD et FindBugs** dans un projet Java utilisant **Maven** ou **Gradle**.
2. Configurez ces outils en tant que parties du processus de build.
3. Exécutez le processus de build et examinez les rapports générés.
4. Identifiez quelques-unes des erreurs ou des avertissements rapportés.
5. Corrigez ces problèmes dans le code source.

KEYCE Informatique & Intelligence Artificielle ISKIIA

ANALYSE DYNAMIQUE



INTRODUCTION.

- L'analyse dynamique est une méthode d'évaluation des logiciels qui se concentre sur l'exécution effective du programme. Contrairement à l'analyse statique qui examine le code source sans l'exécuter, l'analyse dynamique nécessite l'exécution du programme pour observer son comportement réel.
- **Objectifs de l'Analyse Dynamique :**

1. **Détection d'Erreurs à l'Exécution :** Identifier les erreurs, les exceptions et les comportements inattendus pendant l'exécution du programme.
2. **Évaluation de la Performance :** Mesurer la vitesse d'exécution, l'utilisation des ressources et d'autres métriques liées à la performance.
3. **Test Fonctionnel :** Vérifier si le logiciel fonctionne conformément aux spécifications et aux attentes des utilisateurs.

INTRODUCTION.

- **Techniques Courantes d'Analyse Dynamique :**

1. **Tests Unitaires :** Évaluent le comportement d'unités individuelles de code, généralement des fonctions ou des méthodes.
2. **Tests d'Intégration :** Vérifient la manière dont les différentes parties du système interagissent les unes avec les autres.
3. **Tests de Système :** Évaluent le système dans son ensemble pour s'assurer qu'il répond aux exigences globales.
4. **Tests de Performance :** Mesurent la réactivité, la stabilité et la capacité du logiciel à traiter un certain volume de données ou d'utilisateurs.

PROFILAGE DE CODE.

- Le profilage de code est une technique d'analyse dynamique qui vise à mesurer et évaluer les performances d'un programme en identifiant les parties du code qui consomment le plus de ressources, telles que le temps d'exécution, la mémoire, ou l'utilisation du processeur. Cette approche permet d'optimiser le code pour améliorer ses performances. Voici une présentation simple et concise du profilage de code :

- **Objectifs du Profilage de Code :**

1. **Identification des Goulots d'Étranglement :** Trouver les parties du code qui ralentissent l'exécution globale du programme.
2. **Optimisation des Performances :** Identifier les opportunités d'optimisation pour réduire le temps d'exécution ou l'utilisation de la mémoire.
3. **Analyse de l'Utilisation du Processeur :** Mesurer la proportion de temps CPU consacrée à chaque fonction du programme.

PROFILAGE DE CODE.

- **Méthodes Courantes de Profilage de Code :**

1. **Profilage du Temps d'Exécution :** Mesure le temps passé dans chaque fonction du programme pendant son exécution.
2. **Profilage de la Mémoire :** Identifie la consommation de mémoire par chaque fonction ou section du code.
3. **Profilage de l'Utilisation du Processeur :** Analyse la charge du processeur attribuée à chaque fonction.

- **Étapes Typiques du Profilage de Code :**

4. **Identification des Zones Critiques :** Déterminer les parties du code qui nécessitent une attention particulière en termes de performance.
5. **Collecte des Données de Profilage :** Exécution du programme sous l'outil de profilage pour recueillir des données sur le temps d'exécution, la mémoire utilisée, etc.
6. **Analyse des Résultats :** Examiner les résultats du profilage pour identifier les goulots d'étranglement et les zones d'optimisation potentielle.
7. **Optimisation du Code :** Apporter des modifications au code pour améliorer les performances, basées sur les résultats du profilage.

PROFILAGE DE CODE.

- **Avantages du Profilage de Code :**

1. **Optimisation Ciblée :** Permet une optimisation précise des parties spécifiques du code.
2. **Amélioration des Performances :** Identifie les opportunités d'amélioration qui peuvent avoir un impact significatif sur les performances globales.
3. **Détection des Problèmes de Mémoire :** Aide à repérer les fuites de mémoire et les problèmes d'allocation.

- **Limitations du Profilage de Code :**

1. **Surcoût de l'Instrumentation :** L'acte de collecter des données de profilage peut influencer légèrement les performances réelles.
2. **Interprétation Correcte des Résultats :** Les résultats nécessitent une interprétation correcte pour effectuer des optimisations judicieuses.

- En résumé, le profilage de code est une étape cruciale dans l'optimisation des performances d'un logiciel, permettant d'identifier et de résoudre les problèmes qui affectent son exécution en temps réel.

OUTILS DE PROFILAGE DE CODE.

Méthode de Programmation	Java	Python	JavaScript	PHP	C++
Profilage du Temps d'Exécution	VisualVM, YourKit, JVisualVM, Java Mission Control	cProfile, Py-Spy, Pyflame, line_profiler	Chrome DevTools, console.time(), profiling libraries	Xdebug, Blackfire, Xhprof	gperftools (gprof), Valgrind, KCachegrind
Profilage de la Mémoire	VisualVM, YourKit, Eclipse Memory Analyzer	Py-Spy, memory_profiler, objgraph	Chrome DevTools, Memory API	Xdebug, Blackfire, Xhprof, Tideways, New Relic	Valgrind, Massif, Heaptrack, gperftools (gprof)
Profilage de l'Utilisation du Processeur	VisualVM, YourKit	Py-Spy, psutil	Chrome DevTools, console.profile()	Xdebug, Blackfire, Xhprof	gperftools (gprof), Valgrind, perf

Identification des goulots d'étranglement de performance.

1. **Collecte des Données :** Exécutez l'application sous un outil de profilage pour collecter des données sur le temps d'exécution, la mémoire utilisée et l'utilisation du processeur.
2. **Analyse des Résultats :** Examinez les résultats du profilage pour identifier les parties du code qui présentent des performances inférieures à la moyenne.
3. **Identification des Goulots d'Étranglement :** Repérez les fonctions ou sections du code qui consomment beaucoup de ressources ou qui ralentissent l'exécution globale.
4. **Optimisation Ciblée :** Appliquez des optimisations spécifiques aux parties identifiées comme des goulots d'étranglement.

OPTIMISATION DU CODE

1. **Analyse des Résultats de l'Analyse Dynamique :**
    1. **Récapitulation :** Examiner les rapports et les données collectées lors de l'analyse dynamique, y compris le profilage du temps d'exécution, de la mémoire, et de l'utilisation du processeur.
    2. **Identification des Problèmes :** Repérer les sections du code qui consomment beaucoup de temps, d'espace mémoire, ou qui présentent d'autres problèmes de performances.

2. **Priorisation des Problèmes :**
    1. **Hiérarchisation :** Classer les problèmes en fonction de leur impact sur les performances et de leur fréquence d'occurrence.
    2. **Détermination des Priorités :** Identifier les problèmes les plus critiques ou fréquemment rencontrés qui doivent être résolus en premier.

3. **Optimisation Ciblée :**
    1. **Restructuration du Code :** Modifier le code pour éliminer les inefficacités et améliorer la logique de traitement.
    2. **Algorithmes Plus Efficaces :** Remplacer des algorithmes lents par des alternatives plus rapides.
    3. **Optimisation de la Mémoire :** Réduire l'utilisation de la mémoire en évitant les fuites et en optimisant les structures de données.

4. **Tests de Rétrocompatibilité :**
    1. **Régression :** S'assurer que les modifications n'introduisent pas de nouvelles erreurs ou n'altèrent pas le comportement attendu de l'application.
    2. **Tests Automatisés :** Utiliser des tests automatisés pour garantir que les modifications n'affectent pas négativement d'autres parties du code.

5. **Réexécution de l'Analyse Dynamique :**
    1. **Validation :** Vérifier à nouveau les performances de l'application après les optimisations.
    2. **Comparaison :** Comparer les résultats avant et après l'optimisation pour évaluer l'impact des changements.

AVANTAGES/ LIMITATIONS.

- **Avantages de l'Analyse Dynamique :**

1. **Détection d'Erreurs Réelles :** Identifie les erreurs qui se produisent réellement pendant l'exécution.
2. **Évaluation Réaliste de la Performance :** Fournit des données sur la performance dans un environnement d'exécution réelle.
3. **Validation Fonctionnelle :** Vérifie si le logiciel répond aux attentes fonctionnelles.

- **Limitations de l'Analyse Dynamique :**

4. **Couverture Limitée :** Certains scénarios peuvent ne pas être couverts par les tests, laissant des erreurs potentielles non détectées.
5. **Complexité des Tests :** La mise en place de tests dynamiques peut être plus complexe que les analyses statiques.

- En résumé, l'analyse dynamique complète l'analyse statique en fournissant une évaluation réelle du logiciel en cours d'exécution. Cela permet de détecter des erreurs qui ne peuvent être identifiées que dans un environnement d'exécution réelle et de garantir que le logiciel répond efficacement aux exigences.

VOCABULAIRE

La vérification et la validation des logiciels sont des processus essentiels dans le domaine du développement logiciel et de l'assurance qualité. Il existe un certain vocabulaire spécifique à ces processus.Voici quelques termes importants à connaître :

- **Vérification (Verification)** : La vérification est le processus de détermination de la conformité d'un produit logiciel à des spécifications ou à des normes prédéfinies. Elle consiste à s'assurer que le logiciel a été construit correctement.
- **Validation** : La validation est le processus de détermination si le logiciel répond aux besoins et aux attentes de l'utilisateur. Elle vise à s'assurer que le bon produit a été construit.
- **Test de Vérification** : Les tests de vérification sont des tests effectués pour s'assurer que le logiciel respecte les spécifications techniques. Ils incluent généralement des tests unitaires, d'intégration et de système.
- **Test de Validation** : Les tests de validation sont des tests effectués pour s'assurer que le logiciel répond aux besoins de l'utilisateur et aux exigences fonctionnelles. Ils comprennent des tests fonctionnels, de convivialité, de performance, etc.
- **Plan de Test (Test Plan)** : Un plan de test est un document décrivant les objectifs, les ressources, la portée et le calendrier des activités de test. Il spécifie également les critères d'acceptation.
- **Cas de Test (Test Case)** : Un cas de test est une spécification détaillée d'un scénario de test, y compris les données d'entrée, les étapes à suivre et les résultats attendus.
- **Exécution de Test (Test Execution)** : L'exécution de test est le processus de mise en œuvre des cas de test. Les tests sont effectués en suivant les scénarios définis dans les cas de test.

VOCABULAIRE

- **Défaut (Defect ou Bug)** : Un défaut est une non-conformité ou un problème dans le logiciel qui ne répond pas aux spécifications ou aux attentes.
- **Cycle de Test (Test Cycle)** : Un cycle de test est une série d'activités de test effectuées dans une période donnée, généralement liée à une version du logiciel.
- **Bogue Suivi (Defect Tracking)** : Le suivi des bogues est le processus de suivi et de gestion des défauts ou des problèmes identifiés lors des tests.
- **Automatisation des Tests (Test Automation)** : L'automatisation des tests consiste à utiliser des outils et des scripts pour exécuter automatiquement des tests, ce qui accélère le processus de test.
- **Rapport de Test (Test Report)** : Un rapport de test est un document qui résume les résultats des activités de test, y compris les défauts identifiés et les performances du logiciel.
- **Critères d'Acceptation (Acceptance Criteria)** : Les critères d'acceptation sont des normes ou des exigences spécifiques que le logiciel doit satisfaire pour être accepté par le client ou l'utilisateur.
- **Gestion de la Configuration (Configuration Management)** : La gestion de la configuration implique la gestion des versions du logiciel, la gestion des modifications et la traçabilité des artefacts logiciels.
- **Exigences (Requirements)** : Les exigences sont les spécifications décrivant les fonctions, les performances et les caractéristiques attendues du logiciel.
- **Maturité des Processus (Process Maturity)** : La maturité des processus est le niveau de sophistication et d'efficacité des processus de développement et de test dans une organisation, généralement évalué à l'aide de modèles comme CMMI.

KEYCE Informatique & Intelligence Artificielle ISKIIA

MERCI

EVARIS FOMEKONG