République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohamed Khider Biskra

Faculté des sciences exactes et des sciences de la nature et de la vie

Département d'Informatique

Niveau : 1ère année Master

Option : Professionnel

Cours :

Système d'Information Agile 1 (SIA-1)

Réalisé par :
Dr. Ouaar Hanane

Année universitaire 2024/2025

Chapitre 1 :
Les méthodes agiles

Plan de chapitre

1. Introduction

Les méthodes agiles sont un ensemble d'approches de gestion de projet qui privilégient la flexibilité, l'adaptation au changement et la collaboration constante avec les parties prenantes. Nées dans le secteur du développement logiciel au début des années 2000, elles ont depuis été adoptées dans de nombreux autres domaines, comme la gestion de produits, le marketing et même l'éducation. L'idée centrale des méthodes agiles est de travailler par petites itérations, ou cycles, permettant de livrer des résultats fonctionnels de manière régulière tout en s'adaptant en temps réel aux besoins et retours des clients.

Au cœur des méthodes agiles se trouve le **Manifeste Agile**, un document rédigé en 2001 qui définit quatre valeurs essentielles et 12 principes.

Parmi les méthodologies agiles les plus connues, on retrouve **Scrum**, **Kanban**, **Extreme Programming (XP)** et **Lean**, chacune ayant ses propres spécificités mais partageant les mêmes principes agiles de base. L'adoption de ces méthodes permet d'améliorer la réactivité d'une équipe, la qualité des produits et la satisfaction des clients en impliquant régulièrement les parties prenantes tout au long du projet.

Les méthodes agiles sont donc particulièrement adaptées aux projets complexes et évolutifs, où les besoins peuvent changer en cours de route et où une approche flexible est essentielle.

2. Définitions de l'agilité

L'agilité, dans le contexte de la gestion de projet, fait référence à un ensemble de principes et de pratiques qui privilégient la flexibilité, l'adaptation continue et la collaboration. L'agilité vise à répondre efficacement aux changements, à améliorer la réactivité et à livrer des produits ou services de manière incrémentale, tout en tenant compte des besoins évolutifs des parties prenantes. Voici quelques définitions clés :

2.1 Agilité (en gestion de projet) :

C'est une approche qui favorise une planification flexible, l'adaptation rapide aux changements et une collaboration étroite avec les clients et les parties prenantes. L'objectif est de produire des résultats de manière itérative et incrémentale, en s'ajustant au fur et à mesure du projet pour mieux répondre aux besoins des utilisateurs. L'agilité, selon le Manifeste Agile de 2001, repose sur quatre valeurs fondamentales (voir prochaine section).

2.3 Agilité (en développement logiciel) :

Dans le domaine du développement logiciel, l'agilité désigne une approche qui se concentre sur la livraison rapide de petites portions de logiciel, tout en permettant des ajustements fréquents et une intégration continue. Les équipes agiles travaillent en cycles courts (sprints) pour livrer des versions fonctionnelles du produit à chaque itération.

2.4 Agilité (dans l'entreprise) :

De manière plus générale, l'agilité au sein d'une organisation fait référence à la capacité de l'entreprise à s'adapter rapidement aux changements internes et externes, à exploiter les nouvelles opportunités et à ajuster ses processus en fonction des retours du marché ou des clients.

Dans l'ensemble, l'agilité se caractérise par sa capacité à encourager l'adaptation continue, la réactivité face aux besoins changeants et la recherche de l'amélioration constante des processus et des produits. C'est une philosophie qui s'applique au-delà du simple cadre de gestion de projet et peut être utilisée pour transformer l'ensemble d'une organisation en la rendant plus flexible et adaptable.

3. Manifeste agile

Le manifeste agile est une déclaration rédigée par des développeurs en 2001 qui avaient pour objectif de révolutionner les processus de développement de logiciels. De par leur expérience, ils ont convenu de 4 valeurs et 12 principes pour le développement en mode agile.

3.1 Les 4 valeurs agiles

Les 4 valeurs du manifeste Agile, constituent les principes fondamentaux des approches Agile. Elles sont formulées ainsi :

3.1.1 Les individus et leurs interactions plutôt que les processus et les outils

Mettre l'accent sur les personnes et la communication, car ce sont elles qui créent la valeur, pas les outils ni les processus.

Exemple :

Une équipe de développement doit résoudre un bug critique. Au lieu de se perdre dans les processus (comme attendre une autorisation ou remplir des formulaires), les membres de l'équipe se réunissent rapidement pour discuter du problème et collaborer directement pour trouver une solution.

- L'accent est mis sur la communication directe et rapide entre les membres.

Dans la pratique :

- Organiser des **stand-up meetings quotidiens** pour échanger sur les progrès, obstacles et priorités.
- Favoriser la collaboration directe entre les développeurs, testeurs et product owners.

3.1.2 U**n logiciel opérationnel plutôt qu'une documentation exhaustive**

Livrer un produit fonctionnel est plus important que de rédiger une documentation détaillée. La documentation reste utile, mais elle ne doit pas ralentir l'avancement du projet.

Exemple :

Dans un projet, l'équipe livre une première version fonctionnelle d'un produit en 2 semaines (**MVP** : **Minimum Viable Product)**, même si certaines fonctionnalités ne sont pas encore parfaites. Le client peut tester et donner son avis immédiatement, plutôt que d'attendre plusieurs mois pour un produit complet accompagné d'une documentation technique détaillée.

- Le produit fonctionnel est priorisé pour apporter de la valeur rapidement.

Dans la pratique :

- Mettre en place des **sprints courts** pour livrer des versions incrémentales d'un produit.
- Réduire la documentation aux informations essentielles (par exemple, des guides utilisateur simples).

3.1.3 La collaboration avec le client plutôt que la négociation contractuelle

Travailler en étroite collaboration avec le client permet d'ajuster le produit aux besoins réels, plutôt que de s'enfermer dans un contrat rigide.

Exemple :

Un client change d'avis sur une fonctionnalité après avoir vu une démo. Plutôt que de refuser sous prétexte que cela n'était pas dans le cahier des charges initial, l'équipe revoit ses priorités et adapte son travail pour répondre à ce nouveau besoin.

- La priorité est donnée à la satisfaction du client grâce à une collaboration continue.

Dans la pratique :

- Organiser des **réunions régulières avec le client** pour montrer les progrès et ajuster les attentes.
- Mettre en place des outils de collaboration (comme clickup, Jira, Miro ou Slack) pour faciliter les échanges.

3.1.4 L'adaptation au changement plutôt que le suivi d'un plan

Être flexible face aux changements est primordial pour répondre aux besoins évolutifs du client, même si cela implique de revoir les plans initiaux.

Exemple :

Lorsqu'une nouvelle technologie apparaît en cours de projet, l'équipe décide de l'intégrer car elle permet d'améliorer la qualité du produit, même si cela nécessite de modifier les plans initiaux.

- L'équipe s'adapte pour maximiser la valeur livrée, au lieu de rester rigide face aux contraintes initiales.

Dans la pratique :

- Revoir régulièrement les priorités dans un **backlog produit** avec le product owner.
- Mettre en place des rétrospectives après chaque sprint pour identifier ce qui doit être ajusté.

3.2 Application : Les 4 valeurs agiles

3.2.1 Contexte du projet

Une entreprise lance une application mobile pour la réservation de salles de réunion dans les espaces de coworking. L'équipe est composée de :

- **3 développeurs**
- **1 UX/UI designer**
- **1 Product Owner (PO)**
- **1 Scrum Master**
- Le client est une entreprise spécialisée dans les espaces de coworking, et l'objectif est de développer une application intuitive permettant aux utilisateurs de réserver des salles en temps réel.

UI (Interface Utilisateur) : La partie "visible" d'un produit, ce que l'on voit, entend et touche lors de l'utilisation (ex. design graphique).

UX (Expérience Utilisateur) : Le mix de parties visibles et invisibles qui contribuent à l'expérience globale de l'utilisateur (ex. ergonomie, contenu des pages)

UX/UI Designers jouent un rôle essentiel dans les projets Agile pour concevoir des produits qui sont à la fois fonctionnels, esthétiques et centrés sur les besoins des utilisateurs. Leur mission

consiste à allier ergonomie, design visuel et satisfaction utilisateur tout en collaborant étroitement avec l'équipe pour garantir une intégration fluide des interfaces. Aussi :

- Créent des interfaces utilisateur intuitives.
- Collaborent avec le PO pour garantir que l'expérience utilisateur répond aux attentes.

3.2.2 Déroulement du projet et application des 4 valeurs Agile

1. Les individus et leurs interactions plutôt que les processus et les outils

Situation :

L'équipe identifie un problème lors du premier sprint : les développeurs rencontrent des difficultés à comprendre certaines maquettes UX conçues par le designer.

Approche Agile :

Au lieu d'attendre une réunion officielle ou de s'échanger des emails, le Scrum Master organise une discussion rapide entre les développeurs et le designer. Ensemble, ils clarifient les zones floues et ajustent les maquettes en temps réel.

Résultat :

Les ajustements sont apportés rapidement, et l'équipe gagne du temps, car la communication directe a permis d'éviter des malentendus prolongés.

2. Un logiciel opérationnel plutôt qu'une documentation exhaustive

Situation :

Le client souhaite une solution prête à être testée en un mois. Cependant, les spécifications techniques initiales incluent des dizaines de fonctionnalités, et la documentation prendrait du temps à rédiger.

Approche Agile :

L'équipe se concentre sur le développement d'un **MVP (Minimum Viable Product)** qui inclut:

- La réservation de salles en temps réel.
- Un système de notification simple.

Plutôt que de rédiger un document détaillé sur chaque fonctionnalité, l'équipe se limite à des tickets clairs dans un outil comme Jira et des notes succinctes sur les décisions clés.

Résultat :

Un produit fonctionnel est livré en un mois, permettant au client de le tester avec des utilisateurs. La documentation est allégée et évolutive, se concentrant sur les éléments essentiels.

3. La collaboration avec le client plutôt que la négociation contractuelle

Situation :

Après avoir vu la première version de l'application (MVP), le client demande d'ajouter une fonctionnalité pour afficher la disponibilité des salles via un code couleur (vert : disponible, rouge: occupé). Cette demande n'était pas prévue dans le contrat initial.

Approche Agile :

Plutôt que de refuser ou de demander une renégociation lourde du contrat, le Product Owner discute avec le client pour comprendre l'importance de cette fonctionnalité. Après une priorisation dans le backlog, l'équipe l'intègre au sprint suivant.

Résultat :

La nouvelle fonctionnalité est livrée rapidement, augmentant la satisfaction du client et rendant l'application plus intuitive pour les utilisateurs finaux.

4. L'adaptation au changement plutôt que le suivi d'un plan

Situation :

En cours de projet, un concurrent sort une application similaire avec une fonctionnalité innovante : la possibilité de filtrer les salles par équipements (écran, vidéoprojecteur, etc.).

Approche Agile :

L'équipe se réunit lors d'une rétrospective pour analyser l'impact de cette nouvelle concurrence. Le Product Owner décide de modifier les priorités du backlog pour intégrer un système de filtres dans les deux prochains sprints.

Résultat :

L'équipe intègre cette nouvelle fonctionnalité dans les délais, rendant leur produit compétitif sans suivre strictement le plan initial.

3.2.3 Synthèse des résultats

- Grâce à une **communication directe et fluide**, les problèmes sont résolus rapidement.
- Le client reçoit un **MVP fonctionnel** en un mois et participe activement à l'amélioration du produit.

- Les fonctionnalités sont **adaptées aux besoins réels** du client et du marché, renforçant la valeur de l'application.
- L'équipe reste **flexible face aux imprévus** et garde une longueur d'avance sur la concurrence.

3.3 Les 12 principes agiles

En plus des 4 valeurs agiles fondamentales, le Manifeste Agile décrit 12 principes de l'agilité. Ils vont nous offrir des exemples concrets de la manière dont un produit Agile doit se construire.

3.3.1 Livrer de la valeur au client

En raccourcissant le temps entre le cadrage et le début des développements, on raccourcit aussi le temps avant l'utilisation des fonctionnalités. Le fonctionnement en itérations permet de délivrer plus vite le produit aux utilisateurs.

3.3.2 Intégrer les demandes de changement

Pour le Manifeste Agile, le changement n'est pas négatif. Il est même positif. Il vaut mieux se tromper tôt et de réparer ses erreurs rapidement que de se rendre compte trop tard que le chemin emprunté n'était pas le bon.

3.3.3 Livrer fréquemment une version opérationnelle

Si vous souhaitez adopter le Manifeste Agile, vous devrez abandonner certaines mauvaises habitudes telles que graver un diagramme de Gantt pour une durée d'un an sur le marbre.
Découper votre produit en petits morceaux et vos développements en itérations courtes de deux à trois semaines. À la fin de chaque itération, livrez les nouvelles fonctionnalités développées et testez-les.

3.3.4 Assurer une coopération entre le client et l'équipe

Cela peut sembler difficile au premier abord. En effet, c'est comme si les deux parlaient deux langues différentes. Dans un sens, c'est le cas. Mais l'agilité permet de construire une sorte de pont entre les deux parties, leur permettant de se comprendre. L'équipe et le client doivent s'entendre, échanger et travailler ensemble. Certains outils et "rituels" facilitent cette collaboration.

3.3.5 Réaliser les projets avec des personnes motivées

La confiance mutuelle est un point essentiel du Manifeste Agile. En d'autres termes, le management d'une équipe agile doit se baser sur la transparence. Contrôler étroitement une équipe

de développeur ne va ni les engager, ni les motiver. Il est conseillé de les laisser faire ce qu'ils savent faire de mieux : développer.

L'agilité ne veut pas non plus dire aucun contrôle. Cela signifie qu'une équipe s'auto-organise. Dans une équipe agile, chacun a un rôle précis et le rôle de manager n'en fait pas partie. Il devra rester un peu à l'écart et s'appuyer sur les bons KPIs (key performance indicators) et les résultats des itérations pour mesurer le succès.

3.3.6 Privilégier le dialogue en face à face

La rédaction de documentation, les résumés de réunions, les présentations prennent du temps et diminuent la productivité d'une équipe Agile. Il n'y a rien de plus efficace que d'avoir une équipe travaillant sur le même produit dans un même espace de travail.

Les membres peuvent alors se poser directement les questions, obtenir les réponses dans la seconde.

D'autant plus qu'en posant ses questions dans un open-space, sans même prêter attention, chaque membre aura conscience des problématiques que son voisin rencontre.

3.3.7 Mesurer l'avancement sur la base d'un produit opérationnel

Vous ne mesurez pas les progrès en cochant les tâches et en vous déplaçant sur votre calendrier, mais par le taux d'utilisation des fonctionnalités par vos utilisateurs.

L'objectif du produit n'est pas le processus, c'est sa valeur. Le processus est ce qui vous permet de l'atteindre.

3.3.8 Faire avancer le projet à un rythme soutenable et constant

La raison de découper un produit en petites tâches est de garder vos équipes motivées.
Si vous travaillez sur un projet pendant une longue période, vos équipes rencontreront une lassitude. C'est inévitable.

Ne surchargez pas non plus l'équipe avec trop d'heures supplémentaires. Cela aura un impact sur la qualité de votre projet.

3.3.9 Contrôler l'excellence technique et à la conception

Que ce soit dans le code ou dans la méthodologie, la rigueur va favoriser la construction d'un produit de valeur.
Assurez-vous de bien respecter le Manifeste Agile.

3.3.10 Minimiser la quantité de travail inutile

Si votre objectif est de produire rapidement un produit fonctionnel, il faut veiller à ne pas se noyer sous les tâches complexes et inutiles. Gardez vos documentations simples, ne vous embarrassez pas de réunions parasites.

3.3.11 Construire le projet avec des équipes auto-organisées

Le Manifeste Agile favorise la responsabilisation des équipes.

Je vous conseille de donner aux équipes l'autonomie d'agir de manière indépendante. Ils pourront alors prendre des décisions rapidement en cas d'imprévus.

En fait, ils peuvent tout faire avec une plus grande agilité parce que vous leur avez donné la confiance nécessaire pour agir. Une équipe responsabilisée est une équipe qui fera de son mieux pour atteindre son objectif.

3.3.12 Améliorer constamment l'efficacité de l'équipe

Le dernier des 12 principes du manifeste agile est l'amélioration continue de l'efficacité. Ce dont vous avez besoin, c'est d'un groupe en constante évolution, constamment engagé et à la recherche de moyens d'améliorer la productivité.

3.4 Application : les 12 principes agiles

3.4.1 Contexte du projet

Une startup veut créer une plateforme web de gestion des abonnements (Netflix, Spotify, etc.) permettant aux utilisateurs de visualiser et d'optimiser leurs dépenses mensuelles.

L'équipe comprend :

- **1 Product Owner (PO)**
- **4 développeurs full-stack**
- **1 UX/ UI designer**
- **1 Scrum Master**
- Le projet est développé en suivant un cadre Scrum sur des **sprints de 2 semaines**.

3.4.2 Les 12 principes Agile illustrés dans le projet

1. Satisfaire le client grâce à des livraisons rapides et continues

Dès le premier mois, l'équipe livre un **MVP** permettant aux utilisateurs de lier leurs abonnements existants. Les retours du client sont rapidement intégrés dans le backlog pour prioriser les fonctionnalités importantes, comme l'analyse des dépenses.

2. Accueillir favorablement les changements, même tard dans le développement

Lors du 3e sprint, le client demande de modifier l'interface pour afficher des graphiques interactifs. L'équipe s'adapte et intègre cette demande dans les priorités, tout en maintenant un rythme de livraison régulier.

3. Livrer fréquemment un logiciel opérationnel

À la fin de chaque sprint, l'équipe livre une version fonctionnelle de la plateforme. Les livraisons incluent des fonctionnalités comme la liaison des abonnements, la catégorisation des dépenses et les notifications par email.

4. Collaboration entre le client et l'équipe tout au long du projet

Des démos sont organisées à la fin de chaque sprint pour montrer les progrès au client. Ces sessions permettent de discuter des ajustements nécessaires et de prioriser les prochaines tâches ensemble.

5. Construire des projets autour d'individus motivés

L'équipe est autonome dans la gestion des tâches. Les développeurs choisissent eux-mêmes les stories qui correspondent à leurs compétences, et le Scrum Master veille à ce que chacun reste motivé et soutenu.

6. Privilégier la communication directe

Plutôt que d'envoyer des emails, l'équipe organise des **stand-ups quotidiens** pour discuter des problèmes et coordonner les efforts. Une réunion rapide permet de débloquer un développeur face à une dépendance API.

7. Mesurer l'avancement par la livraison de fonctionnalités opérationnelles

Le succès est mesuré par les fonctionnalités déployées, comme la liaison des abonnements avec PayPal et Google. Ces fonctionnalités sont utilisées immédiatement par le client pour tester et donner son feedback.

8. Maintenir un rythme constant et soutenable

L'équipe s'engage sur une quantité de travail réaliste à chaque sprint, évitant le surmenage. Le Scrum Master s'assure qu'aucun membre ne prend une charge excessive.

9. Exigence d'excellence technique et de design

L'équipe suit les bonnes pratiques de développement :

- Mise en place de tests automatisés pour éviter les régressions.
- Design responsive et intuitif grâce à la collaboration entre le designer et les développeurs.

10. Simplicité : l'art de maximiser le travail non fait

L'équipe choisit de limiter les fonctionnalités dans la première version pour se concentrer sur l'essentiel : la gestion des abonnements. Par exemple, le suivi des dépenses par année est reporté à une future version.

11. Les meilleures architectures et conceptions émergent d'équipes auto-organisées

Chaque membre propose des solutions techniques ou des idées UX lors des réunions d'équipe. Par exemple, un développeur a proposé d'utiliser une architecture modulaire pour faciliter l'ajout de nouvelles fonctionnalités.

12. Régulièrement, l'équipe réfléchit à comment devenir plus efficace

À la fin de chaque sprint, une **rétrospective** est organisée. Lors d'une rétro, l'équipe a décidé d'améliorer la gestion des dépendances en intégrant des pipelines CI/CD (intégration et déploiement continus).

Synthèse des résultats

- Le projet progresse rapidement tout en restant flexible face aux changements.
- Les livraisons fréquentes permettent de tester et d'itérer avec le client.
- L'équipe reste motivée, engagée et maintient une bonne qualité de code.
- Grâce à des ajustements constants (rétrospectives), le rythme de travail est soutenable et efficace.

4. Méthodes agiles

Il existe différents frameworks Agile, aucune méthode ne s'étant imposée de manière universelle. Nombre de leaders choisissent d'associer les éléments de différents frameworks afin de créer une approche optimale pour leur équipe, leur secteur d'activité ou leur entreprise dans l'optique d'améliorer leurs performances et d'atteindre les objectifs.

Une fois qu'une organisation décide d'adopter une gestion de développement Agile, il reste encore à choisir la méthodologie la plus adaptée à son projet. En effet, les méthodes Agiles disponibles sont nombreuses et peuvent être source de confusion. Voici un aperçu de certaines des approches les plus répandues.

4.1 Scrum

La présente section explique avec un exemple la méthode agile scrum.

4.1.1 Principe

Courant en développement applicatif, le framework Scrum est une approche Agile qui permet aux membres de l'équipe de s'organiser de manière autonome pour effectuer diverses tâches abordées de façon itérative. En collaborant à la réalisation d'un objectif commun par le biais d'une série de réunions et d'outils structurés, les personnes qui participent à un projet Scrum peuvent faire part de leurs commentaires et progresser de manière incrémentielle au sein d'un cycle de développement tout en élaborant le produit final. Le framework Scrum est recommandé pour les environnements nécessitant des changements fréquents et des capacités d'adaptation.

La méthode agile Scrum particulièrement destinée à la gestion de projets informatiques tient son nom du monde du rugby. Le principe de Scrum est de pouvoir **modifier** la direction prise par le projet au fur et à mesure de son avancement. C'est exactement ce qui se passe lors d'un match de rugby, lors d'une mêlée (« scrum » en anglais).

La mêlée est donc une phase essentielle au rugby comme dans la gestion de projet. Si les conditions de réussite ne sont pas remplies, alors il faut réorienter le projet pour repartir sur de meilleures bases. Le **client** est étroitement impliqué grâce à la **livraison** régulière de prototypes opérationnels permettant de valider les développements. Cette gestion dynamique permet de s'assurer de la correspondance entre le besoin exprimé et le produit livré, et de réorienter au besoin les futurs développements.

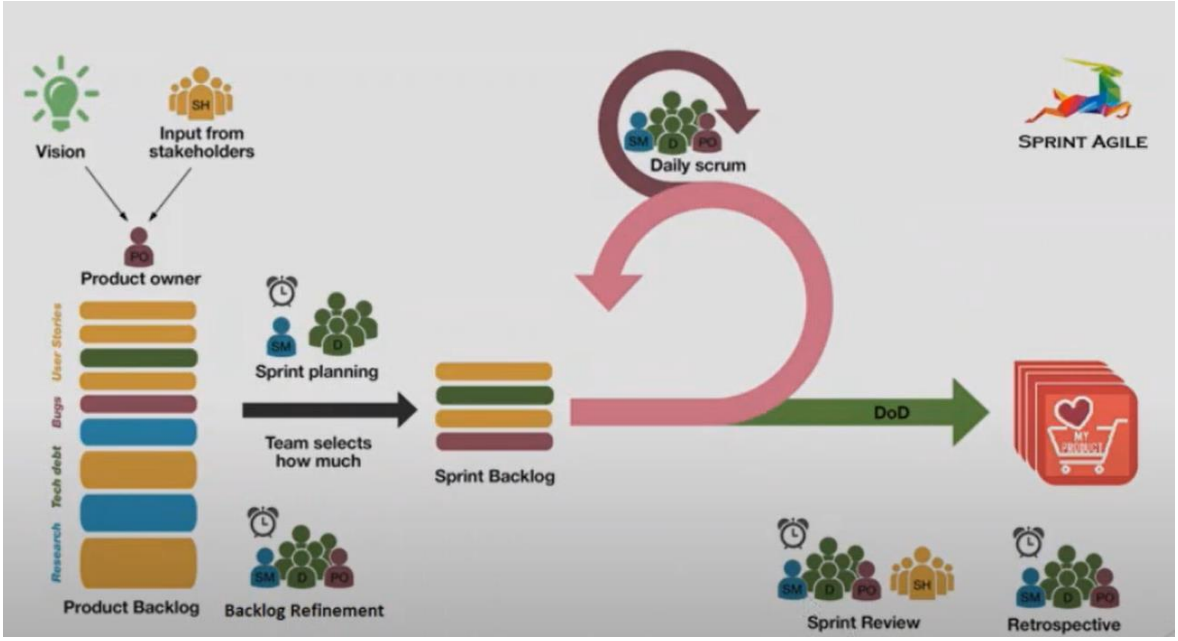


Figure 1 : le framework Scrum

4.1.2 User Story en Agile

Une **User Story** est une description simple et concise d'une fonctionnalité ou d'une valeur que le produit doit offrir à un utilisateur final. Elle est rédigée du point de vue de l'utilisateur, dans un langage non technique, et sert à capturer **ses besoins** ou **attentes**.

Elle est souvent utilisée dans les méthodologies Agiles (comme Scrum ou Kanban) pour définir les tâches à réaliser dans un projet.

La structure classique suit ce modèle :

En tant que [type d'utilisateur], je veux [ce besoin spécifique] afin de [bénéfice attendu].

Exemple :

En tant qu'utilisateur connecté, je veux recevoir une notification de rappel pour mes tâches importantes afin de ne pas oublier de les accomplir.

Les **critères d'acceptation** précisent les conditions qui doivent être remplies pour que la User Story soit considérée comme terminée.

- User Story :

  En tant qu'utilisateur, je veux pouvoir créer une tâche avec une date limite afin de mieux organiser mon emploi du temps.

- Critères d'acceptation :

- L'utilisateur peut entrer un titre et une description pour une tâche.
- L'utilisateur peut définir une date limite via un sélecteur de date.
- La tâche est sauvegardée automatiquement après la création.

Exemple d'User Stories dans un projet Agile

Projet : Application de gestion de tâches

1. **User Story 1 (Création de tâche)**

   En tant qu'utilisateur, je veux pouvoir créer une nouvelle tâche pour organiser mes activités.

   - **Critères d'acceptation** :
     - L'utilisateur peut saisir un titre et une description.
     - La tâche doit apparaître immédiatement dans la liste des tâches après sa création.

2. **User Story 2 (Notifications de rappel)**

   En tant qu'utilisateur, je veux recevoir une notification pour les tâches urgentes afin de ne pas les oublier.

   - **Critères d'acceptation** :
     - L'utilisateur reçoit une notification 30 minutes avant l'échéance.
     - Les notifications ne sont envoyées que pour les tâches avec une échéance définie.

3. **User Story 3 (Tri des tâches)**

   En tant qu'utilisateur, je veux trier mes tâches par date ou priorité afin de mieux visualiser celles qui sont les plus importantes.

   - **Critères d'acceptation** :
     - L'utilisateur peut basculer entre les tris par date ou priorité.
     - Le tri doit être sauvegardé pour la prochaine session.

4.1.3 Exemple applicatif de la méthode scrum

Voici un exemple concret d'application de la méthode Agile Scrum dans le développement d'une application mobile de gestion de tâches.

a. Contexte du projet

Une startup veut développer une application mobile de gestion de tâches pour aider les utilisateurs à organiser leur travail quotidien. L'équipe Scrum est composée de :

- **Product Owner (PO)** : Détermine les fonctionnalités prioritaires selon les besoins des utilisateurs.
- **Scrum Master** : S'assure que la méthodologie Scrum est bien appliquée et résout les obstacles.
- **Équipe de développement** : Composée de développeurs, designers et testeurs.

b. Cycle Scrum appliqué

1. Création du Product Backlog

Le **Product Owner** liste toutes les fonctionnalités souhaitées sous forme de **User Stories**, par exemple :

- **US01** : En tant qu'utilisateur, je veux pouvoir créer une tâche avec un titre et une date limite.
- **US02** : En tant qu'utilisateur, je veux pouvoir modifier ou supprimer une tâche.
- **US03** : En tant qu'utilisateur, je veux être notifié quand une tâche arrive à échéance.

Chaque User Story est priorisée selon la valeur ajoutée pour l'utilisateur.

2. Sprint Planning (Planification du Sprint)

L'équipe sélectionne un ensemble de User Stories à réaliser durant un Sprint (ex. : 2 semaines). Par exemple :

- US01 et US02 sont sélectionnées pour le Sprint 1.

Les tâches techniques nécessaires sont définies dans un **Sprint Backlog**, par exemple :

- Créer l'interface utilisateur pour l'ajout de tâches.
- Développer la base de données pour stocker les tâches.
- Implémenter la fonctionnalité d'édition et de suppression.

3. Développement et Daily Scrum

- Chaque jour, un **Daily Scrum** (15 min) permet à l'équipe de répondre à trois questions :
- Qu’ai-je fait hier ?
- Que vais-je faire aujourd’hui ?
- Quels obstacles rencontrés ?

Exemple :

- Un développeur a terminé la base de données et commence l'interface utilisateur.
- Un autre a un problème d’affichage sur certaines tailles d’écran.

4. Sprint Review (Revue de Sprint)

À la fin du sprint, une démo est présentée au **Product Owner** et aux parties prenantes.

✓ Le PO valide la création, l'édition et la suppression des tâches.

✗ La synchronisation avec le cloud pose problème → reportée au prochain Sprint.

5. Sprint Retrospective

L'équipe identifie les améliorations possibles :

* **Ce qui a bien fonctionné** : Bonne communication et livraison des fonctionnalités principales.

* **À améliorer** : Besoin de mieux estimer le temps de certaines tâches.

* **Actions à prendre** : Utiliser des estimations en "Story Points" plus précises.

c. Répétition des Sprints

Chaque Sprint apporte de nouvelles fonctionnalités jusqu'à la sortie de l'application.

d. Bénéfices de Scrum dans ce projet

- **Livraison rapide** : Fonctionnalités livrées toutes les 2 semaines.
- **Flexibilité** : Adaptation en fonction des retours des utilisateurs.
- **Collaboration efficace** : Communication quotidienne pour résoudre les problèmes rapidement.

Résumé : Grâce à Scrum, l'application est développée en itérations successives, avec une amélioration continue et une forte réactivité aux besoins des utilisateurs.

4.2 Extreme Programming (XP)

Extreme Programming, ou XP, est une méthode agile de gestion de projet particulièrement bien adaptée aux projets de développement informatique. Elle a été conçue par Kent Beck pour accélérer les développements alors qu'il travaillait pour la société Chrysler. L'idée lui est venue alors qu'il devait intervenir sur un logiciel de paie écrit en langage Smalltalk ayant accumulé une dette technique considérable, le rendant particulièrement complexe à maintenir et à faire évoluer.

Le principe fondamental de la méthode XP est de faire collaborer étroitement tous les acteurs du projet et d'opter pour des **itérations** de développement très **courtes**. La planification des tâches reste très souple et l'estimation des charges simplifiée par des projections à très court terme. Ainsi la correspondance entre ce qu'attend le client et les réalisations est garantie. Les

fonctionnalités sont livrées régulièrement, afin d'être testées et validée au travers de prototypes opérationnels. L'Extreme Programming préconise également le travail en **binôme** des développeurs, facilitant ainsi la production d'un code simple, facilement lisible et maintenable.

4.3 Kanban

Kanban est une approche de gestion des workflows **Lean** pensée pour les équipes qui cherchent à simplifier le travail en cours et ont un flux constant de demandes entrantes. Cette approche est efficace, car le processus représente visuellement la **file d'attente** des tâches, en faisant progresser les éléments en fonction des étapes ou des ressources requises. La charge de travail (et non les membres de l'équipe) est gérée par la ou le responsable du projet, qui alimente la file d'attente et permet aux membres de l'équipe de choisir des éléments à développer ou à réviser.

La méthode Kanban est un outil d'aide à la gestion de projet. Elle a initialement été conçue afin de mieux gérer la production, mais elle permet également d'assurer une production de **qualité**. La méthode remplit sa fonction tout en faisant participer activement chaque acteur impliqué.

L'objectif ici consiste avant tout à s'adapter aux besoins du consommateur. Cela conduit à éviter toute surproduction et, de manière générale, tout gaspillage de ressources. De même, les délais de production sont relativement raccourcis, de même que les coûts.

De plus, pour une mise en œuvre plus simple et plus aisée, cette méthode est axée sur une approche visuelle. Il existe donc des **tableaux Kanban** sur lesquels sont inscrites les tâches à faire, celles en cours et celles qui sont terminées, le tout à l'aide de petites cartes.

La méthode Agile Kanban apporte flexibilité et facilité à l'accomplissement des tâches. Elle exclut d'attribuer de gros volumes de tâches à un nombre limité de personnes, mais commande plutôt de scinder chaque activité en petites sous-activités facilement exécutables.

4.4 Feature Driven Development (FDD)

Le Développement Dirigé par les Fonctionnalités, est une méthode de gestion de projet basée sur la **gestion des risques**. Les développements sont organisés en **itérations courtes** autour

de **fonctionnalités testables** par l'utilisateur. **L'utilisateur** est ainsi **impliqué** dans les développements, peut **suivre** l'avancement du projet et la **validation** des fonctionnalités. Aucune méthode de programmation n'est préconisée, c'est réellement la fonctionnalité qui est mise en avant.

Un projet géré par FDD est **découpé** en plusieurs grandes étapes. La première consiste en la constitution d'un **modèle** général du produit, qui va définir le **périmètre** global de réalisation. Vient ensuite la construction de la **liste complète des fonctionnalités** à réaliser. Il est impératif lors de cette étape **d'impliquer au maximum le client**. Ces fonctionnalités sont finalement regroupées en fonction de leurs **caractéristiques communes et priorisées**. Les deux dernières étapes verront le jour sous la forme **d'itérations**, et englobent d'une part la conception technique des fonctionnalités, puis leur réalisation.

Cette méthode donne une importance plus grande à la **phase de conception**, quitte à démarrer la réalisation plus lentement, afin d'avoir un modèle plus **solide**.

4.5 Lean Software Development

Le Lean Software Development est basé sur sept grands principes.

- Éliminer les gaspillages (finition partielle, processus inutiles, fonctionnalités non nécessaires, modification de l'équipe, retards…),
- Favoriser l'apprentissage (multiplication des sources d'apprentissage, synchronisation des équipes…),
- Reporter les décisions (jusqu'au dernier moment raisonnable, pour éviter de longues discussions sources de pertes de temps et les décisions irrévocables),
- Livrer vite (livraisons rapides et régulières de façon à avoir un retour client rapide également),
- Responsabiliser l'équipe (favoriser l'autonomie et le leadership des équipes, partir du principe que les intervenants connaissent leur travail, faciliter le développement de l'expertise),
- Construire la qualité (elle doit être placée au cœur du projet, de la conception à la réalisation),

- Optimiser le système dans son ensemble (mise en place de mesures de performances complètes, pour avoir en permanence une vision globale du produit, et gérer les différentes interactions et dépendances).

Avec la méthode Lean, la qualité est réellement placée au cœur de la **gestion** du projet, en optimisant notamment l'ensemble des processus d'apprentissage, de décision, de livraison et de mesure de performances.

4.6 Rational Unified Process (RUP)

Cette méthode est un mélange des pratiques classiques et agiles de gestion de projet. Les développements sont **itératifs** et **incrémentaux**. Chaque **itération** respecte un **cycle** comprenant quatre phases qui sont **lancement, conception, réalisation et livraison**. Les développements sont guidés par des **cas d'utilisation**. On commence par les fonctionnalités **génériques** pour aller ensuite de plus en plus vers le **spécifique**.

4.7 Agile Unified Process (Agile UP or AUP)

Agile Unified Process (ou Processus Unifié Agile) est une **version simplifiée** du Rational Unified Process, ou RUP. Il s'agit d'une méthode de développement d'applications métier utilisant les techniques agiles du **TDD** (Test Driven Development ou développement piloté par les tests), du **MDD** (Model Driven Development ou développement piloté par le modèle) et de la **gestion du changement**.

La méthode est divisée en quatre phases :

- Lancement : identification du périmètre du projet, définition de la ou des architectures potentielles pour le système, implication des intervenants et obtention du budget.
- Conception : définir l'architecture du système et démonstration de sa pertinence.
- Réalisation : développement du logiciel lors d'un processus incrémental dans l'ordre de priorité des fonctionnalités.
- Livraison : validation et déploiement du système en production.

4.8 Crystal (Clear/Orange)

Mettant l'accent sur la **communication** et les **interactions** plutôt que sur les **processus**, le framework Agile dénommé Crystal permet aux membres de l'équipe de s'exprimer sur les **obstacles** ou sources **d'efficacité** qu'ils constatent afin d'orienter leur **workflow** de manière **autonome**. Selon les règles de collaboration en général, les équipes sont aussi autorisées à **partager** des informations entre elles au lieu de devoir respecter des obligations rigides en matière de documentation et de **reporting**. Le framework Crystal peut s'avérer efficace pour les équipes dont les membres s'entendent bien, et évite la dérive des objectifs.

La méthode Crystal Clear est particulièrement adaptée aux **petites** équipes de développement. Idéalement, l'équipe est composée d'un **architecte** et de deux à six ou sept **développeurs**, situés à proximité les uns des autres, de façon à faciliter la communication, dans un local calme. Des **tableaux blancs** servent de supports afin que tous aient un accès rapide à toutes les informations. Les rythmes de **développement et de livraison sont rapides** (toutes les deux semaines ou une fois par mois) afin que les utilisateurs puissent passer les tests.

Durant tout le processus de développement, l'équipe se remet en question en **permanence** afin d'améliorer continuellement sa façon de travailler.

4.9 Dynamic Systems Development Method (DSDM)

Pour les entreprises qui cherchent à accélérer le rythme de sortie des versions, le framework DSDM peut s'avérer intéressant. La méthode consiste à créer des **stratégies** qui mettent l'accent sur la **publication de versions fréquentes**, avec de nombreuses **itérations**, en tenant compte de la probabilité de **révisions** et **retouches** en cours de route ou par la suite. Les membres de l'équipe Agile sont toujours tenus de respecter les processus et procédures, mais travaillent à un rythme plus soutenu que leurs homologues strictement Scrum.

Cette méthode s'articule autour de neufs grands principes qui sont : la participation des utilisateurs, l'autonomie de l'équipe projet, la transparence des développements, l'adéquation avec le besoin, le développement itératif et incrémental, la réversibilité permanente, la synthèse du projet, les tests automatisés et continus et enfin la coopération entre tous les intervenants.

Le projet commence par une étude de faisabilité afin de décider s'il faut le faire ou non. Un rapport est rédigé, et éventuellement, un prototype est créé pour démontrer la faisabilité de l'application. S'il a été décidé de continuer, une analyse fonctionnelle est réalisée et les spécifications sont rédigées. A partir de ce moment-là, des **itérations** de conception technique et de développement sont mises en place, puis au final, l'application est livrée en production.

4.10 Adaptive software development (ASD)

ASD est une méthode de développement rapide d'applications. Le principe consiste à automatiser et à industrialiser un maximum de processus. Des outils de modélisation sont utilisés et une usine logicielle est mise en place de façon à générer un maximum de code informatique automatiquement. Un atelier de génie logiciel (AGL) permet ensuite aux développeurs de modifier l'application ainsi générée. Pour finir, une usine de livraison assure l'automatisation de l'ensemble des processus de déploiement. La méthode ASD est indépendante de toute méthodologie ou langage de programmation.

4.11 Behavior driven development (BDD)

Dans cette méthode, c'est le **langage naturel** qui est mis en avant. Plutôt que de décrire les solutions techniques à mettre en œuvre, ce sont les objectifs des fonctionnalités qui sont décrites par les utilisateurs. Ces derniers sont donc particulièrement impliqués dans le processus de **fabrication**. Le comportement cible de l'application est donc décrit au travers d'exemples permettant aux développeurs de mieux cerner les objectifs. Les expressions « Etant donné », « quand », « alors » et « et » sont employées dans les scénarios décrits, qui sont également utilisés pour créer des séries de **tests de non régression**. Les scénarios sont écrits collectivement avec le client, les développeurs et toutes les équipes impliquées dans le processus.

4.12 Conception pilotée par le domaine (DDD domain-driven design)

Le Domain Driven Design est une technique de conception d'applications informatiques. La conception est centrée sur le domaine **métier** et non sur les aspects techniques. Elle permet aux équipes techniques et fonctionnelles de communiquer ensemble afin d'obtenir un **modèle** commun de l'application, compréhensible par tous. Les développeurs acquièrent ainsi une meilleure

connaissance du fonctionnel et de son vocabulaire spécifique, et l'équipe métier a une meilleure vision des contraintes techniques. Grâce à cet outil, on obtient une meilleure communication entre les différents intervenants et une conception modulaire facilitant à terme la maintenabilité de l'application et la réutilisabilité de ses composants.

4.13 Test driven development (TDD)

Le développement piloté par les tests est une technique de développement qui associe l'écriture des **tests unitaires**, la programmation et le remaniement du code. Les **tests unitaires sont écrits avant le code**. Chaque test décrit un élément de la fonctionnalité. Le développeur écrit le minimum de code possible pour que le test passe, et qu'il échoue pour des raisons prévisibles. Au fur et à mesure de l'avancement, le code source doit être simplifié autant que nécessaire et continuer à passer les tests. Les tests s'accumulent durant le développement et sont exécutés automatiquement de façon très régulière. Le TDD est toujours **associé à des outils** d'automatisation de tests unitaires liés au langage de programmation utilisé.

4.14 Disciplined Agile Delivery (DAD)

DAD est une méthode d'aide à la décision et d'industrialisation qui a pour objectif de simplifier l'intégration des processus liés à une gestion de projet **incrémentale et itérative**, agile. Elle s'applique de façon particulièrement efficace en association avec une méthode de développement telle que Scrum ou Lean. Il s'agit d'une méthode hybride, destinée à faciliter l'adoption de l'agilité au sein d'une organisation de taille significative. Tous les aspects de l'intégration de l'agilité dans le développement logiciel sont pris en charge en tenant compte du contexte global du projet et surtout de l'entreprise. Lorsqu'une méthode agile doit être appliquée à un projet impliquant des équipes importantes, DAD peut être d'un grand secours pour faciliter l'adoption des différents processus.

4.15 Enterprise Unified Process (EUP)

EUP est une extension des méthodes d'industrialisation comme DAD ou comme les dérivés de Unified Process (UP), ou des méthodes de développement agile comme Scrum. EUP apporte deux autres phases à la fin des cycles des autres méthodes agiles :

- Mise en production : maintenance en condition opérationnelle des systèmes déployés.
- Retrait de production : processus de retrait de la production des systèmes déployés.

La phase de retrait de production peut être mise en œuvre pour plusieurs raisons comme le remplacement complet du système, la fin du support de la version courante, la redondance du système ou encore l'obsolescence de l'application.

5. Itération, cycle de développement, planification adaptative

5.1 Itération

- En **mathématiques**, une itération désigne l'action de répéter un processus. Le calcul itératif permet l'application à des équations récursives. Le terme itération est issu du verbe latin iterare qui signifie « cheminer » ou de iter « chemin ». Le processus d'itération est employé fréquemment en algorithmique.
- Action de répéter, de faire de nouveau; fait d'être répété.
- En **informatique**, procédé de calcul répétitif qui boucle jusqu'à ce qu'une condition particulière soit remplie.
- Il peut être remplacé par différents synonymes tels que "répété", "réitéré", "renouvelé", "recommencé", "répétitif" ou encore "fréquentatif".
- En **gestion de projet** c'est le processus itératif est un cycle de travail répété créé par une équipe pour développer rapidement un prototype de son produit et obtenir du feedback des clients et des parties prenantes.
- Une **itération agile** : L'itération est le principe de répéter un processus plusieurs fois dans le but d'améliorer un résultat.
- Une **approche itérative et incrémentale (et adaptative**) : Dans le monde du développement, nous utilisons cette approche pour améliorer une fonctionnalité. Nous sommes pragmatiques et nous savons bien que rien n'est parfait la première fois. Une fonctionnalité a besoin d'être testée dans des conditions réelles pour s'adapter à ses utilisateurs et pour révéler des bugs éventuels
- Un **processus incrémental** permet de construire un produit petit à petit en le découpant en pièces détachées. Elles sont le plus souvent indépendantes les unes des autres mais ont pour caractéristiques d'améliorer le produit

5.2 cycle de développement

Le cycle de vie du développement logiciel (SDLC) désigne une méthodologie comportant des processus clairement définis pour créer des logiciels de haute qualité. L'un des principaux objectifs du SDLC est de produire un logiciel robuste dans le cadre d'un budget et d'un calendrier précis. Essentiellement, le SDLC décrit un plan détaillé avec des étapes qui comprennent chacune leur propre processus et leurs propres produits livrables. Par rapport aux autres méthodes de production, le SDLC accélère le développement des projets et minimise les coûts. Il existe **différents types de cycles** de développement entrant dans la réalisation d'un logiciel. Ces cycles prennent en compte toutes les étapes de la conception d'un logiciel.

5.3 Planification adaptative

La planification adaptative est un processus de planification qui s'adapte aux conditions changeantes et aux nouvelles informations. Cette approche est similaire à la planification dynamique, mais elle implique des ajustements plus fréquents, car elle s'adapte aux changements qui peuvent survenir plus rapidement. Elle est essentielle dans les environnements complexes, car elle permet aux entreprises de s'adapter aux conditions changeantes et de s'assurer qu'elles prennent les bonnes décisions.

5.3.1 Fonctionnement d'une planification adaptative

La planification adaptative implique un cycle continu de planification et de mise à jour. Les entreprises planifient d'abord leurs objectifs et leurs stratégies, puis surveillent constamment l'environnement pour détecter les changements. Lorsqu'un changement est détecté, l'entreprise peut ajuster ses plans et ses stratégies pour correspondre aux nouvelles conditions.

5.3.2 Exemples de planification adaptative

- Une entreprise peut planifier ses activités de marketing et de publicité en fonction des tendances du marché et des informations sur la concurrence.
- Un détaillant peut ajuster ses plans de stock en fonction des prévisions de ventes.
- Une start-up peut ajuster ses plans de produits en fonction des commentaires des utilisateurs.

5.3.3 Avantages de la planification adaptative

La planification adaptative offre de nombreux avantages :

- Elle permet aux entreprises de rester à jour et de s'adapter aux changements dans leur environnement.
- Elle peut améliorer l'efficacité des plans et des stratégies en prenant en compte les nouvelles informations et en s'assurant que les objectifs sont toujours en ligne avec l'environnement.
- Elle peut améliorer la prise de décision en fournissant des informations plus précises et à jour.

La planification adaptative est un outil puissant pour les entreprises qui cherchent à s'adapter aux conditions changeantes et à prendre des décisions plus éclairées.

6. Equipe agile

Une équipe agile est un groupe de personnes multidisciplinaires qui collaborent pour développer des produits ou des solutions en suivant les principes du **Manifeste Agile**. Ce type d'équipe se distingue par sa flexibilité, son adaptabilité et son focus sur la livraison fréquente de valeur au client.

6.1 Caractéristiques principales d'une équipe agile :

- **Taille réduite** : Généralement entre 5 et 11 personnes, pour maintenir une communication fluide et une prise de décision rapide.
- **Multidisciplinarité** : L'équipe regroupe des compétences variées (développeurs, testeurs, concepteurs, analystes métier, etc.) nécessaires pour accomplir le travail de bout en bout.
- **Auto-organisation** : Les membres de l'équipe sont responsables de leur organisation et de la planification de leurs tâches, sans supervision rigide.
- **Focus sur l'utilisateur final** : Les besoins du client ou de l'utilisateur sont au cœur des décisions et priorités de l'équipe.
- **Cycles courts et itératifs** : Le travail est organisé en **sprints** ou **itérations** (souvent de 1 à 4 semaines), permettant d'adapter rapidement les livrables en fonction des retours.

- **Communication fréquente** : Les membres se réunissent régulièrement (par exemple, lors de la **Daily Scrum**) pour partager leur avancement, identifier les obstacles et ajuster leurs plans.
- **Amélioration continue** : À la fin de chaque cycle, l'équipe organise une **rétrospective** pour identifier ce qui a bien fonctionné et ce qui peut être amélioré.

6.2 Rôles de membres de l'équipe agile Scrum

En général, les équipes Agile répartissent clairement les responsabilités entre différents rôles. La méthodologie en propose d'ailleurs de nombreux établis de manière formelle. Voici les membres qui sont indispensables à la réussite du projet :

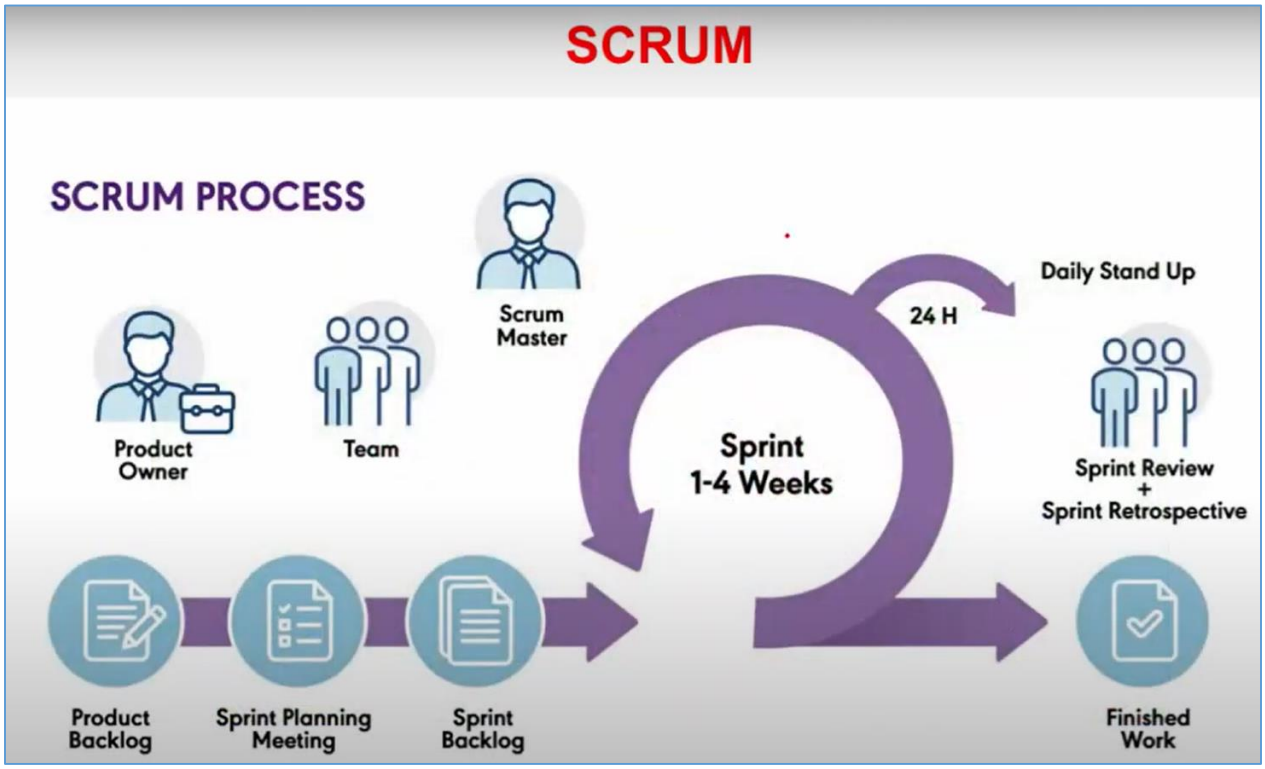


Figure : les membres de méthode agile Scrum

6.2.1 Rôles de product owner (chef de projet)

Au sein d'une équipe Agile, cette personne est chargée de définir les tâches, les échéances et les attentes en fonction des objectifs globaux de l'entreprise. Également chargée de superviser l'objet du projet, elle (c'est-à-dire la ou le gestionnaire de projet en marketing Agile) est le **leader**

désigné de l'équipe. Ce poste est généralement occupé par un membre de la direction : CMO, directeur ou directrice, manager, etc. Son principal objectif est d'aider à fractionner les missions en itérations gérables et coïncidant avec les méthodes de développement Agile.

6.2.2 Rôles de Scrum Master

Cette personne aide à matérialiser les listes de tâches établies par la ou le product owner en créant des processus et des structures d'équipe qui permettent à chacun et à chacune de faire preuve d'efficacité. Elle **supervise** à ce titre les modalités d'exécution du projet et joue un rôle essentiel dans la communication interne et externe de l'équipe.

La ou le Scrum Master, un rôle propre à la méthodologie Scrum filtre les demandes qui parviennent à l'équipe Agile, gère le backlog et anime toutes les réunions Scrum. Ce poste n'est pas forcément occupé par un membre de la direction. Il peut même être assumé à tour de rôle par chaque membre de l'équipe Agile.

Si les responsables produit et les Scrum Masters sont sous les feux de la rampe, d'autres rôles sont aussi importants. Par exemple, une partie prenante interne peut être à l'origine de la demande de projet ou de résultat et être informée de l'avancement dans le cadre de la coordination avec d'autres équipes en vue d'un lancement ou d'une mise à jour. Les parties prenantes formulent des commentaires essentiels qui peuvent affecter l'orientation des tâches.

Certaines personnes comme les spécialistes du développement font aussi partie de l'équipe Agile et constituent la principale source de productivité et de résultats de l'équipe.

6.2.3 Rôles de l'équipe de développement

Elle peut comprendre toutes sortes de personnes, y compris des designers, des rédacteurs, des programmeurs, et bien d'autres. C'est un peu comme concevoir un projet de maison et engager un développeur. Il développe le projet et réalise le travail.

L'équipe Scrum contient entre 2 et 5 développeurs. Elle doit répondre à tous les besoins techniques nécessaires pour livrer le produit ou le service. L'équipe de développement est guidée par le Scrum Master, mais doit être autonome.

Chaque développeur doit être polyvalent et suffisamment responsable pour effectuer toutes les tâches requises. Ils doivent se concentrer sur les développements. Pour cela, le Product Owner se charge de traduire les besoins des utilisateurs en **user story** que les développeurs exécuteront lors du sprint. Pour que le cadre soit respecté, il faut aussi être familier avec ceux des personnes travaillant avec l'équipe sans en faire partie.

6.3 Collaboration et communication

Scrum permet également de **minimiser les risques** liés à la **communication et à la collaboration** en éliminant les silos et en encourageant la transparence.
Les membres de l'équipe peuvent en effet travailler ensemble pour identifier les problèmes et trouver des solutions en temps réel. En outre, les parties prenantes sont impliquées dès le début du processus de développement, ce qui leur permet de fournir des commentaires réguliers sur les produits et de les améliorer en temps réel.

En utilisant Scrum, les équipes peuvent donc faciliter **la collaboration entre les équipes et les parties prenantes**, ce qui se traduit par une meilleure communication, une meilleure compréhension des objectifs et une réduction des risques.

La collaboration est l'un des plus importants principes de l'approche Agile.

Dans une équipe Agile, la collaboration entre les membres est essentielle pour assurer la réussite du projet. Chaque membre de l'équipe apporte des compétences et des connaissances uniques, et la collaboration permet de maximiser les avantages de chaque personne.

Les membres de l'équipe doivent travailler ensemble pour partager des idées, résoudre les problèmes et prendre des décisions ensemble. En collaborant étroitement, l'équipe peut s'assurer que les objectifs du projet sont atteints de manière efficace et efficiente.

Une collaboration efficace dans une équipe Agile nécessite également une communication ouverte et transparente. Les membres de l'équipe doivent être en mesure de communiquer librement et honnêtement pour résoudre les problèmes rapidement et éviter les erreurs coûteuses. La communication doit être continue et régulière pour **permettre à l'équipe de s'adapter rapidement aux changements et aux défis imprévus.**

Enfin, la collaboration ne se limite pas aux membres de l'équipe, mais implique également les parties prenantes externes. Les clients, les utilisateurs finaux et les parties prenantes doivent

être impliqués tout au long du processus de développement pour assurer que leurs besoins sont pris en compte.

On conclusion, la collaboration est essentielle pour réussir un projet Agile et pour satisfaire les parties prenantes.

7. Développement itératif et incrémental

Le **développement itératif et incrémental** est une approche utilisée dans les méthodologies agiles (comme Scrum, Kanban, ou XP) pour construire un produit ou un système progressivement, tout en s'adaptant aux retours et aux changements. C'est une alternative aux approches traditionnelles comme le cycle en V ou le modèle en cascade, qui demandent de tout planifier et concevoir à l'avance.

7.1 Développement itératif

L'approche itérative consiste à développer un produit en plusieurs **cycles (ou itérations)** successifs, où chaque cycle permet d'améliorer ou de corriger ce qui a été produit précédemment.

Caractéristiques principales :

- L'équipe produit une version "intermédiaire" du produit à chaque itération.
- Les retours des parties prenantes ou utilisateurs sont intégrés pour affiner ou ajuster le produit.
- Chaque itération a pour objectif d'améliorer la compréhension des besoins ou de corriger les erreurs identifiées précédemment.

7.2 Développement incrémental

Dans le développement incrémental, le produit est construit en **ajoutant progressivement des fonctionnalités**, ou des "incréments". À chaque étape, une nouvelle partie du produit est ajoutée à ce qui a été construit précédemment.

Caractéristiques principales :

- Le produit est livré de manière **progressive**, par blocs fonctionnels.
- Chaque incrément est testé et peut, dans certains cas, être livré directement au client.
- Les fonctionnalités sont priorisées pour maximiser la valeur à chaque livraison.

7.3 Développement itératif et incrémental combiné

La combinaison des deux approches permet de :

1. **Planifier les incréments** de manière progressive (développement incrémental).
2. **Améliorer chaque incrément** à travers des cycles successifs (développement itératif).

Exemple d'application :

Prenons le développement d'une application mobile :

1. **Sprint 1** : L'équipe livre une version minimale avec une fonction de connexion.
2. **Sprint 2** : Un système de profil utilisateur est ajouté (incrément), et des améliorations à la connexion sont apportées en fonction des retours (itératif).
3. **Sprint 3** : Une messagerie est ajoutée, et des bugs des sprints précédents sont corrigés.

À la fin de chaque sprint (itération), un produit utilisable est disponible, avec des fonctionnalités de plus en plus riches.

7.4 Avantages du développement itératif et incrémental

- **Flexibilité** : Permet d'intégrer facilement des changements en cours de développement.
- **Réduction des risques** : Les erreurs sont identifiées et corrigées rapidement grâce aux cycles fréquents.
- **Valeur continue** : Les parties prenantes voient rapidement des résultats concrets.
- **Collaboration accrue** : Encourage une communication constante entre l'équipe et les parties prenantes.

7.5 désavantages du développement itératif et incrémental

1. Complexité de la gestion

- La planification des itérations et des incréments peut devenir complexe, surtout pour des projets avec de nombreuses dépendances entre les fonctionnalités.
- Nécessite une gestion rigoureuse des priorités, des ressources et du temps pour éviter que le projet ne devienne chaotique.

2. Difficulté à avoir une vision d'ensemble

- L'accent mis sur les cycles courts peut entraîner un manque de vue globale sur le produit final.
- Les équipes peuvent se concentrer trop sur les livraisons immédiates au détriment de la stratégie long terme ou de l'architecture globale.

3. Dépendance aux retours fréquents

- Le succès repose souvent sur la disponibilité des parties prenantes et des utilisateurs pour fournir des retours après chaque incrément ou itération.
- Si les retours sont tardifs, imprécis ou inexistants, cela peut nuire à l'efficacité du processus.

4. Risque de "réécriture" fréquente

- Les ajustements ou changements constants (liés aux itérations) peuvent entraîner une réécriture ou des refontes importantes, ce qui peut augmenter les coûts et les délais.
- Les bases techniques (architecture, code) peuvent devenir instables si elles ne sont pas bien conçues dès le départ.

5. Possible insatisfaction des parties prenantes

- Les parties prenantes peuvent mal comprendre le concept d'incréments : elles peuvent s'attendre à voir un produit entièrement fonctionnel après chaque livraison, ce qui peut créer de la frustration.
- Si le produit intermédiaire ne répond pas aux attentes, cela peut nuire à la confiance dans l'équipe.

6. Risque de perte de qualité

- La pression pour livrer des incréments rapidement peut parfois conduire à une baisse de la qualité (ex. : code mal testé, dette technique accrue).
- Les équipes peuvent négliger les tests ou la documentation pour respecter les délais serrés des sprints.

7. Nécessite une équipe expérimentée

- Les membres de l'équipe doivent être à l'aise avec les méthodologies agiles et capables de travailler dans un cadre auto-organisé.
- Une équipe inexpérimentée peut avoir du mal à s'adapter aux changements fréquents ou à gérer les priorités dynamiques.

8. Coût plus élevé à court terme

- Comparé à une méthode traditionnelle où la planification et la conception sont exhaustives dès le début, le développement itératif et incrémental peut entraîner des coûts initiaux plus élevés, notamment en raison des cycles de tests et ajustements fréquents.

9. Difficulté à arrêter le projet

- Dans certaines situations, les parties prenantes peuvent avoir du mal à définir un point final, car le produit est constamment amélioré.
- Cela peut conduire à un "effet tunnel", où les itérations se poursuivent sans aboutir à une véritable finalisation.

10. Dépendance à une communication efficace

- Une mauvaise communication entre les membres de l'équipe ou avec les parties prenantes peut compromettre les itérations et conduire à des malentendus sur les attentes.

8. Les parties prenantes (**stakeholders**)

Les **parties prenantes** (ou **stakeholders**) sont toutes les personnes, groupes ou organisations ayant un intérêt direct ou indirect dans le projet ou son résultat final. Elles peuvent influencer le projet, être affectées par celui-ci ou en bénéficier. Voici les principaux types de parties prenantes dans un projet (aussi comme un développement logiciel en méthode Scrum) :

8.1 Parties prenantes internes

Les parties prenantes internes, quant à elles, sont des groupes d'intérêt qui ont une influence directe sur les processus et le succès de l'entreprise. Il s'agit notamment du propriétaire, de la direction et des collaborateurs. Par leur travail, les collaborateurs ont un impact direct sur les processus et donc sur le succès de l'entreprise. Ils font donc partie des parties prenantes internes. Plus un produit est réussi, plus les ventes sont élevées et plus il est probable que les emplois soient durables et bien rémunérés.

Par conséquence, ces acteurs font partie de l'organisation ou de l'équipe projet.

- **Équipe Scrum** :
    - **Product Owner** : Responsable de la définition des besoins, de la priorisation des tâches dans le Product Backlog et du lien avec les autres parties prenantes.
    - **Scrum Master** : Responsable de faciliter le processus Scrum, de lever les obstacles et d'assurer que l'équipe respecte les principes Agile.
    - **Développeurs** : Contribuent directement à la création de l'incrément fonctionnel.
    - **Testeurs/QA (Assurance Qualité)** : Garantissent que le produit répond aux attentes et fonctionne correctement.

8.2 Parties prenantes externes

Une partie prenante extérieure à l'entreprise relève du groupe des parties prenantes externes. Cette catégorie de parties prenantes n'a pas d'impact direct sur le succès ou les processus de l'entreprise. Il s'agit par exemple des banques, des bailleurs de fonds, des investisseurs, des actionnaires, des fournisseurs, des associés, des associations, de l'État ou encore des clients. En tant que bailleurs de fonds, les banques, par exemple, ont intérêt à ce qu'une entreprise soit performante, car elles ont tout à gagner à ce que leurs investissements soient fructueux.

Par conséquence, ces acteurs n'appartiennent pas directement à l'équipe projet, mais ont un intérêt dans ses résultats.

- **Clients ou utilisateurs finaux** :
    - Ce sont ceux qui utiliseront le produit final. Ils peuvent être des individus, des groupes ou des entreprises.
    - Exemple : Pour une application mobile, les utilisateurs finaux sont les personnes qui téléchargeront et utiliseront l'application.
- **Commanditaires (Sponsors)** :
    - Ils financent le projet ou apportent les ressources nécessaires.
    - Exemple : Une entreprise investissant dans le développement d'une application pour répondre à ses besoins commerciaux.
- **Direction ou management** :
    - Fournissent la vision stratégique et veillent à ce que le projet soit aligné avec les objectifs de l'organisation.
- **Partenaires ou fournisseurs** :
    - Fournissent des services, des outils ou des produits externes nécessaires à la réalisation du projet.
    - Exemple : Une entreprise tierce qui fournit une API ou un service cloud utilisé par l'application.
- **Régulateurs ou autorités** :
    - Imposent des normes légales, éthiques ou de conformité.
    - Exemple : Dans le cadre d'une application bancaire, une autorité financière peut exiger des règles de sécurité spécifiques.

8.3 Parties prenantes secondaires

Leur implication est indirecte, mais ils peuvent influencer le projet.

- **Équipes marketing et ventes** :
    - Elles peuvent influencer le design du produit en fonction des retours clients ou des besoins du marché.
- **Support technique** :
    - Prend en charge les questions ou problèmes des utilisateurs une fois le produit lancé.
- **Communauté ou utilisateurs potentiels** :

- Donnent des retours ou créent une demande pour le produit, influençant ainsi sa direction.

- **Actionnaires** :
  - Recherchent des bénéfices financiers ou un impact positif de l'entreprise via le projet.

8.4 Rôle des parties prenantes dans Scrum

Dans la méthode Scrum, les **parties prenantes clés** sont principalement impliquées lors des événements suivants :

- **Sprint Review** : Présentation de l'incrément terminé, avec recueil de feedback des parties prenantes.
- **Interactions avec le Product Owner** : Le Product Owner agit comme point de contact entre l'équipe Scrum et les autres parties prenantes pour s'assurer que les priorités du projet sont alignées sur leurs besoins.

En résumé, les parties prenantes incluent tous ceux qui sont intéressés par le projet, directement ou indirectement, et leur rôle peut varier selon leur lien avec l'équipe ou leur impact sur le produit.

8.5 Parties prenantes – un exemple concret

Quelles que soient sa taille et sa forme juridique – une entreprise individuelle ou une grande entreprise –, toute entreprise a des parties prenantes. Prenons un exemple concret : votre entreprise fabrique des produits en céramique. Ceci entraîne nécessairement l'existence de différentes parties prenantes, qui ont des attentes différentes vis-à-vis de votre entreprise.

Parties prenantes	Attentes/Objectifs	Influence
Propriétaire	Réaliser un chiffre d'affaires et des bénéfices aussi élevés que possible.	Très grande : en tant que propriétaire, vous contrôlez les processus et avez le pouvoir de décision.
Collaborateurs	Avoir un emploi durable et bien rémunéré.	Moyenne/Élevée : votre entreprise ne peut pas fonctionner sans collaborateurs formés et qualifiés.
État	Une bonne qualité à un bon prix.	Forte : vos clients sont les parties prenantes les plus importantes dans la mesure où ils achètent vos produits et garantissent par conséquent votre durabilité.
Clients	Une bonne qualité à un bon prix.	Forte : vos clients sont les parties prenantes les plus importantes dans la mesure où ils achètent vos produits et garantissent par conséquent votre durabilité.
Fournisseurs	Plus votre chiffre d'affaires est élevé, plus la quantité commandée est importante, plus le chiffre d'affaires du fournisseur est également élevé.	Relativement importante : selon le nombre de concurrents, l'impact sur votre compétitivité peut être élevé.

9. Conclusion

En conclusion, les méthodes agiles représentent un changement radical par rapport aux approches traditionnelles de gestion de projet, en mettant l'accent sur la flexibilité, la collaboration, et la réactivité. Elles permettent aux équipes de s'adapter rapidement aux changements, de livrer des produits de manière incrémentielle et de mieux répondre aux attentes des clients.

Les principales forces des méthodes agiles résident dans leur capacité à favoriser l'amélioration continue et à maintenir une communication constante entre les équipes et les parties prenantes, ce qui permet de mieux gérer les incertitudes et d'optimiser la qualité des produits.

Cependant, la réussite de l'agilité dépend largement de l'engagement de l'équipe, de la bonne compréhension des rôles, et de la capacité à ajuster les pratiques aux spécificités du projet. Les méthodes agiles ne sont pas une solution universelle : elles sont particulièrement adaptées à des environnements dynamiques, mais peuvent se heurter à des défis dans des contextes plus rigides ou où les processus sont déjà très établis.

Au final, adopter une approche agile demande une culture de transparence, de collaboration et une volonté d'amélioration continue. Lorsque ces principes sont bien intégrés, l'agilité peut transformer profondément la manière dont les projets sont menés et aboutir à des résultats plus efficaces et satisfaisants pour toutes les parties impliquées.

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TP N° 0 : Application :
Génération automatique de la documentation Java

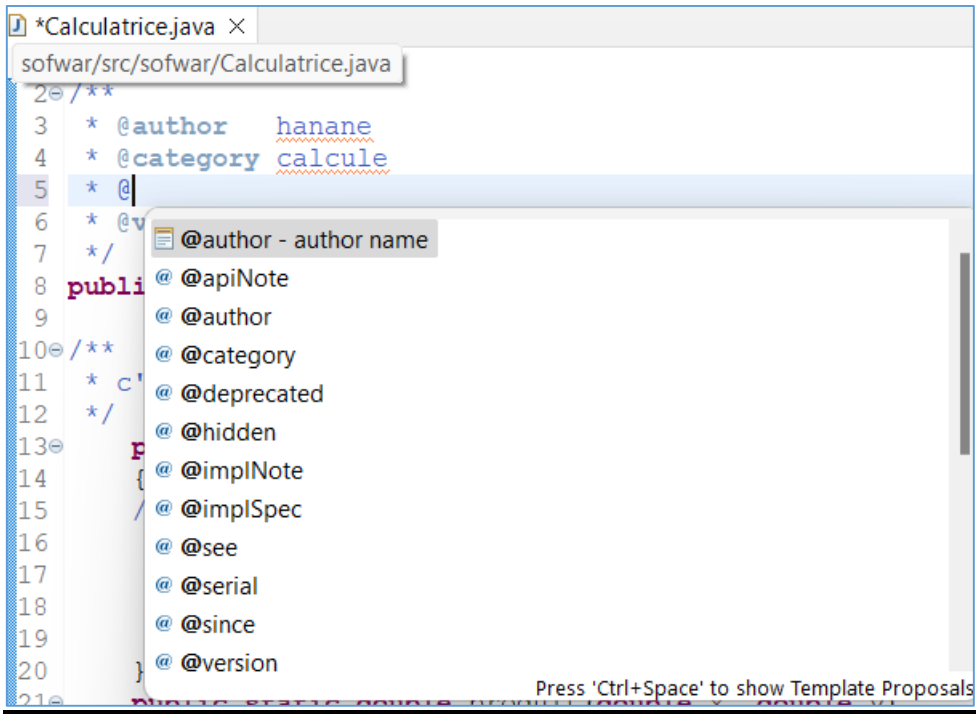
Objectif : Générer automatiquement d'un code java en utilisant la commande Java.doc fournît par le jdk

Etape 1: création d'un projet Java par la suite créer une classe

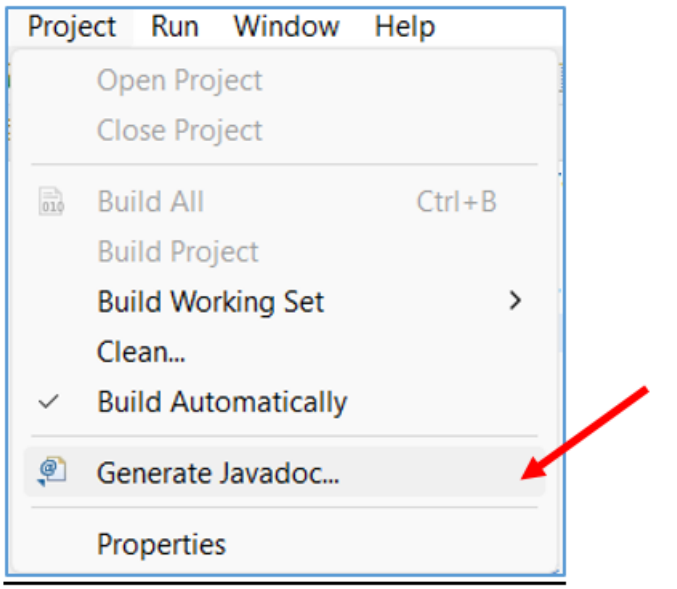
Calculatrice.java

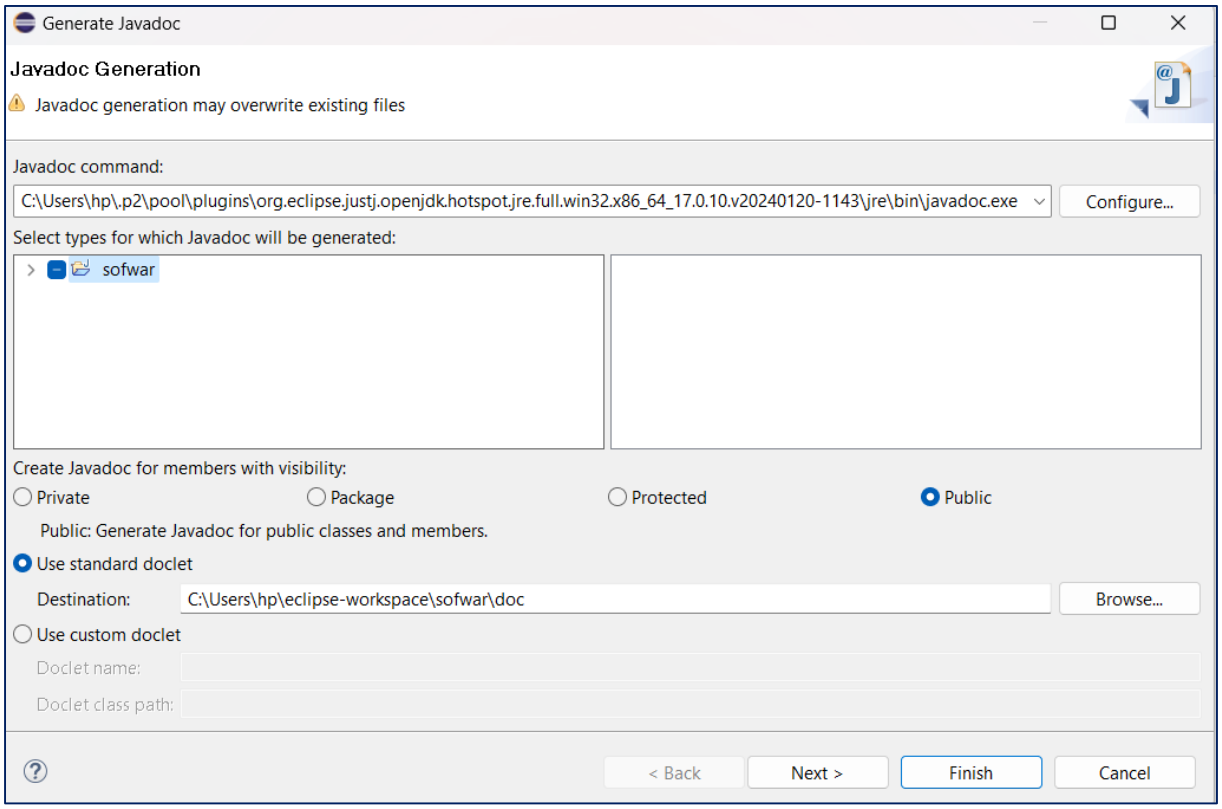
```java
package sofwar;
/**
 * @author   hanane
 * @category calcule
 * @version  1.0
 */
public class Calculatrice {

/**
 * c'est une classe permettent d'effectuer trois operations arithmetique
 */
    public static double summ(double x, double y)
    {return x+y;
    /**
     * @param x est un double
     * @param y est un double
     * @returne est la somme (+) entre x et y
     */
    }
    public static double produit(double x, double y)
    {return x*y;
    /**
     * @param x est un double
     * @param y est un double
     * @returne est le produit (*) entre x et y
     */
    }
    public static double difference(double x, double y)
    {return x-y;
    /**
     * @param x est un double
     * @param y est un double
     * @returne est la soustraction (-) entre x et y
     */
    }
}
```

Etape 2: ajouter les commentaires, pour ce faire taper `/**` , puis ajouter d'autre propriétés par `@`



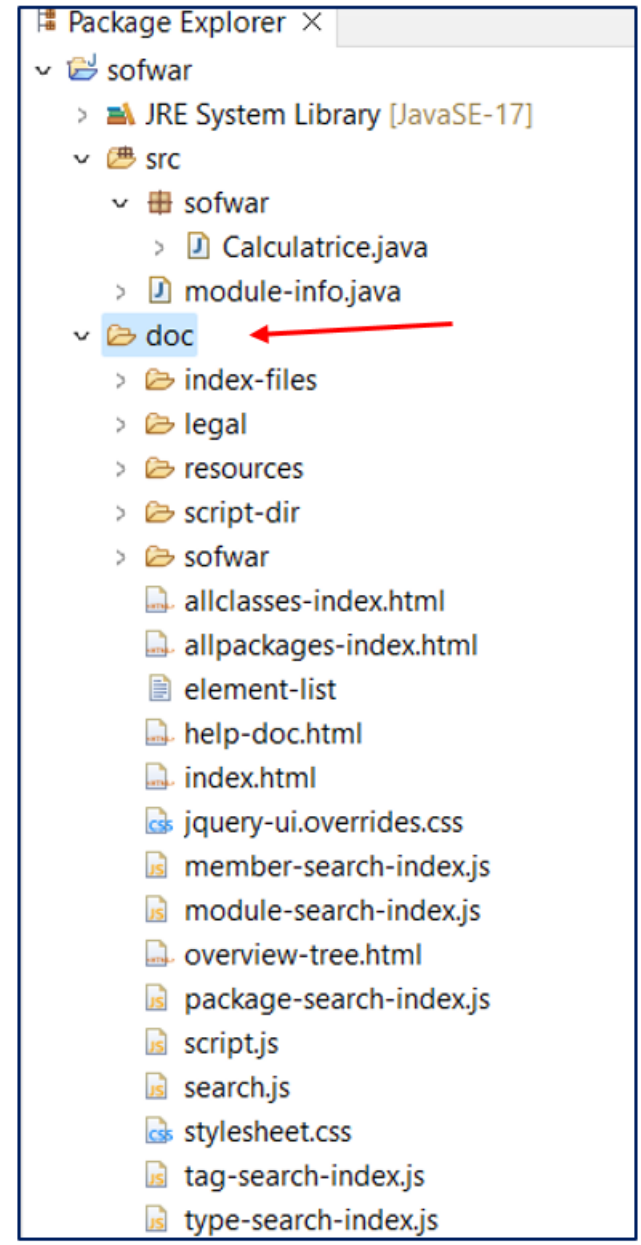
Etape 3: aller au menu project

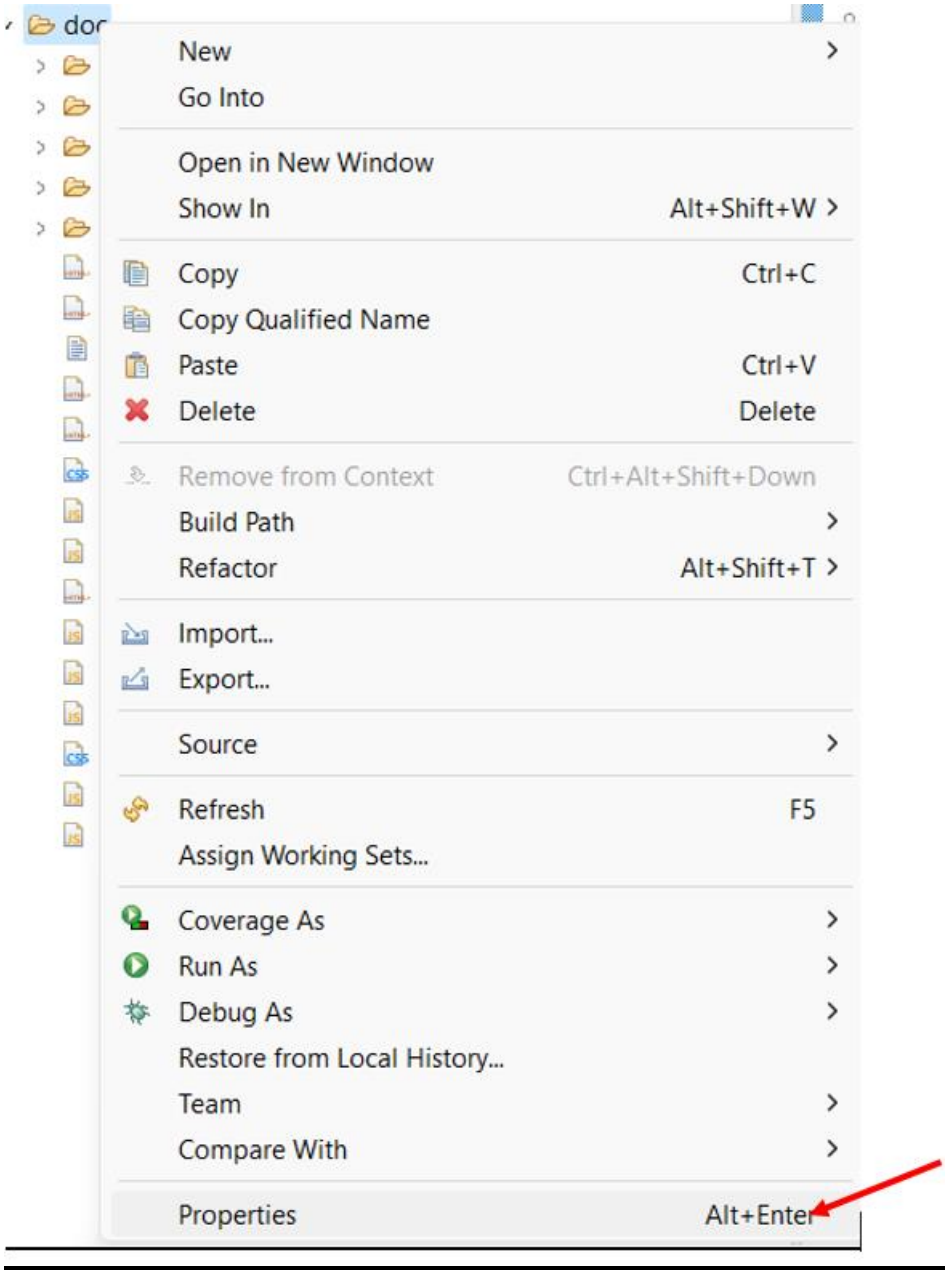


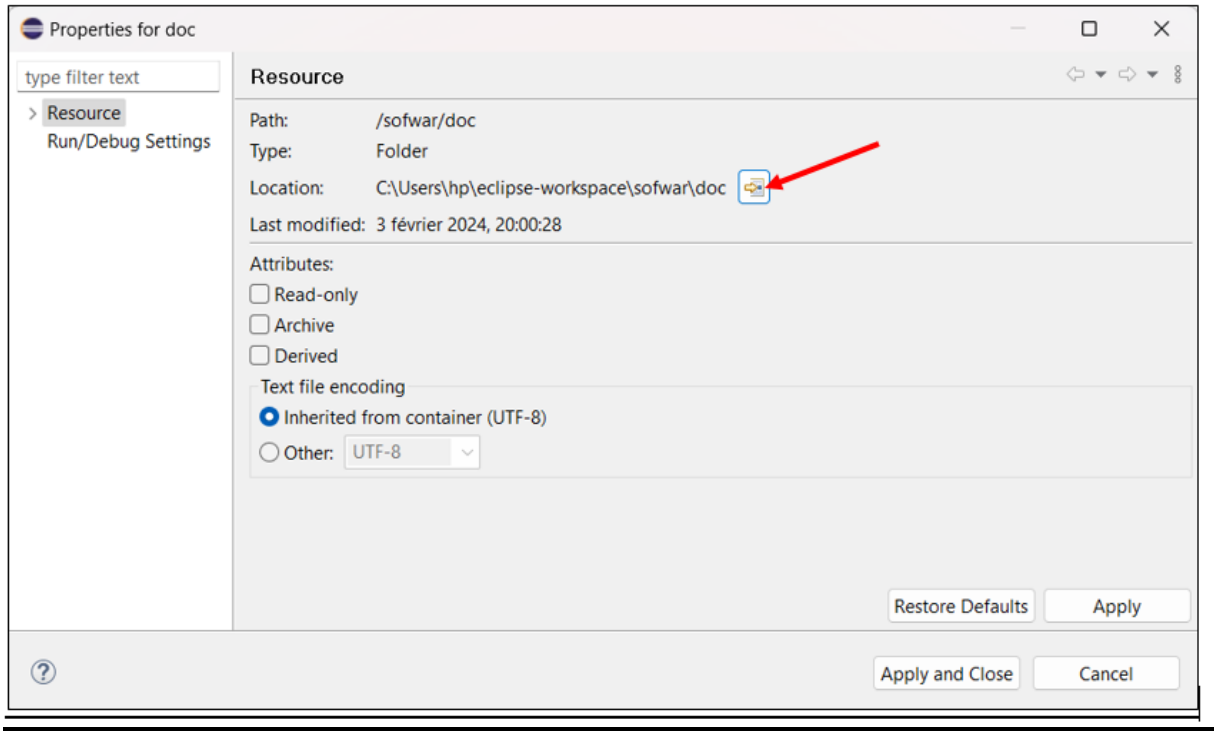


Etape 4: après une compilation

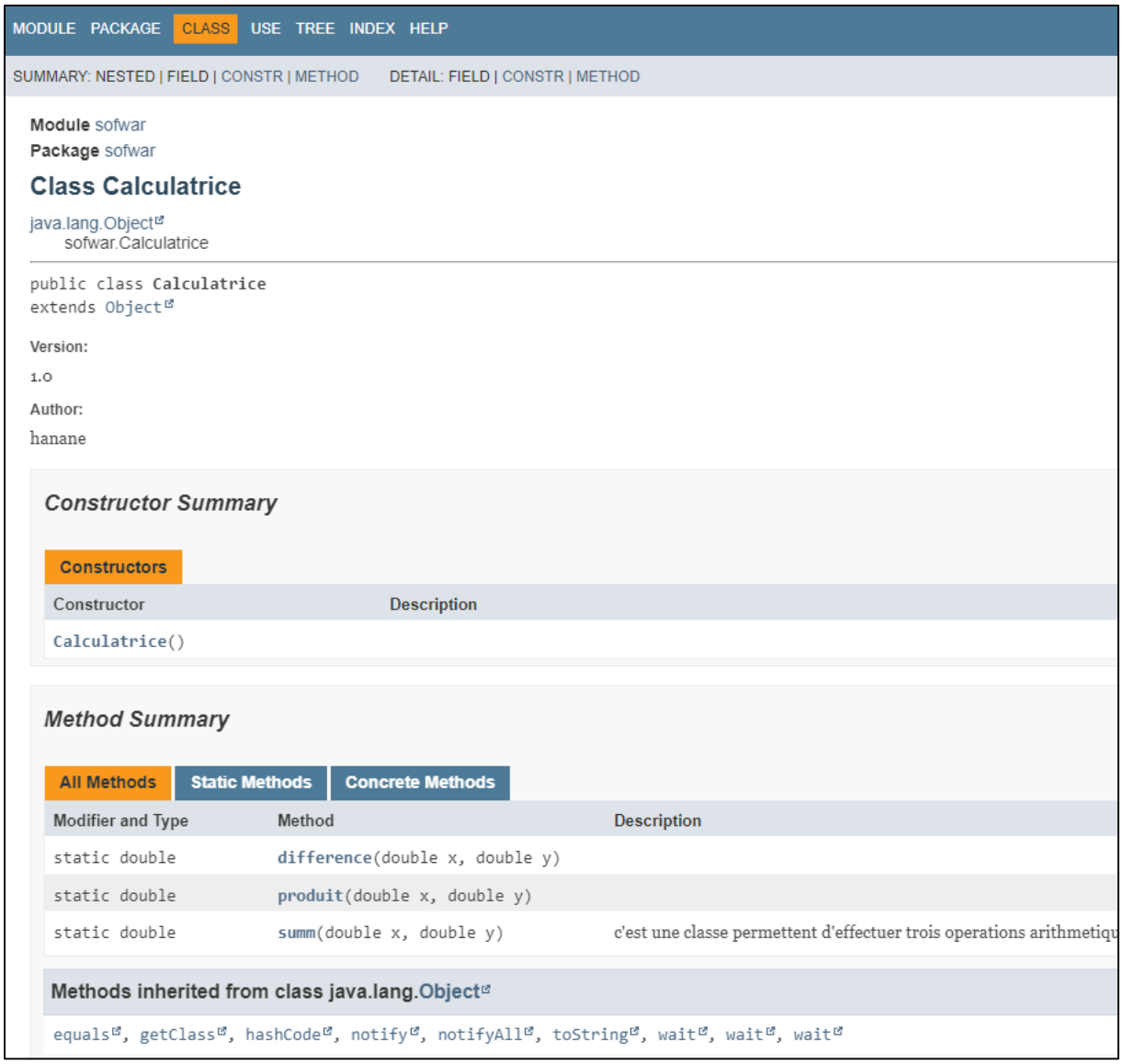
Etape 5: un nouveau dossier s'apparaitre


Package Explorer
sofwar
JRE System Library [JavaSE-17]
src
sofwar
Calculatrice.java
module-info.java
doc
index-files
legal
resources
script-dir
sofwar
allclasses-index.html
allpackages-index.html
element-list
help-doc.html
index.html
jquery-ui.overrides.css
member-search-index.js
module-search-index.js
overview-tree.html
package-search-index.js
script.js
search.js
stylesheet.css
tag-search-index.js
type-search-index.js


New
Go Into
Open in New Window
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Refactor Alt+Shift+T
Import...
Export...
Source
Refresh F5
Assign Working Sets...
Coverage As
Run As
Debug As
Restore from Local History...
Team
Compare With
Properties Alt+Enter



Etape 6: Génération automatique de documentation



MODULE PACKAGE CLASS USE TREE INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module sofwar
Package sofwar

Class Calculatrice

java.lang.Object
sofwar.Calculatrice

```
public class Calculatrice
extends Object
```

Version:

1.0

Author:

hanane

Constructor Summary

Constructors

Constructor	Description
Calculatrice()	

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method	Description
static double	difference(double x, double y)	
static double	produit(double x, double y)	
static double	summ(double x, double y)	c'est une classe permettent d'effectuer trois operations arithmetiqu

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Details

Calculatrice

```
public Calculatrice()
```

Method Details

summ

```
public static double summ(double x,
                          double y)
```

c'est une classe permettent d'effectuer trois operations arithmetique

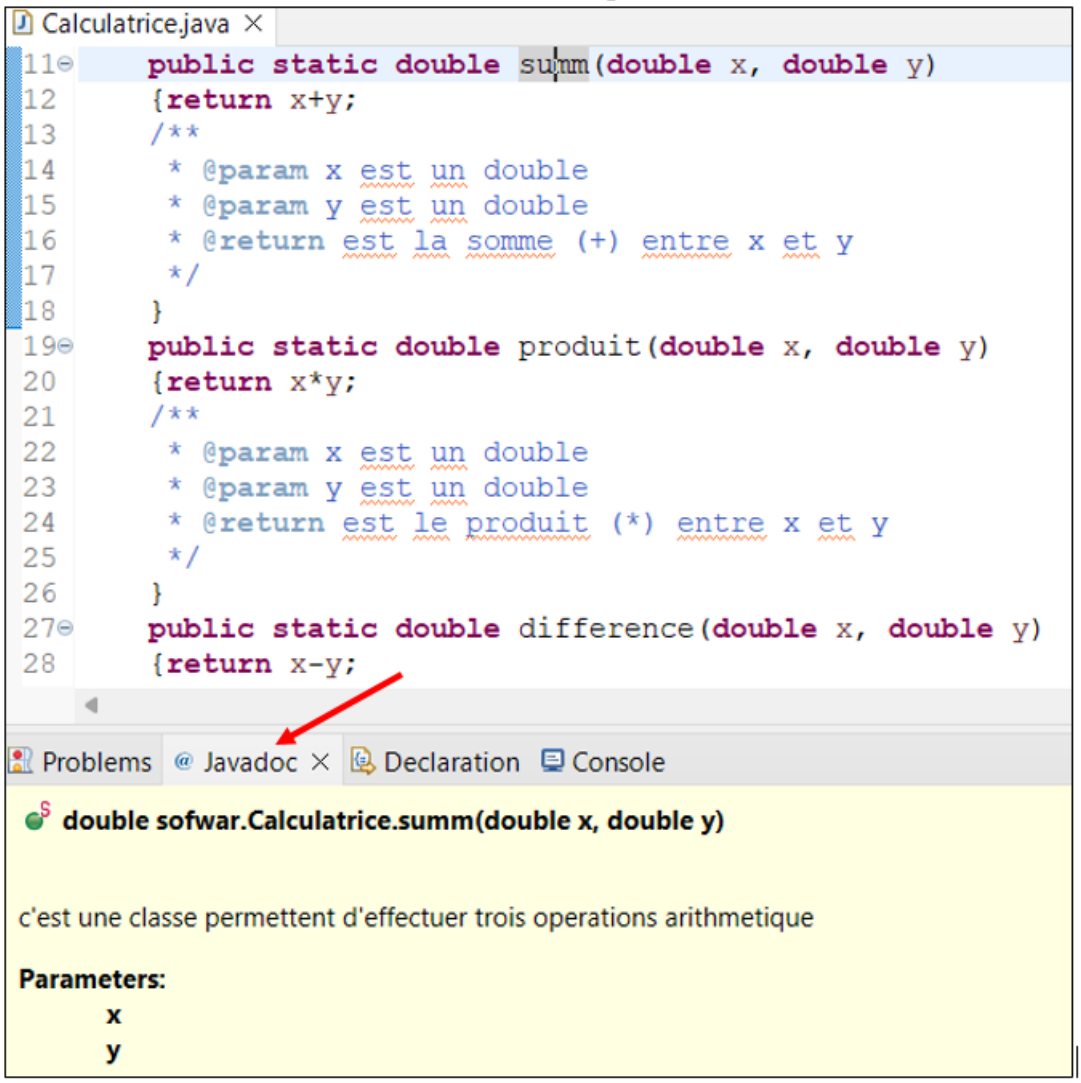
produit

```
public static double produit(double x,
                             double y)
```

difference

```
public static double difference(double x,
                                double y)
```

Etape 7: voir aussi la console javadoc



Application :

- Appliquez ces étapes en proposant un code de votre choix.
- Rapporter votre travail (document pdf)

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TD N°1 : Les méthodes agiles (exemples applicatifs)

Objectif : Donner le maximum d'exemples explicatifs sur les méthodes agiles

La méthode **Agile Scrum** est un cadre de travail collaboratif conçu pour développer des produits complexes de manière itérative et incrémentale. Voici un exemple détaillé de son application, avec un focus sur le **sprint** et le **backlog**.

Exemple complet qui adopte la méthode agile Scrum

Supposons que l'équipe développe une application mobile de gestion de tâches (comme une "to-do list").

1. Product Backlog

Le **Product Backlog** est une liste priorisée de tout ce qui doit être développé pour le produit. Les éléments du backlog sont définis par le **Product Owner** en collaboration avec les parties prenantes.

Exemple d'éléments du Product Backlog pour notre application :

1. Ajouter des tâches (avec titre et description).
2. Marquer une tâche comme terminée.
3. Trier les tâches par priorité.
4. Notifications pour les échéances.
5. Synchronisation entre plusieurs appareils.
6. Mode hors-ligne.

Chaque élément est appelé un **Product Backlog Item (PBI)**. Les PBIs sont souvent accompagnés de critères d'acceptation pour définir leur "fait" (ou **Definition of Done**).

2. Sprint Planning

Un **sprint** est un cycle court et fixe (souvent de 2 à 4 semaines) au cours duquel l'équipe développe un incrément fonctionnel du produit.

Objectif du Sprint Planning :

- Sélectionner les éléments du Product Backlog à développer.
- Planifier leur réalisation en fonction de la **capacité de l'équipe**.

Exemple :

- Durée du sprint : 2 semaines.
- Éléments choisis :
    1. Ajouter des tâches.

2. Marquer une tâche comme terminée.
3. Trier les tâches par priorité.

- L'équipe établit un **Sprint Goal** : "Permettre à l'utilisateur de créer, terminer et organiser ses tâches."

3. Sprint Backlog

Le **Sprint Backlog** est dérivé du Product Backlog. Il contient les PBIs sélectionnés pour le sprint, décomposés en tâches plus petites et plus spécifiques.

Exemple de Sprint Backlog : Pour l'élément "Ajouter des tâches" :

- Créer l'interface utilisateur pour ajouter une tâche.
- Implémenter la sauvegarde des tâches dans la base de données.
- Tester l'ajout de tâches avec différentes données.

4. Exécution du Sprint

Pendant le sprint, l'équipe se concentre sur les éléments du Sprint Backlog. Voici les activités typiques :

- **Daily Scrum** (réunion quotidienne de 15 minutes) : Chaque membre répond à trois questions :
  1. Qu'ai-je fait hier ?
  2. Que vais-je faire aujourd'hui ?
  3. Y a-t-il des obstacles ?
- **Collaboration constante** : Les membres de l'équipe travaillent ensemble pour compléter les tâches.

5. Sprint Review

À la fin du sprint, l'équipe présente l'incrément du produit à toutes les parties prenantes lors de la **Sprint Review**. L'objectif est de montrer le travail réalisé et de recueillir des retours.

Exemple d'incrément :

- Les utilisateurs peuvent ajouter des tâches, les marquer comme terminées et les trier par priorité.
- Une démonstration de l'application montre ces fonctionnalités.

6. Sprint Retrospective

Après la Sprint Review, l'équipe se réunit pour réfléchir à ce qui a bien fonctionné et ce qui peut être amélioré.

Exemple de points abordés :

- Ce qui a bien fonctionné : Bonne collaboration entre les développeurs et les testeurs.
- Ce qui peut être amélioré : Améliorer les estimations pour éviter de prendre trop de tâches.

Répétition des cycles

Une fois le sprint terminé, un nouveau sprint commence. Le **Product Backlog** est mis à jour en fonction des retours et des priorités.

Résumé visuel

1. **Product Backlog** : Liste des fonctionnalités.
2. **Sprint Planning** : Choix des éléments à réaliser dans un sprint.
3. **Sprint Backlog** : Tâches spécifiques pour atteindre l'objectif.
4. **Exécution du sprint** : Travail sur les tâches avec des réunions quotidiennes.
5. **Sprint Review** : Démonstration des résultats.
6. **Sprint Retrospective** : Réflexion sur les améliorations.

Ce cycle continu permet de livrer des incréments de produit fonctionnels, adaptés aux besoins des utilisateurs, tout en s'améliorant constamment.

Exemple complet qui adopte la méthode agile XP

La méthode Agile **Extreme Programming (XP)** est un cadre axé sur le développement logiciel itératif et collaboratif, mettant l'accent sur la qualité du code et l'adaptabilité aux changements. Elle repose sur des pratiques techniques rigoureuses et des principes de collaboration.
Voici un exemple détaillé pour comprendre comment XP fonctionne dans un projet.

Contexte de l'exemple

Un projet d'équipe consiste à développer une plateforme de commerce en ligne avec des fonctionnalités de base, comme la navigation par catégories, le panier d'achat et le paiement en ligne.

1. Principes clés d'XP

- **Communication** : Collaboration constante entre les développeurs, le client et les autres parties prenantes.
- **Simplicité** : Concevoir des solutions simples pour répondre aux besoins actuels.
- **Feedback rapide** : Les retours sont fréquents grâce aux tests et à des cycles de développement courts.
- **Courage** : Faire face aux changements ou aux erreurs sans hésitation.
- **Respect** : Valoriser chaque membre de l'équipe.

2. Cycle de vie d'un projet XP

1. Phase d'exploration

Le client exprime ses besoins sous forme de **stories utilisateurs**. Ces stories sont des descriptions simples des fonctionnalités demandées, rédigées de manière compréhensible.

Exemple de stories utilisateur :

- En tant qu'utilisateur, je veux ajouter un produit à mon panier pour l'acheter plus tard.
- En tant qu'utilisateur, je veux trier les produits par prix pour comparer plus facilement.
- En tant qu'utilisateur, je veux recevoir un email de confirmation après avoir payé.

Ces stories sont regroupées dans un **Backlog**.

2. Phase de planification

L'équipe organise des itérations courtes (généralement 1 à 2 semaines). Pendant une itération :

- L'équipe sélectionne les stories prioritaires en fonction des besoins du client.
- Les développeurs estiment chaque story en points (effort nécessaire).

Exemple de priorisation :

- Priorité haute : Ajouter un produit au panier.
- Priorité moyenne : Trier les produits par prix.
- Priorité basse : Personnalisation du profil utilisateur.

3. Itérations

Chaque itération contient les pratiques clés d'XP pour produire un incrément du produit.

3. Pratiques fondamentales d'XP dans l'exemple

1. Programmation en binôme (Pair Programming)

- Deux développeurs travaillent ensemble sur le même ordinateur : un écrit le code (le "conducteur") pendant que l'autre le critique et réfléchit aux solutions (le "navigateur").
- Exemple : Un développeur code la fonction "Ajouter au panier" tandis que son binôme vérifie la logique et propose des améliorations.

2. Développement piloté par les tests (TDD - Test-Driven Development)

- Avant d'écrire du code, l'équipe rédige un test qui vérifie que la fonctionnalité fonctionne.
- Exemple pour "Ajouter au panier" :
    1. Écrire un test : "Lorsqu'un produit est ajouté au panier, la liste des articles dans le panier doit inclure ce produit."
    2. Exécuter le test (il échoue initialement).

3. Écrire le code pour passer le test.
4. Réexécuter le test (il réussit).

3. Intégration continue

- Chaque développeur intègre son code fréquemment dans un dépôt central, où des tests automatisés vérifient l'intégrité de l'application.
- Exemple : Après avoir terminé la fonctionnalité "Trier les produits par prix", le développeur pousse son code. Les tests vérifient automatiquement que cette modification n'a pas cassé d'autres parties du système.

4. Réfactoration

- Amélioration continue du code sans modifier son comportement.
- Exemple : Si la fonction "Ajouter au panier" fonctionne, mais que son code est complexe ou redondant, l'équipe le simplifie pour qu'il soit plus lisible et maintenable.

5. Petites livraisons

- À la fin de chaque itération, l'équipe livre un incrément utilisable du produit.
- Exemple : Après une première itération, les utilisateurs peuvent :
  - Parcourir les catégories.
  - Ajouter un produit au panier.

6. Proximité avec le client

- Un client ou un représentant est toujours disponible pour répondre aux questions de l'équipe et valider les fonctionnalités.
- Exemple : Lorsqu'une ambiguïté apparaît sur le tri des produits (par prix croissant ou décroissant ?), le client clarifie immédiatement.

4. Retour d’expérience (Rétrospective)

À la fin de chaque itération, l'équipe organise une réunion pour réfléchir sur ce qui s'est bien passé et ce qui peut être amélioré.

Exemple de points discutés :

- Succès : Les tests automatisés ont permis de détecter rapidement un bug.
- Amélioration : Accélérer la rédaction des tests pour ne pas retarder le développement.

5. Résultat final

Après plusieurs itérations, la plateforme de commerce électronique est fonctionnelle :

- Les utilisateurs peuvent rechercher des produits, les trier et les acheter.
- Les tests automatisés garantissent que chaque nouvelle fonctionnalité ajoutée ne casse pas celles déjà développées.

Résumé des pratiques clés de XP appliquées

1. **Collaboration étroite** avec le client.
2. **Stories utilisateurs** pour prioriser les besoins.
3. **Itérations courtes** pour des livraisons fréquentes.
4. **Programmation en binôme** pour améliorer la qualité du code.
5. **Développement piloté par les tests** pour assurer un produit fiable.
6. **Intégration continue** pour maintenir la stabilité du système.
7. **Réfactoration constante** pour garder le code propre.

La méthode XP est particulièrement adaptée aux environnements complexes et évolutifs où les exigences changent fréquemment, tout en assurant un code de haute qualité.

Série d'exercice appliqué sur les diverses méthodes agiles

Exercice 1 : Créer un backlog produit

Objectif : Comprendre la structuration et la priorisation des besoins dans un backlog produit.

Contexte :

Vous travaillez sur le développement d'une application mobile de gestion des tâches. L'application doit inclure les fonctionnalités suivantes :

1. Création de tâches.
2. Ajout d'échéances.
3. Assignation des tâches à des membres de l'équipe.
4. Visualisation des tâches sur un calendrier.
5. Notifications pour les tâches en retard.
6. Intégration avec Google Calendar.

Consignes :

1. **Identifiez les utilisateurs principaux** : Qui utilisera cette application ?
2. **Créez des user stories** pour les fonctionnalités listées ci-dessus. Chaque user story doit suivre le format :
   "En tant que [utilisateur], je veux [objectif] afin de [bénéfice attendu]."
3. **Priorisez** les user stories en fonction de leur valeur métier et de leur complexité.

4. **Estimez** la charge de travail pour chaque user story en utilisant une méthode agile (ex. : estimation en points ou en heures).

Exemple :

- **User story** : "En tant qu'utilisateur, je veux créer une tâche avec un titre et une description afin de structurer mon travail."
- **Priorité** : Haute
- **Estimation** : 3 points

Exercice 2 : Planification d'un sprint (Scrum)

Objectif : Expérimenter la planification d'un sprint pour une équipe agile.

Contexte :

Votre équipe de développement est composée de 4 personnes. Le sprint dure 2 semaines. L'équipe a une capacité moyenne de **40 points** par sprint.

Consignes :

1. Sélectionnez les user stories prioritaires du backlog produit (exercice 1).
2. Déterminez combien de user stories peuvent être incluses dans le sprint, en respectant la capacité de l'équipe.
3. Créez une planification Sprint Backlog avec les tâches associées à chaque user story.

Exemple :

- **Sprint Goal** : "Permettre aux utilisateurs de créer, assigner et visualiser les tâches sur un calendrier."
- **Sprint Backlog** :
    - User Story 1 : Création de tâches (8 points)
    - User Story 2 : Assignation des tâches (5 points)
    - User Story 3 : Visualisation sur un calendrier (12 points)

Exercice 3 : Simulation d'une réunion quotidienne (Daily Scrum)

Objectif : Apprendre à partager les informations clés lors d'une réunion quotidienne.

Consignes :

Imaginez que vous êtes à mi-parcours du sprint. Chaque membre de l'équipe doit répondre aux trois questions clés d'un Daily Scrum :

1. **Qu'avez-vous fait hier ?**
2. **Que ferez-vous aujourd'hui ?**
3. **Quels sont les obstacles rencontrés ?**

Exemple de réponse :

- **Développeur A** :
  - Hier : J'ai terminé l'API pour la création des tâches.
  - Aujourd'hui : Je vais intégrer l'API avec l'interface utilisateur.
  - Obstacles : Aucun.
- **Développeur B** :
  - Hier : J'ai travaillé sur la visualisation des tâches sur le calendrier.
  - Aujourd'hui : Je vais corriger un bug dans l'affichage des dates.
  - Obstacles : Besoin d'aide pour comprendre une partie du design.

Exercice 4 : Introduction au Kanban

Objectif : Apprendre à organiser un tableau Kanban pour visualiser le flux de travail.

Contexte :

Vous gérez un projet simple avec 5 tâches :

1. Concevoir la maquette de l'application.
2. Développer la fonctionnalité de création de tâches.
3. Intégrer les notifications.
4. Effectuer des tests unitaires.
5. Préparer la documentation utilisateur.

Consignes :

1. Créez un tableau Kanban avec les colonnes suivantes :
   - **À faire**
   - **En cours**
   - **Terminé**
2. Placez les tâches dans les colonnes appropriées.

3. Simulez le déplacement des tâches au fur et à mesure de leur avancement.

Exemple :

- **À faire** : Intégrer les notifications, Préparer la documentation.
- **En cours** : Concevoir la maquette.
- **Terminé** : Développer la fonctionnalité de création de tâches, Effectuer des tests unitaires.

Exercice 5 : Rétrospective de sprint

Objectif : Identifier les points forts et les axes d'amélioration après un sprint.

Contexte :

Votre équipe vient de terminer un sprint. Voici un résumé des résultats :

- 80% des user stories ont été complétées.
- Un bug critique a retardé la livraison d'une fonctionnalité.
- La communication au sein de l'équipe a été fluide.

Consignes :

1. Remplissez un tableau de rétrospective avec les colonnes suivantes :
    - **Ce qui a bien fonctionné**
    - **Ce qui n'a pas fonctionné**
    - **Ce que nous devrions améliorer**

Exemple :

- **Ce qui a bien fonctionné** :
    - Bonne communication dans l'équipe.
    - Livraison rapide de certaines fonctionnalités.
- **Ce qui n'a pas fonctionné** :
    - Mauvaise gestion des imprévus (bug critique).
- **Ce que nous devrions améliorer** :
    - Anticiper les risques techniques en effectuant une analyse plus approfondie.

Travail demandé

- Proposer un exemple complet et cohérent afin de le faire appliquer la méthode **scrum**.
- Rapporter votre travail (document pdf)

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TP N° 1 (mini projet): Application des méthodes agiles

Application : SCRUM

Utiliser ClickUp comme un puissant outil de gestion de projet agile qui offre une large gamme de fonctionnalités pour vous aider à planifier, suivre et gérer vos projets.

- Choisissez une étude de cas collaborative, argumentez votre choix
- Vous devez utiliser la méthode agile **scrum**
- Editez le **diagramme de Gantt** afin d'organiser votre tâche collaborative
- Spécifiez les principaux **Backlogs** et ses **sprints**
- Le mini projet est un travail en d'équipe composé de 4 étudiants (1 scrum master by week)

Travail demandé : reporting

- Remplissez le tableau pendant les 4 **sprints** (vous avez max 4 **sprints**)
- Préparez le **cahier charge** le plus pertinent
- Expliquez l'idée principale pour la phase **d'analyse**
- Editez les meilleurs diagrammes pour la phase de **conception**
- Générez la documentation pour la phase de **codage** et de **test**
- Affichez les captures d'écran importantes (les interfaces graphiques)
- Discutez votre opinion sur cette méthode dans la conclusion
- Rapportez votre travail

Weeks	Backlog	Sprint	Scrum master	Scrum team	Admin task	End user task
Week 1	Backlog 1	Sprint 1				
		Sprint 2				
		Sprint 3				
	Backlog 2					
	Backlog 3					
Week 2						
Week 3						
Week 4						

:

ClickUp est une plateforme de gestion du travail et de la productivité tout-en-un conçue pour aider les équipes et les individus à planifier, organiser et collaborer efficacement. Son objectif principal est d'offrir une solution centralisée pour gérer toutes les tâches, projets et flux de travail, en réduisant le besoin d'utiliser plusieurs outils distincts.

Principaux objectifs de ClickUp :

1. **Centralisation du travail** : Rassembler toutes les tâches, projets, documents, et communications dans un seul outil.
2. **Personnalisation** : Permettre une personnalisation complète des espaces de travail, des statuts, des champs personnalisés, et des flux de travail en fonction des besoins de chaque équipe ou projet.
3. **Collaboration en équipe** : Fournir des fonctionnalités comme les commentaires, les mentions, les tâches partagées, et les suivis en temps réel pour améliorer la communication et la collaboration.
4. **Suivi de la productivité** : Offrir des fonctionnalités avancées pour suivre les progrès, mesurer les performances, et analyser les données avec des tableaux de bord et des rapports détaillés.
5. **Automatisation** : Réduire les tâches répétitives grâce à des automatisations intégrées qui permettent d'économiser du temps.
6. **Intégrations** : Se connecter avec d'autres outils et applications comme Slack, Google Drive, Zapier, etc., pour une expérience plus fluide.

En résumé, **ClickUp** vise à améliorer l'organisation, l'efficacité et la transparence des processus de travail tout en s'adaptant aux besoins variés des entreprises ou des utilisateurs individuels.

Chapitre 2 :

Tests en boite blanche (Structurel)

Plan de chapitre

1. Introduction

Les tests en boîte blanche, également connus sous le nom de Clear Box, Glass Box ou Structural Testing, sont une approche de test logiciel essentielle et systématique qui se concentre sur l'évaluation du fonctionnement interne et de l'architecture d'une application logicielle ou d'un système. Cette méthodologie permet aux testeurs et aux développeurs d'examiner le code, les algorithmes, les structures de données et la conception du système depuis l'application dans diverses conditions de test. Les tests en boîte blanche sont principalement utilisés pour les phases de tests unitaires, de tests d'intégration et parfois de tests système du cycle de vie de développement logiciel.

2. Classification des tests

Il existe différentes façons de classifier les tests. Il est possible de les regrouper selon : leur mode d'exécution, leurs modalités, leurs méthodes, leurs niveaux.

a. Le mode d'exécution

- **Le test Manuel** Les tests sont exécutés par le testeur. Il saisit les données en entrée, vérifie les traitements et compare les résultats obtenus avec ceux attendus. Ces tests sont fastidieux et offrent une plus grande possibilité d'erreurs humaines. Ces tests sont très vite ingérables dans le cas d'applications de grandes tailles.
- **Le test Automatique :** L'automatisation des tests est «l'utilisation de **logiciels** pour exécuter ou supporter des activités de tests, par exemple : gestion des tests, conception des tests, exécution des tests ou vérification des résultats». JUnit par exemple dans le monde Java.

b. Les modalités de test

- **Statiques** : Les tests sont réalisés «par l'humain» (testeur), sans machine, par lecture du code dans le but de trouver des erreurs. Il peut s'agir : de l'inspection ou d'une revue de code; d'un travail de collaboration lors d'une réunion: Le programmeur, le concepteur, un programmeur expérimenté, un testeur expérimenté, un modérateur (régulateur)…=> Portent sur des documents (plutôt des programmes), sans exécuter le logiciel.

- **Avantages :** Contrôle systématique valable pour toute exécution, applicables à tout document.

- **Inconvénients** Ne portent pas forcément sur le code réel. - Ne sont pas en situation réelle (interaction, environnement). Vérifications sommaires, sauf pour les preuves. Ces preuves nécessitent des spécifications formelles et complètes**,** donc difficiles.
    - **Dynamiques** : On exécute le système de manière à tester l'ensemble des caractéristiques. Chaque résultat est comparé aux résultats attendus. => Nécessitent une exécution du logiciel, une parmi des multitudes d'autres possibles
- **Avantages : Vérification avec des conditions proches de la réalité. Plus à la portée du commun des programmeurs.**
- **Inconvénients : Il faut provoquer des expériences, donc écrire du code et construire des données d'essais. Un test qui réussit ne démontre pas qu'il n'y a pas d'erreurs. =>Les techniques statiques et dynamiques sont donc complémentaires.**

c. Les méthodes de test :

Lorsqu'un logiciel ou une application sont créés, il est vital de réaliser plusieurs types de tests pour s'assurer que le produit est fini, complet, sécurisé et efficace. Pour réaliser ces tests, plusieurs méthodes sont possibles : les tests en « boîte noire », en « boîte blanche » et en « boîte grise ». La méthodologie de test des logiciels est définie comme les différentes approches, stratégies et types de tests pour tester une application afin de s'assurer que l'application ressemble et fonctionne comme prévu et répond aux attentes des utilisateurs / clients.

À un large niveau, les méthodologies de test impliquent tous les différents types de tests fonctionnels et non fonctionnels pour valider l'application.

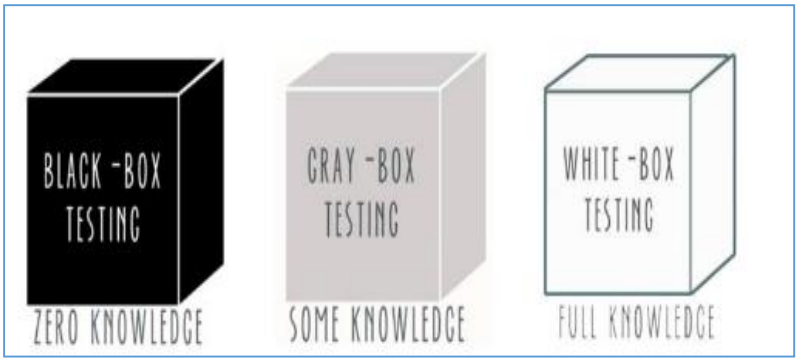


Figure 1 : les trois méthodes de test

Une analogie est souvent utilisée pour différencier ces techniques, en comparant le système testé à une voiture.

- En méthode « boîte noire », on vérifie que la voiture fonctionne en allumant les lumières, en klaxonnant et en tournant la clé pour que le moteur s'allume. Si tout se passe comme prévu, la voiture fonctionne.
- En méthode « boîte blanche », on emmène la voiture chez le garagiste, qui regarde le moteur ainsi que toutes les autres parties (mécaniques comme électriques) de la voiture. Si elle est en bon état, elle fonctionne.
- En méthode « boîte grise », on emmène la voiture chez le garagiste, et en tournant la clé dans la serrure, on vérifie que le moteur s'allume, et le garagiste observe en même temps le moteur pour s'assurer qu'il démarre bien selon le bon processus.

3. Test en boite blanche (white box, Glass box)

a. Objectifs

Un test en boîte blanche consiste généralement à inspecter les **chemins d'exécution** possibles par l'intermédiaire du **code** pour trouver les valeurs d'entrée qui forceraient l'exécution de ces chemins. Le testeur, en général le développeur, vérifie le code d'après sa conception.

b. Quand

Les tests en boîte blanche sont une forme de test d'application qui fournit au testeur une **connaissance complète de l'application testée**, y compris l'accès au **code** source et aux documents de conception.

Les tests en boîte blanche sont une technique de test de logiciels qui consiste à tester la **structure interne** et le **fonctionnement** d'une application logicielle. Le testeur **a accès au code source** et utilise ces connaissances pour **concevoir** des cas de test capables de vérifier **l'exactitude** du logiciel au niveau du **code**.

Les tests en boîte blanche peuvent commencer **tôt** dans le cycle de vie du développement logiciel, avant même le développement d'une interface graphique. Automatisation les tests en boîte blanche peuvent être facilement automatisés pour améliorer la couverture avec moins d'effort.

c. Raisons

- Détection précoce des erreurs logiques et de codage.
- Optimisation de la couverture du code source.
- Identification des chemins inaccessibles ou inutiles.
- Amélioration des performances et de la robustesse du logiciel.
- Il identifie en globale la sécurité internes.
- Pour vérifier les entrées à l'intérieur du code.
- Vérifiez la fonctionnalité des boucles conditionnelles.
- Pour tester la fonction, l'objet et l'instruction à un niveau individuel.

les techniques de test en boite blanche sont particulièrement utilisées dans le développement critique (ex. systèmes embarqués, médical, bancaire) où la fiabilité du code est primordiale.

d. Compréhension de la structure interne de code

Le codage de données consiste à représenter les données de manière à ce qu'elles puissent être stockées, traitées et transmises par un ordinateur. Il existe de nombreux formats de codage de données

La structure du code est la manière dont vous organisez et regroupez vos éléments de code, tels que des variables, des fonctions, des classes, des modules et des fichiers. Un code bien structuré est plus facile à comprendre, déboguer, tester, réutiliser et étendre. Cela vous aide également à éviter les erreurs, les bugs et les duplications.

Découvrez ces lignes spécifiques. Donc, pour mieux lire le code, reculez d'un pas à partir de là. Déterminez comment placer ces informations dans le fichier. Reculez encore d'un pas, puis déterminez d'où viennent les informations.

Chaque fois que vous êtes prêt à coder, assurez-vous d'avoir une liste des conventions de code que vous respectez. Cela contribue à rendre votre code lisible et maintenable. Se renseigner sur les modèles de conception et les mettre en œuvre peut vous aider à y parvenir. Gardez les classes et les fonctions petites et faites-leur faire une seule chose.

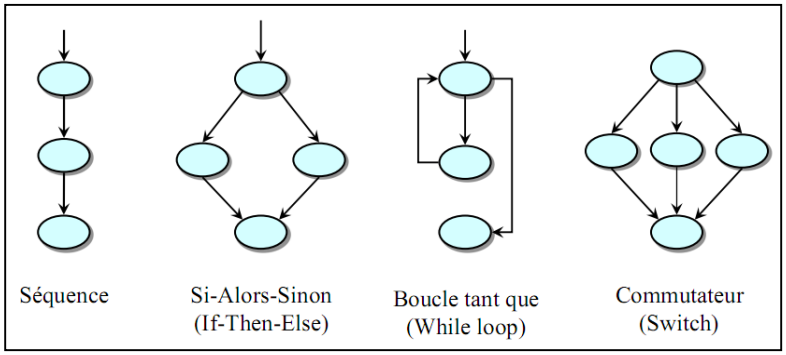
La structuration du code est une partie difficile, mais cruciale du codage. L'écriture d'un code bien structuré nécessite une réflexion appropriée, une compréhension des modèles de conception

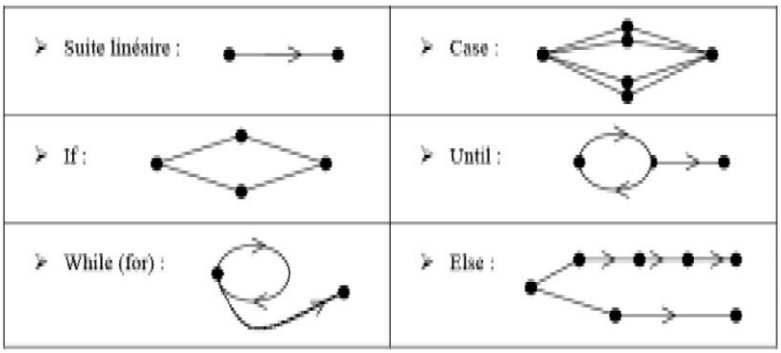
et de l'expérience. L'importance de la structuration du code ne doit pas être sous-estimée : la structuration du code est très importante du point de vue de la lisibilité et de la maintenabilité.

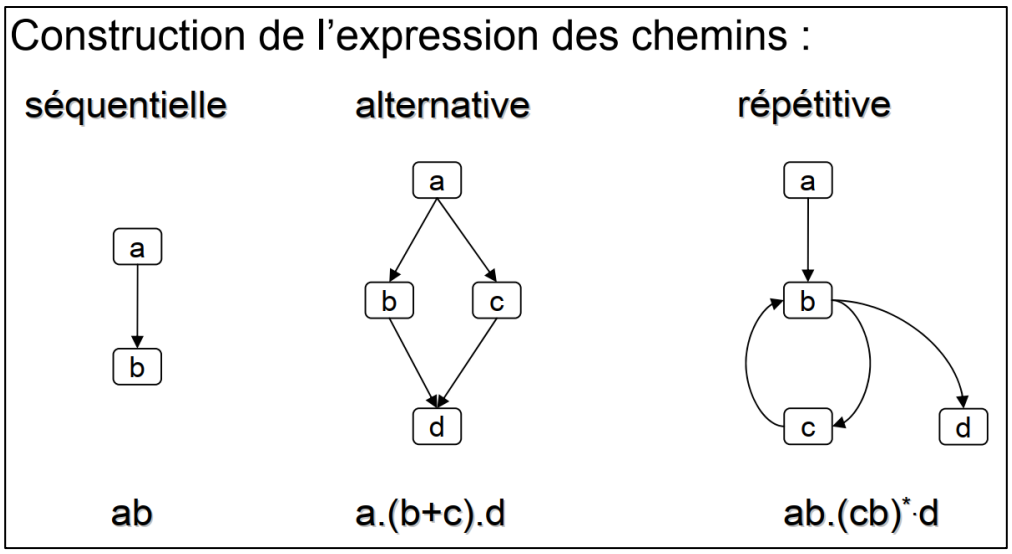
Les tests structurels reposent sur des analyses du code source. Il est possible d'analyser la structure du programme. Le principe est de : Accès au **code** pour déduire des tests à faire, de manière plus complète. Test de couverture : passage par tous les blocs d'instructions solidaires (GFC : Graphe de flot de contrôle. Techniques d'analyse dynamique, en effectuant l'instrumentation du code (logiciel de test).

e. Graphe de flot de contrôle

C'est un Graphe orienté. Les nœuds sont des blocs d'instructions séquentiels. Les arêtes sont des transferts de contrôle. Les arêtes peuvent être étiquetées avec un attribut représentant la condition du transfert de contrôle. Plusieurs conventions existent pour les modèles de graphes de flot avec des différences subtiles








```
read(x);
read(y);
while x != y loop
   if x>y then
        x := x - y;
   else
        y := y - x;
   end if;
end loop;
gcd := x;
```

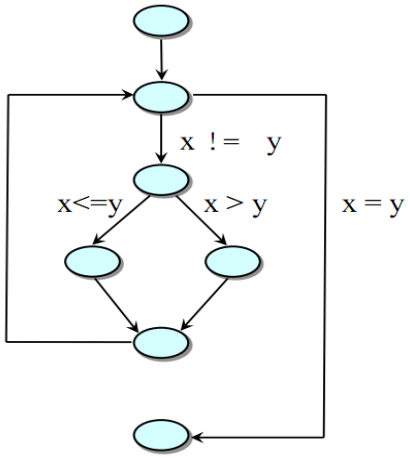


Figure 2 : graphe de flot de control

f. Mesure de couverture de code

Comment la couverture du code est-elle mesurée ? Il peut être calculé à l'aide de la formule: **Pourcentage de couverture du code = (Nombre de lignes de code exécutées)/(Nombre total de lignes de code dans une application) * 100**

a- Calculer la complexité cyclomatique V(G) : Dériver une mesure de la complexité logique. L'utiliser pour définir un ensemble de base des chemins d'exécution. D'où calcul de la complexité cyclomatique. **V(G)** fournit une borne supérieure des tests qui doivent être

exécutés pour garantir la couverture de toutes les instructions des programmes. Le nombre de composants **connectés** : => sous graphe

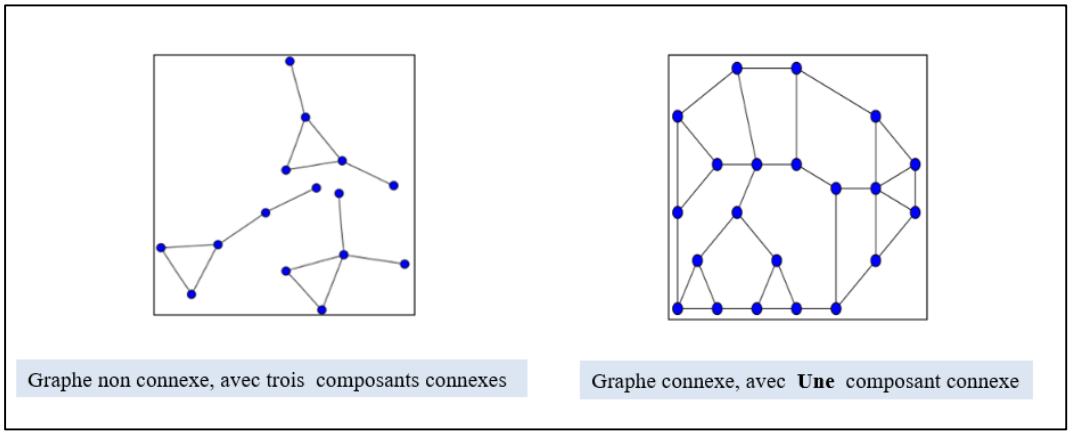


Figure 3 : composants connectés

Le nombre cyclomatique, la complexité cyclomatique ou la mesure de McCabe en 1976 est un outil de métrologie logicielle mesurer la complexité d'un programme informatique. Cette mesure reflète le nombre de décisions d'un algorithme en comptabilisant le nombre de « chemins » linéairement indépendants au travers d'un programme représenté sous la forme d'un graphe.

La complexité cyclomatique V(G) définit le nombre de chemins indépendants dans l'ensemble de base. L'ensemble de couverture des chemins est alors l'ensemble des chemins qui exécuteront toutes les instructions et toutes les conditions au moins une fois dans un programme. Les tests de couverture doivent être appliqués **aux modules critiques.** De nombreuses études industrielles ont montré que plus V(G) est grand, plus la probabilité d'erreur est importante.

$$V(G) = nb\_d'arcs - nb\_de\_nœuds + 2*nb\_de\_composants\_connectés$$

Exemple : le calcule de PGCD

V(G)= 10-9+2*1=3 (trois chemins)

Chemin 1= E, B1,P1, B4,S

Chemin 2= E,B1, P1,B2, P1, B4, S

Chemin 3= E,B1, P1,B3, P1, B4, S

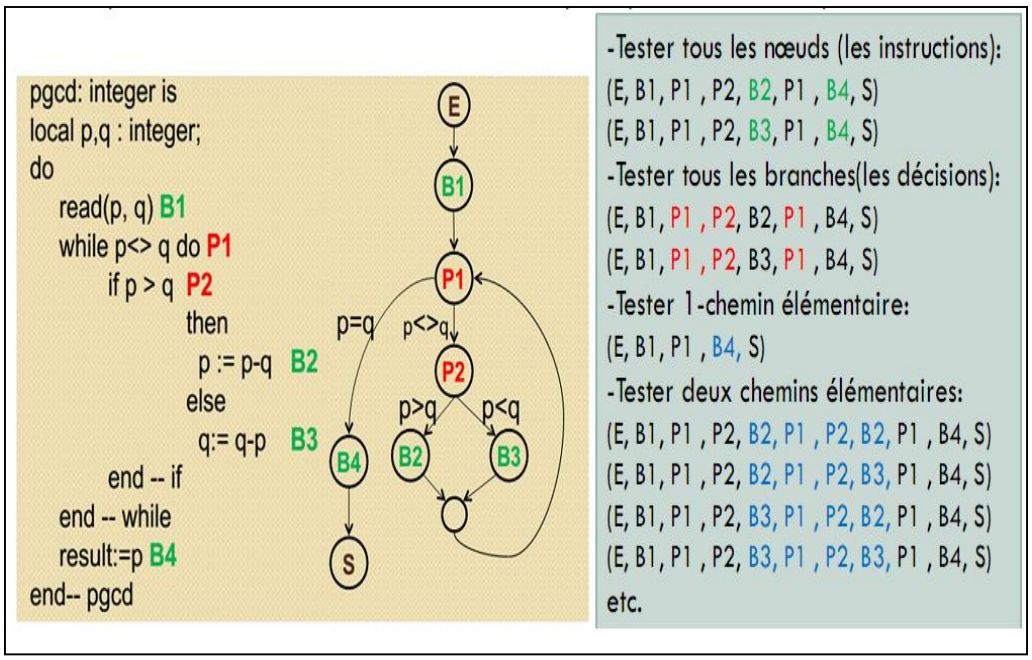


Figure 5 : GFC de PGCD

4. Techniques de test en boite blanche

4.1 Tests des chemins (Path Testing)

4.1.1 Principe

- Analyse et test de tous les chemins d'exécution possibles dans le programme.
- Objectif : S'assurer que chaque chemin est correctement exécuté au moins une fois.

Le **test des chemins** est une technique de test en boîte blanche qui vise à exécuter tous les chemins logiques possibles dans un programme pour s'assurer qu'ils fonctionnent correctement.

- Un programme est représenté sous forme de **graphe de contrôle de flux** (CFG – *Control Flow Graph*), où chaque bloc d'instructions est un **nœud** et chaque transition conditionnelle est un **arc**.
- On identifie tous les **chemins indépendants** à travers ce graphe.
- Chaque chemin doit être testé au moins une fois.

Exemple 1

Prenons un programme simple en Python qui vérifie si un nombre est positif, négatif ou nul :

```python
def verifier_nombre(n):
    if n > 0:
        print("Le nombre est positif")
    elif n < 0:
        print("Le nombre est négatif")
    else:
        print("Le nombre est nul")
```

4.1.2 Représentation en graphe de contrôle de flux (CFG)

1. Début
2. Vérification `n > 0`
    - **Oui** → Affiche « positif » → Fin
    - **Non** → Passe à `n < 0`
        - **Oui** → Affiche « négatif » → Fin
        - **Non** → Affiche « nul » → Fin

Chemins indépendants identifiés

1. `Début → n > 0 → Affichage « positif » → Fin`
2. `Début → n > 0 (faux) → n < 0 → Affichage « négatif » → Fin`
3. `Début → n > 0 (faux) → n < 0 (faux) → Affichage « nul » → Fin`

Cas de test nécessaires pour couvrir tous les chemins

Entrée (n)	Chemin suivi	Sortie attendue
`5`	Chemin 1	"Le nombre est positif"
`-3`	Chemin 2	"Le nombre est négatif"
`0`	Chemin 3	"Le nombre est nul"

Exemple 2 :

Prenons un programme qui vérifie si un utilisateur peut accéder à un système en fonction de son âge et de son statut de membre :

```
def verifier_acces(age, membre):
    if age >= 18:
        if membre:
            print("Accès autorisé")
        else:
            print("Accès refusé : adhésion requise")
    else:
        print("Accès refusé : âge insuffisant")
```

Étapes pour le Test des Chemins

1. **Représentation sous forme de Graphe de Flot de Contrôle (CFG)**
   - Chaque **bloc d'instruction** est un **nœud**.
   - Chaque **test conditionnel** (`if`) crée des **branches** vers des chemins alternatifs.

Graphe de Flot de Contrôle (CFG) :

Chemins indépendants à tester

1. `Début → age >= 18 (faux) → Affichage "Accès refusé : âge insuffisant" → Fin`
2. `Début → age >= 18 (vrai) → membre (faux) → Affichage "Accès refusé : adhésion requise" → Fin`
3. `Début → age >= 18 (vrai) → membre (vrai) → Affichage "Accès autorisé" → Fin`

Cas de Test Requis

Entrée (`age` , `membre`)	Chemin suivi	Sortie attendue
`16, False`	Chemin 1	"Accès refusé : âge insuffisant"
`20, False`	Chemin 2	"Accès refusé : adhésion requise"
`25, True`	Chemin 3	"Accès autorisé"

4.1.3 Avantages du test des chemins

- Assure que chaque condition et chaque branche est testée => **Couverture complète** de tous les scénarios possibles.

- Permet de détecter des **chemins inaccessibles** ou des **bugs logiques** =>**Détection des erreurs logiques** (ex. conditions mal gérées).
- Utile pour les **programmes complexes avec plusieurs conditions imbriquées**.
- **Optimisation du code** en repérant les chemins inutiles.

4.1.4 Désavantages du test des chemins

Bien que le test des chemins soit une technique puissante pour assurer une couverture complète du code, il présente plusieurs **limitations et désavantages** :

1. Explosion combinatoire des chemins

- Le nombre de chemins possibles dans un programme **augmente exponentiellement** avec la complexité du code.
- Pour un programme avec plusieurs boucles et conditions imbriquées, tester tous les chemins devient **impraticable**.

🔻 **Exemple** :

Un programme avec **3 conditions** (`if`) a **2^3 = 8 chemins possibles**, mais avec 10 conditions, on a **2^{10} = 1024 chemins** à tester !

2. Difficulté de mise en œuvre

- Le test des chemins **nécessite une analyse approfondie du code source** et la construction du **graphe de contrôle de flux**.
- Il **exige des compétences avancées** en programmation et en conception de tests.

🔻 **Problème** :

Si le code est mal structuré (ex. spaghetti code), l'analyse des chemins peut être **très compliquée**.

3. Non adapté aux programmes dynamiques

- Le test des chemins **ne prend pas en compte les entrées dynamiques ou imprévues** (ex. entrées utilisateur, interactions réseau).
- Il **ne détecte pas forcément** les erreurs liées à l'exécution en temps réel (ex. concurrence, ressources système).

🔻 **Exemple** :

Un programme qui récupère des **données depuis une API externe** peut générer des chemins différents selon les réponses reçues, rendant le test statique inefficace.

4. Néglige les erreurs logiques et fonctionnelles

- Il se concentre sur la couverture des chemins du **code source** mais **pas sur les exigences métier**.
- Il **ne garantit pas** que le programme **répond aux besoins de l'utilisateur final**.

🔻 **Exemple** :

Un **bug fonctionnel** peut être présent même si tous les chemins ont été testés correctement.

5. Limité par les boucles et récursions

- Les boucles (`for`, `while`) et la récursion peuvent **générer une infinité de chemins**.
- Il est **impossible de tester toutes les itérations** d'une boucle, surtout si elle dépend d'une entrée utilisateur.

🔻 **Exemple** :

Une boucle `while` qui s'exécute en fonction d'un capteur de température peut avoir un **nombre de répétitions inconnu** au moment du test

4.1.5 Conclusion

✅ **Le test des chemins est utile** pour assurer une couverture maximale du code et détecter des erreurs structurelles.

❌ **Mais il devient impraticable** pour les systèmes complexes et ne garantit pas que l'application répond aux attentes des utilisateurs.

🔽**Solution**

Compléter le test des chemins avec d'autres techniques (ex. tests en boîte noire, tests basés sur les exigences, tests fonctionnels).

4.2 Tests des conditions et décisions (Decision/Condition Testing)

4.2.1 Principe

- Vérification des instructions conditionnelles (`if`, `switch-case`, boucles).
- Objectif : Tester toutes les conditions et combinaisons logiques possibles.

Le **test des conditions et décisions** est une technique de test en boîte blanche qui vise à vérifier :

1. **Toutes les conditions booléennes** d'une expression (`if`, `while`, `for`).
2. **Toutes les décisions possibles** en testant les différentes combinaisons des conditions.

4.2.2 Démarche

- **Test des conditions** : Vérifier **chaque condition individuellement** en prenant ses valeurs **Vrai (True) et Faux (False)**.
- **Test des décisions** : Vérifier **toutes les branches possibles** (exécution des blocs `if`, `else`, etc.).
- **Objectif** : S'assurer que **toutes les conditions influencent bien le résultat final** et qu'aucun chemin logique n'est ignoré.

Exemple

Considérons un programme qui vérifie si un utilisateur peut accéder à un service en ligne :

```
def verifier_acces(age, compte_actif):
    if age >= 18 and compte_actif:
        print("Accès autorisé")
    else:
        print("Accès refusé")
```

Analyse des conditions et décisions

- **Conditions** :
    1. `age >= 18`
    2. `compte_actif`
- **Décision** :
    - `(age >= 18) AND (compte_actif) → "Accès autorisé"`
    - Autrement → "Accès refusé"

Cas de Test Requis (Table de Vérité)

`age >= 18`	`compte_actif`	Décision	Sortie attendue
Vrai	Vrai	Vrai	"Accès autorisé"
Vrai	Faux	Faux	"Accès refusé"
Faux	Vrai	Faux	"Accès refusé"
Faux	Faux	Faux	"Accès refusé"

🔹 **Couverture complète** : Toutes les combinaisons des **conditions** et **décisions** sont testées.

Exemple 2

Un système bancaire vérifie si un utilisateur peut effectuer un retrait en fonction de son solde et de la limite de retrait autorisée.

Code à tester

```
def verifier_retrait(solde, montant, limite_retrait):
    if solde >= montant and montant <= limite_retrait:
        print("Retrait autorisé")
    else:
        print("Retrait refusé")
```

Analyse des conditions et décisions

- **Conditions :**
  1. `solde >= montant` (Le solde est suffisant)
  2. `montant <= limite_retrait` (Le montant demandé est dans la limite autorisée)
- **Décision :**
  - `(solde >= montant) AND (montant <= limite_retrait)` → `"Retrait autorisé"`
  - Sinon → "Retrait refusé"

Table des Cas de Test (Conditions et Décision)

`solde >= montant`	`montant <= limite_retrait`	Décision finale	Résultat attendu
Vrai	Vrai	Vrai	"Retrait autorisé"
Vrai	Faux	Faux	"Retrait refusé"
Faux	Vrai	Faux	"Retrait refusé"
Faux	Faux	Faux	"Retrait refusé"

Exécution des tests avec valeurs concrètes

`solde`	`montant`	`limite_retrait`	Condition 1 (`solde >= montant`)	Condition 2 (`montant <= limite_retrait`)	Sortie attendue
500€	200€	300€	✅ Vrai	✅ Vrai	"Retrait autorisé"
500€	400€	300€	✅ Vrai	❌ Faux	"Retrait refusé"
100€	200€	300€	❌ Faux	✅ Vrai	"Retrait refusé"
100€	400€	300€	❌ Faux	❌ Faux	"Retrait refusé"

4.2.3 Avantages du Test des Conditions et Décisions

- **Détecte les erreurs logiques** dans les expressions conditionnelles.
- **Vérifie que chaque condition impacte bien le résultat final**.
- **Permet une couverture plus efficace que les tests simples de décisions**.
- Vérifie **chaque condition indépendamment** pour s'assurer qu'elle influence bien la décision.
- Assure **une couverture complète** des scénarios possibles.
- Permet de détecter **des erreurs logiques** dans les conditions.

4.2.4 Désavantages du Test des Conditions et Décisions

1. Explosion combinatoire

- **Problème** : Lorsque plusieurs conditions sont combinées dans une décision, le nombre de cas de test augmente **exponentiellement**.
- **Conséquence** : Difficile à gérer pour des systèmes complexes.

🔻 **Exemple** :

Une décision avec **4 conditions booléennes** (`if (A && B && C && D)`) entraîne **$2^4 = 16$ tests possibles** !

2. Ne détecte pas les erreurs de logique métier

- **Problème** : Cette approche se concentre sur les conditions du code mais **ne garantit pas** que l'application respecte les exigences métier.
- **Conséquence** : Un programme peut passer tous les tests et **ne pas fonctionner correctement** dans le contexte réel.

🔻 **Exemple** :

Un système bancaire peut valider les tests de retrait, mais **ne pas vérifier les frais bancaires**, provoquant des erreurs financières.

3. Inefficace pour les boucles complexes

- **Problème** : Cette approche **ne couvre pas toutes les exécutions possibles** des boucles (`for`, `while`).
- **Conséquence** : Certains cas d'exécution des boucles peuvent être **ignorés** (ex. boucle infinie, conditions d'arrêt incorrectes).

🔻 **Exemple** :

Un algorithme de recherche peut fonctionner dans les tests unitaires, mais **planter en présence d'une grande quantité de données**.

4. Difficulté à gérer des structures conditionnelles complexes

- **Problème** : Lorsque plusieurs conditions sont imbriquées (`if` dans `if`), il devient difficile d'assurer une **bonne couverture**.
- **Conséquence** : Risque de **tester certaines branches mais pas toutes**, laissant des bugs cachés.

🔻 **Exemple** :

Un code avec plusieurs niveaux d’imbrication :

```
if (A):
    if (B):
        if (C):
            faire_X()
```

Ici, le nombre de **scénarios à tester** devient rapidement **très grand**.

5. Dépendance aux entrées de test

- **Problème** : Les cas de test doivent être bien choisis pour **couvrir toutes les décisions possibles**.
- **Conséquence** : Une mauvaise sélection des entrées peut **laisser passer des bugs**.

🔻 **Exemple** :

Si on ne teste que des valeurs positives pour une fonction de calcul, on risque **d'ignorer les erreurs pour les valeurs négatives ou nulles**.

Conclusion

✅ **Le test des conditions et décisions est utile** pour détecter les erreurs logiques dans les instructions `if`.

❌ **Mais il devient inefficace** face à des systèmes complexes, des boucles et des exigences métier.

🔻 **Solution**

Compléter cette approche avec **d'autres techniques** (ex. tests de mutation, analyse des flux de données, tests fonctionnels).

4.3 Tests des flux de données (Data Flow Testing)

4.3.1 Principe

- Analyse des variables du programme (déclaration, affectation, utilisation).

- Objectif : Identifier les variables inutilisées ou mal initialisées.

Le **test des flux de données** est une technique de test en boîte blanche qui analyse la manière dont les **variables sont définies, utilisées et détruites** dans le code. Il permet de détecter des erreurs liées à l'utilisation incorrecte des variables, comme :

- L'utilisation d'une variable non initialisée.
- L'affectation d'une valeur à une variable sans jamais l'utiliser.
- L'utilisation d'une variable après sa destruction ou sortie de portée.

4.3.2 Démarche

On identifie trois types d'opérations sur les variables :

1. **Définition (D - Define)** : Une variable reçoit une valeur (x = 10).
2. **Utilisation (U - Use)** : La variable est utilisée (y = x + 2).
3. **Destruction (K - Kill)** : La variable est effacée ou sortie de portée.

Un test efficace consiste à s'assurer que :

- Toute variable utilisée a été correctement **définie** auparavant.
- Les valeurs affectées aux variables sont **utilisées** quelque part dans le programme.

Exemple

Prenons une fonction simple qui calcule la moyenne de deux nombres :

```
def calcul_moyenne(a, b):
    somme = a + b  # Définition (D) de `somme`
    moyenne = somme / 2  # Définition (D) de `moyenne`
    return moyenne  # Utilisation (U) de `moyenne`
```

Analyse des Flux de Données

Variable	Définition (D)	Utilisation (U)	Problème possible ?
`somme`	`somme = a + b`	`moyenne = somme / 2`	✅ Correct
`moyenne`	`moyenne = somme / 2`	`return moyenne`	✅ Correct

Ici, il **n'y a pas d'erreur**, car chaque variable est bien définie avant d'être utilisée.

Exemple avec une Erreur

```
def calcul_moyenne(a, b):
    if a > 0:
        somme = a + b  # Définition (D) de `somme`
    moyenne = somme / 2  # Utilisation (U) de `somme` (ERREUR !)
    return moyenne
```

Problème détecté par le test des flux de données :

- Si a <= 0, la variable somme **n'est jamais définie**, ce qui provoquera une **erreur d'exécution** lors de son utilisation (somme / 2).
- **Correction** : Initialiser somme avant le if.

Exemple 2 : **Exemple détaillé d'une limite du test des flux de données : L'explosion du nombre de cas de test**

Problème : Trop de cas à tester

Le test des flux de données exige que chaque variable soit suivie depuis sa **définition (D)** jusqu'à son **utilisation (U)** et éventuellement sa **destruction (K)**.

Si un programme comporte **de nombreuses variables et chemins d'exécution**, le nombre de cas de test devient rapidement ingérable.

Exemple de code : Gestion d'un compte bancaire

```python
def gestion_compte(solde, retrait, depot, mode):
    if mode == "RETRAIT":
        if retrait <= solde:
            solde = solde - retrait  # DÉFINITION (D) de solde
        else:
            return "Fonds insuffisants"
    elif mode == "DEPOT":
        solde = solde + depot  # DÉFINITION (D) de solde
    return solde  # UTILISATION (U) de solde
```

Analyse des flux de données

Variable	Définition (D)	Utilisation (U)	Problèmes possibles ?
`solde`	Modifié après un retrait ou un dépôt	Retourne la valeur finale	✅ Correct
`retrait`	Utilisé uniquement si `mode == "RETRAIT"`	Si non défini, peut causer une erreur	⚠️ Problème potentiel
`depot`	Utilisé uniquement si `mode == "DEPOT"`	Idem que `retrait`	⚠️ Problème potentiel

Nombre de cas de test à prévoir

Pour assurer une bonne couverture, il faudrait tester **toutes les combinaisons possibles** de `solde`, `retrait`, `depot` et `mode` :

1. **Cas normaux** :
   - `mode = "RETRAIT"`, `retrait` inférieur ou égal au `solde`
   - `mode = "DEPOT"`, `depot` ajouté au `solde`
2. **Cas limites** :
   - `retrait` égal au `solde` (solde devient 0)
   - `retrait` supérieur au `solde` (test de l'erreur)
   - `depot` = 0 (aucun changement)
3. **Cas inattendus** :

- `mode` différent de `"RETRAIT"` et `"DEPOT"` (aucun traitement)
- `retrait` ou `depot` non défini (plantage possible)
- `solde` négatif (mauvaise gestion des comptes)

Problème : Si chaque variable a **3 ou 4 valeurs différentes**, le nombre de tests explose très vite (exponentiellement).

4.3.3 Solution pour éviter cette explosion de tests

- **Prioriser les tests critiques** : Tester uniquement les chemins les plus risqués.
- **Utiliser des outils d'analyse statique** : Automatiser l'analyse des flux de données avec des logiciels spécialisés.
- **Compléter avec des tests fonctionnels** : Vérifier si le programme donne les bons résultats, en plus de s'assurer que les flux de données sont corrects.

4.3.4 Avantages du Test des Flux de Données

- **Détecte des erreurs difficiles à repérer**, comme l'utilisation de variables non initialisées.
- **Complète le test structurel** en s'assurant que les données circulent correctement dans le programme.
- **Améliore la qualité du code** en identifiant les variables inutiles ou mal gérées.

4.3.5 Désavantages du Test des Flux de Données

1. Complexité pour les grands programmes

- **Problème** : Plus le programme est long et complexe, plus il contient de variables et d'interactions entre elles.
- **Conséquence** : L'analyse des flux de données devient difficile et nécessite **des outils automatisés** pour tracer toutes les définitions et utilisations.

🔻 **Exemple** : Dans un système de gestion bancaire avec des centaines de variables, analyser manuellement chaque flux de données est presque impossible.

2. Ne détecte pas toutes les erreurs logiques

- **Problème** : Ce test vérifie comment les variables sont définies et utilisées, mais il **ne détecte pas** les erreurs de logique métier.
- **Conséquence** : Un programme peut bien gérer ses variables mais **donner des résultats erronés** s'il y a une erreur dans l'algorithme.

🔻 **Exemple** : Une fonction de calcul des impôts peut bien utiliser ses variables (`revenu`, `taux`), mais appliquer une mauvaise formule de calcul.

3. Inefficace sur les programmes à forte interaction avec l'extérieur

- **Problème** : Lorsque le programme dépend d'**entrées dynamiques** (API, base de données, capteurs IoT), les flux de données peuvent varier et ne pas être entièrement testés.
- **Conséquence** : Certains cas d'erreur ne seront détectés qu'en conditions réelles.

🔻 **Exemple** : Une application météo reçoit des données de capteurs en temps réel. Si un capteur envoie une valeur inattendue (None ou une valeur aberrante), le système peut planter malgré un bon test des flux de données.

4. Explosion du nombre de cas de test

- **Problème** : Pour garantir une couverture complète, il faudrait tester toutes les combinaisons possibles des définitions et utilisations des variables.
- **Conséquence** : Un grand nombre de cas de test est nécessaire, ce qui **prend du temps et des ressources**.

🔻 **Exemple** :

Un programme avec **10 variables utilisées dans différentes conditions** peut nécessiter **des centaines de tests** pour couvrir tous les scénarios possibles.

5. Nécessite des compétences avancées

- **Problème** : L'analyse des flux de données demande une **bonne compréhension du code source** et une expertise en test logiciel.
- **Conséquence** : Tous les testeurs ne sont pas formés à ce type d'analyse, et son implémentation peut être complexe.

🔻 **Exemple** : Dans une équipe de test, seuls les développeurs expérimentés pourront appliquer correctement cette méthode, ce qui limite son utilisation.

4.3.6 Conclusion

✓ **Le test des flux de données est très utile** pour détecter des erreurs liées à la gestion des variables et des définitions inutilisées.

✕ **Mais il devient difficile à appliquer** sur des systèmes complexes, interactifs ou avec un grand nombre de variables.

Solution : Compléter cette approche avec d'autres types de tests comme **les tests fonctionnels, les tests des chemins et les tests de mutation**.

4.4 Tests de couverture (Coverage Testing)

4.4.1 Principe

Les **tests de couverture** sont une technique d'évaluation qui mesure **dans quelle proportion un programme a été testé** en analysant les chemins d'exécution parcourus. L'objectif est d'identifier **les parties du code non testées** et de garantir une meilleure fiabilité du programme.

4.4.2 Types de couverture

Il existe plusieurs niveaux de couverture :

1. **Couverture des instructions** (Statement Coverage)
    - Vérifie que **chaque ligne de code** a été exécutée au moins une fois.

2. **Couverture des branches** (Branch Coverage)
    - Assure que **chaque branche d'une condition (if, else, switch) est testée**.
3. **Couverture des chemins** (Path Coverage)
    - Vérifie que **toutes les combinaisons de chemins possibles dans le code sont exécutées**.
4. **Couverture des conditions** (Condition Coverage)
    - Teste chaque **condition booléenne** d'une expression logique séparément.

Exemple de Tests de Couverture

Prenons le programme suivant en Python :

```python
def verifier_nombre(n):
    if n > 0:
        print("Nombre positif")
    else:
        print("Nombre négatif ou nul")
```

1. **Couverture des instructions (Statement Coverage)**

Objectif : Tester **chaque ligne de code au moins une fois**.

Cas de Test	Valeur de `n`	Instructions exécutées
Cas 1	`5`	✅ `if n > 0` → `print("Nombre positif")`
Cas 2	`-3`	✅ `else` → `print("Nombre négatif ou nul")`

100 % de couverture des instructions, car toutes les lignes sont exécutées.

2.Couverture des branches (Branch Coverage)

Objectif : Tester **chaque branche d'une condition (if et else)**.

Cas de Test	Valeur de `n`	Branche testée
Cas 1	`10`	✅ Branche `if`
Cas 2	`-2`	✅ Branche `else`

100 % de couverture des branches, car **toutes les conditions `if` et `else` ont été explorées**.

3. Couverture des chemins (Path Coverage)

Objectif : Tester **toutes les séquences d'exécution possibles**.

Ici, le programme a seulement **deux chemins possibles** :

1. `if` → `print("Nombre positif")`
2. `else` → `print("Nombre négatif ou nul")`

Comme nous avons déjà testé ces deux chemins, ✅ **100 % de couverture des chemins**.

4.Couverture des conditions (Condition Coverage)

Objectif : Vérifier que chaque **condition logique** (`n > 0`) est testée avec **vrai et faux**.

Cas de Test	Valeur de `n`	Évaluation de `n > 0`
Cas 1	`3`	✅ Vrai
Cas 2	`-1`	✅ Faux

100 % de couverture des conditions, car nous avons testé `n > 0` avec **vrai et faux**.

Exemple plus complexe des Tests de Couverture (Coverage Testing)

Prenons un programme plus avancé qui inclut plusieurs conditions et une boucle :

```python
def calculer_remise(montant, client_fidele):
    if montant > 100:
        if client_fidele:
            remise = montant * 0.20  # 20% de remise
        else:
            remise = montant * 0.10  # 10% de remise
    else:
        remise = 0  # Pas de remise

    return remise
```

1. Couverture des Instructions (Statement Coverage)

Objectif : Vérifier que **toutes les lignes de code** sont exécutées au moins une fois.

Cas de Test	`montant`	`client_fidele`	Lignes exécutées
Cas 1	`150`	`True`	✅ `if montant > 100`, ✅ `if client_fidele`, ✅ `remise = montant * 0.20`
Cas 2	`80`	`False`	✅ `else`, ✅ `remise = 0`

✅ **Toutes les lignes du programme sont exécutées au moins une fois → 100 % de couverture des instructions.**

2. Couverture des Branches (Branch Coverage)

Objectif : Vérifier que **chaque branche d'un `if` est testée**.

Cas de Test	`montant`	`client_fidele`	Branche testée
Cas 1	`150`	`True`	✅ `if montant > 100` → `if client_fidele` (20%)
Cas 2	`150`	`False`	✅ `if montant > 100` → `else` (10%)
Cas 3	`80`	`False`	✅ `else` (remise = 0)

✅ **100 % de couverture des branches**, car chaque condition (`if`, `else`) a été explorée.

3. **Couverture des Chemins (Path Coverage)**

Objectif : Vérifier que **toutes les séquences d'exécution possibles** sont testées.

Nous avons **3 chemins possibles** :

1. `if montant > 100` → `if client_fidele` → `remise = 20%`
2. `if montant > 100` → `else` → `remise = 10%`
3. `else` → `remise = 0`

Cas de Test	`montant`	`client_fidele`	Chemin suivi
Cas 1	`150`	`True`	✅ Chemin 1
Cas 2	`150`	`False`	✅ Chemin 2
Cas 3	`80`	`False`	✅ Chemin 3

✅ **100 % de couverture des chemins**, car chaque chemin a été exécuté au moins une fois.

4. **Couverture des Conditions (Condition Coverage)**

Objectif : Vérifier que **chaque condition logique est testée avec `Vrai` et `Faux`**.

Conditions dans le code :

- `montant > 100`
- `client_fidele`

Cas de Test	`montant`	`client_fidele`	`montant > 100`	`client_fidele`
Cas 1	`150`	`True`	✅ `Vrai`	✅ `Vrai`
Cas 2	`150`	`False`	✅ `Vrai`	✅ `Faux`
Cas 3	`80`	`False`	✅ `Faux`	✅ `Faux`

4.4.3 Avantages de tests de couverture (Coverage Testing)

1. **Identification des parties non testées**
    - Permet de repérer les parties du code qui **n'ont pas été exécutées**, réduisant ainsi le risque d'erreurs non détectées.
2. **Amélioration de la qualité du code**
    - Un haut niveau de couverture améliore la **robustesse** du logiciel et diminue les **risques de bugs** en production.
3. **Détection des erreurs logiques**
    - En s'assurant que toutes les conditions et branches sont couvertes, on réduit les risques d'oubli dans la logique du programme.
4. **Automatisable**
    - Des outils comme **JUnit (Java), pytest-cov (Python), Istanbul (JavaScript)** permettent d'automatiser l'analyse de couverture.
5. **Optimisation des tests**
    - Aide à prioriser les tests et à **éliminer les tests redondants** tout en garantissant une couverture maximale.

4.4.4 Limites de tests de couverture (Coverage Testing)

1. **Ne détecte pas tous les bugs**
    - **Une couverture de 100 % ne signifie pas qu'il n'y a pas de bug !** Elle garantit seulement que chaque partie du code a été exécutée, mais pas qu'elle fonctionne correctement.
2. **Difficile à atteindre à 100 %**
    - Pour des applications complexes, atteindre **100 % de couverture des chemins** peut être **quasiment impossible** à cause de la combinatoire exponentielle des conditions et des boucles.
3. **Ne remplace pas les tests fonctionnels**
    - La couverture **se base sur le code**, mais ne valide pas si le programme répond correctement aux **besoins des utilisateurs**.
4. **Coût et temps d'exécution élevés**
    - Plus le niveau de couverture recherché est élevé, plus **les tests sont nombreux**, ce qui peut être long et coûteux en ressources.

5. **Les tests ne garantissent pas la pertinence des données**
    - Même si un test couvre un bloc de code, il peut **ne pas refléter des scénarios réels**, rendant le test inefficace contre certains bugs.

4.4.5 Conclusion

✅ **Les tests de couverture sont un excellent outil pour améliorer la fiabilité du code** et détecter les zones non testées.

Mais ils ne suffisent pas : il faut les **compléter avec des tests fonctionnels et des tests de performance** pour garantir un logiciel fiable.

4.5 Tests des boucles (Loop Testing)

4.5.1 Principe

- Vérification des boucles (`for`, `while`, `do-while`).
- Objectif : Tester les cas de répétition, les bornes (min/max) et les cas limites.

Le **test des boucles** est une technique de **test en boîte blanche** qui se concentre sur la **vérification du comportement des boucles** dans un programme. Il permet de s'assurer que :

✓ Les boucles fonctionnent correctement avec **différents nombres d'itérations**.
✓ Les **cas limites** sont bien gérés (aucune itération, une seule itération, plusieurs itérations, maximum d'itérations).

✓ Il n'y a **pas de boucles infinies** ou de comportements inattendus.

Ce test est particulièrement utile pour les **boucles for, while et do-while** dans les langages de programmation.

4.5.2 Stratégies de Test

1. **Cas de la boucle non exécutée** (0 itération)
2. **Cas de la boucle exécutée une seule fois** (1 itération)

3. **Cas de la boucle exécutée un nombre moyen de fois** (N itérations normales)
4. **Cas de la boucle exécutée au maximum** (limite supérieure)
5. **Cas d'une boucle infinie** (mauvaise condition de sortie)

◆ Exemple 1 : Test d'une boucle `for`

Code original (calcul de la somme des N premiers nombres entiers)

```python
def somme_n_premiers(n):
    somme = 0
    for i in range(n):  # Boucle de 0 à n-1
        somme += i + 1
    return somme
```

Scénarios de test

Cas testé	Valeur de `n`	Résultat attendu
0 itération	`n = 0`	`0`
1 itération	`n = 1`	`1`
Valeur normale	`n = 5`	`1+2+3+4+5 = 15`
Limite supérieure	`n = 1000`	`500500`
Cas erroné	`n = -5`	À gérer avec une erreur ou un message d'alerte

Exemple 2 : Test d'une boucle while

```python
def compte_a_rebours(n):
    while n > 0:
        print(n)
        n -= 1
    print("Fin !")
```

Scénarios de test

Cas testé	Valeur de `n`	Résultat attendu
0 itération	`n = 0`	Affiche `Fin !` directement
1 itération	`n = 1`	Affiche `1`, puis `Fin !`
Valeur normale	`n = 5`	Affiche `5, 4, 3, 2, 1, Fin !`
Limite supérieure	`n = 1000`	Vérifier le temps d'exécution
Boucle infinie	`n = -5`	**Risque de boucle infinie si mal géré !**

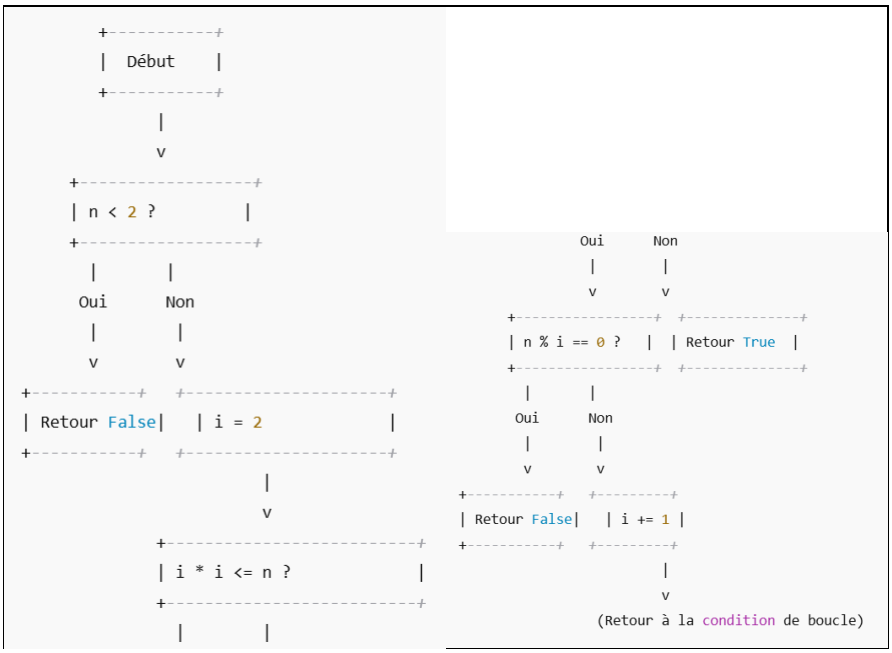
Voici un graphe de flot de contrôle **pour une fonction contenant une boucle `while`**

Exemple de Code

Prenons une fonction qui vérifie si un nombre est premier en utilisant une boucle `while` :

```python
def est_premier(n):
    if n < 2:
        return False  # Cas où n est inférieur à 2

    i = 2
    while i * i <= n:  # Boucle pour vérifier les diviseurs
        if n % i == 0:
            return False  # n n'est pas premier
        i += 1

    return True  # n est premier
```

Graphe de Flot de Contrôle (Control Flow Graph - CFG)



Explication du Graphe

1. **Début** → La fonction commence.
2. **Vérification n < 2** :
   - **Oui** → Retourne False (car n est inférieur à 2).
   - **Non** → Initialise i = 2 et passe à la boucle while.
3. **Boucle while (i * i <= n)** :
   - **Si la condition est fausse** → n est premier, on retourne True.
   - **Sinon**, on teste si n est divisible par i.
4. **Test n % i == 0** :
   - **Oui** → Retourne False (n n'est pas premier).
   - **Non** → Incrémente i et retourne au début de la boucle.
5. **Si la boucle se termine sans trouver de diviseur, n est premier** et on retourne True

Ce graphe de flot de contrôle aide à visualiser toutes les **conditions possibles** dans un programme avec une boucle.

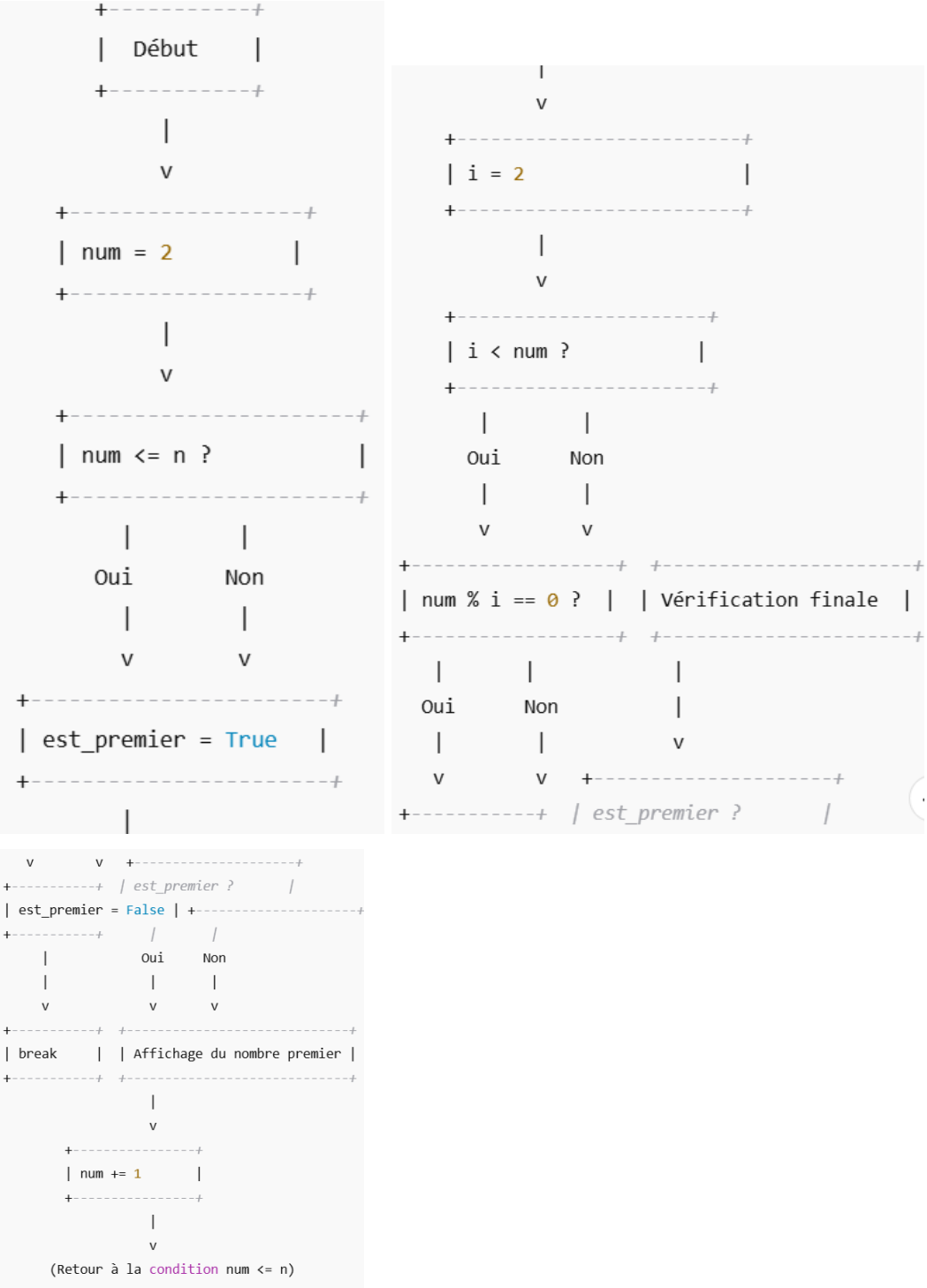
Exemple de Code : Algorithme de recherche de nombres premiers

Ce programme affiche tous les nombres premiers jusqu'à une valeur `n` en utilisant une **double boucle** (`for` imbriqué dans `while`).

```
def afficher_premiers(n):
    num = 2
    while num <= n:  # Boucle pour parcourir les nombres de 2 à n
        est_premier = True
        for i in range(2, num):  # Boucle pour vérifier si num est premier
            if num % i == 0:
                est_premier = False
                break

        if est_premier:
            print(num, "est un nombre premier")

        num += 1  # Passage au nombre suivant
```

Graphe de Flot de Contrôle (CFG)

Voici le **diagramme de contrôle** de ce programme



Explication

1. **Initialisation** : num = 2.
2. **Boucle while (num <= n)** : Tant que num est inférieur ou égal à n, on teste s'il est premier.

3. **Initialisation de est_premier = True**.
4. **Boucle for (i de 2 à num - 1)** :
    - Vérifie si num est divisible par i.
    - **Si oui** → est_premier = False, puis **sortie immédiate** (break).
    - **Si non**, la boucle continue.
5. **Après la boucle for** :
    - Si est_premier est resté True, on affiche num comme étant premier.
6. **Incrémentation de num et retour à la boucle while**.

4.5.3 Avantages du Test des Boucles (Loop Testing)

1. **Identification des erreurs dans les boucles**
    - Permet de détecter les erreurs courantes comme les **boucles infinies**, les **conditions incorrectes** et les **problèmes d'initialisation**.
2. **Test des cas limites**
    - Assure que la boucle fonctionne correctement pour :
        - **0 itération** (la boucle ne s'exécute jamais).
        - **1 itération** (le cas minimal).
        - **plusieurs itérations** (test de stabilité).
3. **Optimisation des performances**
    - Permet de **réduire les itérations inutiles**, améliorant ainsi l'efficacité de l'algorithme.
4. **Amélioration de la robustesse du code**
    - Détecte les problèmes liés à l'utilisation de **compteurs de boucles**, de **conditions d'arrêt incorrectes**, ou de **mauvaise gestion des indices** (ex: dépassement de tableau).
5. **Applicable à tous types de boucles**
    - Fonctionne pour les boucles **for**, **while**, et **boucles imbriquées**, quel que soit le langage de programmation.

4.5.4 Limites du Test des Boucles

1. **Ne couvre pas la logique interne de la boucle**
    - Il vérifie que la boucle fonctionne bien, mais pas si **les calculs à l'intérieur sont corrects**.
2. **Complexité accrue avec des boucles imbriquées**
    - Plus il y a de boucles imbriquées, plus **le nombre de chemins à tester augmente**, rendant l'analyse plus difficile.
3. **Peut être inefficace pour les grandes boucles**
    - Tester toutes les valeurs possibles pour des boucles avec un grand nombre d'itérations peut être **long et coûteux**.
4. **Ne détecte pas forcément tous les bugs**
    - Il peut manquer des erreurs liées aux **données spécifiques** ou aux **interactions avec d'autres parties du programme**.
5. **Dépend du choix des valeurs de test**
    - Un mauvais choix de cas de test peut **ne pas révéler certains défauts** (ex: tester seulement des petites valeurs pour une boucle qui doit fonctionner avec de grands nombres).

4.5.5 Conclusion

Les tests des boucles sont essentiels pour éviter les erreurs critiques telles que les boucles infinies, les sorties incorrectes ou les performances inefficaces.

- Permet d'identifier les **cas limites** (0 itération, 1 itération, plusieurs itérations).
- Aide à détecter les **boucles infinies** et les **sorties incorrectes**.
- Assure que chaque **chemin d'exécution est bien testé**.

En combinant ces tests avec d'autres techniques (test des conditions, test des chemins, test des flux de données), on garantit une meilleure robustesse du programme.

4.6 Tests de mutation (Mutation Testing)

4.6.1 Principe

- Modification volontaire du code (insertion d'erreurs) pour vérifier l'efficacité des tests.
- Objectif : S'assurer que les tests détectent bien les défauts.

Le **test de mutation** est une technique avancée de test logiciel qui vise à **évaluer l'efficacité des cas de test** en introduisant **de petites modifications (mutations)** dans le code source. Ces mutations simulent des **erreurs potentielles** qu'un développeur pourrait commettre.

L'objectif est de vérifier si les tests unitaires **détectent correctement ces erreurs**. Si un test réussit alors qu'il devrait échouer (c'est-à-dire qu'il ne détecte pas la mutation), cela signifie que le test est **faible** et doit être amélioré.

Un bon test doit "tuer" toutes les mutations pour garantir une bonne couverture du code.

◆ Exemple : Fonction de calcul d'un prix avec une réduction

Code originale

```
def calcul_prix(prix_initial, reduction):
    return prix_initial - (prix_initial * reduction)
```

Cas de test existant

```
def test_calcul_prix():
    assert calcul_prix(100, 0.1) == 90
    assert calcul_prix(50, 0.2) == 40
    assert calcul_prix(200, 0) == 200
```

◆ Application du test de mutation

On applique des **mutations artificielles** au code pour voir si les tests les détectent.

Exemple de mutations

Mutation	Modification apportée	Impact attendu
Mutation 1	`return prix_initial - (prix_initial * reduction)` → `return prix_initial + (prix_initial * reduction)`	Le prix devient plus élevé au lieu de diminuer.
Mutation 2	`return prix_initial - (prix_initial * reduction)` → `return prix_initial * reduction`	Mauvais calcul du prix après réduction.
Mutation 3	Suppression de `- (prix_initial * reduction)`	Le prix reste toujours le même, la réduction est ignorée.

Résultat attendu

- Si les tests unitaires sont bien écrits, ils doivent **échouer** lorsque le code muté est exécuté.
- Si un test **réussit malgré une mutation**, cela signifie qu'il est **insuffisant** et ne couvre pas bien le code.

✓ Un test efficace devrait détecter **toutes les mutations** et empêcher le programme de fonctionner avec un code erroné.

4.6.2 Avantages des tests de mutation

1. **Évaluation de la qualité des tests**
   - Permet de mesurer **l'efficacité des tests unitaires** en détectant les tests faibles ou insuffisants.
2. **Détection de cas de test inutiles ou redondants**
   - Met en évidence les tests qui ne détectent pas les erreurs, permettant d'améliorer la suite de tests.
3. **Amélioration de la robustesse du logiciel**
   - Aide à identifier des **bugs potentiels** qui pourraient passer inaperçus avec des tests classiques.
4. **Identification des zones non testées**
   - Si une mutation n'est pas détectée, cela signifie que cette partie du code n'est **pas couverte** par les tests.
5. **Automatisation possible**

- Il existe des outils comme **PIT (Java), MutPy (Python), Stryker (JavaScript)** qui permettent d'exécuter les tests de mutation automatiquement.

4.6.3 Désavantages des tests de mutation

1. **Coût computationnel élevé**
   - Générer et exécuter des centaines de mutations prend **beaucoup de temps et de ressources**.
2. **Difficulté d'interprétation des résultats**
   - Certains tests peuvent échouer **sans qu'il y ait réellement un problème**, rendant l'analyse plus complexe.
3. **Faux positifs et faux négatifs**
   - Certains tests peuvent "tuer" des mutations sans que cela reflète réellement une bonne couverture du code.
   - D'autres mutations peuvent sembler "vivantes" alors qu'elles ne posent pas de risque réel.
4. **Complexité de mise en place**
   - Il faut configurer des outils spécifiques et bien **définir les mutations à tester** pour éviter un **travail inutile**.
5. **Difficilement applicable aux grands projets**
   - Sur de **très gros logiciels**, tester **toutes les mutations possibles** devient pratiquement impossible sans optimisation.

Conclusion

Le **test de mutation** est une technique très puissante pour **évaluer la qualité des tests unitaires**. Il permet de :

✓ Vérifier si les tests couvrent bien toutes les parties critiques du code.
✓ Détecter les tests inefficaces qui ne repèrent pas certaines erreurs.
✓ Renforcer la robustesse du logiciel en améliorant la détection des bugs.

Les tests de mutation sont très efficaces pour évaluer la qualité des tests unitaires, mais ils sont coûteux en temps et en ressources. Ils doivent être **bien planifiés et combinés avec d'autres méthodes de tests** pour être réellement utiles.

🔹 Idéal pour les **petits modules critiques**, mais difficile à appliquer à **grande échelle** sans optimisation.

À utiliser surtout pour améliorer les tests existants et non comme seule méthode de validation.

Cependant, cette approche peut être **coûteuse en calcul**, car elle nécessite de tester plusieurs variantes du code. Elle est donc souvent utilisée avec des outils automatisés comme **PyMutate (Python)**, **PIT (Java)** ou **MutPy**.

5. Classification des outils de test

Les outils de test en boîte blanche sont utilisés pour évaluer la structure interne d'un programme ou d'un système. Ils permettent de vérifier le fonctionnement correct des composants internes en se basant sur le code source, les algorithmes et les structures de données. Voici une classification des techniques et outils associés à ces tests, accompagnée d'exemples.

5.1 Outils d'Analyse Statique

Ces outils examinent le code source sans l'exécuter pour détecter les erreurs de syntaxe, les vulnérabilités de sécurité et les violations des bonnes pratiques.

Exemples :

- **SonarQube** – Analyse statique pour détecter les erreurs de codage et améliorer la qualité du code.
- **Checkstyle** – Vérification du respect des conventions de codage en Java.
- **Flake8** – Analyse de code Python pour vérifier la conformité aux normes PEP 8.
- **Closure Compiler** : pour la vérification syntaxique et le typage en JavaScript1.

- **Linters (comme ESLint pour JavaScript**) : pour détecter des problèmes de style ou de logique.

5.2 Outils d'Analyse Dynamique

Ces outils exécutent le programme et surveillent son comportement pour détecter les erreurs d'exécution, les fuites mémoire et les défauts de performance.

Exemples :

- **Valgrind** – Détection des fuites de mémoire et erreurs d'accès mémoire en C/C++.
- **JProfiler** – Profilage des performances des applications Java.
- **GDB (GNU Debugger)** – Débogage et exécution pas à pas du code en C/C++.

5.3 Outils de Couverture de Code

- **Description** : Mesure du degré auquel le code est testé. Les critères incluent :

Couverture des instructions (chaque ligne de code doit être exécutée au moins une fois).
Couverture des branches (chaque condition doit être évaluée à vrai et à faux).
Couverture des chemins (chaque chemin possible dans le code doit être testé).

Exemples :

- **JaCoCo** – Outil de couverture pour Java.
- **gcov** – Outil de couverture pour le code C/C++.
- **Coverage.py** – Mesure de la couverture du code Python.
- **Visual Studio Code Coverage :** intégré à Visual Studio pour .NET

5.4 Outils de Test Unitaire

Ils permettent d'automatiser les tests unitaires en exécutant des fonctions spécifiques pour vérifier leur bon fonctionnement. Vérification individuelle des fonctions ou méthodes pour s'assurer qu'elles produisent les résultats attendus.

Exemples :

- **JUnit** – Framework de test unitaire pour Java.
- **NUnit :** framework pour .NET.
- **PyTest** – Outil de test pour Python.
- **Google Test (GTest)** – Framework de test unitaire pour C++.

5.5 Outils de Test de Sécurité

Ces outils aident à identifier les failles de sécurité dans le code source et les vulnérabilités potentielles.

Exemples :

- **OWASP ZAP** – Détection des vulnérabilités dans les applications web.
- **Fortify Static Code Analyzer** – Analyse de sécurité pour Java, C, C++, etc.
- **Bandit** – Analyse statique de sécurité pour Python.

5.6 Outils de Test de Mutation

Ils modifient légèrement le code source (mutations) pour vérifier la robustesse des tests unitaires.

Exemples :

- **PIT (Pitest)** – Outil de test de mutation pour Java.
- **MutPy** – Outil de test de mutation pour Python.
- **Mull** – Outil de test de mutation pour C/C++.

5.7 Outils de Test des Flux de Données

Ils analysent comment les variables sont définies et utilisées dans le code, détectant les anomalies potentielles.

Exemples :

- **CodeSonar** – Analyse des flux de données et détection des erreurs critiques.
- **Klocwork** – Analyse des défauts liés à l'utilisation des variables.

Cette classification permet de choisir l'outil approprié selon les besoins du test en boîte blanche. Chaque outil joue un rôle clé dans l'amélioration de la qualité et de la sécurité du logiciel.

5.8 Outils de Test basés sur le graphe de flot de contrôle

- **Description** : Utilisation du graphe représentant les flux d'exécution dans le programme pour identifier les chemins critiques.
- **Exemples d'outils** :
- **Parasoft Jtest** : pour l'analyse des chemins dans le code Java.
- **Control Flow Graph Visualizers** intégrés dans certains IDE.

5.9 Outils de Test basés sur l'exécution symbolique

- **Description** : Technique qui simule l'exécution du programme avec des variables symboliques au lieu de valeurs concrètes, afin de générer automatiquement des cas de test.
- **Exemples d'outils** :
- **KLEE** : framework pour l'exécution symbolique sur LLVM.
- **Pex** (Microsoft) : outil d'exécution symbolique pour .NET.

5.10 Outils de Test basés sur la vérification formelle

- **Description** : Utilisation de méthodes mathématiques pour prouver la correction du programme par rapport à ses spécifications formelles.
- **Exemples d'outils** :
- **SPIN** : vérification formelle basée sur le model-checking.
- **Coq** : assistant de preuve utilisé en programmation formelle.

6. Application de test en boite blanche : JUnit

JUnit est un framework open source pour le développement et l'exécution de tests unitaires en Java. Voici les principes clés de JUnit :

6.1 Principe de fonctionnement

1. **Cas de tests** : Les tests sont exprimés dans des classes sous forme de cas de tests avec leurs résultats attendus. Chaque cas de test est une méthode qui vérifie une fonctionnalité spécifique du code.
2. **Assertions** : Les assertions sont utilisées pour vérifier que le comportement du code correspond aux attentes. JUnit propose plusieurs méthodes d'assertion comme assertEquals, assertTrue, etc.
3. **Exécution automatique** : Les tests peuvent être exécutés automatiquement, ce qui permet de détecter rapidement les erreurs après des modifications du code.
4. **Cycle de vie** : JUnit gère le cycle de vie des tests, permettant de configurer des fixtures (setup et teardown) pour initialiser et nettoyer les ressources nécessaires aux tests.
5. **Organisation des tests** : Les tests peuvent être organisés en suites de tests pour faciliter leur exécution et leur gestion.

6.2 **Avantages**

- **Qualité du code** : JUnit aide à maintenir la qualité du code en détectant les erreurs dès le développement.
- **Non-régression** : Les tests unitaires garantissent que les modifications ne cassent pas les fonctionnalités existantes.
- **Développement incrémental** : JUnit est adapté pour le développement incrémental et la méthode Test-Driven Development (TDD).

6.3 **Versions de JUnit**

- **JUnit 3** : Utilise des classes et méthodes spécifiques pour les tests.
- **JUnit 4** : Introduit les annotations comme @Test, @Before, @After, etc., pour simplifier l'écriture des tests.

6.4 **JUnit verdicts (jugements) :**sont définis grâce aux assertions placées dans les cas de test

Pass (vert) : pas de faute détectée

Fail (rouge) : échec, on attendait un résultat, on en a eu un autre

Error : le test n'a pas pu s'exécuter correctement (exception inattendue, ...)

6.5 Etapes de construction des tests sous Junit

Aucune entrée de table des matières n'a été trouvée.Ecrire la classe a testée dans un projet Java sous Eclipse.

1. Accédez à votre projet Java
2. Créer un nouveau dossier source nommé test (dans ce même projet)
3. Créer un nouveau package (il a le même nom que le package de la classe à tester)
4. Choisissez la classe que vous voulez tester,
5. Allez dans Fichier /New ... / JUnitTest Case
6. Sélectionner les méthodes à tester (ou toutes)
7. Construire un objet qui manipule la méthode à tester.

Pour exécuter les tests :

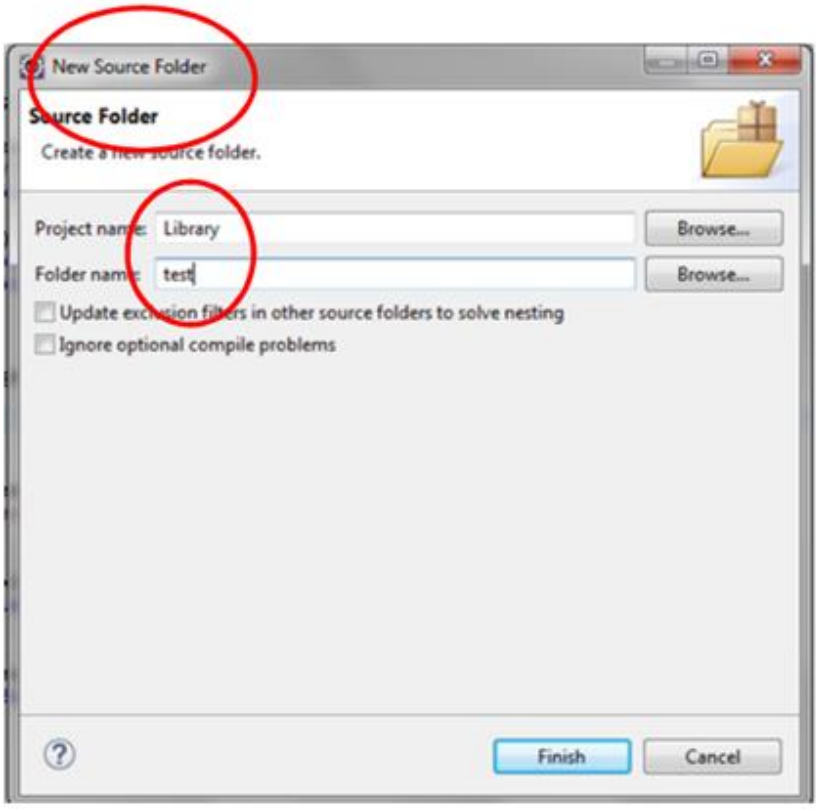
Choisissez Run /Run As /JUnit Test

Démonstration : soit la classe Etudiant en java :

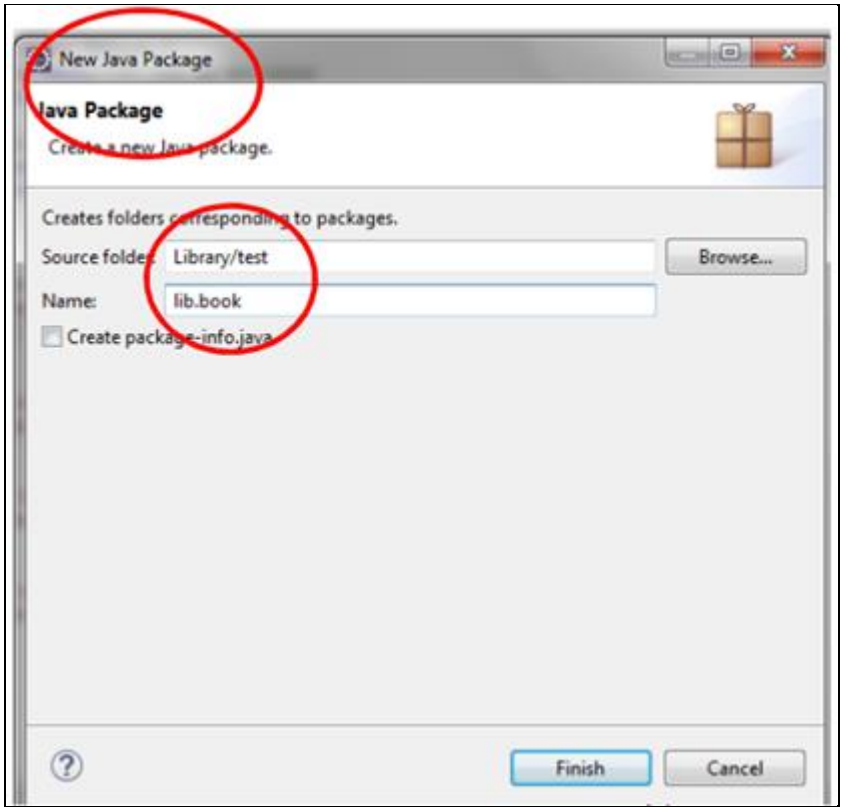
```
package mediateheque;
import java.util.ArrayList;
import java.util.Date;

public class Etudiant {
    String nom;
    String prenom;
    int age;
    float moy, MG;
    Date Dnaissance;
        ArrayList <float> note= new ArrayList <float> ();

    public Etudiant(String nom, String prenom, int age, float moy, Date dnaissance) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
        this.moy = moy;
        Dnaissance = dnaissance;
    }
    public void lectureNote ()
    {}

    float calculerMG (ArrayList note)
    {return  (MG);
    }
```

Figure 1 : Classe Etudiant

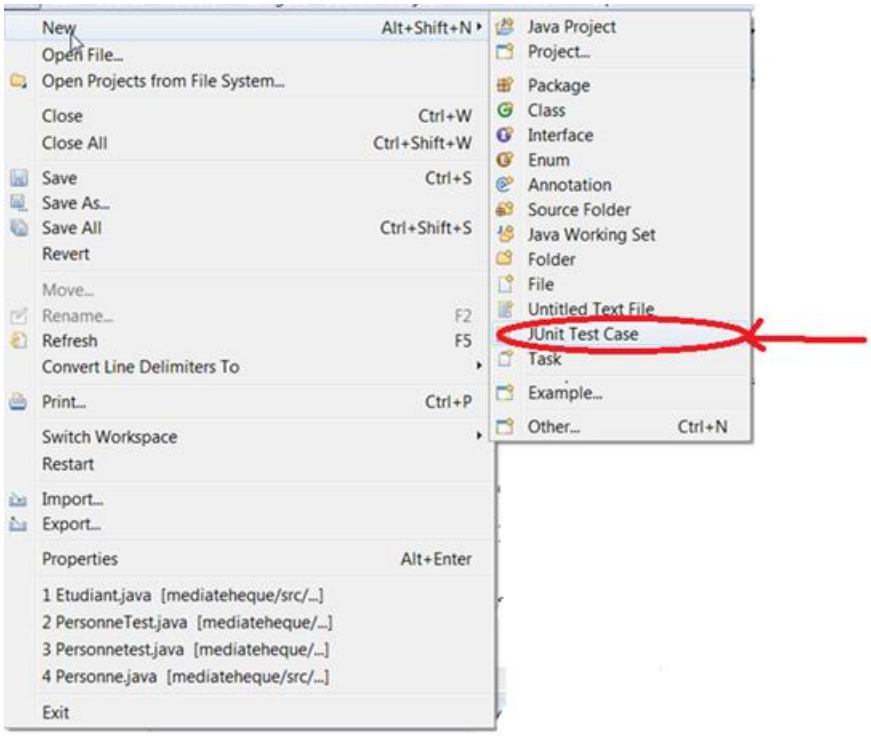
- **Création de nouveau dossier nommé « test »**



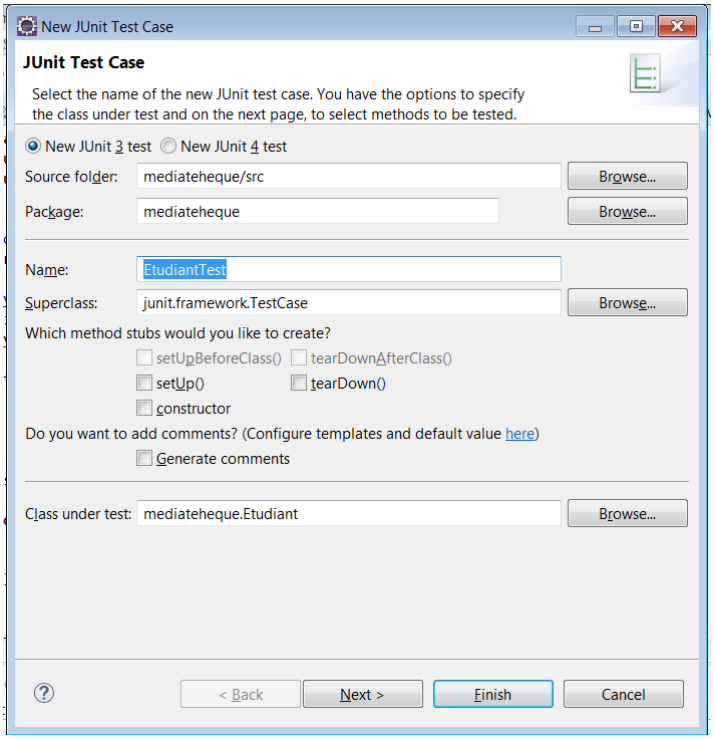
- **Création d'un package porte le même nom que celui à testé**



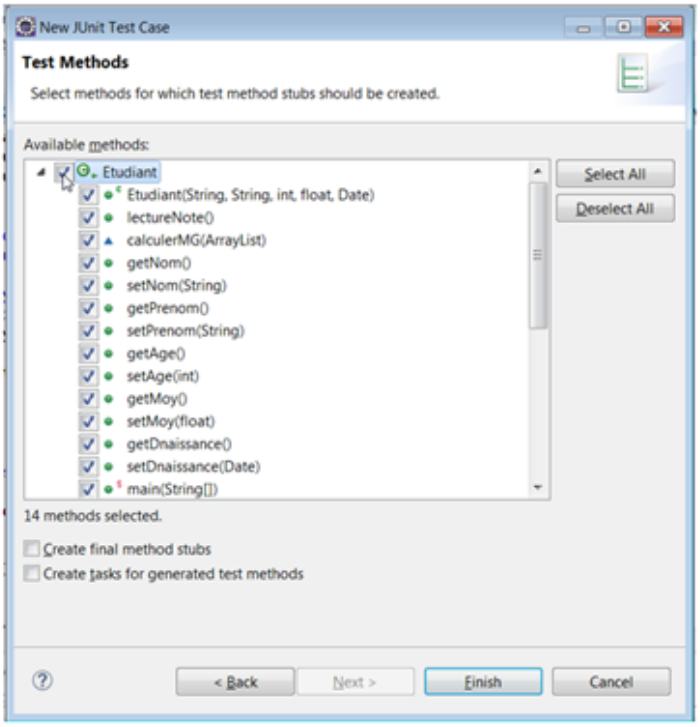
- **Création d'une classe de test à partir des classes java à testées**

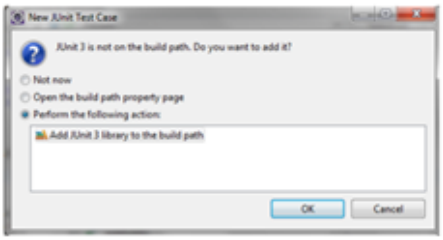


- **La classe de teste porte le même nom que la classe java à testée**



- **Après la création de classe, il faut créer les méthodes de test :**



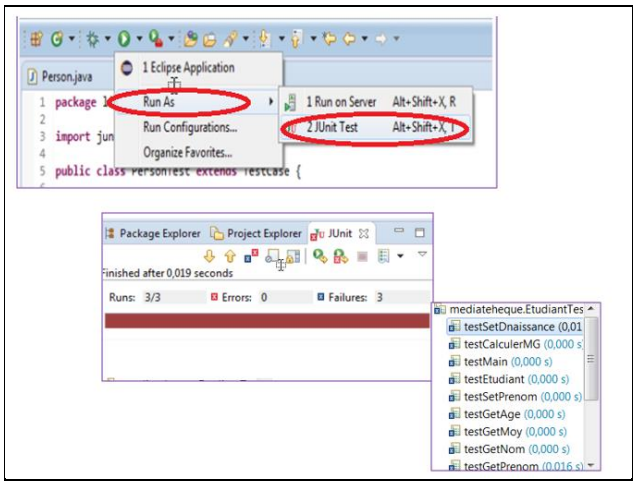


- **JUnit va génère les squelettes des classes de tests :**

```
1  package mediateheque;
2
3  import junit.framework.TestCase;
4
5  public class EtudiantTest extends TestCase {
6
7      public void testEtudiant() {
8          fail("Not yet implemented");
9      }
10
11     public void testLectureNote() {
12         fail("Not yet implemented");
13     }
14
15     public void testCalculerMG() {
16         fail("Not yet implemented");
17     }
18
19     public void testGetNom() {
20         fail("Not yet implemented");
21     }
22
23     public void testSetNom() {
24         fail("Not yet implemented");
25     }
```

Remarque : Pour manipuler une méthode de test il faut créer un objet qui manipule la méthode à testée

- **L'exécution de cas de test se fait par JUnit Test**



6.6 Méthodes d'assertion

Méthode	**Rôle et exemple**
assertEquals ()	Vérifier l'égalité de deux valeurs de type primitif ou objet (en utilisant la méthode equals()). Il existe de nombreuses surcharges de cette méthode pour chaque type primitif, pour un objet de type Object et pour un objet de type String. Exemple utilisant une méthode : assertEquals ("mohamed dib ", p.getName()); Exemple utilisant un attribut : assertEquals ("l'incendie", b.tilel);
assertTrue ()	Vérifier que la valeur fournie en paramètre est vraie. Exemple utilisant une méthode : assertTrue (dic.isEmpty()); Exemples utilisant un attribut : assertTrue(dic.keys instanceof ArrayList); assertTrue (b.author instance of String); Exemple utilisant une expression : assertTrue (p.getMaximumBooks()==3);
assertFalse ()	Vérifier que la valeur fournie en paramètre est fausse. Exemple utilisant une méthode : assertFalse (dic.isEmpty()); Exemple utilisant une expression : assertFalse (p.getMaximumBooks() >3);
assertNull ()	Vérifier que l'objet fourni en paramètre soit null. Exemple : assertNull (b2.getPerson()) ;
assertNotNull ()	Vérifier que l'objet fourni en paramètre ne soit pas null. Exemple : b1.setPerson (p1) ; assertNotNull (b1.getPerson()) ;

assertSame ()	Vérifier que les deux objets fournis en paramètre font référence à la même entité. Exemples identiques :
	Exemple : assertSame ("Les deux objets sont identiques", obj1, obj2); //Obj obj1=new Obj ();
	Obj obj2=obj1 ;
	Exemple : assertTrue ("Les deux objets sont identiques ", obj1 == obj2);

6.7 Application :Exercice corrigé : les tests sous JUnit

Soit la **classe Pile** qui est caractérisée comme montre la figure en face. Afin d'appliquer les tests sous **JUnit** : réaliser les suivantes méthodes de test par les assertions les plus pertinentes :

Pile
Int taille = 10 ;
PileVide ( ) : Boolean ; HauteurPile ( ) : Int ; PilePleine ( ) : Boolean ; Empiler (Int : a) : Int; Dépiler ( ) : Int; Succ (Int: s): Int ;

1 **Vérifier** le **type** de l'attribut «taille » selon son constructeur.

2 **Dépiler** dans une pile **vide** ! (il faut avoir une plie vide), aussi :
Vérifier l'auteur et le **successeur** de la pile dans cette situation.

3 **Empiler** dans une pile **pleine** ! (il faut avoir une plie pleine), aussi :
Vérifier l'auteur et le **successeur** de la pile dans la nouvelle situation.

Il existe une variété des solutions possibles

- **Solution de la question 1 :**

```
import java.util.Pile;
import junit.framework.TestCase;
public class PileTest extends TestCase {
public void testPile(){
        Pile p = new Pile ();
    assertTrue (p.taille instanceof Int);
    assertTrue (p.gettaille()==10);    }
```

- **Solution de la question 2 :**

```
 public void testDépiler (){
      Pile p1 = new Pile ();
      Int a= p1.Dépiler() ;
assertEquals(0,a);
assertEquals(true, p1.PileVide() );
assertEquals(false, p1.PilePleine() ) ;
assertEquals(0, p1.HauteurPile());
assertTrue(p1.PileVide());
assertFalse(p1.PilePleine()) ;

Int b=p1.HauteurPile();
assertTrue(b==0);
assertTrue(p1.position()==0);
Int c=Succ(b) ;
Int c=Succ(p1. HauteurPile();) ;
assertTrue(p1.Succ(0)==1);
assertTrue(p1.Succ(p1.position)==1);
assertEquals(c, p1.position+1);}
```

- **Solution de la question 3 :**

```
public void testEmpiler (){
    Pile p2 = new Pile ();
    Int a= p2. Empiler (10) ;     // pile pleine !
assertEquals(a,11);
assertEquals(false, p2.PileVide() );
assertEquals(true, p2.PilePleine() ) ;
assertEquals(10, p2.position);
assertEquals(10, p2. Hauteur());
assertFalse(p2.PileVide());
assertTrue(p2.PilePleine()) ;

Int b=p2. HauteurPile();
 assertTrue(p2.position()==10);
assertFalse(b<10);
Int c=Succ(p2.position()) ;
Int c=Succ(p2.HauteurPile()) ;
assertTrue(p2.position==10);
assertTrue(c=11) ;
assertEquals(c, p1.position+1);}
```

7. Développement dirigé par les Tests : TDD (Test Driven Development)

7.1 Principe

C'est une méthode de développement logiciel qui consiste à concevoir et écrire des tests **avant** même de développer le code fonctionnel correspondant. Cette approche est largement utilisée dans les environnements agiles et est basée sur un cycle itératif de développement.

Le TDD est une méthodologie qui vise à réduire les anomalies dans un logiciel en favorisant la mise en œuvre fréquente de tests. Les développeurs écrivent d'abord un test unitaire qui échoue, puis ils créent le code minimum nécessaire pour faire passer ce test. Une fois le test réussi, ils

refactorisent le code pour le rendre plus lisible et efficace, tout en conservant la capacité à passer les tests existants.

7.2 Cycle du TDD

Le cycle du TDD se compose de trois étapes principales :

1. **Écrire un test** : Commencez par écrire un test unitaire qui décrit une fonctionnalité ou un comportement attendu du programme. Ce test doit échouer puisqu'il n'y a pas encore de code pour le faire passer.
2. **Faire passer le test** : Écrivez ensuite le code minimum nécessaire pour faire passer le test. Ce code doit être simple et direct, sans trop de complexité.
3. **Refactorer le code** : Une fois le test réussi, refactorisez le code pour le rendre plus lisible, efficace et maintenable, tout en s'assurant que tous les tests existants continuent de passer.

7.3 Avantages du TDD

- **Qualité du code** : Le TDD améliore la qualité du code en garantissant qu'il est testé et validé à chaque étape.
- **Moins de bugs** : En écrivant les tests avant le code, les développeurs réduisent le nombre de bugs et d'erreurs.
- **Maintenance simplifiée** : Le code est plus maintenable car il est conçu pour être testé et refactorisé régulièrement.

7.4 Lois du TDD

Le TDD repose sur trois lois fondamentales :

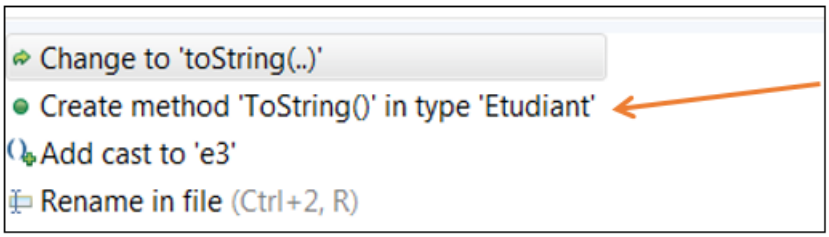
1. **Écrire un test qui échoue avant d'écrire le code**.
2. **Ne pas écrire un test plus compliqué que nécessaire**.
3. **Ne pas écrire plus de code que nécessaire pour faire passer le test**

8.2 Application : Utiliser « first test approach » pour écrire la méthode toString de la classe Etudiant

```
public void testToString() {
    Etudiant e3= new Etudiant ();
    e3.setNom("aaaa bbbb");
    e3.setPrenom("cccc ");
```

```
e3.setAge(40);
    String phrase ="omar (age = 63)";
assertEquals(phrase, e3.ToString());}
```

- **Le compilateur demande d'ajouter la méthode « *ToString* » :**



8.3 Avantages du Test-Driven Development

L'implémentation de la méthode Test-Driven Development offre plusieurs avantages, bien au-delà de la simple validation de l'exactitude fonctionnelle :

Amélioration de la qualité du code : TDD incite les développeurs à rédiger un code propre, modulaire et testable dès le départ. Cela conduit à une meilleure qualité logicielle globale et une plus grande maintenabilité.

Débogage plus rapide : en détectant les bugs tôt dans le processus de développement grâce aux tests continus, il devient plus facile d'identifier et résoudre rapidement les problèmes – économisant ainsi un temps précieux lors des phases de débogage.

Collaboration renforcée : TDD favorise la collaboration entre membres d'une équipe en fournissant une compréhension claire des exigences grâce à des tests bien définis qui agissent comme spécifications exécutables pour les développeurs comme pour les parties prenantes.

Confiance accrue lors du refactoring : avec des suites automatisées couvrant exhaustivement toutes fonctions critiques, le refactoring devient moins risqué car toute régression introduite est rapidement identifiée par des tests échoués.

8.4 Problèmes du Test-Driven Development

Bien que le développement piloté par les tests offre de nombreux avantages, il existe des défis potentiels à considérer :

Courbe d'apprentissage initiale : Adopter le TDD nécessite une familiarité avec les cadres de test et une discipline dans la rédaction de tests unitaires significatifs – un ensemble de

compétences qui pourrait nécessiter un investissement initial en apprentissage pour les équipes nouvelles dans cette approche.

Investissement en temps : Écrire des tests avant la mise en œuvre des fonctionnalités peut sembler contre-intuitif au départ, mais peut finalement faire gagner du temps sur le long terme grâce à la réduction des efforts de débogage plus tard.

Suraccentuation sur les tests unitaires : Bien que les tests unitaires soient cruciaux dans les pratiques TDD, il est important de ne pas négliger d'autres formes d'essais comme l'intégration ou la vérification au niveau du système qui jouent un rôle tout aussi significatif pour assurer l'exactitude logicielle.

8. Avantages et inconvénients de la méthode boite blanche

8.1 Les avantages

- **L'importance** de la couverture du code réside dans sa capacité à fournir un aperçu de la qualité et de l'efficacité d'une suite de tests. Les mesures de couverture de code peuvent aider votre équipe à : Identifier le code découvert qui n'a pas été testé et qui pourrait contenir des bogues ou des problèmes potentiels non résolus par la suite de tests.
- **Anticipation :** effectuer ces tests au cours du développement d'un programme permet de repérer des points bloquants qui pourraient se transformer en erreurs ou **problèmes** dans le futur (par exemple lors d'une montée en version, ou même lors de l'intégration du composant testé dans le système principal).
- **Optimisation :** étant donné qu'il travaille sur le code, le testeur peut également profiter de son accès pour optimiser le code, pour apporter de meilleures performances au système étudié (sans parler de sécurité…).
- **Exhaustivité :** étant donné que le testeur travaille sur le code, il est possible de vérifier intégralement ce dernier. C'est le type de test qui permet, s'il est bien fait, de tester l'ensemble du système, sans rien laisser passer. Il permet de repérer des bugs et vulnérabilités cachées intentionnellement.

8.2 Les inconvénients :

- **Complexité :** ces tests nécessitent des compétences en programmation, et une connaissance accrue du système étudié.
- **Durée** : de par la longueur du code source étudié, ces tests peuvent être très longs.

- **Industrialisation :** pour réaliser des tests en « boîte blanche », il est nécessaire de se munir d'outils tels que des analyseurs de code, des débogueurs… Cela peut avoir un impact négatif sur les performances du système, voire même impacter les résultats.
- **Cadrage :** il peut être très compliqué de cadrer le projet. Le code source d'un programme est souvent très long, il peut donc être difficile de déterminer ce qui est testé, ce qui peut être mis de côté… En effet, il n'est pas toujours réaliste de tout tester, ce qui prendrait trop de temps. Il est également possible que le testeur ne se rende pas compte qu'une fonctionnalité prévue dans le programme n'y a pas été intégrée. Il n'est donc pas dans le scope du testeur de vérifier si tout est là : il ne fait que tester ce qui est effectivement présent dans le code.
- **Intrusion :** cette méthode est très intrusive. Il peut en effet être risqué de laisser son code à la vue d'une personne externe à son entreprise : il y a des risques de casse, de vol, … Choisissez donc toujours des testeurs professionnels !

9 Conclusion

Au cœur des tests en boîte blanche se trouve l'utilisation de critères de couverture de code, qui permettent aux testeurs d'analyser et de mesurer dans quelle mesure le code source de l'application a été testé pendant les tests. Divers critères de couverture du code incluent la couverture des instructions, la couverture des branches, la couverture des conditions, la couverture du chemin et la couverture des fonctions, qui visent à examiner différents aspects du code pour garantir un processus de test complet. Ces mesures de couverture contribuent à l'établissement d'une stratégie d'assurance qualité robuste, minimisant les risques de dysfonctionnement ou de panne du logiciel.

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TD N° 2 : Techniques de test en boite blanche

Exercice : Soient les problèmes reconnus suivants :

Problème 1 : Trier un tableau T de N éléments.

Problème 2 : Calculer le nombre d'occurrence NO de mot « ELLE » dans un tableau T.

Problème 3 : Insérer un élément E dans un tableau T trié

Problème 4 : Appliquer le décalage circulaire à droite d'ordre R d'un tableau T

Travail demandé : Appliquez les techniques de test structurel pour résoudre ces problèmes :

1- Ecrire votre solution sous forme d'un algorithme ou d'un code
2- Etiqueter votre code si nécessaire
3- Elaborer le graphe de flux de contrôle
4- Calculer la complexité cyclomatique V(G)
5- Identifier les chemins possibles
6- Proposer des jeux de test pour chaque chemin
7- Appliquez les 6 techniques de test structurel
8- Raffiner votre solution (code)
9- Rapporter votre travail (document pdf)

Remarque : Le travail est individuel

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

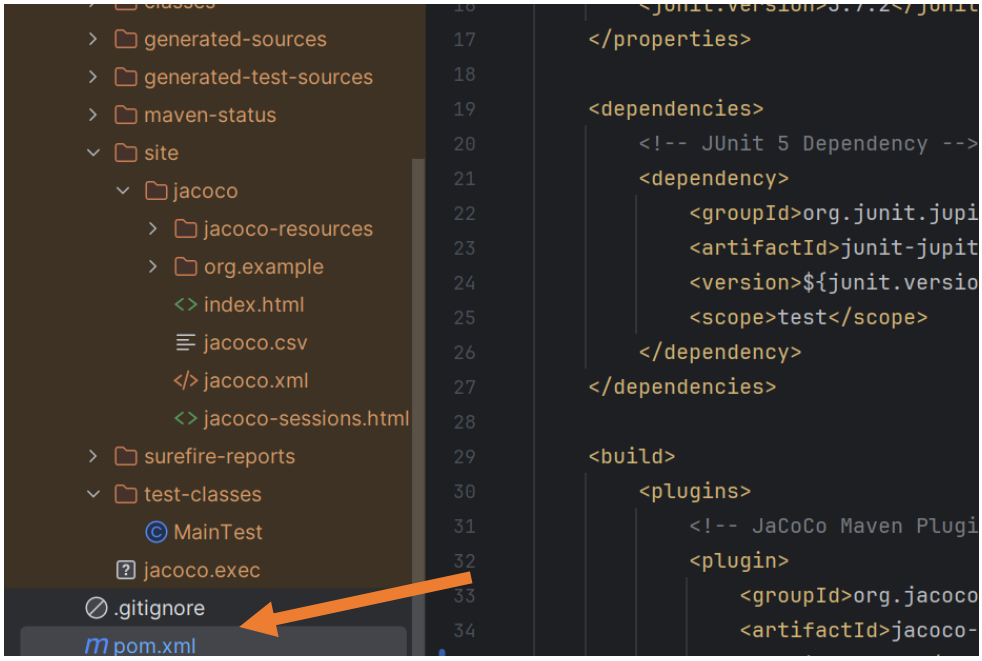
TP N° 2-1 test structurel (white box): Application : L'outil le JAva COde COverage (JACOCO)

JACOCO:

This is a tool that allows you to check the coverage of the code by the tests: it is called code coverage. It looks at what part of the code is covered by the tests. To do this, it simply looks at which lines of code are executed when the tests are launched.

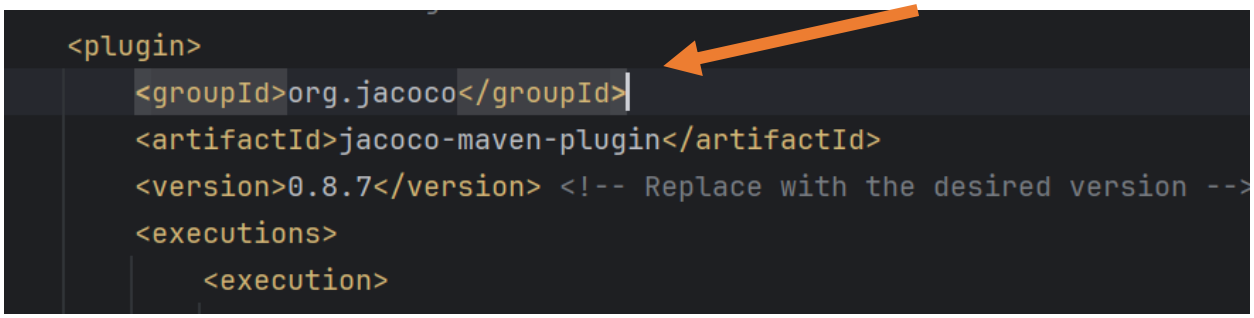
1) Installation steps JACOCO :

Add code xml with pom (project oriente model).xml

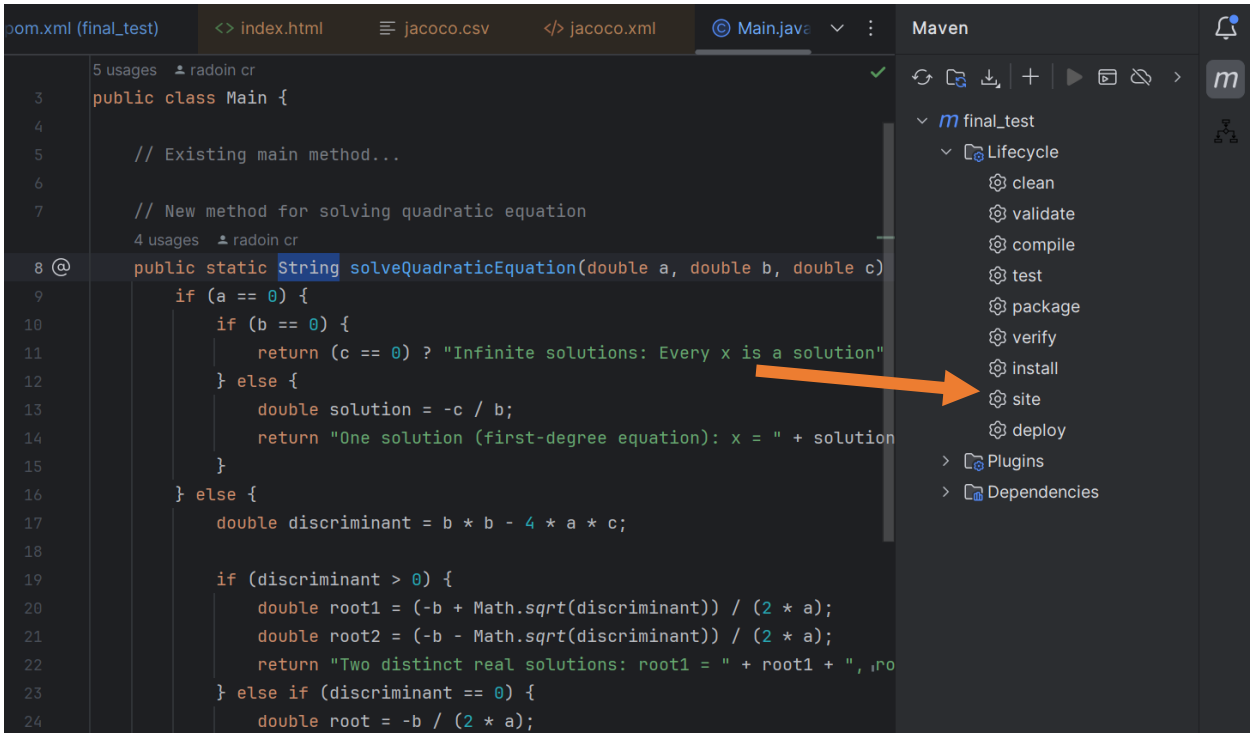


Add Junit library

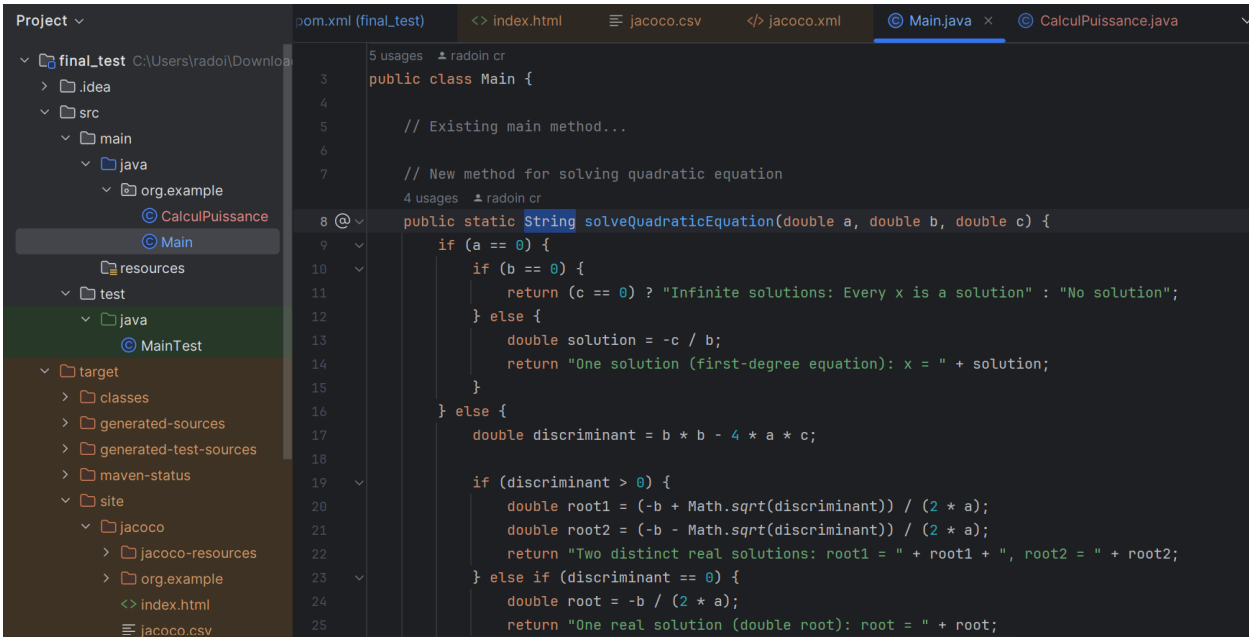
Add jacoco library



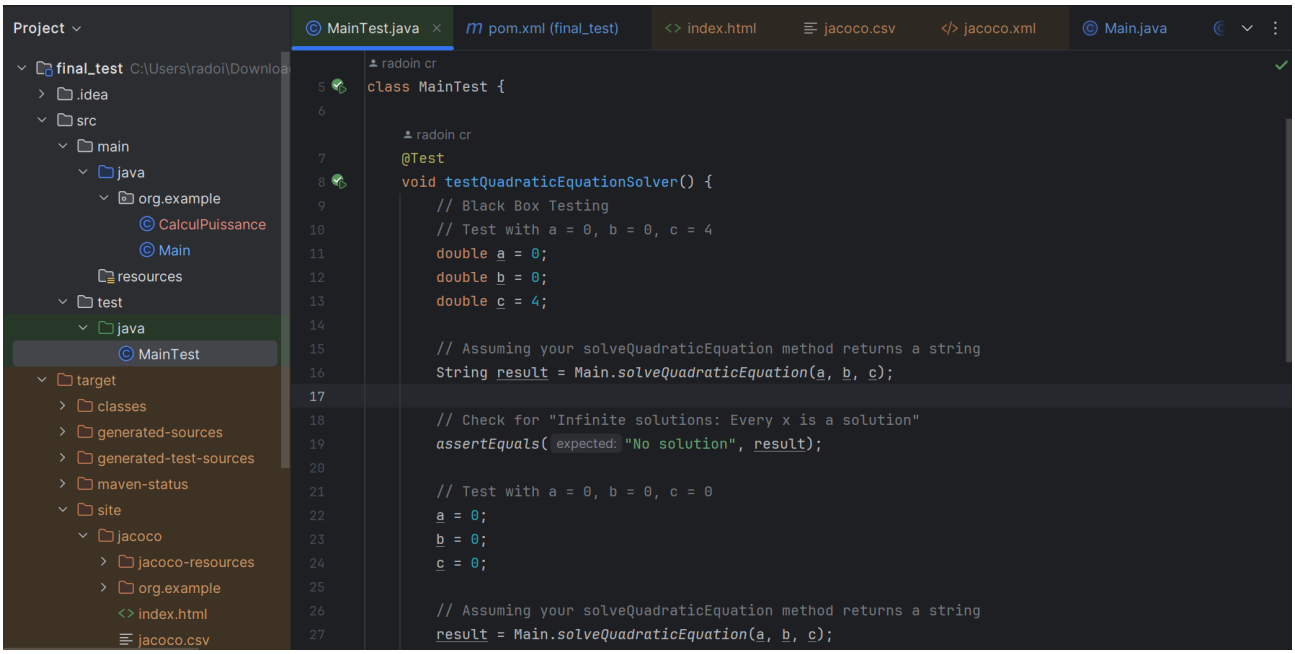
Install JACOCO and JUnit



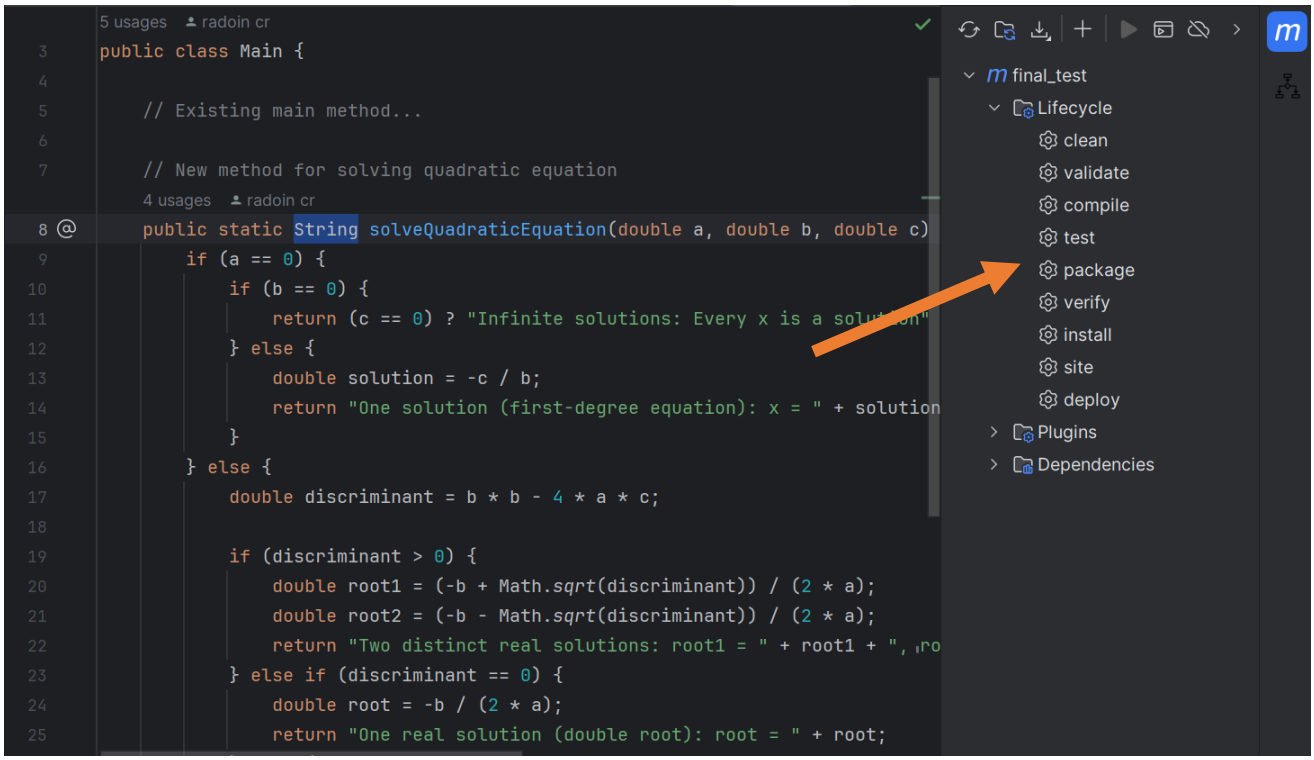
Classe Main



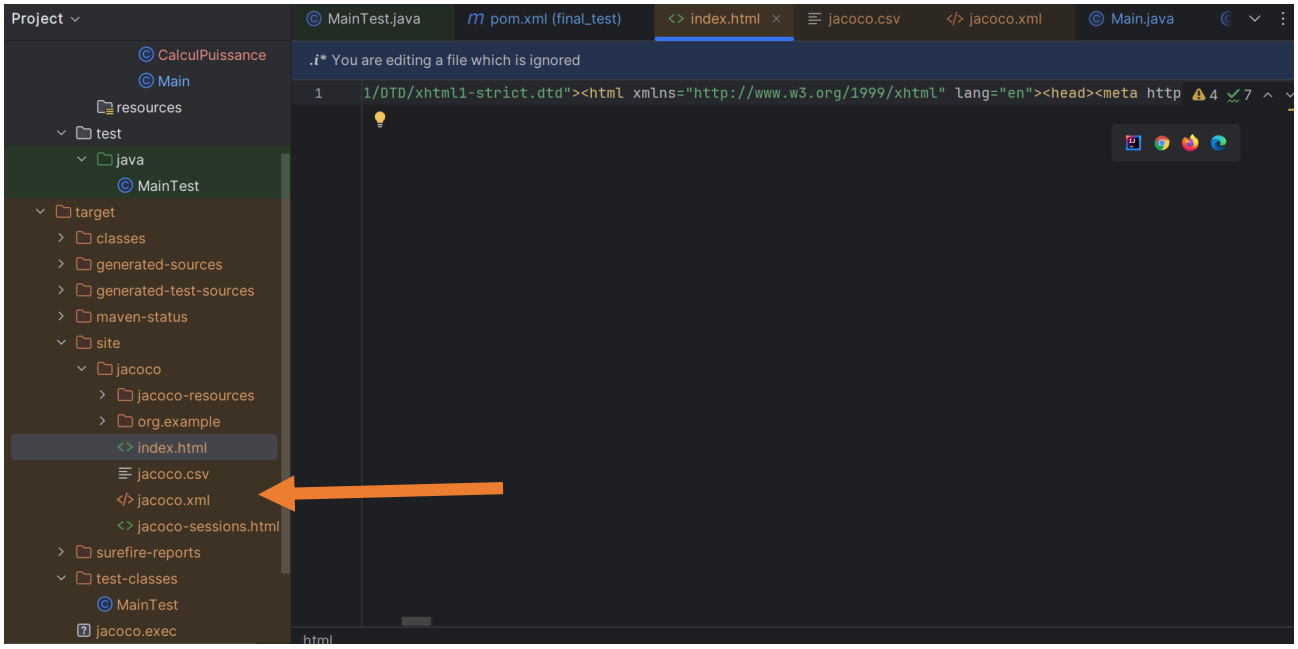
Classe MainTest



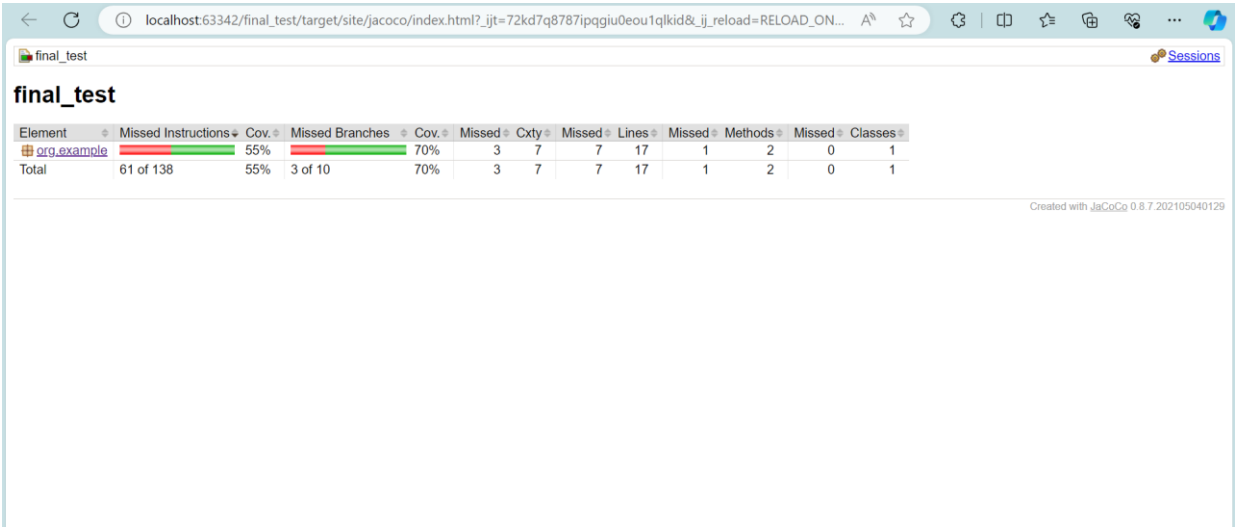
Test Project



Output Test:generate Auto>>create index.html



Percent covrage



Covrage classe java

Main.java

```
1.  package org.example;
2.
3.  public class Main {
4.
5.      // Existing main method...
6.
7.      // New method for solving quadratic equation
8.      public static String solveQuadraticEquation(double a, double b, double c) {
9.          if (a == 0) {
10.             if (b == 0) {
11.                 return (c == 0) ? "Infinite solutions: Every x is a solution" : "No solution";
12.             } else {
13.                 double solution = -c / b;
14.                 return "One solution (first-degree equation): x = " + solution;
15.             }
16.         } else {
17.             double discriminant = b * b - 4 * a * c;
18.
19.             if (discriminant > 0) {
20.                 double root1 = (-b + Math.sqrt(discriminant)) / (2 * a);
21.                 double root2 = (-b - Math.sqrt(discriminant)) / (2 * a);
22.                 return "Two distinct real solutions: root1 = " + root1 + ", root2 = " + root2;
23.             } else if (discriminant == 0) {
24.                 double root = -b / (2 * a);
25.                 return "One real solution (double root): root = " + root;
26.             } else {
27.                 double realPart = -b / (2 * a);
28.                 double imaginaryPart = Math.sqrt(Math.abs(discriminant)) / (2 * a);
29.                 return "Complex solutions: Root1 = " + realPart + " + " + imaginaryPart + "i, Root2 = " + realPart + " - " + imaginaryPart + "i";
30.             }
31.         }
32.     }
33.
34. }
```

Explain:

In test coverage reports using the Jacoco tool, different colors are used to represent the coverage information for each part of the source code. The main colors typically used are green, yellow, and red, and they convey the following general meanings:

Green:

Indicates that the code has been well-tested, with full or nearly full coverage of the source code. Means that tests have been executed for the code, and no errors were detected.

Yellow:

Indicates that some parts of the source code have not been fully tested. There may be some checks, classes, or lines that have not been tested yet.

Red:

Indicates that a significant portion of the source code has not been tested. Points to areas in the code that have not been reviewed by tests and may potentially contain undiscovered errors.

In summary, these colors are used to provide a quick overview of the test coverage for the source code, helping the development team identify areas that may need additional attention and testing.

Application :

- Choose an example from the examples in TD-2, propose your code in Java.
- Describe all the steps: from the integration of the tool to the display of the result.
- Report your work (pdf document).

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TP N° 2-2 Test structurel (white box) JUnit : Application

L'objectif

Ce TP est pour d'écrire et d'exécuter des tests avec JUnit pour une classe Java dont les instances sont des tableaux redimensionnables. À partir de la spécification informelle donnée, vous devez écrire un ensemble de classes JUnit de façon à pouvoir tester des implantations en boîte blanche. Proposez les squelettes des classes de test ainsi que les implantations à tester.

Spécification

On considère la classe **Rarray** implantant des tableaux redimensionnables. Une instance de la classe Rarray est un tableau dont les valeurs sont des objets supposés non **null**. Un même objet peut être présent plusieurs fois dans le tableau.

Le constructeur de la classe **Rarray** permet d'initialiser un **Rarray** vide de la capacité initiale passée en argument, qui doit être strictement positive. Il est toujours possible d'ajouter un élément à un **Rarray**, sa capacité est augmentée si besoin. On peut supprimer une occurrence d'un objet avec la méthode **remove** ou toutes les occurrences d'un objet avec la méthode **removeAll**. Ces deux méthodes renvoient **vrai** si un élément a effectivement été supprimé, **faux** si l'élément passé en argument n'est pas présent dans le tableau. Il est également possible de **vider** entièrement un **Rarray** avec **clear**. La méthode **contains** permet de savoir si un élément est **présent** dans un **Rarray**, la méthode **nbOcc** donne le **nombre** d'occurrences d'un objet et la méthode **size** donne le **nombre total** d'éléments présents dans le tableau.

Le squelette de la classe Java Rarray est le suivant.

```
public class Rarray {
public Rarray(int capacite) throws RarrayError { }
public void add (Object elt) { }
public boolean remove (Object elt) { }
public boolean removeAll (Object elt) { }
public void clear() { }
public boolean contains (Object elt) { }
public int nbOcc(Object elt) { }
public int size() { }
}
```

Exercice 1

1. Complétez les squelettes des trois classes de test fournies en suivant les exemples donnés. Pour chacun des tests, vous préciserez obligatoirement en commentaire l'objectif du test ainsi que le résultat attendu. Pensez à tester aussi bien les cas qui doivent réussir que les cas qui doivent lever une exception : l'objectif est de couvrir un maximum de cas. Pensez également aux cas aux limites.

2. Exécutez vos tests sur chacune des implantations fournies et rédigez un rapport de test sous la forme d'un tableau : pour chaque implantation, dites si les tests ont réussi ou échoué et donnez les raisons apparentes des fautes trouvées. Vous pouvez suivre le modèle suivant :

Implantation	Résultats des tests	Fautes trouvées
rarray13	Échec	Supprimer un objet présent plusieurs fois supprime toutes les occurrences de l'objet

Exercice 2 : complément à la classe **Rarray**

1. On ajoute à la classe **Rarray** des méthodes permettant de manipuler les objets du tableau en fonction de leur indice. La méthode index permet de connaître l'indice d'une occurrence d'un objet et la méthode **get** permet de connaître l'objet présent à un indice. On peut supprimer l'objet se trouvant à un indice donné avec **removeInd** et remplacer un objet par un autre à un indice avec replace. Si l'objet dont on cherche l'indice n'est pas présent dans le tableau, la méthode index lève l'exception **ObjectNotFound** qui hérite de l'exception **RarrayError**. Les trois autres méthodes lèvent l'exception **OutOfRarray** également héritée de **RarrayError** si l'indice passé en paramètre est hors du tableau ou si aucun élément n'est stocké à cet indice.

```
public class Rarray {
...
public int index(Object elt) throws ObjectNotFound { }
public Object get(int ind) throws OutOfRarray { }
public Object removeInd(int ind) throws OutOfRarray { }
public Object replace(int ind, Object elt) throws OutOfRarray { }
}
```

2. Complétez vos tests pour ces méthodes, puis exécutez de nouveau vos tests sur les implantations fournies.

- Editez le rapport de test avec les nouvelles erreurs trouvées :
    * en utilisant la documentation sous eclipse (java.doc de jdk) .
    * en utilisant l'outil jacoco si possible.

Chapitre 3 :

Tests en boite noir (fonctionnel)

Plan de chapitre

1. Introduction

Les **tests de logiciels** sont un domaine incroyablement complexe et intensif. Les entreprises et les développeurs indépendants cherchent tous à améliorer leurs produits à l'aide d'une série de méthodes de test.

L'une des méthodes les plus courantes utilisées par les entreprises pour effectuer des tests est le test de la boîte noire, une technique qui crée une **distance** entre les développeurs et les testeurs afin de fournir des résultats **précis** et **d'éliminer** les préjugés.

Ce chapitre explique ce qu'est un test boîte noire, comment réaliser un test boîte noire et quels sont les avantages de la mise en œuvre d'un test boîte noire dans l'ingénierie logicielle.

2. Objectif et importance

L'objectif des tests de la boîte noire est de vérifier que **le système fonctionne** comme prévu pour l'utilisateur final, tandis que l'objectif des tests de la boîte blanche est de vérifier la **qualité et l'intégrité du code** du logiciel.

Les tests fonctionnels examinent tous les aspects **comportementaux** de l'application, les tests non fonctionnels garantissent des **performances** correctes dans une gamme de conditions d'utilisation.

Les **tests fonctionnels** s'occupent de la **façon** dont les programmes sont exécutés dans une application, les tests non fonctionnels examinent comment l'application **fonctionne** dans un environnement réel. Ce type de test prend en compte des aspects tels que **la vitesse, la fiabilité, l'évolutivité, les performances et la convivialité.**

Ainsi, les tests non fonctionnels sont tout aussi critiques lorsqu'il s'agit de répondre aux **exigences** de l'utilisateur **final**, car non seulement les applications doivent "**fonctionner**", mais elles doivent également "**performer**".

Les **tests fonctionnels se trouvent dans les applications de types :**

Application composée du plusieurs **entités communicantes**

Système **embarqué**

Système à base de **composants**

Protocole…..Peut être **temporisé….**

Transistbor

Algorithmes

Réseau comme internet.

L'importance de test fonctionnel se trouve pour :

a. Test de charge

Les tests de charge valident que l'application **répond** comme requis même lorsqu'un **grand** nombre d'utilisateurs simultanés y accèdent **simultanément**, comme dans des situations réelles. Elle est généralement effectuée sur des serveurs dédiés qui simulent des environnements d'utilisation réels.

b. Tests de résistance

Les tests de résistance évaluent les performances des applications dans des situations critiques, par exemple, dans des conditions d'espace mémoire/disque dur insuffisant. Dans de tels environnements, il est possible de détecter des défauts qui n'auraient pas été découverts dans des situations normales.

c. Tests de récupération

Cela vérifie si les applications récupèrent correctement lorsque les entrées ne sont pas comme prévu ou lorsque l'environnement échoue. Par exemple, lorsqu'un utilisateur tape une entrée non valide qui provoque l'abandon d'un processus de base de données, ou lorsque les systèmes s'arrêtent anormalement en raison d'une panne de courant, etc.

d. Tests de sécurité

Cela vérifie simplement si une application ne présente pas de failles ou de vulnérabilités pouvant être exploitées pour compromettre le système et entraîner une perte de données ou un vol. Il se concentre sur les tests d'authentification, de contrôle d'accès, d'autorisation et d'autres processus sensibles.

e. Tests d'évolutivité

Les tests d'évolutivité vérifient si l'application peut gérer une augmentation du trafic utilisateur, du nombre de transactions, de processus ou du volume de données. L'application doit évoluer pour répondre à ces augmentations de la demande.

f. Tests d'endurance

Les tests d'endurance, également connus sous le nom de tests d'immersion, vérifient si l'application peut supporter une charge soutenue sur une longue durée. Généralement, il est utilisé pour tester les fuites de mémoire dans un système.

g. Tests de fiabilité

Cette forme de test peut être utilisée pour vérifier si une application fournit la même sortie de manière cohérente sur une durée spécifiée. Les tests de fiabilité sont extrêmement

vitaux dans les applications critiques telles que les systèmes aéronautiques, les processus des centrales nucléaires et les équipements médicaux, entre autres.

h. Tests de base

Les tests de référence ou de référence font référence à l'établissement d'une norme pour toute nouvelle application testée. Par exemple, une application peut être capable de gérer une charge de 100 000 utilisateurs lors de sa première série de tests, qui devient alors une référence pour les tests futurs.

3. Principe de ne pas connaitre l'implémentation interne

Les tests "boîte noire" désignent le processus de test d'un système ou d'un logiciel **sans** connaissance préalable de son fonctionnement **interne**. Il ne s'agit pas seulement de ne pas connaître le code source lui-même, mais aussi de ne pas avoir vu la documentation relative à la conception du logiciel. Les testeurs se contentent de fournir des données d'entrée et de recevoir des données de sortie, comme le ferait un utilisateur final.

Le principe de ne pas connaitre l'implémentation interne est qui :

- Impossible de détecter les incomplétudes par rapport à la spécification
- Lors d'une modification du programme, il est souvent difficile de réutiliser les jeux de test précédents
- En cas de test à partir du code objet, le GC (Graphe de Control) n'est pas disponible

L'objectif des tests en boîte noire est d'amener les **utilisateurs** à interagir avec le logiciel d'une manière plus naturelle que d'habitude, sans avoir de préjugés découlant d'une connaissance préalable du logiciel.

Dans cette méthodologie, les personnes chargées de réaliser les tests sont différents de celles qui ont développé le logiciel, ce qui crée une séparation entre les deux équipes.

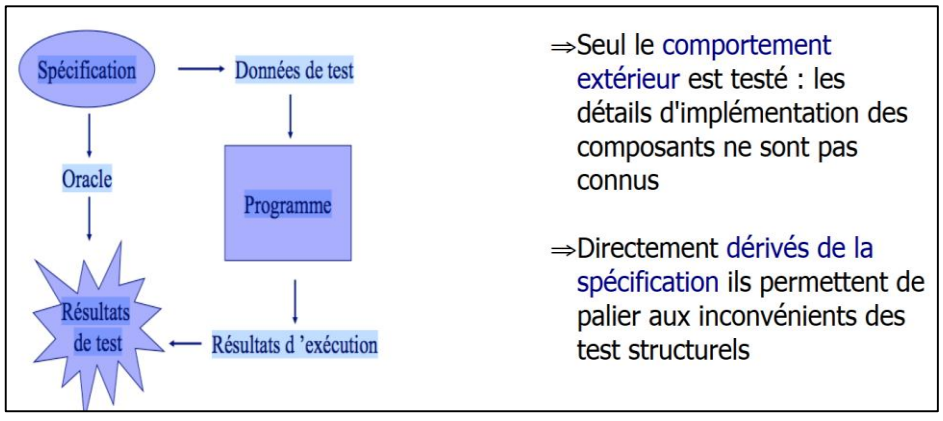


Figure 1: Test de conformité par rapport à la spécification

4. Caractéristiques des tests de la boîte noire

Les tests boîte noire présentent quelques caractéristiques majeures qui les distinguent de toute autre forme d'assurance qualité des logiciels.

4.1 Aucune connaissance interne préalable

Les tests "boîte noire" ne nécessitent aucune connaissance interne préalable du logiciel. Cela peut s'avérer difficile dans certains cas, car les testeurs ont une idée des aspects du logiciel qu'ils testent et de certaines des caractéristiques qu'ils recherchent, mais il s'agit en général de ne pas pouvoir consulter la documentation interne, quelle qu'elle soit.

En d'autres termes, si les informations sont visibles par un utilisateur final dans un magasin d'applications ou sur la page de téléchargement d'un site web, un testeur peut les voir.

4.2 Séparer les testeurs et les développeurs

Les phases de test et de développement sont réalisées par des personnes **différentes** dans une situation de test en boîte noire. Cette différenciation provient du manque de connaissances des testeurs, alors que les développeurs connaissent le code source car ce sont eux qui l'ont développé.

Les entreprises abordent cette question de différentes manières en fonction de leur situation spécifique, certaines choisissant de faire appel à une **organisation externe** pour réaliser les tests et d'autres, plus importantes, disposant de départements de testeurs dédiés à cette tâche.

4.3 Essais à un stade avancé

Il s'agit du stade de développement auquel ce test a lieu. Les tests en boîte noire reposent sur une version relativement avancée d'une application existante, avec une interface utilisateur complète qui permet une navigation totale dans le logiciel et l'accès à la partie frontale de chaque fonctionnalité.

Cela signifie que les tests en boîte noire ne sont possibles qu'à certains stades ultérieurs du processus de test, lorsque tous ces éléments ont été initialement développés. Bien que l'**interface utilisateur** et les contrôles puissent être modifiés au fil du temps, ils doivent exister sous une forme ou une autre pour permettre aux tests de la boîte noire d'accéder à la fonctionnalité.

5. Que testons-nous dans les tests “boîte noire”

Les tests “boîte noire” examinent des aspects spécifiques d’un logiciel, fournissant des informations supplémentaires dans certains domaines du logiciel qui conduisent à des mises à jour améliorant la qualité de vie générale.

Voici quelques-unes des principales parties d’un logiciel que les testeurs examinent dans le cadre d’un test de la boîte noire :

5.1 Fonctionnalité

Certains développeurs utilisent les tests de la boîte noire pour s’assurer qu’un logiciel fonctionne comme prévu pour quelqu’un qui n’a pas de connaissances préalables.

La grande majorité des personnes qui utilisent un logiciel à des fins commerciales le font sans en comprendre les rouages. En testant un logiciel en ayant ces connaissances, vous connaissez les solutions de contournement pour les problèmes existants.

Ces **tests de fonctionnalité** approfondis garantissent que tout le monde profite du meilleur de l’application plutôt que de rencontrer des bogues qui n’ont pas été détectés lors des tests en boîte blanche.

5.2 Interface utilisateur

L’interface utilisateur désigne la manière dont l’utilisateur interagit concrètement avec une application pour lui faire accomplir une série de tâches. Il s’agit notamment des menus avec lesquels l’utilisateur travaille, des boutons spécifiques présents dans une application et de la marque présente dans l’ensemble du logiciel.

Les développeurs passent la majeure partie de leur temps à s’assurer que l’application elle-même fonctionne comme ils le souhaitent, ce qui signifie qu’ils accordent moins d’attention à l’interface utilisateur.

Les tests en boîte noire ne présentent aux testeurs que les fonctionnalités du logiciel destinées à l’utilisateur, ce qui permet d’accorder plus d’attention à l’**interface utilisateur** qu’à la plupart des autres étapes du test.

5.3 La performance

En plus de fonctionner normalement et d’être belle, la façon dont une application fonctionne est essentielle pour plaire aux clients.

La **performance** fait référence à plusieurs facteurs, notamment la **vitesse** à laquelle l’application répond aux entrées de l’utilisateur et les **ressources** qu’elle utilise sur un appareil donné.

Grâce à des formats de test tels que les tests de bout en bout, qui examinent toutes les fonctionnalités d'un logiciel, les développeurs peuvent voir quelle **quantité** de mémoire une application utilise et quelles fonctions sollicitent le plus leurs appareils respectifs, ce qui permet d'orienter les mises à jour relatives à l'efficacité et aux performances dans les versions ultérieures de l'application.

6. Techniques de tests fonctionnels

Les tests en « boite noire » consistent à examiner uniquement les fonctionnalités d'une application, c'est-à-dire si elle fait ce qu'elle est censée faire, peu importe comment elle le fait. Sa structure et son fonctionnement interne ne sont pas étudiés. Le testeur doit donc savoir quel est le **rôle** du système et de ses fonctionnalités, mais ignore ses mécanismes internes. Il a un profil uniquement « utilisateur ».

Cette méthode sert à vérifier, après la finalisation d'un projet, si un logiciel ou une application fonctionne bien et sert efficacement ses utilisateurs. En général, les testeurs sont à la recherche de fonctions incorrectes ou manquantes, d'erreurs d'interface, de performance, d'initialisation et de fin de programme, ou bien encore d'erreurs dans les structures de données ou d'accès aux bases de données externes.

Les tests fonctionnels ou boîte opaque reposent sur une spécification du programme. Le code source du programme n'est pas utilisé. Les tests fonctionnels permettent d'écrire les tests bien avant le «codage ». Il est parfois utile de combiner ces deux méthodes.

Le test de la boîte noire est également connu sous différents noms : Tests basés sur les spécifications. Test comportemental. Tests basés sur les données. Test d'entrée-sortie. Test black box (traduction en anglais).

La méthode de test fonctionnel repose sur six principes : -Partitionnement en classes d'équivalence. -Analyse de la valeur limite. -Test catégorie-partition. -Tables de décision. -Graphes de cause-à-effet. -Fonctions logique

6.1 Partitionnement en classes d'équivalence :

La technique des partitions d'équivalence est utilisée pour réduire le nombre de cas de test nécessaire pour tester un ensemble d'entrées, de sorties, d'intervalles de valeurs ou de temps.

Le découpage en partitions est utilisé pour créer des classes d'équivalence (souvent appelées partitions d'équivalence) qui sont constituées d'ensembles de valeurs qui sont traités de la même manière.

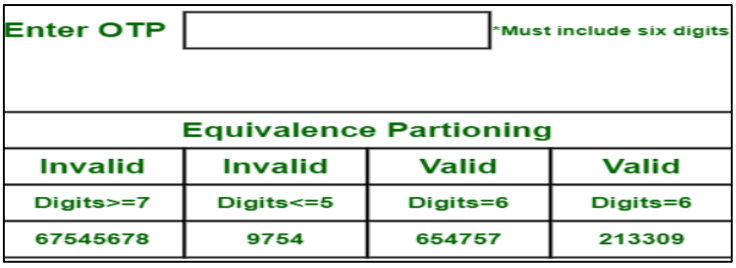
En sélectionnant dans chaque partition une valeur représentative, la couverture de tous les éléments de la même partition est assurée.

Mais les classes d'équivalence doivent être choisies prudemment … Estimer le comportement probable du système à tester.

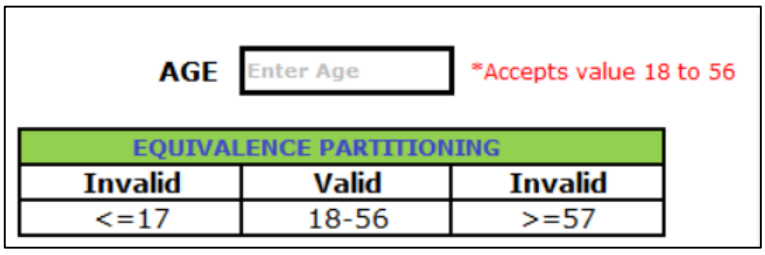
- **Test par classes d'équivalence faible :** choisir une forme de variable d'entrée pour chacune des classes d'équivalences : **max (|A|, |B|, |C|) cas de test**
- **Test par classes d'équivalence fort :** basé sur le produit Cartésien des sous-ensembles de partition (A, B et C), les interactions de classes est calculé comme :

|A| x |B| x |C| cas de test

Exemples 1:



Exemples 2:



Exemple 3 : Soit une fonction **NextDate** avec trois variables : MOIS, JOUR, ANNEE. Elle retourne la date du jour après la donnée date. Limites : 31-12-2040. Sommaire du traitement :

- Si ce n'est pas le dernier jour du mois, la dernière fonction date augmente simplement la valeur **JOUR**.

- À la fin du mois, le jour suivant est **1** et la valeur **MOIS** est augmentée.

- À la fin d'une année, les deux valeurs **JOUR** et **MOIS** sont remises à 1, et la valeur **ANNEE** est **augmentée**.

- Finalement, le problème de l'année **bissextile** rend intéressant la détermination du dernier jour d'un mois.

- NextDate: 36 cas de tests possibles!!!

Cas ID	Mois	Jour	An	Sortie anticipée
SE1	6	14	1900	6/15/1900
SE2	6	14	1912	6/15/1912
SE3	6	14	1913	6/15/1913
SE4	6	29	1900	6/30/1900
SE5	6	29	1912	6/30/1912
SE6	6	29	1913	6/30/1913
SE7	6	30	1900	7/1/1900
SE8	6	30	1912	7/1/1912
SE9	6	30	1913	7/1/1913
SE10	6	31	1900	ERREUR
SE11	6	31	1912	ERREUR
SE12	6	31	1913	ERREUR
SE13	7	14	1900	7/15/1900
SE14	7	14	1912	7/15/1912
SE15	7	14	1913	7/15/1913
SE16	7	29	1900	7/30/1900
SE17	7	29	1912	7/30/1912
SE18	7	29	1913	7/30/1913
SE19	7	30	1900	7/31/1900
SE20	7	30	1912	7/31/1912
SE21	7	30	1913	7/31/1913
SE22	7	31	1900	8/1/1900
SE23	7	31	1912	8/1/1912
SE24	7	31	1913	8/1/1913
SE25	2	14	1900	2/15/1900
SE26	2	14	1912	2/15/1912
SE27	2	14	1913	2/15/1913
SE28	2	29	1900	ERREUR
SE29	2	29	1912	3/1/1912
SE30	2	29	1913	ERREUR
SE31	2	30	1900	ERREUR
SE32	2	30	1912	ERREUR
SE33	2	30	1913	ERREUR
SE34	2	31	1900	ERREUR
SE35	2	31	1912	ERREUR
SE36	2	31	1913	ERREUR

Figure 2 : cas de tests possible

- **Construction de class es d'équivalence**

M1 = { MOIS: MOIS à 30 jours} M2 = { MOIS: MOIS à 31 jours}

M3 = { MOIS: MOIS est Février}

D1 = {JOUR: 1<= JOUR <= 28}

D2 = {JOUR: JOUR = 29} D3 = {JOUR: JOUR = 30} D4 = {JOUR: JOUR = 31}

Y1 = {ANNEE: ANNEE = 1900}

Y2 = {ANNEE: 1812 <= ANNEE <= 2040 et (ANNEE != 1900) et (ANNEE mod 4 = 0)} Y3 = {ANNEE: (1812 <= ANNEE <= 2040 et ANNEE mod 4 != 0)}

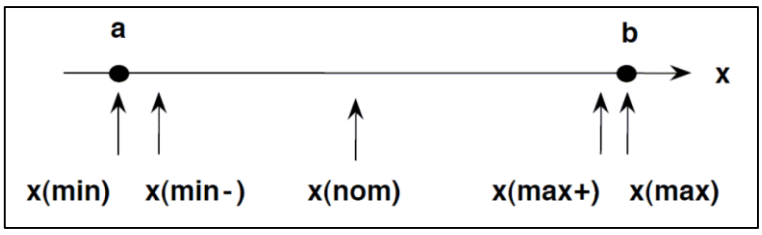
6.2 Analyse de la valeur limite :

Le test des limites est le processus de test entre les **extrémités extrêmes** ou les **limites** entre **les partitions** des valeurs d'entrée.

Ainsi, ces extrémités extrêmes telles que les valeurs Début-Fin, Inférieur-Supérieur, Maximum-Minimum, Juste à l'intérieur-Juste à l'extérieur sont appelées valeurs limites et le test est appelé « test de limite ».

L'idée de base des tests de valeurs limites normales est de sélectionner les valeurs des variables d'entrée à leur niveau :

a. Minimum
b. Juste au-dessus du minimum
c. Une valeur nominale
d. Juste en dessous du maximum
e. Maximum



Remarque : - Dans les tests de limites, le partitionnement des **classes d'équivalence** joue un bon rôle. -Les tests de limites surviennent **après** le partitionnement des classes d'équivalence.

Pourquoi les tests d'équivalence et d'analyse des limites

1. Ces tests sont utilisés pour réduire un très grand nombre de cas de test en morceaux gérables.
2. Des directives très claires sur la détermination des cas de test sans compromettre l'efficacité des tests.
3. Convient aux applications gourmandes en calculs avec un grand nombre de variables/entrées

En pratique, pour des raisons de temps et de budget, il n'est pas possible d'effectuer des tests épuisants pour chaque ensemble de données de test, en particulier lorsqu'il existe un grand nombre de combinaisons d'entrées.

Nous avons besoin d'un moyen simple ou de techniques spéciales permettant de sélectionner intelligemment les cas de test dans le pool de cas de test, de manière à couvrir tous les scénarios

de test. Nous utilisons deux techniques – **Techniques de test de partitionnement d'équivalence et d'analyse des valeurs limites** pour y parvenir

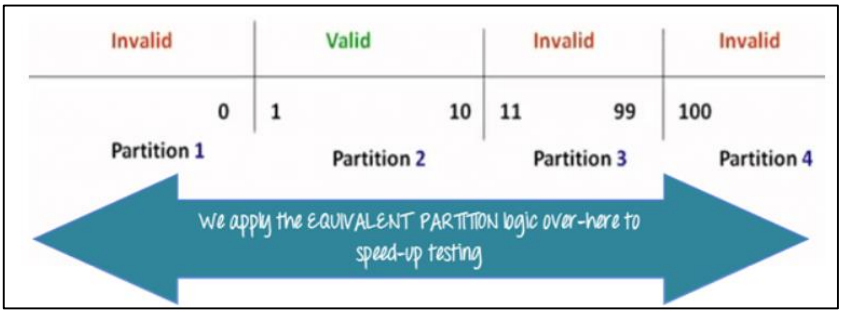
Exemples 1: Considérons le comportement de Order Pizza Text Box Vous trouverez ci-dessous

- Les valeurs de pizza de 1 à 10 sont considérées comme valides. Un message de réussite s'affiche.
- Alors que les valeurs 11 à 99 sont considérées comme invalides pour la commande et qu'un message d'erreur apparaîtra, **"Seules 10 pizzas peuvent être commandées"**

Voici la condition du test

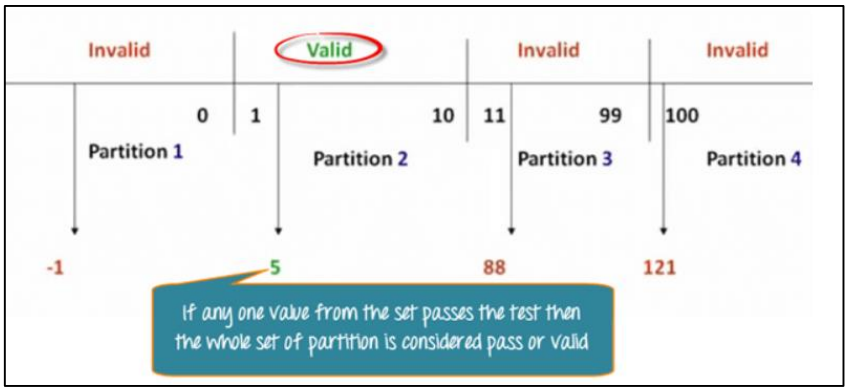
1. Tout nombre supérieur à 10 saisi dans le champ Commander une pizza (disons 11) est considéré comme invalide.
2. Tout nombre inférieur à 1 et égal à 0 ou inférieur est alors considéré comme invalide.
3. Numbers 1 à 10 sont considérés comme valides
4. Tout numéro à 3 chiffres indiquant -100 n'est pas valide.

Nous ne pouvons pas tester toutes les valeurs possibles car si cela est fait, le nombre de cas de test sera supérieur à 100. Pour résoudre ce problème, nous utilisons l'hypothèse de partitionnement d'équivalence où nous divisons les valeurs possibles des tickets en groupes ou ensembles comme indiqué ci-dessous où le système le comportement peut être considéré comme le même.

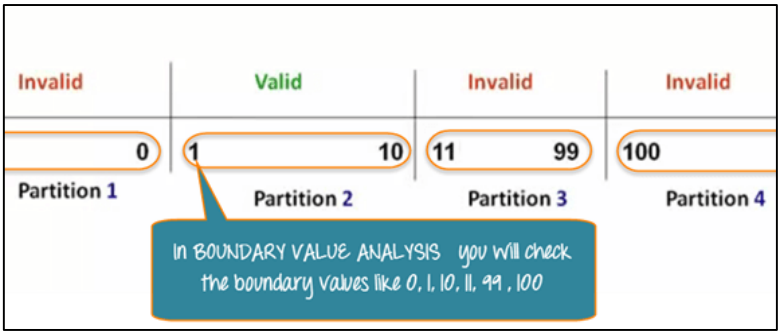


Les ensembles divisés sont appelés partitions d'équivalence ou classes d'équivalence. Ensuite, nous sélectionnons une seule valeur de chaque partition pour les tests. L'hypothèse derrière cette technique est **que si une condition/valeur dans une partition réussit, toutes les autres**

passeront également. Commewise**, si une condition dans une partition échoue, toutes les autres conditions dans cette partition échoueront**.



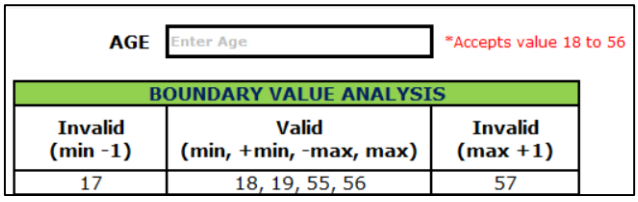
Analyse de la valeur limite– dans Boundary Value Analysis, vous testez les limites entre les partitions d'équivalence



Dans notre exemple précédent de partitionnement d'équivalence, au lieu de vérifier une valeur pour chaque partition, vous vérifierez les valeurs au niveau des partitions comme 0, 1, 10, 11 et ainsi de suite. Comme vous pouvez le constater, vous testez les valeurs à **limites valides et invalides**. L'analyse des valeurs limites est également appelée **vérification de la portée**.

Le partitionnement d'équivalence et l'analyse des valeurs limites (BVA) sont étroitement liés et peuvent être utilisés ensemble. niveaux de test.

Exemple 2: Supposons que nous devons tester un champ qui accepte les âges de 18 à 56 ans.



La valeur limite minimale est de 18

La valeur limite maximale est de 56

Entrées valides : 18,19,55,56

Entrées invalides : 17 et 57

Test case 1: Enter la valeur 17 (18-1) = Invalide

Test case 2: Enter la valeur 18 = Valide

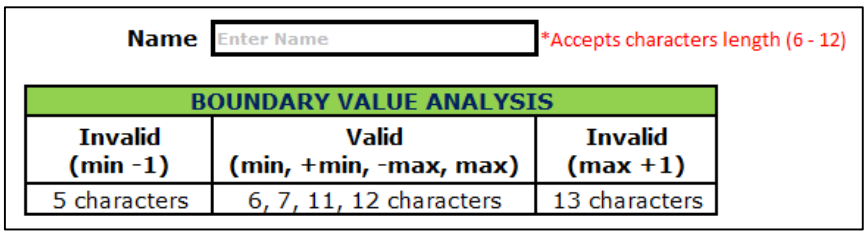
Test case 3: Enter la valeur 19 (18+1) = Valide

Test case 4: Enter la valeur 55 (56-1) = Valide

Test case 5: Enter la valeur 56 = Valide

Test case 6: Enter la valeur 57 (56+1) =Invalide

Exemple 3: Supposons que nous devons tester un champ de texte (Nom) qui accepte une longueur (Text length)comprise entre 6 et 12 caractères.



BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	**Valid (min, +min, -max, max)**	**Invalid (max +1)**
5 characters	6, 7, 11, 12 characters	13 characters

La valeur limite minimale est de 6

La valeur limite maximale est de 12

La longueur de texte valide est 6, 7, 11, 12

La longueur du texte non valide est 5, 13

Test case 1: Text length de 5 (min-1) = Invalide

Test case 2: Text length est exactement 6 (min) = Valide

Test case 3: Text length de 7 (min+1) = Valide

Test case 4: Text length de 11 (max-1) = Valide

Test case 5: Text length est exactement 12 (max) = Valide

Test case 6: Text length de 13 (max+1) = Invalide

Exemple 4: Supposons une fonction F, avec deux variables x1 et x2. Possiblement sans état Limites : a <= x1 <= b, c <= x2 <= d. Dans quelques langages de programmation, un fort typage permet la spécification de tels intervalles. Se concentre sur les **bornes** de l'espace d'entée pour identifier les cas de test. Le raisonnement est que les **erreurs** tendent à se produire **extrêmement** près des valeurs des variables d'entée, ceci est supporté par quelques études.

- **Idées fondamentales :** Les valeurs des variables d'entrée à leur minimum, juste au-dessus du minimum, à une valeur nominale, juste en dessous de leur maximum, et à leur maximum. Convention : min, min+, nom, max-, max. Garde les valeurs de tout sauf une seule variable à leurs valeurs nominales, laissant une seule variable suppose sa valeur extrême.

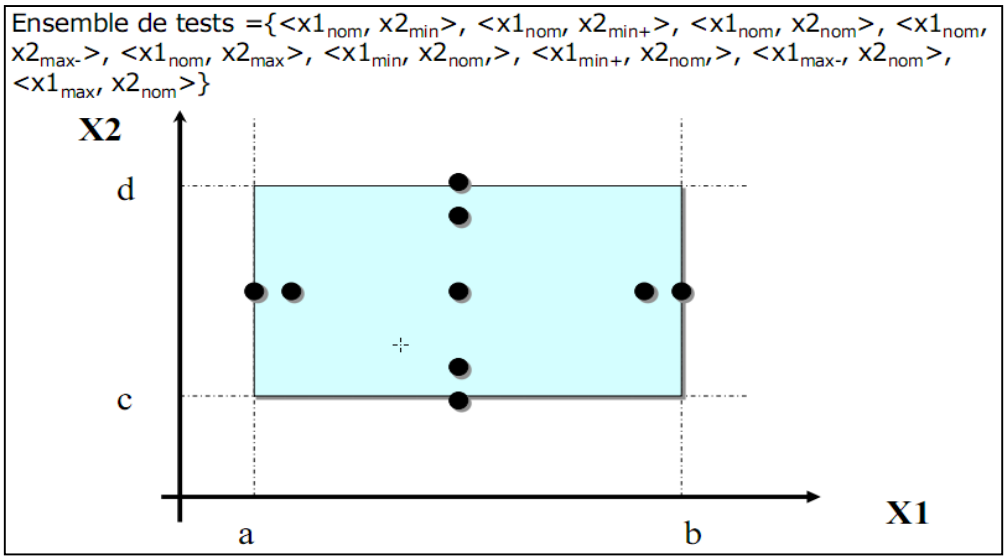


Figure 3 : position des valeurs limites

- **Cas général et limitations** : Une fonction avec n variables nécessitera 4n + 1 cas de tests. Fonctionne bien avec les variables qui représentent des quantités physiques limitées. Sans considération de la nature de la fonction et le sens des variables. Une technique rudimentaire (simple, claire) qui est adéquate au **test de robustesse**.

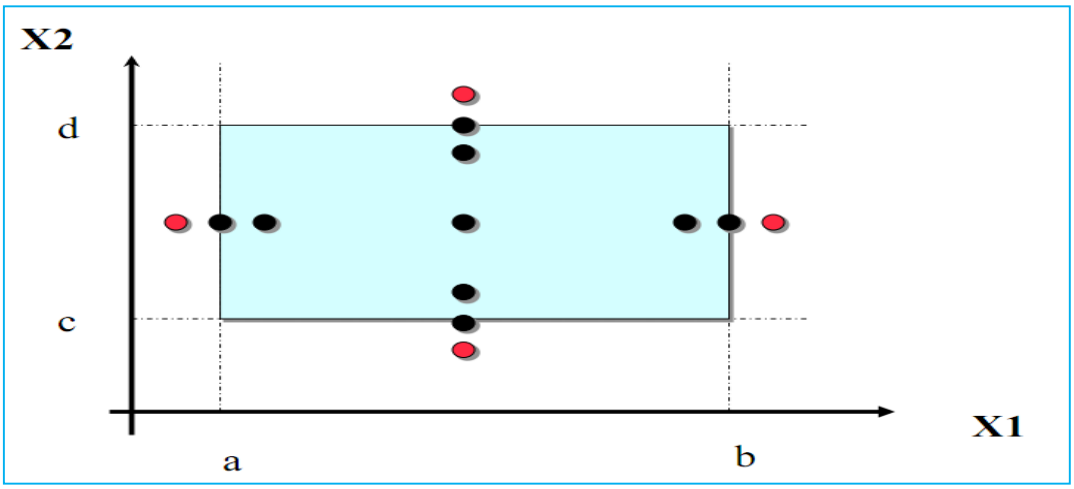


Figure 4 : position de test de robustesse

Remarque :

- Dans le partitionnement d'équivalence, vous divisez d'abord un ensemble de conditions de test en une partition qui peut être prise en compte.
- Dans l'analyse des valeurs limites, vous testez ensuite les limites entre les partitions d'équivalence.
- Convient aux applications gourmandes en calculs avec des variables qui représentent des quantités physiques
- Les tests d'analyse des limites sont utilisés lorsqu'il est pratiquement impossible de tester un grand pool de cas de test individuellement

6.3 Test catégorie- partition :

1. Introduction au Test Catégorie-Partition

Le test catégorie-partition est une méthode de conception de tests qui consiste à diviser les entrées d'un logiciel en **catégories** homogènes, puis à sélectionner un ensemble représentatif de valeurs pour chaque catégorie. L'objectif est de réduire le nombre de tests tout en assurant une couverture optimale des cas possibles.

2. Principe du Test Catégorie-Partition

- Identifier **les paramètres d'entrée** du système à tester.
- Déterminer **les catégories** possibles pour chaque paramètre (par exemple, des plages de valeurs valides et invalides).
- Sélectionner des **valeurs de test** représentatives de chaque catégorie.
- Créer les **cas de test** en combinant les différentes catégories.

Cette approche repose sur l'idée que si un cas fonctionne pour une valeur d'une catégorie, il devrait aussi fonctionner pour toutes les autres valeurs de cette même catégorie.

3. Exemple 1 de Test Catégorie-Partition

Cas d'un formulaire de connexion

Supposons que nous testions un champ de saisie pour un mot de passe, avec les critères suivants :

- Longueur minimale : 8 caractères.
- Longueur maximale : 20 caractères.
- Doit contenir au moins un chiffre.
- Ne doit pas contenir d'espaces.

Étape 1 : Identification des paramètres

- Longueur du mot de passe
- Contient des chiffres

- Contient des espaces

Étape 2 : Définition des catégories

Paramètre	Catégorie Valide	Catégorie Invalide
Longueur du mot de passe	8-20 caractères	<8 ou >20 caractères
Présence de chiffres	Contient un chiffre	Aucun chiffre
Présence d'espaces	Pas d'espace	Contient un espace

Étape 3 : Sélection des valeurs de test

Cas de Test	Mot de Passe	Résultat Attendu
CT1 : Valide	Passw0rd123	Accepté
CT2 : Trop court	Pwd1	Rejeté
CT3 : Trop long	SuperLongPassword12345	Rejeté
CT4 : Sans chiffre	PasswordTest	Rejeté
CT5 : Avec espace	Pass word1	Rejeté

4. Exemple 2 : Test Catégorie-Partition pour un Distributeur de Billets (ATM)

4.1 Contexte

Un distributeur de billets permet aux utilisateurs de retirer de l'argent sous certaines conditions :

- Le montant doit être un multiple de 10 €.
- Le solde du compte doit être suffisant.
- La limite de retrait par jour est de 1000 €.

4.2 Étape 1 : Identification des Paramètres d'Entrée

Les principaux paramètres influençant le retrait sont :

- **Montant demandé**
- **Solde du compte**
- **Plafond de retrait journalier atteint ou non**

4.3 Étape 2 : Définition des Catégories

Paramètre	Catégories Valides	Catégories Invalides
Montant demandé	Multiple de 10€ (ex : 50€)	Non multiple de 10€ (ex : 45€)
Solde du compte	Solde ≥ montant demandé	Solde < montant demandé
Plafond journalier	Montant retiré aujourd'hui < 1000€	Montant retiré aujourd'hui ≥ 1000€

4.4 Étape 3 : Sélection des Valeurs de Test

Cas de Test	Montant demandé	Solde du compte	Plafond journalier	Résultat attendu
CT1 : Retrait valide	50€	500€	500€ retirés	Accepté
CT2 : Montant invalide	45€	500€	500€ retirés	Rejeté (non multiple de 10€)
CT3 : Solde insuffisant	100€	50€	0€ retiré	Rejeté (fonds insuffisants)
CT4 : Plafond atteint	100€	1000€	1000€ retirés	Rejeté (limite dépassée)
CT5 : Cas limite	1000€	2000€	0€ retiré	Accepté

4.5 Analyse des Résultats

Grâce à cette approche, nous avons couvert tous les scénarios critiques (valeurs limites, valeurs invalides et cas standards).

5. Avantages et Inconvénients

Avantages

- Réduction du nombre de tests en évitant la redondance.
- Bonne couverture des cas d'utilisation typiques.
- Facile à comprendre et à appliquer.

Inconvénients

- Nécessite une bonne identification des catégories pertinentes.
- Peut ne pas détecter certains cas extrêmes si les partitions ne sont pas bien définies.

Le test catégorie-partition est une technique efficace pour structurer les tests en fonction des variations des entrées. Il est largement utilisé pour les tests fonctionnels, notamment pour les formulaires, les calculs et les systèmes avec des contraintes d'entrée bien définies.

Ce type de test est couramment utilisé dans les systèmes bancaires, les applications de commerce électronique et les logiciels financiers pour garantir la fiabilité et la robustesse des fonctionnalités.

6.4 Tables de décision (Table de vérité) :

Le principe est de :

* Aide à exprimer les spécifications de test directement dans une forme utilisable.

* Facile à comprendre et supporte la dérivation systématique des tests.

* Supporte la génération des cas de tests automatiques ou manuels.

* Une réponse particulière ou un sous-ensemble de réponse doit être choisie en évaluant plusieurs conditions reliées.

* Idéal pour décrire les situations dans lesquelles un nombre de combinaisons d'actions sont prises selon des ensembles variables de conditions, e.g. systèmes de contrôle.

- **Structure d'une table de décision**

***** La section condition liste les conditions et les combinaisons correspondantes

* La condition exprime l'association entre les variables de décision.

* La section action liste les réponses à être produites quand les combinaisons correspondantes des conditions sont vraies.

* Limitations : les actions résultantes sont déterminées par les valeurs courantes des variables de décision !

* Les actions sont indépendantes de l'ordre de des entrées et de l'ordre dans lequel les conditions sont évaluées.

* Les actions peuvent apparaître plus d'une fois mais chaque combinaison de conditions est unique.

Exemple : Soit a, b, c trois longueurs (entrées) de côté triangulaire :

Son tables de décision ou Table de vérité de type condition /action est présentée comme suite :

conditions											
c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	-	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	-	-	F	T	T	T	T	T	T	T	T
c4: a = b?	-	-	-	T	T	T	T	F	F	F	F
c5: a = c?	-	-	-	T	T	F	F	T	T	F	F
c6: b = c?	-	-	-	T	F	T	F	T	F	T	F
a1: Pas un triangle	X	X	X								
a2: Scalène											X
a3: Isocèles							X		X	X	
a4: Équilatéral				X							
a5: Impossible					X	X		X			

Figure 7 : Table de vérité

- **Tables de décision (Table de vérité) :** c'est l'ensemble de cas de test d'après figure 7

Cas ID	a	b	c	Sortie anticipée
TC1	4	1	2	Pas un triangle
TC 2	1	4	2	Pas un triangle
TC 3	1	2	4	Pas un triangle
TC 4	5	5	5	Équilatéral
TC 5	?	?	?	Impossible
TC 6	?	?	?	Impossible
TC 7	2	2	3	Isocèles
TC 8	?	?	?	Impossible
TC 9	2	3	2	Isocèles
TC 10	3	2	2	Isocèles
TC 11	3	4	5	Scalène

Figure 8 : Cas de test

- **Échelle :** Pour n conditions, il y a peut-être au plus **2^n variantes** (combinaisons uniques de conditions et actions). Mais, heureusement, il y a d'habitude beaucoup moins de variantes explicites … Les valeurs "Don't care" dans les tables de décision aide à réduire le nombre de variantes. "Don't care" peut correspondre à plusieurs cas : les données sont nécessaires mais n'ont pas d'effet; les données peuvent être omises; les cas mutuellement exclusifs (exclusions type-sécurité).

6.5 Graphes de cause-à-effet :

Une technique graphique qui aide à dériver les tables de décision. Vise à supporter l'interaction avec les experts de domaine et l'ingénierie inverse des spécifications, dans le but de tester. Identifie les causes (conditions de entrées, impulsions) et les effets (sorties, changements dans l'état du système). Les causes doivent être formulées d'une manière pour être soit vraies ou fausses (expression booléenne). Précise explicitement les contraintes (environnementales, externes) des causes et des effets. Aide à sélectionner des sous-ensembles de combinaisons de sous- ensembles "significatifs" d'entrées-sorties et à construire de plus petites tables de décision.

- **Structure des graphes de cause-effect :** Un nœud est tiré pour chaque cause et chaque effet. Les nœuds sont placés sur les côtés opposés de la feuille. Une ligne de la cause à l'effet indique que la cause est une condition nécessaire pour l'effet. Si un simple effet a deux causes ou plus, le rapport logique des causes est annoté par des symboles pour un et logique (^) et ou logique (+). Une cause dont la négation est nécessaire est désignée par un non logique (~). Une cause simple peut être nécessaire pour plusieurs effets ; un effet simple peut avoir plusieurs causes nécessaires. Les nœuds intermédiaires peuvent être utilisés pour simplifier le graphe et sa construction.

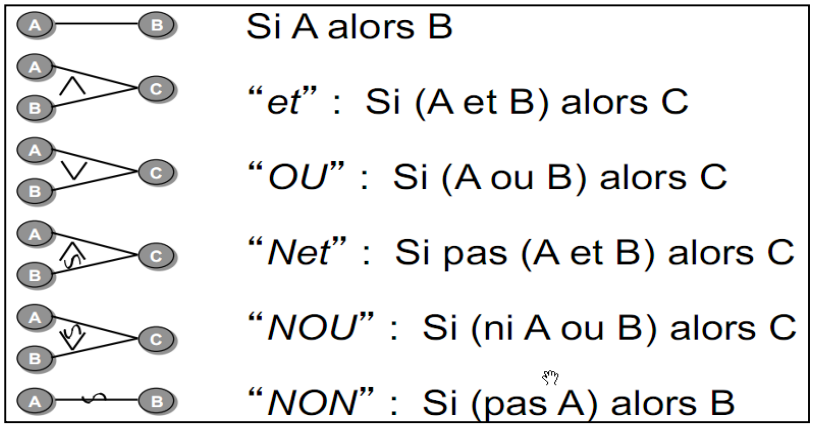


Figure 9 : nœuds graphes de cause-effect

Exemple : prenant le processus de renouvellement d'assurance, son graphe est le suivant :

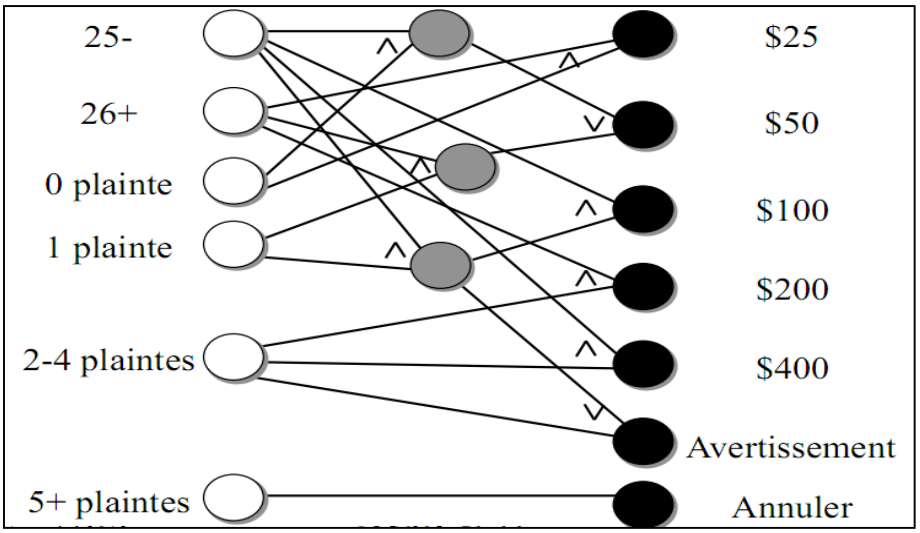


Figure 10 : graphes cause-effect d'assurance

- **Spécification** : d'après le graphe causse-effet, on peut déduire sa table de décision :

	Section condition		Section action		
Variable	**Plaintes**	**Âge**	**Prime augmentée $**	**Envoie d'un avertissement**	**Annulation**
1	0	25-	50	Non	Non
2	0	26+	25	Non	Non
3	1	25-	100	Oui	Non
4	1	26+	50	Non	Non
5	2 à 4	25-	400	Oui	Non
6	2 à 4	26+	200	Oui	Non
7	5+	Tout	0	Non	Oui

Figure 11 : table de décision d'assurance

- **Dériver une table de décision** : Une ligne pour chaque cause ou effet. Les colonnes correspondent aux cas de tests (variables). Nous définissons les colonnes en examinant chaque effet et en listant toutes les combinaisons (conjonctions) des causes qui peuvent mener à cet effet. E.g., deux lignes séparées mènent vers l'effet E3, chacune correspondant à un cas de test, quatre lignes mènent vers E1 mais ne correspondent qu'à deux combinaisons seulement.
- **Discussion** : Le graphe cause-effet peut être utilisé pour générer toutes les combinaisons possibles de causes et vérifier si l'effet correspond à la spécification. Il fournit un test oracle et spécifie les contraintes sur les sorties (effets), aidant à détecter les mauvais états du

système et les combinaisons d'action. Si le graphe est trop large, pour chaque combinaison admissible d'effets, on trouve quelques combinaisons de causes qui déclenchent les combinaisons d'effets en remontant vers le graphe. À cause de contraintes additionnelles sur le graphe, on peut être plus restrictif que les tables de décision classiques.

6.6 Fonctions logiques :

Un prédicat est une expression qui évalue une valeur Booléenne. Les prédicats peuvent contenir des variables booléennes, des variables non booléennes qui sont comparées avec les opérateurs comparateurs {>, <, =, …}, et les appels de fonction (retournent une valeur booléenne). La structure interne du prédicat est créée par les opérateurs logiques {non, et, ou, …}. Une clause est un prédicat qui ne contient aucun des opérateurs logiques, e.g., (a<b). Les prédicats peuvent être écrits de différentes manières logiques équivalentes (algèbre booléenne). Une fonction logique relie n variables d'entrées booléennes (clauses) à une seule variable de sortie booléenne. Pour rendre les expressions plus simple à lire, nous utiliserons la contiguïté (AB) pour l'opérateur et, + pour (A+B) l'opérateur ou et un ~ pour l'opérateur de négation.

Exemple 1 : La chaudière : mettre en état ou hors état l'ignition d'une chaudière se basant sur quatre variables d'entrée :

NormalPressure(A): la pression dans une limite d'opération sécuritaire ?

CallForHeat (B) : la température ambiante sous le point de consigne ?

DamperShut (C) : le conduit du tuyau d'échappement est fermé ?

ManualMode (D) : sélection du manuel d'opération ?

Fonction logique : **Z = A(B~C+D)** ~> **Table de vérité :**

Numéro du vecteur d' entrées	Normalpressure	CallForHeat	DamperShut	ManualMode	Ignition
	A	B	C	D	Z
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0

Numéro du vecteur d'enrées	NormalPressure	CallForHeat	DamperShut	ManualMode	Ignition
	A	B	C	D	Z
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	1

Figure 12 : Table de vérité de la chaudière

Le test en boîte noire des **fonctions logiques** consiste à tester un système ou une application en se basant sur les règles logiques qui régissent son comportement, sans examiner le code source. Ce type de test est utile lorsque le système prend des décisions basées sur des conditions booléennes (Vrai/Faux), des combinaisons d'entrées ou des tables de vérité.

Exemple 2: Système de Contrôle d'Accès

Contexte

Un système permet l'accès à un bâtiment sous les conditions suivantes :

- Un utilisateur peut entrer si **(il possède un badge valide ET connaît le code PIN)** OU **(il est administrateur)**.

Entrée	**Badge valide ?**	**Code PIN correct ?**	**Administrateur ?**	**Accès autorisé ?**
CT1	✗	✗	✗	✗ (Accès refusé)
CT2	✓	✗	✗	✗ (Accès refusé)
CT3	✓	✓	✗	✓ (Accès autorisé)
CT4	✗	✗	✓	✓ (Accès autorisé)
CT5	✓	✓	✓	✓ (Accès autorisé)

Explication

- **Cas CT1** : Aucune condition n'est remplie → Accès refusé.
- **Cas CT2** : Badge valide seul ne suffit pas → Accès refusé.
- **Cas CT3** : Badge valide + Code PIN correct → Accès autorisé.
- **Cas CT4** : Administrateur peut entrer sans badge ni code PIN → Accès autorisé.
- **Cas CT5** : Toutes les conditions sont remplies → Accès autorisé.

Exemple 3 : Test en Boîte Noire – Fonctions Logiques dans un Système de Remise Commerciale

1.Contexte

Une boutique en ligne applique une **remise spéciale** sous certaines conditions :

- Si le client est un **membre VIP**, il bénéficie **automatiquement de 20% de remise**.
- Sinon, il peut bénéficier d'une remise de **10%** s'il **achète pour plus de 100€**.
- Aucune remise ne s'applique si aucune de ces conditions n'est remplie.

2. Définition des Entrées et des Résultats Attendus

Cas de Test	Membre VIP ?	Montant Achat > 100€ ?	Remise Appliquée
CT1	✗	✗	0%
CT2	✗	✓	10%
CT3	✓	✗	20%
CT4	✓	✓	20%

3. Explication

- **CT1** : Le client **n'est pas VIP** et n'atteint pas 100€ → **0% de remise**.
- **CT2** : Le client **n'est pas VIP**, mais il atteint 100€ → **10% de remise**.
- **CT3** : Le client **est VIP**, mais son achat est inférieur à 100€ → **20% de remise** (car VIP).
- **CT4** : Le client **est VIP** et son achat dépasse 100€ → **20% de remise** (la remise VIP est prioritaire).

Avantages de ce Test

- **Valide la logique des remises** sans regarder le code source.
- **Assure une couverture complète** des cas possibles.
- **Facile à représenter avec une table de vérité** pour simplifier l'analyse.

Ce test en boîte noire permet de valider la logique du système **sans connaître l'implémentation interne**.

Les tests en boîte noire sur les **fonctions logiques** s'appuient sur des **tables de décision**, des **matrices de vérité** ou des **arbres de décision** pour vérifier le bon fonctionnement du système face aux différentes combinaisons d'entrées.

Ce type de test est **idéal pour les règles métier**, comme les systèmes de tarification, les droits d'accès ou les calculs financiers.

7. Les meilleurs outils de test en boîte noire

Les tests en boîte noire sont une forme de test qui peut s'appuyer de manière significative sur des outils à portée de main, à la fois pour automatiser vos tests en boîte noire et pour organiser les informations que vous obtenez à partir de vos tests.

L'utilisation de la bonne combinaison d'outils peut vous aider, vous et votre équipe, à travailler de manière beaucoup plus efficace et à mettre en place des processus plus performants dans l'ensemble du département d'assurance qualité. Meilleurs outils pour les tests en boîte noire

1. Selenium

- Type : Test fonctionnel et test UI.
- Description : Un outil open-source pour l'automatisation des tests web.
- Avantages : Prise en charge de plusieurs langages (Java, Python, C#), compatibilité multi-navigateurs.
- Inconvénients : Configuration complexe et maintenance des scripts.

2. JMeter

- Type : Test de performance.
- Description : Outil de test de charge open-source utilisé pour mesurer la performance des applications.
- Avantages : Facile à configurer, puissant pour les tests de charge.
- Inconvénients : Courbe d'apprentissage initiale.

3. Appium

- Type : Test mobile (Android et iOS).
- Description : Automatisation des tests d'applications mobiles.
- Avantages : Support des tests natifs et hybrides, compatibilité avec Selenium.
- Inconvénients : Configuration initiale complexe.

4. Postman

- Type : Test d'API.

- Description : Outil populaire pour tester les API REST et SOAP.
- Avantages : Interface intuitive, automatisation possible.
- Inconvénients : Limitation des tests de charge.

5. TestComplete

- Type : Test fonctionnel et UI.
- Description : Outil de test automatisé supportant diverses technologies.
- Avantages : Interface utilisateur simple, support multi-technologies.
- Inconvénients : Coût élevé.

6. Katalon Studio

- Type : Test fonctionnel et UI.
- Description : Solution tout-en-un pour les tests web, mobiles et API.
- Avantages : Facile à utiliser, intégration avec divers outils DevOps.
- Inconvénients : Moins flexible pour des cas complexes.

7. Cypress

- Type : Test UI.
- Description : Outil moderne pour l'automatisation des tests web.
- Avantages : Rapidité d'exécution, facile à configurer.
- Inconvénients : Support limité aux navigateurs Chromium.

8. Lighthouse

- Type : Test de performance et accessibilité.
- Description : Outil de Google pour auditer les performances web.
- Avantages : Facile à utiliser, recommandations d'optimisation.
- Inconvénients : Limité aux applications web.

Le choix de l'outil dépend du type de test requis et du contexte du projet. Pour les tests UI et fonctionnels, Selenium et Cypress sont recommandés, tandis que JMeter est idéal pour les tests de performance. L'usage combiné de plusieurs outils permet d'assurer une couverture de test optimale.

8. Les avantages de test fonctionnel

- **Simplicité** : ces tests sont simples à réaliser, car on se concentre sur les entrées et les résultats. Le testeur n'a pas besoin d'apprendre à connaître le fonctionnement interne du système ou son code source, qui n'est pas accessible. Cette méthode est donc également non intrusive.
- **Rapidité :** en raison du peu de connaissances nécessaires sur le système, le temps de préparation des tests est très court. Les scénarios sont relativement rapides à créer et à tester, puisqu'ils suivent les chemins utilisateurs, qui sont relativement peu nombreux selon la taille du système.
- **Impartialité :** on est ici dans une optique « utilisateur » et non « développeur ». Les résultats du test sont impartiaux : le système marche, ou il ne marche pas. Il n'y a pas de contestation possible, comme par exemple sur l'utilisation de tel processus plutôt qu'un autre selon l'opinion du développeur.

- **Pas besoin de connaissances techniques :** Une approche "boîte noire" signifie que vous n'avez pas besoin de connaissances techniques pour examiner une application.

  L'objectif des tests en boîte noire est d'examiner le fonctionnement de l'application pour un utilisateur final, et l'utilisateur standard n'a pas de connaissances techniques avancées dans la plupart des cas. Cela peut réduire le coût des tests et aider l'organisation à découvrir plus de bogues à moindre coût, ce qui la rend plus efficace sur le plan financier.
- **Modéliser précisément l'utilisateur :** L'objectif final d'un processus de test en boîte noire est de comprendre quels sont les problèmes d'une application lorsqu'un utilisateur interagit avec elle au quotidien.

  Certains types de tests "boîte noire" – qui visent à reproduire le comportement d'un utilisateur – modélisent le comportement d'un utilisateur avec une grande précision. C'est notamment le cas pour les essais d'acceptation par l'utilisateur, au cours desquels les utilisateurs finaux expérimentent le produit, sans se contenter de modéliser ou de simuler le comportement d'un utilisateur, mais en le mettant réellement en œuvre.

  Une modélisation précise permet de mettre en évidence les bogues qui affectent les flux de travail réels de l'utilisateur.
- **Possibilité d'externaliser les tests :** Les tests en boîte noire sont une forme de test très accessible grâce aux compétences relativement faibles qu'ils requièrent.

  Cela signifie que non seulement les entreprises peuvent embaucher des testeurs ayant un niveau de compétences techniques moins élevé, mais qu'elles peuvent aussi confier leurs tests à des clients enthousiastes. Cette pratique est de plus en plus courante dans l'industrie du

jeu, les entreprises proposant des versions en accès anticipé et mettant à jour le jeu au fil du temps pour résoudre les problèmes rencontrés par les utilisateurs.

Dans ce cas, il est beaucoup plus facile de trouver des bogues, car toutes les fonctionnalités sont beaucoup plus exposées

9. Les défis des tests en boîte noire

Outre les avantages des tests en boîte noire, il existe quelques défis majeurs que vous devrez prendre en compte. En étant conscient de ces défis, vous pouvez vous y adapter et améliorer la qualité de vos tests en réduisant les effets néfastes que peuvent avoir les tests en boîte noire.

Voici quelques-uns de ces défis :

9.1 Difficulté à trouver les causes du problème

L'un des principaux inconvénients des tests en boîte noire est qu'il peut être plus difficile de trouver la cause des problèmes lorsque les testeurs n'ont pas accès au code source. S'ils peuvent décrire l'erreur et le moment où elle se produit, ils ne savent pas quel élément du code source est à l'origine du problème ni pourquoi.

Les testeurs peuvent atténuer quelque peu ce problème en prenant des notes minutieuses, les messages d'erreur détaillés du développeur offrant également des informations supplémentaires pour les futures mises à jour.

9.2 L'automatisation est plus délicate

Comme vous cherchez activement à reproduire la manière dont un utilisateur interagit avec un logiciel, il peut être extrêmement difficile d'automatiser un processus de test en boîte noire.

La première cause est le fait que le testeur n'a pas accès au code source, ce qui rend plus difficile l'élaboration d'un scénario de test précis. Cela va de pair avec le fait que les tests sont conçus pour reproduire autant que possible le comportement humain, l'automatisation étant spécifiquement conçue pour agir de manière **robotique**.

Vous pouvez équilibrer ce problème en automatisant les tâches les moins importantes et en combinant l'automatisation avec des tests manuels lorsque c'est possible.

9.3 Difficultés liées aux tests à grande échelle

Les difficultés susmentionnées liées à l'automatisation signifient que les tests à plus grande échelle sont plus compliqués. Les tests à grande échelle fournissent aux entreprises beaucoup plus de données sur le logiciel et signifient que les bogues sont plus faciles à trouver et à reproduire.

L'exigence de tests manuels en priorité signifie qu'il peut être plus difficile d'organiser des tests à grande échelle. Certaines entreprises y remédient en utilisant un système de "bêta

ouverte", dans lequel toute personne intéressée par le produit peut participer aux tests de préversion et soutenir l'entreprise en donnant son avis sur les premières versions, sur une base volontaire.

10. Inconvénients des tests en boîte noire

- **Superficialité :** étant donné que le code n'est pas étudié, ces tests ne permettent pas de voir, en cas de problème, quelles parties précises du code sont en cause. De plus, les testeurs peuvent passer à côté de problèmes ou vulnérabilités sous-jacentes. Certains problèmes sont également difficilement repérables avec cette méthode, comme par exemple ceux liés à la cryptographie, ou à des aléas (Générateur de nombres aléatoires, Random Number Generator (RNG) de mauvaise qualité. C'est donc l'un des tests les moins exhaustifs.
- **Redondance** : si d'autres tests sont effectués, il est possible que celui-ci perde grandement de son intérêt, puisque son champ d'action a tendance à être inclus dans celui d'autres tests.

11. Test Boite Gris (Gray Box Testing)

Le test boîte grise est une technique de test logiciel qui **combine** les approches du test boîte noire et du test boîte blanche. Il permet aux testeurs d'avoir une compréhension partielle du système interne tout en évaluant ses fonctionnalités externes.

11.1Objectifs du Test Boîte Grise

- Détecter les défauts liés aux structures internes et à la logique métier.
- Vérifier le bon fonctionnement du logiciel en tenant compte des flux de données et des interactions entre les modules.
- Optimiser la couverture des tests en utilisant des informations sur l'architecture et le code source.

11.2 Différence entre Boîte Noire, Boîte Blanche et Boîte Grise

Critère	Test Boîte Noire	Test Boîte Blanche	Test Boîte Grise
Connaissance du code	Aucune	Complète	Partielle
Objectif	Tester les fonctionnalités	Tester l'implémentation	Tester les interactions internes et externes
Qui effectue le test ?	Testeurs	Développeurs	Testeurs expérimentés ou développeurs

11.3 Méthodologie du Test Boîte Grise

a. **Compréhension du Système** : Analyse des spécifications et des documents de conception.
b. **Identification des Cas de Test** : Basé sur les diagrammes de flux de données, les modèles UML, etc.
c. **Exécution des Tests** : Utilisation d'outils automatisés ou manuels pour tester les interactions internes et externes.
d. **Analyse des Résultats** : Vérification des écarts entre les résultats attendus et les résultats réels.
e. **Rapport des Anomalies** : Documentation et remontée des bugs pour correction.

11.4Techniques Utilisées en Test Boîte Grise

- **Test basé sur les matrices de décision** : Vérification des combinaisons d'entrées et de sorties.
- **Analyse des flux de données** : Test des chemins critiques et des points d'interaction entre les modules.
- **Test des conditions limites** : Vérification du comportement du système dans des situations extrêmes.

11.3 Exemples

Exemple 1 de Test Boîte Grise : Validation d'une API Bancaire

1. Contexte

Un testeur doit valider une **API de transfert d'argent** entre comptes bancaires. Il dispose de :

- La **documentation** décrivant les entrées et sorties de l'API.
- Un accès limité au **journal des transactions** pour vérifier les mises à jour.
- Aucune visibilité sur l'implémentation interne du code.

2. Scénario de Test

L'API permet un transfert **si et seulement si** :

1. Le solde du compte expéditeur est **suffisant**.
2. Le montant du transfert est **positif** et ne dépasse pas 5000 €.
3. Le compte destinataire **existe**.

3. Étape 1 : Analyse des Entrées et Sorties de l'API

L'API attend :

- **ID du compte expéditeur**
- **ID du compte destinataire**
- **Montant du transfert**

Elle retourne :

- ✓ **Succès** (transaction validée)
- ✗ **Erreur** (motif de rejet)

4. Étape 2 : Définition des Cas de Test

Cas de Test	Solde Expéditeur	Montant du Transfert	Compte Destinataire Valide ?	Résultat Attendu
CT1 : Transfert valide	6000€	1000€	✓	✓ Succès
CT2 : Solde insuffisant	300€	1000€	✓	✗ Erreur (fonds insuffisants)
CT3 : Montant négatif	5000€	-100€	✓	✗ Erreur (montant invalide)
CT4 : Montant trop élevé	4000€	6000€	✓	✗ Erreur (dépassement de limite)
CT5 : Compte inexistant	7000€	2000€	✗	✗ Erreur (compte invalide)

5. Étape 3 : Vérification des Résultats

Le testeur :

- **Envoie les requêtes** API avec les différentes combinaisons.
- **Vérifie les réponses** de l'API.
- **Consulte les logs** pour voir si la transaction a bien été enregistrée.

Exemple 2 de Test Boîte Grise : Authentification à Deux Facteurs (2FA)

1. Contexte

Une entreprise souhaite tester le fonctionnement de son système **d'authentification à deux facteurs (2FA)**. L'utilisateur doit :

1. **Saisir ses identifiants** (email + mot de passe).
2. **Recevoir un code OTP** par SMS ou email.
3. **Saisir le code OTP** pour accéder à son compte.

Le testeur a accès à :

- **L'interface utilisateur** (comme un test en boîte noire).
- **Les logs serveur** pour voir si le code OTP est bien généré et envoyé (comme un test en boîte blanche).

2. Scénario de Test

Cas de Test	Email et Mot de Passe	Réception du Code OTP	Code OTP Saisi	Résultat Attendu
CT1 : Authentification réussie	Valides	Oui	Correct	✅ Connexion acceptée
CT2 : Mot de passe incorrect	Incorrect	Non généré	-	❌ Échec (Mauvais identifiants)
CT3 : OTP non reçu	Valides	Non reçu	-	❌ Échec (OTP non reçu)
CT4 : OTP erroné	Valides	Oui	Incorrect	❌ Échec (OTP invalide)
CT5 : Expiration OTP	Valides	Oui mais expiré	Expiré	❌ Échec (OTP expiré)
CT6 : Tentatives multiples	Valides	Oui	3 erreurs consécutives	❌ Compte verrouillé

3. Étape 1 : Exécution des Tests

1. **Test boîte noire**
   - Se connecter avec différents comptes et observer les réactions du système.
   - Tester les scénarios d'échec et de succès.
2. **Test boîte blanche partiel**
   - Vérifier dans **les logs** si le code OTP est généré et envoyé correctement.
   - Contrôler si **le délai d'expiration est bien respecté**.
   - Surveiller si un utilisateur bloqué est bien empêché de se reconnecter.

4. Pourquoi est-ce un Test Boîte Grise ?

- Le testeur **ne voit pas directement le code source**, mais peut **analyser les logs et bases de données**.
- Il teste à la fois **les fonctionnalités visibles** et **les processus internes**.

- Il permet de vérifier **les interactions entre plusieurs systèmes** (base de données, envoi OTP, gestion des sessions).

- Le testeur **connaît les règles métiers** et les formats de données.
- Il **ne voit pas le code source**, mais a accès aux **logs** et aux **documents techniques**.
- Il combine **tests fonctionnels (boîte noire)** et **vérifications techniques (boîte blanche)**.

Le test boîte grise est **idéal pour les API, bases de données et systèmes complexes**, car il permet de tester **les interactions internes et externes** du logiciel.

Le test boîte grise est idéal pour **les systèmes de sécurité**, car il permet de détecter :
- Des **failles d'authentification** (contournement de l'OTP, mauvaises expirations).
- Des **problèmes d'intégration** (envoi OTP non fonctionnel).
- Des **erreurs dans les logs** qui aident les développeurs à corriger le système.

11.5 Avantages et Inconvénients

a. Avantages

- Meilleure couverture des tests par rapport au test boîte noire.
- Permet d'identifier des défauts cachés dans les interactions entre les modules.
- Équilibre entre l'efficacité du test boîte noire et la précision du test boîte blanche.

b. Inconvénients

- Nécessite une compréhension partielle du code ou de l'architecture.
- Peut être complexe à mettre en œuvre selon la taille du système testé.
- Dépend souvent de la documentation technique, qui peut être incomplète ou obsolète.

11.6 Outils Utilisés

- **Selenium** (tests d'interfaces et flux de données)
- **JUnit**/**TestNG** (tests unitaires avec accès partiel au code)
- **Postman** (tests des API et interactions entre services)
- White/Grey Box Testing Tools (e.g., **AppScan**, **JMeter**)

11.7 Cas d'Utilisation

- Tests de logiciels embarqués (interaction matériel/logiciel).
- Vérification des API et microservices.
- Tests de sécurité (détection des vulnérabilités dans les flux de données).

Le test boîte grise est une approche hybride qui maximise l'efficacité des tests en utilisant à la fois des techniques de boîte noire et de boîte blanche. Il est particulièrement utile dans les systèmes complexes où une connaissance partielle de l'architecture permet de détecter des défauts plus efficacement.

12. **Etude comparative entre test fonctionnelle et test structurelle**

Test en boîte noire	**Test en boîte blanche**
C'est une manière de tester le logiciel dans laquelle la structure interne du programme ou du code est caché et rien n'est connu à ce sujet.	C'est une façon de tester le logiciel dans lequel le testeur a une connaissance de structure interne du code ou du programme du logiciel.
Cela se fait principalement par des testeurs de logiciels.	C'est principalement fait par les développeurs de logiciels.
Aucune connaissance de l'implémentation n'est nécessaire.	La connaissance de l'implémentation est requise.
Il peut être appelé test externe du logiciel.	C'est le test interne du logiciel.
C'est un test fonctionnel du logiciel.	C'est un test structurel du logiciel.
Ce test peut être lancé sur la base du document de spécification des exigences.	Ce type de test est entamé après le document de conception détaillée.
Aucune connaissance en programmation n'est requise.	Il est obligatoire de connaître la programmation.
C'est le test de comportement du logiciel.	Ce sont les tests logiques du logiciel.
Cela prend moins de temps.	Cela prend beaucoup de temps.

Figure 13 : Etude comparative entre les types de tests.

13. Conclusion

Les **tests fonctionnels et non fonctionnels** sont deux piliers de la famille des tests. Les deux sont tout aussi critiques lorsqu'il s'agit de répondre aux exigences des utilisateurs finaux. Alors que les tests fonctionnels examinent tous les aspects comportementaux de l'application, les tests non fonctionnels garantissent des performances correctes dans une gamme de conditions d'utilisation.

En fin de compte, il existe des différences fondamentales entre les tests de la boîte noire, de la boîte grise et de la boîte blanche, qui dépendent toutes de la manière dont les informations en coulisses sont présentées à l'équipe chargée des tests.

Les tests en boîte noire et en boîte blanche sont les extrêmes de ce spectre, les tests en boîte grise englobant tout, de l'accès à l'ensemble du code source sauf celui d'un tiers à la possibilité de voir uniquement le code d'une fonction spécifique.

Toutes ces méthodes de test ont un rôle à jouer dans le domaine des tests de logiciels. Il est donc indispensable de consacrer du temps et de l'attention à leur apprentissage et à leur mise en œuvre efficace.

Le test de la boîte noire est en fin de compte l'une des parties les plus importantes du processus de test des logiciels. Elle aide les entreprises à s'assurer que ce qu'elles expédient répond aux normes les plus élevées possibles et utilise un changement de perspective pour offrir un aperçu unique de la manière dont une application est perçue et mise en œuvre par un utilisateur externe.

Toute entreprise qui n'ajoute pas à ses processus des tests de boîte noire, à la fois automatisés et manuels, manque une occasion d'améliorer considérablement la qualité de son application. Testez intelligemment et vous récolterez les fruits de vos efforts lorsque vos clients auront accès à votre produit.

Université de Mohamed Khider – Biskra –
Département d'informatique
Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TD N° 3 : Méthode de test en boite noire

Spécification 1 :

Une société vend deux produits A et B au prix unitaire de 5 e pour A et de 10 e pour B. Une commande comprend une certaine quantité du produit A et une certaine quantité du produit B. Le coût d'une commande est la somme totale des prix unitaires des produits commandés, à laquelle on applique une réduction selon les règles suivantes :

Si la somme totale est supérieure ou égale à 200 e, on applique une réduction de 5%, si elle est supérieure ou égale à 1000 e, la réduction est de 20%. Ces deux réductions ne sont pas cumulables et portent sur la somme totale.

La société souhaitant encourager la vente de A, on applique, sur le prix obtenu grâce à la règle précédente, une réduction supplémentaire de 10% si la commande comprend au moins 45 produits A.

Spécification 2 :

La température d'eau peut varie de -40 à +100 Celsius.

- En dessous de 0 degré Celsius, le system met en route le chauffage antigel.
- Au-dessus de 30 degré Celsius, le système met en route le refroidissement.

Spécification 3 :

Dans une université, le système autorise les étudiants à utiliser de l'espace disque en fonction de leurs projets.

- S'ils ont utilisés tout leur espace alloué, ils sont autorisés en accès restreint (ils peuvent lire et supprimer des fichiers mais pas en créer).
- ils doivent toutefois être connectés avec leur identifiant et leur mot de passe

Travail demandé : appliquez les 6 méthodes de test fonctionnel sur les trois spécifications :

1. **Partitionnement en classes d'équivalence**
2. **Analyse de la valeur limite**
3. **Test catégorie- partition**
4. **Tables de décision (Table de vérité)**
5. **Graphes de cause-à-effet**
6. **Fonctions logique**

- Rapporter votre travail.

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TP N° 3 test fonctionnel (black box): Application

Exercice:

On considère une classe **Tableau** permettant de stocker des entiers dans un tableau trié. Le nombre d'éléments du tableau est stocké dans un attribut **taille** et la longueur du tableau dans un attribut **capacite**. La taille est toujours **supérieure ou égale à 0**, la capacité est toujours strictement **positive**, et la taille inférieure ou égale à la capacité. En particulier, on ne peut pas appeler le constructeur de la classe Tableau (int capacite) avec un argument **négatif ou nul**.

```
public class Tableau {
    private int tab[];
    private int taille;
    private int capacite;

    public Tableau(int capacite) { }

    public boolean inserer(int val) { }

    public boolean supprimer(int val) {
        int i = 0;
        while(i < taille && tab[i] != val) {
            i++;
        }
        if(i == taille) {
            return false;
        }
        for(int j = i; j < taille-1; j++) {
            tab[j] = tab[j+1];
        }
        tab[taille-1] = 0;
        taille = taille-1;
        return true;
    }
}
```

On dispose de deux méthodes inserer et supprimer. Si le tableau n'est pas plein, la méthode inserer ajoute l'élément passé en argument au tableau en conservant l'ordre et renvoie true. Si le tableau est plein, elle renvoie false . La méthode supprimer enlève une occurrence de l'élément passé en argument et renvoie true. Elle renvoie false si l'élément n'était pas présent.

On cherche à tester la méthode supprimer de cette classe, de manière « interne », c'est-à-dire en ayant accès à tous les attributs.

1. Proposer un ensemble de tests pour la fonction supprimer.
2. Pour les modifications suivantes du programme, dire si la modification introduit une faute et lorsque c'est possible, donner un cas de test permettant de révéler la faute.

(a) On modifie la condition du if par i == taille-1.

(b) On modifie la condition d'arrêt de la boucle for par j < taille.

(c) On modifie l'initialisation de la boucle for par j=i+1.

(d) On change taille par capacite dans la condition du while et du if.

(e) On supprime la ligne tab[taille-1] = 0.

(f) On fait les modifications (d) et (e) ensemble.

3. On teste maintenant cette méthode depuis l'extérieur de la classe. De quels observateurs aurait-on besoin idéalement pour pouvoir détecter autant de fautes qu'avec les tests internes ?

4. On ajoute les méthodes suivantes à la classe Tableau :

```
int   taille()
int   capacite()
boolean   present(int   val)
```

Quels aspects de l'implémentation ne peut-on plus observer ? Peut-on les tester ? Comment ?

5. Pour deux des tests proposés en question 1, construire la suite d'appels de méthodes (le scénario de test) permettant d'exécuter ce test, depuis l'initialisation des objets jusqu'à la vérification du résultat obtenu.

Travail demandé :

1. Appliquez le test fonctionnel de l'exemple précèdent en Utilisant les deux outils de test fonctionnel :

- **ZAPTEST free Edition**
- ***JIRA***

2. D'après l'utilisation de ces outils de test, complétez le tableau suivant par les phrases les plus adéquates et les plus concises.

Outil	Interface	Rapidité	Qualité	Domaine	Avantage	Critique
ZAPTEST free Edition						
JIRA						

- Rédiger le rapport le plus pertinent

Chapitre 4 :

Tests d'intégration

Chapter outline

1. Introduction

Dans le monde du développement informatique, le **test d'intégration** est une phase de tests, précédée par les tests **unitaires** et généralement suivie par les tests de **validation**, vérifiant le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module ») ; ces modules logiciels sont intégrés logiquement et testés en groupe. Un projet logiciel typique se compose de plusieurs modules logiciels, codés par différents programmeurs.

L'objectif de chaque phase de test est de détecter les erreurs qui n'ont pas pu être détectées lors de la précédente phase. Pour cela, le test d'intégration a pour cible de détecter les erreurs non détectables par le test unitaire

Le but de ce niveau de test est d'exposer les défauts dans l'interaction entre ces modules logiciels lorsqu'ils sont intégrés. Les tests d'intégration se concentrent sur la vérification de la communication des données entre ces modules. C'est pourquoi il est également appelé **'IL'** (Intégration et Tests), **« Test de chaîne »** et parfois **« Test de fil »**.

2. Principe

Le test d'intégration est une technique de test logiciel utilisée pour tester la compatibilité et la synchronisation de différents composants ou modules au sein d'une application. Son objectif est d'identifier les problèmes de **connectivité** ou de **communication** entre les différents **modules** logiciels.

Étant donné que les logiciels **modernes** sont composés de composants **distincts**, le test d'intégration est nécessaire pour garantir que ces composants fonctionnent ensemble de manière **transparente**. Ce test de deuxième niveau est essentiel pour s'assurer que les **problèmes entre les modules individuels** ne compromettent pas la fonctionnalité globale de l'application.

Les tests d'intégration se produisent après les tests **unitaires**, qui sont des tests de chaque composant individuel, et avant les tests de système, qui sont des tests du système dans son ensemble.

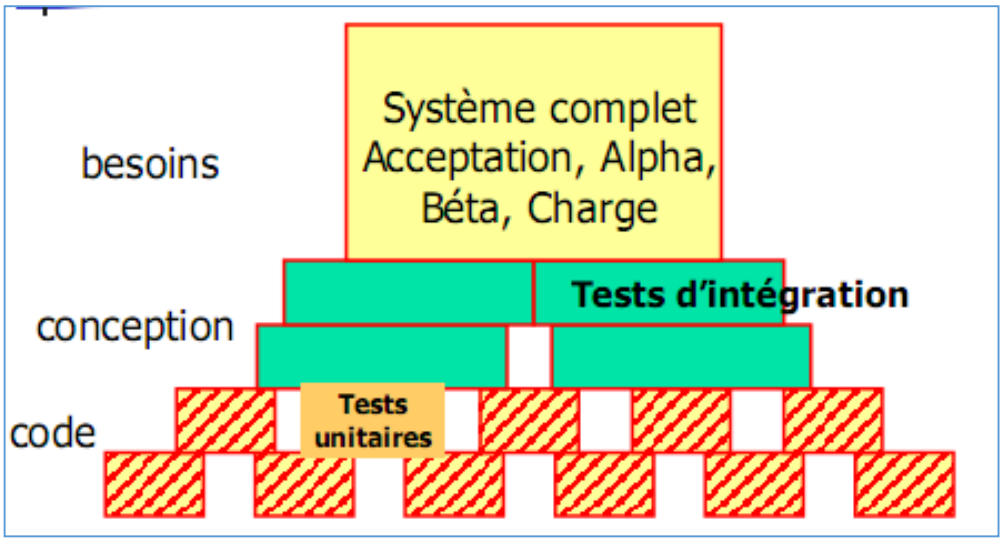


Figure 1 : Les types de tests

Les types de tests se positionnent dans un cycle en V, comme montre la figure :

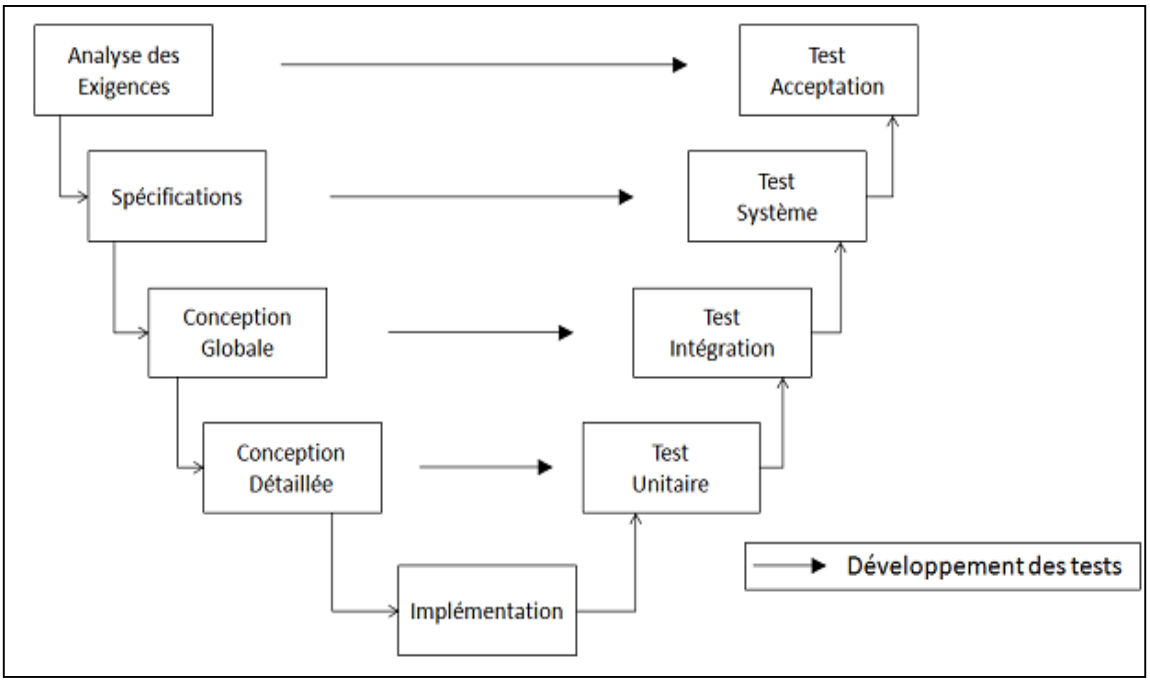


Figure 2 : Positionnement des types de tests dans un cycle en V

3. Objectifs du Test d'intégration

2.1 Analyse du fonctionnement des modules logiciels intégrés : L'objectif principal du test d'intégration est d'examiner la connectivité et la communication entre les différents modules d'une application. Cela implique l'analyse de la logique implémentée et la vérification que les valeurs retournées sont conformes aux exigences du plan de test.

2.2 Assurer une intégration harmonieuse entre les modules et les outils tiers : Un autre objectif du test d'intégration est de tester l'interaction entre les modules et les outils tiers utilisés. Cela inclut la vérification que les données acceptées par l'API sont correctes et que la réponse souhaitée est générée.

2.3 Corriger les défauts de gestion des exceptions : Le test d'intégration permet également d'identifier et de corriger les défauts de gestion des exceptions avant la sortie de la version finale. C'est important car les défaillances d'intégration non détectées peuvent être coûteuses à corriger après la mise en production. En effectuant une analyse approfondie du système lors du test d'intégration, les développeurs peuvent minimiser ces défauts et garantir que la version finale est de haute qualité.

2.4 Réduire les coûts et les risques : Il est important de mettre en place des tests d'intégration dès que possible. Cela permet de vérifier comment les différentes parties de l'application fonctionnent ensemble et d'identifier les éventuelles incompatibilités ou erreurs de communication.

4. Avantages test d'intégration

3.1 Processus de test indépendant : L'un des principaux avantages du test d'intégration est qu'il s'agit d'un processus de test indépendant. Les équipes de QA peuvent commencer le test d'intégration avant que tous les modules ne soient disponibles et testés unitairement. Dès que les modules pertinents sont disponibles, les équipes de QA peuvent procéder au test de leur connectivité et compatibilité.

3.2 Couverture de code élevée : Le test d'intégration permet une analyse approfondie de l'ensemble du système, ce qui conduit à une couverture de code élevée. Cela garantit que presque tous les problèmes de connectivité possibles sont détectés et résolus.

3.3 Détection de bugs : Le test d'intégration agit également comme un détecteur de bugs, permettant la détection de défauts et de problèmes de sécurité dès les premières étapes du cycle de vie de test de logiciel (**STLC**). Cela permet de gagner du temps et donne aux développeurs un meilleur contrôle sur le produit.

3.4 Économie de temps et d'argent : La détection précoce des erreurs permet d'éviter des coûts importants liés aux réparations de dernière minute, aux mises à jour post-lancement et aux temps d'arrêt imprévus. Cela permet également de réduire les coûts globaux de développement du logiciel.

3.5 Évolutivité : Le test d'intégration facilite l'ajout de nouvelles fonctionnalités et de nouvelles versions du logiciel en assurant la compatibilité avec les versions précédentes. Cela permet également de réduire le temps nécessaire pour tester les nouvelles versions.

5. Niveaux de test logiciel

A chaque **phase** (niveau) de développement logiciel est associée son propre type de test comme suit :

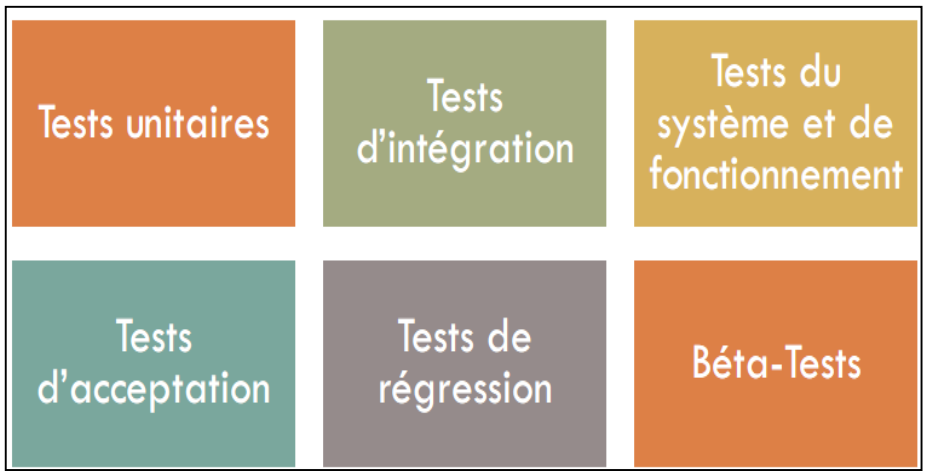


Figure 3 : Les six niveaux de test

4.1 Tests unitaires

Les tests unitaires sont de très bas niveau, près de la source de votre application. Ils consistent à tester les méthodes et fonctions individuelles des classes, des composants ou des modules utilisés par votre logiciel. Les tests unitaires sont en général assez bon marché à automatiser et peuvent être exécutés très rapidement par un serveur d'intégration continue.

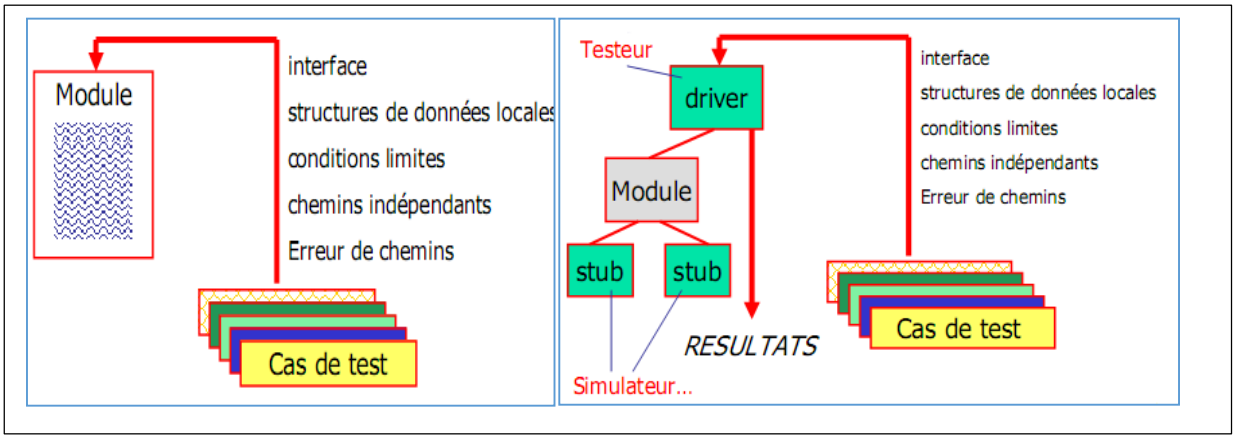


Figure 4 : Structure de test unitaire

4.2 Tests d'intégration

Les tests d'intégration vérifient que les différents modules ou services utilisés par votre application fonctionnent bien **ensemble**. Par exemple, ils peuvent tester **l'interaction** avec la base de données ou s'assurer que les microservices fonctionnent ensemble comme prévu. Ces types de tests sont plus coûteux à exécuter, car ils nécessitent que plusieurs parties de l'application soient fonctionnelles.

4.3 Tests du système et de fonctionnement

Le test système a pour but de vérifier que le logiciel est conforme à ses **spécifications**. En d'autres termes cela revient à s'assurer par des tests que le logiciel remplit bien les **fonctionnalités attendues**. Cette phase vient après celle de l'intégration du logiciel. La méthode de test **boite noire** est généralement utilisée pour ce type de test.

Les tests fonctionnels se concentrent sur les exigences **métier** d'une application. Ils vérifient uniquement la **sortie** d'une action et non les états intermédiaires du système lors de l'exécution de cette action.

Il y a parfois une certaine confusion entre les tests d'intégration et les tests fonctionnels, car ils nécessitent tous les deux plusieurs composants pour interagir. La différence réside dans le fait qu'un test d'intégration peut simplement vérifier que vous pouvez interroger la base de données, tandis qu'un test fonctionnel s'attend à obtenir une valeur spécifique de la base de données, telle que définie par les exigences du produit.

4.4 Tests d'acceptation

Les tests d'acceptation sont des tests formels exécutés pour vérifier si un système répond à ses exigences **métier**. Ils nécessitent que l'application soit **entièrement opérationnelle** et se concentrent sur la **simulation** du comportement des utilisateurs. Ils peuvent aussi aller plus loin, et mesurer la performance du système et rejeter les changements si certains objectifs ne sont pas atteints.

Cette phase de test est l'une des plus importantes, car elle fait intervenir les exigences du **client**. En effet, elle a pour but de vérifier si le logiciel est conforme, en plus de ses spécifications, aux différentes exigences fixées par le client.

4.5 Tests de non régression

Le TNR est donc une pratique de test logiciel qui permet de vérifier qu'une application **fonctionne** toujours comme prévu après toute **modification**, mise à jour ou encore amélioration du code. Les modifications apportées au code peuvent entraîner des **problèmes**, des anomalies ou des dysfonctionnements.

4.6 Test Alpha-Beta

a. **Test Alpha :** Le test alpha est réalisé après les tests d'acceptation. Il a pour objectif de faire tester le logiciel en interne, par des utilisateurs qui n'ont pas participé au développement du projet.

b. **Test Bêta :** Le test bêta est réalisé après le test alpha. Il a pour but de faire tester le logiciel par un échantillon de personnes, le plus souvent externes à l'entreprise (des potentiels utilisateurs par exemple).

6. Type de de test d'intégration

5.1 Big Bang intégration

Si tous les modules marchent bien séparément, pourquoi douter qu'ils ne marcheraient pas ensemble ?

Réunir les modules : **Interfacer**

Intégration de tous les composants à tester en une seule étape. (Intégration massive !)

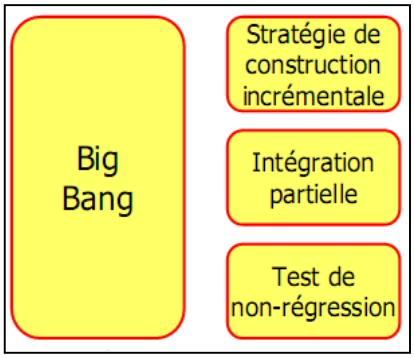


Figure 5 : Big Bang intégration

Les tests d'intégration Big-bang sont une stratégie qui consiste à tester tous les modules d'un système **ensemble après leur développement complet**. Cela peut être une approche **risquée** et stimulante, mais elle peut aussi avoir certains avantages pour les projets complexes qui nécessitent des économies de temps et d'argent.

Le test d'intégration big-bang est un type de test d'intégration qui **combine tous les modules ou composants d'un système en une seule unité et les teste dans leur ensem**ble. Il **n'utilise pas d'étapes intermédiaires ou de stubs pour simuler** le comportement des modules

manquants ou incomplets. Il est généralement effectué **après les tests unitaires** des modules individuels et **avant** les tests du système et les tests d'acceptation.

a. Planification et exécution efficace des tests d'intégration big-bang

Afin d'effectuer efficacement les tests d'intégration big-bang, une planification et une exécution minutieuses sont nécessaires. Cela comprend la définition de la portée et des objectifs des tests d'intégration et l'identification des modules ou des composants qui doivent être intégrés et testés. En outre, des critères et des normes pour les tests d'intégration devraient être établis, ainsi que la conception de scénarios de test et de scénarios couvrant les exigences fonctionnelles et non fonctionnelles du système. L'environnement de test et les données doivent être préparés et rendus cohérents, fiables et capables de gérer la charge et la contrainte des tests d'intégration. Il est également important de coordonner avec les développeurs et les testeurs, d'attribuer des rôles et des responsabilités clairs pour les tests d'intégration, de surveiller les progrès et de rendre compte des résultats. Enfin, les erreurs ou les bogues qui se produisent pendant les tests d'intégration doivent être analysés, défavorisés, hiérarchisés et résolus rapidement.

b. Les avantages des tests d'intégration big-bang

Les tests d'intégration big-bang peuvent être une solution viable pour les projets complexes qui nécessitent des économies de temps et d'argent. Ce type de test est plus rapide et moins cher que les autres méthodes de test d'intégration, car il ne nécessite pas de développer et de maintenir des pilotes ou des **stubs**, ni de suivre un ordre d'intégration spécifique. En outre, il peut tester le système dans son ensemble, révélant les interactions et les dépendances entre tous les modules ou composants qui peuvent ne pas être évidentes dans d'autres méthodes de test d'intégration. Il facilite également la communication et la collaboration entre les développeurs et les testeurs, car ils peuvent travailler sur le même système et partager le même environnement de test et les mêmes données.

c. Les inconvénients des tests d'intégration big-bang

Les tests d'intégration big-bang peuvent présenter des inconvénients et des défis à prendre en considération. Il peut être plus risqué et difficile que d'autres méthodes de test d'intégration, car il peut introduire plus d'erreurs et de bogues, ce qui les rend plus difficiles à identifier et à corriger en raison de la complexité et de l'interdépendance du système. De plus,

cela peut retarder le processus de test et la boucle de rétroaction, car cela nécessite l'achèvement et la disponibilité de tous les modules ou composants, qui peuvent avoir des calendriers et des priorités de développement différents. En outre, il nécessite plus de ressources et de coordination que d'autres méthodes de test d'intégration, telles qu'un environnement de test et des données volumineux et stables, ainsi qu'une équipe d'intégration et une direction solides.

- Besoin de driver et de stub pour chaque module
- Ne permet pas le développement en parallèle
- Rend la localisation des erreurs difficile
- Les erreurs d'interface peuvent facilement passer inaperçues

Exemple : Tester un site web une fois que le frontend et le backend sont entièrement développés.

5.2 Top-down Intégration

- Création de bouchons
- Test tardif des couches basses
- Détection précoce des défauts d'architecture
- Effort important de simulation des composants absents et multiplie le risque d'erreurs lors du remplacement des bouchons.
- La simulation par « couches » n'est pas obligatoire

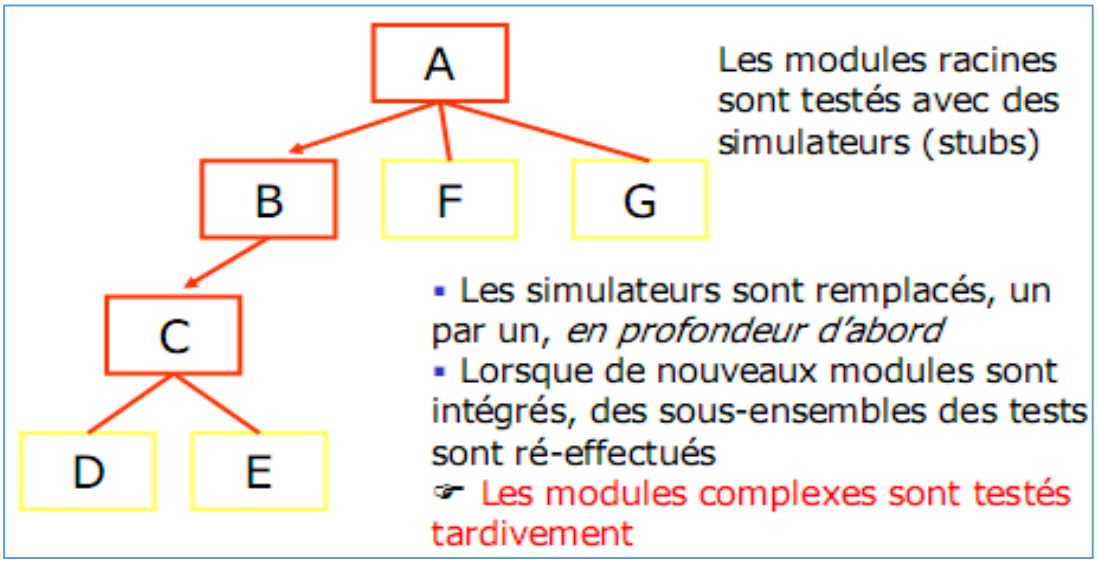


Figure 6 : Top-down Intégration

On teste les modules les plus hauts puis ceux en dessous.

On obtient donc les tests de :

- première étape 1
- seconde étape 1, 2 et 3
- troisième étape 1, 2, 3, 4, 5 et 6
- quatrième étape 1, 2, 3, 4, 5, 6, 7, 8 et 9

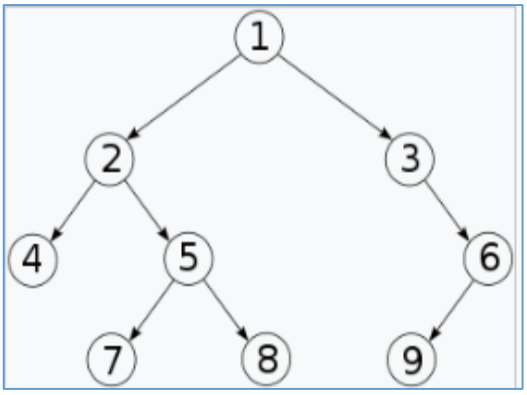


Figure 7 : Top-down exemple

a. Les avantages

- Localisation des erreurs plus facile
- Nécessite peu de driver (ou **Mock**)
- Possibilité d'obtenir un prototype rapidement
- Plusieurs ordres de test/ implémentation(s) possible(s)
- Les erreurs de conception majeures sont détectées en premier dans les modules au plus haut niveau

b. Les désavantages

- Nécessite beaucoup de stubs (bouchon de test)
- Des modules de bas niveau potentiellement réutilisables risquent d'être mal testés

Il convient de noter qu'il est parfois possible de **vérifier** un programme informatique, c'est-à-dire prouver, de manière plus ou moins automatique, qu'il assure certaines propriétés.

Exemple : Tester une application bancaire en intégrant d'abord l'interface utilisateur, puis les services de transaction.

5.3 Bottom-up Intégration

On teste les modules du plus bas niveau puis ceux plus hauts qui les appellent.

Exemple : Tester une API de paiement avant d'intégrer l'interface utilisateur.

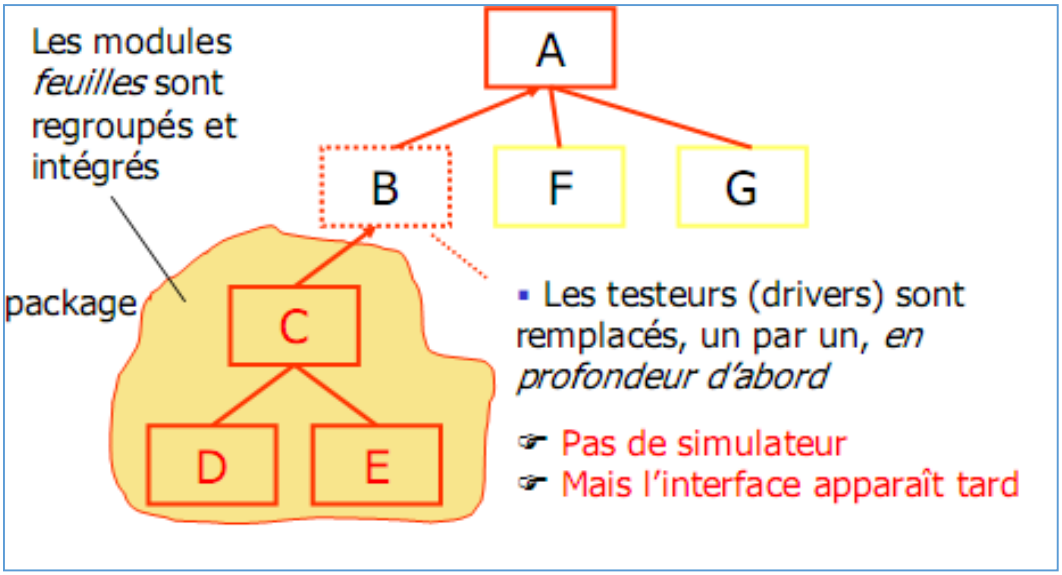


Figure 8 : Bottom-up Intégration

- première étape
  - 7
  - 8
  - 9
- seconde étape
  - 4
  - 5, 7 et 8
  - 6 et 9
- troisième étape
  - 2, 4, 5, 7 et 8
  - 3, 6 et 9
- quatrième étape 1, 2, 3, 4, 5, 6, 7, 8 et 9

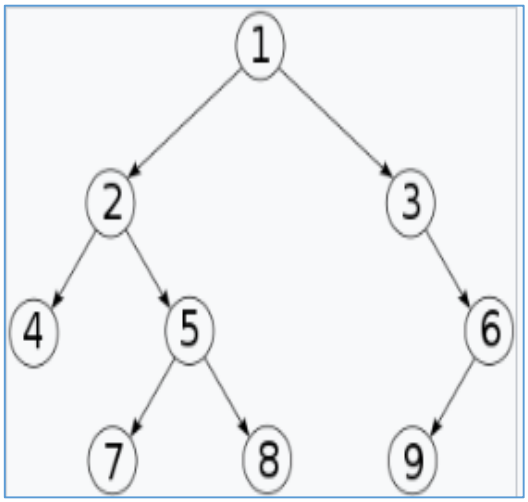


Figure 9 : Bottom-up exemple

a. Avantages -Faible effort de simulation -Construction progressive de l'application s'appuie sur les modules réels. Pas de version provisoire du logiciel -Les composants de bas niveau sont les plus testés, -Définition des jeux d'essais plus aisée -Démarche est naturelle.

- Localisation facile des erreurs
- Aucun besoin de stub
- Les modules réutilisables sont testés correctement
- Les tests peuvent se faire en parallèle avec le développement

b. Inconvénients

- Détection tardive des erreurs majeures
- Planification dépendante de la disponibilité des composants
- Nécessite des drivers
- Les modules de haut niveau sont testés en dernier
- Aucun squelette de système n'est conceptualisé

5.4 Intégration en Sandwich (mixte)

Combinaison des approches descendante et ascendante. Il faut distinguer trois niveaux :

- Niveau logique (haut) ;
- Niveau cible (milieu) ;
- Niveau opérationnel (bas).

On teste **en même temps** les modules de haut et bas niveau, puis on avance vers le centre, méthode réunissant les avantages des deux précédentes.

Exemple : Tester en parallèle les **interfaces utilisateur** (Top-Down) et les **bases de données** (Bottom-Up).

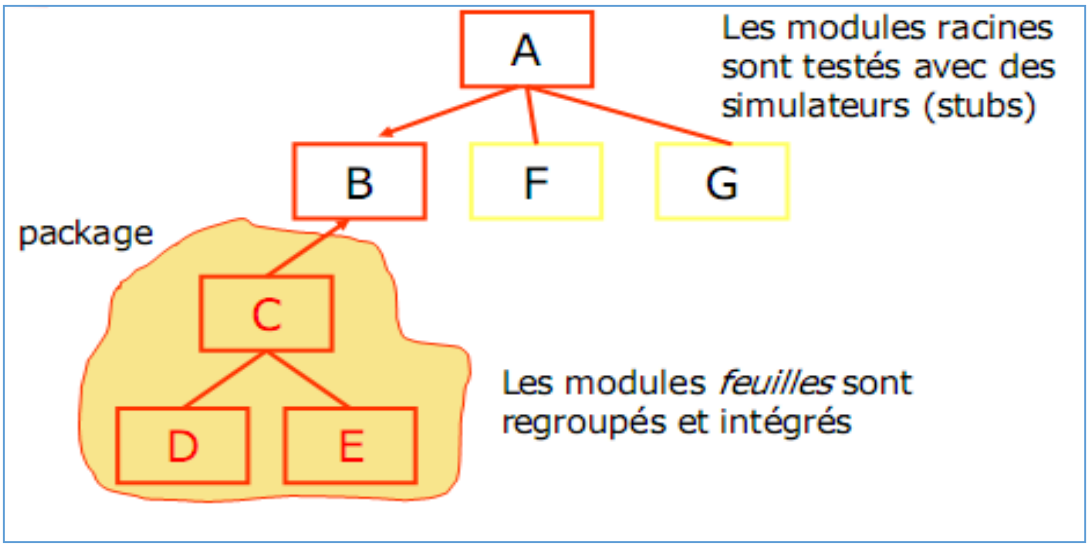


Figure 10 : Intégration en Sandwich

- première étape
  - 1
  - 7 et 8
  - 9
- seconde étape
  - 1, 2 et 3
  - 5, 7 et 8
  - 6 et 9
  - 4
- troisième étape
  - 1, 2, 3, 4, 5 et 6
  - 2, 4, 5, 7 et 8
  - 3, 6 et 9
- quatrième étape
  - 1, 2, 3, 4, 5, 6, 7, 8 et 9

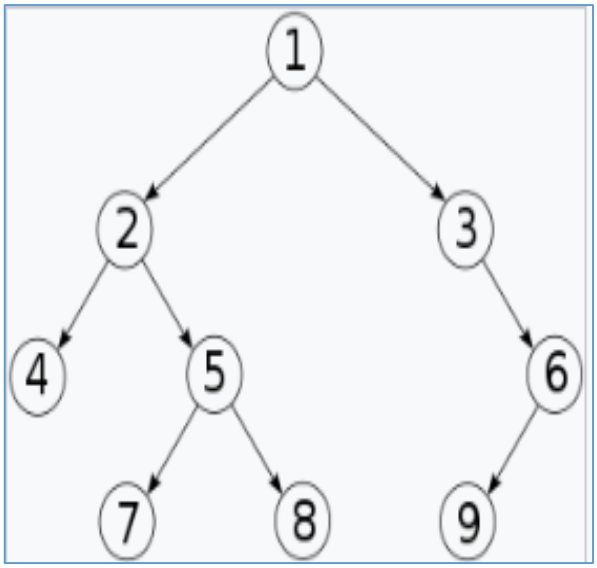


Figure 11 : Sandwich exemple

a. Avantages :

- Suivre le planning de développement de sorte que les premiers composants terminés soient intégrés en premier.
- Prise en compte du risque lié à un composant de sorte que les composants les plus critiques puissent être intégrés en premier.
- Les niveaux haut et bas peuvent être testés en parallèle
- Diminuer les besoins en driver et en stub.

b. La principale difficulté d'une intégration mixte réside dans sa **complexité** car il faut alors gérer intelligemment sa stratégie de test afin de concilier les deux modes d'intégration : ascendante et descendante.

- Plus complexe à planifier
- Le niveau cible peut être difficile à définir

5.5 Techniques Complémentaires

Tests basés sur des Mocks : Simulent des modules externes avec des comportements prédéfinis.

Tests de Contrat : Vérifient si deux modules communiquent correctement selon un contrat d'API.

Tests d'Interface : Vérifient la cohérence des échanges entre les composants via REST, SOAP, ou d'autres protocoles.

5.6 Choix de la Méthode

Le choix dépend du **projet** :

- **Petits systèmes** → Big Bang peut suffire.
- **Systèmes critiques** (banque, avionique) → Approches incrémentales recommandées.
- **Développement Agile** → Tests en continu avec mocks et tests d'API automatisés.

5.7 Développement Agile et Tests basés sur des Mocks

Dans le domaine des tests logiciels, un **mock** (ou objet simulé) est un objet qui imite le comportement d'un objet réel de manière contrôlée. Les mocks sont principalement utilisés dans les tests unitaires pour isoler le composant testé de ses dépendances externes, permettant ainsi de vérifier son fonctionnement de manière indépendante.

1. **Définition**

Les **tests basés sur des mocks** consistent à **remplacer certains composants d'un système par des objets simulés**, appelés **mocks**, pour tester un module en isolant ses dépendances. Ces mocks imitent le comportement des composants réels sans exécuter leur logique complète.

Principe des mocks :

L'utilisation de mocks repose sur la simulation des interactions entre le composant testé et ses dépendances. Cela permet de contrôler les réponses des dépendances et de vérifier que le composant testé interagit correctement avec elles. Les mocks sont particulièrement utiles pour tester les interactions et les attentes de votre code avec une dépendance, en vérifiant si le code appelle la dépendance avec les bons arguments, dans le bon ordre et à la bonne fréquence.

Différence entre mocks et stubs :

Il est important de distinguer les mocks des stubs. Les stubs sont utilisés pour fournir des réponses prédéfinies aux appels effectués pendant le test, sans vérifier les interactions avec le composant testé. En revanche, les mocks permettent non seulement de fournir des réponses, mais aussi de vérifier que certaines méthodes ont été appelées avec des paramètres spécifiques, dans un ordre donné.

2. Pourquoi Utiliser des Mocks ?

- **Tester un module indépendamment** des autres composants.
- **Éviter d'attendre** que les autres parties du système soient développées.
- **Réduire les coûts et la complexité** des tests (pas besoin d'accéder à une vraie base de données, API, etc.).
- **Simuler des scénarios difficiles à reproduire** (erreurs réseau, réponses lentes, etc.).

3. **Différence entre Stub, Mock et Fake**

Type	Définition	Exemple
Stub	Remplace un composant et renvoie des valeurs fixes.	Une fonction renvoie toujours "Utilisateur Valide" pour un test de connexion.
Mock	Simule un objet et vérifie ses interactions.	Vérifier si un email est bien envoyé après une inscription.
Fake	Remplace un composant avec une version simplifiée.	Une base de données en mémoire au lieu d'une vraie base SQL.

4. Exemple : Test d'une API avec Mocks

Contexte : On teste un service de commande en ligne qui dépend d'une API de paiement.
Problème : L'API réelle peut être lente, indisponible, ou entraîner des coûts.
Solution : On utilise un mock pour simuler l'API.

Code Exemple en Python (avec unittest & unittest.mock)

```python
import unittest
from unittest.mock import MagicMock

# Service à tester
class OrderService:
    def __init__(self, payment_api):
        self.payment_api = payment_api

    def process_order(self, amount):
        if self.payment_api.charge(amount):
            return "Commande validée"
        return "Échec du paiement"

# Test avec un Mock
class TestOrderService(unittest.TestCase):
    def test_order_success(self):
        mock_payment_api = MagicMock()
        mock_payment_api.charge.return_value = True  # Simule un paiement réussi

        service = OrderService(mock_payment_api)
        result = service.process_order(100)

        self.assertEqual(result, "Commande validée")
        mock_payment_api.charge.assert_called_once_with(100)  # Vérifie l'appel à l'API

if __name__ == "__main__":
    unittest.main()
```

Explication :

1. **On remplace l'API de paiement par un mock** avec MagicMock().
2. **On définit son comportement** (charge.return_value = True).
3. **On exécute le test sans appeler l'API réelle**.

Exemple de Test Basé sur des Mocks en Java (avec Mockito)

On veut tester un **service de commande** qui dépend d'un **service de paiement externe**.

Problème : L'API de paiement peut être lente ou indisponible.

Solution : Utiliser **Mockito** pour simuler le service de paiement.

2. Dépendances (Maven ou Gradle)

Ajoutez **Mockito** et **JUnit** à votre projet :

Pour Maven (pom.xml)

```xml
<dependencies>
    <!-- JUnit 5 -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.8.1</version>
        <scope>test</scope>
    </dependency>

    <!-- Mockito -->
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>4.0.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Pour Gradle (build.gradle)

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
    testImplementation 'org.mockito:mockito-core:4.0.0'
}
```

3. Code du Service à Tester

```
// Interface du service de paiement
public interface PaymentService {
    boolean processPayment(double amount);
}

// Implémentation du service de commande
public class OrderService {
    private PaymentService paymentService;

    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public String placeOrder(double amount) {
        if (paymentService.processPayment(amount)) {
            return "Commande validée";
        }
        return "Échec du paiement";
    }
}
```

4. Test avec Mockito

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

class OrderServiceTest {

    @Test
    void testOrderSuccess() {
        // 1 Créer un mock du service de paiement
        PaymentService mockPaymentService = mock(PaymentService.class);

        // 2 Définir le comportement simulé (paiement toujours réussi)
        when(mockPaymentService.processPayment(100.0)).thenReturn(true);

        // 3 Tester le service de commande avec le mock
        OrderService orderService = new OrderService(mockPaymentService);
        String result = orderService.placeOrder(100.0);

        // 4 Vérifier le résultat
```

```
        assertEquals("Commande validée", result);

        // 5 Vérifier que processPayment() a bien été appelé avec 100.0
        verify(mockPaymentService).processPayment(100.0);
    }

    @Test
    void testOrderFailure() {
        // Mock du service de paiement avec un paiement refusé
        PaymentService mockPaymentService = mock(PaymentService.class);
        when(mockPaymentService.processPayment(50.0)).thenReturn(false);

        OrderService orderService = new OrderService(mockPaymentService);
        String result = orderService.placeOrder(50.0);

        assertEquals("Échec du paiement", result);
        verify(mockPaymentService).processPayment(50.0);
    }
}
```

5. **Explication**

1 **On crée un mock** du service de paiement avec mock(PaymentService.class).

2 **On définit son comportement** : when(mock.processPayment(100.0)).thenReturn(true).

3 **On exécute la méthode à tester** (placeOrder()).

4 **On vérifie que le résultat est correct** (assertEquals()).

5 **On s'assure que le mock a bien été appelé** (verify(mock).processPayment(100.0)).

6. **Pourquoi Utiliser Mockito ?**

- **Évite de dépendre d'un service externe réel** (ex : API de paiement).
- **Permet de tester uniquement la logique métier**.
- **Exécute les tests rapidement** et sans coût.
- **Facile à intégrer avec JUnit** pour des tests automatisés.

5. Quand Utiliser les Mocks ?

* **Tests d'API** → Simuler des réponses d'API REST ou GraphQL.

* **Tests de bases de données** → Éviter d'écrire dans une vraie base.

*** Tests de services tiers** → Simuler une plateforme de paiement, d'email, etc.

* **Tests unitaires** → Isole un module en simulant ses dépendances.

Les mocks permettent de **tester plus rapidement et efficacement** en isolant les dépendances d'un module. Ils sont très utilisés dans les **tests unitaires et d'intégration**, notamment dans le développement Agile et les architectures basées sur des microservices.

6. Acteurs de test

Les principaux acteurs et activités qui interviennent tout au long du test :

Niveaux de Test	Méthode de Test	Source utilisée	Qui teste ?	But
Unitaire	Boite Blanche + Boite Noire	Conception détaillée ou Code source	Programmeurs	Tester des portions élémentaires dans le modèle utilisé (ex, une classe)
Intégration	Boite Blanche + Boite Noire	Conception Architecture	Programmeurs ou Testeurs	Tester des combinaisons de plusieurs portions
Système	Boite Noire	Spécifications	Testeurs	Tester le système par rapport à ses spécifications
Acceptation	Boite Noire	Exigences (clients)	Du coté du client	Tester le produit par rapport aux exigences du client
Alpha	Boite Noire	Produit	Testeurs n'ayant pas participé au projet	Tester le produit par des testeurs indépendants
Beta	Boite Noire	Produit	Utilisateurs potentiels	Tester le produit par des utilisateurs potentiels

Figure 12 : acteurs de test

6.1 Les Acteurs et Activités dans un Processus de Test

Le processus de test logiciel implique plusieurs **acteurs** et suit des **activités clés** pour garantir la qualité du produit.

1 Acteurs du Test

Les principaux acteurs impliqués dans les tests logiciels sont :

Acteur	Rôle et Responsabilités
Testeur	Exécute les tests, identifie les défauts et rédige les rapports de test.
Analyste de test	Conçoit les cas de test, définit les stratégies de test et analyse les résultats.
Chef d'équipe de test	Gère l'équipe de test, planifie les tests et suit leur avancement.
Développeur	Corrige les bugs détectés et écrit parfois des tests unitaires.
Architecte logiciel	S'assure que le logiciel est testable et conforme aux exigences.
Responsable qualité (QA Manager)	Définit la politique de qualité et supervise les processus de test.
Client / Utilisateur final	Peut participer aux tests d'acceptation et donner un retour sur le produit.
Product Owner (Scrum)	Valide les critères d'acceptation et priorise les tests fonctionnels.
Ingénieur en automatisation	Met en place et exécute les tests automatisés.

---

2 Activités du Test (Cycle de Vie du Test)

Le test suit un **cycle de vie** structuré en plusieurs étapes.

🔻 1. Planification et Stratégie de Test

Objectif : Définir la stratégie et organiser les tests.
Acteurs : Chef de test, Analyste de test, QA Manager.
Activités :

- Définition des objectifs et du périmètre des tests.
- Identification des ressources et des outils nécessaires.

- Estimation du budget et du planning.
- Identification des risques et plans d'atténuation.

🔹 2. Conception et Préparation des Tests

Objectif : Créer les cas de test et préparer l'environnement.
Acteurs : Analyste de test, Testeur, Ingénieur en automatisation.
Activités :

- Rédaction des **cas de test** basés sur les spécifications fonctionnelles et techniques.
- Développement de **scripts de test automatisés** si nécessaire.
- Préparation des jeux de données pour les tests.
- Configuration de l'environnement de test (serveurs, bases de données, outils).

🔹 3. Exécution des Tests

Objectif : Réaliser les tests et identifier les défauts.
Acteurs : Testeur, Ingénieur en automatisation, Développeur.
Activités :

- Exécution des **tests manuels et automatisés**.
- Enregistrement des résultats et comparaison avec les attentes.
- Identification et signalement des **bugs** dans un outil de suivi (JIRA, Bugzilla, etc.).
- Ré-exécution des tests après correction des anomalies (**tests de régression**).

🔹 4. Analyse et Suivi des Défauts

Objectif : Vérifier la qualité du produit et assurer la correction des bugs.
Acteurs : Testeur, Développeur, Chef de test.
Activités :

- Analyse des anomalies détectées.
- Communication des bugs aux développeurs.
- Priorisation des corrections selon la gravité des défauts.
- Exécution de **tests de confirmation** pour vérifier la correction.

🔻 5. Clôture et Rapport de Test

Objectif : Évaluer la qualité et documenter les résultats.
Acteurs : Chef de test, QA Manager, Client.
Activités :

- Rédaction du **rapport de test** avec le bilan des tests effectués.
- Évaluation des indicateurs (nombre de tests réussis/échoués, taux de couverture, nombre de bugs détectés).
- Décision de la **mise en production** ou du **report** si des problèmes critiques subsistent.
- Rétrospective pour améliorer les futurs cycles de test.

Le processus de test est **collaboratif**, impliquant divers acteurs et plusieurs étapes. Une bonne organisation et communication entre les équipes garantissent **un produit final de qualité** et sans anomalies majeures.

6.2 Degré de tests :

Il se traduire selon six niveaux cimentaires :

- **Fatal :** impossible de contenir les tests à causés de la sévérité des défaillances
- **Critique :** les tests peuvent contenir mais l'application ne peut passer en mode production
- **Majeur :** l'application peut passer en mode production mais des exigences fonctionnelles importantes ne sont pas satisfaites
- **Medium :** l'application peut passer en mode production mais des exigences fonctionnelles sans très grand import ne sont pas satisfaites.
- **Mineur :** l'application peut passer en mode production, la défaillance doit être corrigée mais elle sans import sur les exigences métier
- **Cosmétique** : défaillance mineurs relatives à l'interface (couleurs, police,..) mais n'ayant pas une relation avec les exigences du client.

7. Test de régression après l'intégration

7.1 Principe de test de régression (test de non-régression ou encore TNR)

L'origine du mot "*régression*" signifie "*retourner à un stade antérieur de quelque chose*". Le **TNR** est donc une pratique de test logiciel qui permet de **vérifier qu'une application fonctionne toujours** comme prévu après toute **modification**, **mise à jour** ou encore **amélioration du code**.

Les **modifications** apportées au code peuvent entraîner des **problèmes**, des **anomalies** ou des **dysfonctionnements**. **Les TNR ont pour objectif de limiter ces risques**, afin que le code préalablement conçu et testé reste **opérationnel** après de nouvelles **modifications**.

En principe, une application passe par plusieurs tests avant que les modifications ne soient mises en place dans la principale version de développement. Le test de non régression est **la dernière étape,** car il permet de **vérifier le comportement du produit dans son ensemble**.

7.2 Quand faut-il appliquer les tests de non régression

Généralement, les TNR sont utilisés dans les cas suivants :

- Une nouvelle **requête** est ajoutée à une fonctionnalité existante
- Des **corrections** sont apportées au code pour résoudre des défauts
- Ajout d'une **nouvelle** option ou fonctionnalité
- **Optimisation** du code source afin d'améliorer les performances
- Des **mises à jour** sont effectuées
- **Modifications** de la structure

7.3 les différents types de tests de non régression (TNR)

Les tests de non régression se présentent sous plusieurs formes. Ci-dessous, un aperçu **des techniques les plus courantes de test de non régression – TNR**.

a. **Test de non régression correctif :** Le test de correction des performances est apprécié pour sa simplicité, car il nécessite **moins** d'efforts. Il réutilise les tests existants, dans la mesure où aucun changement **majeur** n'a été apporté aux fonctionnalités du produit (pas de changement du scénario de test).

b. **Test de non régression complet :** Comme son nom l'indique, le TNR complet s'agit d'une procédure de test relativement **complexe** qui nécessite de tester à **nouveau** tous les éléments du produit à partir de **zéro**. Ce test contrôle de manière rétrospective toutes les petites modifications qui ont été apportées depuis le début. Cette approche est parfois utilisée lorsque l'on soupçonne que quelque chose a été oublié (ou n'a pas été fait correctement) au cours des phases de test réalisées précédemment. Dans de telles circonstances, il peut être plus efficace de tout **re-tester** depuis le début plutôt que de tenter d'identifier ce qui n'a pas fonctionné. Ce n'est pas une méthode très courante, compte tenu des problèmes de budget, mais elle vous permet d'être plus rassurées quant à la qualité de votre projet.

c. **Test de non régression sélective :** Ce type de test peut être vu comme un juste milieu entre le test de non régression correctif et le test de non régression complet. Il n'est pas aussi simple que le premier, mais pas aussi complet que le second. Le test de régression sélectif permet de **choisir** certains tests dans un ensemble pour inspecter les parties du code qui ont été **affectées**.

d. **Test de non régression progressive :** Quand les tests établis ne sont plus utiles, de nouveaux tests sont créés. C'est souvent le cas lorsque les caractéristiques du produit sont **modifiées**, de **nouveaux** scénarios de test sont alors créés pour répondre à ces changements.

e. **Test de non régression partielle :** le développement de logiciels est vu comme la construction d'une maison LEGO : vous créez d'abord des **blocs** séparés, puis vous les **combinez**, et vous obtenez une maison **entière**. Vos différents **modules** peuvent être en cours de développement depuis un certain temps. Lorsque vous êtes sur le point de les **fusionner** avec une version principale du code, des tests partiels sont effectués, en se basant essentiellement sur les scénarios de test existants.

f. **Test de non régression unitaire :** Il s'agit d'une approche directe qui vise à tester le code en tant que bloc **individuel**, sans tenir compte de toutes les **contraintes**, intégrations et autres éléments qui peuvent s'y rattacher.

7.4 Etapes pour réaliser un test de non régression

Les méthodes de TNR peuvent varier d'une organisation à l'autre. Cependant, il y a quelques **étapes de référence** :

- **Détectez les modifications du code source :** Détectez la modification et son optimisation dans le code source; puis identifiez les composants ou **modules** qui ont été **modifiés**, ainsi que leurs impacts sur les fonctionnalités existantes.

- **Classez par ordre de priorité les changements et les besoins du produit :** Il convient ensuite de **hiérarchiser** ces modifications ainsi que les besoins du produit afin de **structurer** le processus de test avec les **cas** de test et les **outils** de test appropriés.
- **Déterminez le point de départ grâce au TNR :** il faut s'assurer que l'application respecte les critères prédéfinis de conformité avant de procéder à l'exécution du test de régression.
- **Déterminez le point de sortie :** Déterminez un point de sortie pour les **critères** de qualification requis ou les critères minimaux fixés à l'étape 3.
- **Programmez des tests :** Pour finir, identifiez tous les éléments des tests et planifiez le moment voulu pour l'exécution.

8. Approches des jeux de test : Tests exhaustifs

L'objet principal est d'analyser le comportement d'un logiciel dans un environnement donné. Ainsi, il ne sert a priori à rien de tester le bon fonctionnement de la gestion du système de freinage d'une automobile pour une vitesse de 1000 km/h. En général, **tester un programme de façon exhaustive est impossible**. Il faut choisir un sous-ensemble des tests qui **maximise** la probabilité de détecter les erreurs. Il est possible d'utiliser des tests aléatoires, mais leur efficacité est faible pour tester le comportement attendu. Une meilleure approche : déterminer un ensemble de tests fonctionnels qui seront complémentés de tests structurels.

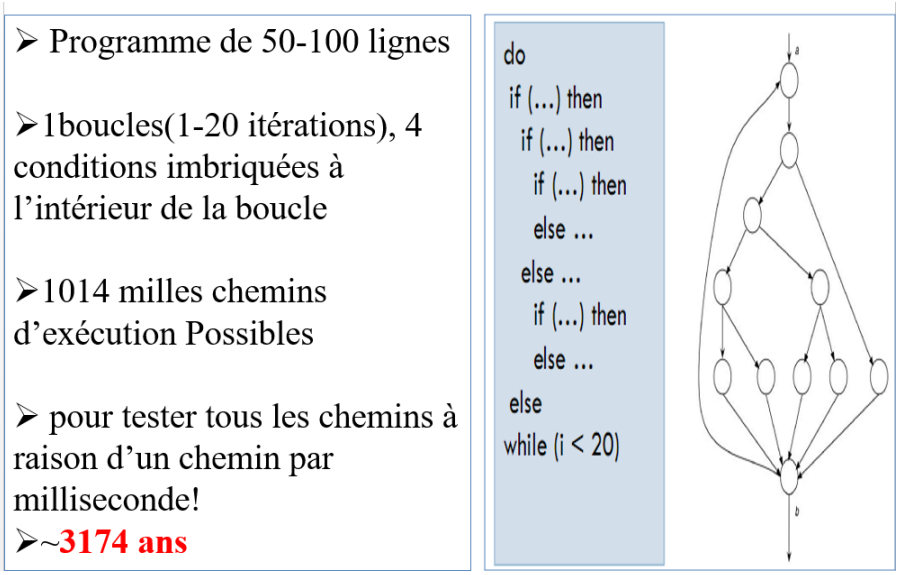


Figure 5 : test exhaustive

➢ **Spécification** : Un programme prend en entrée trois entiers. Ces trois entiers sont interprétés comme représentant les longueurs des côtés d'un triangle. Le programme rend

un résultat précisant s'il s'agit d'un triangle scalène, isocèle ou équilatéral. Produire une suite de cas de tests pour ce programme, 13 cas de test :

1. Cas scalène valide (1,2, 3 et 2,5,10 ne sont pas valides)

2. Cas équilatéral valide

3. Cas isocèle valide (2,2, 4 n'est pas valide)

4. Cas isocèle valide avec les trois permutations (e.g. 3,3, 4; 3,4,3; 4,3,3)

5. Cas avec une valeur à 0

6. Cas avec une valeur négative

7. Cas ou la somme de deux entrées est égale à la troisième entrée

8. cas pour le test 7 avec les trois permutations

9. Cas ou la somme de deux entrées est inférieur à la troisième entrée

10. 3 cas pour le test 9 avec les trois permutations

11. Cas avec les trois entrées à 0

12. Cas avec une entrée non entière

13. Cas avec un nombre erroné de valeur (e.g. 2 entrées, ou 4)

Chacun de ces 13 tests correspond à un défaut constaté dans des implantations de cet exemple triangle. La moyenne des résultats obtenus par un ensemble de développeurs expérimentés est de 7.8 sur 13. La conception de tests est une activité complexe, a fortiori sur de grandes applications

9. Outils et environnement pour les tests d'intégration

- **VectorCast/Ada :** C'est un outil populaire utilisé pour les tests unitaires et d'intégration.

- Il automatise les tests unitaires et d'intégration et réduit le temps et les coûts.
- Il offre des métriques de qualité logicielle pour améliorer et surveiller les principaux indicateurs de test.
- Il crée des modules manquants à l'aide de stubs et de drivers pour simuler la fonctionnalité du code.

- Il génère automatiquement des stubs et des drivers de code.
- Il offre une réutilisation des cas de test pour les tests de régression.
- Il met en évidence les codes à haut risque à l'aide d'une analyse de complexité de code intégrée.

  ➢ **Citrus** : Un framework d'intégration de test open source qui prend en charge une large gamme de protocoles de message et de formats de données.
- Il fonctionne mieux pour les tests d'intégration complexes et offre des capacités de validation supérieures pour plusieurs formats.
- Il prend en charge les demandes et les réponses côté serveur et côté client.
- Citrus prend en charge plusieurs protocoles tels que HTTP, JMS et SOAP.
- Il s'agit d'une solution peu coûteuse pour tous les besoins en matière de tests d'intégration, permettant la validation de la base de données et la re-création des erreurs.
- Il fournit également des plans de test et des documents pour la couverture des tests.

  ➢ **Ldra** : Il propose différents ensembles d'outils de test d'intégration pour répondre aux normes de conformité des différentes organisations.
- C'est un outil à plateforme ouverte et une solution de test rentable pour les organisations.
- La suite d'outils offre – une analyse statique et dynamique, une analyse de la couverture de code, des examens de conception, etc .
- Il permet l'exécution transparente de tests unitaires et d'intégration.
- Il automatise la certification logicielle et les preuves d'approbation.
- Il offre une prise en charge de l'environnement conjoint pour les tests d'intégration dans une large gamme de projets.
- Analyse de la couverture du code.

  ➢ **Protractor** : Il est un framework open-source et un outil d'automatisation de tests pour les applications web.
- Conçu spécifiquement pour les tests d'applications Angular et AngularJS.
- Il peut être utilisé pour les tests d'intégration, les tests de bout en bout et les tests d'applications web dynamiques.
- Il s'exécute en temps réel dans un navigateur réel et simule l'interaction utilisateur.
- Il prend en charge plusieurs navigateurs, notamment Firefox, Chrome, Safari et Internet Explorer.
- Il permet l'exécution de plusieurs instances de votre application.
- Il effectue les tests du point de vue de l'utilisateur final.

- Il s'intègre avec Jenkins/Browser Stack/Grunt pour automatiser le serveur de test.

  En conclusion, les tests d'intégration sont essentiels dans le cycle de vie des tests logiciels (STLC). La mise en œuvre efficace des tests d'intégration peut éviter le besoin de mises à jour et de corrections de bugs après la publication.

10. Conclusion

Dans le monde du **développement** informatique, le **test d'intégration** est une phase de **tests**, précédée par les **tests unitaires** et généralement suivie par les **tests de validation**, vérifiant le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module ») ; dans le test d'intégration, chacun des modules indépendants du logiciel est assemblé et testé dans l'ensemble

Université de Mohamed Khider – Biskra –
Département d'informatique
Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TD N° 4 : Les tests d'intégration

Exercice 1: Appliquer les méthodes de test d'intégration suivantes pour le choix de deux exemples suivants:

- Big Bang intégration
- Top-down Intégration
- Bottom-up Intégration
- Santdwitc Intégration

Exemple 1: LinkedIn

Prenons l'exemple de LinkedIn et considérons que les modules suivants sont des composants indépendants qui communiquent entre eux :

- Module de login
- Module de page d'accueil
- Module de page de profil
- Module de page de réseau (maintenant onglet Développer)
- Module de page de relations
- Module de page de notification
- Module de messagerie
- Le nouveau module de réseau (onglet Reprendre contact)

Vous êtes invité à rédiger un ou plusieurs tests pour vérifier l'intégration entre :

1. Le module de login et le module home page
2. Le module de home page et le module de la page de profil
3. Le module de la page réseau (onglet développer) et le module de la page relations
4. Le module de la page réseau (onglet Reprendre contact) et le module de la messagerie
5. Le module de la page de notification et le module de la home page

Exemple 2 : Une application e-commerce

- **Contexte** : Un utilisateur passe une commande en ligne.
- **Objectif** : Tester l'intégration entre le frontend, le backend, la base de données et les services externes (ex. : système de paiement).
- **Cas de test** :
    1. L'utilisateur ajoute un produit au panier.
    2. Le panier se met à jour correctement dans l'interface (frontend).
    3. La base de données enregistre la commande (backend <-> base de données).
    4. Le système de paiement est appelé, et une réponse correcte est renvoyée (backend <-> service externe).
    5. Un email de confirmation est envoyé (backend <-> service d'envoi d'email).

Exemple 3 : Une API REST

- **Contexte** : Un service utilise une API tierce pour récupérer des données.
- **Objectif** : Vérifier que l'intégration avec l'API fonctionne comme prévu.
- **Cas de test** :
    1. Envoyer une requête GET à l'API pour récupérer les informations d'un utilisateur.
    2. Vérifier que l'API renvoie une réponse dans le format attendu (ex. : JSON).
    3. Valider que ces données sont correctement traitées et affichées par le système.

Exemple 4 : Une application bancaire

- **Contexte** : Transfert d'argent entre deux comptes.
- **Objectif** : Vérifier l'intégration entre les différents services bancaires.
- **Cas de test** :
    1. Un utilisateur initie un transfert d'argent.
    2. Les débits et crédits sont effectués correctement sur les comptes concernés.
    3. Les journaux de transactions se mettent à jour.
    4. Une notification est envoyée aux deux utilisateurs concernés.

Exemple 5 : Système de gestion des utilisateurs

- **Contexte** : Création d'un nouveau compte utilisateur.
- **Objectif** : Vérifier l'intégration entre l'interface utilisateur, le système de gestion des utilisateurs et les services tiers.
- **Cas de test** :
    1. L'utilisateur remplit un formulaire d'inscription.
    2. Les données sont envoyées au serveur, validées, puis enregistrées dans la base de données.
    3. Un email de validation est envoyé via un service externe (ex. : SendGrid).
    4. L'utilisateur clique sur le lien dans l'email, et son compte est activé.

Exercice 2:

1. D'après votre étude dans l'exercice 1, remplir le tableau suivant :

Méthode	Avantage	désavantage	Rapidité	Qualité	Domaine	Temps
Big Bang intégration						
Top-down Intégration						
Bottom-up Intégration						
Santdwitc Intégration						

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TP N° 4 : Tests d'intégration

Développement Agile et Tests basés sur des Mocks

Les **tests basés sur des mocks** consistent à **remplacer certains composants d'un système par des objets simulés**, appelés **mocks**, pour tester un module en isolant ses dépendances. Ces mocks imitent le comportement des composants réels sans exécuter leur logique complète.

Les tests basés sur des mocks sont **indispensables** pour tester un module **sans dépendre de services externes**. Mockito est l'un des frameworks les plus utilisés en Java pour cet usage.

Les mocks permettent de **tester plus rapidement et efficacement** en isolant les dépendances d'un module. Ils sont très utilisés dans les **tests unitaires et d'intégration**, notamment dans le développement Agile et les architectures basées sur des microservices.

Exemple d'utilisation avec Mockito en Java

Considérons une interface `Position` avec une méthode `getLongitude()`. Pour tester un composant qui dépend de cette interface sans implémenter directement sa logique, on peut utiliser Mockito pour créer un mock de `Position` et définir son comportement attendu.

Voici comment écrire un test unitaire en utilisant Mockito :

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

public class PositionTest {

    @Test
    public void testGetLongitude() {
        // Création du mock
        Position positionMock = mock(Position.class);

        // Définition du comportement attendu
        when(positionMock.getLongitude()).thenReturn(45.0);

        // Appel de la méthode à tester
        double longitude = positionMock.getLongitude();

        // Vérification du résultat
        assertEquals(45.0, longitude, 0);
    }
}
```

Dans cet exemple, nous créons un mock de l'interface `Position`, définissons que l'appel à `getLongitude()` doit retourner `45.0`, puis vérifions que cette valeur est bien retournée lors de l'appel.

Travail demandé

Appliquez les Tests basés sur des Mocks sur votre Mini projet (TP1) qui fournit un développement agile utilisant la méthode **Scrum**.

- **Rapporter votre travail**

Chapitre 5 :

Le test d'acceptation « Recette »

Plan de chapitre

1. Introduction

Les tests d'acceptation utilisateur (également appelés tests d'acceptance, **UAT** ou recette client) sont généralement exécutés par les personnes qui **utiliseront** le produit logiciel dans leur pratique opérationnelle. Ils permettent de **s'assurer** que les changements mis en œuvre au travers de la solution informatique fonctionnent **réellement** et qu'ils leur apportent la valeur ajoutée escomptée.

D'après le Standish Group Chaos Report, **seuls 29%** des projets informatiques dans le monde sont considérés comme **réussis** par les Directions de systèmes d'information et les **clients finaux**. Il y a donc de fortes chances pour que vous ayez déjà été impliqué dans un projet qui n'a pas satisfait pleinement les attentes métier.

L'acceptation utilisateur (aussi connue sous le nom de recette client ou d'acceptance) peut aider. Parfois bâclés pour cause de dépassement de délais, les tests jouent en effet un rôle fondamental pour sécuriser la valeur ajoutée perçue et réelle d'un logiciel.

1. Définition des recettes

1.1 La recette informatique en quelques mots

- Si le terme "recette" renvoie à une définition à caractère monétaire, ce n'est pas le cas dans notre contexte. Le recettage d'une application, est une étape importante dans la conception et le déploiement d'un logiciel.
- Le recettage permet de **corriger** les remarques faites par le **client** ou les imperfections remarquées lors du test d'acceptation
- On appelle « **recette** » (ou **essais de réception**) la vérification de la **conformité** de produit logiciel à la **demande** formulée dans le dossier validé de conception générale. La recette est un processus rigoureux et méthodologique effectué dès la réception de la commande. Elle est réalisée conformément au dossier de contrôle établi par la maîtrise d'ouvrage, rassemblant les documents définissant la façon de laquelle le logiciel doit être contrôlé. La recette est parfois dite **provisoire pendant une période** de temps appelée délai de garantie. Cette étape se conclue lors de l'expiration du délai de garantie par la rédaction d'un dossier de recette (procès-verbal de réception définitive) cosigné

par le maître d'œuvre et le maître d'ouvrage contenant les remarques du maître d'ouvrage et éventuellement le refus de l'ouvrage s'il n'est pas conforme au cahier des charges.

- Le recettage, ou recette informatique est une **succession de tests** et de vérifications de toutes les fonctionnalités implémentées sur une application. En découle une phase de correction afin que tout soit en accord avec les attentes définies dans le cahier des charges. C'est une étape très importante dans la gestion de projet.

Après que l'agence ait développé votre application et saisit les contenus, elle doit s'assurer que tout est conforme à ce qui figure dans le cahier des charges. Pour ce faire, elle doit tester et vérifier le **design, le développement, l'intégration, les balises HTML**, etc. C'est à cela que consiste la recette informatique. Cette étape permet d'examiner le degré de **conformité** entre ce qui a été défini en amont (dans le cahier des charges ou dans les maquettes et spécifications fonctionnelles) et la réalité.

Plus le nombre d'acteurs qui traitera la recette sera **élevé**, plus cette dernière sera meilleure ! Il sera question ici de prévenir de potentiels problèmes **d'ergonomie**, bref de déceler d'autres **imperfections**, d'avoir un autre point de **vue**. Veillez toutefois à respecter les **délais** de livraison communiqués au client.

Exemple : La recette site internet consiste également à vérifier la conformité aux maquettes, aux styles et à l'ergonomie. Pour ce faire, il convient de tester le site en question sur les différents terminaux (smartphone, laptop, etc.) et navigateurs (car des bugs peuvent exister sur Edge et non sur Google, et inversement). Vérifiez tous les points : affichage des menus, etc. Lorsque des éléments à corriger ont été détectés, il est important de les rassembler grâce à l'utilisation d'un outil de recette. Ce dernier a l'avantage de conserver l'historique des demandes, de suivre leur résolution et de les classer par ordre de priorité.

L'acceptation de l'utilisateur est effectuée lorsque les **autres tests**, tels que les tests fonctionnels, solution, performance et robustesse, **ont déjà été réalisés**. Elle représente la dernière étape du processus de test avant qu'un produit ne soit mis en production. Les tests d'acceptation des utilisateurs permettent de démontrer que les fonctions requises d'un objet de test fonctionnent de manière adaptée à l'utilisation du monde réel. Par conséquent, les tests d'acceptation des utilisateurs concernent à la fois les **performances** du logiciel et le comportement **humain**. Il est essentiel que les fonctionnalités du logiciel de test et l'intuition de l'utilisateur final soient en harmonie. En pratique, les testeurs observent et documentent ce

qui se passe lorsqu'ils essaient d'exécuter des tâches telles que placer un article dans un panier ou se connecter à un compte client.

1.2 Objectif : validation finale du produit.

La recette est une tâche qui doit être faite avec le plus grand soin. Le client sera consulté pour réaliser un test d'acceptation pour s'assurer de la conformité de l'application. Dès lors que le résultat conviendra à ses attentes, il passera à la validation du projet auprès de son prestataire. Le client **attestera** ainsi du bon fonctionnement du logiciel et de ses fonctionnalités, et **validera** la livraison du projet. Ce dernier pourra donc être déployé dans son environnement réel.

Le recettage permet de **corriger** les remarques faites par le client ou les imperfections remarquées lors du test d'acceptation. Cela permet de mettre déployer un logiciel 100% **fonctionnel**, répondant aux besoins des utilisateurs. Par conséquent, la recette est une étape essentielle au sein d'un projet logiciel.

Le logiciel lui-même est intégré dans l'environnement extérieur (autres logiciels, utilisateurs). On vérifie que le logiciel développé répond aux besoins exprimés dans la phase de spécifications des besoins. Cette phase qui représente environ 10% du temps total consacré au développement se termine par la revue finale. Les documents produits au cours de cette phase sont: **Résultats de la recette**

Le cahier de recette, ou cahier de test, est un document présentant les éléments à vérifier pour mener à bien la mise en production d'un produit numérique.

Le test d'acceptation de l'utilisateur, selon un cahier de recette informatique, est important car il aide à démontrer que le logiciel de test fonctionne de **manière adaptée au monde réel**. Si le résultat attendu n'est pas atteint pendant les tests, l'élément sera documenté et **renvoyé** aux développeurs pour optimisation.

Ce processus sert de vérification **finale** pour assurer que le produit est **bien développé** pour les utilisateurs, ce n'est pas parce que le test d'acceptation de l'utilisateur est le test final effectué avant la mise en production qu'il n'y a pas de bugs découverts ! Ces bugs peuvent **ruiner** l'expérience utilisateur et même rendre le logiciel **inutilisable**.

En raison de ces enjeux, les avantages des tests d'acceptation de l'utilisateur sont nettement supérieurs aux coûts. Les tests d'acceptation des utilisateurs ne prennent généralement que 5 à 10% du temps du projet, mais ils peuvent économiser près de 30% des coûts totaux (comparé à

ne pas effectuer de tests du tout). C'est un bon moyen pour assurer un bon retour sur investissement des projets.

Chaque anomalie, ou bug, est synonyme **d'insatisfaction** chez l'utilisateur et de perte de revenus pour la marque. Les utilisateurs ne devraient pas être ceux qui découvrent des bugs de logiciels ! Il y a plusieurs façons de conduire ces tests : vous pouvez lancer une campagne en laboratoire, ou bien par du **crowd testing**. Dans tous les cas, ce sont bien des humains qui doivent faire ces tests, versus des **tests automatisés**.

1.3 Recette fonctionnelle et techniques

En informatique, la recette (ou test d'acceptation) est une phase de développement des projets, visant à assurer formellement que le produit est conforme aux spécifications. Elle s'inscrit dans les activités plus générales de qualification.

Dans ce cas des nouveaux termes sont utilisés :

- **VA** : La Vérification d'Aptitude intervient après la mise en ordre de marche. Elle a pour objet de constater que les prestations (services), livrées ou exécutées, présentent les caractéristiques techniques qui les rendent aptes à remplir les fonctions précisées dans les documents particuliers du marché. Il s'agit principalement de matériels ou de logiciels et de prestations associées (installation, paramétrage, reprise de données, maintenance, …). Cette constatation peut aussi résulter de l'exécution, dans les conditions fixées par le marché, d'un ou de plusieurs programmes ou **bancs d'essais.**
- **VSR** : La Vérification de Service Régulier (VSR) intervient après la vérification d'aptitude qui succède à la mise en ordre de marche. Elle constitue la dernière étape des opérations de vérifications qualitatives. La VSR ne peut débuter que dans l'hypothèse où la vérification d'aptitude est **positive**. Elle a pour objet de constater que les prestations fournies sont capables d'assurer un service régulier dans les conditions normales d'exploitation prévues dans les documents particuliers du marché.

Si le résultat de la vérification de service régulier est négatif, le pouvoir adjudicateur prend une décision **écrite** qu'il notifie au titulaire, soit :

- d'ajournement avec vérification de la régularité de service pendant une période supplémentaire maximale d'un mois ;

- de réception avec réfaction qui consiste à **réduire le montant** des prestations à verser au titulaire, lorsque les prestations ne satisfont pas entièrement aux prescriptions du marché, mais qu'elles peuvent être reçues en l'état.
- de rejet qui est la décision prise par le pouvoir adjudicateur qui estime que les prestations ne peuvent être reçues, même après ajournement ou avec réfaction.

- **VABF :** La VABF (pour « **vérification d'aptitude au bon fonctionnement** ») est une phase d'un projet informatique, qui précède généralement la mise en production, pendant laquelle le client et son prestataire informatique vérifient, sur la base de scénarios de tests prédéfinis (par le client ou par le prestataire) la conformité de la solution informatique objet du contrat à ses spécifications contractuelles, dans des conditions de test. Généralement la VABF (qu'on appelle souvent « **recette provisoire** ») a donc lieu sur un environnement de test avec des données de test.

- **La maîtrise d'ouvrage**, ou MOA, correspond à l'entité organisatrice d'un projet. La MOA désigne une personne morale dont la principale mission est de conduire la réalisation d'un ouvrage en organisant, notamment, des comités de pilotage. La MOA est le commanditaire, le **client**. Il connait le besoin métier mais **ne sait pas** réaliser le système

- **Le maître d'œuvre (MOE)** est le prestataire qui **construit le système sur le terrain.**

Lors de l'étape de **vérification d'aptitude (VA)** ou vérification d'aptitude au bon fonctionnement (**VABF**) (aptitude à répondre aux besoins exprimés dans le cahier des charges initial) ou recette utilisateur, le client réalise deux catégories de tests différentes.

D'un côté, **une recette technique** est effectuée afin de vérifier que le produit livré est techniquement conforme sur toute la chaîne de processus. De l'autre, la maîtrise d'ouvrage contrôle l'aspect fonctionnel du produit lors de **la recette fonctionnelle.**

- **La recette fonctionnelle :** a pour but la validation des fonctionnalités exprimées dans le cahier des charges et détaillées dans les spécifications fonctionnelles. La MOA procède donc à sa propre série de tests de validation. Les tests techniques, unitaires et d'intégration doivent être complétés par des tests dits « fonctionnels » pour assurer une qualité optimale des projets informatiques.
- **La recette technique :** permet de contrôler les caractéristiques techniques du produit livré, la recette technique regroupe les tests suivants :

- les tests d'exploitabilité et en particulier le respect des exigences d'architecture technique et les tests de performance. - La VSR.
- Si la VABF se déroule correctement et est validée, le client procède alors à la mise en service opérationnelle.

Une période de vérification de service régulier (VSR) commence donc par un premier déploiement. Cette mise en production permet de valider le produit en conditions réelles.
À la différence des étapes précédentes, celle ci se déroule pleinement en environnement de production avec des données réelles.

- Les documents livrables sont les documents qui accompagnent la procédure de recette: Le protocole de recette:
- Le protocole de recette est un document visant à clarifier intégralement la procédure de recette. Il précise scrupuleusement :
- les tâches du client, celles du fournisseur, la liste des documents à communiquer, l'ordre des tests et le planning, les seuils d'acceptation du produit.
- Le cahier de recette[modifier:
- Le cahier de recette est la liste exhaustive de tous les tests pratiqués par le fournisseur avant la livraison du produit.
- La couverture des tests, en particulier ceux de non-régression lorsqu'il s'agit d'une nouvelle version d'un produit existant, pouvant être infinie, le cahier de recette doit préciser toutes les fiches de test passées par le fournisseur, ainsi que celles à passer dans l'environnement du client lors de la VABF.

2. Processus de recette

Étape 1 : Utiliser le retour d'expérience de projets similaires

Si vous avez collaboré à un projet similaire ou êtes en mesure d'accéder aux « lessons learned » de projets comparables, vous êtes en présence d'une **mine d'or**. Cela vous épargnera du temps de préparation et la répétition d'erreurs.

Ce **retour d'expérience** vous permettra de recueillir les défis et les problèmes rencontrés lors des tests d'acceptation utilisateur de projets similaires. Certaines organisations mettent systématiquement en œuvre un processus destiné à recueillir ces informations post-projet, et vous n'aurez donc qu'à les consulter. Dans la plupart des cas, en revanche, de tels documents n'existent pas et il s'agira d'aller à la rencontre des collaborateurs et chefs de projets pour identifier ce qu'ils ont retiré en positif comme en négatif de la phase de recette.

Il faut pas oublier que la même histoire se répète bien souvent : par exemple, si d'autres projets ont du faire face à des difficultés pour obtenir des ressources pour exécuter les tests.

Étape 2 : Décider de ce qu'il faut tester

Les tests d'acceptation utilisateur ont pour objectif de vérifier que le système répond aux besoins des utilisateurs et du client dans le contexte opérationnel réel d'utilisation. Dans cette seconde étape, il s'agit donc de déterminer précisément ce qui doit être testé dans la limite des **processus métier** mis en œuvre. Certains points doivent être particulièrement analysés :

a. **Les interfaces :** Les **interfaces** entre les différents systèmes, ainsi que le passage de relai entre le système informatique et les processus manuels gérés par les utilisateurs. C'est souvent à ces points de passage que l'on rencontre le plus de problèmes.
b. **Les principaux risques métier :** La business analyse doit tenir compte des risques métiers encourus lors de la mise en œuvre d'un changement. Reprenez le registre des risques (pas celui du projet, mais bien celui des risques métier !) et analysez l'impact du changement sur les domaines métier concernés. Tester le système en partant de cette réflexion permet de mettre en lumière des dysfonctionnements qui auraient pu passer inaperçus lors d'un processus de vérification standard.
c. **Le processus métier « end-to-end » :** L'acceptance doit, si possible, tester la manière dont le système impacte les processus métier du début à la fin. Ce n'est pas toujours possible pendant le projet, notamment dans le cadre du gestion de projet agile, mais il faudra néanmoins le faire à un moment donné.
d. **L'analyse d'impact :** L'objectif recherché est d'identifier dans quelle mesure ne pas tester certaines fonctionnalités aurait un impact sur le projet. Parfois, faire l'impasse sur certaines parties du système d'information est moins coûteux ou évite des retards. Les anomalies éventuelles seraient-elles acceptables pour l'entreprise ? L'analyse d'impact permet de prioriser les tests d'acceptation utilisateur.
e. **Les alternatives envisageables :** Ici, l'objectif est d'identifier les autres possibilités de répondre aux besoins métiers si le système informatique est défaillant. En d'autres termes, il s'agit de réfléchir à un fonctionnement du processus métier en mode **dégradé**, afin de prioriser les tests d'acceptance en fonction de leur criticité sur l'activité de l'organisation.

Étape 3 : Préparer la logistique du processus des tests d'acceptance

Je l'ai évoqué dans cet article : la phase de test ne consiste pas uniquement à exécuter des tests, elle s'appuie sur un ensemble d'activités dont la partie « logistique » est un élément majeur. Voici quelques points-clé à ne pas négliger :

a. **Le budget :** Le budget de la phase de recette utilisateur comporte des frais annexes qui peuvent faire perdre du temps si vous ne les anticipez pas. Par exemple :

- Location de salle
- Matériel informatique
- Frais d'impression du matériel de formation
- Dépenses / notes de frais des testeurs
- Contrat de prestation externe etc…

b. **La mobilisation des bonnes personnes :** Mobilisation de *toutes* les bonnes personnes: ce sont les utilisateurs finaux, ceux qui réalisent ou réaliseront les tâches testées qui sont les plus à même de valider le système cible. Cependant, il peut également s'avérer judicieux de faire intervenir d'autres profils :

- **Les collaborateurs influents :** Très utiles dans une perspective de conduite du changement, pour obtenir le soutien des principales parties prenantes de votre projet.

- **Les défenseurs du nouveau processus :** Ce sont eux qui travailleront souvent le plus dur pour s'assurer que le logiciel réponde aux exigences.

- **Les opposants au projet :** À « manipuler » avec précaution ! Ils s'attacheront à démontrer à quel point le nouveau système ne répond pas à leurs besoins en testant tous les cas « aux limites » que vous n'auriez sans doute pas trouvés sans cela…

Ce sont eux également qui vous souffleront les améliorations à apporter à vos supports de formation et autres guides utilisateurs.

On ne va pas se mentir, ce type de testeurs est délicat à maîtriser, et il faut au Business Analyst une bonne dose de « soft skills » pour ne pas se faire trancher en deux par leur sabre laser.

Cela étant, vous pourriez être agréablement surpris par les avantages que leur intervention procure en phase d'acceptance.

Sans compter que certains d'entre eux peuvent même se transformer en ardents défenseurs du nouveau système d'information, au fur et à mesure qu'ils en découvrent les avantages ou lorsqu'ils se rendent compte qu'on est à l'écoute de leurs résistances.

c. **Pilotage de la recette utilisateur :** En plus de faire appel aux bonnes personnes pour effectuer les tests, réfléchissez à la manière dont vous allez piloter la campagne de recette utilisateur. Lieu de déroulement des tests, processus de traitement des anomalies constatées, ou encore formation de l'équipe de testeurs, tout cela doit être préparé avec soin.
d. **Documentation de support de tests :** Réfléchissez en amont à la documentation de référence des tests dont votre équipe aura besoin. Par exemple : les cas de tests, procédures métier, cartographie des processus métier, spécifications fonctionnelles, documents de formation etc…

Étape 4 : Définir la durée des tests d'acceptation

Chaque projet est différent.

Dans certains cas, il est possible de mener à bien rapidement et de manière assez superficielle la phase de test avec un petit groupe de « key users ».

Dans d'autres, en revanche, vous aurez peut-être besoin de planifier la recette client de manière plus approfondie sur une plus longue période de temps.

a. **Définir les objectifs des tests :** Pour déterminer ce qui convient le mieux à votre projet, bien entendu vous pouvez vous baser sur les étapes décrites précédemment, notamment le retour d'expérience, mais il est également nécessaire de prendre du recul sur les **objectifs** de vos tests.

En effet, l'un des bénéfices de la recette utilisateur est de **donner confiance dans le nouveau système** et pas seulement d'identifier les anomalies. On peut donc tout-à-fait inclure les tests d'acceptance dans une démarche globale de conduite du changement, pour par exemple obtenir le soutien des principales parties prenantes.

b. **Préparer et former les testeurs :** Prévoyez également du temps pour préparer et former vos testeurs. Sinon, ils risquent de se concentrer sur le fonctionnement du nouveau système ou du nouveau processus métier, au lieu de rechercher des anomalies techniques.
c. **Prendre le temps d'effectuer les tests :** Prévoyez également suffisamment de temps et de ressources pour effectuer les tests, apporter les modifications nécessaires, tester à nouveau les révisions et prévoir un plan d'urgence.

D'autre part, comme ces tests d'acceptance ont généralement lieu vers la fin du projet ou du sprint pour ce qui est des approches agiles, il peut y avoir une pression exercée afin de réduire la durée de cette phase.

Soyez donc clair et transparent sur votre approche méthodologique, pour éviter que l'on ne vous reproche d'être passé à côté de défauts majeurs.

d. **Assurer la disponibilité des testeurs :** Assurez-vous de la disponibilité de vos testeurs pour la phase de recette. N'oubliez pas que ce sont des experts métiers qui seront vos testeurs, or il est souvent difficile de faire libérer ces personnes de leurs activités habituelles pendant les vacances ou encore, lors d'échéances métier (clôture comptable etc).
e. **Planifier la mise à jour des supports de formation :** Assurez-vous également d'avoir planifié une activité de mise à jour des supports de formation, de la documentation et des procédures destinées aux utilisateurs APRES la fin de l'UAT. Cela permettra de garantir l'exactitude de ce matériel une fois les tests terminés.

Étape 5 : Mettre en place le pilotage des tests d'acceptation utilisateur

Enfin, dernière étape :

- **Identifier les indicateurs de suivi** de la qualité et de la progression des tests d'acceptance.
- **Mettre en place les outils de collecte statistique** pendant le déroulement de la phase de test : nombre d'anomalies constatées, résolues, etc. Ce travail ne se fait pas seul dans son coin, mais en collaboration avec le sponsor et les autres parties prenantes (y compris la maîtrise d'œuvre).

En raison des enjeux commerciaux et stratégiques parfois opposés entre la maîtrise d'ouvrage et la maîtrise d'œuvre, cette réflexion n'est pas forcément aisée, et la recherche de compromis est parfois nécessaire pour équilibrer ces différents objectifs.

3. Exécution et suivi des recettes

Pour une recette fonctionnelle optimale, il est important de disposer de données « réelles », c'est à dire représentatives de ce qui sera rencontré en production. Une stratégie souvent utilisée est de « redescendre » des données de production existantes sur les environnements de test, et de les compléter avec les données manquantes (c'est à dire concernant des fonctionnalités nouvelles).

3.1 Exécution des scénarios de tests

Après avoir mené à bien un processus de préparation et de conception approfondi, il convient de passer au processus d'exécution. Il s'agit d'exécuter le test d'acceptation par **l'utilisateur** au fur et à mesure et de signaler tout bogue survenu au cours du test, y compris le moment où le bogue s'est produit, le message que le logiciel a envoyé en réponse et ce qui a provoqué le problème. Les outils de gestion des tests peuvent **automatiser** ce processus d'exécution dans certains cas. Répétez les tests dans la mesure du possible pour vous assurer que les résultats obtenus sont **fiables**.

Quand le cahier de recette est fourni, que l'environnement de test est opérationnel et doté de jeux de données pertinents, et enfin (surtout) que les développements ont été réalisés, les tests fonctionnels peuvent démarrer.

L'exécution des tests est généralement mixte : manuel et automatisé. Les tests manuels sont exécutés par une équipe de recette voire directement par la maîtrise d'ouvrage, et les tests automatisés sont exécutés par des outils dédiés.

La recette doit être industrialisée, c'est à dire reposer sur des process clairement établis et documentés, et permettre de produire des résultats (rapports de test, création d'entrées dans un outil de ticketting, etc.) à partir d'un ensemble de données en entrée (cas de tests notamment).

Voici une liste non-exhaustive des outils disponibles : Monday, ClickUP, Notion, Squash TA, SilkTest

3.2 Gestion des anomalies

Types d'erreurs et de bogues détectés lors des tests d'acceptation par l'utilisateur

Les tests UAT sont confrontés à différents types de bogues. Comme vous effectuez les tests UAT à la fin de la phase de développement, ces erreurs sont généralement plus mineures que celles qui se produisent au début du processus, notamment :

1. Erreurs visuelles

Les erreurs visuelles se produisent lorsque l'apparence du logiciel est différente de celle prévue par l'utilisateur (du **point de vue de l'interface utilisateur**, par exemple), avec des graphiques qui ne se chargent pas ou qui se chargent de manière incorrecte.
Cela affecte la façon dont les gens interagissent avec l'application et c'est une caractéristique que les développeurs cherchent à corriger avant la publication afin d'améliorer l'expérience de l'utilisateur.

2. Questions relatives aux performances

Les problèmes de performance se produisent lorsque le logiciel accomplit toutes ses tâches, mais de manière inefficace. Ces inefficacités comprennent le fait de nécessiter plus de ressources qu'il n'en faudrait ou de prendre plus de temps que la normale pour accomplir des tâches simples.
Les développeurs y apportent des correctifs d'optimisation à un stade ultérieur du processus.

3. Processus échoués

C'est le cas lorsqu'un processus échoue complètement ou atteint ses objectifs de manière imprécise. Ces problèmes démontrent une faille fondamentale dans le code et nécessitent une réponse de la part des développeurs pour que le logiciel fonctionne à nouveau correctement.

Mesures communes de l'UAT

Lorsqu'une entreprise reçoit des données mesurables en réponse à ses tests UAT, ces données se présentent sous différentes formes. N'oubliez pas que les mesures ne suffisent pas à rendre compte de la situation, et comprenez ce que les utilisateurs pensent du produit et pourquoi, grâce à des discussions approfondies.
Voici quelques-unes des mesures UAT les plus courantes utilisées par les entreprises :

1. Totaux PASS/FAIL

Le nombre total de résultats positifs ou négatifs que vous atteignez dans un test automatisé. Il mesure le nombre d'erreurs qui se produisent, et le suivi de cette mesure vous permet de savoir si les mises à jour ultérieures ont permis de réduire le nombre total d'erreurs.

2. Couverture de l'exécution des tests

Une valeur en pourcentage qui vous indique la proportion du code qui a été testée par votre régime de test UAT.

Des pourcentages plus élevés correspondent à des tests plus complets, une couverture de 100 % garantissant que l'ensemble du code est fonctionnel.

3. Satisfaction des clients

L'UAT est l'étape à laquelle les clients interagissent avec un produit et il est primordial de comprendre leurs sentiments. Demandez aux testeurs leur degré de satisfaction sur une échelle de un à dix, obtenez une moyenne, puis répétez les tests avec les mêmes personnes après les mises à jour, l'objectif étant d'obtenir un degré de satisfaction plus élevé.

4. Livraison et acceptation du produit

La phase de recette est constituée de plusieurs sessions de test au cours desquelles on doit réaliser les tests décrits dans les cahiers de test. Le résultat de ces sessions est conservé sous la forme de rapports de test. Un rapport de test indique pour chaque étape d'un scénario si le résultat attendu a bien été observé. Sinon, le testeur a la possibilité de rapporter le résultat observé afin de fournir une indication sur l'écart par rapport au résultat attendu.

L'utilisation d'un outil informatique permet de créer facilement des rapports de test en proposant au testeur de valider ou non un scénario de test étape par étape. Dans le cas d'une non validation, le testeur peut immédiatement décrire le comportement observé.

Pour le développement de solution de tests automatisés, il existe de nombreuses bibliothèques de programmation qui permettent de produire automatiquement le rapport d'exécution des tests.

4.1 Présentation des résultats

Dans la livraison de divers ajustements du système habituellement d'abord un test d'acceptation endroit. Nous devons enregistrer les résultats de ces tests dans un rapport de test. Le rapport est destiné au client intéressé ou au propriétaire de l'application. Pour obtenir les

résultats des différents test pour pouvoir bien comparer et garantir que nous avons prêté attention à tous les aspects importants, il existe un cadre fixe pour la rédaction d'un rapport de test de réception.

Il faut indiquer dans quelle mesure nous avons pu mettre en œuvre le plan de test et pour quelles raisons le plan de test s'est écarté.

Il faut également indiquer les résultats pour chaque cas de test.

Il faut enregistrer tout cela dans un journal de bord afin que les activités soient visibles par jour et par objet de test. De cette manière, indiquer clairement quels efforts étaient nécessaires pour résoudre certains problèmes.

Voir la date à laquelle le problème est résolu à côté de la date d'observation.

Si des cas de test sont nécessaires pour effectuer le test, il faut le préciser. Il faut rapporter également leur diversité et les résultats par cas de test. La méthode utilisée pour déterminer les cas de test doit également être clairement définie.

Il faut brièvement citer les conclusions et les recommandations les plus importantes sur la base des résultats du test.

informations signaler dans le rapport de test

- ***Affectation / numéro de problème*****:** Le numéro du travail ou le numéro du problème correspond aux numéros attribués au projet ou à la version. En copiant ce numéro, un lien est créé entre le résultat du test et l'affectation sous laquelle le résultat a été obtenu.

- ***Équipe de test*****:** Il est important de toujours indiquer qui a effectué le test. En effet, le rapport peut comporter des incertitudes sur lesquelles nous devrions poser des questions. L'équipe de test peut être composée d'une ou plusieurs personnes.

- ***(Re) numéro de test*****:** Dans les systèmes complexes, il s'avère souvent que les modules du programme ne réussissent pas bien le test en une seule fois. Nous rejetons ensuite l'objet de test. En émettant maintenant un nouveau numéro de test pour chaque test de la commande, nous pouvons déterminer la fréquence à laquelle un module de programme a été rejeté lors des tests d'acceptation. Si cela se produit souvent, nous

pouvons conclure que quelque chose ne va pas. Par exemple, il peut y avoir un problème de communication ou la qualité du logiciel peut être médiocre.

- ***Objets de test*:** Pour chaque objet de test (généralement un module de programme), nous spécifions le nom, indiquons le type de module, enregistrons la date du test, marquons l'effet attendu et enregistrons le résultat.

- ***Résultats*:** Avec les résultats, nous ne notons pas seulement les écarts, mais nous indiquons également si l'opération est conforme aux attentes. De cette manière, il est possible de déterminer si le test a été entièrement réalisé et s'il n'a pas été uniquement testé pour les points modifiés.

4.2 Types de résultats des tests d'acceptation par l'utilisateur

Les différentes formes de tests UAT produisent des résultats et des formats de données uniques. Voici quelques-uns des principaux types de résultats que vous pouvez obtenir à l'issue des tests UAT :

1. **Retour d'information écrit** : Les développeurs reçoivent un retour d'information écrit de la part des testeurs lorsqu'ils réalisent les tests d'acceptation par les utilisateurs. Ces données sont relativement difficiles à analyser car il s'agit d'informations qualitatives plutôt que quantitatives, ce qui signifie que les réponses sont plus nuancées.

2. Messages d'erreur : Certains tests renvoient des messages d'erreur qui indiquent ce qui n'a pas fonctionné dans le processus de test et pourquoi. Les développeurs créent une structure de messages d'erreur qui les informe de la nature et de l'origine du problème, ce qui les aide à trouver une solution potentielle à l'avenir.

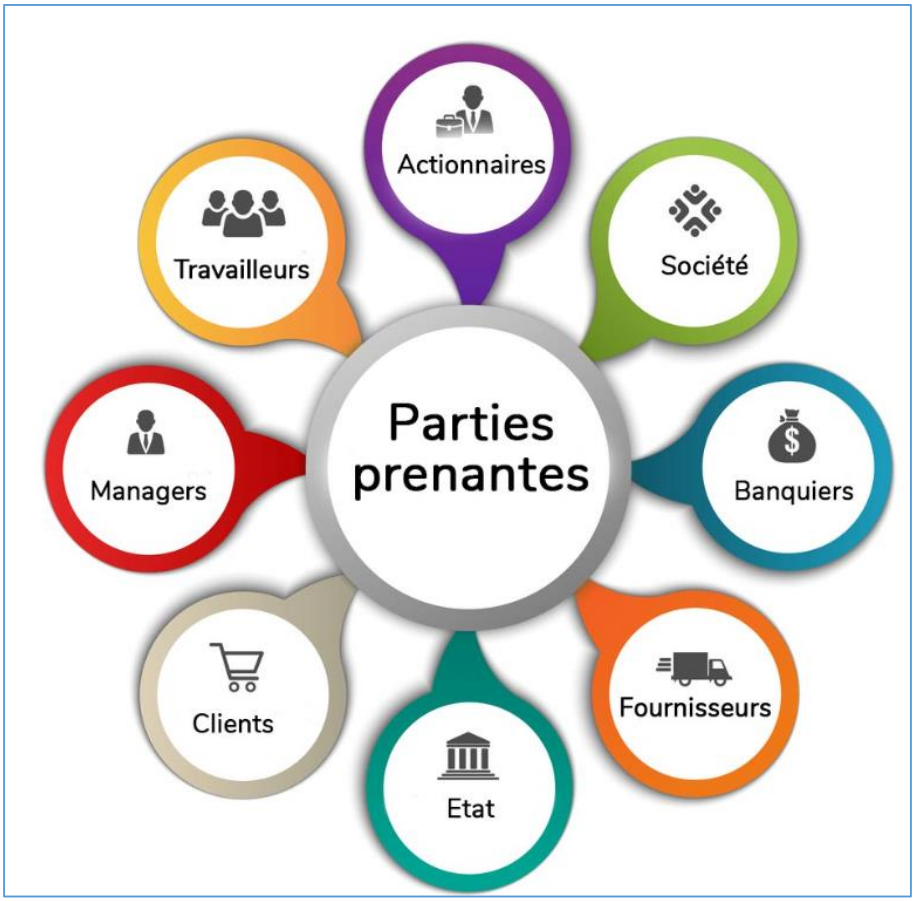
3. **Données** : Les données numériques constituent une autre forme de sortie, y compris le nombre d'erreurs trouvées par un test, la latence entre les entrées de l'utilisateur et les réponses du programme et d'autres chiffres directement liés au travail effectué par l'application. Ces informations permettent d'effectuer des analyses et des révisions après les tests.

4.3 Validation par les parties prenantes

Les parties prenantes sont les personnes qui ont un intérêt quelconque dans le résultat de votre projet. Il s'agit généralement des membres de l'équipe du projet, du chef de projet, des cadres, du promoteur du projet, des clients et des utilisateurs. Pour ce faire il faut suivre les étapes suivantes :

- Préparer la liste complète des **parties prenantes**;

- Identifier les **attentes** et les rôles des **parties prenantes**;
- Développer votre leadership dans le but de maintenir l'engagement des **parties prenantes** conformément aux objectifs du projet;



5. Parties impliqué dans les tests d'acceptation par l'utilisateur

Plusieurs parties sont impliquées dans le processus de test d'acceptation par l'utilisateur, chacune ayant son propre rôle et ses propres responsabilités tout au long du processus. Parmi les personnes les plus importantes jouant un rôle dans le processus UAT, on peut citer

5.1 Développeurs : Les développeurs de l'application compilent la version la plus récente du logiciel et l'envoient aux testeurs, puis apportent les modifications nécessaires une fois que les résultats des tests sont connus.

5.2 Testeurs : Les testeurs sont généralement des personnes qui utiliseront le logiciel, soit dans le cadre de leur travail, soit en tant que passe-temps. Ils examinent toutes les fonctionnalités du logiciel dans le cadre d'une série de tests planifiés à l'avance, avant de communiquer leurs résultats à l'entreprise.

5.3 Gestionnaires : Le personnel d'encadrement organise la collaboration avec les testeurs, en plus de fournir une liste d'exigences pour le test UAT et, dans certains cas, de mener à bien les processus de planification et de préparation des tests.

5.4 Expert du domaine : Dans la mesure du possible, faites appel à un "expert du domaine", ou à une personne possédant des compétences pertinentes dans le domaine, pour effectuer les tests d'acceptation des utilisateurs aux côtés des utilisateurs finaux et fournir des détails supplémentaires lorsqu'ils signalent des problèmes à l'équipe de développement.

	Fonctionnalités	Actions	Résultats
1	Création de compte	Je clique sur le bouton "Création de compte" et j'accède à un formulaire pour renseigner mes informations personnelles	OK : accès au formulaire
2	Mot de passe oublié	Je clique sur le bouton "Mot de passe oublié" afin d'obtenir un lien de connexion en renseignant mon adresse mail	Erreur : rien ne se passe
3	Message "Produit ajouté au panier"	Je clique sur le bouton "Ajouter au panier", l'article se retrouve dans le panier et un message de confirmation "Votre article a bien été ajouté au panier" s'affiche	Ok : le message s'affiche
4	Proposition de livraison en point relais	Dans la rubrique Adresse de livraison, je choisis "Livraison en point relais" et je renseigne mon code postal pour trouver le point relais le plus proche	OK : accès aux points relais disponibles

Le client peut également prendre part à la recette informatique de son projet : on parle de recette utilisateur. Dans certaines agences, le client à accès à l'application avant la date de livraison. Il peut alors tester et retester son application et faire remonter des choses que l'équipe de développement n'a peut-être pas remarqué. Cela peut aussi être fait après la livraison.

6. Les avantages et désavantages à faire du recettage

6.1 Avantages de l'exécution manuelle des tests d'acceptation par l'utilisateur

La réalisation manuelle de tests UAT présente de nombreux avantages, en fonction de la nature de logiciel et de la structure de l'entreprise dans laquelle vous travaillez. Les principaux avantages de la réalisation manuelle des tests UAT par rapport à l'utilisation d'outils d'automatisation sont les suivants :

Réaliser des tests plus complexes : Le premier avantage des tests manuels est la possibilité de réaliser des tests plus complexes qu'avec un outil de test automatisé. L'automatisation implique l'intégration de tests dans le logiciel, ce qui peut signifier que les tests plus complexes prennent plus de temps, car l'équipe écrit de longues chaînes de code pour examiner des problèmes détaillés. Les tests manuels ne nécessitent pas d'exigences de codage aussi complexes, le testeur entrant dans le logiciel et effectuant le test après avoir reçu des instructions, ce qui simplifie considérablement le rôle de l'équipe chargée des tests.

Intégrer les tests d'interface utilisateur et de convivialité : Lorsque vous expédiez un logiciel complet, vous devez tenir compte d'un grand nombre d'éléments autres que la simple fonctionnalité. Alors que les tests automatisés peuvent fournir des informations exclusives sur la fonctionnalité d'un logiciel, les testeurs manuels ont l'avantage de réagir à des éléments que les utilisateurs humains remarqueront. Il s'agit notamment d'informer les développeurs des problèmes potentiels liés à l'interface utilisateur du logiciel, de recommander des modifications de la police de caractères utilisée par le site et de comprendre les problèmes liés au flux de travail que les utilisateurs doivent suivre. Ce type de retour d'information de la part des utilisateurs du manuel contribue à rendre le site plus convivial, plutôt que de se contenter d'offrir la fonctionnalité.

Identifier des questions plus spécifiques : Les tests automatisés sont conçus pour suivre un script très spécifique et déterminer si un logiciel fonctionne ou non, mais cela signifie qu'il n'y a pas de place pour les détails. Les testeurs manuels d'acceptation par l'utilisateur peuvent fournir une identification plus spécifique des problèmes et des défauts dans le programme, ce qui est contraire au système plus binaire de PASS/FAIL d'un système automatisé. Grâce à ce retour d'information détaillé, les développeurs savent exactement où le problème s'est produit et peuvent le résoudre beaucoup plus rapidement qu'ils ne l'auraient fait autrement, ce qui accroît la réactivité de l'entreprise et permet aux clients d'obtenir de meilleurs résultats plus rapidement.

Fournir des réponses plus nuancées : L'utilisation d'un processus de test UAT manuel signifie que vous obtenez des réponses plus nuancées que lorsque vous utilisez des tests automatisés.
La première chose à faire est d'examiner l'image de marque du logiciel et toute capacité potentielle d'amélioration des intégrations avec des logiciels externes, car il s'agit d'un aspect qu'un test automatisé n'a pas été conçu pour prendre en compte.

En outre, un testeur humain peut générer des rapports ad hoc sur la façon dont un flux de travail est ressenti, offrant des conseils et des recommandations spécifiques plutôt qu'une équipe d'assurance qualité examinant les données générées par un test automatisé UAT et émettant des hypothèses sur la base de ces informations.

Travailler de manière plus flexible dans le domaine des essais : La flexibilité est un élément fondamental des tests, et c'est un domaine dans lequel l'utilisation d'un testeur manuel excelle. Il y aura toujours quelque chose qu'un développeur ou une équipe d'assurance qualité n'aura pas pris en compte lors de la création de ses tests, comme l'utilisation d'un logiciel d'une manière particulière ou une fonctionnalité ayant plusieurs fonctions inattendues.
Un testeur UAT manuel qui interagit avec le logiciel de manière inattendue fait apparaître des bogues et des problèmes que les développeurs n'ont peut-être pas envisagés, ce qui les aide à corriger des aspects du logiciel qu'ils n'avaient peut-être même pas envisagés.
C'est d'autant plus important que l'exposition à un plus grand nombre d'utilisateurs signifie que ces utilisations innovantes des fonctions sont presque certaines d'être trouvées après le lancement public.

6.2 Les défis de l'UAT manuel

Il y a plusieurs défis à relever lorsque l'on envisage un test UAT manuel. Résoudre ces problèmes et chercher activement à les atténuer est une nécessité pour quiconque souhaite lancer des tests manuels sans se heurter à des obstacles importants tout au long du processus.

Voici quelques-uns des principaux défis que pose la mise en œuvre de l'UAT manuelle dans les processus de test :

Coût financier plus élevé : L'un des inconvénients des tests manuels par rapport aux tests UAT automatisés est que le coût financier des tests manuels est beaucoup plus élevé. Chaque test manuel nécessite l'intervention d'un membre du personnel rémunéré, et les tests les plus fiables sont ceux que vous effectuez à plusieurs reprises afin d'obtenir des résultats plus cohérents.
C'est beaucoup d'argent que vous devez investir dans vos processus d'assurance qualité.
Le coût augmente encore si l'on tient compte du fait que les résultats des tests sont plus précis lorsqu'ils sont effectués par des membres du personnel ayant un niveau de compétence plus élevé, et que le recrutement de ces employés coûte encore plus cher. Les tests manuels

d'acceptation par l'utilisateur ne sont pas la solution la plus abordable pour de nombreuses entreprises.

Exigences élevées en matière de compétences techniques : Les tests UAT manuels sont un domaine qui nécessite un degré élevé d'interaction avec les logiciels et les services spécifiques, avec l'expertise nécessaire pour comprendre d'où les problèmes sont susceptibles de provenir et pour recommander des réponses potentielles à ces problèmes.
Dans ce cas, il est utile d'avoir des testeurs manuels ayant un haut niveau d'expertise dans l'exécution des tâches d'assurance qualité, comme un « expert du domaine ». S'il manque un expert du domaine dans vos processus de test d'acceptation par l'utilisateur, vous risquez d'obtenir des résultats inexacts et vos testeurs risquent d'utiliser un langage erroné pour décrire les problèmes, ce qui enverra votre équipe de développement sur la mauvaise voie lorsqu'elle cherchera à corriger le logiciel et à résoudre les problèmes.

Risque d'erreur humaine : Alors que les ordinateurs et les machines sont conçus pour effectuer la même tâche encore et encore sans dévier, ce n'est pas le cas pour les personnes. Les gens sont faillibles et peuvent parfois commettre des erreurs, quel que soit le niveau des employés de votre organisation. Les tests manuels laissent place à l'erreur humaine qui peut donner des résultats inexacts ou laisser certains tests incomplets à la fin du processus de test. C'est pourquoi les tests UAT réalisés manuellement ont tendance à être répétés à plusieurs reprises. Un plus grand nombre de tests réalisés par plusieurs testeurs permet de s'assurer qu'un seul cas de test inexact n'a pas d'impact négatif sur le résultat global du processus de développement après le test.

Difficile de tester les tâches répétitives : L'un des principaux avantages de l'automatisation des tests UAT réside dans le fait qu'un développeur est en mesure de réaliser exactement le même test avec les mêmes données et les mêmes étapes à chaque fois. Il n'y a aucun risque de manquer une étape ou de ne pas compléter une partie spécifique du processus.
Ce n'est pas le cas des testeurs manuels. Dans certaines tâches très répétitives, un testeur UAT manuel peut occasionnellement manquer l'une des étapes du test ou enregistrer les informations de manière inexacte. Les tâches qui nécessitent une répétition peuvent être difficiles pour les testeurs qui examinent manuellement les logiciels, en particulier si la répétition se fait sur plusieurs heures et des centaines de cycles.

Besoins importants en ressources : La réalisation manuelle des tests d'acceptation par l'utilisateur est une méthode qui mobilise beaucoup de ressources au sein d'une entreprise.
Il ne s'agit pas seulement du coût financier, mais pour les logiciels plus importants, il peut s'agir d'une plus grande pression sur le personnel, qui examine les données que l'organisation reçoit des tests UAT en plus d'administrer les tests manuels avec sa base d'utilisateurs.
Un tel besoin en ressources signifie que d'autres départements d'une entreprise peuvent recevoir des contraintes sur leurs exigences, car le processus de test demande plus d'attention que la majorité des autres projets de développement.

7. Conclusion

Les tests d'acceptation de l'utilisateur (User Acceptance Test, UAT) sont essentiels pour le lancement réussi et l'entretien de tout projet logiciel, l'acceptation de l'utilisateur garantit que logiciel ou application fonctionnent comme prévu et aussi détermine si l'utilisateur l'accepte ou non.

Les tests d'acceptation utilisateur sont souvent indispensables pour s'assurer qu'un nouveau processus ou un nouveau système apporte la valeur attendue au client et aux utilisateurs dans le contexte de leur activités opérationnelles. Une phase de recette bien pensée et exécutée sécurise la conduite du changement et la réussite du projet. Aussi, accepter des compromis peut s'avérer payant pour trouver le bon équilibre entre la livraison d'un projet dans les délais, le budget et le respect des normes de qualité.

Le cahier de recette était indispensable…..Un produit peu ou mal testé peut devenir une catastrophe pour l'entreprise. Une simple erreur dans un produit mis en production ébranlera la confiance du client de manière irréversible. Certaines erreurs peuvent même avoir des conséquences financières gigantesques. C'est pourquoi il est si important d'organiser des tests des nouveaux produits de manière systématique et d'y impliquer le client et les utilisateurs finaux.

Le cahier de recettage et le plan de tests vous permettront de structurer les tests et leur résultat. Le mettre à jour à chaque phase de test importante et le garderez dans vos archives.
Un plan bien conçu augmente vos chances de délivrer un produit solide et fiable. Mais avant tout il créera un climat de confiance pour l'équipe projet comme pour le client. Les risques potentiels seront donc fortement réduits et la qualité largement augmentée.

Université de Mohamed Khider – Biskra –
Département d'informatique
Niveau : 1ère Master (professionnel)
Module : SIA-1 **Année : 2024/2025**

TD N° 5 : Le test d'acceptation « La recette »

Application

Les tests d'acceptation doivent être détaillés et liés aux exigences fonctionnelles définies dans le cahier des charges pour garantir une validation précise et complète.

Chaque test d'acceptation doit être précis, reproductible, et lié à une fonctionnalité métier ou technique. En documentant des étapes claires et des résultats attendus, vous facilitez la validation des exigences du projet par toutes les parties prenantes.

Le test d'acceptation permet de vérifier que les exigences fonctionnelles et non fonctionnelles sont respectées avant la validation finale.

Exemple 1 : Gestion des tâches

Contexte

- **Module testé** : Gestion des tâches
- **Objectif** : Vérifier que l'utilisateur peut créer, modifier, attribuer et suivre une tâche conformément aux spécifications fonctionnelles.

Scénario de test d'acceptation

Nom du test : Création et gestion d'une tâche

Objectif : Vérifier que l'utilisateur peut créer, modifier et gérer une tâche avec toutes les options nécessaires.

Prérequis :

- L'utilisateur doit être connecté à l'application.
- L'utilisateur doit avoir un accès au projet où il souhaite créer la tâche.

Étapes à réaliser :

1. Aller sur la page ou le tableau de gestion des tâches.
2. Cliquer sur le bouton "Créer une tâche".
3. Remplir les champs obligatoires :
    - Nom de la tâche : "Préparer le rapport mensuel".
    - Description : "Rédiger et soumettre le rapport d'activités pour le mois."
    - Échéance : Sélectionner une date future (par ex., 31/01/2025).
    - Attribuer un responsable : Sélectionner "Utilisateur A".
4. Ajouter des détails supplémentaires :

- Étiquettes : Ajouter "Priorité élevée".
- Sous-tâches : Ajouter "Collecte des données", "Analyse des données".

5. Enregistrer la tâche.
6. Revenir sur la liste des tâches et vérifier que la nouvelle tâche apparaît.
7. Modifier la tâche en changeant son statut à "En cours".
8. Supprimer une sous-tâche et enregistrer les modifications.

Résultats attendus :

- La tâche est créée avec les informations renseignées.
- Les sous-tâches sont correctement affichées sous la tâche principale.
- L'utilisateur peut modifier le statut de la tâche.
- Une sous-tâche peut être supprimée sans affecter la tâche principale.
- L'utilisateur reçoit un message de confirmation pour chaque action réussie.

Critères de succès :

- La tâche est visible dans le tableau de gestion.
- Les modifications sont correctement enregistrées et reflétées.
- Aucune erreur technique ou comportement inattendu n'est rencontré.

Exemple 2 (simple) : Vérification des notifications

Objectif : S'assurer que l'utilisateur reçoit une notification lorsqu'une tâche lui est assignée.

Étapes :

1. Un administrateur crée une tâche et l'assigne à "Utilisateur B".
2. Vérifier que "Utilisateur B" reçoit une notification dans son espace personnel.

Résultat attendu : La notification est reçue immédiatement, avec les détails de la tâche.

Exemple 3 : Gestion des utilisateurs

Nom du test : Création et gestion d'un utilisateur

Objectif : Vérifier que l'administrateur peut ajouter, modifier et désactiver un utilisateur.

Prérequis :

- L'administrateur doit être connecté au système avec des droits d'administration.
- Le formulaire de gestion des utilisateurs doit être accessible.

Étapes à réaliser :

1. Aller dans le menu "Gestion des utilisateurs".
2. Cliquer sur "Ajouter un utilisateur".
3. Remplir les champs obligatoires :
   - Nom : "Jean Dupont".
   - Email : "jean.dupont@example.com".

- Rôle : "Collaborateur".
- Statut : "Actif".

4. Enregistrer le nouvel utilisateur.
5. Vérifier que l'utilisateur apparaît dans la liste des utilisateurs avec les informations correctes.
6. Modifier les informations de l'utilisateur :
   - Changer le rôle à "Administrateur".
   - Mettre à jour le statut à "Inactif".
7. Enregistrer les modifications.
8. Supprimer l'utilisateur de la liste.

Résultats attendus :

- L'utilisateur est ajouté avec succès et visible dans la liste.
- Les modifications sont correctement enregistrées.
- L'utilisateur peut être supprimé sans erreur.
- Une notification de succès est affichée pour chaque action.

Exemple 5 : Création d'un projet

Nom du test : Création d'un projet avec des paramètres personnalisés

Objectif : Vérifier que l'utilisateur peut créer un projet avec des paramètres spécifiques et y ajouter des membres.

Prérequis :

- L'utilisateur doit être connecté avec des droits pour créer un projet.

Étapes à réaliser :

1. Naviguer vers la page "Projets".
2. Cliquer sur "Créer un projet".
3. Remplir les informations suivantes :
   - Nom : "Lancement du produit X".
   - Description : "Coordination des activités pour le lancement du nouveau produit".
   - Échéance : "31/12/2025".
   - Statut initial : "En cours".
   - Ajouter des membres : Sélectionner "Utilisateur A" et "Utilisateur B".
4. Définir les autorisations d'accès :
   - "Utilisateur A" : Accès complet.
   - "Utilisateur B" : Lecture seule.
5. Enregistrer le projet.

6. Vérifier que le projet apparaît dans la liste des projets avec les paramètres renseignés.

Résultats attendus :

- Le projet est visible dans la liste avec le nom et la description corrects.
- Les membres ajoutés apparaissent avec les bonnes autorisations.
- Les notifications sont envoyées aux membres ajoutés.

Exemple 6 : Téléversement de fichiers

Nom du test : Téléversement et gestion de fichiers

Objectif : Vérifier que l'utilisateur peut téléverser, afficher et supprimer un fichier dans un projet.

Prérequis :

- L'utilisateur doit avoir accès à un projet spécifique.

Étapes à réaliser :

1. Accéder à un projet existant.
2. Aller dans la section "Fichiers".
3. Cliquer sur "Téléverser un fichier".
4. Sélectionner un fichier depuis l'ordinateur, par exemple, "rapport_financier.pdf".
5. Valider le téléversement.
6. Ouvrir le fichier depuis l'interface pour visualiser son contenu.
7. Supprimer le fichier téléversé.
8. Confirmer la suppression.

Résultats attendus :

- Le fichier est téléversé avec succès et visible dans la liste des fichiers.
- Le fichier peut être affiché sans erreur.
- Le fichier est supprimé avec une confirmation affichée.

Exemple 7 : Gestion des notifications

Nom du test : Configuration et réception des notifications

Objectif : Vérifier que l'utilisateur peut configurer ses préférences de notification et recevoir des alertes correspondantes.

Prérequis :

- L'utilisateur doit être connecté.

Étapes à réaliser :

1. Aller dans le menu "Paramètres".
2. Accéder à la section "Notifications".
3. Modifier les paramètres suivants :

- Activer les notifications par email pour les nouvelles tâches.
- Désactiver les notifications par SMS.

4. Enregistrer les modifications.
5. Simuler un événement déclencheur (par exemple, assigner une tâche à l'utilisateur).
6. Vérifier que l'utilisateur reçoit une notification par email et aucune notification par SMS.

Résultats attendus :

- Les préférences sont correctement enregistrées.
- Les notifications respectent les paramètres configurés.

Exemple 8 : Rapport de performance

Nom du test : Génération et téléchargement d'un rapport

Objectif : Vérifier que l'utilisateur peut générer et télécharger un rapport détaillé sur les tâches du projet.

Prérequis :

- Le projet doit contenir plusieurs tâches avec des statuts différents.

Étapes à réaliser :

1. Accéder à la section "Rapports" du projet.
2. Cliquer sur "Créer un rapport".
3. Sélectionner les options suivantes :
   - Filtrer par statut : "Terminé" et "En cours".
   - Inclure les colonnes : Nom de la tâche, Responsable, Échéance.
   - Générer un rapport au format PDF.
4. Cliquer sur "Télécharger".
5. Ouvrir le fichier téléchargé pour vérifier son contenu.

Résultats attendus :

- Le rapport est généré avec les bonnes données.
- Le fichier téléchargé contient toutes les informations sélectionnées.
- Aucune erreur n'est rencontrée durant le processus.

Travail demandé

1. Choisissez 4 exemples et appliquez les techniques de test d'acceptation
2. Remplissez le tableau suivant par l'étude et le raffinement des exemples précédents :

Exemple	exigences fonctionnelles	exigences non fonctionnelles	résultats attendus (obtenu ou pas)	parties prenantes	validation finale (qualité)
Exemple 1					
Exemple 2					
Exemple 3					
Exemple 4					

3. Appliquez les tests d'acceptation sur votre mini projet (TP 1).
4. Editez un bon rapport pour ce travail.