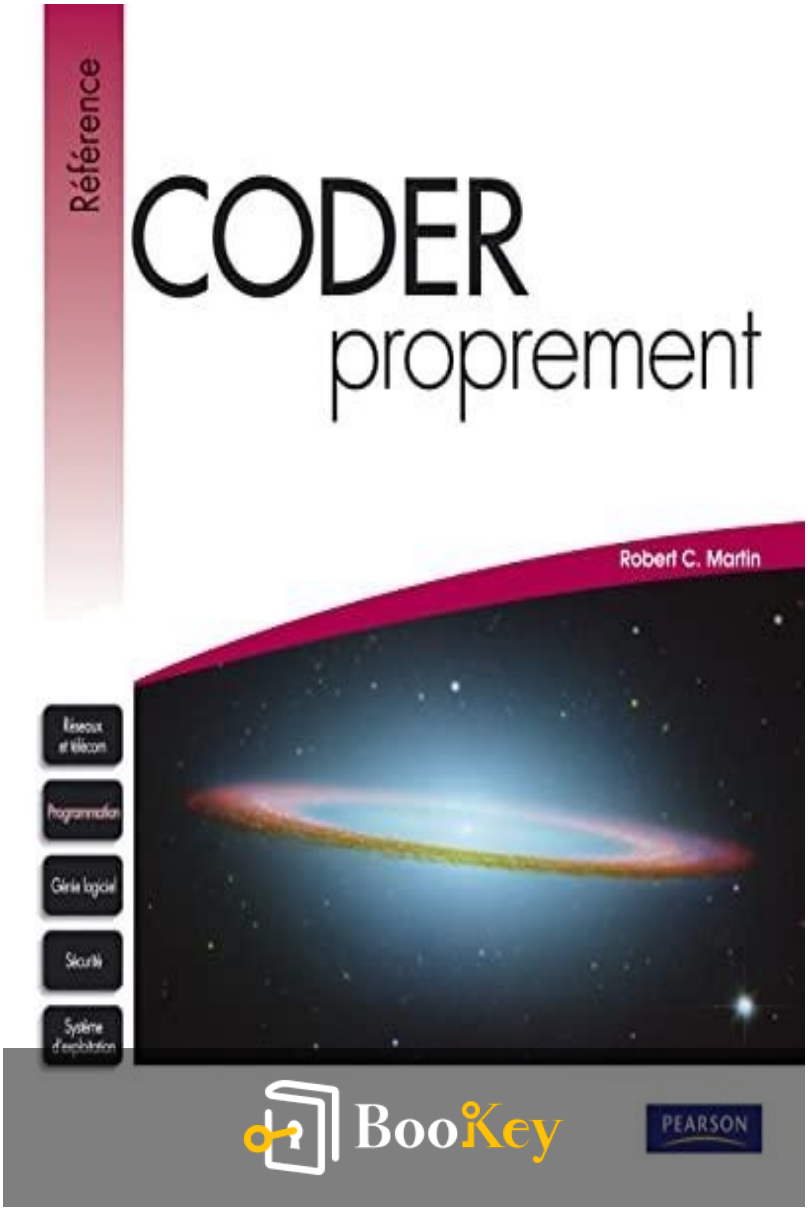
CODER PROPREMENT PDF

Robert C. Martin



CODER PROPREMENT

Transformez vos compétences en programmation grâce aux principes du code propre.

Écrit par Bookey

En savoir plus sur le résumé de CODER PROPREMENT

Écouter CODER PROPREMENT Livre audio

À propos du livre

Dans "CODER PROPREMENT : Un Manuel de l'Artisanat Agile en Logiciel," le célèbre expert en logiciels Robert C. Martin révèle l'importance transformative de l'écriture de code propre et son impact sur les organisations de développement. Chaque année, un code mal écrit fait perdre d'innombrables heures et d'importantes ressources, mais Martin, avec ses collègues d'Object Mentor, propose une solution à travers une exploration approfondie des principes de codage agile. Ce guide perspicace encourage les programmeurs à évaluer de manière critique leur code en examinant ses forces et ses faiblesses, tout en remettant en question leurs valeurs professionnelles et leur engagement envers l'artisanat. Divisé en trois parties, le livre couvre les principes essentiels du codage, présente des études de cas complexes pour une application pratique, et se termine par une précieuse collection d’heuristiques et de "mauvaises odeurs" de code. Les lecteurs apprendront à distinguer le bon code du mauvais, à écrire des fonctions et des classes efficaces, à assurer la lisibilité, à mettre en œuvre une gestion d'erreurs robuste et à adopter le développement piloté par les tests. Un ouvrage incontournable pour les développeurs, les ingénieurs logiciels et quiconque s'engage à produire du code de haute qualité.

À propos de l'auteur

Robert C. Martin, souvent appelé "Oncle Bob", est un ingénieur logiciel renommé, auteur et conférencier avec plus de cinq décennies d'expérience dans le domaine de la programmation informatique et du développement logiciel. Il est co-fondateur de l'Agile Alliance et un fervent défenseur des méthodologies agiles et de l'artisanat logiciel. Martin est surtout connu pour ses œuvres influentes, notamment "CODER PROPREMENT : Un Guide de l'Artisanat Logiciel Agile", où il souligne l'importance d'écrire un code clair, maintenable et efficace. Ses contributions à la communauté des développeurs vont au-delà de l'écriture, car il a également joué un rôle majeur dans l'établissement des meilleures pratiques et la promotion des valeurs de professionnalisme et d'éthique en programmation. Par ses enseignements et ses écrits, Martin continue d'inspirer une nouvelle génération de développeurs à aspirer à l'excellence dans leur métier.

Liste du contenu du résumé

Chapitre 1 Résumé : 2 Noms Significatifs

Section	Points Clés
Introduction	Importance d'un nommage efficace dans le développement logiciel.
Utiliser des Noms Révélateurs d'Intention	Les noms doivent refléter leur objectif ; des noms clairs réduisent le besoin de commentaires.
Éviter la Désinformation	Les noms doivent être clairs et ne pas être trompeurs ou ambigus.
Faire des Distinctions Significatives	Chaque nom doit avoir un but unique pour éviter la confusion.
Utiliser des Noms Prononçables	Les noms doivent être faciles à prononcer pour une communication efficace.
Utiliser des Noms Recherchables	Éviter les noms d'une seule lettre ; des noms plus clairs sont plus faciles à rechercher.
Éviter les Encodages	Éviter d'encoder le type ou la portée dans les noms pour réduire la complexité.
Noms de Méthodes	Les noms de méthodes doivent clairement exprimer des actions, en évitant les noms astucieux.
Choisir un Mot par Concept	Utiliser une terminologie cohérente pour éviter la confusion dans le code.
Ajouter un Contexte Significatif	Contextualiser les noms au sein des classes/fonctions, mais éviter l'encombrement.
Derniers Mots	Un nommage efficace est vital pour la lisibilité ; n'hésitez pas à renommer.

Chapitre 1 : Noms Significatifs

Introduction

Les noms sont omniprésents dans le développement logiciel. Ils concernent les variables, les fonctions, les classes, les packages, et même les fichiers. Par conséquent, il est essentiel de bien les nommer.

Utilisez des Noms Révélateurs d'Intention

Les noms doivent refléter leur but. Un nom clair élimine le besoin de commentaires. Par exemple, `int d;` est vague, tandis que `int elapsedTimeInDays;` est explicite. Un nom clair simplifie la compréhension du code.

Évitez la Désinformation

Les noms doivent être clairs et ne pas induire en erreur. Évitez les noms qui pourraient signifier différentes choses dans différents contextes ou créer de la confusion, par exemple, nommer une liste qui n'est pas une Liste comme `accountList`.

Faites des Distinctions Significatives

Ne modifiez pas les noms arbitrairement juste pour satisfaire le compilateur. Chaque nom doit avoir un but distinct, évitant la confusion entre des variables aux noms similaires.

Utilisez des Noms Prononçables

Les noms doivent être faciles à prononcer pour faciliter la communication. Évitez les conventions de nommage qui mènent à des abréviations complexes qui ne peuvent pas être facilement discutées.

Utilisez des Noms Rechercheables

Évitez les noms à une lettre et les constantes numériques qui sont difficiles à rechercher dans le code. Une constante bien nommée est plus facile à distinguer.

Évitez les Encodages

Ne codez pas le type ou le scope dans les noms. Les encodages créent une complexité inutile sans apporter de valeur, en particulier dans les langages de programmation modernes avec des systèmes de types forts.

Noms des Méthodes

Les noms des méthodes doivent transmettre des actions (par exemple, `postPayment`) pour éviter le mappage mental. Gardez la clarté en évitant des noms astucieux ou accrocheurs qui obscurcissent leurs fonctions.

Choisissez Un Mot par Concept

Utilisez une terminologie cohérente dans votre base de code pour éviter la confusion (par exemple, évitez d'utiliser à la fois `fetch` et `retrieve` pour des fonctions similaires).

Ajoutez un Contexte Significatif

Les noms doivent être contextualisés dans les classes ou les fonctions. Regroupez les variables connexes pour une meilleure compréhension, mais évitez un contexte excessif et inutile qui encombre les noms.

Mots de la Fin

Un nommage efficace peut être difficile, mais il est essentiel pour la lisibilité. N'hésitez pas à renommer des éléments pour

plus de clarté. Utilisez des outils modernes pour maintenir l'organisation et améliorer la qualité du code.

Exemple

Point clé:Utilisez des noms qui révèlent l'intention

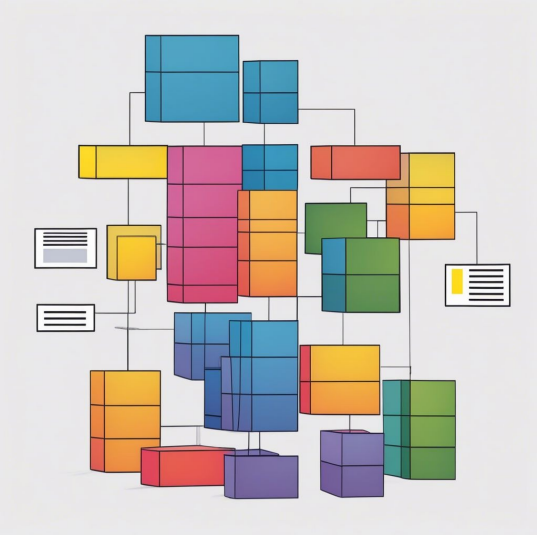
Exemple:Imaginez que vous examinez le code d'un collègue et que vous tombez sur une variable nommée `int x;`. Immédiatement, la confusion s'installe : que est censée représenter cette variable ? Maintenant, imaginez un scénario différent où vous voyez `int userAgeInYears;`. Tout à coup, il est crystal clair que cette variable contient l'âge d'un utilisateur, rendant le code beaucoup plus compréhensible sans un seul commentaire. Cet exemple souligne l'importance des conventions de nommage qui expriment clairement le but des variables, améliorant ainsi la lisibilité globale du code.

Pensée critique

Point clé:L'importance des noms significatifs dans le développement logiciel

Interprétation critique:Robert C. Martin souligne que le choix de noms efficaces est essentiel dans le développement logiciel, car cela améliore la lisibilité et la compréhension du code. Cependant, certains peuvent argumenter qu'une insistance excessive sur les conventions de nommage peut faire négliger d'autres pratiques de codage vitales, telles que la structure du code et la modularité. Il est intéressant de considérer d'autres perspectives, comme celles présentées par Martin Fowler dans "Refactoring: Improving the Design of Existing Code", qui suggère que, bien que des noms clairs soient importants, ils doivent également faire partie d'une préoccupation plus large pour la qualité générale du code.

Chapitre 2 Résumé : 3 Fonctions



Section	Résumé
Évolution des fonctions	La programmation est passée des routines aux fonctions comme principal mode d'organisation du code.
Compréhension de la complexité des fonctions	Une fonction encombrante remplie de code dupliqué et de types complexes nuit à la compréhension, tandis que les fonctions refactorisées clarifient l'intention et restent concises.
Meilleures pratiques pour les fonctions	La taille compte: Les fonctions doivent être petites (moins de 20 lignes). Responsabilité unique: Chaque fonction doit gérer une seule tâche. Niveaux d'abstraction: Maintenez un niveau d'abstraction cohérent.
Composition des fonctions	Les fonctions doivent se lire comme un récit, garantissant un flux fluide des actions de haut niveau vers des implémentations spécifiques.
Évitement des instructions switch	Les instructions switch doivent être évitées car elles diminuent la clarté ; le polymorphisme est préféré pour gérer les variations.
Conventions de nommage	Les noms des fonctions doivent être descriptifs pour clairement transmettre les actions et établir des attentes.
Arguments des fonctions	Minimiser les arguments: Visez zéro argument ; 1-2 acceptables, évitez d'en avoir plus de 3. Évitez les arguments de sortie: Privilégiez les valeurs de retour pour éviter toute confusion. Distinction entre commandes et requêtes: Les fonctions doivent soit agir, soit retourner des informations, mais pas les deux.
Stratégies de gestion des erreurs	Utilisez des exceptions au lieu de codes d'erreur pour un flux de contrôle plus propre, en gardant la gestion des exceptions distincte de la logique métier.
Conclusion	Les fonctions doivent être concises et significatives, ressemblant à des verbes dans une langue,

Section	Résumé
	pour faciliter une communication efficace dans le code.

Chapitre 2 : Fonctions

Évolution des Fonctions

La programmation a évolué des routines vers les fonctions, devenant ainsi la principale méthode d'organisation des pratiques de codage.

Comprendre la Complexité des Fonctions

La fonction présentée dans la Liste 3-1 est encombrante, remplie de code dupliqué, de chaînes étranges et de types de données compliqués, rendant difficile sa compréhension rapide. En revanche, la Liste 3-2 simplifie cela en refactorisant la fonction d'origine, clarifiant son objectif tout en restant concise.

Meilleures Pratiques pour les Fonctions

-

La Taille Compte
 : Les fonctions doivent être petites, idéalement de moins de 20 lignes, ce qui permet une meilleure lisibilité et maintenabilité.
-
Responsabilité Unique
 : Une fonction doit accomplir une seule tâche ou suivre un seul concept, réduisant ainsi la complexité et améliorant la clarté.
-
Niveaux d'Abstraction
 : Les fonctions doivent maintenir un niveau d'abstraction cohérent sans mélanger des opérations de haut niveau avec des détails de bas niveau.

Composition des Fonctions

Les fonctions doivent se lire comme une narration, progressant des actions conceptuelles de haut niveau vers des implémentations spécifiques, suivant une approche descendante. Cela crée un flux fluide, rendant le code plus facile à lire et à comprendre.

Évitement des Instructions Switch

Les instructions switch introduisant plusieurs flux dans une fonction sont déconseillées, car elles diminuent la clarté. À la place, utiliser le polymorphisme peut aider à gérer différentes opérations de manière claire.

Conventions de Nommage

Choisir des noms descriptifs est crucial pour la clarté. Les noms de fonction doivent transmettre clairement leurs actions, aidant à établir des attentes avant que la fonction ne soit lue.

Arguments de Fonction

-
Minimiser les Arguments
: Les fonctions ne devraient idéalement avoir aucun argument, un ou deux étant acceptables. Plus de trois devraient généralement être évités en raison de la complexité.
-
Éviter les Arguments de Sortie
: Les fonctions devraient utiliser des valeurs de retour au lieu d'arguments de sortie pour prévenir toute confusion sur les

données échangées.

-

Séparer Commandes et Requêtes

: Les fonctions devraient soit effectuer une action, soit retourner des informations, mais pas les deux, afin d’éviter toute ambiguïté.

Stratégies de Gestion des Erreurs

Utiliser des exceptions au lieu de codes d'erreur peut mener à un flux de contrôle plus propre dans les fonctions. La gestion des exceptions devrait être séparée de la logique métier pour maintenir la clarté.

Conclusion

La programmation concerne fondamentalement la création d'un langage pour représenter un système. Les fonctions, comme des verbes dans un langage, devraient être concises, ciblées et bien structurées pour communiquer efficacement. Suivre les principes énoncés conduira à un code plus propre et plus maintenable.

Exemple

Point clé:Comprendre l'Importance de la Clarté des Fonctions

Exemple:Imaginez que vous examinez du code et que vous tombez sur une fonction de plus de 50 lignes, avec une logique qui se chevauche et des noms déroutants. Alors que vous essayez de percer son but, la frustration monte. Maintenant, imaginez plutôt ouvrir un fichier et trouver une fonction compacte, pas plus de 20 lignes, nommée 'calculerPaiementMensuel'. Instantanément, vous reconnaissez la tâche qu'elle effectue. Vous parcourez sans effort des lignes claires et organisées qui séparent logiquement les étapes en morceaux gérables - pas de complexité superflue, juste un calcul simple. Cette clarté non seulement vous fait gagner du temps, mais réduit considérablement le risque de bugs lors des modifications futures, vous permettant de vous concentrer sur l'amélioration de la fonctionnalité plutôt que de démêler un enchevêtrement de codes.

Pensée critique

Point clé:L'accent mis sur la taille des fonctions peut ne pas s'appliquer universellement.

Interprétation critique:Bien que l'auteur prône des fonctions plus petites pour améliorer la lisibilité et la maintenabilité, ce point de vue peut ne pas tenir compte des contextes spécifiques où des fonctions plus grandes et plus complexes peuvent encore être utilisées efficacement. Dans certaines paradigmes de programmation, comme la programmation fonctionnelle, le besoin de fonctions composites plus grandes peut surgir naturellement en raison de la nature de la tâche à résoudre (Knuth, D. E. (1997). 'L'Art de la programmation'.) Ainsi, les lecteurs devraient évaluer de manière critique l'approche universelle concernant la taille des fonctions.

Chapitre 3 Résumé : 4 Commentaires

Section	Points Clés
La Nature des Commentaires	Les commentaires peuvent être utiles mais peuvent encombrer le code. De bons commentaires indiquent un échec à exprimer l'intention à travers le code. Les commentaires peuvent devenir obsolètes au fur et à mesure que le code évolue. Le meilleur code devrait être auto-explicatif.
Pratiques Efficaces pour les Commentaires	CODER PROPREMENT au lieu d'ajouter des commentaires pour un code médiocre. Un code efficace peut réduire le besoin de commentaires. Les commentaires bénéfiques incluent : Commentaires Légaux : Nécessaires pour le droit d'auteur. Commentaires Informatifs : Expliquent les valeurs de retour et le but. Explication de l'Intention : Raisons des décisions de codage. Clarifications : Contexte pour les arguments complexes. Avertissements : Alerte concernant les conséquences ou l'utilisation.
Les Dangers des Commentaires	De mauvais commentaires peuvent induire en erreur et créer de la confusion. Commentaires Redondants : Répétition inutile du code. Marmonnements et Bruit : Déclarations vagues et évidentes. Code Commenté : Évitez le code inactif commenté ; utilisez le contrôle de version. Commentaires Obligatoires : Des commentaires forcés mènent à une documentation encombrée.
Meilleures Pratiques pour Écrire des Commentaires	Évitez les commentaires lorsque des noms peuvent expliquer le code. Les commentaires devraient décrire le code à proximité, pas des informations non locales. Réduisez l'information dans les commentaires pour ne pas submerger les lecteurs. Insistez sur la clarté et la concision ; révisez pour l'exactitude.
Conclusion	Visez un code clair pour minimiser les commentaires nécessaires, en utilisant les commentaires avec parcimonie et efficacité lorsque c'est nécessaire.

Chapitre 3 : Commentaires

La Nature des Commentaires

- Les commentaires peuvent être utiles mais encombrent souvent le code.
- De bons commentaires sont un signe d'échec à exprimer une intention à travers le code.
- Avec le temps, les commentaires peuvent devenir obsolètes ou trompeurs à mesure que le code évolue.
- Le meilleur code devrait être explicite, minimisant ainsi le besoin de commentaires.

Pratiques Efficaces de Commentaires

- Les commentaires ne doivent pas remplacer un code

Scanner pour télécharger

Pourquoi Bookey est une application incontournable pour les amateurs de livres

Contenu de 30min
Plus notre interprétation est profonde et claire, mieux vous saisissez chaque titre.

Format texte et audio
Absorbez des connaissances même dans un temps fragmenté.

Quiz
Vérifiez si vous avez maîtrisé ce que vous venez d'apprendre.

Et plus
Plusieurs voix & polices, Carte mentale, Citations, Clips d'idées...

Essai gratuit avec Bookey

Chapitre 4 Résumé : 5 Mise en forme

Section	Résumé
Importance du formatage du code	Améliore la communication et la lisibilité, reflétant le professionnalisme et l'attention aux détails.
Formatage vertical	Les fichiers doivent être petits (<200 lignes) pour une meilleure compréhension ; ils doivent commencer par des concepts de haut niveau.
Ouverture verticale	Utilisez des lignes vides pour séparer les concepts afin d'améliorer la lisibilité et la clarté dans les sections logiques.
Densité verticale	Gardez le code lié dense sans trop de sauts de ligne ou de commentaires qui obscurcissent la clarté.
Distance verticale	Placez les variables et fonctions liées proches les unes des autres pour réduire la confusion ; déclarez les variables au début des classes.
Ordre vertical	Les appels de fonction doivent suivre un ordre logique pour aider les lecteurs à comprendre la structure du programme.
Formatage horizontal	Ayez des lignes de moins de 120 caractères et utilisez efficacement les espaces sans alignements inutiles.
Ouverture et densité horizontales	Utilisez des espaces pour clarifier les relations et maintenir des distinctions claires entre les différents éléments.
Indentation	Indentez le code pour représenter sa structure hiérarchique afin de faciliter la navigation dans les portées.
Règles de l'équipe	Une cohérence entre les membres de l'équipe est essentielle ; chacun doit suivre un ensemble de règles de formatage partagé.
Règles de formatage d'Oncle Bob	Mise en avant de la clarté et de l'uniformité ; favorise la création d'un code fonctionnel, élégant et compréhensible.

Chapitre 5 : Mise en forme

Importance de la mise en forme du code

La mise en forme est cruciale en programmation car elle améliore la communication et garantit que le code est lisible.

Une base de code bien formatée reflète le professionnalisme et l'attention aux détails, ce qui peut influencer la perception de l'ensemble du projet.

Mise en forme verticale

Les fichiers devraient généralement être de petite taille, idéalement moins de 200 lignes, pour faciliter la compréhension. À l'image des articles de journaux, les fichiers sources devraient offrir une structure claire, commençant par des concepts généraux et détaillant progressivement les spécificités.

Ouverture verticale

Utilisez des lignes vides pour séparer des concepts distincts, ce qui aide à la lisibilité. Cela favorise une plus grande clarté dans la mise en page visuelle, permettant aux lecteurs de distinguer facilement les différentes sections logiques du code.

Densité verticale

Le code connexe devrait être dense verticalement,

minimisant les sauts de ligne inutiles pour faciliter la compréhension. Évitez les commentaires excessifs qui peuvent nuire à la clarté.

Distance verticale

Les variables et les fonctions connexes devraient être proches les unes des autres pour éliminer la confusion lors de la navigation dans le code. Les variables d'instance devraient généralement être déclarées au début des classes, tandis que les fonctions étroitement liées devraient suivre le flux naturel des opérations.

Ordre vertical

Les appels de fonction devraient se dérouler logiquement vers le bas, maintenant un ordre qui permet aux lecteurs de saisir facilement la structure du programme.

Mise en forme horizontale

Visez des longueurs de ligne plus courtes, idéalement ne dépassant pas 120 caractères. Utilisez des espaces pour relier des éléments étroitement associés, tout en évitant les

alignements inutiles qui peuvent détourner l'attention de l'objectif du code.

Ouverture et densité horizontales

L'espace horizontal sert à clarifier les relations entre les éléments, améliorant la lisibilité. Équilibrez l'espace pour signifier de fortes associations tout en maintenant une distinction claire lorsque cela est nécessaire.

Indentation

L'indentation représente visuellement la structure hiérarchique du code, rendant plus facile la navigation à travers différents niveaux de portée. Indentez les instructions pour refléter cette structure, maintenant le code propre et compréhensible.

Règles d'équipe

La cohérence au sein d'une équipe de développement est essentielle ; tous les membres devraient respecter un ensemble commun de règles de mise en forme pour maintenir un style cohérent tout au long de la base de code.

Règles de mise en forme d'Oncle Bob

L'auteur, "Oncle Bob," maintient des règles de mise en forme personnelles qui privilégient la clarté et l'uniformité, incarnées dans ses exemples. En respectant ces principes, les programmeurs peuvent créer un code qui est non seulement fonctionnel, mais aussi élégant et facile à comprendre.

Chapitre 5 Résumé : 6 Objets et Structures de Données

Section	Points Clés
Abstraction des Données	Conservez les variables privées pour plus de flexibilité. Les accesseurs et mutateurs peuvent compromettre l'abstraction des données. Utilisez des interfaces pour les opérations sans exposer la structure.
Anti-Symétrie Données/Objets	Les objets encapsulent les données avec des méthodes ; les structures de données exposent les données. Conséquences significatives pour la conception du système. Le code procédural facilite l'ajout de fonctions ; le code orienté objet facilite l'ajout de types de données.
Loi de Demeter	Un module ne devrait pas connaître la structure interne des objets manipulés. Les méthodes devraient communiquer uniquement avec leur classe ou les objets qu'elles possèdent. Les violations peuvent conduire à un code complexe et fortement couplé ("casse de train").
Hybrides	Évitez les structures hybrides combinant les caractéristiques objets et structures de données. Les objets doivent encapsuler le comportement, évitant l'exposition inutile des données.
Objets de Transfert de Données	Les OTD sont des structures simples avec des variables publiques, utilisées pour la communication de données. Les beans ont des variables privées accessibles via des accesseurs/mutateurs, offrant peu d'avantages. Les Enregistrements Actifs servent d'OTD avec des méthodes de manipulation de données, provoquant des hybrides de conception.
Conclusion	Les objets permettent une addition facile de types de données, mais compliquent l'ajout de comportements. Les structures de données facilitent l'ajout de comportements, compliquant l'intégration de types de données.

Section	Points Clés
	Les développeurs doivent évaluer les exigences du système pour faire des choix de conception.

Objets et Structures de Données

Abstraction des Données

- Les variables devraient rester privées pour maintenir la flexibilité des changements.
- Les accesseurs et les mutateurs peuvent exposer des variables privées, compromettant l'abstraction des données.
- Mettre en œuvre l'abstraction des données implique d'utiliser des interfaces qui permettent des opérations sans révéler la structure sous-jacente.

Anti-Symétrie Données/Objets

- Les objets encapsulent des données avec des méthodes, tandis que les structures de données exposent des données sans comportement.
- Cette distinction a des implications significatives pour la conception des systèmes.

- Le code procédural simplifie l'ajout de fonctions sans changer les structures de données, tandis que le code orienté objet (OO) simplifie l'ajout de types de données sans modifier les comportements existants.

Principe de Demeter

- Un module ne devrait pas être au courant de la structure interne des objets qu'il manipule.
- Le principe suggère que les méthodes ne devraient communiquer qu'avec leur propre classe ou les objets qu'elles créent ou détiennent directement.
- Les violations conduisent souvent à un code complexe et fortement couplé, qualifié de "casse-train."

Hybrides

- Les structures hybrides qui combinent des caractéristiques des objets et des structures de données compliquent la conception et devraient être évitées.
- Les objets devraient encapsuler le comportement et ne pas exposer leurs données inutilement.

Objets de Transfert de Données

- Les DTO sont des structures simples avec des variables publiques et aucune fonctionnalité, souvent utilisées pour la communication de données.
- Les beans ont des variables privées accessibles via des accesseurs et des mutateurs, mais ils peuvent apporter peu d'avantages supplémentaires.
- Les enregistrements actifs fonctionnent comme des DTO mais avec des méthodes pour la manipulation des données, conduisant souvent à une hybridation de la conception.

Conclusion

- Les objets offrent la flexibilité d'ajouter de nouveaux types de données mais rendent plus difficile l'ajout de comportements, tandis que les structures de données facilitent l'ajout de comportements mais compliquent les intégrations de types de données.
- Les développeurs compétents évaluent les exigences spécifiques d'un système, choisissant la conception et l'approche appropriées en conséquence.

Chapitre 6 Résumé : Gestion des erreurs

Section	Résumé
Introduction	La gestion des erreurs est fondamentale, mais lorsqu'elle est abusée, elle peut obscurcir la logique principale du code.
Utiliser des exceptions plutôt que des codes de retour	Les exceptions séparent clairement la gestion des erreurs de la logique principale, améliorant ainsi la lisibilité.
Rédigez d'abord votre structure try-catch-finally	Commencer par des structures try-catch-finally clarifie les exceptions attendues et maintient la logique cohérente.
Utiliser des exceptions non vérifiées	Les exceptions non vérifiées offrent de la flexibilité et évitent d'encombrer les signatures de méthode par rapport aux exceptions vérifiées.
Fournir un contexte avec les exceptions	Les exceptions doivent inclure des informations détaillées sur les erreurs pour aider au débogage et à la journalisation.
Définir les classes d'exception en fonction des besoins de l'appelant	Concentrez-vous sur la façon dont les appelants géreront les exceptions ; la normalisation des exceptions améliore la qualité du code.
Définir le flux normal	Encapsuler les comportements de cas spéciaux réduit l'encombrement dans la logique principale, conduisant à un code plus propre.
Ne pas retourner null	Retourner null peut entraîner des vérifications fréquentes et des erreurs d'exécution ; utilisez plutôt des exceptions ou des objets de cas spéciaux.
Ne pas passer null	Évitez de passer null aux méthodes pour prévenir les NullPointerExceptions ; imposez des arguments non nuls.
Conclusion	Un code lisible et robuste considère la gestion des erreurs comme une préoccupation distincte pour améliorer la maintenabilité.

Résumé du Chapitre 6 : Gestion des erreurs

Introduction

La gestion des erreurs est cruciale en programmation car elle garantit que le code se comporte correctement lorsque des situations inattendues surviennent. Cependant, lorsque la

gestion des erreurs devient trop envahissante dans la base de code, elle peut obscurcir la logique principale, rendant le code difficile à lire et à maintenir.

Utilisez des exceptions plutôt que des codes de retour

- Historiquement, de nombreux langages de programmation utilisaient des codes d'erreur et des indicateurs qui encombraient le code d'appel.
- Lancer des exceptions est une approche plus propre ; cela permet à la logique principale de rester dégagée des vérifications d'erreur, améliorant ainsi la lisibilité du code.

Écrivez d'abord votre instruction try-catch-finally

- Commencer par une structure try-catch-finally clarifie quelles exceptions sont attendues et comment elles seront

Scanner pour télécharger

App Store
Coup de cœur

22k avis 5 étoiles

Retour Positif

Fabienne Moreau

ue résumé de livre ne testent
ion, mais rendent également
musant et engageant.
té la lecture pour moi.

Fantastique! Giselle Dubois

Je suis émerveillé par la variété de livres et de langues que Bookey supporte. Ce n'est pas juste une application, c'est une porte d'accès au savoir mondial. De plus, gagner des points pour la charité est un grand plus !

Fi
Le
liv
co
pr

é Blanchet

de lecture
eption de
es,
tous.

J'adore ! Isoline Mercier

Bookey m'offre le temps de parcourir les parties importantes d'un livre. Cela me donne aussi une idée suffisante pour savoir si je devrais acheter ou non la version complète du livre ! C'est facile à utiliser !"

Gain de temps !

Bookey est mon applicat
intellectuelle. Les résum
magnifiquement organis
monde de connaissance

Appli géniale ! Joachim Lefevre

adore les livres audio mais je n'ai pas toujours le temps
'écouter le livre entier ! Bookey me permet d'obtenir
n résumé des points forts du livre qui m'intéresse !!!
Quel super concept !!! Hautement recommandé !

Appli magnifique

Cette application est une bouée de sauve
amateurs de livres avec des emplois du te
Les résumés sont précis, et les cartes me
renforcer ce que j'ai appris. Hautement re

Essai gratuit avec Bookey

Chapitre 7 Résumé : 8 Limites

Section	Résumé
Introduction aux Limites	Ce chapitre souligne l'importance de maintenir des limites claires lors de l'intégration de composants tiers dans le développement logiciel.
Utilisation de Code Tiers	Il existe une tension entre l'applicabilité générale des interfaces et les besoins spécifiques des utilisateurs. Pour améliorer la clarté et la maintenabilité, il est conseillé de ne pas exposer directement des interfaces comme java.util.Map, mais de les encapsuler au sein de classes personnalisées.
Explorer et Apprendre les Limites	Intégrer du code tiers nécessite de réaliser des tests pour mieux comprendre le comportement des bibliothèques. Des tests courts facilitent une intégration efficace plutôt qu'une expérimentation directe en production.
Utilisation de Code Qui N'Existe Pas Encore	Créer des interfaces auto-définies permet aux équipes de travailler indépendamment de composants incomplets, aboutissant à un code plus propre et plus ciblé.
Limites Propres	Gérer efficacement les limites soutient les changements futurs avec un minimum de rework. La séparation des préoccupations et l'utilisation de modèles de conception comme l'Adaptateur aident à maintenir de faibles charges de maintenance.
Conclusion	Des limites claires sont essentielles pour la flexibilité et pour minimiser l'impact des changements tiers. Des stratégies de conception et de test appropriées peuvent atténuer les complexités liées aux dépendances externes.

Chapitre 8 : Limites

Introduction aux Limites

Dans le développement logiciel, il est courant d'intégrer des paquets tiers ou de s'appuyer sur des composants créés par d'autres équipes. Ce chapitre aborde l'importance de maintenir des limites claires entre différents composants logiciels.

Utilisation de Code Tiers

Une tension fondamentale existe entre les fournisseurs d'interfaces, qui visent une large applicabilité, et les utilisateurs, qui nécessitent de la spécificité. Par exemple, l'interface java.util.Map est polyvalente mais peut mener à des abus à cause de méthodes comme clear() et put(). Pour atténuer cela, il est préférable de ne pas exposer Map directement ; à la place, encapsulez-le dans une classe dédiée (par exemple, Capteurs) pour limiter la fonctionnalité disponible et améliorer la clarté et la maintenabilité du code.

Explorer et Apprendre les Limites

Intégrer du code tiers peut être difficile. Écrire des tests pour explorer la fonctionnalité de ces bibliothèques, appelés tests d'apprentissage, peut faciliter la compréhension et l'intégration. Par exemple, lors de l'utilisation de la bibliothèque log4j, créer de petits tests pour vérifier les comportements peut permettre de surmonter les défis d'intégration plus efficacement que d'expérimenter directement en production.

Utiliser du Code Qui N'Existe Pas Encore

Lors du développement contre des interfaces indéfinies ou inconnues, créer une interface auto-définie peut maintenir la clarté. Cette approche permet aux équipes de travailler indépendamment de composants incomplets, menant finalement à un code plus propre et plus ciblé.

Limites Propres

Gérer correctement les limites peut accueillir des changements futurs avec un minimum de retouche. Cela implique de maintenir une séparation claire des préoccupations, limitant les points dans le code où les bibliothèques tierces sont référencées. L'utilisation de modèles comme le Adaptateur peut aider à faire le lien entre les interfaces personnalisées et les API tierces et à garder la charge de maintenance basse, assurant que le système reste adaptable aux changements.

Conclusion

Maintenir des limites nettes dans les systèmes logiciels est crucial pour favoriser la flexibilité et minimiser l'impact des

changements tiers. Des stratégies de conception et de test appropriées peuvent protéger contre les complexités introduites par les dépendances externes.

Pensée critique

Point clé:La nécessité de frontières claires entre les composants logiciels est un principe fondamental du CODER PROPREMENT.

Interprétation critique:Alors que Martin insiste sur l'importance de maintenir des frontières nettes pour améliorer la clarté et la facilité de maintenance, il est crucial de considérer que cette perspective peut négliger des scénarios où une intégration plus étroite et des limites moins rigides pourraient favoriser l'innovation et l'efficacité. Par exemple, bien que l'encapsulation des bibliothèques tierces puisse protéger contre une utilisation abusive, cela pourrait également limiter la flexibilité d'utiliser ces bibliothèques de manière novatrice. Les débats sur la conception des interfaces ne sont pas nouveaux ; des approches telles que celles défendues par Eric Evans dans 'Domain-Driven Design' suggèrent que parfois, une intégration plus profonde, plutôt qu'une séparation stricte, peut mieux servir des modèles de domaine complexes. Les lecteurs devraient évaluer de manière critique si le respect strict des frontières claires, tel que présenté par Martin, est universellement applicable ou si des stratégies

alternatives pourraient s'avérer bénéfiques dans certains contextes.

Chapitre 8 Résumé : 9 Tests Unitaires

Section	Points Clés
Introduction aux Tests Unitaires	L'évolution des pratiques de programmation, un changement notable vers des méthodes structurées comme le TDD qui insiste sur l'écriture de tests avant le code.
Les Trois Lois du TDD	1. Écrire un test unitaire qui échoue avant le code de production. 2. Écrire juste assez d'un test pour qu'il échoue. 3. Écrire juste assez de code de production pour que le test échoué passe.
Maintenir des Tests Propres	Des tests propres sont essentiels au processus de test ; des tests sales entraînent des problèmes de maintenance et réduisent la confiance dans la suite de tests.
L'Importance des Tests Propres	Les tests devraient avoir des normes élevées similaires à celles du code de production, favorisant la flexibilité et réduisant la peur des changements.
Caractéristiques des Tests Propres	Lisibilité: Les tests doivent clairement exprimer leur intention. Langage de Test Spécifique au Domaine: Améliore l'expressivité. Une Assertion par Test: Se concentrer sur un concept pour la lisibilité.
Principes F.I.R.S.T. pour des Tests Propres	Rapide: Une exécution rapide encourage des tests fréquents. Indépendant: Les tests doivent s'exécuter indépendamment. Répétable: Résultats cohérents à travers les environnements. Auto-Vérifiable: Résultats clairs de réussite/échec. Opportune: Créé juste avant le code correspondant.
Conclusion	Des tests unitaires propres sont essentiels pour la santé du projet, favorisant la maintenabilité et la flexibilité, garantissant la qualité du code et permettant des changements. Un refactoring régulier des tests est crucial.

Tests Unitaires

Introduction aux Tests Unitaires

- L'évolution des pratiques de programmation et de tests

unitaires au cours de la dernière décennie, passant de tests ad hoc à des méthodologies structurées comme le développement piloté par les tests (TDD), est significative.
- Le TDD met l'accent sur l'écriture de tests unitaires avant de coder, ce qui améliore le processus de test.

Les Trois Lois du TDD

1. Écrire un test unitaire qui échoue avant d'écrire du code de production.
2. Écrire seulement autant de code de test qu'il est nécessaire pour échouer.
3. Écrire seulement autant de code de production qu'il est nécessaire pour faire passer le test échouant actuel.

Maintenir des Tests Propres

- Des tests propres sont essentiels pour maintenir l'efficacité des tests unitaires face à l'évolution du code de production.
- Des tests sales créent des charges de maintenance et peuvent conduire à un manque de confiance dans la suite de tests.

L'Importance des Tests Propres

- Les tests doivent être maintenus avec les mêmes standards que le code de production.
- Des tests propres favorisent la flexibilité et réduisent la peur de faire des modifications dans le code de production.

Caractéristiques des Tests Propres

-
Lisibilité
: La clarté et la simplicité des tests sont primordiales ; ils doivent exprimer l'intention sans être alourdis par des détails.
-
Langage de Test Spécifique au Domaine
: Créer une API de test peut améliorer l'expressivité et la clarté des tests.
-
Une Seule Assertion par Test
: Viser à minimiser le nombre d'assertions par test, en mettant l'accent sur le test d'un seul concept pour améliorer la lisibilité.

Principes F.I.R.S.T. pour des Tests Propres

-

Rapides

: Les tests doivent s'exécuter rapidement pour encourager une exécution fréquente.

-

Indépendants

: Les tests doivent s'exécuter indépendamment et ne pas dépendre les uns des autres.

-

Répétables

: Les tests doivent donner des résultats cohérents à travers différents environnements.

-

S'auto-Valident

: Les tests doivent fournir des résultats d'acceptation/rejet clairs sans interprétation subjective.

-

Opportuns

: Les tests doivent être créés juste avant le code de production correspondant.

Conclusion

- Des tests unitaires propres sont cruciaux pour la santé d'un

projet, favorisant la maintenabilité et la flexibilité. Ils aident non seulement à garantir la qualité du code, mais facilitent également les modifications et les améliorations du code. Il est essentiel de refactoriser régulièrement les tests et de maintenir leur propreté pour assurer le succès continu du projet.

Pensée critique

Point clé:L'Importance des Tests Propres

Interprétation critique:Bien que l'auteur plaide en faveur d'une adhésion stricte aux principes de tests propres, il est important de reconnaître que toutes les équipes n'ont pas les mêmes ressources ou priorités. Les tests propres, tels que décrits par Martin, favorisent effectivement la maintenabilité et la flexibilité, mais la mise en œuvre de pratiques aussi rigoureuses peut être impraticable dans tous les contextes. Les critiques pourraient soutenir que l'accent mis sur la propreté des tests pourrait entraîner une surcharge inutile pour les projets plus petits ou les équipes manquant d'expertise en tests. Par exemple, dans certains environnements agiles, la rapidité et les itérations rapides peuvent primer sur la propreté exhaustive des tests, mettant en évidence une divergence potentielle de philosophie. Des sources telles que 'The Pragmatic Programmer' d'Andrew Hunt et David Thomas présentent des points de vue alternatifs sur l'équilibre entre les tests et d'autres priorités de développement, suggérant que bien que les tests propres soient idéaux, le pragmatisme devrait guider la pratique.

Chapitre 9 Résumé : 10 Classes

Section	Résumé
Organisation des Classes	Les classes doivent être organisées en commençant par les constantes statiques publiques, suivies des variables statiques privées, des variables d'instance privées, et des fonctions publiques. Les fonctions utilitaires doivent être privées et placées après les fonctions publiques pour une meilleure lisibilité.
Encapsulation	Bien que les variables privées et les fonctions utilitaires soient privilégiées, un certain accès peut être nécessaire pour les tests, mais la confidentialité doit rester une priorité.
Les Classes Doivent Être Petites !	En respectant le Principe de Responsabilité Unique (SRP), les classes doivent être petites et se concentrer sur une seule responsabilité. Les noms de classes doivent refléter leurs responsabilités pour guider leur taille et leur objectif.
Cohésion	Les classes doivent avoir un nombre limité de variables d'instance partagées entre les méthodes pour atteindre une haute cohésion, ce qui les rend plus faciles à comprendre et à maintenir.
Maintenir la Cohésion Résulte en Beaucoup de Petites Classes	Décomposer les fonctions en parties plus petites peut révéler la nécessité de nouvelles classes, et les classes ne doivent pas accumuler de variables d'instance inutiles simplement pour être partagées entre les méthodes.
Organiser pour le Changement	Les classes doivent être conçues pour minimiser les ruptures lors des changements. L'utilisation de sous-classes permet une extension sans modifier le code existant, en respectant le Principe Ouvert-Fermé (OCP).
Isoler du Changement	Les interfaces et les classes abstraites doivent être utilisées pour réduire les dépendances directes et faciliter les tests. Par exemple, une interface `StockExchange` peut permettre des tests cohérents malgré la volatilité des données réelles.
Bibliographie	Les références incluent des textes clés sur les principes de conception orientée objet par des auteurs comme Robert C. Martin et Donald E. Knuth, qui mettent en avant les rôles, responsabilités et pratiques agiles.

Chapitre 10 : Classes

Organisation des Classes

Dans le CODER PROPREMENT, une attention particulière doit être portée aux niveaux supérieurs d'organisation du

code, notamment aux classes. Une classe doit commencer par des constantes publiques statiques, des variables privées statiques, puis des variables d'instance privées, suivies des fonctions publiques. Garder les fonctions utilitaires privées et les organiser après les fonctions publiques améliore la lisibilité.

Encapsulation

Bien que les variables privées et les fonctions utilitaires soient préférées, il arrive parfois que l'accessibilité doive être assouplie pour les tests. Néanmoins, le maintien de la confidentialité doit toujours être une priorité.

Les Classes Doivent Être Petites !

Les classes doivent être petites, respectant le principe d'avoir une seule responsabilité ou raison de changement, connu

Lire, Partager, Autonomiser

Terminez votre défi de lecture, faites don de livres aux enfants africains.

Le Concept

BOOKS FOR AFRICA × ×

Cette activité de don de livres se déroule en partenariat avec Books For Africa. Nous lançons ce projet car nous partageons la même conviction que BFA : Pour de nombreux enfants en Afrique, le don de livres est véritablement un don d'espoir.

La Règle

Gagnez 100 points → **Échangez un livre** → **Faites un don à l'Afrique**

Votre apprentissage ne vous apporte pas seulement des connaissances mais vous permet également de gagner des points pour des causes caritatives ! Pour chaque 100 points gagnés, un livre sera donné à l'Afrique.

Chapitre 10 Résumé : 11 Systèmes

Chapitre 11 : Systèmes

Introduction à la complexité et à la gestion des systèmes

- La complexité nuit au développement logiciel, rendant la planification, la construction et les tests des produits difficiles.
- Comme les villes, les systèmes logiciels nécessitent une organisation d'équipe efficace et une modularité, permettant aux composants individuels de fonctionner sans nécessiter une compréhension exhaustive.

Séparation des préoccupations

- Met l'accent sur l'importance de séparer les processus de construction de l'utilisation de l'application.
- De nombreuses applications mélangent les processus de démarrage avec la logique d'exécution, ce qui peut entraîner des complications lors des tests et de la gestion des

dépendances.

Construction efficace du système

- Le processus de démarrage doit être modulé séparément de la logique d'opération normale pour améliorer la clarté et la maintenabilité.
- Deux méthodes pour réaliser cette séparation sont :
  - Déplacer toute la logique de construction dans la fonction principale.
  - Utiliser des usines pour gérer la création d'objets sans influencer la couche de l'application.

Injection de dépendances (DI)

- La technique DI améliore la séparation des préoccupations en transférant les responsabilités de gestion des dépendances du code de l'application à des conteneurs ou frameworks externes.
- La DI permet une initialisation paresseuse, garantissant que les dépendances ne sont créées que lorsque c'est nécessaire.

Développement itératif des systèmes

- Les systèmes évoluent de manière incrémentielle, commençant par une architecture simple et s'élargissant à mesure que les besoins croissent, ce qui favorise l'agilité et la réactivité au changement.
- Les leçons majeures tirées soulignent que la planification préalable conduit souvent à des systèmes trop sophistiqués qui entravent l'adaptabilité.

Préoccupations transversales

- Aborde les défis communs avec des architectures comme EJB2 qui ne séparent pas correctement les préoccupations.
- Introduit la programmation orientée aspect (AOP) comme une méthodologie pour gérer les préoccupations transversales par des déclarations claires plutôt que par une logique codée en dur.

Proxys Java et AOP

- Les proxys Java facilitent des opérations transversales simples, bien qu'ils introduisent de la complexité.
- Les frameworks AOP comme Spring et AspectJ fournissent des outils pour mieux gérer les préoccupations transversales tout en maintenant des pratiques de CODER

PROPREMENT.

Tests et prise de décision

- Une architecture système modulaire favorise le développement dirigé par les tests et implique de minimiser les pratiques de conception invasives pour maintenir la clarté du code et faciliter les tests.

Normes et langages spécifiques au domaine (DSL)

- Encourage l'utilisation de normes lorsqu'elles apportent une réelle valeur et suggère d'utiliser des DSL pour combler le fossé entre la logique de domaine et l'implémentation du code.

Conclusion

- Une architecture système propre est cruciale pour maintenir la clarté du domaine et permettre des réponses agiles aux changements.
- Les développeurs doivent prioriser la simplicité et la modularité à tous les niveaux pour garantir la qualité et la maintenabilité dans la conception des systèmes.

Exemple

Point clé:Séparation des préoccupations

Exemple:Imaginez que vous développez une application web. Au lieu de regrouper la configuration de votre connexion à la base de données, l'authentification des utilisateurs et la logique de rendu dans un seul bloc de code, vous décidez de créer des modules distincts : un pour établir les connexions à la base de données, un autre pour gérer l'authentification des utilisateurs, et un troisième pour le rendu des pages web. Cette approche simplifie la maintenance, car vous pouvez travailler sur la logique de la base de données sans vous soucier de qui est connecté ou de l'apparence de la page, et cela vous permet de tester chaque module indépendamment, menant ainsi à un logiciel globalement plus fiable et adaptable.

Pensée critique

Point clé:Séparation des Préoccupations

Interprétation critique:La notion de séparer les processus de construction de la logique d'application est présentée comme essentielle à CODER PROPREMENT. Cependant, il est important de se demander si ce paradigme est toujours valable, certains soutenant qu'une approche plus intégrée peut favoriser une meilleure compréhension des interactions complexes dans les logiciels. Une critique de Simon Brown dans 'Architecture Logicielle pour Développeurs' postule que, bien que la séparation puisse aider à la clarté, elle peut également entraîner une fragmentation, rendant plus difficile pour les développeurs de saisir le comportement global du système. Cette perspective encourage les lecteurs à considérer les inconvénients potentiels d'une séparation stricte et à évaluer les besoins spécifiques au contexte dans la conception des systèmes.

Chapitre 11 Résumé : 12 Émergence

Émergence par Jeff Langr

Devenir Propre grâce à un Design Émergent

Dans la conception logicielle, respecter quatre règles simples peut améliorer de manière significative la structure et le design de votre code. Ces règles facilitent l'émergence d'un bon design, comme l'ont souligné les principes de Kent Beck sur le Design Simple :

1. Passe tous les tests
2. Ne contient aucune duplication
3. Exprime l'intention du programmeur
4. Minimise le nombre de classes et de méthodes

Règle de Design Simple 1 : Passe Tous les Tests

Un design doit garantir que le système se comporte comme prévu. La testabilité est cruciale ; les systèmes qui ne peuvent pas être testés ne doivent pas être déployés. Écrire des tests conduit à des classes plus petites, à usage unique, qui

respectent le Principe de Responsabilité Unique (SRP) et minimise le couplage grâce à des techniques comme l'Injection de Dépendance (DIP). L'acte de tester en continu améliore l'adhésion aux principes de la programmation orientée objet, favorisant un couplage plus faible et une cohésion plus élevée.

Règles de Design Simple 2-4 : Refactorisation

Une fois qu'un système est entièrement testé, une refactorisation incrémentale peut se produire. Chaque ajout de code nécessite une réflexion sur la dégradation du design, et si c'est le cas, des modifications doivent être effectuées tout en s'assurant que les tests passent toujours. Cette étape implique l'élimination de la duplication, l'assurance de l'expressivité et la réduction du nombre de classes et de méthodes.

Pas de Duplication

La duplication complique le design, créant des risques supplémentaires et une complexité inutile. Éliminer la duplication — qu'elle soit directe ou dans des implémentations fonctionnelles — est essentiel. Refactoriser

des méthodes similaires en une méthode partagée améliore la clarté et réduit les violations du SRP. Par exemple, en considérant un design pour les politiques de vacances, nous pouvons utiliser le motif Méthode Template pour éliminer la duplication de code à travers différents types.

Expressif

La clarté dans le code est primordiale pour la maintenabilité. Le code doit clairement exprimer son intention pour minimiser les erreurs lors des modifications futures. Atteindre l'expressivité implique de choisir des noms descriptifs, de garder les classes et les fonctions concises et d'utiliser une nomenclature standard. Des tests unitaires bien élaborés servent également de double fonction en tant que documentation par l'exemple, aidant les futurs développeurs à comprendre le code.

Classes et Méthodes Minimales

Bien que minimiser la duplication et améliorer l'expressivité soient cruciaux, la création d'un nombre excessif de petites classes peut entraîner de la confusion. Il est essentiel de trouver un équilibre et d'éviter une complexité inutile.

L'objectif est de maintenir un système global de taille manageable avec des fonctions et des classes petites.

Conclusion

Bien qu'aucune pratique simple ne puisse remplacer l'expérience, les méthodes décrites favorisent l'adhésion aux principes de design établis. Adopter des pratiques de design simples peut orienter les développeurs vers les meilleures pratiques qui nécessitent généralement un apprentissage approfondi au fil du temps.

Exemple

Point clé:Le développement piloté par les tests (TDD) améliore la qualité du code.

Exemple:Imaginez que vous construisez une nouvelle fonctionnalité pour une application. Avant même d'écrire la fonctionnalité, vous commencez par rédiger un test qui définit ce que la fonctionnalité est censée faire. Lorsque vous exécutez le test, il échoue parce que la fonctionnalité n'existe pas encore. Cela vous guide à construire juste assez de code pour le faire passer. Chaque fois que vous améliorez la fonctionnalité, vous écrivez de nouveaux tests ou modifiez ceux qui existent, en vous assurant que tout fonctionne toujours correctement. En vous en tenant à cette méthode, votre base de code reste propre et adaptable, reflétant votre intention tout en rendant clair ce que chaque partie de votre code est censée accomplir.

Pensée critique

Point clé:L'Importance du Développement Dirigé par les Tests

Interprétation critique:Langr souligne qu'un design bien testé est non-négociable pour la fiabilité. Cependant, il est essentiel de remettre en question l'efficacité absolue de cette approche, car la dépendance aux tests pourrait amener les développeurs à devenir trop dépendants des outils au lieu de favoriser une compréhension approfondie. Les critiques peuvent plaider en faveur des tests exploratoires et des méthodes agiles, qui prennent en compte les exigences dynamiques et les scénarios du monde réel (voir (Beck, K. "Développement Dirigé par les Tests : Par l'Exemple") pour des perspectives contraires). Les développeurs devraient peser l'importance d'un test rigoureux et les rendements potentiellement décroissants d'une adhésion stricte aux principes dirigés par les tests.

Chapitre 12 Résumé : 13 Concurrence

Chapitre 13 : Concurrence

Aperçu

Écrire des programmes concurrents propres représente un défi majeur par rapport aux applications à un seul fil. Ce chapitre explore la nécessité de la concurrence, les difficultés qu'elle engendre et fournit des directives pour écrire du code concurrent propre. Il aborde également les complications liées au test des applications concurrentes, soulignant la complexité du sujet.

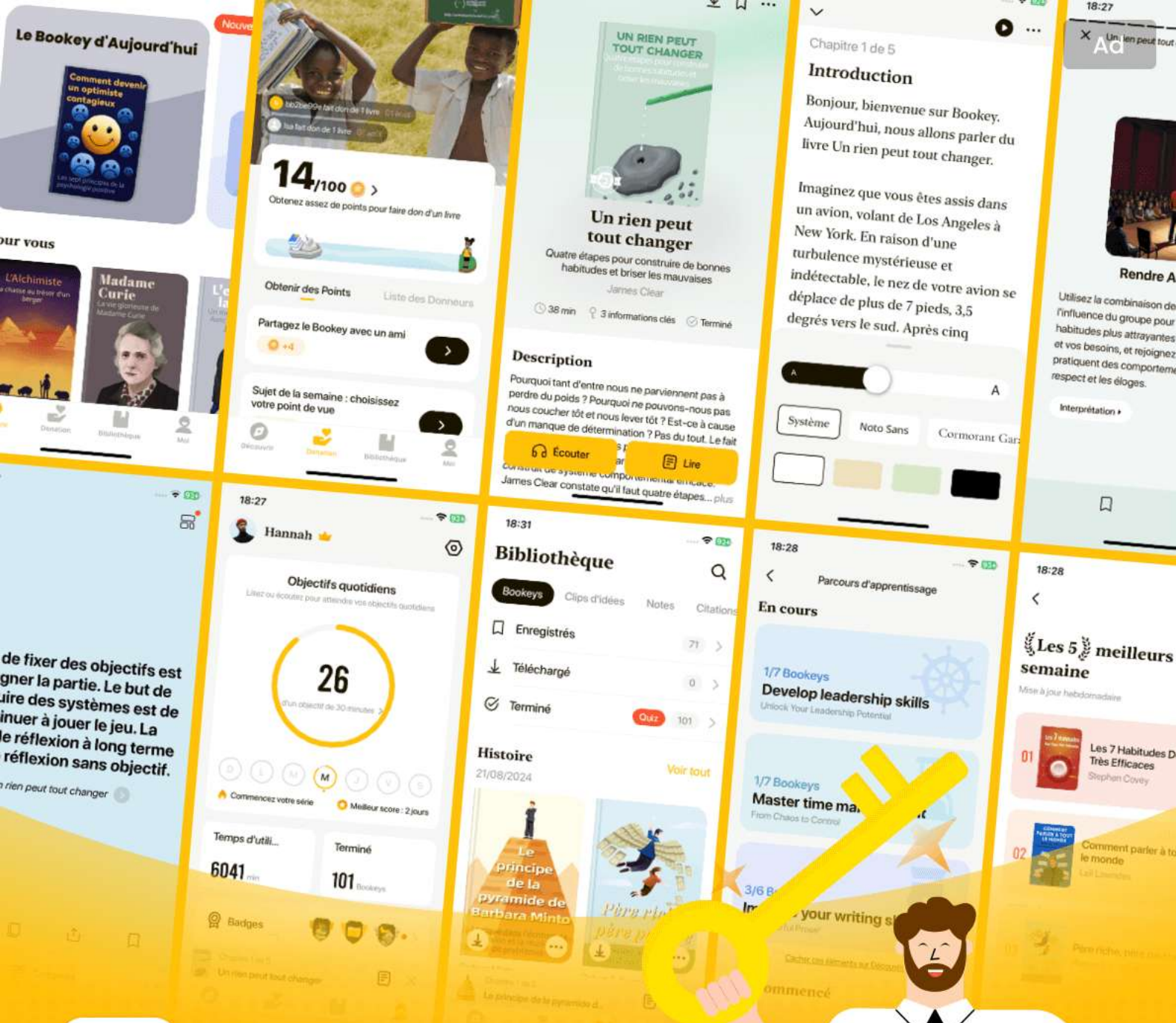
Pourquoi la Concurrence ?

La concurrence agit comme une stratégie de découplage, permettant de séparer le "quoi" de l'exécution du "quand". Cela améliore le débit et la structure de l'application, la rendant plus compréhensible et meilleure pour séparer les préoccupations. Par exemple, les applications web utilisent des servlets asynchrones pour gérer les demandes, facilitant

ainsi le processus de gestion de plusieurs demandes entrantes.
Cependant, la mise en œuvre réelle de la concurrence est semée d'embûches. La performance peut être trompeuse, les exigences de conception changent de manière significative, et même la concurrence gérée par les conteneurs nécessite une attention particulière pour éviter des problèmes tels que le blocage et les mises à jour concurrentes.

Mythes et Idées Reçues

1.
La concurrence améliore la performance
- Pas toujours ; les gains de performance dépendent du temps d'attente partagé entre les fils, ce qui peut être difficile à gérer.
2.
La conception concurrente ne change pas



Les meilleures idées du monde débloquent votre potentiel

Essai gratuit avec Bookey

Télécharger dans App Store

DISPONIBLE SUR Google Play

Scanner pour télécharger

Chapitre 13 Résumé : 14 Raffinement Successif

Résumé du Chapitre 14 : Raffinement Successif

Introduction

Ce chapitre présente une étude de cas illustrant le concept de raffinement successif en revisitant un analyseur d'arguments en ligne de commande implémenté en Java, nommé Args. L'implémentation initiale est décrite comme simple mais finit par devenir désordonnée à mesure qu'elle gagne en complexité.

Implémentation Initiale

La classe Args permet de parser des arguments en ligne de commande définis par une chaîne de schéma. La fonctionnalité de base inclut des arguments booléens, entiers et de chaîne. Cependant, le design devient encombrant en raison d'une fonctionnalité étendue qui introduit de

nombreuses variables d'instance et un traitement des erreurs compliqué.

Processus de Refactoring

Le chapitre détaille le processus de refactoring de ce code désordonné à travers des améliorations successives et incrémentales tout en maintenant une suite de tests fonctionnels. Une approche clé est d'utiliser le développement dirigé par les tests (TDD), garantissant que chaque changement maintienne le système fonctionnel.

-

Modèles de Conception

: L'introduction de l'interface ArgumentMarshaler permet d'aborder de manière structurée la gestion des différents types d'arguments sans encombrer la classe principale Args.

-

Changements Incrémentaux

: Chaque changement est minime et vise à rendre le code plus propre tout en réussissant les tests, déplaçant les fonctionnalités vers leurs classes respectives (par exemple, StringArgumentMarshaler, IntegerArgumentMarshaler).

-

Gestion des Erreurs

: Une nouvelle classe ArgsException consolide le traitement des exceptions de manière à séparer les préoccupations de la classe Args.

Structure Finale

La classe Args refactorisée devient plus modulaire, avec des responsabilités plus claires déléguées à différents composants, ce qui améliore la lisibilité et la maintenabilité. Enfin, le chapitre souligne que le code doit être continuellement gardé propre pour éviter la dégradation qui accompagne le mauvais code.

Conclusion

CODER PROPREMENT est crucial non seulement pour la fonctionnalité immédiate mais aussi pour la longévité et la santé des projets de développement. Un raffinement et une amélioration continus garantissent que le code reste gérable, évitant les pièges de la dette technique accumulée due à de mauvaises implémentations initiales ou à un développement précipité.

Exemple

Point clé:L'importance de l'affinage successif dans le développement de code.

Exemple:Imaginez que vous développez une application en ligne de commande et que vous commencez par implémenter un simple parseur d'arguments. En ajoutant de nouvelles fonctionnalités—comme la gestion de différents types d'arguments et l'amélioration des messages d'erreur—vous remarquez que votre code devient de plus en plus complexe. Chaque fois que vous ajoutez quelque chose, vous pourriez prendre un moment pour refactoriser juste une partie à la fois, en vous assurant que chaque changement est petit et maintient la fonctionnalité, conduisant à un design plus propre et plus modulaire. Ce processus itératif d'affinage peut empêcher votre code de devenir ingérable et garantir qu'il reste adaptable à mesure que votre application évolue.

Chapitre 14 Résumé : 15 Les Internes de JUnit

Les Internes de JUnit

Aperçu du Cadre JUnit

JUnit, un cadre de test Java renommé, est né des efforts collaboratifs de Kent Beck et Eric Gamma. Ils ont développé le cadre lors d'une conversation dans un avion à destination d'Atlanta, synthétisant des éléments provenant du travail précédent de Beck avec Smalltalk. L'accent de ce chapitre est mis sur le `ComparisonCompactor`, un module conçu pour rationaliser les erreurs de comparaison de chaînes.

Module ComparisonCompactor

Le `ComparisonCompactor` identifie les différences entre deux chaînes, présentant les variations dans un format clair (par exemple, `<...B[X]D...>`). Le chapitre inclut une critique approfondie de la structure de test, soulignant l'efficacité des

tests mis en œuvre dans `ComparisonCompactorTest.java`.

Revue des Cas de Test

Les cas de test couvrent divers scénarios :
- Incohérences de message
- Chaînes identiques
- Correspondances contextuelles au début et à la fin
Les tests offrent une couverture complète, garantissant que chaque ligne de code dans `ComparisonCompactor` est exécutée, contribuant ainsi à la confiance dans son fonctionnement.

Examen du Code

1.
Mise en œuvre Initiale
: La mise en œuvre originale du `ComparisonCompactor` est révisée, identifiant des points forts en clarté et en structure.
2.
Identification des Refactorisations
: Les observations sur le code révèlent des opportunités d'amélioration, notamment :
- Conventions de nommage redondantes (par exemple,

préfixes)
   - Encapsulation des conditionnels pour une meilleure lisibilité
   - Inversion des conditions négatives pour plus de clarté
3.
Refactorisation Itérative
 : Le processus de refactorisation se déroule de manière incrémentale, conduisant souvent à des réévaluations de changements précédemment réalisés. L'importance de la lisibilité et du maintien de la cohérence des variables est soulignée tout au long.

Mise en œuvre Finale

La version finale du `ComparisonCompactor` présente un module bien structuré avec des fonctions d'analyse et de synthèse distinctes. La conception suit les meilleures pratiques, garantissant que les fonctions connexes sont regroupées logiquement et que leurs définitions apparaissent près de leur utilisation dans le code.

Conclusion

Le chapitre se termine par une réflexion sur la Règle du Boy

Scout, soulignant le raffinement itératif du code. Il met en avant la responsabilité des développeurs de continuer à améliorer la qualité de la base de code, favorisant une culture d'excellence et de maintenabilité.

Chapitre 15 Résumé : 16 Refactorisation de SerialDate

Refactorisation de SerialDate : Aperçu

Dans ce chapitre, Robert C. Martin effectue un examen détaillé et une refactorisation de la classe SerialDate de la bibliothèque JCommon. Malgré la compétence de l'auteur original, l'analyse met en évidence plusieurs axes d'amélioration, en mettant l'accent sur la critique professionnelle comme outil d'apprentissage pour les développeurs.

Critique et Raisons de la Refactorisation

L'auteur reconnaît la qualité initiale du code mais entame le processus de refactorisation dans le but d'améliorer la couverture des tests et la fonctionnalité globale. Les observations clés comprennent :

1.

Couverture de Test Insuffisante

: Les tests initiaux ne couvraient pas adéquatement la

fonctionnalité de SerialDate, ce qui a conduit à la création d'une suite de tests plus complète.

2.

Bugs et Flaws Logiques

: La refactorisation identifie des bugs spécifiques, tels que le traitement incorrect des conditions limites et des fonctionnalités de méthode qui contredisaient les attentes.

Étapes de Refactorisation : Faire Fonctionner, Puis Rendre Correct

1.

Amélioration de la Couverture des Tests

: La première étape a consisté à écrire de nouveaux tests qui byraient des zones préoccupantes. Les tests existants étaient souvent commentés en raison de cas échoués.

2.

Correction des Bugs et Clarté du Code

Chapitre 16 Résumé : 17 Odeurs et Heuristiques

Chapitre 17 : Odeurs et Heuristiques

Dans ce chapitre, Robert C. Martin compile une liste complète des "Odeurs de Code" et des heuristiques associées qui aident les programmeurs à reconnaître les mauvaises pratiques dans le code et à les refactoriser en CODER PROPREMENT.

Commentaires

-
Informations Inappropriées
 : Les commentaires ne devraient pas contenir de données historiques plus adaptées aux systèmes de versionnage.
-
Commentaire Obsolète
 : Les commentaires obsolètes doivent être mis à jour ou supprimés pour éviter toute confusion.
-

Commentaire Redondant
: Les commentaires qui se contentent de répéter ce que le code exprime sont inutiles.
-
Commentaire Mal Rédigé
: Les commentaires doivent être bien écrits, clairs et concis.
-
Code Commenté
: Enlevez le code commenté pour éviter le désordre et la confusion.

Environnement

-
Construction Nécessite Plus D'une Étape
: Le processus de construction doit être simplifié et efficace.
-
Tests Nécessitent Plus D'une Étape
: L'exécution des tests doit être simple et rapide.

Fonctions

-
Trop D'Arguments

: Limitez les arguments de fonction à trois ou moins pour plus de clarté.

-

Arguments de Sortie

: Évitez d'utiliser des arguments comme sorties ; les fonctions doivent modifier l'état de l'objet auquel elles appartiennent.

-

Arguments de Drapeau

: Éliminez les drapeaux booléens qui troublent l’objectif de la fonction.

-

Fonction Inactive

: Supprimez les fonctions inutilisées pour garder le code propre.

Principes Généraux

-

Langues Multiples Dans Un Fichier Source

: Efforcez-vous d'utiliser une seule langue par fichier pour plus de clarté.

-

Comportement Évident Non Implémenté

: Les fonctions doivent fournir un comportement attendu pour maintenir la confiance.

-

Comportement Incorrect Aux Limites

: Toujours tester les conditions limites.

-

Sécurités Plus Réglées

: Évitez de contourner les mécanismes de sécurité dans le code.

-

Duplication

: Éliminez le code dupliqué car il représente des occasions d'abstraction manquées.

-

Code Au Mauvais Niveau D'Abstraction

: Maintenez une séparation entre les concepts de code de haut niveau et de bas niveau.

-

Classes de Base Dépendant de Leurs Dérivées

: Les classes de base doivent rester indépendantes de leurs dérivées.

Odeurs de Conception

-

Trop D'Informations

: Gardez les interfaces de module petites pour minimiser la complexité.

-

Code Mort

: Enlevez le code qui n'est jamais exécuté.

-

Séparation Verticale

: Gardez la définition proche de l'utilisation pour plus de clarté.

-

Incohérence

: Maintenez des noms et des méthodologies uniformes dans l'ensemble du code.

-

Désordre

: Enlevez les constructions inutiles ou inutilisées du code.

Cohésion et Complexité

-

Cohésion Artificielle

: Ne coupez pas des composants non liés ensemble sans

raison.

-

Envie de Fonctionnalité

: Assurez-vous que les méthodes utilisent les attributs de leur propre classe plutôt que ceux d'une autre.

-

Arguments de Sélecteur

: Évitez d'utiliser des arguments de sélecteur ambigus dans les signatures de méthode.

-

Intention Obscurcie

: Écrivez du code clair et expressif pour communiquer l'intention.

-

Responsabilité Mal Placée

: Placez la fonctionnalité là où elle appartient logiquement.

Algorithme et Logique

-

Comprendre l'Algorithme

: Assurez-vous de comprendre les algorithmes utilisés, ne vous fiez pas uniquement à la réussite des tests.

-

Rendre les Dépendances Logiques Physiques
 : Établissez des dépendances explicites entre les modules.
-
Préférez le Polymorphisme aux If/Else ou Switch/Case
 : Réduisez la complexité grâce à des méthodes polymorphiques plutôt qu'à des commutateurs.

Nommage et Tests

-
Choisissez des Noms Descriptifs
 : Utilisez des noms clairs et informatifs pour les variables et les fonctions.
-
Couverture de Test
 : Maintenez des tests complets pour couvrir tous les problèmes potentiels.
-
Tester les Conditions Limites
 : Portez une attention particulière aux limites des données d'entrée et aux cas extrêmes.

Conclusion

Ce chapitre renforce l'idée que le CODER PROPREMENT découle d'un ensemble de valeurs et de principes plutôt que de règles rigides. Les heuristiques et les odeurs servent à guider les développeurs vers de meilleures pratiques, conduisant finalement à un artisanat amélioré dans le développement logiciel.

Chapitre 17 Résumé : A : Concurrence II

Résumé du Chapitre 17 : Concurrence II

Vue d'ensemble

Ce chapitre offre des éclairages sur la concurrence et le multithreading en programmation, en mettant particulièrement l'accent sur les applications client/serveur et l'amélioration des performances grâce au traitement concurrent.

Exemple Client/Serveur

- Un serveur de base attend des connexions clients, traite les messages entrants, puis renvoie des réponses.
- Un client simple se connecte au serveur, envoie une requête et traite la réponse.

Mise en œuvre du Serveur

- La mise en œuvre du serveur consiste à accepter les

connexions et à les traiter de manière séquentielle.
- Un test de performance valide que le serveur peut traiter les requêtes des clients dans un délai défini.

Identification des Goulots d'Étranglement de Performance

- Les performances peuvent être limitées par l'E/S (par exemple, attendre des données d'un socket) ou par le processeur (par exemple, les tâches de calcul).
- Si le serveur est limité par l'E/S, le multithreading peut améliorer l'efficacité en permettant d'autres opérations pendant les périodes d'attente d'E/S.

Introduction du Multithreading pour Améliorer les Performances

- Le multithreading peut aider à améliorer le débit si l'application est limitée par l'E/S.
- Le chapitre traite de la modification du serveur pour gérer les connexions clients en utilisant des threads séparés.

Séparation des Responsabilités

- La gestion des threads doit être isolée pour maintenir un CODER PROPREMENT; fusionner plusieurs responsabilités dans une seule méthode viole le Principe de Responsabilité Unique.
- Le code conçu pour le multithreading doit se concentrer exclusivement sur la gestion des threads.

Modèles de Conception pour la Gestion des Threads

- Des abstractions appropriées, telles que la séparation de la connexion client et du traitement des requêtes, améliorent la clarté et la maintenabilité du code.

Défis de la Concurrence

- La complexité du multithreading crée des problèmes potentiels comme les conditions de course et les interblocages.
- Comprendre comment gérer efficacement les ressources partagées est crucial pour éviter les problèmes de concurrence.

Prévention des Interblocages

- Le chapitre décrit les conditions nécessaires à l'occurrence des interblocages et les stratégies de prévention, qui incluent l'évitement des attentes circulaires et l'établissement d'une hiérarchie de verrouillage.

Tests de Code Multithread

- Des défis de test émergent en raison de la nature imprévisible de la concurrence.
- Des stratégies comme le Test de Monte Carlo et l'utilisation d'outils (comme ConTest) peuvent aider à exposer plus efficacement les problèmes de threading.

Conclusion

- Une gestion efficace de la concurrence nécessite des approches de conception et de test soigneuses pour éviter les pièges courants.
- Comprendre les principes de la concurrence est essentiel pour construire des applications évolutives et efficaces. Une lecture approfondie des techniques de programmation concurrente, comme les travaux de Doug Lea, est recommandée pour une compréhension plus approfondie.

Pensée critique

Point clé:Utilisation des threads pour améliorer les performances

Interprétation critique:Bien que le chapitre souligne l'utilisation des threads comme solution pour améliorer la performance des serveurs, il est important de se demander si plus de threads équivalent toujours à une meilleure efficacité ; les subtilités de la concurrence peuvent mener à des problèmes complexes et à des rendements décroissants, suggérant que le point de vue de l'auteur n'est peut-être pas universellement applicable dans tous les contextes. Des ouvrages complémentaires, comme "Concurrency in Go" de Katherine Cox-Buday, mettent en avant des paradigmes alternatifs qui peuvent gérer efficacement des tâches concurrentes sans recourir uniquement aux threads traditionnels.

Chapitre 18 Résumé : B: org.jfree.date.SerialDate

Résumé du Chapitre 18 - CODER PROPREMENT

Aperçu de la classe SerialDate

La classe `SerialDate` offre une abstraction pour la manipulation des dates qui évite d'être liée à une implémentation spécifique. Elle propose une représentation plus simple que `java.util.Date`, en se concentrant sur le jour, le mois et l'année sans précision temporelle ni problèmes de fuseau horaire.

Caractéristiques principales :

-
Classe immuable :
La conception garantit que les instances de `SerialDate` sont immuables.
-

Compatibilité avec Excel :

La classe répond à des exigences spécifiques basées sur la gestion des dates par Microsoft Excel.

-

Options de constructeur :

Permet de créer des dates à travers plusieurs méthodes de fabrique, y compris à partir de représentations `int` et de la classe `java.util.Date`.

Méthodes utilitaires :

- Méthodes pour obtenir le jour de la semaine, le mois et l'année.
- Utilitaires de conversion entre chaînes de caractères et représentations de dates.
- Fonctions pour ajouter et soustraire des jours, des mois et des années à une date.

Scanner pour télécharger

Pourquoi Bookey est une application incontournable pour les amateurs de livres

Contenu de 30min
Plus notre interprétation est profonde et claire, mieux vous saisissez chaque titre.

Format texte et audio
Absorbez des connaissances même dans un temps fragmenté.

Quiz
Vérifiez si vous avez maîtrisé ce que vous venez d'apprendre.

Et plus
Plusieurs voix & polices, Carte mentale, Citations, Clips d'idées...

Essai gratuit avec Bookey

Chapitre 19 Résumé : C : Références croisées des heuristiques

Annexe C : Références croisées des heuristiques

Présentation

Cette annexe fournit une référence croisée détaillée de diverses heuristiques et des odeurs de code associées tirées du livre "CODER PROPREMENT" de Robert C. Martin.

Liste des heuristiques

-
C1
: 16-276A, 16-279A, 17-292A
-
C2
: 16-279A, 16-285A, 16-295A, 17-292A
-
C3

: 16-283A, 16-285A, 16-288A, 17-293A

-

C4

: 17-293A

-

C5

: 17-293A

-

E1

: 17-294A

-

E2

: 17-294A

-

F1

: 14-239A, 17-295A

-

F2

: 17-295A

-

F3

: 17-295A

-

F4

: 14-289A, 16-273A, 16-285A, 16-287A, 16-288A, 17-295A

-

G1

: 16-276A, 17-295A

-

G2

: 16-273A, 16-274A, 17-296A

-

G3

: 16-274A, 17-296A

-

G4

: 9-31A, 16-279A, 16-286A, 16-291A, 17-297A

-

G5

: 9-31A, 16-279A, 16-286A, 16-291A, 16-296A, 17-297A

-

G6

: 6-106A, 16-280A, 16-283A, 16-284A, 16-289A, 16-293A, 16-294A, 16-296A, 17-299A

-

G7

: 16-281A, 16-283A, 17-300A

-

G8

: 16-283A, 17-301A

-

G9

: 16-283A, 16-285A, 16-286A, 16-287A, 17-302A

-

G10

: 5-86A, 15-264A, 16-276A, 16-284A, 17-302A

-

Autres heuristiques

: Des heuristiques supplémentaires (G11 à T9) sont listées avec leurs références croisées respectives.

Conclusion

Cette référence croisée sert de ressource essentielle pour les développeurs cherchant à améliorer la qualité du code en identifiant les odeurs potentielles de code et en appliquant les heuristiques appropriées dans leurs pratiques de programmation.

Chapitre 20 Résumé : Index

Résumé du Chapitre 20

Concepts Clés et Principes

- Détection des erreurs d'opérateur, importance de CODER PROPREMENT pour éviter les ambiguïtés.
- Les classes abstraites et les interfaces sont essentielles pour une bonne abstraction et structuration du code.
- La loi de Demeter souligne le couplage minimal et la facilité de maintenance, mettant en avant l'utilisation de fonctions d'accès.

Qualité du Code

- Compréhension du code de mauvaise qualité, du désordre, et de la valeur d'un code propre.
- Importance des commentaires, à la fois comme amplificateurs d'importance et comme maux nécessaires pour clarifier le contexte.
- Gestion de la complexité, en mettant l'accent sur de petites

fonctions, la cohésion des classes et le maintien d'un style cohérent.

Tests et Fiabilité

- Accent sur les cadres de tests automatisés comme JUnit pour assurer des pratiques de CODER PROPREMENT.
- Importance d'écrire des tests qui sont auto-validants et réduisent le nombre de dépendances.
- Nécessité de tests opportunes qui valident selon les principes de conception et préservent contre les changements.

Refactoring et Maintenance

- Attention constante à maintenir la santé du code pour éviter la dette technique.
- Mise en œuvre du refactoring comme un processus continu, et non pas juste un correctif ponctuel.
- Accent sur les conventions de nommage et les identifiants descriptifs comme essentiels pour comprendre et maintenir la clarté du code.

Concurrence et Performance

- Discussion détaillée sur l'évitement des pièges dans la programmation concurrente.
- Stratégies pour s'assurer que les applications multi-threadées sont efficaces et exemptes de scénarios d'interblocage courants.
- Application pratique des modèles de conception qui améliorent la structure du code tout en facilitant l'évolutivité.

Meilleures Pratiques et Modèles

- Utilisation de modèles de conception comme Abstract Factory et Decorator pour favoriser un code propre et maintenable.
- Une compréhension solide des principes SOLID pour la conception orientée objet soutient l'adaptabilité à long terme.
- Modèles d'échec, redondance, et le concept de Ne Pas Répéter Vous-Même (DRY) comme lignes directrices essentielles.

Conclusion

- Respect des principes de CODER PROPREMENT améliore la robustesse, la maintenabilité et l'utilisabilité du code.
- L'apprentissage continu et l'adaptation sont vitaux pour les

développeurs pour rester pertinents et produire des logiciels de haute qualité.
- Accent sur le rôle du programmeur en tant qu'auteur, responsable de fournir clarté et intention à travers le code.

Pensée critique

Point clé:L'importance des pratiques de CODER PROPREMENT est peut-être simplifiée à l'extrême.

Interprétation critique:Alors que Robert C. Martin défend les pratiques de CODER PROPREMENT comme étant fondamentales pour la fiabilité des logiciels, il faut considérer que la qualité du code peut dépendre du contexte et être subjective. Certains développeurs soutiennent qu'un accent excessif sur la propreté du code peut entraîner une 'paralysie d'analyse', où la quête de la perfection freine le progrès et l'innovation (N. M. Thomas, 'Le Programmeur Pragmatique'). De plus, le respect rigoureux de certains principes ne donne pas toujours des avantages tangibles dans des scénarios réels, comme le soulignent les discussions sur la 'Dette Technique' de Martin Fowler. Cela suggère que, bien que le CODER PROPREMENT soit important, il est impératif de l'équilibrer avec des considérations pragmatiques spécifiques à la dynamique du projet et de l'équipe.

Chapitre 21 Résumé : Introduction Préalable

Résumé du Chapitre 21 : Professionnalisme en Programmation

Introduction au Professionnalisme

- L'auteur souligne l'importance du professionnalisme en programmation, s'appuyant sur 421 ans d'expériences personnelles dans le domaine.
- Le chapitre vise à définir les attitudes, disciplines et actions qui caractérisent un programmeur professionnel.

Expériences Initiales

- L'auteur se souvient de ses débuts en tant que programmeur à l'âge de 17 ans, où il était initialement peu professionnel.
- Sa première mission consistait à éditer des manuels d'ordinateur IBM et à écrire un programme basique, mettant en évidence une courbe d'apprentissage abrupte.

Apprendre par les erreurs

- L'auteur détaille ses premières expériences de programmation avec des processus manuels, tels que le codage sur des formulaires et l'utilisation de machines à perforer.
- Il a rencontré des défis avec des erreurs de programme et a appris à dépanner avec l'aide de collègues expérimentés.

Parcours Professionnel

- Après avoir travaillé brièvement dans diverses fonctions, l'auteur est devenu programmeur à plein temps et a contribué de manière significative à un système de comptabilité en temps réel.
- Lui et ses collègues ont quitté leur emploi par frustration face à des augmentations insuffisantes, illustrant la volatilité

Scanner pour télécharger

App Store
Coup de cœur

22k avis 5 étoiles

Retour Positif

Fabienne Moreau

ue résumé de livre ne testent
ion, mais rendent également
musant et engageant.
té la lecture pour moi.

Fantastique! Giselle Dubois

Je suis émerveillé par la variété de livres et de langues que Bookey supporte. Ce n'est pas juste une application, c'est une porte d'accès au savoir mondial. De plus, gagner des points pour la charité est un grand plus !

Fi
Le
liv
co
pr

é Blanchet

de lecture
eption de
es,
tous.

J'adore ! Isoline Mercier

Bookey m'offre le temps de parcourir les parties importantes d'un livre. Cela me donne aussi une idée suffisante pour savoir si je devrais acheter ou non la version complète du livre ! C'est facile à utiliser !"

Gain de temps !

Bookey est mon applicat
intellectuelle. Les résum
magnifiquement organis
monde de connaissance

Appli géniale ! Joachim Lefevre

adore les livres audio mais je n'ai pas toujours le temps
'écouter le livre entier ! Bookey me permet d'obtenir
n résumé des points forts du livre qui m'intéresse !!!
Quel super concept !!! Hautement recommandé !

Appli magnifique

Cette application est une bouée de sauve
amateurs de livres avec des emplois du te
Les résumés sont précis, et les cartes me
renforcer ce que j'ai appris. Hautement re

Essai gratuit avec Bookey

Chapitre 22 Résumé : 1 Professionnalisme

PROFESSIONNALISME

Introduction au Professionnalisme

Devenir un développeur logiciel professionnel signifie embrasser à la fois la fierté et la responsabilité. Les professionnels assument la responsabilité de leurs actions, contrairement aux non-professionnels qui rejettent la faute sur les autres.

Prendre des Responsabilités

Le professionnalisme incarne le principe de prendre des responsabilités pour son travail, illustré par des expériences passées qui soulignent l'importance des tests et de la responsabilité. Un véritable professionnel apprend de ses erreurs et assume les fautes.

Primauté de l'Intérêt

Pour maintenir un standard professionnel, les développeurs doivent s'efforcer de ne pas introduire de bugs dans leur logiciel. Il est vital que le logiciel fonctionne de manière fiable, et les professionnels doivent accepter l’inévitabilité des erreurs tout en travaillant assidûment pour réduire leur occurrence. Des soumissions de code négligentes compromettent la qualité du travail et violent les normes éthiques.

Connaître Son Domaine

Une compréhension approfondie des concepts fondamentaux, des principes et des terminologies en ingénierie logicielle est essentielle. Les professionnels doivent se tenir au courant des avancées tant historiques que contemporaines dans le domaine. Un apprentissage régulier à travers divers formats, y compris des livres et des conférences, est crucial.

Apprentissage et Pratique Continue

La nature rapide du secteur logiciel nécessite un apprentissage et une pratique constants. Les professionnels

perfectionnent leurs compétences par divers exercices, de la même manière que les musiciens pratiquent leur art de manière régulière.

Collaboration et Mentorat

Le véritable apprentissage se produit souvent par la collaboration avec des pairs. Les professionnels en herbe devraient s'efforcer de travailler ensemble et de mentoriser les autres, améliorant ainsi leurs propres connaissances et aidant à la croissance des juniors.

Compréhension de Son Domaine

Un développeur logiciel professionnel doit posséder une compréhension de base du domaine dans lequel il travaille. Une connaissance adéquate permet aux développeurs de mieux comprendre les spécifications commerciales, en reconnaissant les divergences lorsqu'elles se présentent.

S'Identifier à Son Employeur/Client

Le développement professionnel est ancré dans l'empathie et la compréhension des besoins de l'employeur. En alignant les

objectifs personnels avec ceux de l'organisation, un développeur peut fournir de meilleures solutions.

Humilité dans le Professionnalisme

Bien que la confiance soit essentielle en programmation, l'humilité est tout aussi vitale. Reconnaître ses limites et le potentiel d'échec favorise la croissance. De vrais professionnels équilibrent assurance et reconnaissance de leur faillibilité.

Conclusion

Le professionnalisme dans le développement logiciel est un concept multifacette impliquant responsabilité, apprentissage continu, collaboration et humilité. En adhérant à ces principes, les développeurs peuvent susciter le respect et maintenir un haut standard dans leur domaine.

Chapitre 23 Résumé : 2 Dire Non

DIRE NON

Introduction

- Une anecdote personnelle de l'auteur sur un projet de système comptable en temps réel qui a mal tourné à cause de délais irréalistes et de mauvaises décisions de gestion.
- Met en lumière l'importance de dire "non" dans les contextes professionnels, surtout lorsqu'on est confronté à des demandes déraisonnables.

Rôles Adverses

- Discute des dynamiques entre les managers et les développeurs, en soulignant que la confrontation peut mener à de meilleurs résultats.
- Les professionnels sont censés s'opposer aux demandes irréalistes au lieu de simplement approuver tout ce que disent leurs supérieurs.

Négociation pour de Meilleurs Résultats

- Illustre l'importance de la négociation pour atteindre des objectifs communs.
- Encourage les professionnels à représenter de manière assertive leurs estimations et capacités afin d'éviter les malentendus et les déceptions.

Situations à Hauts Enjeux

- Insiste sur le fait que dire non devient d'autant plus crucial lorsque les enjeux sont élevés. Fournir les meilleures informations peut nécessiter de résister à la pression managériale.

Joueur d'Équipe vs. Fausse Coopération

- Définit ce que signifie être un vrai joueur d'équipe : plaider pour des objectifs réalistes plutôt que d'approuver simplement des délais impossibles.
- Met en avant la manière dont certaines personnes exploitent le terme "joueur d'équipe" pour manipuler des situations à des fins personnelles.

Les Dangers du "Essayer"

- Critique la notion de "essayer" de réaliser des tâches, en raisonnant sur la nécessité d'engagements clairs plutôt que de promesses vagues.
- Encourage les professionnels à avoir des plans d'action concrets plutôt que de simuler des efforts ou de prolonger les délais sans fondement.

Agression Passive

- Discute des risques du comportement passif-agressif en milieu de travail et de la nécessité d'une communication directe pour éviter les malentendus et les échecs de projet.

Le Coût de Dire Oui

- Explore une étude de cas d'un développeur qui s'est trop engagé sur un projet avec un calendrier et une portée irréalistes, entraînant une qualité de travail médiocre et un épuisement personnel.
- Met en lumière que, bien que les clients ne puissent pas comprendre les subtilités du codage de qualité, les développeurs doivent maintenir leurs normes et dire non

lorsque cela est nécessaire.

Conclusion

- Conclut en réaffirmant l'importance du professionnalisme dans le développement logiciel.
- Soutient que du bon code n'est pas impossible mais nécessite que les développeurs disent non à des demandes déraisonnables et maintiennent leur intégrité professionnelle.

Pensée critique

Point clé:L'importance de dire 'non' à des demandes déraisonnables dans un contexte professionnel.

Interprétation critique:Bien que la perspective de l'auteur sur l'advocacy de buts réalistes soit précieuse, elle pourrait négliger la complexité des dynamiques de travail et le potentiel de négociation qui pourrait aboutir à des accords amicaux. L'affirmation selon laquelle dire 'non' est toujours la meilleure stratégie doit être abordée avec prudence, car il peut y avoir des situations où le compromis mène à des résultats mutuellement bénéfiques ou favorise un environnement collaboratif. Diverses études, telles que celles de Robert Cialdini sur l'influence et la persuasion, suggèrent que l'équilibre entre assertivité et empathie peut être crucial pour naviguer dans les relations professionnelles, et qu'une approche collaborative peut parfois donner de meilleurs résultats à long terme que des refus catégoriques.

Chapitre 24 Résumé : 3 Dire Oui

DIRE OUI

Introduction à l'ER et l'Engagement

Ce chapitre commence par une anecdote personnelle de Robert C. Martin sur l'invention de la messagerie vocale, précisément "Le Réceptionniste Électronique" (ER), mettant en lumière les défis pour obtenir le soutien et l'engagement d'une entreprise. Il souligne la nécessité de la responsabilité personnelle lorsqu'il s'agit de faire des engagements.

Un Langage d'Engagement

Roy Osherove aborde les trois composantes de l'engagement : dire que vous le ferez, le penser réellement, et le faire réellement. Il souligne que beaucoup de gens ne tiennent pas leurs promesses, utilisant souvent un langage vague qui reflète un manque de véritable engagement. Des mots comme "avoir besoin," "espérer," "souhaiter," et "faisons" signalent des attitudes non engageantes, tandis que des phrases

définitives comme "je le ferai" reflètent une véritable responsabilité.

Reconnaître le Manque d'Engagement

Le texte met en avant l'importance du langage pour comprendre les niveaux d'engagement et décrit des phrases spécifiques qui indiquent un manque d'engagement. Reconnaître ces mots aide les individus à identifier les comportements non engageants en eux-mêmes et chez les autres.

À Quoi Ressemble un Engagement

Un véritable engagement se manifeste par des déclarations claires et définitives sur les actions que l'on va entreprendre, ce qui permet de rendre compte. Le chapitre insiste sur le fait que les engagements doivent être pris uniquement concernant

Lire, Partager, Autonomiser

Terminez votre défi de lecture, faites don de livres aux enfants africains.

Le Concept

BOOKS FOR AFRICA × ×

Cette activité de don de livres se déroule en partenariat avec Books For Africa. Nous lançons ce projet car nous partageons la même conviction que BFA : Pour de nombreux enfants en Afrique, le don de livres est véritablement un don d'espoir.

La Règle

Gagnez 100 points → **Échangez un livre** → **Faites un don à l'Afrique**

Votre apprentissage ne vous apporte pas seulement des connaissances mais vous permet également de gagner des points pour des causes caritatives ! Pour chaque 100 points gagnés, un livre sera donné à l'Afrique.

Chapitre 25 Résumé : 4 Codage

CHAPITRE 25 : CODAGE

Introduction au Codage

- Ce chapitre aborde le codage comme une activité intellectuelle influencée par le comportement personnel, l'humeur et les attitudes.
- L'auteur partage des expériences personnelles concernant l'importance de la confiance et du sens de l'erreur dans le codage.

Préparation

- Le codage demande une concentration intense, car plusieurs facteurs doivent être gérés :
  1. Le code doit fonctionner comme une solution au problème posé.
  2. Répondre aux véritables besoins du client, qui peuvent différer de ses exigences déclarées.
  3. S'assurer que le code s'intègre bien dans le système

existant sans augmenter la complexité.
4. Écrire un code lisible qui exprime clairement l'intention.
- Les distractions peuvent entraîner des résultats de codage médiocres. Il est essentiel d'éliminer les distractions et de maintenir une clarté mentale.

Impact de l'Environnement et de l'État d'Esprit

- Coder lorsqu'on est fatigué ou distrait conduit à de mauvais résultats, comme l'illustre une anecdote impliquant une mauvaise expérience de codage à 3 heures du matin.
- Les préoccupations personnelles peuvent affecter la concentration. Trouver du temps dédié pour traiter les inquiétudes améliore la productivité.

Zone de Flux

- Entrer dans le "flux" ou la "zone" peut sembler productif mais peut conduire à négliger des aspects cruciaux du design.

- Des stratégies comme faire des pauses ou la programmation en binôme peuvent aider à maintenir une perspective plus large.

Musique de Fond et Interruptions

- Les expériences personnelles montrent que la musique peut distraire du codage plutôt que d'aider à la concentration.
- Le professionnalisme implique d'être courtois face aux interruptions ; les distractions gérables peuvent être traitées en collaboration par des techniques comme la programmation en binôme.

Blocages de Codage

- Les blocages mentaux en codage peuvent parfois être surmontés en travaillant avec un partenaire, ce qui favorise un changement physiologique qui encourage la créativité.
- S'engager avec des idées créatives provenant de divers domaines, en particulier la science-fiction, peut inspirer la créativité en codage.

Débogage

- Les expériences de débogage sous haute pression soulignent le besoin de meilleurs outils et pratiques, comme le Développement Dirigé par les Tests (TDD), qui peuvent

réduire considérablement le temps de débogage.

Gestion de Son Temps

- Le développement logiciel nécessite une gestion durable de l'énergie et de la créativité. Reconnaître quand s'éloigner et revenir rafraîchi permet de mieux résoudre les problèmes.

Gestion des Délais

- Une gestion efficace des retards implique des évaluations de progression honnêtes et basées sur des faits. Évitez de tomber dans le piège de la livraison fallacieuse ; établissez une définition claire de « terminé » avec des tests d'acceptation automatisés.

Éthique Professionnelle et Collaboration

- La programmation est complexe et nécessite souvent collaboration ; offrir de l'aide et solliciter de l'assistance est une obligation professionnelle.
- Le mentorat est essentiel pour le développement des programmeurs juniors ; les développeurs chevronnés ont la

responsabilité de les guider.

Conclusion

- Le succès en codage et en développement logiciel repose sur le maintien d'une clarté mentale, la gestion du stress, la promotion de la collaboration et le respect des pratiques éthiques tant dans les contextes personnels que d'équipe.

Pensée critique

Point clé:L'environnement de codage a un impact significatif sur la productivité et la qualité du code.

Interprétation critique:Bien que Robert C. Martin mette l'accent sur l'état d'esprit et le rôle de l'environnement dans le succès de la programmation, cette perspective peut exagérer l'importance du contrôle individuel. Des philosophes comme Karl Popper soutiennent que la cognition humaine est contrainte par des facteurs externes, ce qui suggère que les résultats du codage pourraient également être influencés par des problèmes systémiques ou organisationnels au-delà de la concentration et de la préparation personnelles. Ainsi, bien que la confiance personnelle et la prise de conscience des erreurs soient cruciales, elles ne représentent qu'une partie d'un tableau beaucoup plus large impliquant la dynamique d'équipe et l'environnement de travail, qui doivent également être considérés de manière critique.

Chapitre 26 Résumé : 5 Développement Driven par les Tests

DÉVELOPPEMENT DRIVEN PAR LES TESTS

Introduction au TDD

- Le TDD a émergé il y a plus de dix ans dans le cadre du mouvement de la Programmation Extrême (XP) et a été largement adopté dans les méthodologies Agile.
- L'auteur a initialement abordé le TDD avec scepticisme, mais l'a ensuite adopté après avoir appris directement de son promoteur, Kent Beck.

Expériences clés avec le TDD

- L'expérience de l'auteur en codant avec Kent Beck a mis en lumière l'efficacité du TDD, illustrant de courts cycles avec une exécution rapide des tests et du code.
- La réalisation d'atteindre des cycles de développement rapides similaires à ceux des langages interprétés a été un

tournant.

Les Trois Lois du TDD

1. Aucun code de production ne peut être écrit tant qu'un test unitaire échoue.
2. N'écrivez que ce qu'il faut d'un test unitaire pour qu'il échoue (un échec de compilation est acceptable).
3. N'écrivez que suffisamment de code de production pour faire passer le test actuellement échoué.

Ces lois facilitent un processus d'itération rapide, favorisant le développement simultané du code de test et du code de production.

Avantages du TDD

-
Certitude
 : Des tests fréquents garantissent que les changements de code n'introduisent pas de nouveaux bugs. Une couverture élevée des tests unitaires renforce la confiance dans la base de code.
-
Taux d'injection de défauts

: Le TDD contribue à un taux de défauts plus bas et a montré une réduction significative des défauts dans diverses organisations.

-

Courage

: Avec une suite fiable de tests, les développeurs peuvent refactoriser ou CODER PROPREMENT sans craindre d'introduire de nouveaux problèmes.

-

Documentation

: Les tests unitaires servent de documentation pratique, démontrant efficacement comment le code doit être utilisé.

-

Amélioration de la conception

: Le besoin de tests encourage une meilleure conception, favorisant des structures de code découplées et maintenables.

Adoption professionnelle du TDD

- Le TDD est présenté comme une discipline professionnelle qui améliore les pratiques de développement. Ne pas utiliser le TDD peut être considéré comme non professionnel.

Limitations du TDD

- Le TDD n'est pas une panacée ; suivre ses lois ne garantit pas un bon code ou de bons tests.
- Il existe des situations où le TDD peut ne pas être pratique ou adapté, et les professionnels devraient éviter de s'en tenir rigidement à des pratiques qui peuvent freiner le progrès.

Conclusion

- Le TDD représente une approche disciplinée vitale pour les développeurs modernes, soulignant l'importance des tests, de la conception et de la qualité du code.

Chapitre 27 Résumé : 6 Pratiquer

PRATIQUER

Introduction

Tous les professionnels pratiquent leur art, et ce chapitre se concentre sur la manière dont les programmeurs peuvent améliorer leurs compétences grâce à la pratique.

Quelques éléments de contexte sur la pratique

Bien que la pratique du codage ne soit pas un nouveau concept, elle s'est formalisée autour du tournant du millénaire. Un programme simple comme "Hello, World" sert de rite de passage pour de nombreux programmeurs. Au fil des décennies, l'acte de programmer a évolué, passant de longues attente pour les compilations à des cycles rapides de pratiques de développement modernes comme le Test-Driven Development (TDD).

Délai de réponse

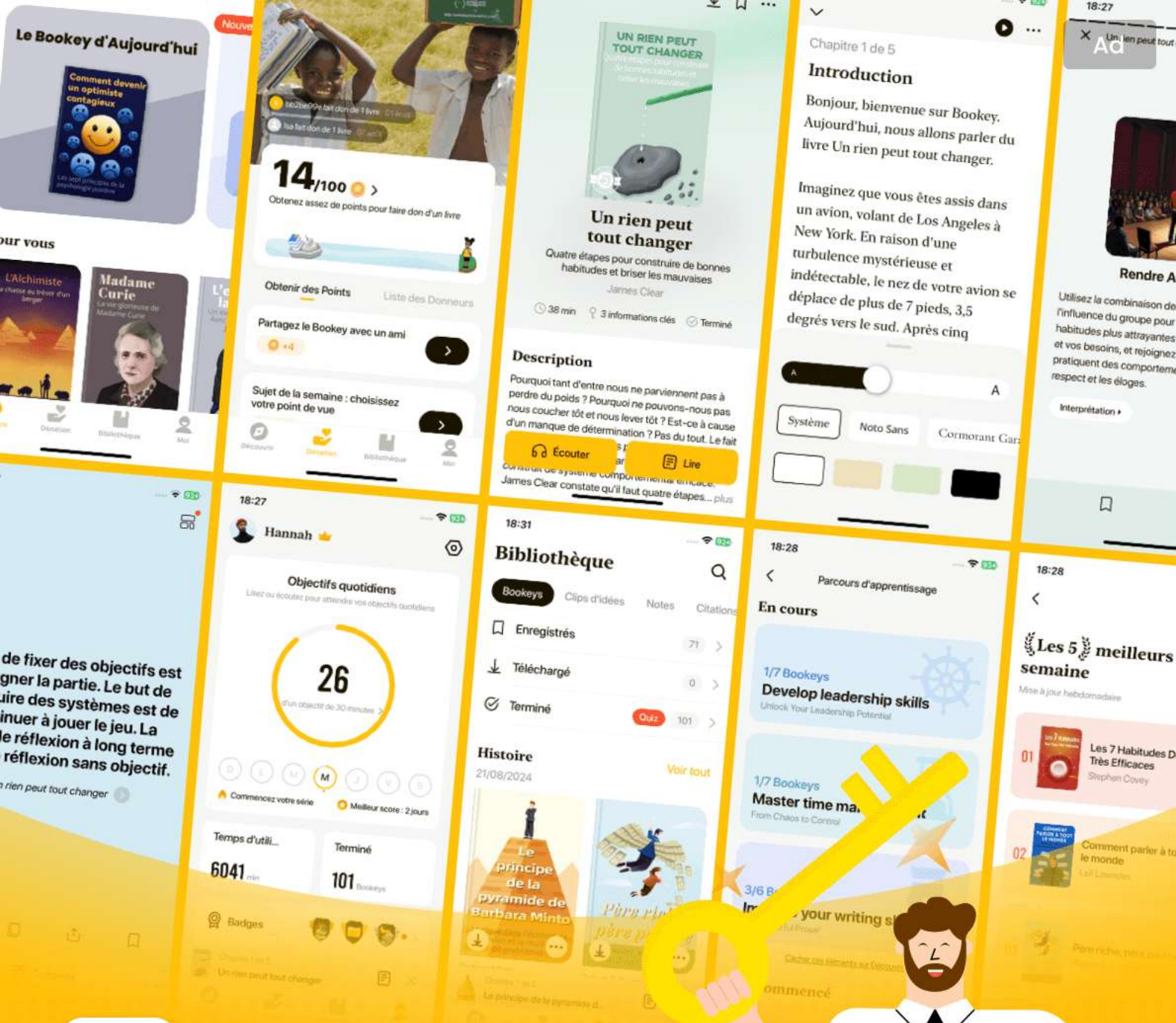
La puissance de calcul disponible aujourd'hui permet aux programmeurs de travailler beaucoup plus efficacement qu'auparavant. Les environnements modernes permettent aux programmeurs de terminer les cycles de compilation et de test en quelques secondes, offrant ainsi une boucle de rétroaction rapide. Cette vitesse incite à prendre des décisions rapides, semblable aux temps de réaction dans les arts martiaux.

Le Dojo de Codage

Le concept du Dojo de Codage a émergé comme un espace où les programmeurs peuvent pratiquer des techniques de codage, souvent dans un cadre communautaire. Les éléments clés incluent :

-

Kata



Les meilleures idées du monde débloquent votre potentiel

Essai gratuit avec Bookey

Télécharger dans App Store

DISPONIBLE SUR Google Play

Scanner pour télécharger

Chapitre 28 Résumé : 7 Tests d'acceptation

TESTS D'ACCEPTATION

Le rôle des développeurs professionnels

- Les développeurs professionnels agissent en tant que communicateurs et bâtisseurs, mettant l'accent sur l'exactitude de la communication avec les membres de l'équipe et les parties prenantes.

Communiquer les exigences

- Les malentendus entre les experts métiers et les programmeurs sont fréquents ; les véritables exigences sont souvent mal comprises.
- Une rencontre avec Tom, un non-programmeur, a illustré les complexités de la transformation d'idées de base en applications réelles, menant à des aperçus sur les besoins des clients.

Précision prématurée

- Les entreprises et les programmeurs rencontrent des défis pour rechercher des exigences et des estimations précises avant le début d'un projet, ce qui entraîne souvent un gaspillage de ressources.
- Le "Principe d'incertitude" indique que les systèmes en direct fournissent des aperçus que les exigences statiques ne peuvent pas capturer, modifiant ainsi la perspective des parties prenantes.

Ambiguïté tardive

- Retarder la précision entraîne des ambiguïtés dues à des désaccords ou des suppositions de compréhension par les parties prenantes, compliquant davantage la communication.

Tests d'acceptation

- Les tests d'acceptation clarifient la définition de "terminé", s’assurant que toutes les parties s’accordent sur les critères d'achèvement du projet.
- Ces tests, développés de manière collaborative, établissent

des attentes claires quant à la satisfaction des exigences.

Communiquer la clarté et l'automatisation

- Les tests d'acceptation sont essentiels pour clarifier les exigences, garantissant que toutes les parties soient alignées sur les fonctionnalités.
- L'automatisation des tests d'acceptation est cruciale pour réduire les coûts associés aux protocoles de test manuels, menant à une meilleure fiabilité et efficacité.

Travail et résistance à l'écriture de tests d'acceptation

- Rédiger des tests d'acceptation détaillés n'est pas un travail supplémentaire ; c'est une partie intégrante du processus de spécification, garantissant que le bon système est livré et que les décisions sur ce que signifie "terminé" sont claires.

Qui écrit les tests d'acceptation ?

- Idéalement, les parties prenantes, les QA ou les analystes métier devraient collaborer sur les tests, avec les développeurs impliqués dans la révision et la connexion des

tests à leurs mises en œuvre.

Rôle du développeur

- Les développeurs doivent implémenter des fonctionnalités seulement après que les tests d'acceptation soient en place, négociant et affinant les tests s'ils sont flous.

Négociation des tests

- Les tests d'acceptation peuvent nécessiter un affinement pour améliorer la compréhension et la clarté, mettant l'accent sur l'aspect collaboratif du développement.

Distinction entre les tests d'acceptation et les tests unitaires

- Les tests d'acceptation se concentrent sur les critères commerciaux du point de vue des parties prenantes, tandis que les tests unitaires traitent du comportement interne du code, ce qui en fait des formes de documentation distinctes.

Défis liés aux interfaces graphiques (GUI)

- Les GUI sont subjectives et volatiles, compliquant l'écriture des tests d'acceptation. Les tests devraient interagir avec les fonctionnalités de la GUI à un niveau supérieur via des API définies.

Intégration continue

- Il est essentiel d'exécuter les tests d'acceptation et les tests unitaires en continu pour détecter les problèmes tôt et maintenir la fiabilité du logiciel.

Conclusion

- La communication sur les détails du projet est intrinsèquement difficile. Les tests d'acceptation automatisés servent de mécanisme formel pour garantir des exigences claires, minimisant les malentendus entre les développeurs et les parties prenantes.

Chapitre 29 Résumé : 8 Stratégies de test

STRATÉGIES DE TEST

Importance de la stratégie de test

Les développeurs professionnels comprennent que le test va au-delà de l'écriture de tests unitaires et d'acceptation. Une stratégie de test complète est essentielle pour chaque équipe de développement.

Chasses aux bugs collaboratives

Lors d'une expérience passée chez Rational, une journée collaborative de "chasse aux bugs" a rassemblé tous les membres de l'équipe pour identifier les bogues, favorisant l'engagement et le sentiment de responsabilité en matière de qualité au sein de l'équipe.

Objectif de l'assurance qualité (AQ)

L'objectif principal de l'équipe de développement devrait être

que l'AQ ne trouve rien à redire. Tous les problèmes découverts par l'AQ doivent inciter à une enquête approfondie et à des actions correctives de la part de l'équipe de développement.

L'AQ comme membre collaboratif de l'équipe

L'AQ devrait travailler aux côtés du développement, agissant comme spécificateurs et caractérisateurs. Cela implique de créer des tests d'acceptation automatisés en collaboration avec le business et de réaliser des tests exploratoires pour révéler les comportements réels du système.

Pyramide d'automatisation des tests

Une stratégie de test structurée est illustrée dans la Pyramide d'automatisation des tests, qui montre la variété et la hiérarchie des tests nécessaires :

1.

Tests unitaires

- Créés par les développeurs pour la spécification de bas niveau.
- Visant une couverture élevée (idéalement dans les 90 %).

- Exécutés dans le cadre de l'intégration continue.

2.

Tests de composants

- Tests d'acceptation axés sur des composants individuels du système.
- Écrits de manière collaborative par l'AQ, le business et le développement.
- Ciblent des scénarios typiques, en mettant l'accent sur les cas de réussite.

3.

Tests d'intégration

- Évaluent l'interaction entre plusieurs composants.
- Rédigés par des architectes et axés sur l'intégrité architecturale.
- Pas intégrés à l'intégration continue en raison de temps d'exécution plus longs.

4.

Tests système

- Tests automatisés pour l'ensemble du système intégré.
- Garantissent un câblage correct et l'interopérabilité des composants du système.

- Exécutés rarement mais essentiels pour la vérification du système.

5.

Tests exploratoires manuels

- Réalisés par des humains pour découvrir des comportements inattendus.
- Créatifs par nature, non scénarisés et ne pouvant pas être entièrement planifiés.
- Se concentrent sur l'expérience utilisateur globale et les particularités du comportement.

Conclusion

Bien que le développement dirigé par les tests (TDD) et les tests d'acceptation soient des composants clés, une stratégie de test holistique doit incorporer divers types de tests. L'exécution fréquente de cette hiérarchie de tests aide à assurer la propreté et la qualité continues du système.

Chapitre 30 Résumé : 9 Gestion du Temps

GESTION DU TEMPS

Utilisation Efficace du Temps

- Huit heures correspondent à 480 minutes, donc maximiser chaque seconde est crucial pour les professionnels.
- Une expérience personnelle en gestion du temps de 1986 impliquait de se réveiller à 5 heures du matin, de faire du vélo jusqu'au bureau, d'utiliser un emploi du temps détaillé et d'allouer du temps pour les interruptions.

Réunions et Leur Coût

- Les réunions peuvent coûter environ 200 $ par heure par participant et gaspillent souvent du temps.
- Deux vérités : les réunions sont nécessaires mais peuvent être de gros gaspilleurs de temps.

Refuser des Invitations à des Réunions

- Soyez sélectif quant à votre participation aux réunions ; n'accepté que si c'est nécessaire pour les tâches immédiates.
- Consultez la direction sur votre participation aux réunions demandées par des figures d'autorité.

Sortir des Réunions Inefficaces

- Si une réunion devient improductive, partez poliment ou demandez à accélérer les discussions.
- Rester dans des réunions improductives est peu professionnel.

Structure des Réunions

- Les réunions doivent avoir un ordre du jour clair et un objectif défini.

Chapitre 31 Résumé : 10 Estimation

ESTIMATION

L'estimation est une tâche critique mais redoutable pour les professionnels du logiciel, influençant la valeur commerciale, les réputations et les relations entre développeurs et parties prenantes du secteur.

UNE EXPÉRIENCE ANCESTRALE

En 1978, l’auteur a rencontré des défis avec un projet logiciel sur des systèmes embarqués fragiles. La solution consistait à découpler les composants logiciels pour permettre des mises à jour indépendantes, ce qui rendait le débogage et le déploiement plus faciles.

PERSPECTIVES SUR L'ESTIMATION

Les parties prenantes commerciales considèrent les estimations comme des engagements, tandis que les développeurs les voient comme des suppositions éclairées. Cette différence peut conduire à des malentendus et à des

relations tendues.

ENGAGEMENT VS. ESTIMATION

Un engagement est une livraison garantie à une date spécifique, nécessitant certitude et responsabilité. En revanche, une estimation n'est qu'une supposition éclairée, sans obligation, permettant ambiguïté et variations.

DISTRIBUTION DE PROBABILITÉ DANS LES ESTIMATIONS

Une estimation efficace nécessite de comprendre que les estimations représentent une gamme de possibilités plutôt qu'une date fixe. Communiquer la probabilité est essentiel pour des attentes plus claires.

LA LOI DE MURPHY ET LES ENGAGEMENTS IMPLICITES

Reconnaître les incertitudes peut conduire à des engagements implicites lorsque les parties prenantes recherchent des dates de réalisation spécifiques, ce qui peut déformer la confiance réelle du développeur.

PERT (TECHNIQUE D'ÉVALUATION ET D'EXAMEN DE PROGRAMME)

Introduit en 1957, PERT aide à la gestion de projet en incorporant trois types d'estimations (Optimiste, Nominative, Pessimiste) pour calculer une durée attendue et un écart type.

AGRÉGER LES ESTIMATIONS DE TÂCHES

Lors de la gestion de plusieurs tâches, appliquer PERT permet de comprendre les délais globaux du projet et de gérer les risques efficacement.

ESTIMATION DES TÂCHES

Impliquer l'avis de l'équipe améliore la précision des estimations. Des techniques comme le Wideband Delphi peuvent générer un consensus lors des efforts d'estimation.

WIDEBAND DELPHI

Le Wideband Delphi de Barry Boehm implique des discussions en équipe et des estimations successives jusqu'à

ce qu'un accord soit atteint. Les méthodes « Doigts Volants » et « Poker Planning » sont des variations informelles soulignant la collaboration.

LA LOI DES GRANDS NOMBRES

Décomposer de grandes tâches en plus petites et les estimer indépendamment peut donner des estimations totales plus précises, atténuant les erreurs d'estimation.

CONCLUSION

Les développeurs professionnels se concentrent sur la fourniture d'estimations pratiques sans faire d'engagements injustifiés. Ils travaillent en collaboration pour garantir un consensus, communiquant les distributions de probabilité pour mieux guider la planification. Les techniques décrites sont des cadres qui se sont révélés efficaces mais ne sont ni exhaustives ni définitives.

Exemple

Point clé:Comprendre la différence entre les engagements et les estimations est essentiel pour une gestion efficace des projets logiciels.

Exemple:Imaginez que vous dirigez une équipe de développement et qu'un intervenant vous demande quand une nouvelle fonctionnalité sera prête. Si vous vous engagez avec assurance à une date précise, vous risquez de créer sans le vouloir une attente irréaliste fondée sur l'incertitude. Au lieu de cela, présentez-le comme une estimation en expliquant les complexités impliquées, en décrivant les différents scénarios avec des résultats optimistes, normaux et pessimistes, et en communiquant clairement que votre estimation est flexible. Ce changement réduit non seulement la pression, mais favorise également la confiance, car les intervenants apprécient la transparence, ce qui conduit à une meilleure collaboration et à un avancement plus fluide du projet.

Chapitre 32 Résumé : 11 La Pression

Résumé du Chapitre 32 : LA PRESSION

L'IMPORTANCE DU CALME SOUS PRESSION

Lorsqu'ils sont confrontés à des situations de forte pression, il est primordial pour les professionnels de garder leur calme et de respecter les pratiques établies. Le comportement d'un développeur sous stress peut avoir un impact significatif sur l'environnement de travail, tout comme un chirurgien lors d'une opération critique.

EXPÉRIENCE PERSONNELLE ET RÉFLEXION

L'auteur raconte son expérience chez Clear Communications, une start-up en difficulté, où le stress chaotique a entraîné des pratiques de travail nuisibles et une réflexion personnelle. Un moment de prise de conscience l'a poussé à changer son approche, en priorisant le professionnalisme au-dessus du chaos induit par la pression.

ÉVITER LA PRESSION

Une des stratégies clés pour rester calme pendant les périodes stressantes est d'éviter les situations qui mènent à la pression. Cela inclut de faire attention aux engagements et de s'assurer que les délais sont réalistes. Les professionnels doivent évaluer les risques associés aux engagements et les communiquer efficacement.

RESTER PROPRE

Maintenir la clarté et la propreté dans les pratiques de codage aide à atténuer la pression. Les professionnels reconnaissent que bâcler du travail entraîne des complications, qui ralentissent finalement le progrès.

DISSIPLINE EN PÉRIODE DE CRISE

Le véritable test de la discipline survient lors de situations de crise. De véritables professionnels respectent leurs disciplines de travail même sous stress. Si les pratiques sont efficaces, elles doivent être maintenues quelles que soient les circonstances.

GESTION DE LA PRESSION

Lorsque la pression devient inévitable, gérer le stress efficacement est crucial. Cela implique de ralentir, de développer un plan clair et de communiquer proactivement avec l'équipe. Éviter les surprises est essentiel pour minimiser le stress supplémentaire.

COMPTER SUR LES DISCIPLINES ET DEMANDER DE L'AIDE

Dans les situations difficiles, s'appuyer sur les pratiques établies devient encore plus critique. Travailler avec un partenaire grâce à la programmation en binôme peut aider à alléger une partie du stress tout en maintenant la concentration et la discipline.

CONCLUSION

L'essence de la gestion de la pression repose à la fois sur des techniques d'évitement et des stratégies d'adaptation. Les professionnels doivent s'efforcer de minimiser la pression en respectant leurs engagements, en restant organisés et en maintenant des lignes de communication claires lorsque des pressions surviennent.

Chapitre 33 Résumé : 12 Collaboration

CODER PROPREMENT : RÉSUMÉ DU CHAPITRE 33

Collaboration dans le développement logiciel

La plupart des logiciels sont créés par des équipes, et une collaboration efficace est essentielle pour le succès de l'équipe. Les programmeurs professionnels doivent interagir avec leurs coéquipiers, évitant ainsi l'isolement.

Expérience personnelle et apprentissage

L'auteur partage une histoire personnelle de ses débuts de carrière, illustrant les défis et les succès de la collaboration avec un collègue, Tim. Ils ont travaillé à l'optimisation d'un générateur de références croisées, apprenant par essais et erreurs. Leur collaboration a mis en lumière les complexités de l'optimisation logicielle.

La nature du programmeur

Les programmeurs préfèrent souvent travailler indépendamment, trouvant les relations interpersonnelles difficiles. Alors que certains s'épanouissent grâce à la collaboration, beaucoup aiment se concentrer intensément sur des problèmes techniques.

Compréhension des objectifs commerciaux

Les professionnels doivent aligner leur travail avec les objectifs de l'entreprise. Une communication efficace avec les managers et les pairs aide les programmeurs à comprendre le contexte plus large de leurs projets. Ignorer les besoins commerciaux peut mener à des licenciements, comme le montre l'histoire personnelle de l'auteur, qui a été renvoyé pour ne pas avoir prêté attention aux priorités commerciales.

Scanner pour télécharger

Pourquoi Bookey est une application incontournable pour les amateurs de livres

Contenu de 30min
Plus notre interprétation est profonde et claire, mieux vous saisissez chaque titre.

Format texte et audio
Absorbez des connaissances même dans un temps fragmenté.

Quiz
Vérifiez si vous avez maîtrisé ce que vous venez d'apprendre.

Et plus
Plusieurs voix & polices, Carte mentale, Citations, Clips d'idées...

Essai gratuit avec Bookey

Chapitre 34 Résumé : 13 Équipes et Projets

ÉQUIPES ET PROJETS

Défis d'allocation des projets

Lors de la gestion de plusieurs petits projets, l'allocation des ressources peut devenir complexe. Souvent, les équipes se composent d'individus partageant leur temps entre divers projets, ce qui entraîne des inefficiences et un manque de cohésion. Le concept de "demi-personne" nuit à l'efficacité des équipes, entraînant plus de confusion qu'un véritable sens de la collaboration.

L'équipe soudée

Une équipe bien fonctionnelle se caractérise par de fortes relations et une collaboration efficace entre ses membres. Une équipe soudée se compose généralement d'un mélange équilibré de programmeurs, de testeurs, d'analystes et d'un

chef de projet, avec une taille optimale d'environ douze membres. Cette structure d'équipe favorise un environnement où les membres complètent les compétences des autres, facilitant le soutien mutuel et des performances élevées.

Structuration des projets

Contrairement à de nombreuses banques et compagnies d'assurance qui créent des équipes pour des projets spécifiques, les organisations qui réussissent construisent des équipes autour de la structure soudée existante. Cela permet aux équipes de gérer plusieurs projets simultanément et d'adapter leur charge de travail en fonction de leurs forces uniques.

Gestion de la vélocité de l'équipe

Les équipes fonctionnent avec une vélocité spécifique, indiquant la quantité de travail qu'elles peuvent accomplir dans un délai donné. En suivant cette mesure, la direction peut prendre des décisions éclairées sur l'allocation des projets et les priorités. La flexibilité des équipes soudées permet des ajustements rapides du focus des projets lorsque nécessaire, contrastant avec des équipes moins capables de

telles réallocations rapides.

Préoccupations des propriétaires de projets

Bien que les propriétaires de projets puissent ressentir une perte de contrôle lorsque les ressources se déplacent fréquemment entre les projets, ce modèle assure une plus grande réactivité aux besoins de l'entreprise. Il permet des ajustements rapides de priorisation basés sur les objectifs organisationnels sans les contraintes de la formation et de la dissolution d'équipes.

Conclusion

Le processus de construction d'une équipe cohésive est plus complexe et précieux que la simple gestion des projets. Maintenir des équipes stables qui poursuivent leur travail d'un projet à l'autre améliore leur capacité à délivrer des résultats efficacement à travers de multiples initiatives.

Chapitre 35 Résumé : 14 Mentorat, Apprentissages, et Artisanat

MENTORAT, APPRENTISSAGE ET ARTISANAT

Introduction à la Déception dans l'Éducation en Informatique

- Robert C. Martin exprime sa déception face à la préparation des diplômés en informatique pour des rôles de programmation.
- De nombreux diplômés manquent d'expérience pratique en code, malgré leurs connaissances théoriques.

Expériences de Mentorat

- Martin partage des anecdotes personnelles d'apprentissage à travers un mentorat structuré et non structuré.
- Parmi ses premières expériences, il a travaillé avec un Digi-Comp I et a appris l'algèbre booléenne à travers un manuel.

- Ses expériences au lycée avec l'ordinateur ECP-18 impliquaient l'observation des techniques de programmation.
- Le mentorat est souligné comme un élément crucial dans le développement des compétences en programmation.

Besoin de Mentorat Structuré dans le Développement Logiciel

- En contraste avec la profession médicale, qui exige un mentorat rigoureux et une pratique supervisée.
- Dans le secteur logiciel, les jeunes diplômés sont souvent plongés dans des rôles critiques sans formation de base suffisante.
- Martin plaide pour un modèle d'apprentissage structuré afin d'élever les compétences et les aptitudes des développeurs de logiciels.

Modèle Proposé d'Apprentissage en Logiciel

-
Maîtres
: Programmeurs expérimentés menant des projets techniques et guidant des développeurs moins expérimentés.
-

Compagnons
: Programmeurs compétents acquérant de l'expérience et apprenant le travail d'équipe sous supervision.
-
Apprentis/Stagiaires
: Nouveaux diplômés étroitement mentorés, assistant principalement les compagnons, avec un accent sur l'apprentissage des principes et pratiques fondamentaux.

Importance de l'Enseignement Technique et des Valeurs

- Insistance sur la nécessité que les aînés du domaine transmettent les valeurs artisanales et les compétences techniques.
- Critique du manque de véritable mentorat technique dans de nombreuses organisations aujourd'hui.

Définition de l'Artisanat dans le Logiciel

- L'artisanat est défini comme l'état d'esprit incarnant la compétence, la qualité et le professionnalisme.
- Il s'acquiert par observation et interaction dans un cadre de mentorat.

Convaincre les Autres d'Adopter l'Artisanat

- Encouragement à modéliser un comportement artisan dans le but de promouvoir ses valeurs au travail.

Conclusion

- La responsabilité de développer des professionnels du logiciel compétents incombe à l'industrie, et non uniquement aux établissements d'enseignement.
- Appel à l'adoption de programmes de mentorat et d'apprentissage structurés dans le développement logiciel.

Chapitre 36 Résumé : A : Outils

OUTILS

Contexte Historique

En 1978, travaillant chez Teradyne, l'auteur décrit l'environnement difficile de la gestion de 80KSLOC de code assembleur M365 stocké sur bande. Le processus était laborieux : les bandes ne se déplaçaient que dans une seule direction, et les erreurs de lecture ou d'écriture entraînaient souvent la répétition d'opérations longues, mettant en évidence l'état primitif de la gestion des outils logiciels à l'époque.

Contrôle de Version

Ce chapitre aborde l'évolution du contrôle de version, en mettant l'accent sur l'utilisation d'outils open-source qui répondent aux besoins des développeurs en raison de leur rapidité et de leur efficacité. La limitation des systèmes de contrôle de version « d'entreprise » est remarquée, suggérant

une approche hybride pour que les développeurs maintiennent leur productivité sans provoquer de réactions négatives de la part de l'entreprise.

Verrouillage Pessimiste vs. Optimiste

Le verrouillage pessimiste restreint l'édition simultanée mais entraîne des inefficacités, car il peut empêcher d'autres de faire des changements nécessaires. Les outils modernes permettent une gestion plus flexible des mises à jour concurrentes grâce à des méthodes de fusion, réduisant ainsi le besoin de vérifications de fichiers individuelles.

Systèmes de Contrôle de Version Modernes

Le texte contraste les outils plus anciens comme CVS et SVN avec les nouveaux systèmes distribués comme git. Git permet la création et la fusion de branches spontanées.

Scanner pour télécharger

App Store
Coup de cœur

22k avis 5 étoiles

Retour Positif

Fabienne Moreau

ue résumé de livre ne testent
ion, mais rendent également
musant et engageant.
té la lecture pour moi.

Fantastique! Giselle Dubois

Je suis émerveillé par la variété de livres et de langues que Bookey supporte. Ce n'est pas juste une application, c'est une porte d'accès au savoir mondial. De plus, gagner des points pour la charité est un grand plus !

Fi
Le
liv
co
pr

é Blanchet

de lecture
eption de
es,
tous.

J'adore ! Isoline Mercier

Bookey m'offre le temps de parcourir les parties importantes d'un livre. Cela me donne aussi une idée suffisante pour savoir si je devrais acheter ou non la version complète du livre ! C'est facile à utiliser !"

Gain de temps !

Bookey est mon applicat
intellectuelle. Les résum
magnifiquement organis
monde de connaissance

Appli géniale ! Joachim Lefevre

adore les livres audio mais je n'ai pas toujours le temps
'écouter le livre entier ! Bookey me permet d'obtenir
n résumé des points forts du livre qui m'intéresse !!!
Quel super concept !!! Hautement recommandé !

Appli magnifique

Cette application est une bouée de sauve
amateurs de livres avec des emplois du te
Les résumés sont précis, et les cartes me
renforcer ce que j'ai appris. Hautement re

Essai gratuit avec Bookey

Chapitre 37 Résumé : Index

Résumé du Chapitre 37 de "CODER PROPREMENT" par Robert C. Martin

Tests d'Acceptation

- Définition : Les tests d'acceptation sont essentiels dans le développement logiciel, servant à valider les exigences et à garantir que le produit répond aux objectifs commerciaux.
- Rôle des Développeurs : Les développeurs jouent un rôle clé dans l'écriture et l'exécution des tests d'acceptation, nécessitant collaboration et communication.
- Automatisation : Les tests d'acceptation automatisés simplifient le processus de développement et s'intègrent bien aux pratiques d'intégration continue.

Rôles Adverses

- La dynamique des rôles adverses au sein des équipes peut aboutir à des conflits et freiner les progrès, soulignant le besoin d'une communication et d'une collaboration efficaces.

Engagement

- Comprendre l'engagement dans le développement logiciel implique de reconnaître son importance pour la cohésion de l'équipe et le succès du projet, tout en gérant les attentes et la discipline.

Communication

- Une communication claire est essentielle, surtout en ce qui concerne les exigences et les changements dans les projets. Les interprétations erronées peuvent entraîner de l'ambiguïté et des échecs de projet.

Discipline en Crise

- En période de crise, l'importance de maintenir la discipline ne peut être sous-estimée, car des décisions précipitées peuvent introduire des défauts et du chaos.

Meilleures Pratiques de Développement

- L'adoption de pratiques comme le développement en

binôme et le mentorat favorise le partage des connaissances et le développement des compétences, essentiels pour améliorer la dynamique de l'équipe.

Estimation

- Les tâches et les estimations de projet doivent prendre en compte l'incertitude et utiliser des techniques telles que le Planning Poker pour atténuer l'anxiété liée à l'estimation.

Assurance Qualité

- L'assurance qualité automatisée contraste avec les méthodes traditionnelles en identifiant activement les défauts tout au long du cycle de développement, contribuant à une meilleure qualité de code.

Gestion du Temps

- Une gestion efficace du temps nécessite de prendre conscience des priorités, des tâches en cours et d'éviter des pièges comme la précipitation ou la complaisance.

Stratégies de Test

- Le chapitre décrit diverses stratégies de test, mettant l'accent sur une approche équilibrée qui intègre des tests unitaires, des tests d'intégration et des tests d'acceptation, tout en garantissant une couverture complète.

Éthique de Travail

- Les développeurs doivent cultiver une forte éthique de travail, ancrée dans un apprentissage continu et un engagement à améliorer leur métier et la qualité globale de la livraison de produits.
En adhérant à ces principes, les équipes peuvent améliorer leur productivité, minimiser les risques et favoriser un environnement de travail sain qui priorise la qualité et la collaboration.

Exemple

Point clé:Les tests d'acceptation automatisés sont cruciaux pour garantir la qualité du produit et s'aligner sur les objectifs commerciaux.

Exemple:Imaginez que vous faites partie d'une équipe de développement travaillant sur une nouvelle fonctionnalité pour une application mobile. Alors que votre chef d'équipe vous attribue la tâche, vous réalisez que les exigences ne sont pas tout à fait claires. Au lieu d'attendre que des malentendus surgissent plus tard, vous prenez l'initiative d'écrire des tests d'acceptation. Ces tests définissent ce que la fonctionnalité doit faire, servant de contrat entre les parties prenantes et votre équipe. Vous collaborez étroitement avec un ingénieur QA, discutant des cas limites potentiels et vous assurant que les tests les couvrent. Au fur et à mesure que vous automatisez ces tests, votre équipe peut les intégrer en toute confiance dans votre pipeline d'intégration continue, permettant à chaque commit d'être validé par rapport à ces critères. Cette approche proactive non seulement fait gagner du temps à long terme, mais favorise également une culture où la qualité est la responsabilité de tous, alignant le produit final plus

étroitement sur les objectifs commerciaux.

Lire, Partager, Autonomiser

Terminez votre défi de lecture, faites don de livres aux enfants africains.

Le Concept

BOOKS FOR AFRICA × ×

Cette activité de don de livres se déroule en partenariat avec Books For Africa. Nous lançons ce projet car nous partageons la même conviction que BFA : Pour de nombreux enfants en Afrique, le don de livres est véritablement un don d'espoir.

La Règle

Gagnez 100 points → **Échangez un livre** → **Faites un don à l'Afrique**

Votre apprentissage ne vous apporte pas seulement des connaissances mais vous permet également de gagner des points pour des causes caritatives ! Pour chaque 100 points gagnés, un livre sera donné à l'Afrique.

Meilleures phrases du CODER PROPREMENT par Robert C. Martin avec numéros de page

Chapitre 1 | Phrases des pages -63

1. Prendre le temps de choisir de bons noms permet d'économiser encore plus de temps par la suite. Prenez donc soin de vos noms et changez-les lorsque vous en trouvez de meilleurs.
2. Si un nom nécessite un commentaire, alors le nom ne révèle pas son intention.
3. Le pouvoir de choisir de bons noms ne saurait être sous-estimé.
4. Les programmeurs doivent éviter de laisser de faux indices qui obscurcissent le sens du code.
5. Les professionnels utilisent leurs compétences pour le bien et écrivent du code que les autres peuvent comprendre.
6. Ne soyez pas trop astucieux avec les noms ; privilégiez la clarté plutôt que la valeur divertissante.

Chapitre 2 | Phrases des pages 64-85

1. Les fonctions ne devraient faire qu'une seule chose. Elles devraient bien le faire. Elles devraient le faire uniquement.
2. Petites ! La première règle des fonctions est qu'elles doivent être petites. La deuxième règle des fonctions est qu'elles doivent être plus petites que ça.
3. Le code devrait se lire comme un récit descendant. Nous voulons que chaque fonction soit suivie de celles du niveau d'abstraction suivant afin que nous puissions lire le programme, descendant un niveau d'abstraction à la fois en descendant la liste des fonctions.
4. Utilisez des noms descriptifs. Il est difficile de surestimer la valeur de bons noms.
5. Préférez les exceptions au retour de codes d'erreur.
6. Ne vous répétez pas (DRY).

Chapitre 3 | Phrases des pages 86-107

1. Ne commentez pas un mauvais code, réécrivez-le.” —Brian W. Kernighan et P. J. Plaugher

2.Un code clair et expressif avec peu de commentaires est de loin supérieur à un code encombré et complexe avec de nombreux commentaires.

3.L'utilisation appropriée des commentaires est de compenser notre incapacité à nous exprimer correctement dans le code.

4.Des commentaires incorrects sont bien pires que pas de commentaires du tout.

5.Chaque fois que vous vous exprimez dans le code, vous devriez vous féliciter. Chaque fois que vous écrivez un commentaire, vous devriez grimacer et ressentir l'échec de votre capacité d'expression.

6.La vérité ne peut être trouvée qu'à un seul endroit : le code. Seul le code peut vraiment vous dire ce qu'il fait.

7.De bons commentaires sont informatifs et nécessaires, mais ils restent un compromis face à l'incapacité du code à s'exprimer adéquatement.

Scannez pour télécharger

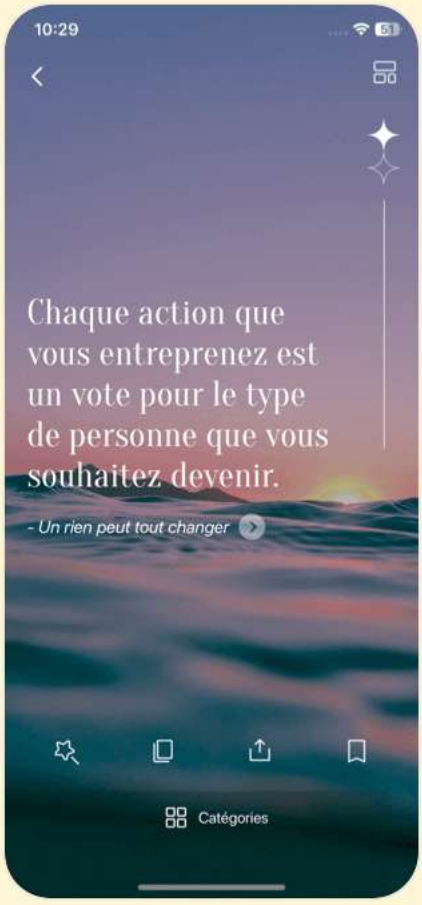
Téléchargez l'appli Bookey pour profiter

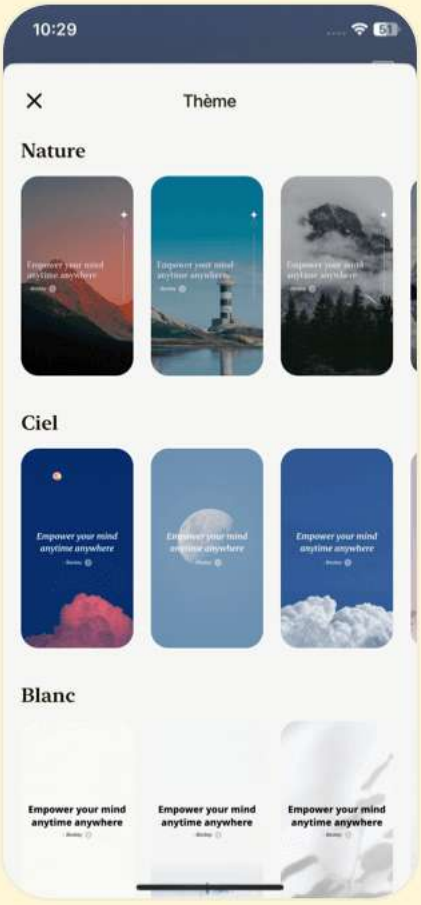
Plus d'un million de citations
Plus de 1000 résumés de livres

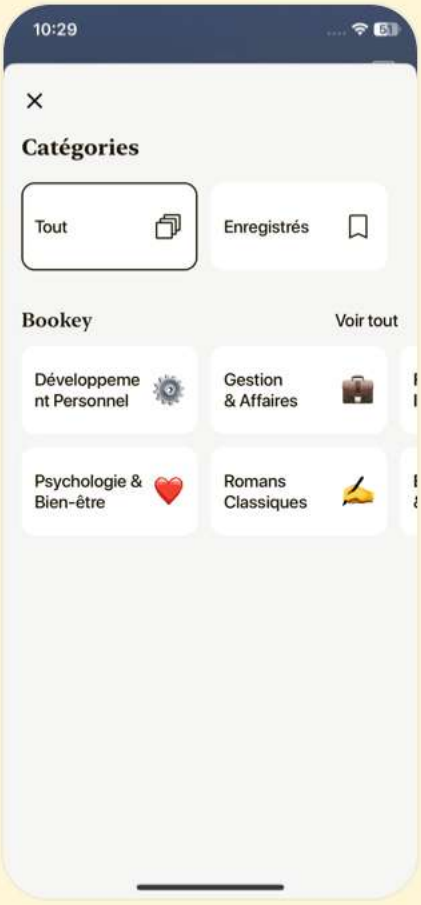
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 4 | Phrases des pages 108-125

1. Si au lieu de cela ils voient une masse de code en désordre qui semble avoir été écrite par une bande de marins ivres, ils sont alors susceptibles de conclure que la même inattention aux détails imprègne chaque autre aspect du projet.
2. Le formatage du code est important. Il est trop important à ignorer et trop important à traiter avec religiosité.
3. La fonctionnalité que vous créez aujourd'hui a de bonnes chances de changer dans la prochaine version, mais la lisibilité de votre code aura un effet profond sur tous les changements qui seront jamais apportés.
4. Les petits fichiers sont généralement plus faciles à comprendre que les grands fichiers.
5. Vous devriez vous assurer que votre code est bien formaté.
6. La dernière chose que nous voulons faire est d'ajouter plus de complexité au code source en l'écrivant dans un méli-mélo de styles individuels différents.
7. Nous devons avoir un style cohérent et fluide. Le lecteur

doit être capable de faire confiance au fait que les gestes de formatage qu’il a vus dans un fichier source signifieront la même chose dans d'autres.

Chapitre 5 | Phrases des pages -135

1. Nous ne voulons pas que qui que ce soit d'autre dépende d'eux. Nous voulons garder la liberté de changer leur type ou leur implémentation selon l'humeur ou l'impulsion.
2. Cacher l'implémentation concerne les abstractions ! Une classe ne se contente pas de rendre ses variables accessibles par des getters et des setters. Elle expose plutôt des interfaces abstraites qui permettent à ses utilisateurs de manipuler l'essence des données, sans avoir à connaître son implémentation.
3. Les choses qui sont difficiles pour l'OO sont faciles pour les procédures, et les choses qui sont difficiles pour les procédures sont faciles pour l'OO !
4. La loi de Demeter dit qu'une méthode f d'une classe C ne doit appeler que les méthodes de : C, un objet créé par f, un

objet passé en tant qu'argument à f, un objet détenu dans une variable d'instance de C.

5. La pire option est d'ajouter aveuglément des getters et des setters. Une réflexion sérieuse doit être donnée à la meilleure manière de représenter les données qu'un objet contient.
6. Les bons développeurs de logiciels comprennent ces enjeux sans préjugés et choisissent l'approche qui convient le mieux à la tâche à accomplir.

Chapitre 6 | Phrases des pages 136-145

1. La gestion des erreurs est importante, mais si elle obscurcit la logique, c'est une erreur.
2. Lorsque vous exécutez du code dans la partie try d'une déclaration try-catch-finally, vous indiquez que l'exécution peut s'interrompre à tout moment et reprendre ensuite au niveau du catch.
3. Définissez les classes d'exception en fonction des besoins de l’appelant.
4. Ne renvoyez pas des valeurs nulles

5.La majeure partie de votre code commencera à ressembler à un algorithme propre et dépouillé.

Scannez pour télécharger

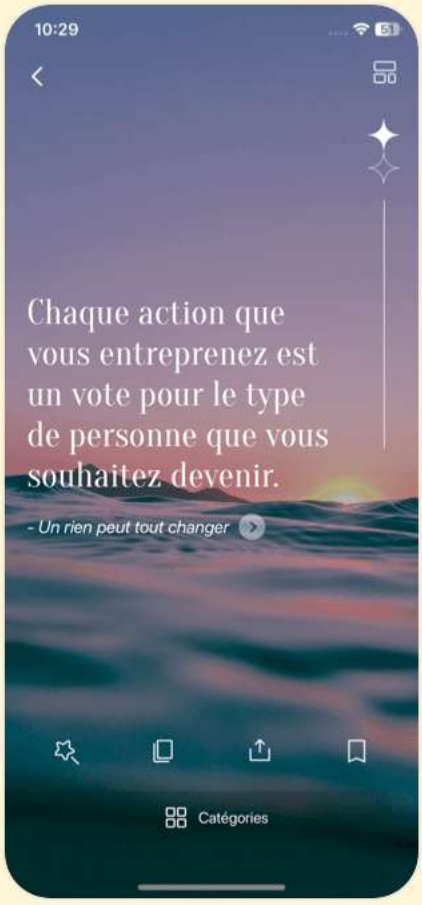
Téléchargez l'appli Bookey pour profiter

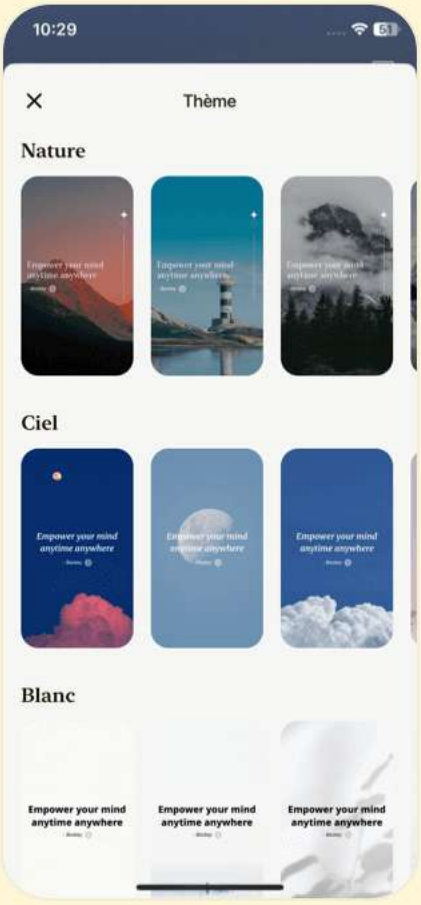
Plus d'un million de citations
Plus de 1000 résumés de livres

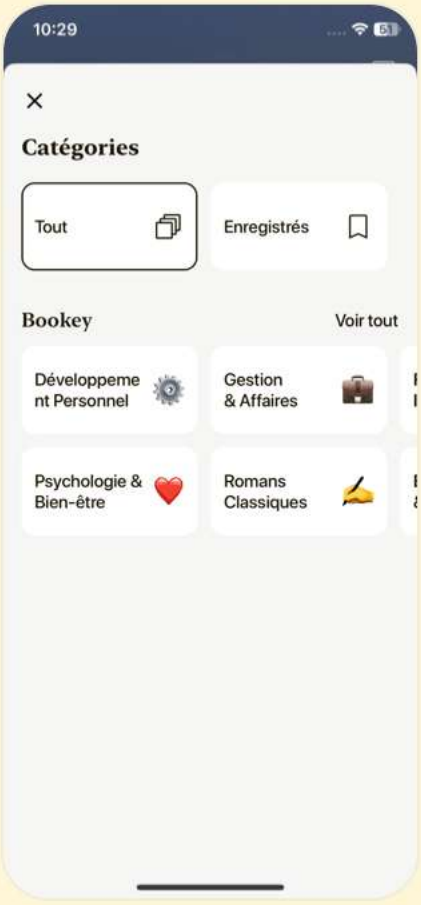
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 7 | Phrases des pages 146-153

1. L'interface à la frontière (Map) est cachée. Elle peut évoluer avec très peu d'impact sur le reste de l'application.
2. Les tests d'apprentissage ne coûtent finalement rien. Nous devions de toute façon apprendre l'API, et écrire ces tests était un moyen simple et isolé d'acquérir cette connaissance.
3. De bonnes conceptions logicielles accueillent le changement sans investissements ni retouches énormes.
4. Il vaut mieux dépendre de quelque chose que vous contrôlez plutôt que de quelque chose que vous ne contrôlez pas, de peur que cela ne finisse par vous contrôler.
5. Nous devrions éviter de laisser trop de notre code connaître les détails particuliers des tiers.

Chapitre 8 | Phrases des pages 154-167

1. Le code de test est aussi important que le code de production. Ce n'est pas un citoyen de deuxième

classe. Cela nécessite réflexion, conception et soin. Il doit être aussi propre que le code de production.

2. Si vous ne gardez pas vos tests propres, vous les perdrez. Et sans eux, vous perdez ce qui maintient votre code de production flexible. Oui, vous avez bien lu. Ce sont les tests unitaires qui maintiennent notre code flexible, maintenable et réutilisable.
3. La morale de cette histoire est simple : le code de test est aussi important que le code de production.
4. Oui, nous avons parcouru un long chemin ; mais il nous reste encore du chemin à faire.
5. Gardez vos tests constamment propres. Efforcez-vous de les rendre expressifs et succincts. Inventez des API de test qui agissent comme un langage spécifique au domaine qui vous aide à écrire les tests.

Chapitre 9 | Phrases des pages 168-185

1. Les classes doivent avoir une seule responsabilité - une seule raison de changer.
2. Nous voulons que nos systèmes soient composés de petites

classes, et non de quelques grandes.

3. Assouplir l'encapsulation est toujours un dernier recours.
4. Le nom d'une classe doit décrire quelles responsabilités elle remplit.
5. Si nous ne pouvons pas dériver un nom concis pour une classe, alors elle est probablement trop grande.
6. Faire fonctionner un logiciel et faire un logiciel propre sont deux activités très différentes.
7. Notre logique Sql restructurée représente le meilleur de tous les mondes.

Scannez pour télécharger

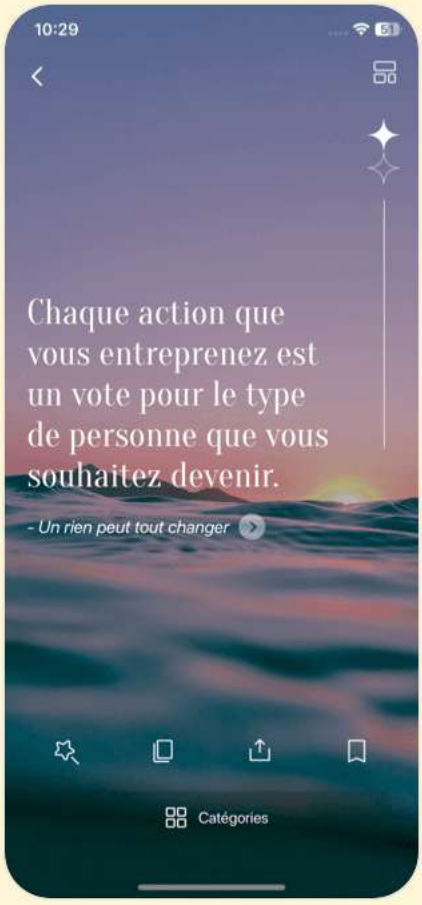
Téléchargez l'appli Bookey pour profiter

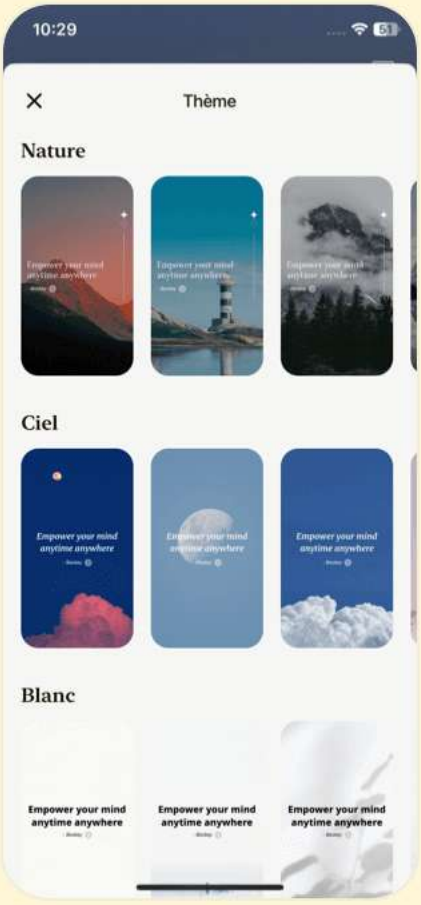
Plus d'un million de citations
Plus de 1000 résumés de livres

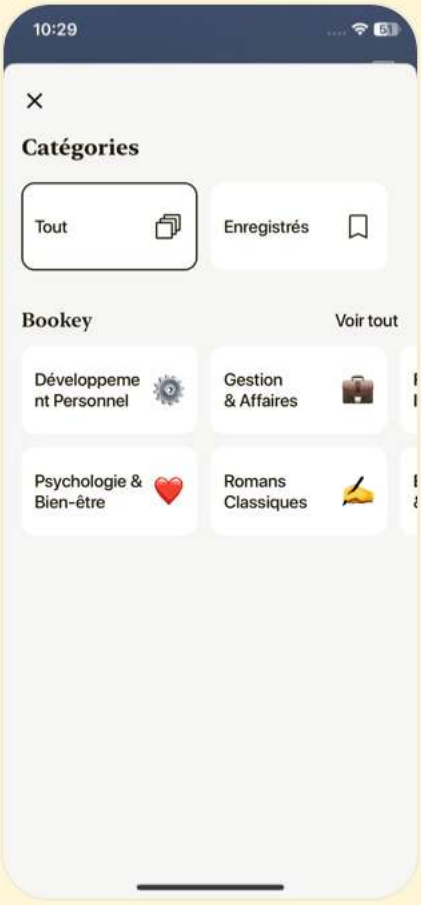
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 10 | Phrases des pages 186-203

1. La complexité tue. Elle épuise les développeurs, rendant la planification, la construction et les tests des produits difficiles." —Ray Ozzie, CTO, Microsoft Corporation
2. La séparation des préoccupations est l'une des techniques de conception les plus anciennes et les plus importantes dans notre métier.
3. C'est un mythe que nous puissions obtenir les systèmes 'justes du premier coup.' Au lieu de cela, nous devrions mettre en œuvre uniquement les histoires d'aujourd'hui, puis refactoriser et étendre le système pour implémenter de nouvelles histoires demain.
4. Une architecture système optimale consiste en des domaines de préoccupation modularisés, chacun étant implémenté avec des objets Java classiques (ou d'autres objets).
5. Nous oublions souvent qu'il est également préférable de reporter les décisions jusqu'au dernier moment possible.

Cela n'est ni paresseux ni irresponsable ; cela nous permet de prendre des choix éclairés avec les meilleures informations possibles.

6. Si l'agilité est compromise, la productivité en souffre et les avantages du TDD sont perdus.

Chapitre 11 | Phrases des pages -209

1. Un système peut avoir une conception parfaite sur le papier, mais s'il n'y a pas de moyen simple de vérifier que le système fonctionne réellement comme prévu, alors tout cet effort sur papier est discutable.
2. Écrire des tests conduit à de meilleures conceptions.
3. La duplication est l'ennemi principal d'un système bien conçu.
4. Plus l'auteur peut rendre le code clair, moins les autres passeront de temps à le comprendre.
5. Le soin est une ressource précieuse.
6. Un design est 'simple' s'il respecte ces règles : Passe tous les tests, Ne contient pas de duplication, Exprime

l'intention du programmeur, Minimise le nombre de classes et de méthodes.

Chapitre 12 | Phrases des pages 210-225

1. Écrire des programmes concurrentiels propres est difficile, très difficile.
2. La concurrency est une stratégie de découplage. Elle nous aide à dissocier ce qui est fait de quand c'est fait.
3. Une concurrence correcte est complexe, même pour des problèmes simples.
4. Écrire un système destiné à rester opérationnel éternellement est différent d'écrire quelque chose qui fonctionne un moment puis se ferme proprement.
5. Traitez les échecs sporadiques comme des problèmes potentiels de multi-threading.
6. Si vous adoptez une approche propre, vos chances de réussir augmentent considérablement.

Scannez pour télécharger

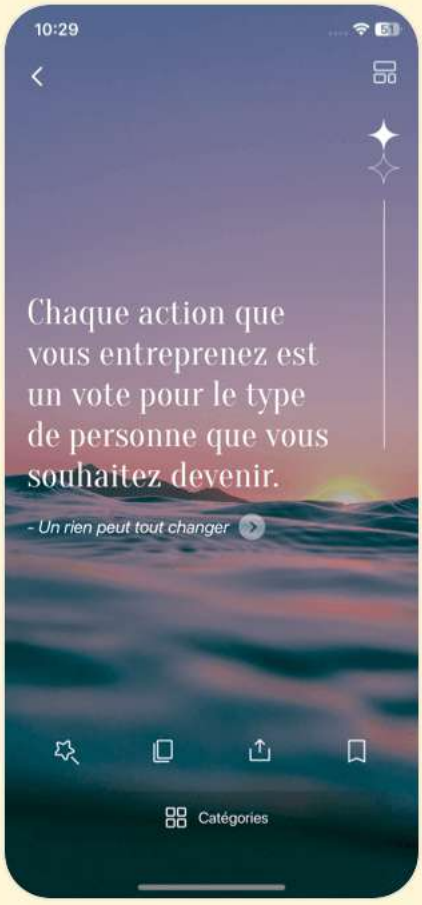
Téléchargez l'appli Bookey pour profiter

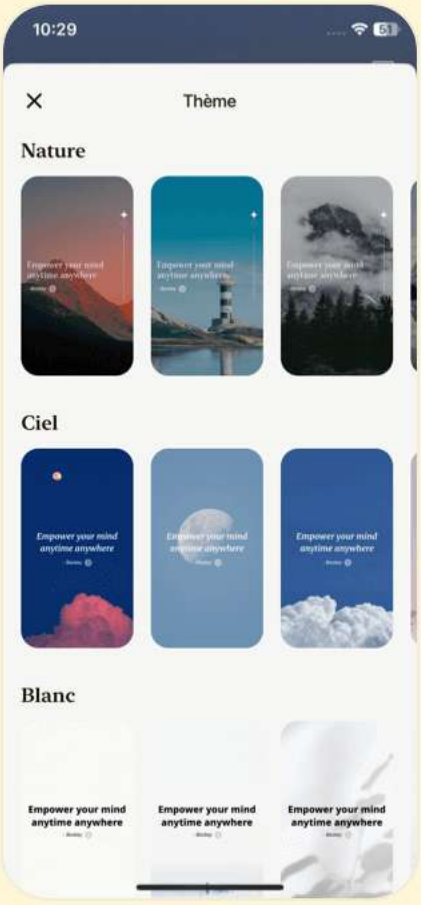
Plus d'un million de citations
Plus de 1000 résumés de livres

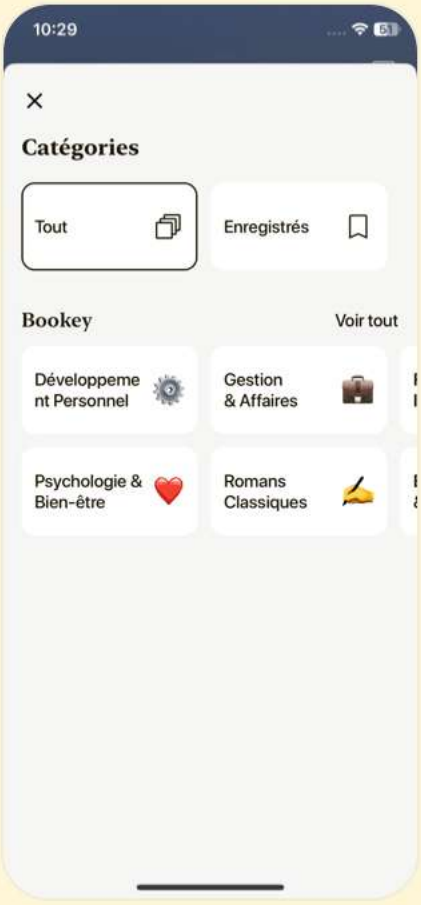
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 13 | Phrases des pages 226-283

1. Pour coder proprement, il faut d'abord écrire du code brouillon et ensuite le nettoyer.
2. La plupart des programmeurs débutants... croient que l'objectif principal est de faire fonctionner le programme. Une fois qu'il est 'fonctionnel', ils passent à la tâche suivante, laissant le programme 'fonctionnel' dans l'état dans lequel ils l'ont finalement fait 'marcher'. La plupart des programmeurs expérimentés savent que c'est un suicide professionnel.
3. Écrire des compositions propres... est une question de perfectionnement successif.
4. Rien n'a un effet dégradant plus profond et durable sur un projet de développement que du mauvais code.
5. Garder le code propre est relativement facile... Si vous avez mis le désordre dans un module le matin, il est facile de le nettoyer l'après-midi.

Chapitre 14 | Phrases des pages 284-299

1. La refactorisation est un processus itératif rempli

d'essais et d'erreurs, convergeant inévitablement vers quelque chose que nous considérons digne d'un professionnel.

2. Même si les auteurs ont laissé ce module en très bon état, la règle du Boy Scout nous dit que nous devrions le laisser plus propre que nous ne l'avons trouvé.
3. Il y a quelques longues expressions et des +1 étranges et autres. Mais dans l'ensemble, ce module est plutôt bon.
4. Chacun d'entre nous a la responsabilité de laisser le code un peu meilleur que nous ne l'avons trouvé.

Chapitre 15 | Phrases des pages 300-317

1. C'est seulement à travers des critiques comme celles-ci que nous apprendrons. Les médecins le font. Les pilotes le font. Les avocats le font. Et nous, les programmeurs, devons aussi apprendre à le faire.
2. Ce n'est pas une activité malveillante. Je ne pense pas non plus que je sois tellement meilleur que David que j'ai le droit de porter un jugement sur son code.

3.En effet, cette classe parle de jours, plutôt que de temps. J'ai envisagé de l'appeler Jour, mais ce nom est également beaucoup utilisé ailleurs. Au final, j'ai choisi DateJournée comme le meilleur compromis.

4.Nous avons maintenant des outils de contrôle de version qui font cela pour nous. Cette histoire devrait être supprimée.

5.Le schéma de l'échec en examinant quels cas de test sont commentés. Ce schéma est révélateur.

6.Il est intéressant de noter que cette fonction a été l'objet d'une réparation antérieure. L'historique des modifications montre que des 'bugs' ont été corrigés...

Scannez pour télécharger

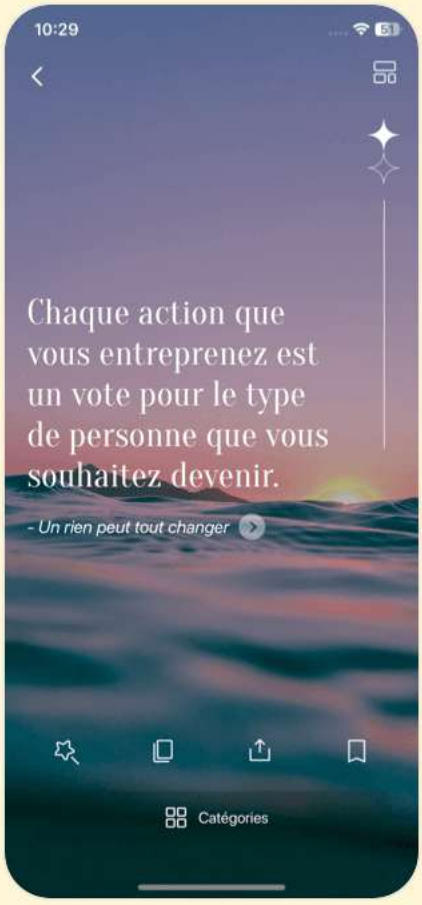
Téléchargez l'appli Bookey pour profiter

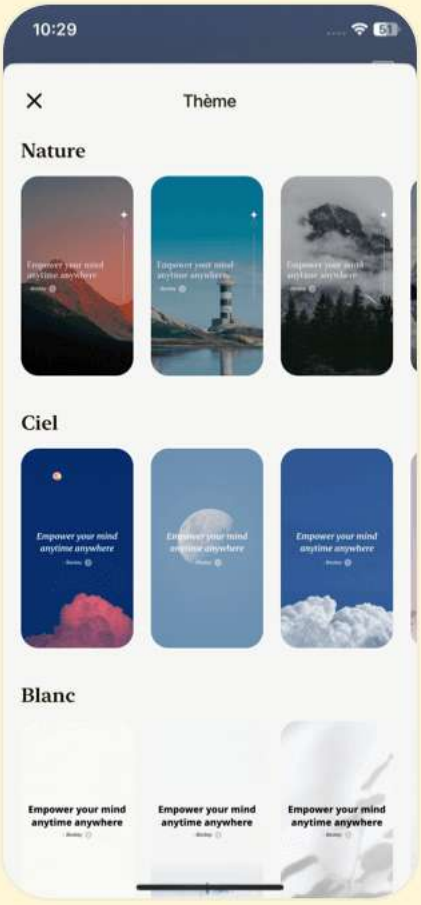
Plus d'un million de citations
Plus de 1000 résumés de livres

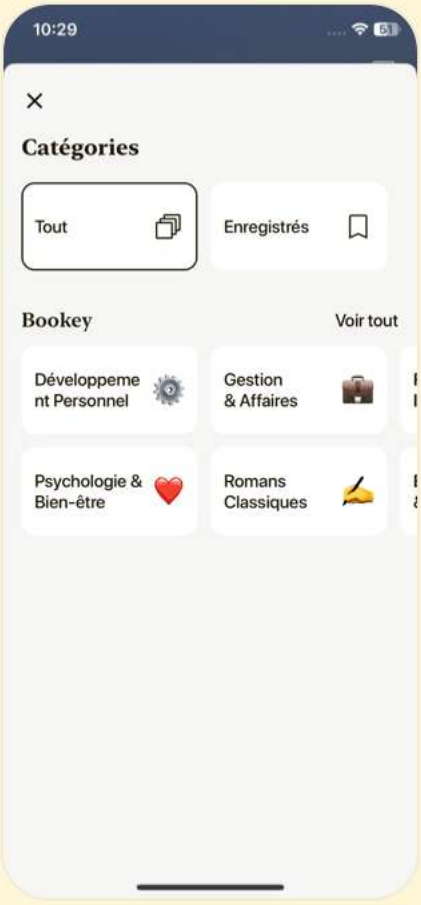
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 16 | Phrases des pages 318-349

1. Informations inappropriées. Il est inapproprié qu'un commentaire contienne des informations qui seraient mieux placées dans un autre type de système.
2. Les commentaires devraient être réservés aux notes techniques concernant le code et la conception.
3. Le code commenté est une véritable abomination.
4. Les fonctions devraient avoir un petit nombre d'arguments.
5. Chaque fois que vous voyez une duplication dans le code, cela représente une occasion manquée d'abstraction.
6. Une bonne conception logicielle nécessite que nous séparions les concepts à différents niveaux et que nous les placions dans différents conteneurs.
7. Moins une classe a de méthodes, mieux c'est.
8. Chaque fonction fait une seule chose.
9. Choisissez des noms descriptifs.
10. Ne négligez pas les tests triviaux.

Chapitre 17 | Phrases des pages -381

1. Il y a deux possibilités : I/O—utiliser une socket, se connecter à une base de données, attendre l'échange de mémoire virtuelle, etc. Processeur—calculs numériques, traitement d'expressions régulières, collecte des déchets, etc.
2. Si le code est lié au processeur, du matériel de traitement supplémentaire peut améliorer le débit, faisant passer notre test. Mais il n'y a qu'un nombre limité de cycles CPU disponibles, donc ajouter des threads à un problème lié au processeur ne le fera pas aller plus vite.
3. Pour garder les systèmes concurrents propres, la gestion des threads devrait se limiter à quelques endroits bien contrôlés. De plus, tout code qui gère des threads ne devrait faire autre chose que de la gestion de threads.
4. Interblocage. Le système ne se rétablit jamais. Cela peut sembler une situation peu probable, mais qui veut d'un système qui se fige complètement toutes les deux semaines ?
5. Comment pouvons-nous écrire un test pour démontrer que

le code suivant est défectueux ?

Chapitre 18 | Phrases des pages 382-441

1. Pourquoi ne pas simplement utiliser java.util.Date ? Nous le ferons, lorsque cela a du sens. Parfois, java.util.Date peut être *trop* précis - il représente un instant dans le temps, exact à 1/1000ème de seconde (avec la date elle-même dépendant du fuseau horaire). Parfois, nous voulons juste représenter un jour particulier (par exemple, le 21 janvier 2015) sans nous préoccuper de l'heure de la journée, du fuseau horaire, ou de quoi que ce soit d'autre. C'est pour cela que nous avons défini SerialDate.
2. Une classe abstraite qui définit nos exigences pour manipuler des dates, sans fixer une mise en œuvre particulière.
3. Cette bibliothèque est distribuée dans l'espoir qu'elle sera utile, mais SANS AUCUNE GARANTIE ; sans même la garantie implicite de COMMERCIALISATION ou

D'ADAPTATION À UN USAGE PARTICULIER.

4. Exigence 1 : correspondre au moins à ce que fait Excel pour les dates ; Exigence 2 : la classe est immuable ;
5. Vous pouvez appeler getInstance() pour obtenir une sous-classe concrète de SerialDate, sans vous préoccuper de la mise en œuvre exacte.

Scannez pour télécharger

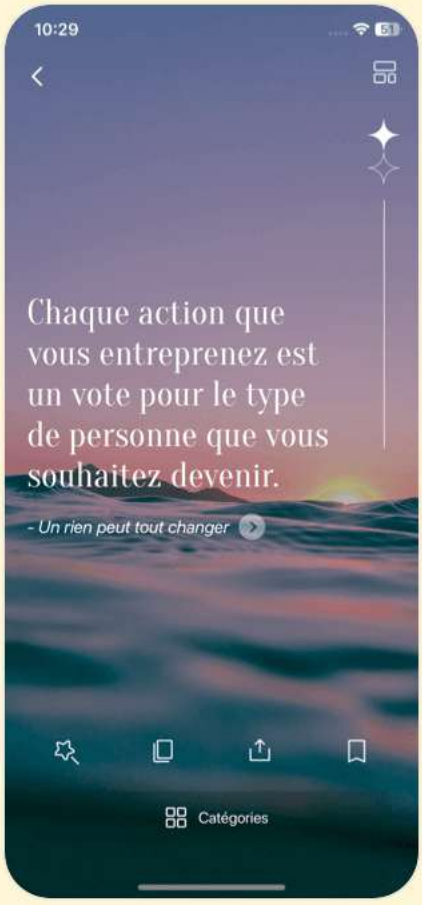
Téléchargez l'appli Bookey pour profiter

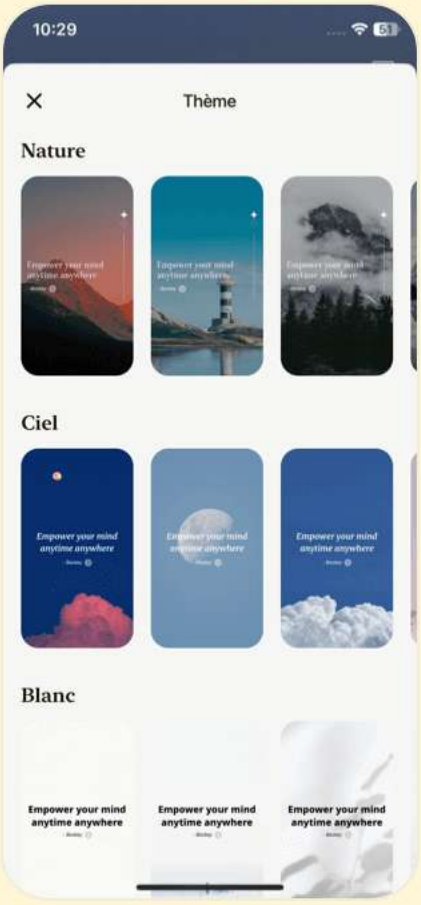
Plus d'un million de citations
Plus de 1000 résumés de livres

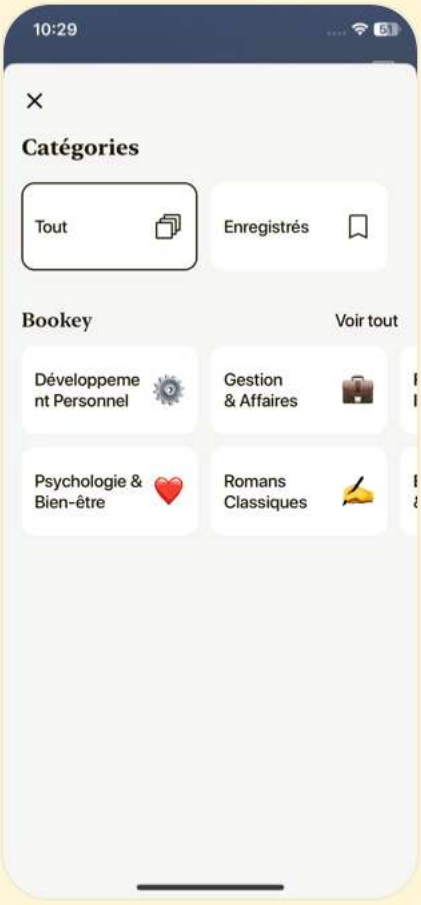
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 19 | Phrases des pages 442-443

1. ‘La seule façon d'aller vite est d'aller bien.’
2. ‘Le code est lu bien plus souvent qu'il n'est écrit.’
3. ‘On ne peut pas écrire un bon code sans un bon design.’
4. ‘La simplicité est l'âme de l'efficacité.’
5. ‘Apprendre à écrire un bon code est un voyage qui dure toute une vie.’

Chapitre 20 | Phrases des pages -465

1. L'art de CODER PROPREMENT est une question de négociation, et il commence par un design à la fois simple et expressif.
2. Le code est comme l'humour. Quand vous devez l'expliquer, c'est mauvais.
3. La règle du scout : Toujours laisser le campement plus propre que vous ne l'avez trouvé.
4. Le code propre se lit comme une prose bien écrite.
5. Un design simple permet aux développeurs de communiquer efficacement et facilite les changements.

Chapitre 21 | Phrases des pages 492-497

1. Le professionnalisme est quelque chose dont notre métier a grandement besoin.
2. Vous voyez, quand j'ai obtenu mon premier emploi en tant que programmeur, le mot professionnel était le dernier que vous auriez utilisé pour me décrire.
3. J'ai appris... que jamais on ne quitte un emploi sans en avoir un nouveau, et qu'on doit toujours partir calmement, sereinement, et seul.
4. Pensez à ce livre comme à un catalogue de mes propres erreurs, un registre de mes propres crimes, et un ensemble de lignes directrices pour vous éviter de marcher dans mes premières traces.

Scannez pour télécharger

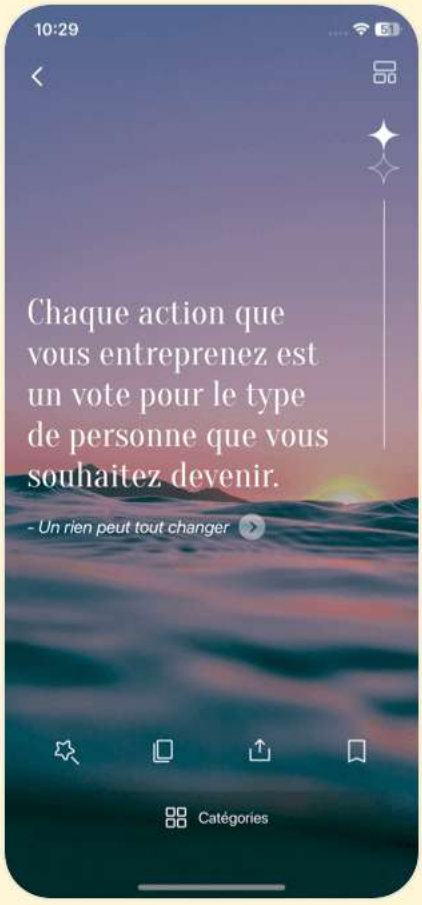
Téléchargez l'appli Bookey pour profiter

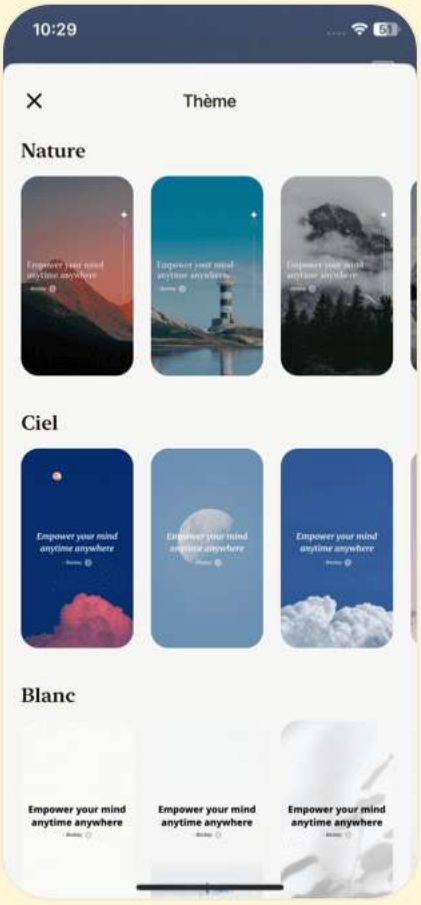
Plus d'un million de citations
Plus de 1000 résumés de livres

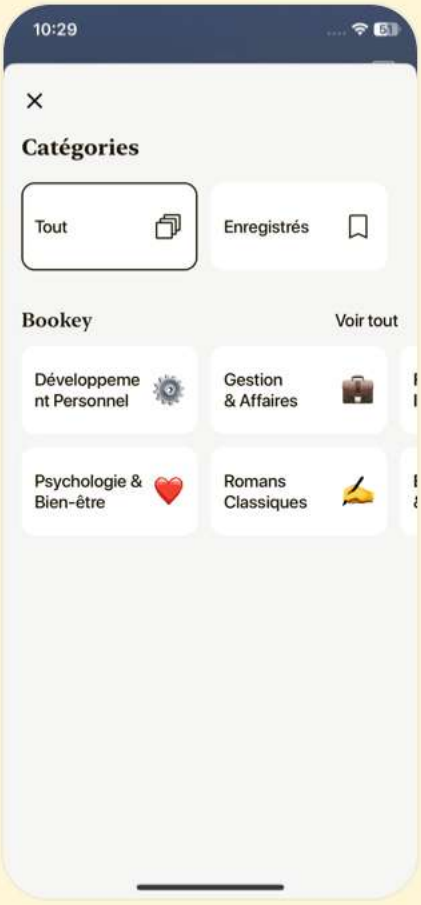
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 22 | Phrases des pages 498-513

1. Le professionnalisme est un terme chargé de connotations. C'est certes un insigne de fierté, mais c'est aussi un marqueur de responsabilité et de redevabilité.
2. Le professionnel écrirait un chèque de 10 000 $ à l'entreprise ! Oui, ça fait un peu différent quand il s'agit de votre propre argent, non ?
3. D'abord, ne pas nuire. Clairement, nous voulons que notre logiciel fonctionne.
4. Le vrai professionnel sait que livrer de la fonction au détriment de la structure est une tâche futile.
5. Votre carrière est votre responsabilité. Ce n'est pas à votre employeur de s'assurer que vous êtes attractif sur le marché.
6. Si vous voulez être un professionnel, vous devez connaître une part importante [de notre domaine] et constamment augmenter cette part.
7. Le meilleur moyen d'apprendre est d'enseigner.

8.Un professionnel a confiance en ses capacités et prend des risques audacieux et calculés en se basant sur cette confiance.

Chapitre 23 | Phrases des pages 514-535

1.Fais; ou ne fais pas. Il n'y a pas d'essai." — Yoda

2.Les professionnels disent la vérité aux puissants. Les professionnels ont le courage de dire non à leurs responsables.

3.Un bon joueur d'équipe n'est pas quelqu'un qui dit oui tout le temps.

4.La seule façon de faire votre travail, à ce moment-là, est de dire 'Non, c'est impossible.'

5.Plus les enjeux sont élevés, plus le non devient précieux.

6.Avez-vous un réservoir d'énergie supplémentaire que vous retenez ?

7.En promettant d'essayer, vous promettez de changer vos plans.

Chapitre 24 | Phrases des pages 536-547

1.Dire. Signifier. Faire. Il y a trois étapes pour

s'engager. 1. Vous dites que vous le ferez. 2. Vous le pensez vraiment. 3. Vous le faites vraiment.

2. Si vous ne trouvez pas ces petits mots magiques, il y a de fortes chances que nous ne voulions pas ce que nous disons, ou que nous ne croyions pas que cela soit réalisable.

3. Vous vous sentirez mal de ne pas l'avoir fait. Vous vous sentirez mal à l'aise de dire à quelqu'un que vous ne l'avez pas fait (si cette personne a entendu votre promesse). Effrayant, n'est-ce pas ?

4. L'ingrédient secret pour reconnaître un réel engagement est de chercher des phrases qui ressemblent à ceci : Je vais . . . d'ici . . .

5. Les professionnels ne sont pas tenus de dire oui à tout ce qu'on leur demande. Cependant, ils devraient s'efforcer de trouver des moyens créatifs pour rendre le 'oui' possible.

Scannez pour télécharger

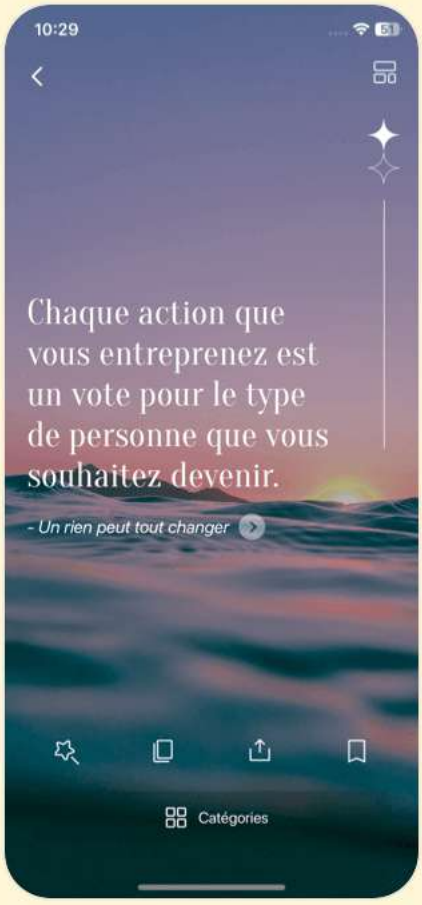
Téléchargez l'appli Bookey pour profiter

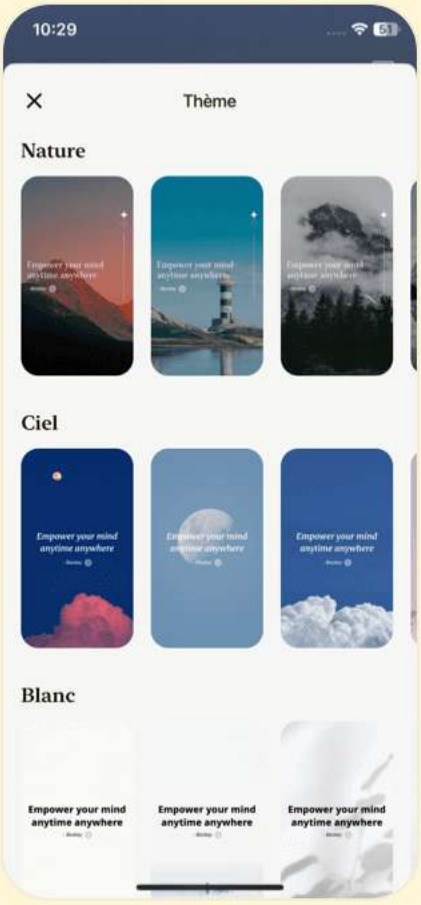
Plus d'un million de citations
Plus de 1000 résumés de livres

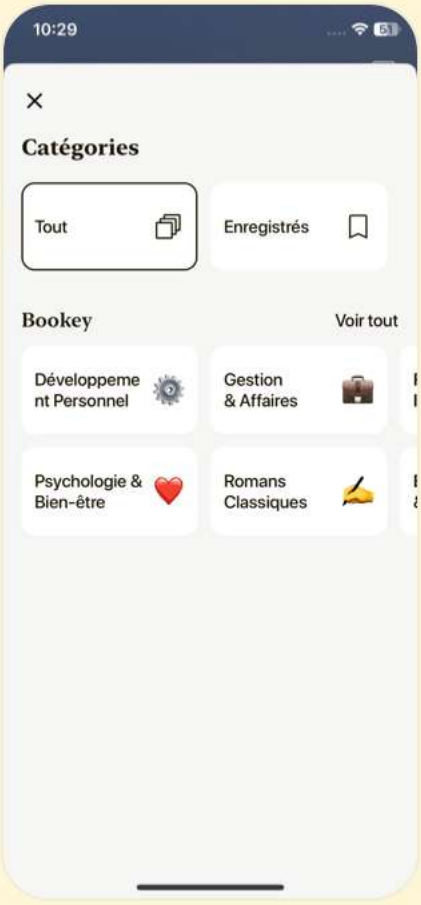
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 25 | Phrases des pages 548-567

1. Une des choses qui m'a aidé à avoir confiance, c'est que je pouvais sentir quand je faisais une erreur.
2. Quand vous ne pouvez pas concentrer et vous focaliser suffisamment, le code que vous écrivez sera faux.
3. La morale de cette histoire est : Ne faites pas de code quand vous êtes fatigué.
4. La créativité et l'intelligence sont des états d'esprit éphémères.
5. La programmation est difficile. Plus vous êtes jeune, moins vous le croyez.
6. Il est peu professionnel de rester bloqué quand l'aide est facilement accessible.
7. Une livraison fausse est peut-être le pire de tous les comportements non professionnels dans lesquels un programmeur peut se laisser aller.
8. Ne laissez personne d'autre avoir de l'espoir.

Chapitre 26 | Phrases des pages 568-575

1. Le verdict est tombé ! La controverse est terminée. GOTO est nuisible. Et le TDD fonctionne.
2. Comment pouvez-vous vous considérer comme un professionnel si vous ne savez pas que tout votre code fonctionne ?
3. Si vous adoptez le TDD comme discipline professionnelle, alors vous écrirez des dizaines de tests chaque jour, des centaines de tests chaque semaine et des milliers de tests chaque année.
4. Lorsque les programmeurs n'ont plus peur de nettoyer, ils nettoient ! Et un code propre est plus facile à comprendre, plus facile à changer et plus facile à étendre.
5. Chacun des tests unitaires que vous écrivez lorsque vous suivez les trois lois est un exemple, écrit dans le code, décrivant comment le système doit être utilisé.
6. La conclusion de tout cela est que le TDD est l'option professionnelle. C'est une discipline qui améliore la certitude, le courage, la réduction des défauts, la documentation et le design.

Chapitre 27 | Phrases des pages 576-585

1. Tous les professionnels pratiquent leur art en s'engageant dans des exercices pour affiner leurs compétences.
2. La nature des déclarations n’a pas changé depuis tout ce temps.
3. La rapidité dépend de la pratique.
4. L'objectif est d'entraîner votre esprit et votre corps à réagir dans une situation de combat particulière.
5. Il est remarquable combien vous pouvez apprendre de ces séances.
6. Pratiquer, c'est ce que vous faites quand vous n'êtes pas payé.
7. De plus, ils pratiquent sur leur temps libre car ils réalisent que c'est leur responsabilité — et non celle de leur employeur — de garder leurs compétences aiguisées.

Scannez pour télécharger

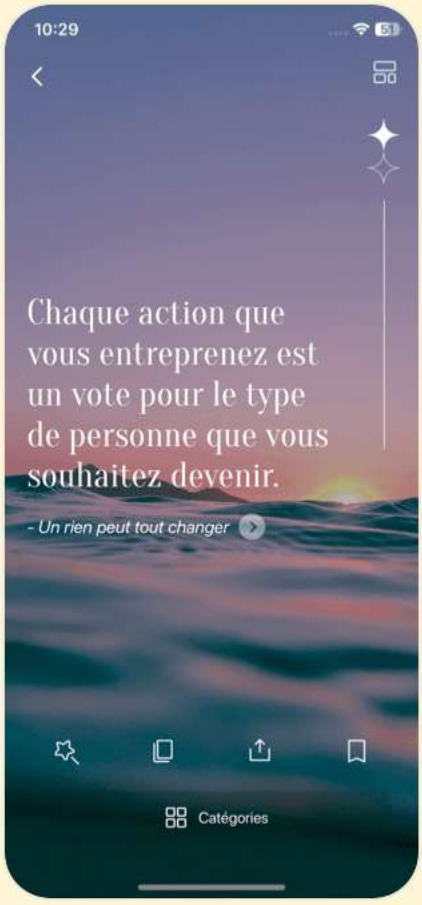
Téléchargez l'appli Bookey pour profiter

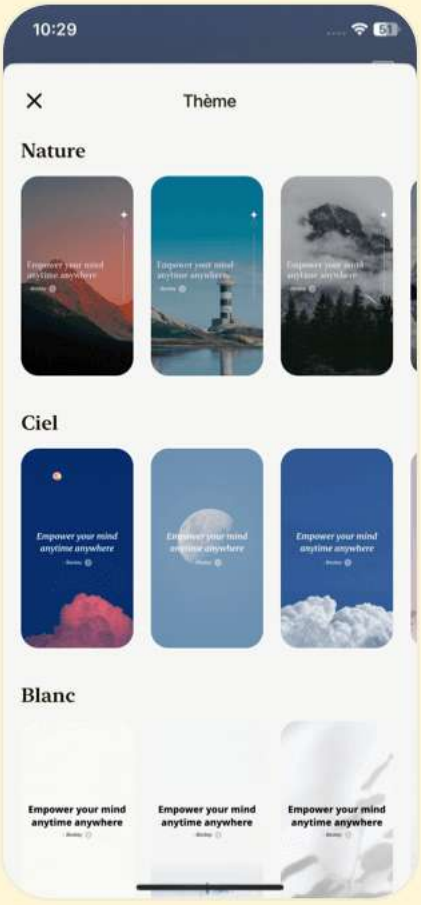
Plus d'un million de citations
Plus de 1000 résumés de livres

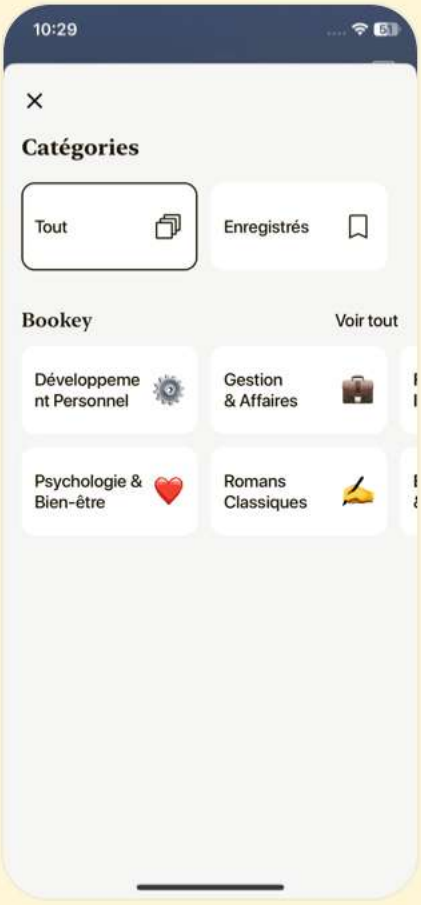
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 28 | Phrases des pages -603

1. Le rôle du développeur professionnel est un rôle de communication ainsi qu'un rôle de développement.
2. Cela nous a pris toute une journée. Il décrivait une fonctionnalité et je l'implémentais sous ses yeux.
3. Le problème, c'est que les choses apparaissent différemment sur papier que dans un système fonctionnel.
4. Plus vous précisez vos exigences, moins elles deviennent pertinentes au fur et à mesure que le système est mis en œuvre.
5. Il est de la responsabilité des développeurs professionnels (et des parties prenantes) de s’assurer que toute ambiguïté est éliminée des exigences.
6. Les développeurs professionnels ne détaillent une exigence que lorsqu'ils sont sur le point de la développer.
7. Lorsqu'un développeur dit qu'il a terminé une tâche, que signifie cela ?
8. Les tests d'acceptation devraient toujours être automatisés.
9. Écrire ces tests est simplement le travail de spécification du

système.

10.Le coût de l'automatisation des tests d'acceptation est si faible par rapport au coût de l'exécution des plans de test manuel qu'il n'a aucun sens économique d'écrire des scripts pour que des humains les exécutent.

Chapitre 29 | Phrases des pages 604-611

1.Les développeurs professionnels testent leur code.

2.Ce que chaque équipe de développement professionnelle a besoin, c'est d'une bonne stratégie de test.

3.L'objectif du groupe de développement devrait être que l'assurance qualité ne trouve rien de défectueux.

4.L'assurance qualité et le développement devraient travailler ensemble pour garantir la qualité du système.

5.L'intention de ces tests est de spécifier le système au niveau le plus bas.

6.Le comportement correct du code et des composants sous-jacents a déjà été vérifié par les couches inférieures de la pyramide.

7.L'objectif est de garantir que le système se comporte bien

lors d'une opération humaine et de découvrir aussi créativement que possible autant de 'particularités' que possible.

Chapitre 30 | Phrases des pages 612-625

1. Huit heures, c'est une période de temps remarquablement courte. C'est seulement 480 minutes ou 28 800 secondes.
2. Vous n'avez pas à assister à chaque réunion à laquelle vous êtes invité.
3. Quand la réunion devient ennuyeuse, partez.
4. La concentration est une ressource rare, un peu comme la manne.
5. Quand vous êtes dans une situation difficile, arrêtez de creuser.
6. Il n'y a pas de spectacle plus triste qu'une équipe de développeurs de logiciels s'enlisant dans un marécage de plus en plus profond.

Scannez pour télécharger

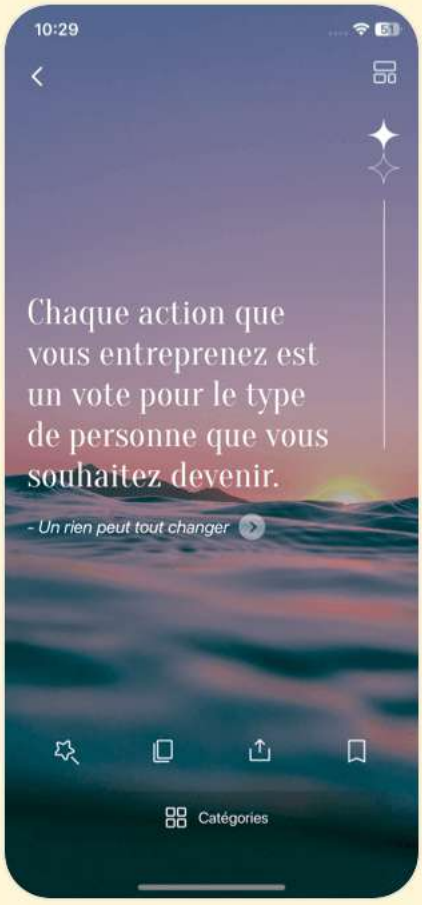
Téléchargez l'appli Bookey pour profiter

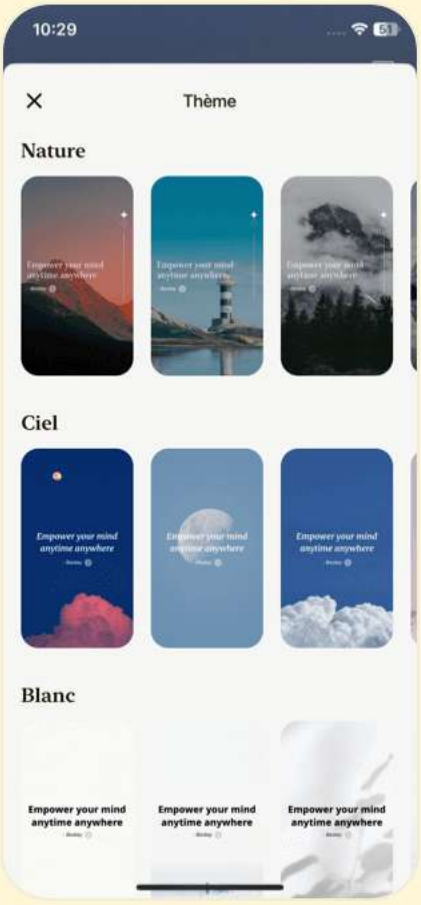
Plus d'un million de citations
Plus de 1000 résumés de livres

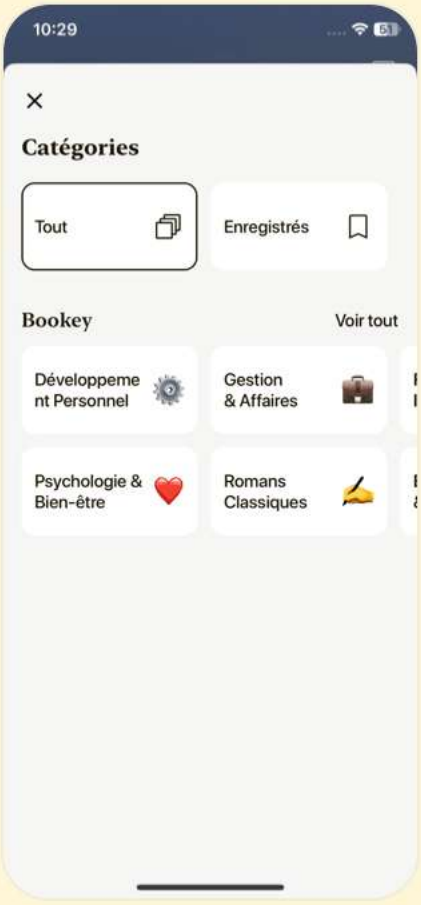
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 31 | Phrases des pages 626-639

1. L'estimation est l'une des activités les plus simples, mais aussi les plus effrayantes auxquelles les professionnels du logiciel sont confrontés.
2. Les entreprises aiment considérer les estimations comme des engagements. Les développeurs aiment voir les estimations comme des suppositions.
3. Un engagement est quelque chose que vous devez réaliser. Une estimation est une supposition.
4. Une estimation est une distribution.
5. La ressource d'estimation la plus importante que vous ayez, ce sont les personnes qui vous entourent.

Chapitre 32 | Phrases des pages 640-647

1. Le développeur professionnel reste calme et décisif sous pression.
2. Tout a changé ce jour-là. J'ai arrêté les heures folles. J'ai arrêté ce mode de vie stressant. J'ai cessé de lancer des stylos et d'écrire des fonctions C de 3 000 lignes.
3. La meilleure façon de rester calme sous pression est

d’éviter les situations qui créent cette pression.

4. Vous savez ce que vous croyez en vous observant dans une crise. Si, en période de crise, vous suivez vos disciplines, alors vous croyez vraiment en ces disciplines.
5. Résistez à cette tentation à tout prix. Se précipiter ne fera que vous enfoncer davantage dans le trou.
6. Le secret pour gérer la pression est de l’éviter quand c’est possible et d’y faire face quand ce n’est pas le cas.

Chapitre 33 | Phrases des pages 648-657

1. La plupart des logiciels sont créés par des équipes. Les équipes sont les plus efficaces lorsque les membres collaborent de manière professionnelle.
2. La seule fois où j'ai été licencié d'un emploi de programmation, c'était en 1976.
3. C'est bien d'être passionné par ce que nous faisons. Mais il est également important de garder un œil sur les objectifs de ceux qui vous paient.
4. La pire chose qu'un programmeur professionnel puisse faire est de s'enfermer dans une tombe de technologie

pendant que l'entreprise s'effondre autour de lui.

5. Les développeurs professionnels ne empêchent pas les autres de travailler dans le code. Ils ne construisent pas de murs de propriété autour du code.
6. Deux têtes valent mieux qu'une. Mais si le travail en binôme est la manière la plus efficace de résoudre un problème en cas d'urgence, pourquoi ne serait-ce pas la manière la plus efficace de résoudre un problème tout court ?
7. Peut-être que nous ne sommes pas entrés dans la programmation pour travailler avec des gens. Tant pis pour nous.

Scannez pour télécharger

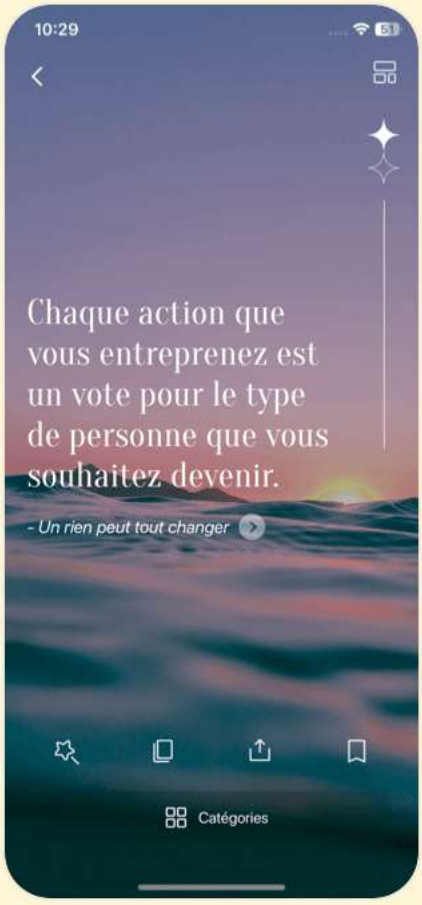
Téléchargez l'appli Bookey pour profiter

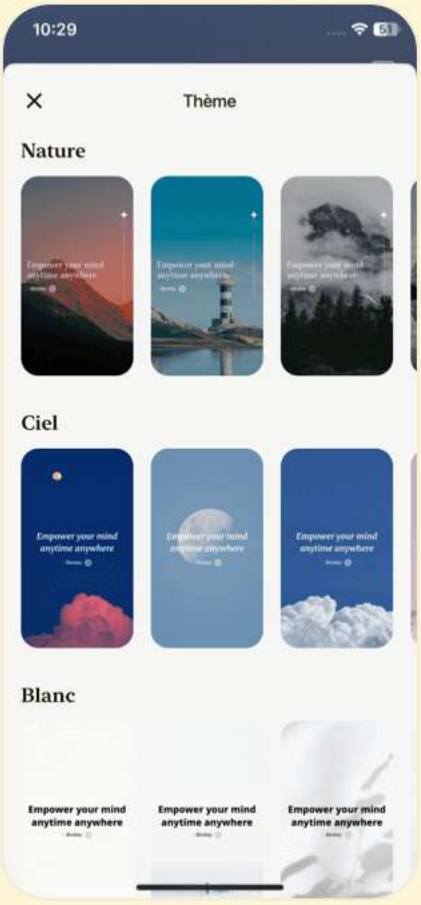
Plus d'un million de citations
Plus de 1000 résumés de livres

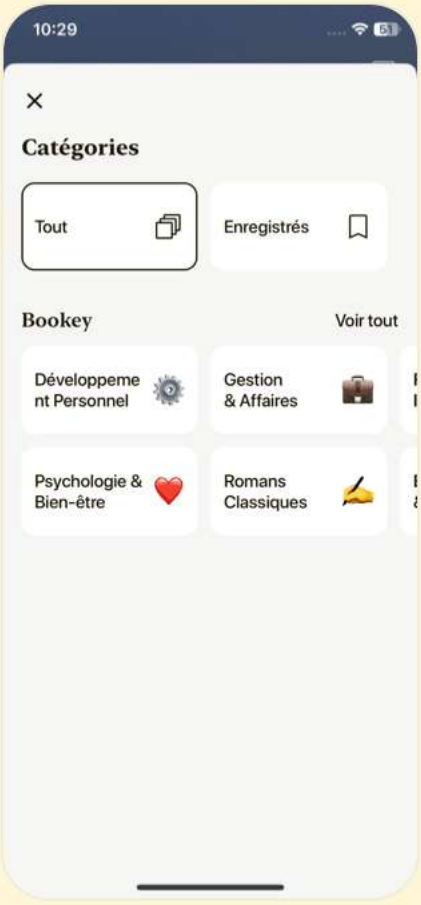
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 34 | Phrases des pages 658-663

1. Il n'existe pas de personne à moitié.
2. Il y a quelque chose de vraiment magique dans une équipe bien rodée.
3. Il faut du temps à une équipe comme celle-ci pour travailler sur ses différences, apprendre à se connaître et vraiment s'harmoniser.
4. L'entreprise ne devrait pas avoir les mains liées par la difficulté artificielle de former et de dissoudre des équipes.
5. L'objectif en formant une équipe est de lui donner suffisamment de temps pour s'harmoniser, puis de la maintenir ensemble comme un moteur pour réaliser de nombreux projets.

Chapitre 35 | Phrases des pages 664-677

1. J'ai constamment été déçu par la qualité des diplômés en informatique.
2. Même les meilleurs programmes de diplôme en informatique ne préparent généralement pas le jeune diplômé à ce qu'il trouvera dans l'industrie.

3.Je n'ai pas tout découvert par moi-même. J'ai été mentoré.

4.Il aurait été bien mieux pour moi si j'avais eu un véritable mentor, quelqu'un pour m'enseigner les ficelles du métier.

5.Que font les médecins ? Pensez-vous que les hôpitaux embauchent des diplômés en médecine et les jettent dans les salles d'opération pour faire des opérations à cœur ouvert dès leur premier jour de travail ?

6.L'artisanat est un mème qui contient des valeurs, des disciplines, des techniques, des attitudes et des réponses.

Chapitre 36 | Phrases des pages 678-695

1. Aujourd'hui, les développeurs de logiciels disposent d'un large éventail d'outils parmi lesquels choisir. La plupart ne valent pas la peine d'être utilisés, mais il y en a quelques-uns avec lesquels chaque développeur de logiciels doit être familiarisé.

2.S'il y a de nouvelles épingles sur le tableau, nous retirerions nos épingles et remettrions notre bande de travail à la personne dont les épingles étaient encore sur le tableau. Ils

devraient faire la fusion.

3. En aucun cas, l'échec ne doit persister pendant un jour ou plus.
4. Le but est que vous devez être en mesure de dire que tous les tests ont réussi rapidement et sans ambiguïté.
5. Le problème, c'est le détail. Les programmeurs sont des gestionnaires de détails. C'est ce que nous faisons.
6. La courbe d'apprentissage est élevée, et le temps de configuration du projet n'est pas négligeable. Ces outils ne sont pas légers.
7. Il se peut que votre entreprise ait investi une petite fortune dans un système de contrôle de version 'entreprise'. Si c'est le cas, je vous présente mes condoléances.
8. Les outils examinent les deux fichiers différents ainsi que l'ancêtre de ces deux fichiers, puis ils appliquent plusieurs stratégies pour déterminer comment intégrer les changements concurrents.
9. Vous pouvez dire si vous êtes prêt à enregistrer du code en fonction de si tous les tests automatisés passent.

10. En fin de compte, tout tourne autour des détails, et ce sont les programmeurs qui gèrent ces détails.

Scannez pour télécharger

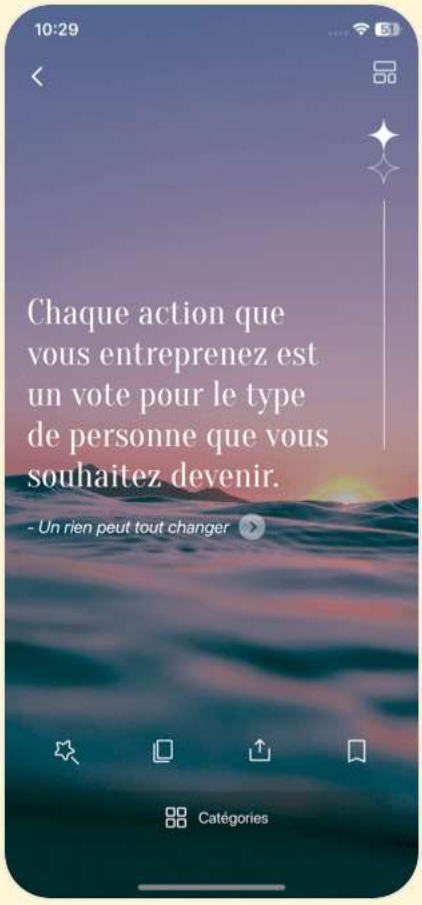
Téléchargez l'appli Bookey pour profiter

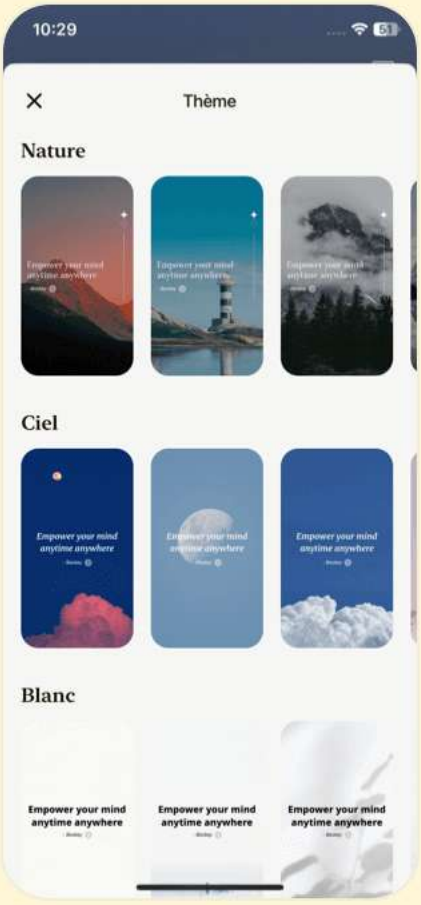
Plus d'un million de citations
Plus de 1000 résumés de livres

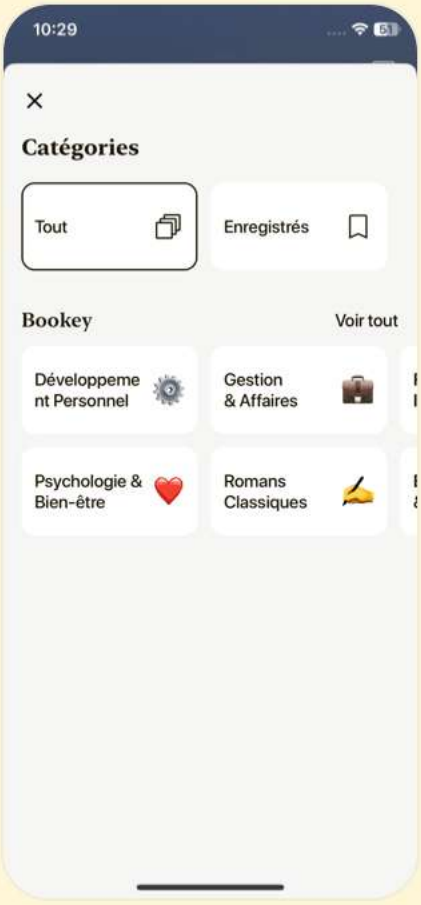
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







Chapitre 37 | Phrases des pages -703

1. La discipline du codage implique un engagement envers la qualité, la clarté et la simplicité. Adoptez l'art de l'artisanat, car un excellent code naît d'une dévotion passionnée.
2. L'éthique de travail va au-delà de la simple exécution de ses tâches ; c'est un engagement envers l'amélioration continue et envers ceux qui vous entourent.
3. La communication est essentielle. Sans elle, le processus de développement peut plonger dans le chaos, et les malentendus peuvent mener à l'échec.
4. Adoptez la peurlessness dans votre travail, surtout quand il s'agit de remettre en question les idées et pratiques existantes. L'innovation découle de la remise en question du statu quo.
5. Un excellent logiciel se construit en collaboration. La dynamique d'équipe favorise la créativité, l'insight et la responsabilité partagée.

Scannez pour télécharger

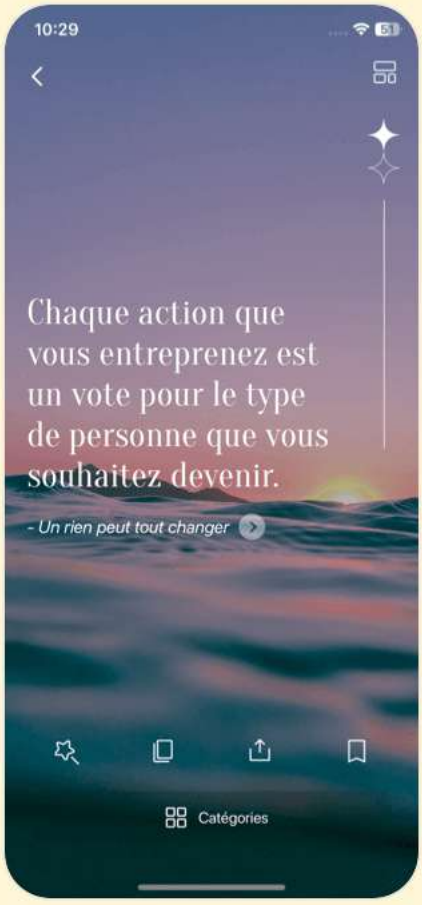
Téléchargez l'appli Bookey pour profiter

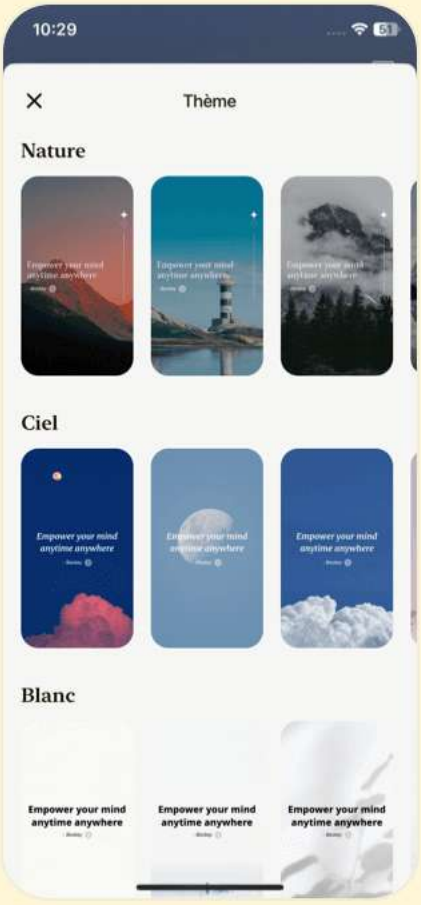
Plus d'un million de citations
Plus de 1000 résumés de livres

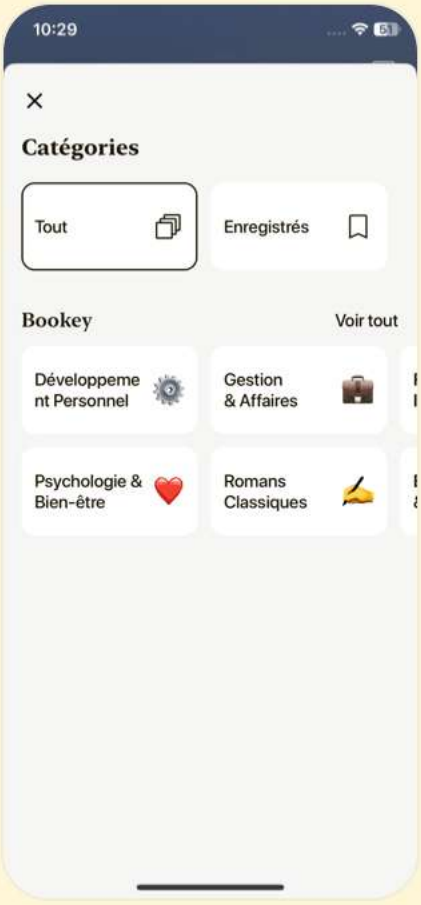
Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play







CODER PROPREMENT Questions

Voir sur le site de Bookey

Chapitre 1 | 2 Noms Significatifs| Questions et réponses

1.Question

Pourquoi les noms significatifs sont-ils importants dans le code ?

Réponse:Les noms significatifs sont cruciaux car ils améliorent la lisibilité et la compréhension du code. Ils doivent transmettre l'objectif et la fonctionnalité de la variable, de la fonction ou de la classe, rendant ainsi plus facile pour les développeurs de maintenir et de modifier le code sans avoir à déchiffrer la logique derrière des noms ambigus.

2.Question

Quelle est la règle concernant les noms révélant l'intention ?

Réponse:La règle est d'utiliser des noms qui révèlent l'intention de la variable, de la fonction ou de la classe. Un bon nom doit aider à répondre à des questions clés comme

pourquoi il existe, ce qu'il fait et comment il est utilisé. Si un nom nécessite un commentaire pour l'expliquer, alors le nom ne fournit pas suffisamment de contexte.

3.Question

Comment éviter la désinformation peut-il aider en programmation ?

Réponse:Éviter la désinformation implique d'utiliser des noms qui représentent avec précision leur but et leur contexte, empêchant ainsi toute confusion. Par exemple, utiliser un nom clair comme 'jeuPlateau' au lieu de 'laListe' rend le code explicite et permet à quiconque lisant le code de comprendre immédiatement ce qui est référencé.

4.Question

Que devez-vous considérer lors de la nomination des variables pour éviter la confusion ?

Réponse:Pour éviter la confusion, il est essentiel de choisir des noms distincts et significatifs qui ne se ressemblent pas (par exemple, éviter 'l' et '1' ou 'O' et '0'). De plus, les noms doivent être suffisamment clairs pour ne pas nécessiter de

traduction en une terminologie plus familière pour la compréhension.

5.Question

Pourquoi les noms de variables d'une seule lettre doivent-ils être évités en dehors des boucles ?

Réponse:Les noms d'une seule lettre peuvent être non informatifs et créer de l'ambiguïté. Ils ne transmettent aucun concept, rendant plus difficile pour d'autres développeurs (et vous-même à l'avenir) de comprendre le code, car ils nécessitent un mapping mental pour saisir ce que la variable représente.

6.Question

Quelle est la signification des noms de classes par rapport aux noms de méthodes ?

Réponse:Les noms de classes doivent être des noms ou des phrases nominales qui représentent avec précision l'objet modélisé, tandis que les noms de méthodes doivent être des verbes ou des phrases verbales décrivant des actions. Cette distinction aide à clarifier le rôle de chaque élément dans le

code, rendant l'ensemble plus intuitif.

7.Question

Comment le contexte influence-t-il les noms de variables ?

Réponse:Le contexte améliore considérablement la clarté des noms de variables. Par exemple, utiliser des noms comme 'addrPrenom' fournit des indices contextuels que ces variables font partie d'une structure d'adresse. Un contexte approprié aide les lecteurs à saisir immédiatement les relations entre les éléments du code.

8.Question

Quels conseils sont donnés concernant les conventions de nommage pour différents contextes comme les domaines ?

Réponse:Lors du choix des noms, il est bénéfique d'utiliser la terminologie appropriée en fonction du contexte ; utilisez des termes d'informatique pour les aspects techniques et des noms de domaine de problème lorsque cela est pertinent, pour garantir que les mainteneurs puissent comprendre la signification sans confusion.

9.Question

Que suggère l'auteur à propos du changement de nom de

variables ou de fonctions ?

Réponse:L'auteur encourage à renommer les variables ou les fonctions lorsque cela est approprié et suggère que ces changements conduisent généralement à une meilleure lisibilité. Il peut y avoir une résistance initiale, mais finalement, de meilleurs noms contribuent positivement à la maintenabilité du code.

10.Question

Comment les programmeurs devraient-ils différencier les concepts similaires dans la nomination ?

Réponse:Les programmeurs devraient systématiquement choisir un seul terme pour chaque concept abstrait et s'y tenir. Mélanger des terminologies comme 'récupérer', 'obtenir' et 'prendre' peut conduire à confusion, donc utiliser un terme de manière cohérente simplifie la compréhension.

Chapitre 2 | 3 Fonctions| Questions et réponses

1.Question

Quelle est la première règle pour écrire des fonctions ?

Réponse:Les fonctions doivent être petites. Cela

signifie que nous devons nous efforcer de garder les fonctions encore plus courtes que 20 lignes chaque fois que cela est possible.

2.Question

Comment pouvons-nous faire en sorte que les fonctions communiquent leur intention ?

Réponse:Pour communiquer clairement l'intention, les fonctions doivent faire une seule chose bien et elles doivent être nommées de manière descriptive. Un bon nom reflète ce que fait la fonction et minimise l'ambiguïté.

3.Question

Pourquoi les fonctions devraient-elles avoir un maximum de deux ou trois arguments ?

Réponse:Avoir trop d'arguments complique la compréhension et le test de la fonction. Moins d'arguments conduisent à des fonctions plus claires et plus faciles à lire, moins sujettes aux erreurs.

4.Question

Quelle est l'importance d'éviter les effets secondaires dans les fonctions ?

Réponse:Les effets secondaires créent des dépendances cachées et un comportement inattendu dans le code, rendant difficile sa compréhension et son débogage. Les fonctions ne doivent effectuer que leur tâche déclarée sans altérer l'état des variables ou systèmes externes.

5.Question

Que devez-vous faire quand vous avez une fonction qui fait plus d'une chose ?

Réponse:Si une fonction fait trop de choses, envisagez de la diviser en petites fonctions qui chacune exécutent une seule tâche. Cela augmente la lisibilité et la maintenabilité.

6.Question

À quoi fait référence la `Règle de réduction` ?

Réponse:La Règle de réduction souligne que le code doit se lire comme un récit hiérarchique, où chaque fonction introduit le niveau d'abstraction suivant, permettant une compréhension et un flux faciles.

7.Question

Pourquoi est-il problématique de mélanger différents niveaux d'abstraction dans une fonction ?

Réponse:Mélanger différents niveaux d'abstraction confond les lecteurs sur ce qui est essentiel par rapport à ce qui n'est qu'un détail, ce qui entraîne des malentendus et des erreurs potentielles dans le code.

8.Question

Quel est le principe de `Séparation des Commandes et des Requêtes` ?

Réponse:Ce principe stipule que les fonctions doivent soit modifier l'état (commandes), soit retourner des informations (requêtes), mais pas les deux, car cela peut créer de l'ambiguïté et réduire la clarté.

9.Question

Quelle est l'importance de la nomenclature dans les fonctions ?

Réponse:La nomenclature est cruciale car un nom bien choisi peut transmettre l'objectif de la fonction, rendant plus facile pour les autres de comprendre et d'utiliser le code sans nécessiter de commentaires excessifs.

10.Question

Quelle approche devriez-vous adopter concernant la

gestion des erreurs dans les fonctions ?

Réponse:Il est préférable d'utiliser des exceptions pour la gestion des erreurs au lieu de retourner des codes d'erreur. Cela sépare la logique de gestion des erreurs de la logique normale et empêche des structures trop imbriquées.

11.Question

Pourquoi devrions-nous éviter d'utiliser des arguments de type drapeau dans les fonctions ?

Réponse:Les arguments de type drapeau indiquent généralement qu'une fonction fait plus d'une chose, ce qui viole le principe de garder les fonctions concentrées sur une seule tâche.

12.Question

Comment la création d'objets d'arguments peut-elle aider avec les arguments de fonction ?

Réponse:En enveloppant plusieurs arguments dans un seul objet, on peut réduire le nombre de paramètres passés à une fonction, ce qui rend la signature de la fonction plus claire et facilite la compréhension des données associées.

13.Question

Quel devrait être le nombre idéal de points de retour d'une fonction ?

Réponse:Les fonctions devraient idéalement avoir un seul point de retour pour éviter toute confusion et améliorer la lisibilité, bien que pour des fonctions très petites, plusieurs retours puissent être acceptables.

14.Question

Que devez-vous faire lorsqu'une fonction devient trop complexe ou fastidieuse ?

Réponse:Refactorisez régulièrement la fonction en la décomposant, en renommant les variables et en simplifiant la logique pour garder le code propre et compréhensible.

15.Question

Quel est le rôle des fonctions en programmation selon le chapitre ?

Réponse:Les fonctions servent de verbes dans un langage spécifique au domaine que les programmeurs créent pour décrire le système qu'ils développent, facilitant ainsi une communication plus claire de l'intention du code.

16.Question

Comment peut-on maintenir les fonctions sur le long terme ?

Réponse:En respectant des principes tels que garder les fonctions petites, les nommer de manière descriptive, éviter les effets secondaires et adhérer au principe de responsabilité unique, les fonctions peuvent rester propres, organisées et plus faciles à maintenir.

Chapitre 3 | 4 Commentaires| Questions et réponses

1.Question

Pourquoi Robert C. Martin soutient-il que les commentaires sont souvent un signe d'échec dans le code ?

Réponse:Martin affirme que les commentaires sont nécessaires lorsque le code ne parvient pas à transmettre clairement son intention. Il pense qu'un code bien écrit devrait être auto-explicatif. Si un programmeur ressent le besoin de commenter, cela indique souvent que le code lui-même doit être mieux écrit pour exprimer son but sans explication

supplémentaire.

2.Question

Quel est le principal danger des commentaires vieillissants dans le code ?

Réponse:À mesure que le code évolue, les commentaires peuvent devenir obsolètes ou inexactes, entraînant de la désinformation. Des commentaires anciens peuvent induire les programmeurs en erreur sur la fonctionnalité du code, créant un risque de malentendu et d'erreurs.

3.Question

Quelle analogie Martin utilise-t-il pour décrire les commentaires dans le code ?

Réponse:Il compare les commentaires à un mal nécessaire, soulignant que bien qu'ils puissent être utiles, ils représentent souvent un échec à communiquer clairement à travers le code lui-même.

4.Question

Que suggère Martin aux programmeurs de faire plutôt que de commenter un code confus ?

Réponse:Il encourage les programmeurs à se concentrer

d'abord sur le nettoyage du code, le rendant plus clair et plus expressif, plutôt que de se fier aux commentaires pour expliquer un désordre.

5.Question

Quels exemples de bons commentaires Martin reconnaît-il ?

Réponse:Martin souligne que les commentaires légaux, les commentaires informatifs et ceux qui donnent des avertissements sur les conséquences peuvent être bénéfiques, surtout lorsqu'ils clarifient des aspects tels que les mentions de droits d'auteur ou des détails importants spécifiques.

6.Question

Comment Martin différencie-t-il les mauvais commentaires des bons commentaires ?

Réponse:Les mauvais commentaires incluent ceux qui sont redondants, trompeurs, inutiles ou qui ne servent que de béquilles pour un code mal écrit. Les bons commentaires, en revanche, devraient être minimaux, pertinents et ajouter de la clarté sans créer de désordre.

7.Question

Pourquoi les commentaires décrivant le code doivent-ils être évités dans le code non public ?

Réponse:Parce que les javadocs et des commentaires similaires dans le code non public ajoutent une complexité et un désordre inutiles. Ils ne sont généralement pas utiles pour un usage interne et peuvent distraire de la compréhension du code.

8.Question

Selon Martin, que doit-on faire avec le code commenté ?

Réponse:Martin s'oppose fermement au fait de commenter du code, suggérant qu'il devrait plutôt être supprimé. Il estime que les systèmes de contrôle de version peuvent mieux gérer l'historique du code que de laisser des vestiges dans la base de code.

9.Question

Que suggère Martin comme solution lorsqu'un commentaire ne se connecte pas clairement au code adjacent ?

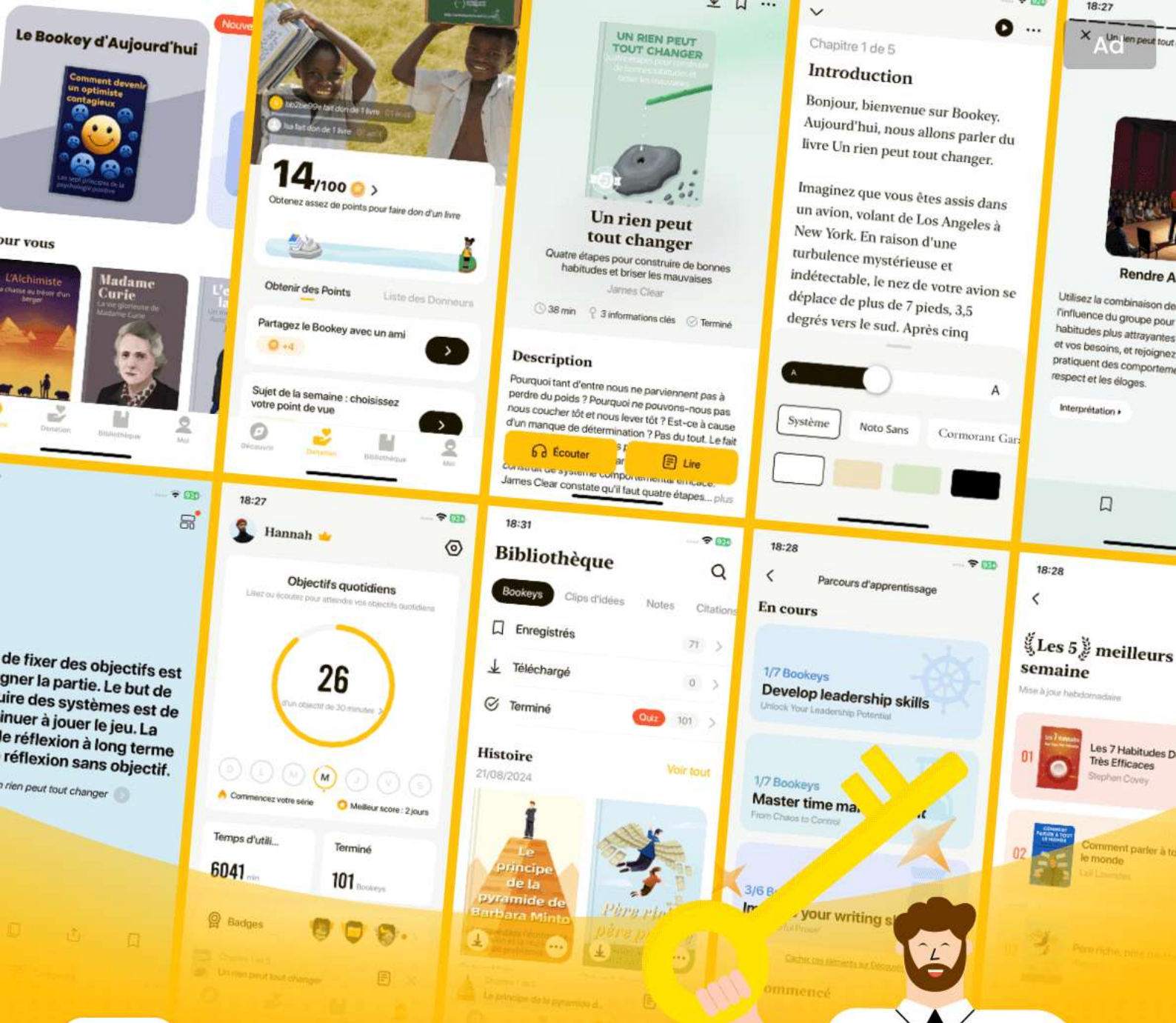
Réponse:Martin conseille que si un commentaire nécessite

une explication supplémentaire ou ne se rapporte pas clairement au code voisin, il devrait être révisé ou éliminé, car la connexion devrait être instinctive et évidente.

10.Question

Quelle est l'importance de la nomination dans l'écriture de code, selon Martin ?

Réponse:Il souligne que des noms bien choisis pour les fonctions, les variables et les classes peuvent souvent remplacer le besoin de commentaires en transmettant l'intention, rendant le code plus compréhensible.



Les meilleures idées du monde débloquent votre potentiel

Essai gratuit avec Bookey

Télécharger dans App Store

DISPONIBLE SUR Google Play

Scanner pour télécharger

Chapitre 4 | 5 Mise en forme| Questions et réponses

1.Question

Pourquoi le formatage du code est-il essentiel pour les développeurs professionnels ?

Réponse:Le formatage du code est primordial car il impacte considérablement la communication. Un code bien formaté améliore la lisibilité, signalant le professionnalisme et l'attention aux détails à quiconque le examine. Il garantit que les développeurs peuvent rapidement comprendre l'intention et la structure du code, facilitant ainsi le débogage, la maintenance et les futures modifications.

2.Question

Comment le formatage vertical peut-il améliorer la lisibilité du code ?

Réponse:Le formatage vertical aide à la lisibilité en veillant à ce que les concepts liés soient visuellement regroupés. Par exemple, séparer les fonctions par des lignes vides permet

aux lecteurs de distinguer différentes sections logiques du code, tout comme des paragraphes dans un article. Cela minimise la confusion et permet aux développeurs de se concentrer sur des fonctionnalités spécifiques sans avoir à trier à travers un code encombré.

3.Question

Que signifie la métaphore du 'journal' en ce qui concerne l'organisation du code ?

Réponse:La métaphore du journal suggère que les fichiers sources doivent être structurés comme des articles de presse. Cela signifie commencer par un en-tête informatif (le nom de la classe), suivi d'un aperçu général (les fonctions principales), puis descendre dans des implémentations détaillées et des algorithmes spécifiques, promouvant ainsi à la fois la clarté et la facilité de navigation.

4.Question

Pourquoi des fonctions étroitement liées doivent-elles être maintenues verticalement proches dans le code ?

Réponse:Garder des fonctions étroitement liées verticalement

proches les unes des autres reflète leur connexion conceptuelle, améliorant le flux logique du code. Cette pratique de design permet aux développeurs de saisir rapidement comment les fonctions interagissent, car l'œil du lecteur suit naturellement le flux des appels sans avoir besoin de naviguer en arrière et en avant à travers des segments de code disparates.

5.Question

Quels sont les risques de ne pas respecter une norme de formatage commune à l'équipe ?

Réponse:Ne pas maintenir une norme de formatage cohérente au sein d'une équipe peut mener à un code désordonné qui semble déconnecté et chaotique. Cette incohérence complique la collaboration et la compréhension, car des styles différents peuvent perturber les membres de l'équipe lors des revues de code et du développement, augmentant finalement la probabilité de bugs et de malentendus.

6.Question

Comment le formatage horizontal peut-il affecter la compréhension du code ?

Réponse:Le formatage horizontal, qui utilise stratégiquement l'espace blanc, aide à différencier les éléments étroitement liés tout en isolant les composants moins reliés. Par exemple, ajouter des espaces autour des opérateurs souligne leur importance dans les expressions, tandis que maintenir un espacement serré entre les noms de fonction et les paramètres préserve leur association, facilitant une compréhension plus rapide.

7.Question

Que suggère Uncle Bob pour les règles de formatage personnelles ?

Réponse:Uncle Bob préconise des règles de formatage simples et claires, telles que l'indentation cohérente et l'utilisation stratégique de l'espace blanc. Ces règles visent à améliorer la lisibilité, maintenir la fonctionnalité à travers divers environnements, et promouvoir la facilité de compréhension à travers la base de code en éliminant la complexité inutile.

8.Question

Pourquoi est-il recommandé de limiter la largeur des lignes de code ?

Réponse:Limiter la largeur des lignes (idéalement autour de 80-120 caractères) améliore la lisibilité en empêchant le défilement horizontal, ce qui peut entraver la compréhension. Des lignes plus courtes permettent aux développeurs de saisir l'information sans perdre de concentration, favorisant une expérience de lecture plus fluide qui ressemble à la lecture d'un texte en sections confortables.

9.Question

Quelle est l'importance de l'indentation dans le code source ?

Réponse:L'indentation représente visuellement la structure hiérarchique du code, indiquant les portées et les relations. Cela rend le code compréhensible d'un coup d'œil, permettant aux développeurs d'identifier rapidement les classes, les méthodes et les blocs, améliorant ainsi la navigabilité sans nécessiter une attention extensive.

10.Question

Comment les variables d'instance doivent-elles être gérées dans une structure de classe selon les principes de formatage ?

Réponse:Les variables d'instance doivent être déclarées en haut d'une classe pour établir un point de référence clair pour les développeurs. Cette convention minimise la distance verticale entre les concepts liés, permettant un accès et une gestion plus intuitifs de l'état à travers les méthodes de la classe.

Chapitre 5 | 6 Objets et Structures de Données| Questions et réponses

1.Question

Pourquoi gardons-nous nos variables privées dans la programmation orientée objet ?

Réponse:Nous gardons nos variables privées pour éviter que du code externe ne dépende d'elles. Cette encapsulation nous permet de changer le type ou l'implémentation de ces variables sans affecter d'autres parties du code.

2.Question

Quelle est la différence entre exposer l'implémentation via des accesseurs et des mutateurs et utiliser des abstractions ?

Réponse:Exposer l'implémentation par le biais d'accesseurs et de mutateurs dévoile la structure interne des données, rendant les changements plus difficiles à l'avenir. En revanche, utiliser des abstractions permet de masquer les détails d'implémentation et de fournir une interface contrôlée pour l'interaction.

3.Question

En quoi les objets diffèrent-ils des structures de données selon le contenu ?

Réponse:Les objets encapsulent leurs données et offrent des méthodes pour opérer sur ces données, tandis que les structures de données exposent leurs données sans inclure d'opérations significatives. Cette distinction influence la conception de nos systèmes.

4.Question

Quelle est la signification de la Loi de Demeter ?

Réponse:La Loi de Demeter stipule qu'une méthode ne devrait appeler que des méthodes de sa propre classe, des objets qu'elle crée, des arguments passés ou des variables d'instance, favorisant ainsi l'encapsulation et réduisant les dépendances entre les classes.

5.Question

Pouvez-vous expliquer le concept de 'train wrecks' dans le code ?

Réponse:Les 'train wrecks' désignent du code ayant de nombreuses chaînes d'appels de méthodes, rendant la lecture et la maintenance difficiles. Cela indique souvent une violation de la Loi de Demeter, car le code démontre trop de connaissances sur la structure interne de différents objets.

6.Question

Pourquoi l'ajout de nouvelles fonctions peut-il être plus facile dans le code procédural par rapport au code orienté objet ?

Réponse:Dans le code procédural, ajouter de nouvelles fonctions nécessite généralement aucune modification des structures de données existantes, ce qui est simple. En

revanche, l'ajout de nouvelles fonctions dans le code OO nécessite souvent des modifications dans toutes les classes existantes qui pourraient être affectées.

7.Question

Qu'est-ce que les Objets de Transfert de Données (DTO) ?

Réponse:Les DTO sont des classes simples avec des variables publiques et sans comportement, principalement utilisées pour transporter des données. Ils simplifient l'échange de données entre des systèmes comme les bases de données et le code de l'application.

8.Question

Quelle est l'approche idéale en matière de conception lors de l'utilisation des Active Records ?

Réponse:Les Active Records doivent être considérés comme des structures de données, avec des entités séparées gérant la logique métier. Cela évite la confusion créée par le mélange des caractéristiques des structures de données avec des comportements orientés objet.

9.Question

Quelle conclusion pouvons-nous tirer sur l'utilisation des

objets par rapport aux structures de données ?

Réponse:Une bonne conception logicielle nécessite un équilibre entre objets et structures de données. Les objets offrent une flexibilité pour ajouter de nouveaux types sans impacter le comportement, tandis que les structures de données permettent d'ajouter facilement de nouveaux comportements. Les développeurs doivent évaluer les besoins de leur système pour déterminer la meilleure approche.

Chapitre 6 | Gestion des erreurs| Questions et réponses

1.Question

Pourquoi la gestion des erreurs est-elle importante dans le CODER PROPREMENT ?

Réponse:La gestion des erreurs est essentielle dans le CODER PROPREMENT car elle aborde la réalité selon laquelle la programmation implique inévitablement de traiter des erreurs dues à des saisies anormales ou des défaillances d'appareils.

Cependant, une gestion efficace des erreurs ne doit pas obscurcir la logique principale du code ; au contraire, elle doit maintenir la clarté et la séparation par rapport à la logique métier pour améliorer la lisibilité et la maintenabilité.

2.Question

Quels sont les avantages d'utiliser des exceptions plutôt que des codes de retour pour la gestion des erreurs ?

Réponse:L'utilisation d'exceptions conduit à un code d'appel plus propre, non encombré par des vérifications d'erreur. Les exceptions permettent au programmeur de séparer la gestion des erreurs de la logique principale du programme. Cette séparation améliore la compréhension et rend l'algorithme plus visuellement clair.

3.Question

Quel est l'objectif d'écrire d'abord la déclaration try-catch-finally ?

Réponse:Écrire d'abord la déclaration try-catch-finally aide à établir le cadre de gestion des erreurs dès le départ, clarifiant

quelles exceptions peuvent se produire et garantissant une gestion des transactions robuste. Cela définit comment le code doit se comporter face aux échecs et maintient des états cohérents, ce qui est crucial dans les opérations complexes.

4.Question

Comment les exceptions doivent-elles être classées pour une meilleure gestion des erreurs ?

Réponse:Les exceptions doivent être définies en fonction de la manière dont elles seront attrapées par l'appelant plutôt qu'en fonction de leur origine. Cette approche minimise la duplication et permet un code plus propre, car la gestion commune peut être abstraite en un seul type d'exception, simplifiant ainsi le processus de gestion des erreurs.

5.Question

Qu'est-ce que le MODÈLE DE CAS SPÉCIAL, et comment améliore-t-il la clarté du code ?

Réponse:Le MODÈLE DE CAS SPÉCIAL consiste à créer une classe ou un objet qui gère les cas spéciaux, permettant au code principal de rester dégagé de la logique

exceptionnelle. Cette encapsulation simplifie le code en gérant le comportement exceptionnel en interne, rendant la logique principale plus facile à lire et à maintenir.

6.Question

Pourquoi est-il conseillé de ne pas retourner null depuis les méthodes ?

Réponse:Retourner null crée un fardeau de maintenance en introduisant la nécessité de vérifications nulles dans tout le code, augmentant la possibilité d'erreurs d'exécution telles que les NullPointerExceptions. Il est préférable de lancer des exceptions ou de retourner des objets de cas spéciaux, ce qui peut prévenir de telles erreurs et maintenir un code plus propre.

7.Question

Quelles sont les implications de passer des paramètres null aux méthodes ?

Réponse:Passer des paramètres null peut entraîner des exceptions d'exécution inattendues et compliquer la gestion des erreurs. Il est recommandé d'éviter de passer null dans

l'ensemble, car cela indique un potentiel défaut dans le code qui peut entraîner une diminution de la fiabilité. Des assertions peuvent être utilisées pour documenter les attentes, mais interdire null empêche que ces erreurs surviennent en premier lieu.

8.Question

Comment les pratiques de gestion des erreurs peuvent-elles améliorer la maintenance du code ?

Réponse:En considérant la gestion des erreurs comme une préoccupation distincte de la logique métier, les programmeurs peuvent rendre leur code plus maintenable. Une séparation claire permet un raisonnement et une compréhension indépendants de la gestion des erreurs, facilitant une meilleure collaboration entre les équipes et favorisant une culture d'écriture de code robuste et propre.

9.Question

Quelles recommandations le chapitre fait-il pour gérer efficacement les exceptions ?

Réponse:Les recommandations incluent l'utilisation

d'exceptions non vérifiées pour un code plus propre, la fourniture de messages d'erreur informatifs et de contexte, l'encapsulation des API tierces pour minimiser les dépendances, l'utilisation de classes d'exception personnalisées selon les besoins des appelants, et l'évitement de retours et de paramètres null pour améliorer la stabilité du code.

Chapitre 7 | 8 Limites| Questions et réponses

1.Question

Quels sont les défis liés à l'intégration de code tiers dans un système ?

Réponse:L'intégration de code tiers entraîne souvent des tensions entre les fournisseurs d'interfaces et les utilisateurs en raison de besoins divergents. Les fournisseurs visent une applicabilité large, tandis que les utilisateurs recherchent des solutions sur mesure. Cette divergence peut engendrer des problèmes aux limites du système, notamment dans des cas comme l'utilisation de java.util.Map, où l'exposition involontaire de méthodes (comme clear()) pourrait introduire des risques de modifications non souhaitées ou de violations des contraintes de conception.

2.Question

Comment pouvons-nous gérer efficacement les limites dans notre code ?

Réponse:Nous devrions encapsuler les interfaces limites comme Map au sein de classes personnalisées (par exemple, Capteurs), limitant leur exposition à nos applications. Cette approche réduit les abus, permet des modifications plus faciles si l'interface change, et aide à maintenir un code propre et compréhensible en confinant la complexité.

3.Question

Qu'est-ce que les tests d'apprentissage et en quoi bénéficient-ils au développement logiciel ?

Réponse:Les tests d'apprentissage sont des tests exploratoires rédigés pour comprendre les API tierces avant leur intégration dans le code de production. Ils permettent aux développeurs d'expérimenter avec de nouvelles bibliothèques en isolation, consolidant ainsi leur compréhension et leurs attentes en matière de comportement. Ils servent également de filet de sécurité, garantissant que les mises à jour futures de la bibliothèque tierce ne cassent pas la fonctionnalité existante.

4.Question

Quelle stratégie de conception pouvons-nous utiliser face à des interfaces inconnues ou non définies ?

Réponse:Lorsque nous sommes confrontés à des interfaces inconnues, nous pouvons définir nos propres interfaces qui représentent nos interactions souhaitées (par exemple, créer une interface Émetteur en attendant l'API réelle). Cela maintient notre implémentation propre, nous permet de travailler productivement en attendant, et facilite ensuite les tests et l'adaptation lorsque l'interface inconnue devient claire.

5.Question

Pourquoi est-il important d'isoler le code tiers du reste de votre application ?

Réponse:Isoler le code tiers empêche un couplage étroit, rendant le système moins vulnérable aux changements des bibliothèques tierces. Cela favorise la maintenabilité et la lisibilité du code, et permet une adaptation plus facile aux nouvelles versions, car les points d'intégration sont bien définis et limités.

Chapitre 8 | 9 Tests Unitaires| Questions et réponses

1.Question

Quel impact la qualité du code de test a-t-elle sur la qualité du code de production ?

Réponse:La qualité du code de test est cruciale pour maintenir le code de production. Des tests clairs et bien structurés aident à s'assurer que tout changement apporté au code de production peut être vérifié facilement, préservant ainsi la flexibilité, la maintenabilité et la réutilisabilité. En revanche, un code de test mal structuré augmente la probabilité d'échec lors des tests et empêche les développeurs d'apporter les modifications nécessaires, conduisant à une baisse de la qualité globale tant des tests que du code de production.

2.Question

Pourquoi la lisibilité est-elle mise en avant dans les tests unitaires ?

Réponse:La lisibilité est mise en avant dans les tests unitaires

car les tests doivent être facilement compris par tous les membres de l'équipe, y compris ceux qui ne les ont peut-être pas écrits. Les tests doivent clairement exprimer leur intention avec un minimum de distractions causées par des détails inutiles. Cette clarté garantit que quiconque peut rapidement comprendre ce que chaque test vérifie sans avoir à plonger dans une logique complexe.

3.Question

Quelles sont les trois lois du développement piloté par les tests (TDD) discutées dans le Chapitre 8 ?

Réponse:1. Vous ne pouvez pas écrire de code de production tant que vous n'avez pas écrit un test unitaire échoué. 2. Vous ne pouvez pas écrire plus d'un test unitaire que ce qui est suffisant pour échouer, même si cela signifie ne pas compiler. 3. Vous ne pouvez pas écrire plus de code de production que ce qui est suffisant pour faire passer le test actuellement échoué.

4.Question

Quelle erreur courante l'équipe encadrée par l'auteur a-t-elle commise avec son code de test ?

Réponse:L'équipe a décidé que son code de test n'avait pas besoin d'être maintenu aux mêmes normes de qualité que son code de production, adoptant une approche 'rapide et sale'. Cela a conduit à des tests peu soignés qui étaient difficiles à gérer et à maintenir, résultant finalement en un échec de ses efforts de test et une dégradation de la qualité de son code de production.

5.Question

Comment des tests unitaires efficaces peuvent-ils contribuer à la confiance dans les modifications du code de production ?

Réponse:Des tests unitaires efficaces fournissent un filet de sécurité qui permet aux développeurs d'apporter des modifications au code de production sans craindre d'introduire de nouveaux bugs. Lorsque les tests sont complets et clairs, les développeurs peuvent refactoriser et améliorer le code plus librement, en sachant qu'ils peuvent rapidement vérifier que leurs modifications ne cassent pas la fonctionnalité existante.

6.Question

Que signifie l'acronyme F.I.R.S.T. dans le contexte des tests propres ?

Réponse:F.I.R.S.T. signifie : 1. **Rapide** - les tests doivent s'exécuter rapidement. 2. **Indépendant** - les tests ne doivent pas dépendre les uns des autres. 3. **Répétable** - les tests doivent produire les mêmes résultats dans n'importe quel environnement. 4. **Autovalidant** - les tests doivent avoir une indication claire de réussite ou d'échec. 5. **Opportune** - les tests doivent être écrits avant ou en même temps que le code de production.

7.Question

Quel est le bénéfice de créer un langage de test spécifique à un domaine dans les tests unitaires ?

Réponse:Créer un langage de test spécifique à un domaine simplifie le processus d'écriture des tests et améliore la lisibilité. En abstrait les actions courantes dans des fonctions spécialisées, les testeurs peuvent exprimer leur intention plus clairement, permettant à la fois aux rédacteurs et aux lecteurs

des tests de comprendre la logique sous-jacente sans être encombrés de détails inutiles.

8.Question

Quelle est la principale conclusion sur la relation entre le code de test et le code de production ?

Réponse:La principale conclusion est que des tests propres sont tout aussi importants que du code de production propre. Maintenir un haut standard pour le code de test préserve la santé du projet en garantissant que le code de production reste flexible, maintenable et de bonne qualité. Négliger le code de test peut conduire à une détérioration globale du projet.

Chapitre 9 | 10 Classes| Questions et réponses

1.Question

Quel est le principal sujet du Chapitre 10 de 'CODER PROPREMENT'?

Réponse:Le principal sujet du Chapitre 10 concerne l'organisation des classes de manière propre, en mettant l'accent sur l'importance de la taille des

classes, de l'encapsulation, de la cohésion et du Principe de Responsabilité Unique (SRP).

2.Question

Pourquoi les classes devraient-elles être petites selon l'auteur ?

Réponse:Les classes devraient être petites parce qu'elles doivent encapsuler une seule responsabilité, rendant leur maintenance, compréhension et modification plus faciles. Une petite classe réduit la complexité et clarifie sa fonction.

3.Question

Que signifie le terme 'Principe de Responsabilité Unique' (SRP) ?

Réponse:Le Principe de Responsabilité Unique stipule qu'une classe ou un module ne devrait avoir qu'une seule raison de changer. Cela signifie que chaque classe doit se concentrer sur une responsabilité particulière pour améliorer la maintenabilité et réduire le couplage.

4.Question

Comment le nom d'une classe peut-il indiquer un problème de taille ?

Réponse:Si le nom d'une classe est ambigu ou contient des mots vagues comme Processor ou Manager, cela suggère souvent que la classe a trop de responsabilités. Des noms clairs et concis aident à indiquer des responsabilités bien définies et singulières.

5.Question

Quel exemple l'auteur donne-t-il pour démontrer une classe avec trop de responsabilités ?

Réponse:L'auteur donne l'exemple de la classe 'SuperDashboard', qui possède environ 70 méthodes publiques et représente ainsi une 'classe Dieu' — une classe qui combine plusieurs responsabilités, la rendant trop complexe.

6.Question

Que se passe-t-il lorsqu'une classe enfreint le SRP ?

Réponse:Lorsqu'une classe enfreint le SRP, elle devient plus difficile à comprendre et à maintenir. Lorsque des modifications sont nécessaires, le risque de briser les fonctionnalités existantes augmente car la classe gère

plusieurs responsabilités et nécessite des tests approfondis.

7.Question

Quelles stratégies peuvent être utilisées pour améliorer l'organisation et la maintenabilité des classes ?

Réponse:Une stratégie consiste à refactoriser de grandes classes en classes plus petites, à responsabilité unique. Une autre stratégie est d'appliquer le Principe d'Inversion de Dépendance (DIP) en concevant des classes qui dépendent d'abstractions et non d'implémentations concrètes, facilitant ainsi les tests et réduisant le couplage.

8.Question

Comment la promotion d'une forte cohésion au sein des classes bénéficie-t-elle à la conception logicielle ?

Réponse:Une forte cohésion signifie que les méthodes et variables d'une classe travaillent étroitement ensemble, ce qui améliore la clarté de l'objectif de la classe, facilite sa compréhension et son débogage, et réduit la probabilité d'interactions non intentionnelles entre différentes parties du code.

9.Question

En quoi l'exemple des classes SQL restructurées illustre-t-il de bonnes pratiques de conception ?

Réponse:Les classes SQL restructurées illustrent une bonne conception en respectant le Principe Ouvert-Fermé, étant organisées en classes dérivées pour différentes commandes SQL, minimisant les changements et risques, et isolant la fonctionnalité des classes pour améliorer la lisibilité et la maintenabilité.

10.Question

Quelle est la relation entre le maintien d'un faible couplage et l'amélioration de la testabilité dans le code ?

Réponse:Maintenir un faible couplage entre les classes les isole des modifications l'une de l'autre, ce qui simplifie les tests et permet de comprendre les parties du système indépendamment. Cela se traduit par des classes plus faciles à tester et à modifier sans affecter l'ensemble du système.

Scanner pour télécharger

Pourquoi Bookey est une application incontournable pour les amateurs de livres

Contenu de 30min
Plus notre interprétation est profonde et claire, mieux vous saisissez chaque titre.

Format texte et audio
Absorbez des connaissances même dans un temps fragmenté.

Quiz
Vérifiez si vous avez maîtrisé ce que vous venez d'apprendre.

Et plus
Plusieurs voix & polices, Carte mentale, Citations, Clips d'idées...

Essai gratuit avec Bookey

Chapitre 10 | 11 Systèmes| Questions et réponses

1.Question

Quel est le message principal concernant la complexité tel qu'indiqué dans le chapitre ?

Réponse:La complexité est nuisible ; elle complique le travail des développeurs et entraîne des difficultés dans la planification, la construction et les tests des logiciels, tout comme elle affecte la gestion d'une ville.

2.Question

Comment le chapitre suggère-t-il de gérer efficacement les systèmes logiciels ?

Réponse:En organisant des équipes pour traiter des préoccupations distinctes, tout comme on gère différents aspects d'une ville, et en séparant la construction des systèmes de leur utilisation.

3.Question

Que souligne le chapitre à propos de la relation entre construction et utilisation dans les systèmes logiciels ?

Réponse:Il est essentiel de distinguer la phase de

construction, où les dépendances sont établies, et la phase d'utilisation, où la logique de l'application s'exécute. Les mélanger entraîne des complications.

4.Question

Expliquez l'idiome de l'initialisation paresseuse et un de ses inconvénients tel que présenté dans ce chapitre.

Réponse:L'initialisation paresseuse permet de retarder la construction d'un objet jusqu'à ce qu'il soit nécessaire, ce qui peut améliorer les performances. Cependant, cela crée des dépendances codées en dur qui compliquent les tests et violent le principe de responsabilité unique.

5.Question

Que signifie 'Modulariser le processus de démarrage' dans le chapitre ?

Réponse:Cela fait référence à l'isolement de la construction et de l'initialisation des objets d'application de la logique principale de l'application, permettant une architecture système plus claire et plus maintenable.

6.Question

Quel rôle joue l'injection de dépendance (DI) dans

l'architecture logicielle selon le chapitre ?

Réponse:L'injection de dépendance aide à séparer la construction des objets dépendants de leur utilisation, en respectant le principe de responsabilité unique, ce qui permet une meilleure gestion et un meilleur test du logiciel.

7.Question

Comment le chapitre lie-t-il la croissance des systèmes au développement urbain ?

Réponse:Tout comme les villes évoluent des petits villages à des systèmes complexes au fil du temps, les systèmes logiciels devraient croître progressivement avec une séparation appropriée des préoccupations, plutôt que de viser un design parfait dès le départ.

8.Question

Que signifie optimiser la prise de décision dans les systèmes logiciels ?

Réponse:En modulant les préoccupations et en maintenant une séparation claire, le chapitre préconise de différer les décisions jusqu'à ce que les informations nécessaires soient

disponibles, conduisant à des choix mieux éclairés.

9.Question

Quelle est l'importance des langages spécifiques au domaine (DSL) dans le développement logiciel ?

Réponse:Les DSL comblent le fossé entre les concepts de domaine et l'implémentation, permettant aux développeurs d'écrire du code qui communique clairement la logique du domaine et réduit le risque de mauvaise interprétation.

10.Question

Pourquoi l'accent sur la simplicité est-il souligné lors de la conception des systèmes ?

Réponse:La simplicité permet une compréhension et une maintenance plus faciles du code, réduisant la complexité qui peut entraîner des bogues et obscurcir la logique du domaine, soutenant finalement le développement agile.

Chapitre 11 | 12 Émergence| Questions et réponses

1.Question

Quelles sont les quatre règles simples qui peuvent aider à créer de bons designs logiciels ?

Réponse:1. Passe tous les tests

2. Ne contient aucune duplication

3. Exprime l'intention du programmeur

4. Minimise le nombre de classes et de méthodes

2.Question

Pourquoi est-il important qu'un design passe tous les tests ?

Réponse:Un design doit produire un système qui fonctionne comme prévu. S'il ne peut pas être vérifié par des tests, son intégrité est discutable. Les systèmes testables permettent de petites classes à but unique, facilitant ainsi le respect du principe de responsabilité unique (SRP).

3.Question

Comment le fait d'écrire des tests conduit-il à de meilleurs designs logiciels ?

Réponse:Écrire des tests incite à un code moins couplé, car les tests sont plus faciles à écrire pour des classes bien structurées. Une attention accrue portée aux tests aligne naturellement l'architecture sur des objectifs de design orienté objet tels que le faible couplage et la haute cohésion.

4.Question

Comment vous assurez-vous que votre code reste propre après avoir ajouté de nouvelles lignes ?

Réponse:Après avoir ajouté quelques lignes de code, faites une pause et réfléchissez au design. Si vous constatez que vous avez dégradé la structure, nettoyez-la en la refactorisant tout en exécutant des tests pour vous assurer que rien n'est cassé.

5.Question

Quel est l'ennemi principal d'un système bien conçu ?

Réponse:La duplication.

6.Question

Pouvez-vous expliquer comment la duplication se manifeste dans le code logiciel ?

Réponse:La duplication peut apparaître sous la forme de lignes de code identiques, de lignes de code similaires qui peuvent être refactorisées pour se ressembler, ou dans des méthodes d'implémentation qui effectuent des fonctions similaires mais sont écrites séparément.

7.Question

Comment pouvez-vous éliminer la duplication dans les méthodes ?

Réponse:En extrayant la fonctionnalité commune dans une méthode partagée. Par exemple, dans les méthodes de mise à l'échelle et de rotation d'image, extrayez la logique de remplacement d'image dans une méthode `remplacerImage` pour réduire la répétition.

8.Question

Quels sont les avantages d'avoir un code expressif ?

Réponse:Un code expressif transmet clairement l'intention du programmeur, facilitant la compréhension du système pour les autres (y compris les futurs mainteneurs), réduisant ainsi les erreurs et abaissant les coûts de maintenance.

9.Question

Quel rôle jouent de bons noms dans l'expressivité du code ?

Réponse:De bons noms fournissent un aperçu immédiat des responsabilités des classes et des fonctions, permettant aux autres de comprendre leur objectif sans avoir à plonger

profondément dans le code.

10.Question

Pourquoi est-il essentiel de garder le nombre de classes et de méthodes faible ?

Réponse:Bien qu'il soit important de maintenir de petites classes et méthodes pour la clarté, en créer trop peut entraîner une complexité inutile. Au lieu de cela, visez à garder l'ensemble du système petit tout en équilibrant la propreté procédurale.

11.Question

Quel principe général les pratiques décrites dans ce chapitre servent-elles ?

Réponse:Elles cristallisent des décennies d'expérience en règles et directives pratiques qui aident les développeurs à adhérer aux bons principes et motifs de conception.

Chapitre 12 | 13 Concurrence| Questions et réponses

1.Question

Quelles sont les principales raisons d'adopter la concurrence en programmation ?

Réponse:La concurrence permet de découpler les

tâches de leur timing, améliorant ainsi le débit et la structure du système. Elle favorise une meilleure gestion des ressources en réponse aux contraintes de temps et permet des applications plus réactives en gérant plusieurs tâches en parallèle.

2.Question

Quels sont les idées reçues courantes concernant les améliorations liées à la concurrence ?

Réponse:Un mythe majeur est que la concurrence améliore toujours la performance ; cependant, cela n'est vrai que lorsqu'il y a un temps d'attente significatif pouvant être partagé entre les threads. Une autre idée reçue est que concevoir des systèmes concurrents ne diffère pas significativement de celui des systèmes à thread unique.

3.Question

Pourquoi gérer les données partagées est-il un défi majeur en programmation concurrente ?

Réponse:Lorsque plusieurs threads tentent d'accéder et de modifier des données partagées, ils peuvent interférer les uns

avec les autres, entraînant des résultats incohérents. Cela nécessite une gestion rigoureuse de la synchronisation, rendant la programmation concurrente de plus en plus complexe.

4.Question

Comment le principe de responsabilité unique (SRP) peut-il s'appliquer à la conception de la concurrence ?

Réponse:Le SRP suggère que le code ne devrait avoir qu'une seule raison de changer. Dans le contexte de la concurrence, cela signifie garder le code lié à la concurrence séparé du code non concurrent pour gérer la complexité et faciliter le débogage.

5.Question

Quelles sont quelques stratégies pour minimiser les erreurs liées à la concurrence ?

Réponse:Vous pouvez limiter la portée des données partagées, utiliser des copies de données autant que possible et vous assurer que les threads fonctionnent de manière aussi indépendante que possible. Ces stratégies réduisent la

probabilité d'interactions non intentionnelles entre les threads.

6.Question

Quelle est l'importance de tester le code concurrent et quels sont les défis spécifiques que cela présente ?

Réponse:Tester le code concurrent est vital car prouver sa correction peut être extrêmement difficile ; les bugs peuvent ne se manifester que dans des conditions spécifiques, entraînant des échecs intermittents. Par conséquent, les tests doivent être approfondis et couvrir diverses configurations et environnements.

7.Question

Quelles sont des méthodes efficaces pour révéler les problèmes de concurrence cachés pendant les tests ?

Réponse:L'utilisation de stratégies telles que l'instrumentation du code, qui force différents chemins d'exécution, peut grandement améliorer la détection de bugs de concurrence rares. De plus, exécuter des tests sur diverses plateformes et configurations peut aider à identifier des

problèmes ne se produisant que dans des environnements spécifiques.

8.Question

Sur quoi un développeur doit-il se concentrer lors de l'écriture de code concurrent ?

Réponse:Les développeurs devraient suivre les principes du CODER PROPREMENT, séparer les préoccupations, minimiser l'état partagé et tester rigoureusement leur code. Ils doivent anticiper les problèmes potentiels de concurrence tel que le blocage et la famine lors de la conception de leurs systèmes.

Scanner pour télécharger

App Store
Coup de cœur

22k avis 5 étoiles

Retour Positif

Fabienne Moreau

ue résumé de livre ne testent
ion, mais rendent également
musant et engageant.
té la lecture pour moi.

Fantastique! Giselle Dubois

Je suis émerveillé par la variété de livres et de langues que Bookey supporte. Ce n'est pas juste une application, c'est une porte d'accès au savoir mondial. De plus, gagner des points pour la charité est un grand plus !

Fi
Le
liv
co
pr

é Blanchet

de lecture
eption de
es,
tous.

J'adore ! Isoline Mercier

Bookey m'offre le temps de parcourir les parties importantes d'un livre. Cela me donne aussi une idée suffisante pour savoir si je devrais acheter ou non la version complète du livre ! C'est facile à utiliser !"

Gain de temps !

Bookey est mon applicat
intellectuelle. Les résum
magnifiquement organis
monde de connaissance

Appli géniale ! Joachim Lefevre

adore les livres audio mais je n'ai pas toujours le temps
'écouter le livre entier ! Bookey me permet d'obtenir
n résumé des points forts du livre qui m'intéresse !!!
Quel super concept !!! Hautement recommandé !

Appli magnifique

Cette application est une bouée de sauve
amateurs de livres avec des emplois du te
Les résumés sont précis, et les cartes me
renforcer ce que j'ai appris. Hautement re

Essai gratuit avec Bookey

Chapitre 13 | 14 Raffinement Successif| Questions et réponses

1.Question

Quel est le thème principal abordé dans le chapitre sur la classe Args ?

Réponse:Le chapitre souligne la nécessité d'un perfectionnement successif dans le développement de code, montrant comment transformer une base de code désordonnée et mal structurée en un code propre et maintenable.

2.Question

Comment peut-on efficacement passer d'un brouillon de code à une implémentation propre ?

Réponse:Le processus consiste à écrire une version initiale du code, à identifier les zones de désordre, et à le refactoriser progressivement par de petits changements incrémentiels tout en s'assurant que le code reste fonctionnel après chaque modification.

3.Question

Pourquoi est-il décrit comme un suicide professionnel de

laisser du code dans un état brut ?

Réponse:Laisser du code dans un état brut peut entraîner une augmentation des taux de bogues, des coûts de maintenance plus élevés, et une complexité qui peut entraver le développement ultérieur, causant finalement des retards et des complications dans le projet.

4.Question

Quel rôle joue le développement piloté par les tests (TDD) dans le processus de refactorisation selon le chapitre ?

Réponse:Le TDD garantit que les modifications apportées pendant le processus de refactorisation ne cassent pas la fonctionnalité existante. Cela implique d'écrire des tests avant les changements de code, permettant aux développeurs de peaufiner leur code tout en maintenant la confiance dans sa correction.

5.Question

Quels défis se posent lors de l'ajout de nouvelles fonctionnalités à une base de code, comme souligné dans le chapitre ?

Réponse:Ajouter de nouvelles fonctionnalités peut compliquer une base de code déjà désordonnée, entraînant des difficultés accrues en matière de débogage et de maintenance. Cela se manifeste dans la façon dont l'ajout de deux nouveaux types d'arguments a transformé un système gérable en un réseau complexe d'interactions et de dépendances.

6.Question

Comment la complexité du code peut-elle se détériorer avec le temps, et quelles stratégies peuvent combattre cela ?

Réponse:La complexité du code peut empirer à mesure que des fonctionnalités sont ajoutées sans maintenance, entraînant des dépendances entrelacées. Pour y remédier, des révisions de code continues, des pratiques de codage propre, et une refactorisation régulière devraient être employées pour garder la base de code gérable.

7.Question

Quel est l'impact à long terme de la négligence de la qualité du code dans les projets logiciels ?

Réponse:Négliger la qualité du code peut conduire à un 'marécage malin' de code qui ralentit le développement, augmente la difficulté de mise en œuvre de futurs changements, et finit par entraver la productivité de l'équipe.

8.Question

Pourquoi est-il plus facile de maintenir du code propre par rapport à du code 'pourrissant' ?

Réponse:Le code propre est plus simple et plus compréhensible, permettant aux développeurs de résoudre rapidement et de corriger les problèmes, tandis que de nouveaux changements peuvent être facilement intégrés sans introduire plus de complexité. En revanche, le code pourrissant devient un enchevêtrement qui complique la gestion des dépendances et des fonctionnalités.

9.Question

Quelle leçon fondamentale sur la programmation l'auteur transmet-il à travers l'exemple de la classe Args ?

Réponse:La programmation est un métier qui nécessite un perfectionnement itératif. Le CODER PROPREMENT ne

surgit pas d'un premier brouillon, mais se cultive à travers des ajustements continus et un engagement à maintenir des normes élevées tout au long du processus de développement.

10.Question

Quelle est l'importance d'une nomination claire et de la taille des fonctions dans le contexte de la lisibilité et de la maintenance du code ?

Réponse:Une convention de nommage claire et des fonctions de taille appropriée améliorent considérablement la lisibilité et la compréhension du code, permettant aux développeurs de saisir rapidement la fonctionnalité et rendant le code plus facile à maintenir et à faire évoluer.

Chapitre 14 | 15 Les Internes de JUnit| Questions et réponses

1.Question

Quelle est l'importance du ComparisonCompactor dans le framework JUnit ?

Réponse:Le ComparisonCompactor est essentiel dans le framework JUnit car il aide à identifier les divergences entre deux chaînes de manière claire et

informative. En générant une sortie formatée qui met en évidence les différences entre les chaînes, il améliore le processus de débogage pour les développeurs, leur permettant de localiser rapidement les erreurs dans leurs tests.

2.Question

Comment Kent Beck et Eric Gamma ont-ils développé JUnit ?

Réponse:Kent Beck et Eric Gamma ont développé JUnit lors d'un vol où Kent voulait apprendre Java et Eric s'intéressait au framework de test Smalltalk de Kent. Leur collaboration a abouti à un framework de test simple mais élégant qui a posé les bases de nombreuses pratiques de test en Java.

3.Question

Quelles améliorations ont été suggérées pour le code du ComparisonCompactor ?

Réponse:Les améliorations incluaient le renommage des variables membres pour supprimer les préfixes inutiles, l'encapsulation des vérifications conditionnelles pour plus de

clarté, la simplification des méthodes pour améliorer la lisibilité et l'assurance que les fonctions reflètent bien leurs objectifs. Ces changements contribuent à rendre le code plus propre et plus facile à maintenir.

4.Question

À quoi fait référence la 'règle du scout' dans le contexte du développement logiciel ?

Réponse:La règle du scout dans le développement logiciel encourage les développeurs à laisser la base de code plus propre qu'ils ne l'ont trouvée. Ce principe prône le refactoring continu et l'amélioration du code existant, garantissant qu'avec chaque contribution, la qualité de la base de code s'améliore.

5.Question

Pourquoi l'analyse de la couverture des tests est-elle importante pour les tests de ComparisonCompactor ?

Réponse:L'analyse de la couverture des tests est importante pour les tests de ComparisonCompactor car elle fournit des informations sur la quantité de code exercée par les tests.

Atteindre une couverture de 100 % indique que toutes les lignes et branches ont été testées, augmentant ainsi la confiance dans la fonctionnalité et la robustesse du code.

6.Question

Quel rôle joue le refactoring dans le développement logiciel ?

Réponse:Le refactoring est un processus critique dans le développement logiciel qui consiste à restructurer le code existant sans changer son comportement externe. Il vise à améliorer la lisibilité du code, à réduire la complexité et à améliorer la maintenabilité, menant finalement à des logiciels plus efficaces et sans erreurs.

7.Question

Comment la version finale du ComparisonCompactor reflète-t-elle les principes de CODER PROPREMENT ?

Réponse:La version finale du ComparisonCompactor incarne les principes de CODER PROPREMENT en séparant clairement les préoccupations entre les fonctions d'analyse et de synthèse, en utilisant des conventions de nommage

descriptives et en maintenant un flux logique qui améliore la lisibilité. Elle démontre des améliorations itératives grâce au refactoring, résultant en une structure de code plus propre et mieux organisée.

8.Question

Que peuvent apprendre les développeurs du passage sur la structure des cas de test ?

Réponse:Les développeurs peuvent apprendre l'importance de la simplicité et de la clarté dans la structuration des cas de test. Les tests bien structurés sont non seulement plus faciles à lire et à comprendre, mais facilitent également la maintenance adéquate et encouragent les meilleures pratiques en matière de test.

9.Question

Comment l'auteur a-t-il démontré la nature itérative du refactoring dans le chapitre ?

Réponse:L'auteur a illustré la nature itérative du refactoring en montrant comment les décisions de refactoring antérieures ont été revisitées et modifiées en fonction d'aperçus

ultérieurs. Cela reflète la nature non linéaire et exploratoire de l'amélioration du code, soulignant que les refactors initiaux peuvent mener à de nouvelles perspectives nécessitant d'autres changements.

Chapitre 15 | 16 Refactorisation de SerialDate| Questions et réponses

1.Question

Pourquoi est-il important de critiquer le code, même lorsqu'il semble 'bon' ?

Réponse:Critiquer le code, même s'il est jugé 'bon', est essentiel pour l'amélioration continue et l'apprentissage. Grâce à ces évaluations, les programmeurs peuvent découvrir des problèmes cachés, améliorer la fonctionnalité et renforcer leur compréhension des meilleures pratiques. Comme le souligne le texte, c'est une pratique standard dans des professions telles que la médecine et l'aviation, favorisant une culture d'excellence et de responsabilité.

2.Question

Que signifie le terme 'refactoring' dans le contexte de la programmation ?

Réponse:Le refactoring se réfère au processus de restructuration du code existant sans changer son comportement externe. Il vise à améliorer la structure, la lisibilité et la maintenabilité du code. Le chapitre souligne qu'à travers un refactoring systématique, on peut CODER PROPREMENT, corriger des bogues et améliorer la couverture de test, rendant finalement le code plus facile à travailler pour les développeurs futurs.

3.Question

Comment l'utilisation des énumérations peut-elle améliorer la clarté du code en programmation ?

Réponse:Les énumérations fournissent un ensemble clair et défini de constantes, rendant le code plus lisible et réduisant les erreurs associées à l'utilisation d'entiers bruts. Dans le cas de la gestion des dates, passer d'identifiants basés sur des entiers à des énumérations comme Mois et Jour rend le code auto-documenté, car il communique l'intention sans

nécessiter de commentaires supplémentaires.

4.Question

Pourquoi devrions-nous minimiser la redondance dans les commentaires de code ?

Réponse:Les commentaires redondants peuvent entraîner de la désinformation et du désordre, obscurcissant la clarté du code. Un point important du texte est que le code lui-même devrait être suffisamment clair pour minimiser le besoin de commentaires. Lorsque des commentaires sont nécessaires, ils doivent apporter une véritable valeur ajoutée, comme des explications de logiques complexes qui ne sont pas immédiatement apparentes.

5.Question

Quel rôle joue le test unitaire dans le processus de refactoring ?

Réponse:Le test unitaire est crucial lors du refactoring car il garantit que les modifications ne cassent pas la fonctionnalité existante. Il fournit un filet de sécurité qui permet aux programmeurs d'apporter des améliorations et des

optimisations en toute confiance, rassurés que les régressions peuvent être immédiatement identifiées. Ce chapitre a souligné l'importance d'atteindre une couverture de test complète en tant que prérequis pour un refactoring sûr.

6.Question

Comment le chapitre illustre-t-il le concept de 'la règle du Boy Scout' ?

Réponse:La règle du Boy Scout suggère que les programmeurs devraient laisser le code plus propre qu'ils ne l'ont trouvé. Le chapitre illustre ce principe en s'attaquant systématiquement à divers aspects de la classe SerialDate, améliorant sa structure, augmentant sa lisibilité, corrigeant des bogues et augmentant la couverture de test, contribuant ainsi à une base de code plus saine pour les développeurs futurs.

7.Question

Quelle est la signification des métriques de couverture de test comme celles rapportées par Clover ?

Réponse:Les métriques de couverture de test, telles que

celles rapportées par Clover, fournissent des informations sur la quantité de code testée par les tests, indiquant le risque potentiel de bogues non détectés. Un pourcentage de couverture plus élevé suggère généralement une meilleure confiance dans la fiabilité du code. Le chapitre décrit comment une compréhension approfondie de la couverture peut guider les efforts de refactoring en soulignant les chemins non testés qui nécessitent une attention particulière.

8.Question

Pourquoi est-il problématique que les classes de base soient conscientes de leurs sous-classes, selon le texte ?

Réponse:Lorsque les classes de base sont conscientes de leurs sous-classes, cela peut créer un couplage étroit, limitant la flexibilité et rendant le code plus difficile à maintenir ou à étendre. Le texte plaide pour l'utilisation de modèles de conception, comme le modèle de Fabrique Abstraite, pour isoler les classes de base des détails d'implémentation spécifiques, permettant des conceptions plus propres et modulaires.

9.Question

Que suggère l'auteur sur le nommage des classes et des méthodes lors du refactoring ?

Réponse:L'auteur souligne que le nommage doit être clair et intuitif, reflétant avec précision le but et le comportement des classes et des méthodes. Un bon nommage aide à comprendre la fonctionnalité du code sans nécessiter de documentation extensive, promouvant le code auto-documenté comme un objectif lors des efforts de refactoring.

10.Question

Comment le processus de refactoring a-t-il contribué à améliorer spécifiquement la classe SerialDate ?

Réponse:Le processus de refactoring pour la classe SerialDate a impliqué d'améliorer sa structure, de corriger des erreurs de conditions aux limites, d'augmenter la couverture de test d'environ 50 % à environ 85 %, d'éliminer le code non utilisé et de renommer les méthodes pour plus de clarté. Ces changements ont collectivement amélioré

l'intégrité, l'efficacité et la lisibilité du code, le rendant plus robuste et plus facile à travailler pour les développeurs.

Lire, Partager, Autonomiser

Terminez votre défi de lecture, faites don de livres aux enfants africains.

Le Concept

BOOKS FOR AFRICA × ×

Cette activité de don de livres se déroule en partenariat avec Books For Africa. Nous lançons ce projet car nous partageons la même conviction que BFA : Pour de nombreux enfants en Afrique, le don de livres est véritablement un don d'espoir.

La Règle

Gagnez 100 points → **Échangez un livre** → **Faites un don à l'Afrique**

Votre apprentissage ne vous apporte pas seulement des connaissances mais vous permet également de gagner des points pour des causes caritatives ! Pour chaque 100 points gagnés, un livre sera donné à l'Afrique.

Chapitre 16 | 17 Odeurs et Heuristiques| Questions et réponses

1.Question

Quelle est l'importance d'identifier les odeurs de code dans un logiciel ?

Réponse:Identifier les odeurs de code est essentiel pour maintenir un CODER PROPREMENT, car elles servent d'indicateurs des zones problématiques dans la base de code qui peuvent conduire à des bugs et des défauts de conception. S'attaquer à ces odeurs aide à améliorer la qualité, la maintenabilité et la lisibilité du code, ce qui conduit à un développement plus efficace et à une collaboration plus facile.

2.Question

Pourquoi les commentaires dans le code doivent-ils se limiter aux notes techniques et à la conception ?

Réponse:Les commentaires doivent se concentrer sur les notes techniques et les aspects de conception car ils visent à fournir une clarté sur l'intention et la fonctionnalité du code,

plutôt que de stocker des données historiques ou des informations redondantes qui peuvent encombrer le code et entraîner de la confusion au fil du temps.

3.Question

Quel est le danger de maintenir du code commenté ?

Réponse:Le code commenté peut devenir obsolète et déroutant avec le temps, car il devient souvent désuet et déconnecté du contexte actuel du code. Cela peut induire les développeurs en erreur en pensant que l'ancienne fonctionnalité est encore nécessaire, ce qui entraîne un encombrement accru et augmente la charge cognitive lors de la lecture du code.

4.Question

Comment les dépendances entre les modules doivent-elles être gérées selon les principes du CODER PROPREMENT ?

Réponse:Les dépendances entre les modules doivent être explicites et physiques plutôt que logiques. Cela signifie que les modules doivent appeler les méthodes et les données

spécifiques dont ils ont besoin auprès de leurs collaborateurs plutôt que de faire des hypothèses sur leur disponibilité, réduisant ainsi les dépendances cachées et promouvant des relations plus claires.

5.Question

Pourquoi est-il important de limiter le nombre d'arguments qu'une fonction peut prendre ?

Réponse:Limiter le nombre d'arguments garde les fonctions plus simples et plus faciles à comprendre. Idéalement, une fonction ne doit pas avoir plus de trois arguments, car des paramètres excessifs peuvent entraîner de la confusion et rendre la fonction plus difficile à utiliser et à mémoriser.

6.Question

Qu'est-ce que le principe de la moindre surprise et comment s'applique-t-il au CODER PROPREMENT ?

Réponse:Le principe de la moindre surprise stipule que le code doit se comporter d'une manière prévisible pour le lecteur ou l'utilisateur, en fonction de leurs attentes. Ce principe améliore la lisibilité et la maintenabilité en

garantissant que les fonctions et les classes fonctionnent de manière conforme aux pratiques communes et aux intuitions.

7.Question

Que signifie le principe DRY et pourquoi est-il crucial dans le codage ?

Réponse:Le principe DRY signifie 'Don't Repeat Yourself' (Ne vous répétez pas). Il est crucial car la duplication de code augmente la charge de maintenance et le potentiel d'erreurs. En évitant la duplication, les développeurs peuvent créer un code plus modulaire et adaptable qui peut être réutilisé sans modification.

8.Question

Que faire du code mort dans un programme ?

Réponse:Le code mort, qui est du code qui n'est jamais exécuté, doit être complètement retiré de la base de code. Le garder ajoute un encombrement inutile et peut confondre les futurs développeurs sur la fonctionnalité prévue du module.

9.Question

Pourquoi les fonctions devraient-elles être conçues pour ne faire qu'une seule chose ?

Réponse:Les fonctions devraient se concentrer sur une seule tâche pour renforcer la clarté et la réutilisabilité. Une fonction à responsabilité unique est plus facile à tester, déboguer et comprendre, ce qui favorise une meilleure organisation et maintenabilité du code.

10.Question

Que signifie encapsuler les conditions limites dans le code ?

Réponse:Encapsuler les conditions limites signifie isoler la logique de traitement des cas particuliers en un seul endroit plutôt que de disperser des vérifications dans toute la base de code. Cette approche simplifie la maintenance et réduit la probabilité d'erreurs lorsque les conditions limites sont modifiées ou doivent être mises à jour.

11.Question

Quel rôle jouent des noms de variables explicites dans la lisibilité du code ?

Réponse:Des noms de variables explicites apportent une clarté sur ce que les variables représentent, rendant le code

plus compréhensible d'un coup d'œil. Cela réduit le besoin de commentaires détaillés et aide les futurs lecteurs du code à saisir rapidement son but et sa logique.

12.Question

Comment l'utilisation de conventions de nommage standard contribue-t-elle au CODER PROPREMENT ?

Réponse:L'utilisation de conventions de nommage standard crée un sentiment de familiarité et de prévisibilité, rendant la base de code plus facile à naviguer pour les développeurs. Lorsque le nommage suit des modèles établis, cela réduit l'ambiguïté et aide à la compréhension, favorisant ainsi la maintenabilité.

13.Question

Pourquoi est-il important de garder l'interface de tout module ou classe petite ?

Réponse:Une petite interface limite le nombre de méthodes exposées aux autres parties du code, diminuant ainsi le couplage et rendant le module plus facile à utiliser et à comprendre. Cela réduit les risques de conséquences

imprévues des modifications et favorise des interactions plus stables.

14.Question

Que signifie le conseil de 'rendre les dépendances logiques physiques' en ce qui concerne la structure du code ?

Réponse:Cela signifie que chaque module doit déclarer explicitement ses dépendances, plutôt que de faire des hypothèses sur ce que d'autres modules fournissent. Cette approche améliore la clarté, diminue les dépendances cachées et facilite la maintenance et le refactoring.

Chapitre 17 | A : Concurrence II| Questions et réponses

1.Question

Quel est l'avantage principal d'utiliser des threads dans une application client/serveur ?

Réponse:L'utilisation de threads permet au serveur de gérer plusieurs requêtes de clients en parallèle, ce qui peut améliorer considérablement le débit, notamment lors des opérations liées aux entrées/sorties. Pendant qu'un thread attend

l'achèvement des opérations d'E/S, un autre thread peut traiter une autre requête de client, optimisant ainsi l'utilisation du processeur.

2.Question

Quelles sont les principales considérations de performance lors du test d'une application client/serveur ?

Réponse:La performance doit être validée par des tests qui vérifient les temps d'achèvement des requêtes clients, par exemple en s'assurant que toutes les requêtes se terminent dans un délai donné. De plus, il est essentiel de comprendre si l'application est liée aux E/S ou au processeur, car cela détermine comment améliorer la performance.

3.Question

Pourquoi est-il problématique de ne pas avoir de limite sur le nombre de threads créés par un serveur ?

Réponse:Sans limites, un serveur peut générer trop de threads, ce qui peut épuiser les ressources système et dégrader la performance. Les systèmes conçus pour de nombreux utilisateurs doivent gérer l'utilisation des threads

par le biais d'une politique de gestion des threads contrôlée, évitant ainsi la surcharge du serveur.

4.Question

Comment le 'Principe de Responsabilité Unique' peut-il être appliqué pour améliorer la conception du serveur dans un contexte multithreadé ?

Réponse:En séparant les préoccupations, telles que la gestion des connexions de sockets, le traitement des clients et la planification des threads, les développeurs peuvent créer une base de code plus propre et plus facile à maintenir. Cela permet des modifications plus simples dans des domaines spécifiques (comme la politique de threading) sans affecter des parties non liées du code.

5.Question

Quelle est une cause courante de blocage dans les applications multithreadées, et comment peut-on l'éviter ?

Réponse:Le blocage se produit lorsque des threads sont coincés en attente de ressources détenues les uns par les autres. On peut y remédier en garantissant un ordre cohérent

d'acquisition des ressources parmi tous les threads, en utilisant des délais d'attente, ou en appliquant des mécanismes de verrouillage appropriés qui minimisent la possibilité de conditions d'attente circulaires.

6.Question

Quelles stratégies de test peuvent révéler des problèmes de threading dans le code ?

Réponse:Des stratégies de test telles que les tests de Monte Carlo (exécution répétée de tests dans des conditions variées) et l'assurance d'exécuter des tests sur différents matériels et charges peuvent augmenter les chances de déceler des problèmes de threading. De plus, l'utilisation d'outils spécialisés (comme ConTest) peut améliorer la détection de bogues concurrentiels.

7.Question

Quel est l'effet de ne pas utiliser de synchronisation lors de l'accès à un état mutable partagé ?

Réponse:Ne pas utiliser de synchronisation peut entraîner des conditions de course où plusieurs threads accèdent et

modifient des données partagées sans coordination, entraînant un comportement de programme incohérent ou inattendu. Dans un cas spécifique, plusieurs threads pourraient écraser des modifications, conduisant à une perte de données ou à des résultats incorrects.

8.Question

Comment le framework Executor de Java améliore-t-il la gestion des threads ?

Réponse:Le framework Executor de Java simplifie la gestion des threads en permettant aux développeurs de créer un pool de threads qui gère efficacement l'exécution des tâches. Il prévient la surcharge de la création et de la destruction fréquentes de threads et aide à gérer des volumes plus importants de tâches concurrentes plus gracieusement.

9.Question

Qu'est-ce que les opérations atomiques et pourquoi sont-elles critiques en programmation concurrente ?

Réponse:Les opérations atomiques sont des opérations qui se terminent en une seule étape du point de vue des autres

threads, ce qui signifie qu'elles ne peuvent pas être interrompues. Comprendre les opérations atomiques est crucial en programmation concurrente pour éviter l'incohérence des données causée par des modifications simultanées.

10.Question

Comment l'exemple démontre-t-il les différences entre les tâches liées aux E/S et celles liées au CPU dans un environnement concurrent ?

Réponse:L'exemple met en contraste les tâches liées aux E/S, où les temps d'attente pour les ressources externes peuvent être superposés aux temps de traitement d'autres tâches, avec les tâches liées au CPU, où des threads supplémentaires n'augmentent pas la performance en raison de la limite de traitement par le CPU.

Chapitre 18 | B: org.jfree.date.SerialDate| Questions et réponses

1.Question

Quel est le but principal de la classe SerialDate dans la manipulation des dates ?

Réponse:La classe SerialDate est conçue pour fournir une représentation simple et immuable des dates, facilitant leur manipulation sans nécessiter la précision d'une java.util.Date, souvent trop détaillée pour de nombreuses applications. Elle permet aux utilisateurs de travailler avec des dates représentant des journées entières tout en abstraisant les détails d'implémentation.

2.Question

Pourquoi l'utilisation de java.util.Date peut-elle parfois être trop complexe ?

Réponse:java.util.Date offre un haut niveau de précision, jusqu'aux millisecondes, ce qui est superflu lorsque l'objectif est simplement de représenter un jour particulier. Les utilisateurs peuvent se retrouver à gérer des complexités liées aux fuseaux horaires et à l'heure de la journée alors qu'ils s'intéressent uniquement à la date elle-même.

3.Question

Comment la classe SerialDate maintient-elle la compatibilité avec le système de dates d'Excel ?

Réponse:La classe SerialDate maintient la compatibilité avec Excel en utilisant un système de numérotation similaire où le numéro de série pour les dates commence le 1er janvier 1900. Elle reconnaît un bug délibéré dans Excel qui considère incorrectement 1900 comme une année bissextile, permettant ainsi une cohérence dans les calculs de dates avec des applications comme Excel.

4.Question

Quelles considérations sont prises en compte dans la conception de la classe SerialDate pour éviter les erreurs ?

Réponse:La classe SerialDate intègre des validations pour garantir que les dates se situent dans une plage définie (de 1900 à 9999) et tenait compte des années bissextiles. Elle met également en œuvre des méthodes pour convertir entre différentes représentations de dates et fournir des utilitaires utiles, comme la vérification des codes valides pour les mois et les jours de la semaine.

5.Question

Pourquoi l'immuabilité est-elle une caractéristique

significative de la classe SerialDate ?

Réponse:L'immuabilité garantit que, une fois qu'une instance de SerialDate est créée, elle ne peut pas être modifiée. Cette caractéristique empêche les modifications accidentelles des données, rendant la classe plus sûre et plus prévisible à utiliser. Elle s'aligne bien avec les principes de la programmation fonctionnelle et renforce la fiabilité des manipulations de dates.

6.Question

Comment le constructeur de la classe SerialDate gère-t-il les entrées de date invalides ?

Réponse:Le constructeur de SerialDate lance une IllegalArgumentException lorsque des dates invalides sont fournies, garantissant que seules des dates valides sont acceptées et que l'intégrité des objets date est maintenue tout au long de leur utilisation.

7.Question

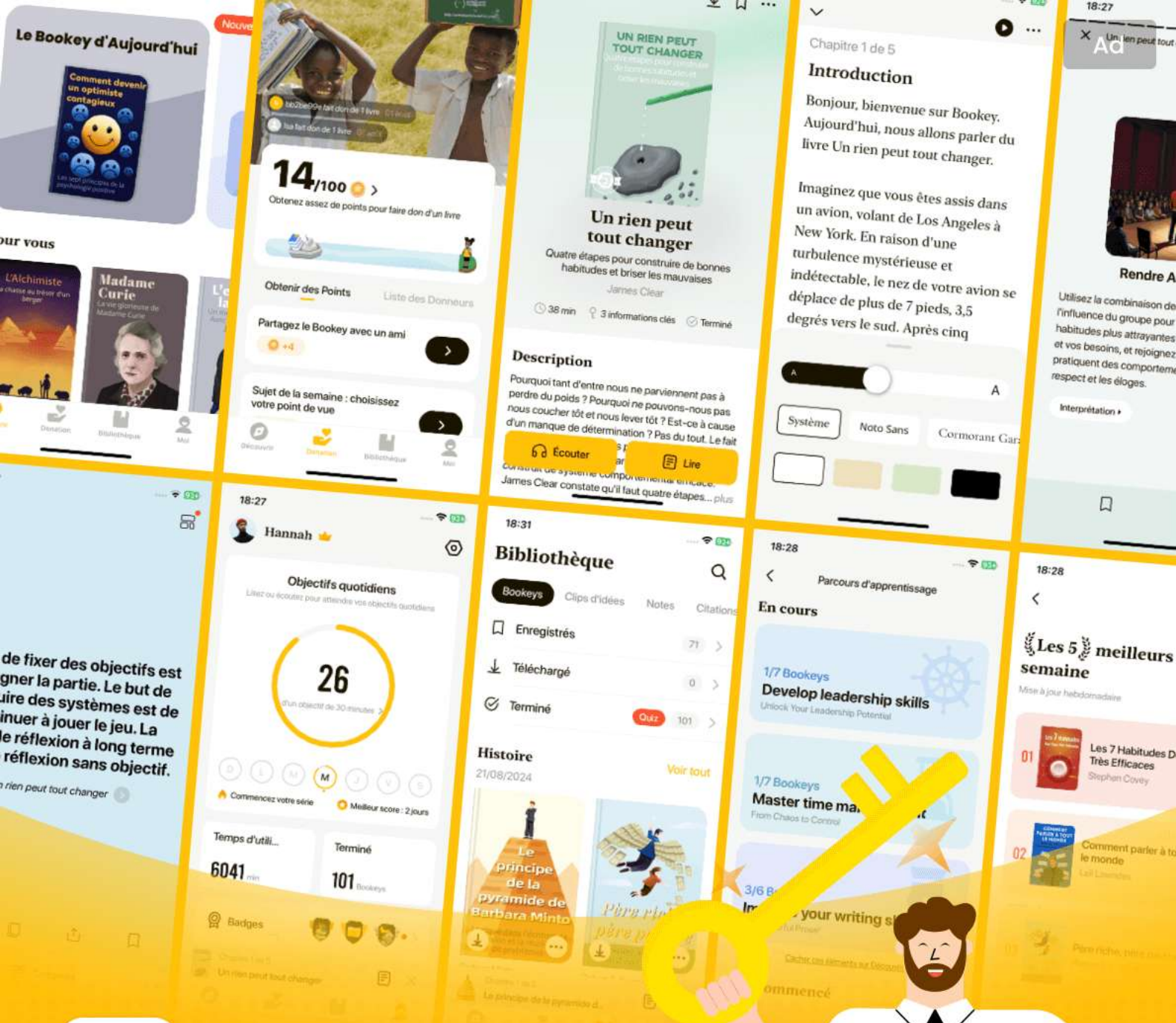
Quel est le but des différentes méthodes 'get' dans la classe SerialDate ?

Réponse:Les méthodes 'get' sont destinées à récupérer des composants spécifiques d'une date, tels que l'année, le mois et le jour du mois, permettant aux utilisateurs d'obtenir facilement et de manipuler ces composants individuels pour diverses fins.

8.Question

Comment la classe SerialDate facilite-t-elle l'ajout de jours, de mois ou d'années à une date donnée ?

Réponse:La classe SerialDate inclut des méthodes statiques qui permet aux utilisateurs de créer de nouvelles instances de SerialDate avec des dates ajustées en ajoutant un nombre spécifié de jours, de mois ou d'années à une date existante, illustrant sa praticité pour la manipulation des dates.



Les meilleures idées du monde débloquent votre potentiel

Essai gratuit avec Bookey

Télécharger dans App Store

DISPONIBLE SUR Google Play

Scanner pour télécharger

Chapitre 19 | C : Références croisées des heuristiques| Questions et réponses

1.Question

Quel est le principal objectif des références croisées des mauvaises odeurs et des heuristiques dans CODER PROPREMENT ?

Réponse:Les références croisées servent de guide pour identifier et relier les différentes mauvaises odeurs de code et les heuristiques associées, permettant aux développeurs d'améliorer systématiquement la qualité de leur code. En se référant à cet appendice, les développeurs peuvent cibler des problèmes de programmation spécifiques et appliquer des heuristiques établies pour les résoudre efficacement.

2.Question

Comment la compréhension des mauvaises odeurs de code bénéficie-t-elle à un développeur ?

Réponse:Comprendre les mauvaises odeurs de code permet aux développeurs de reconnaître des schémas problématiques

dans leur code, ce qui peut entraîner des problèmes de maintenance, des bugs et une dette technique. En étant conscients de ces odeurs, ils peuvent proactivement refactoriser et améliorer le code, ce qui aboutit à des logiciels plus propres, plus efficaces et plus gérables.

3.Question

Pouvez-vous expliquer l'importance d'une heuristique particulière liée à une mauvaise odeur de code ?

Réponse:Par exemple, une mauvaise odeur de code courante est "Méthode Longue", qui se réfère à des méthodes excessivement longues et complexes. L'heuristique qui y est associée pourrait suggérer de diviser la méthode en méthodes plus petites et plus ciblées. Ce changement améliore la lisibilité, facilite les tests unitaires et augmente la maintenabilité globale du code.

4.Question

Que doit faire un développeur s'il identifie une mauvaise odeur de code dans son projet ?

Réponse:Lorsqu'un développeur identifie une mauvaise

odeur de code, il doit analyser la cause sous-jacente, rechercher des heuristiques liées qui traitent la mauvaise odeur spécifique, puis mettre en œuvre des techniques de refactorisation qui respectent ces heuristiques. Ce processus permet non seulement de nettoyer le code, mais aussi de favoriser une meilleure pratique de codage dans l'ensemble.

5.Question

Comment le fait de se référer continuellement aux heuristiques peut-il affecter la qualité du code à long terme ?

Réponse:Se référer continuellement aux heuristiques pendant le codage favorise une approche disciplinée du développement logiciel, menant à moins de problèmes de dette technique et à une base de code plus durable. Cela cultive l'habitude d'écrire du code propre, ce qui peut réduire considérablement le temps passé à déboguer et à maintenir à l'avenir.

6.Question

Que peuvent apprendre les développeurs en comparant différentes mauvaises odeurs de code et leurs heuristiques

associées ?

Réponse:Les développeurs peuvent apprendre la nature interconnectée des différents problèmes dans les pratiques de codage. En comparant différentes odeurs et leurs heuristiques, ils peuvent voir comment s'attaquer à un problème pourrait atténuer d'autres, promouvant une approche holistique du codage qui valorise la qualité et la maintenabilité.

Chapitre 20 | Index| Questions et réponses

1.Question

Quel est l'impact du 'mauvais code' dans un projet logiciel ?

Réponse:Le mauvais code a un effet dégradant, entraînant une complexité accrue et des coûts de maintenance plus élevés. Il réduit la productivité et peut décourager le moral des développeurs. Un exemple clair est lorsqu'un développeur fait face à un bug dans un mauvais code ; le corriger peut nécessiter de trier une logique confuse et une

structure peu claire, consommant un temps et des ressources précieux.

2.Question

Comment les commentaires dans le code peuvent-ils améliorer ou nuire à la qualité du code ?

Réponse:De bons commentaires clarifient l'intention et fournissent du contexte, améliorant ainsi la compréhension, tandis que de mauvais commentaires peuvent induire en erreur, fournir des informations redondantes ou rendre le code plus difficile à lire. Par exemple, un commentaire clair expliquant pourquoi une solution non évidente a été choisie peut aider les futurs mainteneurs, tandis qu'un commentaire qui indique simplement ce qui est déjà évident dans le code ajoute souvent du désordre.

3.Question

Quel rôle jouent les 'tests' dans l'assurance de la qualité du code ?

Réponse:Les tests sont essentiels pour maintenir le CODER PROPREMENT car ils aident à détecter les erreurs tôt, à

vérifier la fonctionnalité et à offrir une sécurité lors des refactorisations. Des tests clairs et bien structurés peuvent mettre en évidence les faiblesse du code et garantir que les modifications futures ne cassent pas la fonctionnalité existante.

4.Question

Quelle est la définition du code propre selon 'Clean Code' ?

Réponse:Le CODER PROPREMENT est défini comme un code simple, facile à lire et à maintenir. Il reflète l'intention, est bien organisé et est dépourvu de complexité inutile. L'essence du CODER PROPREMENT est d'assurer que les développeurs peuvent facilement comprendre et travailler avec le code, réduisant ainsi efficacement le risque de bugs et améliorant la qualité globale du logiciel.

5.Question

Pouvez-vous expliquer le concept de 'séparation des préoccupations' ?

Réponse:La séparation des préoccupations est un principe de

conception qui promeut l'organisation du code en sections distinctes, chacune responsable d'un aspect spécifique de la fonctionnalité. Par exemple, une classe qui gère les interactions avec la base de données ne devrait pas gérer la logique de l’interface utilisateur. Cela rend chaque section plus facile à comprendre, à maintenir et à tester, résultant en une architecture d'application plus robuste.

6.Question

Pourquoi les noms sont-ils importants dans le CODER PROPREMENT ?

Réponse:Les noms sont cruciaux car ils affectent directement la lisibilité et la maintenabilité du code. Des noms descriptifs et révélateurs d'intention aident à réduire la charge cognitive des développeurs. Par exemple, une méthode nommée 'calculerPrixTotal()' est beaucoup plus claire que 'faireCalc()'. Une bonne nomination minimise le besoin de commentaires et d'explications, permettant aux autres de comprendre rapidement le code.

7.Question

Comment la complexité affecte-t-elle la maintenance du code ?

Réponse:Le code complexe augmente la charge cognitive sur les développeurs, rendant plus difficile la compréhension et la modification. Cela peut donner lieu à des bugs et rendre l'ajout de fonctionnalités plus difficile, car les développeurs peuvent ne pas saisir comment différentes parties interagissent. Par conséquent, la gestion de la complexité par la simplification du code et l'adhésion à des principes tels que 'faire une seule chose' aide à faciliter la maintenance.

8.Question

Que peut-on faire pour éviter les 'odeurs de code' et promouvoir des pratiques de programmation propres ?

Réponse:Pour éviter les odeurs de code et promouvoir des pratiques de programmation propres, les développeurs peuvent mettre en œuvre des pratiques telles que le refactoring régulier du code, l'adhésion à des normes de codage solides, l'écriture de tests complets et l'utilisation d'outils pour identifier les problèmes potentiels. La

sensibilisation aux motifs et principes de codage—comme la règle du Boy Scout, qui dit aux développeurs de laisser le code plus propre qu'ils ne l'ont trouvé—aide à garantir une qualité de code continue.

9.Question

Quel est le 'Principe de responsabilité unique' (SRP) et pourquoi est-il important ?

Réponse:Le Principe de responsabilité unique stipule qu'une classe ne doit avoir qu'une seule raison de changer, ce qui signifie qu'elle ne doit avoir qu'un seul emploi ou une seule responsabilité. Ce principe est vital car il conduit à un code plus compréhensible et plus maintenable. Par exemple, une classe gérant à la fois l'accès aux données et le rendu de l'interface utilisateur peut devenir étroitement couplée et plus difficile à modifier, tandis que la séparation de ces préoccupations permet des mises à jour individuelles sans affecter l'autre.

10.Question

Comment le 'développement piloté par les tests' (TDD) peut-il influencer la qualité du code ?

Réponse:Le développement piloté par les tests influence positivement la qualité du code en encourageant les développeurs à écrire des tests avant le code réel. Cette approche conduit à une meilleure conception, car les développeurs doivent considérer comment leur code sera testé, souvent en résultant en conceptions plus simples et plus modulaires. Elle garantit que chaque fonctionnalité est validée, menant finalement à un code plus propre et plus fiable.

Chapitre 21 | Introduction Préalable| Questions et réponses

1.Question

Quelle est la définition du professionnalisme en tant que programmeur selon Robert C. Martin?

Réponse:Le professionnalisme englobe les attitudes, les disciplines et les actions qui définissent l'approche d'un programmeur envers son travail. Cela implique de viser une amélioration continue, de prendre la responsabilité de ses actions, de s'engager dans un apprentissage tout au long de la vie, et de

maintenir des standards éthiques élevés.

2.Question

Comment les premières expériences peuvent-elles façonner le développement professionnel d'un programmeur?

Réponse:Les premières expériences, qu'elles soient positives ou négatives, peuvent façonner significativement la compréhension du professionnalisme d'un programmeur. Par exemple, Martin partage que ses erreurs de débutant – comme démissionner impulsivement sans autre emploi en vue – ont été des moments d'apprentissage critiques qui lui ont enseigné l'importance de l'humilité et de la prise de décision réfléchie.

3.Question

Quel rôle joue l'humilité dans la professionnalisation d'un programmeur?

Réponse:L'humilité est cruciale pour la croissance professionnelle car elle permet aux programmeurs de reconnaître leurs erreurs, d'en tirer des leçons et de chercher des conseils si nécessaire. Martin raconte comment il a dû

'manger son chapeau' en postulant à nouveau pour son emploi après avoir démissionné sur un coup de tête, soulignant que reconnaître ses limites est une étape clé dans le développement professionnel.

4.Question

Comment les conséquences des actions influencent-elles la croissance professionnelle dans le domaine de la programmation?

Réponse:Les conséquences, surtout des erreurs commises au travail, servent de puissantes leçons qui peuvent propulser la croissance professionnelle. Par exemple, Martin évoque avoir été licencié pour avoir manqué des délais critiques, ce qui lui a enseigné l'importance de la communication et de la responsabilité. Tirer des leçons de ces expériences aide à éviter des problèmes similaires à l'avenir.

5.Question

De quelle manière les erreurs partagées entre pairs peuvent-elles impacter la carrière d'un programmeur?

Réponse:Les erreurs partagées peuvent conduire à un apprentissage et à une amélioration mutuels parmi les

programmeurs. L'expérience de Martin, où il a sans le savoir causé le licenciement d'autres personnes, illustre l'importance du leadership et de l'impact que les décisions d'un individu peuvent avoir sur une équipe, soulignant que les programmeurs doivent être conscients de leurs responsabilités envers les autres.

6.Question

Quelle importance Robert C. Martin accorde-t-il à l'apprentissage continu en programmation?

Réponse:L'apprentissage continu est vital pour maintenir le professionnalisme en programmation. Martin souligne que le paysage technologique évolue constamment, et se tenir informé par un apprentissage persistant permet aux programmeurs de s'adapter et d'améliorer leurs compétences, augmentant ainsi leur valeur dans l'industrie.

7.Question

Comment la responsabilité personnelle peut-elle façonner l'avenir de la carrière d'un programmeur?

Réponse:La responsabilité personnelle est essentielle pour la

croissance et le succès dans la carrière d'un programmeur. Reconnaître ses propres erreurs, comme l'a fait Martin, favorise la confiance et le respect entre collègues et employeurs, ouvrant la voie à des opportunités et des promotions futures. Prendre la responsabilité de son travail garantit que les programmeurs peuvent contribuer efficacement à leurs équipes.

8.Question

Quel message global Robert C. Martin transmet-il sur le parcours pour devenir un programmeur professionnel?

Réponse:Le parcours pour devenir un programmeur professionnel est marqué par l'apprentissage continu, l'humilité et la réflexion sur soi. C'est un chemin défini par des succès et des échecs, chaque expérience servant de leçon cruciale pour développer le professionnalisme et améliorer ses compétences dans le recrutement et l'exécution des tâches de programmation.

Chapitre 22 | 1 Professionnalisme| Questions et réponses

1.Question

Que signifie vraiment le professionnalisme dans le développement logiciel ?

Réponse:Le professionnalisme dans le développement logiciel signifie prendre la responsabilité de son travail, être accountable de ses erreurs et prioriser la qualité de son code. Cela implique un engagement non seulement à respecter les délais, mais aussi à s'assurer que le logiciel fonctionne correctement et est maintenable. Il s'agit d'équilibrer la fierté avec la volonté de s'excuser et d'apprendre de ses erreurs.

2.Question

Comment un développeur logiciel peut-il prendre la responsabilité de son travail ?

Réponse:Un développeur peut prendre la responsabilité en testant soigneusement son code avant sa publication, en s'assurant qu'il fonctionne comme prévu. Il doit être proactif

pour détecter les erreurs, limiter les bugs et travailler en continu pour améliorer à la fois la fonctionnalité et la structure de son code. De plus, il devrait s'excuser et apprendre de ses erreurs lorsqu'elles se produisent.

3.Question

Quelle est la signification du 'ne pas nuire' dans les pratiques de codage ?

Réponse:'Ne pas nuire' signifie que les développeurs logiciels doivent s'efforcer d'éviter de créer des bugs ou des échecs qui peuvent impacter les utilisateurs ou les systèmes de manière négative. Tout comme les médecins prêtent un serment de prévenir les dommages, les développeurs doivent se fixer le même standard lors de la création de logiciels, garantissant que ce qu'ils livrent est aussi fiable et exempt de bugs que possible.

4.Question

Quel rôle joue l'apprentissage continu dans le fait d'être un développeur professionnel ?

Réponse:L'apprentissage continu est crucial pour un

développeur professionnel, car la technologie et les méthodologies évoluent constamment. Les développeurs doivent rester à jour sur les nouveaux langages, frameworks et meilleures pratiques. Cela améliore non seulement leurs compétences, mais les maintient également pertinents dans un secteur en évolution rapide.

5.Question

Comment les professionnels peuvent-ils s'assurer que leur code reste flexible et maintenable ?

Réponse:Pour garder le code flexible et maintenable, les professionnels doivent s'engager dans un refactoring régulier et adopter la pratique d'apporter de petites améliorations chaque fois qu'ils travaillent sur du code. Ils doivent également s'assurer que leur code est conçu avec des tests à l'esprit, en utilisant des pratiques comme le développement dirigé par les tests (TDD) pour faciliter les changements futurs.

6.Question

Pourquoi est-il important de connaître son domaine en tant que développeur logiciel ?

Réponse:Comprendre son domaine permet à un développeur de produire des solutions qui répondent réellement aux besoins des utilisateurs et aux objectifs commerciaux. Cela aide à reconnaître les erreurs de spécification et garantit que le logiciel est en accord avec les pratiques et exigences de l'industrie, conduisant à une meilleure qualité et moins de malentendus avec les parties prenantes.

7.Question

Quel état d'esprit un développeur professionnel devrait-il avoir envers son employeur ?

Réponse:Un développeur professionnel devrait s'aligner sur les objectifs de son employeur et considérer ses problèmes comme les siens. Cela signifie rechercher activement des solutions qui répondent aux besoins de l’employeur et collaborer pour développer un logiciel qui contribue au succès de l'organisation.

8.Question

Quel rôle joue l'humilité dans le professionnalisme pour les développeurs logiciels ?

Réponse:L'humilité est cruciale dans le professionnalisme car elle permet aux développeurs de reconnaître leurs limites et d'apprendre de leurs erreurs. Reconnaître que la programmation implique des risques et que personne n'est à l'abri des erreurs favorise un environnement collaboratif où le partage des connaissances et des expériences améliore la croissance et l'apprentissage de l'équipe.

9.Question

Quelles pratiques les développeurs peuvent-ils adopter pour maintenir leurs compétences au fil du temps ?

Réponse:Les développeurs peuvent maintenir leurs compétences par une pratique constante, comme s'engager dans des katas de code, collaborer avec des pairs, assister à des ateliers ou des conférences, mentorant d'autres, et en réservant du temps personnel pour explorer de nouveaux outils et technologies.

10.Question

Pourquoi la collaboration est-elle importante dans le développement logiciel ?

Réponse:La collaboration permet aux développeurs de partager des idées, de résoudre des problèmes plus rapidement et de produire des logiciels de meilleure qualité. Travailler ensemble encourage l'apprentissage mutuel, réduit les erreurs et favorise un sentiment de communauté qui conduit à de meilleurs résultats globaux.

Chapitre 23 | 2 Dire Non| Questions et réponses

1.Question

Quelle est l'importance de dire non à son patron et pourquoi est-ce considéré comme un acte professionnel ?

Réponse:Dire non à son patron est crucial car cela reflète le professionnalisme. Les professionnels sont censés résister aux attentes irréalistes. Il ne s'agit pas simplement de suivre des ordres ; il est question de s'assurer que le travail effectué est réalisable et conforme aux objectifs globaux de qualité et de délais. Dire non permet de négocier et peut conduire à un meilleur résultat pour les projets.

2.Question

Quelles sont les conséquences de ne pas dire non lorsque l'on est sous pression de la part de la direction ?

Réponse:Ne pas dire non peut entraîner des conséquences désastreuses, comme le montre l'histoire d'un projet précipité qui a abouti à un système défectueux incapable de répondre aux attentes des utilisateurs, nuisant tant aux employés qu'à l'organisation. Les professionnels doivent évaluer les délais de projet de manière réaliste ; sinon, ils risquent de créer un travail de mauvaise qualité qui ne satisfera finalement pas les clients.

3.Question

Comment la confrontation peut-elle être bénéfique dans les relations professionnelles, comme entre les programmeurs et les managers ?

Réponse:La confrontation peut mener à une communication plus claire et à de meilleurs résultats de projet. Lorsque les deux parties expriment de manière assertive leurs besoins et leurs limites, elles peuvent négocier une solution qui aligne les objectifs du projet. Ce rôle d'adversaire aide à prévenir les

malentendus et garantit que les capacités de l'équipe et les besoins de la direction sont pris en compte.

4.Question

Quelles stratégies pouvez-vous adopter pour dire non de manière efficace tout en maintenant une relation professionnelle ?

Réponse:Un style de communication clair et assertif est essentiel. Par exemple, énoncez les faits et les délais estimés de manière sans équivoque. Utilisez des exemples ou des expériences passées pour appuyer votre point de vue. Il est également bénéfique de proposer des solutions alternatives ou des compromis qui préservent encore les standards de qualité, montrant qu'en disant non, vous restez néanmoins dédié au succès du projet.

5.Question

Pourquoi l'expression 'Je vais essayer' est-elle considérée comme un signe d'improfessionnalisme dans ce contexte ?

Réponse:Dire 'je vais essayer' implique de l'incertitude et peut suggérer que vous ne vous engagez pas pleinement dans la tâche. Cela indique que vous ne fournissez peut-être pas le

meilleur de vous-même ou que vous n'avez pas planifié pour réussir. Les professionnels devraient énoncer clairement leurs capacités au lieu de tergiverser ; sinon, ils risquent de se mettre en échec en promettant quelque chose qu'ils ne peuvent pas livrer.

6.Question

Quel rôle la communication joue-t-elle pour éviter les pièges de dire trop souvent oui ?

Réponse:Une communication efficace est essentielle pour établir des attentes réalistes. En discutant ouvertement des délais, des capacités et des besoins en ressources, les membres de l'équipe peuvent créer une compréhension mutuelle qui favorise une planification réaliste. S'assurer que tout le monde est sur la même longueur d'onde minimise la tentation de trop promettre et permet une approche plus collaborative de la gestion de projet.

7.Question

Comment la quête de reconnaissance personnelle peut-elle mener à un comportement non professionnel ?

Réponse:Lorsque les individus privilégient la reconnaissance personnelle au détriment de la qualité réelle de leur travail, ils peuvent compromettre leurs standards pour satisfaire des exigences irréalistes. Le désir d'être vu comme un héros peut mener à accepter des délais ou des périmètres impossibles, cultivant ainsi des pratiques de codage non professionnelles et résultant finalement en échec.

8.Question

En réfléchissant à ce chapitre, comment un développeur peut-il équilibrer assertivité et travail d'équipe ?

Réponse:Un développeur peut équilibrer assertivité et travail d'équipe en étant honnête sur ce qui est réalisable et en plaidant pour des délais réalistes tout en restant ouvert à la collaboration sur les solutions. Encourager un dialogue où les objectifs du développeur et du manager sont discutés peut créer un environnement plus harmonieux où les deux parties se sentent entendues et respectées.

9.Question

Quelle est la relation entre le fait de dire non et l'obtention du meilleur résultat possible pour un projet ?

Réponse:Dire non lorsque c'est nécessaire garantit que les objectifs du projet sont réalistes et atteignables. En affirmant les limites, vous protégez la qualité du travail et le moral général de l'équipe. Cet engagement à maintenir des normes peut conduire à un résultat plus réussi, car toutes les parties sont claires sur les attentes et les délais.

10.Question

Pouvez-vous donner un exemple de comment communiquer un non dans un contexte de projet ?

Réponse:Bien sûr ! Si un manager demande une fonctionnalité pour demain, un développeur pourrait répondre : 'J'apprécie l'urgence, mais je dois être transparent : en raison de la complexité de la fonctionnalité, il me faudra au moins trois jours pour la livrer correctement. Pouvons-nous discuter des aspects essentiels à présenter immédiatement, ou si nous pouvons ajuster le délai pour une livraison complète ?' Cette approche fait un non ferme tout en invitant à la négociation.

Chapitre 24 | 3 Dire Oui| Questions et réponses

1.Question

Que signifie vraiment s'engager à quelque chose ?

Réponse:Un véritable engagement implique de dire que vous ferez quelque chose, de le signifier sincèrement et de tenir cette promesse. Cela nécessite une communication claire, un sens des responsabilités personnelles et la mise en place de délais précis pour l'achèvement.

2.Question

Comment pouvons-nous reconnaître un manque d'engagement chez les autres ?

Réponse:Un manque d'engagement se manifeste souvent par un langage vague tel que 'besoin', 'devrait', 'espérer' et 'faisons', qui implique que le locuteur ne prend pas l'entière responsabilité de la tâche.

3.Question

À quoi ressemble une déclaration d'engagement forte ?

Réponse:Une déclaration d'engagement forte exprime clairement l'action que vous allez entreprendre ainsi qu'une date limite spécifique. Par exemple, 'Je vais terminer ceci

d'ici mardi' ne laisse aucune ambiguïté.

4.Question

Que devez-vous faire si des problèmes imprévus surviennent qui pourraient vous empêcher de respecter un engagement ?

Réponse:Vous devez communiquer tôt et honnêtement sur le problème potentiel. En soulevant les problèmes dès qu'ils surviennent, vous permettez à votre équipe de réévaluer et d'ajuster les priorités ou les responsabilités.

5.Question

Pourquoi le changement de langage est-il important lors de la prise d'engagements ?

Réponse:Modifier le langage pour refléter la certitude et la responsabilité personnelle favorise la transparence et la confiance. Cela clarifie ce que vous pouvez réaliser de manière réaliste, établissant ainsi un ton pour la responsabilité.

6.Question

Comment pouvez-vous faire face à la pression de changer d'engagements ou de normes au travail ?

Réponse:Un professionnel doit maintenir ses normes et ne pas transiger sur la qualité. En cas de pression, expliquez pourquoi certaines pratiques sont essentielles pour le succès du projet et explorez des solutions alternatives sans sacrifier la qualité.

7.Question

Quelle est l'importance de dire 'oui' de manière réfléchie dans un environnement professionnel ?

Réponse:Dire 'oui' de manière réfléchie peut améliorer votre réputation en tant que membre fiable de l'équipe qui respecte ses obligations. Cela démontre une attitude responsable et un engagement envers la qualité, ce qui peut favoriser de meilleures dynamiques d'équipe et le succès des projets.

8.Question

Comment le concept de 'essayer' entrave-t-il une communication efficace ?

Réponse:Utiliser des termes comme 'essayer' peut créer de l'ambiguïté et miner la confiance. Cela suggère une incertitude et un manque de responsabilité, tandis qu'un

engagement clair envers un résultat spécifique améliore la transparence.

9.Question

Quel est un exemple de professionnel gérant une demande irréaliste d'un manager ?

Réponse:Dans le scénario où un manager insiste sur un délai serré, un professionnel doit évaluer ses capacités, confirmer honnêtement ses limites et proposer un calendrier réaliste, comme l'a fait Peter dans l'histoire.

10.Question

Pourquoi est-il important de différencier entre la responsabilité personnelle et les dépendances de l'équipe lors de la prise d'engagements ?

Réponse:Reconnaître ce que vous pouvez contrôler par rapport à ce qui dépend des autres vous permet de formuler des engagements réalistes. Vous pouvez toujours soutenir le projet en vous engageant dans des actions qui contribuent à l'objectif tout en étant clair sur la dépendance envers les autres.

Scanner pour télécharger

Pourquoi Bookey est une application incontournable pour les amateurs de livres

Contenu de 30min
Plus notre interprétation est profonde et claire, mieux vous saisissez chaque titre.

Format texte et audio
Absorbez des connaissances même dans un temps fragmenté.

Quiz
Vérifiez si vous avez maîtrisé ce que vous venez d'apprendre.

Et plus
Plusieurs voix & polices, Carte mentale, Citations, Clips d'idées...

Essai gratuit avec Bookey

Chapitre 25 | 4 Codage| Questions et réponses

1.Question

Quelle est l'importance de développer un sens de l'erreur en programmation ?

Réponse:Développer un sens de l'erreur est crucial car cela permet à un programmeur d'identifier et de rectifier rapidement les erreurs, améliorant ainsi non seulement la qualité du code mais aussi l'apprentissage tiré de ces erreurs. Cette compétence favorise une boucle de rétroaction plus rapide, essentielle pour maîtriser la programmation et maintenir la confiance.

2.Question

Comment la fatigue affecte-t-elle la performance en codage ?

Réponse:La fatigue a un impact néfaste sur la performance en codage, comme le montrent des anecdotes personnelles de codage à 3 heures du matin, entraînant de mauvaises décisions de conception. Cela illustre l'importance de

s'assurer d'un repos suffisant et de maintenir un haut niveau de vivacité mentale pour éviter les erreurs et créer un code efficace.

3.Question

Pourquoi maintenir son attention et sa concentration est-il vital en codage ?

Réponse:Maintenir son attention et sa concentration est vital en codage car la programmation nécessite de jongler avec de multiples détails complexes—comme comprendre le problème, respecter les principes d'ingénierie et assurer la lisibilité—simultanément. Les distractions peuvent entraîner des erreurs qui rendent nécessaires des corrections, entraînant une perte de temps et de ressources.

4.Question

Qu'est-ce que la 'Zone' et pourquoi les programmeurs doivent-ils s'en méfier ?

Réponse:La 'Zone' fait référence à un état de hyper-focalisation où les programmeurs se sentent incroyablement productifs. Cependant, cela peut être

trompeur car les décisions prises dans cet état manquent souvent d'une perspective plus large, conduisant à des erreurs qui nécessitent des corrections ultérieures. Prendre du recul et s'engager dans des discussions ou des pauses créatives peut préserver la clarté et améliorer la qualité du codage.

5.Question

Quelles stratégies peuvent être utilisées pour gérer les inquiétudes et les distractions en codage ?

Réponse:Pour gérer les inquiétudes et les distractions, les stratégies comprennent la répartition d'un temps dédié pour aborder les problèmes personnels en dehors des heures de travail, l'utilisation de la programmation en binôme pour maintenir le contexte après des interruptions, et comprendre quand se déconnecter du codage pour permettre à l'inconscient de résoudre les problèmes.

6.Question

Comment les pratiques collaboratives améliorent-elles la performance en programmation ?

Réponse:Les pratiques collaboratives, comme la

programmation en binôme et le mentorat, améliorent la performance en offrant de nouvelles perspectives, en favorisant la résolution efficace de problèmes, et en garantissant un partage des connaissances entre les membres de l'équipe, ce qui est essentiel compte tenu de la complexité de la programmation et de la nécessité d'une communication claire.

7.Question

Quel rôle l'état mental d'un programmeur joue-t-il dans son efficacité ?

Réponse:L'état mental d'un programmeur affecte significativement son efficacité ; le stress, les inquiétudes ou la fatigue peuvent entraver la concentration et la créativité. Trouver des moyens de gérer les défis personnels et s'engager dans des pratiques réparatrices, comme prendre des pauses et maintenir un mode de vie sain, peut conduire à une productivité accrue et à une meilleure qualité de code.

8.Question

Quelle est l'approche recommandée pour gérer les délais de projet ?

Réponse:Pour gérer efficacement les délais de projet, il est conseillé d'estimer les délais de manière réaliste avec des scénarios optimistes, nominaux et pessimistes, de communiquer de manière transparente avec les parties prenantes et d'éviter les pièges de l' 'espoir' qui peuvent entraîner des délais manqués. L'adaptabilité et la clarté dans les plans sont essentielles pour l'intégrité professionnelle.

9.Question

Comment l'apport créatif peut-il prévenir le blocage de l'écrivain en programmation ?

Réponse:L'apport créatif provenant de diverses sources, comme la lecture, l'exploration de différents genres et la participation à des activités qui stimulent l'esprit, peut briser le blocage de l'écrivain. S'engager avec des idées diverses encourage la créativité et aide les programmeurs à connecter plus efficacement les concepts lors du codage.

10.Question

Quelles sont les responsabilités éthiques des programmeurs concernant l'aide aux autres ?

Réponse:Les programmeurs ont une responsabilité éthique d'aider leurs pairs. La collaboration non seulement améliore l'apprentissage et la résolution de problèmes, mais contribue également à un environnement de travail solidaire. Le mentorat et l'ouverture à aider les autres sont cruciaux pour la croissance personnelle et collective dans le domaine.

Chapitre 26 | 5 Développement Driven par les Tests| Questions et réponses

1.Question

Qu'est-ce que le développement piloté par les tests (TDD) et pourquoi est-il important ?

Réponse:Le développement piloté par les tests (TDD) est une discipline de programmation où les tests unitaires sont écrits avant le code réel. Cette méthodologie est cruciale car elle garantit non seulement que le code fonctionne correctement, mais améliore également la fiabilité et la maintenabilité du code en intégrant continuellement les tests dans le processus de développement.

2.Question

Comment l'expérience de la programmation avec Kent Beck a-t-elle changé votre compréhension des pratiques de codage ?

Réponse:Kent Beck a démontré un cycle de codage rapide consistant à écrire un petit test unitaire suivi juste du code de production nécessaire pour faire passer ce test. Cette approche a transformé ma vision sur l'efficacité du codage, prouvant que des tests fréquents peuvent mener à un développement plus rapide et à une meilleure qualité de code, semblable à la programmation dans un langage interprété.

3.Question

Quelles sont les trois lois du TDD ?

Réponse:1. Vous devez écrire un test unitaire échouant avant d'écrire du code de production. 2. N'écrivez pas plus que le code de test unitaire nécessaire pour le faire échouer. 3. N'écrivez pas plus de code de production que nécessaire pour faire passer le test unitaire actuellement échouant. Ce cycle augmente la productivité et encourage une conception

réfléchie.

4.Question

Quels avantages tirez-vous de l'adoption du TDD comme discipline professionnelle ?

Réponse:Adopter le TDD favorise une plus grande certitude dans votre code, améliore le taux d'injection de défauts, encourage le courage de refactoriser et de coder proprement sans craindre de casser des choses, crée une meilleure documentation grâce à des tests automatisés et conduit à une qualité de conception améliorée.

5.Question

Pourquoi le TDD est-il lié à une qualité de code supérieure et à une réduction des défauts ?

Réponse:Parce que le TDD nécessite des tests continus, il entraîne un nombre accru de tests développés et exécutés tout au long du processus de codage. Cela aide non seulement à détecter les défauts tôt, mais assure également une couverture complète du code, réduisant considérablement les chances de bogues.

6.Question

Comment le TDD encourage-t-il de meilleures pratiques de conception ?

Réponse:Écrire des tests d'abord oblige les développeurs à réfléchir de manière critique aux dépendances et à l'isolement des fonctions, empêchant ainsi le code étroitement couplé difficile à tester. Cette focalisation sur la testabilité promeut une meilleure architecture et un design qui soutient la maintenabilité.

7.Question

Pourquoi les professionnels devraient-ils considérer le TDD comme une discipline nécessaire plutôt que comme une option ?

Réponse:Étant donné les avantages substantiels de certitude, de réduction des défauts et de meilleure documentation que le TDD offre, il peut être perçu comme non professionnel de ne pas l'adopter. C'est une discipline qui, si elle est suivie de manière cohérente, propulse l'efficacité d'un développeur et la qualité globale du code.

8.Question

Quelles idées fausses sur le TDD doivent être abordées ?
Réponse:Le TDD n'est pas une solution garantie pour écrire du code parfait ni un dogme strict à suivre dans toutes les circonstances. Il nécessite une application pratique ; il existe des situations où adhérer au TDD peut ne pas être faisable ou bénéfique.

9.Question

Comment les principes du TDD peuvent-ils améliorer les pratiques de codage d'une équipe dans son ensemble ?
Réponse:En adoptant le TDD, les équipes cultivent une culture de responsabilité et de qualité dans leurs pratiques de codage. Cela encourage une compréhension partagée du code à travers des cas de test complets, favorise la collaboration et assure une livraison cohérente de logiciels de haute qualité.

Chapitre 27 | 6 Pratiquer| Questions et réponses

1.Question

Pourquoi la pratique est-elle fondamentale dans le développement logiciel ?
Réponse:La pratique est fondamentale dans le

développement logiciel car elle permet aux programmeurs d’affiner leurs compétences, d’améliorer leur rapidité et leur efficacité, et de se préparer à des scénarios de résolution de problèmes du monde réel. Tout comme les musiciens ou les athlètes, les programmeurs doivent s’engager dans des exercices de perfectionnement des compétences qui leur permettent de réagir rapidement et avec précision sous pression.

2.Question

Comment la pratique de la programmation a-t-elle évolué depuis les débuts ?

Réponse:La pratique de la programmation a évolué de manière significative depuis les débuts, lorsque les possibilités d'entraînement étaient limitées par de longs temps de compilation et des processus de débogage lents. Aujourd'hui, grâce à des boucles de rétroaction rapides grâce à des temps de compilation courts et à une puissance de calcul accrue, les programmeurs peuvent pratiquer et

améliorer leurs compétences beaucoup plus efficacement, renforçant ainsi leur capacité à peaufiner leur art.

3.Question

Quel est le concept de "Coding Dojo" et en quoi cela bénéficie-t-il aux programmeurs ?

Réponse:Un Coding Dojo est un environnement de pratique où les programmeurs se réunissent pour travailler sur des katas, ou des exercices structurés, tout comme les artistes martiaux pratiquent leurs mouvements. Ce cadre collaboratif favorise l’apprentissage entre pairs, expose les participants à des approches diversifiées de la résolution de problèmes et les aide à intégrer les pratiques et techniques de programmation.

4.Question

Qu'est-ce qu'un kata de programmation et pourquoi est-il important ?

Réponse:Un kata de programmation est un exercice de codage bien défini qui est pratiqué à plusieurs reprises pour perfectionner une compétence ou une technique spécifique.

Tout comme le fait de répéter une pièce musicale, pratiquer des katas aide à solidifier des concepts clés de programmation dans la mémoire d’un programmeur, améliorant leur efficacité et leur familiarité avec les méthodes de résolution de problèmes.

5.Question

Quel rôle la diversité dans la résolution de problèmes joue-t-elle pour les programmeurs ?

Réponse:La diversité dans la résolution de problèmes permet aux programmeurs de rencontrer divers défis, les empêchant de devenir stagnants ou trop spécialisés dans un langage ou un domaine. Cette variété d'expérience les prépare aux évolutions de l'industrie et favorise la créativité et l'adaptabilité, des traits cruciaux pour le succès à long terme dans leur carrière.

6.Question

Pourquoi les programmeurs devraient-ils pratiquer pendant leur temps libre ?

Réponse:Les programmeurs devraient pratiquer pendant leur

temps libre car il est de leur responsabilité de maintenir et d’améliorer leurs compétences. Tout comme d'autres professionnels pratiquent en dehors de leur travail rémunéré, les programmeurs ne devraient pas s'attendre à ce que leurs employeurs leur fournissent toutes les opportunités d'apprentissage ; la pratique personnelle conduit à une meilleure préparation et un potentiel de gains plus élevés.

7.Question

Quelle est l'importance de maîtriser différents langages de programmation ?

Réponse:Maîtriser différents langages de programmation est vital pour un programmeur car cela élargit leur ensemble de compétences, offre de nouvelles perspectives sur la résolution de problèmes et les maintient adaptables aux exigences en constante évolution de l'industrie. Être polyglotte aide les programmeurs à rester pertinents et préparés pour diverses opportunités d'emploi dans un domaine en rapide évolution.

8.Question

Quelle est la relation entre la rapidité en programmation et la pratique ?

Réponse:La rapidité en programmation est directement liée à la pratique ; plus un programmeur s'entraîne sur des tâches de codage, plus ses frappes deviennent instinctives et fluides. Tout comme les athlètes ou les musiciens, une pratique cohérente et ciblée permet aux programmeurs d’exécuter rapidement des tâches complexes, permettant à leurs ressources mentales de se concentrer sur des résolutions de problèmes de niveau supérieur.

Scanner pour télécharger

App Store
Coup de cœur

22k avis 5 étoiles

Retour Positif

Fabienne Moreau

ue résumé de livre ne testent
ion, mais rendent également
musant et engageant.
té la lecture pour moi.

Fantastique! Giselle Dubois

Je suis émerveillé par la variété de livres et de langues que Bookey supporte. Ce n'est pas juste une application, c'est une porte d'accès au savoir mondial. De plus, gagner des points pour la charité est un grand plus !

Fi
Le
liv
co
pr

é Blanchet

de lecture
eption de
es,
tous.

J'adore ! Isoline Mercier

Bookey m'offre le temps de parcourir les parties importantes d'un livre. Cela me donne aussi une idée suffisante pour savoir si je devrais acheter ou non la version complète du livre ! C'est facile à utiliser !"

Gain de temps !

Bookey est mon applicat
intellectuelle. Les résum
magnifiquement organis
monde de connaissance

Appli géniale ! Joachim Lefevre

adore les livres audio mais je n'ai pas toujours le temps
'écouter le livre entier ! Bookey me permet d'obtenir
n résumé des points forts du livre qui m'intéresse !!!
Quel super concept !!! Hautement recommandé !

Appli magnifique

Cette application est une bouée de sauve
amateurs de livres avec des emplois du te
Les résumés sont précis, et les cartes me
renforcer ce que j'ai appris. Hautement re

Essai gratuit avec Bookey

Chapitre 28 | 7 Tests d'acceptation| Questions et réponses

1.Question

Quel est le rôle principal d'un développeur professionnel au sein d'une équipe ?

Réponse:Le rôle principal d'un développeur professionnel est d'engager une communication efficace avec les membres de l'équipe et les parties prenantes du business, assurant des échanges précis et sains concernant les exigences et l'avancement du projet.

2.Question

Pourquoi est-il difficile de communiquer les exigences entre les business et les programmeurs ?

Réponse:La communication des exigences est souvent sujette à des erreurs car les parties prenantes peuvent décrire ce qu'elles pensent avoir besoin, mais les programmeurs interprètent souvent ces descriptions différemment. Ce désalignement peut conduire à créer le mauvais produit.

3.Question

Pouvez-vous expliquer le concept de 'précision prématurée' dans les exigences d'un projet ?

Réponse:La précision prématurée fait référence à la tendance des business et des développeurs à rechercher des exigences trop détaillées trop tôt dans le projet. Cela peut conduire à un gaspillage de ressources car les hypothèses initiales changent souvent lorsque le système réel est démontré.

4.Question

Comment le principe d'incertitude s'applique-t-il au développement logiciel ?

Réponse:Le principe d'incertitude dans ce contexte suggère que lorsque les parties prenantes voient leurs exigences exécutées dans le système, elles réalisent souvent que ce qu'elles ont spécifié ne répond pas à leurs véritables besoins. Cette prise de conscience peut changer leur perception et leurs exigences pour la suite.

5.Question

Quelle est la solution au problème de la précision prématurée ?

Réponse:La solution consiste à différer la précision des exigences aussi longtemps que possible, ne les détaillant juste avant le développement. Cependant, les développeurs doivent être attentifs à l'ambiguïté tardive, en s'assurant que toutes les parties prenantes s'accordent sur les spécificités avant le début du développement.

6.Question

Qu'est-ce que les tests d'acceptation ?

Réponse:Les tests d'acceptation sont des tests collaboratifs conçus par les parties prenantes et les programmeurs pour définir clairement quand une exigence est considérée comme 'terminée', garantissant que tout le monde a la même compréhension des livrables du projet.

7.Question

Que signifie 'terminé' dans le contexte des tests d'acceptation ?

Réponse:'Terminé' signifie que tout le code est écrit, tous les tests passent, et que la QA et les parties prenantes ont accepté la fonctionnalité comme satisfaisante aux exigences.

8.Question

Pourquoi est-il important d'automatiser les tests d'acceptation ?

Réponse:Automatiser les tests d'acceptation est important car cela réduit les coûts et garantit une vérification cohérente de la fonctionnalité sans le poids des tests manuels. Les tests automatisés peuvent être exécutés fréquemment, permettant un retour d'information plus rapide et moins de risques d'erreurs.

9.Question

Comment les tests d'acceptation devraient-ils être écrits selon le chapitre ?

Réponse:Idéalement, les tests d'acceptation devraient être écrits collaborativement par les parties prenantes et la QA, avec la revue des développeurs. Ils devraient être créés tard dans le cycle de développement et mis à jour si nécessaire pour maintenir clarté et pertinence.

10.Question

Pourquoi n'est-il pas suffisant d'avoir uniquement des tests unitaires au lieu de tests d'acceptation ?

Réponse:Les tests unitaires et les tests d'acceptation ont des objectifs différents : les tests unitaires se concentrent sur le fonctionnement interne du système du point de vue du programmeur, tandis que les tests d'acceptation définissent le comportement du système du point de vue commercial. Les deux sont cruciaux pour une compréhension et une validation complètes.

11.Question

Pouvez-vous détailler les défis de la spécification des interfaces graphiques (GUIs) dans les tests d'acceptation ?

Réponse:Spécifier les GUIs à l'avance est difficile en raison de leur nature subjective et des changements fréquents. Au lieu de se concentrer sur les aspects visuels, les tests d'acceptation devraient interagir avec les capacités sous-jacentes du système, traitant la GUI comme une API lorsque cela est possible pour éviter la frailité.

12.Question

Quelle est l'importance de l'intégration continue (CI) dans les tests ?

Réponse:L'Intégration Continue garantit que les tests sont exécutés fréquemment et automatiquement chaque fois que des modifications de code sont apportées. Cela permet un retour d'information rapide sur la qualité du code, et tout test échoué doit être traité comme une urgence pour une résolution rapide afin de maintenir l'intégrité du code.

13.Question

Quel est le bénéfice ultime d'écrire des tests d'acceptation automatisés comme décrit dans le chapitre ?

Réponse:Les tests d'acceptation automatisés éliminent l'ambiguïté dans les exigences, servent de document formel des exigences et fournissent une méthode claire pour vérifier si le logiciel développé répond aux besoins des parties prenantes, améliorant ainsi la communication et le succès du projet.

Chapitre 29 | 8 Stratégies de test| Questions et réponses

1.Question

Quelle est l'importance d'une stratégie de test pour les équipes de développement professionnelles ?

Réponse:Une bonne stratégie de test est essentielle car elle ne se limite pas à écrire quelques tests unitaires ou tests d'acceptation. Elle garantit une assurance qualité complète tout au long du processus de développement, conduisant à un logiciel qui respecte les normes de performance et de fiabilité attendues. Elle intègre différents types de tests (unitaires, composants, intégration, système et exploratoire) pour aborder systématiquement la qualité et la robustesse du logiciel.

2.Question

Comment devrait-on caractériser la relation entre les équipes QA et Développement ?

Réponse:La QA et le Développement doivent travailler en collaboration plutôt que de manière antagoniste. La relation idéale repose sur la QA agissant en tant que spécificateurs qui traduisent les exigences commerciales en tests d'acceptation automatisés et en tant que caractérisateurs qui explorent les comportements réels du système. Ce travail

d'équipe vise l'objectif que la QA ne découvre rien de problématique, soulignant ainsi que la qualité est une responsabilité partagée.

3.Question

Quels sont les composants de la pyramide d'automatisation des tests ?

Réponse:La pyramide d'automatisation des tests se compose de plusieurs couches : à la base se trouvent les tests unitaires (les tests les plus fondamentaux se concentrant sur des composants individuels), suivis des tests de composants (couvrant les règles métier au sein des composants individuels), puis les tests d'intégration (testant la manière dont les composants fonctionnent ensemble), et enfin les tests systèmes (s'assurant que l'ensemble du système fonctionne correctement). Au sommet se trouvent les tests manuels exploratoires, où les testeurs humains interagissent de manière créative avec le système pour déceler des problèmes.

4.Question

Quel est le rôle des tests unitaires dans le processus de développement ?

Réponse:Les tests unitaires sont cruciaux car ils sont rédigés par les développeurs pour garantir que les sections individuelles du code (unités) fonctionnent comme prévu avant d'être intégrées dans des systèmes plus larges. Ils offrent une couverture presque totale, permettant aux développeurs de détecter les problèmes tôt et de s'assurer que leur code répond aux exigences spécifiées.

5.Question

Quel est l'objectif des tests manuels exploratoires ?

Réponse:L'objectif des tests manuels exploratoires est d'identifier des comportements inattendus et de confirmer le fonctionnement attendu du système à travers l'intuition et la créativité humaines. Il ne s'agit pas d'atteindre une couverture, mais plutôt de s'assurer que le logiciel fonctionne bien dans un scénario réel.

6.Question

Comment les équipes de développement peuvent-elles

atteindre l'objectif selon lequel la QA ne devrait trouver rien de problématique ?

Réponse:Pour atteindre cet objectif, les équipes de développement doivent mettre en œuvre une stratégie de test structurée impliquant une collaboration avec la QA pour créer et exécuter une hiérarchie de tests (unitaires, composants, intégration, système et exploratoires). L'exécution fréquente des tests fournit un retour d'information immédiat et aide à maintenir des normes élevées de qualité du code tout au long du cycle de développement.

7.Question

Que doivent réfléchir les équipes lorsque la QA trouve un bug ?

Réponse:Lorsque la QA découvre un bug, l'équipe de développement doit réagir avec préoccupation, se demandant comment le bug s'est produit. Cette réflexion devrait mener à une analyse des processus de test et de la qualité du code, suivie de la mise en œuvre de mesures préventives pour éviter

des problèmes similaires à l'avenir.

Chapitre 30 | 9 Gestion du Temps| Questions et réponses

1.Question

Quelles stratégies peuvent être mises en œuvre pour gérer efficacement le temps dans un environnement professionnel ?

Réponse:Dans l'environnement chaotique du développement logiciel, il est crucial d'adopter une approche structurée de la gestion du temps. Une stratégie efficace est d'allouer des blocs de temps spécifiques aux tâches, comme l'indique <CODER PROPREMENT>. Par exemple, utiliser un emploi du temps divisé en intervalles de 15 minutes peut contribuer à maximiser la productivité en assignant des tâches précises durant ces périodes. De plus, incorporer des périodes tampons dans le calendrier permet de gérer les interruptions sans perdre de vue les activités essentielles. Se lever tôt pour assurer un temps ininterrompu avant le chaos de la journée

peut également améliorer la concentration et l'efficacité.

2.Question

Comment devrait-on gérer les réunions pour s'assurer qu'elles soient un usage productif du temps ?

Réponse:Comprendre la nature ambivalente des réunions est vital. Les réunions sont nécessaires, mais entraînent souvent un temps perdu. Pour gérer judicieusement sa présence aux réunions, n'acceptez que les invitations qui sont cruciales pour votre travail. Si une réunion manque d'objectifs clairs ou dérive dans des discussions non pertinentes, n'hésitez pas à partir poliment. Avoir un ordre du jour bien défini et un objectif fixé est la clé pour mener des réunions productives, garantissant que le temps des participants est utilisé efficacement.

3.Question

Qu'est-ce que le 'focus-manna' et comment peut-on le préserver tout en travaillant ?

Réponse:Le focus-manna est une métaphore pour désigner

l'énergie mentale et la concentration nécessaires pour un travail profond et productif. Pour protéger cette ressource, les professionnels doivent reconnaître leurs moments de concentration optimale et planifier des tâches exigeantes en conséquence. Éviter les distractions, prévoir des pauses pour se ressourcer et gérer sa consommation de caféine sont des pratiques essentielles. Il est également important d'équilibrer des périodes de concentration intense avec des activités qui aident à restaurer l'énergie mentale, comme l'exercice ou le temps passé en pleine nature.

4.Question

Quels sont les signes d'alerte de 'l'inversion des priorités' dans un environnement de travail ?

Réponse:L'inversion des priorités se produit lorsque des tâches moins importantes sont privilégiées par rapport à des missions plus critiques, souvent à cause de la peur ou de l'inconfort face au défi principal. Les signes d'alerte incluent le temps excessif passé sur des tâches triviales, le fait de se convaincre de fausses urgences ou l'accumulation d'affaires

inachevées tout en évitant des contributions significatives à un projet. Pour combattre cela, il est essentiel de maintenir une clarté sur les priorités des tâches et de se tenir responsable d'aborder directement les défis les plus pressants.

5.Question

Comment les développeurs peuvent-ils reconnaître et naviguer à travers des 'impasses' et des 'désordres' dans le développement logiciel ?

Réponse:Les développeurs doivent être vigilants et flexibles dans leur approche pour éviter de se coincer dans des impasses, où ils s'engagent excessivement dans une idée ou un design qui ne mènera pas à des résultats fructueux. Des outils comme des revues de code fréquentes ou des discussions avec des pairs peuvent fournir des informations pour identifier ces chemins sans sortie tôt. En ce qui concerne les désordres, reconnaître les signes—comme la baisse de productivité ou l'excès de complexité—et agir rapidement pour refactoriser ou simplifier la base de code peut aider à atténuer les dommages à long terme.

6.Question

Qu'est-ce que la Technique Pomodoro et comment peut-elle améliorer la gestion du temps ?

Réponse:La Technique Pomodoro est une méthode de gestion du temps qui utilise des intervalles de travail concentré, traditionnellement de 25 minutes, appelés 'tomates'. Après chaque intervalle, prenez une courte pause pour vous ressourcer. Cette approche structurée aide à minimiser les distractions et favorise une productivité accrue en créant un sentiment d'urgence et de concentration au sein des blocs de temps définis. Même suivre combien de tomates sont complétées chaque jour peut fournir des aperçus sur la productivité et aider à identifier les schémas de concentration et de distraction.

Lire, Partager, Autonomiser

Terminez votre défi de lecture, faites don de livres aux enfants africains.

Le Concept

BOOKS FOR AFRICA × ×

Cette activité de don de livres se déroule en partenariat avec Books For Africa. Nous lançons ce projet car nous partageons la même conviction que BFA : Pour de nombreux enfants en Afrique, le don de livres est véritablement un don d'espoir.

La Règle

Gagnez 100 points → **Échangez un livre** → **Faites un don à l'Afrique**

Votre apprentissage ne vous apporte pas seulement des connaissances mais vous permet également de gagner des points pour des causes caritatives ! Pour chaque 100 points gagnés, un livre sera donné à l'Afrique.

Chapitre 31 | 10 Estimation| Questions et réponses

1.Question

Pourquoi l'estimation est-elle considérée comme terrifiante pour les professionnels du logiciel ?

Réponse:L'estimation est effrayante car elle impacte significativement la valeur commerciale et la réputation des développeurs. Elle est souvent à l'origine de la méfiance entre les parties prenantes commerciales et les développeurs, entraînant anxiété et échec.

2.Question

Quelle est la différence cruciale entre la façon dont les entreprises et les développeurs perçoivent les estimations ?

Réponse:Les entreprises considèrent les estimations comme des engagements qu'il faut réaliser, tandis que les développeurs les voient comme des suppositions sans obligation.

3.Question

Quelles sont les conséquences de faire un engagement

dans le développement logiciel ?

Réponse:S'engager à respecter une échéance signifie qu'on doit y parvenir, souvent au détriment de son temps personnel et de sa famille, ce qui entraîne une pression immense et un risque de dommages à la réputation si l'engagement n'est pas respecté.

4.Question

Comment les développeurs devraient-ils communiquer leurs estimations pour éviter les malentendus ?

Réponse:Les développeurs devraient communiquer clairement la distribution de probabilité de leurs estimations, indiquant à quel point ils croient que la tâche pourra être accomplie dans un certain délai.

5.Question

Quelle est l'importance de la technique PERT dans l'estimation ?

Réponse:La méthode d'évaluation et d'examen des programmes (PERT) permet aux développeurs de fournir un moyen structuré de calculer les estimations sous forme de

distributions de probabilité, aidant ainsi à gérer les attentes efficacement.

6.Question

Comment le travail d'équipe peut-il améliorer la précision des estimations dans les projets logiciels ?

Réponse:L'intuition collective et l'expérience des membres de l'équipe grâce à des techniques comme le large Delphi peuvent conduire à des estimations plus précises. La discussion et la construction d'un consensus aident à affiner les estimations sur une série d'itérations.

7.Question

Quelle est la loi des grands nombres et comment s'applique-t-elle à l'estimation ?

Réponse:La loi des grands nombres suggère qu'estimer de nombreuses petites tâches individuellement et additionner leurs estimations donne une plus grande précision que d'estimer une tâche plus grande dans son ensemble. Cela aide à atténuer les erreurs individuelles.

8.Question

Comment les professionnels abordent-ils les engagements

et les estimations dans le développement logiciel ?

Réponse:Les développeurs professionnels s'efforcent de ne pas faire d'engagements dont ils ne sont pas sûrs. Au lieu de cela, ils fournissent des estimations probabilistes qui incluent des délais d'achèvement attendus et des variations possibles, garantissant une communication plus claire.

9.Question

Que peut-il se passer si les développeurs manquent une estimation ?

Réponse:Manquer une estimation n'est pas perçu comme déshonorant, mais cela peut entraîner une perception négative si cela se traduit par un non-respect des délais de projet. Cela souligne l'importance de communiquer clairement que les estimations ne sont pas des engagements.

10.Question

Comment des techniques comme le Planning Poker peuvent-elles améliorer le processus d'estimation ?

Réponse:Des techniques comme le Planning Poker engagent l'équipe dans un processus d'estimation collaboratif,

permettant à chacun d'apporter ses idées et d'arriver à un consensus qui reflète un jugement collectif plus précis.

Chapitre 32 | 11 La Pression| Questions et réponses

1.Question

Comment un professionnel doit-il se comporter sous pression lors de situations critiques ?

Réponse:Un professionnel doit rester calme et décisif, donner des ordres clairs et précis, s'en tenir à sa formation et garder son sang-froid, plutôt que de céder au stress et au chaos.

2.Question

Quel a été le tournant dans la vie professionnelle de l'auteur ?

Réponse:Le tournant est survenu lorsque l'auteur a réalisé, après un moment de réflexion durant une promenade, qu'il n'était pas heureux de son comportement ni de l'environnement chargé de pression. Il a décidé de cesser de travailler de longues heures et d'éviter les tendances destructrices de crier et de créer du code en désordre.

3.Question

Quelle est l'une des meilleures stratégies pour éviter la pression dans les environnements de travail ?

Réponse:La meilleure stratégie est d'éviter de s'engager à respecter des délais irréalistes et de s'assurer que le risque est quantifié et communiqué à l'entreprise.

4.Question

Que faire lorsque les délais deviennent écrasants ?

Réponse:Ne paniquez pas. Au lieu de cela, ralentissez pour réfléchir à la situation, communiquez avec votre équipe sur les problèmes et appuyez-vous sur vos disciplines établies pour naviguer dans la situation.

5.Question

Pourquoi est-il important de suivre constamment les disciplines, surtout en cas de crise ?

Réponse:Suivre vos disciplines établies en cas de crise renforce votre croyance en ces méthodes comme efficaces. Si vous les abandonnez en période difficile, cela indique un manque de confiance dans leur efficacité.

6.Question

Quel rôle la communication joue-t-elle lors de situations de forte pression ?

Réponse:La communication est cruciale ; faire savoir à votre équipe et à vos supérieurs les problèmes rencontrés et demander leur avis aide à gérer les attentes et réduit les surprises, qui peuvent aggraver la pression.

7.Question

Comment le travail en binôme avec un autre développeur peut-il aider sous pression ?

Réponse:La programmation en binôme permet de résoudre les problèmes de manière collaborative, où les partenaires peuvent se soutenir mutuellement, maintenir leur concentration et s'assurer de respecter les meilleures pratiques, réduisant ainsi la probabilité d'erreurs et améliorant la productivité.

8.Question

Quelle est la perspective de l'auteur sur les pratiques 'rapides et sales' ?

Réponse:L'auteur pense que 'rapides et sales' est un oxymore

; un travail désordonné vous ralentira inévitablement et aura des conséquences négatives à long terme.

9.Question

Quelles sont les stratégies clés mentionnées pour gérer efficacement la pression ?

Réponse:Évitez la pression autant que possible en gérant vos engagements et en maintenant des pratiques propres, et lorsque la pression est inévitable, restez calme, communiquez, appuyez-vous sur vos disciplines et demandez de l'aide.

10.Question

Comment l'auteur suggère-t-il de maintenir la qualité de la production sous pression ?

Réponse:En gardant les systèmes, le code et le design aussi clairs que possible, les professionnels peuvent éviter la production 'en désordre' qui conduit souvent à une pression supplémentaire et à des délais non respectés.

Chapitre 33 | 12 Collaboration| Questions et réponses

1.Question

Pourquoi la collaboration est-elle soulignée dans les équipes de développement logiciel ?

Réponse:La collaboration dans le développement logiciel est cruciale car la plupart des logiciels sont créés par des équipes. Une collaboration efficace entre les membres de l'équipe améliore la résolution de problèmes, la qualité du code et garantit que tout le monde travaille vers les mêmes objectifs commerciaux. Cet effort collectif conduit à des processus plus efficaces et de meilleurs résultats, bénéficiant finalement à la productivité et au succès global du projet.

2.Question

Que veut dire l'auteur en disant que les programmeurs préfèrent souvent travailler seuls ?

Réponse:L'auteur suggère que de nombreux programmeurs sont introvertis et apprécient la concentration solitaire de la programmation. Bien que certains puissent préférer la clarté et la prévisibilité des interactions avec les machines par

rapport aux relations humaines compliquées, cet état d'esprit peut entraver le travail d'équipe efficace et la collaboration qui sont essentielles dans des contextes professionnels.

3.Question

Quels défis l'auteur et son ami ont-ils rencontrés lors de l'optimisation de leur générateur de renvois ?

Réponse:L'auteur et son ami ont rencontré des difficultés avec les performances de leur générateur de renvois, écrivant d'abord un code inefficace et lent. Ils ont expérimenté divers structures de données et algorithmes sans connaissances avancées, affrontant les difficultés d'optimisation des performances par essais et erreurs, ce qui a mené à de la frustration mais finalement à un apprentissage significatif.

4.Question

Comment l'expérience de l'auteur chez Outboard Marine Corp. a-t-elle façonné sa compréhension du professionnalisme ?

Réponse:L'expérience de l'auteur chez Outboard Marine Corp. a été déterminante car il a initialement négligé l'importance de la politique interne et des objectifs

commerciaux. Être licencié l'a amené à reconnaître la nécessité de professionnalisme dans le développement logiciel, soulignant l'importance d'être présent, de respecter les délais et de comprendre le contexte commercial dans lequel il travaillait.

5.Question

Que signifie pour l'auteur la 'propriété collective' dans les équipes de programmation ?

Réponse:'La propriété collective' fait référence à la pratique de partager le code entre tous les membres de l'équipe, plutôt que d'avoir une propriété individuelle sur des parties spécifiques. Cette approche favorise la collaboration, permet d'avoir des perspectives diverses sur le code et encourage une responsabilité partagée pour tous les aspects du projet, réduisant les silos de code et encourageant l'apprentissage en équipe.

6.Question

Pourquoi l'auteur prône-t-il le pair programming ?

Réponse:L'auteur prône le pair programming car il renforce

la collaboration, augmente l'efficacité et facilite le partage des connaissances entre les membres de l'équipe. Le pair programming permet une revue de code immédiate, réduit les 'silos de connaissances' et aide les membres de l'équipe à apprendre les uns des autres, menant finalement à un meilleur développement logiciel.

7.Question

Quelle est la position de l'auteur sur la communication au sein des équipes de programmation ?

Réponse:L'auteur estime que la communication efficace au sein des équipes de programmation est essentielle. Il souligne la nécessité pour les membres de l'équipe de communiquer ouvertement et fréquemment, de partager leurs frustrations et leurs idées, ce qui peut conduire à une meilleure collaboration et à une résolution de problèmes, contrairement à l'idée que travailler seul est plus productif.

8.Question

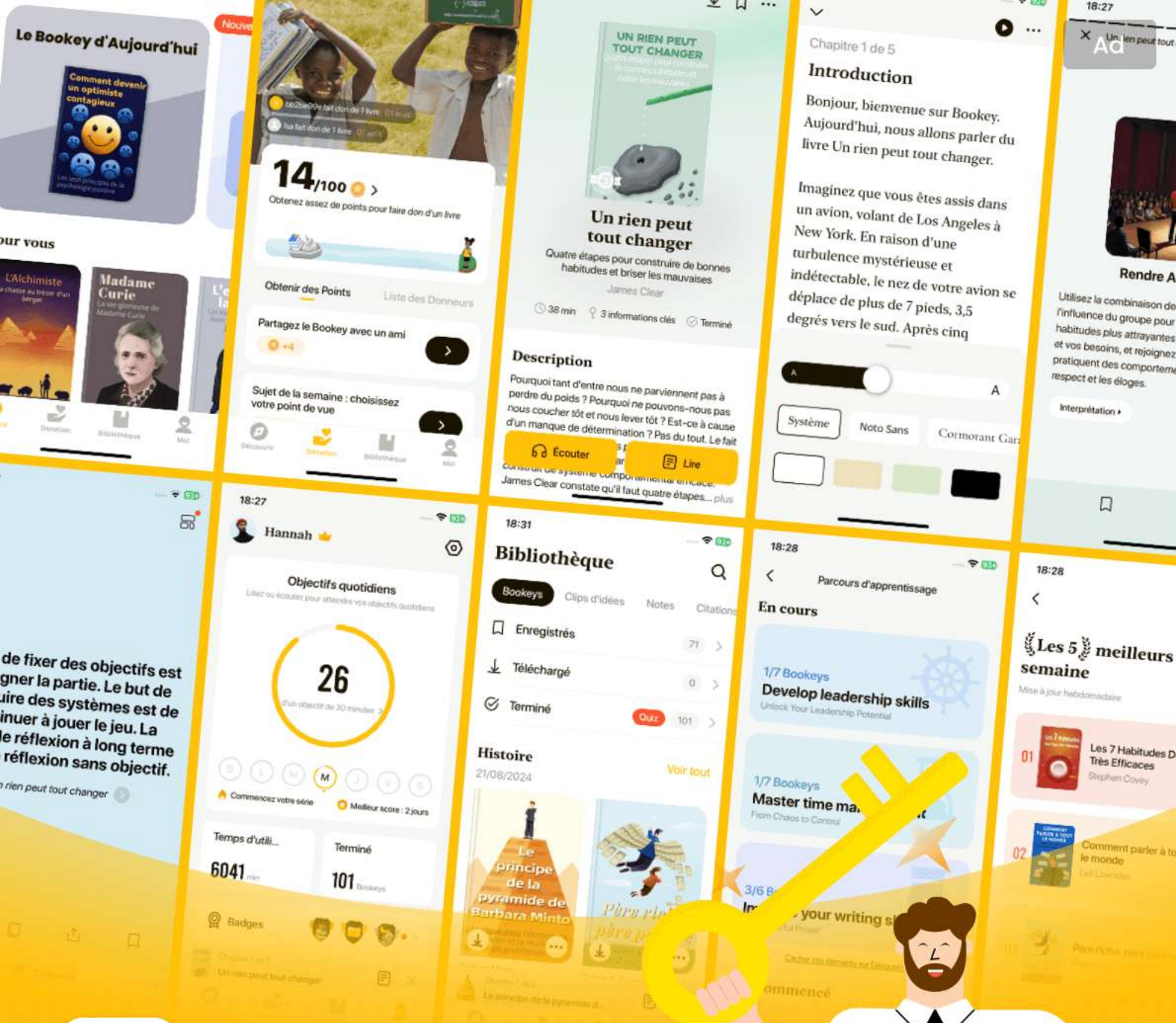
En quoi l'anecdote sur le cervelet est-elle liée à la collaboration en équipe ?

Réponse:L'anecdote sur le cervelet illustre comment une communication inefficace peut mener à l'isolement parmi les programmeurs. L'auteur l'utilise pour souligner l'importance de se faire face et de s'engager dans une interaction directe pour favoriser la collaboration. Frotter les 'cervelets' symbolise le style de travail déconnecté qui entrave le travail d'équipe, tandis que la communication directe promeut une équipe plus forte et cohésive.

9.Question

Quelle conclusion l'auteur tire-t-il des programmeurs et de leur besoin d'interagir avec les autres ?

Réponse:L'auteur conclut que malgré la préférence de nombreux programmeurs pour un travail solitaire, la programmation est intrinsèquement une profession collaborative. Pour réussir et apprécier leur travail, les programmeurs doivent apprendre à interagir avec leurs collègues et le monde des affaires, soulignant que la communication et la collaboration efficaces sont intégrales à l'amélioration des performances individuelles et d'équipe.



Les meilleures idées du monde débloquent votre potentiel

Essai gratuit avec Bookey

Télécharger dans App Store

DISPONIBLE SUR Google Play

Scanner pour télécharger

Chapitre 34 | 13 Équipes et Projets| Questions et réponses

1.Question

Quel est le principal problème d'assigner des programmeurs à plusieurs projets simultanément ?

Réponse:Le principal problème est que le fait de diviser le temps des programmeurs entre plusieurs projets les empêche de former une équipe cohésive. Cela dilue leur efficacité, car ils ne peuvent pas se consacrer pleinement à un projet, ce qui entraîne des inefficacités et une mauvaise collaboration.

2.Question

Qu'est-ce qu'une 'équipe soudée' et pourquoi est-elle importante ?

Réponse:Une équipe soudée est un groupe cohérent d'individus qui ont développé de fortes relations, comprennent les forces et les faiblesses des uns et des autres, et collaborent efficacement. Cette synergie leur permet de résoudre les problèmes plus efficacement, de se soutenir mutuellement et de produire un travail de meilleure qualité.

3.Question

Quelles sont les caractéristiques idéales d'une équipe soudée en termes de composition et de gestion de projet ?

Réponse:Une équipe soudée idéale se compose d'environ une douzaine de membres, incluant généralement des programmeurs, des testeurs, des analystes, et un chef de projet, souvent avec un ratio de 2:1 de programmeurs par rapport aux testeurs. Cette configuration permet un fonctionnement fluide et une exécution efficace des projets.

4.Question

Pourquoi est-il plus efficace de former des équipes autour des personnes plutôt que des projets ?

Réponse:Former des équipes autour d'individus établis permet une meilleure collaboration et continuité. Une équipe qui a réussi à se souder peut gérer plusieurs projets plus efficacement car ses membres peuvent adapter leur flux de travail, prioriser les tâches et tirer parti de leur chimie établie.

5.Question

Comment la direction peut-elle allouer efficacement des ressources entre plusieurs projets tout en travaillant avec

des équipes soudées ?

Réponse:La direction peut s'appuyer sur la vélocité de l'équipe—mesurée en points de travail accomplis dans le temps—pour fixer des objectifs réalistes pour chaque projet. En cas d'urgence, les priorités peuvent être rapidement réajustées grâce à la familiarité de l'équipe et à sa capacité à réallouer les efforts de manière fluide.

6.Question

Que doivent considérer les propriétaires de projet lorsqu'ils travaillent avec des équipes soudées ?

Réponse:Les propriétaires de projet doivent comprendre que bien qu'ils puissent perdre un certain contrôle sur les ressources dédiées, la flexibilité gagnée en ayant des équipes soudées permet aux entreprises de prioriser les projets en fonction des besoins immédiats, au bénéfice ultime des résultats des projets.

7.Question

Quel message clé les organisations doivent-elles retenir concernant la formation d'équipes par rapport à l'exécution de projets ?

Réponse:Les organisations doivent privilégier la formation d'équipes stables et persistantes susceptibles de se souder avec le temps, plutôt que de réorganiser fréquemment les équipes autour de différents projets. Cette approche maximise la productivité et favorise un meilleur environnement de travail.

Chapitre 35 | 14 Mentorat, Apprentissages, et Artisanat| Questions et réponses

1.Question

Pourquoi de nombreux diplômés en informatique, selon l'auteur, ne sont-ils pas bien préparés à la programmation dans le monde réel malgré leur formation ?

Réponse:De nombreux diplômés en informatique manquent de compétences fondamentales en programmation car ils ne suivent souvent pas de cours de programmation durant leurs études. Ceux qui excellent se sont généralement auto-formés avant et pendant leur séjour à l'université. De plus, il existe un décalage entre ce qui est appris dans les milieux académiques et les besoins pratiques de

l'industrie.

2.Question

Quelles anecdotes l'auteur fournit-il pour illustrer son intérêt précoce pour la programmation et comment ont-elles influencé sa carrière ?

Réponse:L'auteur se souvient d'avoir reçu un jouet Digi-Comp I étant enfant, ce qui a éveillé son intérêt pour la programmation. Après avoir eu du mal à le comprendre par lui-même, il a reçu un manuel qui lui a enseigné l'algèbre booléenne, l'amenant à créer son premier programme. Cette expérience fondamentale, associée à l'observation de techniciens et à l'auto-apprentissage grâce à divers mentors et outils, a solidifié sa passion pour la programmation.

3.Question

En réfléchissant à ses expériences, que suggère l'auteur comme essentiel pour un mentorat efficace en programmation ?

Réponse:L'auteur soutient que le mentorat efficace implique une guidance pratique, de l'observation et l'enseignement des principes fondamentaux à travers la pratique et la théorie. Il

souligne que l'industrie de la programmation a besoin d'un mentorat structuré similaire à celui du domaine médical, où les nouveaux diplômés effectuent une pratique supervisée extensive avant d'être considérés comme pleinement qualifiés.

4.Question

Qu'est-ce que le 'artisanat' dans le contexte de la programmation selon l'auteur ?

Réponse:L'artisanat en programmation fait référence à un état d'esprit centré sur la qualité, la compétence et le professionnalisme. Il englobe des techniques, des valeurs et le perfectionnement des compétences à travers l'observation et le temps. L'artisanat s'apprend et se transmet à travers le mentorat et les interactions entre pairs.

5.Question

Comment l'auteur suggère-t-il que la profession de programmation peut améliorer l'intégration de nouveaux développeurs dans l'industrie ?

Réponse:L'auteur suggère un modèle d'apprentissage structuré où les nouveaux diplômés travaillent en étroite

collaboration avec des programmeurs expérimentés (compagnons et maîtres), apprenant par supervision directe, programmation en binôme et retours réguliers pour développer à la fois des compétences et des valeurs professionnelles.

6.Question

Que veut dire l'auteur lorsqu'il dit que l'artisanat est une 'contagion' ?

Réponse:Il utilise 'contagion' au sens métaphorique pour expliquer que l'état d'esprit de l'artisanat se propage par l'observation et l'interaction avec des professionnels compétents. Ce n'est pas enseigné par des conférences, mais par le biais de l'observation et de l'engagement dans la pratique d'un travail qualifié.

7.Question

En fin de compte, quel appel à l'action l'auteur adresse-t-il à ceux de l'industrie du logiciel ?

Réponse:L'auteur appelle les développeurs de logiciels expérimentés à assumer la responsabilité de mentorat de la

prochaine génération, établissant une culture de formation, de guidance et de développement intentionnel des compétences en programmation.

Chapitre 36 | A : Outils| Questions et réponses

1.Question

Quelles sont les leçons clés tirées de l'expérience de l'auteur avec les systèmes de gestion de code source dans les années 1970 ?

Réponse:L'auteur souligne l'importance de la fiabilité et de la redondance dans la gestion du code source. Les défis rencontrés avec les systèmes à bande, tels que les erreurs de lecture/écriture, ont enseigné la nécessité d'avoir des processus de sauvegarde en place, comme le maintien de paires de bandes pour vérification, réfléchissant à l'importance cruciale de gérer les risques associés aux outils de développement logiciel.

2.Question

Pourquoi l'auteur préfère-t-il les outils open-source pour le contrôle de code source plutôt que les systèmes

commerciaux ?

Réponse:L'auteur soutient que les outils open-source sont généralement plus efficaces car ils sont conçus par des développeurs pour les développeurs, ce qui garantit qu'ils répondent aux besoins réels avec des fonctionnalités pratiques telles que la rapidité et la fiabilité. En revanche, les outils commerciaux se concentrent souvent davantage sur les ventes à la direction plutôt que de fournir ce dont les développeurs ont réellement besoin.

3.Question

Quelle est l'importance de passer du verrouillage pessimiste aux pratiques modernes de contrôle de source ?

Réponse:Cette transition reflète une amélioration majeure dans le développement collaboratif. Dans le passé, des systèmes comme des épingles colorées pour verrouiller des fichiers limitaient le développement parallèle. Des outils modernes comme git permettent une édition concurrente sans verrouillage, permettant des flux de travail plus rapides et

plus efficaces tout en aidant les développeurs à gérer les fusions automatiquement.

4.Question

Comment l'approche de l'auteur concernant l'utilisation des IDE a-t-elle évolué au fil du temps ?

Réponse:Au départ, l'auteur préférait des éditeurs comme Emacs et vi, mais il a évolué vers IntelliJ en raison de ses fonctionnalités robustes adaptées aux besoins de développement moderne. Cette évolution souligne l'importance d'exploiter des outils qui améliorent la productivité et la qualité du code grâce à des fonctionnalités avancées.

5.Question

Quelle est la philosophie centrale derrière l'approche de l'auteur en matière de builds continus et de développement piloté par les tests ?

Réponse:L'auteur insiste sur le maintien d'une 'build fonctionnelle' à tout moment, avec un accent sur un retour d'information immédiat chaque fois que du code est intégré. En veillant à ce que tous les tests passent avant de s'engager,

cette pratique favorise la responsabilité et encourage les développeurs à produire un code fiable en continu.

6.Question

Quels éclairages l'auteur fournit-il concernant la misconception de l'MDA (Architecture Dirigée par les Modèles) ?

Réponse:L'auteur critique l'hypothèse de l'MDA selon laquelle remplacer le code par des diagrammes de haut niveau pourrait éliminer la gestion des détails. Il souligne que la programmation implique intrinsèquement la gestion de détails complexes qui ne peuvent pas être facilement abstraits, renforçant que le détail définit l'art de la programmation.

7.Question

Comment l'auteur caractérise-t-il son kit d'outils actuel, et quels outils met-il en avant ?

Réponse:L'auteur décrit son kit d'outils comme épuré, comprenant git pour le contrôle de source, Tracker pour le suivi des problèmes, Jenkins pour le build continu, et IntelliJ pour le codage. Cela reflète une croyance en l'utilisation

d'outils efficaces et puissants qui aident les développeurs à se concentrer sur la programmation de qualité plutôt que de s'enliser dans des systèmes hérités encombrants.

8.Question

Quels principes devraient guider la sélection des systèmes de suivi des problèmes selon l'auteur ?

Réponse:L'auteur préconise de commencer par des méthodes simples et manuelles de suivi des problèmes pour comprendre les besoins de l'équipe avant de passer à des systèmes plus complexes. Il souligne que le suivi des problèmes doit rester gérable, plaidant pour une liste limitée d'éléments exploitables plutôt que des bases de données écrasantes de bogues.

9.Question

Quel rôle l'auteur suggère-t-il que jouent les outils de test unitaire dans le développement logiciel ?

Réponse:Les outils de test unitaire sont essentiels pour garantir un retour d'information rapide sur la qualité du code. L'auteur insiste sur l'importance de la facilité d'utilisation et

des résultats clairs de réussite/échec des tests, promouvant une culture de responsabilité partagée pour maintenir l'intégrité du code tout au long du processus de développement.

10.Question

Comment l'auteur intègre-t-il les exigences commerciales avec la technologie via des outils de test de composants comme FITNESSE ?

Réponse:FITNESSE permet la collaboration entre les parties prenantes commerciales et les développeurs en permettant d'écrire des spécifications dans un format facilement compréhensible. Cet alignement entre les perspectives techniques et commerciales garantit que toutes les parties ont une clarté sur ce qui constitue une fonctionnalité réussie, comblant efficacement le fossé entre les besoins commerciaux et techniques.

Chapitre 37 | Index| Questions et réponses

1.Question

Quelle est l'importance des tests d'acceptation dans le développement logiciel ?

Réponse:Les tests d'acceptation sont cruciaux car ils définissent les critères de succès du point de vue du client. Ils guident les développeurs dans la compréhension des besoins du client et garantissent que le code écrit répond à ces besoins. De plus, ils facilitent la communication entre les développeurs et les parties prenantes, en étant alignés sur les pratiques d'intégration continue.

2.Question

Comment le concept de collaboration améliore-t-il les résultats d'un projet ?

Réponse:La collaboration favorise le travail d'équipe et peut conduire à une meilleure qualité de code et à des solutions innovantes. Elle encourage le partage des connaissances, conduisant à un sentiment de responsabilité collective pour le

projet. Lorsque les membres de l'équipe travaillent bien ensemble, cela réduit les malentendus et améliore l'efficacité globale du processus de développement.

3.Question

Quel rôle joue la discipline dans l'artisanat du logiciel ?

Réponse:La discipline est essentielle pour maintenir l'intégrité du code et respecter les meilleures pratiques. Elle aide à faire des engagements cohérents et à les respecter, renforçant ainsi la confiance dans les compétences de l'équipe et améliorant la qualité du produit.

4.Question

Comment les développeurs peuvent-ils gérer efficacement la pression dans leur environnement de travail ?

Réponse:Une gestion efficace de la pression implique une communication proactive sur les difficultés, l'adoption de techniques de gestion du temps et la priorisation des tâches. Les développeurs devraient également pratiquer la recharge—prendre des pauses et s'éloigner du travail pour prendre du recul, évitant ainsi l'épuisement professionnel.

5.Question

Qu'est-ce que l'approche 'ne pas nuire' et pourquoi est-elle importante ?

Réponse:L'approche 'ne pas nuire' met l'accent sur l'écriture d'un code qui n'introduit pas de nouveaux problèmes ou complexités. Elle est vitale car elle protège l'intégrité structurelle du système tout en promouvant la maintenabilité et la facilité des futures améliorations.

6.Question

Comment le concept de propriété améliore-t-il la qualité du code ?

Réponse:La propriété collective encourage tous les membres de l'équipe à prendre la responsabilité du code, conduisant à une plus grande responsabilité. Cela peut prévenir l'accumulation de 'messes de code' et garantir qu'une haute qualité de code soit maintenue, car chacun est investi et vigilant quant à l'état du projet.

7.Question

Quelle est l'importance de l'apprentissage continu dans la carrière d'un développeur ?

Réponse:L'apprentissage continu permet aux développeurs de se tenir au courant des dernières technologies et pratiques. Il améliore leurs compétences en résolution de problèmes et leur adaptabilité, leur permettant de contribuer plus efficacement à leurs équipes et projets.

8.Question

Pourquoi la communication efficace des exigences est-elle cruciale dans le développement logiciel ?

Réponse:Une communication claire des exigences minimise les ambiguïtés et réduit le risque d'erreurs dans la livraison du projet. Elle assure que toutes les parties prenantes partagent une compréhension des objectifs du projet, entraînant une exécution plus fluide et une satisfaction accrue.

9.Question

Comment accepter l'échec peut-il être bénéfique dans le développement logiciel ?

Réponse:Accepter l'échec fournit des leçons précieuses qui peuvent mener à des succès futurs. Cela encourage une

culture d'expérimentation et d'innovation où les équipes peuvent apprendre de leurs erreurs sans crainte, favorisant la créativité et la résilience dans la résolution de problèmes.

10.Question

Pourquoi l'évitement de la complaisance est-il important dans les pratiques logicielles ?

Réponse:Éviter la complaisance garantit que les équipes recherchent constamment l'amélioration. Cela promeut une approche proactive d'apprentissage et de perfectionnement des processus, ce qui est essentiel pour s'adapter aux technologies changeantes et répondre aux besoins des utilisateurs en évolution.

Scanner pour télécharger

Pourquoi Bookey est une application incontournable pour les amateurs de livres

Contenu de 30min
Plus notre interprétation est profonde et claire, mieux vous saisissez chaque titre.

Format texte et audio
Absorbez des connaissances même dans un temps fragmenté.

Quiz
Vérifiez si vous avez maîtrisé ce que vous venez d'apprendre.

Et plus
Plusieurs voix & polices, Carte mentale, Citations, Clips d'idées...

Essai gratuit avec Bookey

CODER PROPREMENT Quiz et test

Vérifier la réponse correcte sur le site de Bookey

Chapitre 1 | 2 Noms Significatifs| Quiz et test

1. Les noms dans le développement logiciel incluent uniquement des variables et des fonctions.
2. Il est important d'utiliser des noms révélateurs d'intention pour améliorer la clarté du code.
3. Les noms de méthodes devraient obscurcir leurs fonctions pour être plus créatifs et accrocheurs.

Chapitre 2 | 3 Fonctions| Quiz et test

1. Idéalement, les fonctions ne devraient pas avoir plus de trois arguments pour réduire la complexité.
2. Choisir des noms descriptifs pour les fonctions est crucial pour la clarté et la définition des attentes.
3. Utiliser des exceptions au lieu de codes d'erreur conduit à un flux de contrôle plus propre dans les fonctions.

Chapitre 3 | 4 Commentaires| Quiz et test

1. De bons commentaires sont un signe d'échec dans

l'expression de l'intention à travers le code.

2. Les commentaires devraient être utilisés pour remplacer un mauvais code au lieu de le nettoyer.
3. Des commentaires vagues qui encombrent la base de code sont considérés comme une bonne pratique.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 4 | 5 Mise en forme| Quiz et test

1. La mise en forme n'est pas importante en programmation car elle n'affecte pas la lisibilité ou la compréhension.
2. Les fichiers devraient idéalement être limités à 200 lignes pour faciliter la compréhension.
3. Les membres de l'équipe devraient suivre leurs propres styles de mise en forme plutôt qu'un ensemble de règles partagées.

Chapitre 5 | 6 Objets et Structures de Données| Quiz et test

1. Les variables doivent rester publiques pour maintenir la flexibilité des changements.
2. La loi de Demeter suggère qu'un module doit connaître la structure interne des objets qu'il manipule.
3. Les structures hybrides qui combinent des caractéristiques d'objets et de structures de données doivent être évitées dans la conception.

Chapitre 6 | Gestion des erreurs| Quiz et test

1. La gestion des erreurs n'est pas importante en programmation tant que la logique principale est correcte.
2. Lancer des exceptions est préférable à l'utilisation de codes d'erreur car cela améliore la lisibilité du code.
3. Retourner null est une bonne pratique en programmation car cela simplifie la gestion des erreurs.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 7 | 8 Limites| Quiz et test

1. Intégrer du code tiers conduit toujours à une clarté et une maintenabilité du code.
2. Encapsuler des composants tiers dans une classe dédiée améliore la maintenabilité du code.
3. Les tests d'apprentissage sont inutiles lors de l'utilisation de bibliothèques tierces, car elles peuvent être facilement comprises par expérimentation.

Chapitre 8 | 9 Tests Unitaires| Quiz et test

1. Le développement piloté par les tests (TDD) met l'accent sur l'écriture des tests unitaires après avoir écrit le code de production.
2. Les tests ne doivent pas être maintenus avec les mêmes normes que le code de production.
3. Les principes F.I.R.S.T. comprennent le fait de rendre les tests indépendants les uns des autres.

Chapitre 9 | 10 Classes| Quiz et test

1. Les classes doivent commencer par des constantes statiques publiques, suivies de variables statiques

privées, puis de variables d'instance privées, et enfin de fonctions publiques.

2. Il est préférable d'avoir des classes avec plusieurs responsabilités et elles ne doivent pas être refactorisées tant qu'elles sont suffisamment grandes pour gérer toutes les opérations.
3. Maintenir la cohésion dans les classes peut conduire à la création de nombreuses petites classes.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 10 | 11 Systèmes| Quiz et test

1. La complexité est bénéfique au développement logiciel, aidant les produits à être facilement planifiés, construits et testés.
2. L'injection de dépendances (DI) améliore la séparation des préoccupations en déplaçant les responsabilités de gestion des dépendances vers des frameworks externes.
3. Le processus de démarrage doit être intégré à la logique d'opération normale pour améliorer les performances du système.

Chapitre 11 | 12 Émergence| Quiz et test

1. Un design doit garantir que le système fonctionne comme prévu et doit être testé ; les systèmes qui ne peuvent pas être testés ne devraient pas être déployés.
2. Éliminer la duplication dans le code aide à améliorer la clarté et réduit les violations du Principe de Responsabilité Unique (SRP).
3. Avoir autant de petites classes et méthodes que possible est

l'objectif lors de la conception de CODER PROPREMENT.

Chapitre 12 | 13 Concurrence| Quiz et test

1. La concurrence améliore toujours les performances de l'application.
2. Utiliser des copies de données est une pratique recommandée pour éviter les problèmes de données partagées dans la programmation concurrente.
3. Comprendre la concurrence n'est pas nécessaire lorsqu'on utilise la concurrence gérée par les conteneurs.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 13 | 14 Raffinement Successif| Quiz et test

1. La classe Args était initialement bien structurée et facile à maintenir.
2. Ce chapitre encourage l'utilisation du développement piloté par les tests (TDD) durant le processus de refactoring.
3. Le refactoring de la classe Args a abouti à un design plus modulaire avec des responsabilités plus claires.

Chapitre 14 | 15 Les Internes de JUnit| Quiz et test

1. JUnit a été développé uniquement par Kent Beck sans aucune collaboration.
2. Le module `ComparisonCompactor` présente les différences entre deux chaînes de manière claire.
3. La version finale de `ComparisonCompactor` ne suit pas les meilleures pratiques et manque de regroupement logique des fonctions.

Chapitre 15 | 16 Refactorisation de SerialDate| Quiz et test

1. Le refactoring de la classe SerialDate a été initié malgré la grande qualité du code original.

2.Le processus de refactoring a inclus l'amélioration de la couverture des tests en créant une suite de tests plus complète.

3.Le nom de la classe original 'SerialDate' a été maintenu tout au long du processus de refactoring pour éviter toute confusion.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 16 | 17 Odeurs et Heuristiques| Quiz et test

1. Les commentaires ne doivent jamais contenir de données historiques adaptées aux systèmes de versionnement.
2. Les fonctions peuvent avoir un nombre quelconque d'arguments sans affecter la clarté du code.
3. Il est acceptable d'avoir plusieurs langages de programmation dans un seul fichier source pour des raisons de flexibilité.

Chapitre 17 | A : Concurrence II| Quiz et test

1. Le multithreading peut aider à améliorer le débit si l'application est liée au CPU.
2. Les goulets d'étranglement de performance peuvent se produire à la fois en raison de limites d'I/O et d'utilisation du processeur.
3. Les modèles de conception pour la gestion des threads doivent se concentrer uniquement sur la fusion des responsabilités pour simplifier le code.

Chapitre 18 | B: org.jfree.date.SerialDate| Quiz et

test

1. La classe SerialDate est conçue pour être mutable, permettant aux instances d'être modifiées après leur création.
2. SerialDate garantit la compatibilité avec les exigences de gestion des dates d'Excel.
3. La classe SerialDate inclut des méthodes pour retourner le jour de la semaine et pour gérer les conversions de chaînes pour les représentations de dates.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 19 | C : Références croisées des heuristiques| Quiz et test

1. L'annexe fournit une référence croisée détaillée de diverses heuristiques et mauvaises pratiques du livre 'CODER PROPREMENT'.
2. Des heuristiques sont listées dans l'annexe, allant de C1 à T9.
3. Les termes G1 à G10 dans l'annexe font référence à des langages de programmation spécifiques.

Chapitre 20 | Index| Quiz et test

1. Les classes abstraites et les interfaces ne sont pas nécessaires pour une bonne abstraction et une bonne structuration du code.
2. La loi de Demeter souligne le couplage minimal et la facilité de maintenance, mettant en avant l'utilisation de fonctions d'accès.
3. Écrire des tests qui se valident eux-mêmes réduit le nombre de dépendances et n'est pas essentiel pour les pratiques de CODER PROPREMENT.

Chapitre 21 | Introduction Préalable| Quiz et test

1. L'auteur souligne l'importance du professionnalisme en programmation en s'appuyant sur des expériences personnelles spanning over 421 ans.
2. L'auteur a tiré des leçons précieuses sur le professionnalisme et l'importance de quitter un emploi en bons termes après avoir connu le chômage.
3. L'auteur pense que devenir un professionnel est un événement unique qui se termine après avoir acquis quelques expériences initiales.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 22 | 1 Professionnalisme| Quiz et test

1. Les développeurs de logiciels professionnels rejettent principalement la faute sur leurs actions plutôt que de prendre leurs responsabilités.
2. L'apprentissage continu et la pratique sont essentiels pour maintenir un standard professionnel dans le développement de logiciels.
3. Comprendre le domaine dans lequel vous travaillez est optionnel pour un développeur de logiciels professionnel.

Chapitre 23 | 2 Dire Non| Quiz et test

1. Les professionnels devraient toujours accepter les demandes de leurs supérieurs pour être considérés comme des joueurs d'équipe.
2. Dire non est particulièrement important dans des situations à enjeux élevés selon Robert C. Martin.
3. L'auteur considère que les promesses vagues et le concept d''essayer'' sont acceptables dans les milieux professionnels.

Chapitre 24 | 3 Dire Oui| Quiz et test

1. Dire 'faisons' indique un véritable engagement selon le chapitre.
2. Un véritable engagement implique de dire que vous le ferez, de le penser réellement et de le faire.
3. Communiquer les défis lorsque les engagements ne peuvent pas être respectés est inutile et peut entraîner de la confusion.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 25 | 4 Codage| Quiz et test

1. Coder en étant fatigué ou distrait mène à des résultats médiocres.
2. Écouter de la musique améliore toujours la concentration en codant.
3. Une gestion efficace des retards implique des évaluations de progrès honnêtes et basées sur des faits.

Chapitre 26 | 5 Développement Driven par les Tests| Quiz et test

1. Le développement piloté par les tests (TDD) a été largement adopté dans les méthodologies Agile depuis son apparition il y a plus de dix ans.
2. Selon les trois lois du TDD, on peut écrire du code de production avant de créer un test unitaire échouant.
3. Le TDD garantit que les développeurs produiront un code bien conçu sans avoir besoin de refactoring.

Chapitre 27 | 6 Pratiquer| Quiz et test

1. Tous les professionnels pratiquent leur art pour améliorer leurs compétences, y compris les

programmeurs.

2.Le Coding Dojo est un concept qui décourage la collaboration entre programmeurs.

3.S'exercer sur son temps libre est rare pour les programmeurs et n'est pas recommandé pour le développement des compétences.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 28 | 7 Tests d'acceptation| Quiz et test

1. Les développeurs professionnels devraient mettre l'accent sur l'exactitude de la communication avec les membres de l'équipe et les parties prenantes.
2. Les tests d’acceptation ne sont pas importants pour établir des attentes claires et une communication entre les développeurs et les parties prenantes.
3. Les développeurs devraient implémenter des fonctionnalités avant que les tests d’acceptation soient finalisés et acceptés.

Chapitre 29 | 8 Stratégies de test| Quiz et test

1. Une stratégie de test complète n'inclut que les tests unitaires et d'acceptation.
2. Les chasses aux bugs collaboratives favorisent l'engagement de l'équipe et la responsabilité en matière de qualité.
3. L'objectif principal du QA est de s'assurer qu'il trouve des problèmes dans le système.

Chapitre 30 | 9 Gestion du Temps| Quiz et test

1. Les réunions sont toujours une partie nécessaire de la productivité selon Robert C. Martin.
2. La technique Pomodoro consiste à travailler pendant 25 minutes suivies de pauses pour maintenir la concentration.
3. Il est déconseillé de décliner les invitations aux réunions à moins d'avoir des raisons valables.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 31 | 10 Estimation| Quiz et test

1. Les développeurs considèrent les estimations comme des engagements, tandis que les parties prenantes du secteur les voient comme des suppositions éclairées.
2. Une estimation efficace nécessite de comprendre que les estimations représentent une gamme de possibilités plutôt qu'une date fixe.
3. La méthode Wideband Delphi de Barry Boehm implique des discussions en équipe et des estimations séquentielles jusqu'à ce qu'un accord soit atteint.

Chapitre 32 | 11 La Pression| Quiz et test

1. Il est important que les professionnels gardent leur calme sous pression selon le Chapitre 32 de 'CODER PROPREMENT'.
2. Prendre des raccourcis dans les pratiques de codage est une stratégie recommandée pour gérer efficacement la pression.
3. Des pratiques efficaces doivent être maintenues même dans des situations de crise selon les principes exposés dans

'CODER PROPREMENT'.

Chapitre 33 | 12 Collaboration| Quiz et test

1. La plupart des logiciels sont créés par des équipes, et une collaboration efficace est essentielle pour le succès de l'équipe.
2. Les programmeurs trouvent les relations interpersonnelles faciles et agréables, préférant travailler de manière indépendante.
3. La programmation en binôme diminue l'efficacité de la résolution de problèmes et réduit le partage des connaissances entre les programmeurs.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 34 | 13 Équipes et Projets| Quiz et test

1. L'allocation de ressources entre plusieurs petits projets entraîne souvent une meilleure cohésion et efficacité au sein de l'équipe.
2. Une équipe bien rodée se compose généralement d'un mélange de programmeurs, de testeurs, d'analystes et d'un chef de projet, avec une taille optimale d'environ douze membres.
3. Les organisations réussies préfèrent former de nouvelles équipes pour chaque projet plutôt que de construire des équipes autour de structures existantes bien rodées.

Chapitre 35 | 14 Mentorat, Apprentissages, et Artisanat| Quiz et test

1. Robert C. Martin pense que de nombreux diplômés en informatique sont prêts pour des postes de programmation dès la sortie de l'université.
2. Le mentorat structuré dans l'industrie du logiciel est aussi rigoureux que dans la profession médicale selon Martin.

3.L'artisanat dans le logiciel est défini comme l'état d'esprit incarnant compétence, qualité et professionnalisme.

Chapitre 36 | A : Outils| Quiz et test

1. Le verrouillage pessimiste encourage l'édition simultanée entre différents développeurs.
2. Les systèmes de contrôle de version modernes comme git permettent des branches et des fusions spontanées, améliorant ainsi la dynamique de la programmation collaborative.
3. Jenkins n'est pas recommandé pour les processus de build continu en raison de son manque de capacités de retour d'information en temps réel.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play

Chapitre 37 | Index| Quiz et test

1. Les tests d'acceptation ne sont nécessaires qu'à la fin du processus de développement logiciel.
2. Les tests d'acceptation automatisés aident à mieux s'intégrer aux pratiques d'intégration continue.
3. Une communication claire n'est pas importante dans la gestion de projet ; les malentendus sont attendus.

Scannez pour télécharger

Téléchargez l'appli Bookey pour profiter

Plus de 1000 résumés de livres avec des quiz

Essai gratuit disponible !

Télécharger dans App Store

DISPONIBLE SUR Google Play