Analyse et Conception Détaillée des Modules Cartes, Planification et Paiement par Tranches

1. Introduction

Cette analyse présente la conception détaillée de trois modules essentiels de l'application mobile DOHONE : la gestion des cartes de crédit, la planification des opérations financières, et le système de paiement par tranches. La conception s'appuie sur une approche orientée objet intégrant divers design patterns pour assurer la flexibilité, la maintenabilité et l'extensibilité du système, en cohérence avec l'architecture existante du système de gestion de projets.

2. Module Gestion des Cartes de Crédit

2.1 Classe Card (Factory Method, Observer, State)

La classe Card représente l'entité centrale pour la gestion des cartes de crédit. Elle agit comme une Factory pour créer ses opérations associées et comme Observable dans le pattern Observer, notifiant les observateurs enregistrés des changements significatifs. Elle utilise également le pattern State pour gérer les différents états d'une carte.

Attributs :

- id_card (long) : Identifiant unique de la carte (clé primaire).
- card_number (string) : Numéro de la carte (partiellement masqué pour sécurité).
- card_type (string) : Type de carte (VISA, MASTERCARD, DOHONE_PREPAID).
- card_status (enum: ACTIVE, BLOCKED, EXPIRED, PENDING) : Statut actuel de la carte.
- balance (double) : Solde disponible sur la carte.
- credit_limit (double) : Limite de crédit autorisée.
- expiration_date (datetime) : Date d'expiration de la carte.
- holder_name (string) : Nom du porteur de la carte.
- issue_date (datetime) : Date d'émission de la carte.
- last_transaction_date (datetime) : Date de la dernière transaction.
- daily_limit (double) : Limite quotidienne de transaction.
- monthly_limit (double) : Limite mensuelle de transaction.
- id_user (long) : Clé étrangère vers la classe User.
- pin_hash (string) : Hash du code PIN (sécurisé).

- security_code_hash (string) : Hash du code de sécurité.
- observers (list of Observer objects) : Liste des observateurs inscrits pour les notifications.
- currentState (object implementing CardState interface) : État actuel de la carte pour le pattern State.
- operationHistory (list of CardOperation objects) : Historique des opérations effectuées.

Méthodes :

- createOperation(operation_type, amount, description, recipient) : Crée et retourne un nouvel objet CardOperation. Cette méthode incarne le pattern Factory Method, abstrayant l'instanciation des objets d'opération de carte.
- createRechargeOperation(amount, payment_method) : Crée une opération de recharge spécifique.
- createPaymentOperation(amount, merchant_info, payment_details) : Crée une opération de paiement.
- createBalanceCheckOperation() : Crée une opération de consultation de solde.
- changeState(new_card_state_object) : Change l'état interne de la carte en définissant un nouvel objet state, incarnant le pattern State.
- executeCurrentStateAction() : Exécute les actions spécifiques à l'état actuel.
- validateOperation(operation_context) : Valide une opération selon l'état actuel et les limites.
- updateBalance(amount, operation_type) : Met à jour le solde de la carte.
- checkLimits(amount, operation_type) : Vérifie les limites quotidiennes et mensuelles.
- addObserver(observer) : Ajoute un observateur à la liste des observateurs.
- removeObserver(observer) : Supprime un observateur de la liste.
- notifyObservers(message) : Itère à travers les observateurs et appelle leur méthode update.
- generateTransactionReceipt() : Génère un reçu de transaction.
- blockCard(reason) : Bloque la carte pour des raisons de sécurité.
- unblockCard() : Débloque la carte après validation.
- renewCard() : Renouvelle une carte expirée.

2.2 Interface CardState (State Pattern)

L'interface CardState définit les comportements communs pour tous les états possibles d'une carte.

Méthodes :

- canPerformOperation(operation_context) : Détermine si une opération peut être effectuée dans l'état actuel.
- handleOperation(card, operation_context) : Traite une opération spécifique à l'état.
- getAvailableOperations() : Retourne la liste des opérations disponibles dans cet état.
- getStateDescription() : Retourne une description textuelle de l'état.
- validateStateTransition(new_state) : Valide si une transition vers un nouvel état est autorisée.

2.3 Classes Concrètes d'États de Carte

2.3.1 ActiveCardState (implémente CardState)

Représente l'état d'une carte active et fonctionnelle.

Méthodes :

- canPerformOperation(operation_context) : Retourne true pour toutes les opérations standard.
- handleOperation(card, operation_context) : Exécute l'opération après validation des limites et du solde.
- getAvailableOperations() : Retourne toutes les opérations (paiement, recharge, consultation, retrait).

2.3.2 BlockedCardState (implémente CardState)

Représente l'état d'une carte bloquée pour des raisons de sécurité.

Attributs :

- block_reason (string) : Raison du blocage.
- block_date (datetime) : Date du blocage.
- auto_unblock_date (datetime) : Date de déblocage automatique si applicable.

Méthodes :

- canPerformOperation(operation_context) : Retourne true uniquement pour les consultations de solde.
- handleOperation(card, operation_context) : Lance une exception pour les opérations interdites.
- getAvailableOperations() : Retourne uniquement les opérations de consultation.

2.3.3 ExpiredCardState (implémente CardState)

Représente l'état d'une carte expirée.

Méthodes :

- canPerformOperation(operation_context) : Retourne false pour toutes les opérations.
- handleOperation(card, operation_context) : Lance une exception CardExpiredException.
- getAvailableOperations() : Retourne une liste vide.

2.3.4 PendingCardState (implémente CardState)

Représente l'état d'une carte en attente d'activation.

Attributs :

- activation_code (string) : Code d'activation requis.
- pending_since (datetime) : Date de mise en attente.

Méthodes :

- canPerformOperation(operation_context) : Retourne true uniquement pour l'activation.
- handleOperation(card, operation_context) : Traite uniquement les demandes d'activation.

2.4 Classe CardOperation (Template Method, Command)

La classe CardOperation représente une opération effectuée sur une carte. Elle utilise le pattern Template Method pour définir le flux d'exécution standard et le pattern Command pour encapsuler les requêtes.

Attributs :

- id_operation (long) : Identifiant unique de l'opération.
- operation_type (string) : Type d'opération (RECHARGE, PAYMENT, BALANCE_CHECK, WITHDRAWAL).
- amount (double) : Montant de l'opération.
- operation_date (datetime) : Date et heure de l'opération.
- operation_status (enum: PENDING, COMPLETED, FAILED, CANCELLED) : Statut de l'opération.
- description (text) : Description détaillée de l'opération.
- recipient_info (string) : Informations sur le bénéficiaire si applicable.
- transaction_reference (string) : Référence unique de la transaction.
- fees (double) : Frais associés à l'opération.
- id_card (long) : Clé étrangère vers la carte concernée.
- merchant_info (string) : Informations sur le commerçant pour les paiements.

- location_info (string) : Informations de géolocalisation de l'opération.

Méthodes :

- execute() : Template Method définissant le squelette de l'algorithme d'exécution (pre_execution_validation(), perform_operation_action(), post_execution_processing()).
- undo() : Annule l'opération si possible (pattern Command).
- validate() : Valide les paramètres de l'opération.
- calculateFees() : Calcule les frais associés à l'opération.
- logOperation() : Enregistre l'opération dans les logs de sécurité.
- sendNotification() : Envoie une notification de l'opération.
- generateTransactionId() : Génère un identifiant unique pour la transaction.

2.5 Interface CardOperationStrategy (Strategy Pattern)

L'interface CardOperationStrategy permet d'implémenter différentes stratégies pour les opérations de carte.

Méthodes :

- executeOperation(card, operation_context) : Exécute l'opération selon la stratégie définie.
- validateOperation(card, operation_context) : Valide l'opération selon les règles de la stratégie.
- calculateOperationFees(amount, operation_type) : Calcule les frais selon la stratégie.

2.6 Classes de Stratégies d'Opération

2.6.1 RechargeOperationStrategy (implémente CardOperationStrategy)

Stratégie pour les opérations de recharge de carte.

Attributs :

- minimum_recharge_amount (double) : Montant minimum de recharge.
- maximum_recharge_amount (double) : Montant maximum de recharge.
- recharge_fee_percentage (double) : Pourcentage de frais de recharge.

2.6.2 PaymentOperationStrategy (implémente CardOperationStrategy)

Stratégie pour les opérations de paiement.

Attributs :

- merchant_category_rules (object) : Règles spécifiques par catégorie de commerçant.
- fraud_detection_rules (object) : Règles de détection de fraude.

2.6.3 WithdrawalOperationStrategy (implémente CardOperationStrategy)

Stratégie pour les opérations de retrait.

Attributs :

- atm_network_fees (object) : Frais selon le réseau de distributeurs.
- daily_withdrawal_limit (double) : Limite quotidienne de retrait.

3. Module Planification

3.1 Classe Planning (Command, Observer, Template Method)

La classe Planning représente une planification d'opérations financières. Elle utilise le pattern Command pour encapsuler les actions planifiées et le pattern Observer pour notifier les changements.

Attributs :

- id_planning (long) : Identifiant unique de la planification.
- planning_name (string) : Nom de la planification.
- planning_description (text) : Description détaillée de la planification.
- planning_type (enum: TRANSFER, PAYMENT, CARD_RECHARGE, SERVICE_PAYMENT, INSTALLMENT_PAYMENT) : Type de planification.
- scheduled_date (datetime) : Date et heure planifiées pour l'exécution.
- created_date (datetime) : Date de création de la planification.
- last_execution_date (datetime) : Date de dernière exécution.
- next_execution_date (datetime) : Date de prochaine exécution.
- recurrence_pattern (enum: NONE, DAILY, WEEKLY, MONTHLY, YEARLY) : Motif de récurrence.
- recurrence_end_date (datetime) : Date de fin de la récurrence.
- is_active (boolean) : Indique si la planification est active.
- execution_count (int) : Nombre d'exécutions effectuées.
- max_executions (int) : Nombre maximum d'exécutions autorisées.
- amount (double) : Montant de l'opération planifiée.
- recipient (string) : Destinataire de l'opération.
- payment_method (string) : Méthode de paiement préférée.

- id_user (long) : Clé étrangère vers l'utilisateur.
- id_project (long) : Clé étrangère vers un projet (optionnel).
- id_card (long) : Clé étrangère vers une carte (si applicable).
- notification_settings (object) : Paramètres de notification.
- observers (list of Observer objects) : Liste des observateurs pour les notifications.
- scheduledCommands (list of PlanningCommand objects) : Liste des commandes planifiées.

Méthodes :

- scheduleCommand(planning_command) : Planifie une nouvelle commande (pattern Command).
- executeScheduledTasks() : Template Method pour l'exécution des tâches planifiées.
- calculateNextExecutionDate() : Calcule la prochaine date d'exécution selon la récurrence.
- validatePlanningParameters() : Valide les paramètres de la planification.
- checkExecutionConditions() : Vérifie les conditions d'exécution (solde, limites, etc.).
- updateExecutionHistory() : Met à jour l'historique d'exécution.
- pausePlanning() : Met en pause la planification.
- resumePlanning() : Reprend une planification en pause.
- cancelPlanning() : Annule définitivement la planification.
- duplicatePlanning() : Crée une copie de la planification.
- linkToProject(project_id) : Associe la planification à un projet.
- addObserver(observer) : Ajoute un observateur.
- removeObserver(observer) : Supprime un observateur.
- notifyObservers(message) : Notifie tous les observateurs.

3.2 Interface PlanningCommand (Command Pattern)

L'interface PlanningCommand encapsule les commandes planifiées.

Méthodes :

- execute() : Exécute la commande planifiée.
- undo() : Annule l'exécution de la commande si possible.
- isReady() : Vérifie si la commande est prête à être exécutée.
- getDescription() : Retourne une description de la commande.
- getRequiredBalance() : Retourne le solde requis pour l'exécution.

- validateExecution() : Valide les conditions d'exécution.
- logExecution() : Enregistre l'exécution dans les logs.

3.3 Classes Concrètes de Commandes Planifiées

3.3.1 ScheduledTransferCommand (implémente PlanningCommand)

Commande pour les virements planifiés.

Attributs :

- transfer_amount (double) : Montant du virement.
- recipient_phone (string) : Téléphone du destinataire.
- recipient_name (string) : Nom du destinataire.
- transfer_reason (string) : Motif du virement.
- execution_time (datetime) : Heure d'exécution programmée.
- transfer_service (object) : Service de virement à utiliser.

3.3.2 ScheduledCardRechargeCommand (implémente PlanningCommand)

Commande pour les recharges de carte planifiées.

Attributs :

- card_id (long) : Identifiant de la carte à recharger.
- recharge_amount (double) : Montant de la recharge.
- funding_source (string) : Source de financement.
- execution_time (datetime) : Heure d'exécution programmée.
- card_service (object) : Service de carte à utiliser.

3.3.3 ScheduledServicePaymentCommand (implémente PlanningCommand)

Commande pour les paiements de services planifiés.

Attributs :

- service_provider (string) : Fournisseur de service (ENEO, CamWater, etc.).
- account_number (string) : Numéro de compte client.
- payment_amount (double) : Montant du paiement.
- service_type (string) : Type de service (électricité, eau, télévision, etc.).
- execution_time (datetime) : Heure d'exécution programmée.

3.4 Classe PlanningScheduler (Singleton, Observer)

La classe PlanningScheduler gère l'exécution des planifications selon leur calendrier.

Attributs :

- instance (static PlanningScheduler) : Instance unique (pattern Singleton).
- active_plannings (list of Planning objects) : Liste des planifications actives.
- execution_queue (priority queue) : File de priorité des exécutions.
- scheduler_status (enum: RUNNING, PAUSED, STOPPED) : Statut du planificateur.
- execution_thread (Thread) : Thread d'exécution des tâches.

Méthodes :

- getInstance() : Retourne l'instance unique (pattern Singleton).
- addPlanning(planning) : Ajoute une planification au planificateur.
- removePlanning(planning_id) : Supprime une planification du planificateur.
- startScheduler() : Démarre le planificateur.
- stopScheduler() : Arrête le planificateur.
- pauseScheduler() : Met en pause le planificateur.
- processExecutionQueue() : Traite la file d'exécution.
- checkOverduePlannings() : Vérifie les planifications en retard.
- sendExecutionReport() : Envoie un rapport d'exécution.

4. Module Paiement par Tranches

4.1 Classe InstallmentPayment (State, Strategy, Observer)

La classe InstallmentPayment représente un paiement échelonné pour un article. Elle utilise le pattern State pour gérer les différents états du paiement et le pattern Strategy pour les méthodes de paiement.

Attributs :

- id_installment_payment (long) : Identifiant unique du paiement par tranches.
- store_name (string) : Nom de la boutique partenaire.
- store_address (string) : Adresse de la boutique.
- store_contact (string) : Contact de la boutique.

- article_name (string) : Nom de l'article.
- article_description (text) : Description détaillée de l'article.
- article_price (double) : Prix total de l'article.
- article_reference (string) : Référence de l'article.
- article_category (string) : Catégorie de l'article.
- store_gps_location (string) : Coordonnées GPS de la boutique.
- article_photo_path (string) : Chemin vers la photo de l'article.
- initial_payment (double) : Paiement initial (minimum 10%).
- remaining_amount (double) : Montant restant à payer.
- total_paid_amount (double) : Montant total déjà payé.
- subscription_date (datetime) : Date de souscription.
- payment_deadline (datetime) : Date limite de paiement (3 mois maximum).
- last_payment_date (datetime) : Date du dernier paiement.
- payment_plan (object) : Plan de paiement détaillé.
- terms_accepted (boolean) : Acceptation des conditions générales.
- delivery_status (enum: PENDING, READY, DELIVERED) : Statut de livraison.
- delivery_address (string) : Adresse de livraison.
- id_user (long) : Clé étrangère vers l'utilisateur.
- currentState (object implementing InstallmentState interface) : État actuel du paiement.
- paymentStrategy (object implementing PaymentStrategy interface) : Stratégie de paiement.
- observers (list of Observer objects) : Liste des observateurs.
- installmentTransactions (list of InstallmentTransaction objects) : Historique des transactions.
- penalties (list of Penalty objects) : Pénalités appliquées.

Méthodes :

- changeState(new_installment_state_object) : Change l'état du paiement par tranches.
- processPayment(amount, payment_method) : Traite un paiement selon l'état actuel.
- setPaymentStrategy(payment_strategy) : Définit la stratégie de paiement.
- makePayment(amount) : Effectue un paiement selon la stratégie définie.
- canRequestDelivery() : Vérifie si la livraison peut être demandée.
- requestDelivery(delivery_address) : Demande la livraison de l'article.

- calculatePenalties() : Calcule les pénalités de retard.
- generatePaymentPlan() : Génère un plan de paiement personnalisé.
- validateSubscription() : Template Method pour valider la souscription.
- createTransaction(amount, payment_method) : Factory Method pour créer des transactions.
- updatePaymentProgress() : Met à jour le progrès du paiement.
- sendPaymentReminder() : Envoie un rappel de paiement.
- generatePaymentReceipt() : Génère un reçu de paiement.
- addObserver(observer) : Ajoute un observateur.
- removeObserver(observer) : Supprime un observateur.
- notifyObservers(message) : Notifie les observateurs.

4.2 Interface InstallmentState (State Pattern)

L'interface InstallmentState définit les comportements pour les différents états d'un paiement par tranches.

Méthodes :

- processPayment(installment_payment, amount) : Traite un paiement dans l'état actuel.
- canRequestDelivery() : Indique si la livraison peut être demandée.
- canAddPayment() : Indique si des paiements supplémentaires peuvent être ajoutés.
- getStateDescription() : Retourne une description de l'état.
- calculatePenalties(installment_payment) : Calcule les pénalités selon l'état.
- getAvailableActions() : Retourne les actions disponibles dans cet état.

4.3 Classes Concrètes d'États de Paiement par Tranches

4.3.1 ActiveInstallmentState (implémente InstallmentState)

État d'un paiement par tranches actif et en cours.

Méthodes :

- processPayment(installment_payment, amount) : Traite le paiement et vérifie si le paiement est terminé.
- canRequestDelivery() : Retourne false (livraison uniquement quand totalement payé).
- canAddPayment() : Retourne true.
- getStateDescription() : Retourne "Paiement en cours".

4.3.2 CompletedInstallmentState (implémente InstallmentState)

État d'un paiement par tranches complètement soldé.

Méthodes :

- processPayment(installment_payment, amount) : Lance une exception (paiement terminé).
- canRequestDelivery() : Retourne true.
- canAddPayment() : Retourne false.
- getStateDescription() : Retourne "Paiement terminé".

4.3.3 OverdueInstallmentState (implémente InstallmentState)

État d'un paiement par tranches en retard.

Attributs :

- overdue_days (int) : Nombre de jours de retard.
- penalty_rate (double) : Taux de pénalité appliqué.

Méthodes :

- processPayment(installment_payment, amount) : Traite le paiement avec application de pénalités.
- canRequestDelivery() : Retourne false.
- canAddPayment() : Retourne true.
- calculatePenalties(installment_payment) : Calcule les pénalités de retard.
- getStateDescription() : Retourne "Paiement en retard".

4.3.4 CancelledInstallmentState (implémente InstallmentState)

État d'un paiement par tranches annulé.

Attributs :

- cancellation_date (datetime) : Date d'annulation.
- cancellation_reason (string) : Raison de l'annulation.
- refund_amount (double) : Montant remboursé.

Méthodes :

- processPayment(installment_payment, amount) : Lance une exception (paiement annulé).
- canRequestDelivery() : Retourne false.

- canAddPayment() : Retourne false.
- getStateDescription() : Retourne "Paiement annulé".

4.4 Interface PaymentStrategy (Strategy Pattern)

L'interface PaymentStrategy définit les différentes stratégies de paiement pour les tranches.

Méthodes :

- processPayment(amount, installment_payment) : Traite le paiement selon la stratégie.
- validatePayment(amount) : Valide le montant selon les règles de la stratégie.
- calculateTransactionFees(amount) : Calcule les frais de transaction.
- getPaymentMethodName() : Retourne le nom de la méthode de paiement.
- getPaymentLimits() : Retourne les limites de paiement.

4.5 Classes Concrètes de Stratégies de Paiement

4.5.1 MobileMoneyPaymentStrategy (implémente PaymentStrategy)

Stratégie de paiement via Mobile Money.

Attributs :

- mobile_money_service (object) : Service Mobile Money utilisé.
- transaction_limit (double) : Limite de transaction.
- fee_percentage (double) : Pourcentage de frais.

4.5.2 CardPaymentStrategy (implémente PaymentStrategy)

Stratégie de paiement par carte bancaire.

Attributs :

- card_service (object) : Service de carte utilisé.
- supported_card_types (list) : Types de cartes supportés.
- security_validation_level (int) : Niveau de validation sécuritaire.

4.5.3 BankTransferPaymentStrategy (implémente PaymentStrategy)

Stratégie de paiement par virement bancaire.

Attributs :

- supported_banks (list) : Banques supportées.

- processing_time (int) : Temps de traitement en heures.
- minimum_amount (double) : Montant minimum pour virement.

4.6 Classe InstallmentTransaction

La classe InstallmentTransaction représente une transaction individuelle dans le cadre d'un paiement par tranches.

Attributs :

- id_transaction (long) : Identifiant unique de la transaction.
- id_installment_payment (long) : Clé étrangère vers le paiement par tranches.
- transaction_amount (double) : Montant de la transaction.
- transaction_date (datetime) : Date de la transaction.
- payment_method (string) : Méthode de paiement utilisée.
- transaction_reference (string) : Référence de la transaction.
- transaction_status (enum: PENDING, COMPLETED, FAILED, REVERSED) : Statut de la transaction.
- transaction_fees (double) : Frais de la transaction.
- external_reference (string) : Référence externe du système de paiement.
- transaction_description (text) : Description de la transaction.

Méthodes :

- processTransaction() : Traite la transaction.
- reverseTransaction() : Inverse la transaction si possible.
- generateTransactionReceipt() : Génère un reçu de transaction.
- validateTransaction() : Valide la transaction.

5. Classes d'Association et d'Intégration

5.1 Classes d'Observateurs pour les Notifications

5.1.1 CardTransactionObserver (implémente Observer)

Observateur pour les notifications de transactions de carte.

Attributs :

- id_card (long) : Clé étrangère vers la carte.
- id_user (long) : Clé étrangère vers l'utilisateur.

- notification_service (object) : Service de notification.

Méthodes :

- update(message) : Met à jour et envoie des notifications de transaction de carte.

5.1.2 PlanningReminderObserver (implémente Observer)

Observateur pour les rappels de planification.

Attributs :

- id_planning (long) : Clé étrangère vers la planification.
- id_user (long) : Clé étrangère vers l'utilisateur.
- reminder_settings (object) : Paramètres de rappel.

Méthodes :

- update(message) : Envoie des rappels de planification.

5.1.3 InstallmentPaymentObserver (implémente Observer)

Observateur pour les notifications de paiement par tranches.

Attributs :

- id_installment_payment (long) : Clé étrangère vers le paiement par tranches.
- id_user (long) : Clé étrangère vers l'utilisateur.
- notification_service (object) : Service de notification.

Méthodes :

- update(message) : Notifie les changements d'état des paiements par tranches.

5.2 Classes d'Association Many-to-Many

5.2.1 Card_Planning

Classe d'association entre Card et Planning pour lier les cartes aux planifications.

Attributs :

- id_card (long) : Clé étrangère vers Card.
- id_planning (long) : Clé étrangère vers Planning.
- association_date (datetime) : Date de l'association.

- is_primary_funding_source (boolean) : Indique si c'est la source de financement principale.

5.2.2 InstallmentPayment_Planning

Classe d'association entre InstallmentPayment et Planning.

Attributs :

- id_installment_payment (long) : Clé étrangère vers InstallmentPayment.
- id_planning (long) : Clé étrangère vers Planning.
- planned_payment_amount (double) : Montant de paiement planifié.
- payment_frequency (enum) : Fréquence de paiement.

5.2.3 Card_Project

Classe d'association entre Card et Project pour affecter des dépenses de carte à des projets.

Attributs :

- id_card (long) : Clé étrangère vers Card.
- id_project (long) : Clé étrangère vers Project.
- allocated_budget (double) : Budget alloué du projet pour cette carte.
- expense_category (string) : Catégorie de dépense.

6. Facades et Services d'Intégration

6.1 Classe CardManagementFacade (Facade Pattern)

La classe CardManagementFacade fournit une interface simplifiée pour toutes les opérations liées aux cartes.

Attributs :

- card_service (object) : Service principal de gestion des cartes.
- transaction_service (object) : Service de gestion des transactions.
- notification_service (object) : Service de notification.
- security_service (object) : Service de sécurité et validation.
- audit_service (object) : Service d'audit et de logging.

Méthodes :

- createNewCard(user_id, card_type, initial_deposit) : Méthode Facade pour créer une nouvelle carte.
- rechargeCard(card_id, amount, funding_source) : Méthode Facade pour recharger une carte.
- processCardPayment(card_id, amount, merchant_info) : Méthode Facade pour effectuer un paiement.
- blockCard(card_id, reason) : Méthode Facade pour bloquer une carte.
- getCardTransactionHistory(card_id, date_range) : Méthode Facade pour obtenir l'historique des transactions.
- generateCardStatement(card_id, period) : Méthode Facade pour générer un relevé de carte.
- validateCardOperation(card_id, operation_type, amount) : Méthode Facade pour valider une opération.
- manageCardLimits(card_id, daily_limit, monthly_limit) : Méthode Facade pour gérer les limites.
- renewExpiredCard(card_id) : Méthode Facade pour renouveler une carte expirée.

6.2 Classe PlanningManagementFacade (Facade Pattern)

La classe PlanningManagementFacade simplifie la gestion des planifications complexes.

Attributs :

- planning_service (object) : Service principal de planification.
- scheduler_service (object) : Service de planification des tâches.
- validation_service (object) : Service de validation.
- notification_service (object) : Service de notification.
- balance_service (object) : Service de vérification de solde.

Méthodes :

- createScheduledTransfer(transfer_details, schedule_info) : Méthode Facade pour créer un virement planifié.
- createScheduledCardRecharge(card_id, amount, schedule_info) : Méthode Facade pour planifier une recharge de carte.
- createScheduledServicePayment(service_details, schedule_info) : Méthode Facade pour planifier un paiement de service.
- createRecurringPayment(payment_details, recurrence_pattern) : Méthode Facade pour créer un paiement récurrent.

- managePlanningExecution(planning_id, action) : Méthode Facade pour gérer l'exécution des planifications.
- getUserPlanningsWithBalanceCheck(user_id) : Méthode Facade pour obtenir les planifications avec vérification de solde.
- generatePlanningReport(user_id, period) : Méthode Facade pour générer un rapport de planification.
- validatePlanningFeasibility(planning_details) : Méthode Facade pour valider la faisabilité d'une planification.

6.3 Classe InstallmentPaymentFacade (Facade Pattern)

La classe InstallmentPaymentFacade simplifie les opérations de paiement par tranches.

Attributs :

- installment_service (object) : Service principal de paiement par tranches.
- store_service (object) : Service de gestion des boutiques partenaires.
- delivery_service (object) : Service de livraison.
- payment_processing_service (object) : Service de traitement des paiements.
- contract_service (object) : Service de gestion des contrats.

Méthodes :

- createInstallmentSubscription(article_details, payment_plan) : Méthode Facade pour créer une souscription de paiement par tranches.
- processInstallmentPayment(installment_id, payment_amount, payment_method) : Méthode Facade pour traiter un paiement de tranche.
- requestArticleDelivery(installment_id, delivery_address) : Méthode Facade pour demander la livraison.
- calculateInstallmentPlan(article_price, down_payment, duration) : Méthode Facade pour calculer un plan de paiement.
- manageInstallmentStatus(installment_id, new_status) : Méthode Facade pour gérer le statut d'un paiement par tranches.
- generateInstallmentContract(installment_details) : Méthode Facade pour générer un contrat de paiement par tranches.
- getInstallmentPaymentHistory(user_id, period) : Méthode Facade pour obtenir l'historique des paiements par tranches.

- validateInstallmentEligibility(user_id, article_details) : Méthode Facade pour valider l'éligibilité au paiement par tranches.

6.4 Classe IntegratedFinancialServicesFacade (Facade Pattern)

La classe IntegratedFinancialServicesFacade fournit une interface unifiée pour toutes les opérations financières intégrées.

Attributs :

- card_facade (CardManagementFacade) : Facade de gestion des cartes.
- planning_facade (PlanningManagementFacade) : Facade de gestion des planifications.
- installment_facade (InstallmentPaymentFacade) : Facade de paiement par tranches.
- project_service (object) : Service de gestion des projets.
- user_service (object) : Service de gestion des utilisateurs.

Méthodes :

- scheduleCardRechargeForProject(card_id, project_id, amount, schedule) : Intègre recharge de carte, planification et projet.
- linkInstallmentToProject(installment_id, project_id) : Lie un paiement par tranches à un projet.
- createProjectBudgetWithCards(project_id, card_allocations) : Crée un budget de projet avec allocation des cartes.
- scheduleProjectExpenses(project_id, expense_schedule) : Planifie les dépenses d'un projet.
- generateFinancialDashboard(user_id) : Génère un tableau de bord financier intégré.
- processMultiChannelPayment(payment_details, channels) : Traite un paiement via plusieurs canaux.
- validateCrossModuleTransaction(transaction_details) : Valide une transaction impliquant plusieurs modules.

7. Relations entre Classes

7.1 Relations du Module Cartes

Relations One-to-Many :

- User to Card : Un utilisateur peut posséder plusieurs cartes (id_user dans Card).
- Card to CardOperation : Une carte peut avoir plusieurs opérations (id_card dans CardOperation).

Relations Many-to-Many :

- Card to Project : Une carte peut être associée à plusieurs projets et vice versa (via Card_Project).
- Card to Planning : Une carte peut être utilisée dans plusieurs planifications (via Card_Planning).

Relations de Composition :

- Card compose CardOperation : Les opérations sont créées et gérées par la carte.
- CardOperation compose TransactionDetails : Chaque opération contient des détails de transaction.

7.2 Relations du Module Planification

Relations One-to-Many :

- User to Planning : Un utilisateur peut avoir plusieurs planifications (id_user dans Planning).
- Planning to PlanningCommand : Une planification peut contenir plusieurs commandes (composition).
- Planning to ExecutionHistory : Une planification a un historique d'exécutions.

Relations Many-to-Many :

- Planning to Project : Une planification peut être liée à plusieurs projets (via Planning_Project).
- Planning to Card : Une planification peut utiliser plusieurs cartes comme source de financement (via Card_Planning).

Relations de Dépendance :

- PlanningCommand dépend de TransferService, CardService, ou ServicePaymentService selon le type de commande.

7.3 Relations du Module Paiement par Tranches

Relations One-to-Many :

- User to InstallmentPayment : Un utilisateur peut avoir plusieurs paiements par tranches (id_user dans InstallmentPayment).
- InstallmentPayment to InstallmentTransaction : Un paiement par tranches peut avoir plusieurs transactions (id_installment_payment dans InstallmentTransaction).
- Store to InstallmentPayment : Une boutique peut avoir plusieurs paiements par tranches associés.

Relations Many-to-Many :

- InstallmentPayment to Planning : Un paiement par tranches peut être lié à plusieurs planifications de paiement (via InstallmentPayment_Planning).

- InstallmentPayment to Project : Un paiement par tranches peut être affecté à un projet (via InstallmentPayment_Project).

7.4 Relations Inter-Modules

Relations d'Intégration :

- Card to InstallmentPayment : Une carte peut être utilisée pour les paiements par tranches.
- Planning to Card et InstallmentPayment : Les planifications peuvent concerner les cartes ou les paiements par tranches.
- Project to Card, Planning, et InstallmentPayment : Les projets peuvent intégrer tous les modules financiers.

8. Implémentation des Design Patterns - Récapitulatif

8.1 Factory Method Pattern

Implémenté dans :

- Classe Card : createOperation(), createRechargeOperation(), createPaymentOperation()
- Classe Planning : createCommand()
- Classe InstallmentPayment : createTransaction()

Objectif : Abstrait la création d'objets complexes et permet l'extension future avec de nouveaux types.

8.2 Observer Pattern

Implémenté dans :

- Toutes les classes principales (Card, Planning, InstallmentPayment) comme Subjects
- Classes d'observateurs spécialisées pour chaque type de notification
- Association avec les classes de notification existantes

Objectif : Découple les objets notificateurs des objets notifiés, permettant une architecture de notification flexible.

8.3 State Pattern

Implémenté dans :

- Classe Card avec CardState (ActiveCardState, BlockedCardState, ExpiredCardState, PendingCardState)

- Classe InstallmentPayment avec InstallmentState (ActiveInstallmentState, CompletedInstallmentState, OverdueInstallmentState, CancelledInstallmentState)

Objectif : Permet à un objet de changer son comportement selon son état interne, évitant les conditions complexes.

8.4 Strategy Pattern

Implémenté dans :

- Classe Card avec CardOperationStrategy (RechargeOperationStrategy, PaymentOperationStrategy, WithdrawalOperationStrategy)
- Classe InstallmentPayment avec PaymentStrategy (MobileMoneyPaymentStrategy, CardPaymentStrategy, BankTransferPaymentStrategy)

Objectif : Permet de changer l'algorithme utilisé selon le contexte, sans modifier la classe cliente.

8.5 Command Pattern

Implémenté dans :

- Classe Planning avec PlanningCommand (ScheduledTransferCommand, ScheduledCardRechargeCommand, ScheduledServicePaymentCommand)
- Classe CardOperation pour encapsuler les opérations de carte

Objectif : Encapsule les requêtes comme objets, permettant la planification, la mise en file d'attente et l'annulation.

8.6 Template Method Pattern

Implémenté dans :

- Classe CardOperation : execute() définit le squelette d'exécution
- Classe Planning : executeScheduledTasks() définit le processus d'exécution
- Classe InstallmentPayment : validateSubscription() définit le processus de validation

Objectif : Définit le squelette d'un algorithme, laissant les sous-classes implémenter les détails spécifiques.

8.7 Facade Pattern

Implémenté dans :

- CardManagementFacade : Interface simplifiée pour les opérations de carte

- PlanningManagementFacade : Interface simplifiée pour la planification
- InstallmentPaymentFacade : Interface simplifiée pour les paiements par tranches
- IntegratedFinancialServicesFacade : Interface unifiée pour tous les services financiers

Objectif : Fournit une interface simplifiée vers des sous-systèmes complexes.

8.8 Singleton Pattern

Implémenté dans :

- Classe PlanningScheduler : Assure une seule instance du planificateur système

Objectif : Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global.

9. Contraintes Techniques et Sécurité

9.1 Contraintes de Sécurité Spécifiques

Pour les Cartes :

- Masquage obligatoire des numéros de carte (seuls les 4 derniers chiffres visibles)
- Hash sécurisé des codes PIN et de sécurité
- Validation biométrique pour les opérations sensibles
- Logging détaillé de toutes les opérations avec IP et géolocalisation
- Blocage automatique après tentatives frauduleuses

Pour la Planification :

- Validation du solde avant chaque exécution planifiée
- Notifications obligatoires avant exécution d'opérations importantes
- Limitation du nombre de planifications actives par utilisateur
- Audit trail complet de toutes les modifications de planification

Pour les Paiements par Tranches :

- Vérification de l'identité complète avant souscription
- Limitation à 3 mois maximum pour les paiements
- Validation GPS de la boutique partenaire
- Signature électronique des conditions générales

9.2 Contraintes de Performance

Optimisations :

- Cache des informations de carte fréquemment consultées
- Index sur les dates de planification pour les requêtes rapides
- Pagination des historiques de transaction
- Traitement asynchrone des notifications

Limites :

- Maximum 50 planifications actives par utilisateur
- Historique des transactions limité à 2 ans
- Taille maximale des photos d'articles : 5MB
- Timeout des opérations : 30 secondes

9.3 Contraintes d'Intégration

Avec le Système Existant :

- Intégration transparente avec le module User existant
- Compatibilité avec le système de notification existant
- Utilisation des classes Project, Phase, Step, Task existantes
- Respect de l'architecture de base de données existante

APIs Externes :

- Intégration avec les services Mobile Money (MTN, Orange, etc.)
- Connexion aux réseaux de cartes bancaires
- APIs des boutiques partenaires pour la validation des articles
- Services de géolocalisation pour la validation des boutiques

10. Évolutivité et Maintenance

10.1 Points d'Extension

Nouveaux Types de Cartes :

- Framework extensible via le pattern Strategy pour supporter de nouveaux types de cartes
- Interface CardType pour définir de nouveaux comportements spécifiques

Nouvelles Méthodes de Paiement :

- Pattern Strategy permet l'ajout facile de nouvelles méthodes de paiement
- Interface PaymentMethod pour l'extensibilité

Nouveaux Types de Planification :

- Pattern Command permet l'ajout de nouveaux types de commandes planifiées
- Interface PlanningCommandType pour les extensions

10.2 Maintenance et Monitoring

Logging :

- Logs détaillés pour toutes les opérations financières
- Niveaux de log configurables (.info, .debug, .error)
- Rotation automatique des fichiers de log

Monitoring :

- Métriques de performance pour chaque type d'opération
- Alertes automatiques pour les opérations suspectes
- Tableau de bord de monitoring temps réel

Sauvegarde :

- Sauvegarde quotidienne de toutes les données financières
- Réplication en temps réel pour la haute disponibilité
- Plan de reprise après sinistre testé régulièrement

11. Conclusion

Cette conception détaillée des modules Cartes, Planification et Paiement par Tranches s'intègre harmonieusement dans l'écosystème DOHONE existant. L'utilisation cohérente des design patterns assure une architecture flexible et maintenable, tandis que les contraintes de sécurité et de performance garantissent une expérience utilisateur optimale et sécurisée.

L'architecture proposée respecte les principes SOLID, facilite les tests unitaires et d'intégration, et permet une évolution future sans impact sur les composants existants. La séparation claire des responsabilités entre les différentes couches facilite également la maintenance et le débogage du système.