Architecture et Styles Architecturaux

Projet A.S.A.

Ait Yacoub Anis

Atala Salim

Master 2 ALMA - Université de Nantes

Table des matières

Résumé

Dans le cadre du cours "Architectures et Styles Architecturaux" du Master ALMA 2024 de l'Université de Nantes, ce projet vise à concevoir et à implémenter une architecture logicielle basée sur le style composant-connecteur. Nous présentons un méta-modèle (M2) de cette architecture, un modèle client-serveur (M1) dérivé du méta-modèle, et une instance concrète (M0) de ce modèle implémentée en Java. Les extensions apportées incluent la gestion des attachements, l'intégration du pattern Observer, et la mise en place d'un système de règles pour assurer la cohérence et la sécurité des interactions entre les composants.

1 Introduction

L'architecture composant-connecteur est un style architectural qui met l'accent sur la décomposition d'un système en composants modulaires interconnectés par des connecteurs définis. Ce style permet une grande flexibilité et facilite la maintenance et l'évolution des systèmes complexes. Ce rapport détaille la conception et l'implémentation d'un méta-modèle de cette architecture, son application à un modèle client-serveur, et son instanciation concrète.

2 Pré-requis

La réalisation de ce projet nécessite une maîtrise des compétences et connaissances suivantes :

- Programmation Java
- UML (Unified Modeling Language)
- Paradigme Objet
- Métamodèle (concepts tels que Component, Connector, Configuration, Binding, etc.)
- Transition entre les niveaux M2 (métamodèle) et M0 (instance concrète)

3 Méthodologie

3.1 Présentation d'A.S.A.

A.S.A. est une méthodologie de conception architecturale qui permet de structurer et d'organiser les composants d'un système logiciel de manière modulaire et évolutive. Elle s'appuie sur les principes des styles architecturaux composant-connecteur pour faciliter l'interopérabilité et la réutilisabilité des composants.

3.2 Choix de Conception

Les choix de conception ont été guidés par les objectifs suivants :

- Modularité : Faciliter la maintenance et l'évolution du système.
- Réutilisabilité : Permettre la réutilisation des composants dans différents contextes.
- Sécurité : Assurer la sécurité des interactions entre les composants.
- Flexibilité : Faciliter l'adaptation du système aux besoins changeants.

3.3 Écarts entre Spécification et Implémentation

Lors de la transition de la spécification à l'implémentation, certains ajustements ont été nécessaires pour optimiser les performances et la maintenabilité du système. Par exemple, des modifications ont été apportées aux règles de connexion pour mieux gérer les scénarios d'authentification et de transaction sécurisée.

4 Spécification Technique

4.1 Méta-modèle (M2)

4.1.1 Description du Méta-modèle

Le méta-modèle définit les concepts fondamentaux de l'architecture composant-connecteur :

- **Composant** : Unité modulaire encapsulant des fonctionnalités spécifiques.
- **Interface de Composant** : Classe abstraite servant de base pour les interfaces des composants.
  - **ProvideInterface** : Interface fournissant des services (sortants).
  - **RequireInterface** : Interface requérant des services (entrants).
- **Port** : Point de connexion des composants, associé à des services.
- **Service** : Fonctionnalité offerte ou requise par un composant.
- **Connecteur** : Mécanisme permettant l'interaction entre les composants.
- **RulesInterface** : Interface définissant des règles pour les connecteurs.
  - **ProvideRules** : Règles fournies par un connecteur.
  - **RequireRules** : Règles requises par un connecteur.
- **Attachment** : Liaison entre un port et une interface de règles.
- **Glue** : Ensemble de règles liant les services des composants via les connecteurs.
- **Configuration** : Classe centrale reliant les composants et les connecteurs, assurant la cohérence de l'ensemble.

4.1.2 Diagramme UML du Méta-modèle

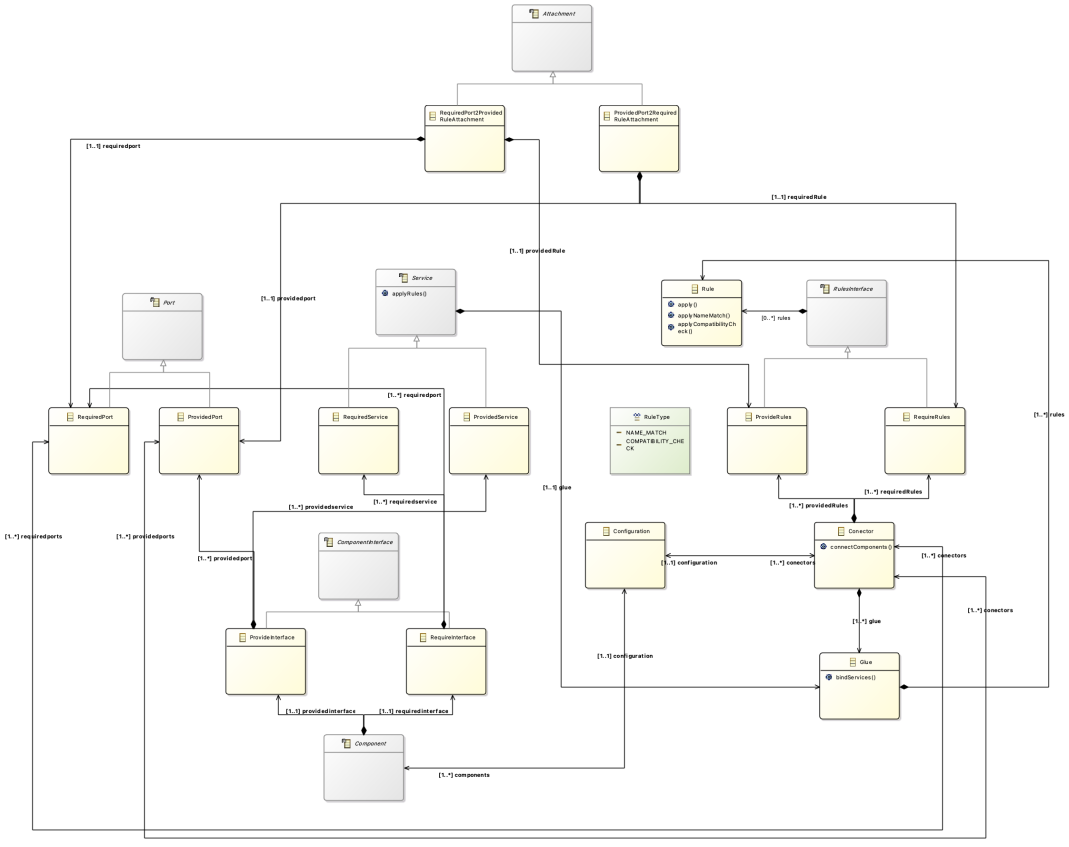

FIGURE 1 – Diagramme UML du Méta-modèle (M2)

4.1.3 Implémentation M2

ComponentInterface.java

```java
package component;

import java.util.List;
import core.Port;
import core.Service;

/**
 * Abstract class representing a generic interface for components.
```

```
 */
public abstract class ComponentInterface {
    private List<Port> ports;
    private List<Service> services;

    public ComponentInterface(List<Port> ports, List<Service> services) {
        this.ports = ports;
        this.services = services;
    }

    public List<Port> getPorts() {
        return ports;
    }

    public List<Service> getServices() {
        return services;
    }
}
```

Listing 1 – Classe `ComponentInterface`

ProvideInterface.java et RequireInterface.java

```
package component;

import core.Port;
import core.Service;
import java.util.List;

/**
 * Represents the provided interface of a component.
 */
public class ProvideInterface extends ComponentInterface {
    public ProvideInterface(List<Port> ports, List<Service> services) {
        super(ports, services);
    }
}
```

Listing 2 – Classe `ProvideInterface`

```
package component;

import core.Port;
import core.Service;
import java.util.List;

/**
 * Represents the required interface of a component.
 */
public class RequireInterface extends ComponentInterface {
    public RequireInterface(List<Port> ports, List<Service> services) {
        super(ports, services);
    }
}
```

Listing 3 – Classe `RequireInterface`

4.2 Connecteurs et Règles

4.2.1 Connector.java

```
package connector;

import component.Component;
import core.Service;
import java.util.List;
```

```java

/**
 * Represents a connector used to connect two components.
 * It holds references to provided and required rules and also has a glue
 * that connects the services with rules.
 */
public class Connector {
    private Glue glue;
    private ProvideRules provideRules;
    private RequireRules requireRules;

    public Connector(Glue glue, ProvideRules provideRules, RequireRules
        requireRules) {
        this.glue = glue;
        this.provideRules = provideRules;
        this.requireRules = requireRules;
    }

    public Glue getGlue() {
        return glue;
    }

    public void setGlue(Glue glue) {
        this.glue = glue;
    }

    public ProvideRules getProvideRules() {
        return provideRules;
    }

    public void setProvideRules(ProvideRules provideRules) {
        this.provideRules = provideRules;
    }

    public RequireRules getRequireRules() {
        return requireRules;
    }

    public void setRequireRules(RequireRules requireRules) {
        this.requireRules = requireRules;
    }

    /**
     * Connects two components by associating the provided and required rules,
     * and applies glue to bind services with rules.
     */
    public void connectComponents(Component component1, Component component2) {
        // Collect all provided services from the first component
        List<Service> providedServices =
            component1.getProvideInterface().getPorts()
                .stream()
                .flatMap(port -> port.getServices().stream())
                .toList();

        // Collect all required services from the second component
        List<Service> requiredServices =
            component2.getRequireInterface().getPorts()
                .stream()
                .flatMap(port -> port.getServices().stream())
                .toList();

        // Match and bind services
        for (Service providedService : providedServices) {
            for (Service requiredService : requiredServices) {
                if (providedService.applyRules(requiredService)) {
                    System.out.println("Connected service: "
```

```
                                + providedService.getServiceName()
                                + " to "
                                + requiredService.getServiceName());
                    }
                }
            }
        }
}
```

Listing 4 – Classe `Connector`

4.2.2 Glue.java

```
package connector;

import core.Rule;
import core.Service;
import java.util.List;

/**
 * Repr sente une glue qui connecte des r gles et des services.
 * Elle contient une liste de r gles qui lient les services aux composants.
 */
public class Glue {
    private List<Rule> rules;

    public Glue(List<Rule> rules) {
        this.rules = rules;
    }

    public List<Rule> getRules() {
        return rules;
    }

    public void setRules(List<Rule> rules) {
        this.rules = rules;
    }

    /**
     * Lie un service fourni    un service requis en utilisant les r gles.
     *
     * @param providedService Le service fourni.
     * @param requiredService Le service requis.
     * @return True si les services ont   t   li s avec succ s, false sinon.
     */
    public boolean bindServices(Service providedService, Service
        requiredService) {
        // Parcourt chaque r gle pour v rifier si les services peuvent  tre
            connect s
        for (Rule rule : rules) {
            if (rule.apply(providedService, requiredService)) {
                // La r gle a   t   appliqu e avec succ s, les services sont
                    connect s
                System.out.println("Services connect s : " +
                        providedService.getServiceName() + " -> " +
                            requiredService.getServiceName());
                return true;
            }
        }
        // Si aucune r gle ne s'applique, les services ne sont pas connect s
        System.out.println("Les services n'ont pas pu  tre  connect s : " +
                providedService.getServiceName() + " -> " +
                    requiredService.getServiceName());
        return false;
    }
```

```
}
```

Listing 5 – Classe `Glue`

4.2.3 ProvideRules.java et RequireRules.java

```
package connector;

import core.Rule;
import rules.RulesInterface;
import java.util.List;

/**
 * Repr sente les r gles fournies.
 */
public class ProvideRules extends RulesInterface {
    public ProvideRules(List<Rule> rules) {
        super(rules);
    }
}
```

Listing 6 – Classe `ProvideRules`

```
package connector;

import core.Rule;
import rules.RulesInterface;
import java.util.List;

/**
 * Repr sente les r gles requises.
 */
public class RequireRules extends RulesInterface {
    public RequireRules(List<Rule> rules) {
        super(rules);
    }
}
```

Listing 7 – Classe `RequireRules`

4.3 Gestion des Règles

4.3.1 Rule.java et RuleType.java

```
package core;

import rules.RuleType;

/**
 * Repr sente une r gle qui peut  tre  appliqu e aux services lors du
    processus de liaison.
 */
public class Rule {
    private RuleType ruleType; // Le type de r gle (par exemple, NAME_MATCH,
        COMPATIBILITY_CHECK)
    private String ruleName;   // Un nom pour la r gle (peut  tre  une
        description)

    public Rule(RuleType ruleType, String ruleName) {
        this.ruleType = ruleType;
        this.ruleName = ruleName;
    }

    public RuleType getRuleType() {
```

```
        return ruleType;
    }

    public void setRuleType(RuleType ruleType) {
        this.ruleType = ruleType;
    }

    public String getRuleName() {
        return ruleName;
    }

    public void setRuleName(String ruleName) {
        this.ruleName = ruleName;
    }

    /**
     * Ex cute la r gle sur le service fourni et le service requis.
     *
     * @param providedService Le service fourni.
     * @param requiredService Le service requis.
     * @return True si la r gle peut  tre  appliqu e, false sinon.
     */
    public boolean apply(Service providedService, Service requiredService) {
        switch (ruleType) {
            case NAME_MATCH:
                return applyNameMatch(providedService, requiredService);
            case COMPATIBILITY_CHECK:
                return applyCompatibilityCheck(providedService,
                    requiredService);
            default:
                return false;
        }
    }

    // Logique pour la r gle NAME_MATCH
    private boolean applyNameMatch(Service providedService, Service
        requiredService) {
        return
            providedService.getServiceName().equalsIgnoreCase(requiredService.getServiceName()
    }

    // Logique pour la r gle COMPATIBILITY_CHECK
    private boolean applyCompatibilityCheck(Service providedService, Service
        requiredService) {
        // Exemple de logique de v rification de compatibilit ,
        // TODO : elle peut  tre  personnalis e.
        return providedService.getGlue().equals(requiredService.getGlue());
    }
}
```

Listing 8 – Classe `Rule`

```
package rules;

/**
 * Enum repr sentant diff rents types de r gles qui peuvent  tre
    appliqu es aux services.
 */
public enum RuleType {
    NAME_MATCH,         // R gle pour la correspondance des noms de services
    COMPATIBILITY_CHECK // R gle pour la v rification de la compatibilit
        des services
}
```

Listing 9 – Enum `RuleType`

4.3.2 RulesInterface.java

```java
package rules;

import core.Rule;
import java.util.List;

/**
 * Classe abstraite repr sentant une interface pour les r gles.
 */
public abstract class RulesInterface {
    private List<Rule> rules;

    public RulesInterface(List<Rule> rules) {
        this.rules = rules;
    }

    public List<Rule> getRules() {
        return rules;
    }

    public void setRules(List<Rule> rules) {
        this.rules = rules;
    }
}
```

Listing 10 – Classe abstraite `RulesInterface`

5 Modèle Client-Serveur (M1)

5.0.1 Description du Modèle

Le modèle client-serveur dérive du méta-modèle M2. Les composants principaux sont :

- **Client** : Composant initiant les requêtes vers le serveur et agissant comme observateur pour recevoir les notifications.
- **Serveur** : Composant traitant les requêtes, gérant l'authentification, les transactions bancaires et les notifications des observateurs en cas de changement d'état.
- **Gestionnaire de Connexion** : Responsable de l'authentification des utilisateurs et de la gestion des tokens de session.
- **Gestionnaire de Sécurité** : Gère la sécurité des sessions utilisateurs, notamment via la génération et la validation des tokens.
- **Base de Données** : Représente la base de données du système, gérant les utilisateurs et les comptes bancaires.
- **Connecteur RPC** : Permet une communication de type Remote Procedure Call entre le client et le serveur en appliquant des règles spécifiques.

5.0.2 Diagramme UML du Modèle Client-Serveur M1

5.1 Implémentation M1

5.1.1 Classes Principales

Client.java La classe `Client` représente le composant client dans le système client-serveur. Elle implémente l'interface `Observer` pour recevoir les notifications du serveur.

```java
package com.alma.server_client;

import com.alma.banking.BankingService;
import com.alma.component.Component;
import com.alma.component.ProvideInterface;
import com.alma.component.RequireInterface;
import com.alma.observer.Observer;
import com.alma.server_client.Server;
```

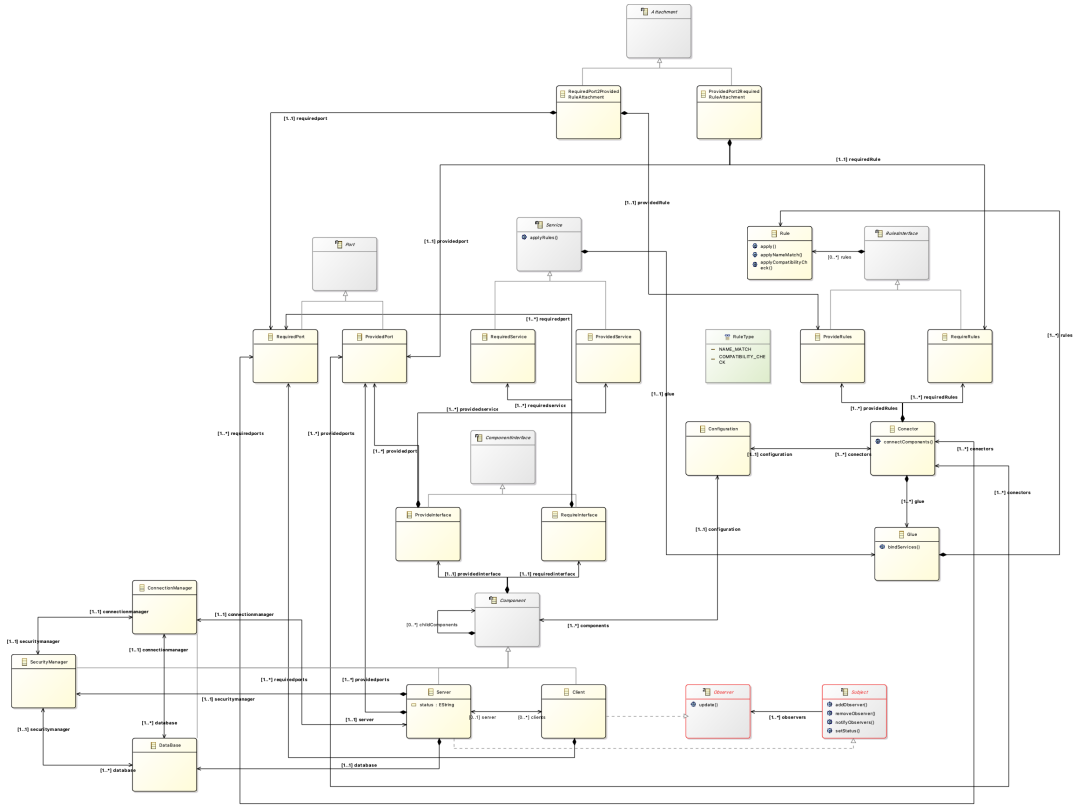

FIGURE 2 – Diagramme UML du Modèle Client-Serveur (M1)

Ce diagramme illustre les interactions entre les composants du modèle client-serveur, tels que le Client, le Serveur, le Gestionnaire de Connexion, le Gestionnaire de Sécurité, la Base de Données et le Connecteur RPC.

```
9
10  /**
11   * Repr sente le client dans le syst me client-serveur.
12   * Impl mente l'interface Observer pour r pondre aux changements d' tat  du
         serveur.
13   */
14  public class Client extends Component implements Observer {
15
16      private BankingService bankingService; // R f rence au service bancaire
17      private String username;
18      private String password;
19      private String sessionToken; // Token de session apr s authentification
20
21      public Client(ProvideInterface providedInterface, RequireInterface
            requiredInterface) {
22          super(providedInterface, requiredInterface);
23      }
24
25      /**
26       * D finit les informations d'identification de l'utilisateur.
27       *
28       * @param username le nom d'utilisateur
29       * @param password le mot de passe
30       */
31      public void setCredentials(String username, String password) {
32          this.username = username;
33          this.password = password;
34      }
35
36      /**
```

```
     * Authentifie le client aupr s du serveur.
     *
     * @param server le serveur pour l'authentification
     */
    public void authenticate(Server server) {
        if (username == null || password == null) {
            System.out.println("Client: Identifiants non d finis.");
            return;
        }
        this.sessionToken = server.login(username, password);
        if (this.sessionToken != null) {
            System.out.println("Client: Authentification r ussie. Token de
                session : " + sessionToken);
        } else {
            System.out.println("Client:  chec  de l'authentification.");
        }
    }

    /**
     * D finit le BankingService pour le client.
     *
     * @param bankingService le service bancaire    utiliser
     */
    public void setBankingService(BankingService bankingService) {
        this.bankingService = bankingService;
    }

    /**
     * Met    jour le client en r ponse    un changement d' tat  du serveur.
     */
    @Override
    public void update(String message) {
        System.out.println("Client: Notification du serveur - " + message);
        // Ajouter une logique sp cifique de mise    jour ici si n cessaire.
    }

    /**
     * Cr e un nouveau compte bancaire en utilisant le BankingService.
     *
     * @param accountNumber le num ro de compte
     * @param initialBalance le solde initial
     */
    public void createAccount(String accountNumber, double initialBalance) {
        if (bankingService != null && sessionToken != null) {
            bankingService.createAccount(accountNumber, initialBalance);
        } else {
            System.out.println("Client: Service bancaire indisponible ou
                utilisateur non authentifi .");
        }
    }

    /**
     * Effectue une transaction en utilisant le BankingService.
     *
     * @param transactionId l'ID de la transaction
     * @param fromAccount   le compte source
     * @param toAccount     le compte destination
     * @param amount        le montant    transf rer
     */
    public void performTransaction(String transactionId, String fromAccount,
        String toAccount, double amount) {
        if (bankingService != null && sessionToken != null) {
            bankingService.performTransaction(transactionId, fromAccount,
                toAccount, amount);
        } else {
            System.out.println("Client: Service bancaire indisponible ou
```

```
                utilisateur non authentifi .");
        }
    }

    // Getters pour les champs priv s

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public String getSessionToken() {
        return sessionToken;
    }

    public BankingService getBankingService() {
        return bankingService;
    }

}
```

Listing 11 – Classe `Client`

ConnectionManager.java La classe `ConnectionManager` gère les connexions et l'authentification des utilisateurs, interagissant avec la base de données et le gestionnaire de sécurité pour valider les identifiants et gérer les tokens de session.

```
package server_client;

import component.Component;
import component.ProvideInterface;
import component.RequireInterface;

/**
 * G re les connexions et l'authentification des utilisateurs.
 */
public class ConnectionManager extends Component {

    private Database db;
    private SecurityManager sm;

    public ConnectionManager(Database db, SecurityManager sm, ProvideInterface
        providedInterface, RequireInterface requiredInterface) {
        super(providedInterface, requiredInterface);
        this.db = db;
        this.sm = sm;
    }

    /**
     * Authentifie un utilisateur et g  n  re un token de session.
     *
     * @param username le nom d'utilisateur
     * @param password le mot de passe
     * @return le token de session si l'authentification r ussit , null sinon
     */
    public String login(String username, String password) {
        // Logique d'authentification
        String storedPassword = db.getPassword(username);
        if (storedPassword != null && storedPassword.equals(password)) {
            String sessionToken = sm.generateSessionToken(username);
            return sessionToken;
        }
        return null;
```

```
    }
}
```

Listing 12 – Classe `ConnectionManager`

Database.java La classe `Database` représente la base de données du système, gérant les informations des utilisateurs et des comptes bancaires.

```
package server_client;

import banking.Account;
import component.Component;
import component.ProvideInterface;
import component.RequireInterface;

import java.util.HashMap;
import java.util.Map;

/**
 * Classe repr sentant la base de donn es.
 */
public class Database extends Component {

    private Map<String, String> userCredentials;
    private Map<String, Account> accounts;

    public Database(ProvideInterface providedInterface, RequireInterface
        requiredInterface) {
        super(providedInterface, requiredInterface);
        userCredentials = new HashMap<>();
        accounts = new HashMap<>();

        // Ajouter des utilisateurs d'exemple
        userCredentials.put("user1", "password1");
        userCredentials.put("user2", "password2");
    }

    /**
     * R cup re le mot de passe pour un nom d'utilisateur donn .
     *
     * @param username le nom d'utilisateur dont le mot de passe est
         r cup rer
     * @return le mot de passe si le nom d'utilisateur existe, null sinon
     */
    public String getPassword(String username) {
        return userCredentials.get(username);  // Retourne null si
            l'utilisateur n'existe pas
    }

    /**
     * Ajoute un nouvel utilisateur avec un nom d'utilisateur et un mot de
         passe    la base de donn es.
     *
     * @param username le nom d'utilisateur du nouvel utilisateur
     * @param password le mot de passe pour le nouvel utilisateur
     * @return true si l'utilisateur a   t   ajout   avec succ s, false si le
         nom d'utilisateur existe  d  j
     */
    public boolean addUser(String username, String password) {
        if (userCredentials.containsKey(username)) {
            return false;  // L'utilisateur existe  d  j
        }
        userCredentials.put(username, password);
        return true;
    }

```

```
    /**
     * Supprime un utilisateur de la base de donn es.
     *
     * @param username le nom d'utilisateur de l'utilisateur    supprimer
     * @return true si l'utilisateur a   t   supprim  avec succ s , false si
         l'utilisateur n'existe pas
     */
    public boolean removeUser(String username) {
        if (userCredentials.containsKey(username)) {
            userCredentials.remove(username);
            return true;  // Utilisateur supprim
        }
        return false;  // L'utilisateur n'existe pas
    }

    /**
     * Cr e un compte dans la base de donn es.
     *
     * @param accountNumber  le num ro de compte
     * @param initialBalance le solde initial
     */
    public void createAccount(String accountNumber, double initialBalance) {
        if (!accounts.containsKey(accountNumber)) {
            Account account = new Account(accountNumber, initialBalance);
            accounts.put(accountNumber, account);
            System.out.println("Database: Account " + accountNumber + " created
                with balance " + initialBalance);
        } else {
            System.out.println("Database: Account " + accountNumber + " already
                exists.");
        }
    }

    /**
     * Supprime un compte de la base de donn es.
     *
     * @param accountNumber le num ro de compte du compte    supprimer
     * @return true si le compte a   t   supprim  avec succ s , false si le
         compte n'existe pas
     */
    public boolean removeAccount(String accountNumber) {
        if (accounts.containsKey(accountNumber)) {
            accounts.remove(accountNumber);
            return true;  // Compte supprim
        }
        return false;  // Le compte n'existe pas
    }

    /**
     * R cup re un compte par son num ro de compte.
     *
     * @param accountNumber le num ro de compte du compte    r cup rer
     * @return le compte s'il existe, null sinon
     */
    public Account getAccount(String accountNumber) {
        return accounts.get(accountNumber);  // Retourne null si le compte
            n'existe pas
    }

    /**
     * Ex cute une requ te dans la base de donn es.
     *
     * @param query la requ te    ex cuter
     */
    public void executeQuery(String query) {
        // Impl mentation de l'ex cution d'une requ te
```

```
    }
}
```

Listing 13 – Classe `Database`

SecurityManager.java La classe `SecurityManager` gère la sécurité des sessions utilisateurs en générant, validant et invalidant les tokens de session.

```
package server_client;

import component.Component;
import component.ProvideInterface;
import component.RequireInterface;

import java.util.HashMap;
import java.util.Map;

/**
 * G re la s curit  des sessions utilisateurs.
 */
public class SecurityManager extends Component {

    private Database db;
    private Map<String, String> sessionTokens = new HashMap<>();

    public SecurityManager(Database db, ProvideInterface providedInterface,
        RequireInterface requiredInterface) {
        super(providedInterface, requiredInterface);
        this.db = db;
    }

    /**
     * G n re un token de session pour un utilisateur.
     *
     * @param username le nom d'utilisateur
     * @return le token de session g n r
     */
    public String generateSessionToken(String username) {
        String token = username + "-TOKEN-12345"; // Pour un vrai syst me,
            utilisez un g n rateur de tokens s curis
        sessionTokens.put(username, token);
        return token;
    }

    /**
     * V rifie si le token de session est valide pour un utilisateur donn .
     *
     * @param username     le nom d'utilisateur
     * @param sessionToken le token de session
     * @return true si la session est valide, false sinon
     */
    public boolean isSessionValid(String username, String sessionToken) {
        return sessionTokens.containsKey(username) &&
            sessionTokens.get(username).equals(sessionToken);
    }
}
```

Listing 14 – Classe `SecurityManager`

Server.java La classe `Server` représente le composant serveur dans le système client-serveur. Elle implémente les interfaces `Subject` et `BankingService` pour gérer les observateurs et fournir des services bancaires.

```
package server_client;

```

```java
import banking.Account;
import banking.BankingService;
import banking.Transaction;
import component.Component;
import component.ProvideInterface;
import component.RequireInterface;
import observer.Observer;
import observer.Subject;

import java.util.ArrayList;
import java.util.List;

/**
 * Repr sente le serveur dans le syst me client-serveur.
 * Impl mente Subject pour notifier les observateurs des changements d' tat .
 */
public class Server extends Component implements Subject, BankingService {

    private final List<Observer> observers; // Liste des observateurs (clients)
    private boolean stateChanged;

    private ConnectionManager cm;
    private SecurityManager sm;
    private Database db;

    public Server(ProvideInterface providedInterface, RequireInterface
        requiredInterface) {
        super(providedInterface, requiredInterface);
        this.observers = new ArrayList<>();
        this.db = new Database(providedInterface, requiredInterface);
        this.sm = new SecurityManager(this.db, providedInterface,
            requiredInterface);
        this.cm = new ConnectionManager(this.db, this.sm, providedInterface,
            requiredInterface);
    }

    // Impl mentation des m thodes de Subject

    @Override
    public void addObserver(Observer observer) {
        if (!observers.contains(observer)) {
            observers.add(observer);
        }
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(String message) {
        if (stateChanged) {
            for (Observer observer : observers) {
                observer.update(message);
            }
            stateChanged = false;
        }
    }

    // M thode pour changer l' tat  et notifier les observateurs

    public void changeState() {
        System.out.println("Server: State changed, notifying observers...");
        stateChanged = true;
        notifyObservers("Server state has changed.");
```

```
66      }
67
68      // M thodes sp cifiques au serveur
69
70      /**
71       * Authentifie un utilisateur et g n re un token de session.
72       *
73       * @param username le nom d'utilisateur
74       * @param password le mot de passe
75       * @return le token de session si l'authentification r ussit , null sinon
76       */
77      public String login(String username, String password) {
78          stateChanged = true;
79          notifyObservers("Attempting to authenticate user: " + username);
80          String sessionToken = cm.login(username, password);
81          if (sessionToken != null) {
82              System.out.println("Server: User '" + username + "' authenticated
                    successfully. Session token: " + sessionToken);
83              notifyObservers("User '" + username + "' authenticated
                    successfully.");
84          } else {
85              System.out.println("Server: Authentication failed for user '" +
                    username + "'.");
86              notifyObservers("Authentication failed for user '" + username +
                    "'.");
87          }
88          return sessionToken;
89      }
90
91      /**
92       * V rifie si la session est valide pour un utilisateur donn .
93       *
94       * @param username     le nom d'utilisateur
95       * @param sessionToken le token de session
96       * @return true si la session est valide, false sinon
97       */
98      public boolean isSessionValid(String username, String sessionToken) {
99          boolean valid = sm.isSessionValid(username, sessionToken);
100         notifyObservers("Session validation for user '" + username + "': " +
                (valid ? "Valid" : "Invalid"));
101         return valid;
102     }
103
104     // Impl mentation des m thodes de BankingService
105
106     @Override
107     public void createAccount(String accountNumber, double initialBalance) {
108         db.createAccount(accountNumber, initialBalance);
109         System.out.println("Server: Account " + accountNumber + " created with
                initial balance " + initialBalance);
110         notifyObservers("Account " + accountNumber + " created successfully.");
111     }
112
113     @Override
114     public void performTransaction(String transactionId, String fromAccount,
            String toAccount, double amount) {
115         Account sourceAccount = db.getAccount(fromAccount);
116         Account destinationAccount = db.getAccount(toAccount);
117         if (sourceAccount != null && destinationAccount != null) {
118             sourceAccount.withdraw(amount);
119             destinationAccount.deposit(amount);
120             Transaction transaction = new Transaction(transactionId,
                    fromAccount, toAccount, amount);
121             System.out.println("Server: Transaction " + transactionId + "
                    completed: " + transaction);
122             notifyObservers("Transaction " + transactionId + " completed
```

```
                successfully.");
        } else {
            System.out.println("Server: Transaction failed due to invalid
                account information.");
            notifyObservers("Transaction " + transactionId + " failed.");
        }
    }

}
```

Listing 15 – Classe `Server`

6 Instance Client-Serveur (M0)

6.1 Structure de l'Application

L'instance concrète (M0) implémente le modèle client-serveur en Java. Elle simule l'interaction entre un client et un serveur, incluant l'authentification, la création de comptes et l'exécution de transactions.

6.1.1 App.java

```
import server_client.Client;
import server_client.Server;
import component.ProvideInterface;
import component.ProvidedPort;
import component.RequiredPort;
import component.RequireInterface;
import core.Service;
import core.Rule;
import rules.RuleType;
import connector.Connector;
import connector.Glue;
import connector.ProvideRules;
import connector.RequireRules;
import attachment.ProvidedPortRequiredRulesAttachment;
import attachment.RequiredPortProvidedRulesAttachment;

import java.util.ArrayList;
import java.util.List;

/**
 * Classe principale de l'application pour tester le syst me bancaire.
 */
public class App {
    public static void main(String[] args) {
        System.out.println("Application lanc e avec succ s !");

        // Cr er les Services
        Service clientService = new Service("ClientService");
        Service serverService = new Service("ServerService");

        // Cr er les Ports
        ProvidedPort clientProvidedPort = new
            ProvidedPort("ClientProvidedPort");
        RequiredPort clientRequiredPort = new
            RequiredPort("ClientRequiredPort");
        ProvidedPort serverProvidedPort = new
            ProvidedPort("ServerProvidedPort");
        RequiredPort serverRequiredPort = new
            RequiredPort("ServerRequiredPort");

        // Cr er ProvideInterfaces et RequireInterfaces pour le Client
        List<core.Port> clientProvidedPorts = new ArrayList<>();
        clientProvidedPorts.add(clientProvidedPort);
```

```
        List<Service> clientProvidedServices = new ArrayList<>();
        clientProvidedServices.add(clientService);
        ProvideInterface clientProvideInterface = new
            ProvideInterface(clientProvidedPorts, clientProvidedServices);

        List<core.Port> clientRequiredPorts = new ArrayList<>();
        clientRequiredPorts.add(clientRequiredPort);
        List<Service> clientRequiredServices = new ArrayList<>();
        RequireInterface clientRequireInterface = new
            RequireInterface(clientRequiredPorts, clientRequiredServices);

        // Cr er ProvideInterfaces et RequireInterfaces pour le Serveur
        List<core.Port> serverProvidedPorts = new ArrayList<>();
        serverProvidedPorts.add(serverProvidedPort);
        List<Service> serverProvidedServices = new ArrayList<>();
        serverProvidedServices.add(serverService);
        ProvideInterface serverProvideInterface = new
            ProvideInterface(serverProvidedPorts, serverProvidedServices);

        List<core.Port> serverRequiredPorts = new ArrayList<>();
        serverRequiredPorts.add(serverRequiredPort);
        List<Service> serverRequiredServices = new ArrayList<>();
        RequireInterface serverRequireInterface = new
            RequireInterface(serverRequiredPorts, serverRequiredServices);

        // Cr er les composants Client et Serveur
        Client client = new Client(clientProvideInterface,
            clientRequireInterface);
        Server server = new Server(serverProvideInterface,
            serverRequireInterface);
        client.setBankingService(server);

        // Enregistrer le client comme observateur du serveur
        server.addObserver(client);

        // Cr er les R gles
        Rule clientToServerRule = new Rule(RuleType.NAME_MATCH,
            "ClientToServerRule");
        Rule serverToClientRule = new Rule(RuleType.NAME_MATCH,
            "ServerToClientRule");

        // Cr er ProvideRules et RequireRules
        List<Rule> provideRulesList = new ArrayList<>();
        provideRulesList.add(serverToClientRule);
        ProvideRules provideRules = new ProvideRules(provideRulesList);

        List<Rule> requireRulesList = new ArrayList<>();
        requireRulesList.add(clientToServerRule);
        RequireRules requireRules = new RequireRules(requireRulesList);

        // Cr er Glue
        List<Rule> glueRules = new ArrayList<>();
        glueRules.add(clientToServerRule);
        glueRules.add(serverToClientRule);
        Glue glue = new Glue(glueRules);

        // Cr er le Connecteur
        Connector connector = new Connector(glue, provideRules, requireRules);

        // Cr er les Attachements
        ProvidedPortRequiredRulesAttachment clientAttachment = new
            ProvidedPortRequiredRulesAttachment(clientProvidedPort,
            requireRules);
        RequiredPortProvidedRulesAttachment serverAttachment = new
            RequiredPortProvidedRulesAttachment(serverRequiredPort,
            provideRules);
```

```
94
95          // Simuler l'interaction entre le Client et le Serveur
96          System.out.println("Simulating interaction between Client and
                Server...");
97
98          // Le Serveur change d' tat  et notifie les observateurs
99          server.changeState();
100
101         // Configurer les informations d'identification du Client
102         client.setCredentials("user2", "password2"); // Exemple de credentials
103
104         // Authentifier le Client via le Serveur en utilisant la m thode du
                Client
105         client.authenticate(server);
106
107         // V rifier si l'authentification a r ussi en utilisant le token du
                Client
108         if (server.isSessionValid(client.getUsername(),
                client.getSessionToken())) {
109             System.out.println("Client authenticated successfully.");
110
111             // Utiliser les services bancaires via le client
112             client.createAccount("ACC123", 1000.0);
113             client.createAccount("ACC456", 500.0);
114             client.performTransaction("TXN001", "ACC123", "ACC456", 200.0);
115         } else {
116             System.out.println("Client authentication failed. Cannot perform
                    banking operations.");
117         }
118
119         System.out.println("Simulation completed successfully!");
120     }
121 }
```

Listing 16 – Classe `App`

6.1.2 Explication des Interactions

Dans `App.java`, nous configurons les interfaces fournies et requises pour le client et le serveur, créons les ports et les services associés, et établissons les règles et les connecteurs qui régissent les interactions. Les attachements lient les ports fournis et requis aux règles appropriées, assurant ainsi que les connexions entre les services respectent les contraintes définies.

7 Évaluation du Projet

7.1 Spécification et Conception

- **Diagrammes UML de Classe au Niveau M2 :** Les diagrammes UML fournis (Figure 1 et Figure 2) illustrent les relations entre les classes du méta-modèle et du modèle client-serveur. Ils montrent clairement les associations entre les composants, connecteurs, services, et règles.
- **Clarté de la Spécification :** La spécification technique est bien définie avec des descriptions détaillées des composants et de leurs interactions, facilitant la compréhension et la mise en œuvre du système.

7.2 Codage et Choix Architecturaux

- **Qualité du Code :** Le code Java est structuré de manière modulaire, respectant les principes de l'architecture composant-connecteur. Les classes sont bien définies avec des responsabilités claires.
- **Implémentation des Patterns :** L'intégration du pattern Observer permet une communication efficace et asynchrone entre le serveur et les clients, améliorant la réactivité du système.
- **Gestion des Règles :** La mise en place d'un système de règles via `Rule`, `RuleType`, et `Glue` assure la cohérence des interactions entre les services.

- **Extensibilité :** L'architecture permet l'ajout de nouveaux composants et connecteurs sans perturber l'ensemble du système, facilitant ainsi l'évolution future.

7.3 Rôle dans le Système Client-Serveur

- Les classes définissent les différents composants du système, chacun ayant des responsabilités spécifiques.
- Les instances permettent de simuler des interactions réelles entre des clients et le serveur.
- Les sous-classes peuvent être utilisées pour étendre les fonctionnalités des composants de base, permettant ainsi une meilleure modularité et réutilisabilité.

8 Extensions et Compléments de l'Architecture

Pour enrichir l'architecture initiale et répondre aux besoins fonctionnels et non fonctionnels du système, plusieurs extensions et compléments ont été implémentés. Ces ajouts incluent la gestion des attachements, l'intégration du pattern Observer, la gestion des règles, ainsi que l'authentification et la sécurisation des sessions.

8.1 Gestion des Attachements

Les attachements permettent de relier les ports des composants aux règles définies, facilitant ainsi la communication et la coordination entre les composants et les connecteurs.

8.1.1 Attachment.java

```java
package com.alma.attachment;

import com.alma.core.Port;
import com.alma.rules.RulesInterface;

/**
 * Represents an attachment between a port and a rules interface.
 */
public abstract class Attachment {
    private Port port;
    private RulesInterface rules;

    public Attachment(Port port, RulesInterface rules) {
        this.port = port;
        this.rules = rules;
    }

    public Port getPort() {
        return port;
    }

    public void setPort(Port port) {
        this.port = port;
    }

    public RulesInterface getRules() {
        return rules;
    }

    public void setRules(RulesInterface rules) {
        this.rules = rules;
    }
}
```

Listing 17 – Classe abstraite `Attachment`

8.1.2 ProvidedPortRequiredRulesAttachment.java

```java
package com.alma.attachment;

import com.alma.component.ProvidedPort;
import com.alma.connector.RequireRules;

/**
 * Represents an attachment between a provided port and required rules.
 */
public class ProvidedPortRequiredRulesAttachment extends Attachment {
    public ProvidedPortRequiredRulesAttachment(ProvidedPort port, RequireRules
        rules) {
        super(port, rules);
    }
}
```

Listing 18 – Classe `ProvidedPortRequiredRulesAttachment`

8.1.3 RequiredPortProvidedRulesAttachment.java

```java
package com.alma.attachment;

import com.alma.component.RequiredPort;
import com.alma.connector.ProvideRules;

/**
 * Represents an attachment between a required port and provided rules.
 */
public class RequiredPortProvidedRulesAttachment extends Attachment {
    public RequiredPortProvidedRulesAttachment(RequiredPort port, ProvideRules
        rules) {
        super(port, rules);
    }
}
```

Listing 19 – Classe `RequiredPortProvidedRulesAttachment`

8.2 Pattern Observer

L'intégration du pattern Observer permet au serveur de notifier les clients des changements d'état ou des événements importants, favorisant ainsi une communication asynchrone et réactive.

8.2.1 Observer.java et Subject.java

```java
package com.alma.observer;

/**
 * Represents an observer in the Observer design pattern.
 * Observers are notified when the subject's state changes.
 */
public interface Observer {

    /**
     * Updates the observer based on changes in the subject with a message.
     */
    void update(String message);
}
```

Listing 20 – Interface `Observer`

```java
package com.alma.observer;

/**
```

```
 * Represents a subject in the Observer design pattern.
 * A subject manages a list of observers and notifies them of state changes.
 */
public interface Subject {
    /**
     * Adds an observer to the subject.
     *
     * @param observer the observer to be added
     */
    void addObserver(Observer observer);

    /**
     * Removes an observer from the subject.
     *
     * @param observer the observer to be removed
     */
    void removeObserver(Observer observer);

    /**
     * Notifies all registered observers of a state change with a message.
     */
    void notifyObservers(String message);
}
```

Listing 21 – Interface `Subject`

8.3 Gestion des Règles

Les règles définissent des contraintes et des logiques spécifiques pour les interactions entre les composants et les connecteurs.

8.3.1 RulesInterface.java et RuleType.java

```
package com.alma.rules;

import com.alma.core.Rule;
import java.util.List;

/**
 * Abstract class representing an interface for rules.
 */
public abstract class RulesInterface {
    private List<Rule> rules;

    public RulesInterface(List<Rule> rules) {
        this.rules = rules;
    }

    public List<Rule> getRules() {
        return rules;
    }

    public void setRules(List<Rule> rules) {
        this.rules = rules;
    }
}
```

Listing 22 – Classe abstraite `RulesInterface`

8.4 Gestion des Règles

Les règles définissent des contraintes et des logiques spécifiques pour les interactions entre les composants et les connecteurs.

8.4.1 RulesInterface.java et RuleType.java

```java
package com.alma.rules;

import com.alma.core.Rule;
import java.util.List;

/**
 * Abstract class representing an interface for rules.
 */
public abstract class RulesInterface {
    private List<Rule> rules;

    public RulesInterface(List<Rule> rules) {
        this.rules = rules;
    }

    public List<Rule> getRules() {
        return rules;
    }

    public void setRules(List<Rule> rules) {
        this.rules = rules;
    }
}
```

Listing 23 – Classe abstraite `RulesInterface`

```java
package com.alma.rules;

/**
 * Enum representing different types of rules that can be applied to services.
 */
public enum RuleType {
    NAME_MATCH,          // Rule for matching names of services
    COMPATIBILITY_CHECK // Rule for checking compatibility of services
}
```

Listing 24 – Enum `RuleType`

8.4.2 Rule.java

```java
package com.alma.core;

import com.alma.rules.RuleType;

/**
 * Represents a rule that can be applied to services during the binding process.
 */
public class Rule {
    private RuleType ruleType; // The type of rule (e.g., NAME_MATCH,
        COMPATIBILITY_CHECK)
    private String ruleName;   // A name for the rule (can be a description)

    public Rule(RuleType ruleType, String ruleName) {
        this.ruleType = ruleType;
        this.ruleName = ruleName;
    }

    public RuleType getRuleType() {
        return ruleType;
    }

    public void setRuleType(RuleType ruleType) {
        this.ruleType = ruleType;
```

```
    }

    public String getRuleName() {
        return ruleName;
    }

    public void setRuleName(String ruleName) {
        this.ruleName = ruleName;
    }

    /**
     * Executes the rule on the provided service and required service.
     *
     * @param providedService The provided service.
     * @param requiredService The required service.
     * @return True if the rule can be applied, false otherwise.
     */
    public boolean apply(Service providedService, Service requiredService) {
        switch (ruleType) {
            case NAME_MATCH:
                return applyNameMatch(providedService, requiredService);
            case COMPATIBILITY_CHECK:
                return applyCompatibilityCheck(providedService,
                    requiredService);
            default:
                return false;
        }
    }

    // Logic for the NAME_MATCH rule
    private boolean applyNameMatch(Service providedService, Service
        requiredService) {
        return
            providedService.getServiceName().equalsIgnoreCase(requiredService.getServiceName()
    }

    // Logic for the COMPATIBILITY_CHECK rule
    private boolean applyCompatibilityCheck(Service providedService, Service
        requiredService) {
        // Example compatibility check logic,
        // TODO : it can be customized.
        return providedService.getGlue().equals(requiredService.getGlue());
    }
}
```

Listing 25 – Classe `Rule`

9 Classes Complémentaires

Afin de compléter l'architecture et de répondre aux exigences fonctionnelles et non fonctionnelles du système, plusieurs classes supplémentaires ont été implémentées, notamment pour gérer les attachements, observer les changements, et appliquer des règles spécifiques.

9.1 Additional Component Classes

ProvidedPort.java

```
package com.alma.component;

import com.alma.core.Port;

/**
 * Represents a port provided by a component.
 */
public class ProvidedPort extends Port {
    public ProvidedPort(String name) {
```

```
        super(name);
    }
}
```

Listing 26 – Classe `ProvidedPort`

RequiredPort.java

```
package com.alma.component;

import com.alma.core.Port;

/**
 * Represents a port required by a component.
 */
public class RequiredPort extends Port {
    public RequiredPort(String name) {
        super(name);
    }
}
```

Listing 27 – Classe `RequiredPort`

ProvidedService.java et RequiredService.java

```
package com.alma.component;

import com.alma.core.Service;

/**
 * Represents a service provided by a component.
 */
public class ProvidedService extends Service {
    public ProvidedService(String serviceName) {
        super(serviceName);
    }
}
```

Listing 28 – Classe `ProvidedService`

```
package com.alma.component;

import com.alma.core.Service;

/**
 * Represents a service required by a component.
 */
public class RequiredService extends Service {
    public RequiredService(String serviceName) {
        super(serviceName);
    }
}
```

Listing 29 – Classe `RequiredService`

9.1.1 Configuration.java

```
package com.alma.configuration;

import com.alma.component.Component;
import com.alma.connector.Connector;
import java.util.List;

/**
 * Represents a configuration in the system.
 * Contains components and connectors.
```

```
 */
public class Configuration {
    private List<Component> components;
    private List<Connector> connectors;

    public Configuration(List<Component> components, List<Connector>
        connectors) {
        this.components = components;
        this.connectors = connectors;
    }

    public List<Component> getComponents() {
        return components;
    }

    public void setComponents(List<Component> components) {
        this.components = components;
    }

    public List<Connector> getConnectors() {
        return connectors;
    }

    public void setConnectors(List<Connector> connectors) {
        this.connectors = connectors;
    }
}
```

Listing 30 – Classe `Configuration`

10 Conclusion

Ce projet nous a permis de concevoir et d'implémenter une architecture composant-connecteur en partant d'un méta-modèle abstrait jusqu'à une application concrète. Cela a renforcé notre compréhension des principes architecturaux et de leur application pratique dans le développement logiciel. Les extensions apportées, telles que la gestion des attachements, l'intégration du pattern Observer, et la mise en place d'un système de règles, ont permis de créer une architecture robuste, flexible et maintenable, répondant ainsi aux exigences fonctionnelles et non fonctionnelles du système client-serveur bancaire.