**Documentation technique**

Table des matières

[La décomposition en modules de l’application 2](#_Toc41741366)

[Les spécifications des interfaces 3](#_Toc41741367)

[Diagrammes UML : package, composants, déploiement, classes 4](#_Toc41741368)

[Les choix de conception que vous avez faits avec argumentaire 5](#_Toc41741369)

[Les patrons de conception utilisés 5](#_Toc41741370)

[Analyse de la conception selon les principes SOLID 6](#_Toc41741371)

# 

# **La décomposition en modules de l’application**

Gestion des contacts

Gestion des messages

Gestion des données & sauvegardes

Gestion serveur

Affichage

# **Les spécifications des interfaces**

Une image contenant capture d’écran

Description générée automatiquement

Liste des contacts et ajout font appel au module de la gestion des contacts.

A gauche la liste des discussions et la possibilité d’en créer une avec le bouton en bas.

Elle s’affiche ensuite à droite avec son titre, les personnes concerner par la discussion.

En bas on crée un message du format que l’on souhaite (module gestion des messages) l’on peut voir les anciens messages plus haut (module gestion des données et sauvegarde).

Un message envoyé va faire appel au module gestion serveur.

# **Diagrammes UML : package, composants, déploiement, classes**

Une image contenant capture d’écran

Description générée automatiquement

Le diagramme des classes UML étant bien trop complexe pour être affiché sur cette page, il sera disponible en version complète et plus claire sur Github.

# **Les choix de conception que vous avez faits avec argumentaire**

Nous avons choisi de découper le projet en module pour qu’il respecte des principes de conception. En effet, notre code est modulaire et facilement applicable à d’autre projet. Il gagne ainsi en efficacité et en performance. Il respect les principes SOLID. Le projet et le code est donc propice aux changements et aux évolutions.

Chaque module fonctionne de façon individuelle et on retrouve 5 modules distincts.

# **Les patrons de conception utilisés**

Notre programme lit des fichiers JSON au démarrage de l’application afin de récupérer les discussions existantes. Afin de respecter les principes SOLID, nous avons mis en place le patron Observateur.

Chaque fichier JSON correspond à une discussion. Le programme va créer un objet Discussion pour chaque fichier et va appeler une fonction update() qui va indiquer à l’interface utilisateur qu’elle doit créer un nouvel élément graphique correspondant à la discussion. Cela facilite grandement la réutilisation d’objets de type Discussion.

Nous utilisons également le patron Façade. Le programme principal se charge uniquement d’afficher l’interface utilisateur. Le code derrière se chargera d’appeler toutes les fonctions nécessaires pour importer, exporter des discussions, gérer les menus contextuels… Cela nous permet d’avoir un programme un peu plus simple.

# **Analyse de la conception selon les principes SOLID**

*Single Responsibilty Principle*

*« A class should have one reason to change. »*

*Une classe à une responsabilité* pour éviter que les classes soient liée. Les modifications apportées à une responsabilité n’impacteront pas les autres. Ainsi on diminue la rigidité et la fragilité du code.

*Open Closed Principle*

*« Classes, methods should be open for extension, but closed for modifications. »*

Une classe, une méthode, un module doit pouvoir être étendu, supporter différentes implémentations. Une nouvelle implémentation aura pour impact l’ajout d’une condition dans la méthode.

#### *Liskov Substitution Principle*

« Subtypes must be substituable for their base types. »

*« Les sous classes doivent pouvoir être substituées à leur classe de base sans altérer le comportement de ses utilisateurs. »* Il ne faut pas modifier les valeurs des attributs de la classe principale d’une manière inadaptée, ne respectant pas le contrat défini par la méthode.

#### *Interface Segregation Principle*

« Clients should not be forced to depend on methods that they do not use. »

Le but de ce principe est d’utiliser les interfaces pour définir des contrats, des ensembles de fonctionnalités répondant à un besoin fonctionnel, plutôt que de se contenter d’apporter de l’abstraction à nos classes

#### *Dependency Inversion Principle*

« High level modules should not depend on low level modules. Both should depend on abstractions. »

« Abstractions should not depend on details. Details should depend on abstractions. »

Le principe de DIP dit que les modules de haut niveau ne doivent pas dépendre de modules de plus bas niveau. Lorsque ces modules dépendent de modules de plus bas niveau, les modifications effectuées dans les modules « bas niveau » peuvent avoir des répercussions sur les modules « haut niveau » et les « forcer » à appliquer des changements.