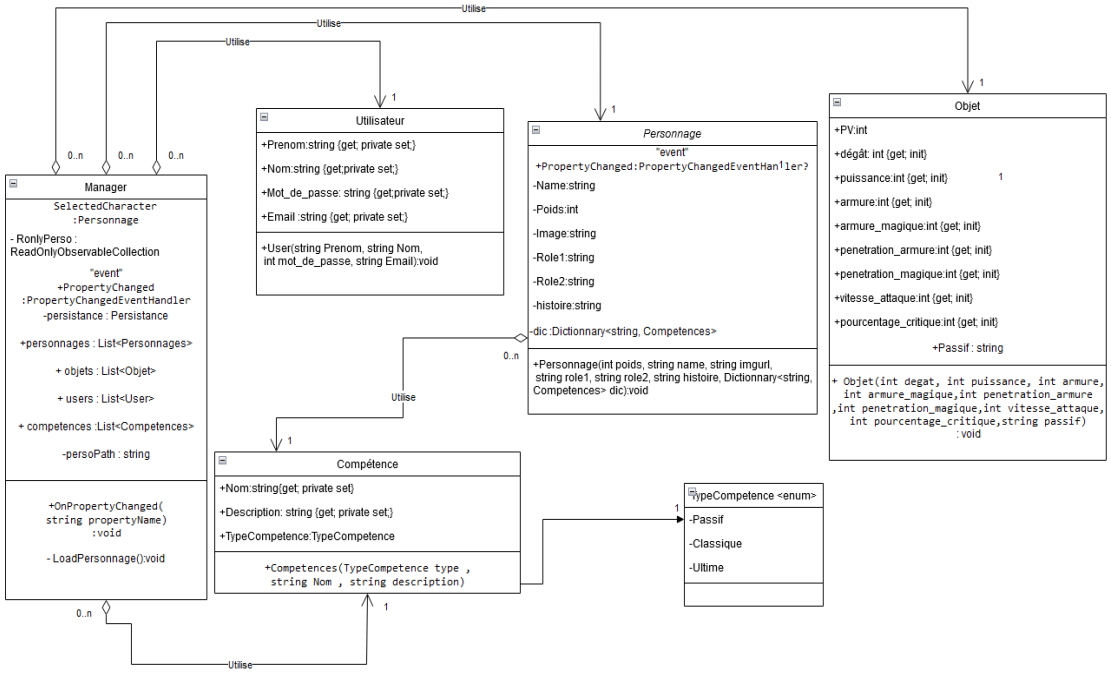
Diagramme de classe



Dans ce diagramme de classe les utilisateurs, les objets et les personnages sont des éléments indépendants entre eux. La classe utilisateur sera utile pour permettre la connexion des utilisateurs, la classes objets est utile pour afficher les informations des équipements et permettre d'enregistrer les ensembles d'équipement. La classe personnage permet de répertorier les personnages et leurs information respective pour ensuite pouvoir l'afficher dans une Page Détaille, elle utilise plusieurs fois la classe compétence (5 fois et ce pour tous les champions, il n'est pas possible de réduire ou augmenté ce nombre), la classe compétence utilise elle-même la classe enum TypeCompetence, une seule fois par compétence.

diagramme de classes mettant en avant la partie persistance

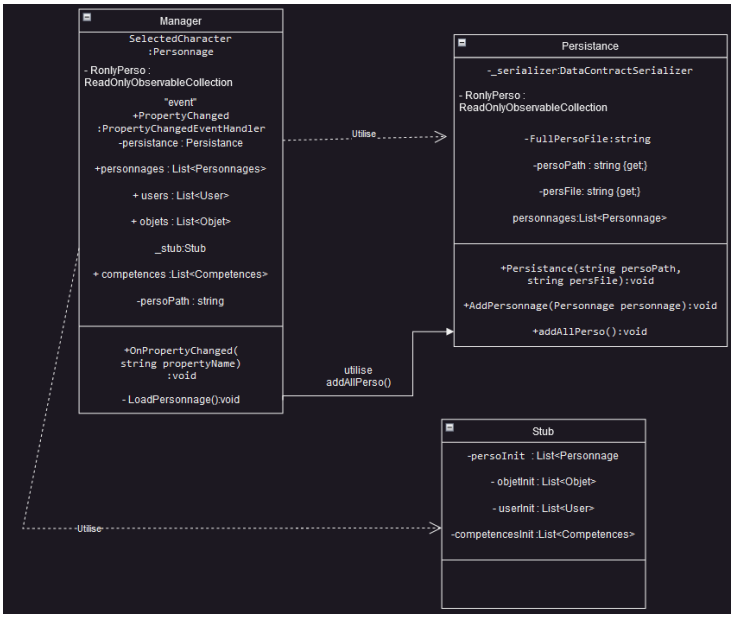
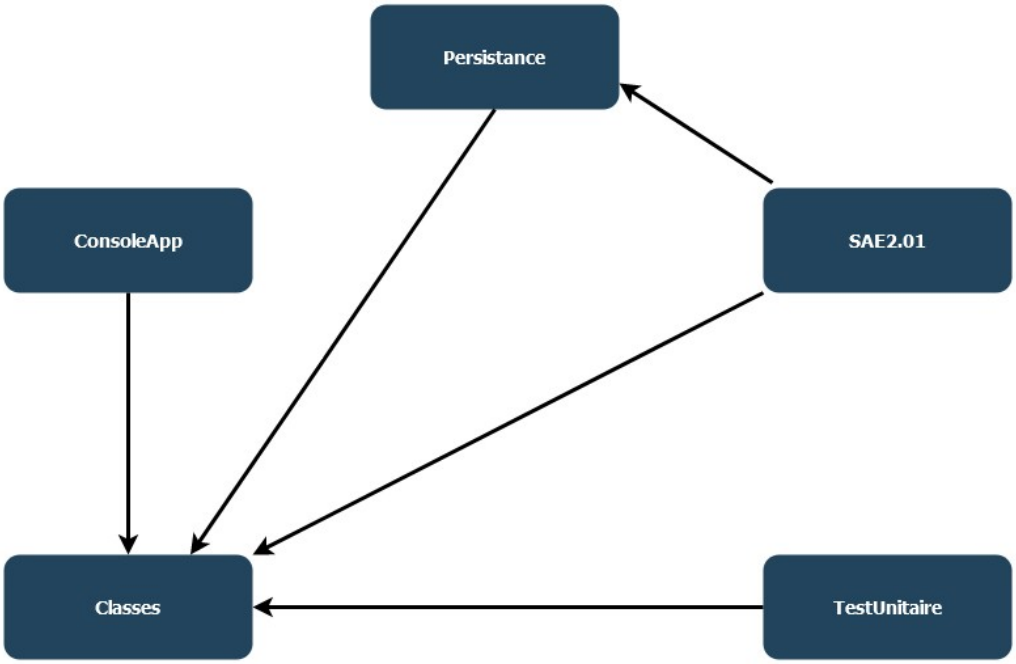


Diagramme de Paquetage



Dans mon diagramme de paquetage il y a 5 projets, tous d'abord Classes dont tous les autres projets sont dépendant et qui contient toutes les classes vues dans le diagramme de classe, nous avons ensuite ConsoleApp qui permet de faire les test simples des classes et TestUnitaire qui contient les tests unitaires des classes. Il y le projet Persistance qui contient le stub et la persistance et dont le projet SAE2.01 dépend, et ce dernier projet contient toutes les pages XAML, leurs .cs et le manager.

description écrite de l'architecture :

Le manager dans l'application joue un rôle essentiel en agissant comme une façade, qui est un patron de conception structurel largement utilisé. La façade offre une interface simplifiée pour accéder à des bibliothèques, des frameworks ou tout autre ensemble complexe de classes. Son objectif principal est de faciliter la structuration du projet en orchestrant les appels aux méthodes des différentes classes. En regroupant l'accès à toutes les fonctions de l'application, le manager simplifie la navigation et l'utilisation du code, offrant ainsi une interface cohérente et unifiée pour les développeurs.

Le stub, quant à lui, peut être considéré comme une variante du patron Procuration. Il agit comme un substitut d'un objet distant ou d'un service, encapsulant l'accès et la communication avec cet objet distant. En fournissant une interface locale similaire à celle de l'objet réel, le stub permet d'interagir avec l'objet distant sans nécessiter la connaissance des détails de l'accès à distance ou de la complexité de la communication.

Dans le contexte de la persistance des données, l'utilisation du patron repository est courante. Celui-ci agit comme une couche intermédiaire entre l'application et la source de données, telle qu'une base de données ou un fichier. Son rôle principal est de fournir une interface cohérente et simplifiée pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur les données persistantes. En masquant les détails techniques spécifiques du stockage, le repository permet à l'application de manipuler les données de manière abstraite et indépendante de la source de données sous-jacente.

Dans le cadre d'un projet MAUI, les différents sous-projets se prêtent bien à l'utilisation du patron composite. Ce dernier permet d'agencer les objets en une structure arborescente, facilitant ainsi la manipulation de ces objets individuellement ou en groupe. En structurant les fichiers du projet en différentes arborescences, les sous-projets du projet MAUI offrent une organisation claire et logique, favorisant la maintenabilité et la modularité du code.

Le patron Bridge peut être utilisé dans un projet MAUI pour séparer l'abstraction de la plateforme spécifique. Par exemple, l'abstraction peut représenter une fonctionnalité d'interface utilisateur telle qu'un bouton ou une zone de texte, tandis que l'implémentation représente la manière dont cette fonctionnalité est mise en œuvre sur différentes plates-formes (iOS, Android, Windows, etc.). Le pont permet de lier ces deux aspects de manière flexible, permettant ainsi d'étendre et de modifier les fonctionnalités de l'application tout en minimisant les dépendances entre la logique métier et les détails d'implémentation spécifiques à une plate-forme.

Le ResourceDictionary correspond au patron structurel "Poids Mouche" car il permet d'économiser les ressources en partageant les données communes entre plusieurs objets. Il permet de stocker les données de chaque élément XAML (comme un ImageButton, Label, etc.) de manière centralisée, évitant ainsi la duplication de ces données et optimisant l'utilisation de la mémoire.

Il convient de noter que tous les projets dépendent du projet "classe" qui contient les classes nécessaires au fonctionnement de l'application. De plus, le projet SAE2.01 présente une dépendance supplémentaire vis-à-vis du projet "Persistance", soulignant ainsi l'interdépendance des différents éléments d'un projet. Cette dépendance peut être gérée en utilisant des mécanismes de dépendance et d'injection de dépendances appropriés pour assurer une bonne modularité et une bonne gestion des dépendances dans le projet MAUI.