I decided to use solid principles

For example I held to creating **single responsibility**, this is clear if you look at the verification, it only has 1 reason to change and its only responsible for verification purposes. Also User is has its own repositories splitted. So if preferences in the preference context need to be manipulated, the preference repository comes in to place and the user persistence will no longer have to do different jobs.

**Open-closed:**

The cupid application can change front end and database this application is domain driven so even the business could technically change but the domain will not suffer in that process.

I also used **Interface Segregation**, this means that I am using interfaces to prevent conflicts when changing implementations.

Finally I used **Dependency Inversion** for dependencies, so with my user adapter the User Service can safely say that there WILL be a save method and it doesn’t matter which implementation is used, this makes it possible to swap easily between for example data sources but also on the other end, frontends.

Afbeelding met tekst, schermopname, diagram, Parallel

Automatisch gegenereerde beschrijving