Gebruikte design patterns

BaseEntity

Elke entiteit die door de database opgenomen wordt zal een id bevatten. Om herhaling te voorkomen in elke class heb ik dit in base class gestoken waar Person & Appointment van zullen overerven.

Person

Patient & Physician zullen Person als base class gebruiken. Ik heb dit zo gedaan aangezien beide entiteiten een naam, geboortedatum, email, … enzo voort hebben. Hierdoor hoeft dit maar 1 keer gedefineerd te worden.

Address

Ik heb besloten om dit in een aparte class te steken om zo zeker seperation of concerns te waarborgen. Momenteel wordt het enkel door Patient gebruikt voor toekomstige facturatie, maar mocht er later besloten worden om dit ook voor de artsen te implementeren, dan gaat dit.

ConsoleUtil

Met deze ConsoleUtil class heb ik het DRY-principe toegepast. Dit betekent dat ik herhalende stukken code, zoals het inlezen en valideren van gebruikersinput, heb samengebracht op één centrale plek. Dit gaat onderhoud vergemakkelijken.

ReadInput

Bij elke vorm van input zal er altijd een prompt voorafgaand gevraagd worden en dit steeds opnieuw gevraagd worden tot de input van het juiste type is. Om deze logica wat te generaliseren heb ik het in deze functie gestoken die vervolgens wordt gebruikt door al de andere functies in ConsoleUtil.

Repository

Elke repository erft een generieke basisrepository Repository<T>. Hier zal de implementatie van algemene CRUD-operaties plaatsvinden. Deze base repository is gekoppeld aan een interface IRepository, waardoor er gewerkt kan worden met de abstractie verder in het project.

Dit zorgt voor meer flexibiliteit in de code. Ook zal testen vergemakkelijken door de onafhankelijkheid en er moet geen rekening gehouden worden met de interne logica, enkel het opgelegde contract voorwaarden van het interface (mock).

In het algemeen is dit ook beter voor dependency management. Als de repository implementatie rechtstreeks gebruikt en de naam vervolgens veranderd, dan moet dit in meerdere files aangepast worden. Dit probleem is er niet mocht je het aanspreken met een interface.

Elke repository heeft bovenop ook nog zijn eigen interface, aangezien ze elk nog verschillen van elkaar met extra functionaliteit eigen aan de entiteit.

Services

Elke entiteit heeft zijn eigen service. Dit heb ik zo gedaan om logica van UI te scheiden. Dit zal voor testen handig zijn, aangezien je niet afhankelijk bent van de UI.

Net zoals de Person entity, zal PersonService functionaliteit bevatten dat toepasselijk is voor zowel Patient als Physician. Hierdoor heb ik besloten om PatientService en PhysicianService te laten overerven van PersonService.

Forms

Form classes zijn form gedeeltes van entiteiten die herhaald (zullen) worden. Dit zou ook in Program.cs gestoken kunnen worden, maar om het aantal lijnen per file te beperken heb ik gekozen om dit in een aparte static class te steken.

AppointmentStatus

Dit is een enum waarmee verschillende statussen centraal en één keer gedefinieerd worden. Het maakt de code overzichtelijker, makkelijker te onderhouden en minder foutgevoelig. De enum is voornamelijk toegevoegd geweest als voorbereiding op toekomstige uitbreidingen van het programma.

Error handling

In een ideaal scenario krijgt elke klasse zijn eigen logger, zodat fouten nauwkeurig kunnen worden getraceerd. Dit wordt gecombineerd met een algemene error handler die onverwachte fouten opvangt en op een nette manier afhandelt.