Richtlijn voor het portfolio onderdeel ‘Programmeren van onafhankelijk werkende applicaties vanuit een OO-benadering (HBO-ICT 2025)

In de lessen C# komen verschillende concepten aan de orde, die we terug willen zien in de opgeleverde onderdelen voor het project. Het gaat hier dan met name om EvL 2 Programmeren van onafhankelijk werkende applicaties vanuit een OO-benadering.

Naast het aantoonbaar doorlopen van de ontwikkelcyclus (requirements → FO → TO → code → testen → oplevering) verwacht je van professionals dat ze feedback vragen/krijgen/verwerken/geven/terugkoppelen. Dit geldt dus ook voor de programmacode. Het is belangrijk om hierbij een gerichte vraag te stellen (b.v. de code geeft invulling aan requirement x, en is ontworpen volgens y en getest volgens z. Kun je feedback geven over de implementatie van het stukje TO in code en of de testen goed aansluiten bij de requirements?). Daarnaast wil je dat de code aan een aantal basisvoorwaarden voor goede code voldoet.

Basisvoorwaarden: Goede code versus matige code:

Realiseer je dat zowel goede als matige code te compileren is, dus syntactisch goed wil nog niet zeggen dat de code goed is. Goede code is goed met reden, bijvoorbeeld is het onderhoudbaar of is het herbruikbaar. Het uitvoeren van peerreviews in de context van softwareontwikkeling is een belangrijk onderdeel van het leerproces. Het stelt je in staat om kritisch te denken, feedback te geven en te ontvangen, en van elkaar te leren. Hier is een set van algemene richtlijnen je kunt gebruiken voor het beoordelen van code geschreven in C#.

Soort	Goede code	Matige code
Leesbaarheid en stijl	**Code Indentatie en Spatiëring**: Goede code heeft consistente indentatie en spatiëring, wat het goed leesbaar maakt / logica goed te volgen / overzichtelijk.	Matige code heeft inconsistente indentatie en spatiëring, wat het moeilijk maakt om de logica te volgen.
	Variabelenamen: Namen zijn beschrijvend en consistent, waardoor het duidelijk is wat er met de functie van variabelen bedoelt wordt.	Namen zijn niet beschrijvend of inconsistent, waardoor het lastig is om de functie van variabelen te begrijpen
	Commentaar en Documentatie: Er zijn voldoende commentaren gegeven, waardoor het doel en de werking van de code goed te begrijpen zijn.	Commentaar beschrijft wat er al staat of er zijn weinig of onduidelijke commentaren, waardoor het doel en de werking van de code niet goed te begrijpen zijn.
	Structuur: De code heeft korte functies (minder dan 10 regels) / classes met één verantwoordelijkheid / één taak, waardoor het goed te begrijpen is, te onderhouden en te testen.	De code heeft langere functies / classes met meerdere verantwoordelijkheden, waardoor het moeilijk is om de functies te begrijpen en te onderhouden / te testen.
Functionaliteit en Logica	**Correctheid**: De code heeft geen bugs en doet volledig wat het moet doen (aangetoond met testen) en is daarmee volledig functioneel..	De code heeft kleine bugs of doet niet volledig wat het moet doen, maar is grotendeels functioneel.

	Foutafhandeling: Er is volledige foutafhandeling en deze pakt alle mogelijke uitzonderingen aan.	Er is enige foutafhandeling, maar het is onvolledig en pakt niet alle mogelijke uitzonderingen aan.
	Randgevallen: Alle randgevallen (b.v. null waarden) worden behandelden correct afgehandeld.	Sommige randgevallen worden behandeld, maar andere worden genegeerd of incorrect afgehandeld.
	Efficiëntie: De code werkt, maar er zijn betere of efficiëntere manieren om hetzelfde te doen.	Methodes hebben meer dan 10 regels.
Architectuur en Ontwerp	**Modulariteit**: De code is gemoduleerd, goed ingedeeld in namespaces, folders & classes, waarmee de verantwoordelijkheden goed zijn belegd (iedere classe één verantwoordelijkheid, iedere functie één taak).	De code is enigszins gemoduleerd, maar sommige functies of klassen zijn te groot of hebben meerdere verantwoordelijkheden.
	Herbruikbaarheid: Aantal delen van de code zijn herbruikbaar doordat elke verantwoordelijkheid is geïsoleerd.	Enkele delen van de code zijn herbruikbaar, maar het vereist extra werk om ze te isoleren.
	Testbaarheid: De code is goed te testen vanwege lage koppeling, vermijden van globale variabelen en goed parametriseerbare functies.	De code is moeilijk te testen vanwege sterke koppelingen, globale variabelen, of andere anti-patronen.
	Koppeling en Cohesie: Er is een lage koppeling tussen de modules en een sterke cohesie.	Er is een zekere mate van koppeling tussen de modules, en de cohesie binnen de modules is niet ideaal.
Best Practices	**Gebruik van Frameworks en Libraries**: Gebruik van bestaande oplossingen uit frameworks en bibliotheken, waardoor er zo weinig mogelijk zelfgeschreven, foutgevoelige code is.	Matig gebruik van bestaande frameworks en bibliotheken, waardoor er mogelijk veel zelfgeschreven, foutgevoelige code is.
	Coding Standards: Best practices, principes en coding standards voor C# zijn gevolgd.	Niet alle best practices, principes of coding standards voor C# zijn gevolgd, wat leidt tot inconsistentie en foutgevoeligheid.
	Versiebeheer: Er is goed gebruik gemaakt van versiebeheer middels branches, pull requests, merge requests waardoor er altijd een werkende versie geleverd kon worden.	Versiebeheer wordt gebruikt, maar de commit-berichten zijn vaag en niet beschrijvend.
Overige	**Creativiteit en Innovatie**: Er is gebruik gemaakt van algoritmen of andere technieken vanuit andere projecten of domeinen op het probleem waar dit niet eerder is gedaan.	De code volgt traditionele methoden zonder nieuwe of creatieve oplossingen te verkennen.
	Algemene Indruk: De code werkt goed, is goed getest, netjes ingedeeld en ziet er verzorgd uit.	De code werkt tot op zekere hoogte maar mist afwerking en finesse. Er zijn veel gebieden voor verbetering.

C# concepten

De concepten die gebruikt zijn in de C# lessen zijn de volgende zijn in onderstaande tabel beschreven en toegelicht:

Concept	Toelichting
OO-principes	Overerving, overriding, overloading, polymorfism & information hiding
C# class objects	Denk hierbij ook aan static classes, interfaces, abstracte classes etc.
Generics	Zowel op class niveau als method niveau. Als je deze niet zelf gemaakt hebt laat dan wat voorbeelden zien dat je dit hebt toegepast.
Delegates en events	Declaratie van functiepointer, delegeren van taken, ontkoppelen van definitie en uitvoering, filteren van informatie etc. Eigen gemaakte event, of in elk geval het toepassen van een event. Multicast delegate (advanced).
WPF	Property binding, viewmodels, views of andere MVVM onderdelen, keuze voor componenten als StackPanel, Grid, Canvas, Window, Page, UserControl, modeless dialogs of mode dialogs.
Exceptions	Try-catch-finally blocks, of beschrijven hoe excepties zijn opgevangen op andere wijze. Houdt rekening met hierarchie in excepties. Eigen ontwikkelde excepties + throws.
List, array & enums	Keuze van juiste collectie / type voor het juiste type probleem. B.v. enum voor weekdag, soort kaart etc. Als een collectie uitbreidbaar moet zijn dan een List, anders volstaat een array etc.
Lambda expressies	Anonymous functions / classes, lamda expressies, standaard delegates als Action, Predicate, Func.
Linq	Querysyntax of method syntax. Gebruik van let en new voor anonieme types.
Gegevensopslag	EntityFramework, overerven van DbContext / DbSet. Relaties in model classes, primary en foreign keys. Gebruik van repositories voor afschermen van database laag.
Unit testen	Gebruik van xUnit of nUnit. Arrange Act en Assert principe. Werk met TestCase of InlineData voor serie testen. Testen van excepties met Assert.Throws etc.

Hopelijk helpt dit document jullie voldoende op weg naar een mooi en goed geschreven portfolio!

Veel succes!

Eugène van Roden