Software Development Opleiding en AI

Vijf Pijlers moderne Software Development Opleiding

AI verandert de wered en een moderne opleiding software delopement zou moeten bestaan uit vijf pijlers.

Dit stuk legt uit waarom.

image.png

Zo was het vroeger

Decennialang draaide programmeren voor een groot deel om het leren van een programmeertaal.

Een ontwikkelaar begon met een idee:

"Ik wil een knop die gegevens opslaat."

Daarna werd dat idee vertaald naar syntax:

```
if user.clicked():
    save_data()
```

Succes hing grotendeels af van het kennen van de juiste trefwoorden, interpunctie en regels van de programmeertaal. De uitdaging was om ideeën te vertalen naar werkende code.

Veel opleidingen waren daarom sterk gericht op syntax, programmeerconstructies en technische details. Studenten leerden variabelen, if-statements, loops, functies, classes, databases en frameworks. Wie de taal beheerste, kon software bouwen.

Dat blijft belangrijk. Zonder basiskennis van code kun je software niet goed begrijpen, controleren of verbeteren. Maar de context waarin software wordt ontwikkeld, verandert snel.

En hoe is het vandaag geworden?

Generatieve AI verandert de relatie tussen idee en code fundamenteel.

De uitdaging is niet langer alleen:

> **Ideeën → Syntax**

maar steeds vaker:

> **Ideeën → Duidelijkheid** -> AI -> Code

Het vermogen om intentie helder uit te drukken wordt een van de waardevolste vaardigheden in softwareontwikkeling.

Een AI kan vaak syntax genereren, code aanvullen, fouten uitleggen, functies herschrijven of testvoorbeelden maken. Maar AI kan niet betrouwbaar bepalen wat de gebruiker werkelijk bedoelt, welke keuzes verstandig zijn, welke risico's acceptabel zijn of wanneer een oplossing goed genoeg is.

Die verantwoordelijkheid blijft bij mensen.

Hoe duidelijker een ontwikkelaar een probleem kan beschrijven, hoe beter AI kan helpen bij het bouwen van een oplossing.

De moderne ontwikkelaar moet daarom vragen kunnen beantwoorden zoals:

- Welk probleem lossen we op?
- Voor wie lossen we het op?
- Waarom is dit probleem belangrijk?
- Hoe ziet succes eruit?
- Wat moet er gebeuren in normale situaties?
- Wat moet er gebeuren als er iets misgaat?
- Welke uitzonderingen zijn mogelijk?
- Welke beperkingen zijn er?
- Welke gegevens gebruiken we?
- Welke gegevens mogen we juist niet gebruiken?
- Welke voorbeelden maken het gewenste gedrag duidelijk?

- Hoe testen we of de oplossing correct werkt?
- Hoe beoordelen we of de oplossing veilig, bruikbaar en onderhoudbaar is?
- Komt het resultaat overeen met de oorspronkelijke bedoeling?

Van programmeur naar softwareontwikkelaar

De rol van de ontwikkelaar verschuift. Waar vroeger veel nadruk lag op het zelf schrijven van iedere regel code, komt er steeds meer nadruk te liggen op begrijpen, ontwerpen, controleren en verbeteren.

Ontwikkelaars treden steeds vaker op als:

- Probleemanalist
- Ontwerper
- Architect
- Programmeur
- Reviewer
- Tester
- Communicator
- Kwaliteitsbewaker

Oude denkwijze

> "Hoe schrijf ik deze loop in Python?"

Nieuwe denkwijze

"Ik heb een systeem nodig dat inzendingen van studenten verwerkt, dubbele inzendingen afwijst, duidelijke feedback geeft en bruikbaar blijft voor docenten met weinig technische kennis."

De AI kan helpen om de loop te schrijven. De ontwikkelaar moet bepalen wat het systeem moet doen, welke kwaliteitseisen gelden, welke uitzonderingen bestaan en hoe succes wordt gemeten.

Dit betekent dat vaardigheden die vroeger soms als 'soft skills' werden gezien, steeds meer technische vaardigheden worden:

- Helder communiceren
- Requirements definiëren
- Voorbeelden geven
- Randgevallen herkennen
- Gebruikers begrijpen
- Resultaten beoordelen
- Instructies verfijnen
- Code lezen en controleren
- Fouten opsporen
- Kwaliteit bewaken
- Veiligheid beoordelen
- Onderhoudbaarheid inschatten

Prompting is geen trucje

Vaak wordt gesproken over "prompt engineering". Daardoor lijkt het alsof werken met AI vooral draait om slimme formuleringen.

In werkelijkheid is een goede prompt meestal het resultaat van goed nadenken.

Een ontwikkelaar die het probleem begrijpt, duidelijke voorbeelden geeft, randgevallen benoemt en succescriteria definieert, zal doorgaans betere resultaten te krijgen dan iemand die alleen probeert een slimme prompt te schrijven.

Een goede prompt is geen magische zin. Een goede prompt is een compacte beschrijving van goed denkwerk.

Zwakke prompt

> "Maak een login-systeem."

Sterkere prompt

> "Maak een eenvoudig login-systeem in PHP met een formulier, server-side validatie, password_hash, password_verify, sessies, foutmeldingen zonder gevoelige informatie en bescherming tegen lege invoer. Leg daarna uit hoe ik dit kan testen."

De tweede prompt is beter omdat de ontwikkelaar beter heeft nagedacht over techniek, veiligheid, invoer, foutafhandeling en testbaarheid.

De kwaliteit van de uitkomst wordt steeds vaker bepaald door de kwaliteit van het denkwerk dat eraan voorafgaat.

Wat betekent dit voor studenten?

Studenten die alleen leren hoe ze code moeten schrijven, missen een belangrijk deel van de nieuwe werkelijkheid.

Technische kennis blijft noodzakelijk, maar wordt aangevuld met vaardigheden zoals analyseren, ontwerpen, testen, samenwerken, communiceren en kritisch beoordelen.

Een succesvolle ontwikkelaar van de toekomst begrijpt niet alleen hoe software werkt, maar ook waarom die software gebouwd wordt.

De waarde verschuift steeds meer van het produceren van code naar het oplossen van problemen.

Dat betekent niet dat studenten minder technisch hoeven te worden. Het betekent juist dat zij techniek beter moeten kunnen plaatsen in een groter geheel.

Een student moet niet alleen kunnen zeggen:

> "Mijn code werkt."

maar ook:

- Waarom werkt deze oplossing?
- Voor welk probleem is dit een goede oplossing?
- Welke keuzes heb ik gemaakt?
- Welke alternatieven waren er?
- Welke risico's zitten erin?
- Hoe kan ik bewijzen dat het werkt?
- Kan iemand anders deze code begrijpen en onderhouden?

Wat moeten aankomende softwareontwikkelaars leren?

De centrale vraag voor een moderne opleiding Software Development is:

Wat moeten wij studenten leren in een wereld waarin AI steeds meer code kan genereren?

Het antwoord is niet: minder programmeren.

Het antwoord is: anders leren programmeren.

Studenten moeten nog steeds code leren schrijven, maar zij moeten daarnaast leren hoe zij softwareproblemen analyseren, oplossingen ontwerpen, AI effectief gebruiken, gegenereerde code controleren en verantwoordelijkheid nemen voor het eindresultaat.

Een moderne softwareopleiding zou daarom moeten bouwen op:

vijf pijlers

1. begrijpen van code
2. Denken in problemen en requirements
3. Ontwerpen voordat je gata bouwen
4. Werken met Ai als hulpmiddel
5. testen, reviewen en verbteren

1. Begrijpen van code

Studenten moeten code kunnen lezen, uitleggen, aanpassen en debuggen. Ook wanneer AI de eerste versie heeft geschreven, moet de student begrijpen wat de code doet.

Belangrijke onderwerpen blijven:

- Variabelen
- Datatypes
- If-statements
- Loops
- Functies
- Arrays en lijsten
- Objecten en classes
- Bestanden
- Formulieren
- Databases

- API's
- Foutmeldingen
- Debugging

Het doel is niet dat studenten alle syntax uit hun hoofd kennen. Het doel is dat zij code kunnen begrijpen, redeneren over gedrag en fouten kunnen herstellen.

2. Denken in problemen en requirements

Software begint niet met code, maar met een probleem.

Studenten moeten leren om een vaag idee om te zetten naar duidelijke requirements.

Vaag idee

"Maak een reserveringssysteem."

Duidelijker uitgewerkt

"Een student moet een lokaal kunnen reserveren voor een projectgroep. Een lokaal mag niet dubbel geboekt worden. Een reservering heeft een datum, starttijd, eindtijd, lokaalnummer en naam van de student. De gebruiker krijgt een duidelijke foutmelding als het lokaal al bezet is."

Studenten moeten leren vragen stellen voordat ze gaan bouwen.

- Wie gebruikt het systeem?
- Wat wil de gebruiker bereiken?
- Welke gegevens zijn nodig?
- Welke regels gelden er?
- Wat mag absoluut niet gebeuren?
- Welke uitzonderingen zijn belangrijk?
- Wanneer is de opdracht klaar?

3. Ontwerpen voordat je bouwt

Studenten moeten leren dat softwareontwikkeling meer is dan direct beginnen met code.

Voor veel opdrachten hoort eerst een eenvoudig ontwerp:

- Een schets van het scherm
- Een lijst met functionaliteiten
- Een datamodel
- Een flowchart
- Een stappenplan
- Een beschrijving van normale situaties en foutgevallen

AI kan helpen om een ontwerp te verbeteren, maar de student moet zelf kunnen uitleggen waarom het ontwerp logisch is.

4. Werken met AI als hulpmiddel

AI moet geen verboden hulpmiddel zijn, maar ook geen automatische oplossing.

Studenten moeten leren om AI professioneel te gebruiken.

Dat betekent:

- Een goede opdracht formuleren
- Context geven
- Voorbeelden geven
- Beperkingen benoemen
- AI vragen om uitleg
- AI vragen om alternatieven
- AI vragen om testgevallen
- AI-output controleren
- Onveilige of onduidelijke code herkennen
- Zelf verantwoordelijkheid nemen voor het eindresultaat

Een student moet niet leren om AI blind te vertrouwen, maar om AI kritisch en doelgericht te gebruiken.

5. Testen, reviewen en verbeteren

Wanneer code sneller geproduceerd wordt, wordt controle belangrijker.

Studenten moeten leren dat software niet klaar is omdat er code staat. Software is pas bruikbaar als deze getest, begrijpelijk, veilig en onderhoudbaar is.

Daarom moeten studenten leren werken met:

- Handmatige tests
- Testscenario's
- Randgevallen
- Foutmeldingen
- Code reviews
- Refactoring
- Security checks
- Performance checks
- Gebruiksvriendelijkheid

Een goede ontwikkelaar vraagt niet alleen:

"Werkt het?"

maar ook:

- Werkt het met verkeerde invoer?
- Werkt het als gegevens ontbreken?
- Werkt het voor een echte gebruiker?
- Is de code begrijpelijk?
- Is de oplossing veilig?
- Kan iemand anders dit later aanpassen?

Welke oefeningen moeten studenten maken?

De opleiding moet studenten niet alleen losse programmeeropdrachten geven, maar opdrachten waarin zij het hele ontwikkelproces oefenen.

Een moderne opdracht bestaat daarom niet alleen uit:

"Maak deze code."

maar uit:

"Analyseer het probleem, ontwerp een oplossing, bouw deze met of zonder AI, test het resultaat en leg je keuzes uit."

Oefening 1: Van vaag idee naar duidelijke opdracht

Studenten krijgen een vaag idee en moeten dit omzetten naar duidelijke requirements.

Voorbeeld

"Maak een app voor huiswerk."

Studenten werken uit:

- Wie gebruikt de app?
- Welke taken moet de gebruiker kunnen doen?
- Welke gegevens worden opgeslagen?
- Welke foutmeldingen zijn nodig?
- Wanneer is de app succesvol?

Doel

Studenten leren dat goede software begint met duidelijke taal.

Oefening 2: Prompt schrijven en verbeteren

Studenten schrijven eerst een slechte prompt en verbeteren deze daarna stap voor stap.

Voorbeeld

> "Maak een quiz."

wordt:

> "Maak een eenvoudige quiz-app in JavaScript met vijf vragen, vier antwoorden per vraag, één goed antwoord, feedback na elke vraag, een eindscore en duidelijke foutafhandeling als er geen antwoord is gekozen."

Doel

Studenten leren dat AI betere resultaten geeft wanneer de opdracht concreet, volledig en controleerbaar is.

Oefening 3: AI-code uitleggen

Studenten krijgen code die door AI is gegenereerd en moeten uitleggen wat de code doet.

Studenten beantwoorden vragen zoals:

- Welke variabelen worden gebruikt?
- Welke functies zijn er?
- Waar wordt invoer gecontroleerd?
- Waar kan een fout ontstaan?
- Wat zou je verbeteren?

Doel

Studenten leren dat zij eigenaar blijven van code, ook als AI heeft geholpen.

Oefening 4: Fouten zoeken in gegenereerde code

Studenten krijgen werkende code met verborgen problemen.

Bijvoorbeeld:

- Een formulier zonder validatie
- Een SQL-query zonder prepared statement
- Een wachtwoord dat als gewone tekst wordt opgeslagen
- Een functie die crasht bij lege invoer
- Een berekening die fout gaat bij nul of negatieve waarden

Doel

Studenten leren kritisch kijken naar code en ontdekken dat "de code draait" niet hetzelfde is als "de software is goed".

Oefening 5: Testgevallen ontwerpen

Studenten bouwen niet meteen nieuwe functionaliteit, maar schrijven eerst testscenario's.

Voorbeeld: reserveringssysteem

- Een lokaal reserveren op een vrije tijd
- Een lokaal reserveren dat al bezet is
- Een reservering maken zonder naam
- Een eindtijd invoeren die vóór de starttijd ligt
- Een reservering annuleren

Doel

Studenten leren vooraf nadenken over normaal gedrag, fout gedrag en randgevallen.

Oefening 6: Code review uitvoeren

Studenten beoordelen elkaars code of AI-gegenereerde code met een vaste checklist.

Checklist

- Is de code begrijpelijk?
- Zijn namen van variabelen en functies duidelijk?
- Wordt invoer gecontroleerd?
- Zijn foutmeldingen begrijpelijk?
- Is de code veilig genoeg?
- Is er dubbele code?
- Kan deze code later makkelijk worden aangepast?

Doel

Studenten leren kwaliteit beoordelen in plaats van alleen functionaliteit opleveren.

Oefening 7: Refactoren

Studenten krijgen rommelige maar werkende code en moeten deze verbeteren zonder het gedrag te veranderen.

Ze verbeteren bijvoorbeeld:

- Naamgeving
- Structuur
- Herhaling
- Functies
- Leesbaarheid
- Commentaar

Doel

Studenten leren dat goede software niet alleen werkt, maar ook begrijpelijk en onderhoudbaar is.

Oefening 8: Bouw dezelfde oplossing op twee manieren

Studenten bouwen eerst een kleine oplossing zelf en laten daarna AI een alternatief maken. Daarna vergelijken zij beide oplossingen.

Vergelijkingsvragen

- Welke oplossing is duidelijker?
- Welke oplossing is korter?
- Welke oplossing is veiliger?
- Welke oplossing is makkelijker aan te passen?
- Welke oplossing begrijp je zelf het beste?

Doel

Studenten leren dat er meerdere oplossingen mogelijk zijn en dat zij keuzes moeten kunnen onderbouwen.

Oefening 9: Mini-project met volledige ontwikkelcyclus

Studenten werken aan kleine projecten waarin alle stappen terugkomen.

Voorbeelden

- Een takenlijst
- Een quiz-app
- Een reserveringssysteem
- Een absentie-overzicht
- Een simpele webshop
- Een dashboard met gegevens
- Een login-systeem
- Een portfolio-website

Elke opdracht bevat:

- Probleembeschrijving
- Requirements
- Schets of ontwerp
- Code
- Gebruik van AI met verantwoording
- Testscenario's
- Code review
- Reflectie

Doel

Studenten leren softwareontwikkeling als proces, niet als losse code-oefening.

Oefening 10: Uitleggen aan een niet-technische gebruiker

Studenten moeten hun oplossing uitleggen aan iemand zonder technische kennis.

Ze leggen uit:

- Welk probleem wordt opgelost?
- Hoe werkt de oplossing globaal?
- Welke keuzes zijn gemaakt?

- Welke beperkingen zijn er nog?
- Wat zou een volgende verbetering zijn?

Doel

Studenten leren communiceren als ontwikkelaar. Dat is essentieel, omdat software altijd wordt gebouwd voor mensen.

Moeten studenten nog programmeertalen leren?

Ja. Studenten moeten nog steeds ervaring opdoen met programmeertalen.

AI maakt programmeertalen niet overbodig. AI maakt het juist belangrijker dat studenten code kunnen begrijpen, beoordelen en verbeteren.

Een student die geen programmeerbasis heeft, kan AI-code moeilijk controleren. Die student ziet dan niet of de oplossing onveilig, onlogisch, onvolledig of slecht onderhoudbaar is.

De vraag is daarom niet:

"Moeten studenten nog syntax leren?"

maar:

"Hoeveel syntax moeten studenten leren, en met welk doel?"

Programmeertalen leren met een ander doel

Vroeger was het doel vaak: syntax uit het hoofd leren en zelfstandig code produceren.

Nu wordt het doel breder:

- Code kunnen lezen
- Code kunnen begrijpen
- Code kunnen aanpassen
- Code kunnen debuggen
- Code kunnen testen
- Code kunnen beoordelen

- AI-code kunnen controleren
- Technische keuzes kunnen uitleggen

Studenten hoeven niet iedere functie of ieder commando uit het hoofd te kennen. Zij moeten wel de basisconcepten begrijpen en voldoende praktijkervaring hebben om te herkennen wat code doet.

Aanbevolen verhouding in de opleiding

Een moderne opleiding Software Development kan de aandacht ongeveer als volgt verdelen:

- **Basis programmeren:** studenten leren zelf code schrijven zonder AI, zodat zij fundament opbouwen.
- **Begrijpend programmeren:** studenten leren code lezen, uitleggen, aanpassen en debuggen.
- **AI-ondersteund ontwikkelen:** studenten leren AI gebruiken als hulpmiddel, maar blijven zelf verantwoordelijk.
- **Ontwerp en requirements:** studenten leren problemen analyseren en oplossingen specificeren.
- **Testen en reviewen:** studenten leren bepalen of software goed, veilig en bruikbaar is.
- **Projectmatig werken:** studenten leren samenwerken, versiebeheer gebruiken en keuzes verantwoorden.

Voor beginnende studenten is het belangrijk dat zij eerst voldoende handmatig programmeren. Zonder basiservaring wordt AI een black box.

Voor gevorderde studenten mag AI steeds meer worden ingezet, maar altijd met controle, uitleg, testen en reflectie.

Welke programmeertalen zijn belangrijk?

Studenten hoeven niet zoveel mogelijk talen te leren. Zij moeten vooral leren denken als ontwikkelaar.

Een goede basis kan bestaan uit:

- **HTML en CSS:** structuur en vormgeving van webpagina's begrijpen.
- **JavaScript:** interactie, logica in de browser en moderne webontwikkeling.

- **Python:** laagdrempelig leren programmeren, automatiseren, data verwerken en AI-concepten begrijpen.
- **PHP of een vergelijkbare backendtaal:** formulieren, sessies, databases en server-side logica begrijpen.
- **SQL:** gegevens opslaan, opvragen, filteren en structureren.
- **Git:** versiebeheer, samenwerken en professioneel werken.

Belangrijker dan de exacte taal is dat studenten de onderliggende concepten begrijpen:

- Invoer en uitvoer
- Data en datastructuren
- Controleflow
- Functies
- Objecten
- Databases
- API's
- Security
- Testing
- Onderhoudbaarheid

Wie deze concepten begrijpt, kan later makkelijker nieuwe talen en frameworks leren.

Een nieuwe richting

De opleiding Software Development moet studenten voorbereiden op een beroepspraktijk waarin AI normaal gereedschap wordt.

Dat vraagt om een verschuiving in de opleiding.

Niet van:

> Programmeren leren → AI gebruiken

Maar naar:

> Softwareontwikkeling leren → AI professioneel inzetten

De opleiding moet studenten leren dat softwareontwikkeling bestaat uit meerdere fasen:

- Probleem begrijpen
- Gebruiker begrijpen
- Requirements formuleren
- Oplossing ontwerpen
- Code schrijven of genereren
- Code begrijpen
- Code testen
- Code verbeteren
- Code veilig maken
- Resultaat uitleggen
- Resultaat onderhouden

AI kan in bijna elke fase helpen, maar AI vervangt die fasen niet.

Een student moet daarom leren wanneer AI nuttig is en wanneer eigen denkwerk noodzakelijk is.

Wat moet een student uiteindelijk kunnen?

Een student die klaar is voor de moderne praktijk kan:

- Een probleem analyseren
- Gebruikersbehoeften beschrijven
- Requirements opstellen
- Een eenvoudige technische oplossing ontwerpen
- Zelf basiscode schrijven
- AI gebruiken om sneller te ontwikkelen
- AI-output controleren en verbeteren
- Foutmeldingen begrijpen
- Code debuggen
- Veilige keuzes maken

- Testscenario's opstellen
- Code reviewen
- Samenwerken met Git
- Een oplossing presenteren aan een gebruiker
- Reflecteren op kwaliteit, veiligheid en onderhoudbaarheid

Daarmee wordt de student niet alleen iemand die code kan maken, maar iemand die softwareproblemen kan oplossen.

Wat betekent dit voor beoordeling?

Ook de beoordeling van studenten moet veranderen.

Als AI code kan genereren, is het onvoldoende om alleen het eindproduct te beoordelen. De opleiding moet ook beoordelen hoe de student tot de oplossing is gekomen.

Beoordeling zou daarom moeten kijken naar:

- De probleemanalyse
- De requirements
- Het ontwerp
- De gemaakte keuzes
- Het begrip van de code
- De kwaliteit van de tests
- De manier waarop AI is gebruikt
- De controle van AI-output
- De veiligheid van de oplossing
- De onderhoudbaarheid van de code
- De reflectie van de student

Een student mag AI gebruiken, maar moet kunnen uitleggen:

- Wat heb je gevraagd?
- Waarom heb je dat gevraagd?
- Wat kwam eruit?

- Wat heb je gecontroleerd?
- Wat heb je aangepast?
- Hoe weet je dat het werkt?
- Wat begrijp je zelf van de oplossing?

Zo wordt AI-gebruik geen manier om denkwerk te vermijden, maar een manier om denkwerk zichtbaar te maken.

Samenvatting

Decennialang draaide softwareontwikkeling vooral om het schrijven van code.

Een ontwikkelaar had een idee, vertaalde dat idee naar een programmeertaal en bouwde daarmee software. Hoe beter je de syntax beheerste, hoe meer je kon maken.

Maar de komst van generatieve AI verandert die werkelijkheid.

Voor het eerst in de geschiedenis kan een computer een groot deel van de programmeercode zelf schrijven. Wat vroeger veel kennis van syntax vereiste, kan vandaag vaak worden gegenereerd op basis van een beschrijving in gewone taal.

Daardoor verschuift de belangrijkste vaardigheid van de ontwikkelaar.

De vraag is niet langer alleen:

> **"Kan ik deze code schrijven?"**

De vraag wordt steeds vaker:

> **"Kan ik duidelijk uitleggen wat er gebouwd moet worden?"**

Want AI kan code genereren, maar begrijpt niet vanzelf welk probleem opgelost moet worden, welke regels gelden, wat gebruikers verwachten of wanneer een oplossing werkelijk goed is.

De waarde verschuift daarom van syntax naar duidelijkheid.

Niet:

> **Idee → Code**

maar steeds vaker:

> **Idee → Duidelijkheid → AI → Code**

Wie software wil ontwikkelen in het tijdperk van AI, moet niet alleen leren programmeren. Die moet vooral leren begrijpen, analyseren, ontwerpen, communiceren en beoordelen.

De rol van de ontwikkelaar verschuift. Minder tijd gaat naar het schrijven van syntax en meer tijd naar het begrijpen van problemen, ontwerpen van oplossingen, beoordelen van resultaten en samenwerken met AI.

De ontwikkelaar van morgen:

- **begrijpt** bestaande code;
- kan **problemen analyseren** en requirements verduidelijken;
- **denkt na** voordat er gebouwd wordt;
- gebruikt **AI** als **hulpmiddel**, niet als vervanging van kritisch denken;
- **test**, **controleert** en **verbetert** het resultaat;
- neemt **verantwoordelijkheid** voor de **kwaliteit** van de software.

Wie alleen leert programmeren, leert een techniek.

Wie leert problemen oplossen, ontwerpen, communiceren en kritisch denken, leert software ontwikkelen.

En juist die vaardigheden worden in het tijdperk van AI steeds waardevoller.

> "De programmeertaal van gisteren was **syntax**. De programmeertaal van morgen is **duidelijkheid**."

--

(Max Bisschop)

Revision #6
Created 2026-07-04 18:55:06 UTC by Max
Updated 2026-07-06 20:19:52 UTC by Max