Odisee
DE CO-HOGESCHOOL

Onderwijsgroep Professionele Opleidingen

Toegepaste Informatica

CityMesh Drones - Opdracht 1

Oumou Bah
Noa Gölger
Juliaan Petitjean
Walid Ahyaten
Younes Bellaga
Oumaima Ahmiti

Tweede Bachelor Toegepaste Informatica

Analysis, Design & Testing: Server Applications
Hans Vandenbogaerde

Academiejaar 2024-2025

Inhoud

Inleiding

In deze opdracht werkten we op basis van eerder opgestelde user stories verder aan de analyse, het ontwerp en de testfase van een serverapplicatie. We vertrokken vanuit een aantal functionele vereisten en vertaalden die naar use case diagrammen, die we vervolgens verder uitwerkten via activity diagrammen en use case beschrijvingen. Op basis daarvan bouwden we een klassendiagram op, vertrekkend van kandidaat-objecten die we selecteerden en verfijnden. We onderzochten ook het gedrag van de belangrijkste objecten via toestandsdiagrammen en gebruikten een sequentiediagram om de interacties binnen een use case visueel te maken. Tot slot hebben we acceptatietesten opgesteld die linken aan de toestand van bepaalde objecten. Al deze onderdelen vormden samen een gestructureerde aanpak om de basis te leggen voor een stabiele en duidelijke serverapplicatie.

1. Deel 1

1.1. User Stories

US001: Als piloot kan ik een lijst met beschikbare drones opzoeken, zodat ik inzicht krijg in welke drones klaar zijn voor gebruik.

US002: Als piloot kan ik beschikbare startplaatsen zien en reserveren, zodat ik mijn drone kan lanceren vanaf een vrije en veilige locatie.

US003: Als dronepiloot kan ik tijdens mijn vlucht alle mogelijke checkpoints in kaart brengen zodat ik precies kan zien waar mensen zich bevinden

US004: Als mechanieker kan ik de status van een gerepareerde drone resetten naar 'klaar voor gebruik', zodat piloten weten dat deze weer operationeel is.

US005: Als mechanieker kan ik een drone markeren voor verzending naar het depot, wanneer reparatie ter plaatse niet mogelijk is, zodat deze drone toch gerepareerd kan worden.

1.1.1. Invest

In dit onderdeel gebruiken we het INVEST-principe om onze user stories te beoordelen op kwaliteit. We hebben per user story nagegaan of die onafhankelijk, onderhandelbaar, waardevol, inschatbaar, klein en testbaar is

US001

- **I (Independent)** - De user story is onafhankelijk en kan los van andere functionaliteiten ontwikkeld worden.
- **N (Negotiable)** - De story is flexibel en kan bijgestuurd worden op basis van input.
- **V (Valuable)** - Geeft waarde aan de piloot door inzicht in beschikbare drones te bieden.
- **E (Estimable)** - Goed in te schatten, omdat het een simpele lijstweergave betreft.
- **S (Small)** - Beperkt in scope en kan in één sprint ontwikkeld worden.
- **T (Testable)** - Testbaar door te verifiëren of de lijst correct wordt weergegeven.

US002

- **I (Independent)** - Kan los van andere functionaliteiten ontwikkeld worden.
- **N (Negotiable)** - De details kunnen aangepast worden (bijv. tijdslimiet op reservatie).
- **V (Valuable)** - Piloten kunnen efficiënter plannen, wat waarde toevoegt.
- **E (Estimable)** - De implementatie is duidelijk te schatten.
- **S (Small)** - Bevat een duidelijke, beperkte functionaliteit.
- **T (Testable)** - Testbaar door te controleren of reservaties correct worden verwerkt.

US003

- **I (Independent)** - Mogelijk afhankelijk van een kaart- of GPS-systeem.
- **N (Negotiable)** - De story is redelijk breed; wat wordt exact bedoeld met "alle mogelijke checkpoints"?
- **V (Valuable)** - Geeft piloten inzicht in de omgeving en mogelijke risico's.
- **E (Estimable)** - Moeilijk in te schatten zonder meer details over hoe de checkpoints worden bepaald.
- **S (Small)** - Kan een grote en complexe functionaliteit zijn.
- **T (Testable)** - Onvoldoende gedefinieerd om te testen; wat betekent "precies zien waar mensen zich bevinden"?

Herschreven: "Als dronepiloot kan ik tijdens mijn vlucht een kaart bekijken met vooraf ingestelde checkpoints en realtime informatie over gedetecteerde personen, zodat ik precies weet waar belangrijke controlepunten en mogelijke menselijke activiteit zich bevinden."

US004

- **I (Independent)** - Kan zelfstandig worden ontwikkeld zonder afhankelijkheden.
- **N (Negotiable)** - Kan aangepast worden (bijv. extra statusopties toevoegen).
- **V (Valuable)** - Geeft piloten duidelijke info over welke drones klaar zijn.
- **E (Estimable)** - Goed in te schatten; status wijzigen is een eenvoudige functionaliteit.
- **S (Small)** - Beperkte scope en eenvoudig te implementeren.
- **T (Testable)** - Testbaar door te controleren of de status correct wordt bijgewerkt.

US005

- **I (Independent)** - Losstaand proces, geen directe afhankelijkheden.
- **N (Negotiable)** - Kan aangepast worden (bijv. extra statussen zoals 'in afwachting van transport').
- **V (Valuable)** - Zorgt ervoor dat defecte drones correct worden verwerkt.
- **E (Estimable)** - Eenvoudige statuswijziging, goed in te schatten.
- **S (Small)** - Beperkte scope en eenvoudig te implementeren.
- **T (Testable)** - Testbaar door te controleren of een drone correct wordt gemarkeerd en doorgestuurd.

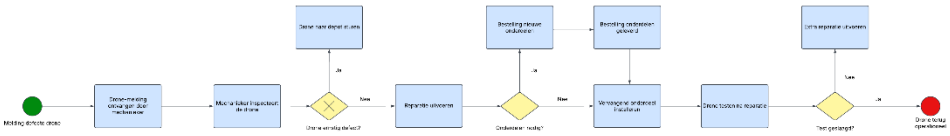
1.2. BPMN

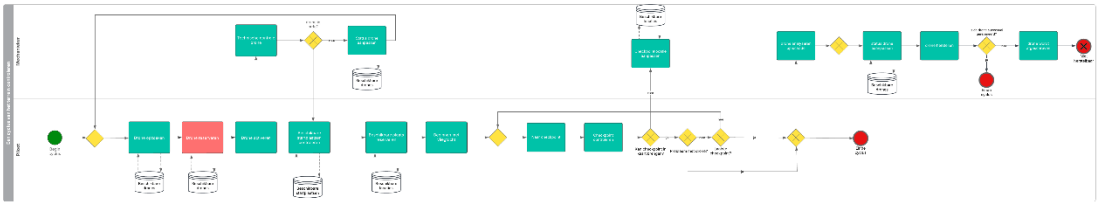
Melding defecte drone

Dit BPMN-diagram stelt het proces voor van hoe een melding over een defecte drone verwerkt wordt. Het toont de interactie tussen de piloot, de mechanieker en het systeem

Een cyclus van het terrein controleren

In onderstaand BPMN-diagram tonen we hoe het proces van het controleren van een terrein verloopt met behulp van een drone. Elke stap van de piloot tot de verwerking van gegevens wordt visueel voorgesteld.





1.3. IPO-Requirements

Hieronder geven we een overzicht van de input, processen en output (IPO) voor de belangrijkste processen uit ons systeem. Dit helpt om duidelijk te zien welke gegevens binnenkomen, wat ermee gebeurt en wat het resultaat is

REQUIREMENTS PER ACTIVITEIT VAN HET PROCES	DRONE-MELDING ONTVANGEN DOOR MECHANIEKER	MECHANIEKER INSPECTEERT DE DRONE	DRONE NAAR DEPOT STUREN	REPARATIE UITVOEREN	BESTELLING NIEUWE ONDERDELEN	BESTELLING ONDERDELEN GELEVERD	VERVANGEND ONDERDEEL INSTALLEREN	DRONE TESTEN NA REPARATIE	EXTRA REPARATIE UITVOEREN
REQ0001 - Het systeem moet defectmeldingen van drones registreren met een uniek meldingsnummer, inclusief drone-ID, defectbeschrijving en tijd/datum van de melding.	I,P								
REQ0002 - Mechaniekers moeten defectmeldingen kunnen bekijken en aanvullen met extra details, inclusief foto's en opmerkingen.		I,P,O							
REQ0003 - Mechaniekers moeten de mogelijkheid hebben om de staat van de drone te inspecteren en de resultaten te registreren.		I,P,O							
REQ0004 - Het systeem moet de verzendstatus van drones kunnen bijhouden via een trackingmechanisme.			P,O						
REQ0005 - Indien een drone als ernstig defect wordt beoordeeld, moet het systeem automatisch een transportaanvraag genereren en registreren.		I,P	P,O						
REQ0006 - Mechaniekers moeten voorgestelde reparaties kunnen goedkeuren of afwijzen voordat deze worden uitgevoerd.		I,P,O							
REQ0007 - Het systeem moet controleren of de benodigde onderdelen voor reparatie beschikbaar zijn en waarschuwen als deze ontbreken.				I,P,O					
REQ0008 - Het systeem moet een bestellogboek bijhouden met verwachte leverdata en meldingen genereren wanneer bestelde onderdelen aankomen.					I,P,O	P,O			
REQ0009 - Installaties en reparaties moeten worden bijgehouden met datum, tijd en verantwoordelijke medewerker.				P,O			P,O		
REQ0010 - Het systeem moet testscenario's genereren en de resultaten registreren per drone na reparatie.								P,O	
REQ0011 - Als een test faalt, moet het systeem automatisch een foutmelding genereren en een extra reparatieoptie voorstellen.								I,P,O	
REQ0012 - Mechaniekers moeten toegang hebben tot testresultaten en herhalingsreparaties kunnen uitvoeren op basis van mislukte tests.								I,P	P,O
REQ0013 - Het systeem moet herhalingstesten genereren en registreren na een extra reparatie.								P,O	I,P,O

REQUIREMENTS PER ACTIVITEIT VAN HET PROCES	DRONE OPZOEKEN	DRONE RESERVEREN	DRONE ACTIVEREN	TECHNISCHE CONTROLE DRONE	STATUS DRONE AANPASSEN	BESCHIKBARE STARTPLAATSEN CONTROLEREN	BESCHIKBARE PLAATS RESERVEREN	BEGINNEN MET VLIEGTOCHT	NAAR CHECKPOINT	CHECKPOINT CONTROLEREN	CHECKPOINT VERANDEREN	DRONE ANALYSEREN OP SCHADE	STATUS DRONE AANPASSEN	DRONE HERSTELLEN
REQ0014 - Het systeem moet piloten in staat stellen om beschikbare drones op te zoeken en hun status te bekijken.	O													
REQ0015 - Piloten moeten drones kunnen reserveren via het systeem en een bevestiging ontvangen.		P												
REQ0016 - Het systeem moet piloten de mogelijkheid bieden om een gereserveerde drone te activeren voordat ze de vlucht starten.			P											
REQ0017 - Mechaniekers moeten een technische controle van een drone kunnen uitvoeren en de resultaten registreren.				P	O									
REQ0018 - De status van een drone moet op elk moment door mechaniekers of piloten kunnen worden aangepast.					P								O	
REQ0019 - Het systeem moet een overzicht tonen van beschikbare startplaatsen voor dronevluchten.						P	O							
REQ0020 - Piloten moeten een startplaats kunnen reserveren en een melding ontvangen bij succesvolle reservering.		P												
REQ0021 - Een piloot moet de mogelijkheid hebben om een vlucht te starten zodra de drone is geactiveerd en een startplaats is gereserveerd.								P						
REQ0022 - Tijdens de vlucht moet het systeem de drone in staat stellen om naar vooraf bepaalde checkpoints te navigeren.								I	P					
REQ0023 - Checkpoints moeten gecontroleerd worden op basis van sensorinformatie en registraties van de drone.										P				
REQ0024 - Piloten moeten de mogelijkheid hebben om de locatie van een checkpoint aan te passen indien nodig.									I		P			
REQ0025 - Na de vlucht moet de drone geanalyseerd worden op mogelijke schade.												P	O	
REQ0026 - Als schade wordt vastgesteld, moet de status van de drone worden aangepast naar 'Defect' of 'Onderhoud nodig'.												I	P	
REQ0027 - Mechaniekers moeten defecte drones kunnen markeren voor reparatie of verzending naar een depot.														P

1.4. Woordenboek

Hier geven we een overzicht van alle belangrijke begrippen binnen ons systeem. We vertrokken vanuit de user stories en requirements en filterden zelfstandige naamwoorden die relevant zijn. Die hebben we verder uitgewerkt met een korte beschrijving, zodat duidelijk is wat elk concept inhoudt.

zelfstandige naamwoorden		gefiltert op relevantie	gefiltert op onafhankelijkheid	gefiltert op dubbels
systeem	7	~~systeem~~	~~meldingen~~	drones
meldingen	2	meldingen	drones	melding
drones	1	drones	melding	mechanieker
meldingsnumm	1	~~meldingsnummer~~	mechanieker	reparaties
melding	3	melding	reparaties	gebruiker
~~gegevens~~	1	mechanieker	gebruiker	verslag
~~drone-ID~~	1	~~staat~~	verslag	
~~beschrijving~~	1	~~resultaten~~		
~~defect~~	2	reparaties		
~~datum/tijd~~	2	gebruiker		
mechanieker	6	verslag		
~~mogelijkheid~~	3			
staat	1			
resultaten	2			
~~verzendstatus~~	1			
~~transportaanvra~~	1			
verslag	1			
~~details~~	1			
reparaties	2			
~~onderdelen~~	3			
~~voorraad~~	1			
~~bestellogboek~~	1			
~~leverdata~~	1			
~~meldingen~~	1			
~~installaties~~	1			
~~datum~~	1			
~~tijd~~	1			
~~medewerker~~	1			
~~testscenario's~~	1			
~~testresultaten~~	2			
~~test~~	3			
~~aanvullende rep~~	1			
~~herhalingstest~~	1			
gebruiker	1			

Zelfstandige naamwoorden die gefilterd zijn op relevantie, onafhankelijkheid en dubbels:

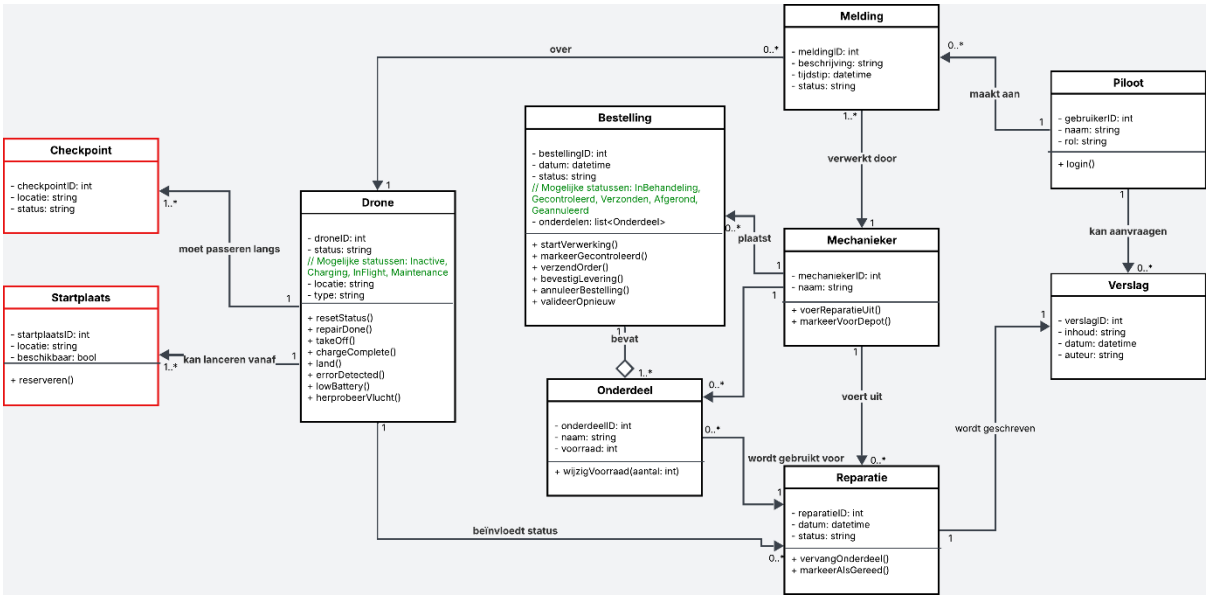
1) Drones
2) Melding
3) Mechanieker
4) Reparaties
5) Gebruiker
6) Verslag
7) Bestelling
8) Onderdeel
9) Checkpoint
10) Startplaats

Uitgeschreven

Drone: Heeft een uniek ID, status, type en locatie. Kan zijn status resetten.
Melding: Wordt aangemaakt door een gebruiker en bevat details zoals een ID, beschrijving, tijdstip en status.
Mechanieker: Verwerkt meldingen, kan reparaties uitvoeren en drones markeren voor verzending naar het depot.
Reparatie: Heeft een ID, datum en status. Kan onderdelen vervangen en beïnvloedt de drone-status.
Gebruiker: Kan meldingen aanmaken en verslagen raadplegen.
Verslag: Bevat informatie over reparaties, zoals inhoud, datum en auteur.
Startplaats: Bevat een ID, locatie en beschikbaarheidsstatus. Kan worden gereserveerd.
Checkpoint: Bevat een ID, locatie en status. Drones moeten deze passeren.
Onderdeel: Mechaniekers moeten weten welke onderdelen beschikbaar zijn. Er moet een voorraadbeheer zijn om te checken of een onderdeel beschikbaar is.
Bestelling: Mechaniekers plaatsen bestellingen voor onderdelen. Een bestelling heeft een status.

1.5. Domeinklassendiagram

Het klassendiagram hieronder toont de belangrijkste entiteiten van ons systeem en hoe ze met elkaar in verband staan. Elke klasse bevat attributen en, indien nodig, methodes gebaseerd op het gedrag van het object.



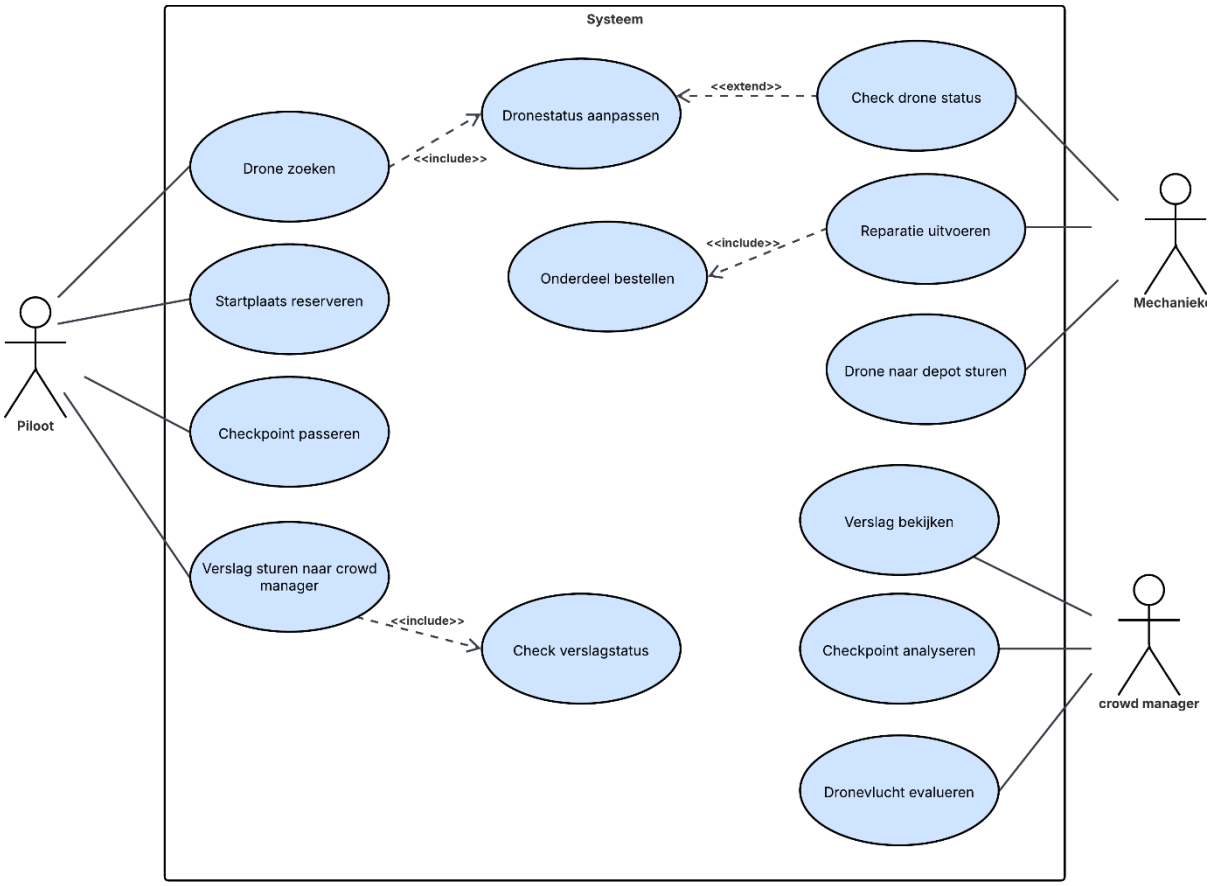
2. Deel 2

2.1. De actoren in onze case

- **Piloot**: is een Piloot die drones bestuurt en verantwoordelijk is voor het uitvoeren van dronevluchten.
- **Mechanieker**: is verantwoordelijk voor het onderhoud en de reparatie van drones.
- **Crowd manager**: monitort en analyseert de data afkomstig van dronevluchten.

2.2. Use case diagram

Dit Use case diagram toont de interacties tussen de actoren piloot, mechanieker, crowd manager en het systeem. Elke use case stelt een actie of functionaliteit voor die de gebruiker kan uitvoeren.



2.3. Use case beschrijving

In dit onderdeel werkten we één specifieke use case (dat het ons het relevantste leek) volledig uit in een gestructureerde beschrijving. Deze beschrijving vormt de basis voor het activity diagram, de UI, en latere testscenario's.

USE CASE Titel	**Verslag sturen naar Crowd manager**
Versie	1.0
Primaire actor	Piloot
Preconditie	De piloot heeft een vlucht uitgevoerd. De dronevlucht heeft automatisch vluchtgegevens geregistreerd (zoals checkpoints, tijd, locatie, enz.). Deze gegevens zijn opvraagbaar via het systeem.
Eindconditie bij succes	Het verslag wordt correct naar de Crowd manager gestuurd. De Crowd Manager heeft toegang tot de inhoud en gekoppelde vluchtdata.
Eindconditie bij falen	Het verslag wordt niet verstuurd. De piloot krijgt een foutmelding en kan het later opnieuw proberen. Als vluchtgegevens niet beschikbaar zijn, kan het verslag niet opgesteld worden.

Hoofdscenario	**Stap**	**Actie**	
	1	De piloot logt in op het systeem.	
	2	De piloot navigeert naar de sectie “Verslag indienen”.	
	3	Het systeem toont een lijst van recente uitgevoerde vluchten.	
	4	De piloot selecteert de juiste vlucht en ziet bijhorende vluchtgegevens.	
	5	De piloot vult de vereiste informatie in (drone-ID, vluchtgegevens, opmerkingen).	
	6	Het systeem controleert of alle velden correct zijn ingevuld.	
	7	De piloot klikt op "Verzenden"	
	8	Het systeem slaat het verslag op en stuurt het naar de Crowd manager.	
Uitbreidingen			
	3.1	Geen vluchten beschikbaar voor de piloot	
		3.1.1	Het systeem toont een melding: “Geen voltooide vluchten beschikbaar.”
	6.1	Niet alle velden werden correct ingevuld	
		6.1.1	Het systeem toont een foutmelding.
	8.1	Verbinding met het systeem is verbroken	

		8.1.1	Het systeem toont een foutmelding en de piloot kan het later opnieuw proberen.
Speciale Vereisten			
	Het systeem moet een bevestiging tonen als het verslag succesvol is verzonden.		

2.4. Vermelding van processen, requirements & user stories

1. Processen die aan bod komen

Een cyclus van het terrein controleren → De piloot voert een vlucht uit en verzamelt gegevens, wat een voorwaarde is voor het verzenden van een verslag.

Een defecte drone herstellen → Als een drone defect raakt tijdens een vlucht, kan dit invloed hebben op het verslag dat naar de Crowd Manager wordt gestuurd.

2. Requirements die aan bod komen

REQ0014 - Het systeem moet piloten in staat stellen om beschikbare drones op te zoeken en hun status te bekijken.

REQ0022 - Tijdens de vlucht moet het systeem de drone in staat stellen om naar vooraf bepaalde checkpoint te navigeren.

REQ0023 - Checkpoint moeten gecontroleerd worden op basis van sensorinformatie en registraties van de drone.

REQ0024 - Piloten moeten de mogelijkheid hebben om de locatie van een checkpoint aan te passen indien nodig.

REQ0025 - Na de vlucht moet de drone geanalyseerd worden op mogelijke schade.

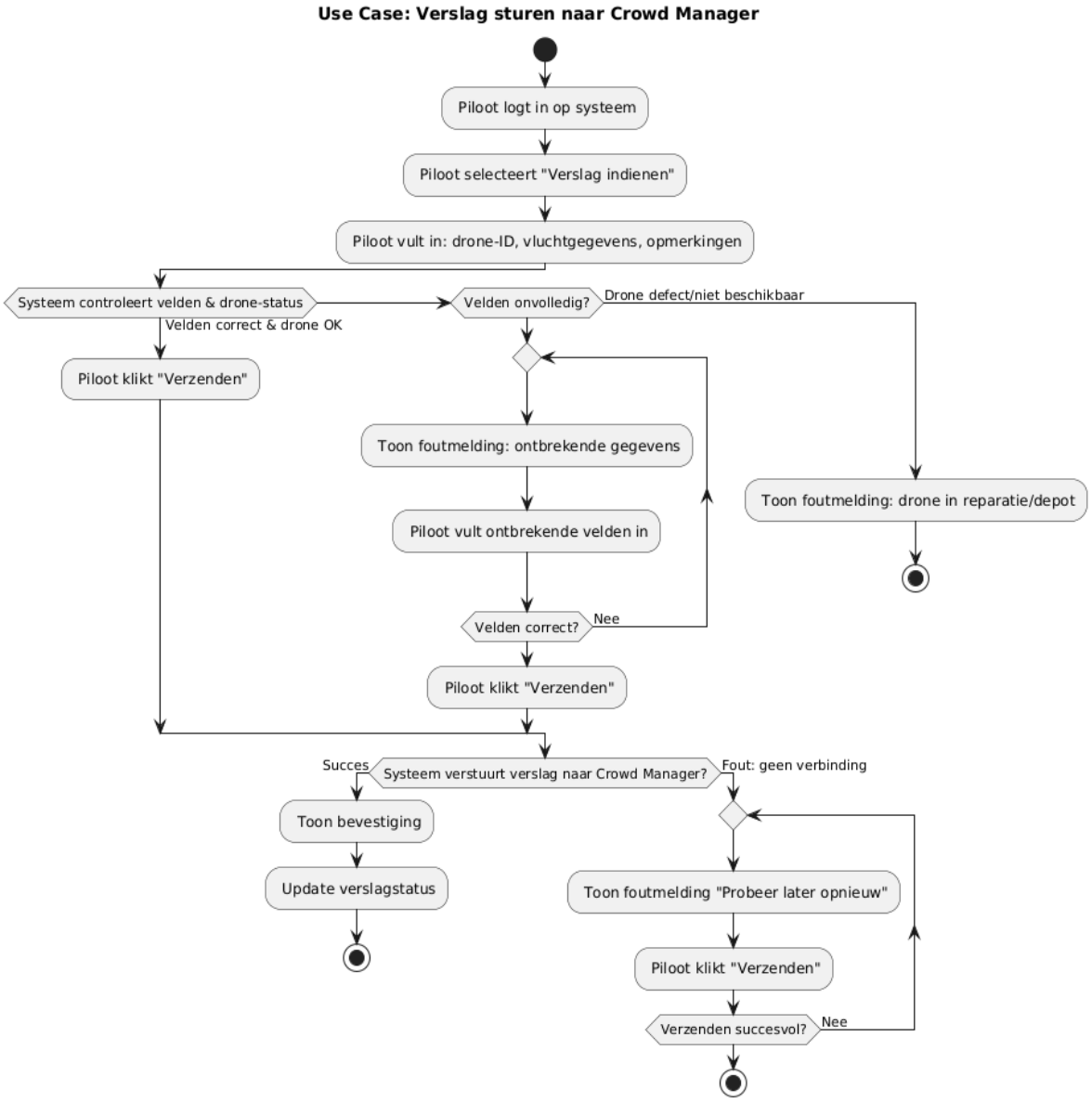
REQ0026 - Als schade wordt vastgesteld, moet de status van de drone worden aangepast naar ‘Defect’ of ‘Onderhoud nodig’.

3. User Stories die aan bod komen

- **US003**: Als dronepiloot kan ik tijdens mijn vlucht alle mogelijke checkpoints in kaart brengen zodat ik precies kan zien waar mensen zich bevinden.
- **US004**: Als mechanieker kan ik de status van een gerepareerde drone resetten naar ‘klaar voor gebruik’, zodat piloten weten dat deze weer operationeel is.

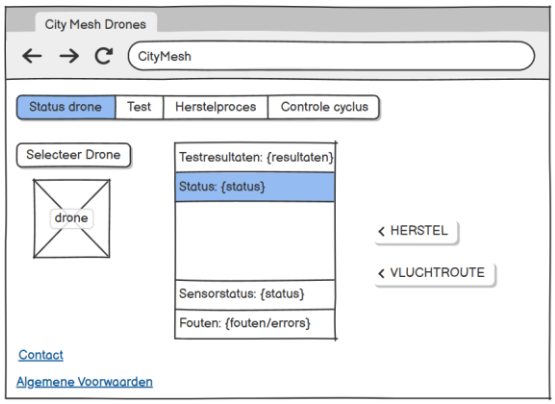
2.5. Activity diagram

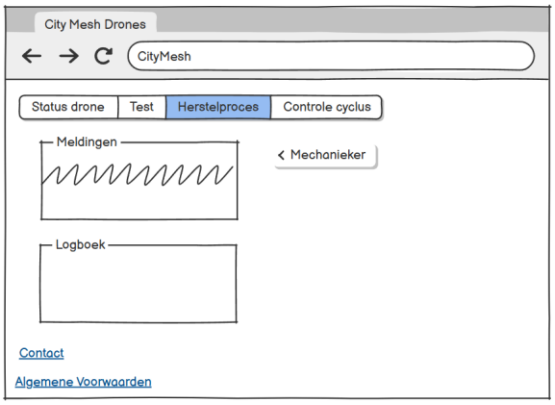
Het activity diagram geeft weer hoe de acties van de piloot verlopen bij het versturen van een verslag. De flow wordt stap voor stap voorgesteld, met mogelijke afwijkingen.

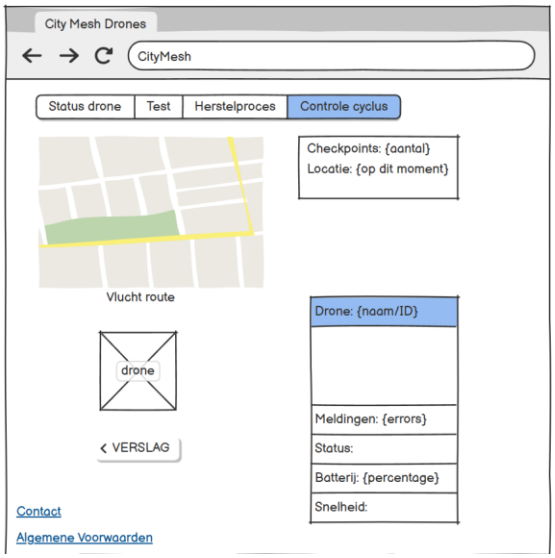


2.6. UI protoype

Hier tonen we een eenvoudig UI-prototype dat weergeeft hoe de piloot een verslag kan indienen. Dit helpt om de use case visueel voor te stellen vanuit het perspectief van de gebruiker.







2.7. Use Case Testing

Hieronder tonen we hoe we een use case omzetten in concrete testscenario's. We vertrekken vanuit een realistische situatie die beschreven staat in de use case en vertalen die naar een testbare reeks stappen. Dit helpt om na te gaan of het systeem zich gedraagt zoals verwacht in praktijkomstandigheden.

Elke test beschrijft een actie en een verwachte uitkomst. Zo kunnen we het gedrag van het systeem stap voor stap controleren.

ID	**Testactie**	**ID**	**Verwacht resultaat**
1	Piloot logt in op het systeem	4	Systeem accepteert de invoer en toont een verzenden knop
2	Piloot navigeert naar de sectie "Verslag indienen"	6	Het systeem slaat het verslag op en stuurt het naar de Crowd manager.
3	Piloot vult de vereiste informatie in	7	De Crowd manager ontvangt een melding en kan het verslag bekijken.
5	Piloot klikt op "verzenden"		

Alternatieve scenario's:

ID	**Testactie**	**ID**	**Verwacht resultaat**
1	Piloot logt in op het systeem	4.1	Het systeem toont een foutmelding en vraagt om ontbrekende informatie
3	Piloot navigeert naar de sectie "Verslag indienen"		
3	Piloot laat verplichte velden leeg		

ID	Testactie	ID	Verwacht resultaat
1	Piloot logt in op het systeem	4	Systeem accepteert de invoer en toont een verzenden knop
2	Piloot navigeert naar de sectie “Verslag indienen”	6.1	Het systeem toont een foutmelding en de piloot kan het later opnieuw proberen.
3	Piloot vult de vereiste informatie in		
5	Piloot klikt op “verzenden”		

3. Deel 3

3.1. Acceptatietesten

US004: *Als Mechanieker kan ik de status van een gerepareerde drone resetten naar 'klaar voor gebruik', zodat piloten weten dat deze weer operationeel is.*

- De drone verandert van toestand: "Defect" → "In Reparatie" → "Gerepareerd" → "Klaar voor gebruik".
- Dit toont niet-triviaal gedrag, want er zijn meerdere toestanden waarin de drone zich kan bevinden.

ID	Voorwaarde	Actie	Verwachte uitkomst
004.1	Drone staat op "Defect"	Mechanieker probeert status te wijzigen naar "Klaar voor gebruik"	Wijziging **niet toegestaan**, foutmelding "Drone moet eerst gerepareerd worden"
004.2	Drone staat op "In Reparatie"	Mechanieker probeert status te wijzigen naar "Klaar voor gebruik"	Wijziging **niet toegestaan**, foutmelding "Reparatie moet eerst voltooid worden"
004.3	Drone staat op "Gerepareerd"	Mechanieker zet status op "Klaar voor gebruik"	Wijziging **geslaagd**, drone is nu **operationeel**
004.4	Drone staat al op "Klaar voor gebruik"	Mechanieker probeert opnieuw te resetten	Geen statuswijziging, systeem toont "Drone is al klaar voor gebruik"

***Scenario*: Drone wordt succesvol gerepareerd**

Given een drone met status "Defect"

When de mechanieker de drone inspecteert en repareert

Then moet de status van de drone veranderen naar "Gerepareerd"

And wanneer de mechanieker de status reset

And moet de drone "Klaar voor gebruik" zijn

***Scenario*: Reparatie mislukt door ontbrekende onderdelen**

Given een drone met status "Defect"

And er zijn geen vervangende onderdelen beschikbaar

When de mechanieker probeert de reparatie te starten maar het systeem detecteert dat onderdelen ontbreken

Then moet het systeem een foutmelding geven: "Reparatie niet mogelijk"

And de status van de drone blijft "Defect"

***Scenario*: Drone moet naar depot worden gestuurd**

Given een drone met status "Defect"

When de mechanieker beoordeelt dat reparatie ter plaatse niet mogelijk is

And de mechanieker de drone markeert voor verzending naar het depot

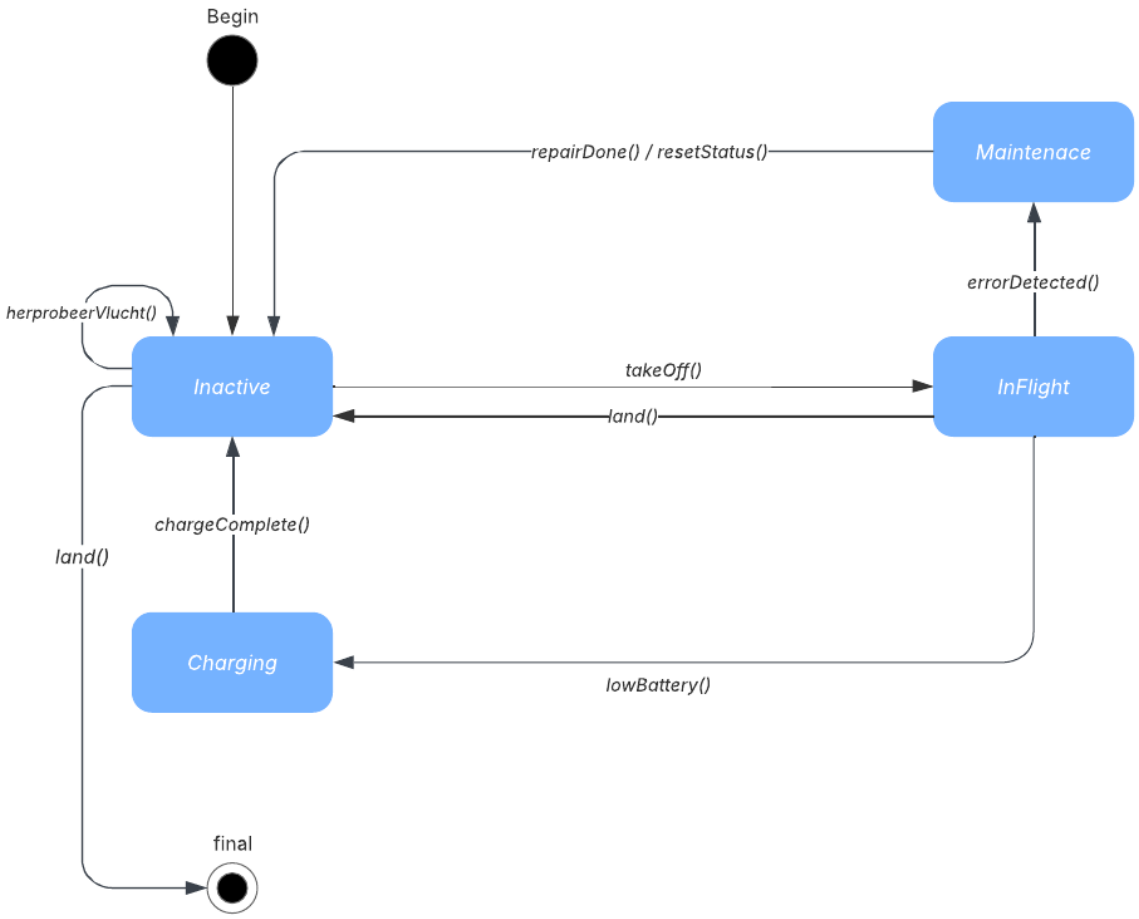
Then moet de status van de drone veranderd worden naar "Naar Depot Verzonden"

→ Referentie toestand van drone in het **ROOD** aangeduid

3.2. Toestandsdiagram

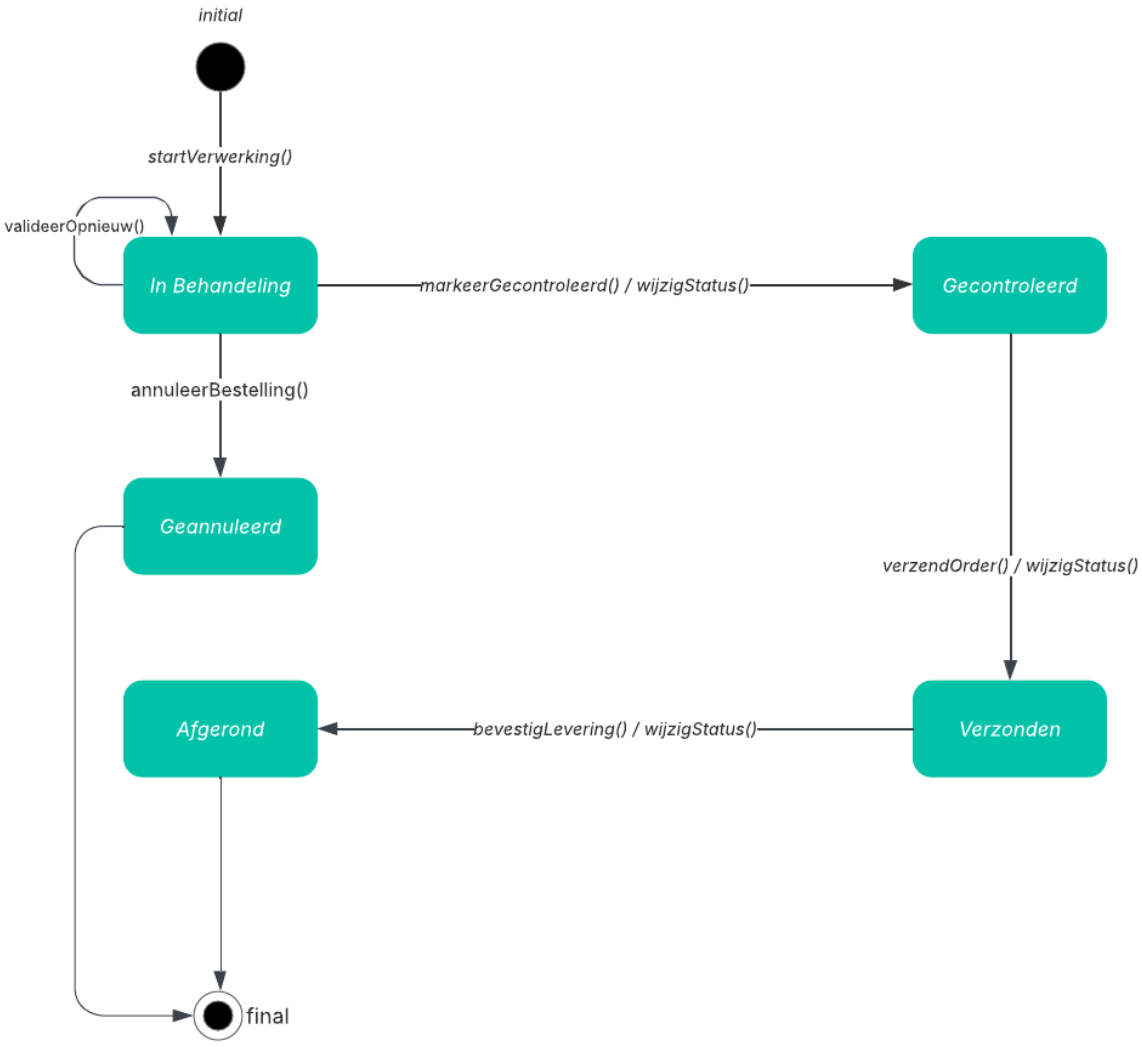
3.2.1. Drone

Dit toestandsdiagram toont de verschillende statussen die een drone kan aannemen, afhankelijk van acties in het systeem (zoals opladen, in vlucht, error, enz.).



3.2.2. Bestelling

In dit diagram zien we de statusveranderingen van een bestelling, vanaf aanmaak tot levering of annulatie.



3.3. Gherkin-acceptatietestscenario's

Hieronder staan enkele acceptatietesten uitgewerkt op basis van de user stories. Elke test vertrekt vanuit een bepaalde beginsituatie en controleert of het systeem correct reageert op een actie.

3.3.1. Drone

***Scenario:* Succesvol opstijgen vanuit inactieve staat**

Given de drone is in status "Inactive"

When de piloot een startcommand geeft

Then moet de drone-status veranderen naar "InFlight"

***Scenario:* Drone detecteert lage batterij tijdens vlucht**

Given de drone is in status "InFlight"

When het systeem "lowBattery()" activeert

Then moet de drone-status veranderen naar "Charging"

And de motoren moeten uitgeschakeld worden

***Scenario:* Drone volledig opgeladen**

Given de drone is in status "Charging"

When de batterij 100% bereikt

Then moet de drone-status veranderen naar "Inactive"

***Scenario:* Kritieke fout tijdens vlucht**

Given de drone is in status "InFlight"

When een sensor "errorDetected()" rapporteert

Then moet de drone-status veranderen naar "Maintenance"

And er moet een waarschuwing naar de mechanieker gestuurd worden

Scenario: ***Drone reset na reparatie***

Given de drone is in status "Maintenance"

When de technicus "repairDone()" uitvoert

And het systeem "resetStatus()" aanroept

Then moet de drone-status veranderen naar "Inactive"

And alle foutlogs moeten gewist worden

Scenario: ***Drone landt en schakelt uit***

Given de drone is in status "Inactive"

When het "final" commando wordt gegeven

Then moeten alle dronesystemen afgesloten worden

And de status moet gemarkeerd worden als "Final"

Scenario: ***Voorkom opstijgen tijdens opladen***

Given de drone is in status "Charging"

When het commando "takeOff()" ontvangen wordt

Then moet het systeem het verzoek afwijzen

And "Kan niet opstijgen tijdens opladen" loggen

3.3.2. Bestelling

Scenario: ***Nieuwe order start verwerking***

Given het systeem is in status "Initial"

When "startVerwerking()" wordt uitgevoerd

Then moet de orderstatus veranderen naar "In Behandeling"

And er moet een tijdsstempel worden geregistreerd

Scenario: ***Order succesvol gevalideerd***

Given de order is in status "In Behandeling"

When "valideerControle()" wordt uitgevoerd

Then moet de order klaar zijn voor controle

And alle vereiste velden moeten correct zijn ingevuld

Scenario: ***Klant annuleert order***

Given de order is in status "In Behandeling"

When "annuleerBestelling()" wordt uitgevoerd

Then moet de orderstatus veranderen naar "Geannuleerd"

And er moet een annuleringsbericht worden verstuurd

Scenario: ***Order gecontroleerd en goedgekeurd***

Given de order is klaar voor controle

When "markeerGecontroleerd()" wordt uitgevoerd

And "wijzigStatus()" wordt aangeroepen

Then moet de order klaar zijn voor verzending

Scenario: ***Order wordt verzonden***

Given de order is goedgekeurd voor verzending

When "verzendOrder()" wordt uitgevoerd

And "wijzigStatus()" wordt aangeroepen

Then moet de orderstatus veranderen naar "Verzonden"

And er moet een trackingnummer worden gegenereerd

Scenario: ***Levering succesvol bevestigd***

Given de order is in status "Verzonden"

When "bevestigLevering()" wordt uitgevoerd

And "wijzigStatus()" wordt aangeroepen

Then moet de orderstatus veranderen naar "Afgerond"

And de klant moet een bevestigingsmail ontvangen

Scenario: ***Orderproces voltooid***

Given de order is in status "Afgerond"

When het eindproces wordt uitgevoerd

Then moet de orderstatus veranderen naar "Final"

And alle resources moeten worden vrijgegeven

Scenario: ***Ordervalidatie mislukt***

Given de order is in status "In Behandeling"

When "valideerControle()" wordt uitgevoerd

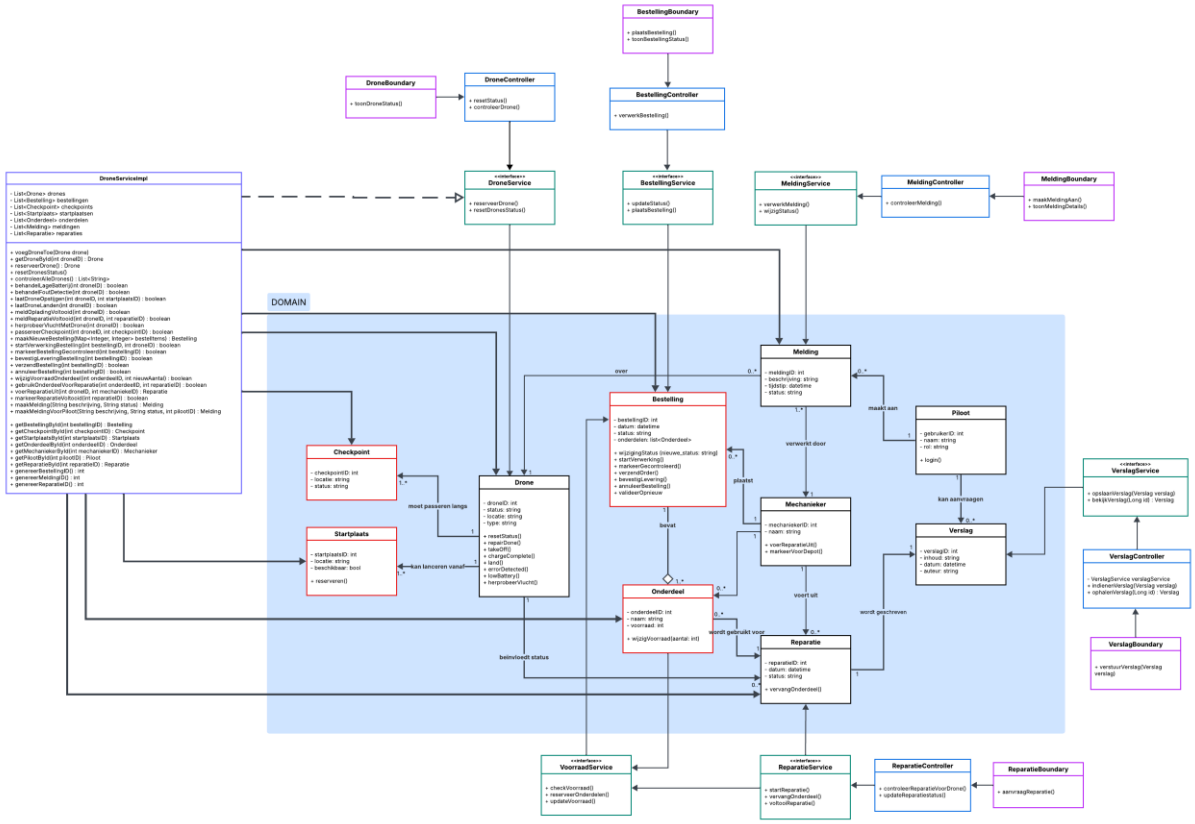
And er ontbreken verplichte gegevens

Then moet de order in "In Behandeling" blijven

And er moet een foutmelding worden gegenereerd

3.4. Service-, Controle- & Boundary-klasse

Volgens het designprincipe van layered architecture hebben we het systeem opgedeeld in service-, controle- en boundaryklassen. Deze verdeling helpt om verantwoordelijkheden gescheiden te houden en het systeem overzichtelijk te maken.



3.5. State Transition Testing

In dit onderdeel testen we of de objecten in het systeem correct door hun verschillende statussen gaan. Op basis van het toestandsdiagram controleren we of bepaalde acties leiden tot de juiste overgang van de ene toestand naar de andere. Zo kunnen we nagaan of het systeem zich correct gedraagt in situaties waarbij de status van een object verandert.

We beschrijven hier testscenario's waarin een object start in een bepaalde status, een actie uitvoert, en vervolgens een verwachte nieuwe status krijgt. Dit helpt om fouten in de logica van statusveranderingen op te sporen.

ID	Begintoestand	Actie	Eindtoestand	Verwachte Output
ST001	Defect	Mechanieker start reparatie	In reparatie	Drone gaat naar 'In Reparatie'
ST002	Reparatie geslaagd	Reparatie geslaagd	Gerepareerd	Drone wordt gemarkeerd als 'Gerepareerd'
ST003	Gerepareerd	Mechanieker reset status	Klaar voor gebruik	Drone is klaar om ingezet te worden
ST004	Defect	Reparatie ter plaatse niet mogelijk	Verstuurd naar depot	Drone wordt verzonden naar het depot
ST005	Gerepareerd	Onjuiste actie: terug naar 'Defect'	Ongeldige overgang	**Foutmelding:** Overgang niet toegestaan
ST006	Defect	Netwerkfout tijdens statusupdate	Blijft in 'Defect'	**Foutmelding:** Actie niet verwerkt
ST007	In Reparatie	Systeemcrash voor statuswijziging	Blijft in 'In Reparatie'	Reparatie niet voltooid

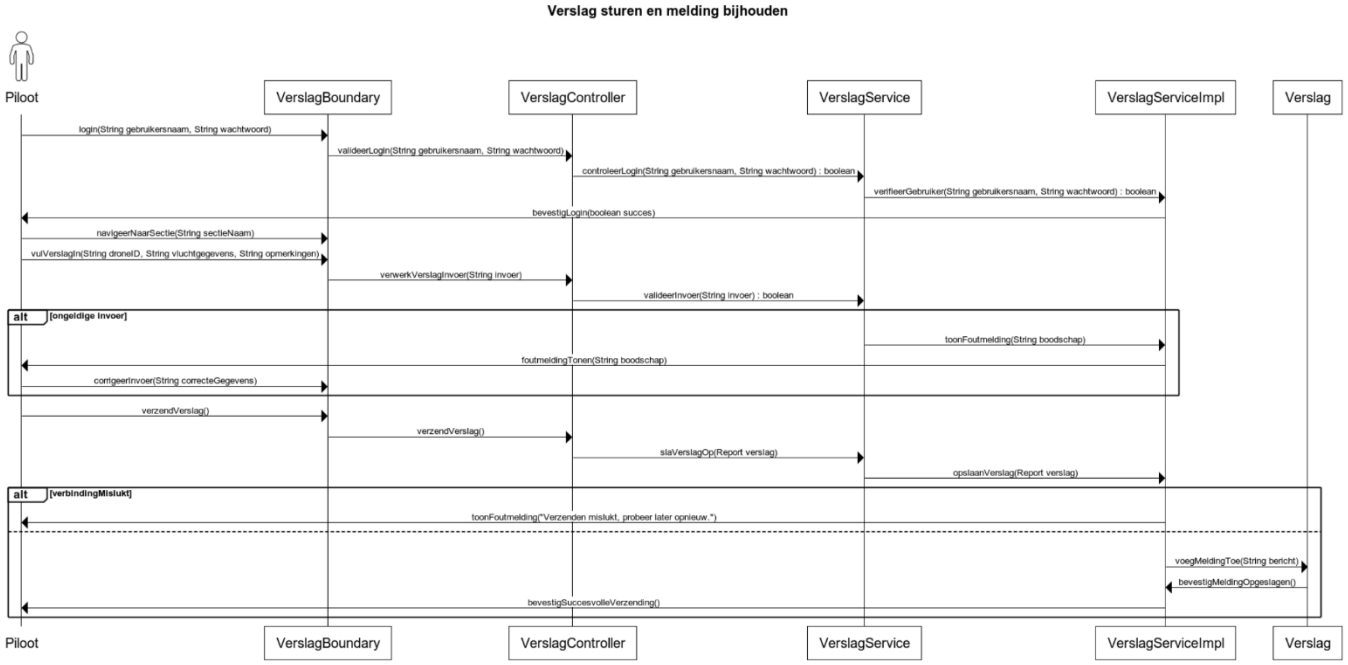
4. Deel 4

Dit sequence diagram toont de volgorde van interacties tussen een piloot en het systeem bij het indienen van een verslag. Zo wordt duidelijk welke methodes aangeroepen worden en in welke volgorde.

4.1. Sequence diagram

Betrokken domeinobjecten:

- Piloot
- Verslag



4.1.1. Java-Code

```
public class Verslag {
    private int VerslagId;
    private String Inhoud;
    private Date Datum;
    private DronePiloot piloot;

    public Verslag() {}

    public int getVerslagId() { return VerslagId; }

    public void setVerslagId(int verslagId) { VerslagId = verslagId; }

    public String getInhoud() { return Inhoud; }

    public void setInhoud(String inhoud) { Inhoud = inhoud; }

    public Date getDatum() { return Datum; }

    public void setDatum(Date datum) { Datum = datum; }

    public DronePiloot getPiloot() { return piloot; }

    public void setPiloot(DronePiloot piloot) { this.piloot = piloot; }
}
```

De volledige code kan je terugvinden in bijlage onderaan.

In bijlage tonen we de belangrijkste klassen en methodes die we zelf hebben geïmplementeerd tijdens dit project. De code is geschreven in Java en sluit zoveel mogelijk aan bij de user stories, use cases, klassendiagrammen en toestandsdiagrammen die eerder in deze paper zijn uitgewerkt.

We hebben ons best gedaan om ervoor te zorgen dat de code goed aansluit bij alles wat we in de vorige fases van het project hebben voorbereid. Denk hierbij aan het respecteren van statussen uit het toestandsdiagram, het overnemen van logica uit de use cases, en het structureren van de klassen zoals voorgesteld in ons klassendiagram.

Daarnaast voegden we ook services toe om het gedrag van de objecten op een duidelijke manier te beheren, zoals het starten van een dronevlucht of het verwerken van meldingen. Op die manier vormt de code een logische vertaling van ons hele analyse- en ontwerptraject

4.2. Unit-testen voor ServiceImpl-klasse

Onderstaande klasse bevat een reeks unit-testen voor de `DroneService`. We hebben deze tests geschreven om te controleren of de methodes uit de service correct reageren op verschillende statussen van een drone. Elke test start vanuit een bepaalde beginsituatie en controleert of de status van de drone verandert zoals verwacht na het uitvoeren van een actie.

Bijvoorbeeld: als een drone op “INACTIVE” staat en je roept `startCharging()` aan, dan moet de status veranderen naar “CHARGING”. Op die manier testen we stapsgewijs of onze implementatie overeenkomt met de logica die we eerder uitwerkten in het toestandsdiagram.

Deze testen geven ons meer vertrouwen dat de drone zich correct gedraagt in verschillende situaties, en helpen om fouten snel op te sporen wanneer we de code aanpassen.

DroneServiceTest.java

```
package be.odisee.citymesh;

import org.joda.time.DateTime;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class DroneServiceTest {
    private DroneService droneService;
    private Drone drone;
    private Melding melding = new Melding(1, "De drone is kapot",
DateTime.now().toDate(), Melding.MeldingStatus.MAINTENANCE_REQUIRED);

    @BeforeEach
    void setUp() {
        droneService = new DroneService();
        drone = new Drone(1, Drone.DroneStatus.INACTIVE, "1,1","DJI");
    }

    @Test
    void testStartCharging() {
        drone.setStatus(Drone.DroneStatus.INACTIVE);
        droneService.startCharging(drone);
        assertEquals(Drone.DroneStatus.CHARGING, drone.getStatus());
    }

    @Test
```

```
    void testChargingToInactive() {
        drone.setStatus(Drone.DroneStatus.CHARGING);
        droneService.resetDroneStatus(drone);
        assertEquals(Drone.DroneStatus.INACTIVE, drone.getStatus());
    }

    @Test
    void testStartDroneVlucht() {
        drone.setStatus(Drone.DroneStatus.RESERVED);
        droneService.startDroneVlucht(drone);
        assertEquals(Drone.DroneStatus.INFLIGHT, drone.getStatus());
    }

    @Test
    void testInFlightToMaintenance() {
        drone.setStatus(Drone.DroneStatus.DEFECT);
        droneService.startReparatie(drone, melding);
        assertEquals(Drone.DroneStatus.MAINTENANCE, drone.getStatus());
    }

    @Test
    void testMaintenanceToInactive() {
        drone.setStatus(Drone.DroneStatus.MAINTENANCE);
        droneService.resetDroneStatus(drone);
        assertEquals(Drone.DroneStatus.INACTIVE, drone.getStatus());
    }

    @Test
    void testReserveerDrone() {
        drone.setStatus(Drone.DroneStatus.INACTIVE);
        droneService.reserveerDrone(drone);
        assertEquals(Drone.DroneStatus.RESERVED, drone.getStatus());
    }

    @Test
    void testResetDroneStatus() {
        drone.setStatus(Drone.DroneStatus.INFLIGHT);
        droneService.resetDroneStatus(drone);
        assertEquals(Drone.DroneStatus.INACTIVE, drone.getStatus());
    }

    @Test
    void testVerwerkDroneLocatie() {
        drone.setStatus(Drone.DroneStatus.INFLIGHT);
        drone.setLocatie("Brussel");
```

```
        assertEquals("Brussel", droneService.verwerkDroneLocatie(drone));
    }

    @Test
    void testVerwerkDroneLocatie_NotInFlight() {
        drone.setStatus(Drone.DroneStatus.CHARGING);
        assertNull(droneService.verwerkDroneLocatie(drone));
    }
}
```

4.3. Gherkin-acceptatietestscenario's

We hebben gekozen voor scenario's die relevant zijn voor de piloot, zoals het inloggen, het invullen en verzenden van een verslag, en het omgaan met fouten zoals verbindingsproblemen of ongeldige invoer. Elk scenario vertrekt vanuit een bepaalde beginsituatie en test of het systeem zich correct gedraagt, zoals ook beschreven in de bijhorende use cases en toestandsdiagrammen.

***Scenario:* Piloot logt succesvol in**

Given een piloot heeft een geldige gebruikersnaam en wachtwoord

When de piloot probeert in te loggen

Then wordt de login gevalideerd

And de piloot krijgt een bevestiging van succesvolle login

***Scenario:* Piloot vult een ongeldig verslag in**

Given de piloot is ingelogd

And de piloot vult het verslag in met ongeldige gegevens

When de piloot probeert het verslag te verzenden

Then toont het systeem een foutmelding

And de piloot krijgt de kans om de invoer te corrigeren

***Scenario:* Piloot dient een correct verslag in**

Given de piloot is ingelogd

And de piloot heeft alle vereiste velden correct ingevuld

When de piloot het verslag verzendt

Then wordt het verslag opgeslagen

And ontvangt de piloot een bevestiging van succesvolle verzending

And wordt er een melding aan het verslag toegevoegd

Scenario: ***Verbindingsprobleem tijdens verzenden***

Given de piloot is ingelogd

And de piloot heeft alle velden correct ingevuld

When de piloot probeert het verslag te verzenden

And er is een verbindingsprobleem

Then toont het systeem een foutmelding

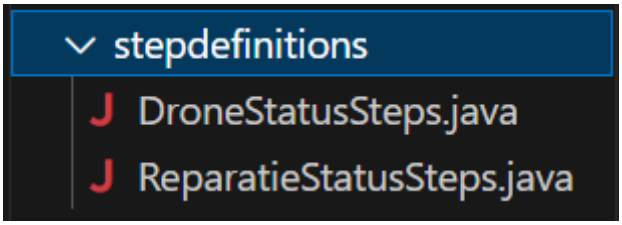
And de piloot kan het later opnieuw proberen

4.3.1. Step definition

Hieronder tonen we de step definitions die we gekoppeld hebben aan onze Gherkin-scenario's voor het testen van de dronefunctionaliteit en reparatieprocessen. Deze code hoort bij de acceptatietesten die we eerder hebben opgesteld.

We gebruikten Cucumber om natuurlijke taal (zoals "Als de piloot een startcommand geeft") te koppelen aan echte Java-code. Zo konden we gedrag zoals het starten van een dronevlucht, opladen bij lage batterij of het uitvoeren van een reparatie automatisch testen.

De stappen sluiten nauw aan bij de user stories, het toestandsdiagram van de drone en de testscenario's die we eerder hebben uitgeschreven. Ze helpen om te controleren of de drone en het systeem correct reageren op acties van de piloot of de mechanieker.



DroneStatusSteps.java

```
package be.odisee.citymesh.stepdefinitions;

import be.odisee.citymesh.Drone;
import be.odisee.citymesh.DroneService;
import io.cucumber.java.nl.Gegeven;
import io.cucumber.java.nl.Als;
import io.cucumber.java.nl.Dan;
import org.jetbrains.annotations.NotNull;

import static org.junit.Assert.*;

public class DroneStatusSteps {
    private Drone drone;
    private DroneService droneService = new DroneService();

    @Gegeven("een drone met status {string}")
    public void een_drone_met_status(@NotNull String status) {
        Drone.DroneStatus droneStatus =
Drone.DroneStatus.valueOf(status.toUpperCase());
        drone = new Drone(1, droneStatus, "Locatie 1", "Standard");
    }
```

```
    @Als("de piloot een startcommand geeft")
    public void de_piloot_een_startcommand_geeft() {
        droneService.reserveerDrone(drone);
        droneService.startDroneVlucht(drone);
    }

    @Als("de drone landt op een geautoriseerde locatie")
    public void de_drone_landt_op_een_geautoriseerde_locatie() {
        droneService.resetDroneStatus(drone);
    }

    @Als("de batterij bijna leeg is")
    public void de_batterij_bijna_leeg_is() {
        droneService.startCharging(drone);
    }

    @Dan("verandert de status van de drone naar {string}")
    public void verandert_de_status_van_de_drone_naar(@NotNull String
verwachteStatus) {
        assertEquals(Drone.DroneStatus.valueOf(verwachteStatus.toUpperCase()),
drone.getStatus());
    }
}
```

ReparatieStatusSteps.java

```
package be.odisee.citymesh.stepdefinitions;

import be.odisee.citymesh.Drone;
import be.odisee.citymesh.DroneService;
import be.odisee.citymesh.Melding;
import io.cucumber.java.nl.Gegeven;
import io.cucumber.java.nl.Als;
import io.cucumber.java.nl.Dan;
import io.cucumber.java.nl.En;
import org.jetbrains.annotations.NotNull;

import static org.junit.Assert.*;

public class ReparatieStatusSteps
{
    private Drone drone;
    private DroneService droneService = new DroneService();
    private Melding melding;
    private String foutmelding;
```

```
    private boolean onderdelenBeschikbaar = true;

    @Gegeven("een defecte drone met status {string}")
    public void een_drone_met_status(@NotNull String status) {
        drone = new Drone(1, Drone.DroneStatus.valueOf(status.toUpperCase()),
"Locatie 1", "Standard");
    }

    @Gegeven("er zijn geen vervangende onderdelen beschikbaar")
    public void er_zijn_geen_vervangende_onderdelen_beschikbaar() {
        onderdelenBeschikbaar = false;
    }

    @Gegeven("de mechanieker beoordeelt dat reparatie ter plaatse niet
mogelijk is")
    public void
de_mechanieker_beoordeelt_dat_reparatie_ter_plaats_niet_mogelijk_is() {

    }

    @Als("de mechanieker de drone inspecteert en repareert")
    public void de_mechanieker_de_drone_inspecteert_en_repareert() {
        if (onderdelenBeschikbaar) {
            droneService.resetDroneStatus(drone);
        }
    }

    @Als("de mechanieker een reparatie wil uitvoeren")
    public void de_mechanieker_een_reparatie_wil_uitvoeren() {
        if (!onderdelenBeschikbaar) {
            foutmelding = "Reparatie niet mogelijk";
        }
    }

    @Als("de mechanieker de drone markeert voor verzending naar het depot")
    public void
de_mechanieker_de_drone_markeert_voor_verzending_naar_het_depot() {
        droneService.reserveerDrone(drone);
    }

    @Dan("moet de status van de drone veranderen naar {string}")
    public void moet_de_status_van_de_drone_veranderen_naar(String
verwachteStatus) {
        assertEquals(Drone.DroneStatus.valueOf(verwachteStatus.toUpperCase()),
drone.getStatus());
```

```
    }

    @Dan("moet het systeem een foutmelding geven: {string}")
    public void moet_het_systeem_een_foutmelding_geven(String
verwachteFoutmelding) {
        assertEquals(verwachteFoutmelding, foutmelding);
    }

    @Dan("de status van de drone blijft {string}")
    public void de_status_van_de_drone_blijft(String verwachteStatus) {
        assertEquals(Drone.DroneStatus.valueOf(verwachteStatus.toUpperCase()),
drone.getStatus());
    }

    @En("wanneer de mechanieker de status reset, moet de drone {string} zijn")
    public void
wanneer_de_mechanieker_de_status_reset_moet_de_drone_zijn(String
verwachteStatus) {
        droneService.resetDroneStatus(drone);
        assertEquals(Drone.DroneStatus.valueOf(verwachteStatus.toUpperCase()),
drone.getStatus());
    }
}
```

4.4. Sequence Diagram Testing

Testgeval 1: Succesvol inloggen

ID	Testactie	Verwacht resultaat
SD001	Piloot voert correcte inloggegevens in	verifieerGebruiker() retourneert true, en bevestigLogin() wordt aangeroepen
SD002	Piloot voert foutieve inloggegevens in	verifieerGebruiker() retourneert false, foutmelding wordt getoond
SD003	Piloot laat gebruikersnaam of wachtwoord leeg	verifieerGebruiker() retourneert false, foutmelding wordt getoond

Testgeval 2: Invoer validatie bij verslag indienen

ID	Testactie	Verwacht resultaat
SD004	Piloot voert correcte invoer in	valideerInvoer() retourneert true en verslag wordt verwerkt
SD005	Piloot laat een verplicht veld leeg	valideerInvoer() retourneert false, foutmelding wordt getoond
SD006	Piloot voert ongeldige data in	valideerInvoer() retourneert false, foutmelding wordt getoond

Testgeval 3: Succesvol verslag indienen

ID	Testactie	Verwacht resultaat
SD007	Piloot verzend correct verslag	slaVerslagOp() wordt aangeroepen en melding word toegevoegd
SD008	Piloot voegt melding toe aan verslag	voegMeldingToe() wordt correct uitgevoerd

5. Deel 5

In dit deel tonen we hoe we gebruik hebben gemaakt van **Dependency Injection (DI)** en **Aspect-Oriented Programming (AOP)** met behulp van Spring. We voegden de nodige dependencies toe in het project om Spring AOP en AspectJ te gebruiken.

Via de configuratieklasse (`AppConfig`) hebben we Spring zo ingesteld dat aspecten automatisch worden opgepikt. De `DroneServiceLoggingAspect` zorgt ervoor dat er telkens een logbericht wordt weergegeven vóór en na het uitvoeren van een methode in de `DroneService`.

Op die manier kunnen we bepaalde zaken (zoals logging) scheiden van de hoofdlogica, zonder de serviceklassen zelf aan te passen. Dit past binnen het principe van “separation of concerns” en toont hoe AOP in een eenvoudige vorm toegepast kan worden in ons project.

5.1. Dependency Injection

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.3.34</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.20</version>
</dependency>
```

5.2. AOP

Spring config. Klasse

```
package be.odisee.citymesh;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@ComponentScan("be.odisee.citymesh")
@EnableAspectJAutoProxy
public class AppConfig {
}
```

Aspect klasse

```
package be.odisee.citymesh.aspects;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.JoinPoint;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class DroneServiceLoggingAspect {

   @Before("execution(* be.odisee.citymesh.DroneService.*(..))")
   public void logBeforeMethod(JoinPoint joinPoint) {
      System.out.println("[AOP] Start methode: " +
joinPoint.getSignature().getName());
   }

   @After("execution(* be.odisee.citymesh.DroneService.*(..))")
   public void logAfterMethod(JoinPoint joinPoint) {
      System.out.println("[AOP] Einde methode: " +
joinPoint.getSignature().getName());
   }
}
```

Conclusie

Tijdens deze opdracht hebben we stap voor stap toegewerkt naar een complete analyse en ontwerp van een serverapplicatie. Door te starten vanuit user stories en die geleidelijk te vertalen naar diagrammen en testscenario's, kregen we inzicht in hoe je op een gestructureerde manier een systeem opbouwt. We brachten de belangrijkste klassen en hun gedrag in kaart, stelden een degelijk klassendiagram op en koppelden dit aan de flow van de applicatie via een sequentiediagram. Ook hebben we getest hoe objecten reageren in verschillende situaties via acceptatietesten die gebaseerd zijn op hun toestanden. Dankzij deze analyse beschikken we nu over een duidelijke basis om verder te bouwen aan een werkend systeem.

Reflecties

Groep

Tijdens dit project hebben we als groep niet alleen gewerkt aan onze technische vaardigheden, maar ook aan onze samenwerking. Inhoudelijk zat het goed: voor sommigen was een groot deel van de leerstof herhaling, terwijl anderen net veel bijleerden over systeemanalyse, ontwerpen en testen. Dankzij de duidelijke uitleg op Toledo en de wekelijkse coachingmomenten konden we alles goed blijven volgen en wisten we waar we aan toe waren.

Wat echt opviel, was hoe vlot de samenwerking verliep. Hoewel we elkaar in het begin niet kenden, klikte het snel binnen de groep. Iedereen nam verantwoordelijkheid, de taken werden eerlijk verdeeld en als iemand ergens mee vastzat, werd er altijd geholpen. Er was echt sprake van teamwork.

Naast de technische kennis, zoals het opstellen van BPMN-diagrammen, user stories en acceptatietests, leerden we ook hoe belangrijk goede communicatie en samenwerking zijn binnen een project. We kijken dan ook tevreden terug op dit groepsproject, zowel inhoudelijk als op vlak van samenwerking.

Individueel

Oumaima:

Tijdens dit project viel het qua leerstof best goed mee. Meer dan de helft was voor mij herhaling, waardoor het allemaal niet zo moeilijk aanvoelde. Er stond ook genoeg uitleg op Toledo via de slides en opnames, wat echt handig was. De wekelijkse coachingmomenten waren ook een grote hulp als er toch nog iets onduidelijk was.

Wat de samenwerking betreft: die verliep verrassend goed. In het verleden heb ik al vaker in groep moeten werken met mensen die ik niet kende, en dat liep niet altijd even vlot. Maar in deze groep voelde ik me meteen welkom. Iedereen was super behulpzaam en er hing een fijne sfeer. De taken werden elke week goed verdeeld, en als iemand ergens op vastzat, werd er zonder probleem geholpen. Het was helemaal geen situatie van “ieder voor zich”, maar echt samenwerken als team.

Noa:

Tijdens dit vak heb ik veel bijgeleerd over analyse, ontwerp en testen van serverapplicaties. Ik heb niet alleen de leerstof beter begrepen, maar ook praktische vaardigheden ontwikkeld, zoals het maken van systeemanalyses, het ontwerpen van softwarearchitecturen en het testen van applicaties om ze betrouwbaarder en efficiënter te maken. Daarnaast heb ik nieuwe klasgenoten leren kennen, wat ik erg leuk vond. De samenwerking binnen de groep verliep vlot, omdat de taken eerlijk werden verdeeld en iedereen bereid was om elkaar te helpen. Dit zorgde voor een fijne werksfeer waarin we samen ideeën konden uitwisselen en

elkaar konden ondersteunen bij uitdagingen. Hierdoor heb ik niet alleen mijn kennis vergroot, maar ook waardevolle connecties opgebouwd binnen de klas.

Younes:

Tijdens dit vak heb ik op een praktische manier geleerd hoe je samenwerkt aan een project met mensen die je in het begin nog niet kent. Dat was in het begin even wennen, maar al snel merkte ik hoe waardevol samenwerking is binnen zo'n projectcontext. Ik heb gewerkt aan BPMN-diagrammen, user stories, acceptatietests in Gherkin en unit testing. Vooral het schrijven van tests en het toepassen van dependency injection deden me inzien hoe belangrijk structuur en betrouwbaarheid zijn in softwareontwikkeling. Daarbij kreeg ik ook veel hulp van mijn teamgenoten – bijvoorbeeld bij het opstellen van de BPMN met Walid of bij het nadenken over de user stories samen met de hele groep.

Ook de samenwerking in onze groep verliep heel vlot. Iedereen nam verantwoordelijkheid en we stonden altijd klaar om elkaar te helpen wanneer iets niet duidelijk was. Dat maakte het werk niet alleen efficiënter, maar ook aangenamer. Door dit vak heb ik niet alleen mijn technische kennis verdiept, maar ook geleerd om beter te communiceren en samen te werken aan een gemeenschappelijk doel.

Oumou:

Dit vak vond ik een stuk moeilijker dan dat van het vorige semester. Er kwamen veel nieuwe dingen aan bod, waardoor ik veel meer tijd moest steken in het begrijpen van alle leerstof. Vooral de verschillende testmethodes waren soms best overweldigend.

Tijdens dit vak (en ook in het vorige semester) werd er constant benadrukt dat testing superbelangrijk is. We hebben allerlei testen gedaan, zoals Gherkin met Cucumber, logische testen en unit testen. Dat klonk op zich logisch, maar pas bij de opdrachten vanaf de use case testing besefte ik pas écht hoeveel verschillende testen er zijn en hoeveel je er moet doen. Op een bepaald moment raakte ik zelfs verward door de vele testvormen en hun specifieke toepassingen.

Het groepswerk verliep eigenlijk beter dan verwacht. We werkten goed samen, iedereen deed eerlijk zijn deel en de communicatie zat goed. Dat maakte het een stuk minder stressvol dan ik had gedacht.

Over mezelf heb ik gemerkt dat ik misschien minder verantwoordelijkheden op mij moet nemen. Ik heb de neiging om te veel op mij te pakken, en dat kan soms best vermoeiend zijn. In de toekomst wil ik daar beter op letten en proberen een betere balans te vinden.

Walid:

De eerste opdrachten gingen goed want we hadden die leerstof al een keer gehad. Daardoor wist ik wat we moest doen. Het was wel moeilijk om te beginnen met een nieuw bedrijf in het project. In het begin was het niet duidelijk wat we precies moesten doen dus dat was even wennen.

Opdracht 3b vond ik het moeilijkst. Vooral het deel over state transition testing vond ik lastig. Het was niet zo makkelijk om goed te snappen wat we moesten doen. Gelukkig had ik een goede groep. Iedereen hielp elkaar en we werkten goed samen. Als iemand iets niet wist dan legde iemand anders het uit. Zo konden we het toch goed maken. Het samenwerken was fijn en dat maakte het werken makkelijker en leuker.

Juliaan:

Ik heb me vooral gericht op het schrijven van de code en unit tests, wat mijn technische vaardigheid heeft versterkt. Door de samenwerking met de groep leerde ik hoe belangrijk duidelijke communicatie is tussen ontwerp en implementatie. Mijn team was fantastisch – iedereen was gemotiveerd, hielp elkaar en maakte het project leuk én succesvol. Soms was ik te gefocust op details, waardoor tijdmanagement een aandachtspuntje bleef. Toch kijk ik tevreden terug, omdat we al bij al een mooi resultaat hebben ontwikkeld. In de toekomst wil ik nog meer letten op documentatie en efficiënter plannen.

Time-sheets & Trello

Via de timesheet hielden we bij wie aan welke taak werkte, op welke datum en hoe lang die persoon eraan bezig was. Dit gaf ons een goed overzicht van de verdeling van het werk en hoeveel tijd er in elke taak kroop. Soms werkten we individueel aan een deel van de opdracht, en dankzij de timesheet konden we ook nagaan hoeveel werk elk groepslid al had verricht.

Younes:

Datum	Begin	Einde	Duur	Activiteit		Totale Duur
15-2-2025	12:30:00	13:40:00	01:10:00	BPMN		18:00:00
22-2-2025	11:00:00	11:45:00	00:45:00	BPMN verbeterd		
25-2-2025	10:30:00	11:15:00	00:45:00	processen, requirements en user stories (deel 2a)		
9-3-2025	11:00:00	12:30:00	01:30:00	Gherkin-acceptatietestscenario's (toestandsdiagrammen)		
18-3-2025	23:00:00	23:30:00	00:30:00	Begonnen met schrijven van unit testen		
20-3-2025	13:30:00	16:00:00	02:30:00	Unit testen voor serviceimpl		
28-3-2025	22:00:00	22:30:00	00:30:00	Injection		
29-3-2025	11:00:00	12:20:00	01:20:00	Dependency injection		
1-4-2025	12:00:00	12:30:00	00:30:00	AI controle gherkin + dependency injection		
1-4-2025	14:00:00	22:30:00	08:30:00	wrap up presentatie (verbetering, nakijken, code, tekst schrijven,...)		
			00:00:00			

Oumaima:

Datum	Begin	Einde	Duur	Activiteit	Totale Duur
18-2-2025	16:50:00	18:50:00	02:00:00	klassendiagram	20:10:00
22-2-2025	14:00:00	18:00:00	04:00:00	use case diagram	
22-2-2025	13:30:00	14:00:00	00:30:00	verbeteren klassendiagram	
23-2-2025	13:00:00	14:00:00	01:00:00	use case beschrijving	
30-3-2025	14:00:00	17:00:00	03:00:00	verbetering use case beschrijving + klassendiagram	
31-3-2025	16:00:00	20:00:00	04:00:00	Paper in orde brengen deel 1 (reflectie schrijven, inleiding, conclusie & alles erin toevoegen...)	
1-4-2025	00:00:00	01:40:00	01:40:00	Paper in orde brengen deel 2	
1-4-2025	18:00:00	22:00:00	04:00:00	Paper afwerken	

Oumou:

Datum	Begin	Einde	Duur	Activiteit	Totale Duur
17/02/2025	16:20:00	18:00:00	01:40:00	Requirements	23:50:00
17/02/2025	19:05:00	19:35:00	00:30:00	User-stories (INVEST)	**04:20:00**
18/02/2025	19:20:00	20:00:00	00:40:00	Weekverslag	**opgeteld: 28u 10min**
25/02/2025	09:30:00	12:00:00	02:30:00	Activity diagram	
25/02/2025	12:10:00	12:50:00	00:40:00	BPMN & Requirements verbeteren	
25/02/2025	21:00:00	21:30:00	00:30:00	Weekverslag	
06/03/2025	19:10:00	21:20:00	02:10:00	Toestands diagram	
11/03/2025	20:20:00	21:00:00	00:40:00	Weekverslag	
18/03/2025	10:00:00	12:50:00	02:50:00	Sequentiediagram	
18/03/2025	20:00:00	21:30:00	01:30:00	Service, control & boundary verbeteren	
19/03/2025	09:00:00	09:30:00	00:30:00	Weekverslag	
24/03/2025	11:50:00	13:30:00	01:40:00	verbeteringen	
25/03/2025	16:00:00	17:00:00	01:00:00	gherkin voor sequence diagram	
25/03/2025	21:00:00	22:30:00	01:30:00	serviceImpl-klasse	
28/03/2025	20:00:00	21:00:00	01:00:00	verbeteringen	
30/03/2025	16:00:00	20:30:00	04:30:00	verbeteringen + UI hermaken	
01/04/2025	18:30:00	22:50:00	04:20:00	paper + verslag + alles nakijken	

Juliaan:

Datum	Begin	Einde	Duur	Activiteit	Totale Duur
17-2-2025	19:00:00	20:00:00	01:00:00	woordenboek en uml klassediagram	19:15:00
20-2-2025	20:00:00	22:30:00	02:30:00	Domein klassen diagram	
25-2-2025	20:30:00	20:45:00	00:15:00	Actoren identificeren	
25-2-2025	23:00:00	01:00:00	03:00:00	use case testing	
3-3-2025	19:30:00	20:30:00	01:00:00	state transition testing	
12-3-2025	20:30:00	00:00:00	03:30:00	code tegoei setup	
19-3-2025	22:00:00	00:00:00	02:00:00	code fixen	
26-3-2025	23:00:00	23:00:00	00:00:00	testjes doen werken	
30-3-2025	21:00:00	00:00:00	03:00:00	step defs (gherkin)	
1-4-2025	20:30:00	23:30:00	03:00:00	code beetje afmaken	
			00:00:00		

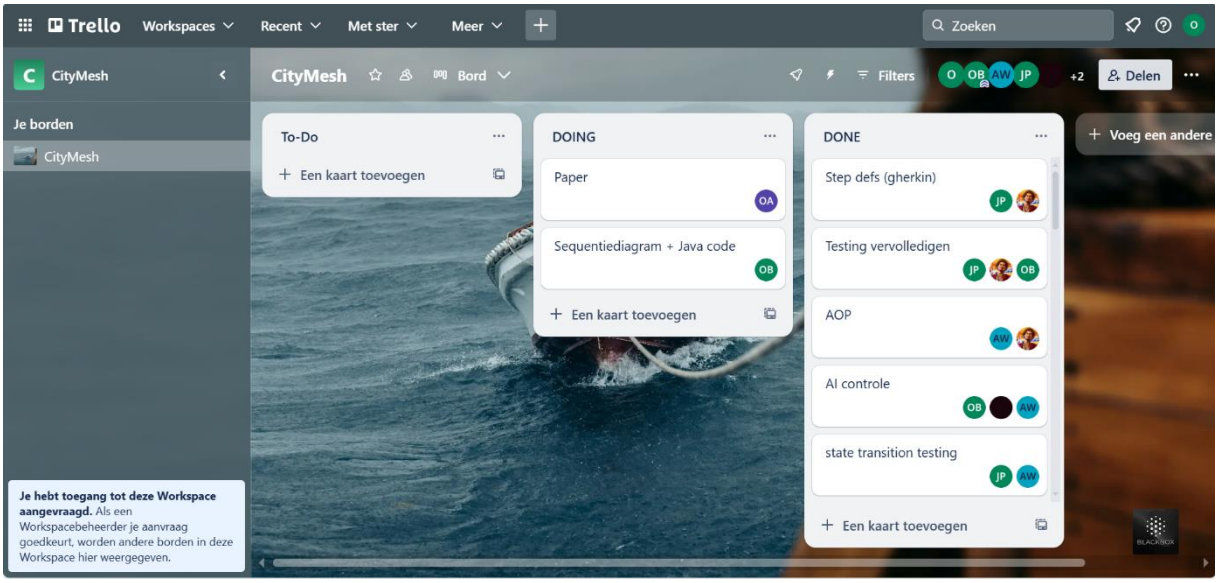
Noa:

Datum	Begin	Einde	Duur	Activiteit	Totale Duur
18-2-2025	20:00:00	21:30:00	01:30:00	Trello, IPO-Requirement (cyclus van het terrein)	12:55:00
20-2-2025	16:00:00	16:40:00	00:40:00	verbetering requirement	
24-2-2025	14:30:00	15:00:00	00:30:00	UI (website maken)	
25-2-2025	21:00:00	22:30:00	01:30:00	UI (papier versie en aanpassingen)	
9-3-2025	18:00:00	19:30:00	01:30:00	Service-, controle- en boundary-klassen	
15-3-2025	20:00:00	20:45:00	00:45:00	Service-, controle- en boundary-klassen (klassediagram) verbeteren	
24-3-2025	15:00:00	16:30:00	01:30:00	Beetje iedereen geholpen	
29-3-2025	16:00:00	18:00:00	02:00:00	Requirement opdracht 1 hermaken verbeteren (feedback leerkracht)	
31-3-2025	12:00:00	14:00:00	02:00:00	AI controle alles	
1-4-2025	13:00:00	14:00:00	01:00:00	Paper	

Walid:

Datum	Begin	Einde	Duur	Activiteit	Totale Duur
14-2-2025	13:00:00	15:00:00	02:00:00	Maken van BPMN	14:40:00
20-2-2025	14:00:00	15:00:00	01:00:00	Aanpassen van het gemaakte BPMN	
25-2-2025	20:00:00	21:00:00	01:00:00	Nakijken en verbeteren van de UI	
7-3-2025	19:00:00	20:00:00	01:00:00	Testontwerp met use case testing	
12-3-2025	19:00:00	20:00:00	01:00:00	verbeteren na feedback	
19-3-2025	19:00:00	21:00:00	02:00:00	State Transition Testing	
22-3-2025	19:00:00	20:30:00	01:30:00	State Transition Testing (verbeteren)	
24-3-2025	18:00:00	19:50:00	01:50:00	sequence testing+andere helpen	
28-3-2025	20:00:00	21:20:00	01:20:00	opdracht 5b	
29-3-2025	21:00:00	22:00:00	01:00:00	AI Controle	
1-4-2025	13:00:00	14:00:00	01:00:00	Paper	

Met Trello hadden we een handig overzicht van alle taken: wat nog moest gebeuren, waar al aan gewerkt werd, en wat volledig afgerond was. Zo wisten we altijd waar we stonden in het project. Het hielp ook om elkaar te herinneren aan deadlines en om te zien wie eventueel ergens vastzat, zodat we elkaar tijdig konden helpen

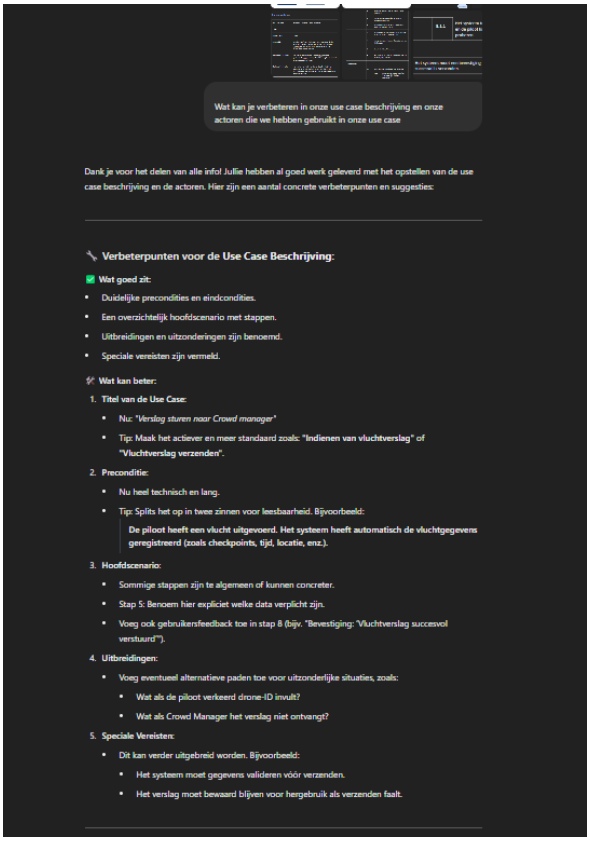


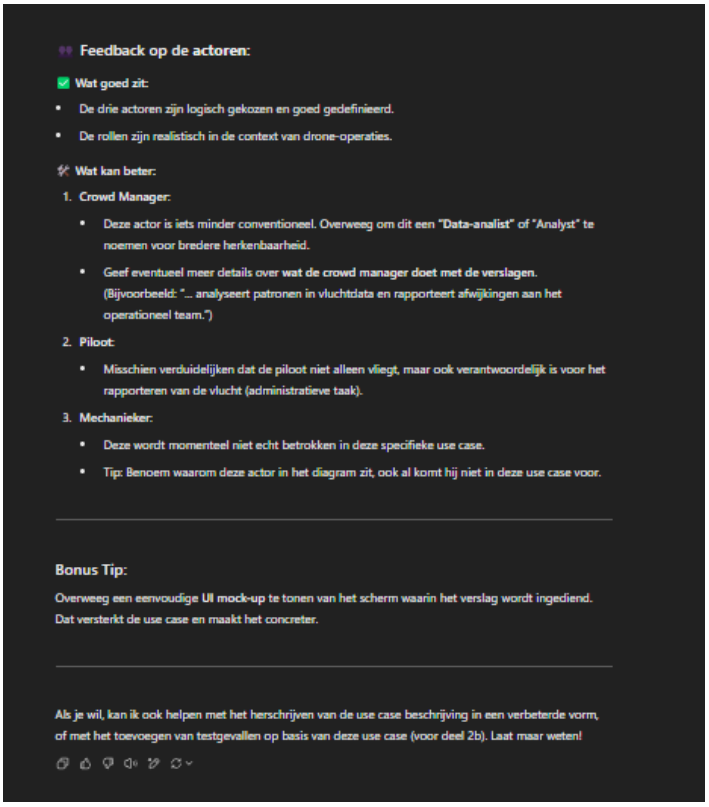
Link naar trello: [CityMesh | Trello]

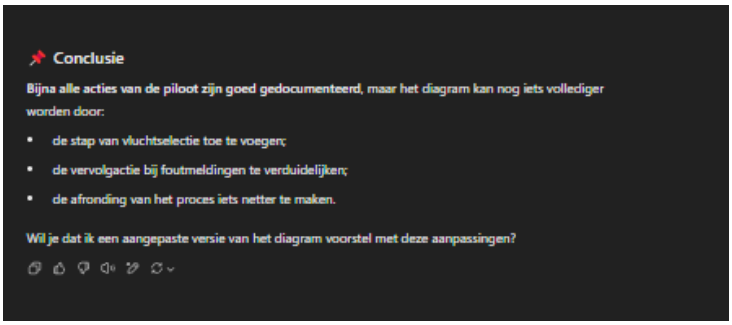
AI Controle

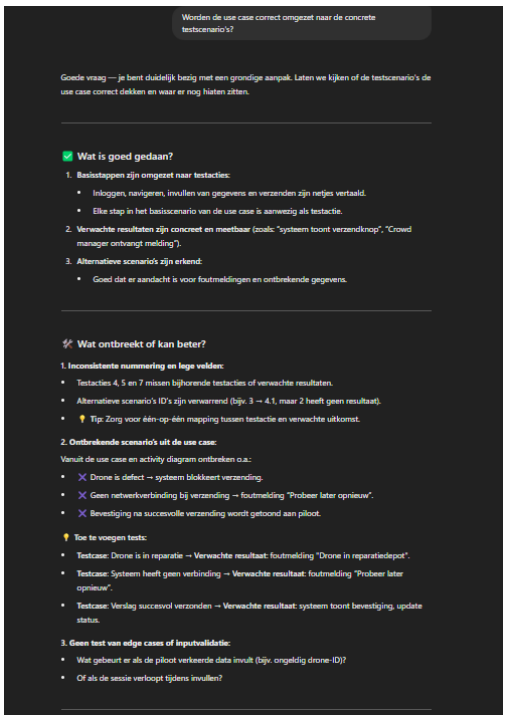
Voor dit onderdeel hebben we gebruikgemaakt van AI-ondersteuning om feedback te krijgen tijdens het uitwerken van verschillende onderdelen. De AI werd enkel ingezet als ondersteuning bij het schrijfproces of ter inspiratie waar nodig, alle inhoud en beslissingen blijven afkomstig van het team zelf.

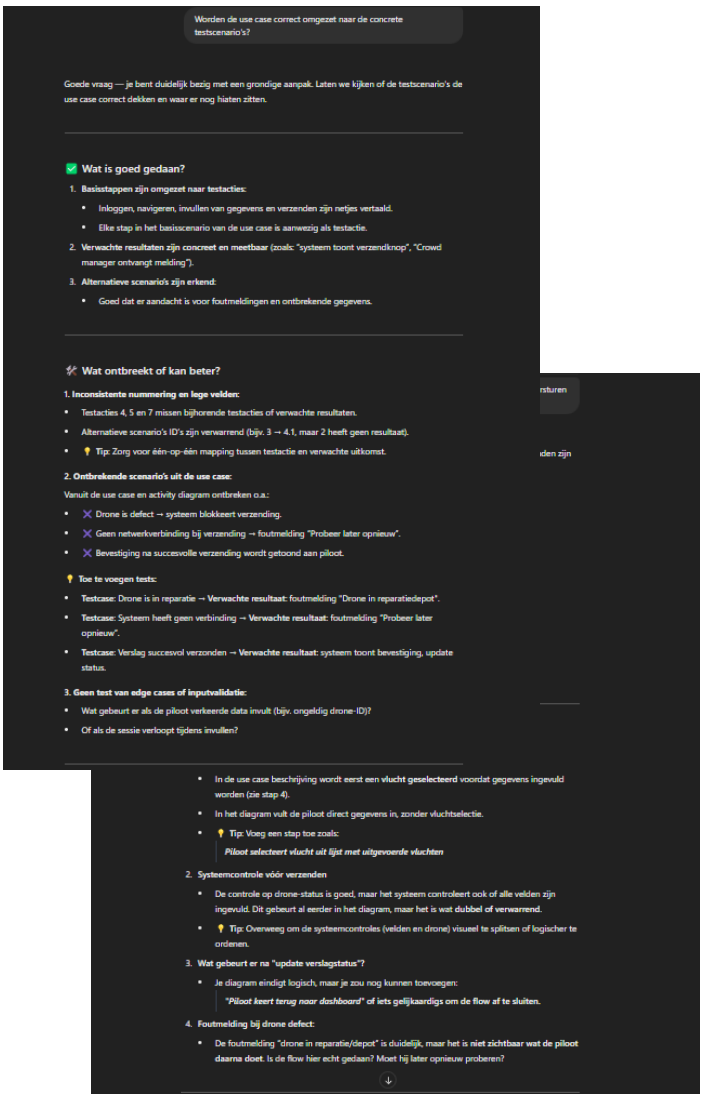
DEEL 2











Deel 3: Acceptatietesten

✅ Wat goed is:

- **Duidelijke US**: De user story (US004) is helder geformuleerd en sluit goed aan bij de praktijk.
- **Stateflow helder**: De toestandsovergangen van de drone zijn logisch en correct uitgewerkt.
- **Niet-triviaal gedrag benoemd**: Goed dat je expliciet benoemt dat dit gedrag niet-triviaal is.
- **Testcases overzichtelijk**: De tabel met testgevallen is netjes uitgewerkt met duidelijke voorwaarden, acties en verwachte uitkomsten.
- **Scenario's zijn realistisch**: Elk scenario is geloofwaardig en representeert een belangrijke use case.

Scenario: Reparatie mislukt door ontbrekende onderdelen

```gherkin
Given een drone met status "Defect"
And er zijn geen vervangende onderdelen beschikbaar
When de mechanieker probeert de reparatie te starten
Then toont het systeem: "Reparatie niet mogelijk"
And blijft de status "Defect"
```

🔧 Suggesties voor verbetering:

1. Taalconsistentie & formulering

- Gebruik consequent de tegenwoordige tijd in scenario's.
- Let op werkwoordgebruik: *"verandert worden"* → *"worden veranderd"* of beter: *"wordt gewijzigd in"*.
- "Resetten" → overweeg een Nederlands alterna[illegible] zoals "bijwerken" of "instellen".

Scenario: Drone moet naar depot worden gestuurd

```gherkin
Given een drone met status "Defect"
When de mechanieker beoordeelt dat reparatie ter plaatse niet mogelijk is
And de drone markeert voor verzending naar het depot
Then wordt de status gewijzigd naar "Naar Depot Verzonden"
```

4. Opmerking over "Referentie toestand in het ROOD"

- Goed idee, maar die aanduiding ontbreekt in de huidige weergave. Je zou eventueel de statusnamen kunnen *vetgedrukt* of **ROOD** mark[illegible]n als je de tekst visueel aanlevert.

2. Formuleringstips voor testgevallen

Overweeg om "Drone staat op..." te vervangen door "De drone heeft status..." voor iets formelere consistentie:

ID	Voorwaarde	Actie	Verwachte uitkomst
004.1	De drone heeft status "Defect"	Mechanieker probeert status te wijzigen naar "Klaar voor gebruik"	Wijziging niet toegestaan, foutmelding: "Drone moet eerst gerepareerd worden"
004.2	De drone heeft status "In Reparatie"	...	...

3. Scenarioformulering

Scenario: Drone wordt succesvol gerepareerd

```gherkin
Given een drone met status "Defect"
When de mechanieker de drone inspecteert en repareert
Then verandert de status naar "Gerepareerd"
And wanneer de mechanieker de status wijzigt
Then wordt de status "Klaar voor gebruik"
```

→ Gebruik consequent `Then`, niet mixen met `And moet`.

Toestandsdiagram: Drone

✅ Wat is goed aan het diagram:

- **Duidelijke toestanden**: De belangrijkste toestanden (`Inactive`, `InFlight`, `Charging`, `Maintenance`) zijn logisch gekozen en herkenbaar.
- **Acties goed benoemd**: Events zoals `takeOff()`, `lowBattery()`, `repairDone()` zijn betekenisvol en sluiten aan bij realistische situaties.
- **Overgangen kloppen logisch**: Bijvoorbeeld van `InFlight` naar `Charging` bij `lowBattery()` of naar `Maintenance` bij `errorDetected()`.

🔍 Verbeterpunten / Opmerkingen:

1. Spelfout:

- `Maintenace` → moet zijn: **Maintenance** (typfoutje)

2. Missende status "Gerepareerd" (in relatie tot je acceptatietesten)

- In je testscenario's is er sprake van een tussens ↓ s **"Gerepareerd"**, maar die zie ik niet terug in dit diagram.

2. Missende status "Gerepareerd" (in relatie tot je acceptatietesten)

- In je testscenario's is er sprake van een tussenstatus **"Gerepareerd"**, maar die zie ik niet terug in dit diagram.
  - Suggestie: Voeg een extra status toe tussen `Maintenance` en `Inactive`, bijvoorbeeld `Repaired` of `ReadyForReset`.
  - Of: verwerk `repairDone()` → `Repaired`, en dan `resetStatus()` → `Inactive`.

3. Ongelijkmatige pijlrichting

- De overgang van `Maintenance` terug naar `Inactive` is top-down, terwijl de rest vooral horizontaal werkt. Overweeg meer consistentie in richting (bijvoorbeeld alles horizontaal), voor leesbaarheid.

4. Dubbele `land()`

- Zowel van `InFlight` als van `Charging` gaat `land()` naar `Inactive`. Dat is logisch voor `InFlight`, maar minder voor `Charging`. Misschien bedoelde je `chargeComplete()` daar?

5. Extra states voor volledigheid (optioneel)

- Als je testscenario's zoals "Naar depot verzonden" wilt dekken, overweeg dan een extra toestand als `SentToDepot` of `OutOfService`.

↓

Toestandsdiagram: Bestelling

🔍 Verbeterpunten of suggesties:

1. Consistentie actie/overgang

De acties zijn correct genoteerd als `event() / actie()`, bijvoorbeeld:

```
verzendOrder() / wijzigStatus()
```

Maar in de praktijk is `wijzigStatus()` vaak impliciet bij het uitvoeren van een event. Overweeg eventueel:

- Of je `wijzigStatus()` in elke overgang wil vermelden.
- Of je het weglaat waar overbodig, voor visuele eenvoud.

2. Geen foutverwerking

Er zijn geen paden voor fouten of uitzonderingen, bv. als de verzending mislukt of levering niet bevestigd kan worden. → Wil je robuuste logica modelleren, dan kun je bijvoorbeeld een status `Mislukt` of `Probleem` toevoegen, afhankelijk van je use case.

3. Beëindiging inconsistent

Je hebt een `final` state vanuit `Geannuleerd` én `Afgerond`. Dat is logisch, maar misschien is het zinvol om ook *te labelen* waarom dat het eindpunt is. Bijvoorbeeld via:

```txt
[Bestelling beëindigd]
```

of

```txt
[Proces voltooid]
```

Zo is het duidelijk dat het pad afhangt van het resultaat (succesvol of geannuleerd).

4. UX-opmerking voor implementatie

Als je dit proces in een UI verwerkt, dan zijn dit de momenten waarop je meldingen/triggers toont aan gebruikers. Bijvoorbeeld:

- Na `verzendOrder()` → notificatie: "Uw bestelling is onderweg."
- Na `bevestigLevering()` → "Bestelling is succes afgerond."

✅ Sterke punten:

- **Duidelijke flow**: Het proces is logisch opgebouwd: van `In Behandeling` → `Gecontroleerd` → `Verzonden` → `Afgerond`.
- **Acties zijn betekenisvol**: Gebruikte events zoals `startVerwerking()`, `verzendOrder()` en `bevestigLevering()` zijn herkenbaar en goed benoemd.
- **Alternatieve paden** goed verwerkt:
  - Annulatie (`annuleerBestelling()`)
  - Herstart van validatie (`valideerOpnieuw()`)

🔍 Verbeterpunten of suggesties:

1. Consistentie actie/overgang

De acties zijn correct genoteerd als `event() / actie()`, bijvoorbeeld:

Sterke punten

1. **Structuur klopt perfect**:
Alle scenario's volgen netjes het *Given–When–Then*-patroon. Goed gedaan!

2. **Duidelijke scenario-namen**:
De titels zoals *"Drone gaat van Inactive naar InFlight"* of *"Bestelling wordt ontvangen en afgerond"* zijn helder en beschrijven het doel van de test concreet.

3. **Consistente stijl**:
De zinnen zijn consequent opgebouwd met actieve werkwoorden en verwijzen logisch naar statussen en acties.

4. **Realistische triggers en condities**:
Je gebruikt situaties zoals "low battery", "foutdetectie" of "bestelling ontvangen", die goed aansluiten bij het domein.

3. **Voeg optioneel validatie toe in Then**
Bijvoorbeeld bij:

```gherkin
Then verandert de status van de drone na
```

Je zou kunnen toevoegen:

```gherkin
And het systeem toont een laadstatus aan
```

Zo test je ook systeemrespons, niet alleen statuswijziging.

4. **Extra foutgevallen kunnen nuttig zijn (optioneel)**
Bijvoorbeeld:

- Wat gebeurt er als de drone probeert te starten tijdens onderhoud?
- Of als een bestelling wordt bevestigd zonder onderdelen?

Dat soort "edge cases" versterken je testsuite nog meer.

Eindscore (indien je dit zou inleveren voor feedback)

9/10 – Zeer goed opgesteld! Kleine verbeteringen kunnen het nog sterker maken, zeker qua precisie in de *When*-stappen.

✅ Algemene Sterke Punten

- **Gebruik van Gherkin-syntaxis is correct**: `Given`, `When`, `Then`, `And` – allemaal juist toegepast.
- **Realistische scenario's**: Zowel normale als foutscenario's komen aan bod.
- **Duidelijke koppeling met je toestandsdiagrammen**.
- **Testbare voorwaarden en uitkomsten**: Elk scenario heeft een helder beginpunt en meetbare eindtoestand.

🔧 Verbetersuggesties per onderdeel

📦 Drone-scenario's

1. Scenario: Drone reset na reparatie

> 💡 **Suggestie: Dit scenario bevat twee opeenvolgende acties die wel afhankelijk van elkaar lijken. Overweeg om `repairDone()` te beschrijven als een voorwaarde, en `resetStatus()` als de werkelijke actie:**

gherkin Kopiëren Bewerken

```
Given de drone is in status "Maintenance"
And de technicus heeft "repairDone()" uitgevoerd
When het systeem "resetStatus()" aanroept
Then moet de drone-status veranderen naar "Inactive"
```

2. Scenario: Drone landt en schakelt uit

> 🟡 **De actie `"final"` is wat vaag – overweeg preciezer te formuleren:**

gherkin Kopiëren Bewerken

```
When het afsluitcommando "shutdown()" wordt uitgevoerd
Then moeten alle dronesystemen worden afgesloten
And de status moet worden gemarkeerd als "Final"
```

3. Scenario: Voorkom opstijgen tijdens opladen

> ✅ **Prima edge case! Je zou eventueel ook kunnen toevoegen:**

gherkin Kopiëren Bewerken

```
And er moet een waarschuwing getoond worden aan de piloot
```

📦 Bestelling-scenario's

4. Scenario: Order gecontroleerd en goedgekeurd

> 💡 **De Given zou explicieter kunnen, om verwarring over status te voorkomen:**

gherkin Kopiëren Bewerken

```
Given de order is in status "Gecontroleerd"
```

Service-, Controle- & Boundary-klassen

🔍 Verbeterpunten / Aandachtspunten:

1. Consistency in naamgeving

- Sommige namen gebruiken `Boundary`, andere `Service`, andere eindigen op `Controller`. Dat is goed, maar zorg ervoor dat het altijd op *rol* slaat.
  - Voorbeeld: `VerstuurEvaluatieService` vs. `VerzendEvaluatieService` (consistent werkwoordgebruik).

2. Boundary-klassen zonder duidelijke input/output

- Het is niet altijd duidelijk *welke gegevens* boundary-klassen verwerken of teruggeven.
  - Tip: Overweeg parameters of returntypes kort te noteren bij methodes. Bijv.: `registreerBestelling(gegevens: BestelDTO): Bevestiging`

3. Domeinlogica-check

- Enkele klassen zoals `Structuur` en `Evaluatie` lijken veel verantwoordelijkheid te dragen.
  - Check of *Single Responsibility Principle* behouden blijft: is `Evaluatie` puur een entiteit, of voert ze ook logica uit?

✅ Wat goed zit:

- **Heldere domeinlaag** (blauw gemarkeerd): De kernlogica zit netjes in entiteiten zoals `Kost`, `Evaluatie`, `Drone`, `Structuur`, enz.
- **Boundary-klassen** (bovenaan en onderaan): Mooi dat je verschillende interfaces (zoals `BestelregistratieBoundary`, `RapportBoundary`, enz.) hebt uitgesplitst.
- **Controle-klassen** (rood omlijnd): Zoals `EvaluatieController`, `ChatpuntController`, enz. zorgen voor goede logische tussenlagen.
- **Service-laag** (rechts onderaan): Je toont dat er aparte services bestaan zoals `VerstuurEvaluatieService`, goed voor herbruikbaarheid en logica die buiten de controller hoort.
- **Gebruik van relaties**: Associaties, compositie en generalisaties zijn correct toegepast.

4. UI of externe bronnen ontbreken (optioneel)

- In een volledig layered model zie je soms ook de link naar **externe systemen of gebruikersinterface**. Als dit diagram daarvoor niet bedoeld is, is dat prima, maar in het kader van een totaalplaatje kan dit nuttig zijn.

5. Documentatie & uitleg van kernscenario's

- Tip: voeg (buiten het diagram) korte beschrijvingen toe zoals:
  - "Scenario: evaluatie aanmaken → controller verwerkt invoer → service valideert → domeinobject wordt opgeslagen"
  - Dat helpt anderen om het diagram *verhalend* te begrijpen.

📌 Suggestie voor uitbreiding (optioneel)

Als je dit gebruikt voor documentatie of overdracht, kun je ook:

- Een legenda toevoegen voor kleurgebruik.
- De relatie tussen boundary/service/controller r[illegible]kleur of lijntype verduidelijken (bijv. stippellijn voor afhankelijkheden).

State Transition Testing

✅ Wat goed is:

- **Duidelijke structuur per testgeval**: elk scenario heeft een logische flow: *begintoestand → actie → eindtoestand → verwachte output*.
- **Niet alleen happy paths**: je hebt ook foutscenario's opgenomen zoals netwerkfouten en ongeldige acties (bijv. ST005 en ST006), wat cruciaal is bij statuslogica.
- **Consistente terminologie**: termen zoals "Mechanieker", "In Reparatie", "Gerepareerd", enz. komen overeen met eerdere diagrammen en testcases.
- **Realistische fouten**: systeemcrash en netwerkproblemen zijn nuttige cases voor robuustheidstesten (ST006 en ST007).

🔍 Verbetersuggesties:

1. Test-ID's uitbreiden met betekenis

Optioneel, maar handig: geef testcases een korte beschrijving erbij of gebruik betekenisvolle ID's zoals `ST_DefectToRepair`, `ST_InvalidBackTransition`, enz.

2. ST002: Dubbels in kolommen

In je tweede test staat zowel bij "Begintoestand" als "Actie" "**Reparatie geslaagd**". Waarschijnlijk bedoel je bij *actie*: `"Reparatie wordt afgerond"`, of `"Reparatie geslaagd gemeld"`.

3. ST005 – Ongeldige overgang

Overweeg hier te noteren:

- **Welke actie werd geprobeerd?** Bijvoorbeeld: `"Mechanieker probeert 'setStatus(Defect)'"`
- **Hoe detecteert het systeem dit?** Eventueel via foutcode of validatiemechanisme

4. ST006 – Netwerkfout verduidelijken

Je zou hier ook kunnen specificeren:

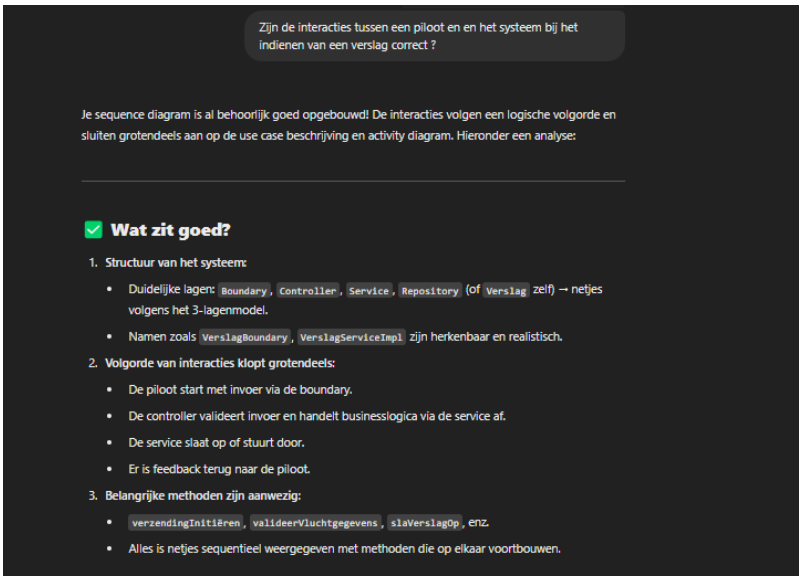
- Of de actie werd herhaald
- Of het systeem automatisch herstelt of een reti…nbiedt

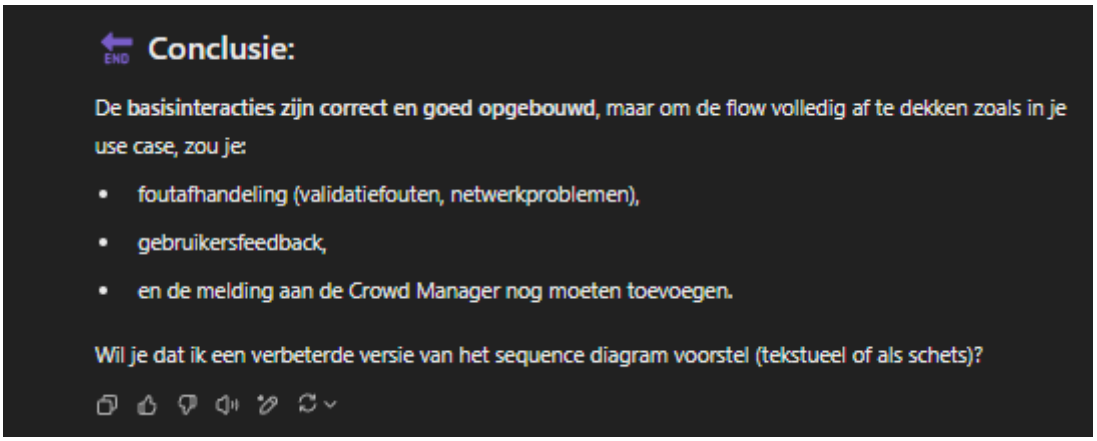
📌 Suggestie: Visualiseer als overgangstabel

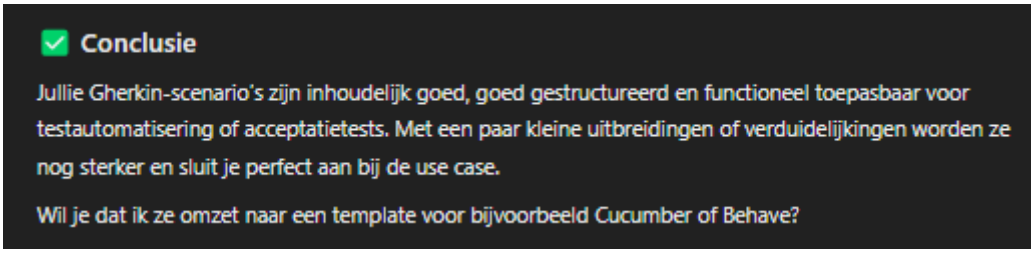
Je zou dit overzicht perfect kunnen omzetten naar een **transition table** voor testspecificatie.
Bijvoorbeeld:

Huidige Status	Actie	Verwachte Nieuwe Status	Verwachte Output
Defect	Start reparatie	In Reparatie	Drone gaat naar 'In Reparatie'
Gerepareerd	Reset status	Klaar voor gebruik	Drone is klaar om ingezet te worden
Gerepareerd	Forceer status naar Defect	Ongeldige overgang	Foutmelding: Overgang niet toegestaan
...	...	...	...

DEEL 4


Zijn de interacties tussen een piloot en en het systeem bij het indienen van een verslag correct ?
Je sequence diagram is al behoorlijk goed opgebouwd! De interacties volgen een logische volgorde en sluiten grotendeels aan op de use case beschrijving en activity diagram. Hieronder een analyse:
Wat zit goed?
1. Structuur van het systeem:
• Duidelijke lagen: Boundary, Controller, Service, Repository (of Verslag zelf) → netjes volgens het 3-lagenmodel.
• Namen zoals VerslagBoundary, VerslagServiceImpl zijn herkenbaar en realistisch.
2. Volgorde van interacties klopt grotendeels:
• De piloot start met invoer via de boundary.
• De controller valideert invoer en handelt businesslogica via de service af.
• De service slaat op of stuurt door.
• Er is feedback terug naar de piloot.
3. Belangrijke methoden zijn aanwezig:
• verzendingInitiëren, valideerVluchtgegevens, slaVerslagOp, enz.
• Alles is netjes sequentieel weergegeven met methoden die op elkaar voortbouwen.


Conclusie:
De basisinteracties zijn correct en goed opgebouwd, maar om de flow volledig af te dekken zoals in je use case, zou je:
• foutafhandeling (validatiefouten, netwerkproblemen),
• gebruikersfeedback,
• en de melding aan de Crowd Manager nog moeten toevoegen.
Wil je dat ik een verbeterde versie van het sequence diagram voorstel (tekstueel of als schets)?


Conclusie
Jullie Gherkin-scenario's zijn inhoudelijk goed, goed gestructureerd en functioneel toepasbaar voor testautomatisering of acceptatietests. Met een paar kleine uitbreidingen of verduidelijkingen worden ze nog sterker en sluit je perfect aan bij de use case.
Wil je dat ik ze omzet naar een template voor bijvoorbeeld Cucumber of Behave?

⭐ Bonus: Interface toevoegen (optioneel)

IDroneService.java

java Copier

```java
package be.odisee.citymesh;

public interface IDroneService {
    void startCharging(Drone drone);
    void startDroneVlucht(Drone drone);
    void reserveerDrone(Drone drone);
    void resetDroneStatus(Drone drone);
    String verwerkDroneLocatie(Drone dro
}
```

DroneService.java

java Copier

```java
@Service
public class DroneService implements IDr
    // bestaande implementatie
}
```

DroneBoundry.java

java Copier

```java
private final IDro ervice droneService
@Autowired
public DroneBoundry(IDroneService droneS
```

✅ Wat je goed hebt gedaan

Element	Feedback
Annotaties	Je gebruikt correct `@Service` voor `DroneService` en `@Component` voor `DroneBoundry`.
Injectie	Je gebruikt **constructor-based injectie** met `@Autowired`, wat de **voorkeurstechniek** is in Spring vanwege betere testbaarheid en immutability.
Component Scan	Via `@ComponentScan("be.odisee.citymesh")` in `Application.java` laat je Spring weten waar het beans moet zoeken. Perfect!
Bean Discovery	Je vertrouwt op **impliciete bean discovery**, precies zoals gevraagd in de opdracht (zie slide 43 van de cursus).
Testmethode	De `testService()` in `DroneBoundry` toont aan dat de injectie werkt — goed als bewijs en debuggingstap.

⚠️ Suggesties voor verbetering

Verbeterpunt	Uitleg
Gebruik van interface	Maak een interface zoals `IDroneService`, en laat `DroneService` deze implementeren. Zo pas je ook het **Dependency Inversion Principle** (DIP) toe uit de cursus (slide 27).
Betere naamgeving Boundry	In moderne conventies zou je `DroneController` of `DroneHandler` gebruiken i.p.v. `Boundry` (tenzij dit bewust zo is gekozen binnen het project).
Unit test maken	Maak een eenvoudige test met een `@SpringBootTest` of `AnnotationConfigApplicationContext` om te tonen dat de injectie effectief werkt — dat wordt meestal verwacht in een demo van DI.

Bijlage Java code

Bestelling.java

```
package be.odisee.citymesh;
import java.util.List;
import java.util.Date;

public class Bestelling {
    private int bestellingID;
    private Date datum;
    private String status;
    private List<Onderdeel> onderdelen;

    public Bestelling(int bestellingID, Date datum, String status,
List<Onderdeel> onderdelen) {
        this.bestellingID = bestellingID;
        this.datum = datum;
        this.status = status;
        this.onderdelen = onderdelen;
    }

    public void wijzigingStatus(String nieuwe_status) {
        this.status = nieuwe_status;
    }

    public int getBestellingID() {
        return bestellingID;
    }

    public void setBestellingID(int bestellingID) {
        this.bestellingID = bestellingID;
    }

    public Date getDatum() {
        return datum;
    }

    public void setDatum(Date datum) {
        this.datum = datum;
    }

    public String getStatus() {
        return status;
    }
```

```
    public void setStatus(String status) {
        this.status = status;
    }

    public List<Onderdeel> getOnderdelen() {
        return onderdelen;
    }

    public void setOnderdelen(List<Onderdeel> onderdelen) {
        this.onderdelen = onderdelen;
    }
}
```

Controller.java

```
package be.odisee.citymesh;

public class Controller {

}
```

Drone.java

```
package be.odisee.citymesh;

public class Drone {
    public enum DroneStatus{
        INACTIVE,
        INFLIGHT,
        CHARGING,
        RESERVED,
        DEFECT,
        MAINTENANCE
    }
    private int droneID;
    private DroneStatus status;
    private String locatie;
    private String type;

    public Drone(int droneID, DroneStatus status, String locatie, String type)
{
        this.droneID = droneID;
        this.status = status;
```

```
        this.locatie = locatie;
        this.type = type;
    }

    public void resetStatus() {
        this.status = DroneStatus.INACTIVE;
    }

    public int getDroneID() {
        return droneID;
    }

    public void setDroneID(int droneID) {
        this.droneID = droneID;
    }

    public DroneStatus getStatus() {
        return status;
    }

    public void setStatus(DroneStatus status) {
        this.status = status;
    }

    public String getLocatie() {
        return locatie;
    }

    public void setLocatie(String locatie) {
        this.locatie = locatie;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

DroneService.java

```
package be.odisee.citymesh;
```

```java
import java.util.Random;

public class DroneService {

    public void startCharging(Drone drone){
        if (drone.getStatus() == Drone.DroneStatus.INACTIVE ||
drone.getStatus() == Drone.DroneStatus.INFLIGHT){
            drone.setStatus(Drone.DroneStatus.CHARGING);
        }
    }

    public void startDroneVlucht(Drone drone){
        if (drone.getStatus() == Drone.DroneStatus.RESERVED){
            drone.setStatus(Drone.DroneStatus.INFLIGHT);
        }
    }

    public void reserveerDrone(Drone drone){
        if (drone.getStatus() == Drone.DroneStatus.INACTIVE ||
drone.getStatus() == Drone.DroneStatus.DEFECT){
            drone.setStatus(Drone.DroneStatus.RESERVED);
        }
    }

    public void resetDroneStatus(Drone drone){
        drone.resetStatus();
    }

    public String verwerkDroneLocatie(Drone drone){
        if (drone.getStatus() == Drone.DroneStatus.INFLIGHT){
            return drone.getLocatie();
        }
        else{
            return null;
        }
    }

    public void startReparatie(Drone drone, Melding melding){
        if (melding.getStatus() == Melding.MeldingStatus.BATTERY_LOW){
            startCharging(drone);
        }
        else {
            drone.setStatus(Drone.DroneStatus.MAINTENANCE);
        }
    }
```

```
    public int[][] genereerMatrix(int size){
        int[][] matrix = new int[size][size];
        Random random = new Random();
        int centre = size/2;

        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                // Bereken de afstand tot het midden
                double distance = Math.sqrt(Math.pow(i - centre, 2) +
Math.pow(j - centre, 2));
                // Hoe dichter bij het midden, hoe hoger het basisgetal
                int baseValue = (int) ((size - distance) * 10) +
random.nextInt(10);
                matrix[i][j] = Math.max(baseValue, 0); // Zorgt ervoor dat de
waarde niet negatief wordt
            }
        }
        return matrix;
    }
}
```

Melding.java

```
package be.odisee.citymesh;

import java.util.Date;
import java.util.ArrayList;
import java.util.List;

public class Melding {

    public enum MeldingStatus {
        BATTERY_LOW,
        MAINTENANCE_REQUIRED,
        OVERCROWDING
    }

    private int meldingID;
    private String beschrijving;
    private Date tijdstip;
    private MeldingStatus status;
```

```
    public Melding(int meldingID, String beschrijving, Date tijdstip,
MeldingStatus status) {
        this.meldingID = meldingID;
        this.beschrijving = beschrijving;
        this.tijdstip = tijdstip;
        this.status = status;
    }

    public int getMeldingID() {
        return meldingID;
    }

    public void setMeldingID(int meldingID) {
        this.meldingID = meldingID;
    }

    public String getBeschrijving() {
        return beschrijving;
    }

    public void setBeschrijving(String beschrijving) {
        this.beschrijving = beschrijving;
    }

    public Date getTijdstip() {
        return tijdstip;
    }

    public void setTijdstip(Date tijdstip) {
        this.tijdstip = tijdstip;
    }

    public MeldingStatus getStatus() {
        return status;
    }

    public void setStatus(MeldingStatus status) {
        this.status = status;
    }
}
```

MeldingService.java

```
package be.odisee.citymesh;
```

```java
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class MeldingService {

    public static List<Melding> generateOvercrowdingReports(int[][] matrix,
int limit) {
        List<Melding> meldingen = new ArrayList<>();
        int meldingIDCounter = 1;
        Date now = new Date();

        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                if (matrix[i][j] > limit) {
                    String beschrijving = "Te veel mensen op locatie (" + i +
", " + j + "): " + matrix[i][j] + " personen.";
                    meldingen.add(new Melding(meldingIDCounter++,
beschrijving, now, Melding.MeldingStatus.OVERCROWDING));
                }
            }
        }
        return meldingen;
    }

    public String verwerkMelding(Melding melding){
        if (melding != null && !melding.getBeschrijving().isEmpty()){
            return String.format("%s gemeld op %s", melding.getBeschrijving(),
melding.getTijdstip());
        }
        else return "Melding is leeg";
    }
}
```

Onderdeel.java

```java
package be.odisee.citymesh;

public class Onderdeel {
    private static int OnderdeelId;
    private String Naam;
    private int vooraad;
```

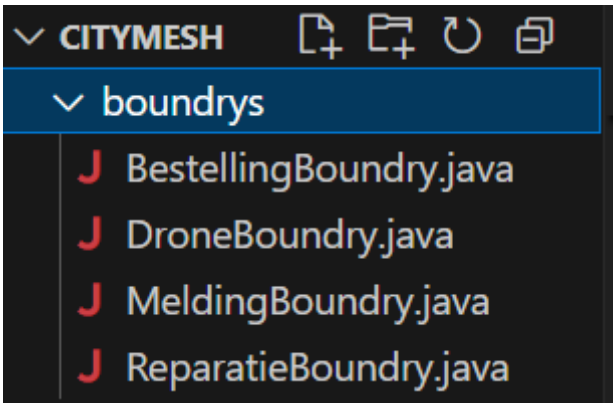
```
    public Onderdeel(String Naam, int vooraad) {
        this.Naam = Naam;
        this.vooraad = vooraad;
        this.OnderdeelId = OnderdeelId + 1;
    }

    public void wijzigVooraad(int wijziging) {
        vooraad = vooraad + wijziging;
    }
}
```

Reparatie.java

```
package be.odisee.citymesh;

import java.util.Date;

public class Reparatie {
    private int reparatieID;
    private Date datum;
    private String status;

    public Reparatie(int reparatieID, Date datum, String status) {
        this.reparatieID = reparatieID;
        this.datum = datum;
        this.status = status;
    }

    public void startReparatie(Drone drone) {
        drone.resetStatus();
        status = "Gerepareerd";
    }

    public int getReparatieID() {
        return reparatieID;
    }

    public void setReparatieID(int reparatieID) {
        this.reparatieID = reparatieID;
    }

    public Date getDatum() {
        return datum;
    }
```

```java
    public void setDatum(Date datum) {
        this.datum = datum;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

Map boundary's:



Voorbeeld: BestellingBoundry.java

```java
package be.odisee.citymesh.boundrys;

public class BestellingBoundry {
}
```