Software Design en Documentatie

1. UML Diagrammen

1.1 Inleiding

Het is inmiddels geen geheim meer dat grote ICT-projecten, en dan met name bij de overheid, in veel gevallen mislukken. Zo zien we vaak krantenkoppen voorbijkomen als "Nederland miljarden kwijt door falend ict" en "Rijksoverheid leert nog steeds niet van mislukte ICT-projecten". In de praktijk blijkt zelfs dat meer dan een derde van alle ICT-projecten bij de overheid dusdanig mislukt dat het hele systeem wordt geschrapt!

> Meer dan een derde van alle softwareprojecten bij de overheid mislukt volledig

Je kunt je dus voorstellen dat het ontwikkelen van een succesvol, grootschalig softwareproject bijzonder ingewikkeld is. Het is echter een valkuil waar we telkens opnieuw in trappen. Veel softwareprojecten lopen spaak door een gebrek aan goede communicatie tussen stakeholders, slechte technische architectuur, complexiteit die onderschat wordt en budgetoverschreidingen waar je op z'n zachts gezegd zwetende handpalmen van krijgt.

Het ontwerpen van een gedegen software-architectuur vóórdat het systeem gebouwd zal worden, is dus ontzettend belangrijk. Door vooraf tot in de details na te denken over de betrouwbaarheid, schaalbaarheid en de manier waarop de verschillende onderdelen van de applicatie met elkaar samenwerken, kunnen we de complexiteit beter inschatten. Bovendien hebben we een universele manier nodig om te kunnen communiceren over de manier waarop de software gebouwd moet worden. Met elkaar, als developers, maar ook met de stakeholders van het project. Hiermee kunnen we ervoor zorgen dat zij bij oplevering niet zeggen: "Oh.... Maar dit is helemaal niet wat we bedoelden!".

Maar ook als ons product het zonder kleerscheuren tot de lanceringsdatum haalt, hebben we documentatie nodig om uit te kunnen leggen hoe de software in elkaar zit. En, net zo belangrijk: *waarom* deze keuzes gemaakt zijn. Niet alleen voor nu, maar ook voor de developers die in de toekomst aan dit project zullen

werken. Dus hoewel documentatie geen sexy reputatie heeft, is het een essentieel onderdeel van jouw workflow als developer.

Om te zorgen dat iedereen dit op dezelfde manier communiceert, zijn er enkele standaarden. In deze cursus zullen we gebruik gaan maken van UML: **Unified Modeling Language**. Dit zorgt ervoor dat we op een visuele manier uit kunnen drukken hoe een systeem werkt. Het gebruikt symbolen (zoals vierkanten en lijnen) om diagrammen te maken die relaties tussen verschillende onderdelen van het systeem weergeven.

Tot welke mate van detail je dit moet uitdenken, hangt erg af van het project. De belangrijkste maatstaaf bij jouw eigen toekomstige projecten, is dat je details toevoegt tot je het probleem volledig begrijpt. Hoewel het verleidelijk is om direct te beginnen met programmeren, zorgt deze aanpak er namelijk voor dat je in een vroeg stadium al mogelijke problemen ontdekt. Je forceert jezelf namelijk om eerst op een abstracte manier naar het probleem te kijken vóórdat je diep in de code zit.

In deze cursus leer je gebruik te maken van klassendiagrammen en sequentiediagrammen. **Klassediagrammen** helpen om de verschillende objecten (classes) uit te tekenen en onderlinge relaties te bepalen. Sequentiediagrammen gebruiken we om de volgorde te specificeren waarin verschillende onderdelen van de applicatie op elkaar inwerken.

Dus, ben je klaar om te leren documenteren?

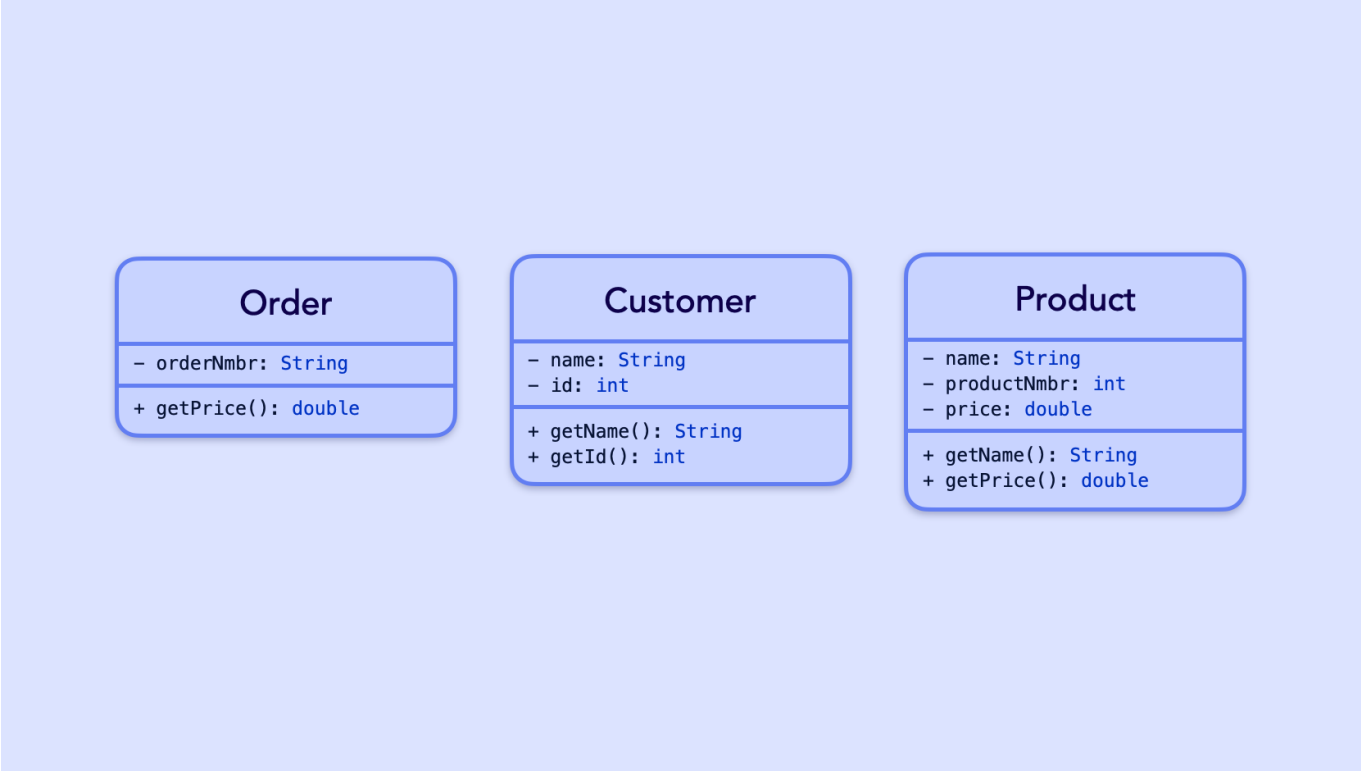
1.2 Klassendiagram: afhankelijkheid

Een klassendiagram is een UML diagram dat de structuur van een applicatie weergeeft door de relaties tussen classes te illustreren. Klassediagrammen worden vaak gebruikt voor de statische elementen van een systeem, zoals het type van de classes, diens properties, methoden en de onderlinge relaties.

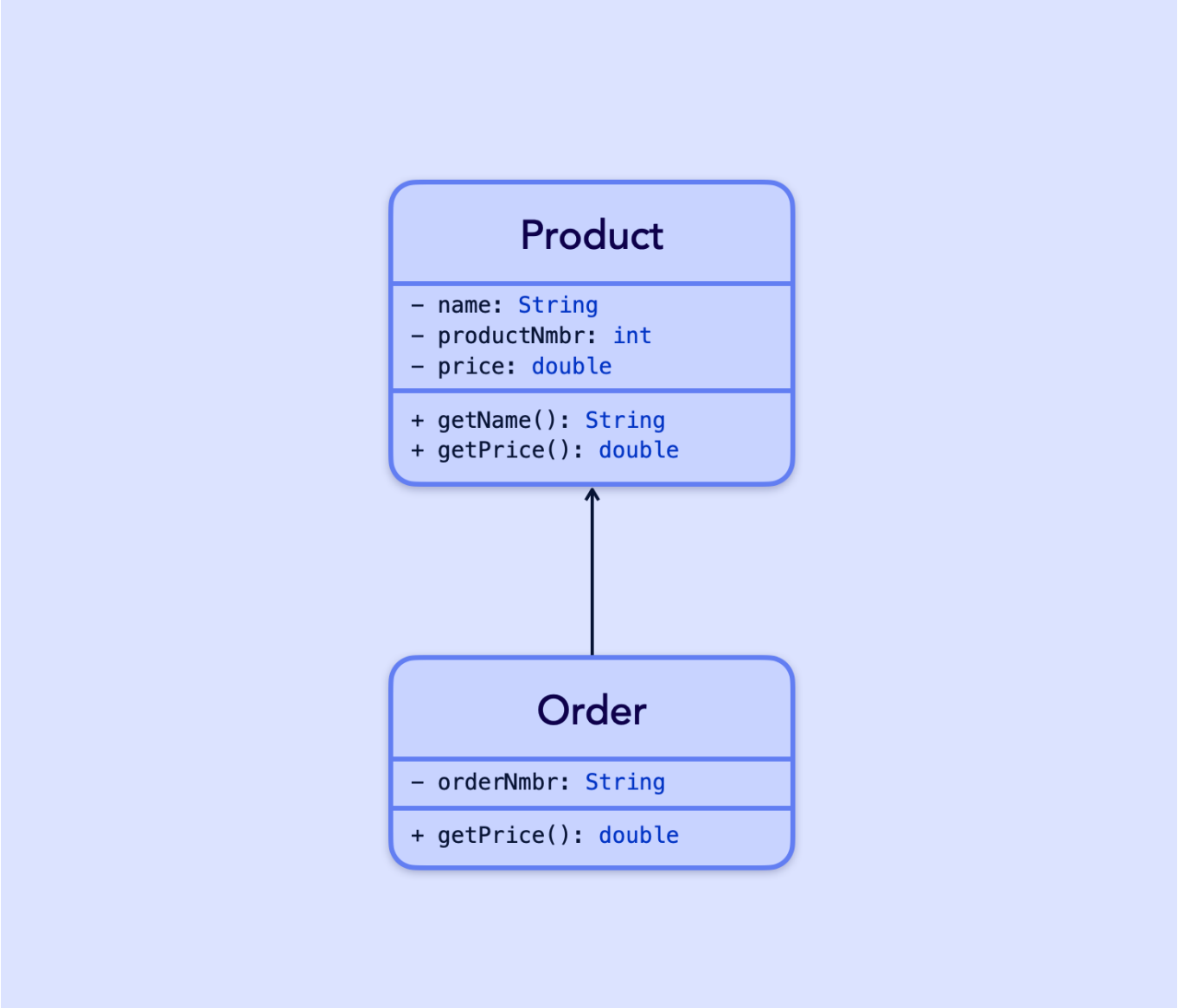
Laten we een klassendiagram gaan modelleren voor een simpele webshop. Wanneer we een applicatie willen maken die zich gedraagd als een webshop, moeten we gaan nadenken over de **entiteiten** die daarbij horen. Een entiteit is een representatie van een generiek stukje data. Denk bij voorbeeld aan een *product*: hier zal de webshop er waarschijnlijk tientallen of honderden van hebben. Bij een aankoop verzamelen we één of meerdere producten in een *bestelling*. De bestelling wordt altijd gedaan door een gebruiker met een naam, adres en account, in dit geval de *klant*. Wanneer we dit vertalen naar classes in onze applicatie, zouden we hier waarschijnlijk een Customer-class, Product-class en Order-class voor maken. Een class wordt in een klassendiagram altijd gerepresenteerd in drie vlakken, één voor de naam, één voor de properties en één voor de methoden. De simpelste versie van onze Customer-class zou er zo uit kunnen zien:

Omdat een klassendiagram gaat over *relaties*, benoem je in dit diagram meestal alleen de methoden en properties van een class die beschikbaar zijn voor andere classes. Dit betekent: properties en methoden die zijn voorzien van de access modifier `public` . Mocht je het toch fijner vinden om alle informatie weer te geven - zoals wij hierboven hebben gedaan - dan kun je zichtbaarheid aanduiden met een `-` teken (private), een +-teken (public) of een #-teken (protected). Je kunt zelfs aangeven dat iets package-private is met ~, maar dit zie je eigenlijk bijna nooit.

Laten we onze Order-class en Product-class hier ook alvast aan toevoegen:



Op dit moment lijkt het alsof onze classes niets met elkaar te maken hebben, maar dat klopt natuurlijk niet! In veel gevallen hebben classen een onderlinge relatie met elkaar of zijn ze van elkaar **afhankelijk**. Dit noemen we binnen klassendiagrammen ook wel eens **associaties**. We spreken al van een afhankelijkheid wanneer class A een property heeft van het type class B. In onze webshop is dit ook zo: een *Order* zal altijd minimaal één *Product* moeten bevatten. Dit kunnen we in een klassendiagram weergeven door er een lijn tussen te tekenen. De *richting* van de afhankelijkheid geven we aan met een zwarte pijl:



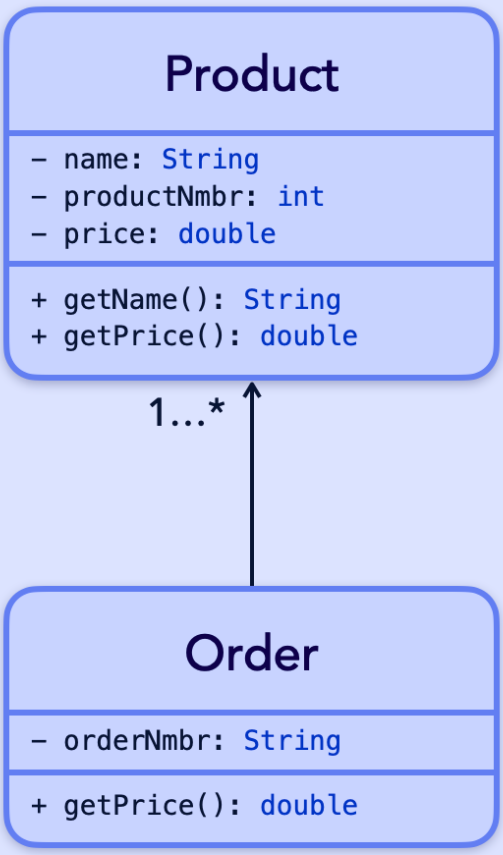
De Order-class is dus afhankelijk van de Product-class

Hierdoor zie je dat de Order-class afhankelijk is van de Product-class, maar niet andersom. Dit noem je een **unidirectionele associatie**. Ofwel: de Order-class is zich bewust van de Product-class, maar de Product-Class is hoeft zich niet bewust te zijn van de Order-class. Een product kan bestaan zonder bestelling, maar een bestelling zonder product is vrij nuttenloos (en dat mag in dit geval dus niet).

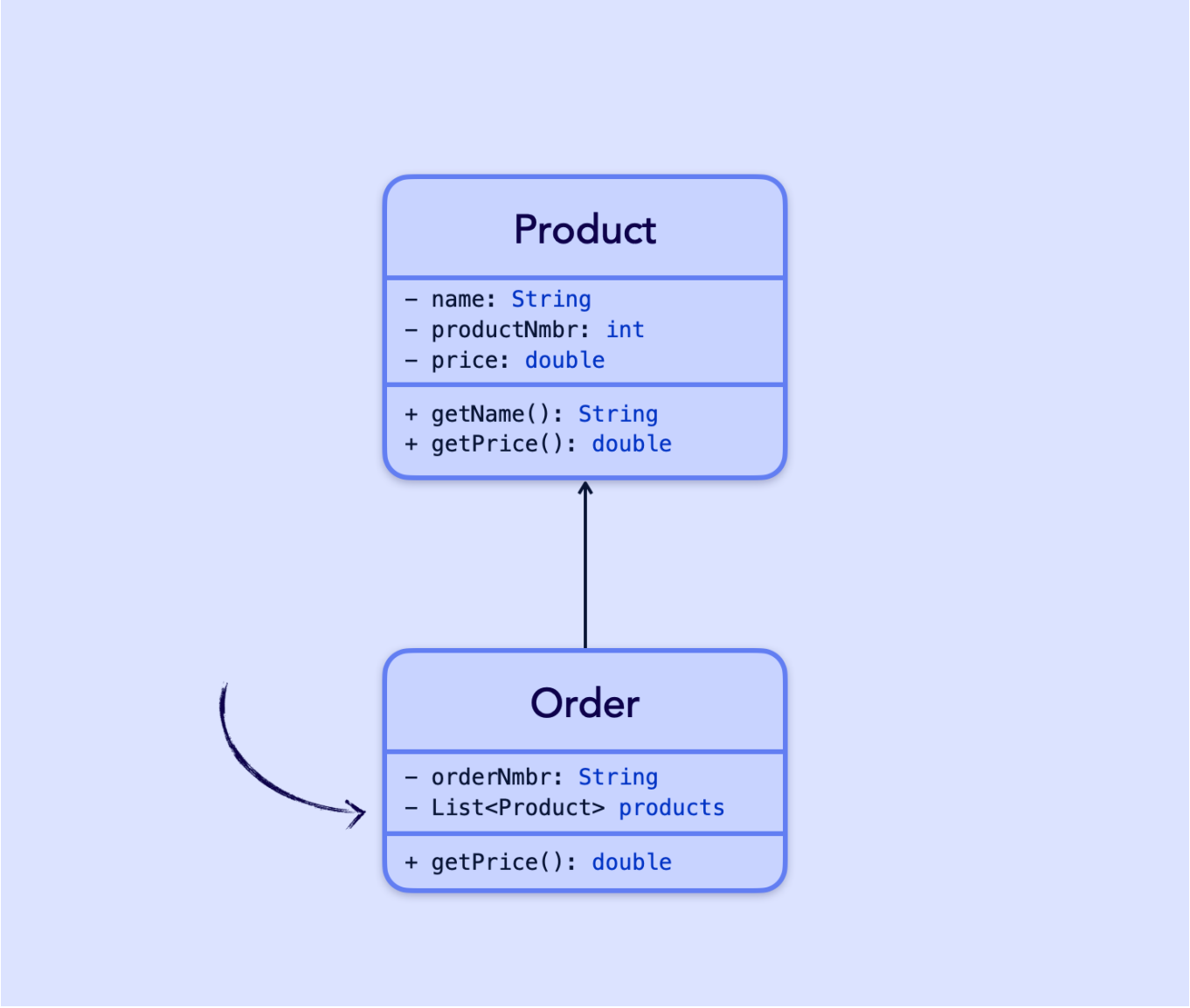
In de huidige weergave missen we echter nog een belangrijk detail: deze relatie heeft de bijzondere eigenschap dat er meerdere producten aan één bestelling kunnen hangen. Dit kunnen we op twee manieren modelleren. De eerste manier is door **kardinaliteiten** toe te voegen aan onze relatieaanduiding. Hierin hebben we de volgende opties:

- 0..1 (optioneel)
- 0..* (0 of meerdere)
- 1..* (minimaal één of meerdere)

- n (vaste waarde, waarbij n het getal is)

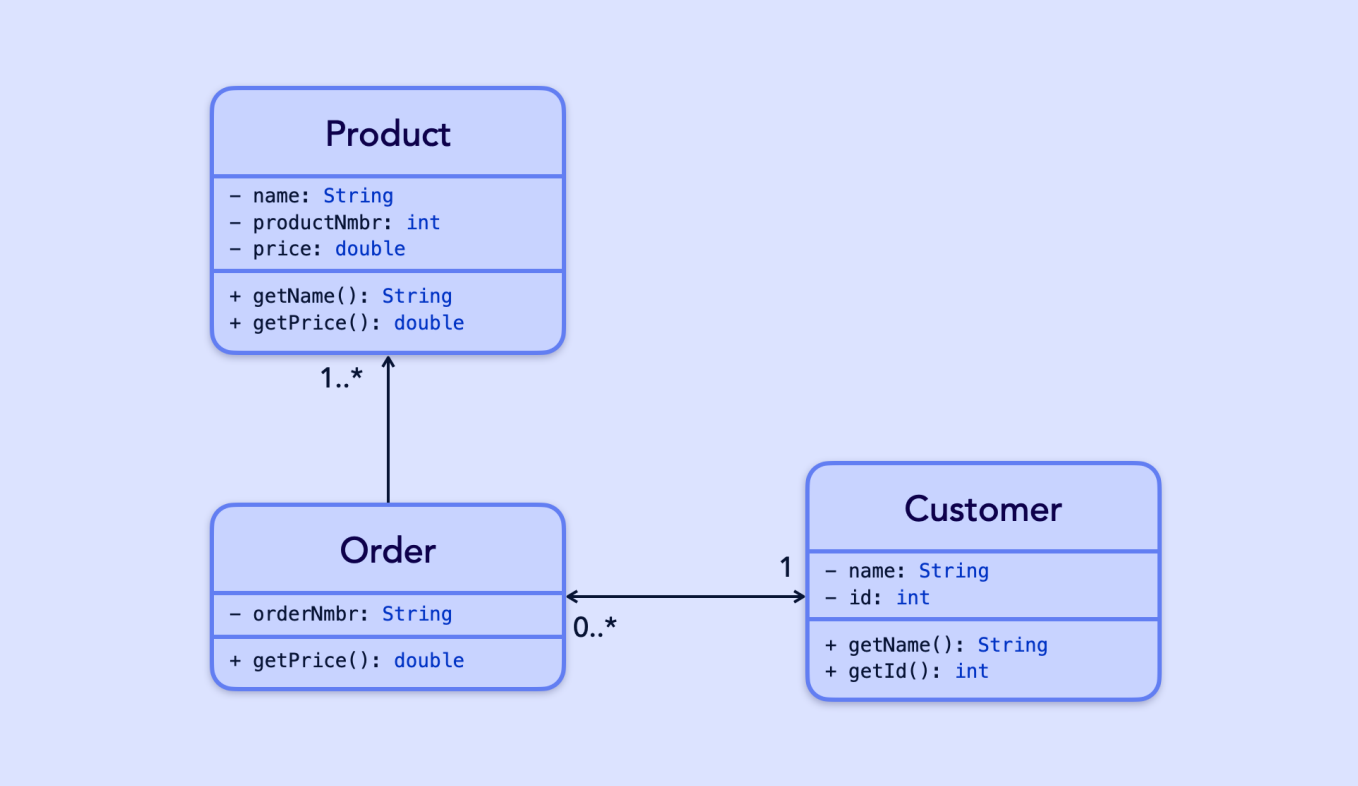


Met deze aanduiding zeggen we: de Order-class heeft altijd minimaal één *of* meerdere Product-instanties. Hierbij verklappen we echter niets over de implementatie van deze relatie. De tweede optie is door het toevoegen van implementatie-details. Gezien de Order-class altijd een lijst van Product-classes zal bevatten, kunnen we dat aangeven door deze property toe te voegen:



Welke van de twee je gebruikt, hangt in de praktijk af van waar je bent in het ontwerpproces. Ben je op hoog niveau bezig of een simpele schets aan het maken? Dan is het aan te raden om gebruik te maken van een meer algemene aanpak met kardinaliteiten. Ben je al verder in het ontwerpproces en ga je jouw implementatie communiceren met andere developers, is het aan te raden de implementatie-details hierin te verwerken.

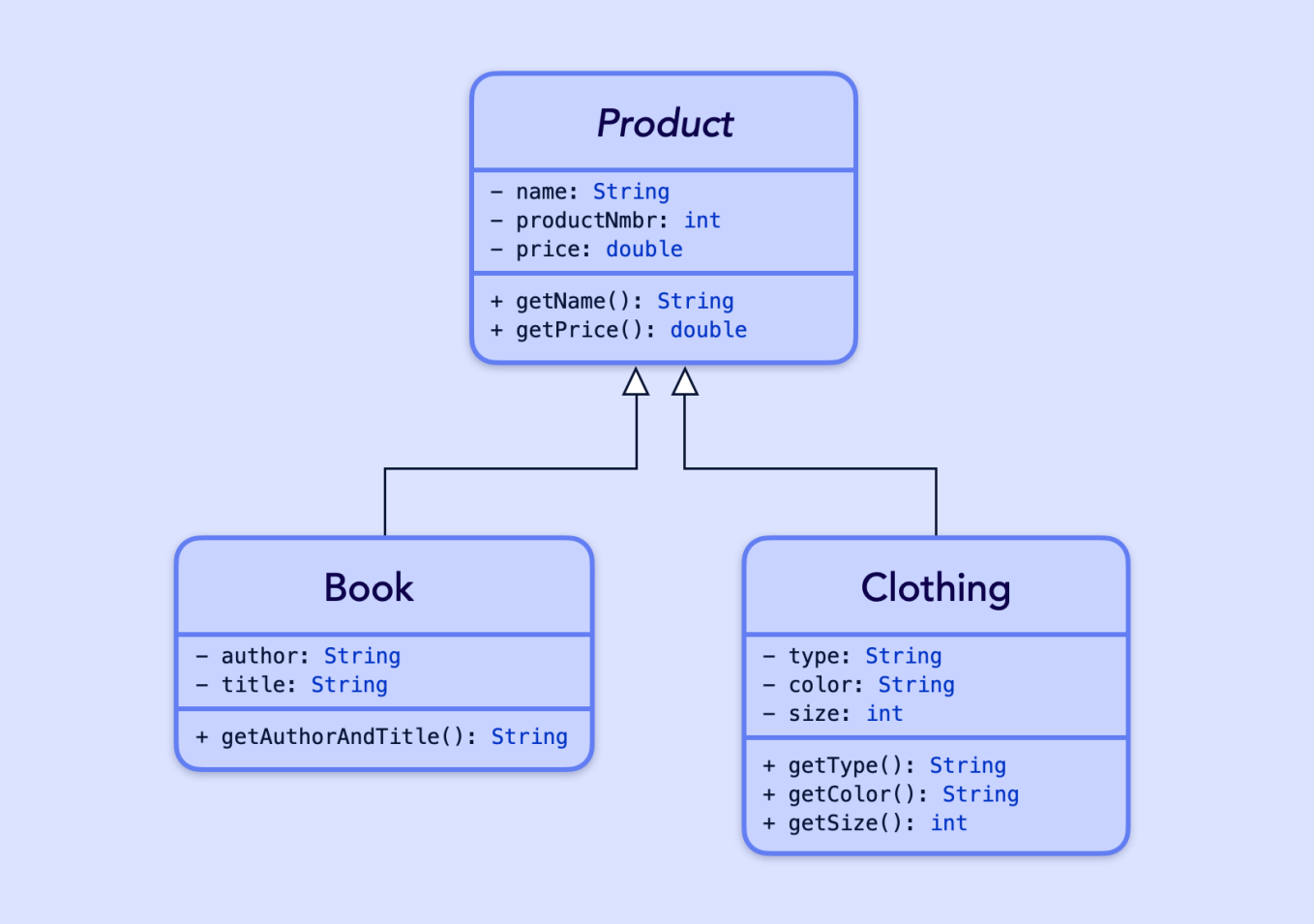
We hebben echter nog een andere afhankelijkheid. Een *bestelling* zal altijd gelinkt moeten zijn aan een *klant*: de Order-class is afhankelijk van de Customer-class. Andersom is dit echter ook het geval, want een klant zal ongetwijfeld zijn bestelling(en) willen inzien. Beide classes moeten zich dus bewust zijn van elkaar: we spreken hier van een **bidirectionele associatie**.



Je kunt er in dit voorbeeld voor kiezen om de pijl twee kanten op te laten wijzen, of aan beide kanten weg te laten. Het resultaat is hetzelfde. Door de kardinaliteiten kun je zien dat een bestelling altijd één klant heeft, maar een klant 0 of meerdere bestellingen kan hebben. Een klant kan immers een account aanmaken zonder ooit iets te kopen in onze webshop.

1.3 Klassendiagram: overerving

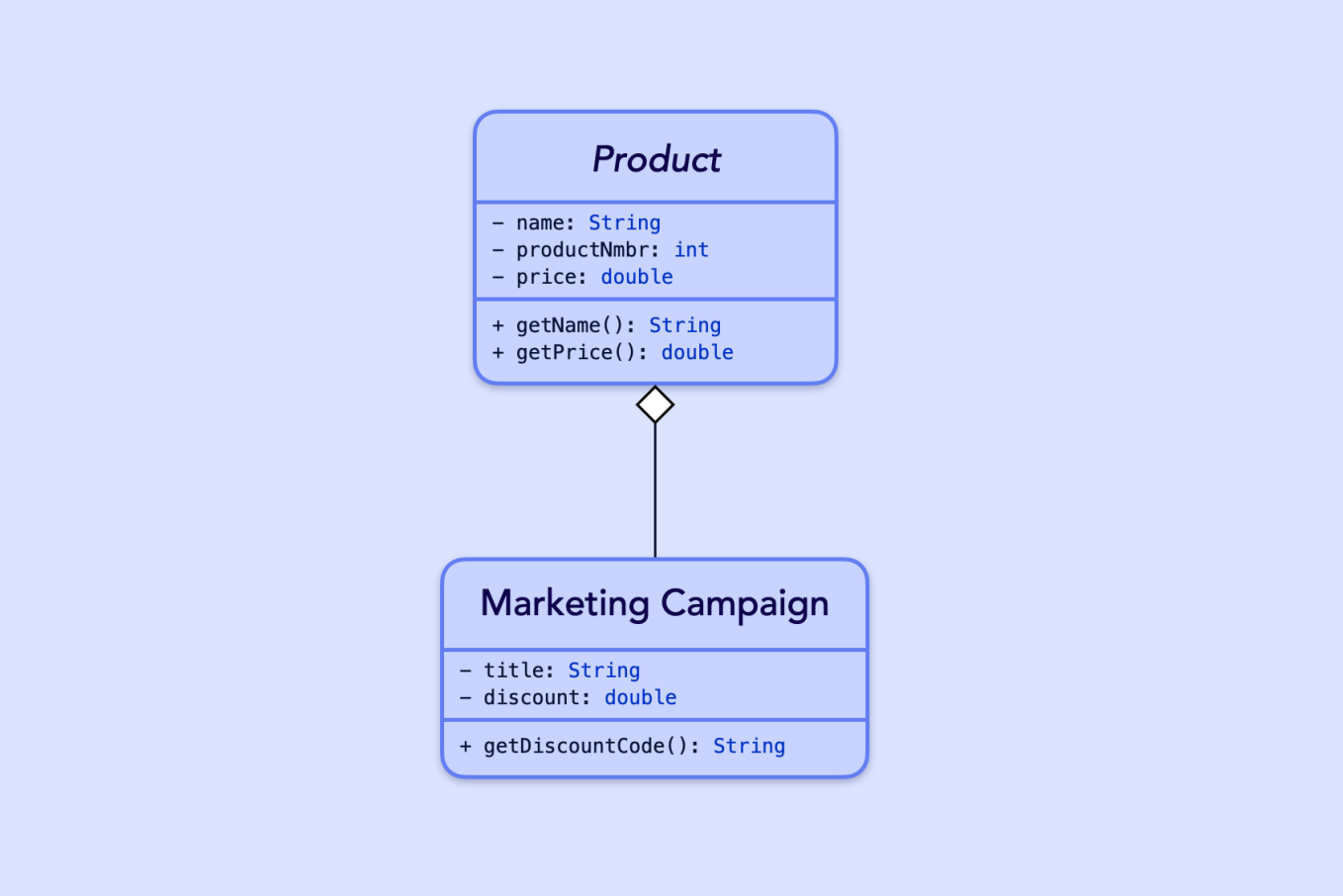
Je hebt inmiddels geleerd dat we vaak gebruik maken van classes die eigenschappen *erven* van een superclass. Ook deze informatie is belangrijk om te verwerken in ons klassendiagram. Zo kunnen we de producten in onze webshop onderverdelen in *boeken* en *kleding*, wat beiden extenties zijn van de Product-class. Dit geven we aan doormiddel van de witte pijlen die van de subclass naar de superclass wijzen:



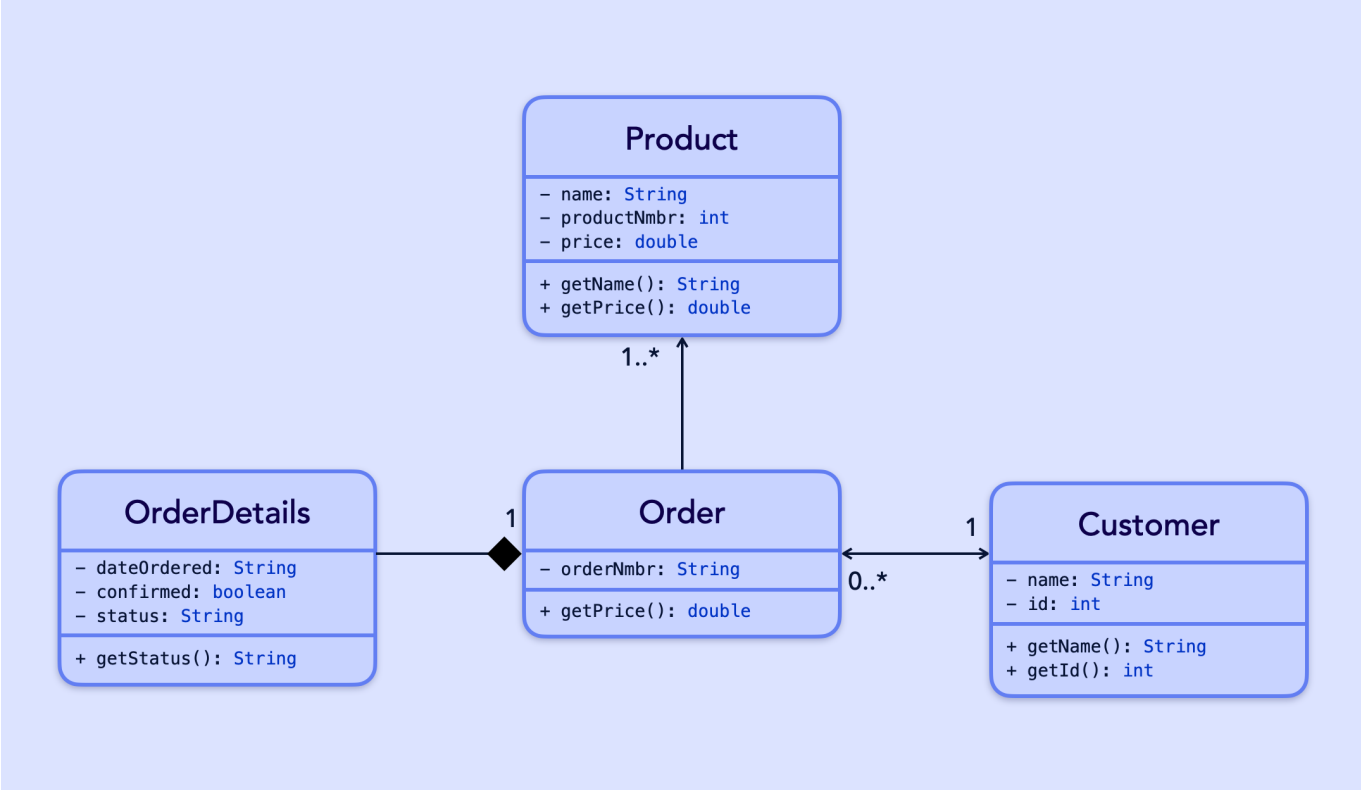
Dit type pijlen geven dus geen afhankelijkheid aan, maar overerving. In dit geval kun je door het maken van deze diagrammen al inschatten dat het niet logisch is om een instantie te mogen maken van de Product-class. Daarom kunnen we er beter een abstracte class van maken. Dit duiden we aan door de naam schuingedrukt te maken.

1.4 Klassendiagram: aggregatie en compositie

Het laatste concept wat je zou kunnen toevoegen aan je klassendiagram - maar niet verplicht is voor de eindopdracht - zijn aggregatie en compositie. Bij **aggregatie** geef je aan dat class A onderdeel *kan* zijn van class B, maar dat hoeft niet. Denk bijvoorbeeld aan een marketing campagne die invloed heeft op de prijs van sommige producten in onze webshop. Een MarketingCampain-class kan onderdeel zijn van de Product-class, maar kan ook los van de Product-class bestaan. Wanneer de sokken - met korting! - verwijderd worden, betekent dit immers niet dat de complete kortingsactie niet meer mag bestaan.



Bij **compositie** is class A een vast onderdeel van class B en kan class A niet bestaan *zonder* deze class. Dit betekent dat wanneer de instantie van Class A verwijderd wordt, Class B automatisch ook de prullenbak in gaat. Denk bijvoorbeeld aan de gegevens over de status van de order. De OrderDetails-class is een vast onderdeel van de Order-class. Wanneer een Order-instantie verwijderd wordt, bijvoorbeeld omdat Karin besluit dat streepjessokken toch niks voor haar zijn, kunnen OrderDetails ook niet meer bestaan.

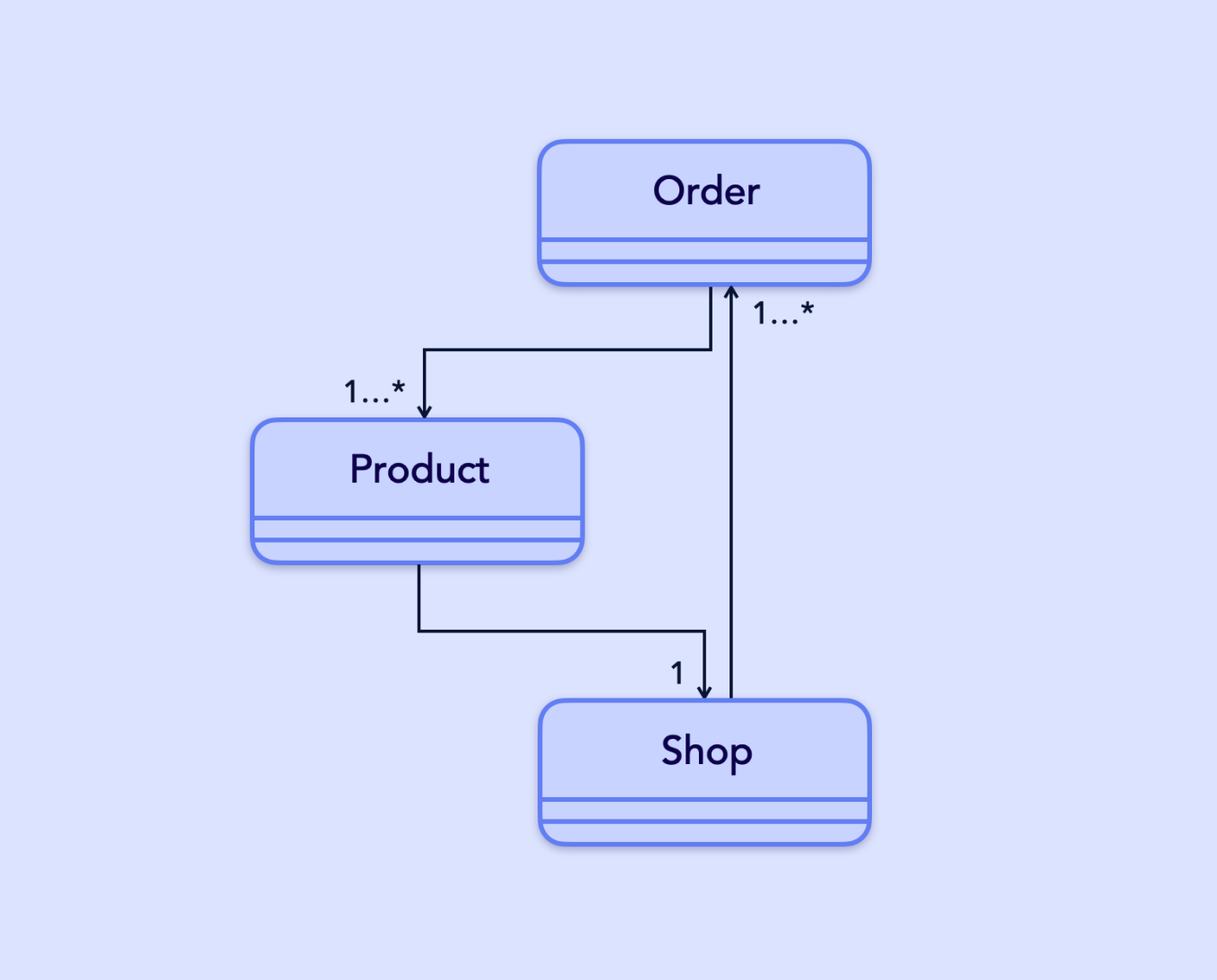


Wellicht vraag je je nu af: als de OrderDetails-class zo'n belangrijk onderdeel is van de Order-class, waarom is het dan überhaupt een aparte class?

Uitstekende vraag! Het opsplitsen van onze classes is onderdeel van een principe dat we **Clean Coding** noemen. Hoewel we hier later in deze cursus dieper op ingaan, kunnen we je alvast vertellen dat Clean Coding een best practise is als het gaat om de opbouw van onze code. Eén van de belangrijkste regels is dat we methoden en classes zo simpel, klein en eenvoudig mogelijk willen houden. Zodra één class meerdere verschillende taken aan het vervullen is, is het tijd om te kijken of we die verantwoordelijkheden niet beter kunnen splitsen.

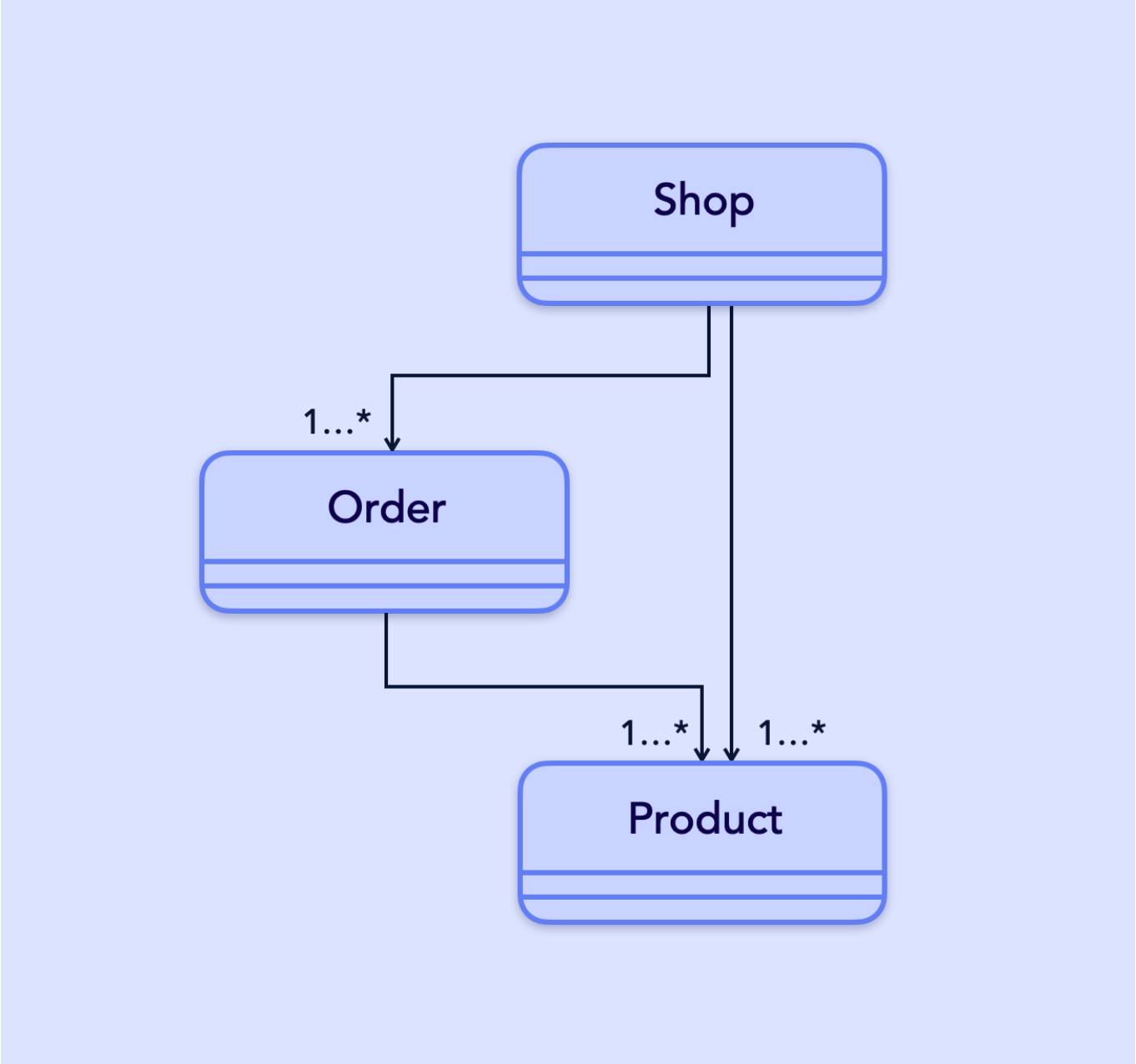
Fouten in de oplossing vinden

Je afhankelijkheden modelleren helpt erg bij de visualisatie van een oplossing. Bovendien kan het helpen bij het vinden van mogelijke fouten in het model. Laten we bijvoorbeeld eens naar het volgende diagram kijken:



De Product-class is afhankelijk van de Shop-class. De Shop-class is afhankelijk van de Order-class en de Order-class is afhankelijk van de Product-class. Dit geeft aan dat we een shop nodig hebben om een product te kunnen maken, een order nodig hebben om een shop te kunnen maken en een product nodig hebben om een order te kunnen maken. *Houston, we have problem.*

In dit geval wordt het onmogelijk om een instantie van één van de classes te maken... Terug naar de tekentafel! Dit is wellicht een beter idee:



Dit geeft aan dat de shop een product en een order nodig heeft en dat een order een product nodig heeft. Nu hebben we geen cirkel meer in ons ontwerp en hebben we zojuist een lastige programmeer-implementatie weten te voorkomen.

1.5 Zelftest klassendiagram

1.6 Sequentie diagrammen

Coming soon...

2. Database development