De basis: Node, met generic type E

De interface Node

Uit het voorgaande kunnen we afleiden dat er weinig informatie nodig is om een boomstructuur op te bouwen. Een boomstructuur heeft als basis een node, die een reference naar een andere node als parent kan hebben. Zo'n node zou een werknemer kunnen zijn, zoals het voorbeeld uit het vorige hoofdstuk, of bijvoorbeeld een directory in een file-systeem. We gebruiken in dit geval geen identifier (zoals PERSNR of MGR) om de relatie met aan te maken: een child node bevat een directe referentie naar de parent node (dus een node **heeft-een** node: de parent).

We weten dus dat een node een parent heeft. Die moeten we kunnen opgeven en ophalen.

Maak een nieuwe interface Node in de package com.vijfhart.casus.tree.

De interface Node heeft de methoden **getParent() en setParent()**. Het lijkt logisch als getParent() dan als return type Node krijgt, en setParent() een parameter van het type Node.

In plaats daarvan maken we in de definitie van Node gebruik van een een generic type E dat een subtype is van Node. Dat maakt het mogelijk om in de code van de applicatie van dat subtype gebruik te maken:

public interface Node<E extends Node<E>> ... {....}

E, het subtype van Node, kunnen we dan gebruiken als returntype in getParent() en als type van de parameter in setParent().

Bij het implementeren van deze interface kunnen we in het volgende hoofdstuk het type E specificeren, zoals:

public class NameNode implements Node<NameNode> { }

Geef de interface Node ook vast een boolean methode isLeaf(). Deze methode geeft aan of de node een leaf node is, zoals connect_by_isleaf bij een hiërarchische query in Oracle. De overweging om deze methode in Node te zetten, in plaats van in de Iterator die nog gemaakt gaat worden, is dat de waarde van isLeaf() eenvoudig is af te leiden uit het aanroepen van setParent() in Node. Aan de andere kant is het aantrekkelijk om Node zo te implementeren dat alleen de parent bekent is, en dat isLeaf() dus pas wordt afgeleid bij het doorlopen van de nodes. Daar zullen we verderop in de casus naartoe werken.

Verder moeten we nodes in een bepaalde volgorde kunnen doorlopen. De interface Comparable is bedoeld om de natuurlijke volgorde van elementen vast te leggen. Zorg er daarom voor dat de interface Node ook Comparable is.

Bedenk in de classes en interfaces van de rest van de casus of het element-type een subtype van Node moet zijn. Gebruik dan weer <E extends Node<E>> om het gewenste type E aan te duiden.