Visitor Design Pattern opdracht

Informatielinks

- https://en.wikipedia.org/wiki/Visitor_pattern (kijk ook naar het Java voorbeeld. Ook voor de C#-ers onder ons, omdat dit een relatief duidelijk voorbeeld is.)
- https://dzone.com/articles/design-patterns-visitor
- https://sourcemaking.com/design_patterns/visitor

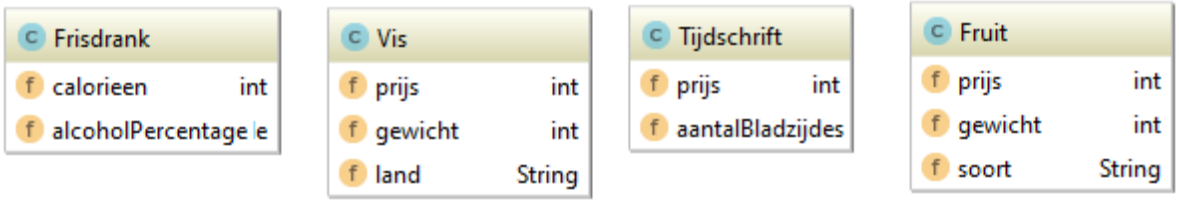
Case

In een supermarkt zijn verschillende producten te koop. Elk product in de supermarkt kan je als iets op zich zelf zien, onafhankelijk van wat er later mee gaat gebeuren. Zo heeft het product vis in de supermarkt een prijs, gewicht en land van herkomst, maar dat het later door een chefkok in een toprestaurant gebruikt gaat worden voor decoratie aan de muur, staat los van het product vis op zich.

Een aantal bedrijven zijn geïnteresseerd in de winkelmandjes van de klanten in de supermarkt. Zo wilt een bedrijf weten hoe *groen* bewust een klant is, door een score aan de inhoud van een winkelmandje te geven. Bijvoorbeeld een hoge score indien het om gezonde milieu bewuste producten gaat. Een ander bedrijf is misschien geïnteresseerd in de totaal prijs, of het aantal calorieën of gewicht van het winkelmandje.

Opdracht 1 – Maak de domein classen van de supermarkt

In de supermarkt software zijn de volgende classen bekend:




```
Vis: int prijs, int gewicht, String land
Fruit: int prijs, int gewicht, String soort
Tijdschrift: int prijs, int aantalBladzijdes
Frisdrank: int calorieen, double alcoholPercentage
```

Dit zijn tevens alle producten die de supermarkt momenteel verkoopt.

Maak deze classen.

Opdracht 2 – Bereidt de domein classen voor op visiting

De supermarkt zou berekenGroenScore() methodes kunnen toevoegen aan Fruit, Vis etc. Maar andere externe bedrijven willen weer ander type analyses kunnen uitvoeren, bijvoorbeeld misschien hoeveel papier een bepaald product verspilt, waardoor er per product een berekenPapierVerspilling() methode zou moeten komen.

De supermarkt programmeurs raken geïrriteerd door de continue wijzingen die ze voor externe bedrijven moeten implementeren. Het continue moeten wijzigen schaadt immers de onderhoudsbaarheid.

De supermarktprogrammeurs zouden tegen de externe bedrijven kunnen zeggen: zoek het zelf maar uit. De externe bedrijven zouden dan bijvoorbeeld de volgende code kunnen programmeren:

```
class Supermarkt {
	public List<Visitable> genereerWinkelmandje() {
		List<Product> winkelmandje = new LinkedList<Product>();
	  winkelmandje.add(new Fruit(2, 200, "Kiwi"));
	  winkelmandje.add(new Fruit(1, 400, "Appel"));
	  winkelmandje.add(new Vis(5, 250, "Alaska"));
	  winkelmandje.add(new Tijdschrift(4, 60));
	  winkelmandje.add(new Frisdrank(100, 12.1));
	  winkelmandje.add(new Frisdrank(200, 0));

		return winkelmandje;
	}
}

class ExternGroenScoreBerekenBedrijf {
	public void analyseerWinkelmandje(List<Visitable> winkelmandje) {
		for (Product product : winkelmandje) {
			// in C# is "instanceof" "is"
			if (product instanceof Fruit) {
				// bereken.....
			} else if (product instanceof Vis) {
				// bereken.....
			} else if (product instanceof Tijdschrift) {
				// bereken.....
			} else if (product instanceof Frisdrank) {
				// bereken.....
			} else {
				println("ERROR!!! Onbekend product");
			}
		}
	}
}
```

Al hoewel de bovenstaande code goed werkt, levert het wel allerlei onderhoudsproblemen op. Instanceof checks zijn vaak al een voorteken/aanwijzing voor het ontstaan van dergelijke problemen. Indien de supermarkt bijvoorbeeld een nieuw product toevoegt, bijv. Snacks, dan gaat het bij de externe bedrijven mis indien ze vergeten daar rekening mee te houden. Dit misgaan gebeurt op runtime (wat een stuk slechter is dan op compile-time): op het moment dat de code wordt gerund, wordt de fout pas ontdekt. Beter zou zijn als nieuwe producten op compiletime worden geforceerd bij de exerne bedrijven, zodat nieuwe producten meteen worden ontdekt. Ook zou het beter zijn als de if-statements (die complexiteit toevoegen) op een één of andere manier verwijderd kunnen worden.

De programmeurs van de supermarkt besluiten daarom om de Visitor design pattern toe te passen, om deze problemen op te lossen. In dit patroon zijn de eerder gegeven classen de *elementen*/visitables uit het patroon, waarop later analyses/operaties uitgevoerd kunnen worden door de visitors.

Maak de eerder gegeven classen/elementen visitable door ze een Visitable (of “Element”) interface te laten implementeren die je zelf moet maken.

Opdracht 3 – Maak een visitor die de groenScore berekent

Het bereken van de groenscore gaat als volgt:

1. Indien het product een vis is, en het een gewicht heeft van minder dan 500 gram, krijgt het een groenscore van 20. Indien het land van herkomst Nederland is, dan een groenscore van 10, indien uit Alaska, dan 4, en anders een score van 12.
2. Indien het product een fruit is en van het soort citrus is, dan een groenscore van 50. Indien een kiwi, dan 80, en anders 30.
3. Indien het product een tijdschrift is, dan krijgt het 10 punten indien het minder dan 50 pagina's heeft, anders 5 punten.
4. Indien een frisdrank, dan krijgt het 20 punten indien er geen alcohol in zit, en anders 2 punten.

Het externe bedrijf dat de groenScore wilt berekenen, gaat de producten als volgt gebruiken en berekenen:

```
public class Client {
    public static void main(String[] args) {
        List<Visitable> winkelmandje = new LinkedList<Visitable>();
        winkelmandje.add(new Fruit(2, 200, "Kiwi"));
        winkelmandje.add(new Fruit(1, 400, "Appel"));
        winkelmandje.add(new Vis(5, 250, "Alaska"));
        winkelmandje.add(new Tijdschrift(4, 60));
        winkelmandje.add(new Frisdrank(100, 12.1));
        winkelmandje.add(new Frisdrank(200, 0));

        GroenscoreVisitor groenscoreVisitor = new GroenscoreVisitor();
        for (Visitable product : winkelmandje) {
            product.accept(groenscoreVisitor);
        }
        System.out.println("De totale groenscore van het winkelmandje is: "
+ groenscoreVisitor.getTotaalGroenscore());
    }
}
```

Zorg ervoor dat alle classen gemaakt worden om de bovenstaande client code werkend te krijgen.

Opdracht 4 – Maak een visitor die het totaal aantal calorieën berekend

Weer een ander extern bedrijf is geïnteresseerd in het totaal aantal calorieën in de winkelmand. Maak een CalorieVisitor die als volgt werkt:

1. Voor Vis geldt: het gewicht * 4 is het aantal calorieën
2. Voor fruit geldt: gewicht * 3 is het aantal calorieën.
3. Voor Frisdrank is het aantal calorieën al bekend
4. Voor iets wat niet eetbaar is geldt dan het 0 calorieën is

Conclusie

Door het toepassen van Visitor design pattern kan je nieuwe analyses/operaties (visitors) toevoegen die werken op producten (de elementen/visitables) zonder dat je de producten zelf hoeft te veranderen. Dit is de essentie van dit patroon, zoals gedefinieerd in het invloedrijke IT boek Gang-Of-Four Gamma boek:

> *Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.*