Moduleopdracht

Ontwerpen en Programmeren

S. Crisan

Studentnummer: 4689887

Datum: 06-07-2019

NCOI

HBO Bachelor Informatica

Ontwerpen en Programmeren

Docent: Erik Mols

Voorwoord

Mijn naam is Sebastiaan Crisan. Van 1 november 2018 t/m 28 Maart 2019 heb ik stagegelopen als developer bij ADchieve in Rotterdam (hoofdkantoor Den Bosch). Ik ben begin september gestart met de opleiding “HBO Bachelor Informatica”. Met behulp van mijn stage heb ik een goede basis werkervaring opgedaan die ik hopelijk samen met de kennis uit mijn studie op het gebied van informatica kan toepassen en mij later te verdiepen.

Als stagiair developer was het mijn taak om een automatische CSS (Comparison Shopping Services) aanmelding van merchants met een Google merchant-id te realiseren via de website van de cliënt. Tevens werk ik aan een automatische reporting-tool die data vanuit Google Ads ophaalt, opslaat in een database en vervolgens aggregeert op basis van de business interests. Ik hoop met behulp van deze opdracht mijn kennis en inzichten te vergroten en deze later toe te kunnen passen bij mijn baan.

Samenvatting

Er is gekozen om de applicatie die ontwikkeld is voor ADchieve, te versimpelen en als Java-applicatie te schrijven in het kader van de moduleopdracht. Eerst zal het doel van de applicatie uitgelegd worden. Vervolgens zullen de functionaliteiten van de applicatie uitgelegd worden. Daarna zullen de klassen van de applicatie weergegeven worden door middel van een UML-klassendiagram en zal uitgelegd worden wat de verschillende elementen in het klassendiagram betekenen, en waarom bepaalde keuzes zijn gemaakt in het ontwerp. Vervolgens zal de daadwerkelijke programmacode behandeld worden, en zal het een en ander toegelicht worden met betrekking tot de keuze van het omzetten van het UML-diagram naar Java-programmacode met de problemen en lastige punten die daarbij kwamen kijken. Tot slot zullen er testen uitgevoerd worden, en uitgelegd worden welke testen zijn uitgevoerd, wat de resultaten zijn, en waarom deze testen zijn uitgevoerd. Bij alle voorgaande onderdelen zal zo veel mogelijk de theorie aan de praktijk gekoppeld worden.

Inhoudsopgave

Inleiding

ADchieve is een snelgroeiend bedrijf, hoofdkantoor gevestigd in Den Bosch, met een tweede nieuwe locatie te Rotterdam. Het bedrijf richt zich op het maximaliseren van winst bij cliënten (merchants), die reclame willen maken via Google Ads en Google Shopping, met behulp van algoritmen en big data-analyse. De moduleopdracht zal in gaan op het deel van de stageopdracht wat betrekking heeft tot het ontwikkelen van de applicatie die zorgt voor het ophalen, opslaan, aggregeren en weergeven van data.

Het doel van de moduleopdracht is het ontwikkelen van een applicatie in Java die een positieve bijdrage levert aan de functionaliteit van ADchieve. De applicatie zal ontwikkeld worden op basis van een UML-klassendiagram. Dit heeft als voordeel dat de structuur alvast is gemaakt waardoor in principe alleen de code nog maar geschreven hoeft te worden, dit is stukken overzichtelijker en biedt houvast bij het ontwikkelproces.

De applicatie zelf is zeer nuttig voor de CEO (Mark). Zo kan hij in één oogopslag zien wat zijn klanten gespendeerd hebben in bijvoorbeeld het huidige kwartaal, wat hij daar zelf aan verdiend heeft en wat de voorspelling zal zijn voor bijvoorbeeld het komende kwartaal. De applicatie die in de moduleopdracht ontwikkeld zal worden, is een versimpelde versie. Er wordt geen verbinding gemaakt met de Google API. In plaats daarvan zal een random number generator gebruikt worden die de spend genereert per dag, per klant.

Hoofdstuk 1: Doel van de applicatie

Het doel van de applicatie is een GUI (*Graphical User Interface*) te creëren voor de CEO van ADchieve (Mark). Hierbij kan hij in één oogopslag zien wat zijn klanten gespendeerd hebben in het huidige kwartaal, en wat hij daarbij verdiend heeft. Ook zullen er tools aanwezig zijn die met één klik op de knop een voorspelling kunnen weergeven voor de winst in bijvoorbeeld het aankomende kwartaal. Om de tool simpel te houden in het kader van de moduleopdracht, zal er hierbij geen verbinding gemaakt worden met de Google API. In plaats daarvan zal een random number generator gebruikt worden, die de spend genereert per dag, per klant.

Hoofdstuk 2: Functionaliteiten van de applicatie

De applicatie moet:

- Een venster weergeven met data
- In dit venster een sectie weergeven met daarin data voor het huidige kwartaal
- De data moet bestaan uit klantrijen, één rij per klant, en kolommen, met daarin
- Huidige datum
- Huidig kwartaal (1,2,3,4)
- Spend huidig kwartaal
- Winst huidig kwartaal (Spend * Tarief)
- Voorspelling winst eind kwartaal (Extrapolated profit)
- Verder moet de applicatie toegang tot de Google API kunnen simuleren
- Hierbij moet uit de “database van het Google Merchant Center”, bijvoorbeeld een .txt file in de simulatie, een lijst met alle klanten opgehaald kunnen worden.

Om de applicatie voor de moduleopdracht simpel te houden zal bij het opstarten, de datum altijd hetzelfde zijn namelijk 01-01-2019, dus het begin van het eerste kwartaal in 2019. De spend voor elke klant tot dan toe is dan ingesteld op 0.

De applicatie heeft:

- Een sectie in het venster waarin een aantal knoppen zitten, namelijk
- Next day (Ga één dag verder)
- Next week (Ga een week verder)
- Next Month (Ga een maand verder)
- Random (Ga een willekeurig aantal dagen verder)
- Hierbij wordt op basis van het aantal dagen, per dag een willekeurige spend berekend en opgeteld bij de “Spend huidig kwartaal” per klant. Verder wordt bij het klikken op de knop de huidige datum correct weergegeven, en daarbij het huidige kwartaal. Verder worden de andere kolommen ook automatisch berekend.

Verder zal de applicatie rechtsboven een knop hebben die het mogelijk maakt de applicatie af te sluiten.

Hoofdstuk 3: Ontwikkelen van het UML-klassendiagram

Voor het ontwikkelen van de applicatie is het handig om eerst een grafische weergave te hebben van de structuur van de applicatie, op basis van de eerdergenoemde requirements. Dit geeft houvast voor het ontwikkelen van de daadwerkelijke programmacode, en zorgt ook voor een duidelijke structuur van de programmacode. Het UML-klassendiagram is opgesteld met behulp van Lucidchart. Eerst zal uitgelegd worden wat elk element betekent in het klassendiagram, hiervoor moet eerst toegelicht worden van OOP (*Object Oriented Programming)* is, en dan zal het daadwerkelijke klassendiagram weergegeven worden met enkele toelichtingen over bepaalde keuzes die gemaakt zijn bij het maken van het klassendiagram, zoals het toevoegen van bepaalde klassen en waarom bepaalde klassen bepaalde connecties hebben, attributen en methoden.

Het gebruik van klassen impliceert OOP (Object Oriented Programming). Dit betekent (Savitch & Mock, 2016) dat er bepaalde objecten gemaakt worden, met bepaalde eigenschappen en functies. Deze objecten kunnen dan als bouwstenen gebruikt worden in de code of in andere code. Ook kunnen nieuwe objecten gemaakt worden die eigenschappen en functies overerven van andere objecten. Al met al brengt dit in ieder geval twee voordelen met zich mee:

- De code wordt overzichtelijk
- De code kan gemakkelijk uitgebreid worden

Over het algemeen ziet een applicatie er dan als volgt uit:

- Applicatie heeft een main method. In deze method worden alle functies aangeroepen die uiteindelijk het programma vormen.
- Applicatie heeft classes, voor elke class een aparte file (uitzonderingen daar gelaten, bijvoorbeeld voor beveiligingsdoeleinden. Wanneer een klasse alleen gebruikt dient te worden binnenin bijvoorbeeld de hoofdklasse van de applicatie.) Het structureren van klassen in aparte files is wel zo overzichtelijk, want als alles in één file zou staan (die van de main method), dan zou je zeer gemakkelijk een programma krijgen met duizenden lines of code, dit wordt al snel zeer onoverzichtelijk.

In deze applicatie worden sommige klassen gebruikt die in geimporteerde java libraries zitten. Deze klassen worden dan uitgebreid, en hier worden nieuwe klassen van gemaakt. Dit wordt gedaan door “class NieuweKlasse extends OudeKlasse”. Met andere woorden, de klasse NieuweKlasse erft eigenschappen over van OudeKlasse. Dit wordt ook wel (Savitch & Mock, 2016) overerving genoemd.

Een klassendiagram (“Tutorial ULM-klassendiagram”, z.d.) geeft in één oogopslag de structuur van de code weer met betrekking tot de klassen – welke klassen zijn er, welke relaties hebben deze klassen onderling en wat zijn de attributen en methoden van deze klassen?

Een klasse (“Tutorial ULM-klassendiagram”, z.d.) wordt weergegeven als een rechthoek met drie rijen. De bovenste rij is de naam van de klasse, de middelste rij heeft de attributen van de klasse, en de onderste rij heeft de methoden van de klasse. Attributen (Savitch & Mock, 2016) zijn eigenschappen van de klassen, bijvoorbeeld voor een Ferrari klasse de attribuut “kleur”, met als waarde bijvoorbeeld “geel” of “rood”. Dit worden ook wel variabelen genoemd. Attributen kunnen bepaalde typen zijn, in het geval van getallen bijvoorbeeld int (hele getallen), of double (getallen met cijfers achter de komma). In het geval van tekst is dit vaak “String”. Klassen worden aangeduid met een hoofdletter, variabelen met een kleine letter. Methoden zijn functies van een klasse. Om de Ferrari als voorbeeld te nemen, bijvoorbeeld “rijden()”.

Attributen en methoden (Savitch & Mock, 2016) kunnen onder andere private (-) of public (+) zijn. Public betekent dat andere klassen toegang hebben tot deze attributen en methoden, private betekent dat deze alleen toegankelijk zijn binnen de klasse waarin ze gedefinieerd staan. Dit is over het algemeen de gang van zaken, en wordt ook wel “encapsulatie” genoemd. Het doel hiervan is het voor de gebruiker verbergen van gevoelige data.

Overerving (“Tutorial ULM-klassendiagram | Lucidchart”, z.d.) wordt weergegeven door middel van een lijn met een gesloten pijl die van de subklasse (klasse die eigenschappen overerft) naar de superklasse (klasse waarvan de eigenschappen overgeërfd worden) wijst.

Een pijl met een witte diamant (“Tutorial ULM-klassendiagram | Lucidchart”, z.d.) aan het voetstuk geeft aan dat de klasse aan het voetstuk een klasse als onderdeel heeft waar de pijl naar wijst.

Een pijl met een gestreepte lijn (“Tutorial ULM-klassendiagram | Lucidchart”, z.d.) geeft een afhankelijkheid aan. Dit betekent dat de ene klasse data gebruikt van de andere klasse, of dat er data uitgewisseld wordt tussen de klassen. Het kan zelfs zijn dat een klasse als parameter gebruikt wordt voor de andere klasse.

Een hoeveelheid wordt aangegeven bij de pijl:
- 1 of 1..1 betekent één staat tot één relatie
- 1..3 betekent bijvoorbeeld één heeft drie van de andere
- 0.. betekent nul t/m oneindig

Nu duidelijk is hoe OOP werkt en hoe men UML-klassendiagrammen afleest, kan het UML-klassendiagram opgesteld worden. Het is handig om eerst de klassen op papier te zetten, op basis van de functionaliteit, en daarna het klassendiagram te maken. Allereerst moet het programma een Main klasse hebben. In Main wordt een SpendReport opgesteld met daarin een GUI met rijen en kolommen. Deze GUI kan gerealiseerd worden door middel van een venster te creëren en daar de JTable klasse (Met de klanten als rijen, en de verschillende te berekenen velden als kolommen) bovenaan aan toe te voegen, in het midden een zelfgemaakte DatePanel (extends JPanel), en onderaan zelfgemaakt ButtonPanel (extends JPanel) met de knoppen. De layout van de GUI kan ingesteld worden door zelf een layout manager aan te maken, bijvoorbeeld een GridManager klasse met als opties één kolom en drie rijen, die we dan “OneColGrid” noemen. De tabel zal in een JScrollPane gezet worden, voor het geval de tabel zeer groot wordt en niet meer in het scherm zal passen.

Nu duidelijk is hoe de applicatie eruit gaat zien, kan nagedacht worden over waar welke functionaliteit geplaatst kan worden. Er is voor gekozen om elke sectie van de applicatie te laten functioneren als manager voor de bepaalde classes die daaraan gerelateerd zijn. Het is bij de tabel bijvoorbeeld handig de visuele componenten te scheiden van wat er achter de schermen gebeurt. Dit kan gedaan worden door een klasse CustomTableModel aan te maken, die alle berekeningen uitvoert en deze dan aan de JTable doorgeeft, die het vervolgens weergeeft. Het simuleren van de Google API wordt gedaan door een klasse genaamd GoogleApi aan te maken, die dan de .txt file die als database fungeert kan aflezen en een lijst met klanten kan opstellen. Het is dus ook zinvol om een Client klasse te maken, met attributen zoals “name” en “spend” (standaard ingesteld op 0). Er is voor gekozen om voor het gemak van de applicatie vanuit de client de extrapolated spend en winst te berekenen, dit zou ook vanuit de tabel gekund hebben indien er later dingen met de echte Google API gedaan worden. Verder is het wellicht nuttig om Client methodes te geven die de naam en spend kunnen ophalen, en de spend kunnen resetten voor wanneer er naar het volgende kwartaal wordt overgegaan.

Het voorlopige UML-diagram bevindt zich als “Bijlage1 – voorlopige klassendiagram”. Nu de structuur duidelijk is, kan een goede basis code geschreven worden voor de applicatie. Echter, wat functionaliteit betreft zijn we er nog niet. Er zijn nog geen functies die rekening houden met zaken zoals, wanneer breekt het nieuwe kwartaal aan? Hoe wordt de extrapolated profit berekend? (Er zal rekening gehouden moeten worden met het aantal dagen per kwartaal, en hoeveel dagen er nog over zijn voor het nieuwe kwartaal aanbreekt.) Er zal een random number generator gemaakt moeten worden die binnen bepaalde limieten een willekeurige spend toevoegt, en een willekeurig aantal dagen kan optellen. Als er bij de overgang van een maand door op de maand knop te drukken een nieuw kwartaal aanbreekt, zal gekeken moeten worden hoeveel dagen er “geskipt” zijn in dat kwartaal en op basis daarvan voor elke dag een random spend berekend moeten worden, en wellicht zijn er

nog andere zaken die nu over het hoofd gezien worden die later aanbreken. Daarom is ervoor gekozen de code te schrijven en gaandeweg te denken over oplossingen.

Hoofdstuk 4: Het omzetten naar en uitbreiden van de programmacode

Omdat de structuur zeer duidelijk was, ging het omzetten naar programmacode in de eerste instantie ook vrij eenvoudig. Er werden Main, met daarin GoogleApi en SpendReport gemaakt. Ook werd een String gedefinieerd, namelijk het pad naar de file waar de klanten in staan. Deze kan vervolgens gebruikt worden in GoogleApi. In GoogleApi werd een method gemaakt die een ArrayList teruggeeft met Client objects, gemaakt uit de file met een lijst van klanten, die de database simuleert. Er werd dus ook een Client klasse gemaakt, met methoden die de naam en spend konden ophalen. Met deze lijst met klanten kon een SpendReport gemaakt worden. Elk SpendReport moet er hetzelfde uitzien, dus dit kon allemaal in de constructor gedefiniëerd worden.

Nu moest een plan van aanpak gemaakt worden. Want hoe zou alles berekend moeten worden? Welke processen moeten doorlopen worden en welke klassen moeten daarvoor aangemaakt worden?

Het proces is weergegeven in een activiteitendiagram, zie “Bijlage 2 - activiteitendiagram”. Hierbij werd al gauw duidelijk dat er een methode toegevoegd moest worden, die het aantal dagen tussen twee data kon berekenen. Immers – als de datum 27 maart is, en er komt een maand bij, dan komen hier m.b.t. de spend een paar dingen bij kijken:

- De datum is nu 27 april.
- Het verschil in dagen tussen 27 april en 27 maart moet berekend worden
- Voor elke dag moet een random spend opgeteld worden
- Echter, het is nu een nieuw kwartaal, de spend reset aan het begin van een kwartaal.
- Er moet een methode gemaakt worden die controleert of het een nieuw kwartaal is.
- Hierbij moet dus bij het verschil in dagen tussen de twee datums, ook nog eens het aantal dagen in het oude kwartaal afgetrokken worden. Hier moet ook een methode voor gemaakt worden.

Er is voor gekozen om een extra klasse aan te maken genaamd DaysCalculator, die bijvoorbeeld zorgt voor het berekenen van het aantal dagen dat we in een bepaald kwartaal zitten.

Ook is de klasse “RandomNumberGen” aangemaakt, die een willekeurige integer kan maken voor het aantal dagen, en een willekeurige double voor de spend die toegevoegd moet worden per dag.

Nu is er een duidelijk beeld van wat er in de updateTable() method moet komen, namelijk het berekenen van het aantal dagen met DaysCalculator, en voor elke Client de spend toevoegen/resetten op basis van bepaalde scenario’s. Hiervoor heeft updateTable(), en daarmee de klasse CustomTableModel, dus het aantal dagen nodig, de clientList met Client objects, en het moet kunnen samenwerken met datePanel. Op basis van wanneer iets gebruikt wordt in de klasse, zijn de eerdergenoemde benodigdheden toegevoegd als arguments voor de methode of als private instanties binnenin de klasse.

Er waren tijdens dit proces wat moeilijkheden, namelijk bij bijvoorbeeld het berekenen van verschillen tussen datums. Eerst werden hier de Calendar en Date klassen voor gebruikt, echter, het proces was naar ervaring zeer stroef. Later werd ontdekt dat de import “java.time” zeer nuttige klassen bevatte, zoals LocalDate en DateTimeFormatter. Er is ook lang nagedacht over hoe de communicatie tussen de klassen het minst omslachtig zou kunnen verlopen. Het was namelijk erg tricky om de actionListeners van de knoppen te maken en bepaalde acties door te geven, totdat de ingeving kwam om deze te definiëren als het resultaat van een methode. Nu kon de methode aangeroepen worden in SpendReport, de ActionListener gemaakt worden, en met behulp van een switch-case de acties te maken.

Na de code getest te hebben, wat in het volgende hoofdstuk behandeld zal worden, is het mogelijk het volledige klassendiagram op te stellen. Deze is bijgevoegd als “Bijlage 3 – uiteindelijke klassendiagram”.

Hoofdstuk 5: Testen

Er is tijdens het ontwikkelen in de eerste plaats (Chamani et al., 2018) gebruikgemaakt van “Exploratory Testing”, dat wil zeggen, tijdens het ontwikkelen in kleine iteraties testen wat je gebouwd hebt op basis van intuïtie. Dit kwam neer op het volgende: Bij elke nieuwe klasse, run code en implementeer error handling (bijvoorbeeld door System.out.println(“Class successfully implemented”)) of een try-catch wanneer de compiler daarom vroeg) en kijk of alles naar behoren werkt. Hetzelfde geldt voor methods. Verder werd er gekeken hoe het programma met verschillende waarden in de .txt file werkt:

Testgeval	Verwachtte resultaat in tabel	Resultaat in tabel	Runtime (seconden)
0 rijen	0 rijen	0 rijen	< 0.25
1 rij	1 rij	1 rij	< 0.25
3 rijen, 1e rij leeg	3 rijen, 1e rij leeg	3 rijen, 1e rij leeg	< 0.25
3 rijen, middelste rij leeg	3 rijen, middelste rij leeg	3 rijen, middelste rij leeg	< 0.25
5 rijen, laatste 3 rijen leeg	5 rijen, laatste 3 rijen leeg	5 rijen, laatste 3 rijen leeg	< 0.25
5 rijen, zelfde waarden	5 rijen, zelfde waarden	5 rijen, zelfde waarden	< 0.25
100 rijen	100 rijen	100 rijen	< 0.25
1000 rijen	1000 rijen	1000 rijen	< 0.25
10000 rijen	10000 rijen	10000 rijen	~ 0.3
100000 rijen	100000 rijen	100000 rijen	~ 3
1000000 rijen	1000000 rijen	1000000 rijen	~ 30
File in andere folder	0 rijen	0 rijen	< 0.25

Qua complexiteit / runtime is er in de applicatie een lineair verband te zien, dit zou eventueel gereduceerd kunnen worden indien de applicatie verder ontwikkeld gaat worden.

Als laatste is er een “stresstest” gedaan in het kader van performance, dit komt neer op herhaaldelijk klikken op willekeurige knoppen en kijken wat er gebeurt. Wanneer er een zichtbare tijd zit tussen het indrukken van de knoppen en de weergave van de waarden, blijft het programma visueel haken en lijkt niet meer op input te reageren. Echter, na een tijd die min of meer overeenkomt met het aantal keer herhaaldelijk klikken, worden de waarden correct weergegeven. Punten voor verbetering zijn dus

- Betere user feedback zoals informatie geven wanneer de applicatie bezig is.
- Eventueel geen klikken accepteren wanneer het programma nog bezig is.
- De runtime verminderen.

Literatuur

Chamani, H., Kruijff, G., Oosting, G., & Van Rooyen, J. (2018). *Aan de slag met software testen: principes, processen en technieken* (2e ed.). Amsterdam, Nederland: Boom.

Savitch, W. J., & Mock, K. (2016). *Absolute Java*. Essex: Pearson.

Tutorial ULM-klassendiagram. (z.d.). Geraadpleegd op 21 juni 2019, van https://www.lucidchart.com/pages/nl/tutorial-klassendiagram