

Abstract

Wij hadden voor het OOP Project de opdracht gekregen om een spreadsheet applicatie in Java te bouwen. Hieraan werden een paar minimumeisen gesteld:

- Het inlezen/wegschrijven van een XML file
- Het maken van een GUI
- De implementatie van 25 (gegeven) formules

De eerste stap was het maken van de planning en de UML. Er is gebruik gemaakt van Agile planning, waarbij we het kwartaal hebben verdeeld in vijf sprints van twee weken. Bij de UML zijn we begonnen met een 'helicopter view' van het systeem om dat vervolgens steeds meer in detail uit te werken. Voor de samenwerking is er vooral gewerkt met pair-programming en was er dagelijks contact via Whatsapp en wekelijks via Skype. De versiebeheer hebben we gedaan in GitHub.

Er is gekozen voor een eigen XML-reader en writer, zodat we het later nog altijd makkelijk kunnen aanpassen en uitbreiden. De GUI is ontworpen in Swing, omdat dit framework het beste geschikt is voor ons programma qua componenten en ondersteuning. De spreadsheet is gemaakt in JTable, omdat dit het snelste werkt bij zo een groot aantal cellen en het heeft de mogelijkheid om data via TableModel rechtstreeks door te geven aan JTable (voor efficiëntie).

Er zijn een aantal extra features verwerkt in ons programma, namelijk:

- PieCharts en BarCharts
- Het kopiëren van meerdere cellen tegelijkertijd
- Een hulpbalkje
- Een formulelijst
- Vijf extra formules (DIVIDE, LARGE, SMALL, SORT, SUB)
- De mogelijkheid om kolommen en rijen toe te voegen

Wij zijn tevreden met de samenwerking en ons eindresultaat! Voor een volgende keer zouden we nog graag de optie om cellen te selecteren met de muis (voor formules) willen implementeren.

Inhoudsopgave

1. Het verloop van het project.....	3
De planning	3
De samenwerking binnen het team.....	3
Overleg	4
Versiebeheer.....	4
2. Ontwerpproces.....	3
De problemen	5
Technologische afwegingen	6
De UML	8
Test Coverage.....	9
3. Extra functionaliteit.....	11
PieCharts en BarCharts.....	11
Formulelijst	12
Kolommen en rijen toevoegen.....	12
Het kopiëren van meerdere cellen tegelijkertijd	12
Selectie van meerdere cellen.....	13
Hulpbalkje	13
Vijf extra formules	13
4. Verbeterpunten.....	14
Verbeterpunten van de software.....	14
Verbeterpunten van het vak.....	14
Verbeterpunten van het proces	15

Het verloop van het project

Introductie

In dit deel zal uitgelegd worden hoe we het project hebben aangepakt. Hoe er gebruik is gemaakt van Agile Planning en pair-programming, hoe de samenwerking en overleg binnen ons team zijn verlopen en wat onze ervaringen zijn met de versiebeheer in GitHub.

De planning

Wij hebben ons redelijk goed aan de planning kunnen houden. Er is gebruik gemaakt van Agile Planning ('wendbare planning') met sprints van twee weken, waarbij er aan het einde van elke sprint een demo werd gehouden en er overleg werd gepleegd met de opdrachtgever. Hierdoor konden we het project goed sturen om uiteindelijk tot het gewenste eindresultaat (product) te komen. Ook bespraken we na elke sprint weer of er aan de planning was gehouden (en of het qua hoeveelheid ook wel haalbaar was). Op basis daarvan werd de volgende sprint uitgepland.

Voor elke sprint werd dus steeds twee weken naar voren uit gepland en spraken we doordeweeks een keer af om een Skype sessie te houden voor overleg. Dan kon er besproken worden wat er wel en niet goed ging en of er taken waren die toch meer tijd vereisten dan gepland, zodat we ze op tijd konden afronden voor de demo.

Wij hebben ons elke week aan de planning kunnen houden, maar rond week 6 was er toch even sprake van tijdsnood. De implementatie van de formules bleek een stuk lastiger te zijn dan we dachten, dus hierdoor was er een beetje tijd verloren en hebben we in week 2.7 extra uren gemaakt. Gelukkig ging het implementeren van de rest van de formules (na de eerste vijf) een stuk sneller.

In week 4 waren we al snel klaar met het uittekenen van de UML, omdat we in de weken daarvoor al veel hadden nagedacht over hoe we het programma wilden opstellen. Daarom was er in week 4 besloten om alvast aan de GUI te beginnen en om in de week van de midterms door te werken. Hierdoor was er aan het einde van het project nog tijd over om extra functionaliteit toe te voegen aan het programma, waaronder een helpbalkje en grafieken. Dit kwam onze planning dus ten goede!

De samenwerking binnen het team

De samenwerking ging erg goed binnen ons groepje. Iedereen heeft zich goed aan zijn taken gehouden en als iemand ergens niet uitkwam, hielpen we elkaar ook. Als er aan het einde van een sprint weer een vergadering/demo was, maakten we voor de volgende sprint (voor de volgende twee weken) een nieuwe planning en

verdeelden we ook gelijk de taken. Hierbij maakten we duidelijke afspraken over wie wanneer wat moest afhebben, zodat we op elkaar konden rekenen. Dat was ook wel belangrijk, omdat veel onderdelen van het programma op elkaar aansluiten (en dus van elkaar af hangen).

Er is vooral gewerkt met pair-programming. We splitsten ons groepje elke keer in teams van twee en drie. Het groepje van twee richtte zich vooral op de GUI en het groepje van drie op de formules (en implementatie). Opdrachten als de UML en het verslag hebben we samen met z'n allen gemaakt.

Overleg

Er is vooral op Whatsapp dagelijks overlegd over de meer 'algemene' groepstaken, zoals de UML en het voorbereiden van de demo's. Ook als iemand een korte vraag of mededeling had, kon dat makkelijk via Whatsapp. Naast de commits op GitHub, wisselden we via mail de overige (tekst)bestanden uit, zoals de planning (back-log) en de taakverdeling. Er is ook een aantal keer in Google Docs 'real-time' samengewerkt aan het eindverslag en de presentatie.

Versiebeheer

Wij hebben gebruik gemaakt van (gedecentraliseerd) versiebeheer in GitHub. Het heeft ons erg geholpen, want op deze manier konden we handig tussenversies van het programma met elkaar vergelijken (dus om te zien welke veranderingen er waren gemaakt tussen twee commits). Dat gaf ons dan snel een overzicht van wie waaraan had gewerkt en welke wijzigingen diegene had gemaakt. Hierdoor was het makkelijker om samen aan hetzelfde stukje code te werken en om elkaar te helpen als iemand er bijvoorbeeld niet uitkwam (om bugs op te lossen bijvoorbeeld).

Bovendien was het met GitHub ook makkelijk om een eerdere versie van het programma terug te halen. Op een gegeven moment was er voor de GUI een knopje gemaakt waarmee kolommen aan de spreadsheet toegevoegd kunnen worden. Maar door een kleine error, die niet zo makkelijk teruggevonden kon worden in de code, bleven de kolommen oneindig toegevoegd worden en crashte het programma steeds. Gelukkig konden we toen heel makkelijk op GitHub de vorige versie (zonder error) van een ander teamlid terughalen en daarmee verder werken.

Toch zijn er ook een paar keer problemen geweest met GitHub. Als in hetzelfde uur een paar keer gecommit werd in dezelfde klasse (door verschillende mensen) liep het programma al snel vast (dan was er sprake van een zogenaamde 'detached head'). Dit kon worden opgelost door iedereen opnieuw de gehele map te laten klonen.

Ontwerpproces

Introductie

In dit deel zal uitgelegd worden hoe we het programma hebben ontworpen en gemaakt. Welke problemen er tijdens het ontwerpproces zijn opgetreden en welke technologische afwegingen er zijn gemaakt. Ook zal er verteld worden hoe de UML ons heeft geholpen met de communicatie binnen het team en hoe het programma in JUnit getest is.

De problemen

Het opslaan van de inhoud van de cellen tijdens runtime

Er is gekozen om de cellen op te splitsen in drie verschillende typen die hun gemeenschappelijke eigenschappen (rijnummer en kolomnummer) erven van een superklasse Cell. Eén celklasse wordt gebruikt voor het opslaan van tekst, een tweede voor getallen en een derde voor het opslaan van formules. Omdat de formules in ons programma altijd met doubles zouden gaan rekenen, is er in de celklassen geen onderscheid gemaakt tussen doubles en integers, maar wordt elk getal als double opgeslagen. Er is gebruik gemaakt van een aparte klasse voor formulecellen, zodat we niet steeds moeten controleren of een string een geldige formule is. Alle instanties van de verschillende celtypes worden opgeslagen in een tweedimensionale arraylist die zich bevindt in de klasse Spreadsheet. Wij maken hierbij gebruik van een arraylist zodat onbeperkt nieuwe cellen toegevoegd kunnen worden.

De formulebalk werd niet goed weergegeven

De eerste werkende versie van de GUI bestond uit een menubalk met daaronder de tabel. Toen we later daartussen een formulebalk wilden toevoegen liepen we tegen het probleem aan dat de tabel verdween en de formulebalk over het volledige scherm werd weergegeven. Dit hebben wij na veel zoekwerk opgelost door de ruimte onder de menubalk met een JSplitPane op te splitsen in twee delen. Het bovenste deel bevat een paneel met de formulebalk en het onderste bevat de tabel.

Het doorgeven van data aan de formules

Bij het aanleveren van data aan formules werd er eerst gebruik gemaakt van een ArrayList met doubles, maar toen wij op het punt stonden om de formules te implementeren, kwamen wij erachter dat dit niet altijd even handig is. Vooral bij formules die een String als invoer krijgen en bij formules die een String of integer als terugwaarde moeten geven. Een ArrayList met doubles kan namelijk alleen doubles doorgeven aan formules (door middel van DoubleParser). Dit probleem is uiteindelijk opgelost door gebruik te maken van ArrayLists met Strings, omdat daarmee zowel Strings als doubles doorgegeven kunnen worden aan de formules.

De ISNUMBER() functie

De ISNUMBER functie controleert of een meegegeven waarde een getal is en retourneert op basis daarvan TRUE of FALSE. Het was ons opgevallen dat de ISNUMBER functie ook TRUE gaf als de letter F in de ingevoerde waarde voorkwam, dus als er bijvoorbeeld de waarde 5F werd ingevuld. Doordat de letter F erin voorkomt is het een String en zou de functie dus eigenlijk FALSE moeten teruggeven. Blijkbaar kwam dit doordat de F als 'Float' (floating-point getal) werd geïnterpreteerd (en dus als een getal). Wij hebben dit kunnen oplossen door een controle in de functie te plaatsen die kijkt of de ingevoerde waarde een F bevat en als dat inderdaad zo is, wordt direct FALSE geretourneerd.

Technologische afwegingen

Een eigen XML-reader en writer

Er is gekozen om een eigen read methode te schrijven in plaats van een standaard XML parser te gebruiken, omdat we de read methode dan later nog altijd makkelijk kunnen aanpassen en uitbreiden. Ook begrepen we daardoor precies hoe in ons programma de waarden in de cellen worden ingelezen en teruggeschreven. Als er fouten (bugs) optraden bij het inlezen van de invoer, konden we sneller opsporen waar het precies in de code was fout gegaan.

Doordat we de write methode ook zelf hebben geschreven, hebben we erin kunnen verwerken dat alleen de cellen met een ingevoerde waarde naar file worden geschreven (dus niet de gehele spreadsheet met lege cellen). Dit is efficiënter en het opslaan gaat daardoor ook sneller.

Het ontwerpen van de GUI in Swing

Onze grootste technologische afweging was het kiezen van het framework voor de Grafische Userinterface. Wij hebben daarvoor onderzoek gedaan naar een aantal bekende frameworks (SWT, AWT, Swing en JavaFX). De frameworks hebben we vergeleken op een aantal punten en er is besloten om de GUI in Swing te maken. Swing heeft namelijk een groot aantal componenten waaruit gekozen kan worden, terwijl AWT en SWT daar wel beperkt in blijken te zijn.

Verder is Swing het oudste framework, waardoor er online dus veel documentatie voor beschikbaar is. JavaFX is juist weer nieuwer en heeft daardoor een modernere uitstraling, maar het is moeilijker te gebruiken dan Swing en het heeft minder documentatie. Bovendien is Swing eenvoudig in gebruik, omdat het al standaard onderdeel van Eclipse is. Ook qua ondersteuning door de verschillende besturingssystemen kwam Swing als beste naar voren. Naast dat een Swing GUI werkt op een ander besturingssysteem wordt ook het uiterlijk van de applicatie aangepast naar wat men gewend is van dat besturingssysteem.

Op basis van deze punten hebben is ervoor gekozen om voor de Grafisch Userinterface het Swing framework te gebruiken.

JTable versus JqGrid en dataTable

Voor de representatie van onze data in een spreadsheet is er vergeleken tussen JTable, JqGrid en dataTable. Er is uiteindelijk gekozen voor JTable, omdat JTable vergeleken met anderen het meest geschikt is voor ons project. Van JTable is veel documentatie beschikbaar en het is qua User Interface meer uitgebreid dan die van JqGrid en dataTable. JTable heeft namelijk meer mogelijkheden om de weergave (van de spreadsheet) aan te passen en heeft ook diverse tabellen die als standaard gebruikt kunnen worden. Dit was doorslaggevend in onze keuze, omdat wij in ons programma de optie om de spreadsheet te vergroten (meer kolommen toevoegen) wilden implementeren.

Ook werkt JTable het snelste van de drie, omdat dataTable al langzaam wordt bij honderd lokale items (cellen). JqGrid kan iets meer aan, maar bij duizend cellen heb je toch echt JTable nodig. Bovendien is er in JTable een mogelijkheid ingebouwd om je data in de methoden van de TableModel interface te zetten, zodat het rechtstreeks doorgegeven kan worden aan JTable. Hierdoor wordt je applicatie efficiënter, omdat het model zelf bepaalt welke interne structuur het beste past bij de ingevoerde data. Hier hebben wij ook gebruik van gemaakt.

JUnit versus andere testframes

Wij hadden voor de tests een beetje onderzoek gedaan naar verschillende testframes en hieruit bleek dat JUnit en TestNG het meest geschikt zijn voor ons programma. Allebei de frames zijn namelijk al standaard geïntegreerd in Eclipse. Ze bieden ook allebei genoeg basis test coverage (hoewel TestNG op een aantal specifieke punten wel meer mogelijkheden biedt). JUnit is daarentegen wel een stuk makkelijker te gebruiken. Daarnaast blijkt JUnit's Eclipse plugin een stuk beter te zijn als het gaat om integratie tests. Dat zijn tests waarbij je software modules samen als een groep test.

Door het makkelijkere gebruik en doordat wij allemaal natuurlijk al ervaring hadden met JUnit (van OOP uit het 1^e kwartaal), is er besloten om ons programma in JUnit te testen.

De UML

Ons programma is ontworpen volgens het Model-view-controller (MVC) model. Dit is een ontwerppatroon waarin het programma in drie eenheden wordt gesplitst: het datamodel (model), de datarepresentatie (view) en de applicatielogica (controller). In ons geval vormt de klasse Spreadsheet het datamodel, de GUI vormt de view en we hebben gebruik gemaakt van een Main Controller (de main() waarmee het programma aangeroepen kan worden).

De UML heeft ons zeker geholpen. Toen wij de UML gingen maken, hebben we het eerst met z'n allen op papier uitgetekend. Voor de uitwerking van het schema is er gebruik gemaakt van Astah. Hierdoor waren we al vanaf het begin van het project aan het vooruitdenken over de uitwerking van de GUI en de formules. Dit heeft voorkomen dat we in de laatste weken voor grote verrassingen kwamen te staan, omdat over het grootste deel van de opzet van het programma dus al nagedacht was.

De UML is ook handig geweest om snel een overzicht te kunnen krijgen van de structuur van het programma: welke klassen zijn er en hoe zijn die verschillende klassen met elkaar gerelateerd? Meestal heb je wel een idee van hoe de klassen eruit moeten zien, maar door de UML kun je het totaalplaatje zien. Hierdoor konden we ook makkelijker taken van elkaar overnemen. Als bijvoorbeeld het werk van een ander teamlid afgemaakt moest worden, was in de UML snel te zien welke attributen/methoden er nog misten.

Bovendien maakte de UML het mogelijk om wijzigingen in het programma duidelijk door te geven. Als bijvoorbeeld een klasse toegevoegd werd of een relatie aangepast moest worden, kon dit gelijk via de UML doorgegeven worden aan de rest van de groep. Het heeft in die zin dus wel degelijk geholpen met de communicatie binnen ons team. We hebben de UML dan ook met elkaar gedeeld in Google Docs.

Test Coverage


Er is in ons programma gekozen om vooral veel met JUNIT-tests te werken. We hebben deze tests ook al gebruikt bij vorige Java opdrachten dus we waren er gemakkelijk mee weg. Bovendien zijn JUNIT-tests ook het beste geschikt voor het testen van de meeste onderdelen in onze applicatie (zie blz. 7).

Formules

De formules waren het gemakkelijkst om te testen en ze waren ook wel nuttig om na te kijken. Zo zijn er veel kleine foutjes in de berekeningen opgelost.





Dit is een algemeen overzicht van wat er getest is:

- functies in normale gevallen
- functies met een lege input.
- ongewone gevallen (negatieve getallen in vierkantswortel, delen door nul, enz)
- excepties (string in een berekening invoeren)

►  formules  83,5 % | 5.717 | 1.131 | 6.848 |
Coverage: 83.5 %









Models

Van de models is bijna alles getest. De **XMLReader** en **XMLWriter** zijn volledig getest en die zijn als eerste nagekeken omdat dit zeer belangrijk was voor de dingen die erop volgde.

►  XMLReader.java  96,5 %
►  XMLWriter.java  92,3 %

Ook **spreadsheet** is een belangrijke klasse die goed getest moet worden omdat het eigenlijk het skelet is waarop ons programma is gebaseerd. Het is het rekenblad waarin alle cellen zitten.

Coverage: 81.4 %

►  XMLWriter.java		91,2 %	104	10	114
►  XMLReader.java		91,1 %	247	24	271
►  Spreadsheet.java		81,4 %	184	42	226
►  DoubleCellTest.java		81,0 %	64	15	79

Zowel de **Cell/DoubleCell/ StringCell** zijn volledig getest op functionaliteit. Hier viel niet al te veel te testen op de constructor en het teruggeven van de inhoud na. Totale coveage van models: 68.2 %

▶ models	68,2 %	1.583	737	2.320
----------	--------	-------	-----	-------

Het testen van de controller/ view

De GUI is zeer moeilijk te testen met een JUNIT-test, omdat er geen echte berekeningen worden gedaan of niet bepaald iets terug wordt gegeven. We begrepen dat het wel mogelijk is, maar dit krijgen we later bij de studie nog.

We hebben dus geprobeerd alle knoppen zo goed mogelijk te testen door ze allemaal uit te voeren. Er is ten eerste nagekeken of er geen errors tevoorschijn komen en ten tweede dat de knop ook degelijk doet wat hij zou moeten doen.

De controllers waren ook lastig om te testen met de JUNIT testen om de zelfde redenen. We krijgen bij volgende vakken meer uitleg over het testen van de view in een applicatie.

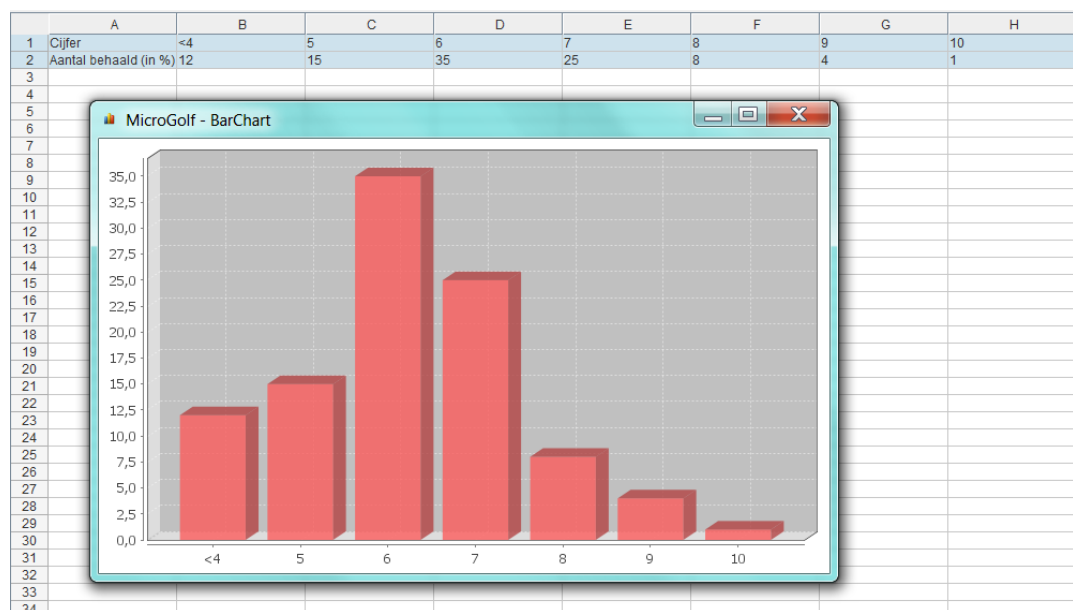
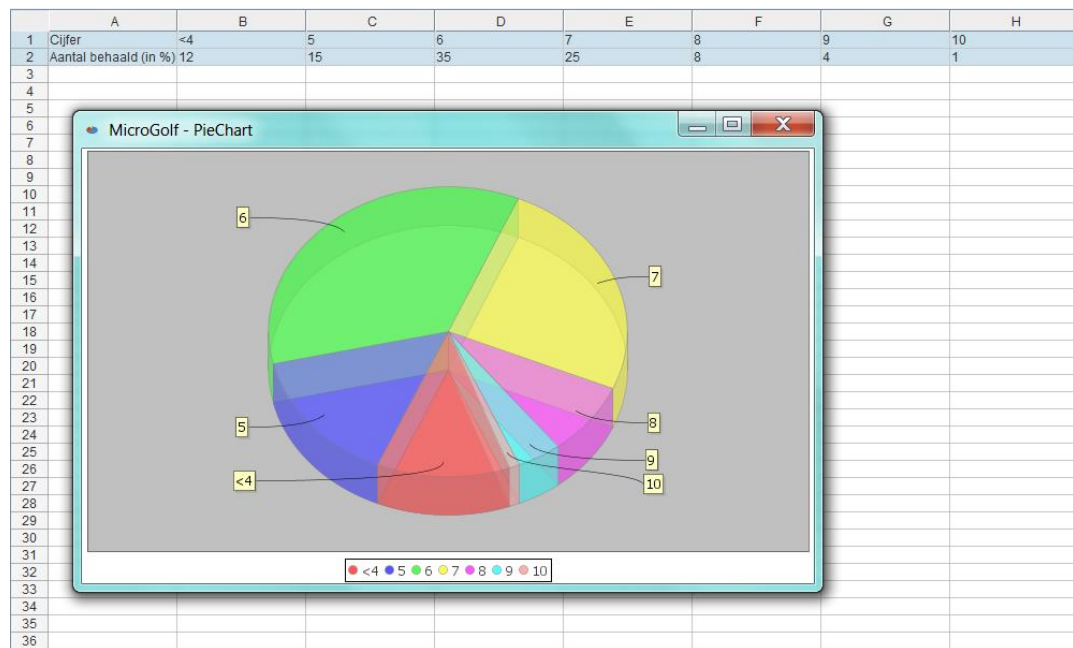
Extra functionaliteit

Er zijn een aantal extra features verwerkt in het programma:

- **PieCharts en BarCharts**

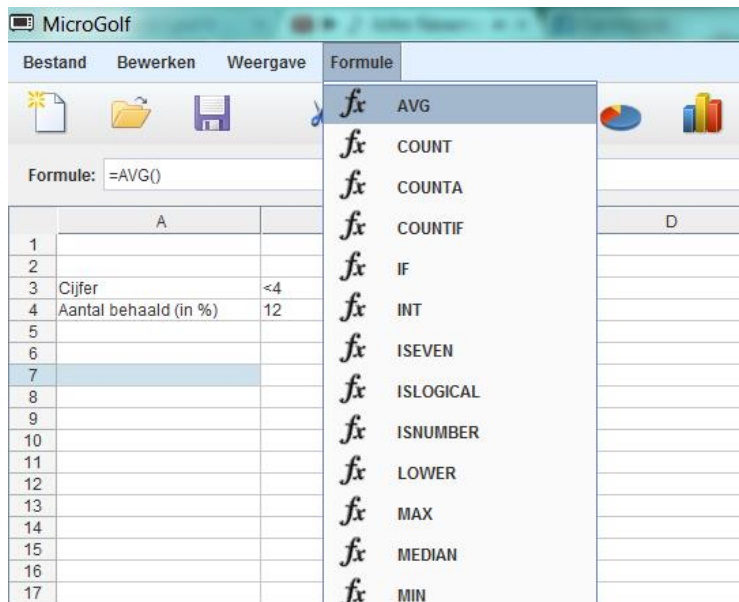
Er kan een PieChart of BarChart gemaakt worden op basis van twee rijen met ingevoerde waarden.

Als voorbeeld ziet u op de onderstaande foto's een PieChart en een BarChart van de cijfers voor een bepaald tentamen:



- **Formulelijst**

Door in het menu op 'Formule' te klikken, wordt de formulelijst geopend. Als je hieruit een formule kiest, wordt de gekozen formule ingevoerd in de formulebalk. Dit werkt ook bij geneste formules (een formule in een formule). Als er in de formulebalk bijvoorbeeld al "=SUM()" is ingevoerd en er een andere formule uit de formulelijst wordt gekozen (bijvoorbeeld AVG), dan vult het programma dit aan tot "=SUM(AVG())". De geneste formule wordt geplaatst waar de cursor staat (dus tussen de haakjes). Na het invoeren van de geneste formule, wordt de cursor ook weer achter het haakje geplaatst, zodat de gebruiker direct waarden kan invoeren of nog een geneste formule kan toevoegen.



- **Kolommen en rijen toevoegen**

Door in het menu op 'Weergave' te klikken, kunnen er kolommen en rijen toegevoegd worden aan de spreadsheet. De nieuwe kolom wordt achteraan toegevoegd met de volgende letter in het alfabet en een nieuwe rij wordt onderaan toegevoegd met het opvolgende nummer. Deze optie werkt ook met de sneltoetsen Ctrl-D (voor het toevoegen van een kolom) en Ctrl-F (voor het toevoegen van een rij).

- **Het kopiëren van meerdere cellen tegelijkertijd**

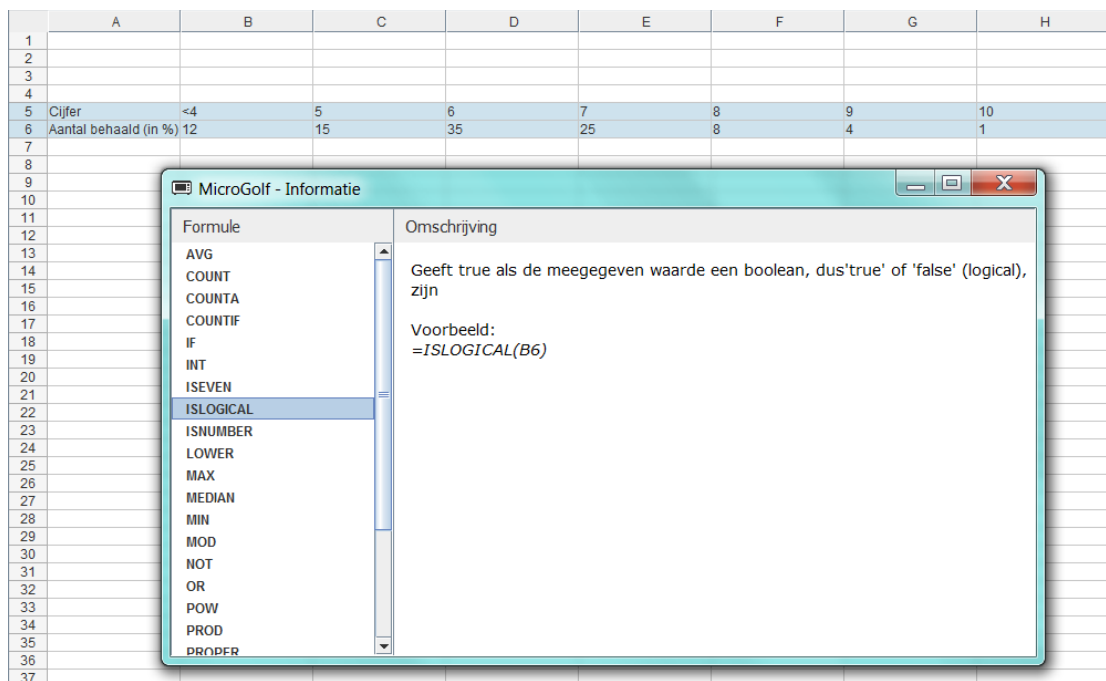
Met het 'kopieren' en 'plakken' knopje kunnen er meerdere cellen tegelijkertijd gekopieerd worden en op een andere plek in de spreadsheet geplaatst worden. Dit is vooral handig als u grote rijen data in een andere volgorde wilt sorteren of naar een andere kolom wilt verplaatsen. Deze optie werkt ook met de sneltoetsen Ctrl-C en Ctrl-V.

- Selectie van meerdere cellen

Om meerdere cellen als invoerwaarde mee te geven aan een formule, kan dat aangegeven worden met een dubbele punt. Als bijvoorbeeld de cellen A1 t/m 5 moeten worden geselecteerd, kan dat als volgt: =SUM(A1 : A5). De dubbele punt staat dus voor tot en met.

- Het helpbalkje

Als er op de hulpknop (i) gedrukt wordt, wordt het helpbalkje geopend met informatie over de verschillende formules. Bij elke formule wordt uitleg gegeven over wat het doet, een voorbeeld van hoe je het gebruikt (met invoerwaarden) en eventueel nog een opmerking.



- Vijf extra formules

Er zijn vijf extra formules geïmplementeerd in ons programma, namelijk:

- DIVIDE: deelt twee getallen door elkaar
- LARGE: geeft het hoogste getal van een lijst, aftellend vanaf de ingevoerde waarde (waarde van het hoogste getal is 1)
- SMALL: geeft het laagste getal van een lijst, optellend vanaf de ingevoerde waarde (waarde van het laagste getal is 1)
- SORT: Sorteert alle getallen in de lijst van klein naar groot
- SUB: Trekt de meegegeven waarden van elkaar af

Deze extra formules zijn onderaan de formulelijst te vinden.

Verbeterpunten

Introductie

In dit deel zal uitgelegd worden wat volgens ons mogelijke verbeterpunten zijn in het project. Wat zou er nog toegevoegd kunnen worden aan ons programma en wat kan er de volgende keer beter aangepakt worden in het vak zelf en ons (werk)proces?

Verbeterpunten van de software

Wij hadden eigenlijk graag nog de optie willen implementeren dat de gebruiker celwaarden aan een formule kan doorgeven door de cellen te selecteren met de muis. Op dit moment kan de gebruiker dat in ons programma alleen doen door de celwaarden op zijn toetsenbord in te voeren, bijvoorbeeld SUM(A1 : B2). Voor sommige functies waarvoor je veel cellen moet selecteren (bijvoorbeeld COUNT) is het veel handiger om dat te doen met je muis. Helaas was hier geen tijd meer voor en hadden wij besloten dat de resterende tijd beter gebruikt kon worden voor andere delen van de GUI. Dit is dus zeker een mogelijk verbeterpunt voor een volgende keer.

Verbeterpunten van het vak

Het vak was goed georganiseerd, maar toch ontstonden er tijdens het project wel een paar onduidelijkheden. In de tweede week was het namelijk nog steeds niet bekend wat er elke week precies voorbereid moest worden voor de demo's/vergaderingen. Ook werd het pas een week vantevoren bekend wanneer de laatste deadline zou zijn van het programma en de definitieve versie van het eindverslag. In de excelsheet uit week 3 stond dat het eindverslag op 31 januari ingeleverd moest worden en de deadline van het programma was toen nog niet bekend. Pas in week 9 werd gemaild dat de deadline van allebei 'vervroegd' was naar de maandag van week 10 (27 januari). De deadlines waren dus niet altijd op tijd bekend bij ons.

Ook vonden wij de opdracht op bepaalde punten vaag geformuleerd. De doelstelling van het bouwen van een spreadsheet kan namelijk erg breed opgevat worden. Uit hoeveel kolommen moet de spreadsheet bestaan? Hoe moeten de cellen geselecteerd worden bij het invoeren van een formule? Onze begeleider heeft deze vragen gelukkig kunnen beantwoorden en als pluspunt hadden we daardoor wel veel vrijheid bij het maken van het programma.

Verbeterpunten van het proces

We hadden meer achtergrondonderzoek moeten doen over de formules, voordat we aan de implementatie daarvan begonnen. Nadat de GUI grotendeels afgerond was, gingen we ervan uit dat de formules dan ook wel makkelijk moesten lukken, maar dat bleek toch een stuk complexer te zijn dan gedacht. Als dat van tevoren goed uitgezocht was op Internet, hadden we kunnen weten dat we er in week 6 toch meer tijd voor moesten inplannen. Doordat het was onderschat, kwamen we in week 6 in tijdsnood en zijn er in week 7 extra veel uren gemaakt.

Achteraf was het waarschijnlijk ook beter geweest als er vaker was gecommit en dan ook in kleinere stukken code. Meestal werd een stuk nieuwe code pas geplaatst als het helemaal af was, maar dat zorgde ervoor dat we dan in *één* keer te maken kregen met grote wijzigingen in het programma waardoor GitHub soms vastliep. Voor een volgende keer zou het beter zijn als er per klasse gecommit wordt en bij de grotere, complexe klassen (zoals FormuleParser bijvoorbeeld) per methode. Ook had er door vaker te committen voorkomen kunnen worden dat een ander teamlid dan in de tussentijd aan hetzelfde onderdeel had zitten werken. Dit is namelijk wel een paar keer gebeurd en hier hadden we tijd mee kunnen sparen.