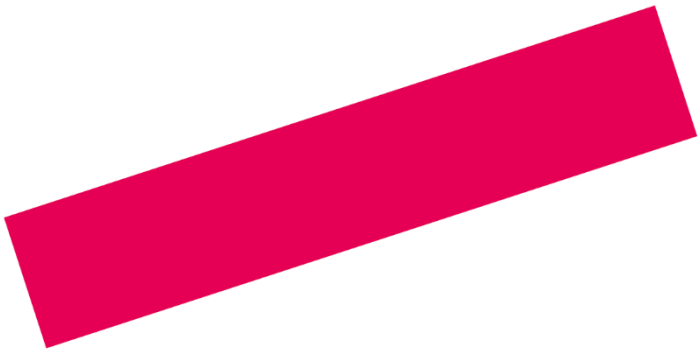


READER
STRUCTURED PROGRAMMING APPLICATION DEVELOPMENT_

Versie SP-AD.1.1

M.C. van der Maas

15 augustus 2021

HAN_UNIVERSITY OF APPLIED SCIENCES

INHOUDSOPGAVE

INLEIDING

Welkom bij het tweede deel van Structured Programming, Structured Programming Application Development (SP-AD). Tijdens SP-B leerde je eerste basis van programmeren. Tijdens deze course gaan we dieper in op codekwaliteit en structuur en gaan we een grotere applicatie schrijven.
In de komende hoofdstukken behandelen we blikken we terug op methoden parameters en returnwaarden en gaan we in op het testen van software. Ook kijken we hoe je verder structuur geeft aan grotere programma's. Nadat alle stof behandeld is geven we in het laatste hoofdstuk inzicht in de eisen die gesteld worden aan de realisatie van de eindopdracht.

1 HERHALING KWALITEITSEISEN METHODEN

Toepassing van methoden is van essentieel belang bij het realiseren van de eindopdracht van SPAD. Omdat deze theorie al behandeld is tijdens SPB volstaan wij hier met een verkort overzicht van de kwaliteitseisen die we stellen aan methoden. Deze zijn:

1. Voor elk codeblok dat een bepaalde omkaderde taak uitvoert schrijven we in een methode.
2. Elke zelfgeschreven methode doet maar één ding, heeft maar één verantwoordelijkheid.
3. Informatie die een methode nodig heeft wordt meegegeven via parameters.
4. Een methode geeft informatie terug aan het hoofdprogramma via de returnwaarde.
5. De naam van de methode wordt geschreven in lowerCamelCase en beschrijft zo nauwkeurig mogelijk wat de methode doet.
6. De naam van een methode bevat een werkwoord: tekenHuis(), berekenGemiddelde(), isGroot().

Voor wat betreft accolades maken we de volgende algemene afspraken:

- In het geval van accolades is het de gewoonte om de openingsaccolade achteraan de regel te plaatsen die het blok inluidt. Bij een methodedefinitie is dat aan het einde van de header.
- Vervolgens spring je één tab (standaard voegt Processing dan 2 spaties in) in voor elke programmaregel die volgt, om vervolgens
- de sluit-accolade weer een tab terug te laten springen. In het vorige fragmenten zie je dat de sluit-accolade precies onder de v van void staat.

Doorloop ook nogmaals alle kwaliteitseisen uit de andere hoofdstukken van de SPB-reader. Deze eisen moet je zo goed mogelijk toepassen in de eindopdracht!

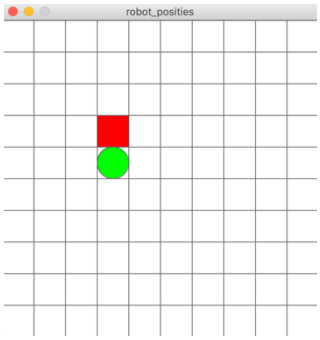
2 SOFTWARE ARCHITECTUUR

Tijdens de gehele course Structured Programming benadrukken we de noodzaak van hoge codekwaliteit en hebben we verschillende kwaliteitsafspraken met elkaar gemaakt: goede naamgeving van variabelen, constanten en methoden, helder en verhelderend commentaar, het scheiden van verantwoordelijkheden door gebruik te maken van eigen methoden (ook wel “Separation of Concerns” genoemd), en ga zo maar door. In dit hoofdstuk gaan we een stapje verder. We gaan onze code nog verder structureren en leren een eenvoudige software architectuur op te stellen.

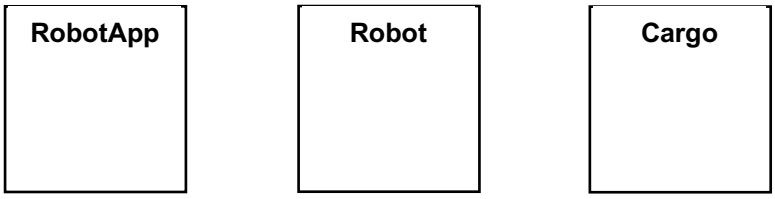
2.1 Modulen definiëren

Naarmate ons programma groter wordt nemen ook de hoeveelheid methoden en variabelen toe. Daarom gaan we die methoden en variabelen beter structureren en onderverdelen. Dat betekent dat we gegevens en methoden die bij elkaar horen vanaf nu bij elkaar gaan bundelen in modulen. Dit noemen we in de programmeerwereld “Encapsulatie”.

Voorbeeld



In SP-B hebben we een vierkante industriële robot gemaakt die een ronde cargo kon oppakken en verplaatsen. Die code kunnen we verbeteren door methoden en variabelen die bij elkaar horen te groeperen in modulen. Alle methoden die betrekking hebben op de robot groeperen we in één module Robot. Ook de methoden die te maken hebben met de cargo horen bij elkaar en groeperen we dus in een aparte module, bijvoorbeeld Cargo. Heel logisch eigenlijk. Tot slot hebben we het hoofdprogramma die alles aanstuurt in een eigen module die we RobotApp noemen.



In Processing maak je een module door een nieuw (leeg) tabblad aan te maken.

Let op de naamgeving van de modulen in bovenstaande afbeelding. We spreken af dat we de modulenaam in UpperCamelCase schrijven, dit in tegenstelling tot methoden en variabelen die we in lowerCamelCase schrijven. Dit doen we in onze diagrammen, en in onze code.

Als de samenhang tussen de methoden in een module heel helder is en de naam van de module eenduidig dan is code heel snel terug te vinden. Ook bij grotere programma's en zelfs als het programma door een ander geschreven is. Mede daarom moeten we, naast een duidelijke eenduidige naam, ervoor zorgen dat de samenhang tussen de methoden die we groeperen zo hoog mogelijk is. Dit noemen we "hoge cohesie".

Modulen gebruiken nooit elkaars variabelen, maar roepen **alleen methoden** aan. We proberen hetgeen wat er gedaan wordt in een module zoveel mogelijk los te koppelen van hoe dat gedaan wordt (de implementatiedetails).

<optionele theorie> Getters en setters

Ook globale variabelen kunnen we in de bijbehorende module declareren. Zijn er globale waarden of constanten in een module waar we *vanuit een andere module echt toegang toe moeten* hebben? Dan spreken we de globale variabelen niet direct aan, maar definiëren we get en set methoden (zg. getters en setters) en benaderen we ze alleen via deze methoden. Getters geven de waarde van een modulevariabele terug via een returnwaarde, bijvoorbeeld:

```
int getIdentity(){
  return identity;
}
```

Met de setters geven we globale variabelen in een andere module een nieuwe waarde via de parameter:

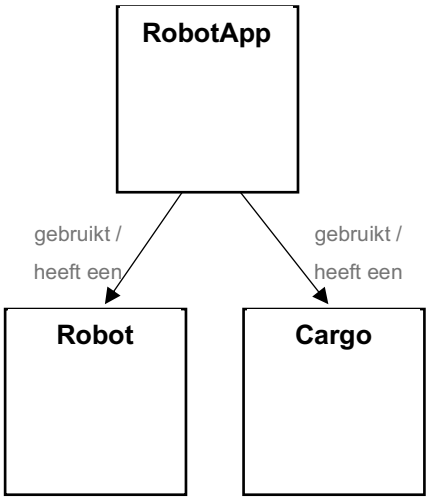
```
void setIdentity(int id){
  identity = id;
}
```

Bijkomend voordeel is dat je nu invoer kunt valideren. Als het id niet negatief mag zijn dan kun je dit dus met een if-statement afvangen.

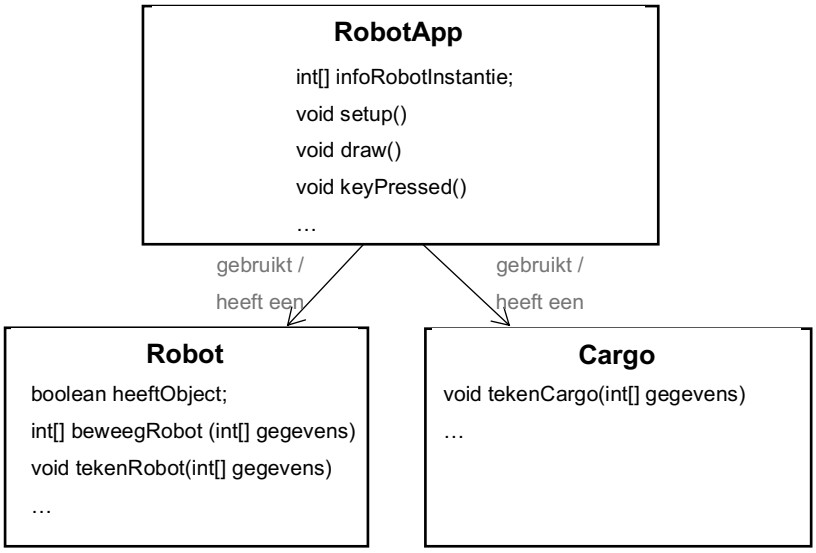
Wanneer je een ontwerp maakt waarbij twee module te veel met elkaar zijn verweven dan zorgt dat voor onnodig werk en verhogen de kansen dat jouw code niet meer goed werkt als je het aan wilt passen.
Als bijvoorbeeld modulen zoals Robot en Cargo te strak gekoppeld zijn, dan kan Robot zijn werk niet goed doen zonder de implementatiedetails van Cargo te weten (zoals de namen van variabelen). Het gebruik van die implementatiedetails in een andere module betekent dat we op twee of meer plaatsen de code moeten veranderen als we iets aan de code willen aanpassen (dat aanpassen van bestaande code noemen we refactoren). En over een maand weten we niet meer goed meer hoe de code ook alweer in elkaar stak. Met fouten en tijdsverspilling om deze op te lossen tot gevolg.

Zorgen we voor een losse koppeling dan kun je de internals van Cargo makkelijker veranderen zonder dat we daar ook de code van de Robot moeten induiken om te voorkomen dat de code breekt. Bovendien maakt het coderen efficiënter omdat je kunt meer code kunt hergebruiken (de module staat tenslotte volledig los van de rest van de code). De Robot module kan je bijvoorbeeld zo in jouw nieuwe game gebruiken.

Tot slot moeten we zoveel mogelijk voor eenzijdige afhankelijkheid zorgen. Als module A een functie aanroept uit module B en module B een functie aanroept uit module A ontstaat er ook een sterke koppeling. Beide module kunnen namelijk niet zonder elkaar en geen van beide modulen zijn dus op zichzelf te hergebruiken. Als module A methoden aanroept uit module B, zonder dat module B iets van A nodig heeft, is module B los van A te hergebruiken.



Het volledige architectuurplaatje krijgen we tenslotte door ook de methoden en eventuele modulevariabelen in het diagram op te nemen. Voor onze robot applicatie ziet het architectuurplaatje er dan als volgt uit:



2.2 Kwaliteitsafspraken

Hiertoe komen we tot de volgende kwaliteitsafspraken:

- Methoden die bij elkaar horen bundelen we in een module.
- Declaraties van variabelen groeperen we.
- De modulenaam wordt UpperCamelCase geschreven
- Methoden in een module hebben altijd een hoge samenhang en
- Modules hebben een lage koppeling:
    - Modules kennen alleen elkaars methoden, en niet elkaars variabelen en constanten:
        - Geef waarden van variabelen door via parameters
        - Maak zoveel mogelijk gebruik van returnwaarden
        - Voor globale variabelen en constanten in modulen: definieer getters en setters wanneer deze in andere modulen toegankelijk moeten zijn.
    - Methoden-aanroepen uit andere modulen gaan één kant op en niet terug. Dus als component A methoden van component B aanroept, mag B geen methoden van A aanroepen.
- Onze ideeën hierover leggen we vast in een software architectuur diagram.

3 TESTEN EN FOUTOPSPORING

Een belangrijk onderdeel bij het programmeren is dat je ook aan kan tonen dat je programma werkt. Je moet dus kunnen laten zien dat het doet wat het moet doen en niets anders. Dit bereik je door je programma te *testen*.

3.1 Tests schrijven

Testen is meer dan je programma uitvoeren, wat acties doen en gewoon kijken of het werkt. Want als je alleen dit doet is er een aantal problemen.

Ten eerste weet je zo niet zeker of je alle belangrijke acties hebt gedaan. Verder hebben programmeurs nogal eens de neiging om alleen het “happy path” te testen (de acties die wel goed gaan omdat de programmeur dit net heeft ontwikkeld). Maar gebruikers doen soms heel vreemde dingen en daar moet het programma ook mee om kunnen gaan. Denk daarbij aan foute waarden, ergens anders op het scherm klikken, een actie middenin afbreken etc.

Daarom gaan we bij het testen meer gestructureerd aan de slag. We bedenken welke verschillende acties (of soms ook waardes) mogelijk zijn en hoe het programma daarop zou moeten reageren. Als iemand bijvoorbeeld in een programma voor een school als cijfer een -1 wil invoeren, dan zou het programma hier een foutmelding moeten geven omdat dit geen geldige waarde is (dit is dan het verwachte resultaat na de actie). Een cijfer 7 zou uiteraard wel geaccepteerd moeten worden. Uitgeschreven zouden dit dus twee testcases kunnen zijn:

Testcase	Actie	Verwacht resultaat
1	-1 als cijfer invoeren	Systeem toont foutmelding: “cijfer is niet geldig, moet 1 t/m 10 zijn”
2	7 als cijfer invoeren	Cijfer wordt geaccepteerd en in systeem opgeslagen

Op die manier kan je dus alle mogelijke acties bedenken en hoe het systeem daarop zou moeten reageren. Al deze acties en verwachte resultaten beschrijf je dan in je *testplan*.
Dit kan je trouwens al doen zonder dat er ook maar een regel code is geschreven, want het gaat vooral om de gevraagde functionaliteit, en die is al bekend.

Nadat je de software (deels) hebt ontwikkeld, ga je de tests uitvoeren om te kijken hoe goed je software werkt. Je gaat hierbij dan observeren wat het daadwerkelijke resultaat is. Als die overeenkomt met wat werd verwacht, dan is die test geslaagd. De samenvatting van alle uitgevoerde tests (per testcase) zet je dan in een *testrapportage*. Hieronder een voorbeeld.

1e keer dat de test zijn uitgevoerd:

Testcase	Actie	Verwacht resultaat	Geslaagd
1	-1 als cijfer invoeren	Systeem toont foutmelding: “cijfer is niet geldig, moet 1 t/m 10 zijn”	Nee
2	7 als cijfer invoeren	Cijfer wordt geaccepteerd en in systeem opgeslagen	Ja

Na aanpassing van de code wordt opnieuw getest. De 2e uitvoer kan er dan als volgt uitzien.

Testcase	Actie	Verwacht resultaat	Geslaagd
1	-1 als cijfer invoeren	Systeem toont foutmelding: “cijfer is niet geldig, moet 1 t/m 10 zijn”	Ja
2	7 als cijfer invoeren	Cijfer wordt geaccepteerd en in systeem opgeslagen	Ja

De test is nu wel geslaagd.

Tests kun je op verschillende niveaus doen. Tijdens de eindopdracht verwachten we van je dat je op twee niveaus test:

- Methodeniveau. Elke methode die je schrijft test je meteen. Dit type test noemen we een unit test.
- Op programmaniveau. Elke programma heeft functionele eisen niet-functionele eisen. We verwachten hier dat je de functionele eisen test. Dit noemen we ook wel functioneel testen.

3.2 Debugging

Inmiddels heb je de ervaring dat jouw programma soms heel andere dingen doet dan je bedoeld had. Meestal gebeurt dit doordat je ergens een foutje hebt gemaakt in een berekening of doordat je een aantal opdrachten net in de verkeerde volgorde hebt gezet. Of misschien klopte jouw gedachten over een bepaalde oplossing gewoon helemaal niet. Dat moet je dan oplossen

Dat oplossen noemen we debuggen. Tot nu toe heb je dit waarschijnlijk gedaan met println commando’s. De meeste IDE’s hebben echter een eigen debugger-tool. Zo ook Processing. Deze tool zal niet getoetst worden, maar kan wel handig zijn om te leren gebruiken.
Onderstaande video van de Processing Foundation geeft meer uitleg over de tool en het gebruik ervan:

https://vimeo.com/140134398

Deze video staat ook op onderwijs-online.

4 EINDOPDRACHT

In deze fase gaan jullie aan de slag met de eindopdracht. Hieronder staat een overzicht van wat er van jullie wordt verwacht tijdens de oplevering.

4.1 Opleverdocument

4.1.1 Algoritmen

- Minimaal 3 van de belangrijkste oplossingsalgoritmen

4.1.2 Software Architectuur

- Architectuurplaatje
- Toelichting waar nodig

4.1.3 Reflectie op codekwaliteit

- Bespreek kort jouw code en toon aan dat die voldoet aan de gestelde code eisen die tijdens SP-B en SP-AD zijn gesteld.

4.1.4 Testrapportage

- Wij verwachten dat je de functionele eisen uit het functioneel ontwerp die bij jouw opdracht hoort test. Het testrapport moet je inleveren als onderdeel van jouw opleverdocument.
- Daarnaast moet je kunnen laten zien dat en hoe je op methodeniveau hebt getest. Dit moet minimaal tijdens het assessment blijken, maar extra punten kunnen worden verdiend als er een coverage rapport aanwezig is of als er een apart test-tabblad met methoden waarin de tests uitgevoerd worden bij de code zit.

4.2 Realisatie

- De opdracht is gerealiseerd conform het functioneel ontwerp en de daarin vermelde MoSCoW specificatie en bevat nauwelijks bugs.
- Jouw code is gebaseerd op het ontwerp en voldoet aan de kwaliteitsafspraken die we bij Gestructureerd Programmeren (SP-B en SP-AD) hebben gemaakt.

OPEN UP
NEW HAN_UNIVERSITY OF APPLIED SCIENCES
HORIZONS.