De schaakklok

Programmaontwerp voor beginners

Kris Coolsaet
Universiteit Gent

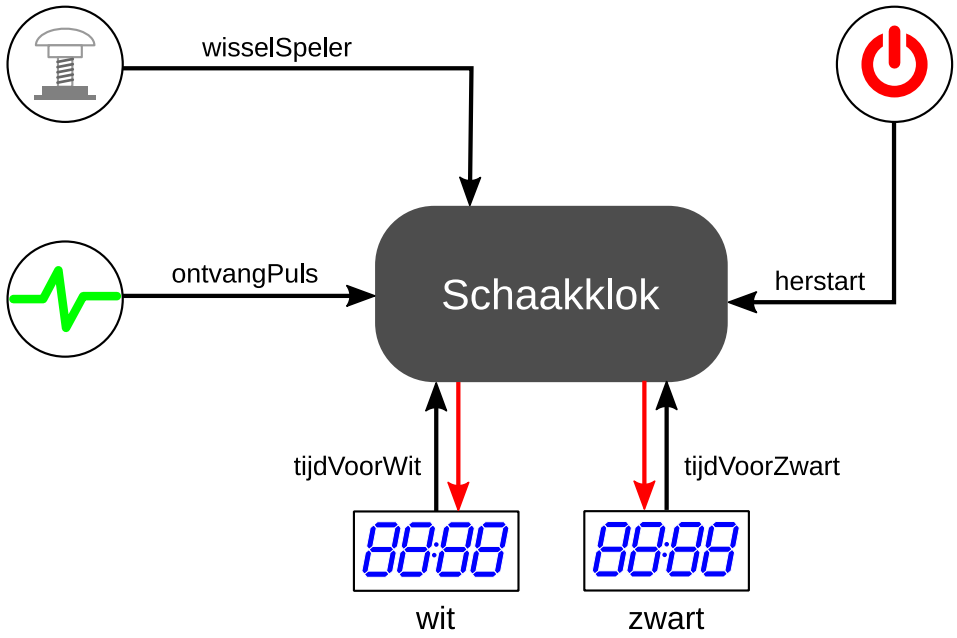
Om je een idee te geven van wat er zo allemaal komt kijken bij het schrijven van een programma, bespreken we in deze tekst hoe een programmeur te werk gaat om een *schaakklok* te simuleren.

1 Structuur van een schaakklok(-programma)

Een schaakklok zorgt ervoor dat de spelers tijdens een wedstrijd de tijd niet overschrijden die hen is toegestaan. Dit apparaat bevat *twee* klokjes, één voor elke speler. Elk klokje toont hoeveel minuten en seconden de overeenkomstige speler nog rest. De klok van de speler die aan zet is, telt elke seconde af, de klok van de andere speler staat stil. Wanneer een speler zijn zet heeft afgewerkt, drukt hij op een grote knop bovenaan de klok. Die zorgt ervoor dat zijn klok stilvalt en dat de klok van de tegenstander begint te lopen.

Het programma dat we hier ontwerpen, zal het 'brein' vormen van de schaakklok. We nemen echter niet elk detail van de schaakklok voor onze rekening: zo veronderstellen we bijvoorbeeld dat de displays van de klokjes weten hoe ze een bepaald getal moeten tonen, we hoeven dus bijvoorbeeld niet expliciet op te geven welke van de zeven segmenten er moeten oplichten om het getal 2 te laten zien. (De segmentdisplays hebben daarvoor hun eigen programma's.)

In het schema hieronder tonen we hoe ons programma *Schaakklok* met de andere delen van het apparaat is verbonden.



In het schema zie je vijf onderdelen van het toestel die ons programma beïnvloeden.

- Een knop (linksboven) die telkens wanneer hij wordt ingedrukt het programma opdraagt om van speler te wisselen. We stellen dit voor alsof de knop het programma de *opdracht* '`wisselSpeler`' geeft[1].

- Een elektronische component die elke seconde een *puls* genereert. Het programma krijgt op die manier elke seconde de opdracht '`ontvangPuls`'.

- Twee digitale klokjes, één voor de speler die met wit speelt, en één voor zwart. De klokjes kunnen aan het schaakklokprogramma de opdracht `tijdVoorWit` (of `tijdVoorZwart`) geven waarop het programma dan 'antwoord geeft' hoeveel tijd er nog rest voor de witte (of de zwarte) speler. (Deze opdracht is een beetje anders dan de vorige, want hij verwacht ook een antwoord terug.)

  Het programma geeft antwoord in de vorm van een tekenreeks die bestaat uit 5 tekens: 2 cijfers, een dubbele punt en 2 cijfers. Een dergelijke tekenreeks wordt in het jargon een *string* genoemd — het Engelse woord voor 'touwtje' (de letters zijn aaneengeregen — *strung together*).

- Een knop (bovenaan rechts) waarmee je de schaakklok kan herstarten bij het begin van een nieuwe wedstrijd. Beide klokjes komen dan op '06:00' te staan, en het 'witte' klokje begint te lopen.

Het schaakprogramma zelf bestaat ook uit een aantal afzonderlijke onderdelen, **objecten** genaamd, die met elkaar samenwerken, zoals geschetst in het schema op de volgende bladzijde.

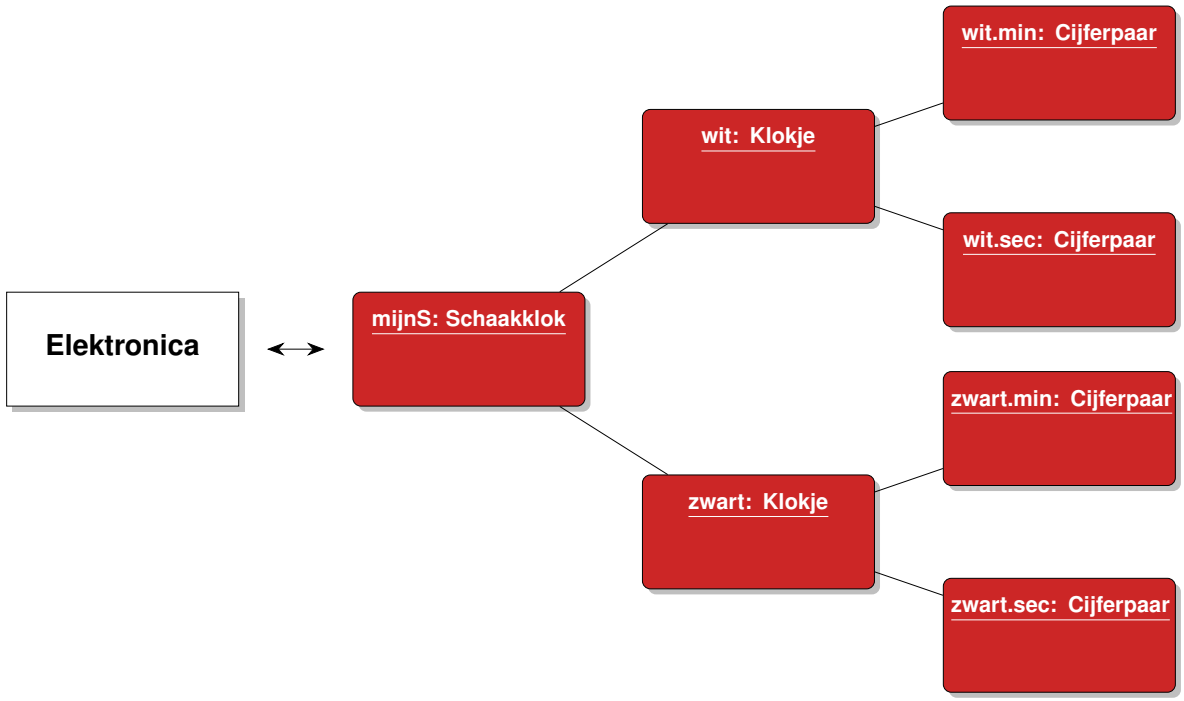
Het is de taak van het programma om de diverse opdrachten die het van de elektronica krijgt, uit te voeren en te beantwoorden, en deze taak wordt verdeeld over de verschillende objecten. Het *Schaakklok*-object gedraagt zich hierbij als manager of kapitein en geeft hoofdzakelijk opdrachten door aan de twee *Klokje*-objecten (de sergeanten) die dan op hun beurt zoveel mogelijk delegeren naar de *Cijferpaar*-objecten (het voetvolk dat het meeste werk doet).

Er zijn twee *Klokje*-objecten, één die het klokje voor de witte speler beheert, en één voor de zwarte speler, en vier *Cijferpaar*-objecten, die telkens twee cijfers van een display voor hun rekening nemen, voor de seconden en de minuten, de witte en de zwarte speler.

Zoek niet al te veel parallellen tussen de onderverdeling van het programma in

[1] Programma's hebben het vaak moeilijk met opdrachten die spaties bevatten. Daarom schrijven we de woorden aaneen, waarbij we elk nieuw woord met een hoofdletter beginnen. Het eerste woord begint echter steeds met een kleine letter.

objecten en de opsplitsing van de schaakklok in elektronische componenten. Objecten in een programma zijn immers in de eerste plaats georganiseerd op een manier die het gemakkelijk maakt om een probleem op te lossen, en dit komt niet altijd strikt overeen met de meest voor de hand liggende opsplitsing van de context waaruit het probleem voortkomt.



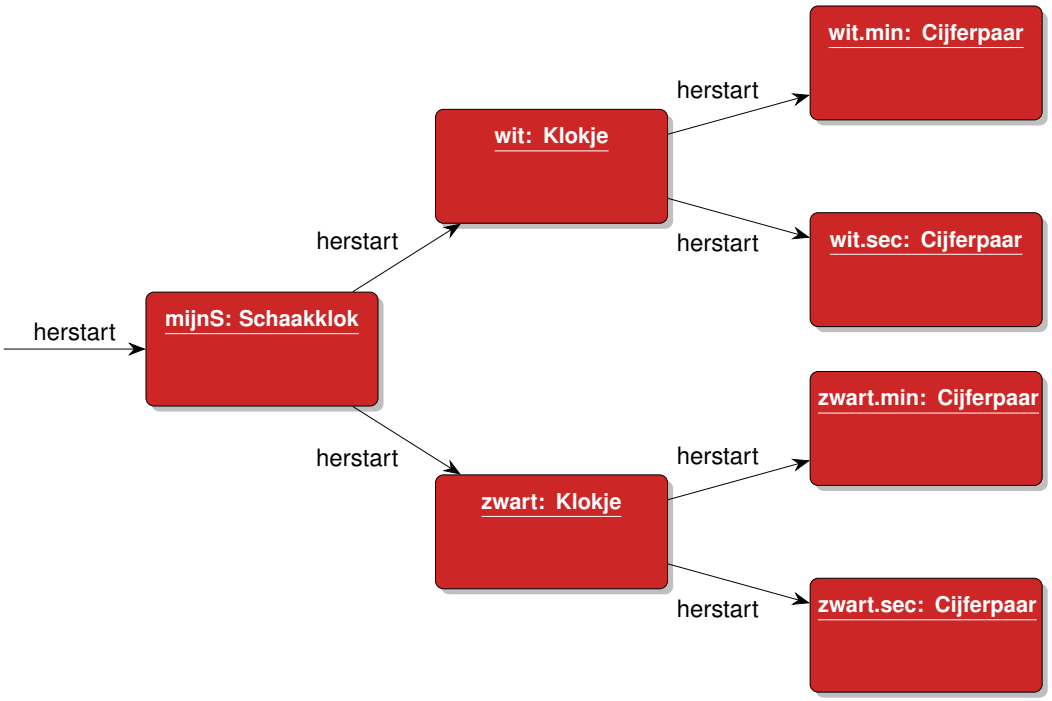
Er zijn hier inderdaad heel wat verschillen:

- In werkelijkheid bestaat het apparaat niet uit twee afzonderlijke klokjes, maar wel uit twee afzonderlijke displays die door één enkele klok (pulsgenerator) worden gestuurd. Het programma gebruikt echter twee *Klokje*-objecten.
- Een werkelijk display is niet opgesplitst in een minuut- en een secondecomponent, maar in afzonderlijke cijfers.

Misschien wat minder duidelijk, maar zeer belangrijk voor de manier waarop we over het programma nadenken, is het feit dat alle communicatie met elektronica moet gebeuren via opdrachten die aan het *Schaakklok*-object worden gegeven. Je kan een *Cijferpaar*-object bijvoorbeeld niet bevelen om een ander getal op het display te tonen, het enige wat dit object kan doen is *bijhouden* welk getal er moet getoond worden (in een stukje van het computergeheugen) en antwoorden wanneer het gevraagd wordt wat dit getal is. Later hierover meer.

1.1 Herstarten

We bekijken nu eerst hoe deze zeven objecten in de praktijk met elkaar samenwerken. De afbeelding hieronder toont wat er binnen het programma gebeurt wanneer de 'herstart'-knop wordt ingedrukt.



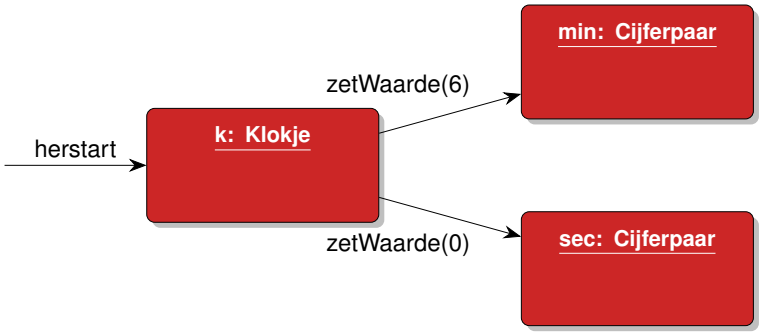
Wanneer het *Schaakklok*-object de opdracht *herstart* krijgt, dan geeft hij die, als een goede manager, gewoon door aan de twee *Klokje*-objecten. Het herstarten van een schaakklok komt er immers gewoon op neer dat alletwee de klokjes worden teruggezet naar hun beginwaarde[2].

In het jargon noemen we het geven van een opdracht aan een object, het **oproepen** van een **methode**. Wanneer we dus de methode *herstart* van de schaakklok oproepen, zal dit object op zijn beurt de methode *herstart* van elk van zijn klokjes oproepen.

De klokjes gedragen zich op een gelijkaardige manier: wanneer *herstart* van een klokje wordt opgeroepen, roept het de gelijknamige methode van de twee cijferparen op die het onder zijn beheer heeft.

[2]Zoals we straks zullen zien, is dit niet helemaal voldoende en moet er nog een kleine actie worden toegevoegd.

Het herstarten van een cijferpaar kunnen we meer gedetailleerd beschrijven: het secondenpaar moet teruggezet worden op 0 en het minutenpaar op 6. Onderstaande afbeelding is daarom een betere voorstelling van wat er in het programma gebeurt:



Dit schema is geldig voor beide klokjes, zowel voor de witte speler als de zwarte[3].

Wanneer de nieuwe methode *zetWaarde* wordt opgeroepen krijgt ze bijkomende informatie mee in de vorm van een **parameter**. In dit geval is de parameter een getal, maar ook andere soorten parameters zijn toegelaten, en een methode kan ook meer dan één parameter hebben.

Wat moet een cijferpaar nu doen wanneer de methode *zetWaarde* wordt opgeroepen? Het uiteindelijke effect moet zijn dat er een nieuw getal op het display verschijnt, maar we hebben reeds eerder uitgelegd dat het cijferpaar daar niet rechtstreeks voor kan zorgen.

In de plaats moet het cijferpaar enkel *bijhouden* wat het aantal minuten is dat op het display moet verschijnen. Objecten kunnen gegevens bijhouden in de vorm van **velden**.

In de figuur hiernaast, tonen we een *Cijferpaar*-object met een veld met de naam *teller* en een waarde 3. (Dit is bijvoorbeeld een momentopname van wanneer het display '3:14' aangeeft.)

De methode *zetWaarde* moet de inhoud van dit veld aanpassen: roepen we de methode op met als parameter 6 (t.t.z., '*zetWaarde*(6)') dan verwachten we dat na afloop het veld *teller* de waarde 6 heeft gekregen, in plaats van 3.

[3]Zoals we later zullen zien, hoeven we beide klokjes niet afzonderlijk te programmeren. Hetzelfde geldt voor de vier cijferparen.

Een object kan meer dan één veld bezitten en veldwaarden hoeven niet noodzakelijk getallen te zijn, maar voorlopig houden we het nog eenvoudig.

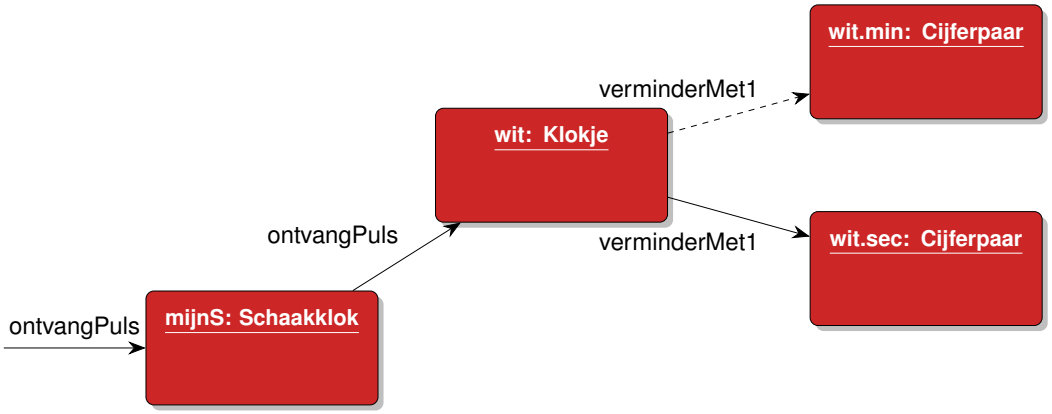
Er is nog een belangrijk detail dat we niet over het hoofd mogen zien als we willen dat de schaakklok correct herstart: we moeten vastleggen dat de witte speler nu aan zet is.

Dit doen we aan de hand van een veld *speler* in het *Schaakklok*-object. De waarde van dit veld is 1 wanneer de witte speler aan zet is, en 2 wanneer de zwarte speler aan zet is.

De methode *herstart* van het *Schaakklok*-object moet dus niet alleen de methode *herstart* oproepen van de beide klokjes, maar ook getal 1 invullen in het *speler*-veld.

1.2 Puls ontvangen

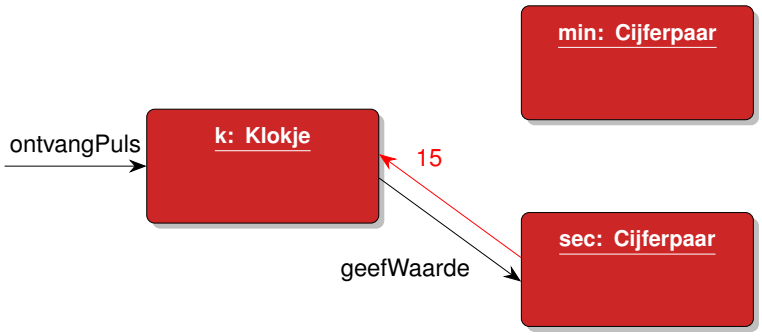
Het schema hieronder geeft aan wat er gebeurt wanneer de schaakklok de opdracht `ontvangPuls` krijgt.



Merk op dat, in tegenstelling tot bij *herstart*, de opdracht nu niet zomaar aan elk object kan worden doorgegeven. Inderdaad, slechts één van de twee klokjes mag op de puls reageren, namelijk het klokje van de speler die op dat moment aan zet is (wit, in dit voorbeeld). Het *Schaakklok*-object kan de informatie uit zijn *speler*-veld gebruiken om te weten naar welk *Klokje*-object hij de opdracht moet doorgeven.

Ook het *Klokje*-object heeft het niet zo eenvoudig. Bij elke puls moet de bijgehouden tijd met één seconde verminderen. Meestal betekent dit dat we aan het secondencijferpaar zullen vragen om de waarde van zijn *teller* -veld met 1 te verlagen — dat is waarop de `verminderMet1` in de figuur duidt — maar elke 60 seconden moet ook het minutencijferpaar de opdracht `verminderMet1` krijgen, en om te kunnen beslissen of dit nodig is, moet het *Klokje*-object weten of de secondenteller op nul staat.

Het *teller* -veld van een *Cijferpaar*-object is niet zichtbaar voor andere objecten. Het *Klokje*-object kan de waarde ervan dus enkel onrechtstreeks te weten komen, namelijk door ze expliciet aan het cijferpaar op te vragen.



In de afbeelding hierboven roept het *Klokje*-object de methode *geefWaarde* op bij het secondencijferpaar en krijgt als antwoord het getal 15 terug (de speler had bijvoorbeeld nog 2 minuten en 15 seconden te gaan). Een methode die een waarde teruggeeft, noemen we een **functie**. Een methode die enkel een opdracht uitvoert (zoals *ontvangPuls* of *zetWaarde*), noemen we een **procedure**.

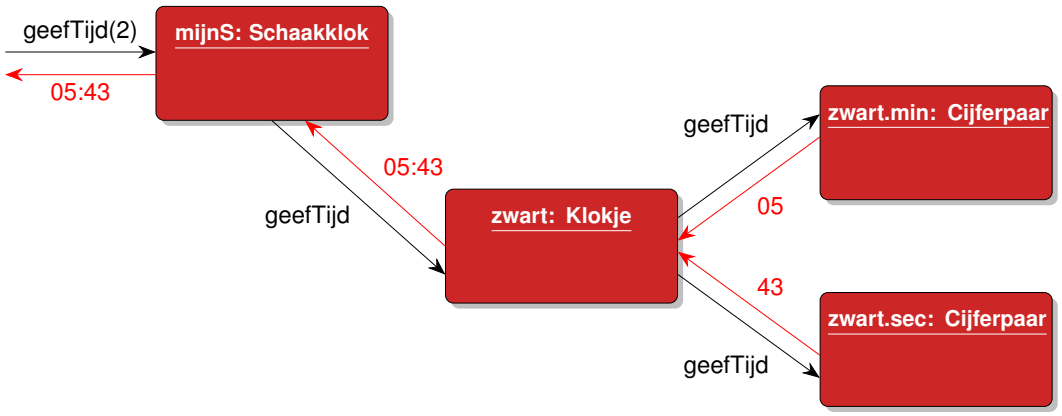
Samenvattend: wanneer de methode *ontvangPuls* van een klokje wordt opgeroepen, doet het klokje het volgende:

- Het roept eerst de functie *geefWaarde* van *sec* op (*sec* is de naam van het secondencijferpaar).
- Is het resultaat verschillend van 0, dan roept het de methode *verminderMet1* op van *sec*.
- Is het resultaat gelijk aan 0, dan roept het voor *min* (het minutencijferpaar) de methode *verminderMet1* op en voor *sec* de methode *zetWaarde* — met parameter 59.

We hebben dit geschetst in de volgende figuur.

1.3 Resterende tijd

De methoden *tijdVoorWit* en *tijdVoorZwart* lijken heel goed op elkaar, alleen gaat het telkens over het klokje van een andere speler. We kunnen daarom evengoed een methode met een parameter gebruiken: *geefTijd*(1) en *geefTijd*(2) waarbij we dezelfde conventie gebruiken als eerder: 1 betekent 'wit', 2 betekent 'zwart'. Merk op dat deze methoden *functies* zijn: we verwachten dat we als resultaat van het oproepen van deze methode een string terugkrijgen — bijvoorbeeld '05:43'.



Het schema hieronder toont hoe deze methode wordt uitgevoerd:

- De schaakklok roept de methode *geefTijd* op van het gepaste klokje. Welk klokje hij kiest, hangt af van de parameter die aan de oorspronkelijke *geefTijd* (..) is meegegeven.

- Het klokje vraagt de tijd op aan elk van de twee cijferparen, met behulp van een methode die we voor het gemak opnieuw *geefTijd* genoemd hebben[4].
- Elke cijferpaar geeft een string terug die uit twee cijfers bestaat.
- Het klokje combineert die met elkaar door er een dubbele punt tussen te plaatsen, en geeft het resultaat door aan het *Schaakklok*-object.

Uiteindelijk geeft het schaakklokprogramma dit laatste resultaat door aan de elektronica.

1.4 Van speler wisselen

De enige methode die we nog niet hebben besproken, is *wisselSpeler*. Deze is echter verrassend eenvoudig: enkel het *Schaakklok*-object zelf hoeft hierop te reageren, bij de klokjes en de cijferparen hoeft er niets te gebeuren.

Het volstaat voor het *Schaakklok*-object om de inhoud van zijn *speler*-veld te veranderen van 1 in 2, en omgekeerd.

2 Het programma — in mensentaal

In een volgende stap willen we wat we hierboven hebben geschetst, samenvatten in een gestructureerde vorm als eerste stap voor het schrijven van een programma. Dit programma zal uit drie delen bestaan: een beschrijving van wat een *Schaakklok*-object doet, wat een *Klokje*-object doet en wat een *Cijferpaar*-object doet.

[4] Welke naam een methode precies krijgt, is niet zo belangrijk. Methoden met dezelfde naam die op verschillende soorten objecten worden opgeroepen, kunnen gerust een totaal ander effect hebben. In dit voorbeeld hebben we af en toe dezelfde namen gebruikt enkel omdat het dat voor ons gemakkelijker maakt om ze te onthouden.

2.1 Klassen

We hebben eerder al opgemerkt dat er weliswaar twee klokjes zijn maar dat die eigenlijk hetzelfde gedrag vertonen: je kan er dezelfde methoden voor oproepen en die hebben hetzelfde effect. Merk bovendien op dat een klokje nergens nodig heeft te weten of het nu bij de witte speler hoort of de zwarte, beide *Klokje*-objecten zijn compleet uitwisselbaar.

Van objecten zoals deze die volledig hetzelfde gedrag vertonen[5], zeggen we dat ze behoren tot dezelfde **klasse**. We geven die klasse een naam (*Klokje*) en noemen de objecten **instanties** van de klasse.

Ook de vier cijferparen zijn instanties van één enkele klasse (*Cijferpaar*). Dat twee van die cijferparen gebruikt worden om minuten voor te stellen en de andere twee voor seconden, is hier opnieuw niet van belang. Alle vier de objecten hebben één veld *teller* en laten dezelfde methoden toe (*geefWaarde*, *zetWaarde*, *verminderMet1* en *geefTijd*).

Merk op dat om tot dezelfde klasse te behoren objecten niet alleen dezelfde methoden moeten toelaten, maar dat die methoden ook hetzelfde moeten doen. Eigenlijk benaderen we de relatie object/klasse hier van de verkeerde kant: in de praktijk zal een programmeur eerst vastleggen wat de klassen zijn die in een programma gebruikt worden en wat de methoden en velden zijn die bij die klasse horen. Pas nadat de klassen volledig zijn gespecificeerd, kan je er instanties van aanmaken.

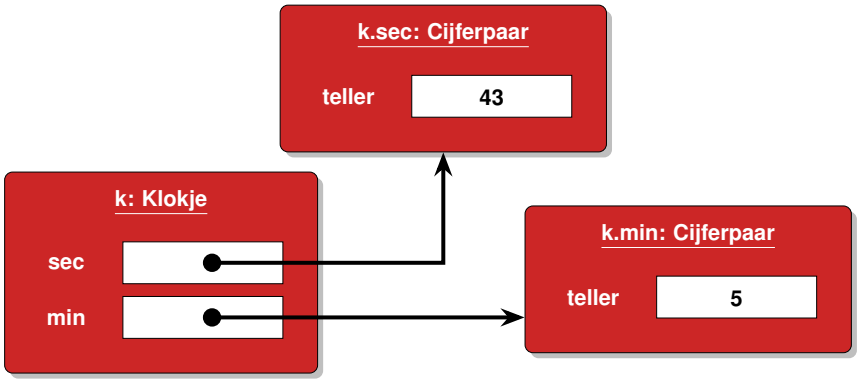
In ons voorbeeld zullen we dus drie *klassen* programmeren — zoals geschetst in onderstaand **klassendiagram** (de pijlen lees je als 'maakt gebruik van').

Bij de start van het programma wordt dan één instantie van *Schaakklok* gecreëerd. Dit object zal dan zelf twee klokjes (*Klokje*-objecten) aanmaken, één voor de witte speler en één voor de zwarte, en de klokjes zelf zullen dan elk twee cijferparen creëren (*Cijferpaar*-objecten), voor de seconden en de minuten.

[5] Objecten hoeven niet exact hetzelfde gedrag te hebben om tot dezelfde klasse te behoren. Soms hangt het verschil in gedrag slechts af van een aantal gegevens die het object 'bij zijn geboorte' meekrijgt. Een 24-uursklok en een 12-uursklok kunnen bijvoorbeeld tot dezelfde klasse behoren als we voor een methode zorgen die de bovengrens (24 of 12) aan het object meedeelt.

2.2 Objecten verwijzen naar andere objecten

Er is een belangrijk aspect van het programma waarop we nog wat dieper moeten ingaan. Hoe zijn de zeven objecten met elkaar verbonden? Hoe weet het 'witte' klokje bijvoorbeeld bij welke van de vier cijferparen hij de methode *geefWaarde* moet oproepen om 'zijn' secondental te weten te komen? Hoe maakt de schaakklok onderscheid tussen zijn twee klokjes? Het antwoord is dat objecten via hun velden naar elkaar kunnen verwijzen. Hieronder schetsen we één *Klokje*-object op het moment dat er nog 5 minuten en 43 seconden resteren.



De *Cijferpaar*-objecten hebben dezelfde structuur als voorheen — één enkel veld *teller* dat een getal bijhoudt. Het *Klokje*-object bezit echter ook twee velden waar we het vroeger nog niet over hebben gehad: een veld *sec* en een veld *min*.

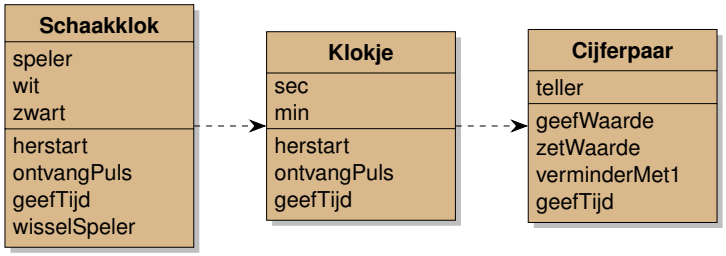
Deze velden bevatten geen getallen, maar *verwijzingen* naar *Cijferpaar*-objecten. Wanneer het klokje dus het aantal resterende minuten wil weten, zal het de methode *geefWaarde* oproepen voor het cijferpaar waarnaar zijn *min*-veld verwijst. (In het kort verwoorden wij dit ook als 'de methode *geefWaarde* van *sec*'.)

Op dezelfde manier bevat een *Schaakklok*-object dus ook twee velden *wit* en *zwart* die naar de twee klokjes verwijzen.

2.3 De implementaties

We hebben nu voldoende informatie om een begin te maken met de programmatuur. Een programma schrijven is weinig meer dan heel specifiek noteren wat de verschillende methoden van een klasse moeten doen. Hieronder doen we dit nog in natuurlijke taal, later gebruiken we hiervoor een *programmeertaal* (zie §5).

Onderstaand schema, een zogenaamd *klassendiagram*, geeft aan wat de velden zijn en de methoden van elk van de drie klassen die we zullen **implementeren** (t.t.z., waarvoor we een programma zullen schrijven). Lees de pijlen in dit diagram als 'gebruikt een object van'.



Dit zijn de drie overeenkomstige implementaties:

Cijferpaar

geefWaarde()
Geef het getal terug dat is opgeslagen in het veld *teller*.

zetWaarde(*w*)
Stop het getal *w* (de waarde van de parameter) in het veld *teller*.

verminderMet1()
Verminder de waarde die is opgeslagen in *teller* met 1.

geefTijd()
Als het getal dat opgeslagen is in *teller* groter is dan 9,
dan geef je dit getal terug (in de vorm van een string),
anders geef je dit getal terug voorafgegaan door het cijfer 0.

We volgen de conventie dat we bij methoden steeds ronde haakjes noteren, ook al zijn er geen parameters.

Klokje

herstart()
Roep *zetWaarde*(0) op voor (het object waarnaar verwezen wordt door) *sec* en roep *zetWaarde*(6) op voor *min*.

ontvangPuls()
Roep *geefWaarde*() op voor *sec*.
Als deze waarde nul is,
dan roep je *zetWaarde*(59) op voor *sec* en *verminderMet1*() voor *min*,
anders roep je *verminderMet1*() op voor *sec*.

geefTijd()
Roep *geefTijd*() op voor *sec* en voor *min*. Combineer deze strings tot één geheel door er een dubbele punt tussen te plaatsen. Geef dit resultaat terug.

Schaakklok

herstart()
Roep *herstart*() op voor zowel *wit* als *zwart*.
Plaats de waarde 1 in het veld *speler*.

geefTijd(*s*)
Als *s* gelijk is aan 1,
dan roep je *geefTijd*() op voor *wit* en geef je het resultaat terug,
anders geef je het resultaat terug van *geefTijd*() voor *zwart*.

ontvangPuls()
Als de waarde van het veld *speler* gelijk is aan 1,
dan roep je *ontvangPuls*() op voor *wit*,
anders roep je *ontvangPuls*() op voor *zwart*.

wisselSpeler()
Als de waarde van *speler* gelijk is aan 1,
dan plaats je 2 in (het veld) *speler*,
anders plaats je 1 in *speler*.

2.4 Initialisatie van de objecten

Er ontbreekt nog één belangrijke component aan de implementaties hierboven: de velden van objecten moeten op één of andere manier een initiële waarde krijgen.

Hiervoor bestaat een bijzondere soort methoden die we **constructoren** noemen. Een constructor van een object wordt automatisch (één keer) opgeroepen vlak nadat het object is aangemaakt. Het is dan de taak van de constructor om in de velden van het object een geldige waarde in te vullen. De naam 'constructor' is

enigszins misleidend: een constructor dient niet om objecten te construeren maar wordt pas opgeroepen nadat het object al is geconstrueerd. Men had beter een naam zoals 'initialisator' gekozen.

Voor de klasse *Cijferpaar* is een constructor vrij eenvoudig te implementeren:

Cijferpaar

<*constructor*>
Plaats de waarde 0 in het veld *teller*.

Welke waarde we hier in *teller* plaatsen is niet zo belangrijk — als er maar een waarde wordt ingevuld. De 'echte' beginwaarde van de teller zal uiteindelijk worden geïnitialiseerd bij het herstarten van de schaakklok[6].

Dit is de constructor van de klasse *Klokje*:

Klokje

<*constructor*>
Maak een nieuw *Cijferpaar*-object aan en plaats (een verwijzing naar) dit object in *sec*.
Maak een nieuw *Cijferpaar*-object aan en plaats het in *min*.

Het is dus zeker toegelaten (en komt in de praktijk vaak voor) dat een object als onderdeel van zijn constructor andere objecten aanmaakt (waarvoor dan op hun beurt hun constructor wordt opgeroepen). De implementatie van de constructor van *Schaakklok* doet iets analoogs:

Schaakklok

<*constructor*>
Maak een nieuw *Klokje*-object aan en plaats het in *wit*.
Plaats ook een nieuw *Klokje*-object in *zwart*.
Geef *speler* de waarde 1 (t.t.z., plaats de waarde 1 in het *speler*-veld).

[6] Het is ook mogelijk om bij de seconden- en minutenparen een andere beginwaarde in te stellen (bijvoorbeeld 6 en 0). We houden het voorlopig liever eenvoudig.

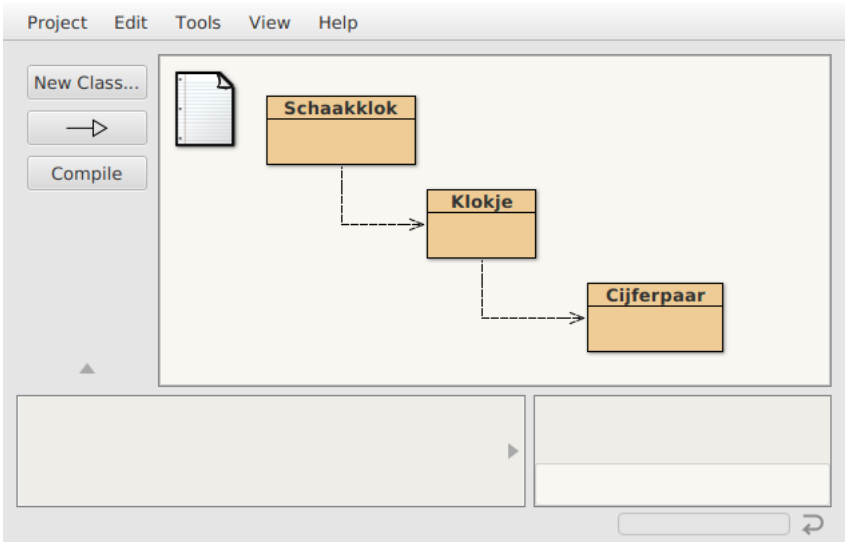
3 Uitproberen met BlueJ

Nu we een goed idee hebben van hoe het schaakklokprogramma in elkaar zit, is de tijd gekomen om het in de praktijk uit te proberen. We doen dit met behulp van **BlueJ**, een softwaretoepassing die speciaal is ontworpen om te helpen bij het aanleren van de programmeertaal Java en die tegelijk toelaat om programma's die in Java zijn geschreven te bestuderen en in detail uit te proberen.

Je kan BlueJ gratis downloaden van `http://bluej.org`. Het bijkomend materiaal voor deze tekst vind je op `http://inigem.ugent.be/sklok.html`.

Het programmeren in Java stellen we nog enkele bladzijden uit. We bestuderen eerst een programma dat reeds voor ons is voorgeprogrammeerd.

- Download *sklok.zip*, pak het bestand uit en open de resulterende map met BlueJ. Je krijgt een venster te zien met daarop het klassendiagram van het programma.

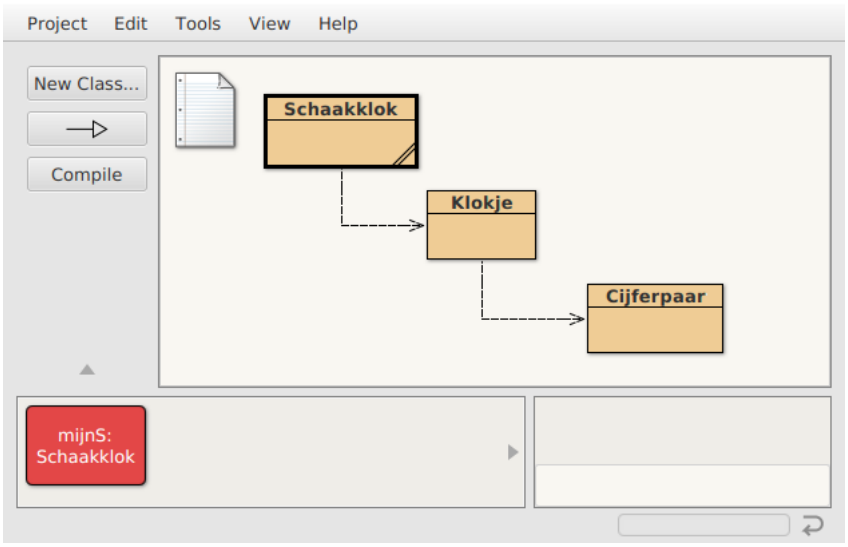


Het hoofdvenster van BlueJ toont altijd een (vereenvoudigd) klassendiagram van het programma. Je herkent de drie klassen *Schaakklok*, *Klokje* en *Cijferpaar*.

- Klik met de rechtermuisknop op het *Schaakklok*-pictogram en kies de menu-optie `new Schaakklok()`.
- Tik `mijnS` in het dialoogvenster dat verschijnt en druk OK.

Je hebt aan BlueJ de opdracht gegeven een nieuw object te maken van de klasse

Schaakklok. BlueJ toont een pictogram voor dit object onderaan het venster, in het zogenaamde *objectenvak*.

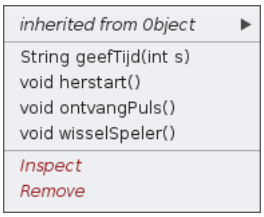


We hebben eerder al gezien dat bij het aanmaken van een *Schaakklok*-object meteen ook twee *Klokje*-objecten en vier *Cijferpaar*-objecten worden gecreëerd. BlueJ plaatst deze zes bijkomende objecten echter niet automatisch in het objectenvak en ze krijgen ook geen naam.

Besef goed dat objecten in een programma in de regel onzichtbaar zijn — ze zijn niet veel meer dan een stukje computergeheugen waarop je bepaalde bewerkingen doet. Wat BlueJ in het objectenvak plaatst, is niet het object zelf, maar een visueel anker dat je toelaat om met het echte object om te gaan.

- Klik met de rechter muisknop op (het pictogram van het object met de naam) *mijnS*.

Je herkent in het menu duidelijk de namen van de vier methoden van de klasse *Schaakklok* (negeer voorlopig de 'String', 'int' en 'void').



Door zo'n menu-optie te selecteren, vraag je aan BlueJ om de overeenkomstige methode op te roepen voor het object waarop je hebt geklikt.

- Roep de methode *herstart* op van het object *mijnS*.
- Roep tweemaal de methode *ontvangPuls* op.
- Roep de methode *geefTijd* op. Er verschijnt een venstertje waarin je de parameter *s* kan invullen. Tik 1 (voor de witte speler).

De methode *geefTijd* geeft een waarde terug (het is een functie, niet een procedure). BlueJ opent een venster waarin je deze waarde kan zien. In dit geval is dit de string[7] `"05:58"`.

- Roep de methode *wisselSpeler* op en vier keer de methode *ontvangPuls*. Zijn de waarden die *geefTijd* teruggeeft correct, zowel voor de witte als de zwarte speler?

Op deze manier kunnen we BlueJ gebruiken om een programma te testen. In bovenstaande voorbeelden hebben we de elektronica van de schaakklok gesimuleerd door met een object van de 'hoofdklasse' *Schaakklok* te interageren.

In de praktijk wordt er echter geen onderscheid gemaakt tussen 'hoofd'- en 'bij'-klassen. Met andere woorden, we kunnen in BlueJ evengoed objecten aanmaken van de andere klassen en daar methoden voor oproepen.

- Maak een nieuw *Cijferpaar*-object aan door rechts te klikken op de *klasse Cijferpaar*. Noem dit object *cp1*.
- Maak een tweede *Cijferpaar*-object aan en noem dit *cp2*.
- Verwijder het object *mijnS* door rechts te klikken en de optie *Remove* te kiezen[8].

Onthouden! Nieuwe objecten maak je door rechts te klikken op een *klasse*. Methoden voer je uit door te klikken op het betreffende *object*.

- Voer *zetWaarde*(9) uit op *cp1* en voer *zetWaarde*(15) uit op *cp2*.
- Roep *geefWaarde*() op voor *cp1*. Welke waarde verwacht je te zien, 9 of 15?
- Wat verwacht je te zien als je *geefTijd*() toepast op *cp1*? Probeer dit uit.

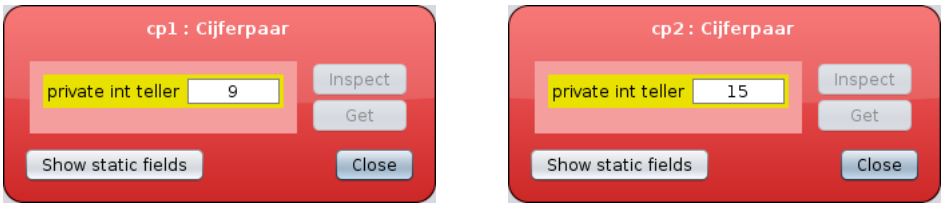
[7] We zullen vanaf nu stringwaarden altijd afdrukken tussen dubbele aanhalingstekens, onder andere om onderscheid te maken tussen getallen zoals 17 en strings zoals `"17"` die toevallig enkel uit cijfers bestaan. Aanhalingstekens helpen ook om aan te geven of een string al dan niet vooraan of achteraan spaties bevat.

[8] In deze tekst gebruiken we de Engelse versie van BlueJ. Je kan echter ook Nederlands instellen als taal via de menukeuze *Tools* | *Preferences* | *Interface*. Het is ook mogelijk om Nederlands als defaultinstelling te configureren. We verwijzen naar de documentatie van BlueJ voor meer informatie.

Elk object heeft zijn eigen stukje computergeheugen om de nodige informatie bij te houden. (In dit geval hebben *cp1* en *cp2* dus elk hun eigen *teller*.) Dit betekent dat een methode die je uitvoert op één object van een klasse geen effect heeft op de andere objecten van die klasse.

In BlueJ kan je ook de interne structuur van een object bekijken.

- Klik rechts op *cp1* en selecteer *Inspect* (rode optie onderaan). Wat is de waarde van het *teller*-veld?
- Sluit het inspectievenster en doe hetzelfde voor *cp2*.



- Roep de methode *verminderMet1* op voor *cp1*.
- Inspecteer *cp1*. Wat is nu de waarde van het veld *teller*?

- Maak een nieuw klokje aan en roep *herstart* op voor dat klokje.
- Haal het inspectievenster op voor dit klokje. Wat zijn de velden die getoond worden en wat is hun waarde?

Zoals we in paragraaf §2.2 hebben uitgelegd, bevat een klokje twee velden *sec* en *min* die elk naar een *Cijferpaar*-object *verwijzen*. Daarom vult BlueJ twee pijlen in de vakjes in en geen expliciete waarden.

- Dubbelklik op de pijl voor het veld *min*. Wat zie je?

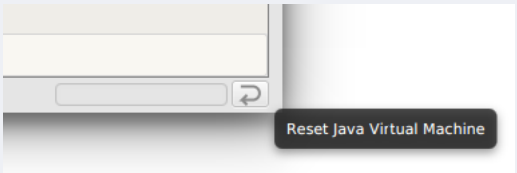
4 Opdrachten in tekstvorm

In paragraaf §2 hebben we het programma voor de schaakklok reeds geschetst in eenvoudige mensentaal. Computers zijn echter nog steeds niet slim genoeg om natuurlijke taal te begrijpen en daarom moeten we het programma meer formeel specificeren, in wat men een *programmeertaal* noemt.

Er bestaan verschillende programmeertalen, en elke taal heeft zo zijn aanhangers en tegenstanders, maar de basisprincipes zijn altijd min of meer dezelfde. Wij zullen de programmeertaal **Java** gebruiken, de taal waarop ook BlueJ is gebaseerd. Zoals je zult zien, biedt BlueJ heel wat hulp om in Java te leren programmeren.

We hernemen het eerste voorbeeld uit paragraaf §3 maar voegen er iets aan toe.

- Tik CTRL-SHIFT-R om het objectenvak leeg te maken[9]. Je bekomt hetzelfde effect door op de knop te klikken helemaal rechts onderaan het BlueJ-venster.



- Maak een nieuw *Schaakklok*-object aan en noem dit *mijnS*.
- Selecteer in het hoofdvenster van BlueJ de menu-optie *View* | *Show Terminal* (CTRL-T). Er verschijnt een 'terminalvenster'.
- Selecteer in dit venster de menu-optie *Options* | *Record method calls* en schakel de menu-optie *Options* | *Clear screen at method call* uit.
- Doe nu verder zoals in het begin van paragraaf §3: roep de methode *herstart* op, tweemaal *ontvangPuls*, *geefTijd*(1), *wisselSpeler*, *ontvangPuls* (vier keer) en tenslotte *geefTijd*(2).

Het terminalvenster bevat nu de volgende lijnen:

```
mijnS.herstart();
mijnS.ontvangPuls();
mijnS.ontvangPuls();
mijnS.geefTijd(1)
    returned String "05:58"
mijnS.wisselSpeler();
```

[9] Op Mac OS/X gebruik je COMMAND (⌘) in plaats van CTRL.

```
mijnS.ontvangPuls();
mijnS.ontvangPuls();
mijnS.ontvangPuls();
mijnS.ontvangPuls();
mijnS.geefTijd(2)
    returned String "05:56"
```

Voor dit 'verslag' gebruikt BlueJ met opzet dezelfde notatie die Java ook gebruikt om een methode op te roepen, behalve bij de twee lijnen die met `returned` beginnen: deze geven gewoon het resultaat weer van de functie-oproep uit de lijn ervoor.

Om een methode op te roepen voor een object, schrijf je dus de naam neer van dit object, gevolgd door een punt, gevolgd door de naam van de methode met haakjes en eventuele parameters. Tot slot komt er een kommapunt om de opdracht af te sluiten.

We kunnen dit soort tekstuele opdrachten ook rechtstreeks uitproberen:

- Rechts van het objectenvak is er een *evaluatievak*. (Met CTRL-E kan je dit laten verdwijnen en verschijnen.)
- Tik in dit vak de tekst '`mijnS.herstart();`' in en druk ENTER. Vergeet de kommapunt niet op het einde van de opdracht.
- Tik daaronder '`mijnS.geefTijd(1)`', *zonder* kommapunt, en druk opnieuw ENTER.

Het resultaat ziet er nu zo uit:

- Maak een *Cijferpaar*-object aan met de naam *sec* en één met de naam *min*.
- Gebruik het *evaluatievak* om de waarde van *sec* in te stellen op 0, en van *min* op 6.
- Controleer of de waarden correct zijn ingesteld. Je kan dit op verschillende manieren doen, door inspectievensters te openen of door *geefWaarde* op te roepen voor die objecten.

De twee opdrachten die je in het evaluatievak hebt ingetikt, doen wat we verwachten dat de *herstart*-methode doet van de klasse *Klokje*. Ze zijn de Java-implementatie van deze methode en vormen een onderdeel van het volledige programma voor deze klasse.

- Dubbelklik op het pictogram van de klasse *Klokje* in het hoofdvenster van BlueJ. Daardoor opent een zogenaamd *editeerpaneel* met daarin het volledige Java-programma voor deze klasse.
- Vind je in dit paneel de twee lijnen terug die je daarnet in het evaluatievak hebt ingetikt?

Inderdaad, ergens halverwege lees je het volgende:

```
public void herstart() {
    sec.zetWaarde(0);
    min.zetWaarde(6);
}
```

Dit is de vertaling in Java van wat we in paragraaf §2 hadden opgeschreven:

herstart()
Roep *zetWaarde*(0) op voor (het object waarnaar verwezen wordt door) *sec* en roep *zetWaarde*(6) op voor *min*.

Ook in de rest van de code (en de code van de twee andere klassen) zal je wellicht heel veel herkennen van het programma dat we eerder in mensentaal hadden opgesteld. Vooraleer we dit echter in detail bekijken, is het nuttig om op voorhand enkele principes kort te overlopen.

5 Programmeren in Java

5.1 Definitie van een klasse

Een Java-toepassing bestaat steeds uit een aantal klassen die elk individueel worden geprogrammeerd. Het 'programma' voor een klasse — voortaan noemen we dit een **klassendefinitie** — bevindt zich een afzonderlijk bestand, en wordt in BlueJ getoond in een apart editeerpaneel.

Een klassendefinitie heeft de volgende algemene structuur:

```
public class Naam {

    // declaraties van velden
    ...

    // definities van constructoren
    ...

    // definities van methoden

}
```

In het editeerpaneel van *Klokje* vind je deze structuur gemakkelijk terug. Een klassendefinitie begint altijd met '**public class**' en daarna de naam van de klasse (steeds met een hoofdletter). Dit wordt gevolgd door de rest van het programma, omsloten door accolades '{..}'. Java gebruikt accolades op heel wat plaatsen om daarmee allerhande elementen te groeperen[10].

Als je het programma van de klasse *Klokje* bekijkt, zie je ook dat tussen de Java-code heel wat Nederlandse tekst te lezen valt (bij BlueJ in grijs en blauw afgebeeld). We noemen dit **commentaar**: tekst die niet door de computer bekeken wordt, maar dient om het programma voor de menselijke lezer te verduidelijken. Er bestaan twee soorten commentaar:

- Uitleg bij klassen en methoden begint met '/∗∗' en eindigt met '∗/' en kan meerdere regels beslaan.
- Ander commentaar begint met '//' en loopt verder tot het einde van de lijn.

[10] In een programma betekenen ronde haakjes '(..)', vierkante haakjes '[..]' en accolades '{..}' iets totaal verschillend. Je kan ze dus niet vrijelijk door elkaar gebruiken zoals in de wiskunde.

5.2 Declaraties van velden

De computer moet van elke klasse weten welke velden haar objecten hebben. Niet alleen moet je de *namen* van die velden vastleggen, maar ook hun *type*: zal het veld een getal bevatten, of een string, of een verwijzing naar één of ander object? Het type van een veld ligt vast: je kan in hetzelfde veld niet eerst een getal stoppen en dan later een string.

We doen dit met zogenaamde **declaratie** van het veld. Hieronder drukken we de declaraties af van de velden van *Schaakklok*:

```
private int speler; // 1 voor wit, 2 voor zwart

private Klokje wit;

private Klokje zwart;
```

Een declaratie heeft een eenvoudige vorm:

- Eerst komt het woord **private**[11].
- Dan komt het **type** van het veld. Het type **int** geeft aan dat het veld enkel (gehele) getallen kan bevatten. Het type *Klokje* verklaart dat je in dit veld een (verwijzing naar een) *Klokje*-object kan opslaan.
- Tot slot komt de naam. Je kan de naam min of meer vrij kiezen. Hij mag echter geen spaties bevatten en moet steeds beginnen met een kleine letter.
- Na dit alles mag je niet vergeten een kommapunt te plaatsen.

Het type **int** is bijzonder omdat velden van dit type geen verwijzingen bevatten maar de gegevens rechtstreeks opslaan. We noemen dit een *primitief* type. De naam van dit type begint met opzet met een kleine letter omdat we het niet met de naam van een klasse zouden verwarren.

- Zoek de veldendeclaraties in de programmacode van *Klokje* en *Cijferpaar*. Wat zijn de namen en de types van deze velden?

[11] Het is te vroeg om uit te leggen wat er zo 'privé' is aan een veld. Voorlopig spreken we af dat we **private** gebruiken voor velden en **public** voor methoden en klassen.

5.3 Definities van methoden — procedures

Op bladzijde 22 hebben we reeds de definitie afgedrukt van de methode *herstart* van *Klokje*. Hieronder staat de definitie van de methode *herstart* van *Schaakklok*. (Je kan die natuurlijk ook zelf in BlueJ opzoeken.)

```
public void herstart() {
    wit.herstart();
    zwart.herstart();
    speler = 1;
}
```

De definitie van een methode bestaat uit twee delen:

- Een **hoofding** waar onder andere de naam van de methode te zien is,
- Het **corpus** van de methode (tussen accolades) dat beschrijft hoe de methode moet worden uitgevoerd.

De definitie van een *procedure* begint steeds met **public void**, gevolgd door de naam van de methode, en dan tussen haakjes de parameterdeclaraties (zie later). (*Functie*methoden komen aan bod in paragraaf §5.5.)

Het corpus van een methode bevat een reeks opdrachten die één na één moeten worden uitgevoerd. Er bestaan verschillende soorten opdrachten. De eerste twee lijnen van het corpus van *herstart* hierboven bevatten een **methode-oproep**, zoals we die al vaak gebruikt hebben.

De derde lijn bevat een zogenaamde **toewijzing** — een opdracht van de vorm '*naam* = *waarde*;'. Hiermee geef je aan dat je de waarde uit het rechterlid wil opslaan in het veld waarvan je de naam aan de linkerkant hebt opgegeven. (We stoppen hier dus het getal 1 in het veld *speler*.)

Dat Java (net zoals vele andere programmeertalen) een gelijkheidsteken gebruikt voor een toewijzing, was een ongelukkige keuze. (Wat wordt er bedoeld met '$a = b$'?) Een notatie met een pijltje, zoals '$a \Leftarrow b$', had veel verwarring kunnen besparen. Onthoud dus dat bij een toewijzing de waarde van de *rechter*kant wordt opgeslagen in het veld van de *linker*kant. Lees '=' steeds als 'wordt' (a wordt b, *speler* wordt 1, ...).

De rechterkant kan ook meer bevatten dan één enkel woord, zoals bij de volgende methodedefinitie uit *Cijferpaar*:

```
public void verminderMet1() {
    teller = teller − 1;
}
```

Aan de rechterkant van het gelijkheidsteken staat nu een **uitdrukking** die door de computer moet worden uitgerekend. De waarde van deze uitdrukking wordt dan gestopt in het veld aan de linkerkant.

We nemen dus de waarde die in *teller* is opgeslagen, trekken er één van af en stoppen dit resultaat in het veld *teller* (waarmee we de oorspronkelijke waarde meteen overschrijven).

Dit toont trouwens nogmaals dat de keuze van het gelijkheidsteken voor een toewijzing niet zo'n goed idee was. Het lijkt erop alsof we hier beweren dat *teller gelijk* is aan *teller* − 1, een wiskundige onmogelijkheid. Er staat echter *teller wordt teller* − 1, of *teller* ⇐ *teller* − 1.

Wanneer een methode parameters heeft, moet je die *declareren* in de hoofding. Net zoals bij velden geeft je zowel het type op als de naam (maar dan zonder **private**). Bekijk bijvoorbeeld de definitie van *zetWaarde* in *Cijferpaar*:

```
public void zetWaarde(int w) {
    teller = w;
}
```

Het effect van *zetWaarde* is dus om de waarde van de parameter *w* op te slaan in het veld *teller*.

5.4 Selectie

Naast de methode-oproep en de toewijzing voorziet Java ook nog enkele andere soorten opdrachten met een meer ingewikkelde structuur. Het schaakklokprogramma maakt bijvoorbeeld ook gebruik van een **selectie** — wat we ook een 'als-dan-anders-' of 'if-then-else'-opdracht kunnen noemen (naar het Engels).

Bij een selectie kiest een programma voor één van twee verschillende (reeksen) opdrachten, afhankelijk van of een bepaalde conditie is voldaan. Bijvoorbeeld:

ontvangPuls()
Als de waarde van het veld *speler* gelijk is aan 1, (**conditie**)
dan roep je *ontvangPuls*() op voor *wit* (**eerste optie**),
anders roep je *ontvangPuls*() op voor *zwart* (**tweede optie**) .

In Java wordt dit:

```
public void ontvangPuls() {
    if (speler == 1) {
        wit.ontvangPuls();
    } else {
        zwart.ontvangPuls();
    }
}
```

De algemene structuur van een selectie is de volgende:

```
if (conditie) {
    opdracht;
     ...
    opdracht;
} else {
    opdracht;
     ...
    opdracht;
}
```

Deze opdracht wordt uitgevoerd op de volgende manier:

- Kijk of de gegeven *conditie* waar is,
- Zo ja, voer dan de eerste reeks opdrachten uit (maar niet de tweede),
- Anders, voer de tweede reeks opdrachten uit (en sla de eerste reeks over).

De conditie moet je steeds tussen ronde haakjes noteren, de opdrachtreeksen tussen accolades. Het is ook de gewoonte om deze opdrachtreeksen wat te laten inspringen ten opzichte van de **if**–**else** die errond staat.

Merk op hoe we de conditie 'de inhoud van *speler* is gelijk aan 1' hebben genoteerd: we gebruiken een *dubbel* gelijkheidsteken in plaats van een enkel (zoals in de

wiskunde). Het enkel gelijkheidsteken heeft reeds de betekenis 'wordt' gekregen, Java had dus een andere notatie nodig voor 'is'. Nog maar eens een bevestiging van een ongelukkige keuze...

Je vindt ook een selectie in de definitie van *wisselSpeler* uit *Schaakklok*:

```
public void wisselSpeler() {
    if (speler == 1) {
        speler = 2;
    } else {
        speler = 1;
    }
}
```

Met een 'wiskundig' trucje had dit ook korter gekund[12].

```
public void wisselSpeler() {
    speler = 3 - speler;
}
```

En een derde voorbeeld van een selectie zie je in de definitie van *ontvangPuls* van *Klokje*:

```
public void ontvangPuls() {
    if (sec.geefWaarde() == 0) {
        sec.zetWaarde(59);
        min.verminderMet1();
    } else {
        sec.verminderMet1();
    }
}
```

De conditie die we hier gebruiken, vraagt misschien wat bijkomende uitleg. Ze bevat namelijk een methode-oproep van een functiemethode. Wat de computer hier doet is het volgende:

[12] Maar is dit ook 'beter'? Beginnende programmeurs hebben wel eens de neiging om programma's in zo weinig mogelijk lijnen te willen schrijven. Vaak is dit contraproductief. Het is belangrijk dat het ook duidelijk blijft voor de menselijke lezer wat een (deel van een) programma doet.

- Ze roept de methode *geefWaarde* op voor het object *sec*. Deze methode geeft een getal terug.
- Dit getal wordt nu vergeleken met de 0 in het rechterlid.

De methode-oproep van een functie wordt dus in zekere zin gewoon vervangen door de waarde die ze teruggeeft, net zoals bij een wiskundige functie zoals een cosinus of logaritme.

- Maak in BlueJ een cijferpaar *cp* aan en geeft het de waarde 33.
- Tik in het evaluatievak de uitdrukking '*cp.geefWaarde*() + 12'. Wat is het resultaat?
- Is de waarde van *cp* nu ook veranderd?
- Tik in het evaluatievak de uitdrukking '*cp.geefWaarde*() == 12' (met dubbel gelijkheidsteken). Wat is hiervan het resultaat?

5.5 Definities van methoden — functies

Tot nog toe hebben we enkel methodedefinities bekeken van procedures, methoden die iets doen (waarden veranderen van velden bijvoorbeeld) maar geen resultaat teruggeven. Methodedefinities van functies zijn niet veel anders, maar doordat ze een waarde teruggeven, zijn er enkele belangrijke verschillen:

- In de hoofding van de functie moet je aangeven wat het type is van de waarde die de methode teruggeeft. Dit type schrijf je waar bij procedures **void** staat.
- Om de waarde effectief terug te geven heb je een **return**-opdracht nodig als laatste opdracht van de methode.

We gebruiken voortaan de technische term **retourneren** in plaats van teruggeven (Engels: *to return*). We spreken ook van het **retourtype** van een functie (Engels: return type).

Bekijk bijvoorbeeld de eenvoudige definitie van *geefWaarde*() uit *Cijferpaar*:

```
public int geefWaarde() {
    return teller;
}
```

De opdracht '**return** *teller*;' retourneert de inhoud van het *teller*-veld en aangezien dit een geheel getal is, staat er een **int** in de hoofding als retourtype.

Een functiemethode kan ook meer ingewikkelde opdrachten bevatten, zoals de selectie in de definitie van *geefTijd* uit *Schaakklok*:

```
public String geefTijd(int s) {
    if (s == 1) {
        return wit.geefTijd();
    } else {
        return zwart.geefTijd();
    }
}
```

De **return**-opdrachten retourneren nu de waarde van een functiemethode-oproep van een andere functie[13]. De functie *geefTijd* van *Klokje* geeft een string terug, dus het retourtype is *String* (met hoofdletter — *String* is de klasse waartoe alle strings behoren).

De methode *geefTijd* uit *Klokje* heeft ook *String* als retourtype:

```
public String geefTijd() {
    return min.geefTijd() + ":" + sec.geefTijd();
}
```

In Java kan je met '+' niet alleen getallen optellen, maar ook strings 'aan elkaar plakken' (het officiële woord is *concateneren*).

Diezelfde '+' kan ook dienen om getallen en strings met elkaar te combineren, zoals in de definitie van *geefTijd* uit *Cijferpaar*:

```
public String geefTijd() {
    if (teller > 9) {
        return "" + teller;
    } else {
        return "0" + teller;
    }
}
```

[13] De functie *geefTijd* van de klasse *Klokje* is een *andere* functie dan de gelijknamige functies uit *Schaakklok* en *Cijferpaar*. Ook al hebben ze dezelfde naam, ze behoren tot een andere klasse. Ze hebben trouwens ook niet hetzelfde aantal parameters.

In de eerste **return**-opdracht gebruiken we een trucje om van een getal een string te maken: we concateneren dat getal met een lege string, t.t.z., een string zonder lettertekens.

Onderstaande programmacode is *niet* toegestaan:

```
public String geefTijd() {
    if (teller > 9) {
        return teller; // niet toegestaan!
    } else {
        return "0" + teller;
    }
}
```

De **return**-opdracht geeft *teller* terug, van het type **int**, terwijl we met het retourtype *String* in de hoofding verklaren een string te zullen retourneren.

5.6 Definities van constructoren

We hebben nu ongeveer alles omgezet naar Java wat we in paragraaf §2 in mensentaal hebben geschetst. Alleen de constructoren ontbreken nog.

De definitie van een constructor heeft dezelfde vorm als die van een methode, alleen de hoofding is verschillend:

- Er staat *geen* type of **void**.
- De naam van de constructor is dezelfde als de naam van de klasse, met hoofdletter.

Dit is bijvoorbeeld de constructor van *Cijferpaar*:

```
public Cijferpaar() {
    teller = 0;
}
```

Het corpus van een constructor bevat alle opdrachten die nodig zijn om de velden van de klasse een beginwaarde te geven.

Bij *Klokje* moeten er twee *Cijferpaar*-objecten aangemaakt worden (nieuwe objecten maak je aan met **new**):

```
public Klokje() {
    sec  = new Cijferpaar();
    min = new Cijferpaar();
}
```

De constructor van *Schaakklok* volgt hetzelfde stramien:

```
public Schaakklok() {
    wit = new Klokje();
    zwart = new Klokje();
    speler  = 1;
}
```