Samenvatting

Leren programmeren is uitdagend en aldus ervaren veel studenten programmeervakken als moeilijk. Programmeeronderwijs is geen uitzondering op het spreekwoord oefening baart kunst. Het is algemeen aanvaard dat het doen de beste manier is om te leren programmeren: hoe meer ervaring, hoe beter de programmeerkunst. Om evenwel iets te leren van al die programmeerervaring is het belangrijk dat studenten op tijd voldoende kwalitatieve feedback krijgen.

Jammer genoeg is net het geven van die feedback heel tijdrovend en arbeidsintensief, zeker als er veel oefeningen en grote aantallen studenten zijn. Enerzijds moeten studenten dus zoveel mogelijk programmeren, maar anderzijds is er weinig tijd om goede feedback te voorzien. Daarom is er een lange en rijke geschiedenis (sinds de jaren 1960) van het gebruik der automatisering om feedback te geven. Het proces om feedback op geautomatiseerde wijze te geven heet geautomatiseerde beoordeling (van het Engelse *automated assessment*).

Lang is relatief natuurlijk, maar wel gepast wetende dat leren programmeren zelf in de jaren 1960 opkwam.

In de meeste gevallen houdt geautomatiseerde beoordeling voor programmeeronderwijs in dat men werkt met testraamwerken voor software. De code die studenten voor een bepaalde oefeningen indienen (we noemen dit een oplossing) wordt getest, en dat minstens op juistheid. Vaak is de feedback wel veel uitgebreider dan enkel een globale juist of fout.

Onze vakgroep heeft, zoals zovele anderen, een online platform gemaakt voor geautomatiseerde beoordelingen: Dodona. Een belangrijke eigenschap is de scheiding tussen het platform zelf (verantwoordelijk voor gebruikersbeheer, cursusbeheer, de gebruikersinterface, enz.) en de *judge* (het testraamwerk verantwoordelijk voor het beoordelen van oplossingen). Zo kan Dodona bijna elke programmeertaal ondersteunen: momenteel is er ondersteuning voor C, Haskell, Java, Kotlin, Prolog, R, Scheme, Bash, C#, JavaScript, Python, HTML, SQL, Markdown, en Turtle.

Tijdens het werken aan en met Dodona stelden we enige tekortkomingen vast in bestaande hulpmiddelen die gebruikt worden in het programmeeronderwijs. Hoofdstuk 1 geeft een gedetailleerd overzicht van de

onderwijscontext en van het Dodona-platform. Samengevat behandelt dit proefschrift vijf van die waargenomen tekortkomingen.

We merkten dat veel oefeningen in Dodona geschikt zijn om te gebruiken in meerdere programmeertalen, althans in theorie. Om een oefening daadwerkelijk te gebruiken in een andere programmeertaal, moet men ze eerst kopiëren, dan handmatig het testplan omzetten naar het formaat dat de judge voor die programmeertaal gebruikt, en ten slotte nog de configuratiebestanden en opgave aanpassen. Dit is veel handwerk. Hoofdstuk 2 biedt een oplossing: **TESTed**, een educatief testraamwerk voor software. Kenmerkend aan TESTed is de mogelijkheid om programmeertaalonafhankelijke oefeningen te schrijven. Dit wil zeggen dat één oefening (met één enkel testplan) opgelost kan worden in meerdere programmeertalen, met ondersteuning voor geautomatiseerde beoordeling. Een oefeningen is dus bruikbaar in meerdere programmeertalen zonder enige bijkomende inspanning.

Met een prototype van TESTed in de hand namen we dan een stapje terug om naar het grote geheel te kijken: wat is er nodig om van een prototype naar een goede oplossing voor het maken van programmeeroefeningen te gaan? We willen TESTed de standaardoptie maken voor lesgevers, in zowel hoger als secundair onderwijs. Hiervoor hebben we **TESTed-DSL** in het leven geroepen, dat we voorstellen in hoofdstuk 3. Het is een domeinspecifieke taal om oefeningen met ondersteuning voor geautomatiseerde beoordeling in meerdere programmeertalen te schrijven. Een domeinspecifieke taal is een formaat dat specifiek ontworpen is voor een bepaald gebruik, wat hier het schrijven van programmeeroefeningen. Door aandacht te besteden aan de ergonomische kant van TESTed-DSL, hebben we ervoor gezorgd dat de taal ook nuttig is voor oefeningen die niet bedoeld zijn om gebruikt te worden in meerdere programmeertalen is. We raden nu alle lesgevers op Dodona aan om TESTed te gebruiken voor het opstellen van oefeningen, zelfs als ze oefeningen willen maken die bijvoorbeeld enkel in JavaScript moeten opgelost worden.

Bij jonge kinderen gebruikt men vaak visuele programmeertalen om te leren programmeren. Een visuele programmeertaal laat gebruikers toe om programma's te maken door stukken van het programma niet tekstueel maar grafisch te manipuleren. Scratch is binnen het onderwijs veruit de meestgebruikte visuele programmeertaal. Programmeren in Scratch bestaat uit het slepen en in elkaar klikken van blokjes (een beetje zoals puzzelstukjes of legoblokjes). Vandaar dat men Scratch ook wel een blokgebaseerde programmeertaal noemt. Een gedetailleerde inleiding over Scratch staat in hoofdstuk 4.

Dodona ondersteunt meerdere programmeertalen, dus oorspronkelijk wilden we ondersteuning voor Scratch toevoegen aan Dodona. Echter,

Scratch is niet alleen een programmeertaal, het is ook een programmeeromgeving. Het werd snel duidelijk dat een platform voor Scratch andere vereisten heeft dan wat we met Dodona konden doen. Daarom gingen we een samenwerking aan met CodeCosmos, een commerciële partner. Aangezien CodeCosmos een educatieve uitgeverij is, die ook oefeningen voor Scratch aanbiedt, heeft ze al een platform voor Scratch. Bovendien heeft ze ook meer ervaring met het maken van oefeningen voor Scratch.

Hoofdstuk 5 stelt **Itch** voor, ons testraamwerk voor Scratch. Het ondersteunt zowel statische testen (wat betekent dat er enkel naar de blokken gekeken wordt, zonder het programma uit te voeren) en dynamische testen (waar het programma uitgevoerd wordt met een bepaalde invoer en de resultaten bekeken worden). Deze combinatie betekent dat Itch een diverse reeks Scratch-programma's kan beoordelen. Scratch lijkt in bepaalde opzichten meer op een spelletje dan op een programmeertaal. Als gevolg hiervan experimenteren kinderen veel en gebruiken ze hun fantasie bij het programmeren. Dit is op zijn beurt een uitdaging bij het testen van Scratch-programma's. Als de opgave bijvoorbeeld "Teken een huis" is, hoe kunnen we een oplossing hiervoor dan beoordelen? Er zijn dus limieten aan de soorten oefeningen die Itch kan beoordelen. De overwegingen die bij deze beslissingen komen kijken worden behandeld in het hoofdstuk.

Als een testraamwerk zoals Itch feedback geeft aan leerlingen, dan is alles soms juist, maar veel vaker zijn er testen die falen. Daarop begint het debugproces: leerlingen moeten achterhalen wat de oorzaak van de gefaalde test is. Dit is notoir moeilijk, want de locatie van de oorzaak in het programma is vaak niet voor de hand liggend. Er zijn gelukkig wel middelen om hiermee te helpen, met als belangrijkste de debuggers. Voor tekstuele programmeertalen zijn er veel debuggers en is er ook veel onderzoek over debuggers. Dodona ondersteunt bijvoorbeeld een debugger voor Python.

Voor Scratch, en blokgebaseerde programmeertalen in het algemeen, is dit evenwel niet het geval. Daarom introduceren we in hoofdstuk 6 een nieuwe debugger voor Scratch: **Blink**. Onze debugger ondersteunt stappen door de code (stapsgewijs de code uitvoeren), de programma-uitvoering pauzeren en verder laten lopen, breekpunten (speciale blokken die de programma-uitvoering pauzeren wanneer ze zelf uitgevoerd worden), en tijdreizen. Een debugger met tijdreizen geeft de mogelijkheid om terug te spoelen in de uitvoering van het programma. Elke stap in de uitvoering wordt opgeslagen, dus kunnen we nadien stap per stap teruggaan. Omdat Scratch voornamelijk gebruikt wordt door een jong publiek, hebben we veel aandacht besteed aan het intuïtief maken van

de debugger. De eerste experimenten in een klas tonen dat leerlingen inderdaad vinden dat de debugger makkelijk om mee te werken is, en dat ze het tijdreizen in het bijzonder nuttig vinden.

We hebben net gezegd dat de debugger het mogelijk maakt om stapsgewijs een programma uit te voeren. We hebben bewust niet beschreven wat we bedoelen met een stap in de context van Scratch. In Scratch bestaat een project namelijk uit verschillende sprites (die getekend worden op het scherm). Elke sprite heeft zijn eigen code, een verzameling stapels (een stapel is een reeks aan elkaar vastgemaakte blokken). Elke stapel van elke sprite wordt gelijktijdig uitgevoerd in Scratch. Een traditionele definitie van een stap (één blok in één stapel per keer) vinden we daarom niet ideaal. In plaats daarvan willen we bij een stap in elke stapel één blok verder gaan.

Dit is evenwel niet mogelijk door de manier waarop Scratch intern werkt (het uitvoeringsmodel). Scratch gebruikt een coöperatief systeem, wat betekent dat het meerdere blokken in dezelfde stapel uitvoert, dan overschakelt naar de volgende stapel en daar meerdere blokken uitvoert, enzovoort. Door snel tussen stapels te wisselen, lijkt het alsof de stapels in parallel uitgevoerd worden. Dit uitvoeringsmodel werd gekozen om een aantal synchronisatieproblemen bij gelijktijdige programma's te vermijden, maar heeft ook nadelen. Zo veroorzaakt het in bepaalde gevallen niet-intuïtief gedrag.

Snel wisselen om parallelisme na te boosten is niet uniek in Scratch: veel systemen werken zo.

In hoofdstuk 7 onderzoeken we of we het uitvoeringsmodel van Scratch zo kunnen wijzigen dat stappen door de code mogelijk wordt zoals hierboven beschreven (één blokje in elke stapel per stap), zonder negatieve effecten op de snelheid en het gedrag van bestaande Scratch-projecten. Aangezien Scratch zoveel gebruikt wordt, kunnen we geen wijzigingen voorstellen die ervoor zorgen dat een groot deel van de bestaande projecten stopt met werken of zich anders gaat gedragen. Om hier met kennis van zaken over te kunnen oordelen, hebben we eerst onderzocht hoe een typisch Scratch-project er in het wild uitziet. Hieruit blijkt dat de meeste Scratch-projecten klein en eenvoudig zijn.

Tot slot sluit hoofdstuk 8 dit proefschrift af door al ons werk, dat we in de verschillende hoofdstukken uit de doeken deden, samen te vatten en te overpeinzen wat de toekomst kan brengen.