# BRAIN STORM

## UniCode

24. New kids on the block

6. Teach yourself programming in ten years

# TABLE OF CONTENTS



Entertaining article, no scientific content. A puzzle or a Cover related article are examples of this category.

Easily readable article on a scientific topic. Should be comprehensible, even without any prior knowledge.

Scientific article that explores a certain topic in depth. Might assume the reader has taken a course that's related to the topic.

# EDITORIAL

*BY: Arnoud van de Meulen, editor*

When you first come to university, usually, everything is confusing. You have no idea where the lecture halls are, there are suddenly deadlines everywhere, and this thing called programming is generally a blur. University life simply consists of a lot of new experiences, each of which can be hard to process.

Being new to student life can be hard. Luckily, you also get to know loads of new people, generally learn a lot of interesting stuff and you can of course enjoy plenty of alcohol and blame it on student tradition! Amidst all the new patterns you have to follow, you find a way to have fun. After five years of studying, this is still true for me: the fact that every term has only ten weeks still manages to escape me every single time, but I learn some really interesting things along the way, even when I have to rush for the latest deadline.

University life seems to have a lot of patterns, be it the four terms of ten weeks, or the regular partying that needs to be done. We could call this some sort of university code, or unicode if you will. Some people would simply say that unicode is only the standard for character encoding, but I think the word has a lot more potential. For example, there is also the code that I produce for my university courses: my very own (usually flawed) UniCode.

This edition of the Brainstorm explores the character of this unicode. There is ample opportunity to learn how to code, with Peter Norvig's guide on how to really learn programming, for example. If you are already proficient in programming, you might enjoy the piece by Topicus, on how to apply this knowledge in multiple programming paradigms.

If you are looking for something a little less serious, you can also read how our chairman views the human mind, or try to get up to speed with the rules a proper university student should always follow, in our section on the UniCode. If you missed out on the introductory camp, or want to reminisce about it already, Rianne's piece on how she experienced 'New Kids on the Block' is just your thing.

I hope you will enjoy reading this edition of the Brainstorm. Don't forget to stay sharp, and remember the UniCode!

*PHOTO: Arnoud van de Meulen*

# BY THE BOARD

BY: *Robin Hermes, chairman*

In the computational theory of the mind, the human brain is viewed as a kind of computer. Expanding on this, I would say human behaviour is like a computer program. Let me take myself as an example.

At the time I was born, I had only a basic set of instructions – kind of like my own assembly language – with a very limited collection of predefined functions. All I could really do was activate a loud warning signal, and turn preprocessed input into raw, useless output. Luckily however, my assembly language also allowed for something else: to define and implement new functions myself. For example, after some practice I learned to properly operate my actuators, and started implementing a walk method.

Then one day, I had sufficiently developed a communication protocol, enabling myself to effectively interact with the world – clearly, quite some debugging was still needed in the beginning.

When I had fine-tuned my communication protocol, it allowed me to include libraries from external sources; new data could rapidly be obtained, allowing my knowledge to grow exponentially.

Now here I am: a self-expanding computer program in a nice wrapper, and part of the board of Cover.

Within the board, we are constantly communicating: keeping each other up-to-date, debating about some pressing matters or discussing our weekends. But besides talking to each other, we are also in touch with our members and the outside world.

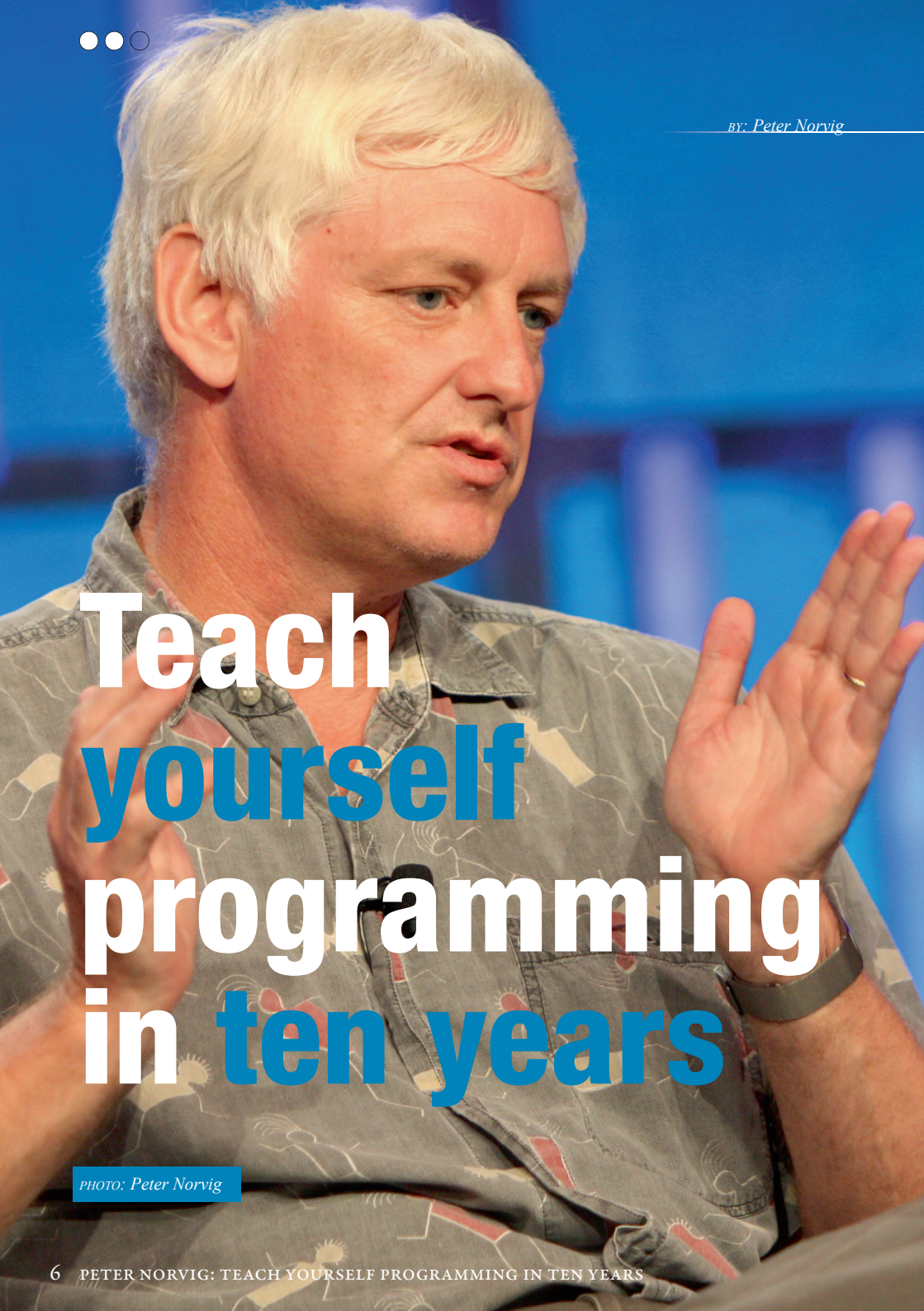
So we are interacting with people all the time to exchange information; to extract useful data from them and to convey our own messages. It is important that during this process, everyone is using the same code page. After all, incorrect decoding of messages can lead to miscommunication and all kinds of unexpected errors.

When we listen to each other carefully and express ourselves clearly, we can make sure that everyone is on the same page and all communication is running smoothly. This way we can break new grounds and get Cover to a higher level. So let's build on those vast knowledge libraries that we already have together, and achieve marvellous things during the coming year!


PHOTO: *Robin Hermes*

# COMIC

BY: Daniel Dennet



BY: Peter Norvig

# Teach yourself programming in ten years

PHOTO: Peter Norvig

**Why is everyone in such a rush?**

Walk into any bookstore, and you'll see *How to Teach Yourself Java in 7 Days* alongside endless variations offering to teach Visual Basic, Windows, the Internet and so on in a few days or hours. I did the following power search at Amazon.com:

> *pubdate: after 1992 and title: days and (title: learn or title: teach yourself)*

and got back 248 hits. The first 78 were computer books (number 79 was *Learn Bengali in 30 days*). I replaced "days" with "hours" and got remarkably similar results: 253 more books, with 77 computer books followed by *Teach Yourself Grammar and Style in 24 Hours* at number 78. Out of the top 200 total, 96% were computer books.

The conclusion is that either people are in a big rush to learn about computers, or that computers are somehow fabulously easier to learn than anything else. There are no books on how to learn Beethoven, or Quantum Physics, or even Dog Grooming in a few days. Felleisen et al. give a nod to this trend in their book *How to Design Programs*, when they say "Bad programming is easy. Idiots can learn it in 21 days, even if they are dummies".

Let's analyze what a title like *Learn C++ in Three Days* could mean:

*Learn:*

In 3 days you won't have time to write several significant programs, and learn from your successes and failures with them. You won't have time to work with an experienced programmer and understand what it is like to live in a C++ environment. In short, you won't have time to learn much. So the book can only be talking about a superficial familiarity, not a deep understanding. As Alexander Pope said, a little learning is a dangerous thing.

*C++:*

In 3 days you might be able to learn some of the syntax of C++ (if you already know another language), but you couldn't learn much about how to use the language. In short, if you were, say, a Basic programmer, you could learn to write programs in the style of Basic using C++ syntax, but you couldn't learn what C++ is actually good (and bad) for. So what's the point? Alan Perlis once said: "A language that doesn't affect the way you think about programming, is not worth knowing". One possible point is that you have to learn a tiny bit of C++ (or more likely, something like JavaScript or Flash's Flex) because you need to interface with an existing tool to accomplish a specific task. But then you're not learning how to program; you're learning to accomplish that task.

> Anyone can cook, but only the fearless can be great

*in Three Days:*

Unfortunately, this is not enough, as the next section shows.

**Teach Yourself Programming in Ten Years**

Researchers (Bloom (1985), Bryan & Harter (1899), Hayes (1989), Simmon & Chase (1973)) have shown it takes about ten years to develop expertise in any of a wide variety of areas, including chess playing, music

composition, telegraph operation, painting, piano playing, swimming, tennis, and research in neuropsychology and topology. The key is deliberative practice: not just doing it again and again, but challenging yourself with a task that is just beyond your current ability, trying it, analyzing your performance while and after doing it, and correcting any mistakes. Then repeat. And repeat again. There appear to be no real shortcuts: even Mozart, who was a musical prodigy at age 4, took 13 more years before he began to produce world-class music. In another genre, the Beatles seemed to burst onto the scene with a string of #1 hits and an appearance on the Ed Sullivan show in 1964. But they had been playing small clubs in Liverpool and Hamburg since 1957, and while they had mass appeal early on, their first great critical success, Sgt. Peppers, was released in 1967. Malcolm Gladwell reports that a study of students at the Berlin Academy of Music compared the top, middle, and bottom third of the class and asked them how much they had practiced.

> The key is challenging yourself with a task that is just beyond your current ability

Everyone, from all three groups, started playing at roughly the same time - around the age of five. In those first few years, everyone practised roughly the same amount - about two or three hours a week. But around the age of eight real differences started to emerge. The students who would end up as the best in their class began to practise more than everyone else: six hours a week by age nine, eight by age 12, 16 a week by age 14, and up and up, until by the age of 20 they were practising well over 30 hours a week. By the age of 20, the elite performers had all totalled 10,000 hours of practice over the course of their lives. The merely good students had totalled, by contrast, 8,000 hours, and the future music teachers just over 4,000 hours.

So it may be that 10,000 hours, not 10 years, is the magic number. (Henri Cartier-Bresson (1908-2004) said "Your first 10,000 photographs are your worst," but he shot more than one an hour.) Samuel Johnson (1709-1784) thought it took even longer: "Excellence in any department can be attained only by the labor of a lifetime; it is not to be purchased at a lesser price." And Chaucer (1340-1400) complained "the lyf so short, the craft so long to lerne." Hippocrates (c. 400BC) is known for the excerpt "ars longa, vita brevis", which is part of the longer quotation "Ars longa, vita brevis, occasio praeceps, experimentum periculosum, iudicium difficile", which in English renders as "Life is short, [the] craft long, opportunity fleeting, experiment treacherous, judgment difficult." Although in Latin, "ars" can mean either "art" or "craft", in the original Greek the word "techne" can only mean "skill", not "art".

**So You Want to be a Programmer**

*Get interested in programming, and do some because it is fun*

Make sure that it keeps being enough fun so that you will be willing to put in your ten years/10,000 hours.

*Program*

The best kind of learning is learning by doing. To put it more technically, "the maximal level of performance for individuals in a

given domain is not attained automatically as a function of extended experience, but the level of performance can be increased even by highly experienced individuals as a result of deliberate efforts to improve." (p. 366) and "the most effective learning requires a well-defined task with an appropriate difficulty level for the particular individual, informative feedback, and opportunities for repetition and corrections of errors." (p. 20-21) The book *Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life* is an interesting reference for this viewpoint.

*Talk with other programmers*
Read other programs. This is more important than any book or training course.

*If you want, put in four years at a college*
This will give you access to some jobs that require credentials, and it will give you a deeper understanding of the field, but if you don't enjoy school, you can (with some dedication) get similar experience on your own or on the job. In any case, book learning alone won't be enough. "Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter" says Eric Raymond, author of *The New Hacker's Dictionary.* One of the best programmers I ever hired had only a High School degree; he's produced a lot of great software, has his own newsgroup and made enough in stock options to buy his own nightclub.

*Work on projects with other programmers*
Be the best programmer on some projects; be the worst on some others. When you're the best, you get to test your abilities to lead a project, and to inspire others with your vision. When you're the worst, you learn what the masters do, and you learn what they don't like to do (because they make you do it for them).

*Work on projects after other programmers*
Understand a program written by someone else. See what it takes to understand and fix it when the original programmers are not around. Think about how to design your programs to make it easier for those who will maintain them after you.

*Learn at least a half dozen programming languages*
Include one language that supports class abstractions (like Java or C++), one that supports functional abstraction (like Lisp or ML), one that supports syntactic abstraction (like Lisp), one that supports declarative specifications (like Prolog or C++ templates), one that supports coroutines (like Icon or Scheme), and one that supports parallelism (like Sisal).

*Remember that there is a "computer" in "computer science"*
Know how long it takes your computer to execute an instruction, fetch a word from memory (with and without a cache miss), read consecutive words from disk, and seek to a new location on disk.

*Get involved in a language standardization effort*
It could be the ANSI C++ committee, or it could be deciding if your local coding style will have 2 or 4 space indentation levels. Either way, you learn about what other people like in a language, how deeply they feel so, and perhaps even a little about why they feel so. Have the good sense to get off the language standardization effort as quickly as possible.

With all that in mind, it's questionable how far you can get just by book learning. Before my first child was born, I read all the *How To* books, and still felt like a clueless novice. 30 Months later, when my second child was due, did I go back to the books for a refresher? No. Instead, I relied on my personal experience, which turned out to be far more useful and reassuring to me than the thousands of pages written by experts. Fred Brooks, in his essay *No Silver Bullet* identified a three-part plan for finding great software designers:

1. Systematically identify top designers as early as possible.
2. Assign a career mentor to be responsible for the development of the prospect and carefully keep a career file.
3. Provide opportunities for growing designers to interact and stimulate each other.

This assumes that some people already have the qualities necessary for being a great designer; the job is to properly coax them along. Alan Perlis put it more succinctly: "Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers". Perlis is saying that the greats have some internal quality that transcends their training. But where does the quality come from? Is it innate? Or do they develop it through diligence? As Auguste Gusteau (the fictional chef in Ratatouille) puts it, "anyone can cook, but only the fearless can be great". I think of it more as willingness to devote a large portion of one's life to deliberative practice. But maybe fearless is a way to summarize that. Or, as Gusteau's critic, Anton Ego, says: "Not everyone can become a great artist, but a great artist can come from anywhere."

So go ahead and buy that Java/Ruby/Javascript/PHP book; you'll probably get some use out of it. But you won't change your life, or your real overall expertise as a programmer in 24 hours, days, or even weeks. How about working hard to continually improve over 24 months? Well, now you're starting to get somewhere...

**References:**


Bloom, Benjamin (ed.) Developing Talent in Young People, Ballantine, 1985.

Brooks, Fred, No Silver Bullets, IEEE Computer, vol. 20, no. 4, 1987, p. 10-19.

Bryan, W.L. & Harter, N. "Studies on the telegraphic language: The acquisition of a hierarchy of habits. Psychology Review, 1899, 8, 345-375

Hayes, John R., Complete Problem Solver Lawrence Erlbaum, 1989.

Chase, William G. & Simon, Herbert A. "Perception in Chess" Cognitive Psychology, 1973, 4, 55-81.

Lave, Jean, Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life, Cambridge University Press, 1988.



Peter Norvig is a Director of Research at Google Inc; previously he directed Google's core search algorithms group. He is co-author of Artificial Intelligence: A Modern Approach, the leading textbook in the field, and co-teacher of an Artifical Intelligence class that signed up 160,000 students, helping to kick off the current round of massive open online classes. He is a fellow of the AAAI, ACM, California Academy of Science and American Academy of Arts & Sciences.

Alumni association Computing Science

# INVARIANT

University of Groningen

If you are currently studying Computing Science at the University of Groningen, you are in luck! This is probably the best time of your life. You work in an interesting environment where new things are invented. You have great parties. You make friends for life. You do what you love and you are learning to do it better.

But unfortunately, everything has to end one day. The day you graduate you get to celebrate that you have done something great, but it will also mean you have to say goodbye to your life as a student. Does this mean you will no longer see your fellow students? Does this mean you will never visit the university again? It doesn't have to be!

Invariant, the alumni association of the study Computing Science at the University of Groningen, is here for you. Our goal is to bring young and old alumni together and help them to keep in touch with the university. We try to do that by organizing activities which have both a professional and a social component.

Have you finished your bachelor and are now doing your master? Then you already are an alumnus, and that means you can become a member. This could be the perfect way to get in touch with people working in the industry, have a look inside interesting companies and find a position at a company for an internship or doing your master's thesis.

So, do you have a degree in Computing Science at the University of Groningen and do you also want to become Invariant? Become a member of our alumni association! You can sign up at our website:

**WWW.INVARIANT.NL**

# IN WELKE TAAL DENK JIJ

BY: Thomas Markus, Topicus

Pidgin: je kent het misschien als multi-protocol messenger app, maar weet je eigenlijk wat het echt betekent? Ten tijde van de kolonisatie is het relatief vaak voorgekomen dat er zogenaamde pidgins zijn ontstaan. Een pidgin is een nieuwe, tweede taal die het resultaat is van de combinatie van verschillende moedertalen. Deze kan ontstaan in een omgeving waarin er geen dominante gemeenschappelijke taal is. Grammaticaregels van de onderliggende talen worden toegepast op woorden uit andere talen waardoor interessante combinaties ontstaan. Omdat geen van de sprekers de pidgin initieel als moedertaal beheerst is de grammatica eenvoudig, zijn de klanken duidelijk herkenbaar en zijn vervoegingen schaars. Voorbeelden van typische zinnen in een pidgin-taal zijn "long time no see" of "mij graag meewillen als mogen". Enkele historische voorbeelden van (uitgestorven) pidgintalen zijn; Russenorsk en Sabir.

Een pidgin kan zich verder ontwikkelen doordat jonge kinderen opgroeien in een gemeenschap waar de pidgin veel gebruikt wordt en daardoor de pidgin als moedertaal verwerven. Als dit gebeurt, spreken we niet langer van een pidgin, maar van een creoolse taal. Door de automatische taalverwerving van kinderen gedurende de kritieke periode (voor de pubertijd) ontstaan automatisch nieuwe syntactische en semantische verfijningen. Veel creoolse talen zijn helaas geen lang leven beschoren en gaan, vaak om pragmatische redenen, (deels) op in een andere dominante taal. Een voorbeeld hiervan is het Jamaicaans.

Je vraagt je misschien nu af waarom we het hebben over de ontwikkeling van natuurlijke talen en wat dit te maken heeft met Computing Science. En terecht! Je kunt je voorstellen dat je als ontwikkelaar een nieuwe taal leert als 'moedertaal' (in mijn geval QBasic) tijdens je 'kritieke periode'.

> Simpelweg blijven hacken 'totdat het werkt' is niet langer een werkende strategie.

Hierna kom je in aanraking met Haskell of Miranda. Tijdens dit leerproces schrijf je wellicht een programma in een imperatieve taal met functionele concepten of andersom. In dit geval zijn er interessante parallellen te trekken met de eerder genoemde pidgin-talen. De grammatica is vereenvoudigd ten opzichte van de moedertaal en slechts een deel van de uitdrukkingskracht van de oorspronkelijke taal blijft over in de combinatie. Het beschreven proces van taalevolutie vertaalt zich redelijk door naar programmeertalen waarbij er de laatste jaren een convergentie zichtbaar is van imperatieve naar functionele programmeerstijlen. Is die cursus functioneel programmeren toch nog nuttig!

Wat is functioneel programmeren eigenlijk

en waarom wil je het wel of juist niet?
Een van de voordelen is dat de stijl van programmeren veel dichter tegen de prachtige eenvoud van de wiskunde en logica aanligt. Hierdoor is de code zeer compact en expressief. Een van de nadelen is dat de de code dicht tegen wiskunde en logica aanligt en daardoor zeer compact en expressief is. Ook een wiskundige formule kan helemaal correct, en toch compleet onleesbaar zijn. De ontwikkelaar moet dus wel in staat zijn om relatief complexe abstracties te interpreteren en het snel uitvoeren daarvan vereist simpelweg veel oefening in vergelijking met imperatieve talen. De toepassing van functioneel programmeren is daarmee dus geen garantie voor succes en hangt nauw samen met de betrokken ontwikkelaars. Enige achtergrondkennis en ervaring met deze abstractere vorm van programmeren is zeker welkom en heeft, academisch gezien, een veel betere 'return on investment' dan het imperatief programmeren waarbij je je wiskundekennis effectief in het afvoerputje werpt. Helaas schrikt het hogere abstractie- en opleidingsniveau van de functionele pracht en praal een groot deel van de potentiële ontwikkelaars af. Simpelweg blijven hacken in bestaande code 'totdat het werkt' is niet langer een werkende strategie. In de praktijk is een organisatie voorzichtig met de toepassing van functioneel programmeren met als voornaamste argument dat code voor zoveel mogelijk ontwikkelaars toegankelijk moet blijven.

Binnen grotere softwareprojecten komt regelmatig een diversiteit aan talen terug. Wat dat betreft is een vacature die specifiek vraagt om een "junior Java-programmeur" of een "C#-architect" best wel gek. Durf jij anno 2014 een Java-programmeur aan te nemen die geen JavaScript, Python of Ruby kent? Eigenlijk behoort een soortgelijke vacature te vragen naar je capaciteiten wat betreft het kunnen vinden van patronen en maken van effectieve abstracties. De specifieke taal die je, toevallig, gebruikt is daarbij maar een bijzaak, want de tijd zit in het begrijpen en bedenken van een oplossing en veel minder in de uitwerking. De keus tussen een uitstekende C#-ontwikkelaar en een junior Java-ontwikkelaar voor een Java-project is redelijk voor de hand liggend. Om dit te illustreren met een voorbeeld; ik zit hier nu op mijn werkplek en om mij heen zie ik vooral software-ontwikkelaars, soms met de wenkbrauwen gefronst, hard nadenken om daarna in één vaart de oplossing te implementeren. De discussies dagelijks op de werkvloer gaan daarbij voornamelijk over schoonheid, elegantie en of het future-proof zijn van een oplossing. Of deze oplossing imperatief, declaratief, logisch of functioneel moet worden opgepikt, varieert daarbij sterk.

Ik suggereer dus dat ieder softwareproject bestaat uit een multiculturele mengelmoes van programmeertalen en stijlen, maar dat botst natuurlijk wel met wat je ziet langskomen aan vacatures. Binnen de meeste grotere codebases staan de klassieke imperatieve talen (Java, C#, C++) nog steeds ruimschoots bovenaan. Maar praktisch iedereen leert tijdens zijn/haar eerste jaar op de universiteit/hogeschool dat functioneel programmeren fantastisch en veel 'beter' is dan imperatief ontwikkelen. Prima, maar toch zie je nergens die berg met Haskell-vacatures klaarliggen. Is het niet een puur academische aangelegenheid? Goede vraag! Wat dit betreft is het heel interessant om te zien dat de grens tussen functioneel en imperatief begint te vervagen. Er zijn bijvoorbeeld Java-libraries zoals Guava die functionele constructen zoals immutable

datastructures, map, zip, lazy evaluation, etc. eenvoudig toegankelijk maken binnen Java, en er zijn soortgelijke libraries voor het .NET-platform. Het is hierbij wel belangrijk om te onderkennen dat dit geen volledige samensmelting is van de twee programmeerparadigma's, maar meer wordt gedaan onder het mom van: "beter goed gejat dan slecht verzonnen". Je zou kunnen zeggen dat hierdoor een pidgin ontstaat.

Het gebruik van functionele constructen binnen imperatieve talen en frameworks voelt soms alsof je een kruiskopschroef vastdraait met een platte schroevendraaier. Het werkt, maar het kan beter. Je kunt ook een stapje verder gaan en een taal als F# of Scala gebruiken waarbij functions wel 'first class citizens' zijn. Dat levert bijvoorbeeld bij Scala prachtige constructies op als pattern matching met behulp van case classes. Voordelen van F# en Scala zijn dat ze eenvoudig kunnen interacteren met het immense software-ecosysteem voor de JVM en .NET en tevens een goed type-systeem hebben. Vooral deze twee eigenschappen maakt ze daadwerkelijk acceptabel voor projecten met een grotere codebase. Het behoort niet zo te zijn dat een grote codebase je remt in het gebruik van nieuwe programmeertechnieken, maar het is wel zo prettig om een bestaande betrouwbare library te kunnen gebruiken voor complexe gegevensuitwisseling van bijvoorbeeld miljoenen leerlinggegevens.

Je kunt het gebruik van functioneel programmeren dus stapsgewijs aanpakken en bibliotheken inzetten met (slappe aftreksels van) functionele talen. Een alternatief is om het gebruik van een functionele taal, zoals Scala, binnen een bestaand ecosysteem van libraries en software in te zetten. Echter, het is waarschijnlijk prettiger om al direct met versie "0.0.1 beta" functioneel te starten. In theorie is dit heel interessant, omdat het onder andere parallelle operaties veel eenvoudiger maakt vanwege het gebrek aan 'shared state'. Hierdoor kun je nagenoeg zonder locks en foutgevoelige synchronisatielogica werken. Een ware verademing ten opzichte van de oude werkwijze waarbij handmatig threads moeten worden beheerd. Een nieuw project opzetten waarbij parallelle computatie en schaalbaarheid belangrijk zijn zonder zwaar te leunen op functionele concepten en technieken is simpelweg niet handig. Deze aanpak vereist natuurlijk wel dat alle ontwikkelaars bekend zijn met het 'abstractere' functioneel programmeren en dat het past binnen het bestaande software- en library-ecosysteem. Voor veel bedrijven en ontwikkelaars gaat deze laatste vlieger helaas niet op ook al zijn veel organisaties

> Is het type van de functionele implementatie wel wenselijk?

(ook kleinere) bezig met servicification: het ontsluiten van (kleine) programma's als een (REST) webservice. Vooral dit laatste geeft je veel meer vrijheid wat betreft technieken door de grotere mate van ontkoppeling en maakt het mogelijk kleinschalig te experimenteren met nieuwe technieken.

Vaak is het praktisch niet haalbaar om helemaal vanaf nul te beginnen, omdat je dan jaren aan kennis, ervaring en bugfixes weggooit. We moeten dit dus stapsgewijs aanpakken en incrementeel de bestaande codebase herschrijven waar dat zinnig is. Dit voorstel klinkt heel redelijk, maar er blijft een

groot verschil zitten tussen de opzet van een functioneel programma en een imperatief/OO-opgezet programma. Voor een klein project kun je nog wel op een bepaalde stijl standaardiseren, maar in de praktijk is dat met een groter project met 100.000 classes, entiteiten en interfaces andere koek. Het willekeurig refactoren van 'oude' imperatieve code naar een prachtige functioneel opgezette implementatie kan behoorlijk vervelend lezen als dit per methode verschilt. Wil jij een boek lezen waarbij per paragraaf willekeurig in het Duits, Engels of Nederlands is geschreven? Je komt er wel doorheen, maar het leest minder prettig dan een boek volledig in één taal. Ook al is de taal wat 'plat'.

Deze problemen betekenen natuurlijk niet dat eens geschreven code heilig is en nooit meer mag veranderen, want actief refactoren en verbeteren van een codebase is cruciaal. Het kan simpelweg altijd beter of mooier. Een functionele taal levert daarentegen niet per direct een snelle applicatie op. In principe zit het voordeel hem voornamelijk in de ontwikkeltijd van de ontwikkelaar, maar die voordelen zijn zacht en ongrijpbaar. Het is dus moeilijk om het argument maken dat je een paar miljoen moet investeren om een bestaand project te herschrijven van een imperatieve taal naar een functionele taal. Zeker omdat het helemaal niet vaststaat dat een functioneel programma per definitie een betere applicatie oplevert. Desalniettemin, dat betekent nog niet dat de keuze geheel arbitrair is: er zijn zeker voorbeelden te geven waarbij de functionele variant van een algoritme minder bug-gevoelig is. Zie bijvoorbeeld het volgende voorbeeld:

```
public class Voorbeeld {
  public static void main(String[] args) {
    // vind het kwadraat van het eerste even getal dat groter is dan 3
    // voor een lijst met nul of meer elementen.
    List<Integer> numbers = Arrays.asList(1,2,3,5,4,9);

    // imperatieve implementatie:
    int result = 0;
    for(int i=0; i <= numbers.size(); i++) {
      int e = numbers.get(i);
      if (e > 3 && e % 2 == 0) {
        result = e*e;
        break;
      }
    }

    // functionele implementatie
    numbers.parallelStream()
      .filter(e -> e > 3)
      .filter(e -> e % 2 == 0)
      .map(e -> e * e)
      .findFirst();
  }
}
```

Het code-voorbeeld is op basis van Java 8. Welk voordeel heeft de functionele implementatie op de imperatieve? Tip: wat is het resultaat van de methode findFirst()? Welke implementatie schaalt beter (pas op: strikvraag)? Wat is het resultaat bij een lege lijst voor de twee implementaties? Welke implementatie is correct? Is het type van de functionele implementatie wel wenselijk? In hoeverre zou je de functionele implementatie kwalificeren als 'pidgin' in vergelijking met een taal als Haskell?


*XKCD: Haskell*

Functioneel programmeren kan dus nuttig zijn; dus hoe gaan we dit nu aanpakken? Hoe komen we uit in het 'paradijs'? Wat doe je als je applicatie bestaat uit 15071 classes en interfaces? Herschrijf je alleen een reeks classes met veel interactie met elkaar zodat ze qua stijl uniform zijn? Hoe bepaal je welke deelgebieden binnen een applicatie kunnen profiteren van een functionele opzet en voor welke andere delen is de imperatieve variant 'natuurlijker'? Het is een spannende tijd waarin we als ontwikkelaar leven met een plethora aan ontwikkelparadigma's. Ook omdat de grotere ontkoppeling van componenten via REST-webservices je meer vrijheid geeft qua tools en technieken.

Ik kijk met veel interesse naar de toenemende invloed van functioneel programmeren op de 'oude' imperatieve garde. Het gebruik van functionele concepten voor bepaalde domeinen kan de foutgevoeligheid van code verlagen en de leesbaarheid ten opzichte van de imperatieve uitwerking verbeteren. De impact die dit heeft op de manier waarop we software ontwikkelen en nadenken over problemen is groot. Ik hoop over een aantal jaren te kunnen zeggen dat we qua frameworks, talen en denkwijzen de overstap gemaakt hebben van een pidgin naar een creoolse taal.

# UNICODE

## UniCode #0

When a uni starts counting at 1, they have to get beer for all unis whom continue counting

## UniCode #8

A real uni always compiles before committing

## UniCode #Φ

Unis always look each other in they eye while toasting

## UniCode #404

Unis only drink with their non-dominant hand

## UniCode #e

Don't walk through closed doors after 20.00 at the BB, it will trip the alarm

When you want to be next at a game, call dibs. Dibs is sacred

## UniCode #π

Unis always make their reports in LaTeX

## UniCode #-1/12

When you are getting N beer at the borrel, order N + 1

BY: Marco Aiello

# Shedding (LED) light on the 2014 Physics Nobel Prize

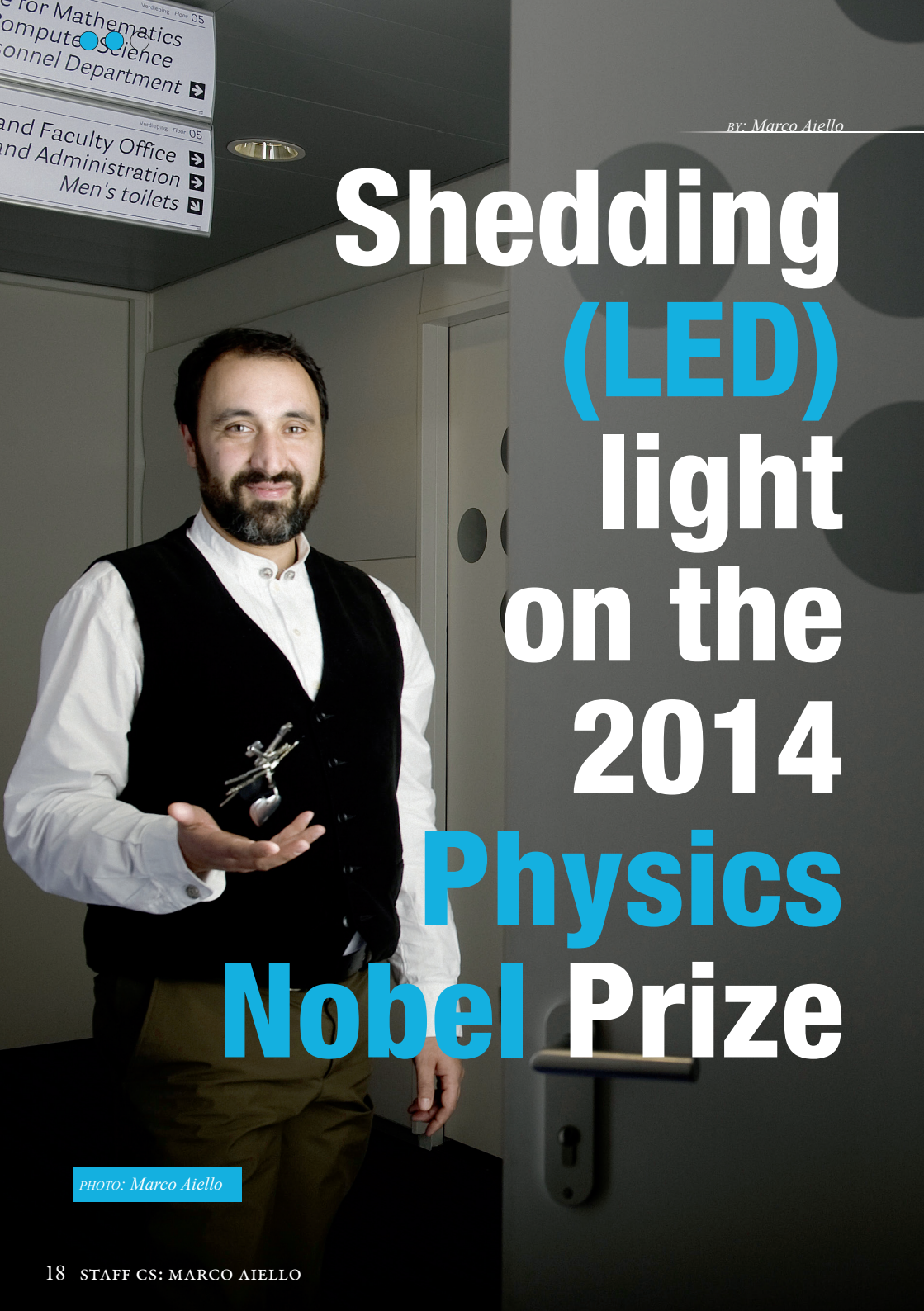



PHOTO: Marco Aiello

In his final will, dated November 27, 1895, inventor and millionaire Alfred Nobel left almost all of his fortune to establish a prize to "those who, during the preceding year, shall have conferred the greatest benefit on mankind". The prize was designated for physics, chemistry, physiology or medicine, literature, and peace. In 1968, thanks to a donation of the Sveriges Riksbank, the prize for Economic sciences was added to the list. As Cover members, we know that there is no Nobel Prize in our fields of study and research: no Nobel for mathematics or computer science. Word has it that Nobel had harsh feelings towards a mathematician having an affair with his wife, though this is unlikely to be true, since he was never married. As for computer science, well there simply was no such field in sight at his time. Mathematicians can console themselves with the Fields medals and the computer scientists with the ACM Turing award, which, incidentally, this year went to one of the most prominent figures in my own field, Distributed Systems: Leslie Lamport.

Being in a faculty of exact sciences, my attention is usually driven by the Nobel Prize in physics. The one that has been awarded to Einstein, Fermi, Bohr and, of course, our very own Groninger Zernike. This prize in popular culture represents more than just an award for a discovery in physics, it extends to represent the highest human intelligence. The name of Einstein is regularly used as a synecdoche to represent the class of people who are extraordinary geniuses. And it is not by accident that these most notable Nobel Prize winners were active in particle physics: the most common field to attract prizes (see the infographic from physicsworld.com on the next page). In fact, particle physics is a field full of unintuitive phenomena which need extraordinary imagination, great mathematical skills, and abstract thinking. However, this year things went in a different direction.

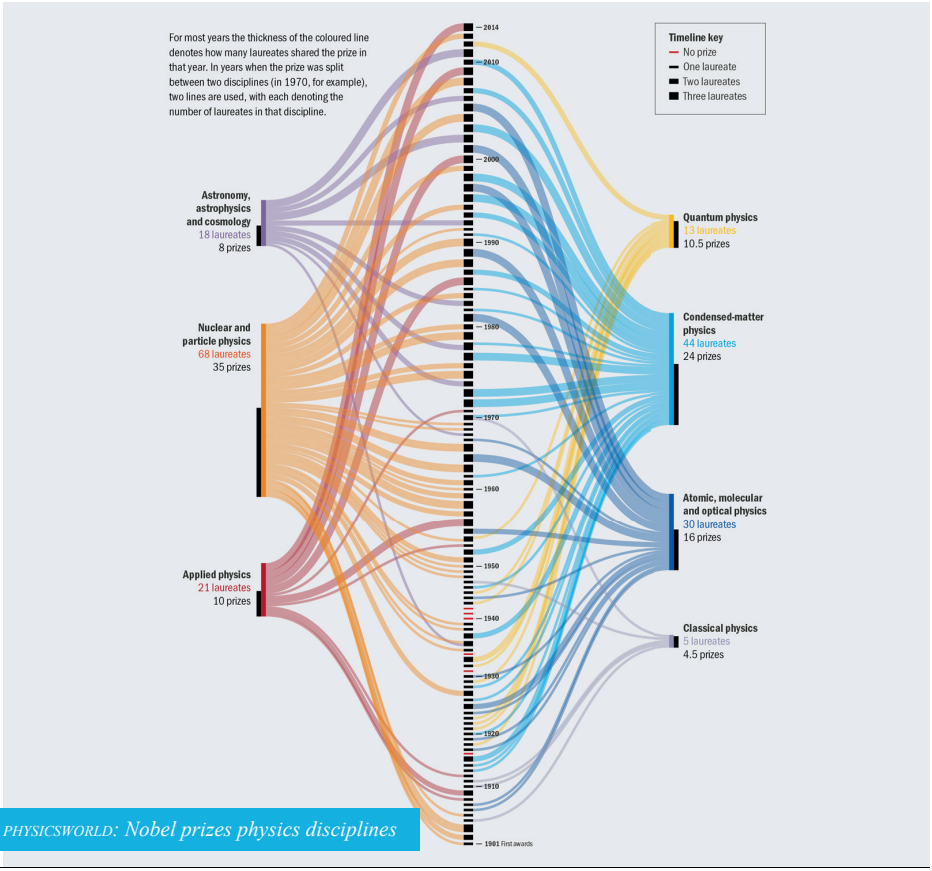
The 2014 Nobel prize award for Physics was awarded to the inventors of blue LED lights: Isamu Akasaki, Hiroshi Amano, and Shuji Nakamura. The motivation citing "for the invention of efficient blue light-emitting diodes, which has enabled bright and energy-saving white light sources." So where is the hard physics? The theorized particles that will be only discovered experimentally many decades later? In fact, there are none. The motivation is simple: "An invention of greatest benefit to mankind; using blue LEDs, white light can be created in a new way. With the advent of LED lamps we now have more long-lasting and more efficient alternatives to older light sources.

"{...} Incandescent light bulbs lit the 20th century; the 21st century will be lit by LED lamps" as the Royal Swedish Academy explained in its press release commenting the award. Commercial LED lights allow us to illuminate while saving nearly 90% of electricity. They allow us to bring light where no or limited infrastructure is present. In fact, the 2014 Nobel Prize winners have solved an engineering problem by which I mean, a problem in the discipline of designing and creating complex structures or new products or systems by using scientific methods that impact society. In this specific case, the problem being that of taking red light emitting diodes, invented in the 60s, and making them emit blue light with a color

So where is the hard physics?

temperature comfortable to the human eye. Isamu Akasaki, Hiroshi Amano, and Shuji Nakamura are in fact three engineers who have been working on the LED light problem since the 80s. Akasaki, now a retired 85 year old emeritus, holds a PhD in Electronic Engineering and has worked most of his life in the Department of Electronics at Nagoya University. Hiroshi Amano joined in 1982 as a PhD student the group of Akasaki, obtained his PhD in electronic engineering and is currently a professor in the Department of Electrical Engineering and Computer Science. Shuji Nakamur also obtained a PhD in electronic engineering and, after a career in the Japanese industry, became a professor at the College of Engineering, University of California, Santa Barbara.

There is more though than engineering to the 2014 Nobel prize award for physics; there is also a tale of research valorization. The story of Nakamura is in this respect exemplary. After his degrees, Nakamura went to work for the Japanese chemical company Nichia. Being fascinated by the results of Akasaki's group, he started working on a solution to the LED problem based on gallium nitride (GaN). At the time, this was considered to be an unfeasible path when considering mass




PHYSICSWORLD: *Nobel prizes physics disciplines*

production. He reported that "in 1990 and 1991, the president [of Nichia] asked me to stop the GaN research immediately. At this time Nichia had no desire to investigate blue LEDs, so they asked me to work on GaAs high-electron mobility transistors (HEMT) instead. I ignored them."

As we know today, he was right to be so perseverant. Something that makes me wonder: who would have the courage and the possibility to take such a decision today? In the current research atmosphere, researchers have to promise short and medium term results that they have to prove feasible to achieve, in order to get funding. My impression is that today high-risk basic and applied research would be ranked poorly by any financing body on the grounds of lack of "utilization".

But let's get back to the story of Nakamura. He, as all employees of the company, would receive an incentive of about 100$ for every filed patent, and he filed about 500 during his employment at Nichia. Though, when he solved the blue LED light problem, he felt something was wrong. First of all, he thought he deserved more, but more importantly he wanted the patent to be licensed to anybody. In 2001 he said that "if Nichia had not monopolized the patents, the blue LED market would have grown 10 times larger". After continuous frustration with his employer, he finally decided to leave Japan and begin an academic career in the United States. Soon after, a legal dispute began. "I decided to take action against Nichia because in Japan there is a special patent law that exists only there and in Germany," he explained. "Even if a researcher invents a patent at a company, using company money and company people, the patent belongs to the inventor, not the company."

In 2004, Nakamura won a landmark ruling in the Tokyo District Court for 20 billion yen (about 150 million euros), recognizing his role in the patent and the potential value of it (Nichia revenues increased by 1000% thanks to their new LED light business). Though, after the appeal of Nichia, he decided to settle for a fraction of that amount. Today Nakamura enjoys his life as a (rich) professor, entrepreneur and now Nobel laureate.

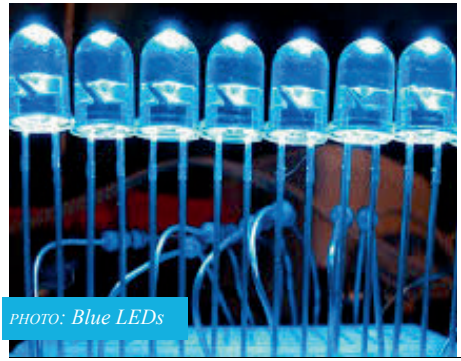


PHOTO: *Blue LEDs*

The 2014 Nobel Prize is something I cheer about. It celebrates engineering, it embraces a vision of valorization of research, it carries a story of rewarding inventors, it gives energy conservation a deep support, and finally, it is about a product that I have enjoyed evermore since installing it in my home and using it on my bike. And if that was not enough, "engineering, valorization and energy" are also three keywords that are central to the current faculty strategy for the future.

2014 has been a great year: a Nobel Prize was awarded to engineers, the Turing award to distributed systems, and an exciting faculty strategy was left to follow.

# PUZZLE

BY: *Isabela Constantin*

Through the Looking-Glass, and What Alice Found There...

...is a novel written by Lewis Carroll as a sequel to "Alice's Adventures in Wonderland". Indeed, her adventure is not over, only this time it takes place in the land beyond the looking-glass. Once she enters through the mirror of her own room, Alice finds herself in a land which turns out to be a huge chessboard. For now, she is just a pawn, located in the second row... However, she can become a queen if you help her to move forward up to the eighth row by writing down the clues in the ☐ at the end and break the code!

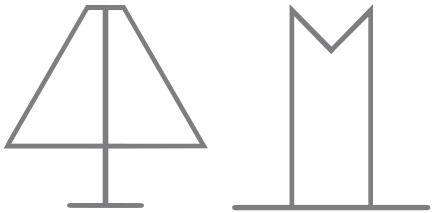
Puzzled as she is, Alice explores the surroundings and encounters the Red Queen, who is willing to give her advice on how to get around, but it's her tea time. No, it's not 5 o'clock in the afternoon, it's 12:08. But well... "it's always time for a tea", as the Mad Hatter used to say. Give the angle between the minute and hour hands of the clock at 12:08 and help Alice have a tea with the Red Queen! Note that down, for it will come handy later, along with the other clues that you'll find in the way. 0 1 In the first move, Alice can advance two fields.

In the fourth row, Alice meets with Humpty Dumpty, which she finds rather difficult to talk to. "When I use a word, it means just what I choose it to mean — neither more nor less.", Humpty Dumpty says. So when the little girl asks him for a clue, the egg replies with a riddle.

$$\begin{array}{r} \text{ALICE} \\ \text{SMILE} + \\ \hline \text{QUEENS} \end{array}$$

Translate "CN SC" in order to put Alice on the right track ! 2 3 4 5

In the fifth field, Alice meets the knight, who offers to give her a ride if she can help him with this strange looking symbols... 6 7



One field further, Alice bumps into Tweedledum, who's very angry. He had a fight with his twin brother Tweedledee, who decided to leave him and travel by himself through other mirror lands. Happily, he comes back for he realizes he misses his own brother. Strangely enough, Tweedledee now looks like a younger version of Tweedledum. Is it karma? Or is something else? Help the twins unveil the mystery and they will point Alice the way to the next field. 8 9 You don't have to be an Einstein to solve it, but that would have helped.

Almost there! It's time to prepare for the grand coronation. Gather all the clues:

| 0 1 | 2 3 | 4 5 | 6 7 | 8 9 |
|---|---|---|---|---|

Pair them and break the code so that Alice can make it to the eighth row and become a queen!

| | | | | |
|---|---|---|---|---|

Do you think you have broken the code? Send it to brainstorm@svcover.nl before 19 January 2015 to win a prize!

BY: Rianne Brandsma

# Intro camp: New kids on the block


PHOTO: Morning gymnastics

September 5th – September 7th 2014, a weekend we will never forget.
Today was the first day of university and we all felt just like we did on the first day of secondary school. Well, not really, because we had already had lectures for like five days, but let's not let that spoil the fun. It was a beautiful sunny day, so waiting outside the Bernoulliborg was not a punishment at all. After waiting for about half an hour, the bus drivers were ready, and it was time for our amazing introduction weekend to begin!

When we arrived at Stadskanaal, we sat in a big circle and got some information about what was going to happen in the next few hours and what rules we had to live up to. After that, we had to drop our stuff and find a bed to sleep in. It was at that same moment that I painfully realized I knew exactly zero of the girls that were present here. So I just threw my belongings on the first empty bed I saw and luckily I ended up with three amazing girls! On the website they told us to bring a sleeping bag, pillow etcetera and they turned out to be completely useless, but who needs space in their bags for clothes anyway. Next we went outside to play some ice-breaker games. We were divided into groups and played all sorts of games like 'krantenmeppen' (hit someone with a newspaper if they take too long to say a name) and 'sta-bal' (yell someone's name, who then has to try to catch the ball and throw it through somebody's legs). Without a doubt the most stunning game I've ever played was the game where you had to throw a ball of wool to create a web. In our case however, the ball was broken, so we made a fool out of ourselves while throwing an imaginary ball.

After these games, it was about 8 o'clock and everyone was starving. We all got a card to play the game 'Gotcha' and were told that we could get our own exclusive 'New Kids on the Block' t-shirt. And then, finally, it was time for dinner! We were told that the soup and the baguettes were ready, so we all ran to our cutlery as quickly as possible and seated ourselves at the tables. Dinner tasted really good, even though I lost my Gotcha card after about ten minutes. Some of us also made our t-shirts a bit more fabulous by turning them into tank tops. With all the energy we gained from dinner, we were very motivated to win the evening games with our group '10'. I think we managed to complete one of the games and at that game we failed terribly. Another group adopted me, so I played a few other games, like making our own song. When the games were done, it was time for the real fun to begin: drinking games and beer. Unfortunately, I wasn't 18 at the time, so I just took my ''you shall not drink''-bracelet and sat in a corner crying for the entire night.

Who needs space in their bags for clothes anyway?

The next morning, I was really grateful for that. We were woken by some really loud noises at 9 o'clock. I can't tell what it was exactly, because opening my eyes at that time was completely impossible. Everyone was, of course, super excited when we found out that we had to participate in morning gymnastics. It was really funny to see who drank a little too much the night before. After that, it was time for breakfast and for some an opportunity to get a little extra sleep. When breakfast was done, we were separated in

groups of four. We played games that you probably remember from your childhood, but then a little different. For example, the games 'koekhappen' en 'spijkerpoepen' were combined, so you had to try to eat the cookie while it was hanging somewhere close to your teammate's ass. We also played 'kratjedraaien' where you have to run around a beer crate 10 times and then run into the other direction. I kept hoping that people would bump into each other, but sadly they didn't. Better luck next year! Then it was time for a nice and welcome lunch. There was a home-made video on the screen, made by the IntroCee, that was really funny to watch. It turned out that the video had something to do with the next game. All of the senior students dressed up as characters and we had to find out what actions to perform in order to earn as much money as possible. We had to catch the fox from the song The Fox, show our fabulous dance moves and play the game chubby bunny for example. But as I was really tired and not very useful at collecting money, I decided to sneak away and be reunited with my bed for a couple of hours (the game was really awesome though). When I woke up, the game was finished and the sun was shining. A perfect time to sit in the grass and enjoy the game of soccer a few guys were playing. That was until we found out that the grass was covered with ants.

Dinner that day consisted of wraps, so there was no need to be hungry in the evening. It was my turn to help with the dishes. Most of the people worked hard, so fortunately it didn't take very long and it even was quite funny. When all of our stomachs were full and the dishes were done, it was time for a pub quiz! Our team was pretty big so in the end, the side of the table with the answering form was the side that answered the questions; when they didn't know it, the other side tried to help. The questions were quite awesome and covered almost every subject. If you liked to watch South Park, played video games like GTA and knew a lot about Groningen, you probably did well. Our team became second, so we could all start another awesome night with our heads held high. Not that I contributed much to our score, but I did beautify the answering forms with hearts, stars and some other pretty bad drawings.

Apparently morning gymnastics isn't enough to scare a group of Computing Science and Artificial Intelligence students, because the alcohol again flowed in abundance. I think I went to bed at about five o'clock, but there were even people who stayed up the entire night. When we were woken at nine a.m. the next morning, I was really relieved that we didn't have to exercise to earn our breakfast this time. After breakfast it was time to clean everything up and pack our bags, because sadly the camp was coming to an end. I was given the task to sweep the floor outside, so me and another boy worked really hard on that. Well not really, but it ended up clean anyway so who cares. I didn't quite get the fact that our breakfast was also the lunch, so I didn't have lunch until 5 o'clock that day, but that was fine. When it was time for our bus ride home, it was a lot more quiet than on the outward journey. Most people went to sleep the moment they sat down. Which is a good sign really, because if you're not exhausted after a weekend like this, then something went wrong. All in all, I really had a great time and I hope I can come again next year. I would also recommend the newbies of next year to join this camp: it's a great opportunity to meet people in a relaxed environment... And to the students that made this camp epic: thanks! You really made our weekend!


















STUDY ASSOCIATION Cover

# COLOPHON

The Brainstorm is a magazine published by study association Cover and is distributed among its members, staff members and other interested people. The Brainstorm comes out at least three times a year in an edition of 500.

Contact
Study association Cover
attn. The Brainstorm
PO box 407
9700 AK Groningen
brainstorm@svcover.nl
www.svcover.nl