# Objective-C, wat is dat?

ALbert Mietus

Vorig jaar vroeg een collega, die net een mooie Apple Notebook gekocht had of ik wist wat 'Objective-C' was. Zijn systeem gebruikte dat veel. Toevalling heb ik, jaren geleden, me verdiept in die *tegenhanger* van C++; en dus ik kon meteen melden dat het een OO programmeertaal is. "Die veel mooier, krachtiger en eenvoudiger is dan C++ of Java", kon ik niet laten om toe te voegen!

Een paar whiteboard-schetsjes gaven al snel aan dat de Objective-C makkelijk te lezen is. Maar ook dat het al weer jaren geleden was dat ik in Objective-C geprogrammeerd had. Na enige pogingen om iets van die verborgen kracht te onthullen, verdween –met de belofte om er ooit iets over te schrijven– 'ObjC' weer in de speekwoordelijke bureaula.

Maar onlangs, toen ik iets meer tijd kreeg en ik een stukje code wou schrijven moest ik weer denken aan die toch wel boute uitspraak. "Krachtig, elegant en snel ..." Kon ik nog een andere programmeertaal kiezen? Daarom werd besloten voor een dubbel project: de implementatie van een syslog-sign-verificator in objective-C. En tegelijkertijd een artikel over die taal schrijven.

## Inhoudsopgave

**1 Waar komt Objective-C vandaan? 1**

**2 Hoe is Objective-C dan anders ... ? 1**
- 2.1 Syntax . . . 2
  - 2.1.1 syntax overzicht . . . 3
- 2.2 Dynamic binding . . . 3
- 2.3 Object allocatie . . . 4
  - 2.3.1 Voorbeeld newMsg:: . . . 5
- 2.4 Super . . . 5
- 2.5 Runtime . . . 5
  - 2.5.1 Voorbeeld respondsTo: . . 6
- 2.6 Selectors . . . 6
  - 2.6.1 Voorbeeld selectors . . . . 6

**3 Wat kan je met Objective-C? 7**
- 3.1 OO-technieken . . . 7
  - 3.1.1 Delegation . . . 7
  - 3.1.2 Target-action paradigma . 7
  - 3.1.3 Archiving . . . 7
- 3.2 Toekomst/gebruik . . . 7
- 3.3 Modelering . . . 8

**4 Bijlages 8**
- 4.1 Kaders e.d. . . . 8
  - 4.1.1 termen . . . 8

# 1 Waar komt Objective-C vandaan?

Als we terugkijken op de korte, maar turbulente, geschiedenis der programmeertalen, dan heeft vooral 'C' geschiedenis geschreven. Naast dat het één van de meest gebruikte talen is, heeft 'C' ook veel andere talen beïnvloed. Moderne talen zijn vaak afgeleid van 'C'; vaker dan van bijvoorbeeld *Pascal*.

'C' is zo'n 3 decennia geleden onstaan als 'K&R-C' en later verbeterd tot 'ANSI-C'. Nog later, zo rond 1985, melde C++ zich als opvolger; eerst onder de naam 'C with Objects'. Het was 'C' met OO (Object Oriented) uitbreidingen.

Objective-C stamt ongeveer uit diezelfde tijd. Ook Objective-C is onstaan als een OO-uitbreiding op '(ANSI-)C'. Maar in tegenstelling tot C++, dat sterk gebaseerd is op Modula, heeft Brad J. Cox, de bedenker van Objective-C, de OO aanpak van Smalltalk gevold.

# 2 Hoe is Objective-C dan anders ... ?

Het leuke van Objective-C is dat het *'anders' is*. Voordat beschreven wordt waarom dat dan zo leuk is, vergelijken we Objective-C eerst met andere talen. Vooral met C++, omdat dat ook

een C variant is. En omdat C++ alom bekend is.

We zullen zien dat de belangrijke verschillen verklaarbaar zijn uit genoemde historie. De ervaringen van de 'taal-bedenkers' hebben gezorgd voor een geheel ander benadering. Maar juist door die verschillen, zorgt enige kennis van Objective-C voor beter begrip van OO; ook in C++.

## 2.1 Syntax

Voordat er ingaan kan worden op de echte, diepere, verschillen tussen Objective-C en C++, moet eerst iets vermeld worden over de syntax van Objective-C. Deze is namelijk '*anders*'. Veel OO-talen gebruiken een notatie als '`anObject.doIt()`'; in Objective-C schrijven we dat als '`[anObject doIt]`'.
De betekenis is gelijk; in beide gevallen wordt een methode aangeroepen van de class van '`anObject`'.

Voor parameters wordt een infix notatie gebruikt, i.p.v. de gebruikelijke lijst tussen haakjes. Elke parameter wordt voorafgegaan door een dubbele punt. Met optioneel een naam; zoals in `[anArray insert: anObject at: 2]`. Nadat men hieraan gewent is, kan het handig zijn in het onthouden van volgorde van parameters. Maar het is natuurlijk slechts syntactische suiker.

Verder is Objective-C eenvoudig; het is ANSI-C met zo'n 7 extra taalconstructies. Bovenstaande is de complexste. Zie het overzicht voor de andere.

#### 2.1.1 syntax overzicht

De syntax van Objective-C is eenvoudig leesbaar. Objective-C is namelijk gewoon ANSI-C met zo'n zeven (7) extra taalconstructies.

De syntax ("`[ontvanger bericht: parameter]`") om een object iets te laten doen is reeds in de hoofdtekst verklaard. Bijzonder hieraan is de notatie met vierkante haken en de parameters. De dubbele punt is hierbij onderdeel van de naam. Het geheel is een expressie, die een waarde kan opleveren.
Gebruikelijk is om, als er geen functionele returnwaarde is, het object zelf te retourneren; zodat een geneste aanroep mogelijk is.

Classes moeten ook gedeclareerd en geïmplementeerd worden. Dit gebeurd met:

**@interface** Hiermee wordt een class gedeclareerd; meestal in een .h file.
Met `@interface B : A` wordt de class B gedefinieerd, die erft van class A. In Objective-C heeft elke class altijd één superclass; gebruik '`Object`' als er geen specifiekere class is.

Direct na dit statement worden de instance variabelen (members, in C++) gedeclareerd. En daaronder class en instance methoden; de eerste vooraf gegaan plus-teken, de laatste door een min-teken. Zoals bij C functies, worden methode declaraties afgesloten met een puntkomma.

**@end** Hiermee wordt een bovenstaande interface definitie afgesloten. En ook het hieronder benoemde *implementatie* deel.

**@implementation** Hiermee wordt de implementatie van een class begonnen; meestal in een .m file. De naam van de class die geïmplementeerd wordt, moet opgegeven worden.
Optioneel kunnen zowel de superclass als de instance variabelen herhaald worden; beter is het om ze weg te laten.

Tussen `@implementation` en de afsluitende `@end` worden alle methoden geïmplementeerd; ongeveer zoals een C-functie.
Zowel in de interface gedeclareerde methoden als elders gedeclareerde methoden kunnen geïmplementeerd worden.

Hiernaast kunnen de keywords `@class`, `@protocol`, `@private`, `@protected` en `@public` gebruikt worden.

Meer informatie kan gevonden worden op het web. O.a. de site van Apple bevat een goede introductie.

## 2.2 Dynamic binding

In de biografie van C++, en haar bedenker Stroustrup, is te lezen waarom C++ gebaseerd is op '*static-* (of *early-*) *binding*': het is een poging om typische fouten (van Stroustrup) te voorkomen. Als de compiler maar zo snel mogelijk de implementatie-functie kan kiezen voor elke methode-aanroep, kan hij ook controleren alle (parameter)typen correct zijn. Maar hierdoor legt de compiler ook direct vast welke code uitgevoerd wordt, voor elke call van een methode; ook als subclasses gebruikt worden. Vaak wordt dit gezien als een (te) sterke OO-restrictie.
In C++ kunnen we een methode dan ook '*virtual*' maken; de compiler gebruikt dan een jumptable om eventueel een subclass implementatie te gebruiken. Een stukje dynamische flexibiliteit is hiermee beschikbaar in C++[1].

In Objective-C is voor een geheel andere aanpak gekozen. De compiler maakt geen keuze in de te gebruiken (methode) implementatie; nooit! Die keuze wordt –altijd– pas runtime gemaakt. Door deze '*late* (of *dynamic*) *bindig*' wordt altijd de passende implementatie

[1] Het is jammer voor de pogingen van Stroustrup, maar het gebruik van *virtual member functions* –dat min of meer gebruikelijk geworden is–, introduceert weer de fouten die hij wou voorkomen. In talen als Java is 'virtual' dan ook ingebouwd. Het voorkomen van fouten wordt anders opgelost. Dit zullen we ook zien in Objective-C.

gebruikt. Ook als subclasses gebruikt worden, ook als die een methode opnieuw implementeert. Dit geeft de programmeur veel vrijheid; en geeft kracht aan de taal.

Immers, niet het type dat de programmeur intikte moet bepalen hoe een instance reageert op een methode (zoals een event), maar de daadwerkelijke (sub)class.

Een klassiek voorbeeld hiervan zien we in mijn **LogMsg** class.
Elke logregel wordt opgeslagen in een object. Met '`[aLogLine timestamp]`' kan de tijd van in zo'n logregel uitgelezen worden. Omdat het scannen van een regel complex is, is een designkeuze gemaakt om o.a. het tijdstip in een aparte (instance) variabele op te slaan. Dit gebeurd door het aanroepen van de interne methode '`-parse`'. De implementatie van `-timestamp` roept dus eerst -parse aan: '`[self parse]`'.

Sommige logregels zijn echter bijzonder; zo zijn er regels die digitale handtekeningen bevatten. Daarvoor wordt een subclass gebruikt: SignatureBlock. Het parsen daarvan is anders. Dus is `-parse` opnieuw geïmplementeerd. De methode `-timestamp` is niet veranderd, die roept immers `[self parse]` al aan.
Een codefragment als `[someLine timestamp]` levert altijd het juiste tijdstip op. Als `someLine` een SignatureBlock is, wordt automatisch de uitgebreide versie van `-parse` gebruikt. Ook als de variabele `someLine` als **LogMsg** gedeclareerd is. Immers, de runtime bepaald de class; niet de programmeur.

## 2.3 Object allocatie

In een OO taal is het essentieel dat objecten aangemaakt kunnen worden. In veel talen gebeurt dat door 'contructors'. Niet in Objective-C! Omdat elke Class een object is, kunnen we het *construeren* van een instance overlaten aan de Class zelf. Default wordt `+new` gebruikt. –een class methode, zonder parameters.
Het gebruik is eenvoudig: "`anInst = [ anObject new ]`"

Dit lijkt sterk op het gebruik van een contructor, maar het is anders. De `+new` is een gewone methode; die als effect heeft dat er een object aangemaakt wordt. De programmeur kan echter ook andere methoden definieren. Gebruikelijk is het om deze wel te laten beginnen met "new".

Het is aan de programmeur om een juiste implementatie te maken van zo'n `+new`*`XXX`* methode. Normaal zal voldoende ruimte gealloceerd worden, waarna initialisatie volgt. De (erfbare) implementatie van **Object** is dan ook: "`return [[self alloc] init]`".
Vaak zal het initialisatie deel uitgebreid worden, maar soms wordt de methode ook geheel anders geïmplementeerd. Bijvoorbeeld door een bestaand object te genbruiken.

### 2.3.1 Voorbeeld newMsg::

In mijn syslog-sign-verificator gebruik ik deze mogelijkheid.
Voor elk logmelding wordt een object van het type **LogMsg** aangemaakt. Sommige berichten zijn echter bijzonder; daarom bestaan twee subclasses: **SignatureBlock** en **CertificateBlock**.

Slechts in de method `+newMsg` is de kennis vast gelegd hoe zo'n bijzonder logmelding herkent kan worden en wanneer welke class nodig is.

```
+ newMsg:(const char*) msg                    //  class: LogMgs
{
  Class use_class = self;                     //  Default, We return a ''LogMsg''
  if (strstr(msg, "@#sigSIG"))
    use_class = [SignatureBlock class];       //  But overrule when needed ! ...
   if (strstr(msg, "@#sigCer"))
    use_class = [CertificateBlock class];     // ... with a subclass

  return [[use_class alloc] initMsg: msg];    // Make & init the object and return
}
```

Met "`[LogMgs newMgs:a_log_line]`" wordt dus altijd een object aangemaakt dat een LogMsg is. Maar indien nodig wordt automatisch subclass van **LogMsg** gemaakt!

## 2.4 Super

De eerder gebruikte methode `-parse` is geïmplementeerd in zowel `LogMsg`, als in SignatureBlock; de laatste is een uitgebreidere variant van de eerste. Die uitgebreidere implementatie begint dan ook met "`[super parse]`". Hiermee wordt de `-parse` implementatie van de vererfde class (hier: `LogMsg`) aangeroepen.

Het object `super` is min of meer gelijk aan `self`; het bevat dezelfde data. Alleen wordt op een andere manier gezocht naar de implementation van een message. De "eigen" implementatie wordt als het ware overgeslagen.
In veel OO-talen is zoiets niet mogelijk. In C++ wordt dan vaak een hulp-functie ingezet. Doordat "`super`" in de Objective-C taal ingebouwd is, is het generiek bruikbaar. Een groot voordeel; wat er voor zorgt dat de code leesbaar en erg inzichtelijk is.

## 2.5 Runtime

Objective-C is een gecompileerde taal, maar met een runtime systeem. Dit runtime systeem omvat de (interne) functies om bovengenoemde dynamic binding te ondersteunen. Maar het biedt veel meer. Zo *ontsluit* het, via de top-class "**Object**" een hoop informatie, die de compiler genereert.

Zo is het mogelijk om –aan elk object– te vragen of het van een bepaalde class is, of daarvan erft. Het is zelfs mogelijk om te vragen of een methode gebruikt kan worden.
De programmeur kan zo testen of een object een bepaalde message kan verwerken en dit slechts versturen indien geen fout oplevert. Zie ook het voorbeeld.

Omdat dit typisch onmogelijk is in C++ of Java, zal de kracht hiervan niet meteen duidelijk zijn. Maar gecombineerd met selectors (zie hieronder), zorgt dit ervoor dat er veel minder "gesubclassed" moet worden. Ook templates (generieke subclasses in C++) zijn hierdoor niet nodig.Vooral voor programmeurs betekend dat: gewoon minder saai werk. Een class als "*Set of ...*" bestaat niet. In Objective-C volstaat in de class "**Set**"; voor elk objecttype.

#### 2.5.1 Voorbeeld respondsTo:

Onderstaande codefragment laat zien dat dynamisch opgevraagd kan worden of een object kan reageren op een message.

```
    ⋮
  if ([testObject respondsTo: @selector(timestamp)])
    t=[testObject timestamp];
  else
    printf("Object %s reageert niet op timestamp", [testObject name]);
    ⋮
```

Slechts als de runtime aangeeft dat dit `testObject` een implementatie voor `-timestamp` heeft geïmplementeerd of geërft, zal de code deze methode gebruiken.

## 2.6 Selectors

We kunnen bovenstaand voorbeeld nog generieker maken omdat messages ook een type hebben: "SEL" (van selector). We kunnen ook variabelen declareren van dit type. En we kunnen die *bewaarde* messages versturen naar een object. Daarvoor gebruiken we "-perform:" (of één van de variaties met parameters). Een codefragment als "[anObject performv: aMessage : args]" is dus letterlijk het meest universele stukje OO-code dat kan bestaan. Er wordt een message verstuurd aan een object. Elk message kan verstuurd worden, naar elke mogelijk instance van elke mogelijk object. En dat met een willekeurig aantal parameters.

Buiten zijn context is dergelijke code volstrekt waardeloos. Maar bedenk dat dit statement onderdeel kan zijn van een implementatie van een class. En dat zowel 'anObject' als 'aMessage' object variabelen kunnen zijn; die gezet kunnen worden middels andere methoden

#### 2.6.1 Voorbeeld selectors

Als voorbeeld van de kracht van selectors, hieronder een OO-module testharnas.

In dit codefragment kan `testObject` *bestookt* worden messages die de gebruiker op de commandline opgeeft. Of in een script. De naam van die methoden, hoeft tijdens compileren niet bekend te zijn!

```
    ⋮
  for (i=1; i<argc; i++) {
    SEL s;
    s = sel_getUid(argv[1]);   // char* 2 SEL conversie
    if ([testObject respondsTo: s]) {
      printf("Testing object %s with %s :", [testObject name], sel_getName(s));
      [testObject perform: s];
    } else ...
  }
    ⋮
```

In deze code wordt tekst, hier als programma-argument, gebruikt om een SELector te maken. Die wordt, conditioneel, verstuurd naar het testObject.

In werkelijk zal deze code natuurlijk uitgebreid worden om parameters en/of returnwaarden te kunnen verwerken. Voor de eenvoud is dat hier niet gedaan.

# 3 Wat kan je met Objective-C?

## 3.1 OO-technieken

De kracht van Objective-C komt uit de eenvoud van de taal. Maar ook door de doordachte (en) dynamische aspecten. Daardoor zijn ook minder bekende OO- (c.q. programeer-) technieken mogelijk. Een aantal worden hieronder behandeld.
Natuurlijk zijn die niet specifiek voor één taal. Ze zijn ook toepasbaar in talen die deze technieken niet direct ondersteunen. Maar in Objective-C programma's worden ze vaak gebruikt, omdat het zo gemakkelijk is.

### 3.1.1 Delegation

Messages zijn types die gemanipuleerd kunnen worden, daardoor is het mogelijk om *'delegatie'* te gebruiken. Dit is een krachtige (OO) faciliteit; die slechts zelden ondersteund word.
Het komt er op neer dat berichten, die een object niet zelf kan verwerken, doorgestuurd worden naar een ander –gedelegeerd– object. Dat object zorgt dan voor het verwerken van het bericht.

In Objective-C is dat eenvoudig. Indien een object een niet geïmplementeerde message ontvangt, zal de runtime proberen of delegatie (voor die class) mogelijk is. Pas als dat niet kan, wordt een error gegenereerd.

Middels deze techniek kunnen bijvoorbeeld "proxy" objecten gemaakt worden. Hiermee kan een remote-object "bedient" worden. Ook kan het gebruikt worden om mooie *abstracte* interfaces te gebruiken.

### 3.1.2 Target-action paradigma

Vaak is een object bedacht voor een bepaalde actie, waarbij een ander object echter bepaald wanneer dat moet gebeuren. Het duidelijk voorbeeld hiervan is te zien bij buttons in een GUI. Als de gebruiker zo'n button *indrukt* verwacht hij dat een actie geactiveerd wordt. In een goed ontwikkelde applicatie zal de ***button-window*** dat niet zelf uitvoeren; maar doorgeven aan een object dat die actie implementeert.

Er zijn vaak meerdere van dergelijke buttons. En heel veel mogelijke acties, met bijbehorende objecten.

In Objective-C kan dit generiek opgelost worden, door het *"target-action" paradigma* te gebruiken. Er kan dan een generieke class "**button**" gemaakt worden. Elk object hiervan heeft twee (instance) variabelen `actie` en `target`, naast de variabelen die bijvoorbeeld gebruikt worden voor het teken van het window. Deze twee kunnen gezet worden tijdens initialisatie, of middels een message. En zo worden gebruikt als de buttons ingedrukt worden. Dan zal de `actie` –een selector– verstuurd worden naar het `target` –een object–, zoals eerder beschreven. De code hiervoor hoeft maar een keer geschreven te worden; heel OO! Het maken van subclasses voor elke actie, of per objectclass is in Objective-C niet nodig.
In een taal als C++ zijn dan vaak subclasses nodig; of templates. Of wordt een ander systeem gebruik; zoals MS-Windows events, waardoor de applicatie minder portable is.

Hoewel GUI's een natuurlijk voorbeeld vormen, is dit veel generieker toepasbaar. Vooral ook in embedded systemen, waar veel acties gemoduleerd worden.

### 3.1.3 Archiving

In Objective-C is het mogelijk om zowel data (objecten) als code (classes) dynamisch te laden, na aanvang van het programma. De runtime biedt hiervoor support. Beide technieken zijn alom bekend in niet OO-talen en in geinterpreteerde (OO-) talen. Objective-C is een van de weinige gecompileerde talen die het ook biedt.

Hierdoor is een programma uitbreidbaar. Ook is het mogelijk om één applicatie te splitsen in meerdere, apart te ontwikkelen en te testen delen. En natuurlijk zijn "*plug-in's*" niet meer weg te denken in moderne systemen.

Hoewel de taal ondersteuning hiervoor ondersteuning biedt; moet elk platform zijn eigen invulling hiervoor geven. Als we hier kijken naar het (verouderde) NextStep platform, dan gebeurde dit middels *Bundels*. Hiermee konden locale settings (zoals taal); maar ook de complete GUI, inclusief code, layout en initiële data gelezen worden. En dus buiten het eigenlijke programma opgeslagen worden.

## 3.2 Toekomst/gebruik

Hoewel Objective-C ongeveer even oud is als C++, is het duidelijk veel minder gebruikt. Commercieel is het vooral gebruikt in NextStep en in Apple's MAC OS-X.
Apple's systeem classes ondersteunen zowel Java

als Objective-C. Enkele faciliteiten zijn echter alleen vanuit Objective-C bruikbaar; de taal Java is dan niet krachtig genoeg.

Objective-C is echter bruikbaar op alle systemen waarvoor de GCC compiler beschikbaar is; zoals (free)BSD en Linux. Een file met de extensie .m zal automatisch als Objective-C code gecompileerd worden. Door `objc/Object.h` te includen zijn de standaard types bekent.

### 3.3 Modelering

Behalve om in te programmeren, kan Objective-C ook gebruikt worden als modeleringstaal. Juist omdat typische OO aspecten kort en bondig op te schrijven zij in Objective-C, leent deze taal zich daar erg goed voor. Natuurlijk is het dan niet nodig om volledig (compileerbare) programma's te schrijven. Een "pseudo Objective-C notatie" volstaat dan ruimschoots.

---

## 4 Bijlagen

### 4.1 Termen

Vrijwel elke OO taal gebruikt een ander terminologie. Onhandig maar onvermijdelijk. Zoveel mogelijk zijn generieke termen gebruikt. Maar voor de volledigheid een kort woordenboek.

**method(e)** Een functie specifiek voor een object. Werkt deze op een instance dan wordt de naam voorafgegaan door een min-teken. Een plus-teken geeft aan dat het een "class method is"; er is dan geen instance.

**message** In ObjC, de methode selector (naam) plus de argumenten die middels een OO-aanroep aangeeft wat een object moet doen.
N.B. De naam van een message begint nooit met een plus- of min-teken; de tekens worden uitsluitend gebruikt bij methoden; om aan te geven of het een om class- c.q instance-methode is.

**selector** De gecompileerde versie van een message. Dit is efficiënter dan de (ascii) naam gebruiken. Er is een 1 op 1 relatie tussen selectors en messages.

**implementation** Een term die soms gebruikt wordt om een specifieke methode voor één specifieke class aan te geven. Meerdere classes kunnen dezelfde method implementeren. De runtime selecteert altijd een van die *implementation*'s.

**self** Het object zelf, binnen een implementation. Ofwel de ontvanger van een message.
In C++ : 'my'

`Dit is versie: $Id: Wat_is_ObjC.hltex,v 1.13 2003/07/02 18:32:18 albert Exp $`