Aantekeningen bij iOS Programming Big Nerd Ranch Guide 3rd edition

**Hoofdstuk 1 – A simple iOS Application**

Geen opmerkingen, is een inleidend hoofdstuk waarin de Quiz applicatie wordt gebouwd. Het MVC model wordt uitgelegd en aan de hand hiervan wordt het object diagram van de Quiz applicatie getoond.

**Hoofdstuk 2 – Objective-C**

Geeft een inleiding in de Objective-C taal, noem het een wrap-up.

Een **object** is een instantie van een class. Een **class** is vergelijkbaar met de struct in C, met dien verstande dat een class naast data ook methoden (functies) kan bevatten.

Een object wordt op de volgende manier gecreeerd en geinitialiseerd:

MyClass \*myObject = [MyClass alloc]; // alloc is een **class** methode

[myObject init]; // init is een **instance** methode

Dit kan efficient worden gecodeerd als:

MyClass \*myObject = [[MyClass alloc] init];

NB meestal kan uit een diversiteit aan init methoden gekozen worden.

NB2 er wordt altijd via een pointer aan een object gerefereerd.

Naar objecten kunnen messages verzonden worden: hiermee wordt een instantie (object) gevraagd een method uit te voeren. Een message als geheel heeft de volgende syntax:

[myObject myClassMethod: argument1

label2:argument2

label3:argument3];

Hier is myObject de receiver van het message, myClassMethod de **selector** (uit te voeren method) en label2 en label3 de selector labels. Argument1, argument2 en argument3 zijn de parameters die worden meegegeven. Praktijkvoorbeeld:

[partyInstance addAttendee: somePerson withDish:steak];

**Strings**

Het @ teken is een shortcut voor het creeren van een string, dat hoeft niet met alloc te gebeuren. Voorbeeld:

NSString \*myString = @”Hello world!”;

int len = [myString length];

NSLog is een voorgedefinieerde methode waarmee van alles op de console kan worden gelogd:

NSLog(@”Dit is een log message: %@ (getal %d)”, myString, 3);

Met %@ kunnen allerlei objecten (waaronder strings) worden getoond, het is aan de NSLog functie te bepalen hoe de lay-out er uit ziet. Onder water wordt het description message naar het argument gestuurd, de implementatie van die methode voor de betreffende klasse bepaalt dan hoe het object wordt weergegeven. Door de description methode te overriden (zie verder) kan een eigen invulling hieraan gegeven worden.

Andere format parameters zijn:

%d voor gehele getallen (bijv 3)

%f voor floating point getallen (bijv 2.5)

%c voor karakters (bijv ‘A’)

**Arrays: NSArray en NSMutableArray**

NSArray is een ‘immutable’ array, de elementen kunnen na instantiatie niet meer aangepast worden. Dit in tegenstelling tot NSMutableArray, wat een subclass is van NSArray. Voorbeeld:

NSMutableArray myMutArr = [[NSMutableArray alloc] init];

MyClass \*myObject = [[MyClass alloc] init];

[myMutArr addObject: myObject]; // voegt myObject aan einde toe

int nObj = [myMutArr count]; // geeft #objecten in myMutArr

[myMutArr addObject: myObject // voegt myObject op positie nObj toe

atIndex: nObj] // NB 1e index is 0!

NSObject \*myObj2 = [myMutArr objectAtIndex:1]

// geeft pointer naar object op positie 1

**Een class (subclass) creeren**

In XCode door File -> New -> New File te kiezen. Vervolgens Cocoa in de Mac OSX sectie selecteren en daarna Objective-C class selecteren. Hierna verschijnt een scherm waarin de naam van de nieuwe class en de superclass moeten worden ingegeven. De superclass staat standaard op NSObject, de moeder van alle classes.

XCode maakt nu een .h header en .m code file aan, beide met de naam van de nieuwe class. Stel dat deze BNRItem heet, dan worden BNRItem.h en BNRItem.m aangemaakt.

BNRItem.h is de interface file van class BNRItem waarin de class wordt gedefinieerd. Deze ziet er als volgt uit:

#import <Foundation/Foundation.h>

@interface BNRItem : NSObject

// hier worden instance variabelen gedefinieerd, bijv:

NSString \*itemName;

// vervolgens worden de class methoden gedefinieerd, bijv:

- (NSString \*)itemName; // getter van inst.var. itemName

- (void)setItemName(NSString \*name); // setter van inst.var. itemName

@end

Voor elke instance variabele kunnen accessor methoden (get en set) worden gedefinieerd, in de .h file **na** de instance variabelen en **voor** de @end. De getter heeft altijd dezelfde naam als de instance variabele zelf (itemName in het voorbeeld), de setter methode naam is ‘set’ gevolgd door de naam van de instance variabele.

BNRItem.m is de implementation file waarin de class methoden worden geimplementeerd die zijn gedefinieerd in BNRItem.h. Tevens kunnen van de superclass geerfde methoden worden overridden, deze hoeven dan niet nog eens in de header file genoemd te worden.

Een eenvoudig voorbeeld van BNRItem.m:

@import “BNRItem.h”

@implementation BNRItem

// hier komen de methode implementaties, bijv.

- (NSString \*)itemName

{

return itemName;

}

- (void)setItemName(NSString \*name)

{

itemName = name;

}

@end

**Initializers**

Een object (= class instance) wordt gecreeerd dmv de alloc methode en kan geinitialiseerd worden dmv de init methode, bijv:

BNRItem \*p = [[BNRItem alloc] init];

In dit geval zal init default waarden voor de instance variabelen moeten invullen zodat het object p netjes geinitialiseerd is voor verder gebruik.

Nu kunnen er meerdere initializers worden gedefinieerd waaraan parameters kunnen worden meegegeven om de respectievelijke instance variabelen te initialiseren. De naam van alle initializers beginnen met ‘init’. Een van de initializers is de zg **designated initializer**, deze zorgt er per definitie voor dat de instance volledig en correct wordt geinitialiseerd. Dit betekent zoveel als dat de belangrijkste, schrijfbare (= non read-only) instance variabelen worden geinitialiseerd.

Voor class BNRItem uit het boek wordt dit:

- (**id**)initWithItemName : (NSString \*)name

valueInDollars : (int)value

serialNumber : (NSString \*)sNumber;

**id** is hier een pointer naar een willekeurig object, vergelijkbaar met void in C. De reden dat het return type niet (BNRItem \*) is maar ‘id’ heeft te maken met subclassing. Wordt nl dit init message naar een instance van een subclass van BNRItem gestuurd, dan zal die een instance van de betreffende subclasse moeten teruggeven in plaats van BNRItem. Door een ‘id’ terug te geven wordt dit ondervangen. Het overriden van deze init methode door de subclass is nl niet mogelijk, omdat deze init methode dan exact dezelfde naam zou moeten krijgen, wat niet is toegestaan in Objective-C.

**isa**

Elk object dat dmv alloc wordt gecreeerd heeft een instance variable **isa**. De alloc heeft tot gevolg dat isa wordt ingesteld op een pointer naar de class die de instance variable creeerde. In ons geval is dat een pointer naar BNRItem.

**self**

Binnen een methode is **self** een impliciete lokale variabele die verwijst naar het object dat het message gestuurd kreeg, in java wordt dit met ‘this’ aangeduid. Hiermee kan een object een message naar zichzelf sturen.

**super**

Wanneer een message naar een object wordt gestuurd, dan zal tijdens de executie van het programma eerst gezocht worden binnen de class van dat object naar die methode (= message). Wordt deze niet gevonden – omdat de methode niet binnen die class is gedefinieerd – dan zal gezocht worden in de superclass van het object, etc totdat de methode wordt gevonden of de top NSObject wordt bereikt en een exceptie wordt gegenereerd.

Wordt een methode voorafgegaan door **super** dan start het zoeken naar de methode een niveau hoger in de superclass. Voorbeeld:

self = [super init];

Dit statement staat standaard aan het begin van een overridden methode (init in dit voorbeeld) en heeft als doel dat wat de methode in de superclass uitvoert behouden blijft.

**Class methods**

Er zijn twee soorten methoden: **instance** methoden en **class** methoden. Instance methoden (= messages), zoals init, worden naar instanties van de class (= objecten) gestuurd. Class methoden, zoals alloc, worden naar de class zelf gestuurd.

Een class methode creeert normaliter nieuwe instanties van de class of geeft juist een globale eigenschap van de class terug. Class methoden werken niet op een instance en hebben geen toegang tot instance variabelen.

In Objective-C worden deze syntactisch van elkaar onderscheiden door het eerste karakter in de declaratie: een ‘-‘ voor instance methoden en een ‘+’ voor class methoden. In Objective-C worden class methoden eerst gedeclareerd en daarna de instance methoden.

Het boek definieert een BNRItem class methode randomItem, die een BNRItem object creeert dat met random waarden wordt geinitialiseerd. Een ander voorbeeld van een class methode is **stringWithFormat**, een class methode van NSString. Deze creeert een string die volgens het opgegeven format geinitialiseerd is.

**Fast enumeration**

Dit is in Objective-C 2.0 geintroduceerd. Het biedt een efficientere manier om een for loop te coderen, zoals:

NSMutableArray \*items = [[NSMutableArray alloc] init];

// code die items toevoegt is hier weggelaten

for (BNRItem \*item in items) {

NSLog(@”%@”, item);

}

Impliciet worden hier alle elementen van items behandeld, beginnend met index 0.

**Hoofdstuk 3: Managing memory with ARC**

**Heap**

Objective-C objecten worden opgeslagen in de **heap**, dat overkomt als een grote hoop geheugen waarin allerlei objecten zijn opgeslagen. In Objective-C wordt mbv pointers aan objecten gerefereerd, de pointer geeft het adres van het object in de heap.

**Stack**

De **stack** wordt gebruikt door methoden om hun lokale variabelen op te slaan. Wanneer een methode wordt aangeroepen wordt een stuk geheugen bovenop de stack voor de methode gealloceerd. Wanneer de methode eindigt wordt het geheugen weer vrijgegeven. Roept een methode zelf een methode aan, dan gebeurt hetzelfde. Op deze manier groeit en krimpt de stack. Qua geheugen management is dit anders georganiseerd dan de heap.

**Object ownership**

In het kader van geheugenbeheer is dit een relevant onderwerp. Geheugen is beperkt en objecten die niet meer nodig zijn moeten het gebruikte geheugen vrijgeven om te voorkomen dat het geheugen na verloop van tijd volloopt.

* Een methode is eigenaar van een object wanneer de methode een lokale pointer variabele heeft die naar een object wijst.
* Een object met een instance variabele die wijst naar een ander object is eigenaar van dat object.

Een object dat geen eigenaar heeft moet worden verwijderd, anders is sprake van een memory leak. Een object met tenminste een owner mag niet worden verwijderd. Stuur je er nl een message naar toe dan crasht het programma.

Vanaf iOS5 zorgt Automatic Reference Counting voor het geheugen beheer. Bij oudere versies van iOS moet dit zelf worden geregeld (dmv retain en release statements). ARC is echter niet zaligmakend, soms moet er worden ingegrepen om problemen op geheugengebied te voorkomen.

Een object verliest een eigenaar als:

* Een variabele die naar het object wijst naar een ander object gaat wijzen
* Een variabele die naar het object wijst op **nil** wordt gezet
* Een variabele die naar het object wijst zelf wordt verwijderd

Is dit het geval dan zorgt ARC ervoor dat een object dat geen eigenaren heeft wordt verwijderd.

In het geval een object een eigenaar heeft is sprake van een **strong reference**. Het is ook mogelijk dat een variabele geen eigenaar wil zijn van het object waarnaar wordt verwezen, dit wordt **weak reference** genoemd. Dit hebben we nodig wanneer sprake is van een zg **retain cycle**: twee of meer objecten hebben sterke referenties naar elkaar. Deze kunnen niet worden verwijderd door ARC. De oplossing is om de referenties weak te maken.

Een praktijkvoorbeeld is de moeder-kind relatie tussen objecten. Het moeder object verwijst naar het kind object en vice versa, zodat de relatie van twee kanten duidelijk is. In geval van strong reference is hier sprake van een retain cycle. Door de referentie van kind naar moeder **weak** te maken, weet ARC dat het kind object op een gegeven moment verwijderd kan worden – dat is als deze alleen nog maar weak references heeft naar andere objecten. Als dat gebeurd is kan ook het moeder object verwijderd worden. Een voorbeeld van een declaratie is:

\_\_weak BNRItem \*container;

Een variant op de weak reference is **unsafe unretained**. Het verschil met de weak reference is dat de unsafe unretained reference niet automatisch op nil wordt gezet als het object waarnaar wordt verwezen wordt verwijderd. Dit is een nogal onveilige situatie en bestaat voornamelijk vanuit backwards compatibility redenen; oudere versies dan iOS5 kennen de weak reference niet.

**Properties**

Een property wordt gedeclareerd in de interface van een klasse, waar de methoden worden gedeclareerd. Een property declaratie heeft tot gevolg dat een instance variabele wordt gedeclareerd en automatisch een setter en getter voor de instance variabele, die hoeven dus niet meer expliciet te worden gedeclareerd in de interface file.

Voorbeeld:

@property NSString \*itemName; // juist voor de @end in de .h file

De setter setItemname en de getter itemName zijn hiermee automatisch gedeclareerd.

Een property heeft een aantal attributen:

* **nonatomic** / **atomic**: heeft met multi-threaded applicaties te maken, dit is buiten de scope van het boek. Wij gebruiken enkel nonatomic. Dit is niet de default optie, dus nonatomic moet altijd expliciet opgegeven worden.
* **readwrite** / **readonly**: een readwrite property declareert zowel een setter als een getter, een readonly enkel een getter. De default is readwrite. Meestal gebruiken we readwrite, soms ook readonly bijv voor een systeemdatum die niet dmv een setter kan worden aangepast.
* **strong** / **weak** / **assign**: de default is assign dat van toepassing is op properties die niet naar objecten verwijzen (zoals een int valueInDollars).
* **copy**: heeft tot gevolg dat in de synthesized setter (zie hieronder) de waarde niet dmv assignment wordt doorgegeven, maar door een copy actie:

- (void)setItemName( NSString \*name)

{

itemName = [name copy]; // ipv itemName = name!!!

}

Dit is een voorbeeld van veilig programmeren. Wordt ‘name’ nl veranderd dan heeft dit geen effect op de instance variabele itemName.

In de .m implementatiefile kan een property **synthesized** worden. Dit heeft tot gevolg dat de code voor de setter en getter automatisch wordt gegenereerd:

@implementation

@synthesize itemName, instVar1, instVar2;

Hiermee wordt de code voor de setters en getters van de properties itemName, instVar1 en instVar2 gegenereerd. Properties hoeven niet in aparte @synthesize statements opgenomen te worden, kan in 1 statement.

Meestal hoeft geen code voor setter of getter geschreven te worden, tenzij er extra werk moet worden uitgevoerd, bijv:

- (void)setContainedItem:(BNRItem \*)i

{

containedItem = i;

[ i setContainer:self] //extra statement dat wordt uitgevoerd

}

De synthesized setter zal de tweede regel code niet bevatten, daarom moet de setter handmatig worden geschreven. Dit levert geen conflict op met de synthesized setter, er is hier sprake van override.

Een property is feitelijk hetzelfde als een instance variabele. Het volstaat dan ook om enkel de property itemName op te nemen, dan wordt automatisch een instance variabele met dezelfde naam gecreeerd.

**Dot syntax**

De volgende regels code zijn identiek:

int value = [item valueInDollars]; // getter

int value = item.valueInDollars;

En deze ook:

[item setValueInDollars:5]; // setter

item.valueInDollars = 5;

De voorkeur is de [ en ] notatie omdat daarmee duidelijk wordt dat er een message naar een object wordt gestuurd.

**Hoofdstuk 4: Delegation and Core Location**

In dit hoofdstuk wordt de WhereAmI applicatie gebouwd die draait op de iPhone. Deze maakt gebruik van het Core Location framework (een framework is een verzameling gerelateerde classes). Maak in XCode een iOS single view application aan met de naam WhereAmI. Dit resulteert in .h en .m files genaamd WhereAmIViewController en WhereAmIAppDelegate.

Nadat het WhereAmI project is aangemaakt in XCode moet het Core Location framework worden toegevoegd. Selecteer hiertoe de WhereAmI target in XCode en voeg deze in de ‘Link Binaries With Libraries’ sectie toe door op het + teken te klikken.

Het Core Location framework omvat de classes die het applicaties mogelijk maken om de geografische locatie van een device te bepalen. Om deze te kunnen gebruiken moet de CoreLocation header file in WhereAmIViewController.h worden geimporteerd.

De class **CLLocationManager** is de interface met de locatie hardware van het device. In WhereAmIViewController.h wordt een location manager pointer gedeclareerd:

CLLocationManager \*locationManager;

De msg initWithNibName:nibNameOrNil:nibBundleOrNil wordt naar WhereAmIViewController gestuurd bij het starten van de applicatie. Dit gebruiken we om de nodige initialisaties uit te voeren:

- (id)initWithNibName:(NSString \*)nibNameOrNil

bundle: (NSBundle \*)nibBundleOrNil

{

self = [super initWithNibName:nibNameOrNil bundle:nibBundleornil];

if (self) {

locationManager = [[CLLocationManager alloc] init];

// stel positie nauwkeurigheid in

[ locationManager setDesiredAccuracy:kCLLocationAccuracyBest];

// zeg loc mgr te starten met zoeken naar locatie

locationManager startUpdatingLocation];

}

return self;

}

De locatiemanager heeft geen property waarin de actuele locatie wordt opgeslagen. Deze wordt op een andere manier, nl dmv **delegation** verstrekt. De locatiemanager stuurt het message locationManager:didUpdateLocations naar zijn delegate. De delegate is een property van de locationmanager die kan worden ingesteld op het object dat de locatie updates moet ontvangen. In dit voorbeeld is dit WhereAmIViewController. Voeg hiertoe in initWithNibName:bundle de volgende regel toe, juist na de declaratie van locationManager:

[locationManager setDelegate:self];

Zodra de locationmanager een nieuwe locatie beschikbaar heeft, stuurt die de locatie door naar zijn delegate dmv het message **locationManager:didUpdateLocations.** NB Voor iOS6 was dit het message locationManager:didUpdateToLocation:fromLocation.

De locationmanager stuurt via dit message een array van locaties door, dit kunnen meerdere locaties zijn sinds de laatste keer dat locaties zijn doorgegeven. Wij hebben alleen de laatste nodig. De code om de locatie binnen te halen en in een lokale variabele op te slaan wordt dan als volgt:

- (void)locationManager: (CLLocationManager \*)manager

didUpdateLocations: (NSArray \*)locations

{

CLLocation \*currentLocation = [locations objectAtIndex:[locations count]-1];

}

Het kan ook voorkomen dat de locationmanager er niet in slaagt om een locatie te vinden. In dat geval geeft de locationmanager dit door via het message locationManager:didFailWithError. Deze methode moet in WhereAmIViewController geimplementeerd worden om de fout incl reden af te vangen:

- (void)locationManager:(CLLocationManager \*)manager

didFailWithError: (NSError \*)error

{

NSLog(@”Locatie niet gevonden, fout %@”, error);

}

Wordt het programma op de iOS simulator uitgevoerd dan kan in XCode een locatie opgegeven worden onderaan het editor gebied.

**Delegation**

Delegation is een object georienteerde interpretatie van **callbacks**. Een callback is een functie die wordt voorzien voor een bepaald event (of meerdere events). Vindt het event plaats dan wordt de functie aangeroepen. Nu zijn er objecten (zoals locationManager) die meerdere callback functies nodig hebben, voor verschillende events. Er is echter geen ingebouwde manier om informatie te delen tussen verschillende callback functies. Dmv delegation wordt dit opgelost: we voorzien 1 delegate die alle events voor een bepaald object afhandelt. In ons voorbeeld handelt WhereAmIViewController bijv de events ‘didUpdateLocations’ en ‘didFailWitherror’ af. Andere voorbeelden van events zijn ‘single tap’, ‘double tap’, ‘long press’. Al deze events kunnen naar de delegate gestuurd worden voor afwikkeling.

**Protocols**

Elk object dat een delegate heeft, heeft een protocol dat vermeldt welke messages het object naar zijn delegate kan sturen. De messages corresponderen met events. De delegate implementeert de methoden (= messages) uit het protocol, waarmee de events worden afgehandeld. Events die als optioneel in het protocol zijn vastgelegd hoeft de delegate niet te verwerken (= implementeren).

Een protocol is een lijst van methoden, het is geen class. Een voorbeeld van een protocol declaratie (deze is niet correct en zeker niet volledig):

@protocol CLLocationManagerDelegate <NSObject>

// onderstaande methoden in dit deel van de declaratie zijn verplicht, voor de hier

// genoemde methode is dat in werkelijkheid niet het geval!!

- (void)locationManager:(CLLocationManager \*)manager

didFailWithError: (NSError \*)error;

@optional

// onderstaande methoden zijn optioneel, hoeft de delegate niet te implementeren

- (void)locationManager:(CLLocationManager \*)manager

didUpdateHeading: (CLHeading \*)newHeading;

@end

Om controle of een delegate messages kan verwerken mogelijk te maken, geeft een class aan dat het zich conformeert aan een protocol, zoals:

@interface WhereAmIViewController : UIViewController <CLLocationManagerDelegate>

Hier wordt vastgelegd dat class WhereAmIViewController als superclass UIViewController heeft en zich conformeert aan het protocol CLLocationManagerDelegate.

Een object dat messages naar zijn delegate stuurt zal dit zonder meer doen voor methoden die verplicht zijn. Heeft de delegate de betreffende methode niet geimplementeerd dan zal een ‘unrecognized selector exception’ optreden en de applicatie zal crashen. Overigens zal de compiler hier al op controleren.

Voor optionele methoden zal het object eerst checken of zijn delegate het bewuste message wel kan verwerken. Hiervoor wordt het **respondsToSelector** message gebruikt. Elk object implementeert deze methode. Een voorbeeld hiervan (NB deze methode maakt geen onderdeel van het Core Location framework):

- (void)finishedFindingLocation:(NSArray \*)locations

{

// Mbv @selector kan een selector (=message) in een waarde omgezet worden

SEL updateMethod = @selector(locationManager:didUpdateLocations);

//locationManager:didUpdateLocations is een optionele methode, daarom

// eerst checken of de delegate deze wel verwerkt

if ([[self delegate] respondsToSelector:updateMethod]) {

// als de methode geimplementeerd is kan het msg verstuurd worden

[[self delegate] locationManager:self didUpdateLocations:locations];

}

}

**Delegation, controllers en memory management**

Vanuit MVC perspectief is WhereAmIViewController een controller object. In het algemeen zijn delegates controller objecten. Een controller object is in het algemeen eigenaar van objecten waarvan ze de delegate zijn. In ons voorbeeld is WhereAmIViewController dus eigenaar van de locationmanager.

Omdat beide objecten naar elkaar verwijzen kan hier een potentiele **retain cycle** onstaan, als er in beide gevallen sprake is van een **strong** reference. Om dit te voorkomen is de delegate property van CLLocationManager niet strong, maar unsafe unretained. Het zou weak moeten zijn, maar vanuit backwards compatibility voor oudere iOS versies is dit niet gedaan.

Het gevolg hiervan is dat de delegate niet automatisch op nil wordt gezet, wanneer het object waarnaar het verwijst (in ons voorbeeld WhereAmIViewController) wordt verwijderd. Bij een weak reference gebeurt dat wel. Dit heeft tot gevolg dat we de **dealloc** methode in WhereAmIViewController moeten overriden:

- (void)dealloc

{

// Vertel de locationmanager om geen berichten meer naar ons te sturen

[locationManager setDelegate:nil];

}

In ons voorbeeld zal dealloc nooit worden aangeroepen omdat de WhereAmIViewController van begin tot eind nodig is. Er zijn echter situaties, bijv een applicatie met meerdere controllers, waarin controllers wel tijdens programma executie worden verwijderd. En als zo’n controller een delegate is van een object moet de delegate manueel op nil gezet worden.

**Hoofdstuk 5: MapKit en Tekst input**

In hoofdstuk 4 hebben we een iOS applicatie WhereAmI gebouwd waarmee de huidige positie van het device opgevraagd kan worden en in de console kan worden weergegeven. In hoofdstuk 5 wordt WhereAmI voorzien van een iOS user interface. De huidige (of in XCode geselecteerde) positie wordt op een landkaart getoond. Daaraan toegevoegd krijgt de gebruiker de mogelijkheid deze locatie te taggen en er een eigen omschrijving aan toe te voegen dmv een MapKit annotatie, die als een rode pin op de kaart verschijnt.

Aan de hand van het object-view diagram (p104) wordt de relatie tussen de verschillende objecten uitgelegd.

De **View** objecten:

* Een **MKMapView** instantie geeft de kaart weer en de labels van de opgenomen locaties. Het heeft WhereAmIViewController als delegate en stuurt het **mapView:didAddAnnotationViews** message naar WhereAmIViewController wanneer een view wordt toegevoegd (meer over views hieronder);
* Diverse instanties van **MKAnnotationView** verschijnen als icons op de MKMapView instantie;
* Een **UITextField** instantie maakt het mogelijk om tekst in te voeren waarmee de huidige locatie gelabeld wordt (= annotatie). Deze heeft WhereAmIViewController als delegate en stuurt het message **textFieldShouldReturn** wanneer de gebruiker klaar is met de tekstinvoer (door op Done of Return te drukken);
* Een **UIActivityIndicatorView** instantie die laat zien dat het device nog werkt door een draaiende spinner te tonen.

Het **Controller** object is WhereAmIViewController. Het is de delegate van het MKMapView, het UITextField en het CLLocationManager object. WhereAmIViewController stuurt de genoemde view objecten (aanpassen scherm) en de model objecten aan.

De **Model** objecten:

* **CLLocationManager**, al uitvoerig behandeld. Stuurt het message **locationManager:didUpdateLocations** naar WhereAmIViewController als een nieuwe locatie is gevonden.
* **BNRMapPoint**, dit is een custom made class die we gebruiken voor de annotaties.

**MapKit framework**

MapKit is nodig om kaarten en locaties te tonen. Het meeste werk wordt gedaan door de **MKMapView** class, o.a. tonen van de kaart, het bijhouden van touches en de weergave van annotaties.

**IBOutlet**

Dit is ahw een compiler directive waarmee wordt aangegeven dat een object in een XIB file wordt gecreeerd. Dit doen we door in de XCode project navigator de WhereAmIViewController.xib file te openen. In WhereAmIViewController.h voegen we de volgende instantie variabelen toe (vetgedrukt is nieuw):

@interface WhereAmIViewController : UIViewController

<CLLocationManagerDelegate>

{

CLLocationManager \*locationManager;

**IBOutlet MKMapView \*worldView;**

**IBOutlet UIActivityIndicatorView \*activityIndicator;**

**IBOutlet UITextField \*LocationTitleField**;

}

@end

Het boek beschrijft nu van p105 – p108 hoe de view objecten moeten worden toegevoegd en hoe de relaties worden gelegd met de File’s Owner, d.i. hier de WhereAmIViewController. Dit gebeurt in de ‘XIB’ user interface. De richting waarin de relaties worden gelegd komt overeen met de richting van de messages. De view objecten sturen messages naar hun delegate, WhereAmIViewController:

* worldview stuurt het bericht **mapView:didAddAnnotationViews**
* LocationTitleField stuurt het bericht **textFieldShouldReturn**
* locationManager stuurt het bericht **locationManager:didUpdateLocations**

WhereAmIViewController stuurt op zijn beurt diverse berichten naar deze objecten en de activityIndicator (start en stoppen). Zie hiervoor het boek.

**H7 View Controllers en H6 Views**

Een view controller – klasse **UIViewController** – controleert een enkel applicatie scherm. Per scherm is het het handigst één view controller te gebruiken. De view controller laadt de view hiërarchie.

NB Bovenaan de view hiërarchie staat het applicatie window, d.i. het beeldscherm.

Bij opstart van de applicatie in **application:didFinishLaunchingWithOptions** worden alle gebruikte view controllers gecreëerd en geïnitialiseerd en wordt één ervan ingesteld als dé view controller. Dit kan ook een zg tabbar controller zijn, die meerdere view controllers bevat (een NSArray van view controllers), waartussen onderaan het scherm geschakeld kan worden dmv tab toetsen. Standaard wordt de view controller op index 0 geladen. Voorbeeld van initialisatiecode:

MyViewControllerClass1 \*mvc1 = [[MyViewControllerClass1 alloc] init];

MyViewControllerClass2 \*mvc2 = [[MyViewControllerClass2 alloc] init];

MyViewControllerClass3 \*mvc3 = [[MyViewControllerClass3 alloc] init];

NSArray \*mvcArr = [NSArray arrayWithObjects: mvc1, mvc2, mvc3, nil];

UITabBarController \*tbc = [[UITabBarController alloc] init];

[tbc setviewControllers:mvcArr];

[[self window] setRootViewController:tbc];

Een view controller kan via de Interface Builder of in code worden geconfigureerd. Vuistregel is om een view controller met een enkele view (zonder subviews zoals andere views, buttons, tekst fields) in code te initialiseren en anders mbv de Interface Builder (XIB).

**Configuratie van de view controller**

Een view controller die in code wordt geconfigureerd creëert zijn view hiërarchie door de **loadView** methode te overriden:

- (void) loadView{

// Creeer een frame ter grootte van het scherm van het device waar de

// applicatie op draait

CGRect frame = [[UIScreen mainScreen] bounds];

MyView \*mv = [MyView alloc] initWithFrame:frame];

// Stel MV in als dé view van deze view controller

[self setView:mv];

}

**NB1** Als alternatief kan dit in de methode **viewDidLoad** gebeuren. Deze wordt aangeroepen na loadView, als de view daadwerkelijk is geladen.

**NB2** Voor een view controller die in de XIB wordt geconfigureerd hoeft dit niet. Wanneer de view van een view controller moet worden geladen – d.i. als bijv het scherm in de tabbar is geselecteerd – wordt ‘automatisch’ de loadView methode aangeroepen. De standaard implementatie hiervan zal de XIB file laden met de configuratie van de view controller. Wordt de view controller in code geconfigureerd dan is er geen XIB file, dit is de reden dat bij het overriden van de **loadView** methode er geen aanroep van super:loadView plaatsvindt, die zou nl. proberen de XIB file te laden (zie direct hieronder).

De designated initializer van een view controller is de methode **initWithNibName:Bundle**. Deze methode wordt aangeroepen als de view controller wordt geïnitialiseerd (vanuit de init methode). Deze methode krijgt 2 parameters mee waarmee de naam van de XIB file van de view controller wordt aangeduid binnen de applicatie bundel. Hieronder een voorbeeld override implementatie met optioneel de initialisatie van de bijbehorende tabbar item als gebruik wordt gemaakt van een tabbar controller:

- (id)initWithNibName:(NSString \*)nibName bundle:(NSBundle \*)bundle  
{  
    // Call the superclass's designated initializer  
    self = [super initWithNibName:nil  
                           bundle:nil];  
    if (self) {  
        // Get the tab bar item  
        UITabBarItem \*tbi = [self tabBarItem];  
  
        // Give it a label  
        [tbi setTitle:@"Time"];  
    }  
  
    return self;  
}

Parameter **nibname** geeft de naam van de XIB file. Is deze nil dan wordt ervan uitgegaan dat deze de naam heeft van de klasse (bijv MyViewControllerClass1). Parameter **bundle** beschrijft de bundel waarbinnen de XIB file moet worden gevonden, is deze nil dan betreft het de applicatie bundel van de onderhavige applicatie.

**Views**

De designated initialiser van een view is methode **initWithFrame**. Deze heeft als parameter een frame van type CGRect dat de afmetingen van het frame beschrijft. Dit frame geeft de positie van de view tov zijn superview (dus relatieve coördinaten). Voor dé view van een view controller is de superview het beeldscherm van het device, zie hierboven.

Een view hiërarchie kan opgebouwd worden door subviews aan een view toe te voegen:

[myView addSubView:mySubView];

of direct aan het applicatie window:

[[self window] addSubView:mySubView];

NB Dit is alleen van toepassing voor views die in programmacode worden gecreëerd en geconfigureerd. Voor views die in XIB worden aangemaakt is dit niet van toepassing c.q. niet nodig.

De methode **drawRect** is dé methode om een view naar wens aan te passen. Deze methode doet standaard niets, maar door deze te overriden kunnen allerlei view attributen (kleur, font, etc) aangepast worden. drawRect wordt aangeroepen wanneer ‘het nodig’ is de view te tekenen. Dit gebeurt bij initialisatie van de view of wanneer de **setNeedsDisplay** methode wordt aangeroepen. De view kan setneedsDisplay zelf overigens ook aanroepen:

[self setNeedsDisplay];

Dit is bijv. nodig wanneer door bijv. een schidbeweging de kleur wordt veranderd. Het veranderen van de kleur doet verder niets, pas na aanroep van setNeedsDisplay wordt de kleur daadwerkelijk aangepast als de view opnieuw wordt getekend dmv. drawRect.

Een scrollview wordt gebruikt om door views te scrollen die groter zijn dan het beeldscherm. De scrollview zelf wordt als view aan het applicatie window gehangen. De view waarover gescrolled wordt, wordt aan de scrollview gehangen:

CGRect screenRect = [[self window] bounds;

UIScrollView \*myScrollView = [[UIScrollView alloc] initWithFrame:screenRect];

[[self window] addview:myScrollView];

CGRect bigRect = screenRect;

bigRect.size.width \*= 2; // bigRect is 2x zo breed als het beeldscherm

bigRect.size.height \*=2; // bigRect is 2x zo hoog als het beeldscherm

UIView \*myBigView = [[UIView alloc] initWithFrame:bigRect;

[myScrollView addView:myBigView];

// Vertel de scrollview hoe groot zijn virtuele wereld is

[scrollView setContentSize:bigRect.size];

Het is standaard mogelijk om een deel van een scherm te scrollen. Wil je alleen hele pagina’s scrollen, dan moet paging enabled worden:

[scrollView setPagingEnabled:YES];

Een scrollview kan in- en uitzoomen op zijn content. Dit kan vanuit technisch oogpunt maar op één view. De scrollview moet hiervoor de min en max zoomlevels weten en de view waarop wordt in/uitgezoomed:

[scrollView setMinimumZoomScale:1.0];

[scrollView setMaximumZoomScale:5.0];

[scrollView setDelegate:self];

De delegate is in het boek de application delegate van Hypnosister, d.i. de delegate van de applicatie zelf.

Om het inzoomen mogelijk te maken moet de methode viewForZoomingInScrollView in de application delegate geïmplementeerd worden en moet de application delegate voldoen aan het UIScrollViewDelegate protocol. Zie het boek voor de code.