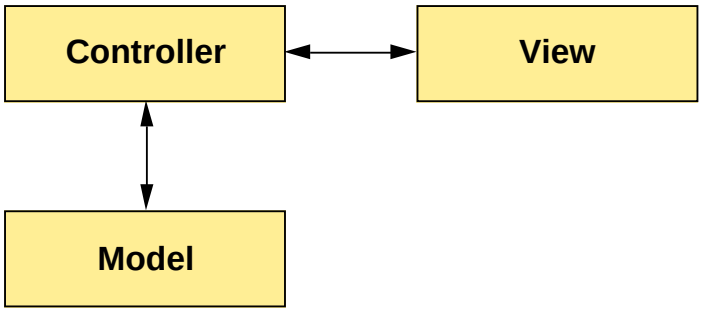
4 MVC-arkitekturen (provavsnitt)

MVC-arkitekturen

I MVC-arkitekturen delas ansvaret inom en applikation upp i tre delar:



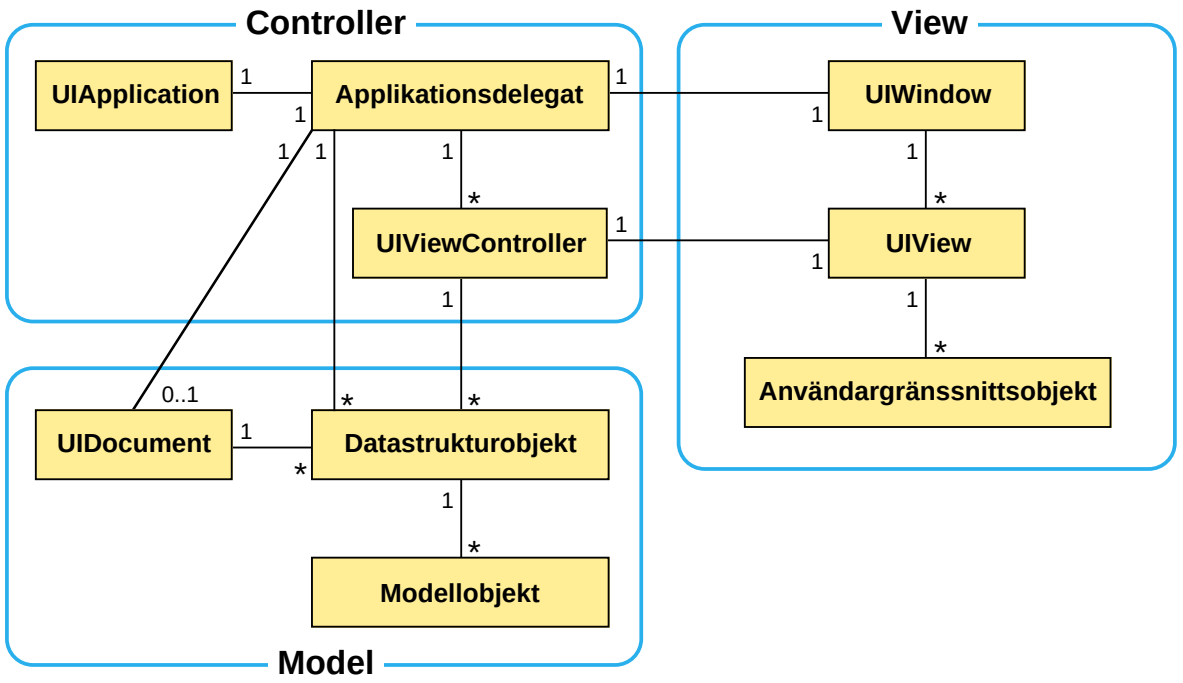
Komponent	*Beskrivning*
Model	Objekt som innehåller applikationens verksamhetsbärande data
View	Objekt som användaren ser och interagerar med inom användargränssnittet
Controller	Objekt som innehåller programlogiken som hanterar användarens kommandon

Inom objektorienterad programmering har det under årens lopp utvecklats arkitekturer som lämpar sig för program med grafiska användargränssnitt. En av de äldsta och mest populära är **MVC**, där applikationen byggs av tre kategorier av komponenter. Det finns flera varianter och förenklingar av MVC-arkitekturen, t ex *modell/vy-separation* som reducerar antalet kategorier till två.

MVC, som är en förkortning av de tre beståndsdelarna **Model**, **View** och **Controller**, är en arkitektur för objektorienterade system som separerar systemet i tre delar:

- De data som hanteras inom en applikation representeras av *modellobjekt* (*Model* i MVC-arkitekturen). Med data menar vi här de verksamhetsbärande data som applikationen handlar om, och inte data som endast krävs för att applikationen ska ha ett användargränssnitt eller följa en viss design. Modellobjekt är helt separerade från applikationens användargränssnitt (och påverkas inte ens av om det finns ett).
- Det användargränssnitt en användare ser i en applikation byggs upp av användargränssnittselement, som kan vara alltifrån fönster och dialoger till enskilda inmatningsfält, kryssrutor och tryckknappar. Dessa användargränssnittselement representeras av *vyobjekt* (*View* i MVC-arkitekturen). Vyobjekt representerar tillståndet och utseendet av användargränssnittet, men ansvarar inte för de verksamhets-bärande data som presenteras eller kan påverkas via dem.
- Den verksamhetsnära programlogik som beskriver hur applikationen ska bete sig i olika situationer och reagera på användarens påverkan representeras av *kontrollobjekt* (*Controller* i MVC-arkitekturen). Kontrollobjekt bildar länken mellan applikationens modell- och vyobjekt: de ser till att data lagras i eller hämtas från modellobjekten, ser till att data presenteras i och att ändrade data hämtas från vyobjekten, reagerar på händelser när användaren manipulerar komponenter i användargränssnittet och synkroniserar uppdateringar mellan olika delar av applikationen.

MVC-arkitekturen i Cocoa Touch



Cocoa Touch-biblioteket är en de renaste tillämpningarna av den ursprungliga MVC-arkitekturen:

- Kontrollobjekten i Cocoa Touch omfattar två nivåer. Hela applikationen representeras av ett objekt av klassen `UIApplication`, som i sin tur delegerar en del av ansvaret till en *applikationsdelegat*, ett objekt av en egenutvecklad klass som implementerar gränssnittet `UIApplicationDelegate`. Applikationsdelegaten hanterar i sin tur ett antal *vykontrollobjekt*. Vykontrollobjekten är av klassen `UIViewController` eller någon av dess underklasser.
- Applikationens fönster representeras av ett objekt av klassen `UIWindow`. I fönstret presenteras ett antal vyer, som var och en täcker en del av eller hela fönstrets yta och representeras av ett *vyobjekt*. Varje vyobjekt hanteras av ett vykontrollobjekt, som innehåller den programlogik som är knuten till vyn. En vy består i sin tur av undervyer och användargränssnittselement, som alla representeras av objekt av bibliotecksklasser i **UIKit** (eller egenutvecklade underklasser till dem).
- De data som applikationen hanterar representeras av *modellobjekt* av egenutvecklade klasser. Det kan vara applikationsdelegaten eller vykontrollobjekten som hanterar modellobjekten, och det är vanligt att de organiseras i datastrukturer. I dokumentorienterade applikationer där olika uppsättningar av data ska lagras i separata filer kan *dokumentobjekt* av underklasser till klassen `UIDocument` underlätta hanteringen av modellobjekten.

De olika komponenterna presenteras ytterligare på nästa sida.

Huvudkomponenter i en iOS-applikation

Komponent	*Beskrivning*
Applikationsobjekt	Ett gemensamt `UIApplication`-objekt för hela applikationen Hanterar applikationsbreda beteenden, t ex händelseslingan
Applikationsdelegat	Implementerar gränssnittet `UIApplicationDelegate` Hanterar initiering av applikationen och växling till bakgrunden
Vykontrollobjekt	Objekt av klassen `UIViewController` eller underklasser Varje vykontrollobjekt hanterar ett vyobjekt och dess undervyer Hanterar programlogik och interaktion med användaren
Fönsterobjekt	Normalt ett `UIWindow`-objekt per applikation Hanterar applikationens fönster och de vyer som visas i det
Vyobjekt	Objekt av klassen `UIView` eller underklasser Varje vyobjekt presenterar en rektangulär yta inom ett fönster Innehåller undervyer och användargränssnittskomponenter
Modellobjekt	Objekt av egenutvecklade klasser Innehåller applikationens verksamhetsbärande data Organiseras ofta i datastrukturer
Dokumentobjekt	Objekt av klassen `UIDocument` eller underklasser Organiserar modellobjekt som ska lagras i dokument

Översikten visar huvudkomponenterna i en iOS-applikation:

- Ett ensamt *applikationsobjekt* av klassen `UIApplication` hanterar beteenden som är gemensamma för hela applikationen, t ex den händelseslinga som sköter den centrala distributionen av händelser inom applikationen. Normalt skapas inte underklasser till `UIApplication`.
- *Applikationsdelegaten* är ett ensamt objekt av en egenutvecklad klass som implementerar gränssnittet `UIApplicationDelegate`. Detta objekt hanterar framför allt tillståndsförändringar inom applikationen, t ex initiering i samband med start och växling till och från bakgrunden.
- *Vykontrollobjekt* är objekt av klassen `UIViewController` eller någon av dess underklasser. Det kan finnas många vykontrollobjekt inom en applikation, både samtidigt och över tiden. Varje vykontrollobjekt hanterar en specifik vy och dess undervyer. Ansvaret omfattar laddning av vyer, installation av vyer i fönster, rotation, programlogik och interaktion med användaren inom vyn.
- Ett *fönsterobjekt* av klassen `UIWindow` representerar applikationens fönster. En applikation har som regel endast ett `UIWindow`-objekt. Det skapas av ramverket och byts aldrig ut under exekveringen. Fönsterobjektet hanterar de vyer som installeras i fönstret. Underklasser förekommer sällan.
- *Vyobjekt* av klassen `UIView` eller någon av dess underklasser representerar vyer och deras beståndsdelar (som räknas som undervyer). Varje vy presenterar innehållet i en rektangulär yta inom fönstret.
- *Modellobjekt* är objekt av egenutvecklade klasser som representerar de verksamhetsbärande data som applikationen ska hantera. Det är vanligt att modellobjekt samlas i datastrukturer.
- *Dokumentobjekt* av underklasser till klassen `UIDocument` används för att hantera modellobjekt i dokumentorienterade applikationer där olika uppsättningar av data ska lagras i separata filer.

Kopplingar mellan källkods- och gränssnittsfiler

Användargränssnitt som vi skapar i **Interface Builder** lagras i *gränssnittsfiler*.

Interface Builder stödjer två former av gränssnittsfiler:

- *Körscheman* (engelsk term *storyboard*) – gränssnitt för en hel applikation
- *Nib-filer* – gränssnitt som hanteras av ett enskilt vykontrollobjekt

Xcode kan skapa kopplingar mellan källkodsfiler och gränssnittsfiler:

Typ av koppling	***Beskrivning***
Action	*Händelsehanterare*: koppling från ett objekt i ett gränssnitt till en metod som reagerar på en händelse i användargränssnittet Markeras i källkoden med `@IBAction` framför `func`
Outlet	Koppling från källkoden till ett enskilt objekt inom ett gränssnitt Markeras i källkoden med `@IBOutlet` framför deklarationen
Outlet Collection	Koppling från källkoden till flera olika objekt inom ett gränssnitt; gruppen är en datastruktur av klassen `NSArray` Markeras i källkoden med `@IBOutlet` framför deklarationen

Användargränssnittsfiler som skapas med hjälp av **Interface Builder** kan vara av två slag: *körscheman* (engelsk term *storyboard*), som beskriver gränssnittet för en hel applikation (t ex `Main.storyboard` i förra kapitlet), eller *nib-filer* (efter den ursprungliga filändelsen `.nib` som föregick den nuvarande `.xib`), som beskriver det gränssnitt som hanteras av ett enskilt vykontrollobjekt.

Ett applikationsprojekt består emellertid också av ett antal källkodsfiler skrivna i något av programspråken **Objective-C** och **Swift**. Det måste naturligtvis finnas någon sorts kopplingar mellan de vanliga källkodsfilerna och gränssnittsfilerna, och verktyget **Xcode** kan hjälpa oss att skapa dem.

Det finns tre typer av kopplingar som kan skapas med hjälp av Xcode:

- En **Action**-koppling beskriver en *händelsehanterare*, dvs en metod i någon av applikationens klasser som anropas när en händelse inträffar i användargränssnittet, t ex när användaren trycker på en knapp på skärmen. Action-kopplingar markeras i källkoden med `@IBAction` framför det reserverade ordet `func`. Xcode kan hjälpa oss att generera såväl de deklarationer som den exekverbara kod som skapar kopplingen. Vi måste givetvis själva fortfarande implementera metoden som reagerar på händelsen, eftersom endast vi vet vad som är rätt reaktion på en händelse i användargränssnittet.
- En **Outlet**-koppling ger oss en koppling från källkoden till ett enskilt objekt i en gränssnittsfil. Kopplingen markeras i källkoden med `@IBOutlet` före deklarationen av den egenskap som refererar till objektet.
- En **Outlet Collection**-koppling är en variant av en Outlet-koppling som ger oss en koppling från källkoden till en grupp av objekt i en gränssnittsfil. Outlet Collection-kopplingar markeras i källkoden med `@IBOutlet` före deklarationen av den egenskap som refererar till objekten, ett objekt av klassen `NSArray`. Outlet Collection-kopplingar är betydligt ovanligare än de andra två formerna av koppling.

En Controller-klass med en Outlet-koppling

ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var centerLabel: UILabel!

    ...
}
```

- Verksamhetsnära programlogik ska finnas i kontrollobjekt
- Klassen `UIViewController` implementerar kontrollobjekt för `UIView`
- Kontrollobjekt behöver känna till vissa vyobjekt via Outlet-kopplingar
- Outlet-kopplingar deklareras som egenskaper i klassen
- Outlet-kopplingar bör vara svaga referenser om de inte uttrycker ägande
- Egenskapsdeklarationer för Outlet-kopplingar ska inledas med `@IBOutlet`

Vi kommer nu att bygga ut vår enkla applikation från föregående kapitel, så att en tryckknapp i användargränssnittet kan påverka texten i Label-objektet.

Att ändra texten i Label-objektet när användaren trycker på en knapp är den här applikationens exempel på "verksamhetsnära" programlogik. Enligt MVC-arkitekturen ska sådan kod finnas i Controller-klassen.

Vyn med Label-objektet som vi skapat i Interface Builder i föregående kapitel är en instans av klassen `UIView` (vyn) som innehåller en instans av klassen `UILabel` (Label-objektet). I **Cocoa Touch** finns klassen `UIViewController` som implementerar kontrollobjekt för vyobjekt av klassen `UIView`. Därför skapar vi vår klass av kontrollobjekt som en underklass till klassen `UIViewController` som deklareras i **UIKit**-ramverket. Detta är exakt vad Xcode genererat i den projektmall vi valt:

```
import UIKit

class ViewController: UIViewController { … }
```

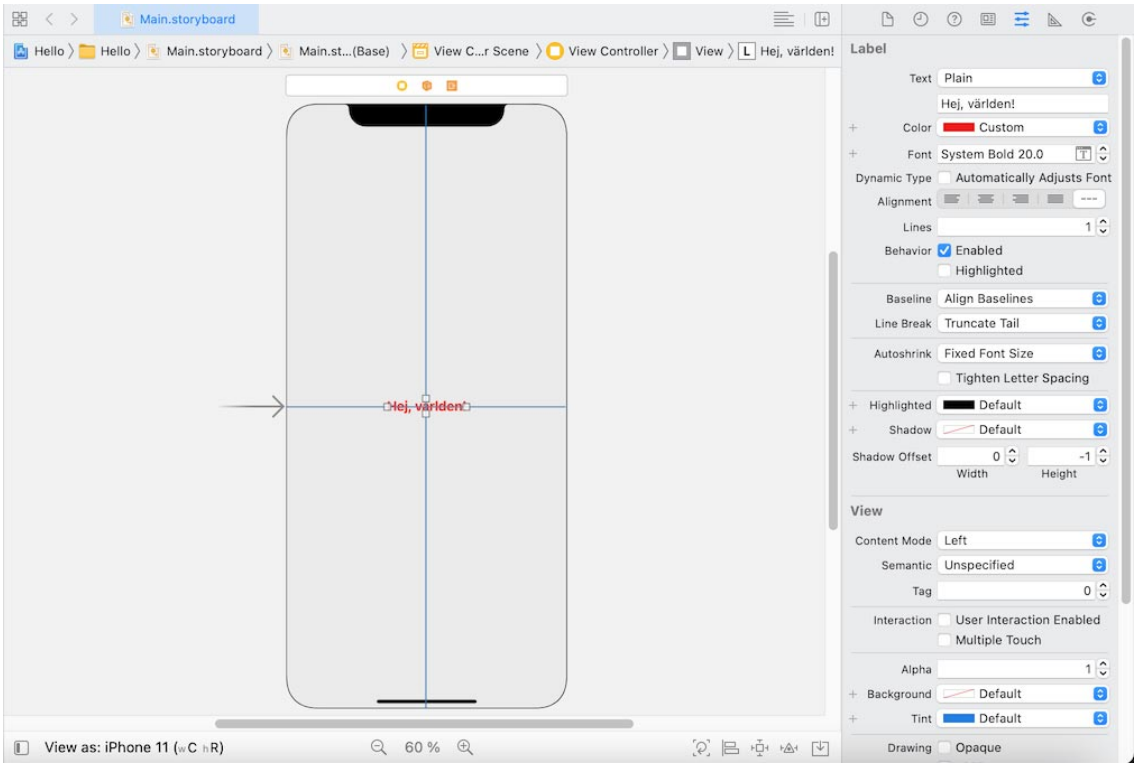
Eftersom vi behöver påverka Label-objektet från kontrollobjektet måste klassen för kontrollobjektet deklarera en egenskap som refererar till det. Egenskapens typ är `UILabel!`, dvs en automatiskt uppackad referens till ett objekt som kan saknas. Den automatiska uppackningen är dock ofarlig i detta fall, eftersom objektet garanterat existerar efter vyn initierats. Eftersom egenskapen *inte* uttrycker ägande (det är vyobjektet som äger Label-objektet), deklarerar vi egenskapen som en *svag referens*:

```
weak var centerLabel: UILabel!
```

Genom att inleda deklarationen med `@IBOutlet` deklareras egenskapen som en Outlet-koppling:

```
@IBOutlet weak var centerLabel: UILabel!
```

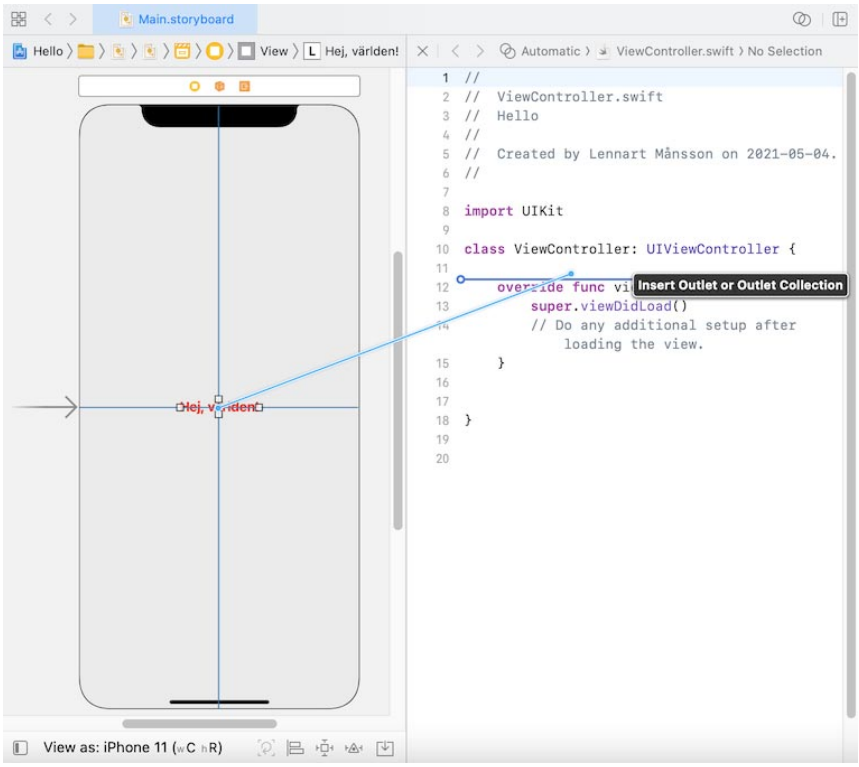
Ett Label-objekt som rymmer mer text



Öppna projektet **Hello** på nytt i Xcode. Klicka på filen `Main.storyboard` i navigatorn. Vyn med det centrerade Label-objektet från föregående kapitel ska nu synas i Interface Builder i editorn.

Label-objektets text ska kunna ändras dynamiskt, så vi måste se till att objektet kan visa mer text är det för tillfället gör. Minska därför storleken på texten till 20 punkter: klicka på Label-objektet, välj femte fliken i inspektorn och ändra typsnittets storlek (egenskapen *Font*) till 20 punkter. Layoutreglerna som sattes i föregående kapitel kommer att se till att Label-objektet förblir centrerat inom vyn.

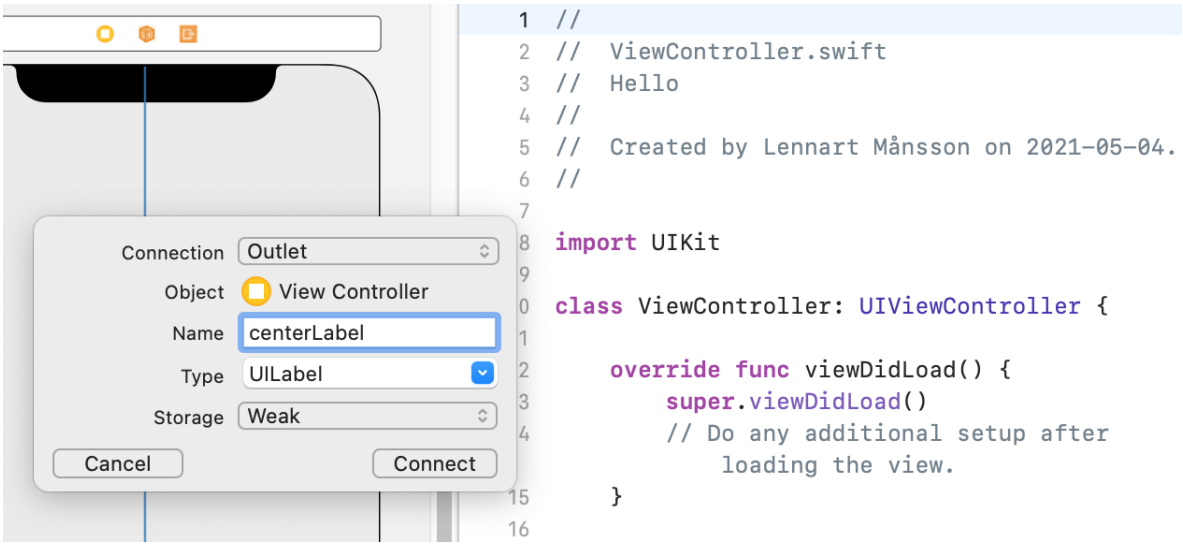
Skapa en Outlet-koppling i Interface Builder



För att ta nästa steg, välj menyvalet **Assistant** i menyn **Editor**. Detta växlar editorn till *assistentläge*, där editorn delas i två paneler. Till vänster visas Interface Builder med samma vy som tidigare. Panelen till höger visar källkodsfilen för klassen `ViewController`. Detta är precis de två vi vill se sida vid sida, och det är ingen tillfällighet: assistentläget försöker räkna ut vad vi troligen vill se i den högra panelen beroende på vad som visas i den vänstra. (Om inte källkodsfilen visas i editorns högra panel kan du enkelt rätta till det genom att välja källkodsfilen i hopplisten längst upp i panelen.)

Håll ned kontrolltangenten och dra från Label-objektet till den högra panelen. En heldragen blå linje följer markören. Dra så att ändpunkten på den blå linjen hamnar mellan `class` och `override` i den högra panelen. En popuptext på mörk botten uppmanar *Insert Outlet or Outlet Collection* när du träffar rätt. Släpp den blå linjen.

Specificera Outlet-koppling



En popupruta ber dig nu specificera detaljer om kopplingen:

- I fältet *Connection*, behåll förvalet **Outlet**. (Alternativet, **Outlet Collection**, gäller en grupp av objekt och är inte aktuellt här.)
- I fältet *Object* är **View Controller** förvalt och kan inte ändras.
- I fältet *Name*, fyll i `centerLabel` som namn på egenskapen du håller på att definiera.
- I fältet *Type* anges egenskapens typ som `UILabel`. Detta är helt korrekt.
- I fältet *Storage* anges att egenskapen är en svag referens (alternativet **Weak**). Detta är också korrekt.

När du specificerat hela Outlet-kopplingen, klicka knappen **Connect**. Lägg märke till att en korrekt egenskapsdeklaration nu lagts till i klassdefinitionen i den högra panelen! Xcode kunde alltså generera koden åt oss som ett resultat enbart av att vi drog Outlet-kopplingen visuellt från Label-objektet till källkodspanelen och fyllde i popuprutan.

Spara dina ändringar.