4 Делегирование и Core Location

В этой главе вы начнете знакомиться с делегированием, представляющим собой повторяющийся шаблон проектирования в среде разработки Cocoa Touch, а также с фреймворком Core Location, обеспечивающим возможности по поиску локации в iOS. Также будет рассмотрено использование отладчика Xcode для поиска и устранения ошибок в коде.

В этой и следующей главе вы создадите приложение Whereami. Это приложение предназначено для поиска географической локации устройства, отображения найденной локации на интерактивной карте и обеспечения возможности отмечания пользователем текущей локации с помощью булавки и надписи.

В меню File (Файл) выберите пункты New ▸ New Project... (Создать ▸ Новый проект). Появится следующее окно, в котором нужно выбрать пункт Application (Приложение) в разделе iOS и тип приложения Single View Application (Приложение с одним представлением). Сконфигурируйте этот проект так, как показано на рис. 4.1.

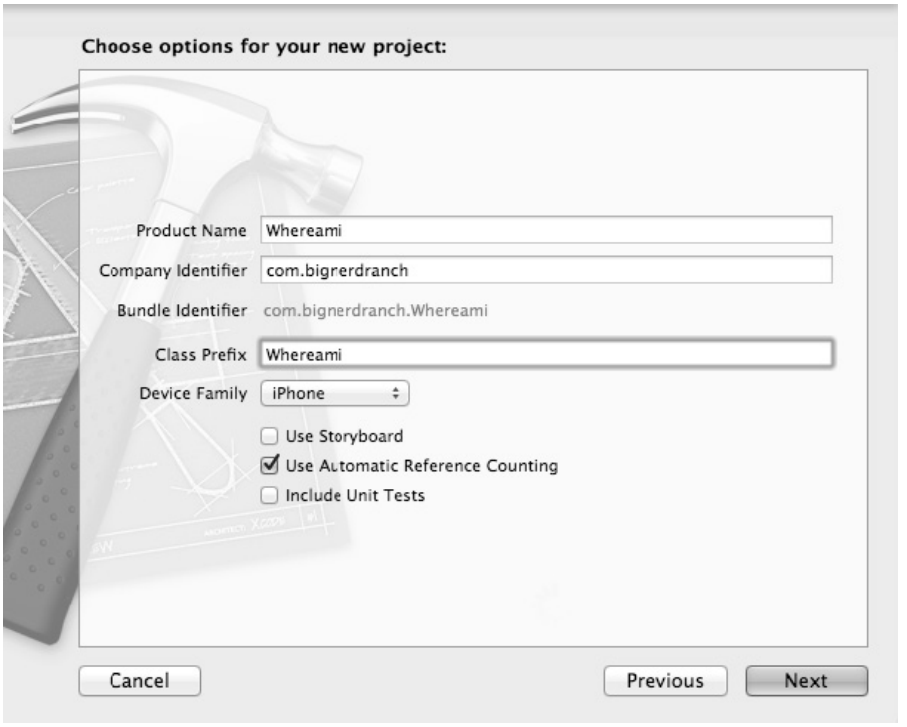

Рис. 4.1. Конфигурирование проекта Whereami

Проекты, цели и фреймворки

В этом разделе мы рассмотрим подробнее, что собой представляет новый проект. *Проект* — это файл, который содержит список ссылок на другие файлы (исходный код, ресурсы, фреймворки и библиотеки), а также ряд настроек, подчиняющиеся правилам для элементов, входящих в проект. Расширение имени файла проекта имеет вид .xcodeproj, например Whereami.xcodeproj.

Любой проект включает хотя бы одну цель. С помощью *цели* на основе файлов проекта создается требуемый продукт. В процессе компиляции и выполнения проекта фактически компилируется и выполняется цель, а не сам проект. Продукт, созданный на основе цели, обычно является приложением, хотя может быть и скомпилированной библиотекой или набором тестовых модулей.

При создании нового проекта и выборе шаблона в Xcode автоматически создается цель. Во время создания проекта Whereami выбирается шаблон iOS application (Приложение iOS), в результате чего в среде Xcode создается цель iOS application, которая получает название Whereami.

В окне навигатора проектов выберите проект Whereami (находится в верхней части окна). Обратите внимание, что проект Whereami и цель Whereami отображаются в области редактора. Выберите цель Whereami, чтобы просмотреть детали и настройки, определяющие ее. Мы не будем рассматривать все настройки в этой главе, но к некоторым из них по мере необходимости будем возвращаться в других главах. Среди настроек, отображающихся в верхней части области редактора, выберите Build Phases (Фазы построения), как показано на рис. 4.2. *Фазы построения цели* — это серия шагов, в результате выполнения которых создается приложение iOS.

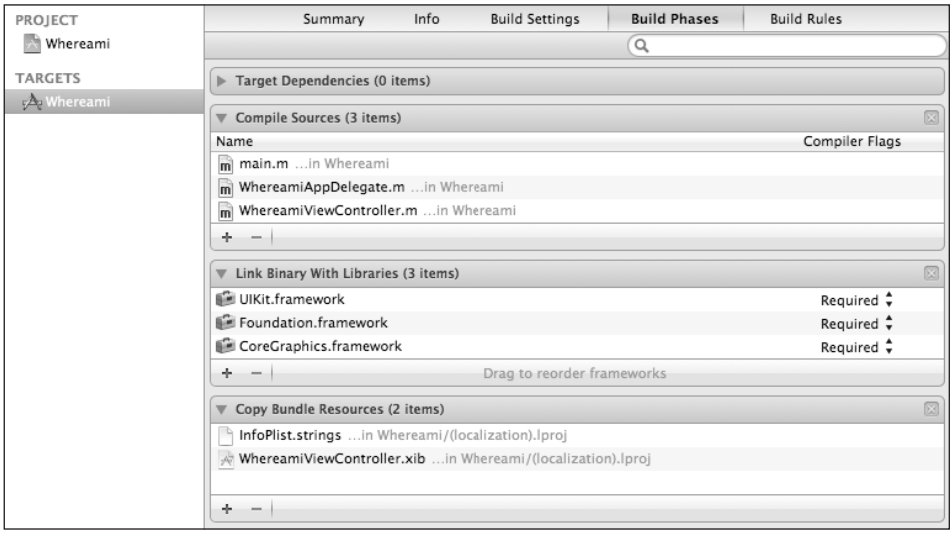

Рис. 4.2. Фазы построения для цели Whereami

При создании iOS-приложений мы проходим через следующие важные фазы построения iOS-приложений: Compile Sources (Компиляция исходного кода), Link Binary With Libraries (Связывание исполняемого файла с библиотеками) и Copy Bundle

Resources (Копирование ресурсов пакета). Эти три фазы будут подробно рассмотрены в конце этой главы. Сейчас же сосредоточимся на фазе Link Binary With Libraries и *фреймворках*.

Фреймворк — это коллекция связанных классов, которые добавляются к цели. Среда Cocoa Touch — это коллекция фреймворков. Одно из преимуществ организации Cocoa Touch в виде фреймворков заключается в том, что программист может добавлять только те фреймворки, которые требуются для данной цели.

Чтобы просмотреть фреймворки, которые уже связаны с вашей целью, щелкните на кнопке закрытия, расположенной возле параметра Link Binary With Libraries. Перед вами появятся следующие три объекта: фреймворк UIKit, включающий классы, которые применяются для создания пользовательского интерфейса iOS; фреймворк Foundation, включающий такие классы, как NSString и NSArray; и фреймворк Core Graphics, подключающий графическую библиотеку, рассмотрение которой начнется в главе 6.

Приложение Whereami также использует фреймворк Core Location, включающий классы, связанные с поиском локации устройства. Чтобы добавить этот фреймворк к цели, щелкните на кнопке плюс (+), находящейся в левом нижнем углу раздела Link Binary With Libraries. Появится лист, отображающий доступные фреймворки (рис. 4.3). В этом списке выберите файл CoreLocation.framework и щелкните на кнопке Add (Добавить).

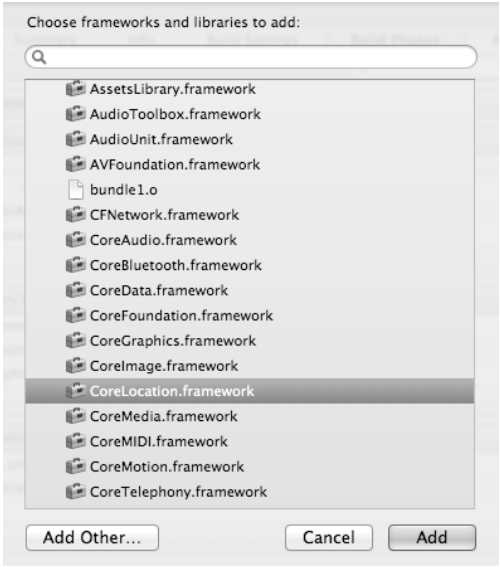

Рис. 4.3. Добавление фреймворка Core Location

Объект CoreLocation.framework появится на фазе Link Binary With Libraries и в окне навигатора проектов. В окне навигатора проектов можно переместить фреймворк в группу Frameworks (Фреймворки), чтобы придать проекту более аккуратный вид, но можно этого и не делать.

Запомните, каким образом фреймворк добавляется в проект, поскольку эту операцию вам придется проделывать довольно часто!

Core Location

Фреймворк Core Location включает классы, с помощью которых приложения определяют географическую локацию устройства. Независимо от типа используемого устройства iOS, созданный вами код Core Location остается неизменным.

Помимо добавления к цели фреймворка Core Location следует импортировать заголовочный файл фреймворка в файлы, которые должны «знать» о классах Core Location. Каждый фреймворк включает заголовочный файл, который импортирует заголовочный файл каждого класса в данный фреймворк. Этому файлу присваивается имя фреймворка, оканчивающееся суффиксом .h.

Откройте файл WhereamiViewController.h и импортируйте заголовочный файл Core Location (оператор импорта находится в верхней части этого файла). Также добавьте переменную экземпляра класса, в которой хранится указатель на экземпляр класса CLLocationManager, — один из классов фреймворка Core Location.

```
#import <UIKit/UIKit.h>
#import <CoreLocation/CoreLocation.h>

@interface WhereamiViewController : UIViewController
{
   CLLocationManager *locationManager;
}
@end
```

Класс CLLocationManager является интерфейсом для оборудования обнаружения местоположения, установленного на устройстве. Экземпляр класса CLLocationManager включает ряд свойств, которые определяет поведение этого класса. Рассмотрим подробнее одно из этих свойств: desiredAccuracy.

Свойство desiredAccuracy задает точность определения локации, выполняемой диспетчером локации. Важность этого свойства обусловлена компромиссом между точностью определения локации, затратами времени и необходимостью экономить заряд аккумуляторной батареи. Точность определения локации также зависит от типа применяемого пользователем устройства, доступности вышек сотовой связи и спутников, а также от наличия точек доступа беспроводной сети.

Откройте файл WhereamiViewController.m и удалите код, находящийся между ключевыми словами @implementation и @end. После этого ваш файл будет выглядеть следующим образом:

```
#import "WhereamiViewController.h"

@implementation WhereamiViewController

@end
```

В файле WhereamiViewController.m переопределите метод initWithNibName:bundle: таким образом, чтобы создать экземпляр класса CLLocationManager, установите значение свойства desiredAccuracy таким образом, чтобы запрашивать наиболее точные данные о локации как можно чаще, а затем запустите CLLocationManager:

```
- (id)initWithNibName:(NSString *)nibNameOrNil
      bundle:(NSBundle *)nibBundleOrNil
{
```

продолжение ⇩

```
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];

    if (self) {
        // Создание объекта диспетчера локации
        locationManager = [[CLLocationManager alloc] init];

        // Необходима как можно большая точность независимо
        // от того, сколько это займет времени или потребует энергии
        [locationManager setDesiredAccuracy:kCLLocationAccuracyBest];

        // Сообщить диспетчеру о немедленном начале поиска локации
        [locationManager startUpdatingLocation];
    }
    return self;
}
```

После отправки сообщения о запуске объекту CLLocationManager он будет продолжать работать во время выполнения приложением других задач, таких как прием данных, вводимых пользователем, или обновление интерфейса.

Получение обновлений от диспетчера CLLocationManager

После компиляции и выполнения приведенного выше кода диспетчер локаций получит информацию о текущей локации, но вы не сможете ее увидеть. Ваше приложение получает сведения о локации у диспетчера локаций. Первое, что приходит в голову, — получить доступ к свойству currentLocation объекта CLLocationManager, в котором хранит сведения о текущей локации. Это логично, но не правильно.

Дело в том, что при нахождении текущей локации диспетчер локаций отсылает сообщение locationManager:didUpdateToLocation:fromLocation: своему *делегату*. А теперь у многих возникает вопрос о том, что же представляет собой *делегат диспетчера*? Сейчас мы постараемся ответить на этот вопрос.

Каждый объект CLLocationManager имеет свойство delegate, и это свойство можно установить таким образом, чтобы оно указывало на объект, который будет получать сообщение «локация найдена». Для приложения Whereami это будет объект WhereamiViewController (рис. 4.4).

Рис. 4.4. Объектная диаграмма приложения Whereami

В файле WhereamiViewController.m обновите метод initWithNibName:bundle:, который применяется для присваивания свойству delegate диспетчера локаций экземпляра класса WhereamiViewController.

```
locationManager = [[CLLocationManager alloc] init];

// Эта строка кода может привести к предупреждению; игнорируйте его
[locationManager setDelegate:self];

[locationManager setDesiredAccuracy:kCLLocationAccuracyBest];
```

Два аргумента метода locationManager:didUpdateToLocation:fromLocation: являются экземплярами класса CLLocation. (Все классы фреймворка Core Location предваряются префиксом CL.) Если объект CLLocationManager имеет достаточное количество данных для определения новой локации, он создает экземпляр класса CLLocation, который содержит значения долготы и широты, представляющие географические координаты устройства (рис. 4.5). Этот экземпляр класса также отображает точность определения локации и (в зависимости от применяемого устройства) высоту над уровнем моря.

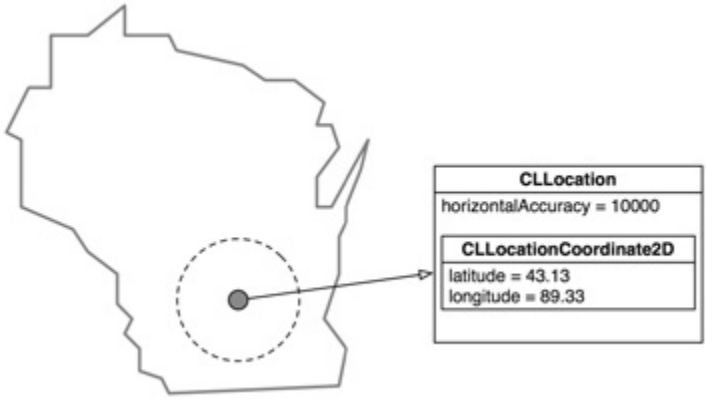

Рис. 4.5. Объект CLLocation

Поскольку объект CLLocationManager отсылает сообщение locationManager:didUpdateToLocation:fromLocation: своему делегату, реализуйте этот метод в файле WhereamiViewController.m. А затем выведите локацию устройства на консоль. (Будьте очень осторожны и не допускайте опечаток либо неправильного использования регистра символов, иначе выполнение метода будет невозможно. Селектор сообщения диспетчера локаций должен точно соответствовать селектору реализованного метода.)

```
- (void)locationManager:(CLLocationManager *)manager
   didUpdateToLocation:(CLLocation *)newLocation
          fromLocation:(CLLocation *)oldLocation
{
   NSLog(@"%@", newLocation);
}
```

Если CLLocationManager не смог найти локацию, сообщите пользователю об этом событии и его причинах. Если локация не найдена, CLLocationManager отсылает другое

сообщение своему делегату — locationManager:didFailWithError:. Реализуйте этот метод в файле WhereamiViewController.m.

```
- (void)locationManager:(CLLocationManager *)manager
      didFailWithError:(NSError *)error
{
   NSLog(@"Could not find location: %@", error);
}
```

Скомпилируйте и запустите приложение на выполнение. Приложение можно скомпилировать как для симулятора, так и для устройства, выбрав соответствующий параметр во всплывающем меню Scheme (Схема), расположенном возле кнопок Run (Выполнить) и Stop (Остановить).

Если приложение Whereami выполняется на симуляторе, вы сможете имитировать его локацию во время выполнения. Если приложение выполняется в среде Xcode, в нижней части области редактора появится панель, на которой находятся несколько значков. Щелкните на значке, находящемся на этой панели, и выберите локацию в раскрывающемся списке (рис. 4.6).

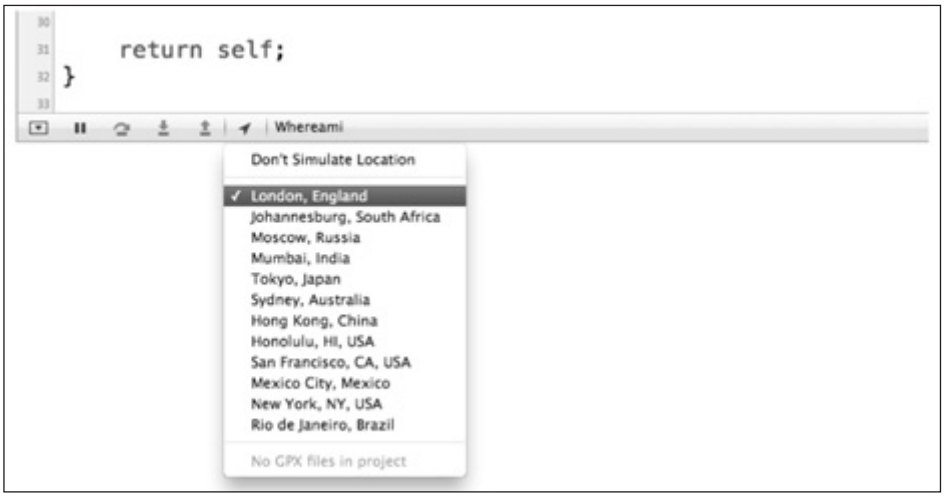

Рис. 4.6. Имитация локации

После предоставления приложению разрешения на использование служб определения локации подождите несколько секунд, требуемых для поиска локации (либо симуляции локации), после чего на консоли отобразится описание объекта локации, которое может выглядеть следующим образом:

```
<+37.33168900, -122.03073100> +/- 100.00m (speed -1.00 mps / course -1.00)
```

Делегирование

При установке значения delegate объекта CLLocationManager и реализации двух методов поиска локации в WhereamiViewController используется шаблон проектирования под названием *делегирование*. Это шаблон является весьма распространенным в среде Cocoa Touch, и многие классы имеют свойство delegate.

Делегирование — это объектно-ориентированный подход к *обратным вызовам*. Обратный вызов — это функция, которая поддерживается в связи с ожидаемым событием и которая вызывается каждый раз, когда происходит событие. Некоторые объекты нуждаются в том, чтобы совершать обратный вызов для нескольких событий. Например, диспетчер локаций ожидает «обратный вызов» в случае, если обнаружена новая локация или произошла ошибка.

Не существует стандартного способа координирования и совместного использования информации двумя (или большим числом) функциями обратного вызова. Для решения этой проблемы используется делегирование. А именно, поддерживается единственный делегат, который получает все сообщения о событиях для конкретного объекта. Объект делегата может хранить, манипулировать, выполнять какие-либо действия либо транслировать связанную информацию, если это требуется.

Давайте сравним делегирование и другой объектно-ориентированный подход к обратным вызовам: *пары «цель-действие»*. Пары «цель-действие» использовались вместе с компонентами UIButtons в приложении Quiz, которое было разработано в главе 1. В паре «цель-действие» имеется объект «цель» (target), которому отправляется сообщение «действие» (action) в случае возникновения определенного события (например, тапа кнопки). Для каждого отдельного события должна создаваться новая пара «цель-действие» (например, для тапа, двойного тапа или длинного нажатия). При использовании делегирования один раз устанавливается делегат, который потом может отсылать сообщения в случае возникновения различных событий. Делегат может реализовывать методы, соответствующие событиям, о которых он хочет «слышать» (рис. 4.7).

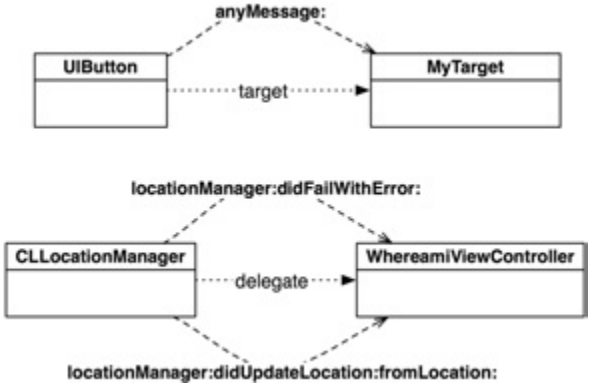

Рис. 4.7. Пары «цель-действие» и делегирование

В парах «цель-действие» в качестве сообщения «действие» может использоваться любое сообщение. Во время делегирования объект может лишь отправлять сообщения делегата из специфического набора, указанного в протоколе.

Протоколы

Для каждого объекта, имеющего делегат, назначается соответствующий протокол, который объявляет сообщения, отсылаемые объектом делегату. Делегат реализует из протокола методы для событий, в которых он заинтересован. Если класс реализует методы из протокола, значит он следует протоколу.

Протокол для делегата CLLocationManager выглядит следующим образом:

```
// Обратите внимание, что было пропущено несколько методов,
// относящихся к реальному определению протокола,
// чтобы уменьшить объем кода
@protocol CLLocationManagerDelegate <NSObject>

@optional

- (void)locationManager:(CLLocationManager *)manager
   didUpdateToLocation:(CLLocation *)newLocation
          fromLocation:(CLLocation *)oldLocation;

- (void)locationManager:(CLLocationManager *)manager
   didUpdateHeading:(CLHeading *)newHeading;

- (BOOL)locationManagerShouldDisplayHeadingCalibration:
          (CLLocationManager *)manager;

- (void)locationManager:(CLLocationManager *)manager
   didEnterRegion:(CLRegion *)region;

- (void)locationManager:(CLLocationManager *)manager
   didFailWithError:(NSError *)error;
@end
```

Этот протокол, подобно всем другим протоколам, объявляется с помощью директивы @protocol, за которой следует имя протокола: CLLocationManagerDelegate. Объект NSObject, заключенный в угловые скобки, ссылается на протокол NSObject и сообщает о том, что CLLocationManagerDelegate включает все методы протокола NSObject. Потом объявляются методы, специфичные для CLLocationManagerDelegate, а сам протокол закрывается директивой @end.

Обратите внимание, что протокол не является классом, — это просто список методов. Вы не сможете создавать экземпляры протокола, иметь переменные экземпляра класса, а сами методы не могут быть реализованы где-либо в протоколе. Реализация протокола остается в каждом классе, который соответствует этому протоколу.

Протоколы, используемые для делегирования, называются *протоколами делегатов*. В соответствии с соглашением о наименовании для протокола делегата используется имя делегирующего класса, к которому добавляется слово Delegate. Имейте в виду, что далеко не все протоколы являются протоколами делегатов, примеры других разновидностей протоколов приведены в следующей главе.

Ранее упомянутые протоколы являются частью iOS SDK, но можно также создавать свои собственные протоколы. Дополнительные сведения по этой теме можно найти в главах 13 и 26.

Методы протокола

В протоколе CLLocationManagerDelegate используется два типа методов: методы, обрабатывающие информацию, связанную с обновлениями, и методы, обрабатывающие запросы на ввод данных. Например, делегат диспетчера локаций реализует метод locationManager:didEnterRegion: в том случае, если хочет «услышать» от диспетчера локаций о том, что устройство находится в определенном регионе. Это и есть информация обновления.

Метод locationManagerShouldDisplayHeadingCalibration: представляет собой запрос на ввод данных. Диспетчер локаций отсылает свой делегат этому сообщению, чтобы запросить, будет ли выполняться калибровка направления движения. Метод возвращает значение Boolean, которое является ответом делегата.

Методы, объявленные в протоколе, могут быть обязательными или необязательными. По умолчанию методы протокола являются обязательными. Если протокол включает необязательные методы, они будут предваряться директивой @optional. Обратите внимание на протокол CLLocationManagerDelegate и вы увидите, что все его методы необязательны. Это справедливо и для протоколов делегатов.

Перед отправкой необязательного сообщения объект сначала запрашивает своего делегата. Если запрос был успешным, отправляется необязательное сообщение путем отсылки другого сообщения, respondsToSelector:. Каждый объект, реализующий этот метод, во время выполнения проверяет, реализовал ли объект данный метод. Можно превратить селектор метода в значение, в результате чего появится возможность передавать его в качестве аргумента с помощью директивы @selector(). Например, объект CLLocationManager может реализовать метод, который выглядит следующим образом:

```
- (void)finishedFindingLocation:(CLLocation *)newLocation
{
   // locationManager:didUpdateToLocation:fromLocation:
   // является дополнительным методом, поэтому проверим его первым.
   SEL updateMethod =
      selector(locationManager:didUpdateToLocation:fromLocation:);

   if ([[self delegate] respondsToSelector:updateMethod]) {
      // Если метод реализован, отправим сообщение.
      [[self delegate] locationManager:self
                  didUpdateToLocation:newLocation
                     fromLocation:oldLocation];
      }
}
```

Если метод, определенный в протоколе, является обязательным, отсылается сообщение без выполнения предварительной проверки. Это означает, что если делегат не реализует этот метод, генерируется исключение нераспознанного селектора, а приложение аварийно завершается.

Чтобы предотвратить появление подобной неприятной ситуации, компилятор будет «настаивать» на том, чтобы класс реализовывал требуемые методы для протокола. Но если компилятор «знает» о результатах проверки реализаций требуемых методов протокола, класс должен явно определить собственное соответствие с протоколом. Все эти действия осуществляются в заголовочном файле класса: протоколы, которым соответствует класс, добавлены в список, разделенный запятыми, который заключен в угловые скобки, в объявлении интерфейса, который следует за суперклассом.

В файле WhereamiViewController.h объявите соответствие класса WhereamiViewController протоколу CLLocationManagerDelegate:

```
@interface WhereamiViewController : UIViewController
   <CLLocationManagerDelegate>
```

Снова скомпилируйте приложение. Теперь, когда объявлено соответствие класса WhereamiViewController протоколу CLLocationManagerDelegate, исчезает предупреждение,

генерируемое строкой кода, в котором был установлен делегат класса locationManager. Если же нужно реализовать дополнительные методы из протокола CLLocationManagerDelegate в классе WhereamiViewController, эти методы будут автоматически завершены в среде Xcode.

Делегирование, контроллеры и управление памятью

С точки зрения шаблона «модель-представление-контроллер» (model-view-controller) объект WhereamiViewController является объектом контроллера. Именно в этом случае делегаты также являются объектами контроллера. Объект контроллера обычно владеет объектами, по отношению к которым является делегатом (рис. 4.8). Например, объект WhereamiViewController владеет объектом CLLocationManager, а WhereamiViewController является делегатом объекта CLLocationManager.

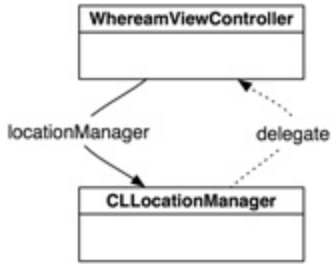

Рис. 4.8. Контроллеры владеют объектами, объекты являются делегатами контроллеров

Как вы помните в главе 3, рассматривалась взаимосвязь, которая может создавать цикл удержания в случае, если оба объекта устанавливают сильные ссылки друг на друга. Чтобы избежать появления подобного цикла, свойство delegate объекта CLLocationManager не является сильной ссылкой. Но в то же время оно не является слабой ссылкой. Для обеспечения обратной совместимости с версиями iOS, несовместимыми с ARC, свойствам delegate присваиваются значения __unsafe_unretained.

Поскольку объект delegate имеет значение *unsafe unretained* (небезопасный неудерживаемый), а не *weak* (слабый), ему автоматически не присваивается значение nil, если объект, на который оно указывает, разрушается. Чтобы выполнить эту операцию, воспользуйтесь методом dealloc объекта delegate.

В файле WhereamiViewController.m переопределите метод dealloc:

```
- (void)dealloc
{
   // Диспетчеру локаций дается указание прекратить отсылать нам сообщения
   [locationManager setDelegate:nil];
}
```

Было бы ошибочным полагать, что реализация метода dealloc для класса WhereamiViewController возможна. Почему? Потому что объект WhereamiViewController никогда не разрушается в приложении — только что реализованный метод dealloc никогда не может быть вызван. Приложение Whereami всегда нуждается в классе WhereamiViewController, поэтому WhereamiViewController всегда имеет владельца.