Глава 6

Организация исходных файлов

До сих пор все рассматриваемые нами проекты состояли из одного исходного файла main.m. Функция main() и все разделы @interface и @implementation наших классов располагались в одном файле. Такая структура пригодна для небольших программ, но не для больших проектов. С ростом программы файлы становятся слишком велики для просмотра или поиска. Возвращаясь к годам учебы — вы же не помещали все написанные вами рефераты в один большой файл? Наверняка, каждая работа располагалась в отдельном файле с именем, позволяющим судить о содержимом файла. Аналогично имеет смысл разделять исходный текст программы на несколько файлов, давая каждому из них свое (желательно — описывающее содержимое файла) имя. Разделение вашей программы на небольшие файлы повышает шансы более быстро найти интересующий вас код и помогает другим быстрее разобраться в вашем проекте. Размещение кода в нескольких файлах облегчает задачу отправки исходного текста интересного класса друзьям: вам будет достаточно отправить только пару файлов, но не весь проект. В этой главе будут рассмотрены идеи и стратегии размещения частей программы в отдельных файлах.

Разделение интерфейса и реализации

Как вы уже знаете, исходный текст классов Objective-C разделяется на две части. Первая часть — интерфейс, предоставляющий вид класса "извне" и содержащий всю необходимую для использования класса информацию. Для того чтобы иметь возможность использовать объекты класса, вызывать его методы, вставлять объекты в другие классы или создавать подклассы, следует предоставить компилятору всю информацию, содержащуюся в разделе @interface.

Вторая часть исходного текста класса — его реализация. Раздел @implementation говорит компилятору Objective-C о том, как заставить класс работать. Здесь содержится код, который реализует методы, объявленные в интерфейсе.

В силу естественного разделения определения класса на интерфейс и реализацию, код класса часто точно так же разбивается на два файла. В первом файле хранятся компоненты интерфейса: директива @interface, открытые определения struct, константы enum, #defines, глобальные переменные extern и т.д. Поскольку Objective-C является наследником C, все это обычно располагается в заголовочном файле, который имеет то же имя, что и имя класса, и расширение .h. Например, у класса Engine заголовочный файл имеет имя Engine.h, а у класса Circle — Circle.h.

Все детали реализации, такие как директива @implementation, определения глобальных переменных, закрытые объявления struct и так далее, располагаются в другом файле — с тем же именем, что и имя класса, и расширением .m. Файлы Engine.m и Circle.m представляют собой файлы реализации соответствующих классов.

ПРИМЕЧАНИЕ

Если вы используете расширение .mm, то тем самым говорите компилятору, что ваш код написан на Objective-C++, который позволяет использовать вместе C++ и Objective-C.

Создание новых файлов в Xcode

При создании нового класса Xcode упрощает вашу жизнь, автоматически создавая для вас .h- и .m-файлы. Выбирая в Xcode пункт меню File⇨New File, вы получите окно наподобие показанного на рис. 6.1, в котором перечислены типы файлов, известные Xcode.

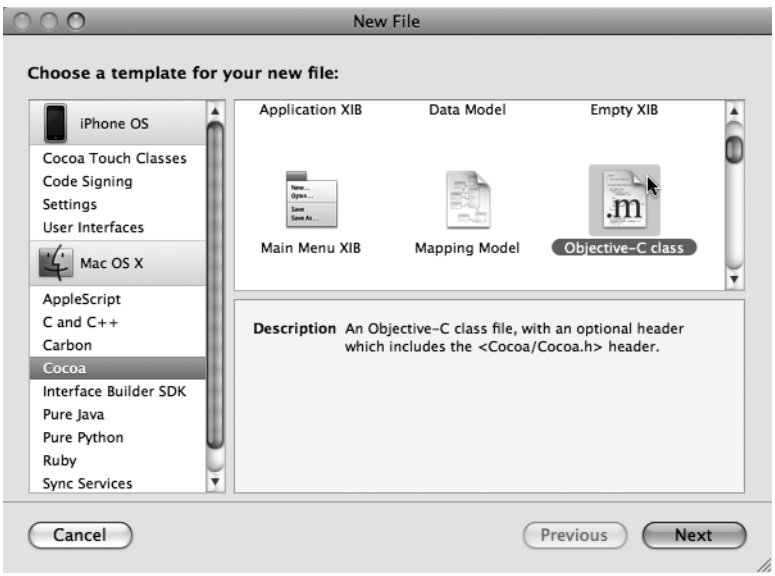

Рис. 6.1. *Создание нового файла в Xcode*

Выберите Objective-C class и щелкните на кнопке Next. Вы получите очередное окно, в котором у вас запросят имя файла (рис. 6.2).

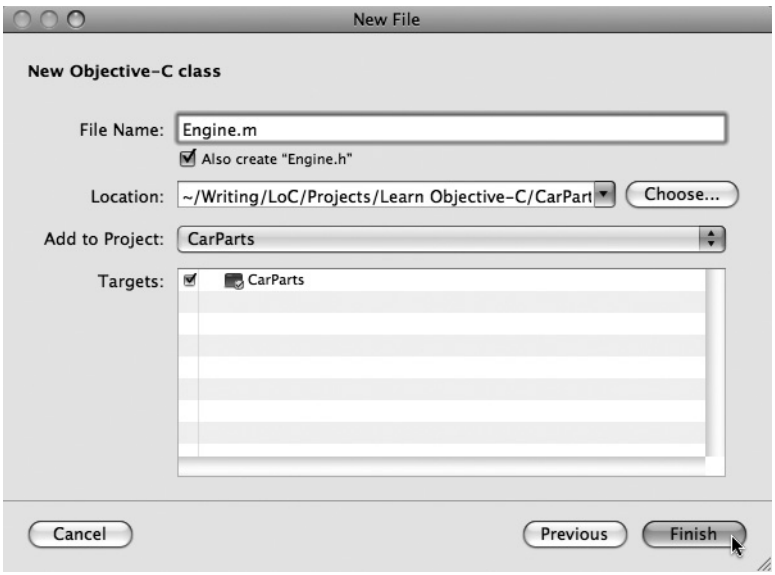

Рис. 6.2. *Именование нового файла*

В этом окне не просто вводится имя файла. Здесь есть переключатель, которым можно воспользоваться, чтобы Xcode создал для вас файл Engine.h. Если у вас открыто несколько проектов, можно воспользоваться выпадающим списком Add to project для выбора проекта, к которому относятся вновь создаваемые файлы. Раздел Targets рассматривать не будем; скажем лишь, что сложные проекты могут иметь несколько конечных целей, каждая из которых имеет собственную конфигурацию исходных файлов и различные правила построения.

После щелчка на кнопке Finish Xcode добавит соответствующие файлы в проект и выведет результат в окне проекта, как показано на рис. 6.3.

Xcode помещает новые файлы в выбранную папку в панели Groups & Files (если перед созданием файлов вы выбрали Source, то файлы будут размещены в этой папке). Эти папки (именуемые Xcode Groups) предоставляют способ организации исходных файлов в вашем проекте. Например, вы можете сделать одну группу для классов вашего пользовательского интерфейса, другую — для классов, работающих с данными, что упростит работу с проектом. При создании групп Xcode в действительности не перемещает никакие файлы и не создает никакие каталоги на вашем жестком диске. Взаимосвязи файлов с группами — сугубо логические; если можно так выразиться, это просто красивые фантазии, поддерживаемые Xcode. Если вы хотите, то можете создать группу, которая будет указывать на определенное место в файловой системе, и Xcode будет помещать вновь создаваемые файлы в указанный вами каталог.

После создания файлов вы можете редактировать их — для этого следует выполнить двойной щелчок на имени интересующего вас файла в списке. Xcode любезно вставляет в файл стандартную заготовку исходного текста, такую как #import <Cocoa/Cocoa.h>, или пустые разделы @interface и @implementation, которые вам предстоит заполнить реальным кодом.

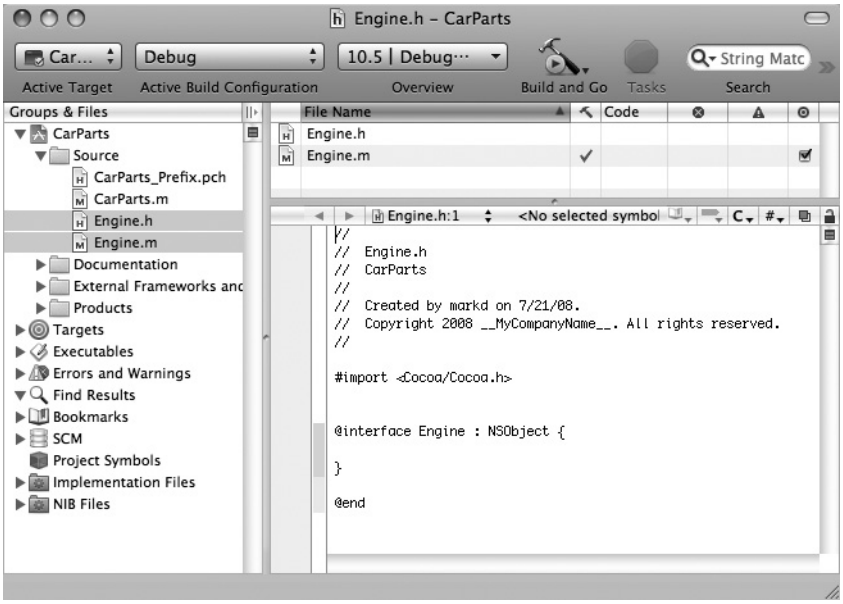

Рис. 6.3. *Новые файлы появляются в окне проекта Xcode*

ПРИМЕЧАНИЕ

До сих пор в данной книге мы вносили в исходный текст программ строку `#import <Foundation/Foundation.h>`**, поскольку использовали только эту часть Cocoa. Но вместо нее можно использовать строку** `#import <Cocoa/Cocoa.h>`**. Такой импорт, среди прочего, подгрузит и заголовочные файлы схемы Foundation.**

Как разбить автомобиль

В программе CarParts–Split (в папке проекта 06.01 CarParts-Split) все классы перемещены в собственные файлы. Каждый класс имеет собственные заголовочный файл (.h) и файл реализации (.m). Давайте рассмотрим, как создается такой проект. Начнем с двух классов, наследующих класс NSObject, — Tire и Engine. Воспользуйтесь командой меню New File и выберите Objective-C Class, после чего введите имя класса — Tire. Сделайте то же самое и для класса Engine. На рис. 6.4 показаны четыре новых файла в списке проекта.

Теперь перенесем раздел @interface класса Tire из файла CarParts-Split.m в файл Tire.h. Этот файл принимает следующий вид.

```
#import <Cocoa/Cocoa.h>

@interface Tire : NSObject
@end // Tire
```

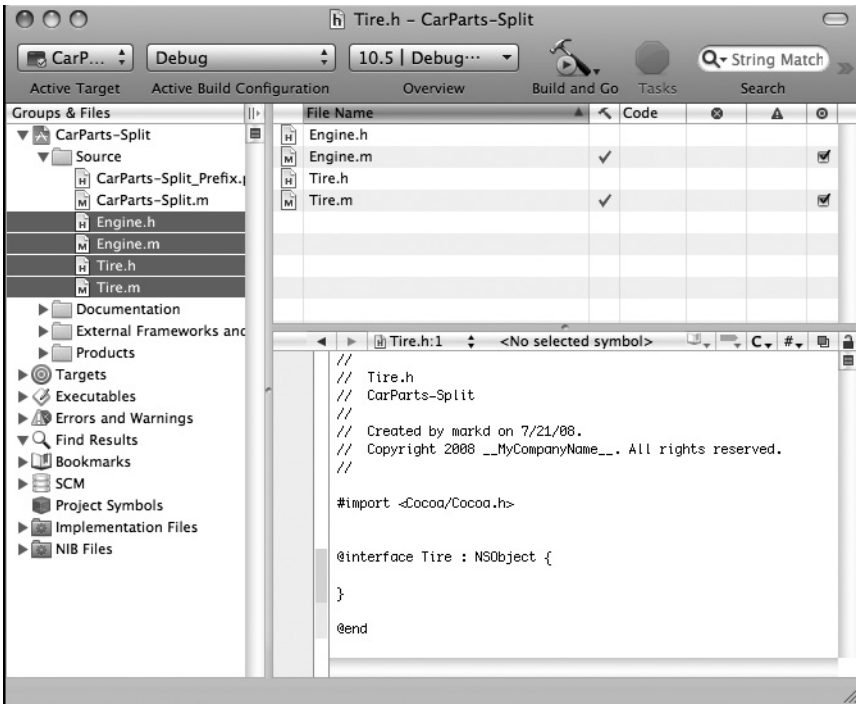

Рис. 6.4. *Добавление в проект Tire и Engine*

Затем из файла CarParts-Split.m в файл Tire.m переносим раздел @implementation класса Tire. Кроме того, в начало файла следует добавить строку #import "Tire.h". Теперь файл Tire.m должен выглядеть следующим образом.

```
#import "Tire.h"

@implementation Tire
– (NSString *) description
{
   return (@"Я – колесо.");
} // description
@end // Tire
```

Первая директива #import представляет определенный интерес. Это не импорт заголовочных файлов Cocoa.h или Foundation.h, который осуществлялся ранее. Здесь импортируется заголовочный файл класса. Это стандартная процедура, и вы должны выполнять ее практически в каждом создаваемом проекте. Компилятору для генерации корректного хода требуется информация о схеме размещения переменных экземпляра в памяти, но автоматической загрузки заголовочных файлов компилятор сам не выполняет. Ему следует явно указать подгружаемые заголовочные файлы при помощи директивы #import. Если при компиляции вы получаете сообщение об ошибке наподобие “Cannot find interface definition for Tire” (не могу найти определение интерфейса для Tire), то это обычно означает, что вы забыли директиву #import с заголовочным файлом класса.

ПРИМЕЧАНИЕ

Обратите внимание на наличие двух различных способов импорта: с применением кавычек и угловых скобок — например, сравните директивы #import <Cocoa/Cocoa.h> **и** #import "Tire.h"**. Версия с угловыми скобками используется для импорта системных заголовочных файлов. Версия с кавычками указывает, что заголовочный файл локален для данного проекта. Если вы видите заголовочный файл в угловых скобках — он предназначен только для чтения в рамках вашего проекта, поскольку им владеет система. Если же имя файла взято в кавычки, то вы (или иной программист проекта) можете вносить в него изменения.**

Теперь выполним те же действия для класса Engine. Перенесем раздел @interface этого класса из файла CarParts–Split.m в Engine.h, который принимает следующий вид.

```
#import <Cocoa/Cocoa.h>

@interface Engine : NSObject
@end // Engine
```

Затем перенесем раздел @implementation в файл Engine.m, который после этой операции выглядит так.

```
#import "Engine.h"

@implementation Engine
– (NSString *) description
{
   return (@"Я – двигатель.  P–p–p–p–p!");
} // description
@end // Engine
```

Если теперь попробовать скомпилировать программу CarParts-Split.m, то вы получите сообщение об ошибке из-за отсутствия объявлений классов Tire и Engine. Исправить эту ошибку очень легко. Достаточно добавить две следующие строки в начало файла CarParts-Split.m, сразу после строки #import <Foundation/Foundation.h>.

```
#import "Tire.h"
#import "Engine.h"
```

ПРИМЕЧАНИЕ

Помните, что директива #import **подобна директиве** #include**, которая обрабатывается препроцессором C. В данном случае препроцессор C, по сути, выполняет добавление содержимого файлов** Tire.h **и** Engine.h **в файл** CarParts-Split.m **перед началом компиляции.**

Вы можете собрать и запустить программу, получив при этом ту же функциональность программы, что и ранее, при использовании классов AllWeatherRadials и Slant6.

```
Я – двигатель slant–6. P–p–p–p–p!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
```

Зависимости между файлами

Зависимость представляет собой взаимоотношение между двумя сущностями. Вопросы зависимостей часто возникают в процессе разработки и написания программ. Зависимости могут иметься между двумя классами: например, Slant6 зависит от Engine из-за отношения наследования. При изменении Engine, таком как добавление новой переменной экземпляра, Slant6 должен быть перекомпилирован, чтобы воспринять это изменение.

Зависимости могут иметься и между двумя и более файлами. CarParts-Split.m зависит от Tire.h и Engine.h. Если какой-то из этих файлов изменится, файл CarParts-Split.m должен быть перекомпилирован, чтобы воспринять эти изменения. Например, Tire.h может содержать константу kDefaultTirePressure, равную 2 атм. Допустим, программист, который работает над программой, решает, что давление по умолчанию следует изменить на 3 атм. и вносит соответствующее изменение в файл Tire.h. Теперь необходимо перекомпилировать файл CarParts-Split.m, чтобы старое значение было заменено новым.

Импорт заголовочного файла устанавливает строгую зависимость между заголовочным файлом и импортирующим его файлом с исходным текстом. Если заголовочный файл изменяется, то все файлы, зависящие от него, требуют перекомпиляции. Представим, что у нас есть сотня .m-файлов, каждый из которых включает некоторый (один и тот же) заголовочный файл — скажем, UserInterfaceConstants.h. Если вы внесете изменение в UserInteraceConstants.h, все сто .m-файлов должны быть перекомпилированы, что может потребовать большого количества времени, какая бы мощная машина не имелась в вашем распоряжении.

Вопрос рекомпиляции может быть еще более неприятным в силу транзитивности зависимости. Например, если Thing1.h импортирует Thing2.h, который в свою очередь импортирует Thing3.h, то любые изменения в Thing3.h приводят к необходимости перекомпиляции файлов, импортирующих Thing1.h. Xcode отслеживает все зависимости файлов за вас.

Минимальная рекомпиляция

А вот хорошая новость: Objective-C предоставляет способ минимизировать перекомпиляцию, вызванную зависимостями. Вопросы зависимостей появляются из-за того, что компилятор Objective-C для успешной работы нуждается в определенной информации. Временами ему требуется полная информация о классе, такая как схема размещения переменных экземпляра и цепочка наследования, а иногда достаточно только имени класса без его полного определения.

Например, при композиции (с которой вы познакомились в предыдущей главе) применяются указатели на объекты, поскольку все объекты Objective-C используют динамически выделяемую память. Компилятору достаточно знать о том, что некоторый элемент является классом, — при этом ему известно, что переменная экземпляра имеет размер указателя (который одинаков во всей программе).

В Objective-C вводится ключевое слово @class, назначение которого — сказать компилятору "это — класс, и я собираюсь обращаться к нему только через указатели". Компилятор может быть спокоен: ему ничего не надо знать об этом классе, кроме того, что это нечто в памяти, доступ к чему осуществляется посредством указателя.

Мы воспользуемся ключевым словом @class при перемещении класса Car в собственный класс. Создадим файлы Car.h и Car.m при помощи Xcode, так же как мы делали это с классами Tire и Engine. Переместим раздел @interface класса Car в файл Car.h, который теперь принимает следующий вид.

```
#import <Cocoa/Cocoa.h>

@interface Car : NSObject
{
  Tire   *tires[4];
  Engine *engine;
}

– (void) setEngine: (Engine *) newEngine;
– (Engine *) engine;
– (void) setTire: (Tire *) tire
      atIndex: (int) index;
– (Tire *) tireAtIndex: (int) index;
– (void) print;
@end // Car
```

Если мы попытаемся воспользоваться этим заголовочным файлом, то получим от компилятора сообщение об ошибке, говорящее о том, что он не понимает, что такое Tire или Engine. Сообщение, скорее всего, будет иметь вид error: parse error before "Tire", что в переводе с компиляторного на русский означает “я этого не понимаю”.

У нас есть два варианта исправления этой ошибки. Первый заключается в импорте Tire.h и Engine.h, которые снабжают компилятор всей имеющейся информацией об указанных классах.

Но есть способ и получше. Если внимательно посмотреть на интерфейс класса Car, то вы увидите, что обращения к классам Tire и Engine осуществляются через указатели, т.е. это как раз тот случай, когда можно применить ключевое слово @class. Вот как выглядит файл Car.h с добавленными строками @class.

```
#import <Cocoa/Cocoa.h>

@class Tire;
@class Engine;

@interface Car : NSObject
{
  Tire   *tires[4];
  Engine *engine;
}

– (void) setEngine: (Engine *) newEngine;
– (Engine *) engine;
– (void) setTire: (Tire *) tire
      atIndex: (int) index;
– (Tire *) tireAtIndex: (int) index;
– (void) print;
@end // Car
```

Здесь достаточно информации для того, чтобы компилятор мог работать с интерфейсом класса Car.

ПРИМЕЧАНИЕ

Ключевое слово @class **является** *опережающей ссылкой* **(forward reference). Это способ сообщить компилятору "доверься мне; ты знаешь, что такой класс есть, а большего тебе пока знать и не надо".**

Ключевое слово @class также полезно в ситуации с *циклическими зависимостями*, т.е. когда класс A использует класс B, а класс B — класс A. Если вы попытаетесь импортировать каждый класс из заголовочного файла другого класса, то получите ошибку компиляции. Но если применить @class B в A.h, а @class A — в B.h, то проблема будет успешно разрешена.

Поехали

Итак, мы сумели исправить заголовочный файл класса Car. Но файлу Car.m для компиляции требуется больше информации о классах Tire и Engine. Компилятор должен знать, от каких классов унаследованы Tire и Engine, чтобы убедиться, что объекты в состоянии отвечать на посылаемые им сообщения. Для этого мы импортируем Tire.h и Engine.h в Car.m. Нам также требуется удалить раздел @implementation класса Car из файла CarParts-Split.m. Теперь файл Car.m имеет следующий вид.

```
#import "Car.h"
#import "Tire.h"
#import "Engine.h"

@implementation Car
– (void) setEngine: (Engine *) newEngine
{
   engine = newEngine;
} // setEngine

– (Engine *) engine
{
   return (engine);
} // engine

– (void) setTire: (Tire *) tire
       atIndex: (int) index
{
   if (index < 0 || index > 3) {
      NSLog (@"Неверный индекс (%d) в setTire:atIndex:",
           index);
      exit (1);
   }
   tires[index] = tire;
} // setTire:atIndex:

– (Tire *) tireAtIndex: (int) index
{
```

```
    if (index < 0 || index > 3) {
        NSLog (@"Неверный индекс (%d) в setTire:atIndex:",
            index);
        exit (1);
    }
    return (tires[index]);
} // tireAtIndex:

– (void) print
{
    NSLog (@"%@", engine);
    NSLog (@"%@", tires[0]);
    NSLog (@"%@", tires[1]);
    NSLog (@"%@", tires[2]);
    NSLog (@"%@", tires[3]);
} // print
@end // Car
```

Теперь можно в очередной раз скомпилировать и запустить программу и получить тот же вывод, что и ранее. Похоже, мы выполнили очередной рефакторинг (но об этом не рекомендуется говорить вслух). Мы улучшили внутреннюю структуру нашей программы, сохранив ее поведение неизменным.

Импорт и наследование

Нам надо освободить из заточения в CarParts-Split.m еще два класса: Slant6 и AllWeatherRadial. Данная задача немного сложнее, поскольку эти классы унаследованы от созданных нами классов: Slant6 наследует Engine, а AllWeatherRadial — Tire. Поскольку мы не просто применяем указатели на классы, а выполняем наследование, не можем воспользоваться трюком с ключевым словом @class в заголовочных файлах. Мы должны вставить #import "Engine.h" в файл Slant6.h и #import "Tire.h" — в файл AllWeatherRadial.h.

И все же, давайте уточним, почему именно мы не можем применить ключевое слово @class в этой ситуации. Дело в том, что компилятор должен все знать о надклассе, чтобы успешно скомпилировать раздел @interface для его подкласса. Компилятор должен знать схему размещения (типы, размеры и порядок) переменных экземпляра в надклассе. Вспомните, что когда вы добавляете переменные экземпляра в подкласс, они размещаются после переменных экземпляра надкласса. Затем компилятор использует эту информацию для определения расположения переменных экземпляра в памяти, начиная со скрытого указателя self, передаваемого при вызове каждого метода. Для корректного вычисления местоположения переменных экземпляра в памяти компилятору требуется полная информация о классе.

Теперь приступим к классу Slant6. Создадим новые файлы Slant6.m и Slant6.h в Xcode, затем вынесем раздел @interface этого класса из CarParts-Split.m. Если вы все сделаете аккуратно, то файл Slant6.h должен иметь следующий вид.

```
#import "Engine.h"

@interface Slant6 : Engine
@end // Slant6
```

Этот файл импортирует только Engine.h, но не <Cocoa/Cocoa.h>. Почему? Мы знаем, что Engine.h импортирует <Cocoa/Cocoa.h>, так что нам не надо импортировать его здесь еще раз самостоятельно. Но если вы поместите в этот файл директиву #import <Cocoa/Cocoa.h>, ничего страшного не произойдет, так как директива #import достаточно интеллектуальна, чтобы избежать многократного включения.

Файл Slant6.m содержит перенесенный из файла CarParts-Split.m раздел @implementation класса Slant6 с директивой импорта заголовочного файла Slant6.h.

```
#import "Slant6.h"

@implementation Slant6
– (NSString *) description
{
   return (@"Я — двигатель slant–6. P–p–p–p–p!");
} // description
@end // Slant6
```

Выполните аналогичные шаги по переносу класса AllWeatherRadial в собственную пару файлов. Вот как после этого должен выглядеть файл AllWeatherRadial.h.

```
#import "Tire.h"

@interface AllWeatherRadial : Tire
@end // AllWeatherRadial
```

А вот вид файла AllWeatherRadial.m.

```
#import "AllWeatherRadial.h"

@implementation AllWeatherRadial
– (NSString *) description
{
   return (@"Я – колесо, которому годится любая погода!");
} // description
@end // AllWeatherRadial
```

От файла CarParts-Split.m теперь осталась лишь жалкая оболочка. Теперь в нем нет ничего, кроме директив #import и единственной функции.

```
#import <Foundation/Foundation.h>
#import "Tire.h"
#import "Engine.h"
#import "Car.h"
#import "Slant6.h"
#import "AllWeatherRadial.h"

int main (int argc, const char * argv[])
{
   Car *car = [Car new];

   int i;
```

```
    for (i = 0; i < 4; i++) {
        Tire *tire = [AllWeatherRadial new];
        [car setTire: tire
            atIndex: i];
    }

    Engine *engine = [Slant6 new];
    [car setEngine: engine];

    [car print];

    return (0);
} // main
```

Если мы скомпилируем и запустим проект, то вновь получим тот же вывод на экран, что и до начала процесса "распыления" проекта по множеству файлов.

Резюме

В этой главе вы познакомились с применением множества файлов для организации вашего кода. Обычно каждому классу отводится два файла: заголовочный файл, содержащий интерфейс класса, и .m-файл с его реализацией. Пользователи класса импортируют посредством директивы #import заголовочный файл класса и получают доступ ко всем его возможностям.

Вы познакомились также с зависимостями между файлами, когда некоторый заголовочный или исходный файл нуждается в информации из другого заголовочного файла. Паутина зависимостей может привести к резкому увеличению времени компиляции и к излишней перекомпиляции — там, где без нее можно обойтись. Аккуратное применение директивы @class может снизить количество импортируемых заголовочных файлов, а значит, и время компиляции.

Далее мы познакомим вас с некоторыми интересными возможностями Xcode, так что не откладывайте книгу в сторону!