Глава 3
ПРОЕКТИРОВАНИЕ ИНТЕРФЕЙСА И API

Когда ваше приложение будет построено, вы, вероятно, захотите сами использовать некоторые его части в будущих проектах. Возможно, вы опубликуете часть кода, чтобы его могли использовать другие. И даже если вы не собираетесь этим заниматься, когда-нибудь код все равно может понадобиться. И тогда вам сильно поможет, если ваши интерфейсы будут написаны в расчете на повторное использование. Это значит, что вы должны задействовать парадигмы, часто встречающиеся в Objective-C, и понимать типичные проблемы.

За последние годы использование кода, написанного другими разработчиками, стало повсеместным явлением — особенно в сообществе открытого кода и в области компонентов, ставших популярными с появлением iOS. Другие пользователи также могут использовать ваш код, поэтому четкость и логичность кода поможет им быстрее и проще интегрировать ваши разработки. И кто знает — возможно, вы напишете библиотеку, которая будет применяться в тысячах приложений!

15 ИСПОЛЬЗУЙТЕ ПРЕФИКСЫ ДЛЯ ПРЕДОТВРАЩЕНИЯ КОНФЛИКТОВ ИМЕН

В отличие от других языков, Objective-C не имеет встроенной поддержки пространств имен. По этой причине в программах легко могут возникнуть конфликты имен, если только вы не примете меры

для их предотвращения. Как правило, из-за конфликтов имен приложения не проходят компоновку с выдачей ошибок о дубликатах символических имен:

```
duplicate symbol _OBJC_METACLASS_$_EOCTheClass in:
    build/something.o
    build/something_else.o
duplicate symbol _OBJC_CLASS_$_EOCTheClass in:
    build/something.o
    build/something_else.o
```

Ошибка возникла из-за того, что символические имена класса и метакласса (см. подход 14) для класса с именем `EOCTheClass` были определены дважды, в двух реализациях `EOCTheClass` в двух разных частях кода приложения — возможно, в двух разных библиотеках, которые вы подключили к своему проекту.

Впрочем, ошибка компоновки — еще не худший вариант. Представьте, что одна из библиотек, содержащих дубликаты, загружается на стадии выполнения. В этом случае динамический загрузчик обнаружит ошибку, что, скорее всего, приведет к аварийному завершению всего приложения.

Проблему можно предотвратить только одним способом: использованием примитивного подобия пространств имен, при котором все имена в программе начинаются с определенного префикса. Выбранный префикс должен быть связан с вашей компанией и/или приложением. Например, если компания называется Effective Widgets, вы можете использовать префикс `EWS` в коде, общем для всех приложений, и префикс `EWB` в коде приложения Effective Browser. Префиксы не исключают конфликты имен полностью, но существенно снижают их вероятность.

Если вы создаете приложения на базе Cocoa, следует помнить, что компания Apple зарезервировала право использования двухбуквенных префиксов для себя, поэтому в этом случае определенно стоит выбирать трехбуквенные префиксы. Например, если разработчик не соблюдал эти рекомендации и решал использовать префикс `TW`, это могло создать проблему. В пакет iOS 5.0 SDK была включена поддержка Twitter, которая использует префикс `TW`; в ней имеется класс `TWRequest` для создания запросов HTTP к Twitter API. Разработчик мог очень легко создать собственный класс `TWRequest` при создании собственного API — например, для компании Tiny Widgets.

Префиксы не должны ограничиваться именами классов. Применяйте их ко всем именам, встречающимся в приложениях. В подходе 25

объясняется важность префиксов имен категорий и их методов, если категория относится к существующему классу. Также часто забывают о другой важной области потенциальных конфликтов — функциях C и глобальных переменных, используемых в файлах реализации классов. Разработчики часто забывают, что эти имена будут присутствовать в таблице символических имен верхнего уровня в откомпилированных объектных файлах. Например, фреймворк AudioToolbox в iOS SDK содержит функцию для воспроизведения звукового файла. Для нее можно назначить функцию обратного вызова, которая будет вызвана по завершении воспроизведения. Возможно, вы решите написать класс для включения этой функциональности в класс Objective-C, который вызывает делегата по завершении звукового файла:

```
// EOCSoundPlayer.h
#import <Foundation/Foundation.h>

@class EOCSoundPlayer;
@protocol EOCSoundPlayerDelegate <NSObject>
- (void)soundPlayerDidFinish:(EOCSoundPlayer*)player;
@end

@interface EOCSoundPlayer : NSObject
@property (nonatomic, weak) id <EOCSoundPlayerDelegate>
delegate;
- (id)initWithURL:(NSURL*)url;
- (void)playSound;
@end

// EOCSoundPlayer.m

#import "EOCSoundPlayer.h"
#import <AudioToolbox/AudioToolbox.h>

void completion(SystemSoundID ssID, void *clientData) {
    EOCSoundPlayer *player =
        (__bridge EOCSoundPlayer*)clientData;
    if ([player.delegate
            respondsToSelector:@selector(soundPlayerDidFinish:)])
    {
        [player.delegate soundPlayerDidFinish:player];
    }
}

@implementation EOCSoundPlayer {
```

```
    SystemSoundID _systemSoundID;
}

- (id)initWithURL:(NSURL*)url {
    if ((self = [super init])) {
        AudioServicesCreateSystemSoundID((__bridge CFURLRef)url,
                                         &_systemSoundID);
    }
    return self;
}

- (void)dealloc {
    AudioServicesDisposeSystemSoundID(_systemSoundID);
}

- (void)playSound {
    AudioServicesAddSystemSoundCompletion(
        _systemSoundID,
        NULL,
        NULL,
        completion,
        (__bridge void*)self);
    AudioServicesPlaySystemSound(_systemSoundID);
}

@end
```

Выглядит вполне безобидно, но, заглянув в таблицу символических имен объектного файла, созданного на базе класса, мы видим следующее:

```
00000230 t -[EOCSoundPlayer .cxx_destruct]
0000014c t -[EOCSoundPlayer dealloc]
000001e0 t -[EOCSoundPlayer delegate]
0000009c t -[EOCSoundPlayer initWithURL:]
00000198 t -[EOCSoundPlayer playSound]
00000208 t -[EOCSoundPlayer setDelegate:]
00000b88 S _OBJC_CLASS_$_EOCSoundPlayer
00000bb8 S _OBJC_IVAR_$_EOCSoundPlayer._delegate
00000bb4 S _OBJC_IVAR_$_EOCSoundPlayer._systemSoundID
00000b9c S _OBJC_METACLASS_$_EOCSoundPlayer
00000000 T _completion
00000bf8 s l_OBJC_$_INSTANCE_METHODS_EOCSoundPlayer
00000c48 s l_OBJC_$_INSTANCE_VARIABLES_EOCSoundPlayer
00000c78 s l_OBJC_$_PROP_LIST_EOCSoundPlayer
00000c88 s l_OBJC_CLASS_RO_$_EOCSoundPlayer
00000bd0 s l_OBJC_METACLASS_RO_$_EOCSoundPlayer
```

Обратите внимание на прячущееся в середине имя `_completion`. Это функция завершения, которая должна обрабатывать завершение воспроизведения звука. И хотя она определяется в файле реализации без объявления в заголовочном файле, она все равно остается символическим именем верхнего уровня. Если в будущем будет создана другая функция с именем `completion`, во время компоновки произойдет ошибка дублирования символических имен:

```
duplicate symbol _completion in:
    build/EOCSoundPlayer.o
    build/EOCAnotherClass.o
```

Еще хуже, если ваш код будет распространяться как библиотека, которую другие разработчики будут использовать в своих приложениях. Получается, что пользователи такой библиотеки уже не смогут создать функцию с именем `completion`; вряд ли они похвалят вас за это.

Итак, функции C тоже всегда должны снабжаться префиксами. Например, в приведенном примере обработчику завершения можно присвоить имя `EOCSoundPlayerCompletion`. У правильного выбора имени также имеется побочный эффект: если символическое имя появится в данных трассировки, вы сможете легко определить, в каком коде возникли проблемы.

К проблемам дублирования символических имен следует относиться особенно осторожно в том случае, если вы используете стороннюю библиотеку в коде, который сам будет распространяться в виде библиотеки. Если используемые вами сторонние библиотеки также используются в приложении, это сильно повышает риск ошибок дублирования символических имен. В таких ситуациях обычно применяется редактирование всего кода используемых библиотек с заменой префикса. Например, если в вашей библиотеке EOCLibrary используется библиотека с именем XYZLibrary, префиксы всех имен в XYZLibrary заменяются префиксами `EOC`. После этого приложение может использовать библиотеку XYZLibrary без риска коллизий имен, как показано на рис. 3.1.

Конечно, замена всех имен в коде — дело довольно утомительное, но это разумная мера для предотвращения конфликтов имен. Почему все это необходимо, почему приложение не может просто не включать XYZLibrary и использовать вашу реализацию? Конечно, это возможно, но представьте ситуацию, в которой приложение подключает третью библиотеку ABCLibrary, в которой также

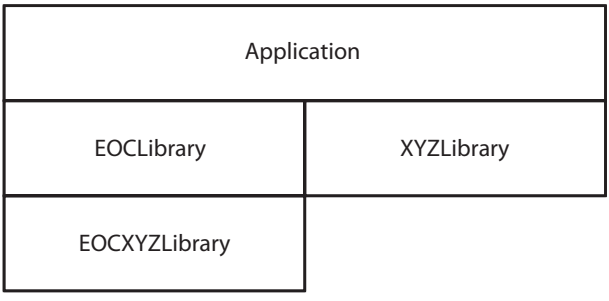

Рис. 3.1. Предотвращение конфликтов имен при повторном включении сторонней библиотеки (самим приложением и другой библиотекой)

используется библиотека XYZLibrary. В этом случае префиксы не используются ни вами, ни автором ABCLibrary и в приложении все равно появятся ошибки дублирующихся символических имен. Или если вы используете версию X библиотеки XYZLibrary, а приложению необходима функциональность из версии Y, ему все равно потребуется своя копия. Если у вас есть опыт использования популярных сторонних библиотек в программировании для iOS, такие префиксы вам наверняка попадались.

УЗЕЛКИ НА ПАМЯТЬ

- Выберите префикс класса по названию компании и/или приложения и используйте его во всем коде.
- Если вы используете стороннюю библиотеку в своей библиотеке, рассмотрите возможность замены ее имен с включением своего префикса.

16 ИСПОЛЬЗУЙТЕ ОСНОВНОЙ ИНИЦИАЛИЗАТОР

Все объекты должны инициализироваться. В некоторых случаях объекты инициализируются в состоянии по умолчанию, но часто объект не может выполнить инициализацию без получения дополнительной информации. Например, класс `UITableViewCell` из UIKit (фреймворк пользовательского интерфейса для iOS) при

своей инициализации должен получать стиль и идентификатор для группировки ячеек разных типов; это позволяет эффективно использовать объекты ячеек, создание которых сопряжено с большими затратами. Инициализатор, который передает объекту необходимую информацию для выполнения этой операции, называется *основным* (designated) инициализатором.

Если экземпляры класса могут создаваться несколькими способами, класс может иметь более одного инициализатора. Это абсолютно нормально, но при этом у него все равно должен быть только один основной инициализатор, используемый всеми остальными инициализаторами. Например, для класса `NSDate` определяются следующие инициализаторы:

```
- (id)init
- (id)initWithString:(NSString*)string
- (id)initWithTimeIntervalSinceNow:(NSTimeInterval)seconds
- (id)initWithTimeInterval:(NSTimeInterval)seconds
                 sinceDate:(NSDate*)refDate
- (id)initWithTimeIntervalSinceReferenceDate:
                                (NSTimeInterval)seconds
- (id)initWithTimeIntervalSince1970:(NSTimeInterval)seconds
```

Основным инициализатором в данном случае является `initWithTimeIntervalSinceReferenceDate:`, как указано в документации класса. Это означает, что все остальные инициализаторы в конечном итоге передают ему управление. Таким образом, сохранение внутренних данных осуществляется только в основном инициализаторе. Если данные по какой-либо причине изменятся, то изменения достаточно внести всего в одном месте.

Для примера возьмем класс, представляющий прямоугольник. Его интерфейс выглядит так:

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

@interface EOCRectangle : NSObject
@property (nonatomic, assign, readonly) float width;
@property (nonatomic, assign, readonly) float height;

@end
```

В соответствии с рекомендациями из подхода 18 свойства доступны только для чтения. Но это означает, что свойства объекта `Rectangle` не могут быть заданы извне, поэтому в класс включается инициализатор:

```
- (id)initWithWidth:(float)width
          andHeight:(float)height
{
    if ((self = [super init])) {
        _width = width;
        _height = height;
    }
    return self;
}
```

Но что, если кто-то захочет создать прямоугольник вызовом `[[EOCRectangle alloc] init]`? Это вполне законно, поскольку суперкласс `EOCRectangle` — `NSObject` — реализует метод с именем `init`, который просто задает всем переменным экземпляров значение 0 (или эквивалент 0 для фактического типа данных). При вызове этого метода ширина и высота прямоугольника сохранят значение 0 от момента создания экземпляра `EOCRectangle` (в вызове `alloc`), когда все переменные экземпляра были обнулены. И хотя может оказаться, что именно такое поведение вам и нужно, с большей вероятностью вы захотите либо задать значения по умолчанию самостоятельно, либо выдать исключение, указывающее, что экземпляр должен инициализироваться только с использованием основного инициализатора. В случае `EOCRectangle` это означает переопределение метода `init`:

```
// Использование значений по умолчанию
- (id)init {
    return [self initWithWidth:5.0f andHeight:10.0f];
}

// Выдача исключения
- (id)init {
    @throw [NSException
             exceptionWithName:NSInternalInconsistencyException
              reason:@"Must use initWithWidth:andHeight:
instead."
           userInfo:nil];
}
```

Обратите внимание на то, как версия, задающая значения по умолчанию, передает управление основному инициализатору. Ее также можно было бы реализовать непосредственным заданием переменных экземпляра `_width` и `_height`. Но если способ хранения данных изменится (например, с использованием структуры для хранения ширины и высоты вместе), логику присваивания придется изменять в обоих инициализаторах. В этом простом примере это не создает особых проблем, но представьте себе более сложный класс

с существенно большим количеством инициализаторов и более сложными данными. В этом случае повышается вероятность того, что разработчик забудет изменить один из инициализаторов, что приведет к логическим несоответствиям.

Теперь представьте, что вам потребовалось субклассировать класс прямоугольника `EOCRectangle` и создать класс с именем `EOCSquare`, представляющий квадрат. Понятно, что такая иерархия разумна, но где должен находиться инициализатор? Естественно, ширина и высота должны совпадать — ведь это следует из самого определения квадрата! Возможно, вам захочется создать инициализатор следующего вида:

```
#import "EOCRectangle.h"

@interface EOCSquare : EOCRectangle
- (id)initWithDimension:(float)dimension;
@end

@implementation EOCSquare

- (id)initWithDimension:(float)dimension {
    return [super initWithWidth:dimension andHeight:dimension];
}

@end
```

Он становится основным инициализатором `EOCSquare`. Обратите внимание на вызов основного инициализатора суперкласса. Обратившись к реализации `EOCRectangle`, мы видим, что она тоже вызывает основной инициализатор своего суперкласса. Очень важно, чтобы эта цепочка вызовов основных инициализаторов сохранялась, но вызывающая сторона может инициализировать объект `EOCSquare` с использованием как `initWithWidth:andHeight:`, так и метода `init`. Такая ситуация нежелательна, поскольку в ней может быть создан «квадрат», у которого ширина и высота не совпадают. В этом проявляется один важный аспект субклассирования: вы всегда должны переопределять основной инициализатор своего суперкласса, если вы используете основной инициализатор с другим именем. В случае `EOCSquare` переопределение основного инициализатора `EOCRectangle` может выглядеть так:

```
- (id)initWithWidth:(float)width andHeight:(float)height {
    float dimension = MAX(width, height);
    return [self initWithDimension:dimension];
}
```

Обратите внимание на вызов основного инициализатора `EOCSquare`. С этой реализацией, как по волшебству, также начинает работать метод `init`. Вспомните, что в `EOCRectangle` его реализация передавала управление основному инициализатору `EOCRectangle` со значениями по умолчанию. Она по-прежнему это делает, но так как метод `initWithWidth:andHeight:` был переопределен, вызывается реализация `EOCSquare`, которая в свою очередь передает управление основному инициализатору. Получается, что все работает, и создать объект `EOCSquare` с разными сторонами не удастся.

Иногда переопределение основного инициализатора суперкласса нежелательно, поскольку оно не имеет смысла. Например, вы можете решить, что присваивание объекту `EOCSquare`, созданному методом `initWithWidth:andHeight:`, размера, равного большему из двух переданных значений, является ошибкой пользователя. В таких ситуациях обычно применяется переопределение основного инициализатора с выдачей исключения:

```
- (id)initWithWidth:(float)width andHeight:(float)height {
    @throw [NSException
        exceptionWithName:NSInternalInconsistencyException
                   reason:@"Must use initWithDimension: instead."
                 userInfo:nil];
}
```

Такое решение выглядит довольно радикально, но иногда оно неизбежно, потому что созданный объект содержит противоречивые внутренние данные. В примере с `EOCRectangle` и `EOCSquare` это будет означать, что вызов `init` также приведет к выдаче исключения, потому что `init` передает управление `initWithWidth:andHeight:`. В таком случае можно переопределить `init` и вызвать `initWithDimension:` с разумным значением по умолчанию:

```
- (id)init {
    return [self initWithDimension:5.0f];
}
```

Однако выдача исключения в Objective-C означает, что ошибка фатальна (см. подход 21), так что выдача исключения из инициализатора должна стать последней мерой, если экземпляр не может быть инициализирован другим способом.

В некоторых ситуациях одного основного инициализатора оказывается недостаточно — например, если экземпляр может создаваться двумя разными способами, которые должны обрабатываться

по отдельности. В качестве примера можно привести протокол `NSCoding` — механизм сериализации, позволяющий объектам управлять процессом своего кодирования и декодирования. Этот механизм широко используется в AppKit и UIKit, двух фреймворках пользовательского интерфейса для Mac OS X и iOS соответственно, для управления сериализацией объектов в формате XML и созданием так называемых NIB-файлов, обычно используемых для хранения макета контроллера представлений. При загрузке из NIB-файла контроллер представления проходит процесс распаковки.

Согласно протоколу `NSCoding`, должен быть реализован следующий метод инициализатора:

```
- (id)initWithCoder:(NSCoder*)decoder;
```

Этот метод обычно не может передать управление основному инициализатору, потому что ему приходится выполнять разную работу для распаковки объекта через декодер. Кроме того, если суперкласс тоже реализует `NSCoding`, должен быть вызван его метод `initWithCoder:`. В результате у класса появляются два основных инициализатора в формальном понимании, так как более чем один инициализатор передает управление инициализатору суперкласса.

Применительно к `EOCRectangle` мы получаем следующее:

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

@interface EOCRectangle : NSObject <NSCoding>
@property (nonatomic, assign, readonly) float width;
@property (nonatomic, assign, readonly) float height;
- (id)initWithWidth:(float)width
          andHeight:(float)height;
@end

@implementation EOCRectangle

// Основной инициализатор
- (id)initWithWidth:(float)width
          andHeight:(float)height
{
    if ((self = [super init])) {
        _width = width;
        _height = height;
    }
    return self;
}
```