Пробелы и форматирование

Табуляция

Отступы перед строками следует делать с помощью табуляции. Настройте редактор Xcode таким образом, чтоб размер табуляции, как и размер отступов, был равен 3-м пробелам.

Объявление методов

Следует использовать один пробел между - или + и возвращаемым типом, а также между параметрами, и нигде более.
Открывающаяся фигурная скобка ставится в той же строке, где и имя метода.

Пример:

```
- (void)doSomethingWith:(GTMFoo *)theFoo rect:(NSRect)theRect {  // ВЕРНО
   ...
}

-(void) doSomethingWith:(GTMFoo *)theFoo rect:(NSRect)theRect {  // НЕВЕРНО
   ...
}

- (void)doSomethingWith:(GTMFoo *)theFoo rect:(NSRect)theRect    // НЕВЕРНО
{
   ...
}
```

Пробел перед открывающейся фигурной скобкой — необязателен.

Если параметров слишком много для размещения в одной строке, предпочтительно размещать каждый параметр в отдельной строке. Строки с параметрами выравниваются по двоеточиям перед типом данных:

```
- (void)doSomethingWith:(GTMFoo *)theFoo
                    rect:(NSRect)theRect
                interval:(float)theInterval {
   ...
}
```

В случае, если имя метода длинне имен параметров, выравнивать параметры следует по левому краю, с помощью 2-х символов табуляции:

```
- (void)short:(GTMFoo *)theFoo
      longKeyword:(NSRect)theRect
      evenLongerKeyword:(float)theInterval {
   ...
}
```

Имена параметров должны быть указаны обязательно:

```
- (void)doSomethingWith:(GTMFoo *)theFoo :(NSRect)theRect {  // НЕВЕРНО
   ...
}
```

Вызов методов

Вызовы методов следует делать в таком же стиле, как и их объявление. Пробелы ставятся только после имени объекта и между параметрами:

```
[myObject doFooWith:arg1 name:arg2 error:arg3];
```

или в случае отдельной строки для каждого параметра:

```
[myObject doFooWith:arg1
               name:arg2
              error:arg3];
```

Не следует использовать такие стили:

```
[myObject doFooWith:arg1 name:arg2  // строка с более чем одним аргументом
              error:arg3];

[myObject doFooWith:arg1
               name:arg2 error:arg3];

[myObject doFooWith:arg1
          name:arg2  // выравнивание по аргументам вместо двоеточий
          error:arg3];
```

@public и @private

Модификаторы @public и @private должны иметь отступ в 1 пробел:

```
@interface MyClass : NSObject {
 @public
   ...
 @private
   ...
}
@end
```

Исключения

Исключения описываются таким образом, чтоб каждый оператор занимал отдельную строку и был отделен пробелом от открывающейся фигурной скобки ({), так же, как и оператор @catch — от объявления объекта исключения:

```
@try {
   foo();
}
@catch (NSException *ex) {
   bar(ex);
}
@finally {
   baz();
}
```

Протоколы

Между именем типа и именем протокола, заключенным в угловые скобки, не должно быть пробела. Это касается объявлений классов, переменных и методов.

Например:

```
@interface MyProtocoledClass : NSObject<NSWindowDelegate> {
```

```
 @private
   id<MyFancyDelegate> delegate_;
}

- (void)setDelegate:(id<MyFancyDelegate>)aDelegate;

@end
```

Имена

При выборе имен следует придерживаться правил Apple, ознакомиться подробнее с которыми можно в следующем документе: Apple Guide.

Имена файлов

Имена файлов должны отражать имена классов, реализацию которых они содержат (с учетом регистра).
Расширения различных типов файлов:

.h	C/C++/Objective-C header file
.m	Objective-C implementation file
.mm	Objective-C++ implementation file
.cc	Pure C++ implementation file
.c	C implementation file

Имена файлов для категорий должны включать имя расширяемого класса, а также (через знак +) функциональность расширения:

GTMNSString+Utils.h или GTMNSTextView+Autocomplete.h

Имена классов

Имена классов (также как имена категорий и протоколов) должны начинаться с заглавной буквы и отделять заглавной буквой каждое слово.

В коде в рамках одного приложения следует избегать префиксов в именах классов. Наличие большого количества классов с одинаковыми префиксами в именах затрудняет чтение кода при том, что не несет никакой выгоды. При разработке кода с целью использования в нескольких приложениях, префиксы допустимы и даже желательны (например `GTMSendMessage`).

Имена категорий

Имя категории должно включать имя расширяемого класса. Например, если мы создаем категорию `NSString` для парсинга, мы должны описать ее в файле с именем `GTMNSString+Parsing.h`, а сама категория будет иметь имя `GTMStringParsingAdditions` (имя категории не совпадает с именем файла, т.к. Этот файл может содержать несколько категорий, имеющих отношение к парсингу).

Имя класса должно быть отделено пробелом от открывающейся скобки с именем категории.

Имена методов

Имена методов должны начинаться с маленькой буквы и отделять заглавной буквой каждое слово. Имя каждого параметра также должно начинаться с маленькой буквы.

Имя метода вместе с именами параметров по возможности должно читаться как связное предложение. (Например: `convertPoint:fromRect:` или `replaceCharactersInRange:withString:`).

Смотрите более детальную информацию в Apple Guide.

Методы, возвращающие значения, должны быть названы так же, как имя переменной, значение которой они возвращают, без префикса "get". Например:

```
- (id)getDelegate;  // НЕВЕРНО

- (id)delegate;    // ВЕРНО
```

Имена переменных

Имена переменных должны начинаться с маленькой буквы и отделять заглавной буквой каждое слово. Давайте настолько осмысленные имена, насколько это возможно. Не заботьтесь о горизонтальной длине строк, т.к. гораздо более важно сделать ваш код интуитивно понятным для последующего читателя. Например:

```
int w;
int nerr;
int nCompConns;
tix = [[NSMutableArray alloc] init];
obj = [someObject object];
p = [network port];                         // НЕВЕРНО

int numErrors;
int numCompletedConnections;
tickets = [[NSMutableArray alloc] init];
userInfo = [someObject object];
port = [network port];                      // ВЕРНО
```

Константы

Константы, объявленые с помощью директивы #define, должны иметь имена в верхнем регистре и отделять каждое слово знаком подчеркивания. Имя константы отделяется от директивы #define пробелом, так же, как и от значения. Допускается выравнивание значений с помощью табуляции.
Например:

```
#define SCREEN_HEIGHT           480
#define SCREEN_WIDTH       320
```

Перечисления

Имя типа перечисления следует начинать с имени класса, к которому оно относится логически (если таковой имеется). Имя каждого из значений перечисления должно содержать имя типа перечисления в качестве префикса. Например, если перечисление определяет типы объекта класса UIButton, то оно будет описано следующим образом:

```
typedef enum {
   UIButtonTypeCustom = 0,
   UIButtonTypeRoundedRect,
   UIButtonTypeDetailDisclosure,
   UIButtonTypeInfoLight,
   UIButtonTypeInfoDark,
   UIButtonTypeContactAdd,
} UIButtonType;
```

Делегаты

Имя делегата должно состоять из имени класса, к которому он относится и слова Delegate.
Имена методов делегата соответствуют тем же правилам, что и имена методов класса.
Имя метода делегата вместе с именами параметров, по возможности, должно представлять собой связное предложение, передающее смысл того, какое событие объекта передается делегату.
Первым параметром метода должен быть сам объект, в котором произошло событие.
Пример делегата для класса UITableView:

```
@protocol UITableViewDelegate

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath;
```

или в случае, когда никаких дополнительных параметров не передается:

```
@protocol SomeClassDelegate

- (void)someClassDidLoad:(SomeClass *)someClass;
```

Управление памятью

Предпочитайте autorelease во время создания

При создании временных объектов, используйте autorelease в строке их создания вместо отдельного release в последующих строках.

```
// НЕВЕРНО (если только это не требуется из соображений производительности)
MyController* controller = [[MyController alloc] init];
// ... code here that might return ...
[controller release];

// ВЕРНО
MyController* controller = [[[MyController alloc] init] autorelease];
```

Методы, возвращающие объекты

Методы, результатом которых являются новые объекты, должны создавать их с использованием autorelease. Последующий retain таких объектов делается (если это необходимо) теми методами, которые вызвали данный метод. Пример:

```
// ВЕРНО
- (NSMutableArray *)createDigits {
   NSMutableArray *arr = [NSMutableArray array];
   for(int digit = 0; digit < 10; digit++) {
      [arr addObject: [NSNumber numberWithInt: digit]];
   }
   return arr;
}

- (void)someMethod{
     NSMutableArray *digits = [[self createDigits] retain];
```

```
}

//  НЕВЕРНО
- (NSMutableArray *)createDigits {
   NSMutableArray *arr = [[NSMutableArray alloc] init];
   for(int digit = 0; digit < 10; digit++) {
      [arr addObject: [NSNumber numberWithInt: digit]];
   }
   return arr;
}

- (void)someMethod{
     NSMutableArray *digits = [self createDigits];
}
```

IBOutlet

Если вы объявили какой-либо IBOutlet, то его обязательно нужно прорелизить в методе dealloc. А если для этого аутлета еще и было созданно property с retain, то необходимо присвоить ему nil в методе viewDidUnload:

```
- (void)viewDidUnload {

     // Release any retained subviews of the main view.

     self.myOutlet = nil;

}
```

Комментарии

Каким бы неприятным ни был процесс написания комментариев, они абсолютно необходимы для того, чтоб код оставался понятным. Данные правила говорят о том, что и где следует комментировать. Но помните: как бы не были важны комментарии, лучший код — тот, который сам себя комментирует. Давать понятные имена классам, методам и переменным — лучше, чем использовать непонятные имена, а затем пытаться объяснить их с помощью комментариев.

Комментарии объявления

Объявления каждого класса, категории и протокола должны иметь комментарий, описывающий их назначение и взаимодействие с другими частями приложения:

```
// A delegate for NSApplication to handle notifications about app
// launch and shutdown. Owned by the main app controller.
@interface MyAppDelegate : NSObject {
   ...
}
@end
```

Комментарии реализации

Особое внимание при добавлении комментариев уделяйте тем фрагментам кода, которые

исполняются несколькими потоками, подробно описывайте все особенности их синхронизации.

Используйте вертикальные линии, чтоб обозначить имена переменных и другие фрагменты кода в комментариях, чтоб не было путаницы с фрагментами, уже содержащими кавычки:

```
// Remember to call |StringWithoutSpaces("foo bar baz")|
```

Свойства vs Переменные класса

В классах объявляйте свойства, а не просто переменные. Это облегчит работу с памятью, позволит делать разные проверки в возвращателях (getters), позволит наследникам переопределять поведение. Пример:

```
@interface ClassA ()

@property (nonatomic, retain) NSArray *dudes;

@end

@implementation ClassA

@synthesize dudes = _dudes;

// …
```

В этом примере при генерации кода для свойства dudes используется переменная класса _dudes. Это нужно для того, чтобы не перепутать случайно обращение к свойству и переменной:

```
// dudes = [NSArray array];       // error: use of undeclared
identifier 'dudes'

self.dudes = [NSArray array];
```

Однако! Не рекомендуется использовать свойства в init и dealloc методах, т.к. возвращатели (getters) и установители (setters) могут иметь побочные эффекты, которые могут обломать (crash) приложение из-за недостроенных/полуразрушенных объектов.

```
- (id)init {
    if (self = [super init]) {
        _dudes = [[NSArray alloc] init];   // эквивалентно retain
    // …

- (dealloc) {
    [_dudes release], _dudes = nil;
    // …
```

```
    [super dealloc];
}
```

Также, рекомендуется метод dealloc располагать ближе к @synthesize (перед/после init), чтобы не забывать удалять объекты.

/* http://stackoverflow.com/questions/1051543/should-i-use-self-keyword-properties-in-the-implementation/1052035#1052035 */

Строковые константы

Для строк рекомендуется использовать следующий способ:

```
// Const.h
extern NSString *const kWTFString;

// Const.m
NSString *const kWTFString = @"WTF is going on here?";
```

Преимущество -- не надо пересобирать весь проект при изменении значения.

/* http://stackoverflow.com/questions/538996/constants-in-objective-c/539039#539039 */

Сравнение чисел с плавающей точкой (floating-point numbers)

Не используйте оператор сравнения == для сравнения чисел типа float и double! Из-за особенностей их представления в цифровой технике (http://en.wikipedia.org/wiki/IEEE_754-2008) такое сравнение некорректно. Пример:

```
if (alpha == 1.0f)        // НЕВЕРНО

#define fequal(a,b) (fabs((a) - (b)) < FLT_EPSILON)
#define fequalzero(a) (fabs(a) < FLT_EPSILON)

if (fequal(alpha, 1.0f))            // ВЕРНО
```

/* http://stackoverflow.com/questions/1614533/strange-problem-comparing-floats-in-objective-c/1614761#1614761 */

Частные вещи (private things) класса

Если свойство/метод является внутренним для класса, не нужно выносить их определение в .h файл. Класс должен предоставлять минимально необходимый публичный интерфейс.

Плохо:

```
// MyAwesomeCat.h
@interface MyAwesomeCat : NSObject {
@public
     int legsNumber; // what is it here for? have you ever seen a cat
with 5+ legs?
     NSString *name;
@private
     NSString *freakingName;
}

@property (nonatomic, assign) int legsNumber;
@property (nonatomic, retain) NSString *name;

@end
```

Хорошо:

```
// MyAwesomeCat.h
@interface MyAwesomeCat : NSObject
     // нет объявлений никаких переменных

@property (nonatomic, readonly, assign) int legsNumber;
@property (nonatomic, copy) NSString *name;

- (id)initWithName:(NSString *)name;

@end

// MyAwesomeCat.m
@interface MyAwesomeCat ()

@property (nonatomic, readwrite, assign) int legsNumber;  //
переопределяем свойство, чтобы внутри можно было читать/писать
@property (nonatomic, copy) NSString *freakingName;

@end


@implementation MyAwesomeCat

@synthesize legsNumber = _legsNumber;
@synthesize name = _name;
@synthesize freakingName = _freakingName;

- (id)initWithName:(NSString *)name {
     if (self = [super init]) {
          _name = [name copy];
```

```
            _legsNumber = 5;    // для устойчивости
        }
        return self;
}

// …

@end
```

Заметьте, что свойство name изначально было retain, но стало copy. Таким образом, внешний класс не сможет никак изменить значение name после присваивания (иначе, если присвоить NSMutableString, это возможно сделать).

/* http://stackoverflow.com/questions/155964/what-are-best-practices-that-you-use-when-writing-objective-c-and-cocoa/156098#156098 2), http://stackoverflow.com/questions/387959/nsstring-property-copy-or-retain/388002#388002 */

Do you want more tips & tricks? Go to "w w w dot", oh no, it's http://stackoverflow.com/questions/155964/what-are-best-practices-that-you-use-when-writing-objective-c-and-cocoa

Паттерны проектирования

Паттерн Observer

Пример из жизни

Студент Вася очень любит ходить на вечеринки. Если быть точнее, они заняли настолько важную часть его жизни, что его выгнали из института. Занявшись поиском работы, Вася понял, что почти ничего не знает и не умеет. Хотя, подождите… У Васи есть очень много знакомых красивых девушек, которых хлебом не корми — дай проникнуть на классную тусовку без приглашения. Также Васю знают во всех соответствующих заведениях его города. Наконец, Вася понимает: чтобы вечеринка удалась (обеспеченные посетители потратили много денег), хорошо бы наполнить ее красивыми девушками (на которых эти деньги будут потрачены).

Так Вася стал сутенером открыл свой бизнес. Владельцы заведений обращаются к Васе (кто же не знает Васю?!) с новостями о модных закрытых мероприятиях. Вася сообщает своим знакомым девушкам о том, что он может их туда провести. Откуда же он берет девушек? Все просто: возьмем Свету. Она познакомилась с Васей в клубе на прошлой неделе. Света давно мечтала побывать на какой-нибудь элитной тусовке, поэтому попросила Васю записать ее номер телефона. Девушка она симпатичная, одета совсем недурно, так что Вася согласился. Так Света **подписалась** на Васины услуги и стала **наблюдателем**. Впрочем, Вася предупредил ее, что позвонит он сам, когда придет время.

Прошел год, Света закончила институт, нашла себе работу и, главное, вышла замуж! Ей больше не хочется ходить на вечеринки, поэтому вчера она позвонила Васе и попросила не донимать ее больше своими идиотскими сообщениями. Так она **отписалась** от (весьма подозрительного) **субъекта** Васи. Впрочем, она знает, что всегда может подписаться снова (она все еще симпатичная, молодая, хорошо одевается).

Тем временем, Вася не стоял на месте, а расширял бизнес. Недавно он познакомился с несколькими весьма обеспеченными футболистами, которые любят расслабиться на вечеринке и всегда готовы угостить симпатичных девушек безалкогольным коктейлем. Естественно, их тоже можно приглашать на разного рода мероприятия (организаторы в восторге, платят Васе больше денег). Расширение бизнеса прошло незаметно: действительно, какая разница, кому звонить? И у девушек, и у футболистов есть телефоны. Иногда, сообщая о новом мероприятии, Вася даже путает, с кем он разговаривает.

Пример покороче

Те, кто внимательно следил за Васиным бизнесом, мог сразу вспомнить новостные рассылки. Действительно, мы можем оставить свой адрес электронной почты на любимом ресурсе и получать спам важные и интересные новости без необходимости каждый день посещать соответствующую веб-страницу. Как и в случае со Светой, мы всегда (по факту, если повезет) можем от рассылки отписаться.

Определение

Банда Четырех

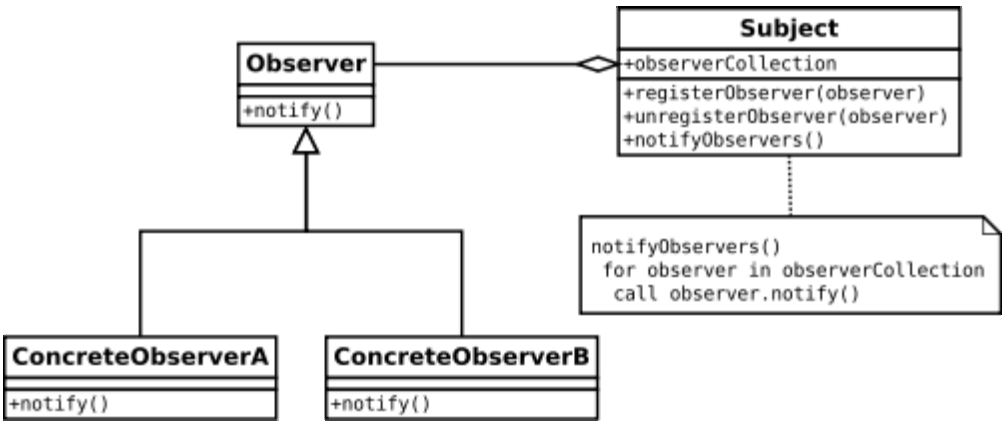
- Название: **Наблюдатель.**
- Классификация: Паттерн поведения.
- Назначение:
- Определяет зависимость типа *один ко многим* между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Комментарии

Итак, паттерн наблюдатель определяет зависимость один ко многим. При этом объект, который сообщает о своих изменениях, называется **субъектом**, а те объекты, которым он о них сообщает — **наблюдателями**.

Стоит отметить важную особенность паттерна наблюдатель: субъект может не знать о наблюдателях практически ничего. Так, Вася не видел никакой разницы между девушками и футболистами.

Структура



Наблюдатель для iOS разработчика

Давайте перейдем к программированию. Сначала мы разберем собственную реализацию паттерна наблюдатель на конкретном примере, а потом разберем механизм оповещений, реализованный в Cocoa.

Собственная реализация

Допустим, мы разрабатываем совершенно новую игру, которая непременно взорвет App Store. После прохождения очередного уровня нам нужно сделать две вещи:

1. Показать поздравительный экран.
2. Открыть доступ к новым уровням.

Стоит отметить, что первое и второе действия никак между собой не связаны. Мы можем выполнять их в произвольном порядке. Также мы подозреваем, что в скором времени понадобится расширить число таких действий (например, мы захотим посылать какие-то данные на свой сервер или проверить, не заработал ли пользователь достижение в Game Center).

Применим изученный паттерн наблюдателя. Начнем с протокола наблюдателя.

```
@protocol GameStateObserver <NSObject>

- (void)completedLevel:(Level *)level withScore:(NSUInteger)score;

@end
```

Здесь все довольно прозрачно: наблюдатели будут реализовывать протокол `GameStateObserver`.

Разберемся с субъектом.

```
@protocol GameStateSubject <NSObject>

- (void)addObserver:(id<GameStateObserver>)observer;
- (void)removeObserver:(id<GameStateObserver>)observer;
- (void)notifyObservers;

@end
```

Перейдем к интересующим нас классам. Пусть состояние текущей игры хранится в объекте класса`GameState`. Тогда его определение будет выглядеть примерно так:

```
@interface GameState : NSObject <GameStateSubject> {
    …
    NSMutableSet *observerCollection;
}

@property (readonly) Level *level;
@property (readonly) NSUInteger score;

…
- (void)updateState;

@end
```

При этом мы считаем, что метод `updateState` вызывается каждый раз, как в игре происходят существенные изменения. Приведем часть реализации `GameState`:

```
@implementation GameState

…
- (void)addObserver:(id<GameStateObserver>)observer {
    [observerCollection addObject:observer];
}

- (void)removeObserver:(id<GameStateObserver>)observer {
    [observerCollection removeObject:observer];
}

- (void)notifyObservers {
    for (id<GameStateObserver> observer in observerCollection) {
        [observer completedLevel:self.level withScore:self.score];
    }
}
```

```
- (void)updateState {
    …
    if (levelCompleted) {
        [self notifyObservers];
    }
}

@end
```

Теперь всякому объекту, которому необходимо знать об успешном проходжении уровня, достаточно реализовать протокол `GameStateObserver` и подписаться на оповещение об успешном завершении. Соотвествующий код будет выглядеть примерно так:

```
GameState *gameState = [[GameState alloc] init];
[gameState addObserver:levelManager];
[gameState addObserver:levelViewController];
```

Здесь `levelViewController` — контроллер, отвечающий за интерфейс игрового процесса, а `levelManager`— объект модели, отвечающий уровни.

Обсуждение

Мы рассмотрели довольно примитивный пример, который, однако, встречается повсеместно. Сразу же видно, что решение достаточно гибкое. Стоит отметить, что в оповещении мы решили передавать в качестве параметров некоторые данные. У такой реализации есть как свои плюсы, так и минусы. Может оказаться удобным использовать следующий вариант протокола `GameStateObserver`:

```
@protocol GameStateObserver <NSObject>

- (void)levelCompleted:(id<GameStateSubject>)subject;

@end
```

Соответствующий вариант `GameStateSubject` выглядел бы так:

```
@protocol GameStateSubject <NSObject>

- (void)addObserver:(id<GameStateObserver>)observer;
- (void)removeObserver:(id<GameStateObserver>)observer;
- (void)notifyObservers;
```

```
@property (readonly) Level *level;
@property (readonly) NSUInteger score;

@end
```

Наблюдатель в Cocoa: Notifications

Оказывается, в Cocoa есть механизм, позволяющий реализовать паттерн наблюдателя. Если быть совсем точным, таких механизмов два. В данный момент мы остановимся на механизме оповещений, а второй оставим на будущее. Далее мы не будем вдаваться во все тонкости, а лишь опишем базовую функциональность.

Неформально, механизм оповещений позволяет делать две вещи: подписываться/отписываться от оповещения и разослать оповещение всем подписчикам. Оповещение представляет из себя экземпляр класса `NSNotification`. Оповещения задаются своим строковым именем `name` типа `NSString`. Помимо имени, оповещение содержит также субъект `object` и дополнительные данные `userInfo` типа `NSDictionary`.

За подписки и доставку оповещений отвечает `NSNotificationCenter`. Чтобы получить к нему доступ, в большинстве случаев достаточно вызвать классовый метод `defaultCenter`.

Чтобы подписаться на сообщение, у центра оповещений имеется метод`addObserver:selector:name:object:`. Первым параметром выступает наблюдатель, вторым — селектор, который будет вызываться при оповещении. Он должен иметь сигнатуру `-(void)methodName:(NSNotification *)`. Третий параметр — имя оповещения, четвертый — субъект. Если в качестве субъекта передать `nil`, то оповещение будет доставляться от произвольного отправителя (при условии совпадения имени). С использованием оповещений код подписки выглядел бы так:

```
GameState *gameState = [[GameState alloc] init];
[[NSNotificationCenter defaultCenter] addObserver:levelManager
    selector:@selector(levelCompleted:)
    name:@"LevelCompletedNotification" object:gameState];
[[NSNotificationCenter defaultCenter] addObserver:levelViewController
    selector:@selector(levelCompleted:)
    name:@"LevelCompletedNotification" object:gameState];
```

Примерный вид метода `levelCompleted:`

```
- (void)levelCompleted:(NSNotification *)notification {
    id<GameStateSubject> subject = [notification object];
    Level *level = subject.level;
    NSUInteger score = subject.score;
}
```

Вообще говоря, от протокола `GameStateSubject` можно избавиться и использовать напрямую `GameState`.

Чтобы отписаться от оповещений, нужно вызвать один из методов `removeObserver:` или`removeObserver:name:object:`.

Отправка оповещения — процесс еще более простой. На то имеются методы`postNotificationName:object:` и `postNotificationName:object:userInfo:`. Первый выставляет значение`userInfo` нулевым `nil`. Новая реализация метода `notifyObservers` следует.

```
- (void)notifyObservers {
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"LevelCompletedNotification" object:self];
}
```