Хрестоматия iOS паттернов. На всякий…

V1.1

Дима Малеев

Оглавление

Привет

Привет, Друг! Очень мило что ты решил прочитать эту книгу. Ну или просто скачал чтобы посмотреть что тут находится. А находятся здесь просто примеры реализации паттернов GoF для iOS. Так как примеры писаны на Objective-C, вероятнее всего их можно использовать и для Mac, но так как я круче програмы чем “Hello, World!” под Mac не писал – то утверждать не могу.
Что тебя ждет дальше? Невероятно, но паттерны! И вероятнее всего граматические ошибки, потому как я просто физически не умею писать без ошибок. Но, я торжественно клянусь, что исправлю каждую ошибку которую ты мне пришлешь на diwingless@gmail.com. Я очень надеюсь, что ты не граммар наци, и поможешь мне исправить ошибки, и конечно же оценишь те знания которыми я попытался поделиться с тобой.

Для кого эта книга?

Книга будет полезна всем iOS разработчикам, потому как книги просто полезно читать. Говорят - это улучшает память. Для начинающих разработчиков, можно будет прочитать про паттерны и примеры их реализации, в будущем про это вас спросят на собеседовании. Для более продвинутых ребят, книга может послужить небольшой напоминалкой про забытое описание паттернов. Книга однозначно не являет собой учебник по Obj-C, и прочитва ее вы врятли сможете написать вторую Angry Birds, однако некоторые проблемы она вам все же решить поможет. Паттерны вообще хорошо знать чтобы структурировать свои знания в голове☺

Почему эта книга?

По моему глубоко личному убеждению – знания должны быть бесплатны. Просто, представьте как далеко было бы человечество, если бы у нас культура была бы направлена не на бесконечное зарабатывание денег, а на продвижение человечества к звездам? К сожалению, я не фармацевт который придумал лекарство от рака, и не ученый который придумал телепортацию☺ Потому, я попытаюсь поделиться знаниями которые есть у меня. Что меня радует – я далеко не первый, и даже решение о написании этой книги пришло мне в голову из-за успеха Андрея Будая с его книжкой, которую я вам очень рекомендую почитать. Внимательный читатель увидит, что книги сами по себе очень похожи, разве что язык примеров другой (ну и сами примеры, кроме одного).

Распространение книги

Книга бесплатна☺ Глубоко писать другое, особенно если вы прочитали предыдущий пункт. Распространятся книга будет путем скачивания откуда угодно. Я не очень знаю какие лицензии в ходу, да и в данном случае мне откровенно все равно. Потому, просто пару правил:

1. Книга не должна продоваться. Книга может только беслптано распространяться в любом виде, но за просто так. Совсем.
2. Книгу можно перепечатывать, копировать себе в блог, отсылать голубиной почтой и даже менять имя автора. Если вспомните вашего покорного слугу – респект вам и уважуха, если нет – это тоже отлично, наверное, у вас была причина.

3. Естественно, автор не несет ответственности за те знания которые вы тут получили☺ Ну серъезно - делайте добро, вы ж не политики.
4. Вы можете дописывать книгу, изменять в любом месте, но тогда будьте добры, меняйте автора.
5. Если вы поменяли контент книги, даже одну буковку – это уже ваша книга☺ Наслаждайтесь! Вы стали автором!
6. Кстати да, менять можно все кроме этих простых правил.
7. После прочтения книги, задумайтесь какими знаниями можете поделиться вы. Вероятнее всего, вы можете расскать уникальные и интересные вещи, которые поменяют что-то в этом мире! Пользуйтесь! Это ваша суперсила!

Напутственное слово

Читайте.

История изменений

Книга же не бумажная, потому имеет возможность меняться и эволюционировать!

V1.0 – базовый контент книги
V1.1 – «подкрашен» код, немного пофиксаны ошибки, книга выложена в опен сорс

Prototype

Прототип – один из самых простых паттернов, который позволяет нам получить точную копию необходимого объекта. Тоесть использовать как прототип для нового объекта

Когда использовать:

1. У нас есть семейство схожих объектов, разница между которыми только в состоянии их полей.
2. Чтобы создать объект вам надо пройти через огонь, воду и медные трубы. Особенно если этот объект состоит из еще одной кучи объектов, многие из которых для заполнения требуют подгрузку даных из базы, веб сервисов и тому подобных источников. Часто, легче скопировать объект и поменять несколько полей
3. Да и в принципе, нам особо и не важно как создается объект. Ну есть и есть.
4. Нам страшно лень писать иерархию фабрик (читай дальше), которые будут инкапсулировать всю противную работу создания объекта.

Да, и есть еще частое заблуждение (вероятнее всего из названия) – прототип – это архитип, которые никогда не должен использоваться, и служит только для создания себе подобных объектов. Хотя, протип, как архитип – тоже достаточно популярный кейс. Собственно, нам ничего не мешает делать прототипом любой объект который у нас в подчинении.

Поверхностное и глубокое копирование
Тут особенно разницы с .NET нету. Есть указатель, есть значение в куче.

Поверхностное копирование – это просто создание нового указателя на те же самые байты в куче. Тоесть, в результате мы можем получить два объекта, которые указывают на одно и тоже значение.

К примеру создадим объект:

```
@interface Person : NSObject <NSCopying>

@property (nonatomic,weak) NSString *name;
@property (nonatomic,weak) NSString *surname;
@property (nonatomic,weak) NSString *age;

@end
```

А теперь давайте просто создадим два объекта и посмотрим что же получится:

```
Person *firstPerson = [[Person alloc] init];
firstPerson.name = @"Dima";
firstPerson.surname = @"Surname";
Person *secondPerson = firstPerson;
NSLog(@"First Person name  = %@ and surname = %@",
                firstPerson.name, firstPerson.surname);
secondPerson.name = @"Roma";
NSLog(@"Second Person name  = %@ and surname = %@",
                secondPerson.name, secondPerson.surname);
NSLog(@"First Person name  = %@ and surname = %@",
                firstPerson.name, firstPerson.surname);
```

Как видим лог достаточно ожидаемый:

2013-01-21 01:31:20.986 PrototypePattern[1961:11303] First Person name = Dima
2013-01-21 01:31:20.987 PrototypePattern[1961:11303] Second Person name = Roma
2013-01-21 01:31:20.987 PrototypePattern[1961:11303] First Person name = Roma

Заметьте, что хоть и меняли мы имя для secondPerson, но и у firstPerson имя поменялось. Просто потому что мы создали два указателя на один и тот же объект.

Для таких задач, стоит использовать глубокое копирование, которое в Objective-C сделанно в принципе очень похоже как и в .NET:

Для этого надо реализовать протокол NSCopying, и перегрузить:

```
-(id) copyWithZone:(NSZone *)zone;
```

И да, не стоит переживать что мы не реализуем метод “copy”. Он уже есть у класса NSObject. Если вызвать этот метод, и не реализовать copyWithZone – то мы получем ошибку типа NSInvalidArgumentException.

Потому интерфейс нашего объекта теперь будет выглядеть следующим образом:

```
@interface Person : NSObject <NSCopying>

@property (nonatomic,weak) NSString *name;
@property (nonatomic,weak) NSString *surname;
@property (nonatomic,weak) NSString *age;

-(id) copyWithZone:(NSZone *)zone;

@end
```

а сама реализация:

```
@implementation Person

-(id) copyWithZone:(NSZone *)zone
{
    Person *copy = [[self class] allocWithZone:zone];
    copy.name = self.name;
    copy.age = self.age;
    copy.surname = self.surname;
    return copy;
}

@end
```

Теперь немного изменим код нашего тестового приложения:

```
Person *firstPerson = [[Person alloc] init];
firstPerson.name = @"Dima";
firstPerson.surname = @"Surname";
Person *secondPerson = firstPerson.copy;
NSLog(@"First Person name  = %@ and surname = %@",
        firstPerson.name, firstPerson.surname);
secondPerson.name = @"Roma";
NSLog(@"Second Person name  = %@ and surname = %@",
        secondPerson.name, secondPerson.surname);
NSLog(@"First Person name  = %@ and surname = %@",
```

```
            firstPerson.name, firstPerson.surname);
```

Ну и естественно лог:

2013-01-21 01:48:36.538 PrototypePattern[2090:11303] First Person name = Dima and surname = Surname
2013-01-21 01:48:36.539 PrototypePattern[2090:11303] Second Person name = Roma and surname = Surname
2013-01-21 01:48:36.540 PrototypePattern[2090:11303] First Person name = Dima and surname = Surname

Как видим, мы в результате получили два независимых объекта, один из которых сделан по подобию первого.

Код примера.

Factory Method.

Еще один порождающий паттерн, довольно прост и популярен.Паттерн позволяет переложить создание специфических объектов, на наследников родительского класса, потому можно манипулировать объектами на более высоком уровне, не заморачиваясь объект какого класса будет создан. Частенько этот паттерн называют виртуальный конструктор, что по моему мнению более выражает его предназначение.

Когда использовать:

1. Мы не до конца уверены объект какого типа нам необходим.
2. Мы хотим чтобы не родительский объект решал какой тип создавать, а его наследники.

Почему хорошо использовать:

Объекты созданные фабричным методом – схожи, потому как у них один и тот же родительский объект. Потому, если локализировать создание таких объектов, то можно добавлять новые типы, не меняя при это код который использует фабричный метод.

Пример:

Давайте представим, что мы такой неправильный магазин в котором тип товара оценивается по цене:) На данный момент товар есть 2х типов – Игрушки и Одежда.

В чеке мы получаем только цены, и нам надо сохранить объекты которые куплены.

Для начала создадим клас Product. Его реализация нас особо не интересует, хотя он может содержать в себе общие для разных типов товаров методы (сделано для примера, мы их особо не используем):

```
@interface Product : NSObject
@property(nonatomic) int *price;
@property(nonatomic, strong) NSString *name;
-(NSInteger *)getTotalPrice:(int)sum;
-(void)saveObject;
@end



@implementation Product
-(NSInteger *) getTotalPrice:(int)sum
{
    return self.price + sum;}

-(void) saveObject
{
    NSLog(@"I am saving an object in to product database");
}
@end
```

Теперь создадим две реализации этого интерфейса.

Игрушка:

```
@interface Toy : Product
@end

@implementation Toy
-(void) saveObject
{
    NSLog(@"Saving object into Toys database");
}
@end
```

И одежда:

```
@interface Dress : Product
@end


@implementation Dress
-(void) saveObject
{
    NSLog(@"Saving object into Dress database");
}
@end
```

Ну теперь мы практически подошли в плотную к нашему паттерну. Собственно, теперь надо создать метод, который будет по цене определять что же за продукт у нас в чеке, и создавать объект необходимого типа.

```
@interface ProductGenerator : NSObject

-(Product *) getProduct:(int)price;
@end

@implementation ProductGenerator

-(Product *) getProduct:(int)price
{

    if ( price > 0 && price < 100 )
    {
        Toy *p = [[Toy alloc] init];
        return p;
    }

    if (price >= 100)
    {
        Dress *p = [[Dress alloc] init];
        return p;
    }
    return nil;
}
@end
```

Ну вот собственно и все. Теперь просто создадим метод, который будет считать и записывать расходы:

```
-(void) saveExpenses:(int)aPrice
{
    ProductGenerator *pd = [[ProductGenerator alloc] init];

    Product *expense  = [pd getProduct:aPrice];

    [expense saveObject];}
```

Попробуем!

```
    [self saveExpenses:50];
    [self saveExpenses:56];
    [self saveExpenses:79];
    [self saveExpenses:100];
    [self saveExpenses:123];
    [self saveExpenses:51];
```

Лог:

2013-01-23 23:27:54.223 FactoryMethodPattern[8833:11303] Saving object into Toys database
2013-01-23 23:27:54.226 FactoryMethodPattern[8833:11303] Saving object into Toys database
2013-01-23 23:27:54.226 FactoryMethodPattern[8833:11303] Saving object into Toys database
2013-01-23 23:27:54.227 FactoryMethodPattern[8833:11303] Saving object into Dress database
2013-01-23 23:27:54.227 FactoryMethodPattern[8833:11303] Saving object into Dress database
2013-01-23 23:27:54.228 FactoryMethodPattern[8833:11303] Saving object into Toys database

Код примера.

Abstract Factory

Абстрактная фабрика – еще один очень популярный паттерн, который как и в названии так и в реализации слегка похож на фабричный метод.
Итак, что же делает абстрактная фабрика:

Абстрактная фабрик дает простой интерфейс для создания объектов которые принадлежат к тому или иному сеймейству объектов.

Отличия от фабричного метода:

1. Фабричный метод порождает объекты одного и того же типа, фабрике же может создавать независимые объекты
2. Чтобы добавить новый тип объекта – надо поменять интерфейс фабрики, в фабричном методе же легко просто поменять внутренности метода, который ответственный за порождение объектов.

Давайте представим ситуацию: у нас есть две фабрики по производству IPhone и IPad. Одна оригинальная, компании Apple, другая – хижина дядюшки Хуа. И вот, мы хотим производить эти товары: если в страны 3-го мира – то товар от дядюшки, в другие страны – товар любезно предоставлен компанией Apple.

Итак, пускай у нас есть фабрика, которая умеет производить и айпады и айфоны:

```
@interface IPhoneFactory : NSObject

-(GenericIPhone *) getIPhone;
-(GenericIPad *) getIPad;

@end
```

Естественно, нам необходимо реализовать продукты которые фабрика будет производить:

```
@interface GenericIPad : NSObject

@property(nonatomic,weak) NSString *osName;
@property(nonatomic, weak) NSString *productName;
@property(nonatomic, strong) NSNumber *screenSize;

@end

//-----

@interface GenericIPhone : NSObject

@property (nonatomic, weak) NSString *osName;
@property (nonatomic, weak) NSString *productName;

@end
```

Но, продукты немного отличаются.

Пускай у нас есть два типа продуктов, оригинальные Apple и продукты которые произведены трутолюбивым дядюшкой Хуа:

```
@interface AppleIPhone : GenericIPhone
@end
```

```
@implementation AppleIPhone

-(id) init
{
    self = [super init];

    self.productName = @"IPhone";
    self.osName = @"iOS";

    return self;
}
@end

//---------
//Айпад
@interface AppleIPad : GenericIPad
@end

@implementation AppleIPad
-(id) init
{
    self = [super init];
    self.productName = @"IPad";
    self.osName = @"iOS";
    self.screenSize = [[NSNumber alloc] initWithFloat:7.7f];

    return self;
}
@end
```

Дядюшкофоны:

```
@interface ChinaPad : GenericIPad
@end

@implementation ChinaPad

-(id) init
{
    self = [super init];
    self.osName = @"Windows CE";
    self.productName = @"Buan Que Ipado Killa";
    self.screenSize = [[NSNumber alloc] initWithFloat:12.5f];

    return self;
}
@end
//--------

@interface ChinaPhone : GenericIPhone
@end

@implementation ChinaPhone

-(id) init
{
    self = [super init];

    self.osName = @"Android";
    self.productName = @"Chi Huan Hua Phone";

    return self;
}
@end
```

Разные телефоны, конечно же, производятся на различных фабриках, потому мы просто обязанны их создать! Приблизительно так должны выглядеть фабрика Apple:

```
@interface AppleFactory : IPhoneFactory
@end

@implementation AppleFactory

-(GenericIPhone *) getIPhone
{
    AppleIPhone *iphone = [[AppleIPhone alloc] init];
    return iphone;
}

-(GenericIPad *) getIPad
{
    AppleIPad *ipad = [[AppleIPad alloc] init];
    return ipad;
}
@end
```

Конечно же у нашего китайского дядюшки тоже есть своя фабрика:

```
@interface ChinaFactory : IPhoneFactory
@end

@implementation ChinaFactory

-(GenericIPad *) getIPad
{
    ChinaPad *pad = [[ChinaPad alloc] init];
    return pad;
}


-(GenericIPhone *) getIPhone
{
    ChinaPhone *phone  = [[ChinaPhone alloc] init];
    return phone;
}
@end
```

Как видим, фабрики одинаковые, а вот девайсы у них получются разные☺

Ну вот собственно и все, мы приготовили все что надо для доменострации! Теперь, давайте напишем небольшой метод который будет возвращать нам фабрику которую мы хотим (кстати, тут фабричный метод таки будет):

```
-(IPhoneFactory *) getFactory
{
    if (_isThirdWorld)
        return [[ChinaFactory alloc] init];

    return [[AppleFactory alloc] init];
}
```

Теперь, давайте как создадим несколько телефонов:

```
_isThirdWorld = false;
IPhoneFactory *factory = self.getFactory;
GenericIPad *ipad = factory.getIPad;
GenericIPhone *iphone = factory.getIPhone;
NSLog(@"IPad named = %@, osname = %@, screensize = %@",
```

```
                ipad.productName, ipad.osName, ipad.screenSize.stringValue);
    NSLog(@"IPhone named = %@, osname = %@", iphone.productName, iphone.osName);
```

Лог будет выглядить следующим образом:

2013-01-26 20:00:56.663 AbstractFactory[13093:11303] IPad named = Buan Que Ipado Killa, osname = Windows CE, screensize = 12.5
2013-01-26 20:00:56.665 AbstractFactory [13093:11303] IPhone named = Chi Huan Hua Phone, osname = Android

Теперь, просто поменяв значение переменной _isThirdWorld на false, и лог будет совсем другой:

2013-01-26 20:02:21.745 AbstractFactory [13115:11303] IPad named = IPad, osname = iOS, screensize = 7.7
2013-01-26 20:02:21.747 AbstractFactory [13115:11303] IPhone named = IPhone, osname = iOS

Код примера.

Builder

Вот представьте что у нас есть фабрика. Но в отличии от фабрики из предыдущего поста, она умеет создавать только телефоны на базе андроида, и еще при этом различной конфигурации. Тоесть, есть один объект, но при этом его состояние может быть совершенно разным, а еще представьте если его очень трудно создавать, и во время создания этого объекта еще и создается миллион дочерних объектов. Именно в такие моменты, нам очень помогает такой паттерн как строитель.

Когда использовать:

1. Создание сложного объекта
2. Процесс создания объекта тоже очень не тривиальный – к примеру получение данных из базы и манипуляция ими.

Сам паттерн состоит из двух компонент – Bulilder и Director. Builder занимается именно построение объекта, а Director знает какой Builder использовать чтобы выдать необходимый продукт. Приступим!

Пускай у нас есть телефон, который обладает следующими свойствами:

```
@interface AndroidPhone : NSObject

@property (nonatomic, weak) NSString *osVersion;
@property (nonatomic, weak) NSString *name;
@property (nonatomic, weak) NSString *cpuCodeName;
@property (nonatomic, strong) NSNumber *RAMsize;
@property (nonatomic, strong) NSNumber *osVersionCode;
@property (nonatomic, weak) NSString *launcher;
@end
```

Давайте создадим дженерик строителя от которого будут наследоваться конкретные строители:

```
@interface BPAndroidPhoneBuilder : NSObject

@property (nonatomic, strong) AndroidPhone* _phone;

-(void) setOSVersion;
-(void) setName;
-(void) setCPUCodeName;
-(void) setRAMSize;
-(void) setOSVersionCode;
-(void) setLauncher;

-(AndroidPhone *) getPhone;

@end

//---

@implementation BPAndroidPhoneBuilder

-(id) init
{
    self = [super init];

    self._phone = [[AndroidPhone alloc] init];
```

```
    return self;
}

-(AndroidPhone *) getPhone
{
    return self._phone;
}

@end
```

Ну а теперь напишем код для конкретных строителий. К примеру, так бы выглядел строитель для дешевого телефона:

```
@interface LowPricePhoneBuilder : BPAndroidPhoneBuilder
@end

//----
@implementation LowPricePhoneBuilder

-(void) setOSVersion
{
    self._phone.osVersion = @"Android 2.3";
}
-(void) setName
{
    self._phone.name = @"Low price phone!";
}
-(void) setCPUCodeName
{
    self._phone.cpuCodeName = @"Some shitty CPU";
}
-(void) setRAMSize
{
    self._phone.RAMsize = [[NSNumber alloc] initWithInt:256];
}
-(void) setOSVersionCode
{
    self._phone.osVersionCode = [[NSNumber alloc] initWithFloat:3.0f];
}
-(void) setLauncher
{
    self._phone.launcher = @"Hia Tsung!";
}
@end
```

И конечно же строительство дорогого телефона:

```
@interface HighPricePhoneBuilder : BPAndroidPhoneBuilder
@end

//----

@implementation HighPricePhoneBuilder

-(void) setOSVersion
{
    self._phone.osVersion = @"Android 4.1";
}
-(void) setName
{
    self._phone.name = @"High price phone!";
}
-(void) setCPUCodeName
{
    self._phone.cpuCodeName = @"Some shitty but expensive CPU";
```

```
}
-(void) setRAMSize
{
    self._phone.RAMsize = [[NSNumber alloc] initWithInt:1024];
}
-(void) setOSVersionCode
{
    self._phone.osVersionCode = [[NSNumber alloc] initWithFloat:4.1f];
}
-(void) setLauncher
{
    self._phone.launcher = @"Samsung Launcher";
}
@end
```

Кто-то же должен использовать строителей, потому давайте создадим объект который будет с помощью строителей создавать дешевые или дорогие телефоны:

```
@interface FactorySalesMan : NSObject

@property (nonatomic, strong) BPAndroidPhoneBuilder *_builder;

-(void) setBulider:(BPAndroidPhoneBuilder *)aBuilder;
-(AndroidPhone *) getPhone;
-(void) constructPhone;

@end

//----

@implementation FactorySalesMan

-(void)setBulider:(BPAndroidPhoneBuilder *)aBuilder
{
    self._builder = aBuilder;
}

-(AndroidPhone *) getPhone
{
    return self._builder.getPhone;
}

-(void) constructPhone
{
    [self._builder setOSVersion];
    [self._builder setName];
    [self._builder setCPUCodeName];
    [self._builder setRAMSize];
    [self._builder setOSVersionCode];
    [self._builder setLauncher];
}

@end
```

Ну и конечно же куда мы без теста и кода:

```
LowPricePhoneBuilder *_cheapPhoneBuilder = [[LowPricePhoneBuilder alloc] init];
HighPricePhoneBuilder *_expensivePhoneBuilder =
                              [[HighPricePhoneBuilder alloc] init];

FactorySalesMan *_salesMan = [[FactorySalesMan alloc] init];
[_salesMan setBulider:_cheapPhoneBuilder];
[_salesMan constructPhone];
AndroidPhone *_phone = [_salesMan getPhone];

NSLog(@"Phone Name = %@, osVersion = %@, cpu code name = %@, ram size = %@, os
```

```
version code = %@, launcher = %@",
       _phone.name, _phone.osVersion, _phone.cpuCodeName, _phone.RAMsize,
       _phone.osVersionCode, _phone.launcher);

    [_salesMan setBulider:_expensivePhoneBuilder];
    [_salesMan constructPhone];
    _phone = [_salesMan getPhone];
    NSLog(@"Phone Name = %@, osVersion = %@, cpu code name = %@, ram size = %@, os
version code = %@, launcher = %@",
       _phone.name, _phone.osVersion, _phone.cpuCodeName, _phone.RAMsize,
       _phone.osVersionCode, _phone.launcher);
```

Как видим, мы создали различных строителей, и сказав директору (FactorySalesMan) какой строитель мы хотим использовать, мы получаем тот девайс который нам необходим:

Традиционный лог:

2013-01-28 00:38:51.863 BuilderPattern[708:11303] Phone Name = Low price phone!, osVersion = Android 2.3, cpu code name = Some shitty CPU, ram size = 256, os version code = 3, launcher = Hia Tsung!
2013-01-28 00:38:51.867 BuilderPattern[708:11303] Phone Name = High price phone!, osVersion = Android 4.1, cpu code name = Some shitty but expensive CPU, ram size = 1024, os version code = 4.1, launcher = Samsung Launcher

Код примера.

Singleton

//реализация паттерна, которая будет приведена тут, подразумевает использование GDC и ARC.

Кто вообще бы мог подумать, что Singleton такой не самый просто паттерн в iOS? Вернее, что есть столько версий. Собственно, в .NET, помнится, наблюдалась точно такая же штука, но там в основном были просто апдейты к самой простой версии паттерна. Я вообще считаю, что сколько людей - столько и версий синглтона.

Итак, давайте начнем с простого – с описания.

Singleton - это такой объект, который существует в системе только в единственном экземпляре. Очень часто используется для хранения каких-то глобальных переменных, например настроек приложения.

Итак, как и все в Obj-C начнем мы естественно с создания интерфейса:

```
@interface SingletonObject : NSObject

@property (nonatomic, weak) NSString *tempProperty;
+(SingletonObject *) singleton;

@end
```

Как видим, обычный объект с одним свойством и класс методом. Естественно, просто от интерфейса мы не получим всего чего ожидаем:

```
@implementation SingletonObject

+(SingletonObject *) singleton
{
    static SingletonObject *singletonObject = nil;

    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        singletonObject = [[self alloc] init];
    });

    return singletonObject;
}
@end
```

Собственно вот и все:) iOS сам за нас позаботится о том, чтобы создан был только один экземпляр нашего объекта. Тут стоит сделать шаг назад и описать проблему, которая являетя головной болью любого кто более-менее блихко работал с потоками:

1. Представьте что есть 2 потока.
2. И тут каждый, одновременно создает singleton.Вроде бы и должен создасться только один объект, но потому что все происходит в один момент – бывают случаи когда создается два объекта.

“Но ведь можно сделать проверку на nil!” – скажете Вы.

А теперь, представьте более сложную ситуацию: объекта singleton не существует. Два потока хотят его создать одновременно:

1. Поток 1 делает проверку или объект существует. Видит что его нет, и проходит этап проверки.

2. Поток 2 делает проверку на существование объекта, и хоть и поток 1 проверку УЖЕ прошел, но объект ЕЩЕ не существует.

Для решения таких проблем, в .NET использовали locks – блокирование кода, для других потоков, пока он исполняется в каком – либо потоке. Собственно dispatch_once делает тоже самое – он просто синхронный:) Потому, ни один поток не может зайти в этот код, пока он занят.

Собственно, без GCD такое создать тоже можно, тогда наш код бы выглядел следующим образом:

```
+(SingletonObject *) singleton
{
    static SingletonObject *singletonObject = nil;
    @synchronized(self)
    {
            if (singletonObject == nil )
            {
                singletonObject = [[self alloc] init];
            }
    }

    return singletonObject;
}
```

Получится тоже самое. Единственное, что dispatch_once() по документации быстрее:) Ну и семантически более правильнее.

Можно вообще бахнуть по хардкору, и создать макрос:

```
#define DEFINE_SHARED_INSTANCE_USING_BLOCK(block) \
static dispatch_once_t pred = 0; \
__strong static id _sharedObject = nil; \
dispatch_once(&pred, ^{ \
_sharedObject = block(); \
}); \
return _sharedObject; \
```

Тогда сама реализация создания объекта будет выглядеть следующим образом:

```
+(SingletonObject *) singleton
{
    DEFINE_SHARED_INSTANCE_USING_BLOCK(^{
        return [[self alloc] init];
    });
}
```

Ну, а использование простое:

```
[[SingletonObject singleton] setTempProperty:@"Hello 2 You!"];
NSLog(@"%@", [[SingletonObject singleton] tempProperty]);
```

Код примера.

Adapter

Тяжело найти более красочно описание паттерна Адаптер, чем пример из жизни каждого, кто покупал технику из США. Розетка! Вот почему не сделать одинаковую розетку всюду? Но нет, в США розетка с квадратными дырками, в Европе с круглыми, а в некоторых странах вообще треугольные. Следовательно – потому вилки на зарядный устройствах, и других устройствах питания тоже различные.

Представьте, что Вы едете в коммандировку в США. У Вас есть, допустим, ноутбук купленный в Европе – следовательно вилка на проводе от блока питания имеет круглые окончания. Что делать? Покупать зарядку для американского типа розетки? А когда вы вернетесь домой – она будет лежать у Вас мертвым грузом?

Потому, вероятнее всего, Вы приобретете один из адаптеров, которые надеваются на вилку, и которая позволяет Вам использовать старую зарядку и заряжаться от совершенно другой розетки.

Так и с Адаптером – он конвертит интерфейс класса – на такой, который ожидается.

Сам паттерн состоит из трех частей: Цели (target), Адаптера (adapter), и адаптируемого (adaptee).

В нашей с вами проблеме:

1. Target – ноутбук со старой зарядкой
2. Adapter – переходник.
3. Adaptee – розетка с квадртаными дырками.

Имплементация паттерна Адаптер в Objective-C может быть 2 (вероятно даже больше, но я вижу две):

Итак, первая – это простенькая имплементация. Пускай у нас есть объект Bird, который реализует протокол BirdProtocol:

```
@protocol BirdProtocol
-(void) sing;
-(void) fly;
@end


@interface Bird : NSObject <BirdProtocol>

@end
//реализация
@implementation Bird

-(void) sing
{
    NSLog(@"Tew-tew-tew");
}

-(void) fly
{
    NSLog(@"OMG! I am flying!");
}
@end
```

И пускай у нас есть объект Raven, у которого есть свой интерфейс:

```
@interface Raven : NSObject

-(void) flySearchAndDestroy;
-(void) voice;

@end

@implementation Raven

-(void) flySearchAndDestroy
{
    NSLog(@"I am flying and seak for killing!");
}

-(void) voice
{
    NSLog(@"Kaaaar-kaaaaar-kaaaaaaar!");
}
@end
```

Чтобы использовать ворону в методах которые ждут птицу:) стоит создать так называемый адаптер:

```
@interface RavenAdapter : NSObject <BirdProtocol>
{
    @private Raven *_raven;
}

@property (nonatomic, strong) Raven *raven;

-(id) initWithRaven:(Raven*) adaptee;
@end

//ну и сама реализация. Как видим мы завернули ворона в обертку протоколом
BirdProtocol
@implementation RavenAdapter

@synthesize raven = _raven;

-(id) initWithRaven:(Raven*) adaptee
{
    self = [super self];
    _raven = adaptee;
    return self;
}

-(void) sing
{
    [_raven voice ];
}

-(void) fly
{
    [_raven flySearchAndDestroy];
}

@end
```

Ну и конечно же тест:

```
-(void) makeTheBirdTest:(id<BirdProtocol>)aBird
{
    [aBird fly];
    [aBird sing];
```

```
}
//и тестовый код:

Bird *simpleBird = [[Bird alloc] init];

Raven *simpleRaven = [[Raven alloc] init];

RavenAdapter *ravenAdapter = [[RavenAdapter alloc] initWithRaven:simpleRaven];

[self makeTheBirdTest:simpleBird];
[self makeTheBirdTest:ravenAdapter];
```

Результат можно легко увидеть в логе:

2013-02-03 15:43:14.447 AdapterPattern[5985:11303] OMG! I am flying!
2013-02-03 15:43:14.449 AdapterPattern[5985:11303] Tew-tew-tew
2013-02-03 15:43:14.449 AdapterPattern[5985:11303] I am flying and seak for killing!
2013-02-03 15:43:14.450 AdapterPattern[5985:11303] Kaaaar-kaaaaar-kaaaaaaar!

Теперь более сложная реализация, которая все еще зависит от протоколов, но уже использует делегацию. Вернемся к нашему несчастному ноутбуку и зарядке: Допустим у нас есть базовый клас Charger:

```
@interface Charger : NSObject
-(void) charge;
@end

@implementation Charger

-(void) charge
{
    NSLog(@"C'mon I am charging");
}
@end
```

И есть протокол для европейской зардяки:

```
@protocol EuropeanNotebookChargerDelegate
-(void) chargetNotebookRoundHoles:(Charger *)charger;
@end
```

Если сделать просто реализацию – то получится тоже самое, что и в прошлом примере:) Потому, давайте добавим делегат:

```
@interface EuropeanNotebookCharger : Charger <EuropeanNotebookChargerDelegate>
{
@private id<EuropeanNotebookChargerDelegate> _delegate;
}

@property (nonatomic, strong) id<EuropeanNotebookChargerDelegate> delegate;
@end

//реализация
@implementation EuropeanNotebookCharger

@synthesize delegate = _delegate;

-(id) init
{
    self = [super self];
    self.delegate = self;
    return self;
}
```

```
-(void) charge
{
    [_delegate chargetNotebookRoundHoles:self];
    [super charge];
}

-(void) chargetNotebookRoundHoles:(Charger *)charger
{
    //and yeah you can do smth with charger.
    NSLog(@"Charging with 220 and round holes!");
}

@end
```

Как видим, у нашего класа есть свойство которое реализует тип EuropeanNotebookChargerDelegate. Так как, наш клас этот протокол реализует, он может свойсту присвоить себя, потому когда происходит вызов:

```
   [_delegate chargetNotebookRoundHoles:self];
```

просто вызывается свой же метод. Вы увидите дальше, для чего это сделано. Теперь, давайте глянем что ж за зверь такой – американская зарядка:

```
@interface USANotebookCharger : NSObject
-(void) chargeNotebookRectHoles:(Charger *) charger;
@end

@implementation USANotebookCharger

-(void) chargeNotebookRectHoles:(Charger *) charger;
{
    NSLog(@"Charge Notebook Rect Holes");
}
@end
```

Как видим, в американской зарядке совсем другой метод и мировозрение. Давайте, создадим адаптер для зарядки:

```
@interface USANotebookEuropeanAdapter : Charger <EuropeanNotebookChargerDelegate>

@property (nonatomic, strong) USANotebookCharger *usaCharger;

-(id) initWithUSANotebookCharger:(USANotebookCharger *) charger;

-(void) charge;
@end

@implementation USANotebookEuropeanAdapter

-(id) initWithUSANotebookCharger:(USANotebookCharger *) charger
{
    self = [super init];
    self.usaCharger = charger;

    return self;
}

-(void) chargetNotebookRoundHoles:(Charger *) charger
{
    [self.usaCharger chargeNotebookRectHoles:charger];
}

-(void) charge
{
```

```
    EuropeanNotebookCharger *euroCharge = [[EuropeanNotebookCharger alloc] init];
    euroCharge.delegate = self;
    [euroCharge charge];
}
@end
```

Как видим, наш адптер реализует интерфейс EuropeanNotebookChargerDelegate и его метод chargetNotebookRoundHoles. Потому, когда вызывается метод charge – на самом деле создается тип европейской зарядки, ей присвается наш адаптер как делегат, и вызывается ее метод charge. Так как делегатом присвоен наш адаптер, при вызове метода chargetNotebookRoundHoles, будет вызыван этот метод нашего адаптера, который в свою очередь вызывает метод зардяки США:)

Давайте посмотрим тест код и вывод лога:

```
//сам метод
-(void) makeTheNotebookCharge:(Charger *) aCharger
{
    [aCharger charge];
}

//тест код
EuropeanNotebookCharger *euroCharger = [[EuropeanNotebookCharger alloc] init];

[self makeTheNotebookCharge:euroCharger];

USANotebookCharger *charger = [[USANotebookCharger alloc] init];
USANotebookEuropeanAdapter *adapter =
         [[USANotebookEuropeanAdapter alloc] initWithUSANotebookCharger:charger];
[self makeTheNotebookCharge:adapter];
```

Лог нам выведет:

2013-02-03 15:57:42.624 AdapterPattern[6179:11303] Charging with 220 and round holes!
2013-02-03 15:57:42.626 AdapterPattern[6179:11303] C'mon I am charging
2013-02-03 15:57:42.626 AdapterPattern[6179:11303] Charge Notebook Rect Holes
2013-02-03 15:57:42.627 AdapterPattern[6179:11303] C'mon I am charging

Код примера.

Bridge

Представьте себе, что у нас есть что-то однотипное, к примеру у нас есть телефон и куча наушников. Если бы у каждого телефона был свой разъем, то мы могли бы пользоваться только одним типом наушников. Но Бог миловал! Собственно таже штука и с наушникам. Они могут выдавать различный звук, иметь различные дополнительные функции, но основная их цель – просто звучание:) И хорошо, что во многих случаях штекер у них одинаковый (я не говорю про различные студийные наушники:)).

Собственно, Мост (Bridge) позволяет разделить абстракцию от реализации, так чтобы реализация в любой момент могла быть поменяна, не меняя при этом абстракции.

Когда использовать?

1. Вам совершенно не нужна связь между абстракцией и реализацией.
2. Собственно, как абстракцию так и имплементацию могут наследовать независимо.
3. Вы не хотите чтобы изменения в реализации имело влияния на клиентский код.

Давайте создадим теперь базовую абстракцию наушников:

```
@interface BaseHeadphones : NSObject

-(void) playSimpleSound;
-(void) playBassSound;

@end
```

И теперь два элемента – дорогие наушники и дешевые:)

```
//Наушники обычные - китайские
@interface CheapHeadphones : BaseHeadphones

@end

@implementation CheapHeadphones

-(void) playSimpleSound
{
    NSLog(@"beep - beep - bhhhrhrhrep");
}

-(void) playBassSound
{
    NSLog(@"puf - puf - pufhrrr");
}

@end
```

```
//наушники дорогие, тоже китайские
@interface ExpensiveHeadphones : BaseHeadphones
@end

@implementation ExpensiveHeadphones
```

```
-(void) playSimpleSound
{
    NSLog(@"Beep-Beep-Beep Taram - Rararam");
}

-(void) playBassSound
{
    NSLog(@"Bam-Bam-Bam");
}

@end
```

И собственно плеер, через который мы будем слушать музыку:

```
@interface MusicPlayer : NSObject

@property (nonatomic, strong) BaseHeadphones *headPhones;

-(void) playMusic;

@end

@implementation MusicPlayer

-(void) playMusic
{
    [self.headPhones playBassSound];
    [self.headPhones playBassSound];
    [self.headPhones playSimpleSound];
    [self.headPhones playSimpleSound];

}
@end
```

Как видите, одно из свойств нашего плеера – наушники. Их можно подменять в любой момент, так как свойство того же типа, от которого наши дешевые и дорогие наушники наследуются.
Тест!

```
MusicPlayer *p = [[MusicPlayer alloc] init];
CheapHeadphones *ch = [[CheapHeadphones alloc] init];
ExpensiveHeadphones *ep = [[ExpensiveHeadphones alloc] init];
p.headPhones = ch;
[p playMusic];
p.headPhones = ep;
[p playMusic];
```

И конечно же log:

2013-02-06 23:03:52.378 BridgePattern[3397:c07] puf – puf – pufhrrr
2013-02-06 23:03:52.379 BridgePattern[3397:c07] puf – puf – pufhrrr
2013-02-06 23:03:52.380 BridgePattern[3397:c07] beep – beep – bhhhrhrhrep
2013-02-06 23:03:52.380 BridgePattern[3397:c07] beep – beep – bhhhrhrhrep
2013-02-06 23:03:52.380 BridgePattern[3397:c07] Bam-Bam-Bam
2013-02-06 23:03:52.381 BridgePattern[3397:c07] Bam-Bam-Bam
2013-02-06 23:03:52.381 BridgePattern[3397:c07] Beep-Beep-Beep Taram – Rararam
2013-02-06 23:03:52.381 BridgePattern[3397:c07] Beep-Beep-Beep Taram – Rararam

Код примера.

Facade

Многие сложные системы состоят из огромной кучи компонент. Так же и в жизни, очень часто для совершения одного основного действия, мы должны выполнить много маленьких.

К примеру, чтобы пойти в кино нам надо:

1. Посмотреть расписание фильмов, выбрать фильм, посмотреть когда есть сеансы, посмотреть когда у нас есть время.
2. Необходимо купить билет, для этого ввести номер карточки, секретный код, дождаться снятия денег, распечатать билет.
3. Приехать в кинотеатр, запарковать машину, купить попкорн, найти места, смотреть.

И все это для того, чтобы просто посмотреть фильм, который нам, очень вероятно, не понравится.

Или же возьмем пример Amazon – покупка с одного клика – как много систем задействовано в операции покупки? И проверка Вашей карточки, и проверка Вашего адреса, проверка товара на складе, проверка или возможна доставка даного товара в даную точку мира… В результате очень много действий которые происходят всего по одному клику.

Для таких вот процесов был изобретен паттерн – Фасад (Facade) который предоставляет унифицированный интерфейса к большому количеству интерфейсов системы, в следствии чего систему стает гораздо проще в использовании.

Давайте, попробуем создать систему которая нас переносит в другую точку мира с одного нажатия кнопки! С начала нам нужна система которая проложит путь от нашего места перебывания в место назначения:

```
@interface PathFinder : NSObject

-(void) findCurrentLocation;
-(void) findLocationToTravel:(NSString *) location;
-(void) makeARoute;

@end

@implementation PathFinder

-(void) findCurrentLocation
{
    NSLog(@"Finding your location. Hmmm, here you are!");
}

-(void) findLocationToTravel:(NSString *)location
{
    NSLog(@"So you wanna travell to %@", location);
}

-(void) makeARoute
{
    NSLog(@"Okay, to travell to this location we are using google maps....");
    //looking for path in google maps.
}
```

```
@end
```

Естественно нам необходима сама система заказа транспорта и собственно путешествия:

```
@interface TravellEngine : NSObject

-(void) findTransport;
-(void) orderTransport;
-(void) travel;

@end

@implementation TravellEngine

-(void) findTransport
{
    NSLog(@"Okay, to travell there you will probabply need dragon!Arghhhhh");
}

-(void) orderTransport
{
    NSLog(@"Maaaam, can I order a dragon?... Yes... Yes, green one... Yes, with
fire!... No, not a dragon of death... Thank you!");
}

-(void) travel
{
    NSLog(@"Maaan, you are flying on dragon!");
}

@end
```

Ну и какие же путешествия без билетика:

```
@interface TicketPrinitingSystem : NSObject

-(void) createTicket;
-(void) printingTicket;

@end

@implementation TicketPrinitingSystem

-(void) createTicket
{
    NSLog(@"Connecting to our ticketing system...");
}

-(void) printingTicket
{
    NSLog(@"Hmmm, ticket for travelling on the green dragon.Interesting...");
}

@end
```

А теперь, давайте создадим единый доступ ко всем этим системам:

```
@interface TravellSystemFacade : NSObject

-(void) travellTo:(NSString *)location;

@end

@implementation TravellSystemFacade
```

```
-(void) travellTo:(NSString *)location
{

    PathFinder *pf = [[PathFinder alloc] init];
    TravellEngine *te = [[TravellEngine alloc] init];
    TicketPrinitingSystem *tp = [[TicketPrinitingSystem alloc] init];


    [pf findCurrentLocation];
    [pf findLocationToTravel:location];
    [pf makeARoute];

    [te findTransport];
    [te orderTransport];
    [tp createTicket];
    [tp printingTicket];
    [te travel];
}

@end
```

Как видим, наш фасад знает все про все системы, потому в одном методе он берет и транспортирует нас куда следует. Код теста элементарен:

```
TravellSystemFacade *facade = [[TravellSystemFacade alloc] init];
[facade travellTo:@"Lviv"];
```

Давайте посмотрим лог:

2013-02-09 17:46:28.442 FacadePattern[2410:c07] Finding your location. Hmmm, here you are!
2013-02-09 17:46:28.444 FacadePattern[2410:c07] So you wanna travell to Lviv
2013-02-09 17:46:28.445 FacadePattern[2410:c07] Okay, to travell to this location we are using google maps….
2013-02-09 17:46:28.446 FacadePattern[2410:c07] Okay, to travell there you will probabply need dragon!Arghhhhh
2013-02-09 17:46:28.446 FacadePattern[2410:c07] Maaaam, can I order a dragon?… Yes… Yes, green one… Yes, with fire!… No, not a dragon of death… Thank you!
2013-02-09 17:46:28.447 FacadePattern[2410:c07] Connecting to our ticketing system…
2013-02-09 17:46:28.447 FacadePattern[2410:c07] Hmmm, ticket for travelling on the green dragon.Interesting…
2013-02-09 17:46:28.448 FacadePattern[2410:c07] Maaan, you are flying on dragon!

Код примера.

Mediator

Медиатор – паттерн которые определяет внутри себя объект, в котором реализуется взаимодействие между некоторым количеством объектов. При этом эти объекты, могут даже не знать про существования друг друга, потому взаимодействий реализованых в медиаторе может быть огромное количество.

Когда стоит использовать:

1. Когда у вас есть некоторое количество объектов, и очень тяжело реализовать взаимодействие между ними. Яркий пример – умный дом. Онозначно есть неколько датчиков, и несколько устройств. К примеру, датчик температуры следит за тем какая на даный момент температура, а кондционер умеет охлаждать воздух. При чем кондиционер, не обязательно что знает про существования датичка температуры. Есть центральный компьютер, который получает сигналы от каждого из устройств и понимает, что делать в том или ином случает.
2. Тяжело переиспользовать объект, так как он взаимодействует и коммуницирует с огромным количеством других объектов.
3. Логика взаимодействия должна легко настраиваться и расширяться.

Собственно, пример медиатора даже писать безсмысленно, потому как это любой контроллер который мы используем во время нашей разработки. Посудите сами – на view есть очень много контролов, и все правила взаимодействия мы прописываем в контроллере. Элементарно.

И все же пример не будет лишним Давайте все же создадим пример который показывает создание аля умного дома.

Пускай у нас есть оборудование которое может взаимодействоать с нашим умным домом:

```
@class CentrallProcessor;

@interface SmartHousePart : NSObject
{
@private CentrallProcessor *_processor;
}

-(id) initWithCore:(CentrallProcessor *) processor;

-(void) numbersChanged;

@end

@implementation SmartHousePart

-(id) initWithCore:(CentrallProcessor *)processor
{
    self = [super init];
    _processor = processor;

    return self;
}

-(void) numbersChanged
{
    [_processor valueChanged:self];
}
```

```
@end
```

Теперь, создадим сердце нашего умного дома:

```
@interface CentrallProcessor : NSObject

@property (nonatomic, weak) Thermometer *_thermometer;
@property (nonatomic, weak) ConditioningSystem *_condSystem;

-(void) valueChanged:(SmartHousePart *) aPart;

@end

@implementation CentrallProcessor

-(void) valueChanged:(SmartHousePart *) aPart
{
    NSLog(@"Value changed! We need to do smth!");

    //detecting that changes are done by thermometer
    if ( [aPart isKindOfClass:[Thermometer class]])
    {
        NSLog(@"Oh, the change is temperature");

        [[self _condSystem] startCondition];
    }
}

@end
```

Тут очень интересный момент – класс CentrallProcessor должен знать про существование SmartHousePart, собственно и SmartHousePart должен знать про существование CentrallProcessor. Если все сделать простым import – то проект не скомпилируется.Потому, в SmartHousePart.h мы добавили @class CentrallProcessor; для того чтобы XCode знал что за объект будет использован, и при этом не импортил файл заголовков CentrallProcessor.

Заголовки же мы импортим в SmartHousePart.m.

Дальше в классе CentrallPart в методе valueChanged мы определяем с какой деталью и что произошло, чтобы адекватно среагировать. В нашем примере – изменение температуры приводт к тому что мы включаем кондиционер.

А вот, и код термометра и кондиционера:

```
@interface Thermometer : SmartHousePart

@property (nonatomic) int temperature;

-(void) temperatureChanged:(int) temperature;

@end

@implementation Thermometer

-(void)temperatureChanged:(int)temperature
{
    self.temperature = temperature;
    [self numbersChanged];
}

@end
```

```
@interface ConditioningSystem : SmartHousePart

-(void) startCondition;

@end

@implementation ConditioningSystem

-(void) startCondition
{
    NSLog(@"Conditioning...");
}

@end
```

Как видим в результате у нас есть два объекта, которые друг про друга не в курсе, и все таки они взаимодействуют друг с другом посредством нашего медиатора CentrallProcessor.

Код для тестинга:

```
CentrallProcessor *proccessor = [[CentrallProcessor alloc] init];

Thermometer *therm = [[Thermometer alloc] initWithCore:proccessor];
ConditioningSystem *condSystem =
                [[ConditioningSystem alloc] initWithCore:proccessor];

proccessor._condSystem = condSystem;
proccessor._thermometer = therm;
[therm temperatureChanged:45];
```

И конечно же лог:

2013-02-12 18:45:06.790 MediatorPattern[8809:c07] Value changed! We need to do smth!
2013-02-12 18:45:06.793 MediatorPattern[8809:c07] Oh, the change is temperature
2013-02-12 18:45:06.793 MediatorPattern[8809:c07] Conditioning…

Код примера.

Observer

Что такое паттерн Observer? Вот вы когда нибудь подписывались на газету? Вы подписываетесь, и каждый раз когда выходит новый номер газеты вы получаете ее к своему дому. Вы никуда не ходите, просто даете информацию про себя, и организация которая выпускает газету сама знает куда и какую газету отнесту. Второе название этого паттерна – **Publish – Subscriber.**

Как описывает этот паттерн наша любимая GoF книга – Observer определяет одно-ко-многим отношение между объектами, и если изменения происходят в объекте – все подписанные на него объекты тут же узнают про это изменение.

Идея проста, объект который мы называем Subject – дает возможность другим объектам, которые реализуют интерфейс Observer, подписываться и отписываться от изменений происходящик в Subject. Когда изменение происходит – всем заинетерсованным объектам высылается сообщение, что изменение произошло. В нашем случае – Subject – это издатель газеты, Observer это мы с вами – те кто подписывается на газету, ну и собсвтенно изменение – это выход новой газеты, а оповещение – отправка газеты всем кто подписался.

Когда используется паттерн:

1. Когда Вам необходимо сообщить всеи объектаь подписанным на изменения, что изменение произошло, при этом вы не знаете типы этих объектов.
2. Изменения в одном объекте, требуют чтоб состояние изменилось в других объектах, при чем количество объектов может быть разное.

Реализация этого паттерна возможно двумя способами:

1. Notification

Notificaiton – механизм использования возможностей NotificationCenter самой операционной системы. Использование NSNotificationCenter позволяет объектам комуницировать, даже не зная друг про друга. Это очень удобно использовать когда у вас в паралельном потоке пришел push-notification, или же обновилась база, и вы хотите дать об этом знать активному на даный момент View.

Чтобы послать такое сообщение стоит использовать конструкцию типа:

```
NSNotification *broadCastMessage = [NSNotification
                                    notificationWithName:@"broadcastMessage"
                                    object:self];
NSNotificationCenter * notificationCenter =
[NSNotificationCenter defaultCenter];
```

Как видим мы создали объект типа NSNotification в котором мы указали имя нашего оповещения: “broadcastMessage”, и собственно сообщили о нем через NotificationCenter.

Чтобы подписаться на событие в объекте который заинтересован в изменении стоит использовать следующую конструкцию:

```
NSNotificationCenter * notificationCenter =
```

```
[NSNotificationCenter defaultCenter];
[notificationCenter addObserver:self
                        selector:@selector(update:)
                            name:@"broadcastMessage" object:nil];
```

Собственно, из кода все более-менее понятно: мы подписываемся на событие, и вызывается метод который задан в своействе selector.

2. Стандартный метод.

Стандартный метод, это реализация этого паттерна тогда, когда Subject знает про всех подписчиков, но при этом не знает их типа. Давайте начнем с того, что создадим протоколы для Subject и Observer:

```
@protocol StandardObserver <NSObject>

-(void) valueChanged:(NSString *)valueName newValue:(NSString *) newValue;

@end

@protocol StandardSubject <NSObject>

-(void) addObserver:(id<StandardObserver>) observer;
-(void) removeObserver:(id<StandardObserver>) observer;
-(void) notifyObjects;

@end
```

Теперь, давайте создадим реализацию Subject:

```
@interface StandardSubjectImplementation : NSObject <StandardSubject>
{
    @private NSString *_valueName;
    @private NSString *_newValue;
}

@property (nonatomic, strong) NSMutableSet *observerCollection;

-(void)changeValue:(NSString *)valueName andValue:(NSString *) newValue;

@end

@implementation StandardSubjectImplementation

-(NSMutableSet *) observerCollection
{
    if (_observerCollection == nil)
        _observerCollection = [[NSMutableSet alloc] init];

    return _observerCollection;
}


-(void) addObserver:(id<StandardObserver>)observer
{
    [self.observerCollection addObject:observer];
}

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

-(void) notifyObjects
{
```

```
        for (id<StandardObserver> observer in self.observerCollection) {
            [observer valueChanged: _valueName newValue:_newValue];
        }
    }

    -(void)changeValue:(NSString *)valueName andValue:(NSString *) newValue
    {
        _newValue = newValue;
        _valueName = valueName;
        [self notifyObjects];
    }

    @end
```

Ну и куда же без обсерверов:

```
    @interface SomeSubscriber : NSObject <StandardObserver>
    @end

    @implementation SomeSubscriber
    -(void) valueChanged:(NSString *)valueName newValue:(NSString *)newValue
    {
        NSLog(@"And some subscriber tells: Hmm, value %@ changed to %@", valueName,
    newValue);

    }
    @end

    //и второй подписчик:
    @interface OtherSubscriber : NSObject <StandardObserver>

    @end

    @implementation OtherSubscriber

    -(void) valueChanged:(NSString *)valueName newValue:(NSString *)newValue
    {
        NSLog(@"And some another subscriber tells: Hmm, value %@ changed to %@",
    valueName, newValue);
    }

    @end
```

Собственно – все:) теперь демо-код:

```
    StandardSubjectImplementation *subj = [[StandardSubjectImplementation alloc]
init];
    SomeSubscriber *someSubscriber = [[SomeSubscriber alloc] init];
    OtherSubscriber *otherSubscriber = [[OtherSubscriber alloc] init];

    [subj addObserver:someSubscriber];
    [subj addObserver: otherSubscriber];

    [subj changeValue:@"strange value" andValue:@"newValue"];
```

И естественно log:

2013-02-16 17:31:43.176 ObserverPattern[24332:c07] And some subscriber tells: Hmm, value strange value changed to newValue
2013-02-16 17:31:43.177 ObserverPattern[24332:c07] And some another subscriber tells: Hmm, value strange value changed to newValue

Ну и конечно же без использования KVO описание паттерна выглядило бы неполным.

Одна из моих самых любимых особенностей Obj-C – это key-value coding. Про него очень клево описанно в официальной документации, но если объяснять на валенках – то это возможность изменять значения свойств объекта с помощью строчек – которые указывают именно само название свойства. Как пример такие две конструкции идентичны:

```
kvoSubj.changeableProperty = @"new value";

[kvoSubj setValue:@"new value" forKey:@"changeableProperty"];
```

Такая гибкость дает нам доступ к еще одной очень замечательной возможности, которая называется key-value observing. Опять же, все круто описанно в документации, но если объяснять на валенках:) то это возможность подписаться на изменение любого свойства, у любого объекта который KV compilant, любым объектом. На самом деле легче объяснить на примере.

Давайте создадим клас с одним свойством, которое мы будем менять:

```
@interface KVOSubject : NSObject

@property (nonatomic, strong) NSString *changeableProperty;

@end

@implementation KVOSubject

@end
```

И создадим объект который будет слушать изменение свойства changeableProperty:

```
@interface KVOObserver : NSObject
@end

@implementation KVOObserver

-(void) observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
change:(NSDictionary *)change context:(void *)context
{
    NSLog(@"KVO: Value changed;");
}

@end
```

Как видим, этот класс реализует только один метод: observeValueForKeyPath. Этот метод будет вызван когда поменяется свойство объекта за которым мы наблюдаем.

Теперь тест:

```
KVOSubject *kvoSubj = [[KVOSubject alloc] init];
KVOObserver *kvoObserver = [[KVOObserver alloc] init];

[kvoSubj addObserver:kvoObserver forKeyPath:@"changeableProperty"
             options:NSKeyValueObservingOptionNew context:nil];

[kvoSubj setValue:@"new value" forKey:@"changeableProperty"];

//because kvoSubj will be deallocated after this functions ends we need to remove
observer information.
```

```
    [kvoSubj removeObserver:kvoObserver forKeyPath:@"changeableProperty"];
```

Как видно из примера, мы для объекта за которым мы наблюдаем, выполняем функцию addObserver – где устанавливаем кто будет наблюлать за изменениями, за изменениями какого свойства мы будем наблюдать и остальные опции. Дальше меняем значение свойства, и так как мы все это проделываем на нажатие кнопки – в конце мы удаляем наблюдателя с нашего объекта, что бы память не текла. Лог говорит сам за себя:

2013-02-17 11:41:58.051 ObserverPattern[26689:c07] KVO: Value changed;
2013-02-17 11:41:58.052 ObserverPattern[26689:c07] KVO: Value changed;

Код примера.

Composite

Вы задумывались как много в нашей жизни древовидных структур? Начиная собственно от самих деревьев, и заканчивая структурами компаний. Да даже, ладно компаний – целые страны используют древовидные структуры, чтобы построить власть.

Во главе компании или страны частенько стоит один человек, у него есть с 10 помошников. У них тоже есть с десяток помошников, и так далее… Если нарисовать их отношения на листе бумаги – увидим дерево!

Очень часто, и мы используем такие типы даных, которые лучше всего храняться в древовидной структуре. Возьмите к примеру стандартный UI: в начале у нас есть View, в нем находяться Subview, в которых могут быть или другие View, или все такие компоненты. Та же самая структура:)

Именно для хранения таких типов данных, а вернее их организации, используется паттерн – Композит.

Когда использовать такой паттерн?

Собственно когда вы работаете с древовидными типами данных, или хотите отобразить иерархию даных таким образом.

Давайте разберем более детально структуру:

В начале всегда есть контейнер в котором находятся все остальные объекта. Контейнер может хранить как другие контейнеры – ветки нашего дерева, так и объекты которые контейнерами не являюстся – листья нашего дерева. Не сложно представить, что контейнеры второго уровня могут хранить как другие контейнеры, так и листья.

Давайте пример!

Начнем с создания протокола для наших объектов:

```
@protocol CompositeObjectProtocol <NSObject>

-(NSString *) getData;
-(void) addComponent:(id<CompositeObjectProtocol>)aComponent;

@end
```

Создадим объект листа:

```
@interface LeafObject : NSObject <CompositeObjectProtocol>

@property (nonatomic, strong) NSString *leafValue;

@end

@implementation LeafObject

-(NSString *) getData
{
    return [[NSString alloc] initWithFormat:@"<%@/>",self.leafValue ];
}
```

```
-(void) addComponent:(id)aComponent
{
    NSLog(@"Can't add component. Sorry, man");
}

@end
```

Как видим наш объект не может добавлять себе детей (ну он же не контейнер:)), и может возвращать свое значение с помощью метода getData.

Теперь нам очень необходим конейтнер:

```
@interface Container : NSObject <CompositeObjectProtocol>

@property (nonatomic,strong) NSMutableArray *components;

@end

@implementation Container

-(NSMutableArray *) components
{
    if (_components == nil)
        _components = [[NSMutableArray alloc] init];

    return _components;
}

-(void) addComponent:(id<CompositeObjectProtocol>)aComponent
{
    [self.components addObject:aComponent];
}

-(NSString *) getData
{
    NSMutableString *valueToReturn = [[NSMutableString alloc] init];
    [valueToReturn appendString:@"<ContainerValues>"];

    for ( id<CompositeObjectProtocol> object in _components)
    {
        [valueToReturn appendString:[object getData]];
    }
    [valueToReturn appendString:@"</ContainerValues>"];

    return valueToReturn;

}

@end
```

Как видим, наш контейнер может добавлять в себя детей, которые могут быть как типа Container так и типа LeafObject. Метод getData же, бегает по всем объектам в массиве components, и вызывает тот же самый метод в детях. Вот собственно и все.

Теперь, конечно же пример:

```
Container *rootContainer = [[Container alloc] init];
LeafObject *object = [[LeafObject alloc] init];
object.leafValue = @"level1 value";
[rootContainer addComponent:object];

Container *firstLevelContainer1 = [[Container alloc] init];
LeafObject *object2 = [[LeafObject alloc] init];
```

```
object2.leafValue = @"level2 value";
[firstLevelContainer1 addComponent:object2];
[rootContainer addComponent:firstLevelContainer1];

Container *firstLevelContainer2 = [[Container alloc] init];
LeafObject *object3 = [[LeafObject alloc] init];
object3.leafValue = @"level2 value 2";
[firstLevelContainer2 addComponent:object3];
[rootContainer addComponent:firstLevelContainer2];


NSLog(@"%@", rootContainer.getData);
```

И конечно же лог:

```
2013-02-17 13:04:09.470 CompositePattern[27392:c07]
<ContainerValues>
      <level1 value/>
      <ContainerValues>
            <level2 value/>
      </ContainerValues>
      <ContainerValues>
            <level2 value 2/>
      </ContainerValues>
</ContainerValues>
```

Код примера.

Iterator

Я задумался о том, какой бы пример из жизни привести, чтобы показать пример как работает паттерн итератор, и оказалось что это не такой простое задание. И как показывает практика, самый простой пример – это обычная вендинг машина. (сам пример взят из книги Pro Objective-C Design Patterns for iOS.) . У нас есть контейнер, который разделен на секции, каждая из которых содержит определенный вид товара, к примеру набор бутылок Coca-Cola. Когда мы заказываем товар, то нам выпадет следующий из колекции. Образно говоря, команда cocaColaCollection.next. Две независимые части – контейнер и итератор.

Паттерн итератор позволяет последовательно обращаться к колекции объектов,не особо вникая что же это за коллекция.

Разделяют два вида итераторов – внутрений и внешний. Как видно из названия, внешний иетратор – итератор про который знает клиент, и собственно он сам(клиент) скармливает коллекцию по которой надо бегать итатору. Внутрений иетратор – это внутрения кухня самой колекции, которая предоставляет интерфейс клиенту для итерирования.

При внешнем итераторе, клиенту надо:

1. Вообще знать про существование итератора, хоть это и дает больше контроля.

2. Создавать и управлять итератором.

3. Можно использовать различные итераторы, для различных алгоритмов итерации.

При внутренем:

1. Клиенту совершенно не известно существование итератора. Он просто дергает интерфейс коллекции.

2. Коллекция сама создает и управляет итератором

3. Коллекция может менять различные итераторы, при этом не трогая код клиента.

Когда использовать итератор:

1. Вам неообходимо доступиться к объектам коллекции, без того чтобу щупать внутренности коллекции.

2. Вам надо обходить объекты коллекции различными способами (вспомните Композит – у вас коллекция может быть древовидной)

3. Вам необходимо дать унифицированный интерфейс для различных подходов итерации.

Самый просто пример внешнего итератора, это использование класса NSEnumerator:

```
NSEnumerator *enumerator = [internallArrayCollection objectEnumerator];
NSString* element;

while (element = [enumerator nextObject]) {
    NSLog(@"%@",element);
   }
```

Как видим, мы просто вызываем у объекта internallArrayCollection метод objectEnumerator – и получаем необходимый нам иетратор. Вообще можно не заморачиваться, и использовать обычный цикл for:

```
for (NSString *element in internallArrayCollection)
{
    NSLog(@"%@",element);
}
```

Я не уверен что смогу правильно объяснить разницу между созданием итератора и цикла for (если она есть), потому этот момент будет упущен.

Одним из примеров реализации внешнего итератора – может быть итерация с помощью блоков:

```
[internallArrayCollection enumerateObjectsUsingBlock:
 ^(id obj, NSUInteger idx, BOOL *stop) {
     if([obj localizedCaseInsensitiveCompare:@"Dima"] == NSOrderedSame)
     {
         NSLog(@"Dima has been found!");
         *stop = YES;
     }
 }];
```

Радость этого метода в том, что сам алгоритм итерации может написать другой программист, вам же необходимо будет только использовать блок написанный этим программистом. Все выглядит приблизительно так:

```
//создание блока поиска Dima в массиве строчек
void (^simpleDimaSearchBlock)(id, NSUInteger, BOOL*) =
^(id obj, NSUInteger idx, BOOL *stop){
    if([obj localizedCaseInsensitiveCompare:@"Dima"] == NSOrderedSame)
    {
        NSLog(@"Dima has been found!");
        *stop = YES;
    }
};


[internallArrayCollection enumerateObjectsUsingBlock:simpleDimaSearchBlock];
```

Приятно же:)

Теперь давайте создадим свой итератор, а то и два:) Пускай у нас будет коллекция товаров, одни их них будут сломаны, другие же – целыми. Создадим два итератора, которые будут бегать по разным типам товаров. Итак, для начала сам клас товаров:

```
@interface ItemInShop : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic) BOOL isBroken;

-(id) initWithArgs:(NSString *)aName andQaulity:(BOOL)isBroken;
```

```
@end

@implementation ItemInShop

-(id) initWithArgs:(NSString *)aName andQaulity:(BOOL)isBroken
{
    self = [super init];
    self.name = aName;
    self.isBroken = isBroken;
    return self;
}

@end
```

Как видим не густо – два свойства, и инициализатор.
Теперь давайте создадим склад в котором собственно товары то и будут:

```
@interface ShopWarehouse : NSObject
{
    @private NSMutableArray *goods;
    @private GoodItemsEnumerator *goodItemsEnumerator;
    @private BadItemsEnumerator *badItemsEnumerator;
}


-(void) addItem:(ItemInShop *)anItem;

-(NSEnumerator *) getBrokenItemsEnumerator;
-(NSEnumerator *) getGoodItemsEnumerator;

@end



@implementation ShopWarehouse

-(id) init
{
    self = [super init];

    goods = [[NSMutableArray alloc] init];

    return self;
}

-(void) addItem:(ItemInShop *)anItem
{
    [goods addObject:anItem];
}

-(NSEnumerator *) getBrokenItemsEnumerator
{
    badItemsEnumerator = [[BadItemsEnumerator alloc] initWithItems:goods];
    return badItemsEnumerator;
}

-(NSEnumerator *) getGoodItemsEnumerator
{
    goodItemsEnumerator = [[GoodItemsEnumerator alloc] initWithItems:goods];
    return goodItemsEnumerator;
}

@end
```

Как видим, наш склад умеет добавлять товары, а также возвращать два таинственных объекта под названием GoodItemsEnumerator и BadItemsIterator.

Собственно их назначение очевидно, давайте посмотрим на реализацию. Для начала создадим базовый клас для обоих:

```
@interface BasicEnumerator : NSEnumerator
-(id)initWithItems:(NSMutableArray *)anItems;
-(NSArray *)allObjects;
-(id) nextObject;
@end
```

Как видим, это просто интерфейс, который предполагает реализацию 3х методов: инициализация, вернуть все объекты, и вернуть следующий объект. Давайте создадим два итератора как и задумывалось:

```
@interface BadItemsEnumerator : BasicEnumerator
{
    @private NSMutableArray *itemsArray;
    @private NSEnumerator *internalEnumerator;
}
@end

@implementation BadItemsEnumerator
-(id) initWithItems:(NSMutableArray *)anItems
{
    self = [super init];
    itemsArray = [[NSMutableArray alloc] init];
    for ( ItemInShop *item in anItems)
    {
        if (item.isBroken)
            [itemsArray addObject:item];
    }
    internalEnumerator = [itemsArray objectEnumerator];

    return self;
}

-(NSArray *)allObjects
{
    return itemsArray;
}

-(id) nextObject
{
    return [internalEnumerator nextObject];
}

@end
```

Я не привожу код GoodItemsIterator, потому как разнится они будут только в одной строчке:

```
    if (!item.isBroken)
```

Как видим во время инициализации, мы создаем свою копию данных, в которых только плохие товары. Так же создаем свой внутренний иетратор, из стандартных Cocoa.

Ну что, тестим:

```
    //создание тестовых даных
shopWarehouse = [[ShopWarehouse alloc] init];

[shopWarehouse addItem:[[ItemInShop alloc]initWithArgs:@"Item1" andQaulity:NO]];
[shopWarehouse addItem:[[ItemInShop alloc]initWithArgs:@"Item2" andQaulity:NO]];
```

```
[shopWarehouse addItem:[[ItemInShop alloc]initWithArgs:@"Item3" andQaulity:YES]];
[shopWarehouse addItem:[[ItemInShop alloc]initWithArgs:@"Item4" andQaulity:YES]];
[shopWarehouse addItem:[[ItemInShop alloc]initWithArgs:@"Item5" andQaulity:NO]];
```

Сам тест:

```
GoodItemsEnumerator *goodIterator = [shopWarehouse getGoodItemsEnumerator];
BadItemsEnumerator *badIterator = [shopWarehouse getBrokenItemsEnumerator];

ItemInShop *element;
while (element = [goodIterator nextObject]) {
    NSLog(@"Good Item = %@", element.name);
}

while (element = [badIterator nextObject]) {
    NSLog(@"Bad Item = %@", element.name);
```

И конечно же лог:

2013-02-25 01:18:10.401 IteratorPattern[5000:c07] Good Item = Item1
2013-02-25 01:18:10.403 IteratorPattern[5000:c07] Good Item = Item2
2013-02-25 01:18:10.403 IteratorPattern[5000:c07] Good Item = Item5
2013-02-25 01:18:10.404 IteratorPattern[5000:c07] Bad Item = Item3
2013-02-25 01:18:10.405 IteratorPattern[5000:c07] Bad Item = Item4

Код примера.

Visitor

Вот у каждого дома вероятнее всего есть холодильник. В ВАШЕМ доме, ВАШ холодильник. Что будет если холодильник сломается? Некоторые пойдут почитают в интернете как чинить холодильник, какая модель, попробуют поколдовать над ним, и разочаровавшись вызовут мастера по ремонту холодильников. Заметьте – холодильник ваш, но функцию “Чинить Холодильник” выполняет совершенно другой человек, про которого вы ничего не знаете, а попросту – обычный визитер.

Паттерн визитер – позволяет вынести из наших объектов логику, которая относится к этим объектам, в отдельный клас, что позволяет нам легко изменять / добавлять алгоритмы, при этом не меняя логику самого класа.

Когда мы захотим использовать этот паттерн:

1. Когда у нас есть сложный объект, в котором содержится большое количество различных элементов, и вы хотите выполнять различные операции в зависимости от типа этих элиментов.
2. Вам необходимо выполнять различные операции над класами, и при этом Вы не хотите писать вагон кода внутри реализации этих классов.
3. В конце – концов, вам нужно добавлять различные операции над элементами, и вы не хотите постоянно обновлять классы этих элементов.

Чтож, давайте вернемся к примеру из прошлой статьи, только теперь сложнее – у нас есть несколько складов, в каждом складе можнт хранится товар. Один визитер будет смотреть склады, другой визитер будет называть цену товара в складе.

Итак, для начала сам товар:

```
@interface WarehouseItem : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic) BOOL isBroken;
@property (nonatomic) int price;


-(id) initWithArgs:(NSString *)aName andQuality:(BOOL) isBrokenState
andPrice:(int)aPrice;

@end



@implementation WarehouseItem

-(id) initWithArgs:(NSString *)aName andQuality:(BOOL)isBrokenState
andPrice:(int)aPrice
{
    self = [super init];
    self.name = aName;
    self.isBroken = isBrokenState;
    self.price = aPrice;
    return self;
}
@end
```

И естественно сам склад:

```
@interface Warehouse : NSObject
{
@private NSMutableArray *_itemsArray;
}

-(void) addItem:(WarehouseItem *) anItem;
-(void) accept:(id<BasicVisitor>) visitor;

@end

@implementation Warehouse
-(void) addItem:(WarehouseItem *)anItem
{
    if (!_itemsArray)
        _itemsArray = [[NSMutableArray alloc] init];


    [_itemsArray addObject:anItem];
}

-(void) accept:(id<BasicVisitor>) visitor
{
    [visitor visit:self];
    for (WarehouseItem *item in _itemsArray)
        [visitor visit:item];
}

@end
```

Как видим, наш склад умеет хранить и добавлять товар, но также обладает таинственным методом accept который принимает в себя визитор и вызвает его метод visit. Чтобы картинка сложилась, давайте создадим протокол BasicVisitor и различных визиторов:

```
@protocol BasicVisitor <NSObject>

-(void) visit:(id)anObject;

@end
```

Как видим, протокол требует реализацию только одного метода. Теперь давайте перейдем к самим визитерам:

```
@interface QualityCheckerVisitor : NSObject <BasicVisitor>

@end

@implementation QualityCheckerVisitor

-(void) visit:(id)anObject
{
    if ([anObject isKindOfClass:[WarehouseItem class]])
    {
        if ([anObject isBroken])
        {
            NSLog(@"Item: %@ is broken", [anObject name]);
        }
        else
        {
            NSLog(@"Item: %@ is pretty cool!", [anObject name]);
        }
    }

    if ([anObject isKindOfClass:[Warehouse class]])
    {
```

```
            NSLog(@"Hmmm, nice warehouse!");
            return;
        }

    }

    @end
```

Если почитать код, то сразу видно что визитер при вызове своего метода visit определяет тип объекта который ему передался, и выполняет определнные функции в зависимоти от этого типа. Данный объект просто говорит или вещи на складе поломаны, а так же что ему нравится склад:)

```
    @interface PriceCheckerVisitor : NSObject <BasicVisitor>

    @end



    @implementation PriceCheckerVisitor

    -(void) visit:(id)anObject
    {
        if ([anObject isKindOfClass:[WarehouseItem class]])
        {
            NSLog(@"Item: %@ have price = %i", [anObject name], [anObject price]);
        }

        if ([anObject isKindOfClass:[Warehouse class]])
        {
            NSLog(@"Hmmm, I don't know how much Warehouse costs!");
            return;
        }

    }


    @end
```

В принципе этот визитер делает тоже самое, только в случае склада он признается что растерян, а в случае товара говорит цену товара!

Теперь давайте запустим то что у нас получилось! Код генерации тестовых даных:

```
    _localWarehouse = [[Warehouse alloc] init];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item1"
andQuality:NO andPrice:25]];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item2"
andQuality:NO andPrice:32]];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item3"
andQuality:YES andPrice:45]];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item4"
andQuality:NO andPrice:33]];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item5"
andQuality:NO andPrice:12]];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item6"
andQuality:YES andPrice:78]];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item7"
andQuality:YES andPrice:34]];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item8"
andQuality:NO andPrice:51]];
    [_localWarehouse addItem:[[WarehouseItem alloc] initWithArgs:@"Item9"
andQuality:NO andPrice:25]];
```

И собственно сам тестовый код:

```
PriceCheckerVisitor *visitor = [[PriceCheckerVisitor alloc] init];
QualityCheckerVisitor *qualityVisitor = [[QualityCheckerVisitor alloc] init];

[_localWarehouse accept:visitor];
[_localWarehouse accept:qualityVisitor];
```

Итак, при вызове метода accept нашего склада, визитер в начале проведывает наш склад, а потом проведывает каждый товар на этом склаже. При этом мы можем менять как визитера так и алгоритм, и это не повлечет изменения в коде клиента:)

Традиционный лог:

2013-02-26 00:19:47.756 VisitorPattern[8748:c07] Hmmm, I don't know how much Warehouse costs!
2013-02-26 00:19:47.759 VisitorPattern[8748:c07] Item: Item1 have price = 25
2013-02-26 00:19:47.759 VisitorPattern[8748:c07] Item: Item2 have price = 32
2013-02-26 00:19:47.760 VisitorPattern[8748:c07] Item: Item3 have price = 45
2013-02-26 00:19:47.761 VisitorPattern[8748:c07] Item: Item4 have price = 33
2013-02-26 00:19:47.762 VisitorPattern[8748:c07] Item: Item5 have price = 12
2013-02-26 00:19:47.763 VisitorPattern[8748:c07] Item: Item6 have price = 78
2013-02-26 00:19:47.763 VisitorPattern[8748:c07] Item: Item7 have price = 34
2013-02-26 00:19:47.764 VisitorPattern[8748:c07] Item: Item8 have price = 51
2013-02-26 00:19:47.765 VisitorPattern[8748:c07] Item: Item9 have price = 25
2013-02-26 00:19:47.765 VisitorPattern[8748:c07] Hmmm, nice warehouse!
2013-02-26 00:19:47.766 VisitorPattern[8748:c07] Item: Item1 is pretty cool!
2013-02-26 00:19:47.767 VisitorPattern[8748:c07] Item: Item2 is pretty cool!
2013-02-26 00:19:47.767 VisitorPattern[8748:c07] Item: Item3 is broken
2013-02-26 00:19:47.768 VisitorPattern[8748:c07] Item: Item4 is pretty cool!
2013-02-26 00:19:47.769 VisitorPattern[8748:c07] Item: Item5 is pretty cool!
2013-02-26 00:19:47.837 VisitorPattern[8748:c07] Item: Item6 is broken
2013-02-26 00:19:47.837 VisitorPattern[8748:c07] Item: Item7 is broken
2013-02-26 00:19:47.837 VisitorPattern[8748:c07] Item: Item8 is pretty cool!
2013-02-26 00:19:47.838 VisitorPattern[8748:c07] Item: Item9 is pretty cool!

Код примера.

Decorator

//уже после того как я решил оформить все в книгу, в блог посте в котором изначально я описал этот паттерн, некто по имени Саша сказал что я привожу не класический пример этого шаблона. И он действительно прав, пример тут использует категории, хотя в класическом исполнении сам декоратор должен быть отдельным объектом. Классическая реализация будет добавлена позже

Класный пример декоратора – различные корпуса для новых телефонов. Как-то я сразу с конца начал:) Для начала у нас есть телефон. Но так как телефон дорогой, мы будем очень счастливы если он не разобьется при любом падении – потому мы покупаем чехол для него. Тоесть, к уже существующему предмету мы добавили функционал защиты от падения. Ну еще мы взяли стильный чехол – теперь наш телефон еще и выглядит отлично. А потом мы докупили съемный объектив, с помощью которого можно делать фотографии с эффектом “рыбьего глаза”. Декорировали наш телефон дополнительным функционалом:)

Вот, приблизительно так выглядит реально описание паттерна декоратор. Теперь описание GoF:
Декоратор добавляет некий функционал уже существующему объекту.

Когда использовать этот паттерн:

1. Вы хотите добавить определенному объекту дополнительные возможности, при этом не задевая и не меняя других объектов
2. Дополнительные возможности класа – опциональны

Радость Objective-C в даном случае – это использование категорий. Я не буду детально описывать категории в этой книге, но в двух словах все же расскажу: Категории – это возможность расширить любой объект дополнительными методами (ТОЛЬКО МЕТОДАМИ) без унаследования от него. Давайте возьмем супер простой пример – декорирование Cocoa классов. К примеру добавим новый метод для объекта NSDate:

К примеру, нам нужно иметь возможность любую дату в нашем приложении как то определенно отформатировать и получить в виде строчки. Для начала создадим категорию:

```
@interface NSDate (StringDate)

-(NSString *) convertDateToString;

@end

@implementation NSDate (StringDate)

-(NSString *) convertDateToString
{
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"yyyy/dd/MM"];

    return [formatter stringFromDate:self];
}

@end
```

Как видим наша категория определяет только один метод “convertDateToString”, который дату форматирует в какой-то совсем странный формат, но у нас такая задача:)
Теперь план-капкан, сделать эту категорию пользуемой для всех объектов NSDate в нашем приложении. Для этого в файле appName-Prefix, добавляем строчку:

```
#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
    #import "NSDate+StringDate.h"
#endif
```

строчка выделенна жирным, чтобы не потеряться.

Вы будете смеятся, но в принципе вот и все:) Примерный код тестирования выглядит следующим образом:

```
NSDate *dateNow = [NSDate date];

NSLog(@"Date is %@", [dateNow convertDateToString]);
```

Традиционный log:

2013-03-01 00:30:18.328 DecoratorPattern[11731:c07] Date is 2013/01/03

Код примера.

Chain of responsibility

Паттерн с моим любимым названием:)

Представьте себе очередь людей которые пришли за посылками. Выдающий посылки человек, дает первую посылку первому в очереди человеку, он смотрит на имя-фамилию на корбке, видит что посылка не для него, и передает посылку дальше. Второй человек делает собственно тоже самое, и так пока не найдется получатель.

Цепочка ответственности (chain of responsibility) – позволяет вам передавать объекте по цепочке объектов-обработчиков, пока не будет найден необходимый объект обработчик.

Когда использовать этот паттерн:

1. У вас более чем один объект-обработчик.
2. У вас есть несколько объектов обработчика, при этом вы не хотите специфицировать который объект должен обрабатывать в даный момент времени.

Как всегда – пример:

Представим что у нас есть конвеер, который обрабатывает различные предметы которые на нем: игрушки, электронику и другие.
Для начала создадим классы объектов которые могут быть обработаны нашими обработчиками:

```
//базовый объект
@interface BasicItem : NSObject
@end

@implementation BasicItem
@end

//игрушка
@interface Toy : BasicItem
@end

@implementation Toy
@end

//электроника
@interface Electronics : BasicItem
@end

@implementation Electronics
@end

//и мусор
@interface Trash : BasicItem
@end

@implementation Trash
@end
```

Теперь создадим обработчики:

```
@interface BasicHandler : NSObject
{
```

```
@private BasicHandler *_nextHandler;
}

@property (nonatomic, strong) BasicHandler *nextHandler;
-(void) handleItem:(BasicItem *) item;
@end

@implementation BasicHandler
@end
```

Как видим, наш базовый обработчик, умеет обрабатывать объекты типа BasicItem. И самое важное – он имеет ссылку на следующий обработчик (как в нашей очереди, про людей передающие посылку).
Давайте создадим код обработчика игрушки:

```
@interface ToysHandler : BasicHandler
@end

@implementation ToysHandler

-(void) handleItem:(BasicItem *)item
{
    if ([item isKindOfClass:[Toy class]])
    {
        NSLog(@"Toy found. Handling");
    }
    else
    {
        NSLog(@"Toy not found. Handling using next handler");
        [self.nextHandler handleItem:item];
    }
}

@end
```

Как видим, если обработчик получает объект класса Toy – то он его обрабатывает, если нет – то обработчик передает объект следующему обработчику.
По аналогии создадим два следующих обработчика: для электроники, и мусора:

```
//хэндлер электроники
@interface ElectronicsHandler : BasicHandler
@end

@implementation ElectronicsHandler

-(void) handleItem:(BasicItem *)item
{
    if ([item isKindOfClass:[Electronics class]])
    {
        NSLog(@"Electronics found. Handling");
    }
    else
    {
        NSLog(@"Electronics not found. Handling using next handler");
        [self.nextHandler handleItem:item];
    }
}

@end


//хэндлер мусора
@interface OtherItemsHandler : BasicHandler
@end
```

```
@implementation OtherItemsHandler

-(void) handleItem:(BasicItem *)item
{
    NSLog(@"Found undefined item. Destroying");
}

@end
```

Как видим OtherItemsHandler в случае, когда до него дошло дело – объект уничтожает, и не пробует дергать следующий обработчик.

Давайте тестировать:

```
BasicHandler *toysHandler = [[ToysHandler alloc] init];
BasicHandler *electronicsHandler = [[ElectronicsHandler alloc] init];
BasicHandler *otherItemHandler = [[OtherItemsHandler alloc]init];

electronicsHandler.nextHandler = otherItemHandler;
toysHandler.nextHandler = electronicsHandler;

BasicItem *toy = [[Toy alloc] init];
BasicItem *electronic = [[Electronics alloc] init];
BasicItem *trash = [[Trash alloc] init];


[toysHandler handleItem:toy];
[toysHandler handleItem:electronic];
[toysHandler handleItem:trash];
```

Как видим мы в начале создаем обработчики, потом скрепляем их в цепь, и пытаемся обработать различные элементы.Традиционно лог:

2013-03-02 15:35:35.668 ChainOfResponsibility[16777:c07] Toy found. Handling
2013-03-02 15:35:35.671 ChainOfResponsibility[16777:c07] Toy not found. Handling using next handler
2013-03-02 15:35:35.672 ChainOfResponsibility[16777:c07] Electronics found. Handling
2013-03-02 15:35:35.673 ChainOfResponsibility[16777:c07] Toy not found. Handling using next handler
2013-03-02 15:35:35.673 ChainOfResponsibility[16777:c07] Electronics not found. Handling using next handler
2013-03-02 15:35:35.674 ChainOfResponsibility[16777:c07] Found undefined item. Destroying

Код примера.

Template Method

Вы заметили как много в нашей жизни шаблонов? Ну к примеру – наше поведение когда мы приходим в незнакомый дом:

1. Зайти
2. Поздороваться с хозяевами
3. Раздеться, молясь о том что у нас носки не дырявые
4. Пройти, и охать удияляясь какая большая/уютная/забавная квартира.

Или же когда мы приходим в квартиру, где происходит ремонт:

1. Зайти
2. Поздороваться с хозяевами
3. Не раззуваться, потому как грязно!
4. Поохать когда хозяин квартиры поведает нам смелость его архитектурной мысли!

В целом, все происходит практически одинаково, но с изюминкой в каждом различно случае:) Наверное потому то, это и называется шаблоном поведения. Шаблонный метод задает алгоритму пошаговоую инструкцию. Элементы алгоритма же, определяются в наследующих класах.

Сам паттерн ну очень интуитивный, и я уверен что многие давно уже использовали его. Потому давайте попробуем сделать пример.
Вернемся к старой практике, писать примеры по созданию телефонов!

Итак, напишем наш шаблонный клас, с помощью которого будем создавать телефон:

```
@interface AnyPhoneTemplate : NSObject
//it will be template method
-(void) makePhone;
-(void) takeBox;
-(void) takeMicrophone;
-(void) takeCamera;
-(void) assemble;
@end

@implementation AnyPhoneTemplate
//our template method
-(void) makePhone
{
    [self takeBox];
    [self takeCamera];
    [self takeMicrophone];
    [self assemble];
}

-(void) takeBox
{
    NSLog(@"Taking a box");
}

-(void) takeCamera
{
    NSLog(@"Taking a camera");
}
```

```
-(void) takeMicrophone
{
    NSLog(@"Taking a microphone");
}

-(void) assemble
{
    NSLog(@"Assembling everythig");
}

@end
```

Как вы уже наверное догадались – сам шаблонный метод, это метод makePhone – который задает последовательность вызовов методов необходимых для складывания телефонов. Давайте теперь научим нашу программу создавать айфоны:

```
@interface iPhoneMaker : AnyPhoneTemplate

-(void) design;

@end

@implementation iPhoneMaker

-(void) takeBox
{
    [self design];
    [super takeBox];
}


-(void) design
{
    NSLog(@"Putting label 'Designed in California'");
}
@end
```

Как видим у сборщика яблочных телефонов есть один дополнительный метод – design, а также перегруженный метод takeBox в котором дополнительно вызывается метод design и после этого вызывается родительский метод takeBox.

На очереди сборка Android:

```
@interface AndroidMaker : AnyPhoneTemplate

-(void) addRam;
-(void) addCPU;

@end

@implementation AndroidMaker

-(void) assemble
{
    [self addCPU];
    [self addRam];
    [super assemble];
}

-(void) addCPU
{
    NSLog(@"Installing 4 more CPUs");
}
```

```
-(void) addRam
{
    NSLog(@"Installing 2Gigs of RAM");

}

@end
```

Как видим у сборщика андроида аж целых два дополнительных метода, и перегружен метод assemble.

Тест здесь конечно же – элементарный:

```
AndroidMaker *android = [[AndroidMaker alloc] init];
iPhoneMaker *iphone = [[iPhoneMaker alloc] init];


[android makePhone];
[iphone makePhone];
```

Традиционный log:

2013-03-03 22:56:28.996 TemplateMethod[21040:c07] Taking a box
2013-03-03 22:56:28.998 TemplateMethod[21040:c07] Taking a camera
2013-03-03 22:56:28.999 TemplateMethod[21040:c07] Taking a microphone
2013-03-03 22:56:29.000 TemplateMethod[21040:c07] Installing 4 more CPUs
2013-03-03 22:56:29.000 TemplateMethod[21040:c07] Installing 2Gigs of RAM
2013-03-03 22:56:29.001 TemplateMethod[21040:c07] Assembling everythig
2013-03-03 22:56:29.001 TemplateMethod[21040:c07] Putting label 'Designed in California'
2013-03-03 22:56:29.002 TemplateMethod[21040:c07] Taking a box
2013-03-03 22:56:29.003 TemplateMethod[21040:c07] Taking a camera
2013-03-03 22:56:29.003 TemplateMethod[21040:c07] Taking a microphone
2013-03-03 22:56:29.003 TemplateMethod[21040:c07] Assembling everything

Код примера.

Strategy

Если Ваша девушка злая, вы скорее всего будете общаться с ней осторожно. Если на вашем проетке завал, то вероятнее всего вы не будете предлагать в команде вечерком дернуть пива или поиграть в компьютерные игры. В различных ситуациях, у нас могут быть очень разные стратегии поведения. К примеру, в приложении вы можете использовать различные алгоритмы сжатия, в зависимости от того с каким форматом картинки вы работаете, или же куда вы хотите после этого картинку деть. Вот мы и добрались до паттерна Стратегия.

Также отличным примером может быть MVC паттерн – в разных случаях мы можем использовать разные контроллеры для одного и того же View (к примеру авторизированный и не авторизированный пользователь).

Паттерн Стратегия определяет семейство алгоритмов, которые могут взаимозаменяться.

Когда использовать паттерн:

1. Вам необходимы различные алгоритмы
2. Вы очень не хотите использовать кучу вложеных If-ов
3. В различных случаях ваш класс работает по разному.

Давайте напишем пример – RPG игра, в которой у вас есть различные стратегии нападения Вашими персонажами:) Каждый раз когда вы делаете ход, ваши персонажи делают определенное действие. Итак, для начала управление персонажами!

Создадим базовую стратегию:

```
@interface BasicStrategy : NSObject
-(void) actionCharacter1;
-(void) actionCharacter2;
-(void) actionCharacter3;
@end
```

Как видно из кода стратегии – у нас есть 3 персонажа, каждый из которых может совершать одно действие! Давайте научим персонажей нападать!

```
@interface AttackStrategy : BasicStrategy
@end

@implementation AttackStrategy

-(void) actionCharacter1
{
    NSLog(@"Character 1: Attack all enemies!");
}

-(void) actionCharacter2
{
    NSLog(@"Character 2: Attack all enemies!");
}

-(void) actionCharacter3
{
    NSLog(@"Character 3: Attack all enemies!");
```

```
}
@end
```

Как видим, при использовании такой стратегии наши персонажи нападают на все что движется! Давайте научим их защищаться:

```
@interface DefenceStrategy : BasicStrategy

@end

@implementation DefenceStrategy

-(void) actionCharacter1
{
    NSLog(@"Character 1: Attack all enemies!");
}

-(void) actionCharacter2
{
    NSLog(@"Character 2: Healing Character 1!");
}

-(void) actionCharacter3
{
    NSLog(@"Character 3: Protecting Character 2!");
}
@end
```

Как видим во время защитной стратегии, наши персонажи действуют по-другому – кто атакует, кто лечит, а некоторый даже защищают:) Ну, теперь как-то надо это все использовать. Давайте создадим нашего игрока:

```
@interface Player : NSObject

@property (nonatomic, strong) BasicStrategy *_strategy;

-(void) makeAction;
-(void) changeStrategy:(BasicStrategy *) strategy;

@end

@implementation Player

-(void) makeAction
{
    [self._strategy actionCharacter1];
    [self._strategy actionCharacter2];
    [self._strategy actionCharacter3];
}

-(void) changeStrategy:(BasicStrategy *)strategy
{
    self._strategy = strategy;
}

@end
```

Как видим наш игрок может только менять стратегию и действовать в зависимости от этой стратегии.
Код для тестирования:

```
Player *p = [[Player alloc] init];
AttackStrategy *a = [[AttackStrategy alloc] init];
DefenceStrategy *d = [[DefenceStrategy alloc] init];
```

```
[p changeStrategy:a];
[p makeAction];
[p changeStrategy:d];
[p makeAction];
```

Собственно все предельно ясно:) В первом случае наши персонажи будут активно атоковать, а после смены стратегии уйдут в глухую оборону.
Традиционный лог:

2013-03-04 23:57:44.797 StrategyPatterns[22420:c07] Character 1: Attack all enemies!
2013-03-04 23:57:44.799 StrategyPatterns[22420:c07] Character 2: Attack all enemies!
2013-03-04 23:57:44.800 StrategyPatterns[22420:c07] Character 3: Attack all enemies!
2013-03-04 23:57:44.800 StrategyPatterns[22420:c07] Character 1: Attack all enemies!
2013-03-04 23:57:44.801 StrategyPatterns[22420:c07] Character 2: Healing Character 1!
2013-03-04 23:57:44.801 StrategyPatterns[22420:c07] Character 3: Protecting Character 2!

Код примера.

Command

Стоять, лежать, сидеть – все это команды которые нам очень часто давали учителя физкультуры. Так как это очень часто происходит в нашей жизни, глупо было бы предполагать что кто нибудь не придумает шаблон с одноименным названием.

Итак, паттерн – команда – позволяет инкапулировать всю информацию необходимую для выполнения определенных операций, которые могут быть выполнены потом, используя объект команды.

Образно говоря, если взять наш с вами пример физрука, родители давным давно инкапсулировали в нас команду ”Сидеть”, потому физрук использует ее чтобы мы сели, не объясняя при этом как это сделать.

Когда использовать паттерн:

Ну, собственно ответ один, и выходит он из описания, когда вы хотите инкапсулировать определенную логику в отдельный класс команду. Отличный пример – do/undo операции. У вас, вероятнее всего, будет так называмы CommandManager, которые будет запоминать что делает команда, и при желании отменять предыдущее действите если выполнить команду undo (кстати, это может быть и просто метод).

Собственно, есть два пути реализации этого паттерна:

Для начала создадим базовую команду:

```
@interface BaseCommand : NSObject

-(void) execute;
-(void) undo;

@end
```

Как видим у нашей команды аж два метода – сделать, и вернуть обратно изменения.

Теперь реализации наших команд:

```
@interface FirstCommand : BaseCommand
{
@private NSString *_originalString;
@private NSString *_currentString;
}

-(id) initWithArguments: (NSString *) anArgument;
-(void) printString;

@end


@implementation FirstCommand

-(id) initWithArguments:(NSString *)anArgument
{
    self = [super init];
    _originalString =  anArgument;
    _currentString = anArgument;
```

```
    return self;
}
-(void) execute
{
    _currentString = @"This is a new string";
    [self printString];

    NSLog(@"Execute command called");
}

-(void) undo
{
    _currentString = _originalString;
    [self printString];

    NSLog(@"Undo of execute command called");
}

-(void) printString
{
    NSLog(@"Current string is equal to %@", _currentString);
}

@end
```

Как видим, наша первая команда просто умеет менять одну строчку. При чем всегда хранит оригинал, чтобы можно было отменить изменение.

Вторая наша команда:

```
@interface SecondCommand : BaseCommand
{
@private int _originalNumber;
@private int _currentNumber;
}

-(id) initWithArgs: (int) aNumber;
-(void) printNumber;

@end

@implementation SecondCommand

-(id) initWithArgs:(int)aNumber
{
    self = [super init];

    _originalNumber = aNumber;
    _currentNumber = aNumber;

    return self;
}

-(void) execute
{
    _currentNumber++;

    [self printNumber];
}

-(void) undo
{
    if (_currentNumber > _originalNumber)
        _currentNumber--;

    [self printNumber];
```

```
}

-(void) printNumber
{
    NSLog(@"current number is %i", _currentNumber);
}
@end
```

Вторая команда делает тоже самое, но с числом.

Давайте теперь создадим объект который будет получать команду и выполнять ее:

```
@interface CommandExecutor : NSObject
{
@private NSMutableArray *_arrayOfCommands;

}

-(void) addCommand:(BaseCommand *) aCommand;
-(void) executeCommands;
-(void) undoAll;

@end

@implementation CommandExecutor

-(id) init
{
    self = [super init];

    _arrayOfCommands = [[NSMutableArray alloc] init];

    return self;
}

-(void) addCommand:(BaseCommand *) aCommand
{
    //id<CommandProtocol> item = aCommand;

    [_arrayOfCommands addObject:aCommand];
}

-(void) executeCommands
{
    for (BaseCommand *command in _arrayOfCommands)
    {
        [command execute];
    }
}

-(void) undoAll
{
    for (BaseCommand *command in _arrayOfCommands)
    {
        [command undo];
    }
}
@end
```

Как видим, наш менеджер может получать очередь команд, и выполнять их все, или даже отменять все действия (пример простой и с багами:)). Итак, наш тестовый код:

```
CommandExecutor *commandE = [[CommandExecutor alloc] init];
```

```
    BaseCommand *cmdF = [[FirstCommand alloc] initWithArguments:@"This is a test
string"];
    BaseCommand *cmdS = [[SecondCommand alloc] initWithArgs:3];

    [commandE addCommand:cmdF];
    [commandE addCommand:cmdS];

    [commandE executeCommands];
    [commandE undoAll];
```

И конечно же лог:

2013-03-06 22:40:47.392 CommandPattern[9871:c07] Current string is equal to This is a new string
2013-03-06 22:40:47.393 CommandPattern[9871:c07] Execute command called
2013-03-06 22:40:47.393 CommandPattern[9871:c07] current number is 4
2013-03-06 22:40:47.394 CommandPattern[9871:c07] Current string is equal to This is a test string
2013-03-06 22:40:47.394 CommandPattern[9871:c07] Undo of execute command called
2013-03-06 22:40:47.395 CommandPattern[9871:c07] current number is 3

2. Второй метод реализации паттерна – это уже использование внутреностей самой Cocoa – NSInvocation:

NSInvocation – это объект, который можно использовать чтобы передать возможно вызова метода одного класса – другому, и при это передать в него несколько аргументов, а так же объект который вызывал этот метод.

Давайте напишем в наш CommandExecutor добавим два метода и одно приватное поле:

```
    //private field
    @private NSInvocation *_specificCommand;

    //two methods:
    -(void) setSpecificCommand:(NSInvocation *)aCommand
    {
        _specificCommand = aCommand;
    }

    -(void) executeSpecificCommand
    {
        [_specificCommand invoke];
    }
```

Как видим, теперь наш объект сохраняет объект типа NSInvocation и может его запустить когда требуется. Давайте тепреь в нашем основном контроллере напишем функцию которую мы будем вызывать:

```
    -(void) methodInMainController:(int) aFirstArgument andString:(NSString
    *)aStringArgument
    {
        NSLog(@"Method called with first argument = %i and second argument = %@",
                                                aFirstArgument, aStringArgument);
    }
```

А теперь создадим объект типа NSInvocation который и будет в результате нашей командой:

```
NSMethodSignature *signature = [self methodSignatureForSelector:
                               @selector(methodInMainController:andString:)];
NSInvocation *invocationToPass =[NSInvocation
                                    invocationWithMethodSignature:signature];
[invocationToPass setTarget:self];
[invocationToPass setSelector:@selector(methodInMainController:andString:)];

int intArgument = 3;
NSString *stringArgument = @"This is a string argument";

[invocationToPass setArgument:&intArgument atIndex:2];
[invocationToPass setArgument:&stringArgument atIndex:3];

CommandExecutor *executor = [[CommandExecutor alloc] init];
[executor setSpecificCommand:invocationToPass];
[executor executeSpecificCommand];
```

Как видим, в самом начале мы создаем объект который хранит сигнатуры нашего метода, и создаем объект NSInvocation. После этого мы передаем в него аргументы – не стоит удивляться что индекс начинается с цифры 2 – index 0 и index 1 зарезервированы под target и selector:)

Ну и конечно же лог:

2013-03-06 23:18:26.624 CommandPattern[10479:c07] Method called with first argument = 3 and second argument = This is a string argument

Код примера.

Flyweight

Я задумался, как объяснить да и перевисти этот паттерн на примеры из реальной жизни, и потерпел полнейшее фиаско! Потому сразу к описанию и примерам!

Flyweight – паттерн который помогает нам отделять определенную информацию, для того чтобы в будущем делиться этой информацией с многими объектами.

Как пример возьмем любую стратегичискую игру – представьте что у вас 1 тысяча солдат одного типа – если каждый будет лезть на диск и пробовать подгрузить картинку с диска – вероятнне всего у вас или память закончится или производительность будет желать лучшего. Очень классно этот пример рассмотрен в книге Андрея Будая ”Дизайн-патерни — просто, як двері”.

Потому не найдя ничего лучше, я решил просто портировать пример.

Когда использовать этот паттерн?

1. У вас ооочень много однотипых объектов в приложении
2. Много объектов сохранены в памяти, от чего производительность вашего приложения страдает
3. Вы видите, что несколько объектов которые могут быть разшарены – спасут вас от создания тонны других объектов

Итак пример:
Пускай мы пишем игру, где есть два типа персонажей – гоблины и драконы. Для начала создадим базовый класс для всех юнитов:

```
@interface BasicUnit : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic) int health;
@property (nonatomic, strong) UIImage *image;

@end
```

Как видим у каждого юнита есть свойство image – которое является типом UIImage и может потребвать подгрузки картинки для каждого юнита. Как же сделать загрузку только единожды? Ну собвстенно с этим то и справится наш паттерн!

```
@interface FlyweightImageFactory : NSObject

+(UIImage *) getImage:(NSString *)imageName;

@end

@implementation FlyweightImageFactory

NSMutableDictionary *_imageDictionary;

+(UIImage *)getImage:(NSString *)imageName
{
    if (!_imageDictionary)
        _imageDictionary = [[NSMutableDictionary alloc] init];

    if (![_imageDictionary objectForKey:imageName])
    {
        [_imageDictionary setObject:[UIImage imageNamed:[[NSString alloc]
```

```
    initWithFormat:@"%@.jpeg",imageName]] forKey:imageName];
            NSLog(@"Loading image of the %@", imageName);
        }

        return [_imageDictionary objectForKey:imageName];
    }

    @end
```

Как видим, наш flyweight иммет только один класс метод, который картинку по имени то и возвращает. Если картинки под таким именем нету в его словаре – то она грузится из бандла, если же есть – то просто передается ссылка на нее. Каждый раз когда картинка грузится из бандла мы логируем сообщение, это сделано для того чтобы увидеть сколько раз происходит подгрузка изображения из бандла.

Теперь нам просто нужно в конструкторе наших юнитов загружать картинку не на прямую, а через наш паттерн:

```
    @implementation Dragon

    -(id) init
    {
        self = [super init];

        self.name = @"Dragon";
        self.health = 150;
        self.image = [FlyweightImageFactory getImage:@"dragon"];
        return self;

    }

    @end

    @implementation Goblin

    -(id) init
    {
        self = [super init];

        self.name = @"goblin";
        self.health = 20;
        self.image = [FlyweightImageFactory getImage:@"goblin"];
        return self;

    }

    @end
```

Ну и конечно же тест:

```
    NSMutableArray *units = [[NSMutableArray alloc] init];
    for ( int i = 0 ; i < 500; i++)
    {
        [units addObject:[[Dragon alloc] init]];
    }
    for ( int i = 0 ; i < 500; i++)
    {
        [units addObject:[[Goblin alloc] init]];
    }
```

И как ожидается, хоть мы и создаем 1 тысячу юнитов, лог срабатывает только два раза:

2013-03-09 11:08:45.002 FlyweightPattern[5595:c07] Loading image of the dragon
2013-03-09 11:08:45.006 FlyweightPattern[5595:c07] Loading image of the goblin

Код примера.

Proxy

Ох, все кто работает в большой компании – ненавидит доступ к интернету через прокси:) Что делает прокся? Ну многие из нас уверенны, что в основном она режет скорость интернета, хотя вероятнее всего она делает еще очень много положительных вещей:

1. Логирует кто куда ходит.
2. Смотрит, чтобы не ходили куда не следует.
3. Смотрит, чтобы по нашему коннекшену к нам не ходили.

…и так далее. Все эти активности взяты из головы, но они показывают использование прокси в реальной жизни – давать стандартный доступ к чему-либо, при этом обворачивая стандартные вызовы в проксю и добавляют свою логику.

Паттерн прокси – подменяет реальный объект, и шлет ему запросы через свои интерфейсы. При этом может добавлять дополнительную логику, или создавать реальный объект если тот еще не создан.

Как пример – вы можете иметь обычных и премиум пользователей приложения. К примеру – премиум пользователи могут скачивать файлы на большей скорости, чем обычные пользователи. Потому, как объекту, который отвечает за скачивание файлов в вашем приложении, не обязательно знать про существование разных типов пользователей, вы оборачиваете этот объект в прокси, которая в свою очередь знает про таких пользователей, и говорит объекту скачивания на какой скорости пользователь должен получить файл.

Когда использовать паттерн:

1. Возомжно, у вас есть два сервера – тестовый и продуктовый. Когда Вы дебажите – скорее всего вы будете пользовать тестовый сервер, ну а когда компилируете приложение для продакшена – скорее всего реальный. Эту логику можно реализовать в проксе
2. Добавление различных валидаций, и проверок безопасности
3. Миллион других возможных ситуаций.

Давайте создадим пример. Пускай у нас есть объект который отвечает за скачку файлов:

```
@interface FileDownloader : NSObject

-(void) slowDownload;
-(void) fastDownload;

@end

@implementation FileDownloader

-(id) init
{
    self = [super init];
    NSLog(@"Downloader created");
    return self;
}
```

```
-(void) slowDownload
{
    NSLog(@"Slooooooowly downloading...");
}

-(void) fastDownload
{
    NSLog(@"Shuuuuuh, already downloaded...");
}
@end
```

Как видим, наш объект умеет скачивать быстро и медленно. При том, ему все равно какой пользователь и есть ли коннект к интернету.
Давайте создадим нашу прокси:

```
@interface FileDownloaderProxy : NSObject
{
@private FileDownloader *_downloader;
}


@property (nonatomic) bool isPremiumUser;

-(void) slowDownload;
-(void) fastDownload;


@end

@implementation FileDownloaderProxy


-(void) fastDownload
{
    if (!_isPremiumUser)
    {
        [self slowDownload];
        return;
    }

    if (!_downloader)
        _downloader = [[FileDownloader alloc] init];

    [self checkNetworkConnectivity];

    [_downloader fastDownload];
}

-(void) slowDownload
{
    if (!_downloader)
        _downloader = [[FileDownloader alloc] init];

    [self checkNetworkConnectivity];

    [_downloader slowDownload];
}

-(void) checkNetworkConnectivity
{
    NSLog(@"Checking network connectivity...");
}
@end
```

Как видим проксятник незначительно умнее:

1. Он знает про тип пользователя, и даже если дернули метод fastDownload но пользователь не премиум – будет вызван метод slowDownload.
2. Он умеет проверять доступ к интернету (пусть это и просто выписка лога).
3. Он проверяет, или тип объекта FileDownloader создан, и если нет – создает его.

Ну что, протестируем:

```
FileDownloaderProxy *proxy = [[FileDownloaderProxy alloc] init];

[proxy setIsPremiumUser:NO];
[proxy fastDownload];

[proxy setIsPremiumUser:YES];
[proxy fastDownload];
```

Традиционный лог:

2013-03-10 13:27:50.312 ProxyPattern[10775:c07] Downloader created
2013-03-10 13:27:50.313 ProxyPattern[10775:c07] Checking network connectivity…
2013-03-10 13:27:50.313 ProxyPattern[10775:c07] Slooooooowly downloading…
2013-03-10 13:27:50.314 ProxyPattern[10775:c07] Checking network connectivity…
2013-03-10 13:27:50.314 ProxyPattern[10775:c07] Shuuuuuh, already downloaded…

Код примера.

Memento

Ах, как же не хватает в жизни таких штук как Quick Save и Quick Load. На худой конец Ctrl + Z. Это я Вам как геймер давнейший говорю! Частенько, такой функционал очень полезен для реализации в приложении. Очень правильно также защитить наше записанное состояние от других класов, чтобы в них не смогли внести изменения.

Итак, что же за паттерн такой? Memento – паттерн который позволяет, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта, чтобы позже восстановить его состояние.

Состояние, как таковое, может сохраняться как в файловую систему, так и в базу даных. Яркий пример использование – может быть сворачивание и выключение вашего приложения – во время выключения приложения вы можете сохранить все данные с формы, или настройки, или еще что вам угодно в базу даных через CoreData, чтобы восстановить при включении.

Когда использовать паттерн:

1. Вам необходимо сохранять состояние объекта как слепок(snapshot) за определенный период
2. Вы хотите скрыть интерфейс получения состояния объекта.

В даном паттерне используется три ключевые объекта: Caretaker (объект который скрывает реализацию сохранения состояния объекта), originator (собственно сам объект) и конечно же сам Memento (объект который сохраняет состояние originator).

Давайте небольшой пример:

```
@interface OriginatorState : NSObject

@property (nonatomic) int intValue;
@property (nonatomic) NSString *stringValue;

@end

@implementation OriginatorState
@end
```

Допустим, у нас есть состояние, в котором всего лишь два знаечение – целочисельное и строчка.

```
@interface Originator : NSObject
{
@private OriginatorState *_localState;
}

-(void) changeValues;
-(OriginatorState *) getState;
-(void) setState:(OriginatorState *)oldState;
@end

@implementation Originator
```

```
-(id) init
{
    self = [super init];

    _localState = [[OriginatorState alloc] init];
    _localState.intValue = 100;
    _localState.stringValue = @"Hello World!";

    return self;
}

-(void) changeValues
{
    _localState.intValue++;
    _localState.stringValue = [NSString stringWithFormat:@"%@ %@",
_localState.stringValue, @"!"];

    NSLog(@"Current state int = %i, string = %@", _localState.intValue,
_localState.stringValue);
}

-(OriginatorState *) getState
{
    return _localState;
}

-(void) setState:(OriginatorState *)oldState
{
    _localState = oldState;

     NSLog(@"Load completed. Current state: int = %i, string = %@",
_localState.intValue, _localState.stringValue);
}
@end
```

Как видим, мы можем изменять состояние объекта состояния, а так же получить состояние и загрузить состояние.

Пускай у нас есть Memento – объект который будет заведовать состояние нашего объекта:

```
@interface Memento : NSObject
{
@private OriginatorState *_localState;
}

-(id) initWithState:(OriginatorState *)state;
-(OriginatorState*) getState;
@end

@implementation Memento

-(id) initWithState:(OriginatorState *)state
{
    self = [super init];

    _localState = [[OriginatorState alloc] init];
    [_localState setIntValue:state.intValue];
    [_localState setStringValue:state.stringValue];

    return self;
}

-(OriginatorState *) getState
{
    return _localState;
```

```
}
@end
```

Тоесть наш объект Memento – умеет хранить состояние, и конечно же отдавать состояние:)
Ну и теперь, соеденим все это в единый паззл создавая Caretaker:

```
@interface Caretaker : NSObject
{
@private Originator *_originator;
@private Memento *_memento;
}

-(void) changeValue;
-(void) saveState;
-(void) loadState;
@end

@implementation Caretaker

-(void) changeValue
{
    if (!_originator)
        _originator = [[Originator alloc] init];

    [_originator changeValues];
}

-(void) saveState
{
    _memento = [[Memento alloc] initWithState:[_originator getState]];
    NSLog(@"Saved state. State int = %i and string = %@",[[_memento getState]
intValue ], [[_memento getState] stringValue ]);
}

-(void) loadState
{
    [_originator setState:[_memento getState]];
}

@end
```

Как видим Caretaker умеет держать в себе сохраненное состояние(для примера, оно все очень просто, но здесь может быть и стек состояний, и так далее), а так же загружать его:)

Давайте протестим:

```
Caretaker *crtaker = [[Caretaker alloc] init];

[crtaker changeValue];
[crtaker saveState];
[crtaker changeValue];
[crtaker changeValue];
[crtaker changeValue];
[crtaker loadState];
```

Лог как пример работы паттерна:

2013-03-11 23:23:30.711 MementoPattern[14985:c07] Current state int = 101, string = Hello World! !
2013-03-11 23:23:30.712 MementoPattern[14985:c07] Saved state. State int = 101 and string = Hello World! !

2013-03-11 23:23:30.712 MementoPattern[14985:c07] Current state int = 102, string = Hello World! ! !
2013-03-11 23:23:30.713 MementoPattern[14985:c07] Current state int = 103, string = Hello World! ! ! !
2013-03-11 23:23:30.713 MementoPattern[14985:c07] Current state int = 104, string = Hello World! ! ! ! !
2013-03-11 23:23:30.713 MementoPattern[14985:c07] Load completed. Current state: int = 101, string = Hello World! !

Код примера.