iPHONE NA PRÁTICA

APRENDA PASSO A PASSO A DESENVOLVER SOLUÇÕES PARA IOS

Fabio Marzullo

Novatec

Editor: Rubens Prates
Revisão gramatical: Alessandro Thomé
Editoração eletrônica: Carolina Kuwabata
Capa: Cesar Marzullo

ISBN: 978-85-7522-297-3

Histórico das impressões:

Abril/2012 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Fax: +55 11 2950-8869
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

CAPÍTULO 1

Orientação a objetos

Pensei bastante antes de introduzir este capítulo no livro. A ideia sempre foi apresentar um conteúdo que pudesse lhe oferecer um meio rápido de desenvolver aplicativos para iOS sem que você tivesse que se preocupar em separar o "joio do trigo" na vasta quantidade de informação à disposição online.

Acredito que eu tenha atingindo esse objetivo, e você perceberá que há neste livro material suficiente para programar bem em iOS. Porém uma coisa é certa: se você não tem conhecimento em orientação a objetos, poderá enfrentar alguma dificuldade. Não que a falta dessa experiência específica seja impeditiva, mas seria bom tê-la, pois lhe facilitaria a compreensão do que está sendo proposto nos exemplos e exercícios.

Se você já possui alguma experiência em programação orientada a objetos (doravante denominado OO), sinta-se à vontade para seguir ao tópico sobre o estilo arquitetural MVC, no qual explico como deve utilizá-lo em suas aplicações para iOS. Se você não se sente tão seguro, vale a leitura, mesmo porque trago um direcionamento sobre como o programador que não está acostumado com linguagens como Java ou .Net deve pensar suas soluções, transportando-as do mundo real para o mundo virtual.

Na verdade, programar orientado a objetos é muito simples, porque de uma maneira muito transparente pensamos dessa forma diariamente. Quando imaginamos um carro, um ônibus, um caminhão, uma moto, sabemos que estes possuem características similares, como número de rodas, número de assentos, cor, velocidade, aceleração, tipo de painel, estofado etc. Sabemos que têm comportamento semelhante, pois podemos ligá-los, acelerá-los, freá-los, virá-los para a esquerda ou a direita, entre outras ações, e tal pensamento se propaga por quase tudo que observamos em nosso cotidiano.

Quando transportamos esse conhecimento para o mundo virtual – no caso, o desenvolvimento de software –, algumas linguagens permitem que esse modelo mental seja mais facilmente representado a partir do que vemos no mundo real. Por exemplo, ao levar um carro, um ônibus, um caminhão ou uma moto ao mundo

virtual, nós os transformamos em entidades que farão parte da nossa solução de software. Como consequência somos obrigados a identificar as características e o comportamento necessários para representá-los dentro do software, o que nos leva a entender o conceito de abstração.

Abstração

Abstrair significa:

> "...o processo ou resultado de generalização por redução do conteúdo da informação de um conceito ou fenômeno observável, normalmente para reter apenas a informação que é relevante para um propósito particular."
> – Fonte: Wikipédia

Em termos mais simples, para o analista ou desenvolvedor de software, abstrair é o processo de identificar o conjunto de características (atributos) e o comportamento (método) que serão necessários para representar uma entidade do mundo real dentro do mundo virtual mediante as regras estabelecidas pelo contexto de negócio analisado.

Por exemplo: digamos que você deseje desenvolver dois softwares, um para um salão de belezas e outro para um banco. Dentro de seus processos de negócio, ambos necessitam controlar as pessoas que frequentam tanto o salão quanto o banco.

Ao dar os primeiros passos para a criação da solução – neste caso estamos falando do entendimento do domínio do negócio –, de imediato vislumbramos em ambos os sistemas a criação de uma entidade denominada `Pessoa`. Sem dúvida você concordará que pessoas frequentam ambos os estabelecimentos, mas o que ainda não sabemos é identificar quais características e qual o comportamento que queremos associados a cada pessoa em cada sistema.

Esse tipo de informação você consegue conversando com o especialista do negócio, normalmente o seu cliente, e conforme as reuniões se aprofundam no tema, melhores são as chances de você definir quais as informações necessárias para representar uma pessoa em ambos os contextos.

Devemos entender essa fase como a de descobrimento dos processos de negócio, como funcionam, o que oferecem de serviços, suas atividades, recursos necessários para completar as atividades etc. Ao final deve-se possuir o entendimento dos atributos e métodos que irão compor as entidades, ou, em termo técnico, as classes.

Esse procedimento, visto pela perspectiva de um arquiteto de software, significa desenvolver um modelo mental da sua aplicação para que você consiga chegar a um modelo explícito de como será (ou deveria ser) a sua solução final.

Continuando com o exemplo, imagino que você já deve ter algumas características e algum comportamento associado às pessoas que frequentam ambos os estabelecimentos. De qualquer forma, vamos construir o raciocínio passo a passo.

Imagine a Renata, uma mulher mãe de dois filhos nos seus 36 anos de idade e que frequenta o salão de beleza "+Mulher". Ela, todo sábado, às 9h da manhã, vai ao salão fazer as unhas, os pés e eventualmente faz o cabelo. Procura sempre sua manicure e cabeleireira preferidas, usa esmalte e xampu de marca apropriada e conversa com as amigas sobre a semana.

Sendo uma profissional de sucesso, faz questão de ajudar a compor a renda familiar. Logo, mensalmente acessa sua conta no banco "+Dinheiro" (são do mesmo dono ☺), paga suas contas, faz suas aplicações, transferências e poupa para anualmente viajar com a família.

Conhecemos a nossa personagem, agora vamos identificar as informações que cada sistema deve armazenar para melhor representá-la. Nesse processo de abstração devemos elencar as informações que são relevantes. Rapidamente conseguimos pensar em características e comportamentos conforme vistos na tabela 1.1.

Tabela 1.1 – Comparativo entre pessoa em um salão e em um banco

Pessoa (salão)	
Características	Comportamento
Nome	Pintar
Endereço	Marcar
Contato	Cortar
Cor do cabelo	Lavar
Cor da pele	Alisar

Pessoa (banco)	
Características	Comportamento
Nome	Transferir
Endereço	Poupar
Contato	Sacar
Agência	Encerrar
Número da conta	Aplicar

Perceba que, apesar de estarmos trabalhando com a mesma entidade `Pessoa`, existirão inúmeras possibilidades de representá-la mediante o contexto de negócio vislumbrado. Uma heurística possível seria identificar as informações requeridas pelas atividades existentes dentro dos processos de negócio nos quais estamos atuando.

Podemos pensar em inúmeras outras informações que ajudem a representar a entidade `Pessoa`, mas o fato que chama atenção é o de que para uma mesma `Pessoa`, informações distintas são delineadas e usadas para representá-la em cada sistema. Essa abordagem deve ser usada no processo de abstração da entidade, e esse é um dos principais conceitos que você deve entender em OO. Logo, o resultado esperado desse processo de abstração é a identificação de um conjunto de entidades dispostas em um modelo denominado "modelo conceitual" e que representará essas entidades como classes de objetos, detalhando a maneira como devem interagir dentro do sistema.

Classes e objetos

Partindo do entendimento de que é necessário construir um modelo conceitual com as entidades identificadas no domínio de negócio da aplicação, precisamos lançar mão de um mecanismo que permita representá-las logicamente em um elemento concreto, visível e manipulável dentro do sistema. Na prática, o que estamos fazendo é agrupar informações e funcionalidades em padrões semânticos que para nós podem ser vistos como *classes de objetos*.

Tecnicamente, uma classe é uma representação computacional de um conjunto de elementos do mundo real que são abstraídos e generalizados dentro do mundo virtual. Cada classe contém características e comportamento que são mais ou menos genéricos, dependendo de como os queremos representados dentro da solução de software.

Uma vez identificadas as características que devemos mapear, há a necessidade de representar as entidades de forma mais precisa. Quero dizer, uma vez que temos o conhecimento genérico de que, para um salão de beleza, precisamos das informações da `Pessoa`, como nome, endereço, cor do cabelo etc., e de que em um banco também precisamos de conta e agência, existe a necessidade de saber codificar esses elementos em uma linguagem de programação.

Por exemplo, como representar a Renata ou a Lúcia ou a Thatiana ou a Lara ou quem quer que frequente o salão de belezas de modo a conseguirmos manipulá-las dentro do sistema? Como diferenciar as características próprias a cada pessoa e entender e mapear o seu comportamento? Como representar e distinguir unicamente cada `Pessoa`?

Para tanto precisaremos de três novos elementos:

- O objeto, a instância que representará uma `Pessoa` em específico.
- Os atributos, que serão responsáveis por armazenar as informações de cada característica.
- Os métodos, que serão responsáveis por permitir a replicação do comportamento associado àquela `Pessoa`.

Tecnicamente, quando instanciamos um objeto a partir de uma classe, estamos na verdade criando cópias em memória da estrutura da classe e preenchendo essas estruturas com valores específicos para cada atributo. No caso da classe `Pessoa`, cópias de sua estrutura são feitas e preenchidas com os valores apropriados, como nome, endereço, contato etc. Ou seja, para representar a pessoa Renata seria necessário preencher as características nome, endereço, contato e outros com os valores específicos a ela. A figura 1.1 apresenta uma visão simplificada desse processo.

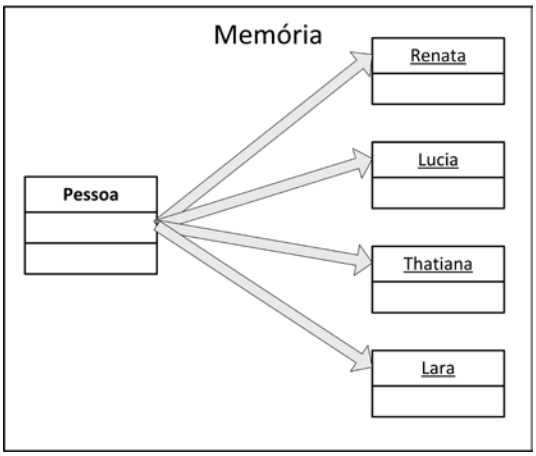

Figura 1.1 – Memória Classe x Objeto.

Para clarear a ideia nada melhor que olharmos o código que teríamos que criar para a classe `Pessoa`. Aproveito então para apresentar um exemplo comparativo, com a descrição de uma classe em Java e outra em Objective-C. Assim, se você já teve algum contato com a linguagem Java, o código a seguir não terá nenhuma novidade.

```
public class Pessoa {

    private String nome;
    private String endereco;
    ...
    public Pessoa() {
        // construtor
    }

    public String getNome() {return nome;}
    public void setNome(String valor) {
        this.nome = valor;
    }

    public String getEndereco() {return endereco;}
    public void setEndereco(String valor) {
        this.endereco = valor;
    }
}
```

Observando a classe em Java notamos dois atributos `nome` e `endereco`, um método construtor e os métodos de acesso. Comparativamente, na linguagem Objective-C a representação da mesma classe `Pessoa` pode causar estranheza. Para representá-la é necessário criar dois arquivos, um arquivo de cabeçalho:

```
@interface Pessoa: Object
{
    NSString *nome;
    NSString *endereco;
}
@property (nonatomic, retain) NSString *nome;
@property (nonatomic, retain) NSString *endereco;
@end
```

e outro arquivo de implementação:

```
@implementation Pessoa
@synthesize nome;
@synthesize endereco;
@end
```

Faça uma pausa e analise bem as diferenças. Tente se acostumar com a sintaxe, pois entender como uma classe é definida em Objective-C lhe trará mais agilidade ao longo do livro. Da mesma forma, o processo de instanciação de objetos difere significativamente. Em Java temos:

- Para a classe `Pessoa.java`:

```
Pessoa p = new Pessoa();
p.nome = "Lara";
p.endereco = "Rua das Nuvens";
...
```

E em Objective-C temos a seguinte sintaxe de instanciação de objetos:

- Para a classe `Pessoa.h/Pessoa.m`:

```
Pessoa *p = [[Pessoa alloc] init];
p.nome = @"Lara";
p.endereco = @"Rua das Nuvens";
...
```

Independente da sintaxe, em ambos os casos o processo de instanciação cria uma representação única, uma identidade para a classe, o que permite a seleção e manipulação do objeto criado na memória.

Atributos

Quando identificamos características estamos identificando os atributos que estarão presentes na classe. Esse mapeamento é importante para entendermos os tipos de dados com os quais trabalharemos. Por exemplo, fomos bem-sucedidos em identificar que as características importantes para o sistema de bancos são nome, endereço, contato, agência, número da conta etc., mas para que o mapeamento em

código funcione precisamos identificar o domínio de valores que cada característica deve representar.

No caso de nome, sabemos que é composto por letras, logo a definição do seu tipo invariavelmente será alguma que consiga tratá-lo da forma mais adequada possível como a classe `String` (Java) ou `NSString` (Objective-C). Por outro lado, o número de conta-corrente de um banco e seu dígito verificador normalmente são tratados como números, e como tal poderão ser representados como `Integer` (Java) ou `NSInteger` (Objective-C).

Além disso, os atributos podem ser definidos como de classe, ou seja, que são acessados por meio da referência à classe. Neste caso estamos falando de atributos estáticos. Podem ser de instância, que só podem ser acessados após a instanciação de objetos da classe, e em algumas linguagens também podem ser finais, que de forma prática indicam que, quando ocorre a primeira atribuição de valor, esse atributo não poderá ser alterado, o que nos leva a entender esse comportamento como a criação de um valor constante em memória.

Métodos

No caso do comportamento temos a representação de uma ação ou resposta de um objeto a estímulos ocorridos em tempo de execução. Também conhecido como mensagens, o comportamento deve ser representado por métodos na classe. Esses métodos possuem sintaxe formal e podem receber informações externas por meio de parâmetros e comunicar o resultado por meio de valores que são retornados pelo método.

Um tipo particular de método são os denominados métodos de acesso. Ação comum em todos os sistemas OO é a manipulação direta ou indireta dos atributos de uma classe. Apesar de não ser elegante e fugir da ideia de encapsulamento, a manipulação direta é comum dentro da classe, quando normalmente ocorre a atribuição de valores diretamente ao atributo. No caso da manipulação indireta temos a criação de métodos que "protegem" o acesso externo a esses atributos. Afinal, ninguém quer correr o risco de que em algum momento o número da conta-corrente de uma pessoa no sistema de controle bancário possa conter o nome do correntista.

Essa proteção é possível com a criação de métodos de acesso que, dependendo da linguagem, possuem características que o diferenciam dos outros métodos, como a inclusão de prefixos como "get/set". Veremos mais sobre métodos de acesso no próximo capítulo.

Além desses, temos um método muito importante denominado construtor. Este é fundamental para o ciclo de vida da classe, pois, como o nome diz, é responsável

por construir o objeto em tempo de execução. Os trechos de código em negrito demonstram como esses métodos devem ser definidos e utilizados.

Exemplo em Java:

```
// Definição do construtor
public class Pessoa {
    public Pessoa() {
        // construtor
    }
}
// Chamada ao método para construção do objeto
Pessoa p = new Pessoa ();

//--------------------------
```

Exemplo em Objective-C:

```
// Definição do construtor em Objective-C
@implementation Pessoa
- (id) init
{
self =[super init];
    if (self != nil)
    {
        // código…
    }
    return self;
}
@end

/* O processo normal de construção ou instanciação de um objeto em Objective-C é o de
   executar o método alloc para reserva de memória e em seguida o init para sua inicialização */
Pessoa *p [Pessoa alloc] init];
```

Atenção! Por padrão, o nome dos métodos construtores em Objective-C são sempre precedidos pela palavra `init`. Supondo que quiséssemos adicionar um novo construtor à classe `Pessoa`, de maneira a inicializá-la com o seu nome, escreveríamos da seguinte forma:

```
@implementation Pessoa
- (id) initWithName: (NSString *) pnome
{
    self =[super init];
    if (self != nil)
    {
        self.nome = pnome;
    }
```

```
    return self;
}
@end
```

Mensagens

Para que ocorra interação (ou colaboração) entre os objetos é necessário que haja um mecanismo de comunicação eficiente e de fácil implementação, de forma que o fluxo de informação existente no processo de negócio real seja replicado dentro da solução de software. Nesse caso utilizamos o mecanismo de troca de mensagens para viabilizar o fluxo de informações ao longo do tempo de execução do sistema.

Um envio de mensagem, essencialmente, deve possuir:

- um emissor;
- um receptor;
- um seletor de mensagens (nome do método);
- parâmetros (opcionais);
- e uma mensagem pode retornar um valor.

Em Objective-C essa abordagem de troca de mensagens é muito bem definida, pois há uma diferenciação sintática entre métodos de comportamento ou mensagens e os métodos que dão acesso aos atributos da classe. Quando enviamos uma mensagem a algum objeto, devemos conhecer sua identidade (ou seja, estar de posse da sua referência em memória) para poder enviar tal mensagem. O exemplo a seguir apresenta essa diferenciação.

Seja a instanciação de um objeto da classe `Pessoa`:

```
Pessoa * p = [[Pessoa alloc] init];
```

Para enviarmos uma mensagem a essa pessoa solicitando que "ande", fazemos da seguinte forma:

```
/* Deve-se utilizar colchetes e enviar a mensagem ao objeto referenciado por p com um
   espaço entre a referência e o método */

[p andar];
```

A sintaxe é estranha, mas com o tempo você se acostuma.

Finalmente, ainda existem outros tipos de métodos, como os abstratos, que delegam sua implementação às classes herdeiras (veremos sobre herança mais adiante), ou seja, as subclasses são obrigadas a implementar esse método; e métodos finais, que não podem ser redefinidos ao longo da árvore hierárquica, ou seja, não podem ser redefinidos pelas subclasses.

Encapsulamento

Princípio importante em programação OO, o encapsulamento determina que a implementação de um objeto só permitirá acesso aos seus atributos por meio de uma interface visível e bem definida, ou seja, por meio de métodos que certifiquem a atribuição adequada de valores conforme as regras de negócio da aplicação.

Se quiséssemos visualizar esse princípio, teríamos algo como o definido pela figura 1.2, em que os métodos criam uma barreira, uma cerca de proteção em torno dos atributos. Isso obriga elementos do mundo externo a manipular indiretamente os atributos. Logo, consultar, alterar e excluir valores somente seria possível pela chamada aos métodos de acesso.

Figura 1.2 – Encapsulamento.

Um benefício explícito dessa técnica é o desacoplamento entre objetos, pois não há conhecimento de como ocorre a implementação da classe, permitindo que mudanças nas regras de negócio não afetem o relacionamento com outros objetos e, por consequência, ao isolar o conteúdo da classe, aumenta-se a capacidade de evolução e manutenção do sistema.

Visibilidade

Como mecanismo de auxílio ao encapsulamento, a visibilidade é definida por meio de palavras-chave que impõem como um objeto deve (ou pode) acessar recursos

de outros objetos. O próprio conceito de encapsulamento nos ajuda a entender como utilizar esse mecanismo.

Também denominados modificadores de acesso, temos como recurso as palavras-chave:

- `public`
- `protected`
- `private`

Seguindo a definição da UML, o modificador `public` determina que um recurso esteja acessível a qualquer tipo de manipulação externa. Já o modificador `protected` define que um determinado recurso pode também ser acessado pelos seus filhos dentro de uma árvore de herança, e o modificador `private` permite apenas acesso ao recurso quando feito de dentro da própria classe.

Interfaces

Uma interface é um artifício da linguagem que define um comportamento específico, um protocolo de troca de mensagens que impõe à classe a implementação daquele comportamento. Esses métodos definidos na interface são obrigatoriamente abstratos, pois não possuem implementação, deixando a cargo da classe proceder com a codificação.

Em Objective-C, interfaces são vistas como protocolos. Comparativamente à linguagem Java, protocolos são criados como forma de definir padrões de comunicação que são necessários conforme imposições das regras de negócio. Veremos mais sobre protocolos em Objective-C no próximo capítulo.

Herança

O mecanismo de herança me remete a uma história interessante. Ano passado, ao explicar o conceito em uma de minhas aulas da disciplina de Arquitetura de Software do MBA da Escola Politécnica da UFRJ (*www.poli.ufrj.br*), ouvi de um aluno que a pior coisa que existia ao programar orientado a objetos era o mecanismo de herança. Confesso que recebi essa informação com bastante surpresa. Ora! Como uma pessoa que programa utilizando uma linguagem OO diz com tanta veemência que herança é uma característica ruim e que não deve ser utilizada? Após alguma discussão em sala, conseguimos chegar às razões que levavam o aluno a pensar dessa forma, o que no final somente me fez reafirmar meus conceitos de que herança é uma excelente forma de você deixar o seu código organizado e preparado para o reuso.

Caso você não saiba o que o termo significa, eu explico. Voltemos ao exemplo do salão de beleza. Qualquer arquiteto de software que queira criar uma solução para um determinado problema definirá algumas classes que, apesar de semelhantes, possuem comportamento especializado. Quero dizer, tomando como base a entidade `Pessoa`, observamos um mapeamento semântico genérico capaz de representar todos os tipos de pessoas que frequentam ou trabalham no salão.

Na prática, entretanto, para representar todos os tipos de pessoas relacionados aos processos de negócio do salão, a classe `Pessoa` acabaria contendo muitos atributos e métodos que não seriam completamente utilizados de acordo com o tipo de pessoa observada em um dado momento. Para resolver isso há a necessidade de dividir a classe, ou seja, no caso do salão a generalização associada à classe `Pessoa` teria especializações definidas em classes como as de `Empregado` e `Cliente`. Ambas representam pessoas, mas cada uma possui características distintas que não fazem sentido se pensadas genericamente, ou seja, se postas na classe `Pessoa` em conjunto.

Nesse caso temos dois momentos, o de generalização e o de especialização de conceitos. No primeiro temos a capacidade de identificar um comportamento comum e fatorá-lo de forma a reutilizá-lo. No segundo temos a capacidade de representar de forma mais completa e específica características e comportamentos para todas as entidades que estão envolvidas no processo de negócio.

Em ambos os momentos é possível observar como a herança nos ajuda. Há um processo de organização estrutural hierárquica que permite a reutilização dos conceitos atribuídos à classe pai em direção às classes filhas. De forma mais técnica, estamos falando de uma classe se tornar subclasse e agregar informação a cada nível hierárquico associado, como apresentado na figura 1.3.

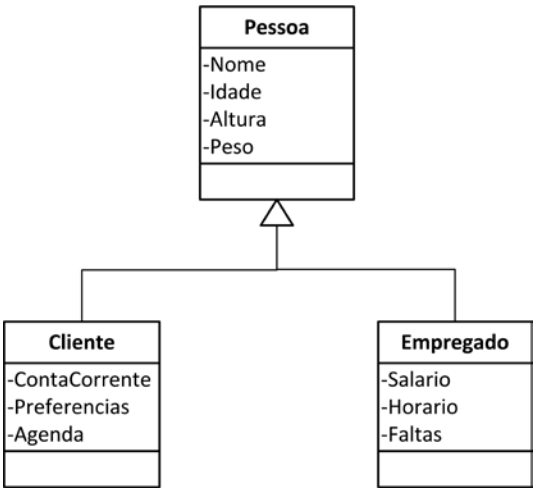

Figura 1.3 – Exemplo de herança.

A herança nos permite reaproveitar características (atributos) e comportamentos (métodos). O interessante nessa história é que o esforço despendido para criar a classe genérica é diluído conforme ocorre a sua (re)utilização, pois para cada subclasse que a utiliza há uma economia em se tratando de custo e tempo de desenvolvimento.

É claro que nem tudo são flores. Como bem definido no meu livro *SOA na Prática*, não existe esse negócio de "ganha-ganha" em desenvolvimento de software. Há sempre a necessidade de se negociar o que em um momento ganharemos e em outro perderemos. Em herança temos o mesmo dilema, e apesar de ser algo extremamente inteligente e indispensável em programação OO, pode nos trazer problemas, principalmente conforme aumenta a profundidade da árvore de herança. Uma vez que tenhamos uma profundidade muito grande da raiz até as folhas, conforme apresentado na figura 1.4, na ocorrência de um erro que impossibilite um objeto no meio da árvore de ser acessado em tempo de execução, todos aqueles que herdam suas características também ficarão inacessíveis.

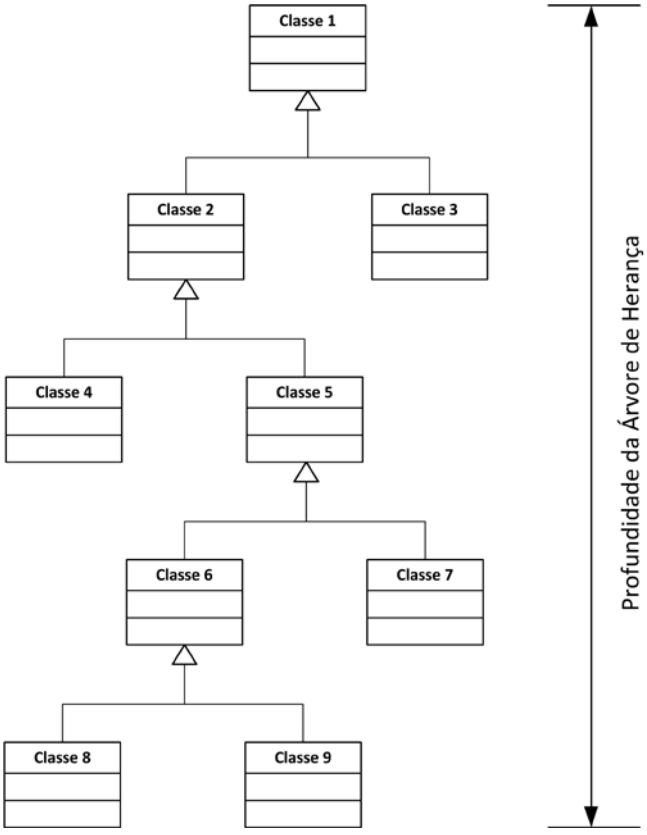

Figura 1.4 – Árvore hierárquica.

Pelo que concluí da discussão em sala, esse, a meu ver, é o principal problema que existe na utilização de herança. Você talvez possa enumerar alguns outros, e se quiser pode até me enviar (quem sabe eu os coloco em futuras versões deste livro), mas isso não justifica não utilizá-la, uma vez que os benefícios quando da definição da arquitetura do sistema e a sua consequente evolução superam tais problemas.

Para completar, temos dois tipos de herança, que são a simples e a múltipla. A simples determina que uma subclasse só pode herdar diretamente de uma única classe, sendo esse comportamento bem definido nas maioria das linguagens OO "modernas" (o que também é o caso de Objective-C). No caso da herança múltipla, uma subclasse pode herdar de várias classes. Em um primeiro momento isso é interessante, mas pode causar uma série de efeitos colaterais que podem tornar complicada a identificação do comportamento que desejamos para a subclasse.

Polimorfismo

Polimorfismo é um mecanismo que ajuda a redefinir e reaproveitar o comportamento de uma classe. Tecnicamente, estamos falando de "overriding", em que o conteúdo de um método herdado é substituído, mas seu nome é mantido, e "overloading", em que também há a manutenção do nome do método, mas com diferenças estruturais de implementação. O exemplo clássico do animal que anda ajuda a entender o polimorfismo. Todo animal anda, mas homens e cachorros andam de formas distintas. Para que esse comportamento esteja replicado dentro do sistema o arquiteto concebe uma classe denominada `Animal` que possui um método `andar()` e duas outras classes, `Pessoa` e `Cachorro`, que também possuirão o mesmo método `andar()`. No entanto as implementações internas deverão conter lógica de programação que se adéque ao tipo de andar que cada um necessita. Esse conceito é mais bem explorado no item sobre polimorfismo em Objective-C no próximo capítulo.

Estilo arquitetural Model-View-Controller (MVC)

Apesar de não ser um conceito clássico de orientação a objetos, eu aproveito este capítulo para apresentar o estilo arquitetural Model-View-Controller (MVC). Ele é um mecanismo extremamente relevante para o desenvolvimento de aplicações para iOS, e você precisa entendê-lo se deseja programar corretamente.

Um estilo arquitetural determina uma abordagem para composição estrutural de um sistema. Ele define um padrão com o qual o arquiteto deve se orientar para compor, por meio de uma heurística, seus componentes e então conectá-los.

No MVC essa heurística é vista por meio da separação semântica que é atribuída aos componentes que possuem um determinado propósito. Estamos falando da divisão em camadas definidas por:

- representação ou modelo;
- controle;
- apresentação ou visão.

A camada de representação nos permite isolar o domínio do problema no qual estamos trabalhando. Desenvolver sistemas significa representar processos, atividades, regras, entre outros, do mundo real para o virtual, e para tanto precisamos entender como suas entidades se relacionam. Você deve pensar em como as informações relevantes a esses processos, regras e atividades serão representados no sistema e como iremos acessá-los.

Quando falamos em controle pensamos em um núcleo autocontido e desacoplado que tem o objetivo de regular como ocorre o fluxo de informações que trafega dentro da aplicação. Nesse caso o controlador regula o fluxo de informação do modelo para a apresentação ou visão. Esse núcleo, no meu entender, é responsável não só por efetuar os repasses de dados, mas também por transformá-los e direcioná-los àqueles que têm permissão de vê-los.

Finalmente, a apresentação é responsável por expor aos usuários as informações que são representadas pelos modelos e transformadas pelos controladores. Se bem desenvolvida, sua aplicação pode ser acoplada a inúmeras formas de apresentação, bastando que o controlador a reconheça e passe a fornecer as informações solicitadas.

Toda arquitetura de software deve ser pensada em termos de componentes, conectores e de um conjunto de restrições que determinam regras de uso desses elementos. Sendo esse o caso, poderíamos pensar o MVC da seguinte maneira:

- Componentes:
  - Modelo.
  - Visão.
- Conector:
  - Controle.
- Restrições
  - A visão e o controle se conhecem mutuamente.
  - O modelo não deve conhecer suas visões.
  - O modelo não deve conhecer seus controladores.

O estilo MVC é conhecido por sua excelente capacidade de organizar os componentes do sistema dentro de um padrão estrutural que permite escalabilidade sem impor esforço desnecessário em alterações e evoluções. Essencialmente, o MVC possui algumas vantagens que são preponderantes em qualquer projeto de software, a saber:

- Podem ser criadas diversas visões para um mesmo modelo.
- Como consequência do item anterior, permite a criação de interfaces mais sofisticadas para interação com o usuário.
- As responsabilidades da aplicação são objetivamente divididas.
- Permite um crescimento consistente da aplicação.
- Permite a distribuição dos elementos que compõem a aplicação.

Não é à toa que o estilo MVC é tão difundido. São poucas as suas desvantagens. Na verdade, a única que me parece mais relevante é em relação a sua utilização em aplicações pequenas ou muito simples, pois pode criar um número desnecessário de componentes. Todavia, ela se torna uma vantagem no momento em que a aplicação precisa crescer e agregar novos conceitos e regras.

É importante que você tenha o completo domínio desse estilo, porque tudo que você irá desenvolver dentro da plataforma iOS será utilizando o MVC. Em praticamente todos os templates que você utilizar, conexões que fizer entre componentes no "Interface Builder" e formulários que criar no Xcode você estará utilizando explícita ou implicitamente o MVC.

Modelo MVC no iOS

No iOS o modelo MVC possui algumas peculiaridades interessantes. De início parece que estamos trabalhando com um modelo MVC comum, mas existem algumas nuances que precisam ser muito bem entendidas para que a programação ocorra sem dificuldades.

Como visto na figura 1.5, em um primeiro momento o MVC possui uma divisão natural entre controlador, modelo e visão. No entanto, perceba que há uma diferenciação entre as fronteiras, deixando claro que modelo e visão não devem se conhecer. Todo o processo de troca de mensagens ocorre por intermédio do controlador.

Toda e qualquer interação com o usuário é feita por intermédio da visão. Logo, deve haver uma maneira de informar ao controlador o que o usuário deseja. Nesse momento temos dois elementos que entram em cena: os "outlets" e os alvos (targets), vistos na figura 1.6.

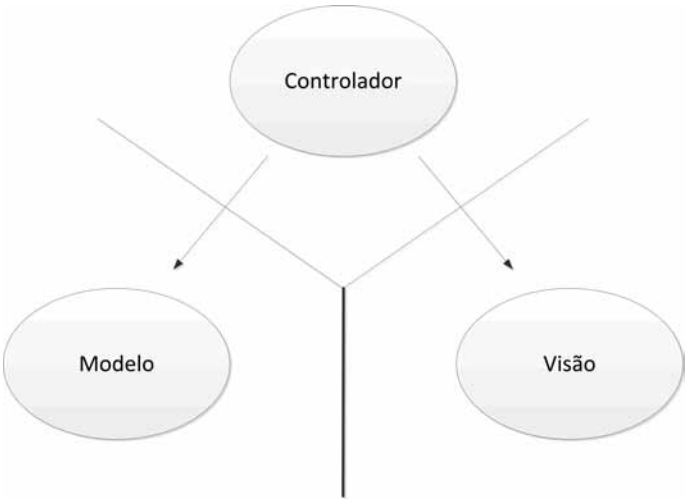

Figura 1.5 – Modelo MVC no iPhone.

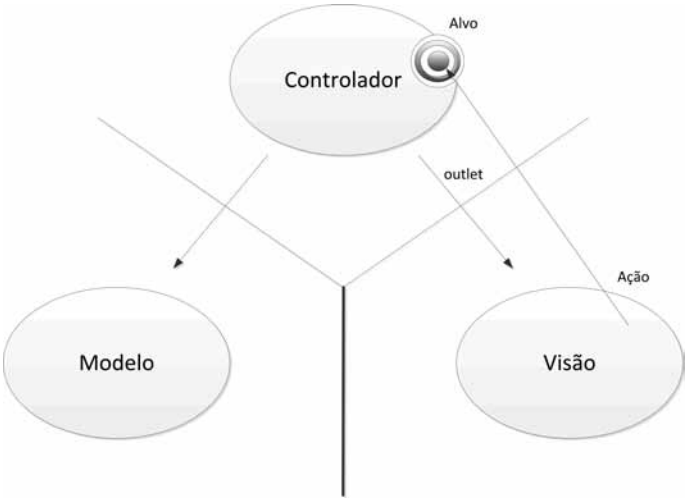

Figura 1.6 – O MVC no iOS: ações, alvos e outlets (adaptado das aulas do prof. Hegarty).

Outlets e targets são elementos abstratos que nos oferecem formas de conexão fracamente acopladas entre controlador, visão e modelo. Um outlet é um canal de comunicação que você cria entre controlador e um componente na visão, como um botão, um rótulo, uma listagem. Já o alvo, ou target, é um ponto fixado no controlador que é dado à visão para notificá-lo de uma interação com o usuário. É como se o controlador criasse um alvo em si próprio e dissesse para a visão mirá-lo quando quisesse enviar alguma mensagem. O controlador cria o alvo e o repassa à visão como uma forma de envio de mensagens. Essa comunicação é transparente e estruturada e não oferece risco de acoplamento entre os componentes.

Esses elementos permitem a sincronização da visão com o controlador e informam ao controlador quando sincronizar com o modelo. Por exemplo, você apaga um item de uma listagem e quer que essa modificação seja refletida no modelo. É por intermédio dos outlets e dos targets que ocorre sincronismo entre visão e controlador, auxiliados por um terceiro mecanismo representado pelo padrão de projeto delegate.

O padrão de projeto delegate determina uma abordagem de programação em que um determinado objeto repassa a execução de suas tarefas a outro objeto. Nesse momento ocorre o que chamamos de inversão de responsabilidade, cabendo ao segundo objeto executar as ações a ele delegadas.

O delegate é usado para garantir que a interação entre visão e controlador seja transparente e que a sincronização não obrigue um acoplamento entre as partes. Tecnicamente, o controlador é sempre cadastrado como delegate da visão, o que é feito via um protocolo seguro que impõe ao controlador a implementação de uma série de métodos que serão usados pelas visões para efetuar a comunicação.

O caminho inverso também é importante de ser entendido. Para apresentarmos as informações contidas no modelo também precisamos do controlador para repassá-las à visão. Esse procedimento é feito por meio da implementação de outro protocolo, o data source. Perceba que, apesar de o modelo ser o dono da informação, são os controladores que implementam os métodos de envio de dados às visões. Os controladores sempre fazem o papel de data source, da mesma forma que sempre fazem o papel de delegates (Figura 1.7).

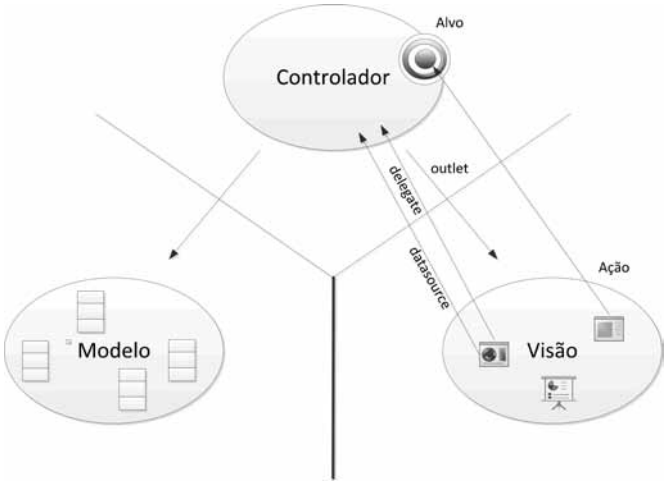

Figura 1.7 – O MVC no iPhone: delegates e data sources (adaptado das aulas do prof. Hegarty).

Veremos mais sobre este conceito no exemplo do próximo capítulo em que você criará um aplicativo denominado "Good Bye Steve". Portanto, não se preocupe e não desanime. Estamos apenas começando.