EXPLICADORES.NET

2023

EAGS SIN

POO

Prof. Camila Dias

PROGRAMAÇÃO ORIENTADA A OBJETO

Bibliografia e conteúdo

SINTES, Anthony.Aprenda Programação Orientada a Objeto em 21 Dias.
São Paulo:
Makron Books, 2002.

PROGRAMAÇÃO ORIENTADA A OBJETOS
Introdução à programação orientada a objetos.

- Encapsulamento.
- Método.
- Propriedades.
- Construtores.
- Herança.
- Polimorfismo.
- Introdução à UML.
- Introdução à Análise Orientada a Objetos.
- Introdução ao Projeto Orientado a Objetos.
- Reutilizando projetos através de padrões de
- projeto.
- Padrões avançados de projeto.
- OO e programação de interface com o usuário.
- Construindo software confiável através de testes.
- Prática da orientação a objetos.

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática

Representação de uma Classe

PESSOA
+ Id:int() + Nome:String(50) + End:String(50)
+CadastrarPessoa() +ListarPessoa() +ExcluirPessoa()

No código:

JAVA:

```
public class Pessoa {
 public String Id;
 public String nome;
 public String end;

}
```

Na prática

PROGRAMAÇÃO ORIENTADA A OBJETO

ENCAPSULAMENTO

No código: O uso do private Determina o conceito de Encapsulamento

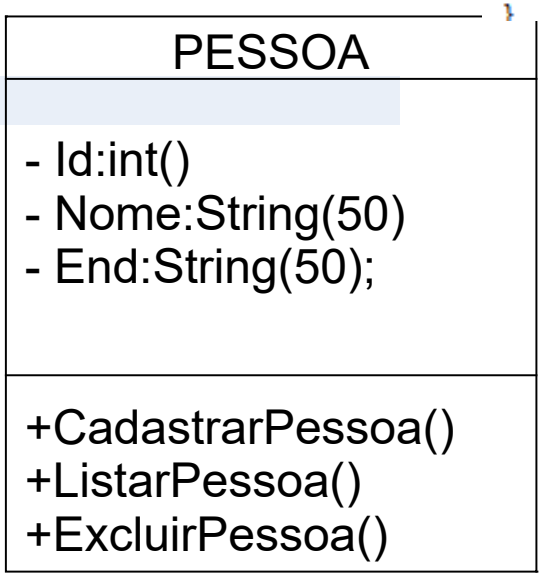
JAVA:

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;

}
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();

    }
}
```



PROGRAMAÇÃO ORIENTADA A OBJETO

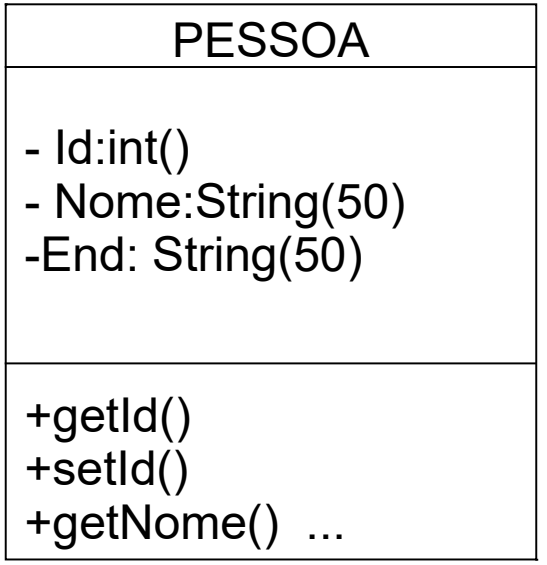
Na prática

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

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

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();
        p.setNome("Ana");

        System.out.println("Seja bem vinda" + p.getNome());
    }
}
```



PROGRAMAÇÃO ORIENTADA A OBJETO

Uma classe em PHP

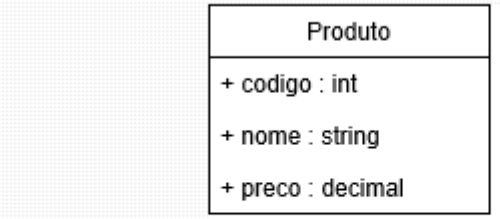


Figura 1. Diagrama da classe Produto

Nome da classe

```php
<?php
public class Produto
{
    private $codigo;
    private $nome;
    private $preco;

    public getCodigo() { return $this->codigo; }
    public setNome($nome) { $this->nome = $nome; }
}
```

Atributos

Getters,Setters

PROGRAMAÇÃO ORIENTADA A OBJETO

```php
<?php

class Produto {

    private $codigo;
    private $nome;
    private $preco;

    public function getCodigo()
    {
        return $this->codigo;
    }
    public function setCodigo($codigo)
    {
        $this->codigo = $codigo;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nome)
    {
        $this->nome = $nome;
    }
    public function getPreco()
    {
        return $this->preco;
    }
    public function setPreco($preco)
    {
        $this->preco = $preco;
    }
}
```

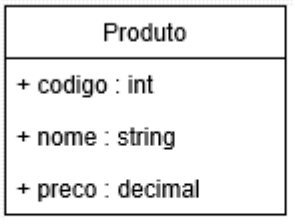


Figura 1. Diagrama da classe Produto

PROGRAMAÇÃO ORIENTADA A OBJETO

PESSOA
- Id:int() - Nome:String(50) - End:String(50)
CadastrarPessoa() ListarPessoa() ExcluirPessoa()

```php
<?php

class Pessoa {

    private $id;
    private $nome;
    private $end;

    public function getId()
    {
        return $this->Id;
    }
    public function setId($id)
    {
        $this->id= $id;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nome)
    {
        $this->nome = $nome;
    }
    public function getEnd()
    {
        return $this->end;
    }
    public function setEnd($end)
    {
        $this->end = $end;
    }
}
```

PROGRAMAÇÃO ORIENTADA A OBJETO

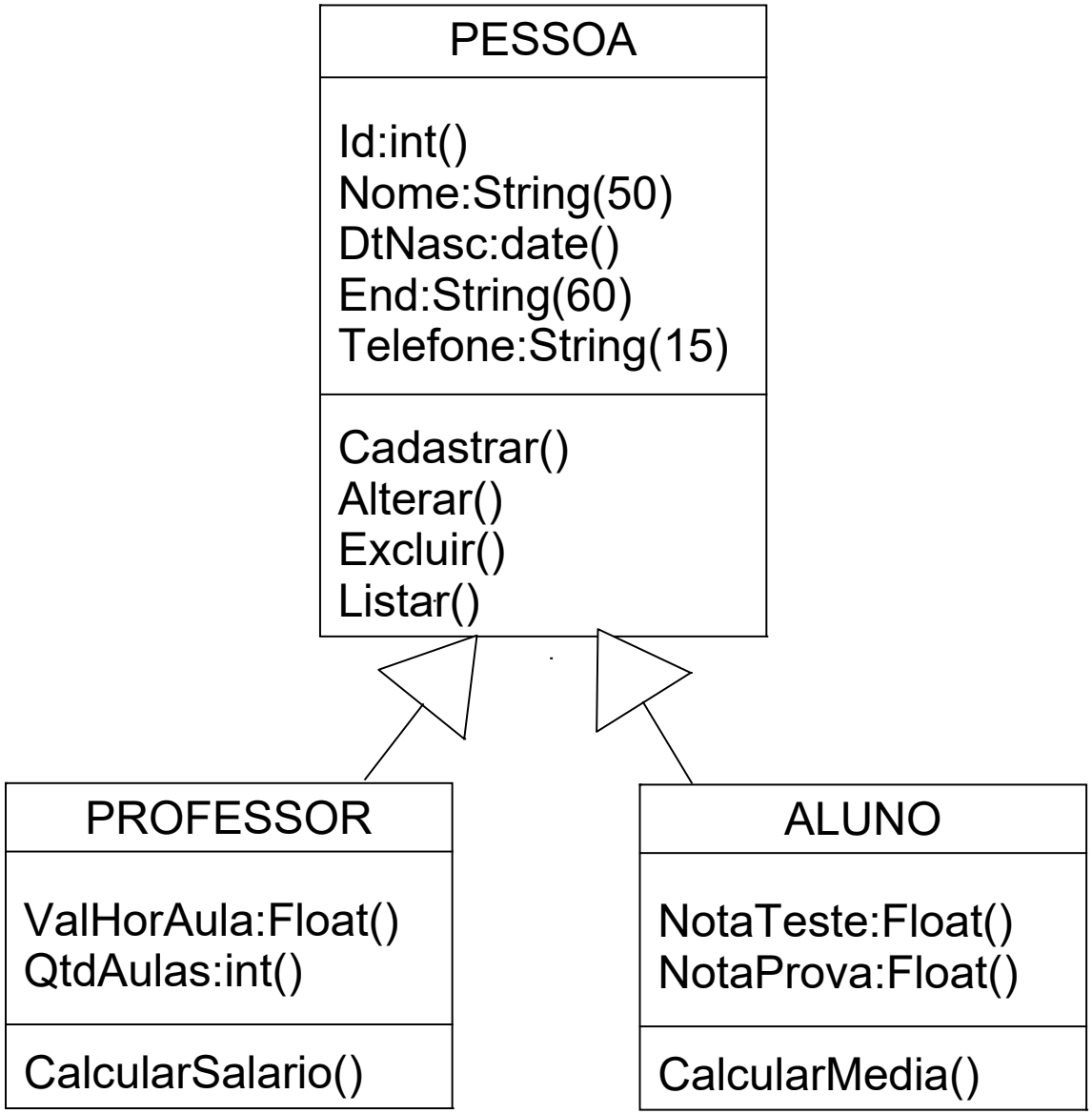
PESSOA
- Id:int() - Nome:String(50) - End: String(50)
CadastrarPessoa() ListarPessoa() ExcluirPessoa()

```php
<?php

class Pessoa {

    private $id;
    private $nome;
    private $end;

    public function getId()
    {
        return $this->Id;
    }
    public function setId($id)
    {
        $this->id= $id;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nome)
    {
        $this->nome = $nome;
    }
    public function getEnd()
    {
        return $this->end;
    }
    public function setEnd($end)
    {
        $this->end = $end;
    }
}
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática

HERANÇA



PROGRAMAÇÃO ORIENTADA A OBJETO

Código em JAVA:

Pessoa.Java

```java
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

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

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

Professor.Java

```java
public class Professor extends Pessoa{
    private double valorHoraAula;
    private int qtdAulas;


    public double getValorHoraAula() {
        return valorHoraAula;
    }

    public void setValorHora(double valorHoraAula) {
        this.valorHoraAula = valorHoraAula;
    }

    public int getQtdAula() {
        return qtdAulas;
    }

    public void setQtdAula(int qtdAulas) {
        this.qtdAulas = qtdAulas;
    }


}
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Código em PHP:

Pessoa.php

```php
<?php

class Pessoa {

    private $id;
    private $nome;
    private $end;

    public function getId()
    {
        return $this->Id;
    }
    public function setId($id)
    {
        $this->id= $id;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nome)
    {
        $this->nome = $nome;
    }
    public function getEnd()
    {
        return $this->end;
    }
    public function setEnd($end)
    {
        $this->end = $end;
    }
}
```

Professor.php

```php
<?php

  class Professor extends Pessoa {

      private $valorHoraAula;
      private $qtdAulas;

      public function getValorHoraAula()
      {
          return $this->valorHoraAula;
      }
      public function setValorHoraAula($valorHoraAula)
      {
          $this->valorHoraAula = $valorHoraAula;
      }
     public function getQtdAulas()
      {
          return $this->qtdAulas;
      }
      public function setQtdAulas($valorqtdAulas)
      {
          $this->qtdAulas = $qtdAulas;
      }
```

PROGRAMAÇÃO ORIENTADA A OBJETO

POLIMORFISMO

Polimorfismo

- Princípio pelo qual as instâncias de duas classes ou mais classes derivadas de uma mesma super-classe podem invocar métodos com a mesma assinatura, mas com comportamentos distintos.

Se o encapsulamento e a herança são os socos um e dois da POO, o polimorfismo é o soco para nocaute seguinte. Sem os dois pilares, você não poderia ter o polimorfismo, e sem o polimorfismo, a POO não seria eficaz. O polimorfismo é onde o paradigma da programação orientada a objetos realmente brilha e seu domínio é absolutamente necessário para a POO eficaz.

Polimorfismo significa muitas formas. Em termos de programação, o polimorfismo permite que um único nome de classe ou nome de método represente um código diferente, selecionado por algum mecanismo automático. Assim, um nome pode assumir muitas formas e como pode representar código diferente o mesmo nome pode representar muitos comportamentos diferentes.

PROGRAMAÇÃO ORIENTADA A OBJETO

Conceitos sobre Polimorfismo:

1- De inclusão
2- Paramétrico
3- Sobreposição *(Sobrescrita , overwrite)
4- Sobrecarga *(Overhead)

O Polimorfismo de inclusão, às vezes chamado de polimorfismo puro, permite que você trate objetos relacionados genericamente.

Também associado a Sobreposição ou sobrescrita

O Polimorfismo paramétrico permite que você crie métodos e tipos genéricos. Assim como o polimorfismo de inclusão, os métodos e tipos genéricos permitem que você codifique algo uma vez e faça isso trabalhar com muitos tipos diferentes de argumentos.
Também associado a sobrecarga.

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRESCRITA**

JAVA:

```java
public class Professor extends Pessoa{
    private double valorHoraAula;
    private int qtdAulas;

    public String getNome() {
        System.out.println("Olá professor: "+nome);
        return nome;
    }

    public double getValorHora() {
        return valorHoraAula;
    }

    public void setValorHora(double valorHoraAula) {
        this.valorHoraAula = valorHoraAula;
    }

    public int getQtdAula() {
        return qtdAulas;
    }

    public void setQtdAula(int qtdAulas) {
        this.qtdAulas = qtdAulas;
    }
```

```java
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {
        System.out.println("Olá aluno: "+nome);
        return nome;
    }
    public double getNota1() {
        return nota1;
    }

    public void setNota1(double nota1) {
        this.nota1 = nota1;
    }

    public double getNota2() {
        return nota2;
    }

    public void setNota2(double nota2) {
        this.nota2 = nota2;
    }
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

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

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();
        p.setNome("Ana");

        System.out.println("Seja bem vinda" + p.getNome());
    }
}
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {...4 linhas }
    public double getNota1() {...3 linhas }

    public void setNota1(double nota1) {...3 linhas }

    public double getNota2() {...3 linhas }

    public void setNota2(double nota2) {...3 linhas }


    public void imprimir(){
        System.out.println("Nome: "+ nome);
        System.out.println("Endereço: "+ end);
        System.out.println("Idade: "+ idade);
        System.out.println("Nota1: "+ nota1);
        System.out.println("Nota2: "+ nota2);

    }

    public void imprimir(double nota1, double nota2){
        double md= (nota1+nota2)/2;
        System.out.println("A média é:"+ md);
    }
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
10
11      public static void main(String[] args) {
12
13
14          Aluno a = new Aluno();
15
16
17          a.setNome("Matheus");
18          a.setEnd("Rua suvaco da minhoca");
19          a.setIdade(19);
20          a.setNota1(7.9);
21          a.setNota2(10.0);
22
23          a.imprimir();
24
25
26      }
27
```

Sem nenhum parâmetro!!!

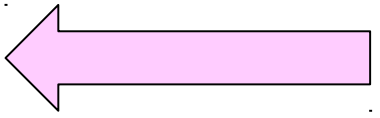
```
Saída -
Nome: Matheus
Endereço: Rua suvaco da minhoca
Idade: 19
Nota1: 7.9
Nota2: 10.0
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {...4 linhas }
    public double getNota1() {...3 linhas }

    public void setNota1(double nota1) {...3 linhas }

    public double getNota2() {...3 linhas }

    public void setNota2(double nota2) {...3 linhas }


   public void imprimir(){
        System.out.println("Nome: "+ nome);
        System.out.println("Endereço: "+ end);
        System.out.println("Idade: "+ idade);
        System.out.println("Nota1: "+ nota1);
        System.out.println("Nota2: "+ nota2);

   }

   public void imprimir(double nota1, double nota2){
        double md= (nota1+nota2)/2;
        System.out.println("A média é:"+ md);
   }
```

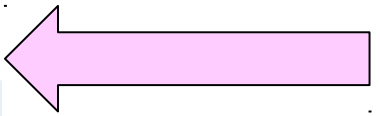


PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
11      public static void main(String[] args) {
12
13
14          Aluno a = new Aluno();
15
16
17          a.setNome("Matheus");
18          a.setEnd("Rua suvaco da minhoca");
19          a.setIdade(19);
20          a.setNota1(7.9);
21          a.setNota2(10.0);
22
23          a.imprimir(a.getNota1(), a.getNota2());
24
25
26      }
27
```



Com 2 parâmetros!!!

```
Saída -
run:
A média é:8.95
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Para o uso de INTERFACES em PHP usamos a palavra reservada: **IMPLEMENTS**

```php
<?php

  class Assinatura extends Produto implements Expiravel {

      private $dataExpiracao;

      public function getDataExpiracao()
      {
          return $this->dataExpiracao;
      }
      public function setDataExpiracao($dataExpiracao)
      {
          $this->dataExpiracao = new \DateTime($dataExpiracao);
      }
      public function getTempoRestante()
      {
          return $this->dataExpiracao->diff(new \DateTime());
      }
  }
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Para o uso de INTERFACES em PHP usamos a palavra reservada: **IMPLEMENTS**

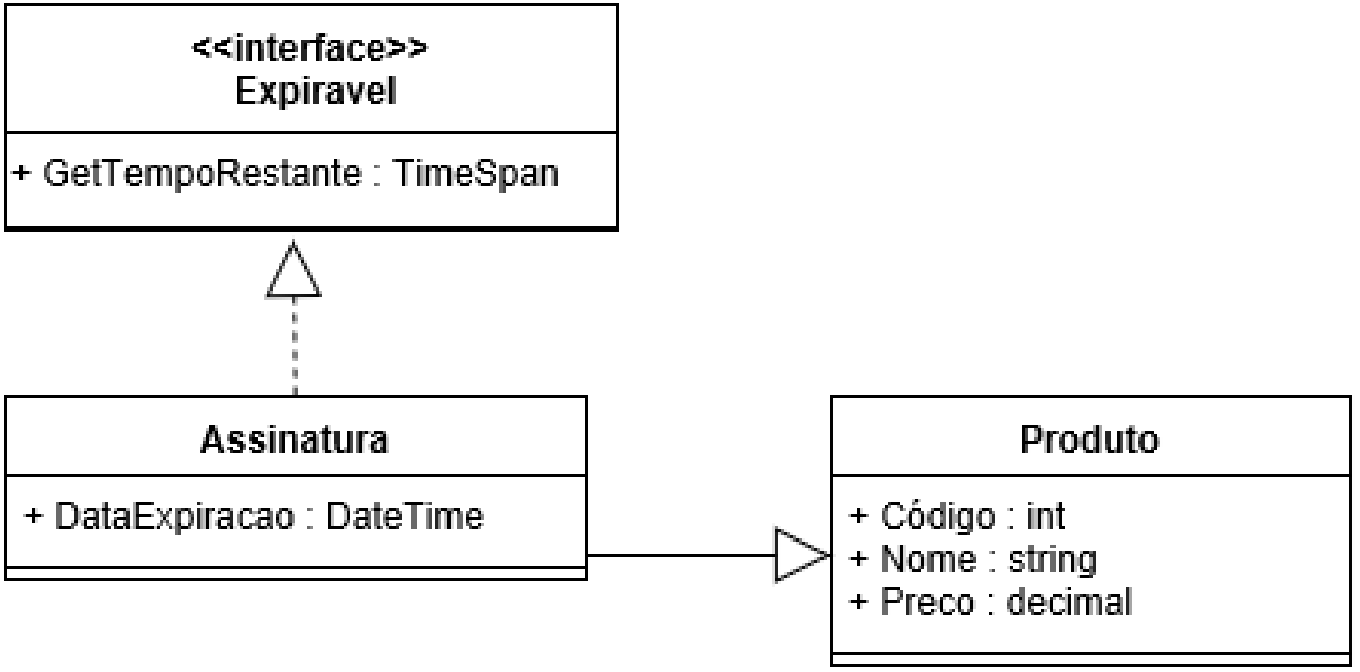


Figura 3. Diagrama de classes com interface

PROGRAMAÇÃO ORIENTADA A OBJETO

Mais exemplos práticos sobre POO:

https://www.devmedia.com.br/sobrecarga-e-sobreposicao-de-metodos-em-orientacao-a-objetos/33066

PROGRAMAÇÃO ORIENTADA A OBJETO

Bibliografia e conteúdo

SINTES, Anthony.Aprenda Programação Orientada a Objeto em 21 Dias.
São Paulo:
Makron Books, 2002.

PROGRAMAÇÃO ORIENTADA A OBJETOS
Introdução à programação orientada a objetos.

- Encapsulamento.
- Método.
- Propriedades.
- Construtores.
- Herança.
- Polimorfismo.
- **Introdução à UML.**
- Introdução à Análise Orientada a Objetos.
- Introdução ao Projeto Orientado a Objetos.
- Reutilizando projetos através de padrões de
- projeto.
- Padrões avançados de projeto.
- OO e programação de interface com o usuário.
- Construindo software confiável através de testes.
- Prática da orientação a objetos.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Basicamente, UML (Unified Modeling Language) é uma **linguagem de notação** (um jeito de escrever, ilustrar, comunicar) para uso em projetos de sistemas.

Esta linguagem é expressa através de diagramas. Cada **diagrama** é composto por **elementos** (formas gráficas usadas para os desenhos) que **possuem relação** entre si.

Os diagramas da UML se dividem em dois grandes grupos: diagramas **estruturais** e diagramas **comportamentais**.

Diagramas estruturais devem ser utilizados para especificar **detalhes da estrutura do sistema** (parte estática), por exemplo: classes, métodos, interfaces, namespaces, serviços, como componentes devem ser instalados, como deve ser a arquitetura do sistema etc.

Diagramas comportamentais devem ser utilizados para especificar **detalhes do comportamento do sistema** (parte dinâmica), por exemplo: como as funcionalidades devem funcionar, como um processo de negócio deve ser tratado pelo sistema, como componentes estruturais trocam mensagens e como respondem às chamadas etc.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

UML deixa as coisas claras

UML ajuda muito a deixar o escopo claro, pois centraliza numa única visão (o diagrama) um determinado conceito, utilizando uma linguagem que todos os envolvidos no projeto podem facilmente entender.

Mas ajuda desde que **utilizada na medida certa**, ou seja, apenas **quando realmente é necessário**.

O maior problema na produção de software, a maior dor, em qualquer país do mundo, chama-se **comunicação ruim**.

Vejamos um rápido exemplo **didático** de como se dá a comunicação em equipes de produção de software:

> *\/* João quer A, explica à equipe algo "parecido" com B. Marcos entende que João quer C, e explica para Claudia que é para fazer D. Claudia faz um "D que mais se parece um E", e entrega um "meio E" para João. E João queria um A... */*

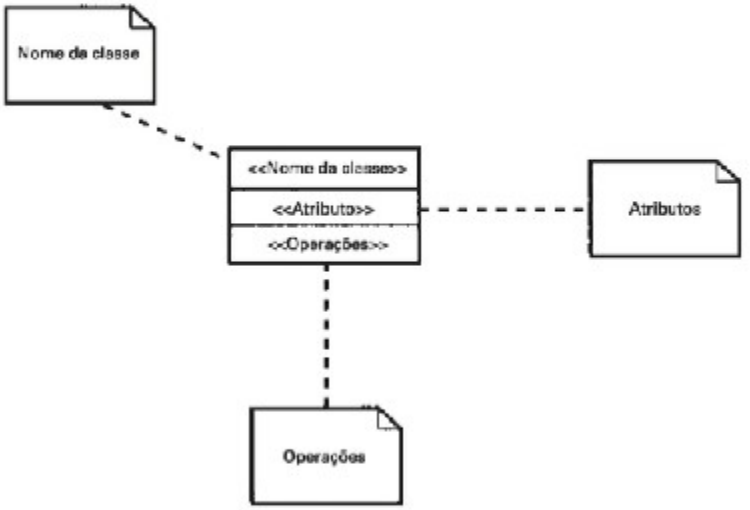
PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Introdução à Unified Modeling Language

Quando um construtor constrói uma casa, ele não faz isso aleatoriamente. Em vez disso, o construtor constrói a casa de acordo com um conjunto de cópias heliográficas detalhadas. Essas cópias heliográficas dispõem o projeto da casa explicitamente. Nada é deixado ao acaso.

FIGURA 8.1
A notação de classe da UML.

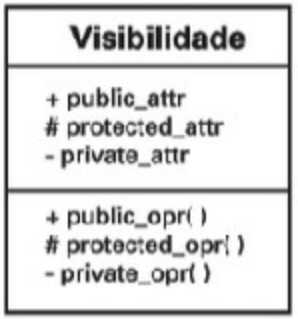


Dentro do modelo, você pode usar os caracteres -, # e +. Esses caracteres transmitem a *visibilidade* de um atributo ou de uma operação. O hífen (-) significa privado, o jogo da velha (#) significa protegido e o sinal de adição (+) significa público (veja a Figura 8.2).

PROGRAMAÇÃO ORIENTADA A OBJETO

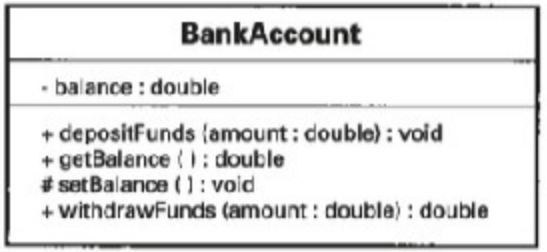
UML

FIGURA 8.2
A notação da UML para especificar visibilidade.



A Figura 8.3 ilustra a completa classe `BankAccount` dos dias 5 e 7.

FIGURA 8.3
Uma classe totalmente descrita.



PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Às vezes, uma nota ajudará a transmitir um significado que, de outro modo, ficaria perdido ou seria ignorado, como a nota da Figura 8.4.

FIGURA 8.4
Um exemplo detalhado de nota.

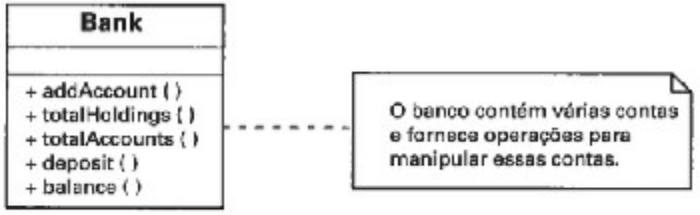
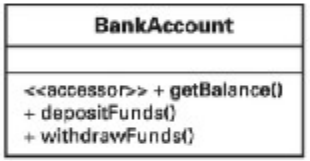
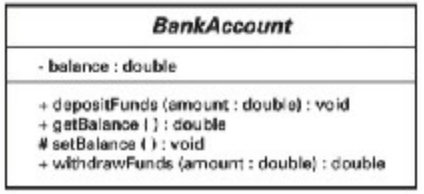


FIGURA 8.5
Um estereótipo que qualifica a operação.



Finalmente, você pode se lembrar que a classe `BankAccount` foi originalmente definida como uma classe concreta. Entretanto, o Dia 7 redefiniu a classe `BankAccount` como uma classe abstrata. A UML fornece uma notação para transmitir que uma classe é abstrata: o nome da classe abstrata é escrito em itálico. No caso de `BankAccount`, o nome deve ser escrito em itálico, conforme ilustrado na Figura 8.6.

FIGURA 8.6
O objeto `BankAccount` *abstrato.*



PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Modelando um relacionamento de classe

As classes não existem no vácuo. Em vez disso, elas têm relacionamentos complexos entre si. Esses relacionamentos descrevem como as classes interagem umas com as outras.

NOVO TERMO Um *relacionamento* descreve como as classes interagem entre si. Na UML, um relacionamento é uma conexão entre dois ou mais elementos da notação.

A UML reconhece três tipos de alto nível de relacionamentos de objeto:

- Dependência
- Associação
- Generalização

Embora a UML possa fornecer notação para cada um desses relacionamentos, os relacionamentos não são específicos da UML. Em vez disso, a UML simplesmente fornece um mecanismo e vocabulário comum para descrever os relacionamentos. Entender os relacionamentos, independentemente da UML, é importante em seu estudo de OO. Na verdade, se você simplesmente ignorar a notação e entender os relacionamentos, estará bem adiantado nos estudos.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Dependência

Dependência é o relacionamento mais simples entre objetos. A dependência indica que um objeto depende da especificação de outro objeto.

NOTA

Especificação é uma maneira diferente de dizer interface ou comportamento.

NOVO TERMO

Em um *relacionamento de dependência*, um objeto é dependente da especificação de outro objeto. Se a especificação mudar, você precisará atualizar o objeto dependente.

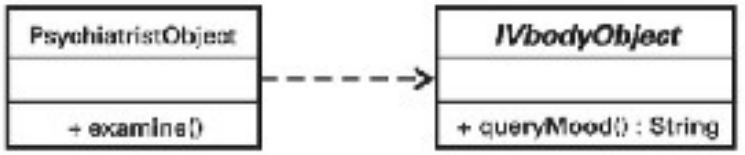


FIGURA 8.8
Um relacionamento de dependência simples.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Associação

Os relacionamentos de associação vão um pouco mais fundo do que os relacionamentos de dependência. As associações são relacionamentos estruturais. Uma associação indica que um objeto contém — ou que está conectado a — outro objeto.

NOVO TERMO Uma *associação* indica que um objeto contém outro objeto. Nos termos da UML, quando se está em um relacionamento de associação, um objeto está conectado a outro.

Como os objetos estão conectados, você pode passar de um objeto para outro. Considere a associação entre uma pessoa e um banco, como ilustrado na Figura 8.9.

FIGURA 8.9
Uma associação entre uma pessoa e um banco.

A Figura 8.9 mostra que uma pessoa *empresta de* um banco. Na notação UML, toda associação tem um nome. Neste caso, a associação é chamada de *empresta de*. A seta indica a direção da associação.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

NOVO TERMO O nome da associação é um nome que descreve o relacionamento.

Cada objeto em uma associação também tem um papel, conforme indicado na Figura 8.10.

FIGURA 8.10
Os papéis na associação.

NOVO TERMO Na associação, o papel de `Pessoa` é devedor e o papel de `Banco` é credor.

NOVO TERMO O papel da associação é a parte que um objeto desempenha em um relacionamento.

Finalmente, a multiplicidade indica quantos objetos podem tomar parte em uma associação.

NOVO TERMO A *multiplicidade* indica quantos objetos podem tomar parte na instância de uma associação.

A Figura 8.11 ilustra a multiplicidade da associação entre `Pessoa` e `Banco`.

FIGURA 8.11
Multiplicidade.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Essa notação nos informa que um banco pode ter um ou mais devedores e que uma pessoa pode utilizar 0 ou mais bancos.

Você especifica suas multiplicidades através de um único número, uma lista ou com um asterisco (*).

Um único número significa que determinado número de objetos — não mais e não menos — podem participar da associação. Assim, por exemplo, um 6 significa que seis objetos e somente seis objetos podem participar da associação.

* significa que qualquer número de objetos pode participar da associação.

Uma lista define um intervalo de objetos que podem participar da associação. Por exemplo, 1..4 indica que de 1 a 4 objetos podem participar da associação. 3..* indica que três ou mais objetos podem participar.

Quando você deve modelar associações?

Você deve modelar associações quando um objeto contiver outro objeto — o relacionamento *tem um*. Você também pode modelar uma associação quando um objeto usa outro. Uma associação permite que você modele quem faz o que em um relacionamento.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

A UML também define dois tipos de associação: *agregação* e *composição*. Esses dois subtipos de associação o ajudam a refinar mais seus modelos.

Agregação

Uma agregação é um tipo especial de associação. Uma agregação modela um relacionamento *tem um* (ou *parte de*, no jargão da UML) entre pares. Esse relacionamento significa que um objeto contém outro. *Pares* significa que um objeto não é mais importante do que o outro.

NOVO TERMO Um *relacionamento todo/parte* descreve o relacionamento entre objetos onde um objeto contém outro.

NOVO TERMO Uma *agregação* é um tipo especial de associação que modela o relacionamento 'tem um' de relacionamentos todo/parte entre pares.

Importância, no contexto de uma agregação, significa que os objetos podem existir independentemente uns dos outros. Nenhum objeto é mais importante do que o outro no relacionamento.

Considere a agregação ilustrada pela Figura 8.12.

FIGURA 8.12
Agregação entre um banco e seus clientes.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Composição

A composição é um pouco mais rigorosa do que a agregação. A composição não é um relacionamento entre pares. Os objetos não são independentes uns dos outros.

A Figura 8.13 ilustra um relacionamento de composição.

FIGURA 8.13
Composição entre um banco e suas filiais.

Aqui, você vê que `Banco` pode conter muitos objetos `Filial`. O losango fechado diz que esse é um relacionamento de composição. O losango também diz quem 'tem um'. Neste caso, `Banco` 'tem um', ou contém, objetos `Filial`.

NOTA

Um losango fechado simboliza a composição. O losango toca o objeto que é considerado o todo do relacionamento. O todo é constituído de partes. No exemplo anterior, `Banco` é o todo e os objetos `Filial` são as partes.

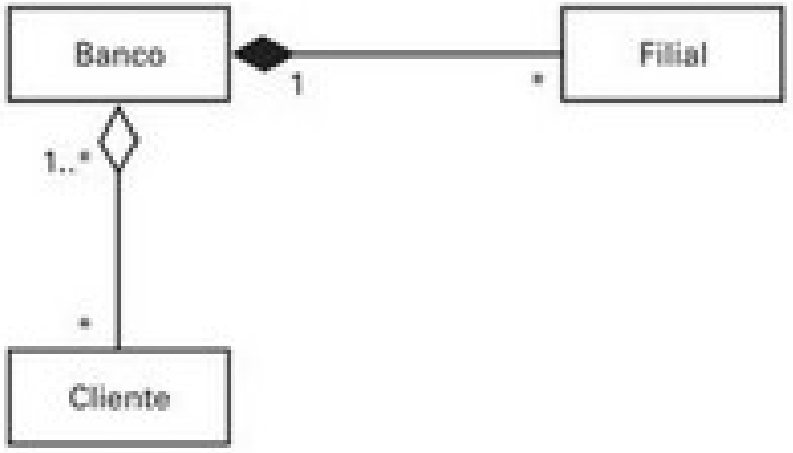
Como esse é um relacionamento de composição, os objetos `Filial` não podem existir independentemente do objeto `Banco`. A composição diz que, se o banco encerrar suas atividades, as filiais também fecharão. Entretanto, o inverso não é necessariamente verdade. Se uma filial fechar, o banco poderá permanecer funcionando.

Um objeto pode participar de uma agregação e de um relacionamento de composição ao mesmo tempo. A Figura 8.14 modela tal relacionamento.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

FIGURA 8.14
Banco em um relacionamento de agregação e de composição, simultaneamente.



PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Generalização

Um relacionamento de generalização é um relacionamento entre o geral e o específico. É a herança.

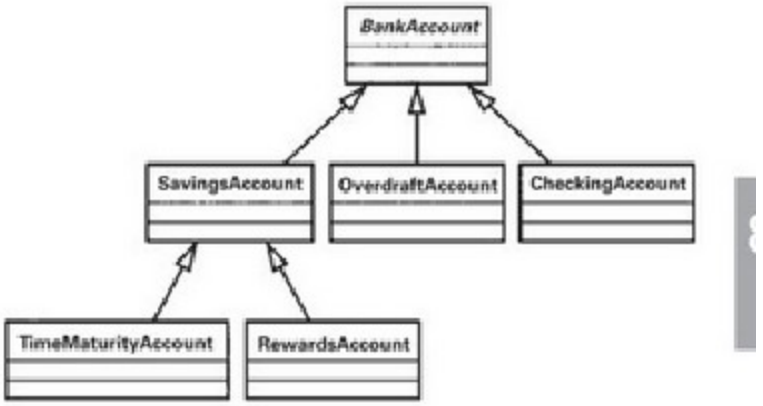
NOVO TERMO Um *relacionamento de generalização* indica um relacionamento entre o geral e o específico. Se você tem um relacionamento de generalização, então sabe que pode substituir uma classe filha pela classe progenitora.

A generalização incorpora o relacionamento *'é um'* sobre o qual você aprendeu no Dia 4. Conforme foi aprendido no Dia 4, os relacionamentos 'é um' permitem que você defina relacionamentos com capacidade de substituição.

Através de relacionamentos com capacidade de substituição, você pode usar descendentes em vez de seus ancestrais, ou filhos em vez de seus progenitores.

A UML fornece uma notação para modelar generalização. A Figura 8.15 ilustra como você modelaria a hierarquia de herança `BankAccount`.

FIGURA 8.15
A hierarquia de herança `BankAccount`.

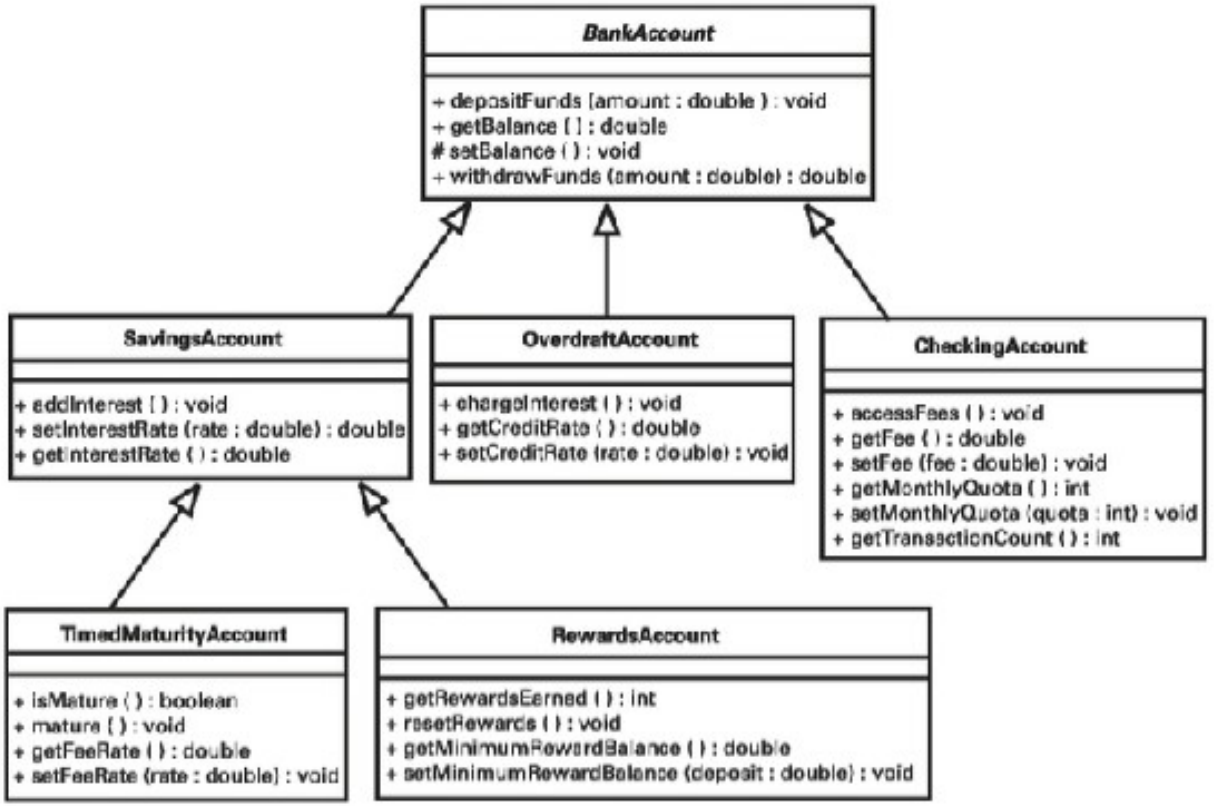


Uma linha cheia com uma seta fechada e vazada indica um relacionamento de generalização.

PROGRAMAÇÃO ORIENTADA A OBJETO

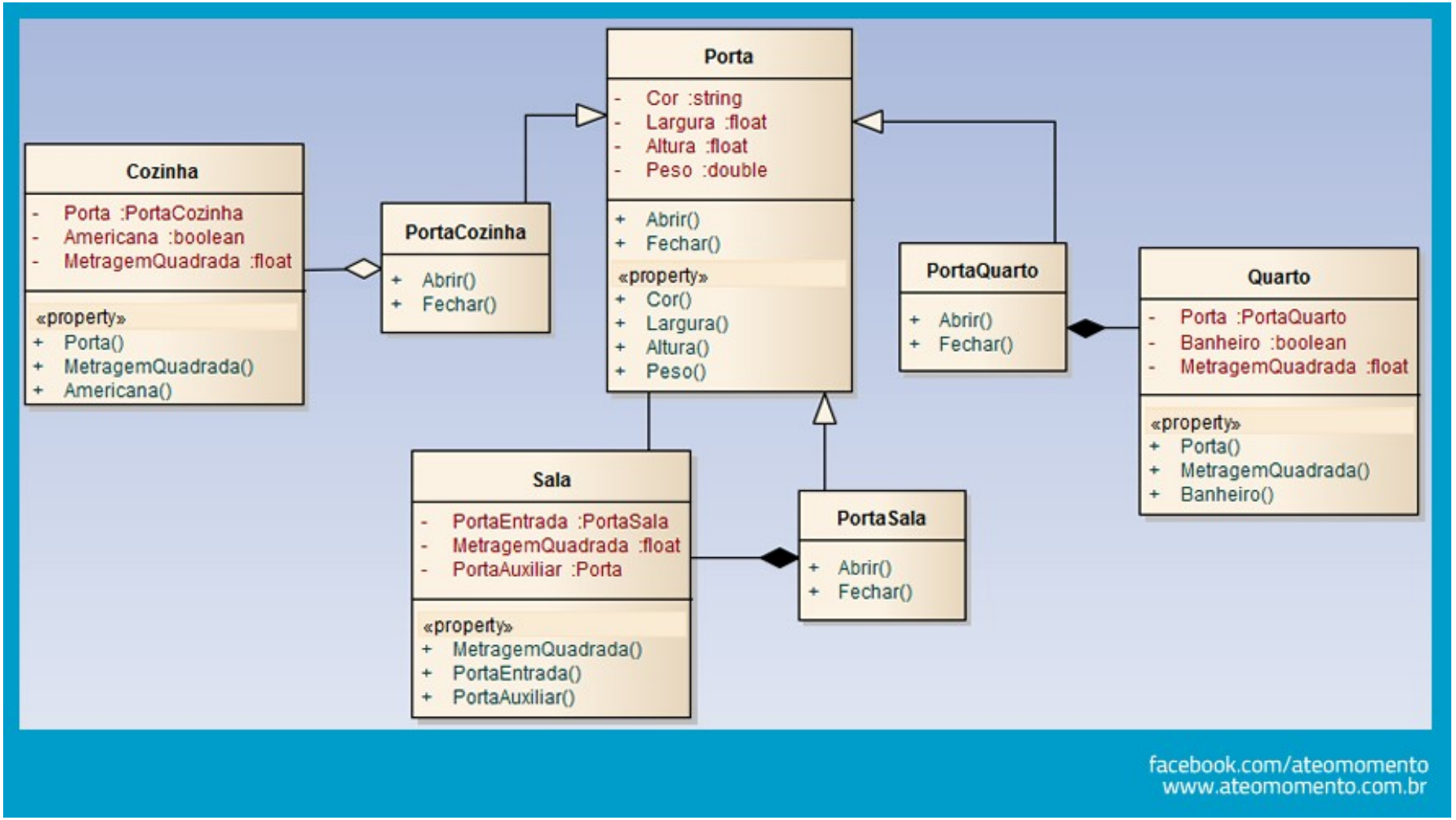
UML

FIGURA 8.17
Uma hierarquia de herança BankAccount *mais detalhada.*



PROGRAMAÇÃO ORIENTADA A OBJETO

UML - Diagrama de Classes Completo



PROGRAMAÇÃO ORIENTADA A OBJETO

Bibliografia e conteúdo

SINTES, Anthony.Aprenda Programação Orientada a Objeto em 21 Dias.
São Paulo:
Makron Books, 2002.

PROGRAMAÇÃO ORIENTADA A OBJETOS
Introdução à programação orientada a objetos.

- Encapsulamento.
- Método.
- Propriedades.
- Construtores.
- Herança.
- Polimorfismo.
- Introdução à UML.
- **Introdução à Análise Orientada a Objetos.**
- **Introdução ao Projeto Orientado a Objetos.**
- Reutilizando projetos através de padrões de
- projeto.
- Padrões avançados de projeto.
- OO e programação de interface com o usuário.
- Construindo software confiável através de testes.
- Prática da orientação a objetos.

PROGRAMAÇÃO ORIENTADA A OBJETO

POO (Projeto Orientado a Objetos)

NOVO TERMO *POO* é o processo de construir o modelo de objeto de uma solução. Dito de outra maneira, POO é o processo de dividir uma solução em vários objetos constituintes.

NOVO TERMO O *modelo de objeto* é o projeto dos objetos que aparecem na solução de um problema. O modelo final de objeto pode conter muitos objetos não encontrados no domínio. O modelo de objeto descreverá as várias responsabilidades, relacionamentos e estrutura do objeto.

O processo de POO o ajuda a descobrir como você vai implementar a análise que completou durante a AOO. Principalmente, o modelo de objeto que conterá as classes principais do projeto, suas responsabilidades e uma definição de como elas vão interagir e obter suas informações.

Pense no POO em termos da construção de uma casa para uma família. Antes de construir a casa de seus sonhos, você decide quais tipos de ambientes deseja em sua casa. Através de sua análise, você pode achar que deseja uma casa que tenha uma cozinha, dois quartos, dois banheiros e um lavabo, uma sala de estar e uma sala de jantar. Você também pode precisar de um gabinete de leitura, uma garagem para dois carros e uma piscina. Todos os ambientes e a piscina compreendem a idéia e o projeto de sua casa.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML + Análise Orientada a Objetos + Projeto OO

Introdução à Análise Orientada a Objetos.

AOO é uma estratégia orientada a objetos para entender um problema. Você usa AOO para ajudar a entender o núcleo do problema que deseja resolver. Após entender seu problema, Você pode começar a projetar uma solução. É aí que o Projeto Orientado a Objetos entra em ação.

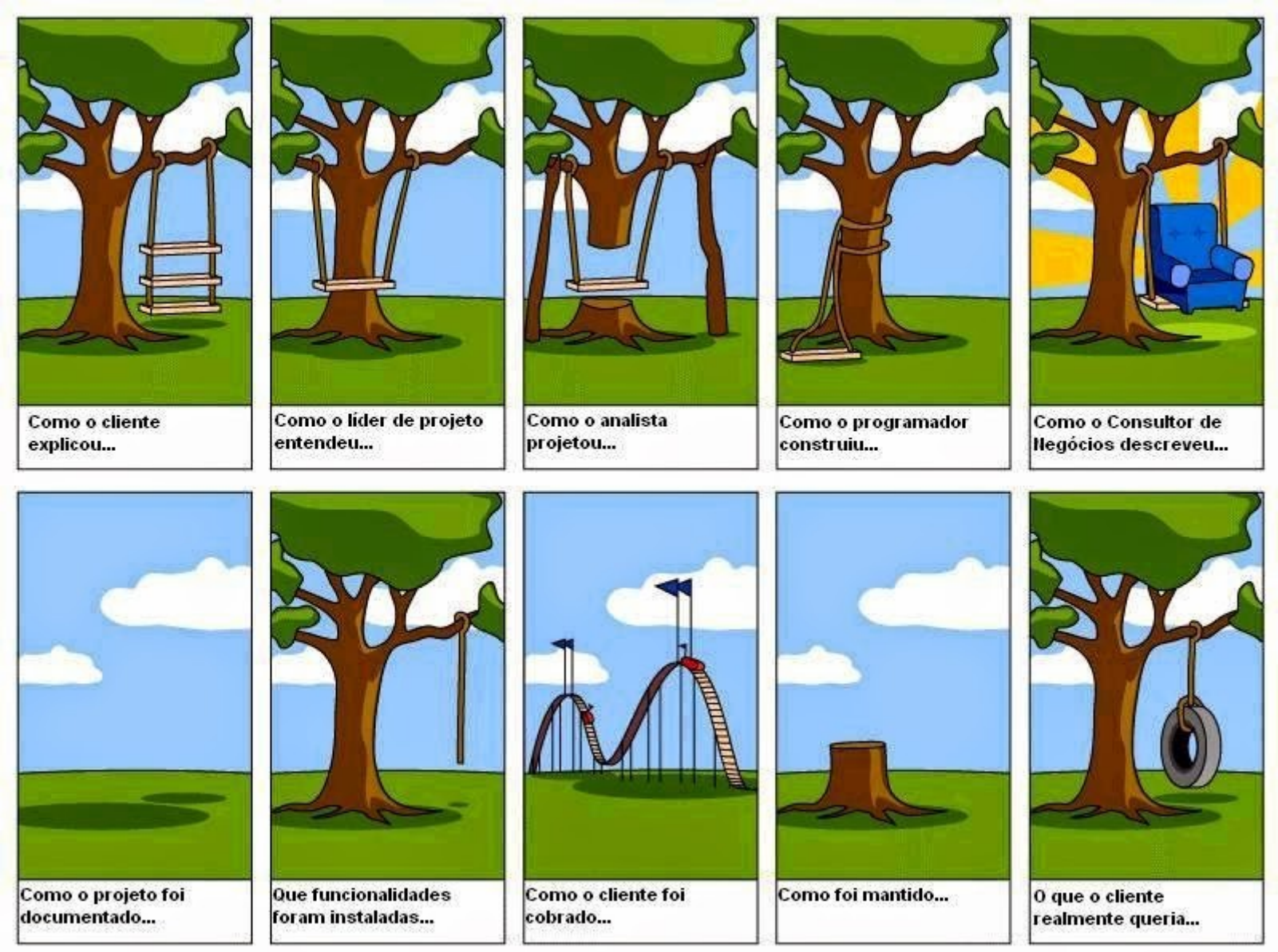
O processo de desenvolvimento de software

Existem muitas maneiras de desenvolver software quanto existem desenvolvedores. Entretanto, uma equipe de desenvolvimento de software precisa de uma estratégia unificada para desenvolver software. Nada será feito, se cada desenvolvedor fizer sua própria atividade. As metodologias de software definem uma maneira comum de encarar o desenvolvimento de software. Uma metodologia frequentemente conterá uma linguagem de modelagem (como a UML) e um processo.

Um processo de software mostra os vários estágios do desenvolvimento de software.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML + Análise Orientada a Objetos + Projeto OO



PROGRAMAÇÃO ORIENTADA A OBJETO

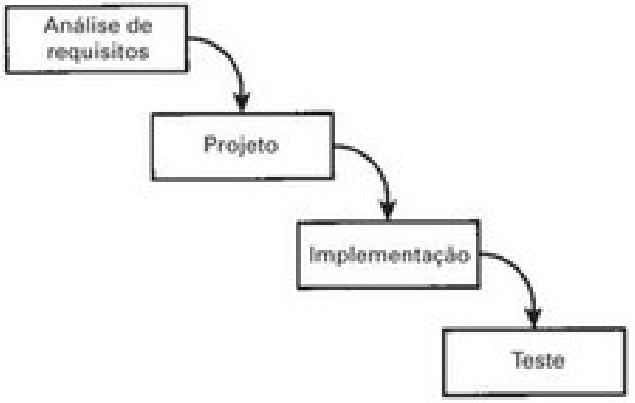
UML + Análise Orientada a Objetos + Projeto OO

PROCESSO EM CASCATA

O processo em cascata você vai de um estágio para o próximo. Entretanto, uma vez que você complete um estágio, não há volta - exatamente como descer uma cascata ou um penhasco.

O processo de cascata tenta evitar alteração, proibindo mudar quando um estágio está concluído. Tal estratégia protege os desenvolvedores de requisitos que mudam constantemente. Entretanto, tal processo rígido frequentemente resulta em um software que não é o que você ou seu cliente quer.

FIGURA 9.1
O processo de cascata.



Conforme a Figura 9.1 ilustra, o processo de cascata é seqüencial e unidirecional. O processo é constituído de quatro estágios distintos:

1. Análise de requisitos
2. Projeto
3. Implementação
4. Teste

PROGRAMAÇÃO ORIENTADA A OBJETO

UML + Análise Orientada a Objetos + Projeto OO

PROCESSO ITERATIVO

O processo iterativo

É o oposto do processo em cascata. O processo iterativo permite alterações em qualquer ponto do processo de desenvolvimento. O processo iterativo permite alteração adotando uma estratégia iterativa e incremental para o desenvolvimento de software.

Um processo iterativo é uma estratégia iterativa e incremental para desenvolvimento de software. Outro modo de pensar a respeito do processo é como uma estratégia evolutiva. Cada iteração aperfeiçoa e elabora gradualmente um produto básico em um produto amadurecido

Uma estratégia iterativa

Ao contrário do processo de cascata, o processo iterativo permite que você continuamente volte e refine cada estágio do desenvolvimento.

Por exemplo, se você descobrir que o projeto simplesmente não funciona ao executar a implementação, pode voltar e fazer um projeto adicional e uma nova análise. É esse refinamento contínuo que torna o processo iterativo.

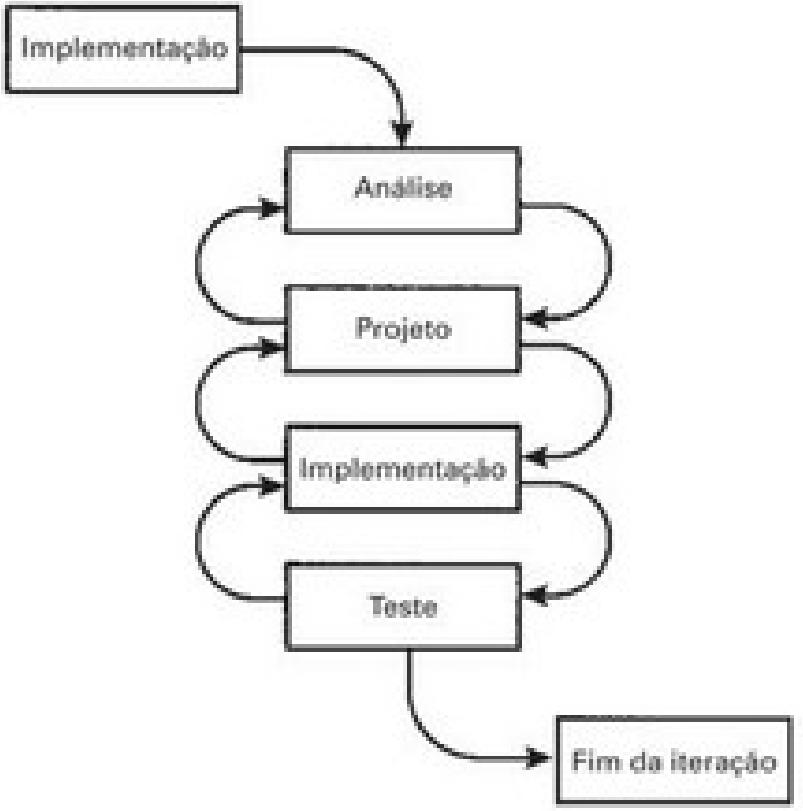
Uma estratégia incremental

Ao seguir um processo iterativo, você não conclui simplesmente uma iteração grande que constrói o programa inteiro. Em ve disso, o processo iterativo divide o trabalho de desenvolvimento em várias iterações pequenas.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML + Análise Orientada a Objetos + Projeto OO

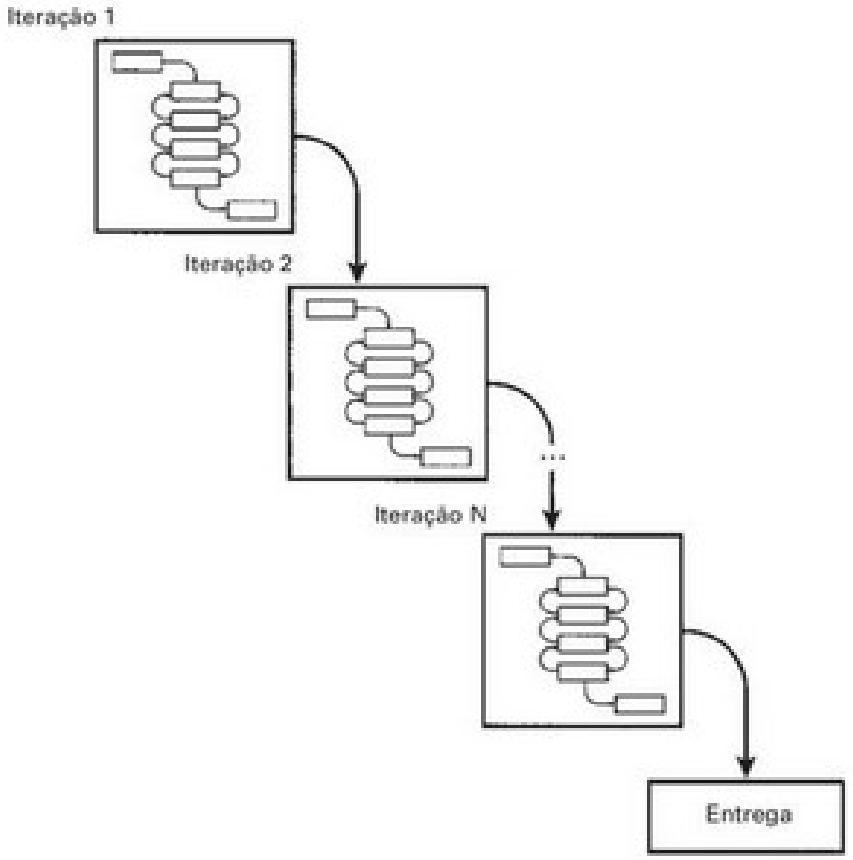
FIGURA 9.2
Uma iteração.



PROGRAMAÇÃO ORIENTADA A OBJETO

UML + Análise Orientada a Objetos + Projeto OO

FIGURA 9.3
O processo iterativo.



Cada iteração do processo introduz uma pequena melhoria incremental no programa. Essa melhoria pode ser um novo recurso ou um refinamento de um recurso já existente.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML + Análise Orientada a Objetos + Projeto OO

4 processos da Interação:

- Análise
- Projeto
- Implementação
- Teste

Após o estágio de teste, você também pode ter estágios de lançamento e manutenção. Esses são estágios importantes no ciclo de vida de um projeto de software. Entretanto, para os propósitos da lição de hoje, esses estágios serão omitidos. Hoje, você vai focalizar análise, projeto, implementação e teste.

As metodologias 'reais' freqüentemente enumeram estágios adicionais. Entretanto, quando você está aprendendo pela primeira vez, esses quatro estágios são aqueles que mais importam. Por isso, este livro se concentra nesses quatro estágios. O restante deste dia abordará a análise orientada a objetos.

PROGRAMAÇÃO ORIENTADA A OBJETO

Sistema: é o termo da AOO para um conjunto de objetos que interagem. Você pode dizer que esses objetos constituem um sistema ou modelo do problema.

Usando casos de estudo para descobrir o uso do sistema

Ao começar a analisar um problema, você primeiro precisa entender como seus usuários utilizarão ou interagirão com o sistema. Esses usos compreendem os requisitos do sistema e prescrevem o sistema que você cria. Atendendo os requisitos de seus usuários, você produz um sistema útil.

NOVO TERMO Os *requisitos* são os recursos ou características que o sistema deve ter para resolver determinado problema.

NOVO TERMO Um modo de descobrir esses usos é através de *análise de casos de uso*. Através da análise você definirá vários casos de uso. Um caso de uso descreve como um usuário vai interagir com o sistema.

NOVO TERMO *Análise de casos de uso* é o processo de descoberta de *casos de uso* através da criação de cenários e histórias com usuários em potencial ou existentes de um sistema.

NOVO TERMO Um *caso de uso* descreve a interação entre o usuário do sistema e o sistema — como o usuário utilizará o sistema do seu próprio ponto de vista.

A criação de casos de uso é um processo iterativo. Existem vários passos que você deve dar durante cada iteração, para formalizar seus casos de uso. Para definir seus casos de uso, você deve:

1. Identificar os atores.
2. Criar uma lista preliminar de casos de uso.
3. Refinar e nomear os casos de uso.
4. Definir a seqüência de eventos de cada caso de uso.
5. Modelar seus casos de uso.

PROGRAMAÇÃO ORIENTADA A OBJETO

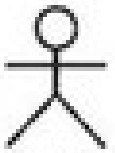
Identifique os atores

O primeiro passo na definição de seus casos de uso é definir os atores que usarão o sistema.

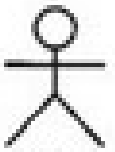
NOVO TERMO Um *ator* é tudo que interage com o sistema. Pode ser um usuário humano, outro sistema de computador ou um chimpanzé.

Você precisa pedir aos seus clientes para que descrevam os usuários do sistema. As perguntas podem incluir as seguintes:

- Quem principalmente usará o sistema?
- Existem outros sistemas que usarão o sistema? Por exemplo, existem quaisquer usuários que não são seres humanos?
- O sistema se comunicará com qualquer outro sistema? Por exemplo, há um banco de dados já existente que você precise integrar?
- O sistema responde ao estímulo gerado por alguém que não seja usuário? Por exemplo, o sistema precisa fazer algo em certo dia de cada mês? Um estímulo pode ser proveniente de fontes normalmente não consideradas ao se pensar do ponto de vista puramente do usuário.



Usuário Convidado



Usuário Registrado

Considere uma loja da Web on-line. Uma loja on-line permite que usuários convidados naveguem pelo catálogo de produtos, verifique o preço dos itens e solicite mais informações. A loja também permite que usuários registrados comprem itens, assim como controla seus pedidos e mantém informações dos usuários.

A partir dessa breve descrição, você pode identificar dois atores: usuários convidados e usuários registrados. Cada um desses dois atores interage com o sistema.

A Figura 9.4 ilustra a notação UML para um ator: um desenho de pessoa com um nome. Você deve dar a cada um de seus atores um nome não ambíguo.

PROGRAMAÇÃO ORIENTADA A OBJETO

Um usuário pode assumir muitos *papéis* diferentes enquanto interage com um sistema. Um ator descreve o *papel* que o usuário pode assumir enquanto interage com o sistema.

Quando você começar a definir seus casos de uso, crie uma lista preliminar de atores. Não se atrapalhe ao identificar os atores. Será difícil descobrir todos os atores na primeira vez.

Em vez disso, encontre atores suficientes para começar e adicione os outros à medida que os descobrir.

Os atores são os instigadores de casos de uso. Agora que você já identificou alguns atores, pode começar a definir os casos de uso que eles executam.

PROGRAMAÇÃO ORIENTADA A OBJETO

Diagramas de caso de uso

Assim como a UML fornece uma maneira de documentar e transmitir projeto de classe, também existem maneiras formais de capturar seus casos de uso. De especial interesse são os diagramas de caso de uso, diagramas de interação e diagramas de atividade. Cada um ajuda a visualizar os vários casos de uso.

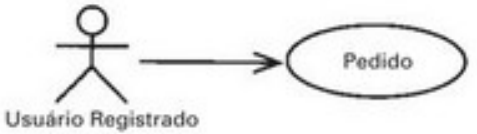
Os diagramas de caso de uso modelam os relacionamentos entre casos de uso e os relacionamentos entre casos de uso e atores. Embora a descrição textual de um caso de uso possa ajudá-lo a entender um caso de uso isolado, um diagrama o ajuda a ver como os casos de uso se relacionam uns com os outros.

A Figura 9.4 mostra como modelar atores. A Figura 9.5 ilustra a notação UML para um caso de uso: uma elipse rotulada.

FIGURA 9.5
O caso de uso na UML.

Coloque um ator e um caso de uso juntos no mesmo diagrama e você terá um diagrama de caso de uso. A Figura 9.6 é o diagrama de caso de uso Pedido.

FIGURA 9.6
O caso de uso Pedido.

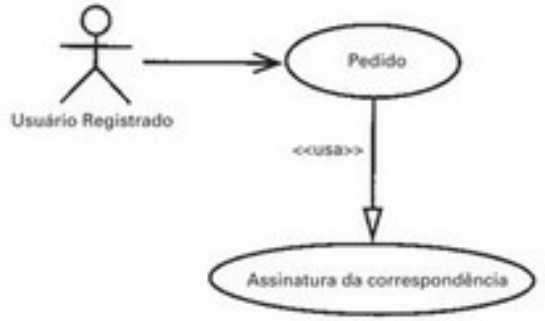


Esse diagrama é muito simples; entretanto, examinando-o, você pode ver que o usuário registrado executa o caso de uso *Pedido*.

PROGRAMAÇÃO ORIENTADA A OBJETO

Os diagramas podem ser um pouco mais complicados. O diagrama também pode mostrar os relacionamentos existentes entre os próprios casos de uso. Conforme você já leu, um caso de uso pode conter e usar outro. A Figura 9.7 ilustra tal relacionamento.

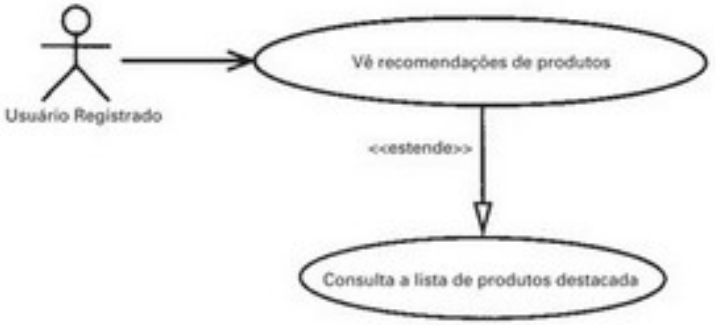
FIGURA 9.7
Um relacionamento usa.



Aqui, você vê que o caso de uso Registro usa o caso de uso Assinatura da correspondência. Como parte do processo de registro, o usuário pode optar por receber e-mails e notificações.

A Figura 9.8 ilustra o segundo tipo de relacionamento, o relacionamento estende.

FIGURA 9.8
Um relacionamento estende.



PROGRAMAÇÃO ORIENTADA A OBJETO

Diagramas de seqüência

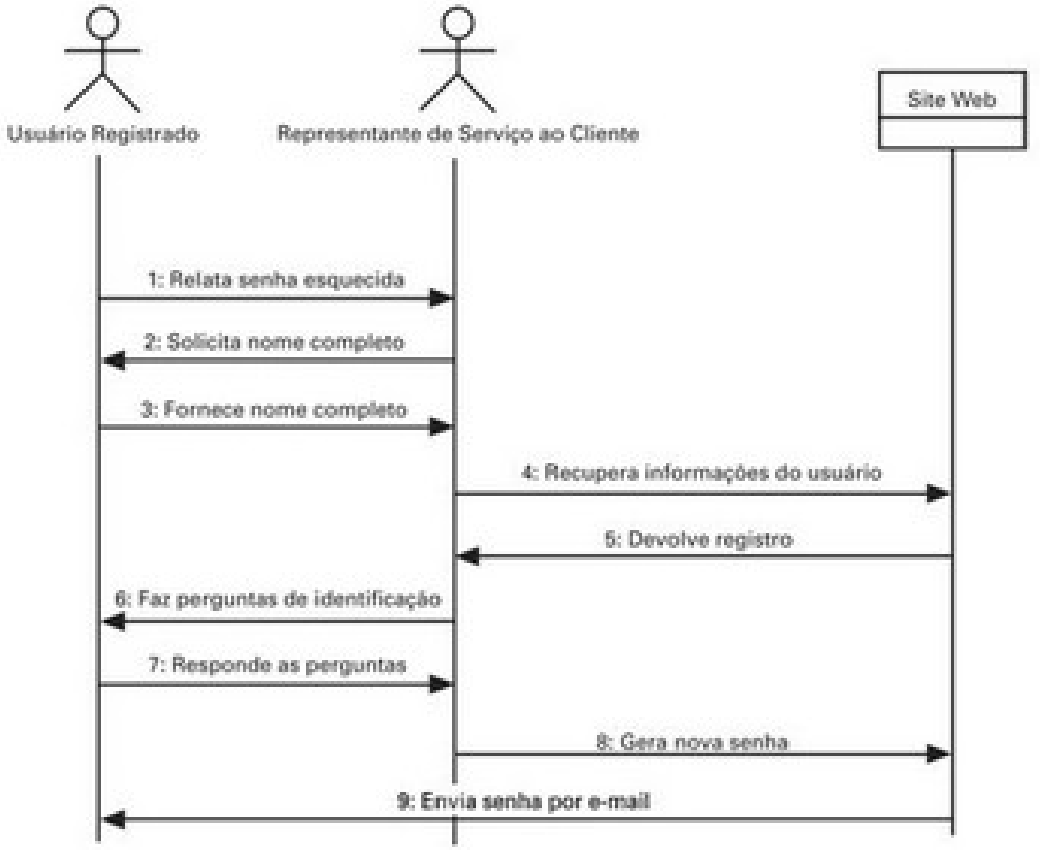
Um diagrama de seqüência modela as interações entre o usuário registrado, o representante de serviço ao cliente e o site Web, com o passar do tempo. Você deve usar diagramas de seqüência quando quiser chamar a atenção para a seqüência de eventos de um caso de uso, com o passar do tempo. A Figura 9.9 apresenta um diagrama de seqüência para o caso de uso *Senha Esquecida*.

Conforme você pode ver na ilustração, um diagrama de seqüência representa os eventos entre cada ator e o sistema (o site Web). Cada participante do caso de uso é representado no início do diagrama como uma caixa ou como um desenho de pessoa (mas você pode chamar ambos de caixa).

Uma linha tracejada, conhecida como *linha da vida*, sai de cada caixa. A linha da vida representa o tempo de vida da caixa durante o caso de uso. Assim, se um dos atores fosse embora durante o caso de uso, a linha terminaria na última seta que termina ou se origina no ator. Quando um ator deixa um caso de uso, você pode dizer que seu tempo de vida terminou.

PROGRAMAÇÃO ORIENTADA A OBJETO

FIGURA 9.9
A diagrama de seqüência Senha Esquecida.



NOVO TERMO Uma *linha da vida* é uma linha tracejada que sai de uma caixa em um diagrama de seqüência. A linha da vida representa o tempo de vida do objeto representado pela caixa.

As setas se originam na linha da vida para indicar que o ator enviou uma mensagem para outro ator ou para o sistema. Quando você desce na linha da vida, pode ver as mensagens conforme

PROGRAMAÇÃO ORIENTADA A OBJETO

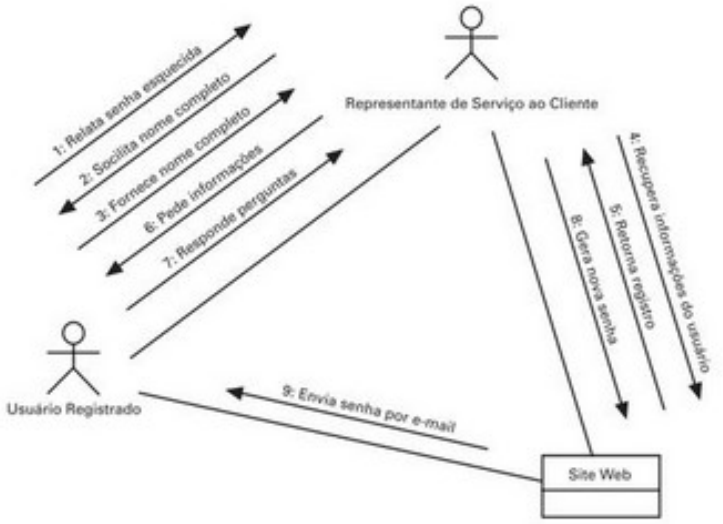
Diagramas de colaboração

Você deve usar diagramas de seqüência se tiver a intenção de chamar a atenção para a seqüência de eventos com o passar do tempo. Se você quiser modelar os relacionamentos entre os atores e o sistema, então deve criar um diagrama de colaboração.

A Figura 9.10 modela o caso de uso *Senha Esquecida* como um diagrama de colaboração.

Em um diagrama de colaboração, você modela uma interação conectando os participantes com uma linha. Acima da linha, você rotula cada evento que as entidades geram, junto com a direção do evento (para quem ele é dirigido). Ele também ajuda a numerar os eventos, para que você saiba em qual ordem eles aparecem.

FIGURA 9.10
O diagrama de colaboração Senha Esquecida.



DICA

Use diagramas de seqüência para modelar a seqüência de eventos em um cenário, com o passar do tempo.

Use diagramas de colaboração para modelar os relacionamentos entre os atores em um cenário.

PROGRAMAÇÃO ORIENTADA A OBJETO

Diagramas de atividade

Os diagramas de interação modelam bem as ações seqüenciais. Entretanto, eles não podem modelar processos que podem ser executados em paralelo. Os diagramas de atividade o ajudam a modelar processos que podem ser executados em paralelo.

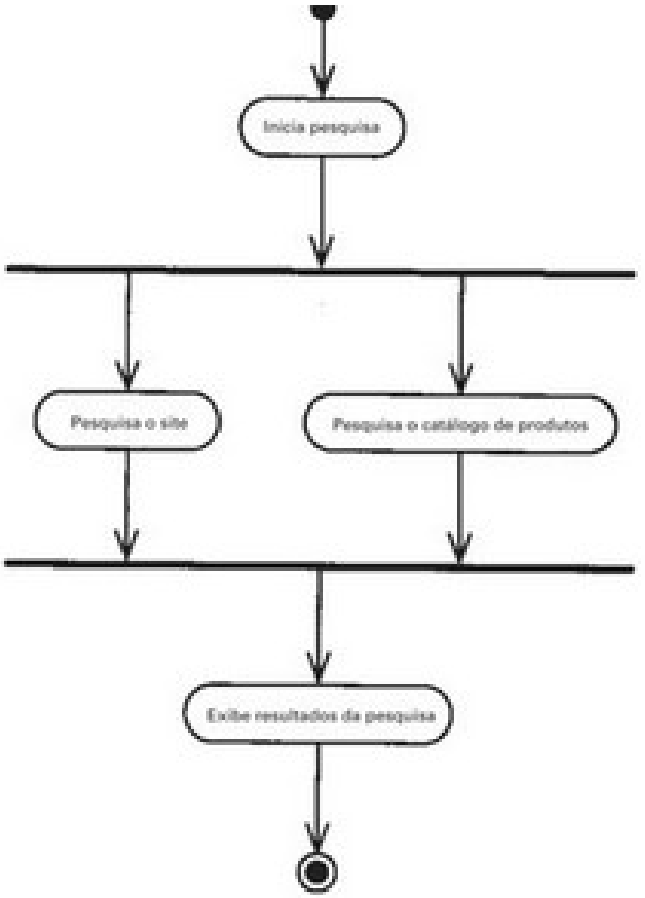
Considere outro caso de uso *Pesquisa*. Esse caso de uso pesquisa o site Web e o catálogo de produtos simultaneamente, usando o caso de uso *Pesquisa o catálogo de produtos* e o caso de uso *Pesquisa o site*. Não há motivo pelo qual essas duas pesquisas não possam ser executadas simultaneamente. O usuário ficaria impaciente se tivesse de esperar que todas as pesquisas terminassem seqüencialmente.

A Figura 9.11 modela esses processos através de um diagrama de atividade.

Uma elipse representa cada estado do processo. A barra preta grossa representa um ponto onde os processos devem ser sincronizados — ou reunidos —, antes que o fluxo de execução possa ser retomado. Aqui, você vê que as duas pesquisas são executadas em paralelo e depois reunidas, antes que o site possa exibir os resultados.

PROGRAMAÇÃO ORIENTADA A OBJETO

FIGURA 9.11
O diagrama de atividade Pesquisa.



Vamos ver de perto os diagramas de interação e os diagramas de atividade nos próximos dias. Entretanto, ambos se mostram úteis ao se analisar um sistema.

PROGRAMAÇÃO ORIENTADA A OBJETO

Bibliografia e conteúdo

SINTES, Anthony.Aprenda Programação Orientada a Objeto em 21 Dias.
São Paulo:
Makron Books, 2002.

PROGRAMAÇÃO ORIENTADA A OBJETOS
Introdução à programação orientada a objetos.

- Encapsulamento.
- Método.
- Propriedades.
- Construtores.
- Herança.
- Polimorfismo.
- Introdução à UML.
- Introdução à Análise Orientada a Objetos.
- Introdução ao Projeto Orientado a Objetos.
- **Reutilizando projetos através de padrões de**
- **projeto.**
- **Padrões avançados de projeto.**
- **OO e programação de interface com o usuário.**
- **Construindo software confiável através de testes.**
- **Prática da orientação a objetos.**

PROGRAMAÇÃO ORIENTADA A OBJETO

NOVO TERMO Um *padrão de projeto* é um conceito de projeto reutilizável.

Verifica-se que você pode reutilizar padrões de projeto em todo o seu projeto, exatamente como se você reutilizasse uma classe dentro de seu programa. Essa reutilização traz as vantagens da reutilização da POO (Programação Orientada a Objetos) para o POO (Projeto Orientado a Objetos). Quando usa padrões de projeto, você sabe que baseou seu projeto em projetos confiáveis e comprovados pelo uso. Tal reutilização permite que você saiba se está na trilha certa para uma solução confiável. Quando você reutiliza um padrão de projeto, está usando um projeto que outros usaram com êxito, muitas vezes anteriormente.

Padrões de projeto

O livro *Design Patterns — Elements of Reusable Object-Oriented Software*, de Gamma, Helm, Johnson e Vlissides, apresentou pela primeira vez o conceito de padrões de projeto para muitos na comunidade de OO. Esse trabalho de base não apenas definiu um conjunto de padrões de projeto reutilizáveis, mas também definiu formalmente o padrão de projeto.

De acordo com esse trabalho, um padrão de projeto consiste em quatro elementos:

- O nome do padrão
- O problema
- A solução
- As conseqüências

PROGRAMAÇÃO ORIENTADA A OBJETO

O nome do padrão

Um nome identifica exclusivamente cada padrão de projeto. Assim como a UML fornece uma linguagem de projeto comum, os nomes dos padrões fornecem um vocabulário comum para descrever os elementos de seu projeto para outros. Outros desenvolvedores podem entender seu projeto rápida e facilmente, quando você usa um vocabulário comum.

Um simples nome reduz um problema inteiro, a solução e as conseqüências a um único termo. Assim como os objetos o ajudam a programar em um nível mais alto e abstrato, esses termos permitem que você projete a partir de um nível mais alto e abstrato e não fique preso aos detalhes que se repetem de um projeto para outro.

O problema

Cada padrão de projeto existe para resolver algum conjunto distinto de problemas de projeto e cada padrão de projeto descreve o conjunto de problemas para o qual foi feito para resolver. Desse modo, você pode usar a descrição do problema para determinar se o padrão se aplica ao problema específico que está enfrentando.

PROGRAMAÇÃO ORIENTADA A OBJETO

A solução

A solução descreve como o padrão de projeto resolve o problema e identifica os objetos arquitetonicamente significativos na solução, assim como as responsabilidades e relacionamentos que esses objetos compartilham.

É importante notar que você pode aplicar um padrão de projeto a uma classe inteira de problemas. A solução é uma solução geral e não apresenta uma resposta para um problema específico ou concreto.

Suponha que você quisesse encontrar a melhor maneira de percorrer os itens do carrinho de compras do capítulo 10, "Introdução ao POO (Projeto Orientado a Objetos)". O padrão de projeto Iterator propõe um modo de fazer justamente isso; entretanto, a solução apresentada pelo padrão Iterator não é dada em termos de carrinhos de compras e itens. Em vez disso, a solução descreve o processo de varredura de qualquer lista de elementos.

Quando você começar a usar um padrão de projeto, deve mapear a solução geral para seu problema específico. Às vezes, pode ser difícil fazer esse mapeamento; entretanto, o próprio padrão de projeto precisa permanecer genérico para que permaneça aplicável a muitos problemas específicos diferentes.

As conseqüências

Não existe um projeto perfeito. Todo bom projeto terá bons compromissos e todo compromisso assumido terá seu próprio conjunto especial de conseqüências. Um padrão de projeto enumerará as principais conseqüências inerentes ao projeto.

PROGRAMAÇÃO ORIENTADA A OBJETO

Os principais padrões :

- Adapter
- Proxy
- Iterator

PROGRAMAÇÃO ORIENTADA A OBJETO

O padrão ADAPTER

Um adaptador é um objeto que transforma a interface de outro objeto

TABELA 11.1 O padrão Adapter

Nome do padrão	Adapter, Wrapper
Problema	Como reutilizar objetos incompatíveis
Solução	Fornecer um objeto que converta a interface incompatível em uma compatível
Conseqüências	Tornar incompatíveis objetos compatíveis; resulta em classes extras — talvez muitas —, se você usar herança ou precisar manipular cada subclasse de uma forma diferente

PROGRAMAÇÃO ORIENTADA A OBJETO

O padrão ADAPTER

Converter a interface de uma classe por outra esperada pelos clientes . O que possibilita que classes com interfaces incompatíveis trabalhem em conjunto – ou que, de outra forma, seria impossível. Também conhecido como Wrapper(adaptador).

Algumas vezes, uma classe de um toolkit, projetada para ser reutilizada não condiz com a interface específica de um domínio requerida por uma aplicação.
O uso do padrão está condicionado a:
1.Usar uma classe existente, mas sua interface não corresponde à interface requerida;

1.Criar classes reutilizáveis que cooperam com classes não-relacionadas ou não previstas, ou seja, classes com interface inicialmente incompatível.

PROGRAMAÇÃO ORIENTADA A OBJETO

O padrão ADAPTER

Os participantes são:

1.Target (Alvo)–define a interface específica do domínio do cliente;
2.Client (cliente) –colabora com objetos compatíveis com Target;
3.Adaptee (Adaptação) –interface existente de necessita de adaptação;
4.Adapter (Adaptador) –adapta a interface Adaptee à interface Target.

Para adaptações de classes:
Um adaptador de classe não funcionará quando quisermos adaptar uma classe e todas as suas subclasses;
Permite a Adapter substituir algum comportamento de Adaptee, já que Adapter é uma subclasse

Para adaptações de objetos:

•Permite a um único Adapter adaptar um Adaptee e suas subclasses;
•Torna mais difícil redefinir o comportamento de um Adaptee. Conseguido através de uma subclasse de Adaptee que é referenciada por Adapter.

PROGRAMAÇÃO ORIENTADA A OBJETO

Implementando um adaptador

A Figura 11.1 resume a hierarquia `MoodyObject` apresentada no Capítulo 7, "Polimorfismo: hora de escrever código".

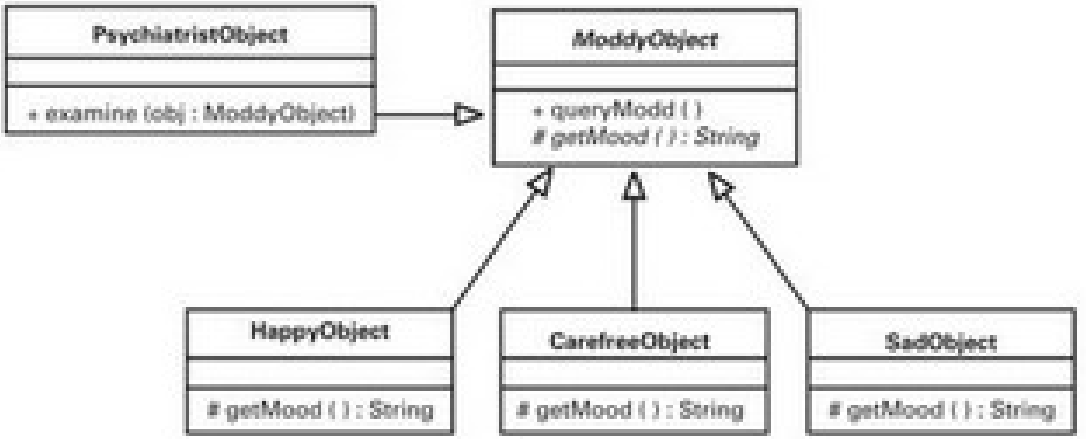


FIGURA 11.1 *A hierarquia* `MoodyObject`.

`PsychiatristObject` só pode examinar um objeto se for um `MoodyObject`. O método `examine()` de `PsychiatristObject` depende do método `queryMood()` de `MoodyObject`. Qualquer outro tipo de objeto precisará encontrar um psiquiatra diferente.

A Figura 11.2 apresenta a nova hierarquia `Pet`.

PROGRAMAÇÃO ORIENTADA A OBJETO

Quando usar o padrão Adapter

O padrão Adapter é útil quando você quer usar um objeto que tem uma interface incompatível. O padrão Adapter permite que você reutilize diretamente objetos que, de outro modo, precisaria alterar ou jogar fora.

Os adaptadores também são úteis no sentido de uma ação preventiva. De tempos em tempos, você precisará empregar bibliotecas de terceiros em seus programas. Infelizmente, as APIs das ferramentas de terceiros podem variar substancialmente entre as versões, especialmente para novos produtos ou para tecnologias que estão amadurecendo. As APIs também podem variar bastante em relação às bibliotecas de um produto concorrente.

O padrão Adapter pode ajudar a evitar que seu programa tenha que trocar de APIs e que fique preso ao fornecedor. Criando uma interface adaptadora que você controla, é possível trocar para novas versões de uma biblioteca a qualquer momento. Basta criar uma subclasse do adaptador para cada biblioteca que você queira usar.

Também é importante notar que um adaptador pode ser simples ou complicado e o nível de complexidade depende do objeto que está sendo empacotado. Alguns adaptadores se resumem simplesmente a mapear um pedido para o método correto. Outros precisarão realizar mais processamento.

Use o padrão Adapter, quando:

- você quiser usar objetos incompatíveis em seu programa;
- você quiser que seu programa permaneça independente de bibliotecas de terceiros.

A Tabela 11.1 destaca o usuário do padrão Adapter.

O padrão Proxy

Normalmente, quando um objeto quer interagir com outro, ele faz isso atuando diretamente sobre o outro objeto. Na maioria dos casos, essa estratégia direta é a melhor estratégia, mas existem ocasiões em que você desejará controlar o acesso entre seus objetos de forma transparente. O padrão Proxy trata desses casos.

Publicação/assinatura e o padrão Proxy

Considere o problema de um serviço de eventos publicação/assinatura onde um objeto vai registrar seu interesse em um evento. Quando o evento ocorrer, o publicador notificará os objetos assinantes do evento.

A Figura 11.3 ilustra um possível relacionamento publicação/assinatura.

FIGURA 11.3
Um relacionamento publicação/assinatura.

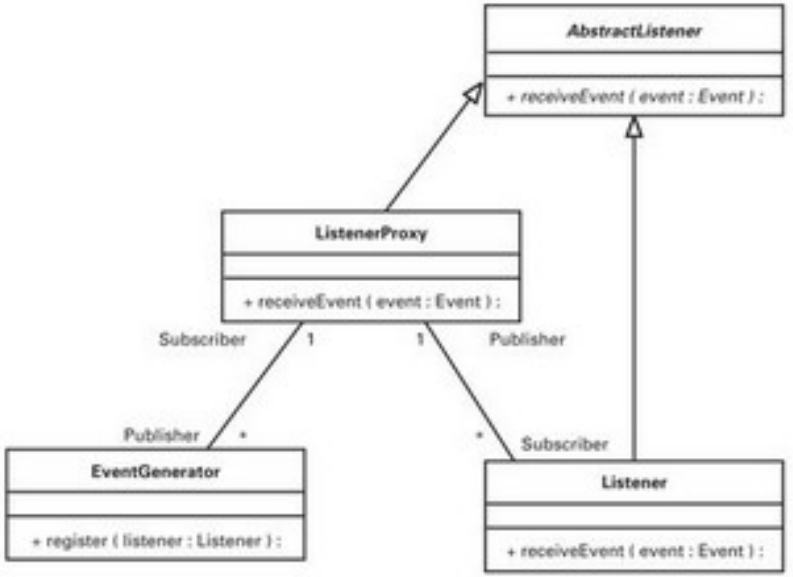
Na Figura 11.3, muitos `Listeners` registram seu interesse nos eventos gerados por um `EventGenerator`. Quando o `EventGenerator` gerar um evento, ele colocará o evento em cada um de seus objetos `Listeners`.

Embora essa solução funcione, ela coloca uma grande carga sobre o `EventGenerator`. O `EventGenerator` não tem apenas a responsabilidade de gerar eventos, como também é responsável por controlar todos os `Listeners` e colocar os eventos neles.

O padrão Proxy apresenta uma solução elegante para esse problema. Considere a Figura 11.4.

Em vez de conter seus `Listeners` diretamente, o `EventGenerator` pode conter um `ListenerProxy`. Quando o gerador precisa disparar um evento, ele o dispara uma vez para o proxy. Então, fica por conta do proxy controlar e atualizar todos os `Listeners`.

FIGURA 11.4
Uma solução proxy.



O padrão Proxy geral

O cenário publicação/assinatura descreve um possível uso para um proxy; entretanto, você pode usar proxies em muitos lugares.

Primeiro, o que é um proxy?

Um *proxy* é um substituto ou lugar reservado que intermedia o acesso ao objeto de interesse real. Você pode usar um proxy em qualquer lugar onde precise de um substituto ou lugar reservado para outro objeto. Em vez de usar um objeto diretamente, você usa o proxy. O proxy cuida de todos os detalhes da comunicação com o objeto (ou objetos) real.

NOVO TERMO Um *proxy* é um substituto ou lugar reservado que intermedia o acesso ao objeto de interesse real. Para todos os efeitos, o *substituto* é indistinguível do objeto real que intermedia.

Quando usar o padrão Proxy

Use substitutos quando:

- Você quiser adiar uma operação dispendiosa. Considere um objeto que extrai informações de um banco de dados. Embora seu programa talvez precise saber a respeito dos objetos que pode extrair, ele não precisa extrair todas as informações até que realmente acesse o objeto. Um substituto pode representar o objeto real e carregar as informações extras quando elas forem necessárias.

  Outro exemplo é um substituto de arquivo, que permite gravar em um arquivo, mas que apenas realiza a operação de Entrada e Saída no arquivo real, quando você já tiver terminado. Em vez de fazer várias gravações lentas, o substituto fará uma única gravação grande.
- Você quer proteger de forma transparente o modo como um objeto é usado. A maioria dos objetos é mutante, como as classes da linguagem Java. Um substituto protetor poderia tornar uma coleção de classes imutável, interceptando pedidos que de outro modo alterariam uma coleção de classes. Filtrando todas as chamadas de método através de um substituto, você pode governar de forma transparente as operações permitidas sobre um objeto.
- O objeto real existe remotamente, através de uma rede ou processo. Os substitutos são importantes para a computação distribuída. Um substituto pode fazer com que um recurso distribuído pareça como se fosse um recurso local, encaminhando os pedidos através de uma rede ou através de um espaço de processo.
- Quando você quer executar ações adicionais de forma transparente, ao usar um objeto. Por exemplo, talvez você queira contar o número de vezes que um método é chamado sem que o chamador saiba. Um substituto de contagem poderia controlar o acesso a um objeto.

A Tabela 11.2 destaca o usuário do padrão Proxy.

TABELA 11.2 O padrão Proxy

Nome do padrão	Proxy, Surrogate
Problema	Precisa controlar o acesso a um objeto
Solução	Fornecer um objeto que intermedie o acesso a outro objeto de forma transparente
Conseqüências	Introduz um nível de procedimento indireto no uso do objeto

O padrão ITERATOR

Se você lembrar das lições sobre acoplamento e responsabilidades, deverá ouvir uma voz em sua mente, gritando, "Código ruim!" Sempre que você quiser fazer um laço sobre uma coleção de cartas, precisará adicionar um método que saiba como fazer um laço sobre esse tipo de coleção específico. Na verdade, a lógica não é tão diferente assim de um caso para outro. Os únicos elementos que mudam são os métodos e atributos que você acessa na coleção. Agora você vê um alerta: código repetido.

O problema é que cada um dos métodos listados anteriormente é dependente da implementação da coleção, como um Deck ou um Array. Quando programa, você sempre quer garantir que não fique acoplado a uma implementação específica. O acoplamento torna seu programa resistente à mudança.

Pense a respeito. O que acontecerá se você quiser mudar a coleção que contém suas cartas? Você precisará atualizar ou adicionar um novo método que faça o laço. Se você não tiver sorte, simplesmente mudar a implementação da coleção poderia necessitar de alterações por todo o seu programa.

Embora um único objeto possa conter apenas um laço, seu programa inteiro provavelmente vai repetir os laços muitas vezes, em muitos lugares diferentes.

Outro problema com esses exemplos provém do fato de que os mecanismos de navegação estão codificados no método. É por isso que você precisa de um método para o laço para frente e outro para o laço inverso. Se você quiser fazer um laço aleatoriamente pelas cartas, então precisará de um terceiro método (e um para cada tipo de coleção). Você não precisará apenas de vários métodos, como também precisará implementar novamente a lógica de navegação, sempre que definir um laço. Infelizmente, tal duplicação de lógica é um sintoma de responsabilidade confusa. A lógica de navegação deve aparecer em um e apenas um lugar.

Felizmente, o padrão `Iterator` resolve muitos dos problemas de forte acoplamento e confusão de responsabilidade, colocando a lógica do laço ou iteração em seu próprio objeto.

A Figura 11.5 ilustra a interface `Iterator`.

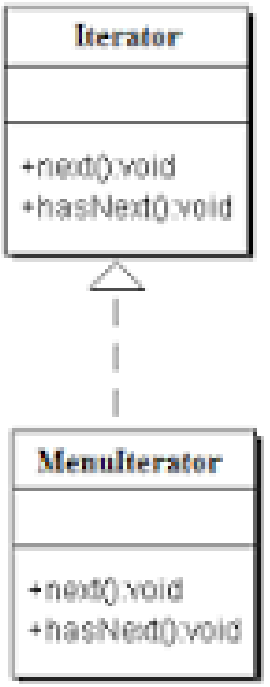
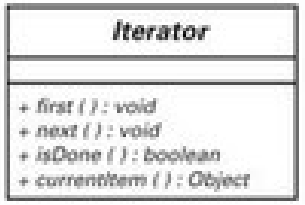


FIGURA 11.5
A interface `Iterator`.



A interface `Iterator` fornece uma interface genérica para iteragir sobre uma coleção. Em vez de escrever seus laços e métodos para usar uma coleção especifica, você pode simplesmente programar a interface genérica do `Iterator`. A interface `Iterator` oculta completamente a implementação da coleção subjacente.

Quando usar o padrão Iterator

Existem várias razões para se usar o padrão Iterator:

- Você pode usar um iterador quando quiser ocultar a implementação de uma coleção.
- Você pode usar um iterador quando quiser fornecer diferentes tipos de laços sobre uma coleção (como laço para frente, laço inverso, laço filtrado etc.).
- Você pode usar um iterador para manter a interface de uma coleção simples. Você não precisará adicionar métodos para ajudar a fazer o laço sobre o conteúdo. Basta deixar os usuários do objeto utilizarem um iterador.
- Você pode definir uma classe de coleção base que retorne um iterador. Se todas as suas coleções herdarem dessa base, o iterador permitirá que você trate todas as suas coleções genericamente. Na verdade, `java.util.Collection` faz justamente isso. Essa utilização também é a forma geral do padrão `Iterator`. O exemplo `Deck` é uma versão abreviada do padrão `Iterator`. A classe `Deck` não herda de uma classe de coleção base abstrata; assim, você não pode tratá-la genericamente.
- Os iteradores também são úteis para fornecer acesso otimizado às coleções. Algumas estruturas de dados, como a tabela hashing, não fornecem um modo otimizado de fazer a iteração sobre os elementos. Um iterador pode fornecer tal ordenação, ao custo de um pouco de memória extra. Entretanto, dependendo de seu aplicativo, a economia de tempo pode mais do que compensar a sobrecarga de memória.

A Tabela 11.3 destaca o usuário do padrão Iterator.

TABELA 11.3 O padrão Iterator

Nome do padrão	Iterator, Cursor
Problema	Fazer laço sobre uma coleção sem se tornar dependente da implementação da coleção
Solução	Fornecer um objeto que manipule os detalhes da iteração, ocultando assim os detalhes do usuário
Conseqüências	Navegação desacoplada, interface da coleção mais simples, lógica de laço encapsulada

O padrão Abstract Factory

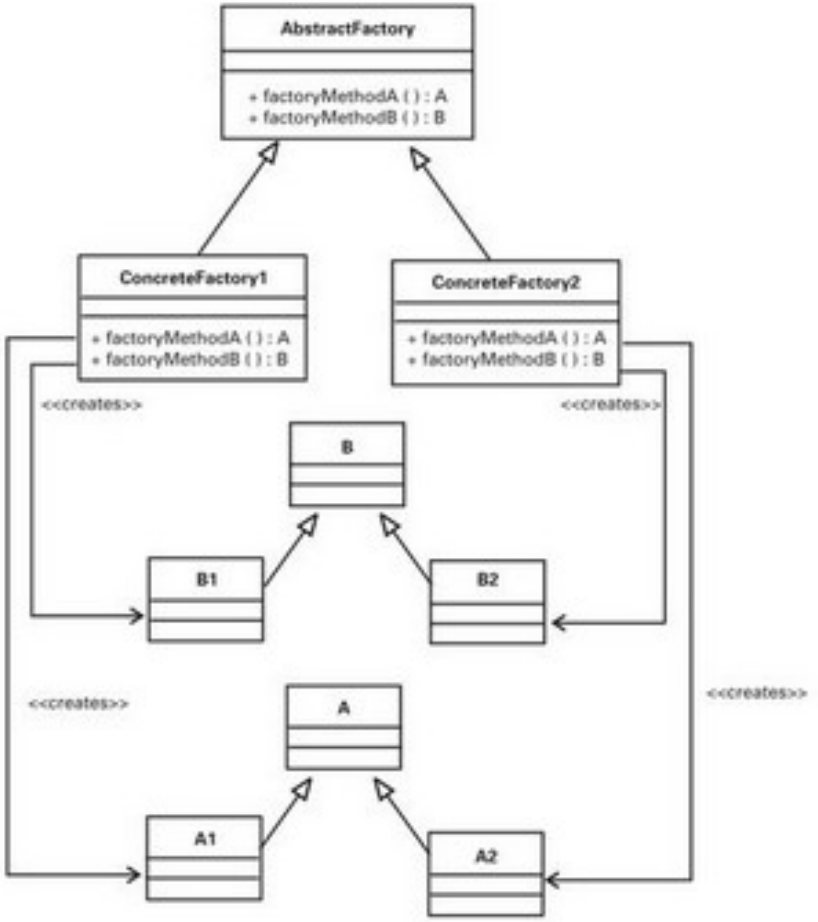
O Capítulo 7, "Polimorfismo: hora de escrever algum código", mostrou como você pode combinar herança e polimorfismo para escrever software 'à prova de futuro'. Os relacionamentos com capacidade de conexão da herança, combinados com o polimorfismo, permitem que você conecte novos objetos em seu programa, a qualquer momento; entretanto, há um inconveniente.

Para que seu programa possa instanciar esses novos objetos, você deve entrar no código e alterá-lo para que ele instancie os novos objetos, em vez dos antigos. (E você precisará fazer isso em todos os lugares onde os objetos antigos são instanciados!). Não seria ótimo se houvesse um modo mais fácil de conectar seus novos objetos?

O padrão Abstract Factory resolve esse problema através da delegação. Em vez de instanciar objetos através de seu programa, você pode delegar essa responsabilidade para um objeto chamado *factory*. Quando um objeto precisar criar outro, ele solicitará que o factory faça isso. Usando um factory, você pode isolar toda a criação de objeto em um único local. Quando você precisar introduzir novos objetos em seu sistema, só precisará atualizar o factory, para que ele crie uma instância de suas novas classes. Os objetos que usam o factory nunca saberão a diferença.

A Figura 12.1 ilustra o projeto geral do padrão Abstract Factory.

FIGURA 12.1
O padrão Abstract Factory.



Quando usar o padrão Abstract Factory

Use o padrão Abstract Factory, quando:

- Você quiser ocultar o modo como um objeto é criado.
- Você quiser ocultar a classe atual do objeto criado.
- Você quiser um conjunto de objetos usados juntos. Isso evita que você use objetos incompatíveis juntos.
- Você quiser usar diferentes versões de uma implementação de classe. Um Abstract Factory permite que você troque essas diferentes versões em seu sistema.

A Tabela 12.1 destaca o usuário do padrão Abstract Factory.

TABELA 12.1 O padrão Abstract Factory

Nome do padrão	Abstract Factory
Problema	Precisa de uma maneira de trocar objetos plugáveis de forma transparente
Solução	Fornecer uma interface abstrata que providencie métodos para instanciar os objetos
Conseqüências	Permite que você troque facilmente novos tipos de classe em seu sistema; entretanto, é dispendioso adicionar tipos não relacionados

O padrão Singleton

Quando projetar seus sistemas, você verá que algumas classes deveriam ter logicamente apenas uma instância, como um factory ou um objeto que acesse algum recurso não compartilhado (conexão de banco de dados, região de memória etc.). Nada, entretanto, impedirá que um objeto instancie outro. Como você impõe seu projeto?

O padrão Singleton fornece a resposta dessa pergunta. O padrão Singleton impõe seu projeto colocando a responsabilidade da criação e da intermediação do acesso à instância no próprio objeto. Fazer isso garante que apenas uma instância seja criada, além de fornecer um único ponto de acesso para essa instância. A Figura 12.2 ilustra a assinatura de uma classe singleton (carta única de determinado naipe).

FIGURA 12.2
O padrão Singleton

Quando usar o padrão Singleton

Use o padrão Singleton quando você quiser restringir uma classe a ter apenas uma instância.

A Tabela 12.2 destaca o usuário do padrão Singleton.

TABELA 12.2 O padrão Singleton

Nome do padrão	Singleton
Problema	Deve existir apenas uma instância de um objeto no sistema em determinado momento.
Solução	Permitir que o objeto gerencie sua própria criação e acesso através de um método de classe.

TABELA 12.2 O padrão Singleton (*continuação*)

Conseqüências	Acesso controlado à instância do objeto. Também pode dar acesso a um número definido de instâncias (como apenas seis instâncias), com uma ligeira alteração no padrão. É um pouco mais difícil herdar um singleton.

O padrão Typesafe Enum

Ao instanciar objetos `Card`, você precisa passar uma constante de número e de naipe válida. A utilização de constantes dessa maneira pode levar a muitos problemas. Nada o impede de passar qualquer `int` que você queira. E, embora você deva referenciar apenas o nome da constante, isso abre a representação interna de `Card`. Por exemplo, para conhecer o naipe da carta (`Card`), você deve recuperar o valor `int` e depois compará-lo com as constantes. Embora isso funcione, essa não é uma solução limpa.

O problema reside no fato de que naipe e número são objetos propriamente ditos. `int` não resolve isso, pois você precisa aplicar um significado a `int`. Novamente, isso confunde a responsabilidade, pois você precisará reaplicar esse significado sempre que encontrar um `int` que represente um número ou um naipe.

Linguagens como C++ têm uma construção, conhecida como *enumeração*; entretanto, as enumerações se reduzem simplesmente a um atalho para declarar uma lista de constantes inteiras. Essas constantes são limitadas. Por exemplo, elas não podem fornecer comportamento. Também é difícil adicionar mais constantes.

Em vez disso, o padrão Typesafe Enum fornece uma maneira OO de declarar suas constantes. Em vez de declarar simples constantes inteiras, você cria classes para cada tipo de constante. Para o exemplo `Card`, você criaria uma classe `Rank` (número) e uma classe `Suit` (naipe). Então, você criaria uma instância para cada valor de constante que quisesse representar e a tornaria publicamente disponível a partir da classe (através de public final, exatamente como as outras constantes).

Quando usar o padrão Typesafe Enum

Use o padrão Typesafe Enum, quando:

- Você se achar escrevendo numerosas primitivas públicas ou constantes de String.
- Você se achar impondo identidade em um valor, em vez de derivar a identidade do próprio valor. A Tabela 12.3 destaca o usuário do padrão Typesafe Enum.

TABELA 12.3 O padrão Typesafe Enum

Nome do padrão	Typesafe Enum.
Problema	As constantes inteiras são limitadas.
Solução	Criar uma classe para cada tipo de constante e depois fornecer instâncias de constante para cada valor de constante.
Conseqüências	Constantes OO extensíveis. Constantes úteis que têm comportamento. Você ainda precisa atualizar código para usar as novas constantes, quando elas forem adicionadas. Exige mais memória do que uma constante simples.

Como desacoplar a UI usando o padrão Model View Controller

O padrão de projeto MVC (Model View Controller) fornece uma estratégia para o projeto de interfaces com o usuário que desacoplam completamente o sistema subjacente da interface com o usuário.

NOTA

MVC é apenas uma estratégia para o projeto de interfaces com o usuário orientadas a objetos. Existem outras estratégias válidas para o projeto de interface com o usuário; entretanto, a MVC é uma estratégia testada que é popular no setor do software. Se você acabar realizando o trabalho da interface com o usuário, especialmente em relação à Web e ao J2EE da Sun, encontrará o MVC.

O Document/View Model, popularizado pelas Microsoft Foundation Classes e o padrão de projeto PAC (*Presentation Abstraction Control*), fornecem alternativas à MVC. Veja o livro *Pattern-Oriented Software Architecture A System of Patterns*, de Frank Buschmann et al, para uma apresentação completa dessas alternativas.

O padrão MVC desacopla a UI do sistema, dividindo o projeto da UI em três partes separadas:

- O modelo, que representa o sistema
- O modo de visualização, que exibe o modelo
- O controlador, que processa as entradas do usuário

Cada parte da tríade MVC tem seu conjunto próprio de responsabilidades exclusivas.

O modelo

O modelo é responsável por fornecer:

- Acesso à funcionalidade básica do sistema
- Acesso às informações de estado do sistema
- Um sistema de notificação de mudança de estado

O modelo é a camada da tríade MVC que gerencia o comportamento básico e o estado do sistema. O modelo responde às consultas sobre seu estado a partir do modo de visualização e do controlador e aos pedidos de mudança de estado do controlador.

Um sistema pode ter muitos modelos diferentes. Por exemplo, um sistema de banco pode ser constituído de um modelo de conta e um modelo de caixa. Vários modelos pequenos repartem melhor a responsabilidade do que um único modelo grande.

Não deixe o termo *modelo* confundi-lo. Um modelo é apenas um objeto que representa o sistema.

O controlador é a camada da tríade MVC que interpreta a entrada do usuário. Em resposta à entrada do usuário, o controlador pode comandar o modelo ou o modo de visualização para que mude ou execute alguma ação.

O modo de visualização é a camada da tríade MVC que exibe a representação gráfica ou textual do modelo. O modo de visualização recupera todas as informações de estado a respeito do modelo, a partir do modelo.

Em qualquer caso, o modelo não sabe absolutamente que um modo de visualização ou controlador está fazendo uma chamada de método. O modelo só sabe que um objeto está chamando um de seus métodos. A única conexão que um modelo mantém com a UI é através do sistema de notificação de mudança de estado.

Se um modo de visualização ou controlador estiver interessado na notificação de mudança de estado, ele se registrará no modelo. Quando o modelo mudar de estado, percorrerá sua lista de objetos registrados (freqüentemente chamados de receptores ou observadores) e informará cada objeto da mudança de estado. Para construir esse sistema de notificação, os modelos normalmente empregarão o padrão Observer.

O padrão Observer

O padrão `Observer` fornece um projeto para um mecanismo de publicação/assinatura entre objetos. O padrão `Observer` permite que um objeto (o observador) registre seu interesse em outro objeto (o observável). Quando o observável quiser notificar os seus observadores de uma alteração, ele chamará um método `update()` em cada observador.

A Listagem 13.2 define a interface `Observer`. Todos os observadores que quiserem se registrar com o objeto observável devem implementar a interface `Observer`.

LISTAGEM 13.2 *Observer.java*

```
public interface Observer {
   public void update();
}
```

Um observável fornecerá um método, através do qual os observadores podem registrar e anular o registro de seu interesse em atualizações. A Listagem 13.3 apresenta uma classe que implementa o padrão `Observer`.

O modo de visualização

O modo de visualização é responsável por:

- Apresentar o modelo para o usuário
- Registrar no modelo notificação de mudança de estado
- Recuperar informações de estado do modelo

O modo de visualização é a camada da tríade MVC que exibe informações para o usuário. O modo de visualização obtém informações de exibição do modelo, usando a interface pública deste, e também se registrará no modelo para que ele possa ser informado da mudança de estado e se atualize de acordo.

Um único modelo pode ter muitos modos de visualização diferentes.

O controlador

O controlador é responsável por:

- Interceptar os eventos do usuário do modo de visualização
- Interpretar o evento e chamar os métodos corretos do modelo ou modo de visualização
- Registrar-se no modelo para notificação de mudança de estado, se estiver interessado

O controlador atua como a cola entre o modo de visualização e o modelo. O controlador intercepta eventos do modo de visualização e depois os transforma em pedidos do modelo ou do modo de visualização.

NOTA

Um modo de visualização tem apenas um controlador e um controlado tem apenas um modo de visualização. Alguns modos de visualização permitem que você configure seu controlador diretamente.

Cada modo de visualização tem um controlador e toda a interação com o usuário passa por esse controlador. Se o controlador for dependente das informações de estado, ele também será registrado no modelo para notificação de mudança de estado.

Testando software OO

A OO não evitará que erros aconteçam em seu software. Mesmo os melhores programadores cometem erros. Os erros são normalmente considerados como um defeito de software que surge de um erro de digitação, de um erro na lógica ou apenas por um engano bobo cometido durante a codificação. Embora a implementação de um objeto seja uma fonte de erros comum, eles aparecem em outras formas.

Erros também podem resultar quando um objeto usa outro incorretamente. Os erros podem até ser provenientes de falhas básicas na análise ou no próprio projeto. Por sua própria natureza, um sistema OO é repleto de objetos interagindo. Essas interações podem ser a fonte de todos os tipos de erros.

Felizmente, você pode proteger seu software de erros, através de testes de software, onde é possível validar a análise, o projeto e a implementação de seu software.

ALERTA

Assim como a OO, o teste não é uma solução mágica; é extremamente difícil testar seu software completamente. O número total de caminhos possíveis através de um programa não trivial torna difícil e demorado obter cobertura total do código. Assim, mesmo um código testado pode abrigar erros ocultos.

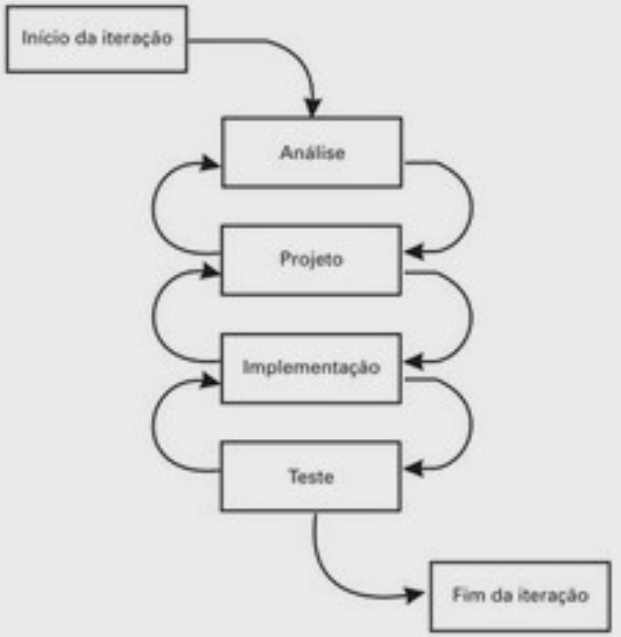
O melhor que você pode fazer é realizar uma quantidade de testes que garantam a qualidade de seu código, enquanto também permitam cumprir seus prazos finais e permanecer dentro do orçamento. A 'quantidade' real de testes que você realizará dependerá da abrangência do projeto e de seus próprios níveis de bem-estar.

Testes e o processo de desenvolvimento de software iterativo

A Figura 14.1 ilustra a iteração apresentada pela primeira vez no Capítulo 9, "Introdução à análise orientada a objetos". O teste é a última etapa de uma iteração.

Antes de você sair de uma iteração, o teste é uma etapa importante. O estágio de testes verifica se todas as alterações feitas por você durante essa iteração não danificaram qualquer funcionalidade existente. O estágio de testes também verifica se toda nova funcionalidade adicionada agora funciona corretamente. Por esses motivos, os testes realizados antes de se sair de uma iteração são freqüentemente referidos como testes *funcionais* ou de *aceitação*.

FIGURA 14.1
Uma iteração.



NOTA

O teste no final de uma iteração é um marco importante. Para sair da iteração, seu sistema deve passar pelos testes; entretanto, também devem ocorrer testes durante outros estágios de uma iteração. As lições de hoje mostrarão a você como usar testes eficientemente, durante a implementação e estágios de testes da iteração.

Torne os testes um objetivo e algo que você faz por todo o desenvolvimento. Se você não pensar nos testes até o final do desenvolvimento, poderá descobrir que não é possível testar seu software. Em vez disso, você precisa desenvolver pensando nos testes. Você deve tornar os testes uma parte integrante do processo de desenvolvimento.

Se erros forem encontrados, você deverá voltar e corrigi-los. Normalmente, você voltará para a implementação e tentará corrigir o problema no código. Às vezes, isso é tudo que você precisará fazer: bastará corrigir a implementação, testar tudo novamente e prosseguir. Entretanto, os erros podem ser provenientes de uma falha de projeto ou mesmo de um requisito ignorado ou mal-entendido. Talvez você precise voltar ao projeto ou à análise, antes de poder corrigir um erro na implementação.

Após corrigir um erro, não é suficiente apenas testar o erro corrigido. Em vez disso, você precisa realizar todos os testes. Ao corrigir um erro, você pode introduzir facilmente um ou mais erros novos!

Um mal-entendido na análise significa que o sistema não funcionará conforme o cliente espera. Um sistema deve funcionar conforme o esperado e o cliente que conduz o sistema deve concordar com o que é esperado do comportamento. Não apenas você precisa testar o código quanto a

falhas de implementação, como também precisa testar o código para ver se ele funciona conforme o esperado.

Para testar um sistema, você precisa escrever e executar casos de teste. Cada caso de teste testará um aspecto específico do sistema.

NOVO TERMO Um *caso de teste* é o bloco de construção básico do processo de teste. O processo de teste executa vários *casos de teste* para poder validar completamente um sistema. Cada *caso de teste* consiste em um conjunto de entradas e saídas esperadas. O teste executará um caminho específico através do sistema (caixa branca) ou testará algum comportamento definido (caixa preta).

Um caso de teste exercita uma funcionalidade específica para ver se o sistema se comporta como deveria. Se o sistema se comportar conforme o esperado, o caso de teste passa. Se o sistema não se comportar conforme o esperado, o caso de teste falha. Um caso de teste falho indica que existe um erro no sistema. Você sempre quer que todos os seus casos de teste passem 100% das vezes. Não tente ignorar um caso de teste falho, se cem outros casos passarem. Todo teste deve passar ou você não poderá continuar seu trabalho!

Existem duas maneiras de basear seus casos de teste: teste de caixa preta e de caixa branca. Uma estratégia de teste eficaz terá uma mistura de casos de teste baseados em caixa preta e em caixa branca.

NOVO TERMO O *teste de caixa preta* testa se o sistema funciona conforme o esperado. Dada uma entrada especifica, o *teste de caixa preta* testa se a saída ou comportamento correto, visivel externamente, resulta conforme definido pela especificação da classe ou do sistema.

NOVO TERMO No *teste de caixa branca*, os testes são baseados unicamente na implementação de um método. Os testes de caixa branca tentam atingir 100% de cobertura do código.

Ao testar classes individuais, o teste de caixa preta é baseado nos requisitos funcionais da classe. Ao testar o sistema inteiro, o teste de caixa preta é baseado nos casos de uso. Em qualquer caso, o teste de caixa preta verifica se um objeto ou sistema se comporta conforme o esperado. Por exemplo, se um método deve somar dois números, um teste de caixa preta enviará dois números para o método e, em seguida, verificará se a saída é ou não a soma correta dos dois números. Se um sistema deve permitir que você adicione e remova itens de um carrinho de compras, um teste de caixa preta tentará adicionar e remover itens do carrinho.

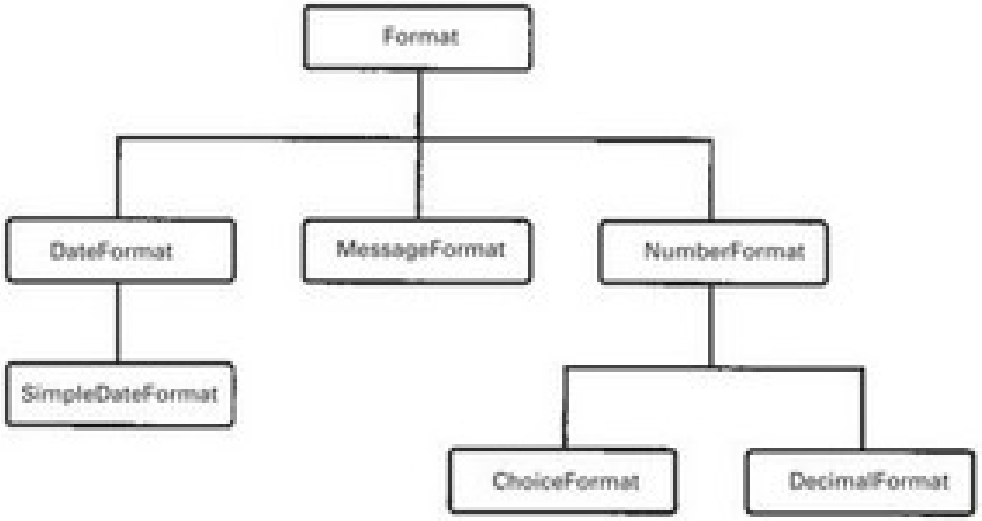
O teste de caixa branca, por outro lado, é baseado na implementação de um método. Seu objetivo é garantir que cada desvio do código seja executado. O teste de caixa preta é avaliado para cobrir apenas de um terço à metade do código real. Com o teste de caixa branca, você projeta seus testes de modo a exercitar cada desvio do código e na esperança de eliminar todos os erros latentes.

Aprendendo a navegar na teia emaranhada da herança

Os conceitos de 'é um' e composição mudam a natureza da discussão sobre herança da ambiciosa reutilização da implementação para inter-relacionamentos de classe. Uma classe que herda de outra deve se relacionar com essa classe de alguma maneira, para que os relacionamentos ou hierarquias de herança resultantes façam sentido.

NOVO TERMO Uma *hierarquia de herança* é um mapeamento do tipo árvore de relacionamentos que se formam entre classes como resultado da herança. A Figura 4.3 ilustra uma hierarquia real extraida da linguagem Java.

FIGURA 4.3
Um exemplo de hierarquia de `java.text`.



DICA

Os testes de caixa branca, exceto quanto aos programas mais simples, raramente podem alcançar uma cobertura razoável da combinação de caminhos através do programa. Existem dois passos que você pode dar para melhorar a eficiência de seus testes:

- Escreva seus programas de modo que eles tenham um número mínimo de caminhos.
- Identifique caminhos críticos e certifique-se de testá-los.

Por exemplo, se um método divide dois números e existe um desvio de erro que é executado quando você tenta dividir por 0, será preciso garantir que exista um caso de teste que exercite essa condição de erro. A não ser que seja especificado na documentação da interface, você saberia a respeito desse desvio examinando o próprio código. Assim, os testes de caixa branca devem ser baseados no próprio código.

Em qualquer caso, os testes de caixa preta e de caixa branca governam o modo como você cria seus casos de teste. Cada um desempenha um papel importante na forma de teste que você pode executar.

Formas de teste

No todo, existem quatro formas importantes de teste. Esses testes variam de testes de nível mais baixo, que examinam os objetos individuais, até os testes de nível mais alto, que examinam o sistema inteiro. A execução de cada um ajudará a garantir a qualidade global de seu software.

Teste de unidade

O teste de unidade é a unidade de nível mais baixo dos testes. Um teste de unidade examina apenas um recurso por vez.

NOVO TERMO Um *teste de unidade* é o dispositivo de teste de nível mais baixo. Um teste de unidade envia uma mensagem para um objeto e depois verifica se ele recebe o resultado esperado do objeto. Um teste de unidade verifica apenas um recurso por vez.

Em termos de OO, um teste de unidade examina uma única classe de objeto. Um teste de unidade verifica um objeto enviando uma mensagem e verifica se ele retorna o resultado esperado. Você pode basear os testes de unidade no teste de caixa preta e no de caixa branca. Na verdade, você deve realizar ambos, para garantir que seus objetos funcionem corretamente. Embora cada classe escrita deva ter um teste de unidade correspondente, você provavelmente deve escrever o caso de teste antes de escrever a classe. Você vai ler mais sobre esse ponto posteriormente.

Hoje, focalizaremos o teste de unidade, pois ele é fundamental para a escrita de software OO confiável. Na verdade, você deve realizar os testes de unidade por todo o desenvolvimento.

Teste de integração

Os sistemas OO são constituídos de objetos que interagem. Enquanto os testes de unidade examinam cada classe de objeto isoladamente, os testes de integração verificam se os objetos que compõem seu sistema interagem corretamente. O que poderia funcionar isoladamente pode não funcionar quando combinado com outros objetos! As fontes comuns de erros de integração são provenientes de erros ou mal-entendidos a respeito dos formatos de entrada/saída, conflitos de recurso e seqüência incorreta de chamadas de método.

NOVO TERMO Um *teste de integração* verifica se dois ou mais objetos funcionam em conjunto corretamente.

Assim como os testes de unidade, os testes realizados durante os testes de integração podem ser baseados nos conceitos de caixa branca e de caixa preta. Você deve ter um teste de integração para cada iteração importante no sistema.

Teste de sistema

Os testes de sistema verificam se o sistema inteiro funciona conforme descrito pelos casos de uso. Enquanto executa testes de sistema, você também deve testar o sistema de maneiras não descritas pelos casos de uso. Fazendo isso, você pode verificar se o sistema manipula e se recupera normalmente de condições imprevistas.

DICA

Tente fazer estes testes em seu sistema:

Testes de ação aleatória

Os testes de ação aleatória consistem em tentar executar operações em ordem aleatória.

Testes de banco de dados vazio

Os testes de banco de dados vazio garantem que o sistema pode falhar normalmente, caso exista um problema maior no banco de dados.

Casos de uso mutantes

Um caso de uso mutante transforma um caso de uso válido em um caso de uso inválido e garante que o sistema possa se recuperar corretamente da interação.

Você ficaria surpreso com o que um usuário pode tentar fazer com seu sistema. Ele pode não cair sob um dos casos de uso 'normais'; portanto, é melhor estar preparado para o pior.

NOVO TERMO

Um *teste de sistema* examina o sistema inteiro. Um teste de sistema verifica se o sistema funciona conforme mencionado nos casos de uso e se ele pode manipular normalmente situações incomuns e inesperadas.

Os testes de sistema também incluem testes de esforço e de desempenho. Esses testes garantem que o sistema satisfaça quaisquer requisitos de desempenho e possa funcionar sob as cargas esperadas. Se possível, é melhor executar esses testes em um ambiente que corresponda o máximo possível ao ambiente de produção.

Os testes de sistema são um aspecto importante dos testes de aceitação. Os testes de sistema verificam unidades funcionais inteiras simultaneamente, de modo que um único teste pode mexer com muitos objetos e subsistemas diferentes. Para sair de uma iteração, o sistema deve passar nesses testes com êxito.

Teste de regressão

Um teste é válido apenas enquanto o que for testado não mudar. Quando um aspecto do sistema mudar, essa parte — assim como todas as partes dependentes — deverá ser novamente testada. Teste de regressão é o processo de repetição dos testes de unidade, integração e de sistema após as alterações serem feitas.

NOVO TERMO Os *testes de regressão* examinam as alterações nas partes do sistema que já foram validadas. Quando uma alteração é feita, a parte que foi alterada — assim como todas as partes dependentes — deve ser novamente testada.

É absolutamente fundamental testar novamente, mesmo depois de uma pequena alteração. Uma pequena alteração pode introduzir um erro que poderia danificar o sistema inteiro. Felizmente, para executar o teste de regressão basta executar novamente seus testes de unidade, integração e sistema.

Resumo

Hoje, você aprendeu sobre os testes e o que pode fazer como desenvolvedor para garantir a qualidade de seu trabalho. Ao todo, existem quatro formas gerais de teste:

- Teste de unidade
- Teste de integração
- Teste de sistema
- Teste de regressão

Em seu trabalho diário, o teste de unidade é sua primeira linha de defesa contra erros. O teste de unidade também tem as vantagens de obrigá-lo a considerar seu projeto do ponto de vista do teste e de fornecer a você um mecanismo que torna mais fácil refazer as coisas.

Você também viu a importância do tratamento correto das condições de erro e da manutenção da documentação. Todas essas práticas aumentam a qualidade de seu código.