POO-SINTES - NARCISO

Introdução à programação orientada a objeto

A OO inclui qualquer estilo de desenvolvimento que seja baseado no conceito de "objeto" , a OO contém tudo que pode ser denominado como orientado a objetos. A Programação Orientada a Objetos estrutura um programa, dividindo-o em vários objetos de alto nível

1.Precursores da POO

Programação procedural

As linguagens procedurais permitem ao programador reduzir um programa em procedimentos refinados para processar dados.

- A programação procedural limita a reutilização de código.

Deficiências da programação modular:

Os módulos não são extensíveis (não pode fazer alterações incrementais em um módulo sem que abra o código a força para fazer alterações diretamente)
Não é possível basear um módulo em outro (isso só é possível através da delegação.)
Um módulo pode definir um tipo mas não pode compartilhar o tipo de outro módulo.

Procedimentos refinados

- ➢ Definem a estrutura global do programa.
- ➢ Limita a reutilização de código

Programação modular

- ➢ Divide o programa em vários componentes ou módulos
- ➢ Um módulo contém informações de estado que podem mudar a qualquer momento.

Estado

- ➢ É o significado combinado das variáveis internas do objeto

Variável interna

- ➢ Valor mantido dentro de um objeto

2. Uma estratégia de POO para software usando objetos

Implementação

- ➢ Define como algo é feito
- ➢ É o código

Domínio

- ➢ É um espaço onde um problema reside
- ➢ É o conjunto de conceitos que representam os aspectos importantes do problema que você está tentando resolver.

O que é um objeto ?

- ➢ É uma instância de uma classe.
- ➢ Uma construção de software que encapsula estado e comportamento.
- ➢ Em uma linguagem de programação OO pura, tudo é um objeto.

O que é uma classe ?

- ➢ Define os atributos e comportamentos comuns compartilhados por um tipo de objeto
- ➢ Usada para criar ou instanciar objetos.
- ➢ defina quais mensagens seus objetos respondem.

Atributos

- ➢ São características de uma classe visíveis externamente.

Um objeto pode exercer o comportamento de outro, executando uma operação sobre esse objeto.

Comportamento

- ➢ É uma ação executada por um objeto, (quando passada uma mensagem ou em resposta a uma mudança de estado) **é algo que um objeto faz**

Um objeto pode exercer o comportamento de outro, executando uma operação sobre esse objeto.

Instanciação

- ➢ É a criação de objetos.
- ➢ ("cria um novo objeto de um objeto já existente na classe")

Construtores

- ➢ São métodos usados para inicializar objetos durante sua instanciação.

(Você chama a criação de objetos de instanciação porque ela cria uma instância do objeto da classe).

- ➢ No construtor, pode-se notar o uso de this.

This

- ➢ É uma referência que aponta para instância do objeto. (a instância usa essa referência para acessar suas próprias variáveis e métodos. Faz referência para o atributo

Acessores

- ➢ Dão acesso aos dados internos de um objeto, eles ocultam o fato de os dados estarem em uma variável, ou serem calculados.
- ➢ Permitem que você mude ou recupere o valor e têm "efeitos colaterais" sobre o estado interno.

Mutantes

- ➢ Permitem que você altere o estado interno de um objeto

6. Relacionamentos de objeto

Os objetos podem existir independentemente uns dos outros. Dois objetos podem aparecer no carrinho de compras ao mesmo tempo, Se esses dois objetos separados precisarem interagir, eles interagirão passando mensagem um para o outro.

Mensagem

Como os objetos se comunicam, as mensagens fazem com que um objeto realize algo.
As mensagens permitem que os objetos permaneçam independentes

'Passar uma mensagem' é o mesmo que chamar um método para mudar o estado do objeto para exercer um comportamento.

7. Vantagens e objetivos da OO:

NATURAL :: MANUTENÍVEL :: OPORTUNO :: EXTENSÍVEL :: REUTILIZÁVEL :: CONFIÁVEL

NA - MA - OPO - EX - RE- CO

NATURAL: Os programas naturais são mais inteligíveis. A POO permite que você modele um problema em um nível funcional e não em nível de implementação.

CONFIÁVEL: Programas OO, bem projetados e cuidadosamente escritos são confiáveis. A OO aprimora os testes, permitindo que você isole conhecimento e responsabilidade em um único lugar.

Permite que você faça alterações em uma parte do programa sem afetar outras.

Permite que você teste e valide cada componente independentemente. Uma vez que tenha validado, você pode reutilizá-lo com confiança.

REUTILIZÁVEL

Uma vez que um problema esteja resolvido, você deve reutilizar a solução. Você pode reutilizar prontamente classes orientadas a objetos bem feitas. Assim como os módulos, você pode reutilizar objetos em muitos programas diferentes. A POO introduz herança para permitir que você estenda objetos existentes e o polimorfismo, para que você possa escrever código genérico. A OO não garante código genérico.

MANUTENÍVEL

Um código orientado a objetos bem projetado é manutenível. Como uma mudança na implementação é transparente, todos os outros objetos se beneficiarão. A linguagem natural do código deve permitir que outros desenvolvedores também o entendam.

EXTENSÍVEL

Quando você construir uma biblioteca de objetos, também desejará estender a funcionalidade de seus próprios objetos. O software não é estático. Ele deve crescer e mudar com o passar do tempo, para permanecer útil. A POO apresenta ao programador vários recursos para estender o código. Esses recursos incluem herança, polimorfismo, sobreposição, delegação e uma variável de padrões de projetos.

OPORTUNO

Quando você divide um programa em vários objetos, o desenvolvimento de cada parte pode ocorrer em paralelo. Vários desenvolvedores podem trabalhar nas classes independentemente. Tal desenvolvimento em paralelo leva a tempo de desenvolvimento menores.

4 armadilhas a serem evitadas ao aprender OO pela primeira vez:

- **Pensar na OO simplesmente como uma linguagem**
- **Medo da reutilização**

- **Pensar na OO como solução paraa tudo**
- **Programação egoísta**

Encapsulamento: aprenda a manter os detalhes consigo mesmo

Os 3 pilares da POO: encapsulamento, herança, polimorfismo

Encapsulamento

É a característica da OO de ocultar partes independentes da implementação;
Permite que atinjam uma funcionalidade e ocultam os detalhes de implementação do mundo exterior.

Forma a base da herança e do polimorfismo.
Tornar transparentes as alterações no objeto
Não causar efeitos inesperados entre o objeto e o restante do programa
Poder reutilizar o objeto em qualquer parte

Permite que você divida em várias partes menores e independentes. Cada parte possui implementação e realiza seu trabalho independentemente das outras partes. Ocultando os detalhes internos ou seja, a implementação de cada parte, através de uma interface externa.

Interface: lista os serviços fornecidos por um componente. Define o que uma entidade externa pode fazer com um objeto. É o painel de controle do objeto.
As mudanças na implementação não mudam o código que usa a classe, desde que a mudança no código que exerce essa interface (a manutenção da implementação não fica exposta) (declaração dos métodos) (é oq será exposto)

Implementação: define como um componente fornece um serviço. Define os detalhes internos do componente. (É o código.)
Define como algo é feito.

3.Público, privado, protegido

Público > Garante o acesso a todos os objetos
Protegido > Garante o acesso à instância, ou seja, para aquele objeto e para todas as subclasses
Privado > Garante o acesso apenas para a instância, ou seja, para aquele objeto.

4. Por que você deve encapsular ?

Independência: Você pode reutilizar o objeto em qualquer parte, Quando você encapsular corretamente, eles não estarão vinculados a nenhum programa em particular. Você poderá usá-lo quando seu uso fizer sentido. Para usar o objeto em qualquer lugar, você simplesmente exerce sua interface.

O encapsulamento: Permite que você torne transparentes as alterações em seu objeto. Desde que não altere sua interface.
O encapsulamento permite que você atualize seu componente, forneça uma implementação mais eficiente ou corrija erros. Tudo isso sem ter de tocar nos outros objetos. Os usuários de seu objeto se beneficiarão automaticamente com todas as alterações que você fizer.

Usar um objeto encapsulado não causará efeitos colaterais inesperados entre o objeto e o restante do programa.
As 3 características do encapsulamento:
> Abstração
>Ocultação da implementação
>Divisão de responsabilidade
Ocul-Div-ABs

5. O que é abstração ?

A abstração tem duas vantagens. Primeiro, ela permite que você resolva um problema facilmente. Ajuda a obter reutilização. Muitas vezes, os componentes de software são demasiadamente especializados, Essa

especialização, combina com uma interdependência desnecessária entre os componentes, torna difícil reutilizar um código existente em outra parte. A abstração permite que você resolva um problema uma vez e depois use essa solução por todo o domínio desse problema

6. Abstração eficaz

- Trate do caso geral e não do caso específico.
- Ao confrontar vários problemas diferentes, procure o que for comum a todos, tente ver um conceito e não um caso específico.
- A abstração é valiosa, mas não descuide do problema na esperança de escrever um código abstrato.
- A abstração pode não estar prontamente aparente.
- É quase impossível escrever uma abstração que funcione em todas as situações.

7. Guardando seus segredos através da ocultação da implementação

A ocultação da implementação tem duas vantagens:

- **Protege seu objeto de seus usuários**
- **Protege os usuários de seu objeto do próprio objeto.**

8. Protegendo seu objeto através do TAD (Abstract data type - tipo abstrato de dados)

TAD

É um conjunto de dados e um conjunto de operações sobre esses dados. Permitem que você defina novos tipos na linguagem que são seguros de usar e que crie novas palavras de programação , onde você expressa uma nova ideia. ocultando dados internos e o estado. Você pode usar tads sem depender da herança e o polimorfismo. (É um tipo abstrato de dados, um tipo que você pode criar.)

Tipos

Definem as diferentes espécies de valores que podem ser usados no programa

Objeto de primeira classe

É aquele que pode ser usado exatamente da mesma maneira que um tipo interno. (Usa como se fosse uma primitiva)

Objeto de segunda classe

É um tipo de objeto que você pode definir, mas não necessariamente usar, como faria com um tipo interno. (Usa de outra forma)

9. protegendo outros de seus segredos através da ocultação da implementação

Proteger seu objeto de uso não projetado é uma vantagem da ocultação da implementação. (Não vincular código, o objetivo da OO é criar códigos fracamente acoplado, (manutenível)

Código fracamente acoplado

É independente da implementação de outros componentes.

Código fortemente acoplado

É fortemente vinculado à implementação de outros componentes.

Código dependente

É dependente da existência de determinado tipo. é inevitável, entretanto, existem graus para a dependência aceitável e para sua superdependência.
Existem graus para dependência, você não pode eliminar a dependência totalmente, entretanto, você deve se esforçar para minimizar a dependência entre objetos.
Você limita tal dependência programando uma interface bem definida. Os usuários só podem se tornar dependentes quanto ao que você decide colocar na interface. O código fortemente acoplado anula o objetivo do encapsulamento: criar objetos independentes e reutilizáveis.

10.Divisão da responsabilidade: preocupando-se com seu próprio usuário

A ocultação da implementação é apenas um passo na direção da escrita de código fracamente acoplado. Para ter realmente código fracamente acoplado, deve-se ter uma divisão de responsabilidade correta. Cada objeto deve executar uma função (sua responsabilidade) e executá-la bem. Significa que o objeto é coesivo. Não faz sentido encapsular muitas funções aleatórias e variáveis. Elas precisam ter um forte vínculo conceitual entre si. Todas as funções devem trabalhar no sentido de uma responsabilidade comum.
O encapsulamento está completamente ligado à ocultação de detalhes. Se um objeto tiver responsabilidades demais, sua implementação se tornará muito confusa e difícil de manter e estender. Quando um objeto fica grande demais, ele quase se torna um programa completo e cai nas armadilhas procedurais, Como resultado, você se depara com todos os problemas que encontraria em um programa que não usasse nenhum encapsulamento.

Encapsulamento efetivo

É abstração mais ocultação da implementação mais responsabilidade.
Retire a abstração e você terá um código que não é reutilizável. Retire a ocultação da implementação e você ficará com um código fortemente acoplado e frágil. Retire a responsabilidade e você ficará com um código centrado nos dados, procedural, fortemente acoplado e descentralizado.
Você não pode ter um encapsulamento efetivo, mas a falta de responsabilidade o deixa com a pior situação de todas: programação procedural em um ambiente orientado a objetos.

11.Estudo de caso - pacote de primitivas JAVA

Linguagem orientada

Uma linguagem orientada a objetos pura suporta a noção de que tudo é um objeto.
 Em uma linguagem puramente orientada a objetos, tudo: classes, primitivas, operadores e até blocos de código é considerado objeto.
Uma linguagem orientada a objetos não considera tudo um objeto
Vantagens das primitivas: não é preciso instanciar uma nova instância usando new.
É muito mais eficiente do que usar um objeto, pois ela não sofre sobrecarga associada aos objetos.

Na linguagem Java, nem tudo é um objeto. Você não pode tratar primitivas como objetos. Isso significa que você não pode usá-las em lugares que exigem um objeto.

Pacote

É um objeto cujo único propósito é conter outro objeto ou primitiva. Um pacote fornecerá qualquer número de métodos para obter e manipular o valor possuído. (**pacote= uma sacola de objetos**)

Cada variável e cada método está ligado a alguma instância de objeto. Para chamar o método ou acessar a variável, você deve ter uma instância do objeto.
Os métodos e variáveis de classe não estão vinculados a nenhuma instância. Em vez disso, você acessa métodos de classe através da própria classe.

Variáveis de classe

São variáveis que pertencem a classe não a uma instância específica. As variáveis de classe são compartilhadas entre todas as instâncias da classe.

Métodos de classe

São métodos que pertencem a classe e não a uma instância específica. A operação executada

pelo método não depende do estado de qualquer instância.

As variáveis de classe funcionam da mesma maneira. Você não precisa de uma instância para acessá-las.

Herança

Abrange o desejo do programador de evitar trabalho repetitivo
É impossível introduzir erros na subclasse que quebrem dependências não expostas da superclasse
Ela simplifica o código

É um mecanismo que permite a você basear uma nova classe na definição de uma classe previamente existente. Sua nova classe herda todos os atributos e comportamentos presentes na classe previamente existente. Quando uma classe herda de outra, todos os métodos e atributos aparecerão automaticamente na interface da nova classe.

Delegação

É o processo de um objeto passar uma mensagem para outro objeto, para atender algum pedido.

2. Por que herança ?

A herança permite à classe que está herdando redefinir qualquer comportamento de que não goste. Tal recurso permite que você adapte seu software, quando seus requisitos mudarem. A herança permite que você agrupe classes relacionadas. A POO permite que você agrupe e classifique suas classes.

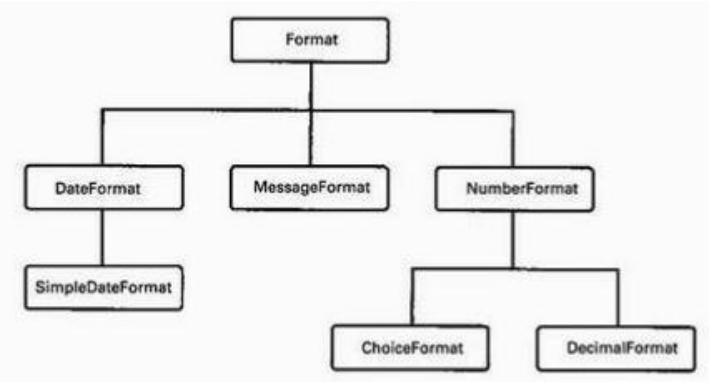
“É UM” (herança) descreve o relacionamento em que uma classe é considerada do mesmo tipo de outra.

“TEM UM “ (composição) Descreve o relacionamento em que uma classe contém uma instância de outra classe.

Composição

Significa que uma classe é implementada usando-se variáveis internas (chamadas variáveis membro), que contém instâncias de outras classes.

Uma hierarquia de herança é um mapeamento do tipo árvore de relacionamentos que se formam entre classes como resultados da herança.



Classe filha (subclasse)

É a classe que está herdando.

Classe progenitora (mãe) (superclasse)

É a classe da qual a filha herda diretamente.

Herança

É um mecanismo que permite estabelecer relacionamento ‘é um’ entre classes. Esse relacionamento permite que uma subclasse herda os atributos e comportamentos de sua superclasse. Quando uma filha herdar de uma progenitora, a filha obterá todos os atributos e comportamentos que a progenitora possa ter herdado de outra classe.
Uma classe pode ter apenas uma progenitora física.
Algumas linguagens

- **herança múltipla:** permitem que uma classe tenha mais de uma progenitora

Mecânica da herança

Todos os métodos atributos disponíveis na interface da progenitora aparecerão na interface da filha. Uma classe construída de herança pode ter três tipos importantes de métodos e atributos:

- **Sobreposto:** Herda o método ou atributo da progenitora. Mas fornece uma nova definição
- **Novo:** Adiciona um método ou atributo completamente novo.
- **Recursivo:** Herda um método ou atributo da progenitora.

Tipos de herança

Existem 3 maneiras principais de usar herança:
1° Reutilização de implementação
2° Diferença

3° Substituição de tipo

Herança para implementação

Herança possibilita que uma nova classe reutilize implementação de outras classes. A herança torna o código automaticamente disponível. Como parte da nova classe. Como mágica, sua nova classe nasce com funcionalidade.

Quando programa com herança de implementação, você está preso à implementação que herda. Uma classe que herda pode sobrepor esses métodos protegidos para alterar a implementação. A sobreposição pode diminuir o impacto da herança de uma implementação mal feita ou inadequada.

Herança para diferença

Permite que você programe especificando apenas como uma classe filha difere de uma classe progenitora.

Significa herdar uma classe e adicionar apenas o código que torne a nova classe diferente da classe herdada.

Permite que você adicione apenas o código necessário para descrever a diferença entre a classe progenitora e a classe filha. Permite que você programe através de incrementos.
Quando programa pela diferença, você escreve um código mais correto em um tempo curto. Assim como a herança de implementação, você pode fazer essas alterações incrementais sem alterar o código existente.

Através da herança, existem duas maneiras de programar pela diferença:

- **Adicionar novos comportamentos e atributos**
- **Redefinindo comportamentos e atributos antigos**

Cada caso é conhecido como especialização.

Especialização

É o processo de uma classe filha ser projetada em termos de como ela é diferente de sua progenitora. A definição de classe da filha incluirá apenas os elementos que a tornam diferente de sua progenitora.

Não permite que você remova da filha comportamentos e atributos herdados. Uma classe não obtém herança seletiva.

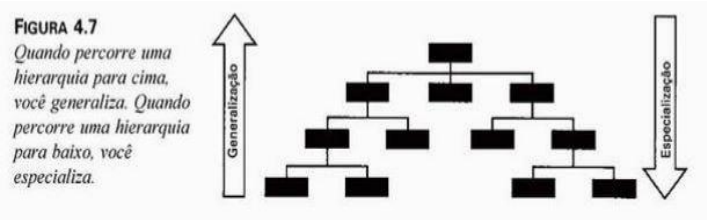


FIGURA 4.7
Quando percorre uma hierarquia para cima, você generaliza. Quando percorre uma hierarquia para baixo, você especializa.

Toda classe que aparece depois dela a hierarquia de classes é uma descendente da classe herdada.

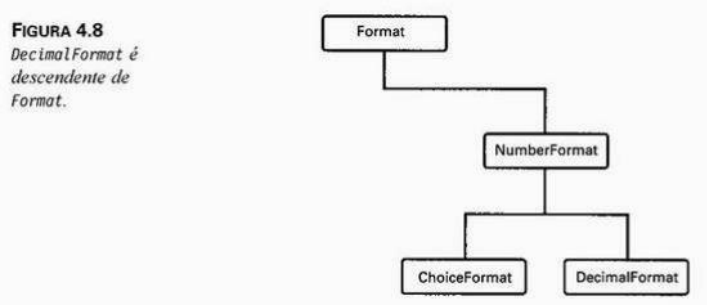


FIGURA 4.8
DecimalFormat é descendente de Format.

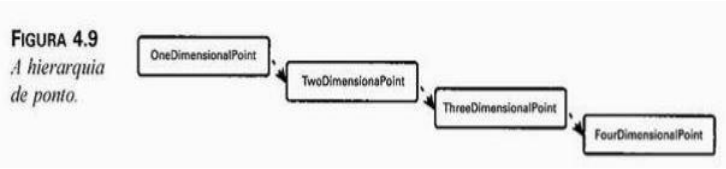


FIGURA 4.9
A hierarquia de ponto.

A classe raiz (classe base) é a superior da hierarquia de herança.

Classe folha

É uma classe sem filhas.

Herança múltipla

Permite que um objeto herde diretamente de mais de uma classe. É um aspecto controverso da POO. A herança múltipla pode ser valiosa, se usada cuidadosamente e corretamente

Herança para substituição de tipo

O tipo final da herança. Permite que você descreva relacionamentos com capacidade de substituição.
Uma filha quando herda de sua progenitora, você diz que a filha ‘ é uma ‘ progenitora.

Subtipo

É um tipo que estende outro tipo através de herança.

A capacidade de substituição aumenta sua oportunidade de realização Permite que você escreva código genérico em vez de várias instruções.

Dicas para herança eficaz

- Use para reutilização de interface e para definir relacionamentos de substituição
- Estender uma implementação se passar no teste do ‘É UM’
- Prefira composição em vez de herança para reutilização de implementação simples
- Sempre use a regra ‘é um’
- Mantenha suas hierarquias de classe relativamente rasas
- Projete e remova características comuns das classes base abstrata (As classes base abstrata permite que você defina um método sem fornecer implementação.)
- Use interfaces bem definidas entre a progenitora e a filha
- Se você adicionar métodos especificamente para uso por subclasses, certifique-se de torná-los protegidos.
- Evite abrir a implementação interna de seu objeto para subclasses
- Nunca se esqueça de que a substituição é o objetivo número 1
- Programe pela diferença para manter o código fácil de gerenciar.
- Prefira à composição à herança para reutilização de implementação

Laboratório : usando classes abstratas para herança planejada

A boa prática diz que, ao ver código comum, você o coloca em uma classe base, planeja para que outras classes herdem dela.
As subclasses se especializam em relação à classe base e fornecem o que está falando.

Método abstrato

É um método declarado, mas não implementado. somente classes abstratas podem ter métodos abstratos.

Declarando métodos abstratos, você obriga suas subclasses a se especializarem em relação à classe base, fornecendo implementação para os métodos abstratos.

O uso de classes abstratas define o contrato que as subclasses devem atender para utilizar a classe base.

Polimorfismo

Um único nome pode representar um código diferente. Permite que um único nome expresse muitos comportamentos diferentes.

É o distúrbio das múltiplas personalidades do mundo do software, pois um único nome pode expressar muitos comportamentos diferentes.

Variável polimórfica

Pode conter muitos tipos diferentes. Em uma linguagem tipada, as variáveis polimórficas estão restritas a conter valores específicos. Em uma linguagem dinamicamente tipada, uma variável polimórfica pode conter qualquer valor.

Linguagem tipada

É uma linguagem de programação que usa variáveis com tipos específicos.
Dinamicamente tipada (fracamente tipada) durante a execução do programa podem alterar o tipo de dado contido em uma variável.

4 formas de polimorfismo

1° Polimorfismo de inclusão
2° Polimorfismo paramétrico
3° Sobreposição
4° Sobrecarga
PARAIN2S

Polimorfismo de inclusão (puro)

Permite que você trate objetos relacionados genericamente. É útil porque diminui a quantidade de código que precisa ser escrito. Permite que um objeto expresse muitos comportamentos diferentes, em tempo de execução

Polimorfismo paramétrico

Permite a criação de métodos e tipos genéricos. Os métodos e tipos genéricos permitem que você codifique algo uma vez e faça isso trabalhar com muitos tipos diferentes de argumentos. Permite que

um objeto ou método opere com vários tipos de parâmetros diferentes.

Polimorfismo de sobreposição

Permite que você sobreponha um método e saiba que o polimorfismo garantirá que o método correto sempre será executado.
Os métodos abstratos (adiados), retarda a definição para classes descendentes,

Polimorfismo de sobrecarga (AD-HOC)

Permite que você use o mesmo nome de um método para muitos métodos diferentes. Cada método difere apenas do número e no tipo de seus parâmetros. Permite que você declare o mesmo método várias vezes. Cada declaração difere simplesmente no número e no tipo de argumento.

Conversão

Conversão e sobrecarga frequentemente andam lado a lado. A conversão também pode fazer com que um método pareça como se fosse polimórfico. A conversão ocorre quando um argumento de um tipo é convertido para o tipo esperado, internamente.

Para obter um polimorfismo eficaz, siga estas dicas:

- Siga as dicas do encapsulamento e da herança eficazes
- Sempre programe para interface e não para a implementação
- Pense e programe genericamente
- Defina a base do polimorfismo estabelecendo e usando relacionamentos com capacidade de substituição. Garantirão que você possa adicionar novos subtipos em seu programa e o código correto será executado
- Separar completamente a interface e a implementação, Separa os dois permite uma capacidade de substituição mais flexível, obtendo-se assim mais oportunidade de polimorfismo
- Use classes abstratas para separar a interface da implementação. Todas as classes não-folha devem ser abstratas; programe apenas para essas classes abstratas.

Aprendendo a aplicar a OO

Introdução à UML

É uma linguagem de modelagem padrão. consiste em várias notações gráficas que você pode usar para descrever a arquitetura interna de seu software. Os programadores, arquitetos e analistas de software usam linguagens de modelagem para descrever graficamente o projeto do software.

UML

É uma notação gráfica para descrever projeto de software. Incluí várias regras para distinguir entre desenhos corretos e incorretos. São essas regras que tornam à UML, uma linguagem de modelagem e não apenas um punhado de símbolos para desenho. Uma linguagem de modelagem ilustra o projeto que você criará enquanto segue uma metodologia. A UML não é a única linguagem de modelagem, ela é um padrão aceito. Ao modelar software, é importante fazer isso em uma linguagem comum, desse modo, outros desenvolvedores podem rápida e facilmente entender seus diagramas de projeto. A UML fornece um vocabulário comum que os desenvolvedores podem usar para transmitir seus projetos.

Metodologia (processo) X UML

- **Metodologia (processo):** descreve como projetar software
- **UML:** Ilustra o projeto

Modelando suas classes

Uma notação gráfica o isola dos detalhes, para que você possa exprimir-se de entender a estruturas de alto nível de um programa.

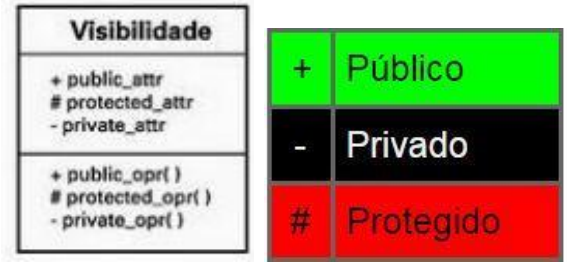
Notações básicas de classe

A UML ajuda a transmitir seu projeto fornecendo um rico conjunto de notações para descrever suas classes. Usando essa notação, outros podem ver facilmente as principais classes que compõem o projeto de seu programa. A UML permite definir as classes, assim como descrever os relacionamentos de alto nível entre as classes.

A UML faz diferença entre operações e métodos.

- **Operação:** É um serviço que você pode solicitar de qualquer objeto de uma classe.
- **Método:** É uma implementação específica da operação.

Caracteres de permissões

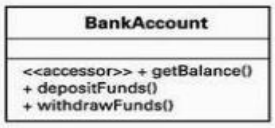


Notações avançada de classe

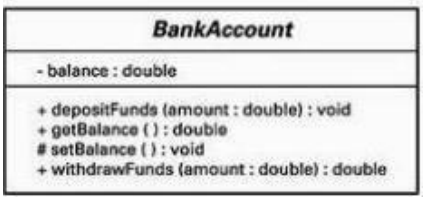
O uso correto dessa notação o ajuda a criar modelos mais descritivos. A UML permite que você amplie o vocabulário da própria linguagem, através do uso de estereótipos.

Estereótipo

É um elemento da UML que permite que você amplie o vocabulário da própria linguagem UML. Um estereótipo consiste em uma palavra ou frase incluída entre sinais de menor e maior duplos (<<>>). Você pode colocar um estereótipo acima ou ao lado de um elemento existente.



A UML fornece uma notação para transmitir que uma classe é abstrata: o nome da classe abstrata é escrito em *itálico.*



Modelando um relacionamento de classe

As classes têm relacionamentos complexos entre si. Esses relacionamentos descrevem como as classes interagem umas com as outras.

Relacionamento

Descreve como as classes interagem entre si, é uma conexão entre dois ou mais elementos da notação.

3 tipos de alto nível de relacionamentos de objeto:

DEPASSGEN

- **Dependência**
- **Associação**
- **Generalização**

A UML simplesmente fornece um mecanismo e vocabulário comum para descrever os relacionamentos.

Especificação = interface / comportamento

Dependência

Um objeto é dependente da especificação de outro objeto. Se a especificação mudar, você precisará atualizar o objeto dependente.

Quando você deve modelar dependência ?

Quando quer mostrar que um objeto usa o outro.

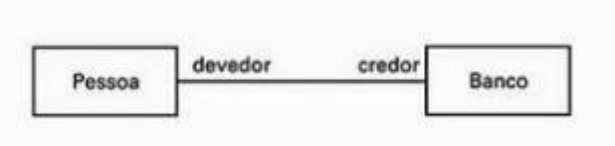
Associação

Indica que um objeto contém outro objeto. (um objeto está conectado a outro.

Como os objetos estão conectados, você pode passar um objeto para o outro. Na notação UML, toda associação tem um nome.

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

Cada objeto em uma associação tem um papel.



Multiplicidade

Indica quantos objetos podem tomar parte na instância de uma associação.

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

Único número (6)	Somente aquele número
Asterisco (*)	qualquer número de objetos podem participar da associação
Lista 1..4	intervalo (indica que de 1 a 4 objetos podem participar da associação.

Quando você deve modelar associações ?

Quando um objeto contiver outro objeto. o relacionamento " TEM UM ". Quando um objeto usa outro. Uma associação permite que você modele quem faz o que um relacionamento.

A UML também define dois tipos de associação

- Agregação
- Composição

Esses dois subtipos de associação o ajudam a refinar mais seus modelos.

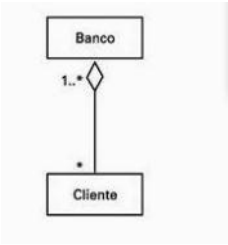
Agregação

É um tipo especial de associação que modela o relacionamento “tem um” de relacionamentos todo/parte entre pares.
Uma agregação modela um relacionamento entre pares. Significa que um objeto contém outro. Pares significa que um objeto não é mais importante do que o outro.

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

Agregação significa que os objetos podem existir independentemente uns dos outros. Nenhum objeto é mais importante do que o outro no relacionamento.

EX: Banco e cliente, um banco pode existir independentemente de um cliente, e assim vice versa.



(Losango aberto simboliza agregação)

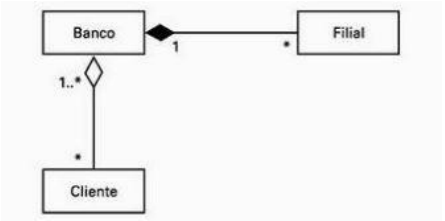
Quando você deve modelar a agregação ?
Quando o objetivo de seu modelo for descrever a estrutura de um relacionamento de pares. Uma agregação mostra explicitamente o relacionamento estrutural todo/parte

Composição

Os objetos não são independentes uns dos outros “TEM UM” (um banco tem uma filial)
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 filias também fecharão.
Entretanto, o inverso não é necessariamente verdade, se uma filial fechar, o banco poderá permanecer funcionando.

(Losango fechado simboliza a composição)

Um objeto pode participar de um relacionamento de composição e agregação ao mesmo tempo



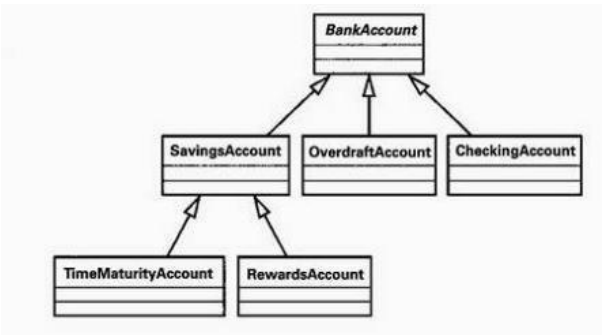
Quando você deve modelar uma composição ?
Quando o objetivo de seu modelo for descrever a estrutura de um relacionamento. A composição não modela relacionamento todo/parte de pares. A parte depende do todo. Isso significa que quando o objeto banco for destruído, os objetos filial também serão destruídos.

Agregação e composição são simplesmente refinamentos ou subtipos da associação. Isso significa que você pode modelar agregação e composição como uma associação simples.

Generalização

É um relacionamento entre o geral e o específico. É a herança.
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 ‘’. 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.



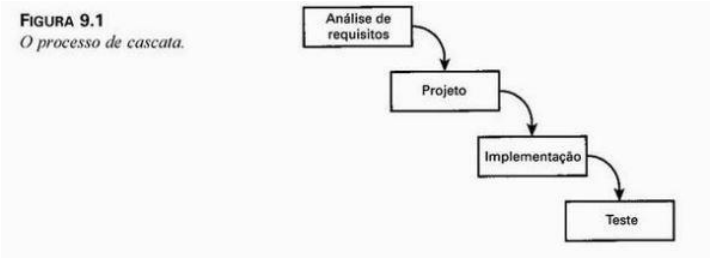
Introdução a análise orientada a objetos

O processo de desenvolvimento de software

Uma equipe de desenvolvimento de software precisa de uma estratégia unificada para desenvolver software. Uma metodologia frequentemente conterá uma linguagem de modelagem (como a UML) e um processo.

Processo de Software

Mostra os vários estágios do desenvolvimento de software.



(O processo de cascata)

Quando segue o processo de cascata, uma vez que complete um estágio, não há volta. O processo de cascata tenta evitar alterações, proibindo mudar quando um estágio está concluído. Tal processo rígido frequentemente resulta em software que não é o que você ou seu cliente quer.
O processo de cascata não pode enfrentar a realidade do moderno desenvolvimento de software - requisitos que mudam constantemente.

O processo iterativo

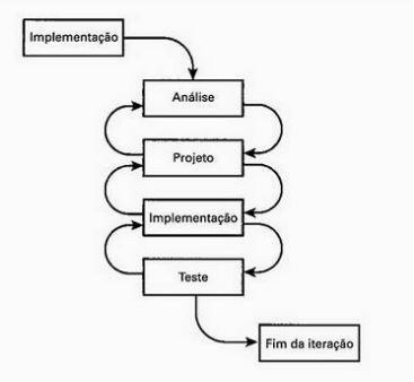
É uma estratégia iterativa e incremental para desenvolvimento de software.
Permite alterações em qualquer ponto do processo de desenvolvimento. Permite alterações adotando uma estratégia **ITERATIVA e INCREMENTAL** para o desenvolvimento de software.

Uma estratégia iterativa

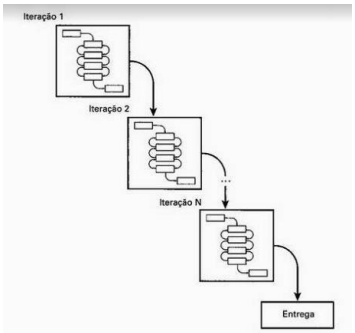
Permite que você continuamente volte e refine cada estágio do desenvolvimento. Pode voltar e fazer um projeto adicional e uma nova análise. É esse refinamento contínuo que torna o processo iterativo.

Estratégia incremental

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



(Uma iteração)



(O processo iterativo)

O progresso constante fornece a você retorno constante. Você pode usar esse retorno como o modo de garantir se está no caminho certo. Como você recebe retorno constante, é mais provável que identifique o problema mais cedo e com isso será mais fácil refazer uma iteração ou duas para corrigi-lo.

Uma metodologia de alto nível

Seleciona e escolhe as técnicas que se mostram eficazes a partir de outras metodologias. A metodologia consiste em um processo iterativo, no qual uma interação tem 4 estágios.
PRIMTAR

- Projeto
- Implementação
- Teste
- Análise de Requesitos

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.

AOO (Análise Orientada a Objetos)

É o processo usado para entender o problema que você está tentando resolver. É um processo que usa

uma estratégia orientada a objetos para ajudá-lo a entender o problema que está tentando resolver. Para projetar uma solução para um problema, A resposta dessa pergunta são os requisitos do sistema. Os requisitos informam a você o que os usuários querem fazer com o sistema e quais tipos de respostas eles esperam receber.
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.

Esses objetos são instâncias de classes derivadas de objetos concretos ou abstratos no domínio do problema que está sob estudo.
Usando o modelo de domínio, você começa a identificar os objetos que pertencem ao seu sistema. Um modelo de domínio corretamente construído pode resolver muitos problemas no mesmo domínio.

Usando casos de estudo para descobrir o uso do sistema.

Requisitos: são recursos ou características que o sistema deve ter para resolver um determinado problema.

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

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.

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. 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 sequência de eventos de cada caso de uso
5 ° Modelar seus casos de uso

Identifique os atores
O primeiro passo é definir os atores que usarão o sistema.

Ator
É tudo que interage com o sistema. (humano, computador, etc.)

(os atores na UML)

Um determinado usuário do sistema pode assumir o papel de muitos atores diferentes. Um ator é um papel. Ex: um usuário poderia entrar no site como convidado e posteriormente se conectar como registrado.

Refine e nomeie os casos de uso
Você desejará procurar oportunidades de dividir ou combinar casos de uso.

Dividindo casos de uso
Cada caso de uso deve executar um objetivo principal. Um caso de uso pode conter outro. Assim, se uma instância de caso de uso exige que outra faça seu trabalho, ela pode usá-la. Um caso de uso também pode estender o comportamento de outro caso de uso.

Combinando casos de uso
Você não quer casos de uso redundantes. Deverá combinar as variantes em um único caso de uso.

Variante
É uma versão especializada de outro caso de uso mais geral.

Casos de uso resultantes
Após concluir o refinamento de seus casos de uso, você deve nomear cada caso de uso.

Defina a sequência de eventos de cada caso de uso

A sequência de passos que um usuário usa para completar um caso de uso é conhecida como cenário. Um caso de uso é constituído de vários cenários

Cenário

É uma sequência ou fluxo de eventos entre o usuário e o sistema.
Você deve primeiro especificar os cenários de cada caso de uso

Condições prévias

São aquelas condições que devem ser feitas para que um caso de uso comece. **Condições posteriores** são os resultados de um caso de uso. Após completar o caso de uso, o sistema conterá um pedido para usuário.

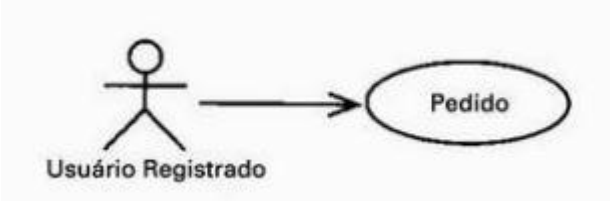
Diagrama de casos de uso

É uma maneira de documentar e transmitir projeto de classe, maneiras formas de capturar seus casos de uso. De especial interesse são:
CA-IN-A

- **Diagramas de caso de uso**
- **Diagrama de interação**
- **Diagrama de atividade**

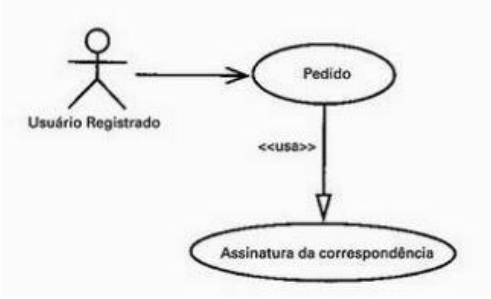
Os diagramas de caso de uso modelam os relacionamentos entre casos de uso e os relacionamentos entre casos de uso e atores.

Coloque um ator e um caso de uso juntos no mesmo diagrama e você terá um diagrama de caso de uso.



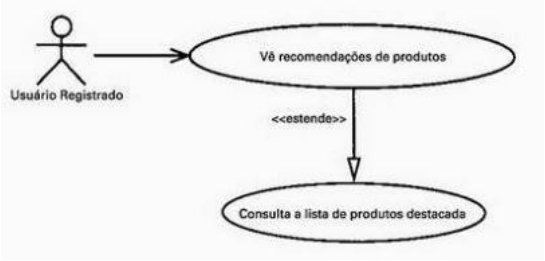
(O caso de uso pedido)

O diagrama também pode mostrar os relacionamentos existentes entre os próprios casos de uso. Um caso de uso pode conter e usar outro caso de uso.



(Um relacionamento usa)

Você vê que o caso de uso Registro usa o caso de uso Assinatura da correspondência.



(Relacionamento estende)

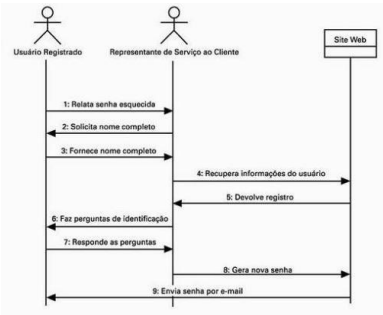
Enquanto você estuda seus casos de uso, pode encontrar meios de extrair características comuns e colocá-las em casos de uso abstrato.

Diagramas de Interação

Os diagramas de caso de uso ajudam a modelar os relacionamentos entre caso de uso. Os diagramas de interação ajudam a capturar as interações entre vários atores.

Diagrama de sequência

Modela as interações entre o usuário, o representante, com o passar do tempo. Você deve usar diagramas de sequência quando quiser chamar a atenção para sequência de eventos de um caso de uso, com o passar do tempo.



Quando um ator deixa um caso de uso, você pode dizer que seu tempo de vida terminou.

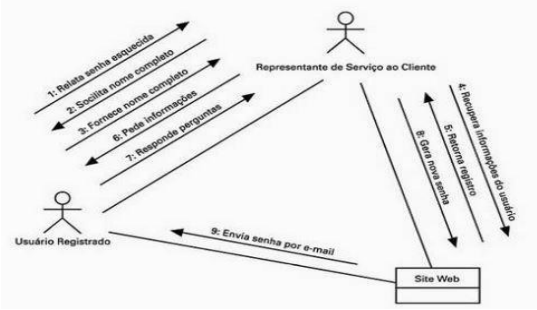
Linha de vida

É uma linha tracejada que sai de uma caixa em um diagrama de sequência. 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 para o sistema. O tempo corre de cima para baixo em um diagrama de sequência. Assim, subindo na linha da vida, você pode reproduzir a sequência de eventos de trás para frente.

Diagrama de colaboração

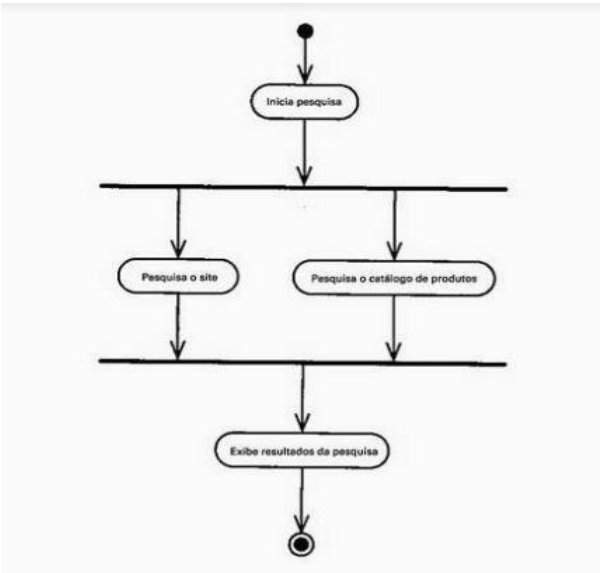
Você usa o diagrama de sequências se tiver a intenção de chamar a atenção para a sequência de eventos com o passar do tempo. Se você quiser modelar os relacionamentos entre os atores e o sistema, então deve criar um diagrama de colaboração.



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.

Diagrama de atividade

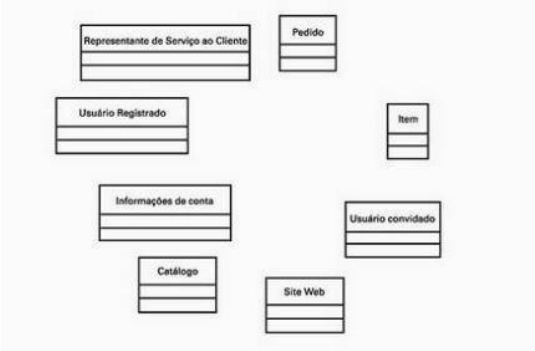
Os diagramas de interação modelam as ações sequenciais. Eles não podem modelar processos que podem ser executados em paralelo. Os diagramas de atividade o ajudam modelar processos que podem ser executados em paralelo.



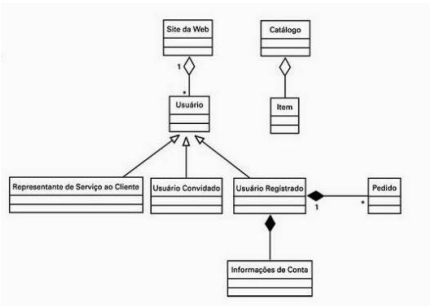
Uma elipse representa cada estado do processo. A barra preta grossa representa um ponto onde os processos devem ser sincronizados - ou reunidos antes que o fluxo de execução seja retomado.

Construindo o modelo de domínio

Os casos de uso também o ajudam a capturar o vocabulário do sistema. Esse vocabulário constrói o domínio do problema. O vocabulário do domínio identifica os principais objetos do sistema. O modelo de domínio lista os objetos que você precisará para modelar corretamente o sistema



(os objetos do domínio)



(relacionamentos dos objetos do domínio)

O modelo de domínio é importante por vários motivos. ele modela seu problema independentemente de quaisquer preocupações com implementação. Ele modela o sistema em um nível conceitual que proporciona uma flexibilidade para resolver muitos problemas diferentes dentro do domínio.

Constrói a base do modelo de objeto que, finalmente se tornará seu sistema. A implementação final pode adicionar novas classes e remover outras. O domínio, fornece algo para que você comece e construa seu projeto - um esqueleto.

Um modelo de domínio bem definido estabelece claramente um vocabulário comum para seu problema. Encontrando-se um vocabulário comum, todos os envolvidos no projeto poderão encará-lo a partir de uma posição e um entendimento iguais

Introdução ao POO (Projeto Orientado a Objetos)

Ajuda a pegar o domínio que você encontrou na AOO e a projetar uma solução. AOO o ajudou a descobrir muitos dos objetos de domínio do problema, O POO o ajuda a descobrir e projetar os objetos que aparecerão na solução específica do problema.

POO

É o processo de construir o modelo de objeto de uma solução. É o processo de dividir uma solução em vários objetos constituintes.

Modelo de objeto

É o projeto dos objetos que aparecem na solução de um problema. (descreverá as várias responsabilidades, relacionamentos e estrutura do objeto.)

Modelo final de objeto: pode conter muitos objetos não encontrados no domínio

Como você aplica POO (Projeto Orientado a Objeto)?

É um processo iterativo que identifica os objetos e suas responsabilidades em seu sistema, e como esses objetos se relacionam.

Passos para construir seu modelo de objeto

1. Gerará uma lista inicial de objetos
2. Refinará as responsabilidades de seus objetos
3. Desenvolverá os pontos de interação
4. Detalhará os relacionamentos entre objetos
5. Construirá seu modelo

Objeto

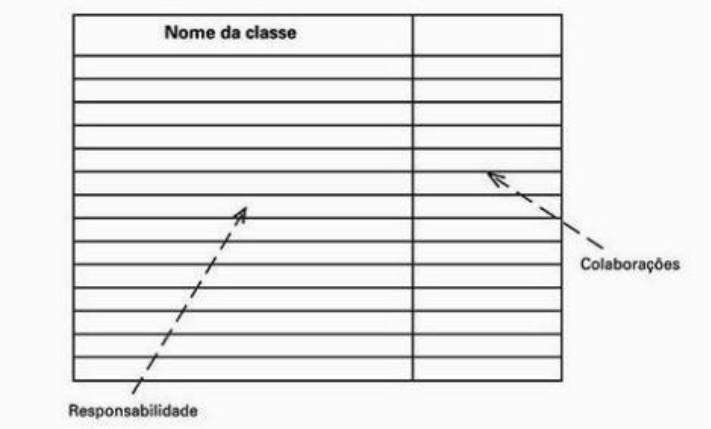
Delega trabalho para colaboradores.

Colaboração

É o relacionamento onde os objetos se interagem para realizar o mesmo propósito.

O que são cartões CRC ? (Classe Responsabilidade Colaboração)

Uma ficha de arquivo 4x6 com linhas. Os cartões ajudam a definir o propósito de um objeto, chamando a atenção para as responsabilidades do objeto. Quando usa cartões CRC, você simplesmente cria um cartão para cada classe. Os cartões são de baixa tecnologia. Se você achar que o cartão não é grande o suficiente, são boas as chances que é preciso dividir as classes.
Vantagem: Você não fica preso a um computador.



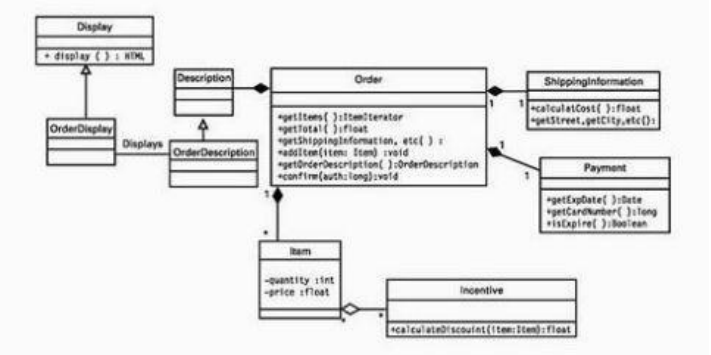
Em vez de ter de modelar vários projetos alternativos, você pode apenas movimentar seus cartões.
Ponto de interação: é qualquer lugar onde um objeto use outro.

Interface

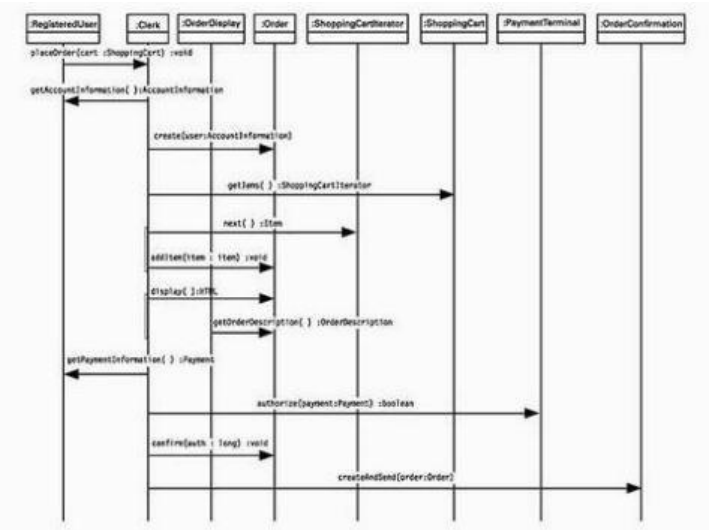
Você precisa de uma interface bem definida, onde um objeto usa outro.

Agentes

Faz a mediação entre dois ou mais objetos para atingir algum objetivo.



(modelo pedido)



(Diagrama de sequência de pedido)

Reutilizando projetos através de padrões avançados

Padrão de projeto

É um conceito de projeto reutilizável.
Quando você reutiliza um padrão de projeto, está usando um projeto que outros usaram com êxito, muitas vezes anteriormente.

4 elementos dos padrões de projeto:

SO-CON-PRO-NOPADRÃO

- ➢ **nome do padrão**
- ➢ **problema**
- ➢ **solução**
- ➢ **consequência**

Nome do padrão

Identifica exclusivamente cada padrão de projeto. 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.

Problema

Você pode usar a descrição do problema para determinar se o padrão se aplica ao problema específico.

Solução

Descreve como o padrão de projeto resolve e identifica os objetos arquitetonicamente.

Consequências

Ajuda os outros a entender as escolhas que você fez e a determinar se o projeto pode ajudar a resolver seus próprios problemas.

Os padrões de são:

- Projetos reutilizáveis que provaram funcionar no passado
- Soluções abstratas para um problema de projeto geral
- Soluções para problemas recorrentes
- Um modo de construir um vocabulário de projeto
- Um registro público da experiência do projeto
- Uma solução para um problema

Os padrões não são:

- Uma solução para um problema específico
- A resposta mágica para todos os seus problemas
- Uma muleta, você mesmo ainda precisa fazer seu projeto funcionar
- Classes concretas, Bibliotecas, soluções prontas

O padrão Adapter (empacotador)

Apresenta uma solução alternativa que resolve o problema de incompatibilidade, transformando a interface incompatível naquela que você precisa.
(é como se fosse um adaptador de tomada.)

Adaptador

É um projeto que transforma a interface de outro projeto. É útil quando você quer usar um objeto que tem uma interface incompatível.

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

O padrão PROXY

Trata de ocasiões em que você desejará controlar o acesso entre seus objetos de forma transparente.

FIGURA 11.3
Um relacionamento publicação/assinatura.

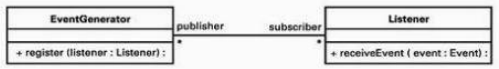
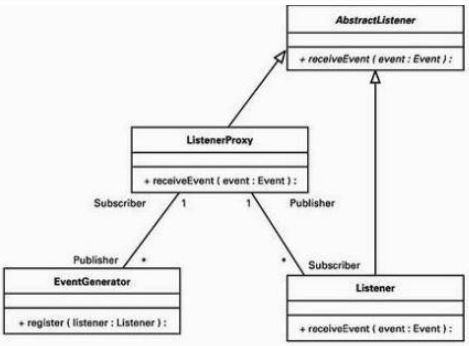


FIGURA 11.4
Uma solução proxy.



Um **PROXY** é um substituto ou lugar reservado que intermedia o acesso ao objeto de interesse real. O **PROXY** cuida de todos os detalhes da comunicação com o objeto (ou objetos real).

PROXY: intermedia o acesso ao objeto
Você pode considerar um substituto como um modo de enganar seus objetos

Substituto: Permite que você coloque responsabilidades nele, sem ter de incorporar essas responsabilidades no usuário do substituto. Seu substituto pode executar todas as responsabilidades sem que os outros objetos saibam o que você está fazendo.

As responsabilidades que você pode colocar no substituto são infinitas.

Quando usar o padrão PROXY

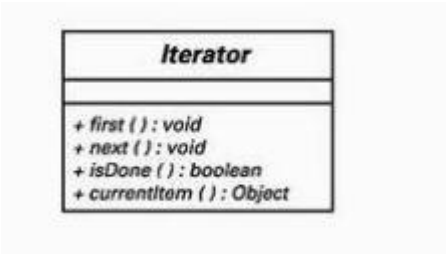
- Quando você quiser adiar uma operação dispendiosa
- Quiser proteger de forma transparente o modo como um objeto é usado
- O objeto existe remotamente, Um substituto pode fazer com que um recurso distribuído pareça como se fosse um recurso local.
- Quando você executar ações adicionais de forma transparente, ao usar um objeto.

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

Resolve muitos dos problemas de forte acoplamento e confusão de responsabilidade, colocando a lógica do laço ou interação em seu próprio objeto.



A interface **ITERATOR** fornece uma interface genérica para interagir sobre uma coleção. Um **ITERATOR** dará acesso ao conteúdo do objeto

3 Vantagens de usar um **ITERATOR** para percorrer uma coleção:

- Não será vinculado a uma coleção específica
- Pode retornar seus elementos em qualquer ordem que achar conveniente
- Torna simples mudar a coleção subjacente, quando isso for necessário.

Quando usar o padrão ITERATOR

- Quando quiser ocultar a implementação de uma coleção
- Quando quiser fornecer diferentes tipos de laços sobre uma coleção
- Para manter a interface de uma coleção simples
- Definir uma classe de coleção base que retorne um **ITERATOR**
- Para fornecer acesso otimizado para às coleções

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

Padrões avançados de projeto

Considerando 3 padrões importantes:

- **Abstract factory**
- **Singleton**
- **Typsafe Enum**

Abstract factory

Permitem que você conectem 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.
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** (você pode isolar toda a criação de objeto em um único local.

Delegação: faz chamada a um método de outra classe
Composição: Um método possui outro.

O padrão **abstract factory** usa herança e capacidade de conexão. A classe base e factory define todos os métodos de criação de objeto e cada subclasse **factory** define quais objetos cria, sobrepondo os métodos.

Analisador XML

Pega um documento e o transforma em uma representação de objeto.

Um método **factory** nada mais é do que um método que cria objetos. Pode aparecer em uma classe normal ou em uma **Factory** abstrata. Em qualquer caso, ele cria objetos, ocultando assim a classe real do objeto criado.

Quando usar o padrão Abstract factory:

- Ocultar o modo de como um objeto é criado
- Ocultar a classe atual do objeto criado
- Conjunto de objetos usados juntos
- Diferentes versões de uma implementação de classe

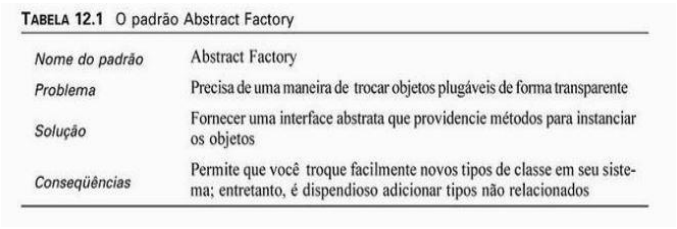


TABELA 12.1 O padrão Abstract Factory

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 (permite apenas uma pessoa utilizar)

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.

Quando usar o padrão Singleton:

Quando você quiser restringir uma classe a ter apenas uma instância.

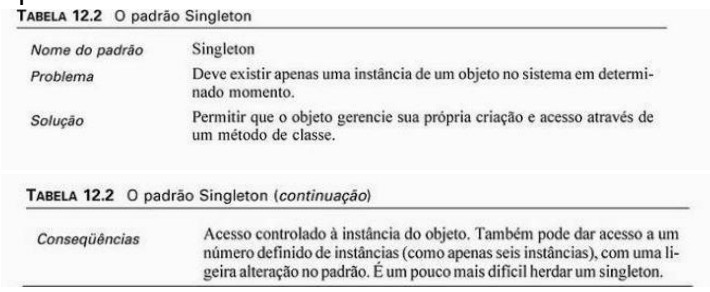


TABELA 12.2 O padrão Singleton

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

Cria o tipo certo “novo tipo”

Quando usar o Padrão Typesafe Enum

- 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

Construindo software confiável através de testes

O teste é a última etapa de interação.
(análise, projeto, implementação e teste)

Testes funcionais ou testes de aceitação

Para testar um sistema, você precisa escrever e executar **CASOS DE TESTE.** Cada caso de teste testará um aspecto específico do sistema.

CASO DE TESTE: É um bloco de construção básico do processo de teste.

O teste executará um caminho específico através do sistema **(CAIXA BRANCA)** ou testará algum comportamento definido **(CAIXA PRETA)**
2 maneiras de basear seus casos de teste:
CAIXA PRETA E CAIXA BRANCA

CAIXA PRETA

testa se o sistema funciona conforme o esperado.
Testa se a saída ou comportamento resulta conforme definido pela especificação da classe ou do sistema.

CAIXA BRANCA

é baseado na implementação de um método. Seu objetivo é garantir que cada desvio do código seja executado. Você projeta seus testes de modo a exercitar cada desvio na esperança de eliminar todos os erros latentes.

4 FORMAS DE TESTE

- **TESTE DE UNIDADE**
- **TESTE DE INTEGRAÇÃO**
- **TESTE DE SISTEMA**
- **TESTE DE REGRESSÃO**

TESTE DE UNIDADE

Examina apenas um recurso por vez. É o dispositivo de teste de nível mais baixo. Envia uma mensagem para um objeto e depois verifica se ele recebe o resultado esperado do objeto. Você deve basear os testes de unidade no teste de caixa preta e caixa branca. Você deve escrever o caso de teste antes de escrever a classe.

TESTE DE INTEGRAÇÃO

Verifica se 2 ou mais objetos funcionam em conjunto corretamente

Verificam se os objetos que compõem o sistema interagem corretamente O que poderia funcionar isoladamente pode não funcionar quando combinado com outros objetos!
Você deve ter um teste de interação para cada interação importante no sistema.

TESTE DE SISTEMA

Verifica se o sistema inteiro funciona conforme descrito pelos casos de uso. Você verifica se o sistema manipula e se recupera de condições imprevistas.

- **teste de ação aleatória:** tenta executar operações em ordem aleatória,

- **teste de banco de dados vazio:** garante que o sistema pode falhar normalmente, caso exista um problema maior no banco de dados.
- **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.

Estes testes garantem que o sistema satisfaça quaisquer requisitos de desempenho e possa funcionar sob as cargas esperadas. Para sair de uma interação, o sistema deve passar nesses testes com êxito.

TESTE DE REGRESSÃO

Examinam as alterações nas partes do sistema que já foram validadas.

É o processo de repetição dos testes de unidade, integração e de sistema após as alterações serem feitas.

Combinando desenvolvimento e teste

Para testar enquanto desenvolve, você precisa escrever testes de unidade para cada classe que criar.

Por que você deve escrever testes de unidade

O teste de unidade é sua primeira linha de defesa contra erros. É muito mais fácil de manipular que tentar rastrear um erro durante o teste de integração ou sistema. Uma classe está pronta quando todos os testes passam!

Escrevendo testes de unidade

ESTRUTURA

é um modelo de domínio reutilizável. Contém todas as classes comuns a um domínio inteiro de problemas e serve como a base para um aplicativo específico no domínio.

A classe de uma estrutura define o projeto geral de um aplicativo.

JUNIT

Fornece classes para escrever testes de unidade
Fornece a você várias opções para a execução de seus casos de teste. Essas opções caem em duas categorias:

- **Estáticas**
- **Dinâmicas**

Classe anônima

É uma classe que não têm nome, são definidas ao serem instanciadas. elas permitem que você sobreponha um método ao instanciar um objeto.

Acessório de teste

Define o conjunto de objetos sobre os quais um teste operará. Permitem que você compartilhe o mesmo acessório dentre um conjunto inteiro de casos de teste, sem ter que duplicar o código.

Objeto falsificado (simuladores)

É um substituto simplista de um objeto real. Ele é chamado de objeto falsificado porque o objeto foi falsificado para propósito de teste. O substituto não aparecerá no sistema real, apenas no código de teste.

Escrevendo código excepcional

- Um erro e uma condição de erro não são a mesma coisa.
- ERRO=DEFEITO
- CONDIÇÃO DE ERRO= FALHA PREVISÍVEL

Escrevendo documentação eficaz

- **Código fonte como documentação**

É uma forma de documentação. Quando outras pessoas precisam pegar e manter seu código, é importante que ele seja legível e bem organizado. O código fonte é a forma mais importante da documentação, pois é a única documentação em que você tem de manter.

- **Convenções de codificação**

- Nomes das classes= MAIÚSCULAS
- Os nomes de métodos= minúscula
- Nome contendo várias palavras = JuanHenri
- Os nomes de variável= minúscula
- Constantes= MAIÚSCULAS
- Variáveis normais= minúsculas

Constantes

Podem servir como uma forma de documentação. Use constantes quando você se achar utilizando um valor codificado. Uma constante bem nomeada pode dar uma ideia do objeto de seu código.

Comentários

// configura id

Nomes

Primeira palavra minúscula e a segunda palavra iniciando com letra maiúscula.

testCase ou utilizando hífen Teste-case

Cabeçalhos de métodos e classes

Quando você escrever uma classe ou um método, sempre se certifique de incluir um cabeçalho.
Um cabeçalho de método incluirá uma descrição, uma lista de argumentos, uma descrição do retorno, assim como condições de uma execução e efeitos colaterais

OO e programação da interface com usuário

(UI) fornece a interface entre o usuário e seu sistema.

Como desacoplar a UI usando o padrão Model View Controller

O padrão **MVC** fornece uma estratégia para o projeto de interfaces com o usuário que desacoplam completamente o sistema subjacente da interface com o usuário.

MVC= é apenas uma estratégia para o projeto de interfaces com o usuário orientadas a objetos. Usada popularmente no setor de software.

O padrão MVC desacopla a UI do sistema, dividindo o projeto da UI em 3 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

Gerencia o comportamento básico e o estado do sistema.
Um modelo é apenas um objeto que representa o sistema.
(Um sistema pode ter muitos modelos diferentes.)

O modelo é responsável por fornecer:

- Acesso à funcionalidades básicas do sistema
- Acesso às informações de estado do sistema
- Um sistema de notificação de mudança de estado.

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

O modelo de visualização é a camada tríade MVC que exibe a representação gráfica ou textual do modelo.

A única conexão que um modelo mantém com a UI é através do sistema de notificação de mudança de estado.

Padrão observer

Fornece um projeto para um mecanismo de publicação/assinaturas entre objetos.

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

Exibe informações para o usuário. Obtém informações de exibição do modelo.
(Um único modelo pode ter modos de visualizações diferentes.)

O controlador

É responsável por:

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

O controlador atua como 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.

Um modo de visualização tem apenas um controlador e um controlador tem apenas um modo de visualização.