Casos de Uso

Diagrama de Casos de Uso

Objetivo

O Diagrama de *Casos de Uso* tem o objetivo de auxiliar a comunicação entre os analistas e o cliente.
Um diagrama de Caso de Uso descreve um cenário que mostra as funcionalidades do sistema do ponto de vista do usuário.
O cliente deve ver no diagrama de Casos de Uso as principais funcionalidades de seu sistema.

Notação

O diagrama de Caso de Uso é representado por:

- atores;
- casos de uso;
- relacionamentos entre estes elementos.

Estes relacionamentos podem ser:

- associações entre atores e casos de uso;
- generalizações entre os atores;
- generalizações, *extends* e *includes* entre os casos de uso.

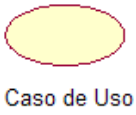
casos de uso podem opcionalmente estar envolvidos por um retângulo que representa os limites do sistema.

Em maiores detalhes:

- ***Atores***

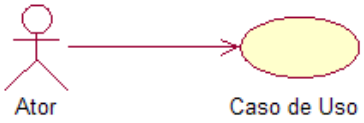
Um ator é representado por um boneco e um rótulo com o nome do ator. Um ator é um usuário do sistema, que pode ser um usuário humano ou um outro sistema computacional.

- ***Caso de uso***



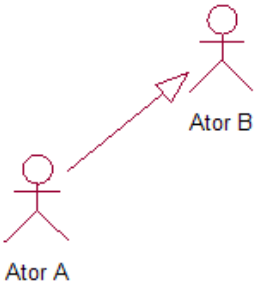
Um *caso de uso* é representado por uma elipse e um rótulo com o nome do *caso de uso*. Um *caso de uso* define uma grande função do sistema. A implicação é que uma função pode ser estruturada em outras funções e, portanto, um *caso de uso* pode ser estruturado.

- ***Relacionamentos***
  - Ajudam a descrever *casos de uso*
  - Entre um ator e um *caso de uso*
    - Associação



Define uma funcionalidade do sistema do ponto de vista do usuário.

  - Entre atores
    - Generalização



- Os *casos de uso* de B são também *casos de uso* de A
- A tem seus próprios *casos de uso*

- Entre *casos de uso*
  - *Include*

    Um relacionamento *include* de um *caso de uso* A para um *caso de uso* B indica que B é essencial para o comportamento de A. Pode ser dito também que B *is_part_of* A.

  - *Extend*

    Um relacionamento *extend* de um *caso de uso* B para um *caso de uso* A indica que o *caso de uso* B pode ser acrescentado para descrever o comportamento de A (não é essencial). A extensão é inserida em um ponto de extensão do *caso de uso* A.

    Ponto de extensão em um *caso de uso* é uma indicação de que outros *casos de uso* poderão ser adicionados a ele. Quando o caso de uso for invocado, ele verificará se suas extensões devem ou não serem invocadas.

    Você entendeu?! Provavelmente, não. É que *extend* é unanimemente considerado um conceito obscuro.

    Vamos a novas explicações.

    Quando se especifica B *extends* A, a semântica é:

    - Dois *casos de uso* são definidos: **A** e **A *extended by* B**;
    - B é uma variação de A. Contém eventos adicionais, para certas condições;
    - Tem que ser especificado onde B é inserido em A.

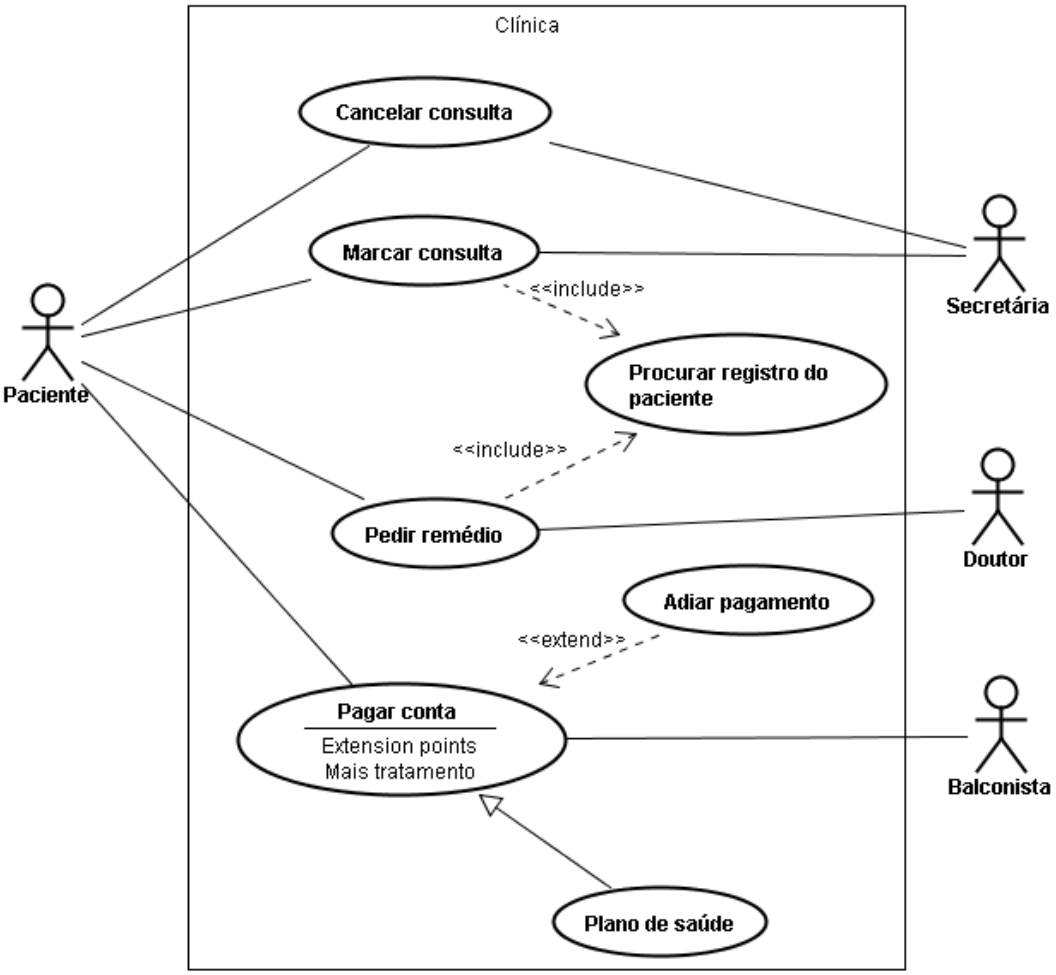
  - Generalização ou Especialização (é_um)

    caso de uso B é_um *caso de uso* A (A é uma generalização de B, ou B é uma especialização de A).

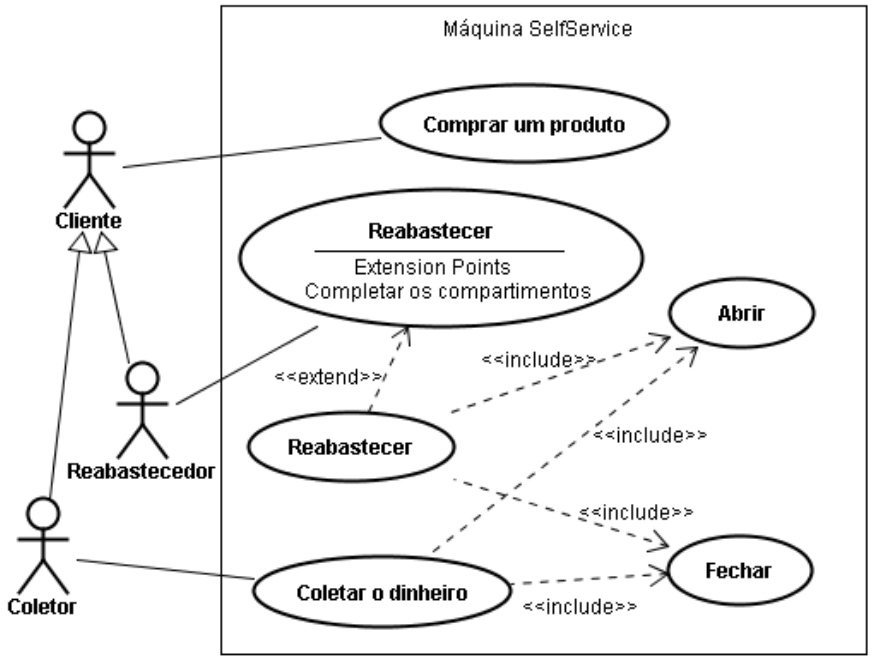
    Um relacionamento entre um caso de uso genérico para um mais específico, que herda todas as características de seu pai.

- ***Sistema***
  - Limites do sistema: representado por um retângulo envolvendo os *casos de uso* que compõem o sistema.
  - Nome do sistema: Localizado dentro do retângulo.

Exemplo 1


Clínica
Cancelar consulta
Marcar consulta
<<include>>
Procurar registro do paciente
<<include>>
Pedir remédio
Adiar pagamento
<<extend>>
Pagar conta
Extension points
Mais tratamento
Plano de saúde
Paciente
Secretária
Doutor
Balconista

Exemplo 2


Máquina SelfService
Comprar um produto
Reabastecer
Extension Points
Completar os compartimentos
Abrir
<<include>>
<<extend>>
Reabastecer
<<include>>
<<include>>
Fechar
Coletar o dinheiro
<<include>>
Cliente
Reabastecedor
Coletor

Caso de Uso – Include, Extend e Generalização

Published by Plínio Ventura on December 28, 2014

Caso de Uso e Programação

Fazer um Caso de Uso, dependendo do ponto de vista, não é algo muito diferente do que programar. É possível fazer um bom trabalho, sob um mesmo ponto de vista, tanto na modelagem quanto na codificação.

Sabia que é possível utilizar uma linguagem de programação Orientada a Objetos e fazer um software com programação **não orientada à objetos**?

Sim, é possível fazer em C# ou Java, por exemplo, um software programado de maneira “quase” estruturada. Muito disso é POG!

Nessa mesma linha de raciocínio, é possível utilizar modelagem de caso de uso em um projeto, mas no fim das contas, ter algo mais próximo de diagramas de fluxo de dados do que de diagramas de Caso de Uso.

Obs.: infelizmente na área de software muitos profissionais **dão pouca importância à qualidade**, não se apegam os detalhes. Fazem as coisas por fazer, sem saber a fundo o que estão fazendo, ou porque estão fazendo.

Nem sempre é culpa do profissional, é uma área com muitos dirigentes despreparados.

Os diagramas de Caso de Uso são relevantes?

Relevante é. Mas **depende da qualidade** do que foi produzido.

Sempre haverá o profissional arrogante que vai analisar o diagrama de caso de uso e diz: “perdeu tempo fazendo isso? Um desenho com bonecos de palito, bolinhas e setinhas ligando as coisas?”.

Outro alguém pode falar: “para que especificar. Até minha mãe faz um diagrama melhor… desenhar bonecos e bolas não tem sentido, não agrega nada ao projeto!”.

> */* Já ouvi um gerente sênior falando da mãe dele, como foi descrito. Acontece… */*

Excetuando a ironia e falta de gentileza, realmente, fazer um “desenho” (diagrama) com bonecos de palito e bolas, ligando estas coisas, sem ter sentido semântico algum nisso, com **baixa qualidade** no material produzido, não tem utilidade mesmo.

É perder tempo, tempo que poderia ser empregado em coisas mais úteis ao projeto.

Mas se for um **trabalho bem feito**, se for um diagrama **produzido com qualidad**e, que realmente explora as possibilidades da técnica de modelagem de caso de uso, utiliza a técnica corretamente e **ajuda** a toda a equipe a entender e implementar o escopo do projeto da melhor forma, aí gera valor, aí **torna-se relevante**.

Relacionamento entre Casos de Uso

Relacionamentos entre Casos de Uso, principalmente para os profissionais que estão tendo o primeiro contato com o assunto, quase sempre geram alguma confusão. É natural.

As dúvidas sobre **Inclusão** (Include), **Extensão** (Extend) e **Generalização** [ou Herança] (Generalization) ocorrem com frequência, são comuns.

Existem alguns profissionais que defendem que "isso é bobagem, não precisa, include só resolve", mas isso é como usar uma linguagem orientada a objetos mas programar o software no paradigma "procedural"; compila a executa, mas fica **um monte de benefícios para trás**, além do que, depois que a modelagem acaba fica a confusão de entender "para que serve aquilo que eu fiz".

> */* No contexto do parágrafo acima, caso você seja um profissional que se preocupa com qualidade no que faz, recomendo muito que veja este meu vídeo sobre "fazer certo da primeira vez!" */*

Vamos ao que significa cada um dos três tipos de relacionamento citados. Consideremos que temos três Casos de Uso – A, B e C, e com base nos três vamos descrever cada um dos relacionamentos.

Include

Quando o caso de uso **A** "inclui" o caso de uso **B**, significa que **sempre** que o caso de uso **A** for executado o caso de uso **B** também

será executado. A direção do relacionamento é do caso de uso que está **incluindo** para o caso de uso **incluído**.

Extend

Quando o caso de uso **B** estende o caso de uso **A**, significa que quando o caso de uso **A** for executado o caso de uso **B** **poderá** (poderá – talvez não seja) ser executado também. A direção do relacionamento é do caso de uso **extensor** (aqui o caso de uso B) para o caso de uso **estendido** (aqui o caso de uso A).

Generalization

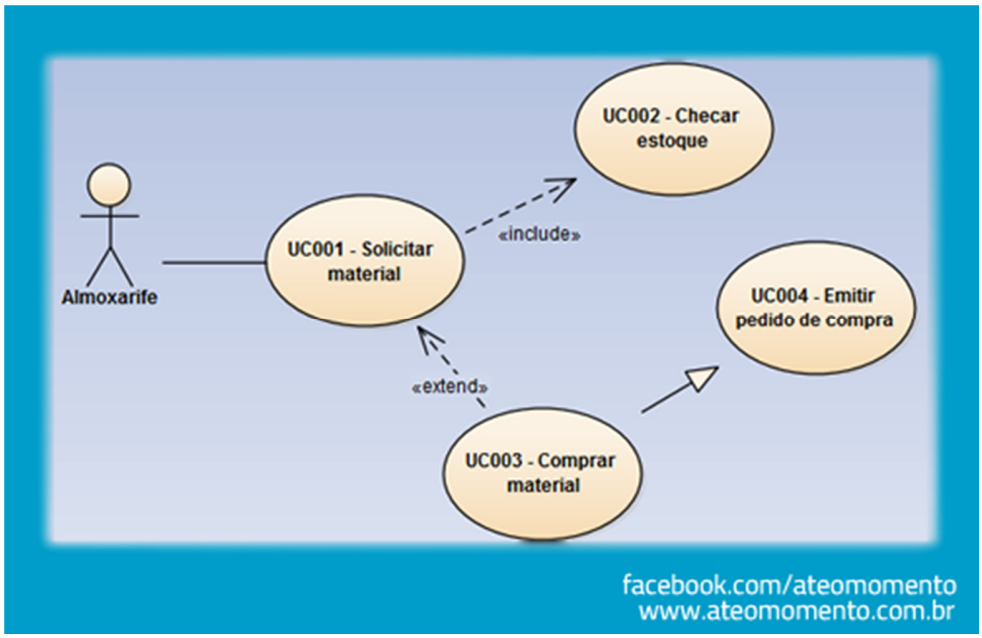
Quando o caso de uso **B** generaliza o caso de uso **C** isso significa que, além de fazer tudo que nele está especificado (ele = B), ele **também executará tudo que está especificado** no caso de uso **C**.

Muitos profissionais falam que isso não deve ser compreendido como a herança da orientação a objetos, mas na minha opinião deve ser sim, apenas (em tempo de modelagem de caso de uso) estamos num nível de abstração diferente, mas o produto final desta modelagem será software codificado.

A direção do relacionamento é sempre do **generalizador** (aqui o caso de uso B) para o **generalizado** (caso de uso C).

Exemplificando

Abaixo um diagrama com um cenário semelhante ao utilizado acima, ilustrando os relacionamentos.



No diagrama temos quatro Casos de Uso, e três relacionamentos diferentes: Include, Extend e Generalization.

Explicando o Include

O caso de uso “Solicitar Material” faz include no caso de uso “Checar Estoque”. Isso se dá porque**sempre** que houver a solicitação de material **sempre** haverá a consulta ao estoque para saber se o material está disponível.

Se sempre haverá, o relacionamento correto é o include.

Explicando o Extend

O caso de uso “Comprar Material” estende o caso de uso “Solicitar Material”. Isso se dá porque quando houver a solicitação de material, **caso o material não exista em estoque** (após consulta via o caso de uso “Checar estoque”) **poderá** ser solicitado a compra do item.

Mas também poderá não ser solicitada a compra, pois o item pode existir em estoque. Se **poderá** ser solicitada a compra (e não **sempre** será solicitada a compra) o relacionamento correto é o extend.

Explicando o Generalization

O caso de uso “Comprar Material” generaliza o caso de uso “Emitir pedido de compra”. Isso se dá porque no caso de uso “Emitir pedido de compra” **existe**especificação de como se realiza o pedido de compra, **processo que não se dá somente no contexto do almoxarifado**, mas é o mesmo em qualquer área do negócio.

Dessa forma, não justifica-se duplicar a especificação pertinente em outro caso de uso, basta **reaproveitar** o que já está pronto mas generalizado a ponto de poder ser aproveitado por alguém que o especialize.

A importância do uso correto nos relacionamentos

Especificações são feitas para serem **interpretadas,** e com base na interpretação, viabilizar a produção de software executável.

Quanto **mais qualidade** houver na especificação, **mais fácil** será de entendê-la, e **mais correta** será a interpretação de quem utilizá-la.

Essa facilidade gera **velocidade** na produção dos outros modelos (incluindo o modelo de código fonte, casos de teste etc.), **diminui a quantidade de defeitos** em potencial (quanto mais clara

uma especificação, menor a chance dela ser interpretada de forma errada) e gera outros benefícios diversos.

Concluindo

A clareza e corretude nos relacionamentos entre os casos de uso influencia diretamente na qualidade do projeto.

Muitos profissionais acham que o diagrama serve apenas para “colar as bolinhas” para que alguém os identifique e consiga “ver o que tem dentro”, ou seja, ver os cenários do Caso de Uso. Vimos que vai muito além disso…

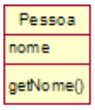
Diagrama de Classes

Um diagrama de três faces

Entidades

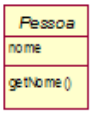
- **Classe**

  Representação gráfica



Classe Concreta

Uma classe é representada na forma de um retângulo, contendo duas linhas que separam 3 partes. A primeira contém no nome da classe, a segunda os atributos da classe e a última os métodos da mesma.

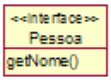


Classe Abstrata

A representação de uma classe abstrata em UML é quase igual à representação de uma classe concreta, a única diferença é o estilo da fonte do nome da classe, que, neste caso, está em itálico.

- **Interface**

  Representação Gráfica



Representação *Icon*

Representação *Label*

Perspectivas:

- Conceitual

  Apenas classes são utilizadas. Neste tipo de perspectiva, uma classe é interpretada como um conceito. Apenas atributos são utilizados.

- Especificação

  Tanto classes como interfaces são utilizados neste tipo de perspectiva. O foco consiste em mostrar as principais interfaces e classes juntamente com seus métodos.

  Não é necessário mostrar todos os métodos, pois o objetivo deste diagrama nesta perspectiva é prover uma maior entendimento da arquitetura do software a nível de interfaces.

- Implementação

	Nesta perspectiva, vários detalhes de implementação podem ser abordados, tais como: ◦ visibilidade de atributos e métodos; ◦ parâmetros de cada método, inclusive o tipo de cada um; ◦ tipos dos atributos e dos valores de retorno de cada método.

Diagrama de Classes
Um diagrama de três faces

Objetivo

Descrever os vários tipos de objetos no sistema e o relacionamento entre eles.

Perspectivas

Um diagrama de classes pode oferecer três perspectivas, cada uma para um tipo de usuário diferente. São elas:

- Conceitos ou Entidades (exemplo; mais exemplos)
  - Representa os conceitos do domínio em estudo.
  - Perspectiva destinada ao cliente.
- Classes (exemplo)
  - Tem foco nas principais interfaces da arquitetura, nos principais métodos, e não como eles irão ser implementados.
  - Perspectiva destinada as pessoas que não precisam saber detalhes de desenvolvimento, tais como gerentes de projeto.
- Classes de Software (exemplo)
  - Aborda vários detalhes de implementação, tais como navegabilidade, *tipo* dos atributos, etc.
  - Perspectiva destinada ao time de desenvolvimento.

Um diagrama de classes contém:

- Entidades
- Relacionamentos
- Exemplo contendo as notações

Exemplo Completo

Especificação do exemplo
Solução 1
Solução 2 (para quem tiver mais curiosidade)

Motivação

UML
- História
- Diagramas

Bibliografia

Diagrama de Atividades

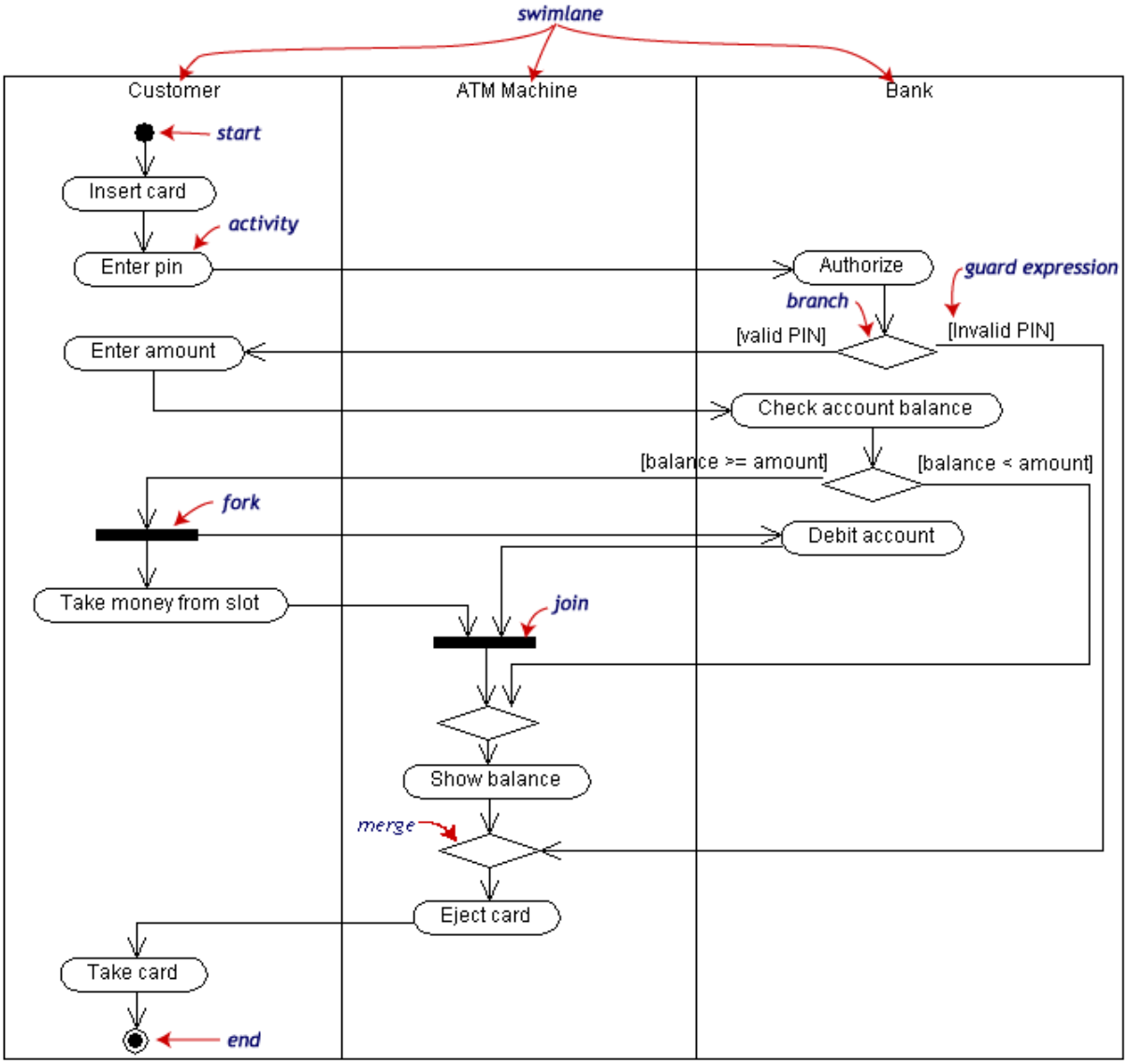
O objetivo do diagrama de atividades é mostrar o fluxo de atividades em um único processo. O diagrama mostra como um atividade depende uma da outra.

Um diagrama de atividade pode ser regiões denominadas *swimlanes*. Estas regiões esão associadas a um objeto do modelo. Desta forma, dentro de cada região, encontram-se as atividades relativas ao objeto da região.

As atividades são conectadas através de arcos (transições), que mostram as dependências entre elas.

Exemplo:

> Descrição do exemplo: Retirando dinheiro de um caixa eletrônico (para cartões de crédito).

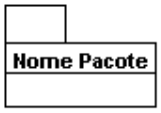


Diagramas de Pacotes

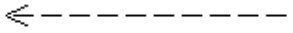
Objetivo principal: agrupar classes em pacotes.

Notação:

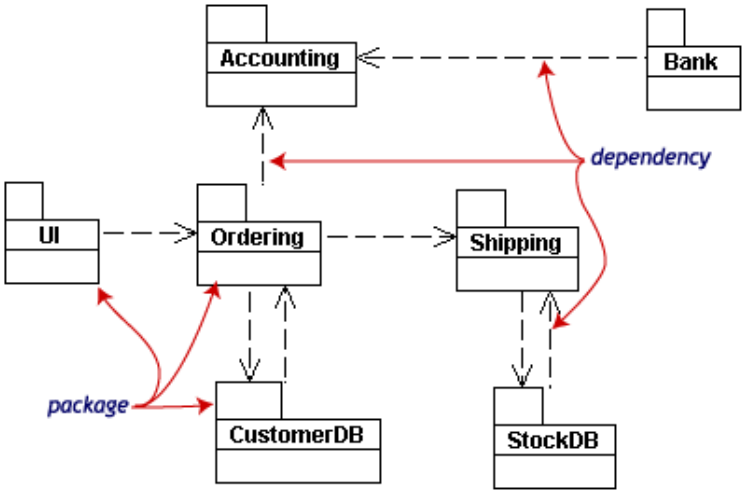
Pacote



Dependência



Exemplo:



Diagramas de Estado

Em um diagrama de estado, um objeto possui um comportamento e um estado.

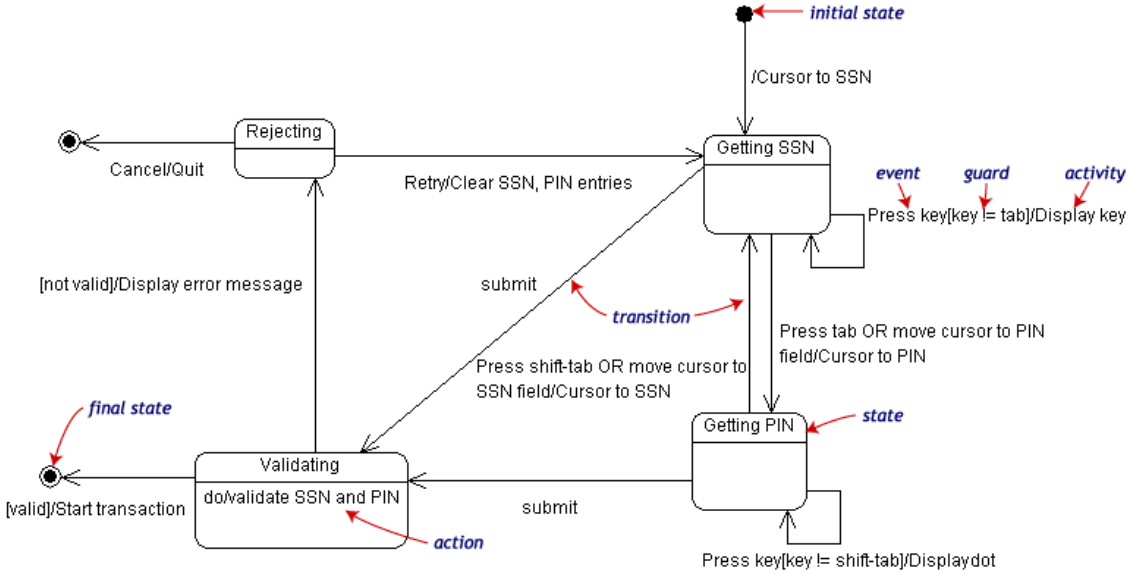
O estado de um objeto depende da atividade na qual ele está processando.

Um diagrama de estado mostra os possíveis estados de um objeto e as transações responsáveis pelas suas mudanças de estado.

Exemplo:

Descrição do exemplo: Modelagem do sistema de login. Para que o usuário seja autenticado, ele deve fornecer dois valores: SSN (Social Security Number) e o PIN (Personal ID Number). Após a submissão é feita uma validação.

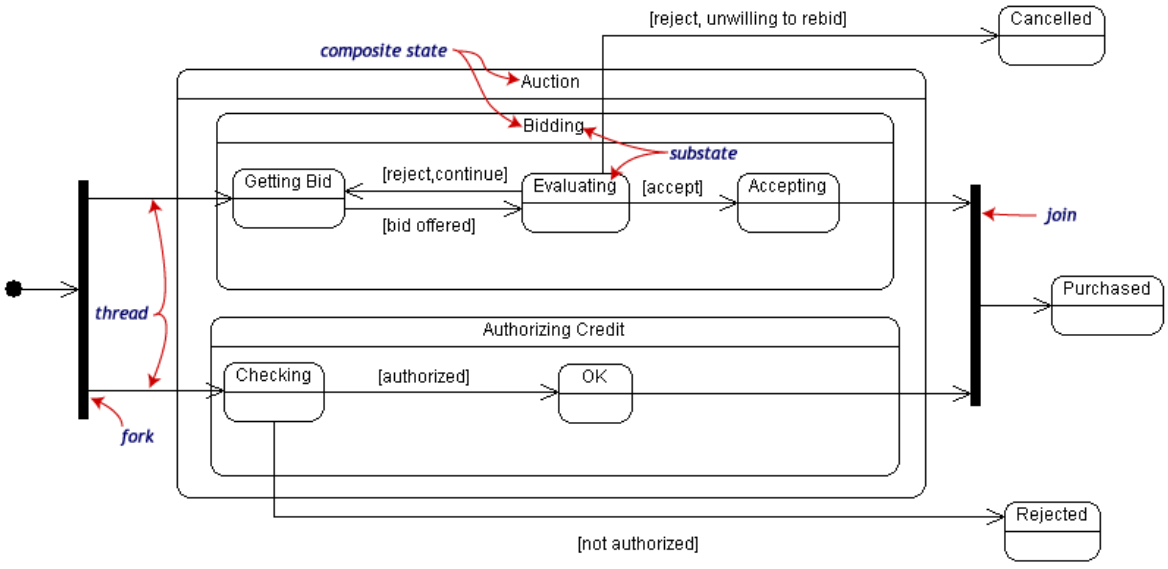
Diagrama de estado para o objeto *Login*.



Um diagrama de estados pode estar aninhado.Estados relacionados podem estar agrupados em um único estado.

Exemplo:

Descrição do exemplo: um sistema de leilão. Para que um lance seja efetuado com sucesso, a oferta tem que ser válida e o cliente tem que ter crédito suficiente.



}

Diagramas de Interação

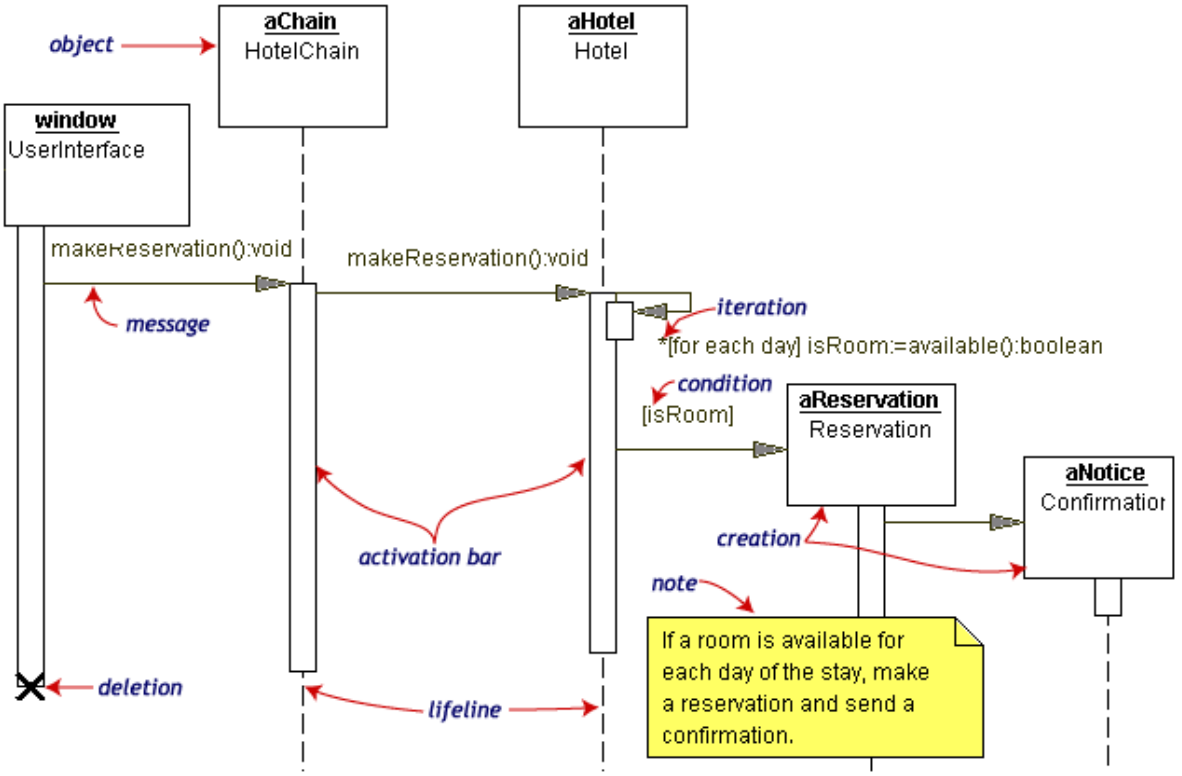
Diagramas de Seqüência

Consiste em um diagrama que tem o objetivo de mostrar como as mensagens entre os objetos são trocadas no decorrer do *tempo* para a realização de uma operação.

Em um diagrama de seqüência, os seguintes elementos podem ser encontrados:

- Linhas verticais representando o tempo de vida de um objeto (*lifeline*);
- Estas linhas verticais são preenchidas por barras verticais que indicam exatamente quando um objeto passou a existir. Quando um objeto desaparece, existe um "X" na parte inferior da barra;
- Linhas horizontais ou diagonais representando mensagens trocadas entre objetos. Estas linhas são acompanhadas de um rótulo que contém o nome da mensagem e, opcionalmente, os parâmetros da mesma. Observe que também podem existir mensagens enviadas para o mesmo objeto, representando uma iteração;
- Uma condição é representada por uma mensagem cujo rótulo é envolvido por colchetes;
- Mesagens de retorno são representadas por linhas horizontais tracejadas. Este tipo de mensagem não é freqüentemente representada nos diagramas, muitas vezes porque sua utilização leva a um grande número de setas no diagrama, atrapalhando o entendimento do mesmo. Este tipo de mensagem só deve ser mostrada quando forfundamental para a clareza do diagrama.

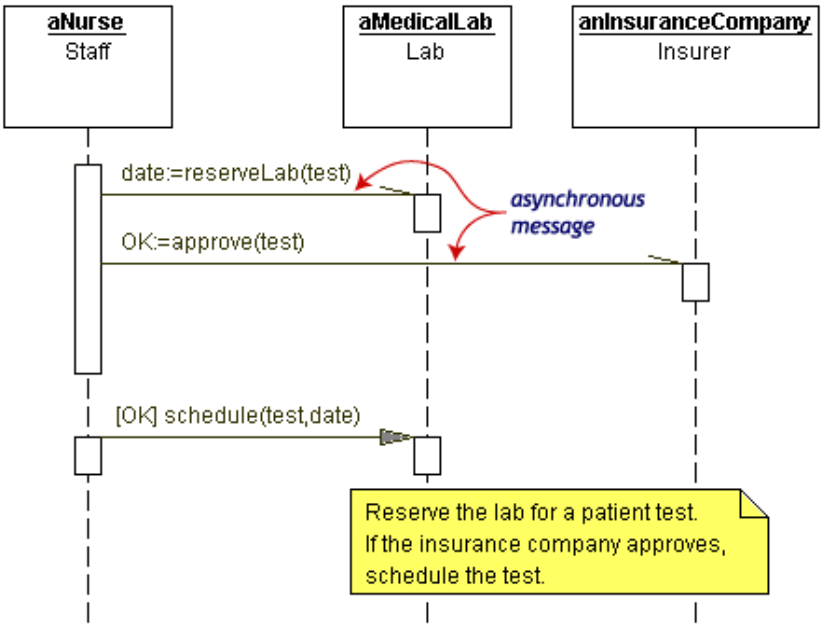
Observe a figura abaixo.



Representado processos concorrentes

Este tipo de diagrama também permite representar mensagens concorrentes assíncronas (mensagens que são processadas em paralelo sem um tempo definido para a sua realização).

Exemplo:


aNurse
Staff
aMedicalLab
Lab
anInsuranceCompany
Insurer
date:=reserveLab(test)
asynchronous
message
OK:=approve(test)
[OK] schedule(test,date)
Reserve the lab for a patient test.
If the insurance company approves,
schedule the test.

Diagramas de Interação

Diagramas de Colaboração

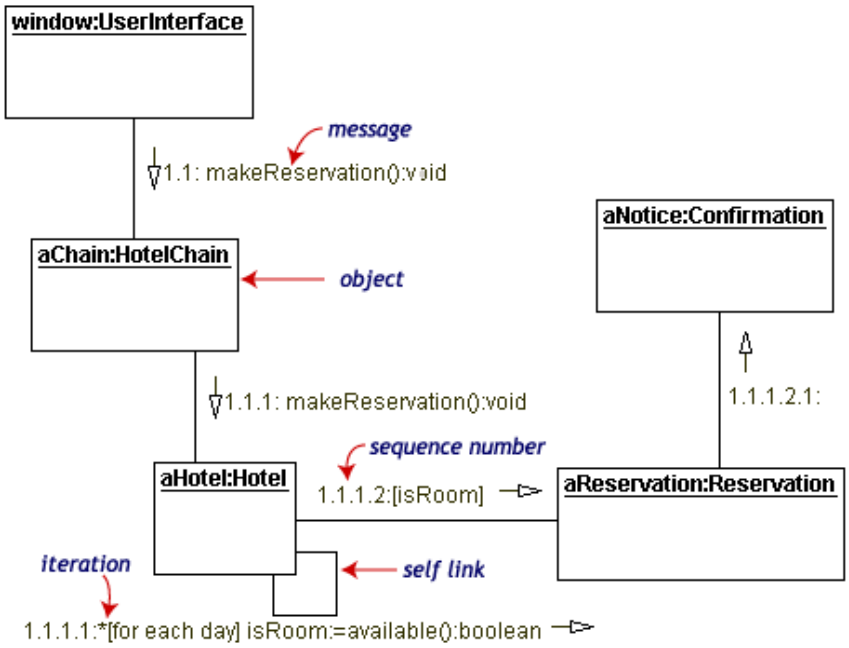
A grande diferença entre um diagrama de colaboração e um de seqüência consiste no fato de que o tempo não é mais representado por linhas verticais, mas sim através de uma numeração, que pode ser de duas formas:

- simples (1,2,3,...)
- composta (1.1, 1.2, 1.2.1, ...)

Um objeto é representado como um retângulo, contendo no seu interior um rótulo, que informa o nome do objeto e o nome da classe, separados por dois pontos. Detalhe: ambos podem ser omitidos.

A troca de mensagens entre os objetos segue o mesmo padrão que o apresentado nos diagramas de seqüência.

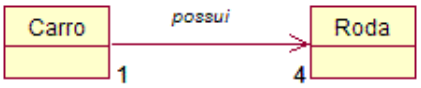
Veja o exemplos abaixo:



Relacionamentos

- Papel

  Descreve o relacionamento.



- Multiplicidade (utilizado em todas as perspectivas de forma uniforme)

  Notações possíveis

Tipos	***Significa***
0..1	Zero ou uma instância. A notação n..m indica n para m instâncias.
0..* ou *	Não existe limite para o número de instâncias.
1	Exatamente uma instância.
1..*	Ao menos uma instância.

  Exemplos:

- Associação (utilizado em todas as perspectivas)

  Representação Gráfica

  Associação

  Perspectiva:

  - Conceitual

    Define um relacionamento entre duas entidades conceituais do sistema.

  - Especificação

    Define responsabilidades entre duas classes. Implica que existem métodos que tratam desta responsabilidade.

  - Implementação

    Permite saber quem está apontando para quem, através da representação gráfica da navegabilidade. Além disto, é possível compreender melhor de que lado está a responsabilidade.

```
public class A {
```

```
public class A {
    private B b;
    public A( ){
    }
    public void setB(
B b ){
          this.b = b;
    }
    public B getB( ) {
        return b;
    }
}
```

```
public class B {
    public B( ){
    }
}
```

- Herança ou Generalização (utilizado em todas as perspectivas)

  Representação Gráfica

  Perspectiva:

  Seja B uma generalização (extensão) de A.

  - Conceitual

    Considera que B é um subtipo ou um tipo especial de A. O que é válido para A, também é válido para B.

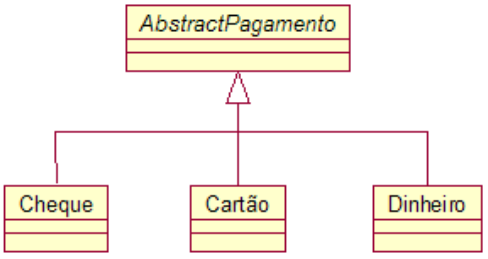
  - Especificação

    Ocorre uma herança de interface.

  - Implementação

    Ocorre uma herança de implementação.

  Exemplo de uma herança de implementação:

- Navegabilidade (utilizado apenas na perspectiva de implementação)

  Um relacionamento sem navegabilidade implica que ele pode ser lido de duas formas, isto é, em suas duas direções. Ex.:

Uma empresa possui um trabalhador, como também um trabalhador trabalha em uma empresa.

Utilizando a propriedade de navegabilidade, podemos restringir a forma de ler um relacionamento. Isto é, em vez de termos duas direções, teremos apenas uma direção (de acordo com a direção da navegação). Ex.:

Uma empresa possui um trabalhador.

- Agregação (utilizado apenas na perspectiva de implementação)

  Definição

  Agregação é uma associação em que um objeto é parte de outro, de tal forma que a parte pode existir sem o todo.

  Em mais baixo nível, uma agregação consiste de um objeto contendo referências para outros objetos, de tal forma que o primeiro seja o todo, e que os objetos referenciados sejam as partes do todo.

  A diferença entre os relacionamentos de associação e agregação ainda é algo de bastante discussão entre os gurus. De forma geral, utiliza-se agregação para enfatizar detalhes de uma futura implementação (perspectiva de implementação).

  Representação gráfica

  Agregação com navegabilidade


```
public class A {
    private B b;
    public A( ){
    }
    public void setB( B b ){
        this.b = b;
    }
    public B getB( ) {
       return b;
    }
}
```

```
public class B {
    public B( ){
    }
}
```

- Composição (utilizado apenas na perspectiva de implementação)

  Definição

  Em mais baixo nível, em termos de passagem por parâmetro, seria uma *passagem por valor*. Enquanto que agregação seria uma *passagem por referência*.

  O todo *contém* as partes (e não *referências* para as partes). Quando o todo desaparece, todas as partes também desaparecem.

  Representação Gráfica


```
public class A {
    private B b;
    public A( ){
        b = new B();
    }
}
```

```
public class B {
    public B( ){
    }
}
```

- Implementação (utilizado apenas na perspectiva de implementação)

  Em Inglês: *realization*

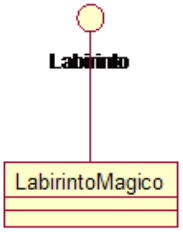
  Definição

  Utilizado para indicar que uma classe implementa uma interface

  Representação Gráfica

  Exemplo

  Implementação de uma interface representada por um círculo



  Implementação de uma interface representada por um retângulo