Projeto de Programas

Paulo Torrens

paulotorrens@gnu.org

Departamento de Ciência da Computação
Centro de Ciências e Tecnológias
Universidade do Estado de Santa Catarina

2020/1

Modelo Entidade-Relacionamento

- Ao projetar um software, é necessário consideração sobre o armazenamento das informações necessárias para a execução
- Em muitos projetos, especialmente os de maior porte, as informações são salvas através de um software externo, o **banco de dados**
- Embora existam outras variações, bancos de dados costumam funcionar através de **relacionamento** entre **entidades**
  - Entidades são representadas através de **tabelas**
  - Bancos que não usam o modelo relacional são tradicionalmente chamados de bancos NoSQL
  - A linguagem **SQL** (*Structured Query Language*) é uma linguagem padronizada para manipulação de dados relacionais

Modelo Entidade-Relacionamento

- Ao projetar um software, é necessário consideração sobre o armazenamento das informações necessárias para a execução
- Em muitos projetos, especialmente os de maior porte, as informações são salvas através de um software externo, o **banco de dados**
- Embora existam outras variações, bancos de dados costumam funcionar através de **relacionamento** entre **entidades**
  - Entidades são representadas através de **tabelas**
  - Bancos que não usam o modelo relacional são tradicionalmente chamados de bancos NoSQL
  - A linguagem **SQL** (*Structured Query Language*) é uma linguagem padronizada para manipulação de dados relacionais

Modelo Entidade-Relacionamento

- Ao projetar um software, é necessário consideração sobre o armazenamento das informações necessárias para a execução
- Em muitos projetos, especialmente os de maior porte, as informações são salvas através de um software externo, o **banco de dados**
- Embora existam outras variações, bancos de dados costumam funcionar através de **relacionamento** entre **entidades**
  - Entidades são representadas através de **tabelas**
  - Bancos que não usam o modelo relacional são tradicionalmente chamados de bancos NoSQL
  - A linguagem **SQL** (*Structured Query Language*) é uma linguagem padronizada para manipulação de dados relacionais

Modelo Entidade-Relacionamento

- Ao projetar um software, é necessário consideração sobre o armazenamento das informações necessárias para a execução
- Em muitos projetos, especialmente os de maior porte, as informações são salvas através de um software externo, o **banco de dados**
- Embora existam outras variações, bancos de dados costumam funcionar através de **relacionamento** entre **entidades**
  - Entidades são representadas através de **tabelas**
  - Bancos que não usam o modelo relacional são tradicionalmente chamados de bancos NoSQL
  - A linguagem **SQL** (*Structured Query Language*) é uma linguagem padronizada para manipulação de dados relacionais

Modelo Entidade-Relacionamento

- Ao projetar um software, é necessário consideração sobre o armazenamento das informações necessárias para a execução
- Em muitos projetos, especialmente os de maior porte, as informações são salvas através de um software externo, o **banco de dados**
- Embora existam outras variações, bancos de dados costumam funcionar através de **relacionamento** entre **entidades**
  - Entidades são representadas através de **tabelas**
  - Bancos que não usam o modelo relacional são tradicionalmente chamados de bancos NoSQL
  - A linguagem **SQL** (*Structured Query Language*) é uma linguagem padronizada para manipulação de dados relacionais

Modelo Entidade-Relacionamento

- Ao projetar um software, é necessário consideração sobre o armazenamento das informações necessárias para a execução
- Em muitos projetos, especialmente os de maior porte, as informações são salvas através de um software externo, o **banco de dados**
- Embora existam outras variações, bancos de dados costumam funcionar através de **relacionamento** entre **entidades**
  - Entidades são representadas através de **tabelas**
  - Bancos que não usam o modelo relacional são tradicionalmente chamados de bancos NoSQL
  - A linguagem **SQL** (*Structured Query Language*) é uma linguagem padronizada para manipulação de dados relacionais

Modelo Entidade-Relacionamento

- Uma forma de representar modelos relacionais é através do **diagrama ER**
- Conceitualmente, há três tipos de informação necessárias para a representação de um modelo:
  - **Entidades**, representando objetos de interesse para o negócio, como Cliente, Livro, Empréstimo, etc
  - **Atributos**, que são campos de uma entidade, salvando uma informação relevante a ela; por exemplo, um Cliente deve ter um nome, e-mail, endereço, etc
  - **Relacionamentos**, que são formas de se "navegar" dentro dos dados, e representam relações entre as entidades existentes; um relacionamento geralmente é representado por um verbo: um Cliente efetua um Empréstimo, um Empréstimo contém um Livro, etc

Modelo Entidade-Relacionamento

- Uma forma de representar modelos relacionais é através do **diagrama ER**
- Conceitualmente, há três tipos de informação necessárias para a representação de um modelo:
  - **Entidades**, representando objetos de interesse para o negócio, como Cliente, Livro, Empréstimo, etc
  - **Atributos**, que são campos de uma entidade, salvando uma informação relevante a ela; por exemplo, um Cliente deve ter um nome, e-mail, endereço, etc
  - **Relacionamentos**, que são formas de se "navegar" dentro dos dados, e representam relações entre as entidades existentes; um relacionamento geralmente é representado por um verbo: um Cliente efetua um Empréstimo, um Empréstimo contém um Livro, etc

Modelo Entidade-Relacionamento

- Uma forma de representar modelos relacionais é através do **diagrama ER**
- Conceitualmente, há três tipos de informação necessárias para a representação de um modelo:
  - **Entidades**, representando objetos de interesse para o negócio, como Cliente, Livro, Empréstimo, etc
  - **Atributos**, que são campos de uma entidade, salvando uma informação relevante a ela; por exemplo, um Cliente deve ter um nome, e-mail, endereço, etc
  - **Relacionamentos**, que são formas de se "navegar" dentro dos dados, e representam relações entre as entidades existentes; um relacionamento geralmente é representado por um verbo: um Cliente efetua um Empréstimo, um Empréstimo contém um Livro, etc

Modelo Entidade-Relacionamento

- Uma forma de representar modelos relacionais é através do **diagrama ER**
- Conceitualmente, há três tipos de informação necessárias para a representação de um modelo:
  - **Entidades**, representando objetos de interesse para o negócio, como Cliente, Livro, Empréstimo, etc
  - **Atributos**, que são campos de uma entidade, salvando uma informação relevante a ela; por exemplo, um Cliente deve ter um nome, e-mail, endereço, etc
  - **Relacionamentos**, que são formas de se "navegar" dentro dos dados, e representam relações entre as entidades existentes; um relacionamento geralmente é representado por um verbo: um Cliente efetua um Empréstimo, um Empréstimo contém um Livro, etc

Modelo Entidade-Relacionamento

- Uma forma de representar modelos relacionais é através do **diagrama ER**
- Conceitualmente, há três tipos de informação necessárias para a representação de um modelo:
  - **Entidades**, representando objetos de interesse para o negócio, como Cliente, Livro, Empréstimo, etc
  - **Atributos**, que são campos de uma entidade, salvando uma informação relevante a ela; por exemplo, um Cliente deve ter um nome, e-mail, endereço, etc
  - **Relacionamentos**, que são formas de se "navegar" dentro dos dados, e representam relações entre as entidades existentes; um relacionamento geralmente é representado por um verbo: um Cliente efetua um Empréstimo, um Empréstimo contém um Livro, etc

Modelo Entidade-Relacionamento

- O intuito do diagrama ER é representar, **com níveis variados de precisão**, as necessidades das informações de um sistema
  - Muitas vezes, cabe ao programador ou DBA (*Database Administrator*) implementar o banco de dados baseando-se no diagrama
- Diagramas ER não apresentam um padrão efetivo, porém possuem algumas convenções que facilitam o entendimento
  - Entidades são representadas através de retângulos
  - Atributos são ou representados através de círculos conectados à entidade, ou apresentados dentro dela em forma similar a um diagrama UML
  - Relacionamentos são representados através de losangos conectados às entidades que relacionam

Modelo Entidade-Relacionamento

- O intuito do diagrama ER é representar, **com níveis variados de precisão**, as necessidades das informações de um sistema
  - Muitas vezes, cabe ao programador ou DBA (*Database Administrator*) implementar o banco de dados baseando-se no diagrama
- Diagramas ER não apresentam um padrão efetivo, porém possuem algumas convenções que facilitam o entendimento
  - Entidades são representadas através de retângulos
  - Atributos são ou representados através de círculos conectados à entidade, ou apresentados dentro dela em forma similar a um diagrama UML
  - Relacionamentos são representados através de losangos conectados às entidades que relacionam

Modelo Entidade-Relacionamento

- O intuito do diagrama ER é representar, **com níveis variados de precisão**, as necessidades das informações de um sistema
  - Muitas vezes, cabe ao programador ou DBA (*Database Administrator*) implementar o banco de dados baseando-se no diagrama
- Diagramas ER não apresentam um padrão efetivo, porém possuem algumas convenções que facilitam o entendimento
  - Entidades são representadas através de retângulos
  - Atributos são ou representados através de círculos conectados à entidade, ou apresentados dentro dela em forma similar a um diagrama UML
  - Relacionamentos são representados através de losangos conectados às entidades que relacionam

Modelo Entidade-Relacionamento

- O intuito do diagrama ER é representar, **com níveis variados de precisão**, as necessidades das informações de um sistema
  - Muitas vezes, cabe ao programador ou DBA (*Database Administrator*) implementar o banco de dados baseando-se no diagrama
- Diagramas ER não apresentam um padrão efetivo, porém possuem algumas convenções que facilitam o entendimento
  - Entidades são representadas através de retângulos
  - Atributos são ou representados através de círculos conectados à entidade, ou apresentados dentro dela em forma similar a um diagrama UML
  - Relacionamentos são representados através de losangos conectados às entidades que relacionam

Modelo Entidade-Relacionamento

- O intuito do diagrama ER é representar, **com níveis variados de precisão**, as necessidades das informações de um sistema
  - Muitas vezes, cabe ao programador ou DBA (*Database Administrator*) implementar o banco de dados baseando-se no diagrama
- Diagramas ER não apresentam um padrão efetivo, porém possuem algumas convenções que facilitam o entendimento
  - Entidades são representadas através de retângulos
  - Atributos são ou representados através de círculos conectados à entidade, ou apresentados dentro dela em forma similar a um diagrama UML
  - Relacionamentos são representados através de losangos conectados às entidades que relacionam

Modelo Entidade-Relacionamento

- O intuito do diagrama ER é representar, **com níveis variados de precisão**, as necessidades das informações de um sistema
  - Muitas vezes, cabe ao programador ou DBA (*Database Administrator*) implementar o banco de dados baseando-se no diagrama
- Diagramas ER não apresentam um padrão efetivo, porém possuem algumas convenções que facilitam o entendimento
  - Entidades são representadas através de retângulos
  - Atributos são ou representados através de círculos conectados à entidade, ou apresentados dentro dela em forma similar a um diagrama UML
  - Relacionamentos são representados através de losangos conectados às entidades que relacionam

Modelo Entidade-Relacionamento

- Alguns atributos especiais podem ser classificados como **chaves primárias**, geralmente sublinhados no diagrama
  - A ideia de uma chave primária é **identificar** um elemento daquela entidade, sendo um valor único; por exemplo, um Cliente pode ser identificado por um UUID ou por seu CPF
- Relacionamentos possuem uma **cardinalidade**
  - Ao se criar uma relação entre entidades A e B, é necessário informar quantos A se relacionam a B, e quantos B se relacionam a A
  - Cada lado de uma relação possui uma cardinalidade, podendo ser **opcional** (zero ou um), **única** (apenas um), **múltipla** (zero ou mais), etc
  - Um para muitos, muitos para muitos, etc
  - Por exemplo, tendo **Cliente 1** $\longleftrightarrow$ **N Empréstimo** como uma relação, um cliente pode efetuar múltiplos empréstimos, mas cada empréstimo registrado é atrelado a apenas um cliente

Modelo Entidade-Relacionamento

- Alguns atributos especiais podem ser classificados como **chaves primárias**, geralmente sublinhados no diagrama
  - A ideia de uma chave primária é **identificar** um elemento daquela entidade, sendo um valor único; por exemplo, um Cliente pode ser identificado por um UUID ou por seu CPF
- Relacionamentos possuem uma **cardinalidade**
  - Ao se criar uma relação entre entidades A e B, é necessário informar quantos A se relacionam a B, e quantos B se relacionam a A
  - Cada lado de uma relação possui uma cardinalidade, podendo ser **opcional** (zero ou um), **única** (apenas um), **múltipla** (zero ou mais), etc
  - Um para muitos, muitos para muitos, etc
  - Por exemplo, tendo **Cliente 1** $\longleftrightarrow$ **N Empréstimo** como uma relação, um cliente pode efetuar múltiplos empréstimos, mas cada empréstimo registrado é atrelado a apenas um cliente

Modelo Entidade-Relacionamento

- Alguns atributos especiais podem ser classificados como **chaves primárias**, geralmente sublinhados no diagrama
  - A ideia de uma chave primária é **identificar** um elemento daquela entidade, sendo um valor único; por exemplo, um Cliente pode ser identificado por um UUID ou por seu CPF
- Relacionamentos possuem uma **cardinalidade**
  - Ao se criar uma relação entre entidades A e B, é necessário informar quantos A se relacionam a B, e quantos B se relacionam a A
  - Cada lado de uma relação possui uma cardinalidade, podendo ser **opcional** (zero ou um), **única** (apenas um), **múltipla** (zero ou mais), etc
  - Um para muitos, muitos para muitos, etc
  - Por exemplo, tendo **Cliente 1** $\longleftrightarrow$ **N Empréstimo** como uma relação, um cliente pode efetuar múltiplos empréstimos, mas cada empréstimo registrado é atrelado a apenas um cliente

Modelo Entidade-Relacionamento

- Alguns atributos especiais podem ser classificados como **chaves primárias**, geralmente sublinhados no diagrama
  - A ideia de uma chave primária é **identificar** um elemento daquela entidade, sendo um valor único; por exemplo, um Cliente pode ser identificado por um UUID ou por seu CPF
- Relacionamentos possuem uma **cardinalidade**
  - Ao se criar uma relação entre entidades A e B, é necessário informar quantos A se relacionam a B, e quantos B se relacionam a A
  - Cada lado de uma relação possui uma cardinalidade, podendo ser **opcional** (zero ou um), **única** (apenas um), **múltipla** (zero ou mais), etc
  - Um para muitos, muitos para muitos, etc
  - Por exemplo, tendo **Cliente 1** $\longleftrightarrow$ **N Empréstimo** como uma relação, um cliente pode efetuar múltiplos empréstimos, mas cada empréstimo registrado é atrelado a apenas um cliente

Modelo Entidade-Relacionamento

- Alguns atributos especiais podem ser classificados como **chaves primárias**, geralmente sublinhados no diagrama
  - A ideia de uma chave primária é **identificar** um elemento daquela entidade, sendo um valor único; por exemplo, um Cliente pode ser identificado por um UUID ou por seu CPF
- Relacionamentos possuem uma **cardinalidade**
  - Ao se criar uma relação entre entidades A e B, é necessário informar quantos A se relacionam a B, e quantos B se relacionam a A
  - Cada lado de uma relação possui uma cardinalidade, podendo ser **opcional** (zero ou um), **única** (apenas um), **múltipla** (zero ou mais), etc
  - Um para muitos, muitos para muitos, etc
  - Por exemplo, tendo **Cliente 1** $\longleftrightarrow$ **N Empréstimo** como uma relação, um cliente pode efetuar múltiplos empréstimos, mas cada empréstimo registrado é atrelado a apenas um cliente

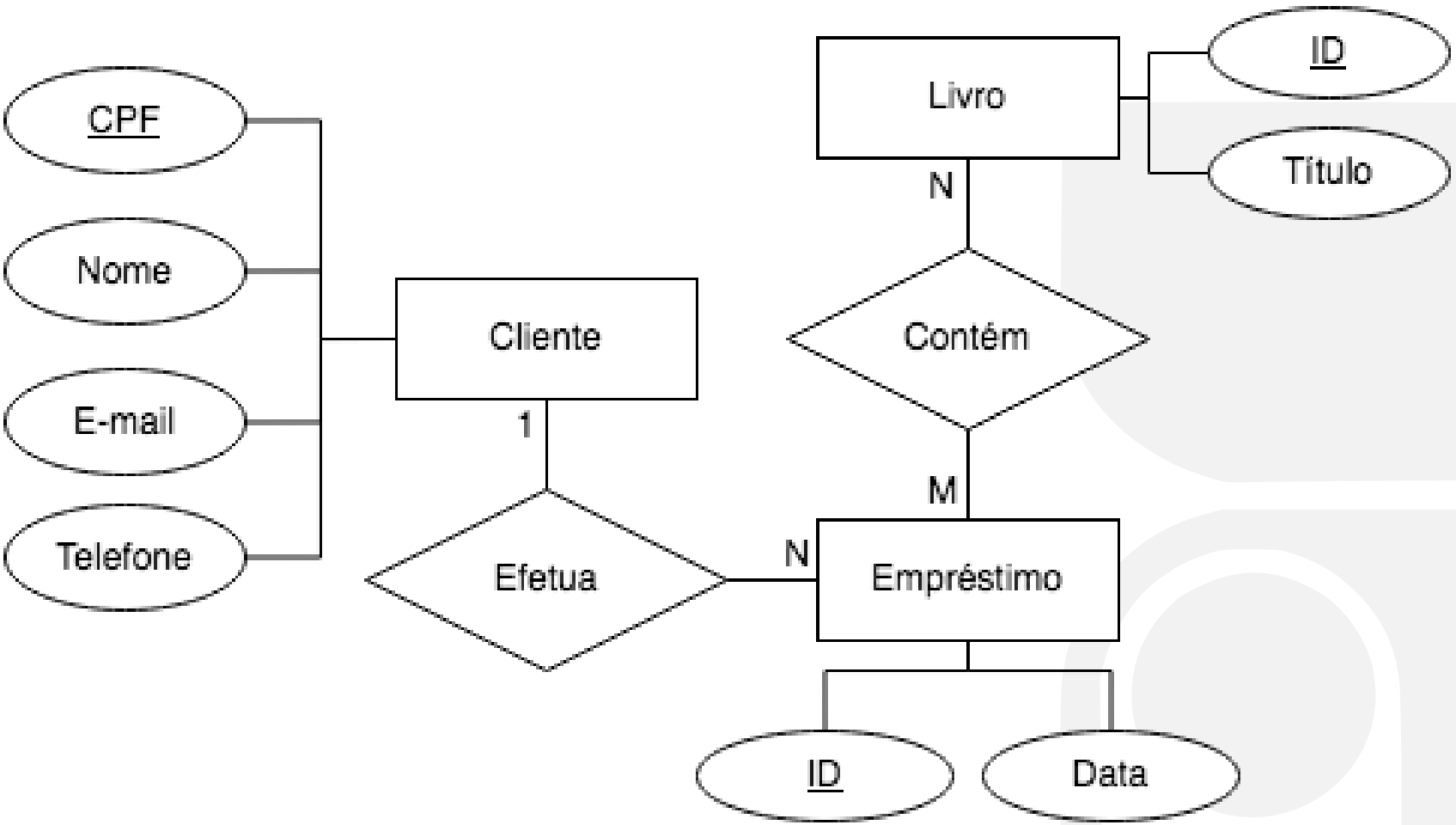
Modelo Entidade-Relacionamento

- Alguns atributos especiais podem ser classificados como **chaves primárias**, geralmente sublinhados no diagrama
  - A ideia de uma chave primária é **identificar** um elemento daquela entidade, sendo um valor único; por exemplo, um Cliente pode ser identificado por um UUID ou por seu CPF
- Relacionamentos possuem uma **cardinalidade**
  - Ao se criar uma relação entre entidades A e B, é necessário informar quantos A se relacionam a B, e quantos B se relacionam a A
  - Cada lado de uma relação possui uma cardinalidade, podendo ser **opcional** (zero ou um), **única** (apenas um), **múltipla** (zero ou mais), etc
  - Um para muitos, muitos para muitos, etc
  - Por exemplo, tendo **Cliente 1** $\longleftrightarrow$ **N Empréstimo** como uma relação, um cliente pode efetuar múltiplos empréstimos, mas cada empréstimo registrado é atrelado a apenas um cliente

Modelo Entidade-Relacionamento

- Alguns atributos especiais podem ser classificados como **chaves primárias**, geralmente sublinhados no diagrama
  - A ideia de uma chave primária é **identificar** um elemento daquela entidade, sendo um valor único; por exemplo, um Cliente pode ser identificado por um UUID ou por seu CPF
- Relacionamentos possuem uma **cardinalidade**
  - Ao se criar uma relação entre entidades A e B, é necessário informar quantos A se relacionam a B, e quantos B se relacionam a A
  - Cada lado de uma relação possui uma cardinalidade, podendo ser **opcional** (zero ou um), **única** (apenas um), **múltipla** (zero ou mais), etc
  - Um para muitos, muitos para muitos, etc
  - Por exemplo, tendo **Cliente 1** $\longleftrightarrow$ **N Empréstimo** como uma relação, um cliente pode efetuar múltiplos empréstimos, mas cada empréstimo registrado é atrelado a apenas um cliente

Modelo Entidade-Relacionamento (notação de Chen)



Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

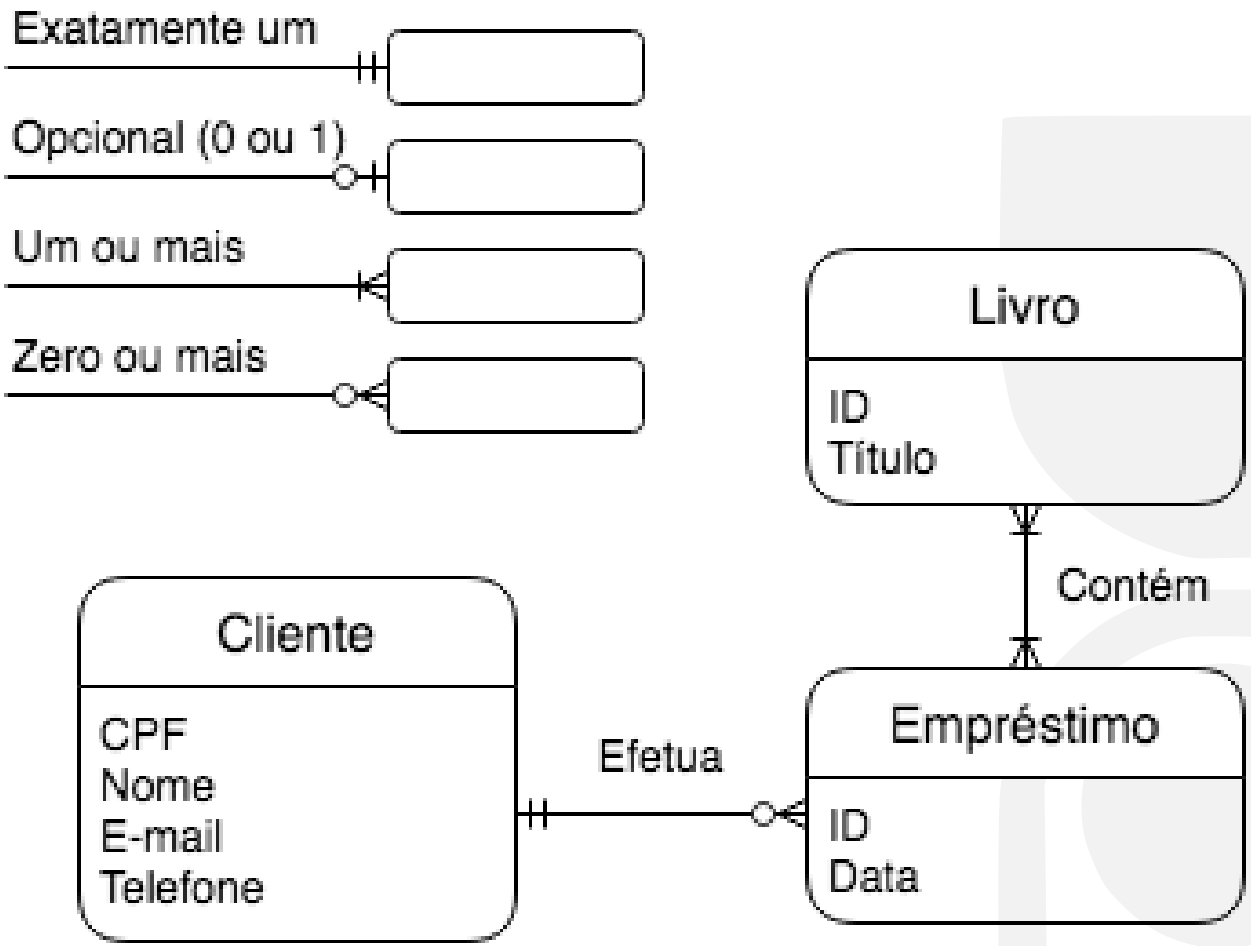
Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

Notação *crow's foot*

- O tipo de notação demonstrada anteriormente se chama notação de Chen
- Outra alternativa para representar modelos ER é através da notação *crow's foot*
- Entidades são representadas através de retângulos com duas seções, similar a um diagrama de classes UML
  - Atributos são anotados diretamente dentro das entidades
- Relacionamentos são representados diretamente por linhas conectando entidades, onde as pontas das setas representam a cardinalidade
  - Similar à notação de Chen, a cardinalidade de uma entidade em um relacionamento é anotada diretamente perto da entidade
  - Um traço representa uma cardinalidade obrigatória
  - Um círculo representa uma cardinalidade opcional
  - O "pé de corvo" representa uma cardinalidade múltipla

Modelo Entidade-Relacionamento (notação *crow's foot*)



Padrão UML

- Historicamente, várias notações gráficas para modelagem de sistemas foram propostas, sem uma forma prevalente
- O padrão UML (do inglês, *Unified Modeling Language*) foi proposto em 1994 como uma forma padronizada de se modelar sistemas **orientados a objetos**, e se tornou popular para o projeto de *software*
- O UML é descrito por um **metamodelo**: ele é modelado através de diagramas que descrevem diagramas
- Atualmente na versão 2.5.1, o UML contém vários tipos de diagramas distintos, se dividindo em duas categorias
  - Diagramas estruturais, cujo objetivo é representar partes estáticas de um projeto, como o diagramas de classe
  - Diagramas comportamentais, cujo objetivo é representar iterações e fluxos dinâmicos em um sistema, como por exemplo os diagramas de atividade e sequência

Padrão UML

- Historicamente, várias notações gráficas para modelagem de sistemas foram propostas, sem uma forma prevalente
- O padrão UML (do inglês, *Unified Modeling Language*) foi proposto em 1994 como uma forma padronizada de se modelar sistemas **orientados a objetos**, e se tornou popular para o projeto de *software*
- O UML é descrito por um **metamodelo**: ele é modelado através de diagramas que descrevem diagramas
- Atualmente na versão 2.5.1, o UML contém vários tipos de diagramas distintos, se dividindo em duas categorias
  - Diagramas estruturais, cujo objetivo é representar partes estáticas de um projeto, como o diagramas de classe
  - Diagramas comportamentais, cujo objetivo é representar iterações e fluxos dinâmicos em um sistema, como por exemplo os diagramas de atividade e sequência

Padrão UML

- Historicamente, várias notações gráficas para modelagem de sistemas foram propostas, sem uma forma prevalente
- O padrão UML (do inglês, *Unified Modeling Language*) foi proposto em 1994 como uma forma padronizada de se modelar sistemas **orientados a objetos**, e se tornou popular para o projeto de *software*
- O UML é descrito por um **metamodelo**: ele é modelado através de diagramas que descrevem diagramas
- Atualmente na versão 2.5.1, o UML contém vários tipos de diagramas distintos, se dividindo em duas categorias
  - Diagramas estruturais, cujo objetivo é representar partes estáticas de um projeto, como o diagramas de classe
  - Diagramas comportamentais, cujo objetivo é representar iterações e fluxos dinâmicos em um sistema, como por exemplo os diagramas de atividade e sequência

Padrão UML

- Historicamente, várias notações gráficas para modelagem de sistemas foram propostas, sem uma forma prevalente
- O padrão UML (do inglês, *Unified Modeling Language*) foi proposto em 1994 como uma forma padronizada de se modelar sistemas **orientados a objetos**, e se tornou popular para o projeto de *software*
- O UML é descrito por um **metamodelo**: ele é modelado através de diagramas que descrevem diagramas
- Atualmente na versão 2.5.1, o UML contém vários tipos de diagramas distintos, se dividindo em duas categorias
  - Diagramas estruturais, cujo objetivo é representar partes estáticas de um projeto, como o diagramas de classe
  - Diagramas comportamentais, cujo objetivo é representar iterações e fluxos dinâmicos em um sistema, como por exemplo os diagramas de atividade e sequência

Padrão UML

- Historicamente, várias notações gráficas para modelagem de sistemas foram propostas, sem uma forma prevalente
- O padrão UML (do inglês, *Unified Modeling Language*) foi proposto em 1994 como uma forma padronizada de se modelar sistemas **orientados a objetos**, e se tornou popular para o projeto de *software*
- O UML é descrito por um **metamodelo**: ele é modelado através de diagramas que descrevem diagramas
- Atualmente na versão 2.5.1, o UML contém vários tipos de diagramas distintos, se dividindo em duas categorias
  - Diagramas estruturais, cujo objetivo é representar partes estáticas de um projeto, como o diagramas de classe
  - Diagramas comportamentais, cujo objetivo é representar iterações e fluxos dinâmicos em um sistema, como por exemplo os diagramas de atividade e sequência

Padrão UML

- Historicamente, várias notações gráficas para modelagem de sistemas foram propostas, sem uma forma prevalente
- O padrão UML (do inglês, *Unified Modeling Language*) foi proposto em 1994 como uma forma padronizada de se modelar sistemas **orientados a objetos**, e se tornou popular para o projeto de *software*
- O UML é descrito por um **metamodelo**: ele é modelado através de diagramas que descrevem diagramas
- Atualmente na versão 2.5.1, o UML contém vários tipos de diagramas distintos, se dividindo em duas categorias
  - Diagramas estruturais, cujo objetivo é representar partes estáticas de um projeto, como o diagramas de classe
  - Diagramas comportamentais, cujo objetivo é representar iterações e fluxos dinâmicos em um sistema, como por exemplo os diagramas de atividade e sequência

Diagrama de classes

- Através do **diagrama de classes** é possível representar a estrutura planejada para as classes em um sistema orientado a objetos
- Detalhes: `www.uml-diagrams.org/class.html`
- O diagrama é representado através de um retângulo com **três compartimentos** distintos
  - O primeiro compartimento contém o **nome da classe**, centralizado e em negrito, dentre outras possíveis informações
  - O segundo compartimento contém **atributos**, que são os campos de uma classe, e pode ser omitido
  - O terceiro compartimento contém **operações**, que são os métodos de uma classe, e pode ser omitido
  - Chamamos atributos e operações coletivamente de *features*
- As *features* podem pertencer à própria classe (estáticas), sendo sublinhadas, ou à instâncias delas, além de possuírem uma visibilidade (pública, privada, protegida, ou do pacote)

Diagrama de classes

- Através do **diagrama de classes** é possível representar a estrutura planejada para as classes em um sistema orientado a objetos
- Detalhes: `www.uml-diagrams.org/class.html`
- O diagrama é representado através de um retângulo com **três compartimentos** distintos
  - O primeiro compartimento contém o **nome da classe**, centralizado e em negrito, dentre outras possíveis informações
  - O segundo compartimento contém **atributos**, que são os campos de uma classe, e pode ser omitido
  - O terceiro compartimento contém **operações**, que são os métodos de uma classe, e pode ser omitido
  - Chamamos atributos e operações coletivamente de *features*
- As *features* podem pertencer à própria classe (estáticas), sendo sublinhadas, ou à instâncias delas, além de possuírem uma visibilidade (pública, privada, protegida, ou do pacote)

Diagrama de classes

- Através do **diagrama de classes** é possível representar a estrutura planejada para as classes em um sistema orientado a objetos
- Detalhes: `www.uml-diagrams.org/class.html`
- O diagrama é representado através de um retângulo com **três compartimentos** distintos
  - O primeiro compartimento contém o **nome da classe**, centralizado e em negrito, dentre outras possíveis informações
  - O segundo compartimento contém **atributos**, que são os campos de uma classe, e pode ser omitido
  - O terceiro compartimento contém **operações**, que são os métodos de uma classe, e pode ser omitido
  - Chamamos atributos e operações coletivamente de *features*
- As *features* podem pertencer à própria classe (estáticas), sendo sublinhadas, ou à instâncias delas, além de possuírem uma visibilidade (pública, privada, protegida, ou do pacote)

Diagrama de classes

- Através do **diagrama de classes** é possível representar a estrutura planejada para as classes em um sistema orientado a objetos
- Detalhes: `www.uml-diagrams.org/class.html`
- O diagrama é representado através de um retângulo com **três compartimentos** distintos
  - O primeiro compartimento contém o **nome da classe**, centralizado e em negrito, dentre outras possíveis informações
  - O segundo compartimento contém **atributos**, que são os campos de uma classe, e pode ser omitido
  - O terceiro compartimento contém **operações**, que são os métodos de uma classe, e pode ser omitido
  - Chamamos atributos e operações coletivamente de *features*
- As *features* podem pertencer à própria classe (estáticas), sendo sublinhadas, ou à instâncias delas, além de possuírem uma visibilidade (pública, privada, protegida, ou do pacote)

Diagrama de classes

- Através do **diagrama de classes** é possível representar a estrutura planejada para as classes em um sistema orientado a objetos
- Detalhes: `www.uml-diagrams.org/class.html`
- O diagrama é representado através de um retângulo com **três compartimentos** distintos
  - O primeiro compartimento contém o **nome da classe**, centralizado e em negrito, dentre outras possíveis informações
  - O segundo compartimento contém **atributos**, que são os campos de uma classe, e pode ser omitido
  - O terceiro compartimento contém **operações**, que são os métodos de uma classe, e pode ser omitido
  - Chamamos atributos e operações coletivamente de *features*
- As *features* podem pertencer à própria classe (estáticas), sendo sublinhadas, ou à instâncias delas, além de possuírem uma visibilidade (pública, privada, protegida, ou do pacote)

Diagrama de classes

- Através do **diagrama de classes** é possível representar a estrutura planejada para as classes em um sistema orientado a objetos
- Detalhes: `www.uml-diagrams.org/class.html`
- O diagrama é representado através de um retângulo com **três compartimentos** distintos
  - O primeiro compartimento contém o **nome da classe**, centralizado e em negrito, dentre outras possíveis informações
  - O segundo compartimento contém **atributos**, que são os campos de uma classe, e pode ser omitido
  - O terceiro compartimento contém **operações**, que são os métodos de uma classe, e pode ser omitido
  - Chamamos atributos e operações coletivamente de *features*
- As *features* podem pertencer à própria classe (estáticas), sendo sublinhadas, ou à instâncias delas, além de possuírem uma visibilidade (pública, privada, protegida, ou do pacote)

Diagrama de classes

- Através do **diagrama de classes** é possível representar a estrutura planejada para as classes em um sistema orientado a objetos
- Detalhes: `www.uml-diagrams.org/class.html`
- O diagrama é representado através de um retângulo com **três compartimentos** distintos
  - O primeiro compartimento contém o **nome da classe**, centralizado e em negrito, dentre outras possíveis informações
  - O segundo compartimento contém **atributos**, que são os campos de uma classe, e pode ser omitido
  - O terceiro compartimento contém **operações**, que são os métodos de uma classe, e pode ser omitido
  - Chamamos atributos e operações coletivamente de *features*
- As *features* podem pertencer à própria classe (estáticas), sendo sublinhadas, ou à instâncias delas, além de possuírem uma visibilidade (pública, privada, protegida, ou do pacote)

Diagrama de classes

- Através do **diagrama de classes** é possível representar a estrutura planejada para as classes em um sistema orientado a objetos
- Detalhes: `www.uml-diagrams.org/class.html`
- O diagrama é representado através de um retângulo com **três compartimentos** distintos
  - O primeiro compartimento contém o **nome da classe**, centralizado e em negrito, dentre outras possíveis informações
  - O segundo compartimento contém **atributos**, que são os campos de uma classe, e pode ser omitido
  - O terceiro compartimento contém **operações**, que são os métodos de uma classe, e pode ser omitido
  - Chamamos atributos e operações coletivamente de *features*
- As *features* podem pertencer à própria classe (estáticas), sendo sublinhadas, ou à instâncias delas, além de possuírem uma visibilidade (pública, privada, protegida, ou do pacote)

Diagrama de classes

- É possível demonstrar relações entre classes UML através de **relacionamentos**, representados através de linhas conectando duas classes com pontas distintas
  - **Generalização** representa a ideia que uma classe herda *features* de uma classe pai, representando a ideia de herança em linguagens orientadas a objetos, como, por exemplo, as classes Cachorro e Mamífero (oposto: **especialização**)
  - **Associação** representa que duas classes são, semanticamente, associadas em um sistema, e trocam mensagens entre si, como, por exemplo, as classes Vôo e Avião
  - **Composição** representa que uma classe compõe outra classe que não existe independentemente, como, por exemplo, as classes Casa e Cômodo (um cômodo não existe sem uma casa)
  - **Agregação** representa que uma classe agrega outra classe, a qual pode existir independentemente, como, por exemplo, as classes Matéria e Aluno (se a matéria for cancelada, os alunos continuam existindo)

Diagrama de classes

- É possível demonstrar relações entre classes UML através de **relacionamentos**, representados através de linhas conectando duas classes com pontas distintas
  - **Generalização** representa a ideia que uma classe herda *features* de uma classe pai, representando a ideia de herança em linguagens orientadas a objetos, como, por exemplo, as classes Cachorro e Mamífero (oposto: **especialização**)
  - **Associação** representa que duas classes são, semanticamente, associadas em um sistema, e trocam mensagens entre si, como, por exemplo, as classes Vôo e Avião
  - **Composição** representa que uma classe compõe outra classe que não existe independentemente, como, por exemplo, as classes Casa e Cômodo (um cômodo não existe sem uma casa)
  - **Agregação** representa que uma classe agrega outra classe, a qual pode existir independentemente, como, por exemplo, as classes Matéria e Aluno (se a matéria for cancelada, os alunos continuam existindo)

Diagrama de classes

- É possível demonstrar relações entre classes UML através de **relacionamentos**, representados através de linhas conectando duas classes com pontas distintas
  - **Generalização** representa a ideia que uma classe herda *features* de uma classe pai, representando a ideia de herança em linguagens orientadas a objetos, como, por exemplo, as classes Cachorro e Mamífero (oposto: **especialização**)
  - **Associação** representa que duas classes são, semanticamente, associadas em um sistema, e trocam mensagens entre si, como, por exemplo, as classes Vôo e Avião
  - **Composição** representa que uma classe compõe outra classe que não existe independentemente, como, por exemplo, as classes Casa e Cômodo (um cômodo não existe sem uma casa)
  - **Agregação** representa que uma classe agrega outra classe, a qual pode existir independentemente, como, por exemplo, as classes Matéria e Aluno (se a matéria for cancelada, os alunos continuam existindo)

Diagrama de classes

- É possível demonstrar relações entre classes UML através de **relacionamentos**, representados através de linhas conectando duas classes com pontas distintas
  - **Generalização** representa a ideia que uma classe herda *features* de uma classe pai, representando a ideia de herança em linguagens orientadas a objetos, como, por exemplo, as classes Cachorro e Mamífero (oposto: **especialização**)
  - **Associação** representa que duas classes são, semanticamente, associadas em um sistema, e trocam mensagens entre si, como, por exemplo, as classes Vôo e Avião
  - **Composição** representa que uma classe compõe outra classe que não existe independentemente, como, por exemplo, as classes Casa e Cômodo (um cômodo não existe sem uma casa)
  - **Agregação** representa que uma classe agrega outra classe, a qual pode existir independentemente, como, por exemplo, as classes Matéria e Aluno (se a matéria for cancelada, os alunos continuam existindo)

Diagrama de classes

- É possível demonstrar relações entre classes UML através de **relacionamentos**, representados através de linhas conectando duas classes com pontas distintas
  - **Generalização** representa a ideia que uma classe herda *features* de uma classe pai, representando a ideia de herança em linguagens orientadas a objetos, como, por exemplo, as classes Cachorro e Mamífero (oposto: **especialização**)
  - **Associação** representa que duas classes são, semanticamente, associadas em um sistema, e trocam mensagens entre si, como, por exemplo, as classes Vôo e Avião
  - **Composição** representa que uma classe compõe outra classe que não existe independentemente, como, por exemplo, as classes Casa e Cômodo (um cômodo não existe sem uma casa)
  - **Agregação** representa que uma classe agrega outra classe, a qual pode existir independentemente, como, por exemplo, as classes Matéria e Aluno (se a matéria for cancelada, os alunos continuam existindo)

Diagrama de classes (geral)

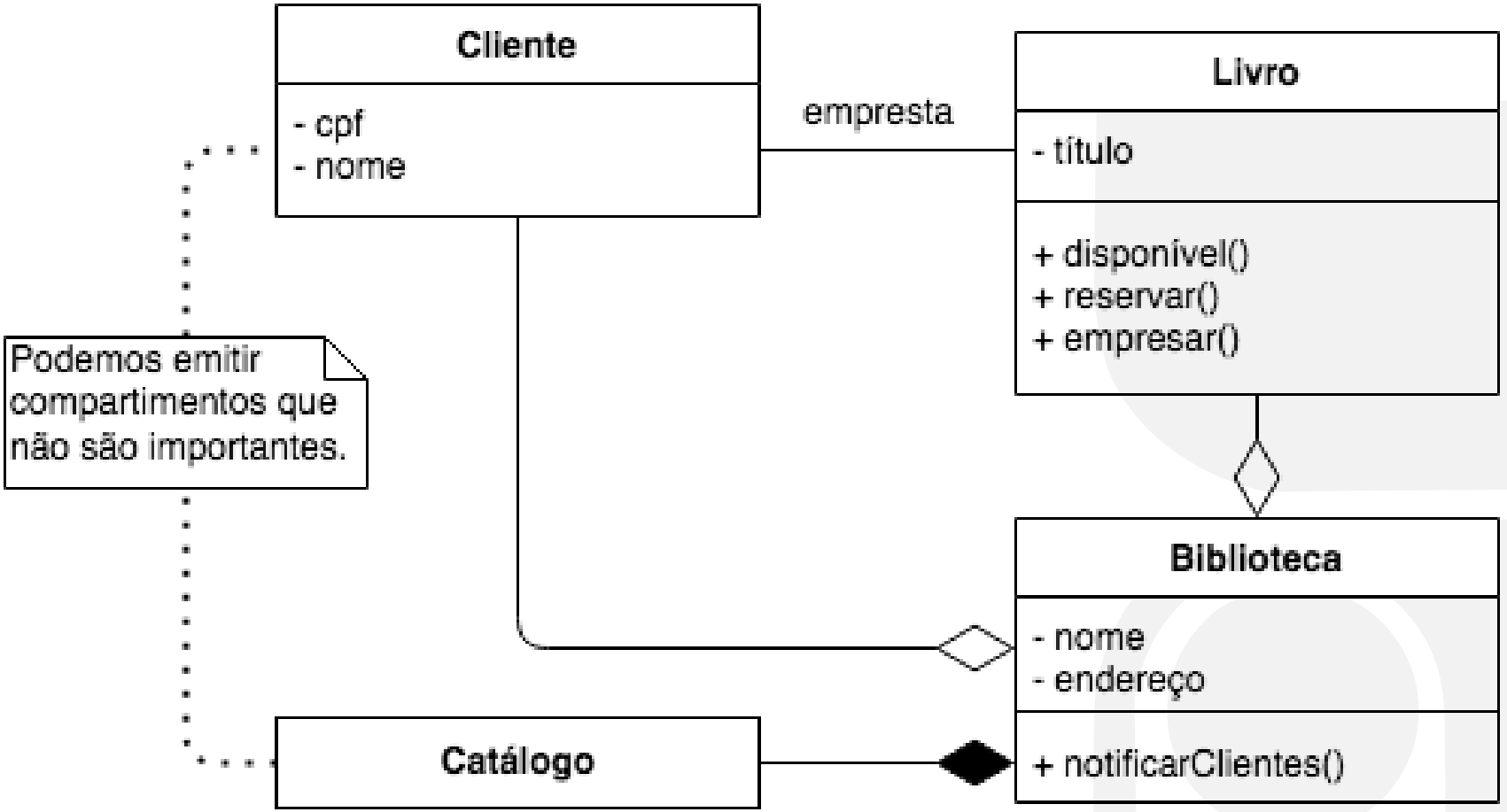


Diagrama de classes (generalização)

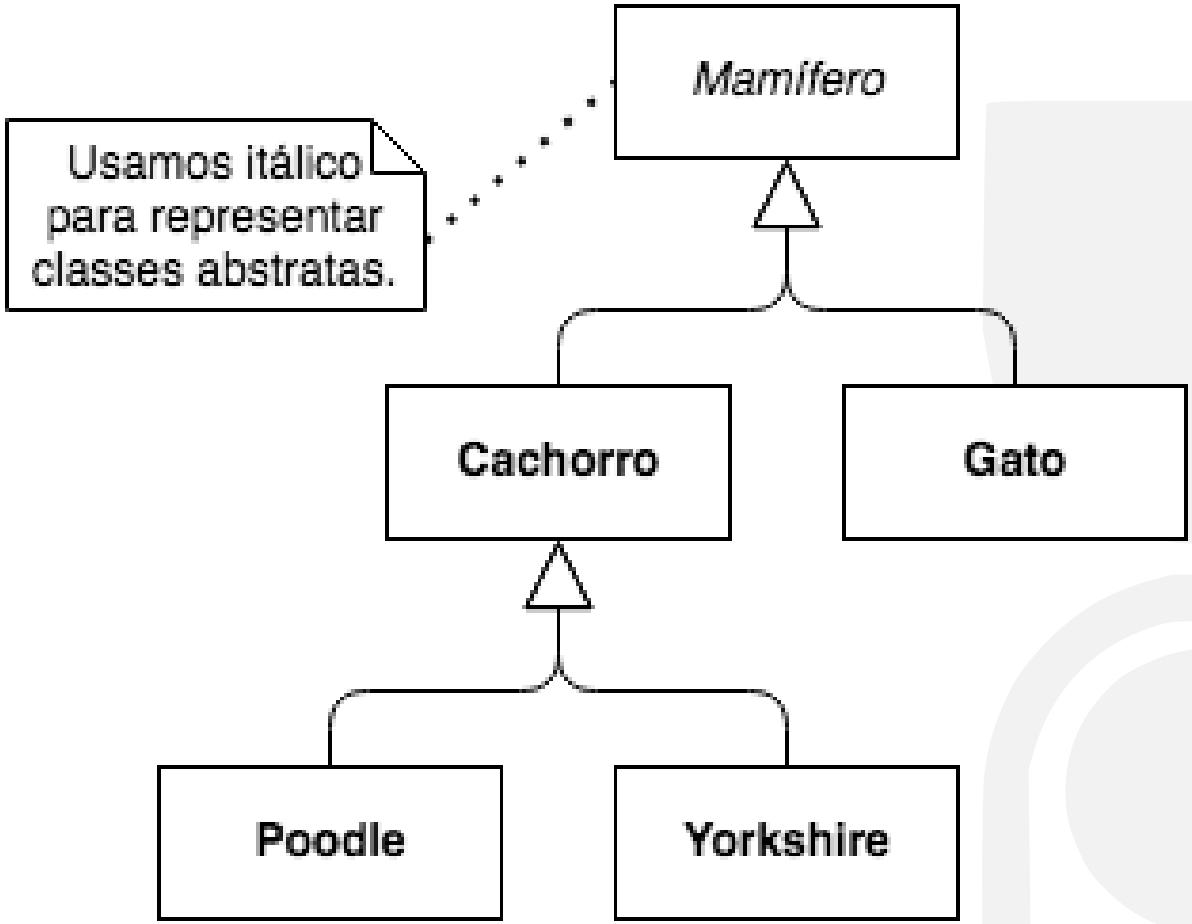


Diagrama de classes (composição)

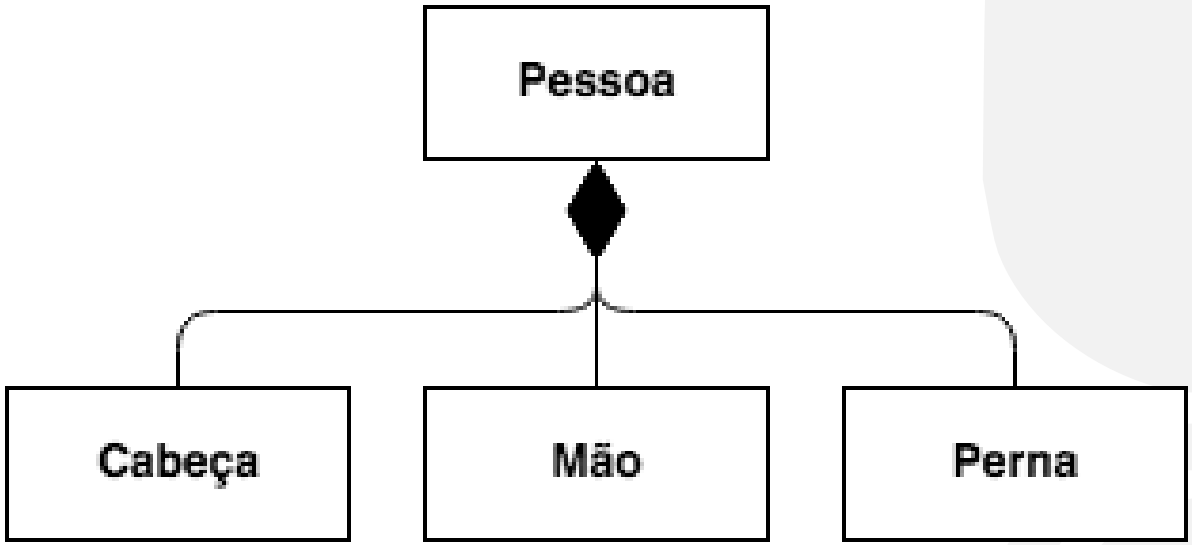


Diagrama de classes (agregação)

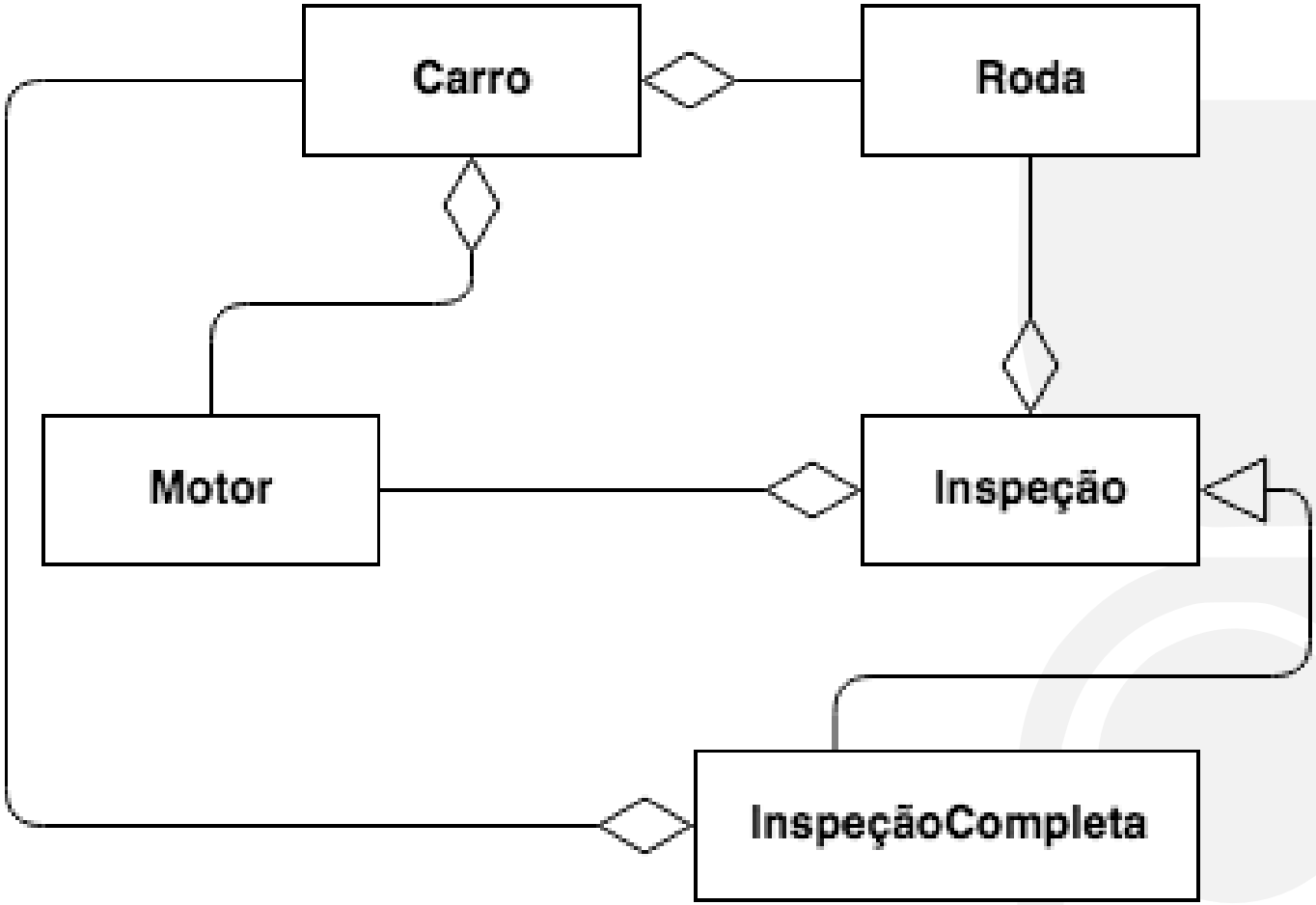


Diagrama de atividade

- O **diagrama de atividades** representa uma sequência (ou melhor dizendo, um fluxo) de **ações** necessárias para se completar uma tarefa durante a execução de um sistema
- `www.uml-diagrams.org/activity-diagrams.html`
- Diagramas de atividade representam uma das principais formas de se representar **comportamento** dentro de diagramas UML, junto ao diagrama de máquina de estado e de sequência
- Atividades são representadas dentro de um retângulo com pontas arredondadas, com o nome da atividade em negrito no campo superior esquerdo (o "nome da função")
- Atividades são essencialmente **assíncronas**, possuindo um **estado inicial**, representado por um círculo preenchido
- Ações **fluem** de umas para as outras até o **estado final**, que é representado por um círculo preenchido dentro de outro círculo

Diagrama de atividade

- O **diagrama de atividades** representa uma sequência (ou melhor dizendo, um fluxo) de **ações** necessárias para se completar uma tarefa durante a execução de um sistema
- `www.uml-diagrams.org/activity-diagrams.html`
- Diagramas de atividade representam uma das principais formas de se representar **comportamento** dentro de diagramas UML, junto ao diagrama de máquina de estado e de sequência
- Atividades são representadas dentro de um retângulo com pontas arredondadas, com o nome da atividade em negrito no campo superior esquerdo (o "nome da função")
- Atividades são essencialmente **assíncronas**, possuindo um **estado inicial**, representado por um círculo preenchido
- Ações **fluem** de umas para as outras até o **estado final**, que é representado por um círculo preenchido dentro de outro círculo

Diagrama de atividade

- O **diagrama de atividades** representa uma sequência (ou melhor dizendo, um fluxo) de **ações** necessárias para se completar uma tarefa durante a execução de um sistema
- `www.uml-diagrams.org/activity-diagrams.html`
- Diagramas de atividade representam uma das principais formas de se representar **comportamento** dentro de diagramas UML, junto ao diagrama de máquina de estado e de sequência
- Atividades são representadas dentro de um retângulo com pontas arredondadas, com o nome da atividade em negrito no campo superior esquerdo (o "nome da função")
- Atividades são essencialmente **assíncronas**, possuindo um **estado inicial**, representado por um círculo preenchido
- Ações **fluem** de umas para as outras até o **estado final**, que é representado por um círculo preenchido dentro de outro círculo

Diagrama de atividade

- O **diagrama de atividades** representa uma sequência (ou melhor dizendo, um fluxo) de **ações** necessárias para se completar uma tarefa durante a execução de um sistema
- `www.uml-diagrams.org/activity-diagrams.html`
- Diagramas de atividade representam uma das principais formas de se representar **comportamento** dentro de diagramas UML, junto ao diagrama de máquina de estado e de sequência
- Atividades são representadas dentro de um retângulo com pontas arredondadas, com o nome da atividade em negrito no campo superior esquerdo (o "nome da função")
- Atividades são essencialmente **assíncronas**, possuindo um **estado inicial**, representado por um círculo preenchido
- Ações **fluem** de umas para as outras até o **estado final**, que é representado por um círculo preenchido dentro de outro círculo

Diagrama de atividade

- O **diagrama de atividades** representa uma sequência (ou melhor dizendo, um fluxo) de **ações** necessárias para se completar uma tarefa durante a execução de um sistema
- `www.uml-diagrams.org/activity-diagrams.html`
- Diagramas de atividade representam uma das principais formas de se representar **comportamento** dentro de diagramas UML, junto ao diagrama de máquina de estado e de sequência
- Atividades são representadas dentro de um retângulo com pontas arredondadas, com o nome da atividade em negrito no campo superior esquerdo (o "nome da função")
- Atividades são essencialmente **assíncronas**, possuindo um **estado inicial**, representado por um círculo preenchido
- Ações **fluem** de umas para as outras até o **estado final**, que é representado por um círculo preenchido dentro de outro círculo

Diagrama de atividade

- O **diagrama de atividades** representa uma sequência (ou melhor dizendo, um fluxo) de **ações** necessárias para se completar uma tarefa durante a execução de um sistema
- `www.uml-diagrams.org/activity-diagrams.html`
- Diagramas de atividade representam uma das principais formas de se representar **comportamento** dentro de diagramas UML, junto ao diagrama de máquina de estado e de sequência
- Atividades são representadas dentro de um retângulo com pontas arredondadas, com o nome da atividade em negrito no campo superior esquerdo (o "nome da função")
- Atividades são essencialmente **assíncronas**, possuindo um **estado inicial**, representado por um círculo preenchido
- Ações **fluem** de umas para as outras até o **estado final**, que é representado por um círculo preenchido dentro de outro círculo

Diagrama de atividade

- Estados intermediários, representados em retângulos de bordas arredondadas, são usados para invocar ações dentro do fluxo
- Como a ideia é que informações fluam entre estados, usamos losangos para representar pontos de **decisão** (*decision*) e **união** (*merge*) de informações
  - As setas partindo de um ponto de decisão apresentam **condições** (*guards*), representados entre colchetes sobre elas
- Com o fim de se representar diagramas de forma genérica, é possível representar pontos de **divisão** (*fork*) e de **junção** (*join*), utilizando linhas fortes, marcando, respectivamente, o início e o término de fluxos concorrentes

Diagrama de atividade

- Estados intermediários, representados em retângulos de bordas arredondadas, são usados para invocar ações dentro do fluxo
- Como a ideia é que informações fluam entre estados, usamos losangos para representar pontos de **decisão** (*decision)* e **união** (*merge*) de informações
  - As setas partindo de um ponto de decisão apresentam **condições** (*guards*), representados entre colchetes sobre elas
- Com o fim de se representar diagramas de forma genérica, é possível representar pontos de **divisão** (*fork*) e de **junção** (*join*), utilizando linhas fortes, marcando, respectivamente, o início e o término de fluxos concorrentes

Diagrama de atividade

- Estados intermediários, representados em retângulos de bordas arredondadas, são usados para invocar ações dentro do fluxo
- Como a ideia é que informações fluam entre estados, usamos losangos para representar pontos de **decisão** (*decision)* e **união** (*merge*) de informações
  - As setas partindo de um ponto de decisão apresentam **condições** (*guards*), representados entre colchetes sobre elas
- Com o fim de se representar diagramas de forma genérica, é possível representar pontos de **divisão** (*fork*) e de **junção** (*join*), utilizando linhas fortes, marcando, respectivamente, o início e o término de fluxos concorrentes

Diagrama de atividade

- Estados intermediários, representados em retângulos de bordas arredondadas, são usados para invocar ações dentro do fluxo
- Como a ideia é que informações fluam entre estados, usamos losangos para representar pontos de **decisão** (*decision)* e **união** (*merge*) de informações
  - As setas partindo de um ponto de decisão apresentam **condições** (*guards*), representados entre colchetes sobre elas
- Com o fim de se representar diagramas de forma genérica, é possível representar pontos de **divisão** (*fork*) e de **junção** (*join*), utilizando linhas fortes, marcando, respectivamente, o início e o término de fluxos concorrentes

Diagrama de atividade

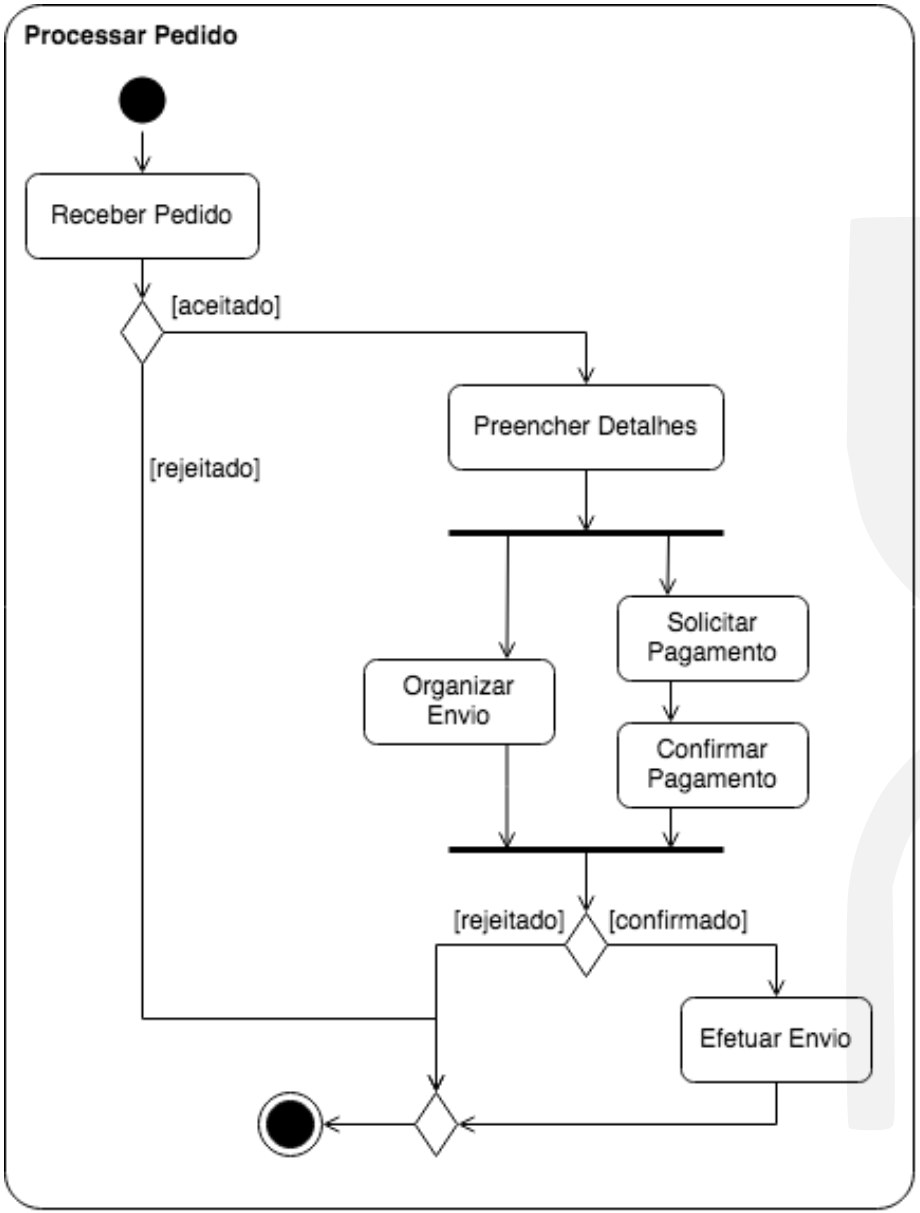


Diagrama de atividade

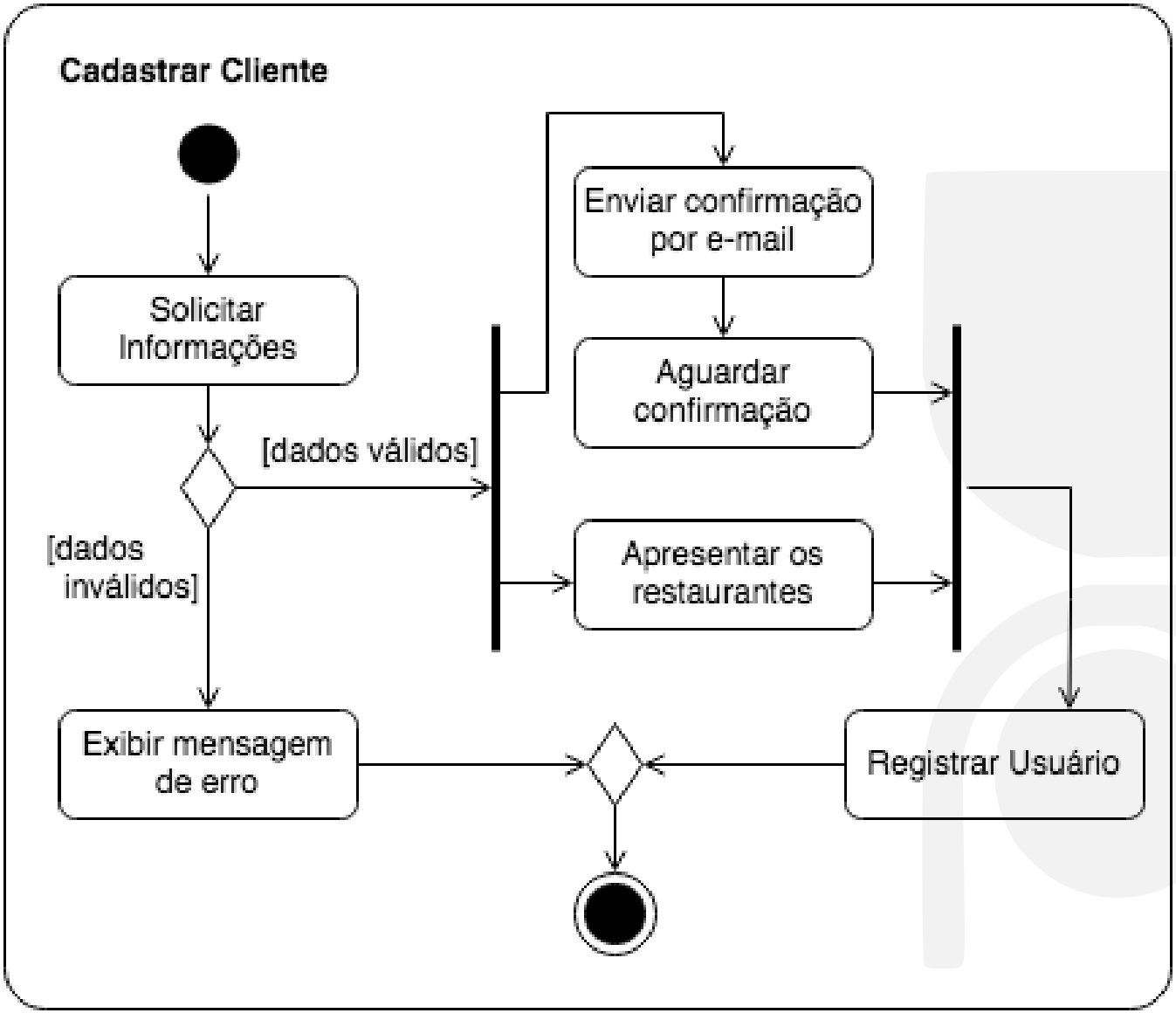


Diagrama de sequência

- O **diagrama de sequência**, similar ao diagrama de atividades, oferece uma forma de se descrever **comportamentos** e algoritmos em UML
- Diferente do diagrama de atividades, que focam no fluxo das informações, o diagrama de sequência foca na **interação** entre diferentes objetos (ou atores!) em um sistema
  - Objetos são instâncias de classes envolvidas em um comportamento
  - Um **ator** é um agente externo ao sistema, como um usuário ou outro sistema, capaz de interagir com o sistema, como solicitar o início de eventos
- Interações entre objetos são representadas por **envios de mensagens**, como, por exemplo, chamadas de métodos
- A ênfase é dada na **ordem** em que **ações** ocorrem, mostrando quais condições devem ser satisfeitas, ações disparadas, etc

Diagrama de sequência

- O **diagrama de sequência**, similar ao diagrama de atividades, oferece uma forma de se descrever **comportamentos** e algoritmos em UML
- Diferente do diagrama de atividades, que focam no fluxo das informações, o diagrama de sequência foca na **interação** entre diferentes objetos (ou atores!) em um sistema
  - Objetos são instâncias de classes envolvidas em um comportamento
  - Um **ator** é um agente externo ao sistema, como um usuário ou outro sistema, capaz de interagir com o sistema, como solicitar o início de eventos
- Interações entre objetos são representadas por **envios de mensagens**, como, por exemplo, chamadas de métodos
- A ênfase é dada na **ordem** em que **ações** ocorrem, mostrando quais condições devem ser satisfeitas, ações disparadas, etc

Diagrama de sequência

- O **diagrama de sequência**, similar ao diagrama de atividades, oferece uma forma de se descrever **comportamentos** e algoritmos em UML
- Diferente do diagrama de atividades, que focam no fluxo das informações, o diagrama de sequência foca na **interação** entre diferentes objetos (ou atores!) em um sistema
  - Objetos são instâncias de classes envolvidas em um comportamento
  - Um **ator** é um agente externo ao sistema, como um usuário ou outro sistema, capaz de interagir com o sistema, como solicitar o início de eventos
- Interações entre objetos são representadas por **envios de mensagens**, como, por exemplo, chamadas de métodos
- A ênfase é dada na **ordem** em que **ações** ocorrem, mostrando quais condições devem ser satisfeitas, ações disparadas, etc

Diagrama de sequência

- O **diagrama de sequência**, similar ao diagrama de atividades, oferece uma forma de se descrever **comportamentos** e algoritmos em UML
- Diferente do diagrama de atividades, que focam no fluxo das informações, o diagrama de sequência foca na **interação** entre diferentes objetos (ou atores!) em um sistema
  - Objetos são instâncias de classes envolvidas em um comportamento
  - Um **ator** é um agente externo ao sistema, como um usuário ou outro sistema, capaz de interagir com o sistema, como solicitar o início de eventos
- Interações entre objetos são representadas por **envios de mensagens**, como, por exemplo, chamadas de métodos
- A ênfase é dada na **ordem** em que **ações** ocorrem, mostrando quais condições devem ser satisfeitas, ações disparadas, etc

Diagrama de sequência

- O **diagrama de sequência**, similar ao diagrama de atividades, oferece uma forma de se descrever **comportamentos** e algoritmos em UML
- Diferente do diagrama de atividades, que focam no fluxo das informações, o diagrama de sequência foca na **interação** entre diferentes objetos (ou atores!) em um sistema
  - Objetos são instâncias de classes envolvidas em um comportamento
  - Um **ator** é um agente externo ao sistema, como um usuário ou outro sistema, capaz de interagir com o sistema, como solicitar o início de eventos
- Interações entre objetos são representadas por **envios de mensagens**, como, por exemplo, chamadas de métodos
- A ênfase é dada na **ordem** em que **ações** ocorrem, mostrando quais condições devem ser satisfeitas, ações disparadas, etc

Diagrama de sequência

- O **diagrama de sequência**, similar ao diagrama de atividades, oferece uma forma de se descrever **comportamentos** e algoritmos em UML
- Diferente do diagrama de atividades, que focam no fluxo das informações, o diagrama de sequência foca na **interação** entre diferentes objetos (ou atores!) em um sistema
  - Objetos são instâncias de classes envolvidas em um comportamento
  - Um **ator** é um agente externo ao sistema, como um usuário ou outro sistema, capaz de interagir com o sistema, como solicitar o início de eventos
- Interações entre objetos são representadas por **envios de mensagens**, como, por exemplo, chamadas de métodos
- A ênfase é dada na **ordem** em que **ações** ocorrem, mostrando quais condições devem ser satisfeitas, ações disparadas, etc

Diagrama de sequência

- Objetos ou atores possuem uma **linha de vida** representa por uma linha vertical tracejada partindo deles, que demonstra o tempo de existência do **participante**
- **Execuções** são representados por retângulos na linha de vida de um participante, e dizemos que um participante é **ativado** ao receber um estímulo esterno, representado por uma linha entre execuções
  - Acima da linha, que parte da execução ativa para servir de estímulo para ativar outra ação, descrevemos o sinal enviado, por exemplo, o método chamado
  - Uma linha tracejada pode **retornar** para representar uma mensagem de retorno, resultado da ação chamada, se desejada
- Desvios (*loops*, condicionais, etc) podem ser representados através de **fragmentos combinados**, seções demarcadas por um retângulo, possivelmente contendo condições (*guards*) entre colchetes, similar aos diagramas de atividade

Diagrama de sequência

- Objetos ou atores possuem uma **linha de vida** representa por uma linha vertical tracejada partindo deles, que demonstra o tempo de existência do **participante**
- **Execuções** são representados por retângulos na linha de vida de um participante, e dizemos que um participante é **ativado** ao receber um estímulo esterno, representado por uma linha entre execuções
  - Acima da linha, que parte da execução ativa para servir de estímulo para ativar outra ação, descrevemos o sinal enviado, por exemplo, o método chamado
  - Uma linha tracejada pode **retornar** para representar uma mensagem de retorno, resultado da ação chamada, se desejada
- Desvios (*loops*, condicionais, etc) podem ser representados através de **fragmentos combinados**, seções demarcadas por um retângulo, possivelmente contendo condições (*guards*) entre colchetes, similar aos diagramas de atividade

Diagrama de sequência

- Objetos ou atores possuem uma **linha de vida** representa por uma linha vertical tracejada partindo deles, que demonstra o tempo de existência do **participante**
- **Execuções** são representados por retângulos na linha de vida de um participante, e dizemos que um participante é **ativado** ao receber um estímulo esterno, representado por uma linha entre execuções
  - Acima da linha, que parte da execução ativa para servir de estímulo para ativar outra ação, descrevemos o sinal enviado, por exemplo, o método chamado
  - Uma linha tracejada pode **retornar** para representar uma mensagem de retorno, resultado da ação chamada, se desejada
- Desvios (*loops*, condicionais, etc) podem ser representados através de **fragmentos combinados**, seções demarcadas por um retângulo, possivelmente contendo condições (*guards*) entre colchetes, similar aos diagramas de atividade

Diagrama de sequência

- Objetos ou atores possuem uma **linha de vida** representa por uma linha vertical tracejada partindo deles, que demonstra o tempo de existência do **participante**
- **Execuções** são representados por retângulos na linha de vida de um participante, e dizemos que um participante é **ativado** ao receber um estímulo esterno, representado por uma linha entre execuções
  - Acima da linha, que parte da execução ativa para servir de estímulo para ativar outra ação, descrevemos o sinal enviado, por exemplo, o método chamado
  - Uma linha tracejada pode **retornar** para representar uma mensagem de retorno, resultado da ação chamada, se desejada
- Desvios (*loops*, condicionais, etc) podem ser representados através de **fragmentos combinados**, seções demarcadas por um retângulo, possivelmente contendo condições (*guards*) entre colchetes, similar aos diagramas de atividade

Diagrama de sequência

- Objetos ou atores possuem uma **linha de vida** representa por uma linha vertical tracejada partindo deles, que demonstra o tempo de existência do **participante**
- **Execuções** são representados por retângulos na linha de vida de um participante, e dizemos que um participante é **ativado** ao receber um estímulo esterno, representado por uma linha entre execuções
  - Acima da linha, que parte da execução ativa para servir de estímulo para ativar outra ação, descrevemos o sinal enviado, por exemplo, o método chamado
  - Uma linha tracejada pode **retornar** para representar uma mensagem de retorno, resultado da ação chamada, se desejada
- Desvios (*loops*, condicionais, etc) podem ser representados através de **fragmentos combinados**, seções demarcadas por um retângulo, possivelmente contendo condições (*guards*) entre colchetes, similar aos diagramas de atividade

Diagrama de sequência

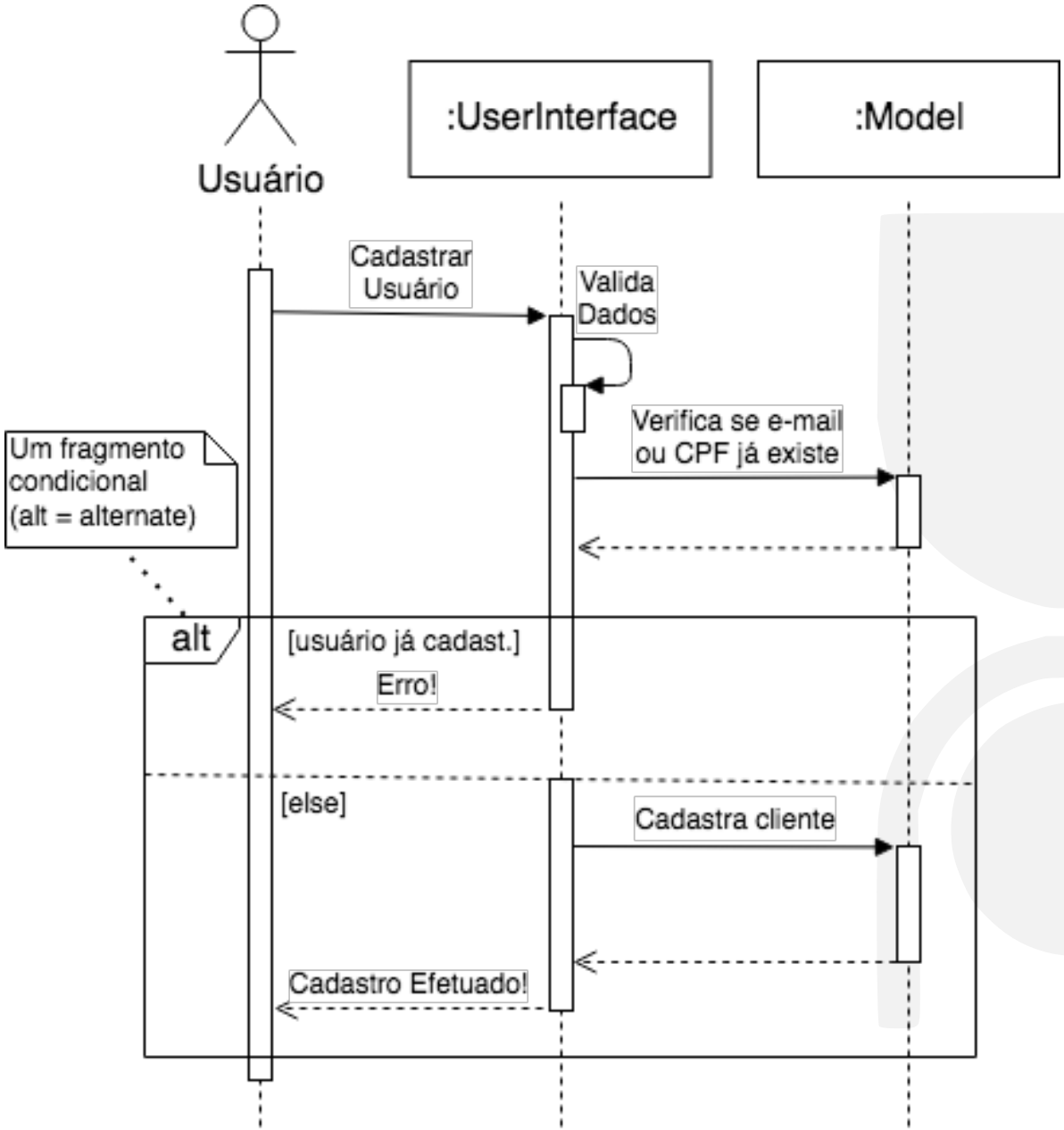
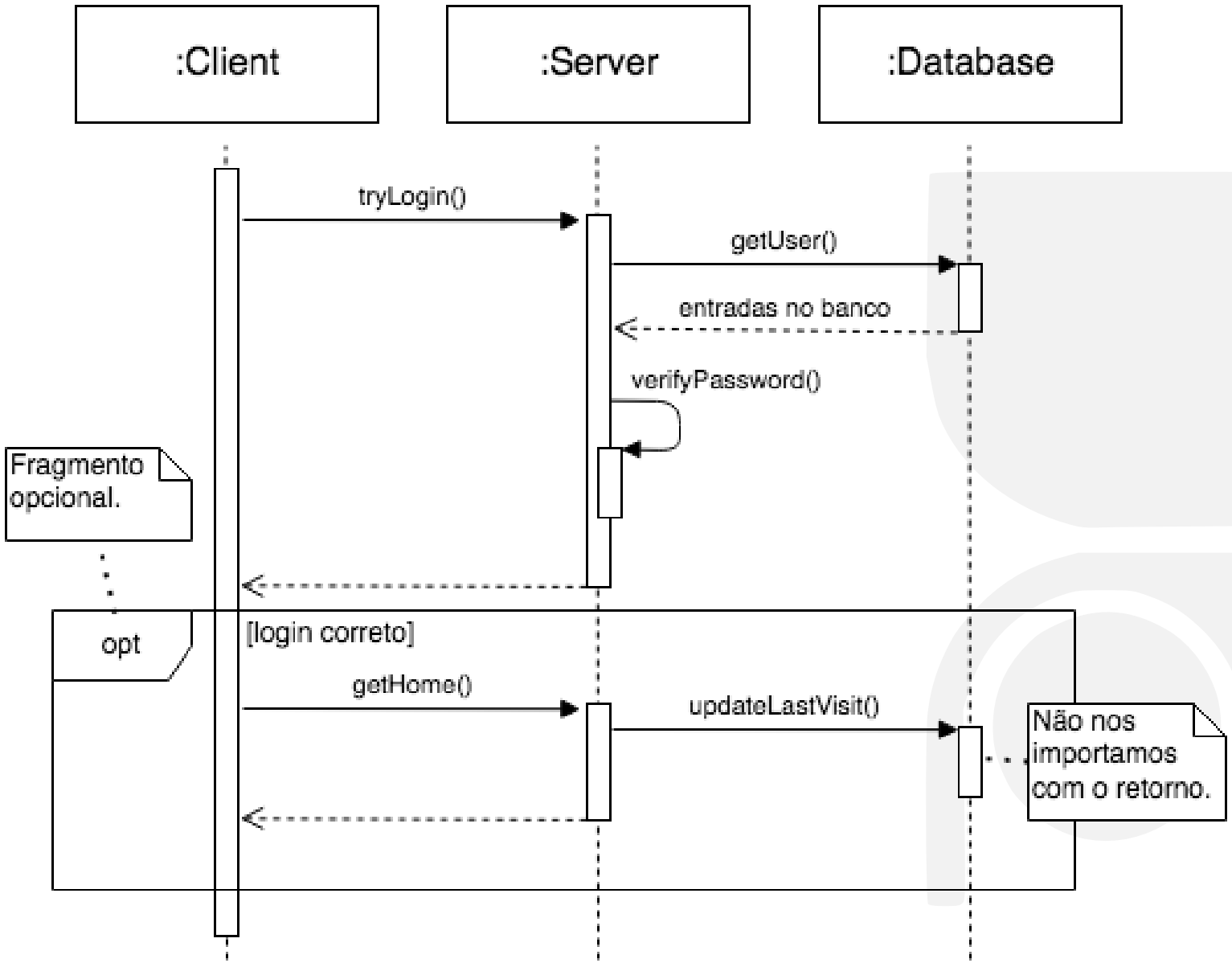


Diagrama de sequência




What the client says they want
What the [redacted] client actually needs

Orientação a objetos, recapitulação

- O principal objetivo do padrão UML é auxiliar na modelagem e especificação de projetos **orientados a objeto**
- Como há algumas variações em terminologia, iremos dar preferência para terminologia usada dentro do UML
- Dos principais conceitos usados na programação orientada a objetos, está, justamente, o conceito de **objeto**: uma estrutura que existe durante a execução de um programa, contendo um conjunto de **atributos** (estado) e **operações**
- A maioria das linguagens introduz o conceito de **classes**, que servem como especificações para objetos: ao dizer que um objeto é uma **instância** de uma classe, temos que o objeto adere à especificação fornecida pela classe
- Dos conceitos básicos, temos as ideias de abstração, encapsulamento, e polimorfismo

Orientação a objetos, recapitulação

- O principal objetivo do padrão UML é auxiliar na modelagem e especificação de projetos **orientados a objeto**
- Como há algumas variações em terminologia, iremos dar preferência para terminologia usada dentro do UML
- Dos principais conceitos usados na programação orientada a objetos, está, justamente, o conceito de **objeto**: uma estrutura que existe durante a execução de um programa, contendo um conjunto de **atributos** (estado) e **operações**
- A maioria das linguagens introduz o conceito de **classes**, que servem como especificações para objetos: ao dizer que um objeto é uma **instância** de uma classe, temos que o objeto adere à especificação fornecida pela classe
- Dos conceitos básicos, temos as ideias de abstração, encapsulamento, e polimorfismo

Orientação a objetos, recapitulação

- O principal objetivo do padrão UML é auxiliar na modelagem e especificação de projetos **orientados a objeto**
- Como há algumas variações em terminologia, iremos dar preferência para terminologia usada dentro do UML
- Dos principais conceitos usados na programação orientada a objetos, está, justamente, o conceito de **objeto**: uma estrutura que existe durante a execução de um programa, contendo um conjunto de **atributos** (estado) e **operações**
- A maioria das linguagens introduz o conceito de **classes**, que servem como especificações para objetos: ao dizer que um objeto é uma **instância** de uma classe, temos que o objeto adere à especificação fornecida pela classe
- Dos conceitos básicos, temos as ideias de abstração, encapsulamento, e polimorfismo

Orientação a objetos, recapitulação

- O principal objetivo do padrão UML é auxiliar na modelagem e especificação de projetos **orientados a objeto**
- Como há algumas variações em terminologia, iremos dar preferência para terminologia usada dentro do UML
- Dos principais conceitos usados na programação orientada a objetos, está, justamente, o conceito de **objeto**: uma estrutura que existe durante a execução de um programa, contendo um conjunto de **atributos** (estado) e **operações**
- A maioria das linguagens introduz o conceito de **classes**, que servem como especificações para objetos: ao dizer que um objeto é uma **instância** de uma classe, temos que o objeto adere à especificação fornecida pela classe
- Dos conceitos básicos, temos as ideias de abstração, encapsulamento, e polimorfismo

Orientação a objetos, recapitulação

- O principal objetivo do padrão UML é auxiliar na modelagem e especificação de projetos **orientados a objeto**
- Como há algumas variações em terminologia, iremos dar preferência para terminologia usada dentro do UML
- Dos principais conceitos usados na programação orientada a objetos, está, justamente, o conceito de **objeto**: uma estrutura que existe durante a execução de um programa, contendo um conjunto de **atributos** (estado) e **operações**
- A maioria das linguagens introduz o conceito de **classes**, que servem como especificações para objetos: ao dizer que um objeto é uma **instância** de uma classe, temos que o objeto adere à especificação fornecida pela classe
- Dos conceitos básicos, temos as ideias de abstração, encapsulamento, e polimorfismo

Orientação a objetos, recapitulação

- Atributos são **encapsulados** dentro de uma classe, isto é, dados são contidos dentro de um objeto e (geralmente) não são acessíveis diretamente (ou ao menos não deveriam)
- As noções de relacionamento usadas para o diagrama de classes no UML representam formas de se compor *features* da especificação de uma classe
- Dentre eles, por exemplo, podemos destacar o conceito de especialização, mais popularmente chamado de **herança**: ao definirmos uma classe filha B que herda de uma classe pai A, temos que os atributos e operações definidas em A também estarão definidas em B (relação *is-a*)
- Tal propriedade é uma das principais formas de **polimorfismo** em linguagens orientados a objetos: se um objeto é uma instância de B, e a classe B é uma especialização da classe A, temos que o objeto também é uma instância de A

Orientação a objetos, recapitulação

- Atributos são **encapsulados** dentro de uma classe, isto é, dados são contidos dentro de um objeto e (geralmente) não são acessíveis diretamente (ou ao menos não deveriam)
- As noções de relacionamento usadas para o diagrama de classes no UML representam formas de se compor *features* da especificação de uma classe
- Dentre eles, por exemplo, podemos destacar o conceito de especialização, mais popularmente chamado de **herança**: ao definirmos uma classe filha B que herda de uma classe pai A, temos que os atributos e operações definidas em A também estarão definidas em B (relação *is-a*)
- Tal propriedade é uma das principais formas de **polimorfismo** em linguagens orientados a objetos: se um objeto é uma instância de B, e a classe B é uma especialização da classe A, temos que o objeto também é uma instância de A

Orientação a objetos, recapitulação

- Atributos são **encapsulados** dentro de uma classe, isto é, dados são contidos dentro de um objeto e (geralmente) não são acessíveis diretamente (ou ao menos não deveriam)
- As noções de relacionamento usadas para o diagrama de classes no UML representam formas de se compor *features* da especificação de uma classe
- Dentre eles, por exemplo, podemos destacar o conceito de especialização, mais popularmente chamado de **herança**: ao definirmos uma classe filha B que herda de uma classe pai A, temos que os atributos e operações definidas em A também estarão definidas em B (relação *is-a*)
- Tal propriedade é uma das principais formas de **polimorfismo** em linguagens orientados a objetos: se um objeto é uma instância de B, e a classe B é uma especialização da classe A, temos que o objeto também é uma instância de A

Orientação a objetos, recapitulação

- Atributos são **encapsulados** dentro de uma classe, isto é, dados são contidos dentro de um objeto e (geralmente) não são acessíveis diretamente (ou ao menos não deveriam)
- As noções de relacionamento usadas para o diagrama de classes no UML representam formas de se compor *features* da especificação de uma classe
- Dentre eles, por exemplo, podemos destacar o conceito de especialização, mais popularmente chamado de **herança**: ao definirmos uma classe filha B que herda de uma classe pai A, temos que os atributos e operações definidas em A também estarão definidas em B (relação *is-a*)
- Tal propriedade é uma das principais formas de **polimorfismo** em linguagens orientados a objetos: se um objeto é uma instância de B, e a classe B é uma especialização da classe A, temos que o objeto também é uma instância de A

Orientação a objetos, recapitulação

- Linguagens costumam nos dar liberdade de projetarmos e escrevermos código da forma que decidirmos, mas existem **boas práticas** que nos guiam para projetarmos sistemas
- Uma das *guidelines* mais populares para o desenvolvimento de sistemas orientados a objetos são os princípios **SOLID**:
  - *Single-responsability principle*: classes devem executar tarefas bem especificadas, tendo uma única responsabilidade
  - *Open–closed principle*: entidades devem estar abertas para extensão, mas fechadas para modificações (do código fonte)
  - *Liskov substitution principle*: objetos em um sistema podem ser substituídos por objetos de um subtipo sem alterar a corretude de um programa
  - *Interface segregation principle*: sistemas devem dar preferência para a separação de interfaces, ao invés de uní-las em uma única grande especificação
  - *Dependency inversion principle*: código deve ser feito se baseado nas formas mais abstratas possíveis, ao invés de nas mais concretas

Orientação a objetos, recapitulação

- Linguagens costumam nos dar liberdade de projetarmos e escrevermos código da forma que decidirmos, mas existem **boas práticas** que nos guiam para projetarmos sistemas
- Uma das *guidelines* mais populares para o desenvolvimento de sistemas orientados a objetos são os princípios **SOLID**:
  - *Single-responsability principle*: classes devem executar tarefas bem especificadas, tendo uma única responsabilidade
  - *Open–closed principle*: entidades devem estar abertas para extensão, mas fechadas para modificações (do código fonte)
  - *Liskov substitution principle*: objetos em um sistema podem ser substituídos por objetos de um subtipo sem alterar a corretude de um programa
  - *Interface segregation principle*: sistemas devem dar preferência para a separação de interfaces, ao invés de uní-las em uma única grande especificação
  - *Dependency inversion principle*: código deve ser feito se baseado nas formas mais abstratas possíveis, ao invés de nas mais concretas

Orientação a objetos, recapitulação

- Linguagens costumam nos dar liberdade de projetarmos e escrevermos código da forma que decidirmos, mas existem **boas práticas** que nos guiam para projetarmos sistemas
- Uma das *guidelines* mais populares para o desenvolvimento de sistemas orientados a objetos são os princípios **SOLID**:
  - *Single-responsability principle*: classes devem executar tarefas bem especificadas, tendo uma única responsabilidade
  - *Open–closed principle*: entidades devem estar abertas para extensão, mas fechadas para modificações (do código fonte)
  - *Liskov substitution principle*: objetos em um sistema podem ser substituídos por objetos de um subtipo sem alterar a corretude de um programa
  - *Interface segregation principle*: sistemas devem dar preferência para a separação de interfaces, ao invés de uní-las em uma única grande especificação
  - *Dependency inversion principle*: código deve ser feito se baseado nas formas mais abstratas possíveis, ao invés de nas mais concretas

Orientação a objetos, recapitulação

- Linguagens costumam nos dar liberdade de projetarmos e escrevermos código da forma que decidirmos, mas existem **boas práticas** que nos guiam para projetarmos sistemas
- Uma das *guidelines* mais populares para o desenvolvimento de sistemas orientados a objetos são os princípios **SOLID**:
  - *Single-responsability principle*: classes devem executar tarefas bem especificadas, tendo uma única responsabilidade
  - *Open–closed principle*: entidades devem estar abertas para extensão, mas fechadas para modificações (do código fonte)
  - *Liskov substitution principle*: objetos em um sistema podem ser substituídos por objetos de um subtipo sem alterar a corretude de um programa
  - *Interface segregation principle*: sistemas devem dar preferência para a separação de interfaces, ao invés de uní-las em uma única grande especificação
  - *Dependency inversion principle*: código deve ser feito se baseado nas formas mais abstratas possíveis, ao invés de nas mais concretas

Orientação a objetos, recapitulação

- Linguagens costumam nos dar liberdade de projetarmos e escrevermos código da forma que decidirmos, mas existem **boas práticas** que nos guiam para projetarmos sistemas
- Uma das *guidelines* mais populares para o desenvolvimento de sistemas orientados a objetos são os princípios **SOLID**:
  - *Single-responsability principle*: classes devem executar tarefas bem especificadas, tendo uma única responsabilidade
  - *Open–closed principle*: entidades devem estar abertas para extensão, mas fechadas para modificações (do código fonte)
  - *Liskov substitution principle*: objetos em um sistema podem ser substituídos por objetos de um subtipo sem alterar a corretude de um programa
  - *Interface segregation principle*: sistemas devem dar preferência para a separação de interfaces, ao invés de uní-las em uma única grande especificação
  - *Dependency inversion principle*: código deve ser feito se baseado nas formas mais abstratas possíveis, ao invés de nas mais concretas

Orientação a objetos, recapitulação

- Linguagens costumam nos dar liberdade de projetarmos e escrevermos código da forma que decidirmos, mas existem **boas práticas** que nos guiam para projetarmos sistemas
- Uma das *guidelines* mais populares para o desenvolvimento de sistemas orientados a objetos são os princípios **SOLID**:
  - *Single-responsability principle*: classes devem executar tarefas bem especificadas, tendo uma única responsabilidade
  - *Open–closed principle*: entidades devem estar abertas para extensão, mas fechadas para modificações (do código fonte)
  - *Liskov substitution principle*: objetos em um sistema podem ser substituídos por objetos de um subtipo sem alterar a corretude de um programa
  - *Interface segregation principle*: sistemas devem dar preferência para a separação de interfaces, ao invés de uní-las em uma única grande especificação
  - *Dependency inversion principle*: código deve ser feito se baseado nas formas mais abstratas possíveis, ao invés de nas mais concretas

Orientação a objetos, recapitulação

- Linguagens costumam nos dar liberdade de projetarmos e escrevermos código da forma que decidirmos, mas existem **boas práticas** que nos guiam para projetarmos sistemas
- Uma das *guidelines* mais populares para o desenvolvimento de sistemas orientados a objetos são os princípios **SOLID**:
  - *Single-responsability principle*: classes devem executar tarefas bem especificadas, tendo uma única responsabilidade
  - *Open–closed principle*: entidades devem estar abertas para extensão, mas fechadas para modificações (do código fonte)
  - *Liskov substitution principle*: objetos em um sistema podem ser substituídos por objetos de um subtipo sem alterar a corretude de um programa
  - *Interface segregation principle*: sistemas devem dar preferência para a separação de interfaces, ao invés de uní-las em uma única grande especificação
  - *Dependency inversion principle*: código deve ser feito se baseado nas formas mais abstratas possíveis, ao invés de nas mais concretas