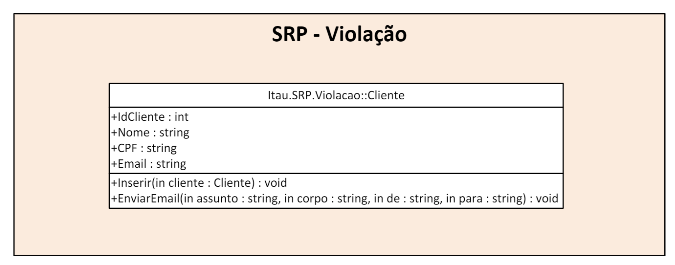
**Violando o princípio SRP (Single Responsability Principle)**

Conforme diagrama abaixo, temos a classe “Cliente” que viola o principio SRP, uma vez concentra múltiplas responsabilidades que não deveria tais como:

* Conter atributos que a categorizam como uma entidade.
* Conter um método para inserção de dados de cliente.
* Conter um método para envio de e-mail.



A violação do princípio SRP tem como consequências:

* **Falta de coesão:** A classe, está assumindo responsabilidades que não são suas.
* **Alto acoplamento:** Com mais responsabilidades, aumentam as dependências**.**
* **Baixa reutilização:** Devido ao alto acoplamento, gera-se dificuldades para reutilização dessas classes.

Conforme exemplos de código fonte abaixo, percebe-se que a classe Cliente possui atributos que a categorizam como uma entidade, possui um método de inserção de informações que gerencia conexões com o banco de dados e por fim, possui um método que transmite mensagens de e-mail.

Exemplo C#

[TODO]

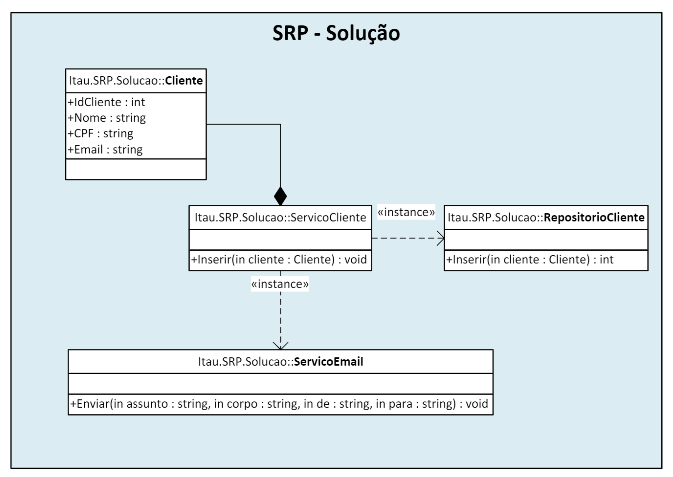
Exemplo Java

[TODO]

**Aplicando o princípio SRP (Single Responsability Principle)**

Conforme diagrama abaixo, percebe-se uma nova estrutura capaz de executar as mesmas ações, porém, com responsabilidades devidamente segregadas:

* **Classe Cliente**: Anteriormente concentrava todas as responsabilidades, agora é um “Model” que agrega a classe de serviço de clientes, e possui os atributos para transporte de dados e mapeamento com a tabela correspondente no banco de dados.
* **Classe ServicoCliente**: Possui a responsabilidade de conter as regras específicas de clientes, além de requisitar o repositório para acesso ao banco de dados e solicitar envio de mensagens de e-mail para classes também específicas.
* **Classe RepositorioCliente**: Possui a responsabilidade de gerenciar conexões como banco de dados e efetivar a inclusão dos dados pertinentes aos clientes.
* **Classe ServicoEmail**: Possui a responsabilidade de fazer a transmissão de mensagens de e-mail.



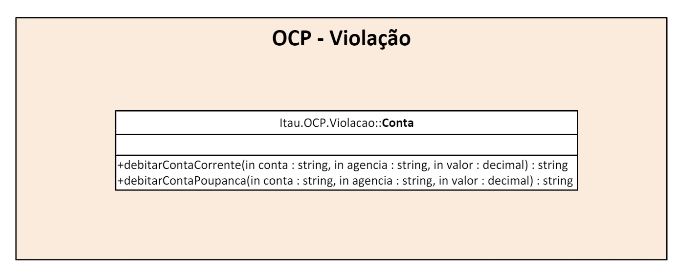
Conforme exemplos de código fonte abaixo, percebe-se que a aplicação da segregação de responsabilidades conforme cada uma das classes presentes no contexto:

Exemplo C#

Exemplo Java

**Violando o princípio OCP (Open Closed Principle)**

Conforme diagrama abaixo, temos a classe “Conta” que viola o princípio OCP uma vez que concentra métodos para débito nas contas do tipo corrente e poupança. Neste caso, temos também um sério problema de acoplamento, pois, caso seja necessária a implementação de um novo tipo de débito em conta de investimentos, teríamos que criar este novo método no escopo da classe, gerando mais dependência e a cada uma dessas evoluções, inviabilizará a manutenção.



Conforme exemplos de código fonte abaixo, percebe-se que a classe Conta, possui os métodos para debito em conta corrente e poupança, deixando bem claro que, para implementar um novo tipo de débito nessa estrutura, deverá ser realizada uma modificação.

Exemplo C#

[TODO]

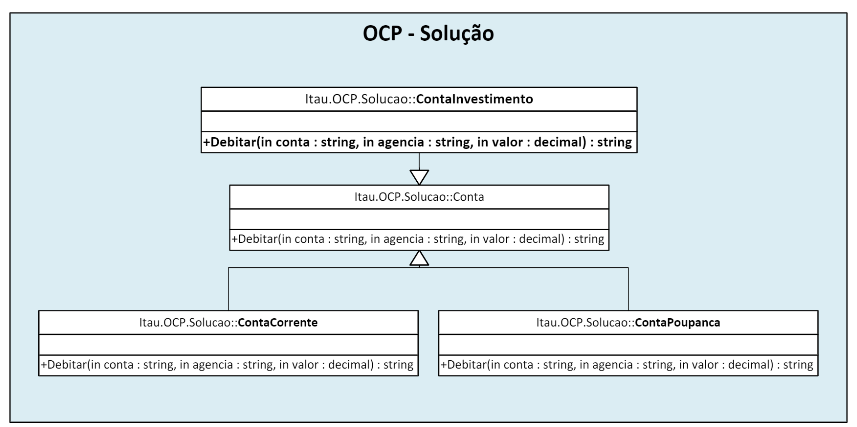
Exemplo Java

[TODO]

**Aplicando o princípio OCP (Open Closed Principle)**

Conforme diagrama abaixo, percebe-se uma nova estrutura preparada para extensão de novos comportamentos associados a débitos em contas, onde realizamos a implementação do débito em conta de investimento respeitando o princípio OCP, onde o método “Debitar” foi estendido.

* Classe Conta: Abstração agnóstica aos tipos de conta para débito que uma instituição financeira pode possuir, com um método “Debitar” para ser subscrito por classes ancestrais.
* Classe ContaCorrente: Classe específica para conta corrente que herdará as características da classe Conta e implementa o método “Debitar”, porém, aplicando débito em conta corrente.
* Classe ContaPoupanca: Classe específica para conta poupança que herdará as características classe Conta e implementa o método “Debitar”, porém, aplicando débito em conta poupança.
* Classe ContaInvestimento: Classe específica para conta investimento que herdará as características classe Conta e implementa o método “Debitar”, porém, aplicando débito em conta de investimentos.



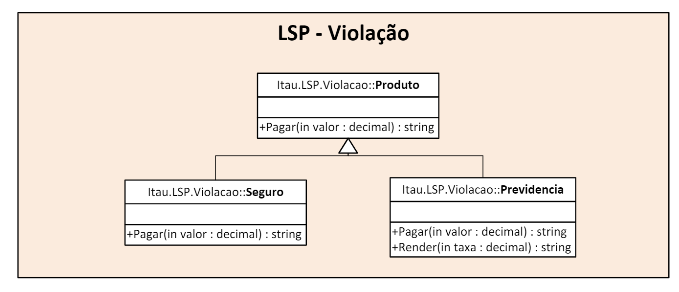
Conforme exemplos de código fonte abaixo, percebe-se que a nova estrutura hierárquica que viabiliza a extensão do método para realização de débitos conforme os tipos de conta a partir da herança das características da classe Conta.

Exemplo C#

Exemplo Java

**Violando o princípio LSP (Liskov Substitution Principle)**

Conforme diagrama abaixo, temos a classe “Previdencia” que viola o princípio LSP uma vez diferente da classe “Seguro”, possui um método que faz sentido para o negócio, porém, prejudica a estrutura, dado que a necessidade de render faz com que tenha um comportamento distinto da estrutura de produto, que determina que apenas pagam e não rendem.



Conforme exemplos de código fonte abaixo, percebe-se que a classe Previdencia, possui o método “Render” que fere a estrutura, não permitindo que substitua fielmente a classe Produto.

Exemplo C#

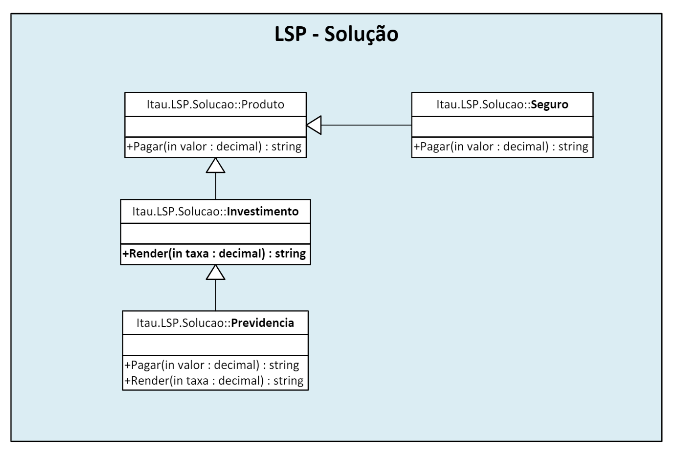
[TODO]

Exemplo Java

[TODO]

**Aplicando o princípio LSP (Liskov Substitution Principle)**

Conforme diagrama abaixo, percebe-se que para respeitar o principio LSP, foi gerada uma classe abstrata “Investimento”, que herdará a classe “Produto”, de forma que seja herdada pela classe “Previdencia” e tenha o método “Render” subscrito. Dessa forma, a classe “Previdencia” pode substituir de maneira fiel as classes ancestrais para rendimentos de previdência e pagamentos de produtos.



Conforme exemplos de código fonte abaixo, percebe-se que a nova estrutura permite que as tanto a sub classe para previdência, quanto a sub classe para produtos, podem substituir as classes ancestrais.

Exemplo C#

Exemplo Java