

9. Modelo de Projeto de Domínio

Introdução

Após a fase de análise, é necessário criar um modelo que represente a solução do problema da forma que deve ser implementado. O problema já foi especificado em forma de requisitos (por exemplo, casos de uso ou histórias de usuários) e diagrama de atividades. Além disso, foi criado um diagrama de classes de análises, que é uma visão dos principais conceitos que compõem o problema, e como estes se relacionam para representar as regras de negócio do problema.

O projeto do software, assim como o projeto de uma casa, é uma atividade complexa que envolve diversas etapas. Dentre estas etapas uma das mais importantes é definir o modelo de classes de projeto. Este modelo de classes define como será a estrutura de classes implementadas pelo sistema. Este artefato pode ser comparado a uma planta baixa em um projeto de uma casa.

O modelo de classes de projeto é constituído de várias camadas (o número de camadas depende da arquitetura do sistema), mas a camada principal é a camada de domínio, responsável por implementar toda a lógica do sistema (é uma evolução da camada de análise). Além da camada de domínio, outras camadas irão existir (camada de persistência, camada de interface, entre outras) no sistema. Para definir quais camadas irão compor o sistema é preciso estabelecer a arquitetura do sistema.

Este capítulo terá como foco descrever técnicas que auxiliem a criar o modelo de classes de projeto da camada de domínio, e em capítulos subsequentes serão abordados outros temas necessários para finalizar o projeto do software.

O diagrama de classes de análise foi o primeiro passo para transformar os requisitos em diagrama de classes. No entanto, este diagrama representa apenas as regras principais do domínio do problema, e precisa evoluir para caracterizar a solução final do problema, principalmente considerando as particularidades da orientação objetos. Assim, neste capítulo será especificado como explicitar o diagrama de classes de projeto da camada de domínio, tomando por base o diagrama de classes de análise e os requisitos do sistema.

Principais Diferenças Entre Diagrama de Análise e Projeto

O modelo de classes de projeto inicialmente é idêntico ao diagrama de classes de análise, uma vez que ele é uma

evolução do diagrama de análise. No entanto, quanto mais técnicas orientadas objetos utilizarmos, mais distintos estes modelos serão, já que, o modelo de projeto deve ser modelado da forma que pretendemos implementar o sistema. Assim, além de métodos, outras características serão acrescentadas no modelo de projeto, dentre elas, destacam-se:

- a) **Adição de métodos:** O diagrama de classes de análises é um diagrama que mostra as principais classes (ou conceitos) que descrevem o problema a ser resolvido, e como elas se comunicam (dá uma visão geral das regras de negócio do problema). No entanto, para implementar essas classes, devemos definir os seus métodos, que descrevem parte da responsabilidade das classes.
- b) **Adição de direção nas associações:** Até o presente momento foram-se identificadas as associações entre as classes existentes, contudo todas são bidirecionais. No entanto, na fase de projeto podem ser definidas as direções de navegação das associações, auxiliando a criar regras de negócios mais robustas.
- c) **Detalhamento de atributos:** No modelo de análise é comum atribuir apenas atributos que caracterizem as classes. Na fase de projeto, deve-se, contudo, descrever todos os atributos de uma classe (uma vez que deve ser similar ao código gerado).
- d) **Alteração da estrutura das classes:** No diagrama de projeto é comum que novas classes surjam, de forma a implementar estruturas do projeto. Assim, é comum que o diagrama de classes de projeto da camada de domínio não corresponda ao modelo de classes de análise.
- e) **Definição de regras de visibilidade aos métodos e atributos:** No modelo de projeto são definidas as regras de visibilidades aos métodos e atributos, pois é na fase de projeto que são definidas todas as regras principais do modelo, que devem ser respeitadas e mantidas na codificação.

Classes de Associação

As classes de associação são um importante instrumento para definir e descrever regras de negócios que ocorrem no domínio do problema. Contudo, a maioria das linguagens de programação não suportam este recurso. Dessa maneira, caso no modelo de análise existam classes de associações, essas devem ser convertidas no modelo de projeto.

A classe de associação representa uma associação que ocorre entre duas classes, e que esta associação em si,

possui propriedades. Um exemplo de classe de associação é mostrado na Figura 9.1, o qual existe uma classe **Funcao**, que representa a relação entre uma **Pessoa** e uma **Empresa**.

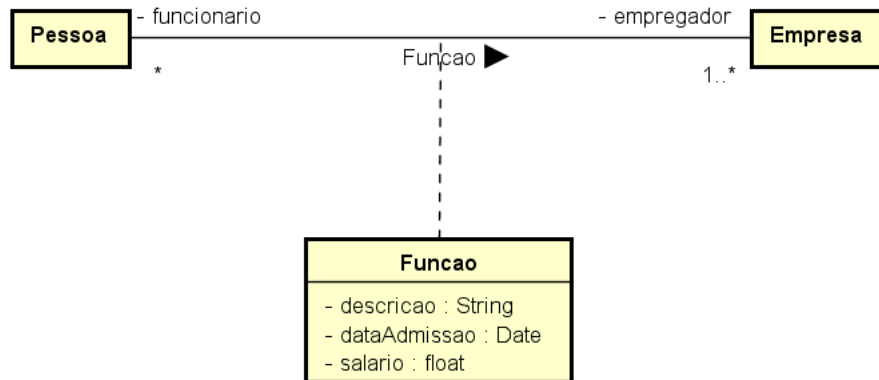


Figura 9.1: Exemplo de Classe de Associação

No diagrama de classes de projeto é prudente transformar uma classe de associação em associações entre classes, já que não existe este tipo de relacionamento na codificação, e, portanto, haveria uma discrepância entre o modelo e o código.

Realizar a transformação de um relacionamento de classe de associação é simples, uma vez que a classe de associação está relacionada com as duas outras classes. O único detalhe que deve ser observado é a multiplicidade, para no novo relacionamento mantê-la adequadamente. Por exemplo, na Figura 9.1, é lido que uma pessoa pode exercer uma ou diversas funções; e uma empresa possui várias funções.

Para definir a multiplicidade entre **Funcao** e **Pessoa**; e **Funcao** e **Empresa**, temos que ter em mente a finalidade que a classe de associação foi criada. A classe de associação foi criada para representar o papel específico que uma **Pessoa** pode exercer em uma **Empresa**. Portanto, os relacionamentos a partir de **Funcao** são sempre *um*, como pode ver na Figura 9.2.

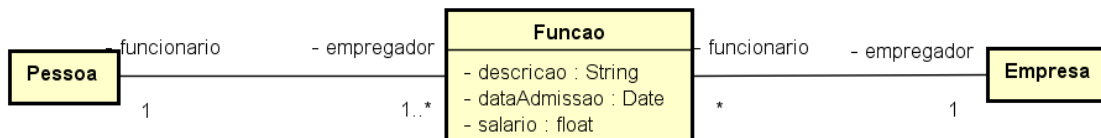


Figura 9.2: Transformando Classe de Associação em Associações

Na Figura 9.2, tem-se o modelo equivalente ao modelo de classes de associação (foram-se mantidos os papéis para melhor entendimento do modelo). Neste modelo, vemos que uma pessoa por exercer uma ou várias funções e uma empresa pode

ter várias funções. No entanto a **Funcao** (que é a relação específica entre a pessoa e a empresa) é única.

Delegação

O conceito de delegação, que já foi explorado, no capítulo 5 “Modelo de Classes de Análise”, e será explorado novamente neste capítulo, de forma mais contundente. A delegação é o “ato de conferir poder e representatividade para”. Na orientação objetos, a delegação é o ato de conceder a outro objeto a responsabilidade de executar o ou controlar a execução de determinada responsabilidade (por meio de mensagem).

A delegação é importante para alcançar a alta coesão, pois classes que não são responsáveis por determinada responsabilidade, delegam à outra classe essa responsabilidade. Além disso, a delegação é importante para se conseguir um baixo acoplamento, pois classes passam responsabilidades para a classe mais próxima, e essa, se não for responsável pela solicitação, delega a outra classe.

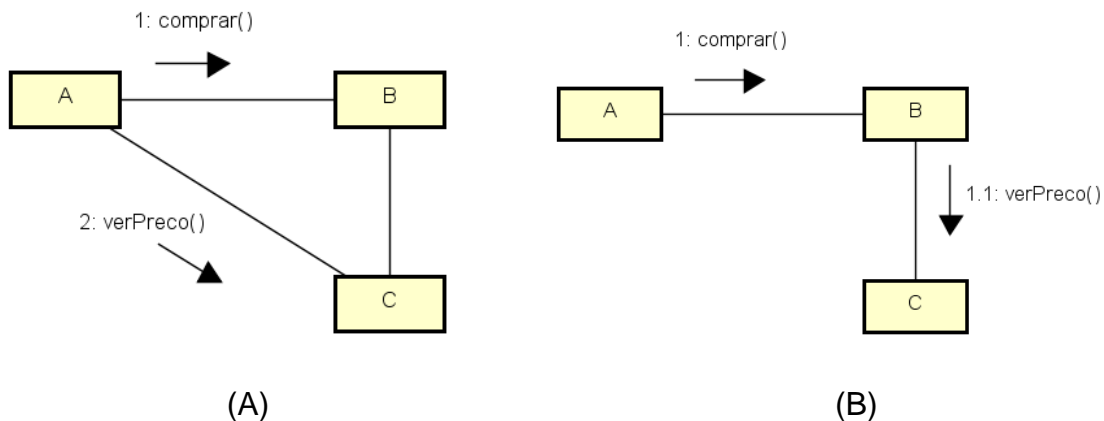


Figura 9.3: Transformando Classe de Associação em Associações

Na Figura 9.3, podemos ver a parte (A), o qual a classe A precisa se comunicar com duas classes para finalizar a sua execução (a classe **A** chama o método *comprar()* da classe **B** e o método *verPreco()* da classe **C**). Já no lado (B) vemos a mesma execução, no entanto a classe **B** tem a responsabilidade de ver o preço e a delega para a classe **C**. Com isso, diminuimos o acoplamento, pois a classe **A** se comunica diretamente apenas com a classe **B**, e essa delega a responsabilidade de *verPreco* para a classe **C**.

Diagrama de Comunicação

Na fase de projetos, uma das principais etapas é a adição de métodos nas classes. A adição de métodos deve levar em consideração a responsabilidades das classes e também os seus relacionamentos com outras classes. Assim, serão criados métodos que executam responsabilidade inerentes à própria classe, e também métodos que deleguem responsabilidades a outras classes, diminuindo o acoplamento e aumentando a coesão.

Para atribuir responsabilidade as classes (incluindo a adição de métodos) iremos utilizar os Padrões *Grasp* (*General Responsibility Assignment Software*). No entanto, para utilizar alguns dos padrões *Grasp*, temos que ter conhecimento do diagrama de comunicação. Este material apresenta de forma superficial o diagrama de comunicação, para informações mais detalhadas sobre este diagrama, consulte um livro específico sobre a UML.

O diagrama de comunicação é um tipo de diagrama comportamental da UML, mais especificamente um diagrama de interação. O objetivo dos diagramas de interação é mostrar a troca de mensagens em uma colaboração (um grupo de objetos que cooperam), para atingir um objetivo. Um Diagrama de Interação é composto por:

- Objetos
- Ligações
- Mensagens

Na UML existem quatros tipos de diagramas de interação, com diferentes propósitos:

- **Sequência:** enfatiza o ordenamento das mensagens trocadas entre os objetos.
- **Comunicação:** enfatiza a organização estrutural dos objetos que trocam mensagens.
- **Interação Geral:** definidos para visualizar o fluxo geral de controle, logo, não mostram em detalhes as mensagens trocadas pelos objetos.
- **Tempo:** descreve as mudanças no estado ou condição de um objeto de uma classe durante um tempo.

Os diagramas de comunicação e de sequência são semanticamente equivalentes (ou seja, pode-se representar a mesma informação por meio de diferentes diagramas). No entanto, eles são estruturalmente distintos, pois têm ênfases diferentes. Neste momento, o diagrama de comunicação é mais apropriado de ser utilizado, uma vez que sua ênfase

é na organização estrutural e é possível observar mais facilmente se as relações definidas no diagrama de classes estão sendo respeitadas. Como exemplo, a Figura 9.4 apresenta um diagrama de comunicação e os elementos que o compõe.

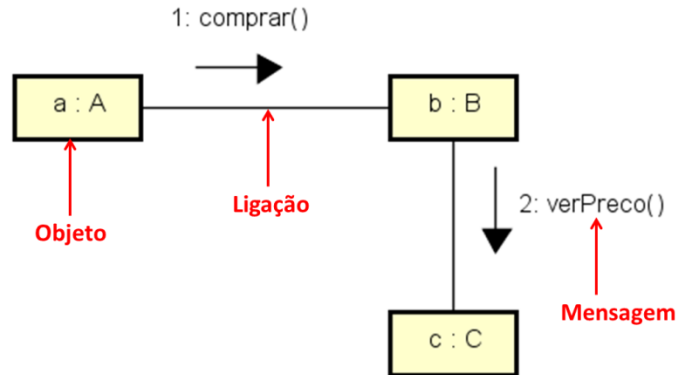


Figura 9.4: Exemplo de Diagrama de Comunicação

Pode-se observar que um diagrama de comunicação mostra explicitamente quais objetos se relacionam diretamente. Essas relações devem respeitar as associações definidas do diagrama de classes. Além disso, como estamos falando de um diagrama comportamental, este diagrama utiliza objetos ao invés de classes (pois estamos demonstrando como eles interagem em tempo de execução), e quais mensagens (métodos, *constructor*, *destructor*) são trocadas.

O diagrama de comunicação será utilizado para definir operações nas classes. Como pode ser visto na Figura 9.4, a direção de uma mensagem indica a classe que deve conter a operação que trata a mensagem correspondente (ex. objeto **a** está invocando o método *comprar()* do objeto **b**).

Por meio de um diagrama de comunicação consegue-se examinar:

- A troca de mensagens entre os objetos sob o ponto de vista temporal (mais facilmente visualizado no diagrama de sequência);
- As interações dos objetos dentro do contexto de suas relações estruturais, especificando as mensagens trocadas em função destas relações.

Além disso, como principais aplicações, pode-se destacar:

- Visualização, especificação, construção e documentação da dinâmica de uma sociedade particular de objetos.

- Podem ser usados para modelar o fluxo de controle de um caso de uso. No contexto de um caso de uso, uma interação representa um cenário.

O diagrama de comunicação, como mencionado anteriormente, enfatiza a organização estrutural dos objetos que participam de uma interação. Os objetos participantes da interação são colocados como se fossem vértices em um grafo, e as ligações que conectam os objetos são colocadas como se fossem os arcos de um grafo. As mensagens que os objetos enviam e recebem são colocadas de forma numerada junto a cada ligação ou arco. Como exemplo, a Figura 9.5 apresenta outro diagrama de comunicação.

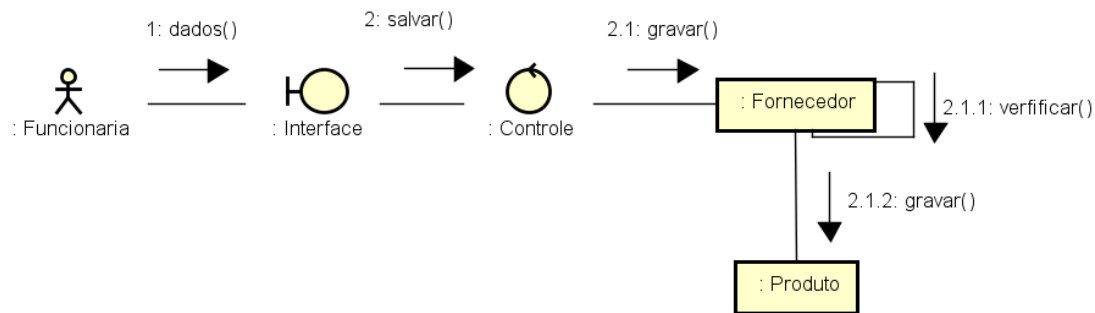


Figura 9.5: Outro Exemplo de Diagrama de Comunicação

No diagrama apresentado na Figura 9.5, percebe-se que estão sendo utilizadas classes estereotipadas. O ator **Funcionaria** insere dados na **Interface**, que por sua vez chama o método *salvar()* de **Controle**. Dentro do método *salvar()* é chamado o método *gravar()* de **Fornecedor**. Dentro do método *gravar()* de **Fornecedor** é chamado primeiro o método *verificar()* do próprio objeto **Fornecedor** e quando este finaliza sua execução, ainda dentro do método *gravar()* de **Fornecedor**, é chamado o método *gravar()* de **Produto**.

Contratos

O diagrama de classes é o principal artefato de um projeto OO. Este diagrama contempla todas as classes que compõem o sistema, assim como seus métodos e relações com outras classes. A criação deste diagrama não é elementar, pois se deve pensar nas responsabilidades de cada classe, assim como suas colaborações com outras classes. Assim, para planejar este diagrama é necessário entender as funcionalidades desejadas do sistema, identificar quais classes são

necessárias para implementar cada funcionalidade, suas responsabilidades e colaborações.

Contudo, utilizando-se modelos de desenvolvimento iterativos, normalmente o diagrama de classes é construído incrementalmente. Cada iteração é feita baseada em contratos, que são acordos que descrevem o que deve ser implementado.

Os contratos são definidos, de forma geral nos requisitos, e variam de acordo a técnica utilizada nos requisitos. Assim, conforme a metodologia de desenvolvimento utilizada, o método para identificação de responsabilidades das classes pode variar. Neste material será explorada identificação de responsabilidades das classes a partir de contratos definidos como casos de uso.

Por meio dos casos de usos, tem-se, portanto, como possíveis usuários trocam informações com o sistema, sem mostrar como a informação é processada internamente no sistema. Um caso de uso é um contrato firmado entre o analista e o *stakeholder*, que descreve uma funcionalidade desejada do software, e como ela deve se comportar, tendo ao menos quatro partes:

- a) Pré-condições: Estabelecem uma condição inicial para que uma determinada funcionalidade possa iniciar sua execução.
- b) Fluxo Principal: Descreve como a funcionalidade deve ser executada de forma a conseguir atingir a pós-condição.
- c) Exceções: Descreve exceções que possam ocorrer na execução do fluxo principal, assim como seus possíveis tratamentos e finalização da execução.
- d) Pós-Condições: Estabelecem o que acontece com as informações manipuladas durante a execução da funcionalidade, e estabelecem informações que devem ser informadas aos atores que interagem com os casos de uso.

Como exemplo de um contrato, vamos analisar o modelo de caso de uso de *Finalizar Pedido* do sistema de *Compra Online*. Analisando o Quadro 9.1, tem-se um dos contratos definidos para implementar o sistema de comércio *online*. Alguns autores definem que um caso de uso contém vários contratos, considerando que cada fluxo é um contrato. No entanto, neste material é considerado que um caso de uso é um contrato, e cada fluxo alternativo é um subcontrato. Assim, neste contrato fica estabelecido que:

- a) Pré-condições: Esta funcionalidade só é acessada por autenticação por meio de *login*.
- b) Fluxo Principal: O sistema deve seguir as regras definidas nos passos descritos no caso de uso para realizar um Pedido.
- c) Exceções: São descritos todos possíveis passos alternativos (subcontratos), problemas e falhas que podem ocorrer ao longo de um pedido, assim como suas execuções e finalizações.
- d) Pós-Condições: Estabelece que o sistema deve emitir e gravar todas as informações geradas no pedido e atualizar o *status* do pedido para aguardando pagamento.

Quadro 9.1 –Caso de Uso Finalizar Pedido

Nome: Finalizar Pedido	
Ator: Usuário	
Pré-Condição: O usuário deve ter feito "log-in" e obtido autorização do sistema	
Fluxo Principal	Fluxos Alternativos
Fluxo de Principal (caminho básico):	3.a Produto em Promoção
12. O usuário acesso sua cesta de compra.	3.a.1 (Ponto de extensão – Calcular Desconto de Produto em Promoção)
13. O sistema mostra as descrições, quantidade e o preço de cada produto.	3.a.2 Retorna ao Fluxo Principal no passo 4.
14. O sistema calcula o valor total de cada produto.	5.a Endereço Inexistente
15. O cliente seleciona "finalizar pedido".	5.a.1 O sistema emite uma mensagem que o endereço é inexistente.
16. O cliente fornece seu nome e endereço.	5.a.2 Retorna ao Fluxo Principal no passo 5.
17. Se o cliente fornece apenas o CEP, o sistema coloca automaticamente a cidade e o estado.	6.a CEP Inválido
18. O sistema calcula o valor total do pedido.	6.a.1 O sistema emite uma mensagem que o endereço é inexistente.
19. O cliente seleciona o método de pagamento.	6.a.2 Retorna ao Fluxo Principal no passo 6.
20. O cliente fornece as informações sobre o cartão.	7.a Cupom de Desconto
21. O cliente submete os dados ao sistema.	7.a.1 O sistema calcula o valor do desconto.
22. O sistema verifica as informações fornecidas e marca o pedido como "pendente", e o número de pedido (NP) é dado ao cliente.	7.a.2 Retorna ao Fluxo Principal no passo 8.
	9.a Pagamento via boleto
	9.a.1 – O sistema abre uma nova aba com as informações sobre boleto
	9.a.2 – O sistema emite um boleto com todas as informações sobre o pagamento
	9.a.3- O cliente fecha essa tela
	9.a.4 – Retorna ao Fluxo Principal no Passo 10.
	9.b Pagamento por depósito
	9.b.1 – O sistema abre uma nova aba com as informações de depósito
	9.b.2 – O cliente insere as informações do depósito
	9.b.3- O cliente submete os dados ao sistema
	9.b.4 – Retorna ao Fluxo Principal no Passo 10.
Pós-Condição: O pedido deve ter sido gravado no sistema e marcado como aguardando confirmação do pagamento.	
Pontos de Extensão: 3.a Produto em Promoção - Calcular Desconto de Produto em Promoção	

Portanto, um sistema é definido por meio de um conjunto de contratos. A principal tarefa no projeto da camada de domínio é para cada contrato e subcontrato existente:

- a) Identificar as classes necessárias para implementar as regras definidas no contrato ou subcontrato.
- b) Identificar as colaborações necessárias entre as classes para implementar as regras definidas no contrato ou subcontrato.
- c) Identificar e atribuir responsabilidade a cada classe.

A identificação das responsabilidades, que é “um contrato ou obrigação da classe”, se divide em dois tipos básicos:

- 1) **Fazer algo**, que destacam as responsabilidades de:
 - i. Identificar e atribuir métodos inerentes a cada classe para realização do contrato ou subcontrato.
 - ii. Identificar invocações de métodos entre as classes para realização do contrato ou subcontrato.
 - iii. Identificar instanciação de objetos para realização do contrato ou subcontrato.
- 2) **Conhecer algo**, que são as responsabilidades das classes em conhecer coisas do tipo:
 - i. Dados privados encapsulados.
 - ii. Objetos relacionados.
 - iii. Informação derivada ou calculada.

Atribuição de Métodos as Classes

Neste momento, iremos começar a atribuir responsabilidades as classes, primeiramente, pela identificação e atribuição dos métodos. A atribuição de métodos está relacionada aos contratos existentes e compromissos esperados a determinada classe na sua definição.

Como exemplo, vamos analisar o modelo de classes definido para o sistema de comércio *online* apresentado na Figura 9.6.

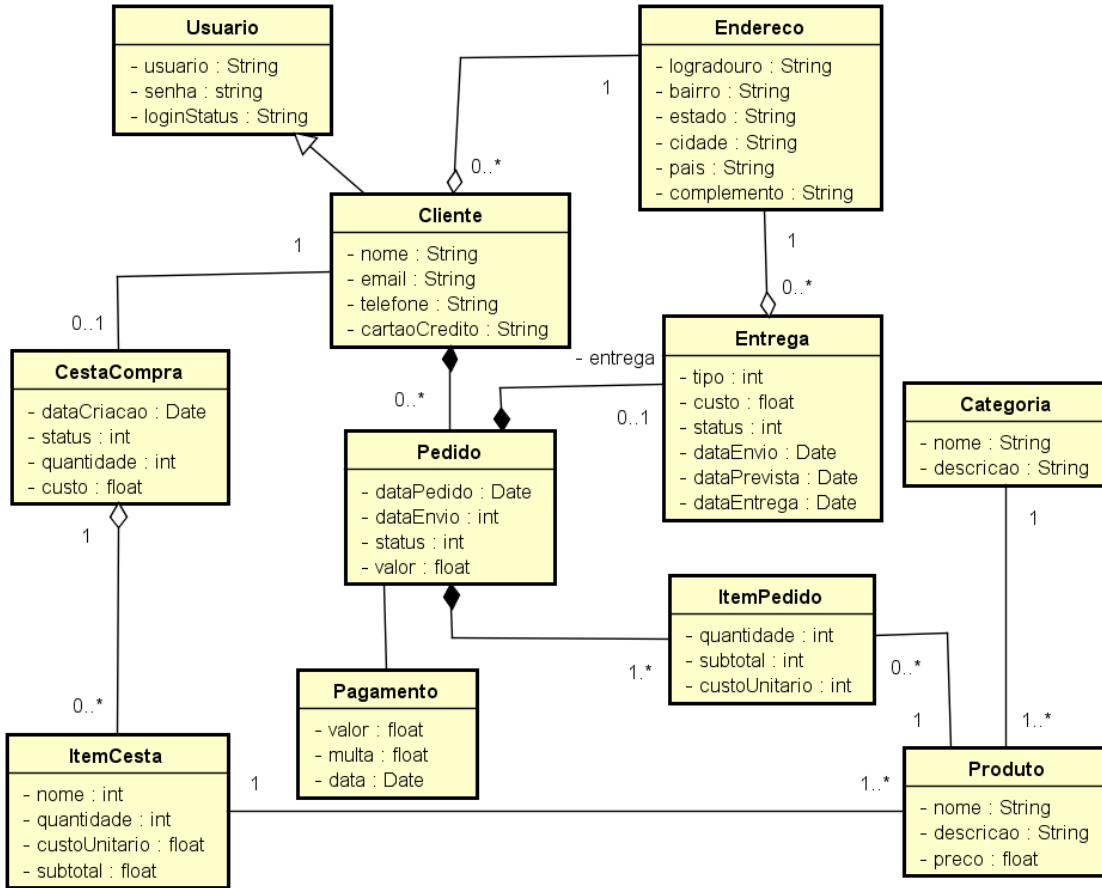


Figura 9.6: Modelo de Classes para o Sistema de Comércio Online

A partir deste modelo, vamos analisar o contrato estabelecido no Quadro 9.1, mais especificamente a regra 7 do fluxo principal: " *O sistema calcula o valor total do pedido*".

- De quem é a responsabilidade de calcular o valor total do Pedido?
 - A classe **Pedido** tem esta responsabilidade, mas será que outras classes não possuem participação nesta responsabilidade?

Neste momento quer-se traduzir as responsabilidades descritas nos contratos para classes e métodos. Esta tarefa é influenciada pela granularidade da responsabilidade. Quanto mais complexa for a responsabilidade, mais classes participarão. Assim, nesta etapa o propósito é a identificação e atribuição dos métodos, que são implementados para cumprir responsabilidades. Estes métodos

podem agir sozinhos ou em colaboração com outros métodos e objetos. Para realizar esta tarefa vamos utilizar o padrão *GRASP expert*.

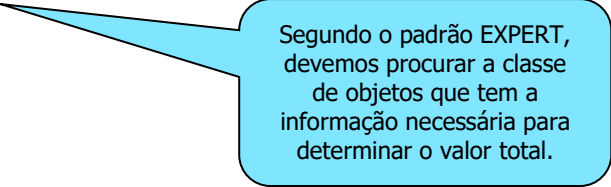
Padrão Grasp Expert

Padrões são pares de problema/solução que codificam orientações e princípios frequentemente relacionados com a atribuição de responsabilidades. Um padrão é um par nomeado problema/solução que pode ser aplicado em novos contextos, com conselhos sobre sua aplicação em novas situações e uma discussão sobre as consequências de seu uso. Assim, em geral, padrões capturam princípios fundamentais da engenharia de software e normalmente, não contêm novas ideias nem soluções originais, apenas codificam soluções existentes comprovadamente eficientes.

O padrão *Grasp expert* auxilia na definição das responsabilidades das classes, uma vez que este tem como par problema/solução:

- **Problema:** Qual é o princípio básico pelo qual responsabilidades são atribuídas em projetos orientados a objetos?
- **Solução:** Atribuir responsabilidade para o especialista na informação (a classe que tem a informação necessária para cumprir a responsabilidade).

Como exemplo, vamos analisar a regra 5, exposta anteriormente: **Quem deve ser responsável por conhecer o valor total de um pedido?**



Segundo o padrão EXPERT, devemos procurar a classe de objetos que tem a informação necessária para determinar o valor total.

Uma vez definida a responsabilidade que se deseja implementar, precisamos de dois artefatos para aplicar o padrão *Grasp expert*:

- a) **Diagrama de Classes:** para identificar as classes participantes da responsabilidade.
- b) **Diagrama de Comunicação:** para mostrar como as classes devem se comunicar (trocar mensagem para cumprir a responsabilidade).

Observando o diagrama de classes da Figura 9.6, vemos que a classe **Pedido** é responsável por conhecer o valor total do pedido, pois conforme mostrado na Figura 9.7, possui a informação necessária para cumprir a responsabilidade.

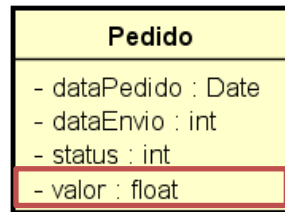


Figura 9.7: Classe Pedido

Assim, iniciamos definindo o método responsável por calcular o valor total e por manipular o atributo **valor**. Para isso, vamos utilizar o diagrama de comunicação para atribuir o método à classe **Pedido**, e atualizar a classe no diagrama de classes (Figura 8).

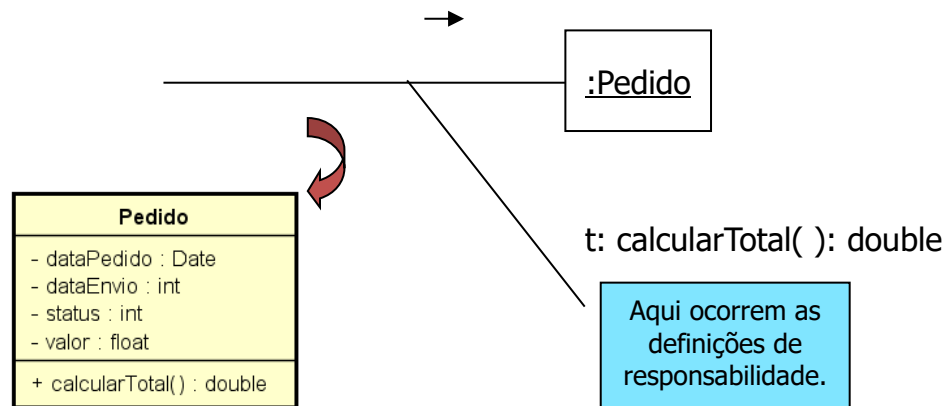


Figura 9.8: Criando a responsabilidade de Calcular o Total do Pedido

Como já visto anteriormente, é muito comum que uma classe não realize toda uma responsabilidade sozinha e delegue parte da responsabilidade a outras classes. Assim, a partir de definida a classe responsável, devemos perguntar:

- **A classe consegue sozinha calcular o valor total?**
Se não, o que ela precisa?

Para a classe **Pedido** calcular o valor total do pedido, ela precisa conhecer o subtotal de cada item do pedido. Dessa maneira, quem deve ser responsável por conhecer o subtotal de um item do pedido?

Observando o diagrama de classe, vemos que para calcular o subtotal tem-se:

- **Informação necessária:** ItemPedido.quantidade e Produto.preco
- **Pelo padrão Grasp expert**, a classe **ItemPedido** deve ser a responsável.

Da mesma maneira que feito anteriormente, utilizamos o diagrama de comunicação para auxiliar a definir o método na classe, e atualizamos o diagrama de classes, como mostrado na Figura 9.9.

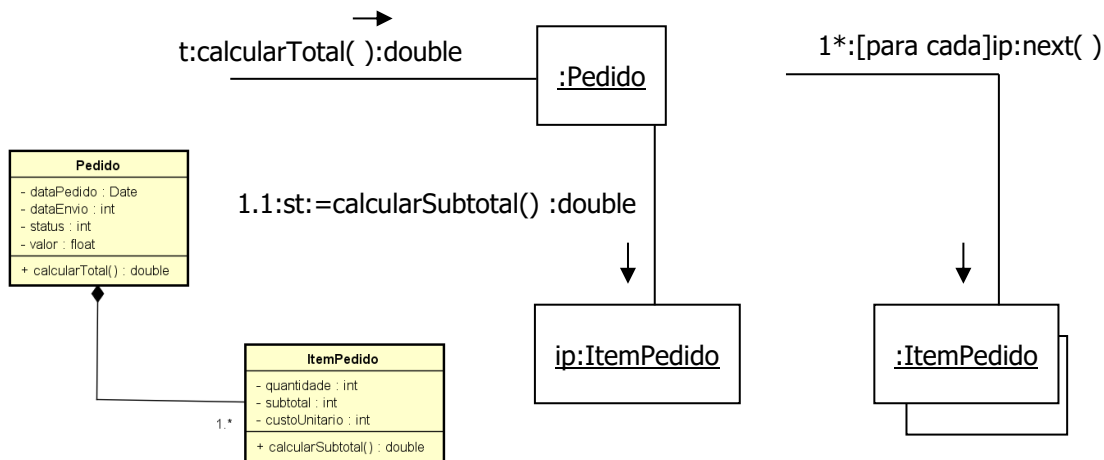


Figura 9.9: Criando a responsabilidade de Calcular o Subtotal de cada Item

Porém, para cumprir a responsabilidade de conhecer ou informar seu subtotal, um **ItemPedido** precisa conhecer o preço do Item. Portanto, o **ItemPedido** deve mandar uma mensagem para **Produto** solicitando o preço do item (Figura 9.10).

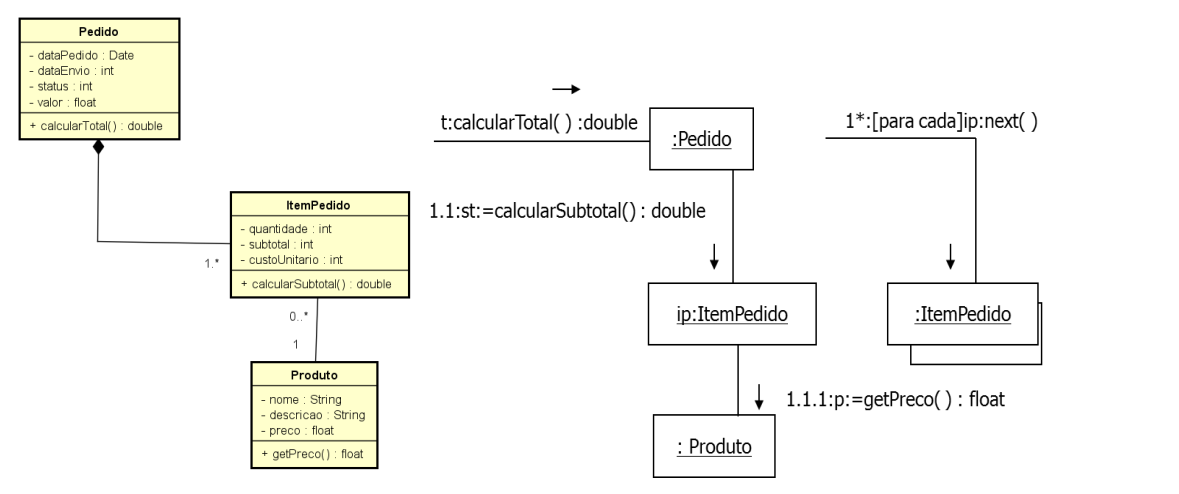


Figura 9.10: Criando a responsabilidade para pegar o preço do Produto

Assim, para cumprir a responsabilidade de conhecer e informar o total do pedido, três responsabilidades foram atribuídas a três classes de objetos, conforme apresentado no Quadro 9.2.

Quadro 9.2 – Responsabilidades Atribuídas para Calcular o Valor Total de um Pedido

Classe	Responsabilidade
Pedido	Conhecer total do Pedido
ItemPedido	Conhecer o subtotal do item
Produto	Conhecer o preço do produto

Instanciação de Objetos

Em um sistema orientado objetos basicamente existem dois tipos de chamadas a métodos:

- Chamada a métodos de objetos, que foram instanciados em algum momento;
- Chamada a métodos de classes estáticas, que classes que contêm apenas membros estáticos, não podendo serem instanciadas. Assim, pode-se invocar seus métodos a qualquer momento sem necessidade de instanciar a classe.

Foi-se até o momento, aplicado o padrão *Grasp expert* para atribuir métodos as classes. No entanto, a não ser que a classe seja estática, ela deve estar instanciada antes de qualquer invocação ocorrer. Assim, uma responsabilidade que precisa ser definida, é identificar classes responsáveis por

instanciar outros objetos para se cumprir determinado contrato. Para isso, será aplicado o padrão *Grasp creator*.

Padrão Grasp Creator

O Padrão *Grasp Creator* auxilia a identificar classes que são responsáveis por instanciar objetos. Para tanto, seu par Problema/Solução é o seguinte:

- **Problema:** Quem deveria ser responsável pela criação de uma nova instância de alguma classe?
- **Solução:** Atribuir à classe B a responsabilidade de criar uma nova instância da classe A se uma das seguintes condições for verdadeira:
 1. B agrega objetos de A
 2. B contém objetos de A
 3. B registra instâncias de objetos de A
 4. B usa objetos de A
 5. B tem os valores iniciais que serão passados para objetos de A, quando de sua criação.

Analisando o diagrama da Figura 9.6, vemos que quando existe as condições 1 e 2, que implica na existência dos relacionamentos de agregação e composição a identificação de classes que instanciam os objetos fica facilitado. Veja por exemplo a Figura 9.11.

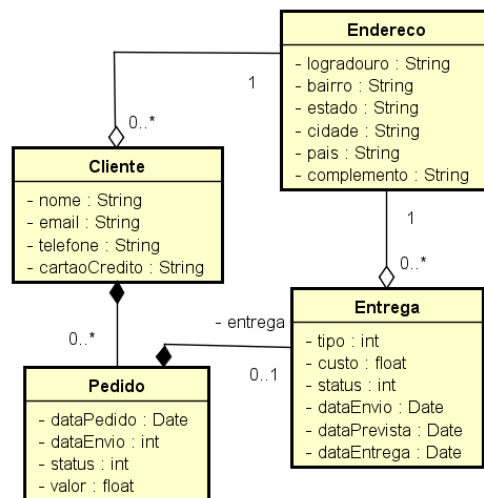


Figura 9.11: Classes com relacionamento de Agregação e Composição

Por meio da Figura 9.11, fica claro que a classe **Cliente** é responsável por instanciar um objeto da classe **Pedido**, e esta por sua vez responsável por instanciar um objeto de

Entrega. Tanto a classe **Cliente** quanto **Entrega** podem instanciar **Endereco**. No entanto, é preciso observar apenas, que se as duas classes forem manipular o mesmo objeto este não pode ser instanciado novamente.

As outras condições dizem respeito basicamente, que o candidato a criador é o objeto que conhece os dados iniciais do objeto a ser criado ou usa o objeto. Assim, se voltarmos ao exemplo do cálculo do valor do pedido, apesar de **ItemPedido** precisar de produto para calcular o valor do subtotal, ele não conhece os dados iniciais do **Produto**, assim provavelmente não é o criador deste objeto.

Para deixar claro o momento em que a criação do objeto ocorre podemos utilizar o diagrama de comunicação.

Exemplo

Quem deve ser responsável por criar uma instância de ItemPedido?

Pelo padrão, **Pedido** deve ser o responsável, de acordo com a condição 2 (**Pedido** contém objetos de **ItemPedido**)

Utilizando o diagrama de comunicação para definir a instanciação, podemos apresentar conforme mostra a Figura 9.12.

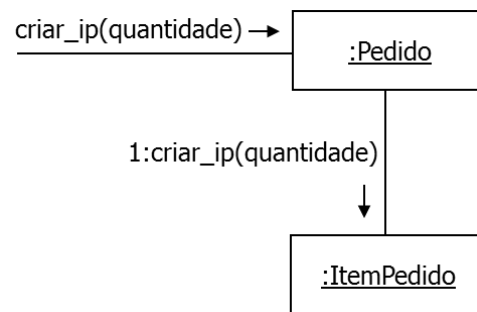


Figura 9.12: Instanciação de um Objeto

Normalmente a classe que instancia não precisa ter um método específica para criar o objeto de outra classe, no entanto o diagrama de comunicação não oferece suporte para representar a criação de métodos. Mais adiante, quando for explanado sobre o diagrama de sequência, será visto que este oferece suporte para representar instanciação de objetos.

Coesão e Acoplamento

Como já apresentado e explicado no Capítulo 7 (Princípios e Conceitos de Projeto de Software) um dos grandes objetivos de um projeto de software é conseguir um produto de qualidade. Para isso, um dos pilares da programação orientada a objetos é a independência funcional, que preconiza que o software deve ser constituído de módulos independentes entre si, facilitando o reuso e a manutenibilidade destes módulos. Em sistemas de software, é possível determinar o grau de independência funcional dos módulos, utilizando para isso métricas de coesão e acoplamento. Assim, é essencial no projeto do software criar modelos altamente coesos e com baixo acoplamento.

Coesão

A coesão indica se uma classe tem uma função bem definida no sistema. Assim, devemos analisar se as classes que estão definidas têm um grau adequado de coesão. Um conjunto de testes que podem ser aplicados para analisar a coesão são:

- a) Examinar uma classe e decidir se todo o seu conteúdo está diretamente relacionado ao que é descrito pelo nome da classe.
- b) Verificar se existe um subconjunto de métodos e campos que poderiam facilmente ser agrupado à parte, com outro nome de classe.
- c) Verificar se o valor de algum atributo determina a possibilidade de outro atributo ser nulo ou não.

Vamos analisar o diagrama da Figura 9.13, o qual apresenta uma classe *Pedido* e *Pagamento*. Por meio deste diagrama, é possível perceber (em destaque), que na classe *Pagamento* item **c** é satisfeito, uma vez o atributo *dataPagamento* possibilita o atributo *multa* ser nulo ou não.

Ainda no modelo apresentado na Figura 9.13, vemos que tanto o item **a** como **b** ocorrem neste modelo. O item **a** ocorre, pois, a multiplicidade 0..*, indica que um pedido pode ter vários pagamentos (indicando o parcelamento do pagamento, por exemplo). Contudo, por meio dos atributos, vemos que a classe *Pagamento* possui tanto dados sobre o pagamento gerado como sobre o pagamento efetuado, que são duas funcionalidades diferentes (Imagine que você efetuou um pedido em site de comércio eletrônico, e gerou o boleto, mas não o pagou).

Considerando o apresentado, o entendimento é que todo pedido deveria estar associado a um pagamento pendente, e este sim, estar associado a 0 ou vários pagamentos efetuados.

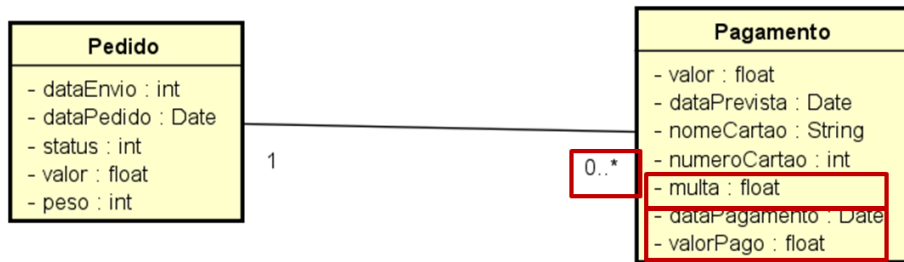


Figura 9.13: Exemplo de Baixa Coesão

Sabendo dos problemas de coesão que o modelo da Figura 9.13 apresenta, este mo9.delo pode ser evoluído para o modelo apresentado na Figura 14. Este modelo faz a diferenciação entre o pagamento pendente (classe *Pagamento*) e o Pagamento realmente efetuado. O modelo da Figura 9.14, resolve o problema do item **a** (agora temos uma classe com responsabilidade sobre o Pagamento gerado e outra sobre o pagamento efetuado). Contudo, na classe *Pagamento* ainda vemos o item **b** (os atributos *nomeCartao* e *numeroCartao* poderiam facilmente ser agrupado à parte), e na classe *PagamentoEfetuado*, tem-se o item **c**, pois a *multa* ainda depende do atributo *valorPago*.

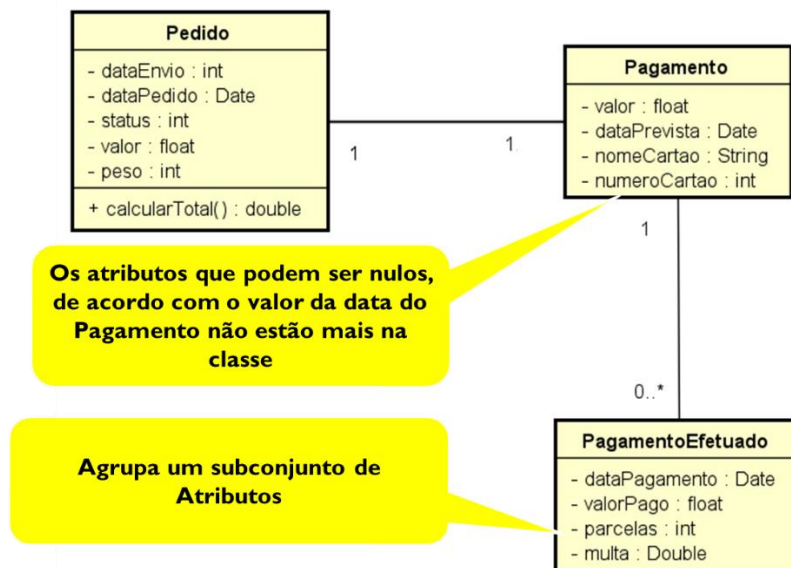


Figura 9.14: Primeira Evolução do Modelo com Baixa Coesão

O modelo da Figura 9.14, pode então ser evoluído para o modelo da Figura 9.15, o qual acrescenta duas novas classes: *Multa* e *Cartao*. Neste modelo consideramos que o pagamento pode ser efetuado utilizando apenas um cartão.

Observando o modelo da Figura 9.15, percebe-se que o Pagamento Efetuado está relacionado a várias multas, uma vez

que um mesmo pagamento pode ter várias multas (se o pagamento for parcelado, e mais de uma parcela sofrer atrasos). Sabendo disso, a coesão da classe *PagamentoEfetuado* não está adequada, pois não se sabe se o atributo *valorPago* se refere ao valor do pagamento total ou ao valor do pagamento da parcela.

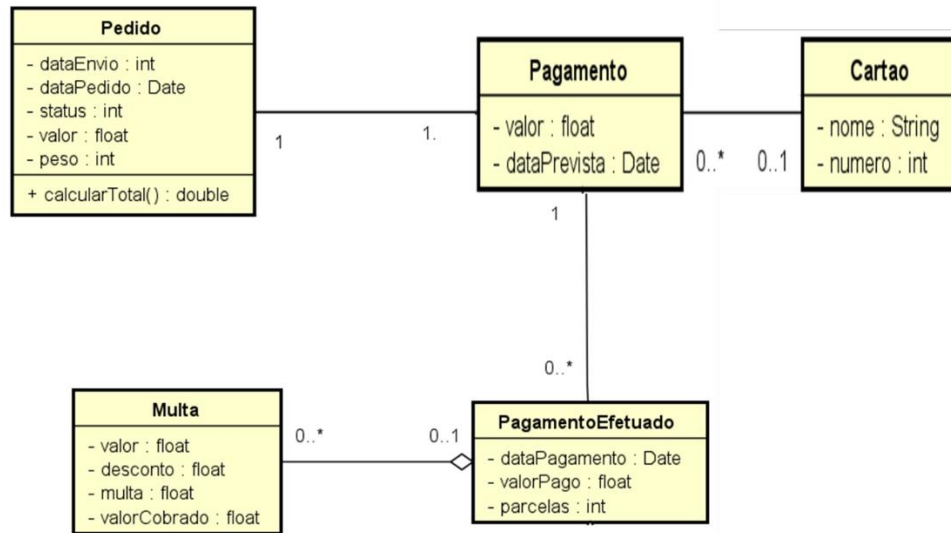


Figura 9.15: Segunda Evolução do Modelo com Baixa Coesão

Para tratar este problema e aumentar a coesão do pagamento efetuado, vamos dividir essa responsabilidade em duas classes, uma para tratar o pagamento a vista, e outra o parcelado. Este novo modelo é apresentado na Figura 9.16. Neste novo modelo, cada pagamento gerado é associado a apenas um pagamento efetuado, e este pode ou não ter parcelas associadas. Além disso, a multa está associada a no máximo um único objeto *PagamentoEfetuado* e no máximo a um único objeto *Parcela*, indicado exatamente a qual objeto a multa ocorreu.

Por fim, este modelo pode ainda se melhorado, considerando uma classe abstrata para tratar o pagamento efetuado, tendo como subclasses, tanto o pagamento a vista como o parcelado. Este novo modelo é apresentado na Figura 9.17, assim a multa é associada a cada pagamento e um pagamento efetuado pode ser a vista, ou composto por várias parcelas.

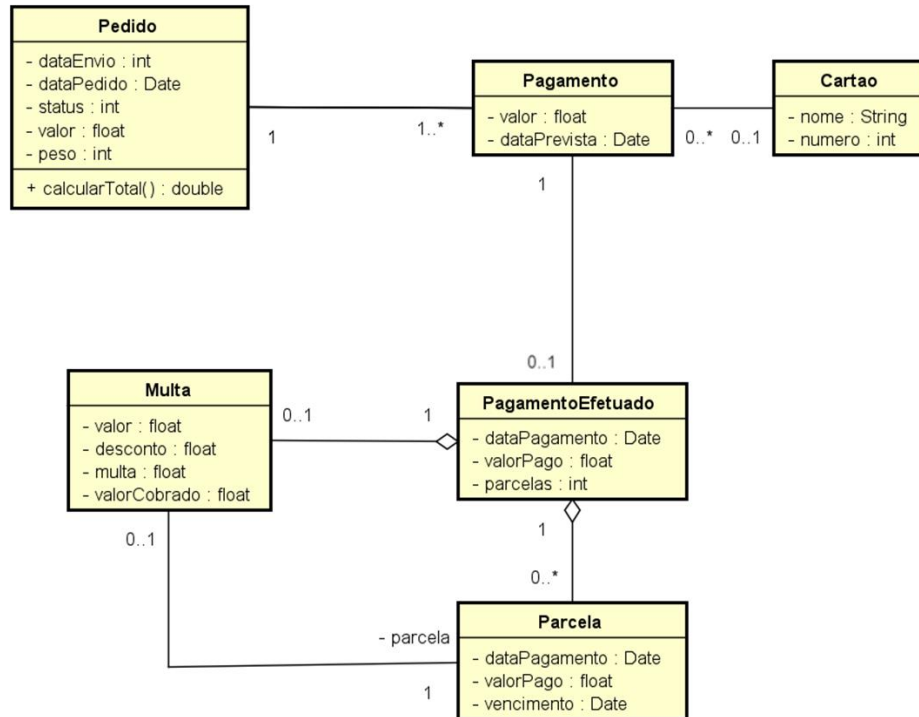


Figura 9.16: Terceira Evolução do Modelo com Baixa Coesão

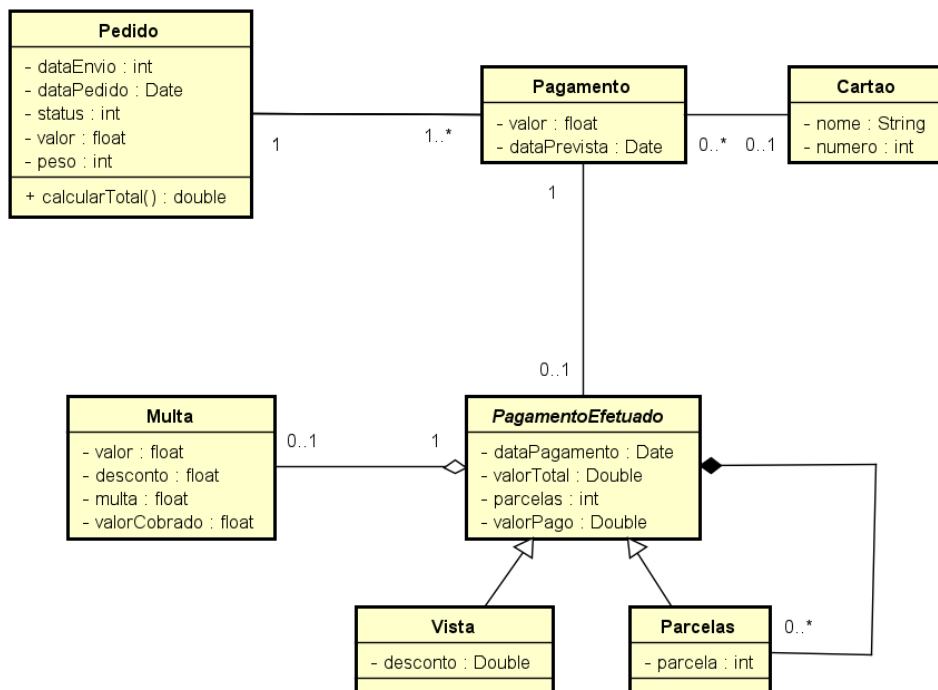


Figura 9.17: Quarta Evolução do Modelo com Baixa Coesão

Outro aspecto que deve ser levado em consideração para definir a coesão está na instanciação de objetos. Como exemplo, vamos analisar:

Acoplamento

No projeto também devemos nos preocupar com nível de acoplamento entre as classes, lembrando que buscamos no projeto de software o acoplamento fraco (ou baixo), que indica que um objeto não depende de muitos outros.

De maneira a ficar mais claro o entendimento sobre acoplamento, vamos analisar duas situações. Na primeira, vamos pensar em como o padrão GRASP *Creator* pode auxiliar a diminuir a coesão.

Quem deve ser responsável por criar um Pagamento e associá-lo ao Pedido?

O Padrão *Creator* nos ajuda a responder esta questão. Vamos imaginar uma situação como a apresentada na Figura 9.18, o qual a classe controladora tem essa reponsabilidade. Se utilizarmos essa solução a classe controladora irá ficar cada vez mais sobrecarregada e sem coesão. Se utilizarmos o Padrão *Creator* será percebido que o **Pedido** tem as informações iniciais que serão passadas para a criação de pagamento. Assim, conforme apresenta a Figura 9.19, definimos um método *realizarPagamento* em **Pedido**, que irá criar um pagamento (caso não exista) e delegar essa responsabilidade à classe **Pagamento**.

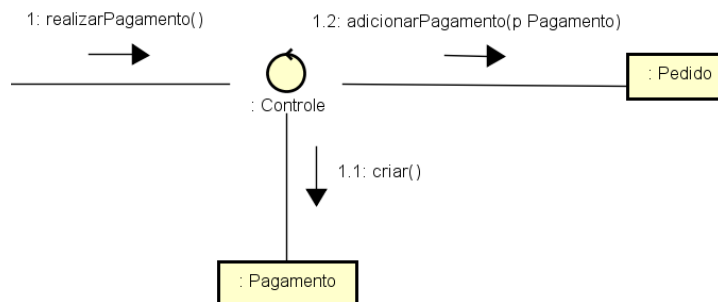


Figura 9.18: Criação de Objetos que irá sobrecarregar a classe Controladora

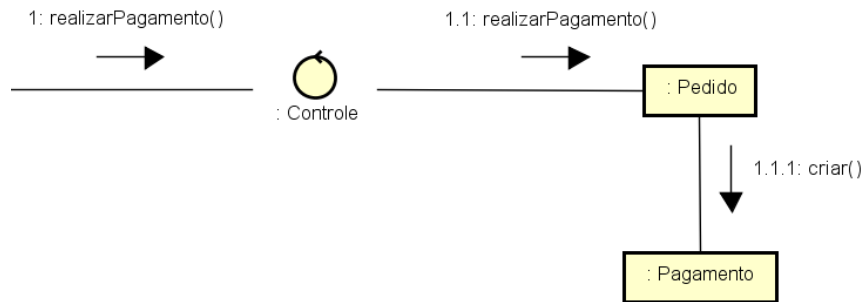


Figura 9.19: Criação de Objetos para melhorar a Coesão

Na segunda situação, vamos utilizar o sistema de biblioteca (Figura 9.20). Vamos supor que queremos realizar a devolução do livro. Qual classe deve ser responsável por essa tarefa?

A devolução é feita a partir de um aluno (que contraiu o Empréstimo), portanto é sensato partir da classe **Aluno**.

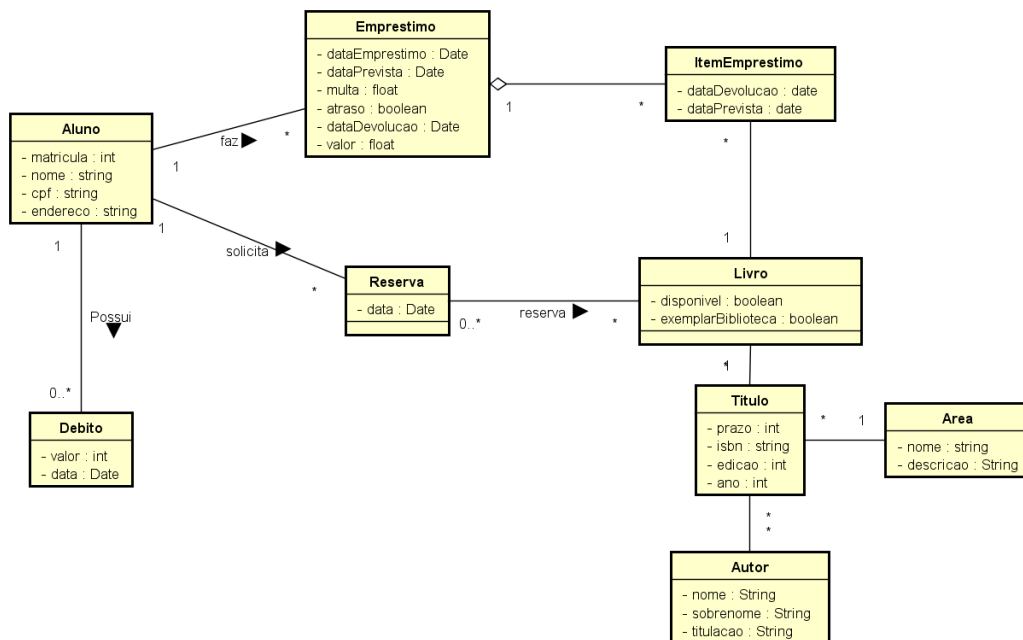


Figura 9.20: Modelo de Classes de Análise para Biblioteca

Se essa responsabilidade for atribuída à classe **Aluno**, teríamos uma colaboração como apresentada no diagrama de comunicação da Figura 9.21. O problema desta solução é que o aluno teria uma responsabilidade de devolver o livro, e para isso teríamos o aumento de acoplamento, pois a classe **Aluno** teria uma dependência com a classe **ItemEmpréstimo**, que não estava previsto no diagrama de classes inicial. Isto estaria ferindo a Lei de Deméter, pois **Aluno** não está falando com os mais próximos.

Para resolver esta situação, o método *devolver* de **Aluno** é um método que apenas delega a responsabilidade à classe **Emprestimo**, conforme é mostrado na Figura 9.22, e essa então colabora com a classe **ItemEmprestimo** para realizar a devolução. Nesta solução manteríamos o acoplamento do diagrama inicial planejado e as classes estariam apenas se comunicando com as mais próximas.

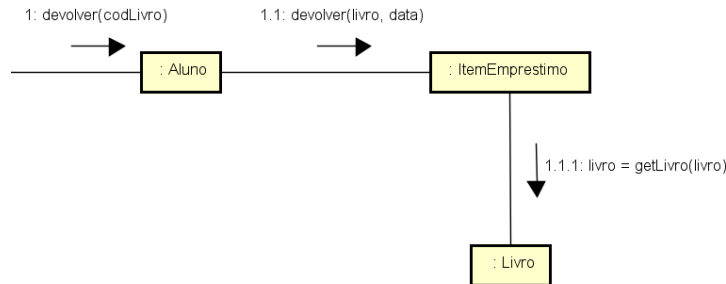


Figura 9.21: Atribuição da responsabilidade de Devolução a classe Aluno

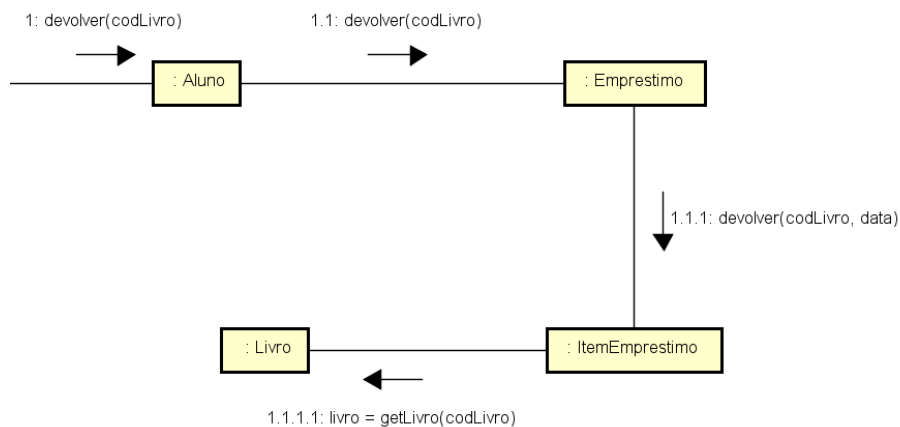


Figura 9.22: Delegando a responsabilidade de Devolução a classe Emprestimo

Controlador

Uma classe controladora é muito importante em sistemas OO, pois este tipo de classe faz a comunicação entre a uma interface GUI (*Graphic User Interface*) e as classes de domínio. Dessa forma, as classes de domínio ficam desacopladas da tecnologia utilizada para implementar a interface gráfica.

Uma classe controladora é um objeto que não é de interface GUI e é responsável pelo tratamento de eventos do sistema, definindo métodos para as operações do sistema.

Padrão Grasp Controller

O padrão *Grasp Controller* auxilia a identificar e definir o objeto responsável, fora da camada de apresentação, que deve receber e coordenar a solicitação da execução de uma operação.

O objeto Controlador responde a uma questão básica no projeto de sistemas OO:

- ***Como conectar a camada de apresentação à camada da lógica do negócio?***

O controlador é o primeiro objeto fora da camada de interface com o usuário a recebe ou trata uma mensagem para o sistema. Existem basicamente duas alternativas possíveis para definir um objeto controlador:

- Um objeto Controlador para todo o sistema (*Facade*).
- Um objeto Controlador por Caso de Uso (ou por cenário de Caso de Uso)

Com o intuito de entender melhor o papel de uma classe controladora, vamos observar a descrição do caso de uso **Emprestar Livro** (Quadro 9.3). Neste caso de uso vemos que existem diversas interações entre o ator e o sistema até que sua execução finalize. Entre cada interação, existem ações que devem ser executadas e dependendo do retorno destas ações, o sistema deve ir por um caminho ou outro.

Como exemplo, vamos imaginar que no passo três o usuário não esteja cadastrado. Neste caso o sistema deve emitir uma mensagem e finalizar a execução da funcionalidade. No entanto, caso contrário, ele deveria continuar a execução da funcionalidade. A coordenação da execução destas ações deve ser coordenada por uma classe controladora.

Quadro 9.3: Caso de Uso Emprestar Livro

Caso de Uso: Emprestar Livro	
Fluxo Principal	Fluxos Alternativos
<p>Fluxo de Principal (caminho básico):</p> <ol style="list-style-type: none"> 8. O aluno apresenta os livros ao funcionário e o sua identificação 9. O funcionário insere a identificação e os livros no sistema 10. O sistema verifica se o Aluno está cadastrado 11. O sistema verifica se o Aluno possui Pendências 12. O sistema cria um empréstimo 13. Para Cada Livro <ol style="list-style-type: none"> 6.1 O sistema verifica se o livro pode ser emprestado 6.2 O Sistema cria um item de empréstimo 6.3 O sistema associa o livro ao item 14. O sistema Calcula a Data de Devolução (ponto de Inclusão Calcula Data de Devolucao) 8. O sistema grava os dados do empréstimo 9. O sistema imprime os dados do empréstimo 	<p>3.a Aluno não cadastrado</p> <ol style="list-style-type: none"> 3.a.1 - O sistema informa que o aluno não esta cadastrado 3.a.2 - O sistema finaliza o caso de uso
	<p>4.a Aluno possui débitos</p> <ol style="list-style-type: none"> 4.a.1 - O sistema informa que o aluno está em débito 4.a.2 - O sistema finaliza o caso de uso
	<p>6.1.a Livro Reservado</p> <ol style="list-style-type: none"> 6.1.a.1 O sistema informa que o livro está reservado e não pode ser emprestado 6.1.a.2 O sistema informa a data de devolução do livro 6.1.a.3 Retorna ao passo 6
	<p>6.1.b Livro de Não pode ser Emprestado</p> <ol style="list-style-type: none"> 6.1.b.1 O sistema informa que o livro é exemplar que não pode ser emprestado 6.1.b.3 Retorna ao passo 6.

Neste caso, a classe controladora funcionaria conforme é ilustrado na Figura 9.23. Após o ator **Funcionária** inserir os dados na GUI, esta instanciaria um objeto controlador, que é responsável por coordenar a execução do fluxo. A controladora não faz verificações, somente coordena a tarefa, delegando a sua execução para os outros objetos do sistema. Assim, ela recebe os dados enviados pelo sistema, e começa a gerenciar a execução do empréstimo. Para isso, a primeira regra descrita no caso de uso, é verificar se o aluno está cadastrado. A classe controladora então instancia um objeto aluno e chama o método responsável por verificar se o aluno está cadastrado. A partir do retorno deste método, a classe controladora irá coordenar as próximas ações do fluxo.

Caso o aluno não esteja cadastrado, a classe controladora irá emitir uma mensagem de erro para interface, conforme é mostrado na Figura 9.24. No entanto, se o aluno estivesse cadastrado, continuaria o fluxo, e, portanto, a classe controladora instanciaria objetos de Titulo para continuar a execução do empréstimo (Figura 9.25).

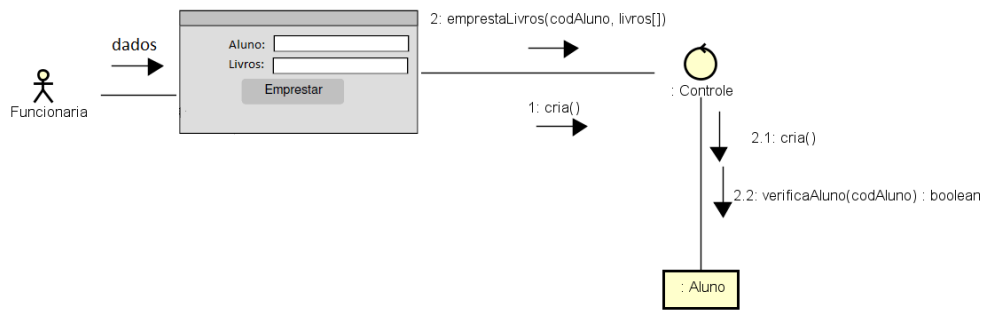


Figura 9.23: Diagrama de Comunicação mostrando o funcionamento da Classe Controladora

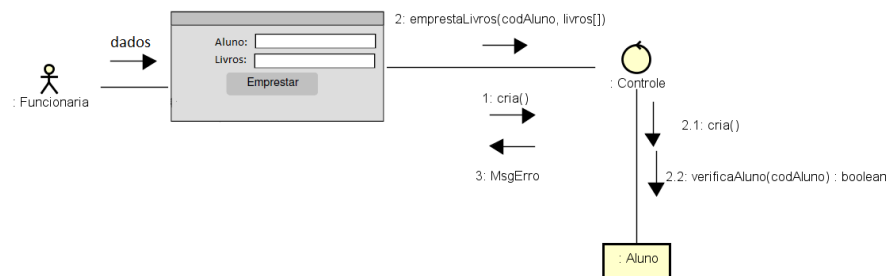


Figura 9.24: Diagrama de Comunicação mostrando Aluno não Cadastrado

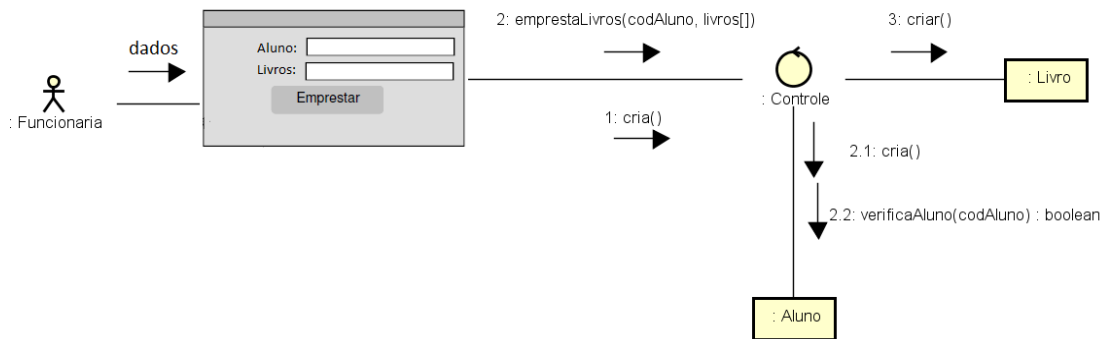


Figura 9.25: Diagrama de Comunicação mostrando Aluno Cadastrado

Quanto aos tipos de classes controladoras, um controlador fachada deve ser um objeto (do domínio) que seja o ponto principal para as chamadas provenientes da interface com o usuário ou de outros sistemas, e pode ser uma abstração de uma entidade física, por exemplo, *TerminalDeAtendimento*, ou pode ser um conceito que represente o sistema, por exemplo *Biblioteca*. Este tipo de classe controladora é adequada quando não há uma grande quantidade de eventos de sistema ou quando não é possível redirecionar mensagens do sistema para

controladores alternativos (ex: outros subsistemas). O padrão Facade pode ser utilizado para realizar esta implementação, é apresentado na próxima seção.

Outra opção é criar um controlador diferente para cada caso de uso, por exemplo, o `ControladorDeEmprestarLivro` será responsável pelas operações `iniciarEmpréstimo`, `emprestarLivro` e `encerrarEmpréstimo`. Este tipo de classe controladora não representa um objeto do domínio, e sim uma construção artificial para dar suporte ao sistema. Esta alternativa é mais viável caso perceba-se que a escolha de controladores fachada deixe a classe controladora com alto acoplamento e/ou baixa coesão (controlador inchado por excesso de responsabilidades). Além disso, é uma melhor solução quando existem muitos eventos envolvendo diferentes processos.

Deve-se tomar cuidado com classes controladoras mal projetadas, pois ocasionam baixa coesão, pois terão falta de foco e tratamento de muitas responsabilidades. Além disso, deve-se sempre analisar se a classe controladora está mesmo desempenhando o papel de controladora e não executando tarefas necessárias para atender o evento, sem delegar para outras classes (coesão alta, não especialista).

Como benefícios caso se tenham controladoras bem definidas, o sistema terá:

- Objetos de interfaces e da camada de apresentação que não têm a responsabilidade de tratar eventos do sistema.
- Aumento das possibilidades de reutilização de classes e do uso de interfaces.
- Conhecimento do estado do caso de uso controlador pode armazenar estado do caso de uso, garantindo a sequência correta de execução de operações.

A classe controle é essencial para a separação *view-controller*, o que facilita a reutilização de componentes específicos de negócio, além de possibilitar o uso de padrões arquiteturais como o MVC (*Model View Controller*).

Por fim, devemos nos preocupar com quais classes de domínio a classe Controladora estará acoplada, para que não tenha com um forte acoplamento. Supondo que usemos uma classe controladora de fachada chamada `CBiblioteca`, a quais classes do modelo apresentado na Figura 9.20 a classe controladora deve ser relacionar? A ideia é manter o acoplamento original, e analisar principalmente o padrão *Grasp Creator*:

- A classe Aluno Possui dados para instanciar Débito e Empréstimo e Reserva.
- A classe ItemEmpréstimo é uma agregação de Empréstimo.
- A classe Livro deve ser associada a um Título.

As outras classes podem ter acesso direto, como pesquisar por autor, área ou aluno. Assim, a classe controladora poderia estar acoplada a estas classes conforme ilustra a Figura 9.26.

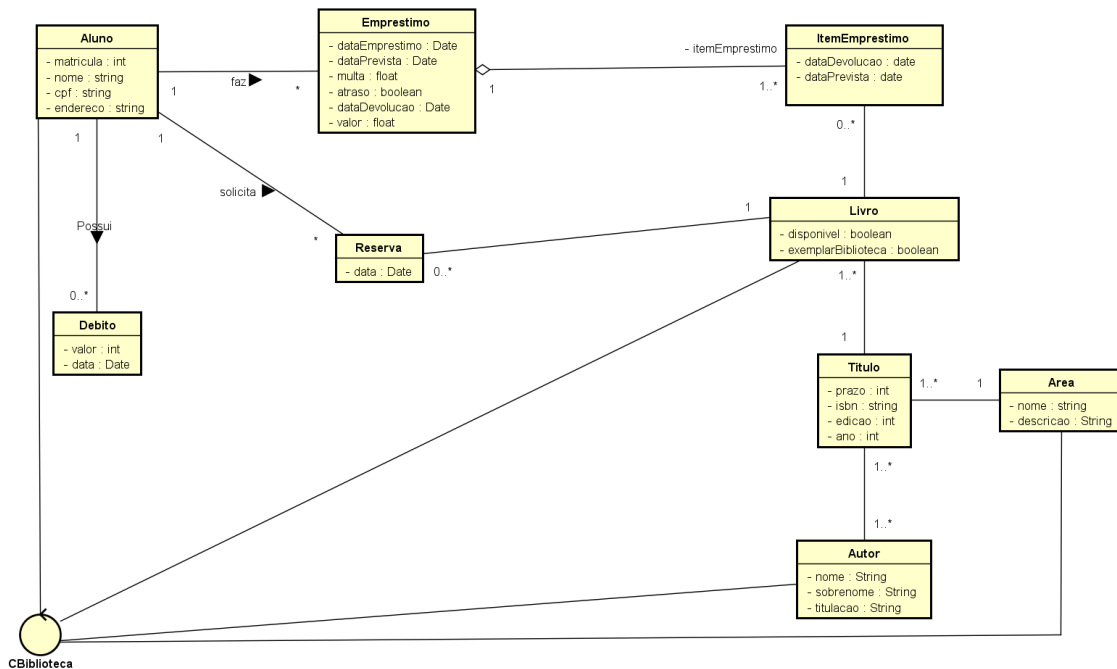


Figura 26: Diagrama de Classe de Biblioteca com Classe Controladora

Um outro exemplo do uso da controladora seria no sistema de comércio eletrônico. A responsabilidade de calcular o valor do pedido (como foi visto no padrão GRASP expert) é da classe pedido. Contudo este cálculo é mais complexo do que apenas saber o valor total. Pode ser necessário várias verificações e cálculos, tais como o cálculo do frete e descontos e promoções. Assim, toda essa lógica complexa não é responsabilidade do pedido e sim de uma classe controladora, que deve coordenar todas essa execução.

Padrão de Projeto Facade

O padrão de projeto Facade pode ser utilizado para definir uma classe controladora, apesar de não ser seu único propósito. A definição para este padrão de acordo com Gamma

(1995) é “fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Facade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.”. Dentre suas principais aplicabilidades, podem ser destacadas:

- Prover interface simples para subsistema complexo.
- Muitas dependências entre clientes e classes que implementam uma abstração.
- Criar camadas no subsistema.

O problema que o padrão Facade busca solucionar é apresentado na Figura 9.27, o qual um cliente que utiliza um subsistema precisa conhecer detalhes de diversas classes para poder utilizá-lo. Assim, o Facade simplifica a utilização de um subsistema complexo apenas implementando uma classe que fornece uma interface única e mais razoável.

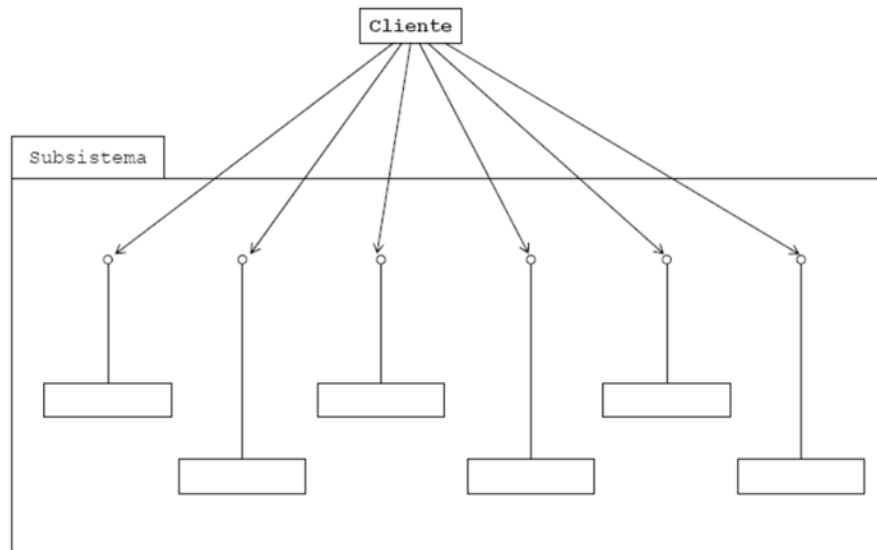


Figura 9.27: Um Cliente Acessando Diversas Classes de um Subsistema

A aplicação do padrão Facade é simples, como é apresentado na Figura 9.28. Este padrão tem como participantes a classe Facade que tem como responsabilidade:

- Conhecer quais classes do subsistema seriam responsáveis pelo atendimento de uma solicitação.
- Delegar solicitações de clientes a objetos apropriados do subsistema.

Além da classe Facade, existem as classes do subsistema, que não têm conhecimento da classe Facade e têm as responsabilidades de:

- Implementam as funcionalidades do subsistema.
- Responder a solicitações de serviços da Facade.

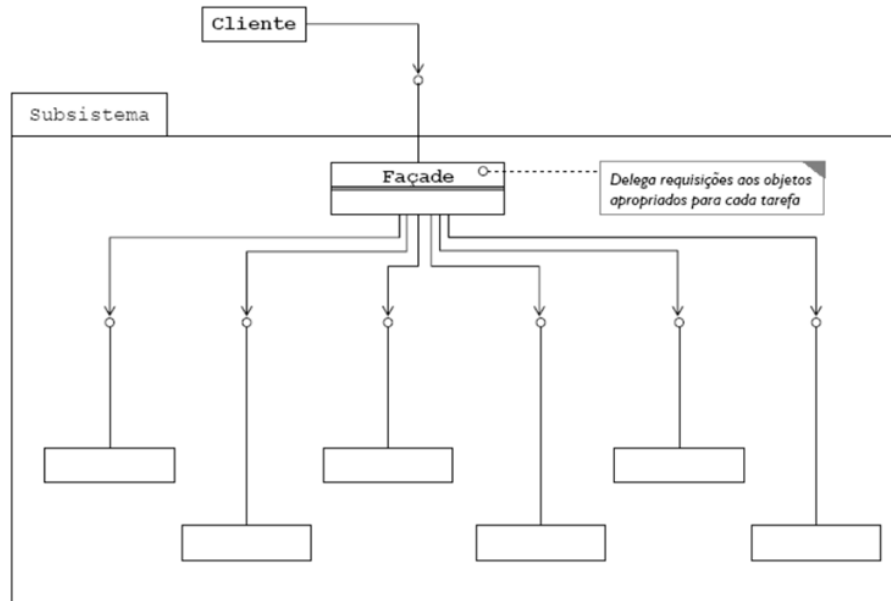


Figura 9.28: Visão Geral do Padrão Facade

Para entender como o padrão Facade pode funcionar como uma controladora, imaginemos a implementação (de forma bem simples) do sistema de comércio eletrônico. Vamos imaginar que este sistema possa ser acessado via uma interface *web* ou por uma interface *mobile*. Além disso, vamos considerar a camada de domínios é um subsistema. A implementação poderia ser feita como apresentada na Figura 9.29, o qual a classe Facade é uma controladora e as interfaces os clientes.

Neste exemplo, tanto a interface *web* como *mobile* acessam a fachada, ou seja, o acesso ao domínio é feito por uma interface única, tornando o domínio independente da interface.

Vemos também que o padrão Facade foi utilizado com finalidade de criar camadas. A camada acima precisa dos serviços da camada abaixo, que por sua vez desconhece a camada superior. Além disso, as classes do subsistema desconhecem o Facade, um dos princípios do padrão, e a classe controladora, que neste exemplo representa o Facade, conhece as classes do subsistema e delega responsabilidade à estas classes.

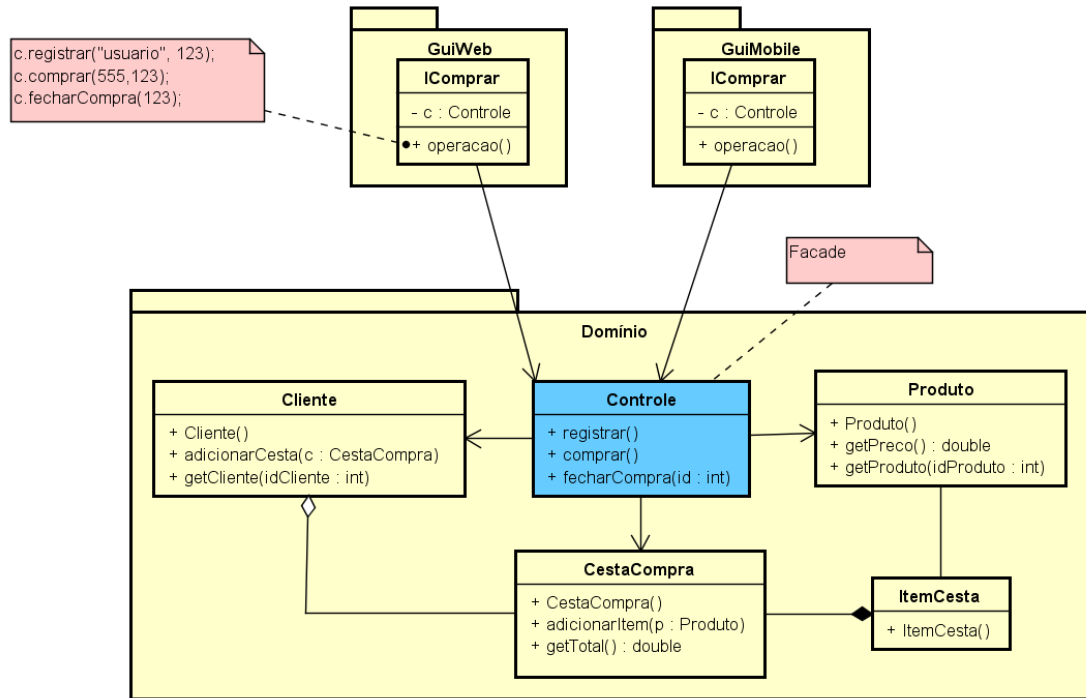


Figura 9.29: Aplicando o Facade como Controladora

Quando se utiliza o padrão Facade deve-se tomar cuidado com o acoplamento, evitando delegar responsabilidades que não são inerentes a classe Facade. Na Figura 9.29, percebe-se que a **Controle** não está respeitando o padrão *Grasp Creator*, uma vez que a controladora está sendo responsável por criar a **CestaCompra**, aumentando o acoplamento. Para resolver este problema, pode ser aplicado o padrão *Creator*, atribuindo a **Cliente** a responsabilidade de criar instâncias de **CestaCompra** e inserindo o método delegado `adicionarItem()` em **Cliente**, como observado na Figura 9.30.

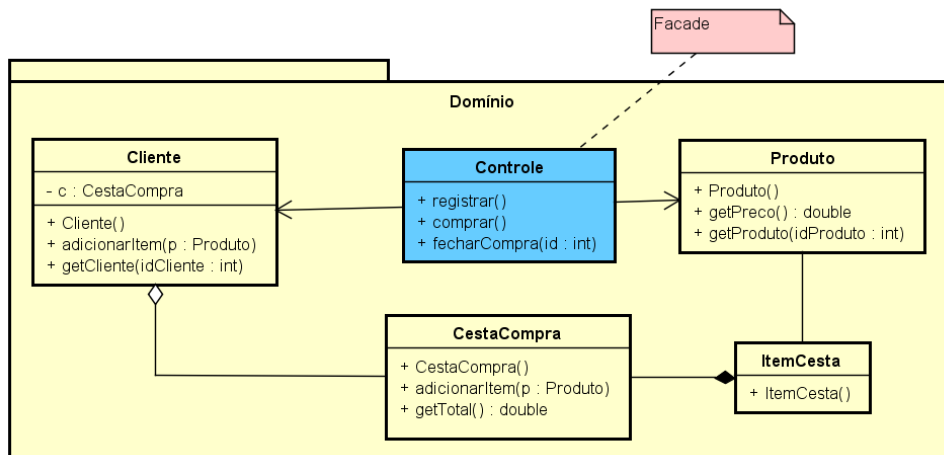


Figura 9.30: Eliminando Acoplamento Indesejado

A implementação do modelo apresentado na Figura 9.30 seria correspondente ao código da Figura 9.31. A classe de interface chama os métodos da classe **Controle**, que por sua vez é responsável por delegar responsabilidades as classes do domínio. Neste trecho de código, vemos que a classe **Cliente** é responsável por registrar um novo usuário, além de instanciar objetos do tipo **CestaCompra** e delegar a inserção de novos produtos a essa classe.

```
class IComprar {
    ...
    Controle c;
    c.registrar("usuario", 123);
    c.comprar(555,123);
    c.fecharCompra(123);
    ...
}

public class Cliente {
    ...
    CestaCompra c = new
    CestaCompra();

    public Cliente(String nome, int id)
    {
        ...
    }
    public void adicionaItem(Produto p)
    {
        c.adicionaItem(p);
    }
    ...
}

public class Controle {
    ...
    public void registrar(String nome, int id)
    {
        Cliente c = new Cliente(nome, id);
    }

    public void comprar(int produto, int idCliente)
    {
        Cliente c = getCliente(idCliente);
        Produto p = getProduto(idProduto);
        c.adicionaProduto(p);
    }
    ...
}
```

Figura 9.31: Trecho de Código implementado usando o padrão Facade

Como consequências do uso do padrão Facade, podem ser destacados:

- Os componentes do subsistema ficam encapsulados e ocultos aos clientes, isso traz como implicações:
 - o Reduz o número de objetos que os clientes lidam.
 - o O Subsistema fica mais fácil de usar e modificar.
- Fraco acoplamento entre subsistema e seus clientes.
- Não impede que aplicações usem classes do subsistema, caso elas precisem.

Pacotes

Um conceito importante em projeto de software são os pacotes, que são dispostos pela UML e são mecanismos de propósito gerais para a organização de elementos da modelagem em grupos.

Os pacotes são utilizados para a organização lógica do sistema, e visam agrupar itens que possuem relações, organizando os elementos do modelo, de maneira que sua compreensão seja facilitada. Além disso, com pacotes é possível controlar o acesso a seus conteúdos, permitindo criar regras específicas e auxiliando a definir, principalmente a arquitetura lógica do sistema.

Os pacotes são representados graficamente como disposto na Figura 9.32.

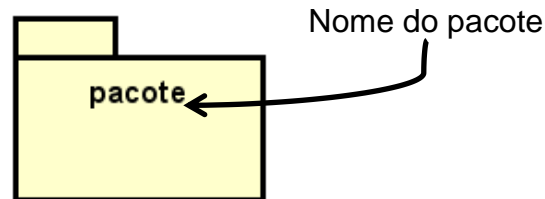


Figura 9.32: Representação Gráfica de um Pacote

Em um projeto de classes, as classes ficam agrupadas em pacotes e dentro dos pacotes podem existir subpacotes. A relação que existe entre pacotes é a de importação, que assegura uma permissão unilateral para que os elementos de um pacote tenham acesso aos elementos pertencentes ao outro pacote.

Normalmente um elemento pertencente a um pacote é público. Isso significa que o elemento está visível em relação ao conteúdo de qualquer pacote que faça a importação do pacote que contém o elemento. No entanto, elementos protegidos só podem ser vistos por elementos filhos, e elementos privados não podem ser vistos fora do pacote em que são declarados. Por exemplo, na Figura 9.33 o pacote **Negocio** importa o pacote **AcessoDados**. As classes do pacote de negócio conseguem ver a classe **Execucao**, mas não podem ver **Conexao**.

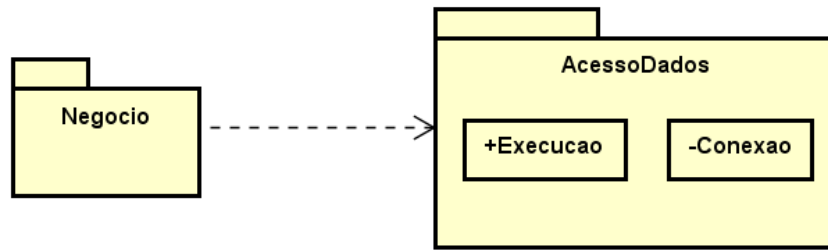


Figura 9.33: Visibilidade entre Classes de Pacotes

Existem duas formas básicas de se utilizar os pacotes em um projeto. Uma das maneiras é como mostrado na Figura 9.34, o qual são exibidos os pacotes, suas comunicações e as classes pertencentes a cada um dos pacotes.

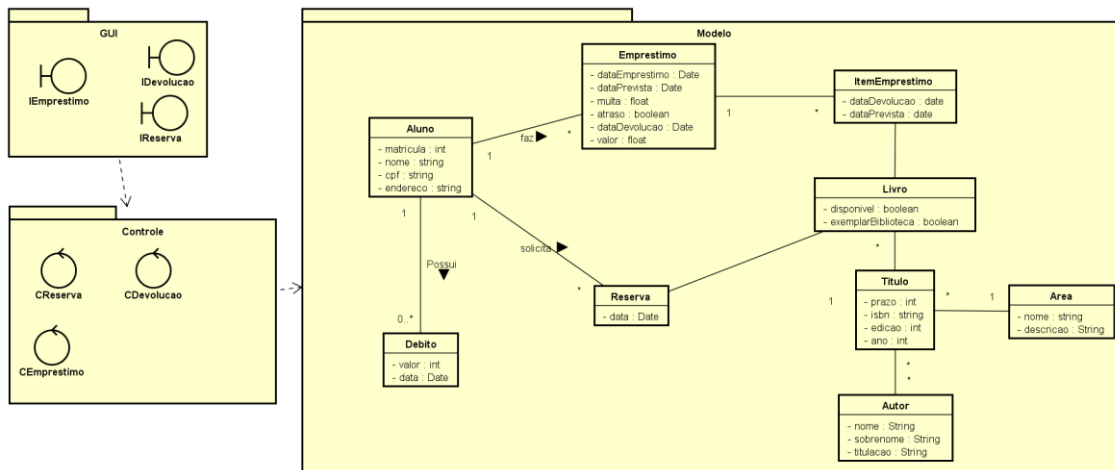


Figura 9.34: Diagrama apresentado Pacotes e suas Classes

No entanto, quando o sistema é muito complexo e existem muitas classes, pode ser difícil entender o sistema se todos os pacotes e suas classes forem expostas. Assim, é muito comum criar um diagrama de pacotes a parte, o qual apenas exhibe os pacotes e suas relações, sem mostrar as classes de cada pacote, conforme é mostrado na Figura 9.35.

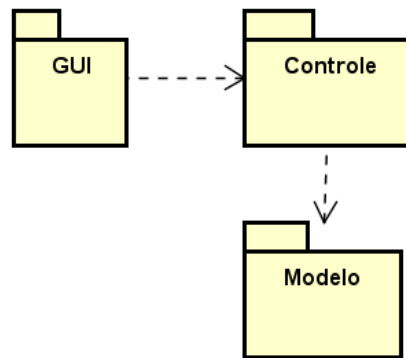
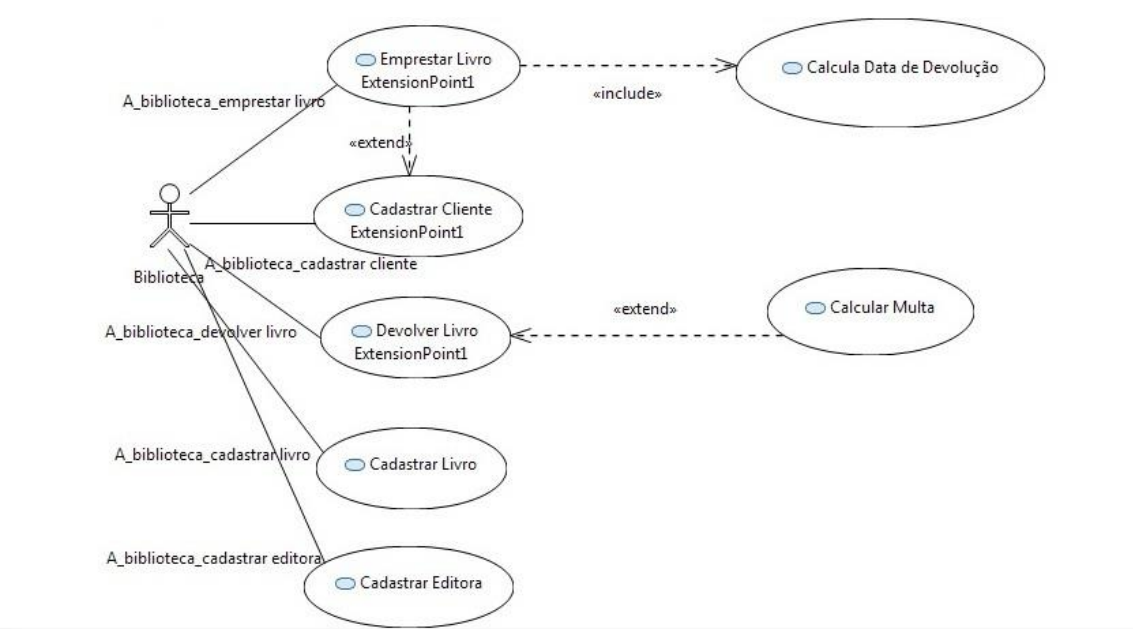


Figura 9.35: Diagrama Apresentado apenas Pacotes

Exercícios

Considere o seguinte modelo de casos de uso do sistema de Biblioteca:



Para o caso de Uso Calcular Data de Devolução, considere a seguinte descrição:

Calcular Data de Devolução	
- Fluxo Principal	- Fluxo Alternativo
1. Pega o número de livros do empréstimo.	2.a Caso o cliente empreste 3 ou mais livros.
1.1 Pega o prazo de devolução de cada livro.	2.a.1 - Adiciona mais 2 dias para cada livro. após o 2º livro emprestado.
1.2 Calcula a data de devolução do livro.	2.a.2 - Calcula a nova data.
2. Seleciona o maior prazo dentre todos os livros.	2.a.3 - Retorna ao passo 3.
3. Retorna a data de devolução dos livros.	

Considerando as descrições acima e o diagrama de classes apresentado na Figura 9.20, faça:

- 1) Aplique o padrão *Grasp Expert* para o caso de uso Calcular Data Devolução da seguinte forma:

- a. Gere o diagrama de comunicação, definindo os métodos de cada classe;
 - b. Atualize o diagrama de classe de análise, com os métodos encontrados;
 - c. Implemente o caso de uso, de acordo com as regras definidas no caso de uso, com o diagrama de comunicação e o diagrama de classes, e verifique se funciona o modelo criado. *(Utilize o código disponível em `CodigoBibliotecaExercicio1.rar`, disponível na pasta materiais - Precisa apenas implementar as partes faltantes, indicadas nos comentários).*
- 2) Agora, vamos expandir a implementação, trabalhando com o caso de uso Emprestar Livro:
 - a. Aplique o Padrão *Grasp Expert* e *Grasp Creator* no caso de uso Emprestar Livro (Quadro 9.3).
 - b. Atualize o diagrama de classe, com os métodos encontrados;
 - c. Implemente o caso de uso, de acordo com as regras definidas no caso de uso, com o diagrama de comunicação e o diagrama de classes, e verifique se funciona o modelo criado. *(Utilize o código disponível em `CodigoBibliotecaExercicio2.rar`, disponível na pasta materiais - Precisa apenas implementar as partes faltantes, indicadas nos comentários).*
- 3) Aplique o padrão de Coesão e Acoplamento e verifique se novas classes devem ser criadas no modelo proposto.
- 4) Na Figura 9.18, 9.19 e 9.20 é descrita uma solução para um baixo acoplamento na devolução de Livro. No entanto, esta solução não levou em conta a coesão das classes. Considerando a devolução de livros, este modelo está adequadamente coeso?
 - a. Descreva um caso de uso para a devolução de Livros.
 - b. Aplique o Padrão *Grasp Expert* e *Grasp Creator* no caso de uso.
 - c. Atualize o diagrama de classe, com os métodos encontrados e com novas classes caso considere que existem classes no sistema que não estão coesas.
- 5) Para o sistema de Biblioteca implementado no Exercício 2, aplique o padrão controlador e faça:

- a. Crie uma interface gráfica onde serão inseridos os empréstimos (mostre os dados dos livros a serem emprestados - será necessário criar a classe título, autor e área);
 - b. Crie um arquivo texto que contenha as informações sobre os livros;
 - c. Crie uma classe controladora para controlar os empréstimos;
 - d. Persista os empréstimos em um arquivo;
- 6) Atualize o Modelo da Figura 9.26, para a nova versão depois de todas as mudanças. Pense quais classes de interface seu sistema deveria ter, e considere o padrão de uma classe controladora por caso de uso.
- 7) Considerando a descrição do sistema de estacionamento, apresentado anteriormente, faça:
 - a. Aplique o padrão *Expert* e *Creator* nos casos de uso descritos anteriormente;
 - b. Veja se a coesão e acoplamento estão adequados
 - c. Atualize o diagrama de classes para que reflita as alterações encontradas.
- 8) Considerando a descrição do sistema de distribuidora de produtos, e os casos de uso Atender Pedido e Gerar Requisição descritos abaixo, e o diagrama de classes de análise faça:
 - a. Aplique o padrão *Expert* e *Creator* nos casos de uso descritos.
 - b. Veja se a coesão e acoplamento estão adequados.
 - c. Atualize o diagrama de classes para que reflita as alterações encontradas.
 - d. Aplique o Padrão *Controller*.
 - e. Organize o modelo em pacotes.

Caso de Uso: Atender Pedido	
Fluxo Principal	Fluxos Alternativos
1. O cliente informa os seus dados. 2. O sistema verifica se o cliente está cadastrado. 3. O sistema cria uma instância do pedido com situação "Pendente" associando-a ao cliente. 4. Para cada item pedido 4.1. O cliente informa o produto e a quantidade desejada. 4.2. O sistema verifica se o produto existe no cadastro. 4.3. O sistema cria uma instância do item pedido com situação "Pendente".	2.a Cliente não Cadastrado 2.1. O sistema emite msg01 "Cliente não cadastrado" 2.2. Abandonar o use case.
	4.2.a Produto não Cadastrado 4.2.1. O sistema emite msg01 "Produto não cadastrado" e retorna ao passo

Caso de Uso: Gerar Requisição	
Fluxo Principal	Fluxos Alternativos
<p>1. 1. Para cada pedido com situação “Pendente”</p> <p>1.1. Para cada item pedido com situação “Pendente”</p> <p>1.1.1. O sistema obtém o fornecedor associado ao produto.</p> <p>1.1.2. O sistema verifica se existe uma instância de requisição para o Fornecedor</p> <p>1.1.2.1. Caso não exista o sistema cria uma instância de requisição com situação “Pendente” associando-a ao Fornecedor</p> <p>1.1.3. O sistema cria instância de item requisição associando-a à requisição.</p> <p>1.1.4. O sistema altera a situação do item pedido para “Requisitado”.</p> <p>2. O sistema altera a situação do pedido para “Requisitado”.</p> <p>3. O sistema envia as requisições para os fornecedores.</p>	<p>1.1.1..a Não existe Fornecedor para o Produto</p> <p>1.1.1.1. O sistema emite msg04 “Produto sem Fornecedor!”.</p> <p>1.1.1.2. O sistema continua com o próximo item pedido no passo 1.1.1 ..</p>
	<p>1.1.3.a Já existe item requisição associado à requisição com situação “Pendente”</p> <p>1.1.3.1. O sistema adiciona a quantidade do produto ao item requisição.</p>

