Projeto de *Software* usando UML

Sumário

Capítulo I : Casos de Uso

1 Modelo de Casos de Uso

O modelo de Casos de Uso foi proposto por I. Jacobson como um instrumento para descrição das intenções ou requisitos para um sistema computacional. A construção do Modelo de Casos de Uso corresponde a uma das fases iniciais de um projeto de *software* pois envolve a determinação dos usos que o sistema terá, ou seja, do que ele deverá fornecer como serviços.

O modelo de Casos de Uso é diferente da visão funcional utilizada no passado nas abordagens de projeto estrututado. Ao invés de focalizar as funções (atribuições técnicas) do sistema, o modelo de Casos de Uso captura os usos ou aplicações completas do sistema. Este modelo busca responder a questão: Que usos o sistema terá? ou Para que aplicações o sistema será empregado?

Os modelos de Casos de Uso são descritos através de Diagramas de Casos de Uso na UML. De uma forma geral, cada projeto de *software* conterá um Diagrama de Casos de Uso. Para sistemas mais extensos, é possível decompor o diagrama em um conjunto de subdiagramas.

Uma vez construído o modelo de Casos de Uso, o restante do projeto pode ser guiado baseando-se neste modelo. Dito de outra forma, a partir do modelo de Casos de Uso, o restante do projeto irá se preocupar com a forma de realização dos casos de uso (que classes e objetos são necessários, como e quando cada um atuará, etc).

O modelo de Casos de Uso é um instrumento eficiente para determinação e documentação dos serviços a serem desempenhados pelo sistema. Ele é também um bom meio para comunicação com os clientes no processo de definição dos requisitos do sistema.

2 Diagramas de Casos de Uso

Os casos de uso de um projeto de *software* são descritos na linguagem UML através de **Diagramas de Casos de Uso**. Estes diagramas utilizam como primitivas Atores, Casos de Uso e Relacionamentos. Como ocorre também com outros diagramas, pode-se ainda utilizar as primitivas Pacote e Nota nos diagramas de casos de uso. As subseções a seguir descrevem estas primitivas.

2.1 Atores

Atores são representações de entidades externas mas que interagem com o sistema durante sua execução. Basicamente, a interação de atores com o sistema se dá através de comunicações (troca de mensagens). As entidades externas representadas pelos atores podem ser :

- Pessoas: usuário, secretária, aluno, professor, administrador,etc.
- Dispositivos: impressora, máquina ou outro equipamentos externo.
- *Hardwares*: placa de modem, placa de controle, etc.
- *Software*: sistema de banco de dados, aplicativos, etc.

É importante observar que atores representam, na verdade, papéis desempenhados por pessoas, dispositivos ou outros *softwares* quando estiverem interagindo com o sistema. Por exemplo, um ator cujo identificador seja Aluno não representa um aluno específico mas sim um aluno qualquer, ou seja, uma pessoa qualquer que esteja interagindo com o sistema na qualidade de aluno. Desta forma, um ator pode representar um entre vários indivíduos, equipamentos ou *softwares*. De forma análoga, uma entidade externa pode ser representada na forma de vários atores. Isto ocorre quando a entidade tem mais de um papel (tipo de participação ou interação) no sistema. Por exemplo, o indivíduo João da Silva poderia ser representado em um sistema na forma do ator Usuário, pois ele interage com o sistema nesta qualidade, e também na forma do ator Administrador, pois ele também interage com o sistema para este outro fim que é a administração do *software*.

Atores são representados através de retângulos (notação genérica para classe) usando a simbologia padrão da UML ou através de ícones humanos. As duas notações são sintaticamente equivalentes, porém a segunda é seguramente mais intuitiva. A desvantagem do uso de ícones humanos ocorre quando o ator representa equipamentos, dispositivos de *hardware* ou outros *softwares*. Neste caso, a figura humana não coincide com a natureza do ator. É possível, entretanto, através de mecanismos de extensão, criar grafismos especiais ou especializados na UML para indicar tipos de atores.

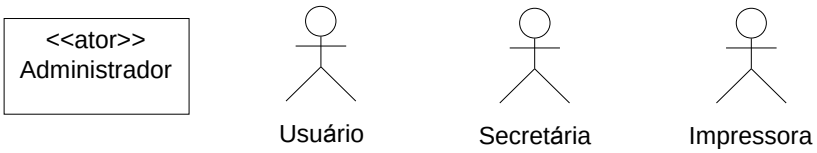

Figura I.1 – Exemplos de Atores

Observe que a notação usando retângulos exige a inserção de um classificador para indicar a natureza daquilo que se está representando. No caso de atores, deve-se incluir o classificador (ou estereótipo) <<ator>> antes do nome do ator. Quando se utiliza o ícone humano, basta indicar o nome do ator abaixo do ícone.

O levantamento dos atores que interagem com um certo sistema depende de um trabalho de estudo que envolve discussões com os clientes. Procura-se neste estudo levantar as pessoas que serão beneficiadas e que usarão o sistema. Além disso, deve-se levantar os dispositivos e *softwares* com os quais o sistema deverá se comunicar. Para muitos projetos, pode não ser fácil levantar corretamente o conjunto de atores na primeira tentativa. O estudo dos casos de uso e dos relacionamentos com atores pode permitir refinar o conjunto de atores definidos. O estudo das classes do sistema, a ser feito na próxima fase, também irá contribuir para o refinamento do levantamento de atores.

Embora a UML não imponha restrições, costuma-se considerar determinados atores como atores implícitos. Desta forma estes atores não aparecem no diagrama de casos de uso embora eles estejam presentes e participem da execução dos casos de uso. Os atores implícitos representam essencialmente dispositivos e *softwares* que são sempre usados e que não impõem protocolos especiais de comunicação. Desta forma, a supressão deste atores não traz nenhum efeito significativos sobre os modelos e simplifica as representações. Os exemplos mais comuns de atores que se pode considerar como implícitos são : monitor de vídeo, teclado, mouse, auto-falante, microfone, unidade de disco e sistema operacional. Estas entidades serão atores legítimos mas cuja inclusão no modelo de casos de uso não traz contribuição para a modelagem.

2.2 Casos de Uso

A descrição dos serviços a serem oferecidos pelo sistema é feita na UML discriminando-se os Casos de Usos do sistema. Cada Caso de Uso descreve uma aplicação ou uso completo do *software*. O conceito de caso de uso não deve ser confundido com o de módulo já que um caso de uso não é um componente do sistema mas sim um de seus empregos possíveis. Também não se deve confundir o conceito de caso de uso com o de função que possui um escopo muito mais limitado, traduzindo freqüentemente apenas um recurso ou utilidade do sistema. Um caso de uso é muito mais abrangente, envolvendo todo um conjunto de transações que juntas constituem um serviço completo oferecido pelo sistema.

A especificação das funcionalidades de um sistema na forma de casos de uso permite uma visão mais abrangente das aplicações do sistema favorizando um levantamento mais completo e preciso de suas atribuições.

2.3 Relacionamentos entre Atores e Casos de Uso

Os relacionamentos em um diagrama de casos de uso podem envolver dois atores, dois casos de uso ou um ator e um caso de uso, conforme descrito nas subseções seguintes.

2.3.1 Relacionamentos entre Atores

Como os atores são entidades externas, os relacionamentos entre eles serão relações externas ao sistema. Embora estas relações possam ser desprezadas, pois não fazem parte do sistema e nem são de conhecimento do sistema, é possível incluí-las no modelo de casos de uso. De certa forma, estas relações descrevem parte do modelo de negócios da empresa.

As duas relações mais comuns entre atores são a comunicação (ou associação) e a especialização (ou generalização). Um relacionamento de comunicação indica que os dois atores, de forma uni ou bidirecional, realizam uma comunicação (troca de informação ou de mensagem) que possui um significado formal para o sistema modelado. No exemplo da figura I.2, o ator Aluno comunica-se com o ator Secretaria no sentido que transmitir uma Solicitação de Histórico Escolar. Trata-se de uma comunicação explícita cuja ocorrência deve ter alguma repercussão ou significado para o sistema. Um relacionamento de generalização, por outro lado, representa uma relação conceitual entre atores indicando que um ator é um caso especial de outro ator mais genérico. Esta relação indica que o ator especializado inclui todos os atributos do ator genérico e inclui ainda atributos adicionais. Como ilustra a figura I.2, o ator Administrador é um ator especializado do ator Usuário. Isto significa que o Administrador é um Usuário com atributos ou características extras. De certa forma, o ator especializado estende o ator genérico.

Figura I.2 – Exemplos de Relações entre Atores

2.3.2 Relacionamentos entre Atores e Casos de Uso

O relacionamento entre um ator e um caso de uso expressa sempre uma comunicação entre eles, pois o ator sendo uma entidade externa não poderia possuir qualquer relacionamento estrutural com o sistema computacional. A notação UML para este tipo de relacionamento é um segmento de reta ligando ator e caso de uso, como ilustrado na figura I.3. Em diagramas mais complexos pode-se utilizar cadeias de segmentos de retas para se evitar cruzamentos.

Como atores podem ter vários propósitos, no que diz respeito a suas interações com o sistema, podem existir mais de um relacionamento de um ator com diferentes casos de usos. De forma análoga, um mesmo caso de uso pode envolver a participação de mais de um ator. A figura I.3 ilustra estas situações. O caso de uso Emitir Histórico Escolar envolve a participação de dois atores : Secretaria e Impressora. O ator Secretaria participa em dois casos de uso: Emitir Histórico e Registrar Novo Aluno.

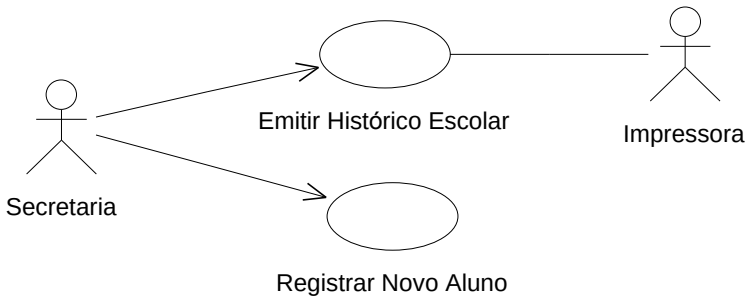

Figura I.3 – Exemplos de Relações entre Atores e Casos de Uso

A utilização de setas nas relações entre atores e casos de uso pode ter duas interpretações distintas. Na primeira, utilizam-se setas para indicar qual ator ativa o caso de uso. Ativação neste caso significa, quem lança ou inicia a execução do caso de uso. Para sistemas interativos, freqüentemente é o ator Usuário quem ativa o caso de uso. Na situação mais comum, cada caso de uso só teria um ator contendo uma seta em seu relacionamento de comunicação. É possível, entretanto, haver mais de um ator ativador para um mesmo caso de uso significando que um deles ativaria este caso de uso. O exemplo na figura I.3 aplica esta interpretação das setas. A segunda interpretação para as setas é a indicação do sentido do fluxo de dados nas comunicações. Neste caso todas as relações entre atores e casos de uso teriam setas em uma ou nas duas extremidades descrevendo o sentido da comunicação com cada ator. As duas interpretações são possíveis mas deve-se optar por uma delas em cada projeto e indicar explicitamente a interpretação adotada.

2.3.3 Relacionamentos entre Casos de Uso

Ao contrário do relacionamento entre ator e caso de uso, as relações entre casos de uso nunca serão do tipo comunicação. Isto ocorre porque casos de uso são aplicações completas do sistema e, por conseqüência, não existe sentido em fazer-se comunicar dois “usos do sistema”. Todas as relações entre casos de uso serão sempre estruturais. Existem três tipos de relações entre casos de uso (inclusão, extensão e generalização) conforme descritos a seguir.

§ Relacionamento de Inclusão

Um relacionamento de inclusão é uma relação estrutural através da qual um caso de uso insere em seu interior um outro caso de uso. O caso de uso incluído (subcaso de uso) não

representa um serviço completo do sistema mas uma porção de um serviço. Isoladamente, um subcaso de uso não teria sentido. Ele será sempre um integrante de um caso de uso maior e completo.

O relacionamento de inclusão se aplica a duas situações principais. A primeira aplicação da Inclusão é para o detalhamento de casos de uso através de decomposição. Nesta situação, extraem-se partes significativas de um caso de uso criando-se subcasos de uso ligados ao caso de uso maior através de relações de inclusão. Embora raro, é possivel ainda decompor subcasos de uso em outros subcasos de uso. Uma segunda aplicação do relacionamento de inclusão é a de colocar em evidência partes comuns a dois ou mais casos de uso. Nesta situação o subcaso de uso é incluído por mais de um caso de uso maior. A figura I.4 ilustra estas duas situações.

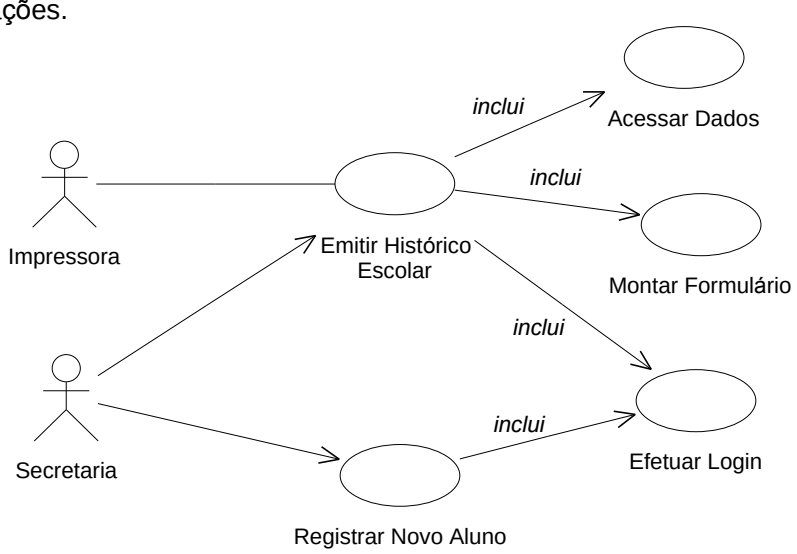

Figura I.4 – Exemplo de Relacionamento de Inclusão entre Casos de Uso

§ Relacionamento de Extensão

Um relacionamento de extensão é uma relação estrututal entre dois casos de uso através da qual um caso de uso maior é estendido por um caso de uso menor. A extensão significa que o caso de uso que estende inclui serviços especiais em um caso de uso maior. A definição de um relacionamento de extensão inclui a especificação de uma condição de extensão. Esta condição habilita a extensão, ou seja, indica quando aplicar o relacionamento. A notação para o relacionamento de extensão é ilustrada na figura I.5.

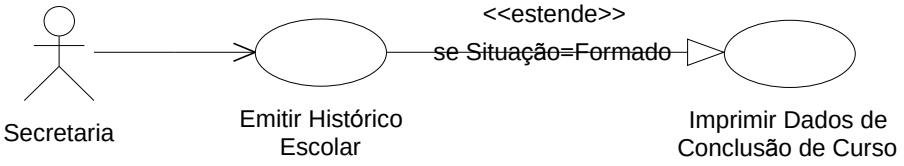

Figura I.5 – Exemplo de Relacionamento de Extensão entre Casos de Uso

A diferença principal entre os relacionamentos de inclusão e extensão é o caráter de excepcionalidade da extensão. Extensões são usadas para modelar casos especiais e de exceção que ocorrem somente em certas situações (dado pela condição). Desta forma, para a maioria das ocorrências do caso de uso estendido, a extensão não se aplicará.

§ Relacionamento de Generalização

Um relacionamento de generalização é uma relação estrutural entre um caso de uso mais geral e um caso de uso mais específico. O caso de uso mais geral representa o caso genérico cujo serviço se aplica a várias situações. O caso de uso mais específico representa a aplicação do caso uso mais geral em uma situação particular incluindo elementos adicionais ou estendendo este caso. Visto no outro sentido, o caso de uso mais geral é uma generalização (abstração) do ou dos casos de uso mais específicos. Neste sentido, o caso de uso geral, representa as partes comuns de casos de uso especializados.

A notação UML para a generalização envolve a ligação dos dois casos de uso através de um segmento de reta e a colocação de um triângulo na extremidade do caso de uso mais geral. A figura I.6 apresenta um exemplo de relacionamento de generalização.

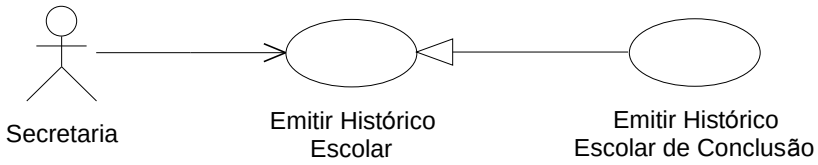

Figura I.6 – Exemplo de Relacionamento de Generalização entre Casos de Uso

No exemplo da figura I.6, tem-se um caso de uso mais geral, chamado Emitir Histórico Escolar, que permite a secretaria imprimir históricos de alunos. Quando o aluno conclui na totalidade o curso, ele pode solicitar um histórico de conclusão. Este histórico é semelhante ao histórico normal mas é mais detalhado incluindo informações adicionais. O caso de uso Emitir Histórico Escolar de Conclusão é portanto semelhante ao caso de uso Emitir Histórico Escolar mas com alguns elementos adicionais. Pode-se, como feito no exemplo, estabelecer uma relação de especialização entre os dois casos de uso.

A relação estrutural definida por um relacionamento de generalização implica a incorporação (herança) dentro do caso de uso especializado de todo o serviço especificado pelo caso de uso geral, incluindo, adaptando ou excluindo alguns serviços oferecidos pelo caso de uso geral.

O relacionamento de generalização não pode ser confundido com os de inclusão e extensão pois a generalização se aplica, na maior parte dos casos, a compartilhamentos de maior dimensão. A inclusão e extensão envolvem partes menores de casos de usos. A natureza da generalização também é distinta pois trata-se de especificar modelos (casos de uso) genéricos aplicáveis a diferentes situações. A inclusão e a extensão apenas põem em evidência partes de casos de uso maiores.

2.4 Decomposição de Diagramas de Casos de Uso

Um projeto de *software,* normalmente, conterá um único Diagrama de Casos de Uso descrevendo o conjunto de serviços oferecidos pelo sistema. Para sistemas maiores ou mais complexos, entretanto, é possível a construção de vários diagramas de casos de uso elaborados a partir da decomposição de um diagrama principal.

A decomposição de um diagrama de casos de uso pode ser feita em UML utilizando o conceito de Pacote (*package*). Um pacote é um encapsulador que não possui uma semântica específica dentro dos projetos. Ele é usado essencialmente para separar ou agrupar elementos do projeto sem criar necessariamente com isso algum vínculo entre os elementos.

Utilizando pacotes pode-se criar um primeiro diagrama contendo todos os pacotes maiores do sistema e, a seguir, tomar cada pacote e expandí-lo em um novo diagrama. Pode-

se construir uma hierarquia com vários níveis de decomposição conforme a dimensão do sistema e o número de casos de uso e atores.

Os elementos (casos de uso e atores) dentro de um pacote podem ter relacionamentos com elementos de outros pacotes. Neste caso existe uma dependência entre pacotes. As dependências devem ser explicitamente definidas utilizando como notação um segmento de reta tracejado com uma seta no sentido do pacote que depende para o pacote que contém as dependências. A figura I.7 ilustra a notação utilizada para pacotes e dependências.

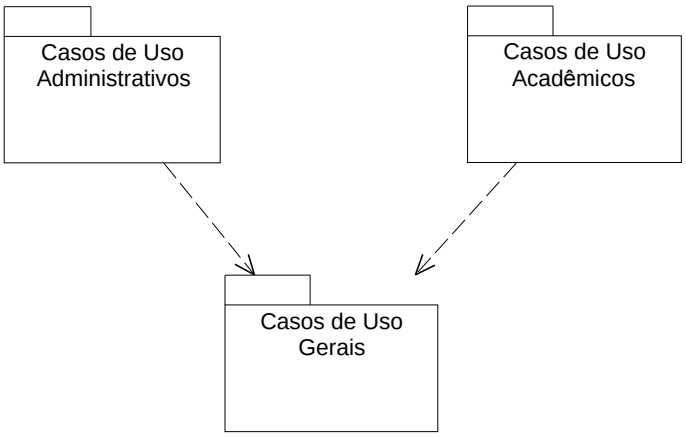

Figura I.7 – Exemplo de Pacotes e Dependências

Não existe uma norma para separação dos casos de uso e atores em pacotes. Pode-se, por exemplo, agrupar dentro de um pacote casos de uso de naturezas semelhantes (casos de uso de cadastro, casos de uso de emissão de relatórios, etc) ou casos de uso envolvendo os mesmos atores. De forma geral, procura-se reunir casos de uso que possuem relacionamentos em um mesmo pacote.

Quando um ator ou caso de uso tiver que aparecer em mais de um pacote, define-se este ator ou caso de uso em um pacote e copia-se o ator ou caso de uso nos demais pacotes. Neste caso, deve-se indicar nos demais pacotes qual o pacote de origem daquele ator ou caso de uso.

3 Exemplo

Para ilustrar a aplicação do conceito de caso de uso, desenvolve-se nesta seção um exemplo de modelagem para um sistema de controle acadêmico. Embora, o desenvolvimento completo de um modelo de casos de uso possa envolver várias iterações de refinamento, para fins didáticos o exemplo deste seção apresentará a modelagem através de 4 fases.

Fase 1 : Levantamento dos Atores

O sistema de controle acadêmico considerado neste exemplo será utilizado na secretaria de um determinado curso. No que diz respeito aos indivíduos envolvidos, somente o pessoal da secretaria terá acesso ao sistema. Entre as pessoas que atuam na secretaria e poderiam utilizar o sistema estão o chefe da secretaria, a secretária, alguns professores e alguns estagiários. Na verdade, apesar de se tratarem de indivíduos diferentes, quando estiverem utilizando o sistema todos assumirão o mesmo papel, ou seja, todos atuarão na forma de um ator abstrato que pode ser denominado Secretaria.

Preliminarmente, supõe-se que alguns documentos deverão ser impressos pelo sistema, o que sugere a criação de um ator denominado Impresora com o qual o sistema irá interagir para a impressão de documentos (histórico escolar, diário de classe, etc). O ator impressora poderia ser considerado um ator implícito mas pode ser ilustrativo fazê-lo aparecer explicitamente no modelo.

Como o volume de informações (alunos, professores, disciplinas, etc) pode ser grande optou-se pelo uso de um Sistema Gerenciador de Banco de Dados para armazenamento dos dados acadêmicos. Como se trata de um sistema computacional independente com o qual o sistema de controle acadêmico irá interagir, ele deve ser considerado também como um ator. Neste exemplo, este ator será denominado SGBD.

Portanto, os atores que foram inicialmente levantados são:

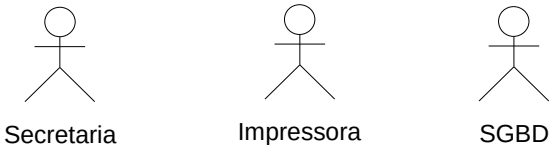

Figura I.8 – Atores do sistema de controle acadêmico.

Na prática, nem sempre é possível determinar todos os atores e definí-los corretamente na primeira tentativa. Se for considerada uma abordagem de projeto por refinamentos sucessivos, a lista de atores poderia ser melhorada, assim como a definição deste atores, a medida que o projeto avance quando mais informações estiverem disponíveis.

Fase 2 : Levantamento dos Casos de Uso Principais

Nesta fase busca-se definir a lista dos grandes serviços que o sistema deverá oferecer. O levantamento dos casos de uso corresponde a uma análise de requisitos e deve ser desenvolvido a partir de informações coletadas dos clientes. Através de questionários e reuniões com os clientes procura-se definir quais são as aplicações ou usos desejados para o sistema a ser desenvolvido.

Para o sistema de controle acadêmico considera-se que os clientes (usuários, professores e administração da escola) desejam que o sistema ofereça os seguintes serviços :

- possibilidade de cadastramento de todos os alunos matriculados no curso. Isto implica um serviço para inclusão de novos alunos e para manutenção da base de dados dos alunos. Este uso do sistema será representado pelo caso de uso Cadastrar Aluno.
- possibilidade de cadastramento de todos os professores que ministram disciplinas no curso. Isto implica um serviço para inclusão de novos professores e para manutenção da base de dados de professores. Este uso do sistema será representado pelo caso de uso Cadastrar Professor.
- possibilidade de registro das disciplinas oferecidas no curso incluindo o registro de novas disciplinas e a manutenção da base de dados de disciplinas. Este serviço ou uso do sistema será representado pelo caso de uso Cadastrar Disciplina.
- possibilidade de registro da matrícula de alunos em disciplinas a cada semestre. Este serviço será representado pelo caso de uso Registrar Matrícula.
- possibilidade de emissão da confirmação de matrícula para cada aluno contendo a lista de disciplinas nas quais um aluno se matriculou para aquele semestre. Este serviço será representado pelo caso de uso Emitir Confirmação de Matrícula.

- possibilidade de emissão do diário de classe para cada disciplina contendo a lista de alunos matriculados naquele semestre. Este serviço será representado pelo caso de uso Emitir Diário de Classe.
- possibilidade de lançamento das notas obtidas pelos alunos em cada disciplina ao final de cada semestre. Este serviço será representado pelo caso de uso Registrar Nota.
- possibilidade de emissão do histórico escolar para cada aluno contendo a lista de disciplinas cursadas e respectivas notas. Este serviço será representado pelo caso de uso Emitir Histórico Escolar.

O conjunto de casos de uso levantados representa os serviços ou usos esperado pelos clientes que utilizarão o sistema. Assim como para os atores, nem sempre é possível efetuar um levantamento completo e definitivo dos casos de uso em uma primeira tentativa. Ao longo do processo de refinamento, novos casos de uso poderiam aparecer ou outros sofrerem alterações.

A figura I.9 ilustra os casos de uso definidos para o sistema acadêmico.

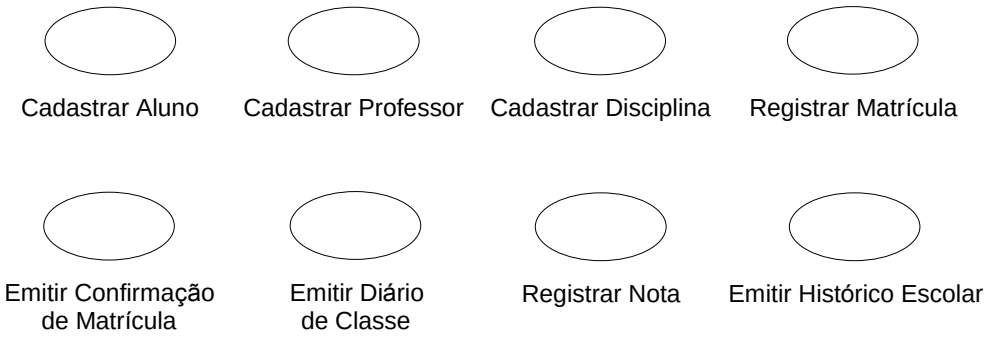

Figura I.9 – Casos de Uso do sistema de controle acadêmico.

Fase 3 : Definição dos Relacionamentos

Nesta fase são estabelecidos os relacionamentos de comunicação entre os atores e os casos de uso indicando quais atores participam (se comunicam) com quais casos de uso. Para o exemplo em estudo, o resultado seria aquele apresentado na figura I.10. Neste diagrama de casos de uso, adotou-se o uso de setas nas relações para indicar qual ator é responsável pela ativação dos casos de uso.

Fase 4 : Detalhamento dos Casos de Uso

Nesta fase é feito um detalhamento dos casos de uso através de decomposições ou especializacões. O grau de detalhamento necessário é um aspecto subjetivo. Cabe ao projetista julgar qual o bom nível de detalhamento para cada projeto. Não se deve exagerar nas decomposições sob o risco de se estar influenciando ou direcionando o processo de projeto. Deve-se lembrar que os diagramas de casos de uso são especificações do que o sistema deve fazer e não de como ele deverá realizar os serviços.

Como abordagem geral para esta fase existem as seguintes sugestões:

- Procure estimar a dimensão de cada caso de uso. Para casos de uso muito extensos, crie subcasos de uso que identifiquem partes do processo envolvido naquele caso de

uso. Relacione os subcasos de uso com caso de uso maior através de relações de inclusão.

Para o sistema de controle acadêmico, considerou-se que os três casos de uso para cadastramento (aluno, professores e disciplinas) têm uma dimensão maior e incluem serviços internos (inclusão, consulta, alteração e exclusão) que podem ser destacados. Assim, optou-se por decompor cada um destes casos de uso em quatro subcasos de uso.

- Compare par a par os casos de uso tentando identificar partes comuns nos serviços associados a cada caso de uso. Quando dois ou mais casos de uso possuem parte em comum de dimensão significativa, esta parte em comum pode ser colocada em evidência através de um subcaso de uso.

  Para o sistema de controle acadêmico, foi decidido que o usuário deverá se identificar através de nome e senha para ter acesso aos serviços de cadastramento e registro de matrícula e notas. Assim, todos os casos de uso associados teriam uma fase inicial idêntica em seus processos que corresponderia a realização de *login*. Esta parte comum pode ser indicada através de um subcaso de uso comum.

- Quando dois ou mais casos de uso tiverem grande parte de seus serviços semelhante, verifique a possibilidade de definição de um caso de uso geral que cubra a parte comum deste casos de uso. Especifique, então, um relacionamento de generalização entre o caso de uso geral e os casos de uso especializados.

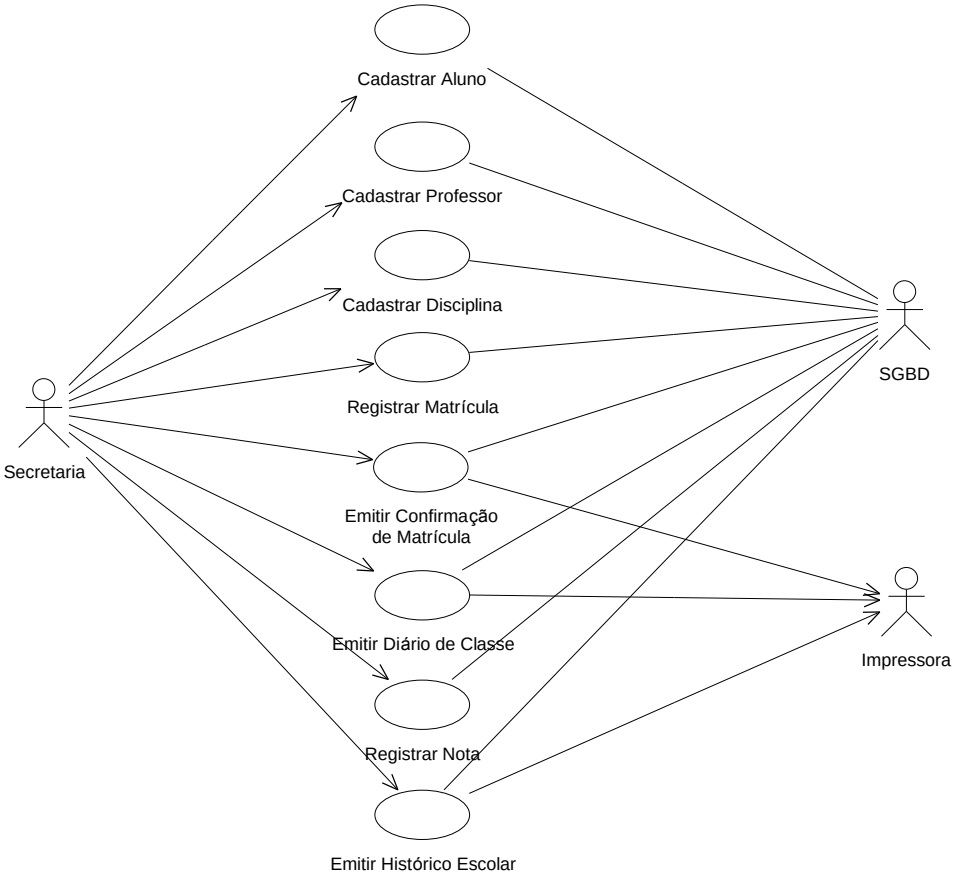

Figura I.10 – Relacionamentos entre Atores e Casos de Uso.

A figura I.11 apresenta o diagrama de casos de uso final para o sistema de controle acadêmico. Na verdade, este diagrama poderia ser melhorado e detalhado a partir das primeiras iterações do projeto. À medida que o projeto avance e a forma de realização dos casos de uso seja definida, pode-se verificar a necessidade de alteração ou adaptação do diagrama de casos de uso.

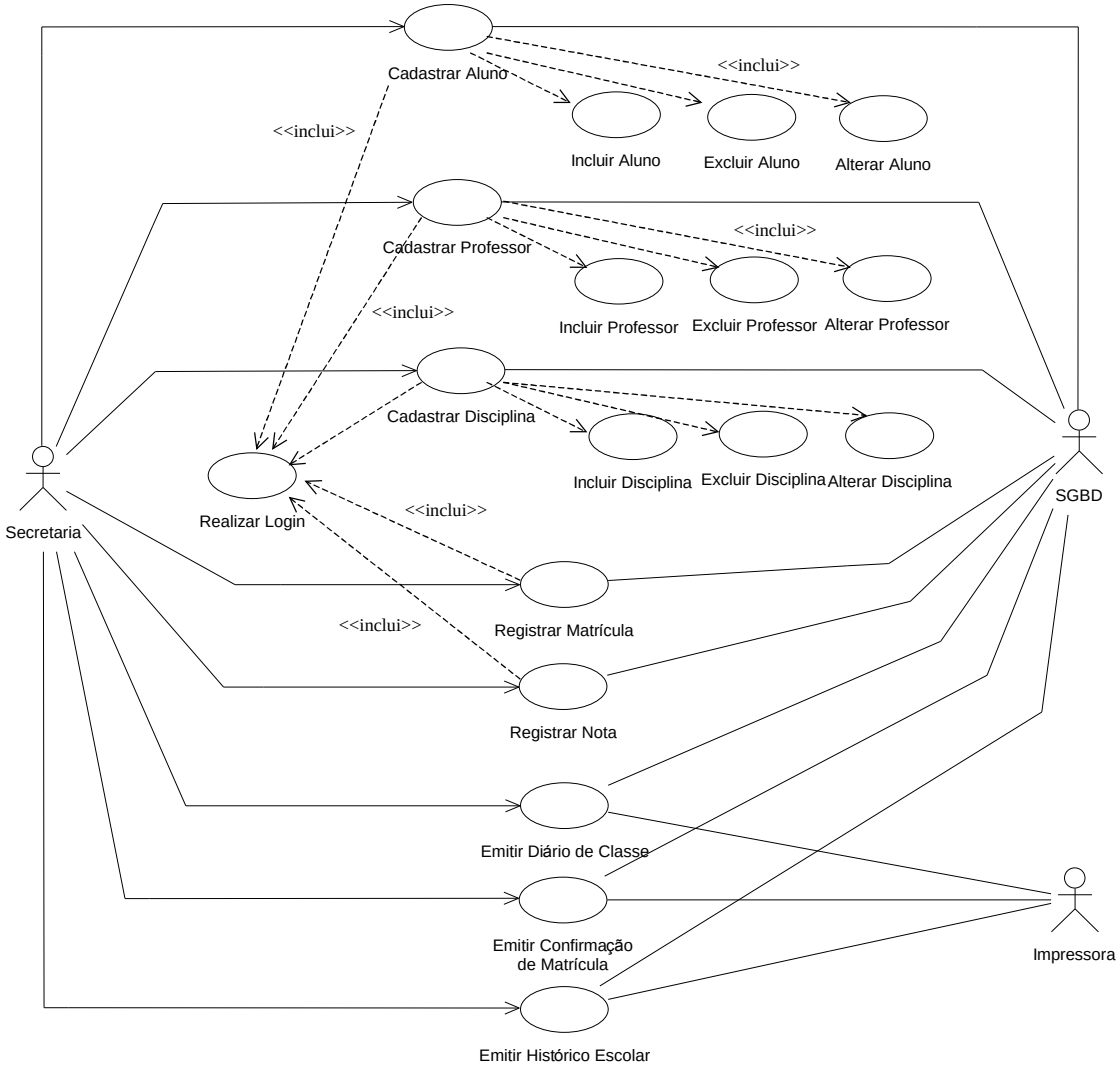

Figura I.10 – Diagrama de casos de uso final para o sistema de controle acadêmico.

4 Conclusão

O modelo de casos de uso é uma ferramenta útil na descrição dos requisitos funcionais de um sistema computacional. Ele permite especificar o conjunto de funcionalidades ou serviços que um software deve oferecer e as relações do sistema com entidades externa (atores) necessárias para a realização destes serviços.

A notação UML para diagramas de casos de uso é em grande parte intuitiva permitindo que os modelos gerados possam ser apresentados aos clientes para discussões e revisões.

Deve-se sempre observar que o modelo de casos de uso é diferente da visão funcional no sentido que ele não apresenta encadeamentos de funções (por conseqüência, não descreve processos) e se limita a uma visão macroscópica dos serviços do sistema sem induzir a forma de realização (projeto) do *software*. O modelo de casos de uso oferece uma visão mais abstrata das funcionalidades do sistema favorizando um trabalho de especificação mais abrangente.

Por fim, o modelo de casos de uso pode também ser útil como ferramenta para planejamento do desenvolvimento de sistemas computacionais (estimativas por caso de uso) e como base para o desenvolvimento de projetos de *software* (projeto baseado em casos de uso).

Capítulo II : Levantamento de Classes

1 Conceito de Classe e Objeto

As classes e objetos são as principais primitivas ou elementos de composição de *softwares* orientados a objetos. Na verdade, um sistema orientado a objetos é composto por classes e por um conjunto de objetos que colaboram ou interagem para execução dos serviços (casos de uso) oferecidos pelo sistema. As seções seguintes apresentam resumidamente os conceitos de objeto e de classe.

1.1 Objetos

Um objeto pode ser visto ou entendido segundo uma visão conceitual e segundo uma visão de implementação. Estas duas visões não são antagônicas, na realidade, elas se complementam apresentando duas formas de interpretação dos objetos.

Visão conceitual

A visão conceitual é uma forma mais abstrata de se observar um objeto. Nesta visão um objeto é um elemento componente de um sistema computacional que representa, dentro do sistema, alguma entidade ou coisa do mundo real ou que define uma porção do sistema com limites e atribuições bem definidos.

Uma das intenções da orientação a objetos é tentar aproximar as representações internas de um sistema computacional da forma como as entidades existem e se relacionam no mundo real, sejam elas materiais ou abstratas. Considerando-se um sistema de controle acadêmico, por exemplo, existem no sistema de controle acadêmico no mundo real entidades materiais (secretária, professores, alunos, diários de classe, etc) e entidades abstratas (disciplina, notas, freqüência, etc) envolvidas. Quando se desenvolve um sistema orientado a objetos procura-se criar no interior do *software* representações das entidades externas na forma de objetos. Assim, poderiam ser criados objetos como aluno, diário de classe e disciplina para representarem entidades externas dentro do sistema. A intenção desta aproximação entre entidades externas e representação interna é facilitar a interpretação da estrutura do sistema computacional e agrupar informações que juntas têm um significado comum.

Objetos não representam apenas informações em um sistema computacional. Eles podem também representar, e implementar, processos (ou seja, algoritmos ou procedimentos). Considerando novamente o sistema de controle acadêmico, o trabalho manual de composição de um diário de classe poderia ser representado no sistema computacional por um objeto responsável por este processamento.

Além das responsabilidades individuais, as entidades no mundo real praticamente sempre se relacionam de forma a juntas cooperarem para a execução de tarefas. Por exemplo, os dados dos alunos (entidades) são utilizados pela secretaria (entidade) para compor (processo) o diário de classe (entidade) para uma disciplina (entidade). De forma análoga, os objetos que representam estas entidades irão se relacionar para juntos cooperarem na execução de serviços (casos de uso). Um exemplo de relacionamento é a comunicação, através da qual um objeto pode trocar informações ou comandar outro objeto.

Um objeto possui três características :

- **Identidade** : cada objeto possui um nome ou identificador que o distingue dos demais objetos e que permite que os outros objetos o reconheçam e o enderecem.
- **Atributos** : cada objeto pode possuir informações que são registradas na forma de um conjunto de atributos.
- **Comportamento** : cada objeto pode possuir um conjunto de habilidades de processamento que juntas descrevem seu comportamento.

Como exemplo considere um objeto que represente uma disciplina de matemática. A identidade deste objeto poderia ser “Cálculo Numérico”. Seus atributos poderiam ser “Carga Horária”, “Pré-requisitos”, “Ementa”, “Alunos Matriculados” e “Notas”. Seu comportamento poderia incluir a capacidade de “Calcular a Média da Turma”, “Inscrever um Novo Aluno” e “Registrar Notas dos Alunos”.

Visão de Implementação

Em termos de implementação, um objeto pode ser entendido como um pequeno módulo de *software*. Neste sentido, um objeto pode ser executado como qualquer módulo de *software*, pode receber dados de entrada e pode produzir resultados. Na orientação a objetos, procura-se construir estes módulos de *software* (objetos) de forma que eles tenham um alto grau de modularidade (pequena dependência entre um módulo e outro) e que cada módulo possua atribuições ou responsabilidades próprias que o caracterize.

Enquanto nas linguagens de programação procedurais como Pascal e C, os programas são construídos como conjuntos de funções ou procedimentos e variáveis, nas linguagens orientadas a objetos os programas são construídos pensando-se em agrupamentos de funções e variáveis formando objetos. Pode-se considerar que na orientação a objetos foi introduzido um novo nível de estruturação nos programas conforme discutido na próxima seção.

Em termos de implementação, um objeto pode interagir com outros objetos. Por exemplo, um objeto poderia se comunicar com outro. Esta comunicação, como será vista em detalhes na seção 3, poderia ser realizada através de chamada de função ou procedimento do objeto para algum outro ou através de um mecanismo de envio de mensagem suportado pela maioria dos sistema operacionais. Desta forma, a execução de um *software* orientado a objetos seria feita através da execução em cadeia ou em concorrência do conjunto de seus objetos que permanentemente estariam se comunicando para trocar informações ou comandar a execução de funções. Pode-se dizer, então, que a execução de um *software* orientado a objetos se dá pela colaboração de um conjunto de objetos.

As três características de um objeto (identidade, atributos e comportamento) se traduzem da seguinte forma na implementação :

- A identidade do objeto será determinada por um nome ou identificador inventado pelo programador para cada objeto. Por exemplo, um objeto que representa uma disciplina de matemática poderia ser nomeado “objMatematica”. Qualquer nome pode ser utilizado, desde que se respeite as normas para criação de identificadores da linguagem de programação utilizada.
- Os atributos de um objeto são traduzidos na forma de variáveis internas do objeto. Pode-se definir variáveis de qualquer tipo para composição dos atributos dos objetos. Por exemplo, um objeto representando um aluno poderia ter os atributos “nome” (texto com 30 caracteres), “sexo” (um caracter, M ou F), “coeficiente de rendimento” (valor real), entre

outros. Não se deve confundir aqui atributo com valor do atributo. Neste exemplo, “nome” é um atributo cujo valor pode ser “Marcia” ou “Carlos”.

- O comportamento do objeto é descrito por uma ou mais funções ou procedimentos. Estas funções descrevem os procedimentos ou habilidades do objeto. Por exemplo, o objeto objMatematica poderia ter uma função para cálculo da média das notas dos alunos matriculados.

1.2 Classes

As classes são elementos fundamentais na composição de *softwares* orientados a objetos. Elas são empregadas tanto na composição do projeto dos sistemas computacionais quanto em sua implementação. Entretanto, ao contrário dos objetos, as classes não se traduzem em elementos executores no código de implementação. O entendimento do conceito de classe pode ser feito também segundo um ponto de vista conceitual e um ponto de vista de implementação.

Visão conceitual

Em um ponto de vista conceitual, as classes podem ser entendidas como descrições genéricas ou coletivas de entidades do mundo real. Mantém-se aqui a intenção de aproximar entidades externas de representações internas. Desta forma, a definição das classes de um sistema deverá procurar inspiração nas entidades do mundo real.

Ao contrário dos objetos, que representam entidades individualizadas, as classes são representações genéricas. Elas são abstrações de um coletivo de entidades do mundo real. Isto significa que elas representam um modelo comum para um conjunto de entidades semelhantes. Imaginando, por exemplo, os alunos envolvidos em um sistema acadêmico, cada aluno (Marcia, Vinicius, Cláudia, etc) é uma entidade individualizada que poderia ser representada por um objeto. Observando-se estes objetos e comparando-os, pode-se constatar que o conjunto de seus atributos (nome, sexo, idade) e seus comportamentos são análogos, embora obviamente os valores dos atributos sejam diferentes.

A partir da observação de atributos e comportamento comum a um conjunto de entidades similares, é possível estabelecer um modelo genérico para este coletivo de entidades. Este modelo conteria os atributos e comportamento comuns a este coletivo. Desta forma, seria possível definir um modelo genérico para todos os alunos no exemplo anterior. Este modelo genérico é, em conseqüência, uma abstração dos alunos que denomina-se Classe na orientação a objetos.

Considera-se uma classe um modelo genérico porque ela estabelece um formato padrão para a representação, na forma de objetos, de todas as entidades externas associadas. A definição das classes de um sistema passa necessariamente pela observação das entidades externas buscando-se determinar coletivos de entidades semelhantes que possam ser abstraídas em uma representação genérica.

Visão de Implementação

A implementação de classes se faz através da criação de tipos, de maneira semelhante aos tipos de dados. As linguagens de programação, de uma forma geral, oferecem um conjunto de tipos de dados padrões, como inteiro, real, caracter, booleano e cadeia de caracteres, e permitem que o programador crie novos tipos como enumerações, estruturas, uniões e registros. A partir de um tipo de dado é possível, então, criar-se instâncias que são as variáveis dos programas. A figura II.1 mostra exemplos de variáveis criadas a partir de vários tipos de dados da linguagem C e a definição de um novo tipo chamado Aluno.

```
/* definição de um novo tipo de dado */
struct Aluno {
      int codigo;
      char nome[30];
      float coeficiente;
      char sexo;
};

/* definição de variáveis */
int         valor;
float       resultado;
char        texto[10];
Aluno       al;
```

Figura II.1 – Exemplo de definição de variáveis na linguagem C.

Embora possam conter mais do que apenas variáveis, as classes são também tipos e, por conseqüência, não são por elas próprias elementos executáveis ou que possam armazenar dados dentro de um programa. As classes, assim como os tipos de dados, são modelos para a criação de variáveis. Desta forma, a partir da criação de uma classe é possível criar instâncias ou variáveis desta classe. As variáveis criadas a partir de uma classe são denominadas "objetos". Isto coincide com a visão conceitual vista anteriormente onde definiu-se que classes eram modelos para objetos semelhantes. Em termos de implementação, uma classe é, de fato, um modelo (ou tipo) usado para criação de objetos.

Na orientação a objetos, impõe-se que todo objeto seja criado a partir de uma classe. Assim, a declaração das classes de um sistema deve ocorrer antes da definição dos objetos. A figura II.2 apresenta um trecho de programa na linguagem C++, ilustrando a declaração de uma classe (CAluno) e a definição de dois objetos (al e estudante) desta classe. Observe que a classe **CAluno** possui quatro atributos (nome, coeficiente, sexo e codigo) e uma função (definir). Na sintaxe da linguagem C++, a definição do corpo das funções é feita fora da declaração da classe. Desta forma, a função **definir** aparece declarada dentro da classe mas sua definição completa só aparece a seguir. Após a declaração da classe, objetos podem ser criados livremente, como é o caso dos objetos **al** e **estudante**.

```
/* definição de uma classe */
class CAluno {
      char nome[30];
      float coeficiente;
      char sexo;
public:
      int codigo;
      void definir(char *n, float c, char s, int i);
};

void Caluno::definir(char *n, float c, char s, int cod)
{
      strcpy(nome, n);
      coeficiente = c;
      sexo = s;
      codigo = cod;
}
/* definição de objetos */
CAluno al;
CAluno estudante;
```

Figura II.2 – Exemplo de definição de uma classe e de objetos na linguagem C++.

Em termos de implementação pode-se considerar que a orientação a objetos introduziu um nível adicional de estruturação nos programas. Na programação procedural os programas são organizados essencialmente em três níveis de estruturação: instruções, funções (ou procedimentos) e arquivos. Assim, o código fonte de uma programa é distribuído primeiramente em arquivos (.c, .h, .pas, etc), que por sua vez contém conjuntos de funções, que finalmente são descritas por um conjunto de instruções, conforme ilustrado na figura II.3. Na orientação a objetos os programas são organizados em arquivos, que contêm classes, que incluem funções que são descritas por conjuntos de instruções, conforme ilustra a figura II.4.

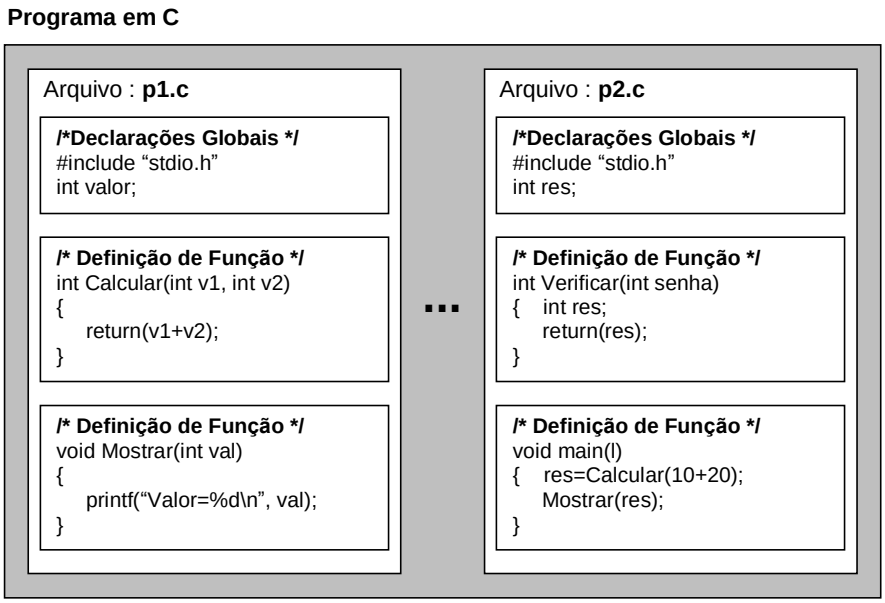

Figura II.3 – Exemplo de estruturação de um programa em linguagem C.

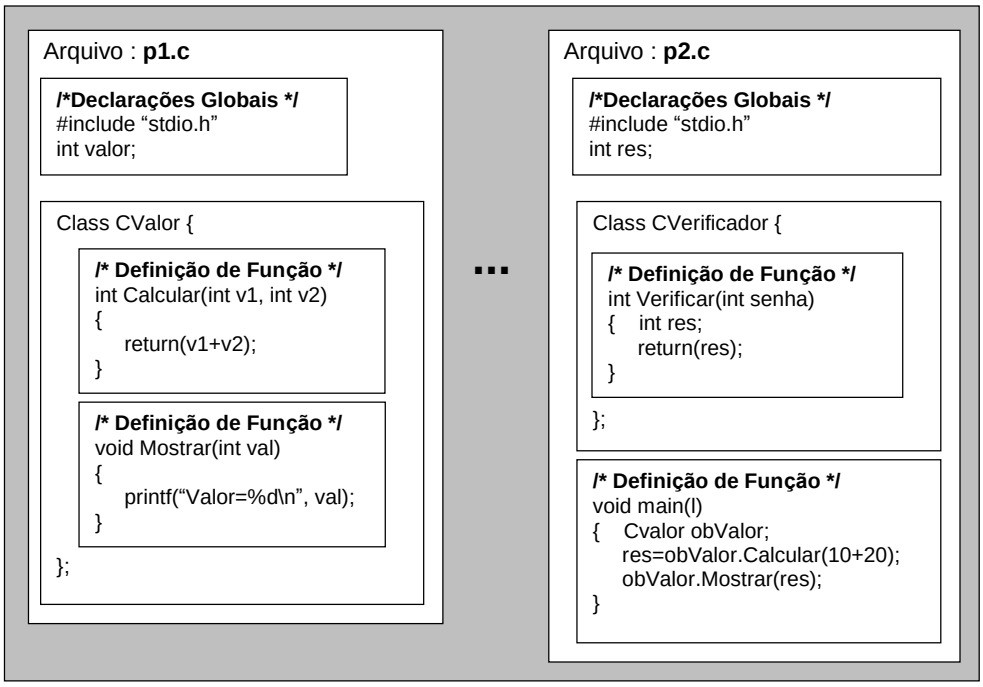

Figura II.4 – Exemplo de estruturação de um programa em linguagem C++.

2 Notação UML para Classes e Objetos

2.1 Notação para Classes

A notação para classes em UML é um retângulo com três compartimentos conforme ilustrado na figura II.5. Quando não se desejar mostrar detalhes da classe em um certo diagrama, pode-se suprimir a apresentação dos atributos e métodos, conforme ilustrado na classe da direita na figura II.5.

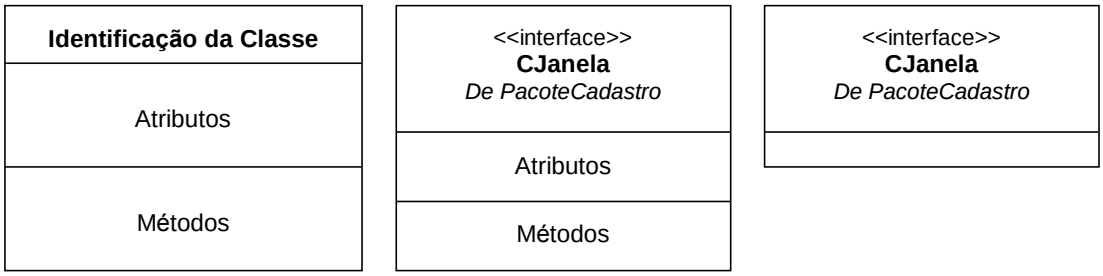

Figura II.5 – Notação UML para classes.

2.1.1 Compartimento de Identificação

O compartimento superior é utilizado para identificação da classe. Nele deverá aparecer, necessariamente, o nome da classe e, opcionalmente, um estereótipo e um identificador de pacote. O nome da classe é uma cadeia de caracteres que identifica exclusivamente esta classe dentro do sistema. Como sugestão, o nome da classe deveria utilizar apenas letras e palavras concatenadas mantendo-se a primeira letra de cada palavra em maiúscula. Muitos projetistas e programadores utilizam também a letra C (de classe) como prefixo para todos os nomes de classes.

Exemplos de nomes de classes :

- CAluno
- CturmaEspecial
- CJanelaDeInterfaceComUsuario

Um estereótipo pode ser incluído no compartimento de identificação de uma classe conforme ilustrado na figura II.5. Estereótipos são classificadores utilizados para indicar, neste caso, tipos de classes. Pode-se, por exemplo, criar classes do tipo interface ou do tipo controle. Para indicar o tipo de uma classe através de um estereótipo inclui-se o tipo da classe entre os sinais << >> acima de seu nome. No exemplo da figura II.5, a classe Cjanela foi definida como sendo do tipo <<interface>>.

Por fim, o compartimento de identificação pode incluir também o encapsulamento da classe, ou seja, o pacote dentro do qual a classe está declarada. O conceito de pacote é o mesmo que aquele discutido no capítulo 2.4. A especificação do pacote é incluída abaixo do nome da classe e escrita em itálico na forma *"de nome-do-pacote*".

2.1.2 Compartimento de Definição de Atributos

Neste segundo compartimento são definidos todos os atributos da classe. Atributos são variáveis membro da classe que podem ser usadas por todos os seus métodos tanto para acesso quanto para escrita. Em linguagem de programação, os atributos da classe são definidos a partir dos tipos padrões da linguagem e de tipos definidos pelo programador. Na UML tem-se uma notação similar a das linguagens de programação.

Notação para definição de atributos :

[Visibilidade] Nome-Atributo [Multiplicidade]:[Tipo]=[Valor] [{Propriedades}]

- **Visibilidade**

A visibilidade indica se o atributo pode ou não ser acessado do exterior da classe por funções que não sejam membro da classe. Existem três especificadores de visibilidade :

+ : indica visibilidade pública. Neste caso o atributo é visível no exterior da classe. Tanto funções membro da classe quanto funções externas podem acessar o atributo.

\- : indica visibilidade privada. Neste caso o atributo não é visível no exterior da classe. Somente funções membro da classe podem acessar o atributo.

\# : indica visibilidade protegida. Neste caso o atributo também é privado mas funções membro de classes derivadas também têm acesso ao atributo.

A definição da visibilidade do atributo é opcional. Entretanto, se ela não for especificada, assume-se por padrão visibilidade privada.

- **Nome-do-Atributo**

O nome do atributo é definido por uma cadeia de caracteres que identificam exclusivamente o atributo dentro da classe. Sugere-se o emprego unicamente de letras começando-se por uma letra minúscula e concatenando-se palavras mantendo a primeira letra em maiúsculo.

Exemplo : valorDoPagamento
nomeDoAluno

Alguns autores utilizam o prefixo “m_” (de minha) para todos os nomes de atributos.

Exemplo : m_valorDoPagamento
m_nome_doAluno

- **Multiplicidade**

A multiplicidade é uma especificação opcional na definição de um atributo. Na verdade, a multiplicidade é utilizada somente quando se tratar de atributo múltiplo ou seja vetores e matrizes. A multiplicidade indica portanto a dimensão de atributos e é descrita entre colchete.

Exemplo: valores[5] → vetor de 5 valores
matrizDeValores[10,20] → matriz de 10 linhas e 20 colunas

- **Tipo**

O tipo de um atributo é um classificador que indica o formato (classe ou tipo de dado) dos valores que o atributo pode armazenar. Pode-se, aqui, utilizar os tipos padrões da linguagem de programação que se pretende usar.

Exemplo: resultado: int
salario: float
sexo : char

- **Valor Inicial**

O valor inicial indica o valor ou conteúdo do atributo imediatamente após a sua criação. Deve-se observar que algumas linguagens de programação (como C++) não suportam esta atribuição inicial.

Exemplo: resultado: int = 10

- **Propriedades**

As propriedades podem ser utilizadas para incluir comentários ou outras indicações sobre o atributo como por exemplo se o seu valor é constante. As propriedades são indicadas entre chaves.

Exemplo :

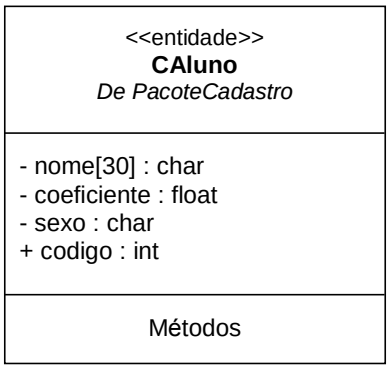

Figura II.6 – Exemplo de definição de atributos de uma classe

2.1.3 Compartimento de Definição de Métodos

Neste compartimento são declarados os métodos (ou seja, as funções) que a classe possui. A sintaxe da declaração pode seguir aquela da linguagem de programação que se pretende utilizar na implementação do sistema.

Notação para definição de atributos :

[Visibilidade] Nome-Método (Parâmetros):[Valor-de-Retorno] [{Propriedades}]

- **Visibilidade**

A visibilidade indica se o método pode ou não ser chamado do exterior da classe por funções que não sejam membro da classe. Existem três especificadores de visibilidade:

- **+** indica visibilidade pública. Neste caso o método é visível no exterior da classe. Tanto funções membro da classe quanto funções externas podem acessar ao método.
- \- indica visibilidade privada. Neste caso o método não é visível no exterior da classe. Somente funções membro da classe podem acessar ao método.
- # indica visibilidade protegida. Neste caso o método também é privado mas funções membro de classes derivadas também têm acesso ao método.

A definição da visibilidade do método é opcional. Entretanto, se ela não for especificada, assume-se por padrão visibilidade privada.

- **Nome-do-Método**

O nome do método é definido por uma cadeia de caracteres que identificam exclusivamente o método dentro da classe. Sugere-se o emprego unicamente de letras começando-se por uma letra maiúscula e concatenando-se palavras mantendo a primeira letra em maiúsculo.

Exemplo: CalcularValor
ArmazenarDados

- **Parâmetros**

Os parâmetros são variáveis definidas junto aos métodos e que são utilizadas pelos métodos para recebimento de valores no momento da ativação. Os parâmetros podem também ser utilizados para retorno de valores após a execução do método. Cada parâmetro é especificado usando a notação :

Nome-do-Parâmetro:Tipo=Valor-Padrão

O Nome-do-Parâmetro é uma cadeia de caracteres que identifica o parâmetro dentro do método. Tipo é um especificador de tipo de dado padrão da linguagem de programação. Valor-Padrão é a especificação de uma constante cujo valor será atribuído ao parâmetro se em uma chamada ao método não for especificado um valor para o parâmetro.

Exemplo: CalcularValor(val1:int, val2:float=10.0)
ArmazenarDados(nome:char[30], salario:float=0.0)

- **Valor-de-Retorno**

O Valor-de-Retorno indica se o método retorna algum valor ao término de sua execução e qual o tipo de dado do valor retornado. Pode-se utilizar aqui os tipos padrões da linguagem de programação que se pretende utilizar ou novos tipos definidos no projeto (inclusive classes).

Exemplo: CalcularValor():int // retorna um valor inteiro
ArmazenarDados(nome:char[30]): bool // retorna verdadeiro ou falso

2.2 Notação para Objetos

A notação UML para um objeto é um retângulo com a identificação do objeto em seu interior. A figura II.7 ilustra objetos representados na UML. A identificação do objeto é composta pelo nome do objeto, seguida de dois-pontos (:) e a indicação da classe do objeto. Esta indentificação deve ser sublinhada. Na verdade, o nome do objeto é opcional. Quando o nome

não for indicado, entende-se que se está fazendo referência a um objeto qualquer de determinada classe.

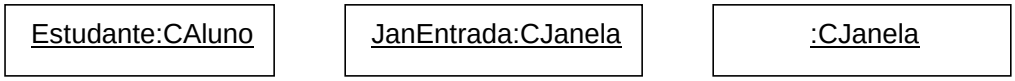

Figura II.7 – Exemplo da notação UML para objetos

3 Levantamento das Classes

Uma vez identificados os Atores e Casos de Uso do sistema, pode-se iniciar efetivamente o projeto do *software*. Do ponto de vista estrutural o projeto deve definir os componentes do sistema e as relações necessárias entre eles. Em um sistema orientado a objetos, os componentes estruturais do sistema são as classes.

A definição das classes necessárias para a realização de todas as funcionalidades do sistema envolve um processo de síntese, ou seja, de criação. Não existe um algoritmo ou técnica precisa para o estabelecimento de classes. Cabe ao projetista, baseado em sua experiência e criatividade determinar quais classes irão compor o sistema. A definição das classes exige um estudo detalhado dos requisitos do sistema e das possibilidades de separação dos dados e processos em classes.

Existem três técnicas básicas para enfrentar a dificuldade de definição de classes: (i) definir as classes por partes, (ii) proceder por refinamentos e (iii) utilizar estereótipos.

3.1 Definição das Classes por Caso de Uso

A definição de classes por partes significaria, em uma abordagem estruturada, estudar as classes dividindo-se o sistema em módulos ou subsistema. Na orientação a objetos, e em particular na UML, sugere-se que o levantamento das classes do sistema seja feita por caso de uso. Desta forma, tomaria-se isoladamente cada caso de uso para definição das classes necessárias a sua implementação.

Nesta abordagem, não se deve considerar os casos de uso como subsistemas. Embora, o levantamento das classes seja feito por caso de uso, elas não são exclusivas de certos casos de uso. Uma mesma classe pode ser empregada em mais de um caso de uso com o mesmo ou com papéis diferentes.

3.2 Definição das Classes por Refinamentos

Para sistemas maiores ou mais complexos, pode ser muito difícil fazer um levantamento completo das classes em uma primeira tentativa. Pode-se, então, empregar uma técnica por refinamentos na qual define-se, inicialmente, grandes classes (também chamadas classes de análise) e em refinamentos seguintes retorna-se para decompor estas classes em classes mais detalhadas e menores. Segue-se, nesta visão, uma abordagem *top-down* partindo-se de classes mais abstratas em direção a classes mais refinadas ou detalhadas.

3.3 Estereótipos para Classes

É possível utilizar o conceito de estereótipo para auxiliar no levantamento de classes. A UML define três estereótipos padrões para classes de análise :

- **entidade** : identifica classes cujo papel principal é armazenar dados que juntos possuem uma identidade. Este tipo de classe freqüentemente representa entidades do mundo real como aluno, professor, disciplina, etc.
- **controle** : identifica classes cujo papel é controlar a execução de processos. Estas classes contém, normalmente, o fluxo de execução de todo ou de parte de casos de uso e comandam outras classes na execução de procedimentos.
- **fronteira** : identifica classes cujo papel é realizar o interfaceamento com entidades externa (atores). Este tipo de classe contém o protocolo necessário para comunicação com atores como impressora, monitor, teclado, disco, porta serial, modem, etc.

Existem algumas regras gerais que, se utilizadas, podem auxiliar no levantamento das classes necessárias para cada caso de uso e permitir uma boa distribuição de papéis entre as classes.

Regras para definição de classes para um caso de uso:

- Definir uma classe do tipo fronteira para cada ator que participe do caso de uso

Considerando que atores são entidades externas, é necessário que o sistema comunique-se com eles utilizando protocolos específicos. Cada ator pode exigir um protocolo próprio para comunicação. Desta forma, um bom procedimento é criar uma classe específica para tratar a comunicação com cada ator. Assim, seriam definidas, por exemplo, uma classe para interface com o usuário, uma classe para interface com a impressora, uma classe para interface com a porta serial, etc.

- Definir pelo menos uma classe do tipo controle para cada caso de uso

Cada caso de uso implica um encadeamento de ações a serem realizadas pelo sistema. O algoritmo ou conhecimento do processo a ser realizado deverá estar definido em alguma parte do sistema. Este conhecimento pode estar distribuído em vários componentes do sistema, ou centralizado em um ou poucos componentes. A vantagem de centralizar o processo em um ou poucos componentes é a maior facilidade de entendimento do processo e sua modificação futura. Neste sentido pode-se criar uma classe de controle para cada caso de uso de forma que ela contenha a descrição e comando do processo associado ao caso de uso.

- Definir classes de controle auxiliares

Em certos casos de uso o processo nas classes de controle pode ser longo ou conter subprocessos. Nestes casos, pode-se extrair partes do processo da classe de controle principal e atribuí-los a classes de controle auxiliares. Assim, o fluxo principal do processo ficaria sob a responsabilidade da classe de controle principal e os subprocessos seriam controlados ou executados pelas classes de controle auxiliares.

- Definir uma classe do tipo entidade para cada grupo de dados

Grupos de dados que junto definem entidades abstratas ou do mundo real deveriam ser representados por classes. Sugere-se, portanto, que para cada caso de uso faça-se uma análise dos dados manipulados e identifique-se grupos que serão representados, cada um, por uma classe. Trata-se aqui de identificar dados que são manipulados no caso de uso e que, por conseqüência, devem estar em memória. Grupos de dados armazenados em arquivos ou bancos de dados não precisam possuir classes que os representem na totalidade. As classes do tipo entidade estão associadas unicamente a dados em memória.

4 Exemplo de Levantamento de Classes

Nesta seção será considerado o sistema de controle acadêmico, discutido no capítulo IV, para ilustrar o processo de levantamento de classes. Será feito o levantamento para dois dos casos de uso daquele sistema: Cadastrar Aluno e Emitir Diário de Classe.

4.1 Classes para o Caso de Uso Cadastrar Aluno

Conforme ilustrado na figura II.8, a realização do caso de uso Cadastrar Aluno envolve a comunicação com dois atores : Secretaria e SGBD. Desta forma, pode-se identificar duas classes do tipo fronteira para implementar o interfaceamento com estes dois atores. Estas duas classes poderiam se chamar : CInterfaceSecretaria e CIntefaceSGBD.

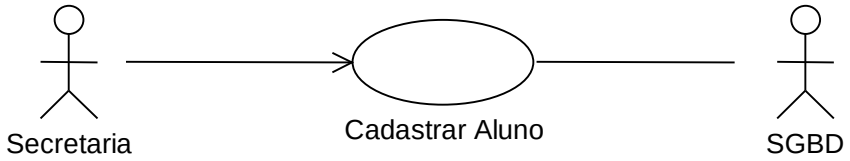

Figura II.8 – Caso de uso Cadastrar Aluno

Para descrever o processo do caso de uso e comandar as demais classes será definida uma classe de controle denominada CControleCadastrarAluno. Como para este caso de uso existem três subprocessos (inclusão, alteração e exclusão) poderiam ser criadas três classes de controle auxiliares, uma para cada subprocesso, que serão: CControleInclusaoCadAluno, CControleAlteracaoCadAluno e CControleExclusaoCadAluno.

Por fim, analisando-se os dados manipulados pelo caso de uso, percebe-se a existência de apenas um grupo de dados referentes às informações do aluno de está sendo incluído, alterado ou excluído. Desta forma, apenas uma classe do tipo entidade será definida e será denominada CAluno.

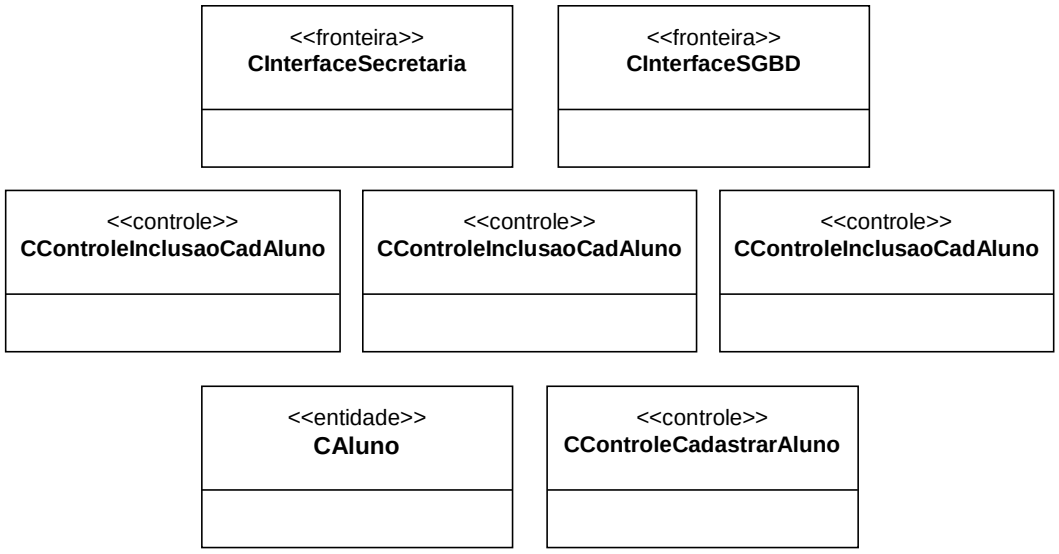

Figura II.9 – Classes para o caso de uso Cadastrar Aluno

4.2 Classes para o Caso de Uso Emitir Diário de Classe

Conforme ilustrado na figura II.10, a realização do caso de uso Emitir Diário de Classe envolve a comunicação com dois atores: Secretaria e SGBD. Desta forma, pode-se identificar duas classes do tipo fronteira para implementar o interfaceamento com estes dois atores. Estas duas classes poderiam se chamar: CInterfaceSecretaria e CIntefaceSGBD.

Nota-se que estas duas classes já foram definidas para o caso de uso Cadastrar Aluno. Neste ponto haveriam duas possibilidades: utilizar as mesmas classes de interface para os dois casos de uso ou definir novas classes para interfaceamento com os dois atores em cada caso de uso. A decisão por uma desta duas soluções dependeria, basicamente, da dimensão que alcançariam estas classes se elas agrupassem o interfaceamento em ambos os casos de uso. Como sugestão para este tipo de situação, sugere-se começar com uma solução unificada e na seqüência, a partir de um conhecimento melhor dos papéis das classes, estudar a possibilidade de decompor estas classes em classes mais especializadas.

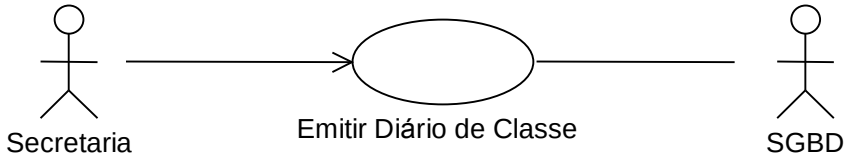

Figura II.10 – Caso de uso Cadastrar Aluno

Para descrever o processo do caso de uso e comandar as demais classes será definida uma classe de controle denominada CControleEmitirDiario. Não se observa, neste momento, subprocessos neste caso de uso. Por conseqüência não serão definidas classes de controle auxiliares.

Com relação às classes do tipo entidade, deve-se considerar quais dados compõem o documento a ser impresso. Um diário de classe normalmente contém o código e nome da disciplina, código e nome do professor, carga horária, e a lista de alunos com o nome e número de registro. Várias possibilidades se apresentam para representação em memória destes dados. Poderiam, inclusive, ser utilizadas aqui técnicas de normalização de Entidades e Relacionamentos.

Uma primeira alternativa para representação dos dados do caso de uso Emitir Diário de Classe seria a criação de uma única classe contendo todos os dados necessários para a composição do documento impresso. Esta classe poderia se chamar CDiarioClasse. Outra possibilidade seria extrair desta classe os dados sobre disciplina, professor e alunos, e colocá-los em três outras classes como ilustrado na figura II.11.

<<entidade>> **CDiarioClasse**	<<entidade>> **CDisciplina**	<<entidade>> **CProfessor**	<<entidade>> **CAluno**
codigoTurma: char[8] listaAlunos	cargaHoraria:int codigoDisciplina:char[10] nomeDisciplina:char[30]	codigo:int nome:char[30] categoria:char	registro:int nome:char[30] curso:char

Figura II.11 – Classes de entidade para o caso de uso Emitir Diário de Classe.

5 Conclusão

Como discutido neste capítulo, as classes e objetos são elementos de grande importância pois são os elementos constitutivos em sistemas computacionais orientados a objetos. A notação UML para classes e objetos segue, basicamente, a sintaxe já utilizada em outras linguagens orientadas a objetos.

Com relação ao projeto de *softwares* orientados a objetos, uma grande dificuldade está na definição de quais classes são necessárias para a realização dos casos de uso. Existem algumas dicas gerais que podem ser seguidas que não diminuem, entretanto, a responsabilidade do projetista no processo inventivo associado ao estabelecimento do conjunto de classes de um sistema.

Capítulo III : Estudo das Interações entre Objetos

Em um sistema computacional orientado a objetos os serviços (casos de uso) são fornecidos através da colaboração de grupos de objetos. Os objetos interagem através de comunicações de forma que juntos, cada um com suas responsabilidades, eles realizem os casos de uso. Neste capítulo discute-se a interação entre os objetos de um sistema computacional e apresentam-se duas notações para a descrição destas interações: os diagramas de seqüência e os diagramas de colaboração.

1 Diagrama de Seqüência

O diagrama de seqüência é uma ferramenta importante no projeto de sistemas orientados a objetos. Embora a elaboração dos diagramas de seqüência possa consumir um tempo considerável para sistemas maiores ou mais complexos, eles oferecem a seguir as bases para a definição de uma boa parte do projeto como: os relacionamentos necessários entre as classes, métodos e atributos das classes e comportamento dinâmico dos objetos.

1.1 Utilização do Diagrama de Seqüência

Um diagrama de seqüência é um diagrama de objetos, ou seja, ele contém com primitiva principal um conjunto de objetos de diferentes classes. O objetivo dos diagramas de seqüência é descrever as comunicações necessárias entre objetos para a realização dos processos em um sistema computacional. Os diagramas de seqüência têm este nome porque descrevem ao longo de uma linha de tempo a seqüência de comunicações entre objetos.

Como podem existir muitos processos em um sistema computacional, sugere-se proceder a construção dos diagramas de seqüência por caso de uso. Assim, tomaria-se separadamente cada caso de uso para a construção de seu ou seus diagramas de seqüência. De uma forma geral, para cada caso de uso constrói-se um diagrama de seqüência principal descrevendo a seqüência normal de comunicação entre objetos e diagramas complementares descrevendo seqüências alternativas e tratamento de situações de erro.

Utiliza-se também o termo **cenário** associado com os diagramas de seqüência. Um cenário é uma forma de ocorrência de um caso de uso. Como o processo de um caso de uso pode ser realizado de diferentes formas, para descrever a realização de um caso de uso pode ser necessário estudar vários cenários. Cada cenário pode ser descrito por um diagrama de seqüência. No exemplo do caso de uso Cadastrar Aluno do sistema de controle acadêmico, pode-se considerar os cenários de inclusão, alteração e exclusão de aluno.

1.2 Notação UML para Diagramas de Seqüência

Notação Básica

A notação UML para descrição de diagramas de seqüência envolve a indicação do conjunto de objetos envolvidos em um cenário e a especificação das mensagens trocadas entre estes objetos ao longo de uma linha de tempo. Os objetos são colocados em linha na parte superior do diagrama. Linhas verticais tracejadas são traçadas da base dos objetos até a parte inferior do diagrama representando a linha de tempo. O ponto superior destas linhas indica um instante

inicial e, à medida que se avança para baixo evolui-se o tempo. Retângulos colocados sobre as linhas de tempo dos objetos indicam os períodos de ativação do objeto. Durante um período de ativação, o objeto respectivo está em execução realizando algum processamento. Nos períodos em que o objeto não está ativo, ele está alocado (ele existe) mas não está executando nenhuma instrução. A figura III.1 ilustra a estrutura básica de um diagrama de seqüência para o cenário Inclusão de Aluno do caso de uso Cadastrar Aluno.

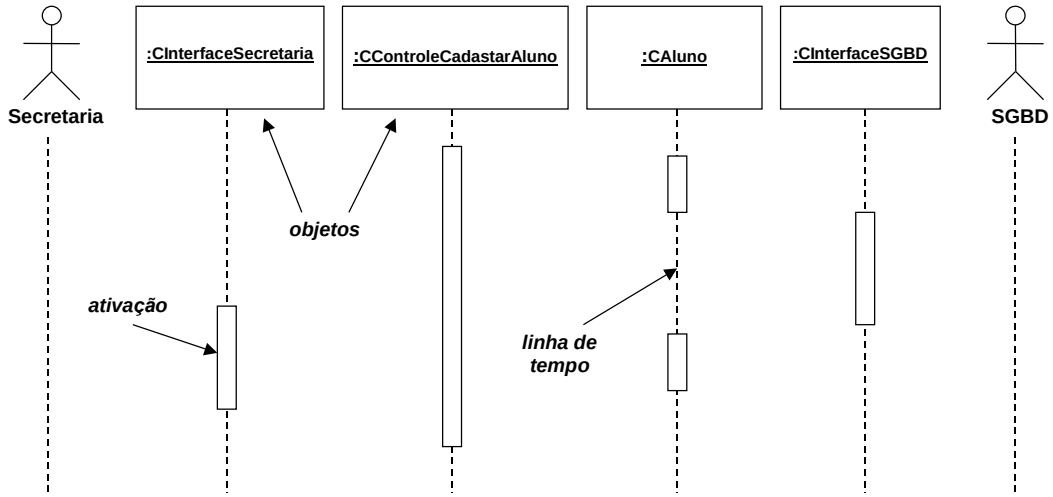

Figura III.1 – Estrutura básica de um diagrama de seqüência.

Outra primitiva importante dos diagramas de seqüência é a troca de mensagem. Esta primitiva é utilizada para indicar os momentos de interação ou comunicação entre os objetos. Utiliza-se como notação para trocas de mensagens segmentos de retas direcionados da linha de tempo de um objeto origem para a linha de tempo de um objeto destino. Uma seta é colocada na extremidade do objeto destino. As linhas representando troca de mensagens são desenhadas na horizontal pois presume-se que a troca de uma mensagem consome um tempo desprezível.

O objeto destino de uma mensagem deve receber a mensagem e processá-la. Desta forma, no recebimento de uma mensagem o objeto destino é ativado para tratamento da mensagem. A figura III.2 ilustra a troca de mensagens entre objetos e entre atores e objetos.

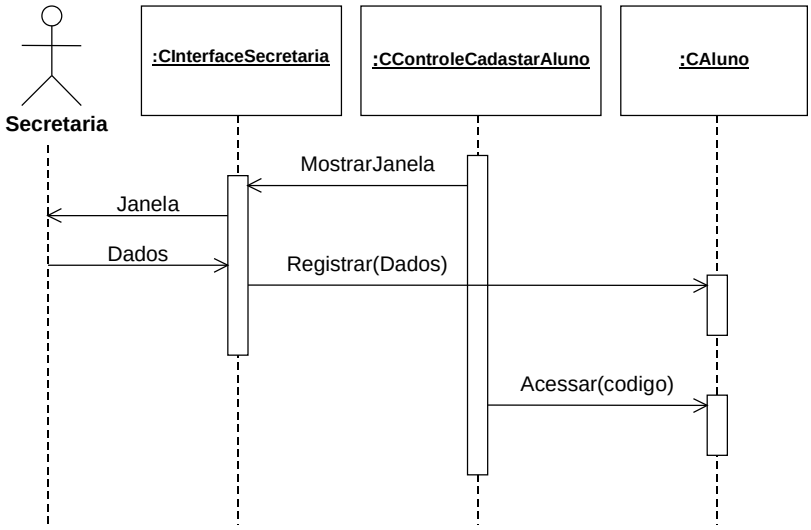

Figura III.2 – Exemplo de troca de mensagens em um diagrama de seqüência.

Significado das Mensagens

As mensagens trocadas entre um objeto e outro ou entre um objeto e um ator podem ter três significados:

- **Chamada de Função ou Procedimento**

  Uma das interações mais freqüentes entre objetos é a chamada de função. Uma chamada de função significa que um objeto está solicitando a execução de uma função (um método) de um outro objeto. Este conceito segue estritamente o processo de chamada de função ou de procedimento utilizado nas linguagens de programação. Obviamente, para que um objeto possa chamar um método de outro objeto é necessário que o método seja declarado como público na classe respectiva. Como será visto a seguir, a sintaxe, no caso de mensagens que representem chamadas de função, é semelhante àquela das linguagens de programação.

- **Envio de Mensagem**

  Objetos também podem comunicar-se com outros objetos através do envio explícito de uma mensagem. O envio de uma mensagem, ao contrário da chamada de função, não é uma interação direta entre dois objetos. Na verdade, quando um objeto envia uma mensagem a outro objeto, a mensagem é roteada ou encaminhada por algum mecanismo de entrega de mensagens. Normalmente, este serviço é prestado pelo sistema operacional através de mecanismos como *Message Queues* (filas de mensagens) ou serviços de notificação.

- **Ocorrência de Evento**

  Existem também outras formas de interação entre objetos através de eventos. Esta é também a forma padrão de interação entre objetos e atores. Basicamente, um evento é algum acontecimento externo ao *software* mas que é a ele notificado pois lhe diz respeito. A notificação, ou seja, a indicação de que um determinado evento ocorreu é, na maioria dos casos, feita pelo sistema operacional. Eventos podem também ser gerados pelo *software* para o sistema operacional. Exemplos são as saídas para dispositivos (monitor, porta serial, disco) feita através de serviços do sistema operacional.

  Alguns exemplos de eventos são:

Evento	**Origem**	**Destino**
Clique do mouse	mouse	algum objeto
Movimentação do mouse	mouse	algum objeto
Dados no *buffer* do teclado	teclado	algum objeto
Dados no *buffer* da serial	porta serial	algum objeto
Projeção de dados no monitor	algum objeto	monitor
Bip do autofalante	algum objeto	autofalante
Colocação de dados no *buffer* da serial	algum objeto	porta serial
Interrupção	dispositivo de *hardware*	algum objeto
Eventos do sistema operacional (*timer*)	sistema operacional	algum objeto

Sintaxe das Mensagens

A sintaxe geral para mensagens em diagramas de seqüência é:

Predecessor/ *[Condição] Seqüência: Retorno:= Nome_Mensagem(Argumentos)

- **Predecessor/**

  As mensagens em um diagrama de seqüência podem ser numeradas para indicar a ordem de ocorrência (ver adiante). Toda mensagem para ser enviada depende de que a mensagem imediatamente anterior tenha sido enviada indicando, assim, uma dependência temporal que define a ordem de envio das mensagens. A mensagem imediatamente anterior é o predecessor natural de cada mensagem e não precisa ser indicada pois entende-se isto implicitamente.

  Algumas mensagens dependem de mais de um predecessor. Na verdade isto só ocorre em processos concorrentes. Quando houver mais de um objeto ativo (mais de uma thread) em execução algumas mensagens podem depender de seu predecessor imediato mas também de algum predecessor associado à outro objeto ativo. Tem-se, desta forma, um conjunto de mensagens que não seguem uma seqüência. Neste caso pode-se indicar o predecessor não imediato indicando-se sua numeração de seqüência seguida de uma barra inclinada (/). Se houverem vários predecessores não imediatos deve-se separá-los com vírgulas.

- **Condição**

  Pode-se, opcionalmente, incluir uma condição entre colchetes em uma mensagem. Desta forma, para que a mensagem seja enviada é necessário que a condição seja satisfeita. Uma condição é descrita por uma expressão de igualdade envolvendo atributos do objeto ou outras variáveis locais e constantes.

  Exemplos: `[x<10], [res=OK], [valor=5], [flag=true].`

  A inclusão de um asterisco (*) antes de uma condição permite especificar iterações ou repetições. Neste caso, a condição representa uma expressão lógica de controle da repetição. A condição será avaliada e, se for verdadeira, a mensagem será enviada uma primeira vez. Ao término do envio, a condição será reavaliada. Enquanto a condição for verdadeira a mensagem continuará a ser enviada e a expressão de condição reavaliada. Quando a condição se tornar falsa, a mensagem não é enviada e avança-se para a próxima mensagem no diagrama. Note que a mensagem pode nunca ser enviada se já na primeira avaliação da condição o resultado for falso. É possível, também, utilizar a própria variável de retorno (ver adiante) como membro da expressão de condição.

  Exemplo: `*[x<10] calcular(x)`

  Neste exemplo, enquanto o valor da variável `x` for menor do que dez, a mensagem `calcular(x)` será enviada.

- **Seqüência**

  As mensagens em um diagrama de seqüência ocorrem em uma determinada seqüência indicada pela ordem de aparecimento no diagrama. Pode-se incluir junto às mensagens uma numeração de seqüência para indicar explicitamente a ordenação de ocorrência das mensagens. As mensagens são enviadas na ordem indicada por seus números de seqüência. Esta informação é, na maioria das vezes desnecessário pois o grafismo dos diagramas de seqüência já indica a cronologia de ocorrência das mensagens. Em diagramas de colaboração, entretanto, a indicação do número de seqüência pode ser

útil. O uso da numeração de seqüência é também útil quando existem concorrências no diagrama de seqüência pela existência de mais de uma *thread*.

A notação para seqüência é a colocação de um valor inteiro seguido de dois-pontos (:) antes do nome da mensagem.

Pode-se utilizar níveis na numeração das mensagens. Isto pode ser útil para a identificação de blocos de mensagens. Por exemplo, todas as mensagens com número de seqüência prefixado com "1." representam a fase de inicialização e todas as mensagens com número de seqüência prefixado com "2." representam a fase de processamento.

Quando houver mais de um objeto ativo, indicando um fluxo de mensagens não seqüencial, pode-se utilizar letras na numeração das mensagens. Cada letra prefixando um conjunto de números de seqüência indicaria mensagens de um certa *thread*.

- **Retorno**

  Determinadas mensagens podem produzir um valor de retorno ao término de seu tratamento. Este valor deve ser retornado ao objeto que enviou a mensagem. O caso mais comum de valor de retorno ocorre na chamada de funções. Muitas funções permitem produzir um valor que é retornado ao objeto que fez sua chamada. O objeto chamador, neste caso, deve indicar uma variável para receber o valor de retorno.

  A notação para valor de retorno é a indicação de um nome de variável seguida de dois-pontos antes do nome da mensagem.

  Exemplo: `res:=registrar(codigo)`

  Neste exemplo a variável `res` receberá o valor de retorno da função registrar. Deve-se notar que a variável `res` é uma variável do objeto que está enviado a mensagem (fazendo a chamada) e que ela deve ter sido declarada como um atributo do objeto ou como uma variável local.

- **Nome_Mensagem**

  O nome da mensagem é o identificador da mensagem ou função que está sendo chamada. Quando se tratar de chamada de função é necessário que a função seja declarada como uma das funções membro do objeto de destino da mensagem.

- **Argumentos**

  Os argumentos de uma mensagem são valores (constantes ou variáveis) enviados junto com a mensagem. No caso de chamadas de função os argumentos devem coincidir com os parâmetros definidos para a função na classe do objeto destino. Não é necessário indicar o tipo de dado do argumento pois os tipos são definidos na classe do objeto de destino.

Tipos de Mensagens

Nos exemplos das figuras III.1 e III.2 utilizou-se como notação gráfica para as trocas de mensagens segmentos de reta com um seta aberta em uma das extremidades. Esta é a notação genérica para mensagens que pode ser utilizada nas primeiras versões dos diagramas de seqüência. Em diagramas mais detalhados, entretanto, será utilizada outra notação de forma a indicar também o tipo das mensagens. Existem dois tipos de mensagens chamadas mensagens síncronas e assíncronas.

- **Mensagens Síncronas**

  Mensagens síncronas são mensagens que implicam um sincronismo rígido entre os estados do objeto que envia a mensagem e os do objeto de destino da mensagem. Um sincronismo entre objetos podem ser entendido, de uma forma geral, como uma dependência na evolução de estado de um objeto sobre o estado de um segundo objeto. De uma forma mais direta, pode-se dizer que uma mensagem síncrona implica que o objeto que enviou a mensagem aguarde a conclusão do processamento da mensagem (entendida como um sinal de sincronismo) feito pelo objeto destino, para então prosseguir seu fluxo de execução.

  O exemplo mais comum de mensagem síncrona é a chamada de função. Em uma chamada de função o objeto que faz a chamada é empilhado e fica neste estado até a conclusão do processamento da função chamada. Trata-se, portanto, de um sincronismo rígido entre o objeto chamador e o objeto chamado. Alguns sistemas operacionais oferecem também mecanismos de troca de mensagens síncronas de forma que o objeto que envia a mensagem fique em estado de espera até a conclusão do tratamento da mensagem.

  A notação UML para uma mensagem síncrona é a de um segmento de reta com uma seta cheia em uma das extremidades (figura III.3).

- **Mensagens Assíncronas**

  Mensagens assíncronas são mensagens enviadas de um objeto a outro sem que haja uma dependência de estado entre os dois objetos. O objeto de origem envia a mensagem e prossegue seu processamento independentemente do tratamento da mensagem feita no objeto destino. Os mecanismos de envio de mensagens suportados pelos sistemas operacionais são o exemplo mais comum deste tipo de mensagem. Além disso, de uma forma geral, todas as comunicações entre atores e objetos representam trocas de mensagem assíncronas. Considere, por exemplo, uma operação para apresentação de uma mensagem no monitor. Um objeto executa uma instrução printf (ou print ou write) e o sistema despacha a mensagem para o ator (o monitor). O objeto prossegue seu processamento independentemente do desfecho na operação. Observe que os *softwares* não bloqueiam quando o monitor não apresenta alguma informação.

  A notação UML para uma mensagem assíncrona é a de um segmento de reta com uma meia seta em uma das extremidades (figura III.3).

Além desta diferenciação entre mensagens síncronas e assíncronas, pode-se também indicar quando uma mensagem consome tempo. A notação UML para mensagens deste tipo é ilustrada na figura III.3.

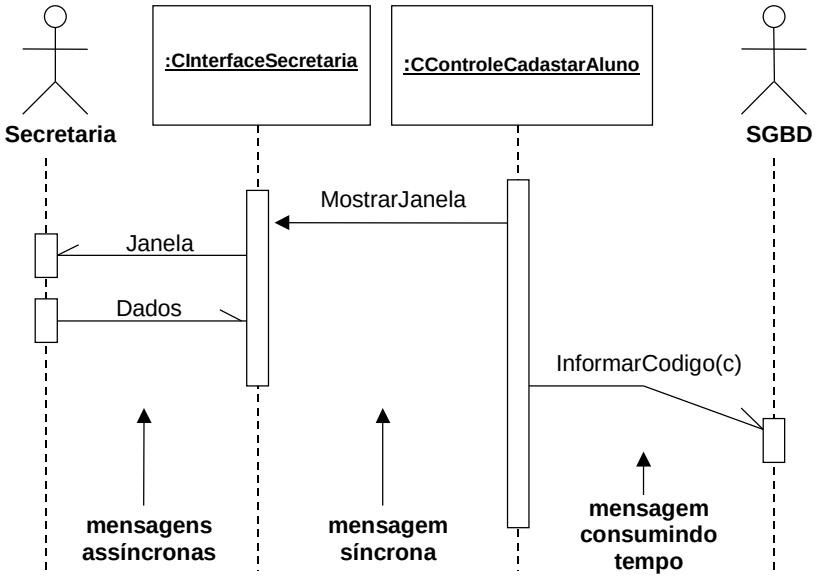

Figura III.3 – Notação para mensagens síncronas, assíncronas e que consomem tempo.

Notação Complementar

Além da notação vista nas seções anteriores existem ainda alguns elementos complementares da notação, conforme descrito a seguir.

- **Criação e Destruição de Objetos**

  Objetos que participam de um caso de uso e que aparecem nos diagramas de seqüência podem não existir desde o início da execução do caso de uso e podem não permanecer existindo até a conclusão do caso de uso. Dito de outra forma, objetos podem ser criados (alocados) e destruídos (desalocados) ao longo da execução de um caso de uso. Existem duas formas de se representar estas situações.

  A primeira forma de especificar criação e destruição de objetos é incluindo-se o objeto no alinhamento de objetos na parte superior do diagrama de seqüência e indicando os eventos de criação e destruição através de mensagens síncronas. A figura III.4 ilustra esta forma de especificação.

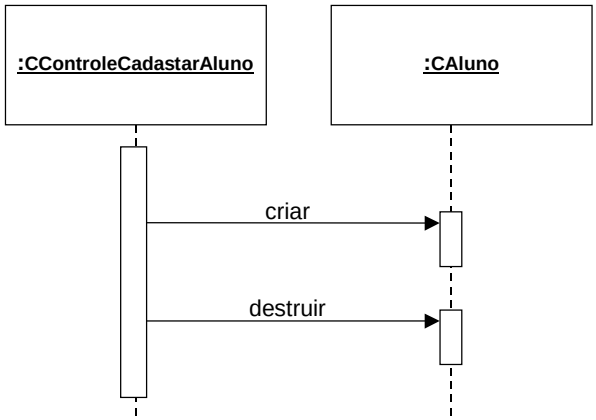

Figura III.4 – Notação básica para criação e destruição de objetos.

Outra forma de especificar a criação e destruição de um objeto é fazê-lo aparacer no diagrama somente após sua criação e eliminar sua linha de tempo após sua destruição, conforme ilustrado na figura III.5. Utiliza-se neste caso um símbolo de X para indicar o ponto de destruição do objeto.

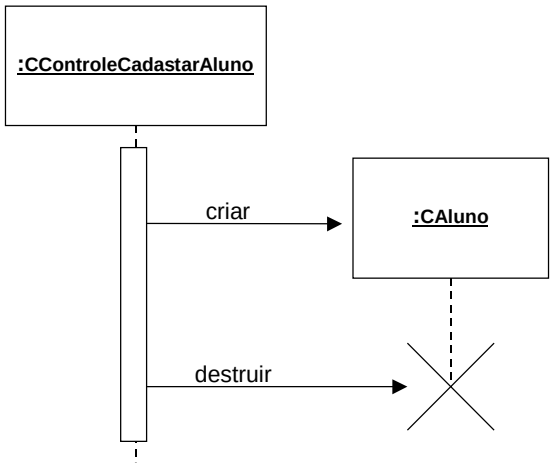

Figura III.5 – Notação altenativa para criação e destruição de objetos.

Nas duas notações é possível incluir parâmetros na mensagem criar. Isto correponderia aos argumentos passados para a função construtora do objeto.

- **Autodelegação**

  Um objeto pode enviar mensagens para outros objetos e pode, também enviar mensagens para ele próprio. Uma mensagem enviada de um objeto para ele próprio chama-se uma autodelegação. Mensagens de autodelegação podem ser síncronas ou assíncronas. O caso mais comum de mensagens assíncronas é o envio de uma mensagem de um objeto para ele mesmo através de mecanismos de envio de mensagens do sistema operacional. O caso mais comum de mensagens de autodelegação síncronas é a chamada de função de um objeto pelo próprio objeto.

  A notação UML para autodelegação é ilustrada na figura III.6.

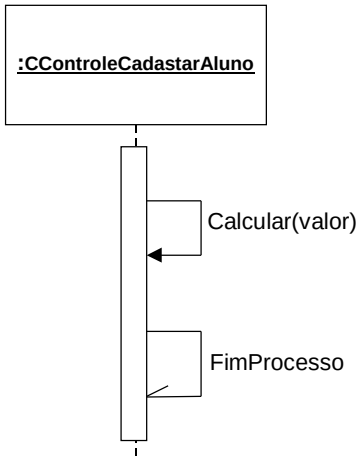

Figura III.6 – Notação UML para autodelegação.

- **Objeto Ativo**

  Um objeto ativo é um objeto que está em execução em uma *thread* própria. Isto significa que o objeto têm um fluxo de execução distinto do fluxo de execução dos demais objetos e está em execução concorrente. Este tipo de ocorrência é cada vez mais comum em sistema operacionais multitarefa como o Windows da Microsoft.

  A notação UML para objetos ativos é a indicação com borda larga do objeto ativo. Esta notação pode ser utilizada em qualquer diagrama de objetos inclusive nos diagramas de seqüência.

  Deve-se observar que objetos ativos introduzem um comportamento diferente e mais complexo nos diagramas de seqüência. Os diagramas passam a incluir concorrência. Tipicamente, a comunicação entre um objeto ativo e outros objetos se faz através de mensagens assíncronas. A figura III.7 apresenta um pequeno exemplo de uso de um objeto ativo.

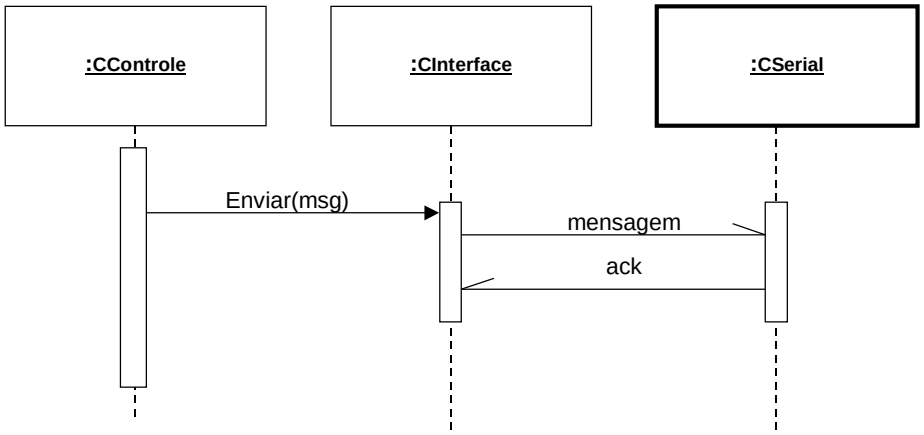

Figura III.7 – Exemplo de objeto ativo

- **Retorno de Mensagem Síncrona**

  Uma notação existente na UML, embora em desuso, é a de retorno de mensagem síncrona. Esta notação é utilizada para indicar o término de uma chamada síncrona e representaria o sinal de retorno da chamada. Não se trata, entretanto, de uma mensagem explícita de algum objeto mas de um sinal de controle. Na prática este retorno significa o desempilhamento da função chamada ou o desbloqueio da instrução de envio no caso de mensagens síncronas.

  A notação UML para retorno de mensagem síncrona é uma linha tracejada no sentido contrário do da chamada síncrona com a indicação “retorno”. A figura III.8 ilustra o uso de retorno de mensagem síncrona.

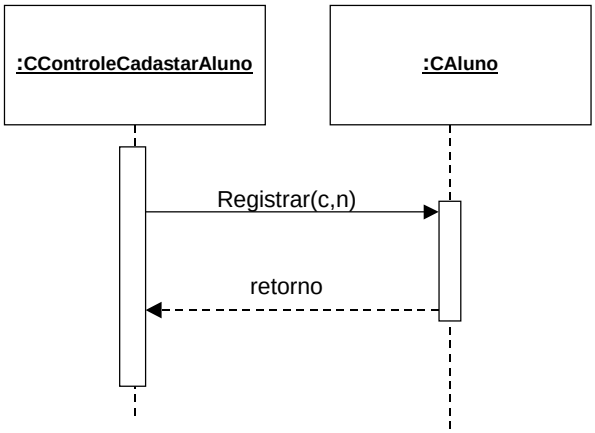

Figura III.8 – Notação UML para retorno de mensagem síncrona.

Notações Não Padronizadas

- **Sobreativação**

  Uma sobreativação é a ativação de um objeto que já estava ativado. Em termos de programa, uma sobreativação representa um empilhamento de função de um mesmo objeto. Um caso simples de sobreativação é a autodelegação, em particular a chamada de função de um objeto pelo próprio objeto. A função que fez a chamada é empilhada e a função chamada é ativada. Outro exemplo de sobreativação ocorre quando um objeto chama uma função de outro objeto e é empilhado aguardando a conclusão da chamada. O segundo objeto, então, faz uma chamada de função ao primeiro objeto ativando uma função de um objeto que já estava ativado (embora empilhado).

  A UML não apresenta uma notação para sobreativação. Uma notação especial é proposta no sistema CASE Together que utiliza um retângulo adicional para indica a sobreativação. A figura III.9 ilustra esta notação.

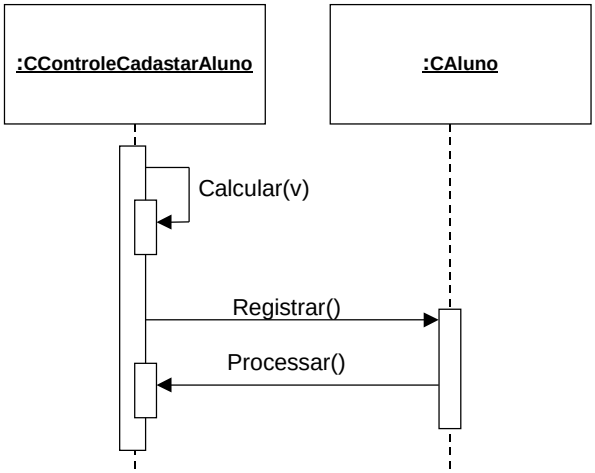

Figura III.9 – Notação para sobreativação.

- **Teste condicional**

  Segundo a norma UML, testes condicional são sempre associados às mensagens e são especificados como expressões lógicas entre colchetes antes do nome da mensagem. No sistema CASE Together, utiliza-se a instrução `if (expressão) then else` para indicar testes condicionais e blocos de comandos associados. Esta solução permite a construção efetiva de algoritmos através de diagramas de seqüência. A figura III.10 ilustra o uso desta notação. Deve-se observar que este artifício permite contornar um deficiência da UML que é a especificação de várias mensagens sob uma mesma condição.

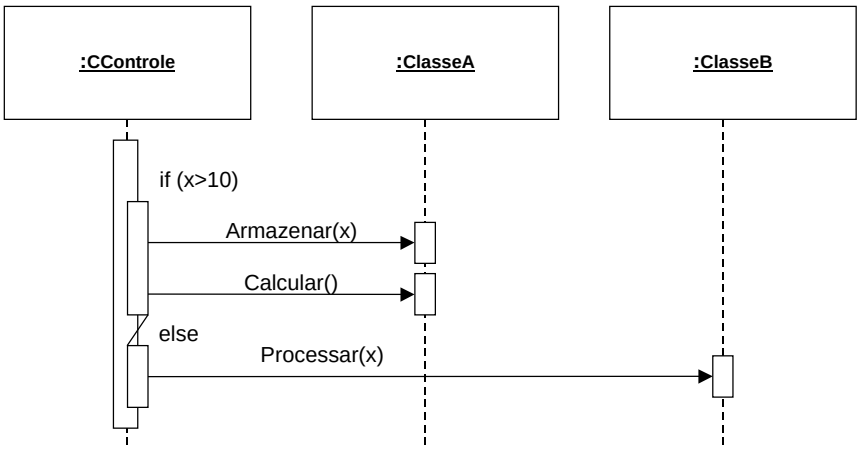

Figura III.10 – Notação para teste condicional.

- **Laços**

  Laços são estruturas de repetição controladas utilizadas em diversos tipos de processamentos. A UML inclui como especificador de repetição a notação `*[cond]`. Esta notação, entretanto, só permite indicar repetições sobre uma única mensagem. No sistema CASE Together, adotou-se uma notação adicional permitindo especificar repetições sobre blocos utilizando a noção de sobreativação. Utiliza-se para controle da repetição uma instrução while ou repeat com uma expressão condicional tal como ocorre nas linguagens de programação. A figura III.11 ilustra uma repetição utilizando esta notação.

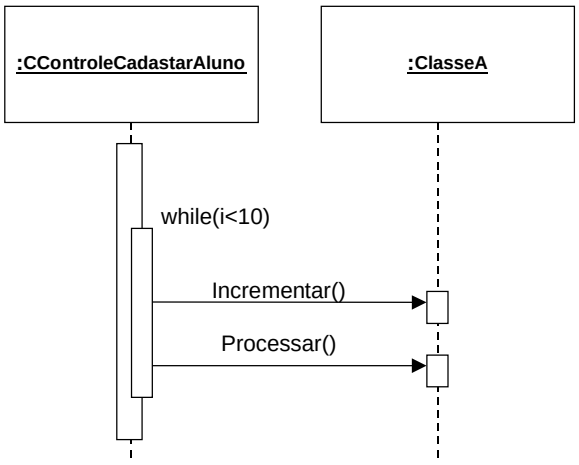

Figura III.11 – Notação para laços.

- **Múltiplos aninhamentos**

  Múltiplos aninhamentos ocorrem quando existem múltiplas sobreativações de um objeto. Em termos de programa isto significa múltiplo empilhamento, ou seja, diversas funções de um objeto são chamadas sem que as precedentes tenham concluído sua execução. Estes aninhamentos podem ocorrer em chamadas de função, dentro de blocos condicionais e dentro de repetições. A figura III.12 apresenta um exemplo de múltiplo aninhamento. Deve-se observar que aninhamentos com um número indefinido de níveis (como nas recursividades) não podem ser representados diretamente com esta notação.

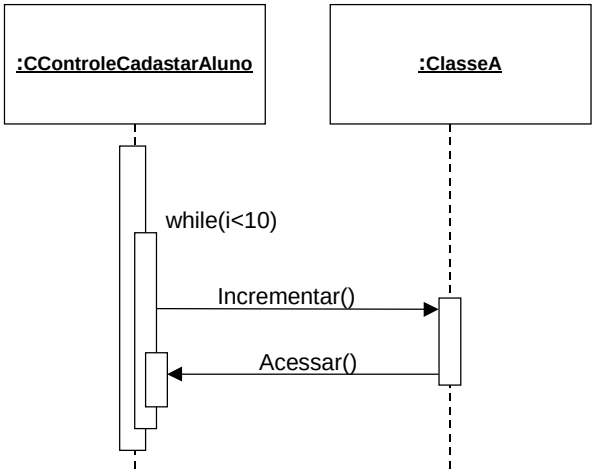

Figura III.12 – Múltiplo aninhamento.

2 Diagrama de Colaboração

Os diagramas de colaboração derivam dos diagramas de seqüência. Eles podem, inclusive, ser gerados automaticamente a partir dos diagramas de seqüência (como ocorre no *software* Rational Rose). Os diagramas de colaboração descrevem grupos de objetos que colaboram através de comunicações.

2.1 Utilização dos Diagramas de Colaboração

Os diagramas de colaboração são diagramas de objetos. As primitivas neste tipo de diagrama são os objetos, atores e comunicações. As comunicações são conexões entre objetos e entre objetos e atores indicando que este elementos se comunicam e relacionando as mensagens trocadas entre eles.

Pode-se considerar que um diagrama de colaboração é um diagrama de seqüência sem as linhas de tempo. Eles apresentam, portanto, todas as mensagens entre pares de objetos de forma agrupadas sem especificar a cronologia de ocorrência.

O objetivo na construção de diagramas de colaboração é exatamente agrupar as mensagens entre pares de objetos de forma a fazer-se um levantamento das necessidades de comunicação. Na seqüência esta informação será utilizada para a definição das relações estruturais entre as classes de forma a se estabelecer canais físicos de comunicação para os objetos. Através destes canais (relacionamentos) todas as mensagens serão trocas entre os objetos das classes respectivas.

2.2 Notação para Diagramas de Colaboração

A notação UML para diagramas de colaboração inclui a notação para objetos e atores, como já foi visto anteriormente, e a notação para comunicações. Um comunicação é especificada através de um segmento de reta ligando um objeto à outro ou um objeto a um ator. Sobre este segmento de reta relaciona-se, separadamente, as mensagens enviadas em cada sentido. Observa-se que as comunicações podem ser uni ou bidirecionais conforme o sentido do conjunto de mensagens trocas entre os objetos.

A figura III.13 apresenta um exemplo de diagrama de colaboração correspondendo ao exemplo da figura III.2. Observe que costuma-se numerar as mensagens de maneira a indicar a ordem de ocorrência. Nos diagramas de seqüência costuma-se não fazer esta numeração pois o próprio diagrama já indicar o ordenamento pelo seu grafismo.

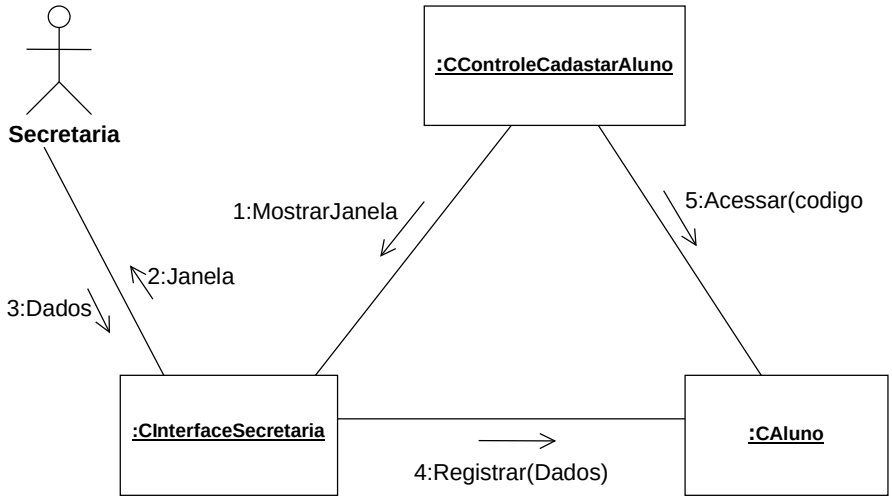

Figura III.13 – Exemplo de diagrama de colaboração.

3 Exemplos de Diagramas de Seqüência e Colaboração

A figura III.14 apresenta um exemplo de diagrama de seqüência para o caso de uso Cadastrar Disciplina do sistema de controle acadêmico discutido nos capítulos anteriores. O diagrama mostra o cenário de Inclusão de uma Disciplina no Cadastro.

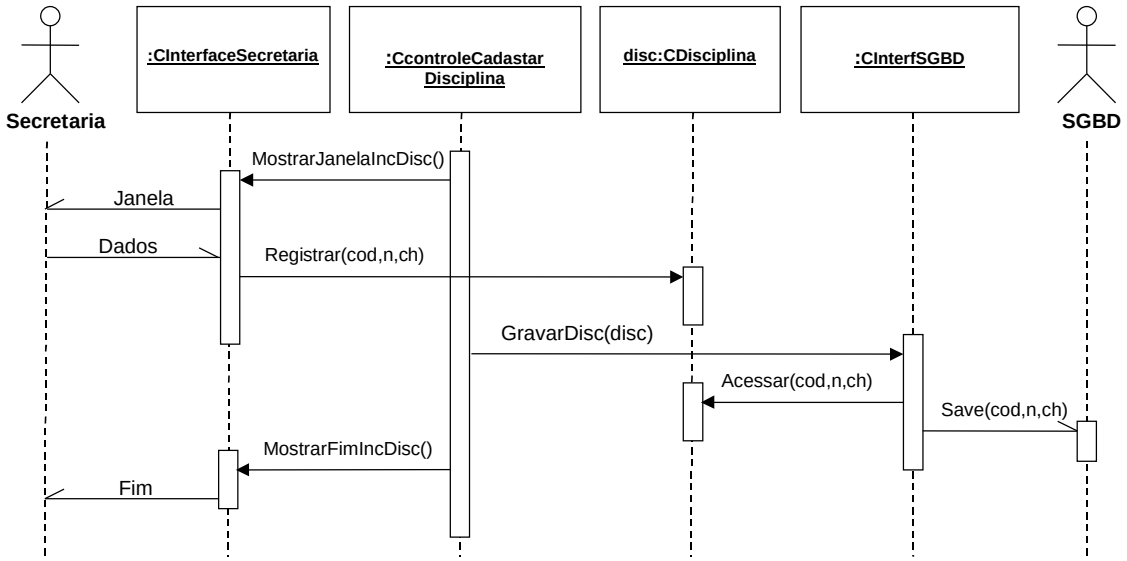

Figura III.14 – Diagrama de seqüência do caso de uso Cadastrar Disciplina.

A figura III.15 apresenta o diagrama de colaboração obtido a partir do diagrama de seqüência da figura III.14.

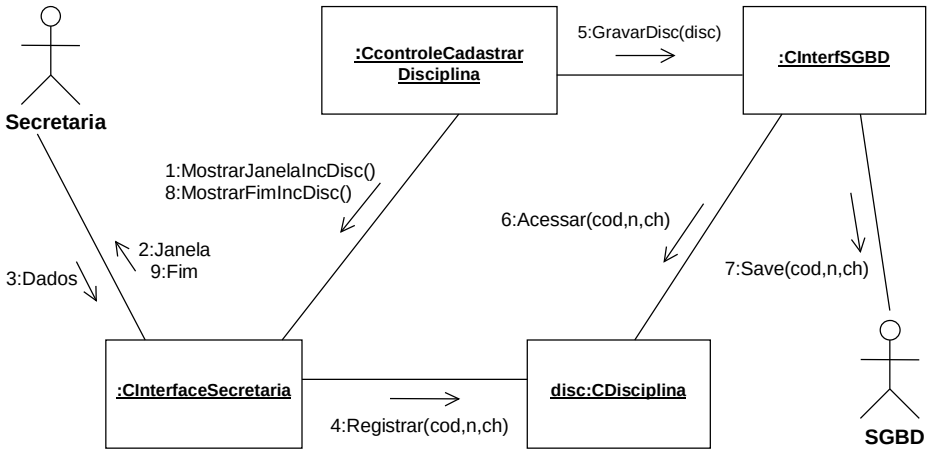

Figura III.15 – Diagrama de colaboração do diagrama de seqüência da figura III.14.

A figura III.16 apresenta o diagrama de seqüência para o caso de uso Emitir Diário de Classe do sistema de controle acadêmico discutido nos capítulos anteriores.

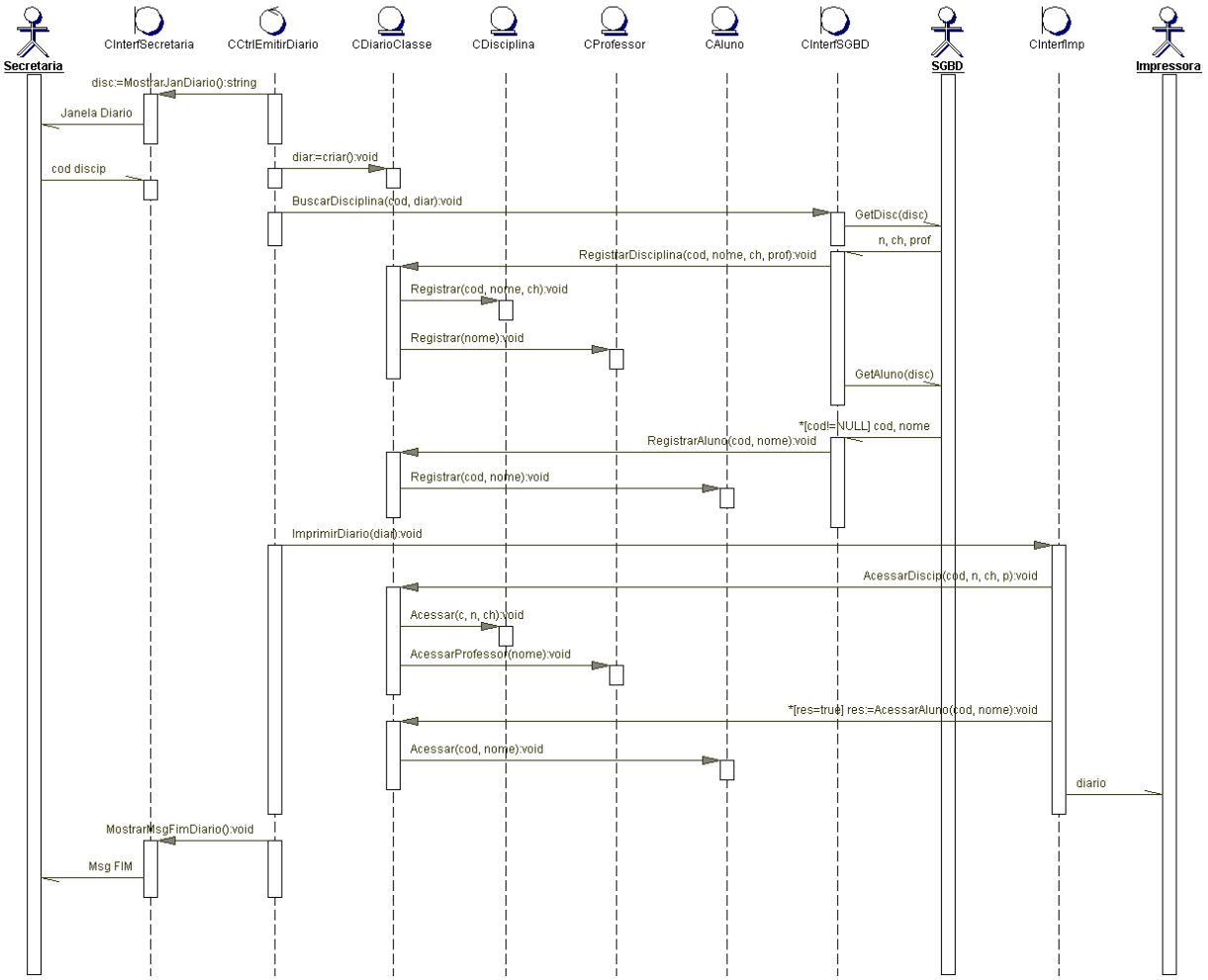

Figura III.16 – Diagrama de seqüência do caso de uso Emitir Diário de Classe.

O diagrama na figura III.16 foi elaborado utilizando a ferramenta CASE Together 5.02. As descontinuidades na ativação de certos objetos são devidas aos automatismos da própria ferramenta. Note, por exemplo, que o objeto da classe CCtrlDiario deveria estar ativo ao longo de todo o diagrama pois ele comanda a execução do caso de uso.

A figura III.17 apresenta o diagrama de colaboração obtido a partir do diagrama de seqüência da figura III.16.

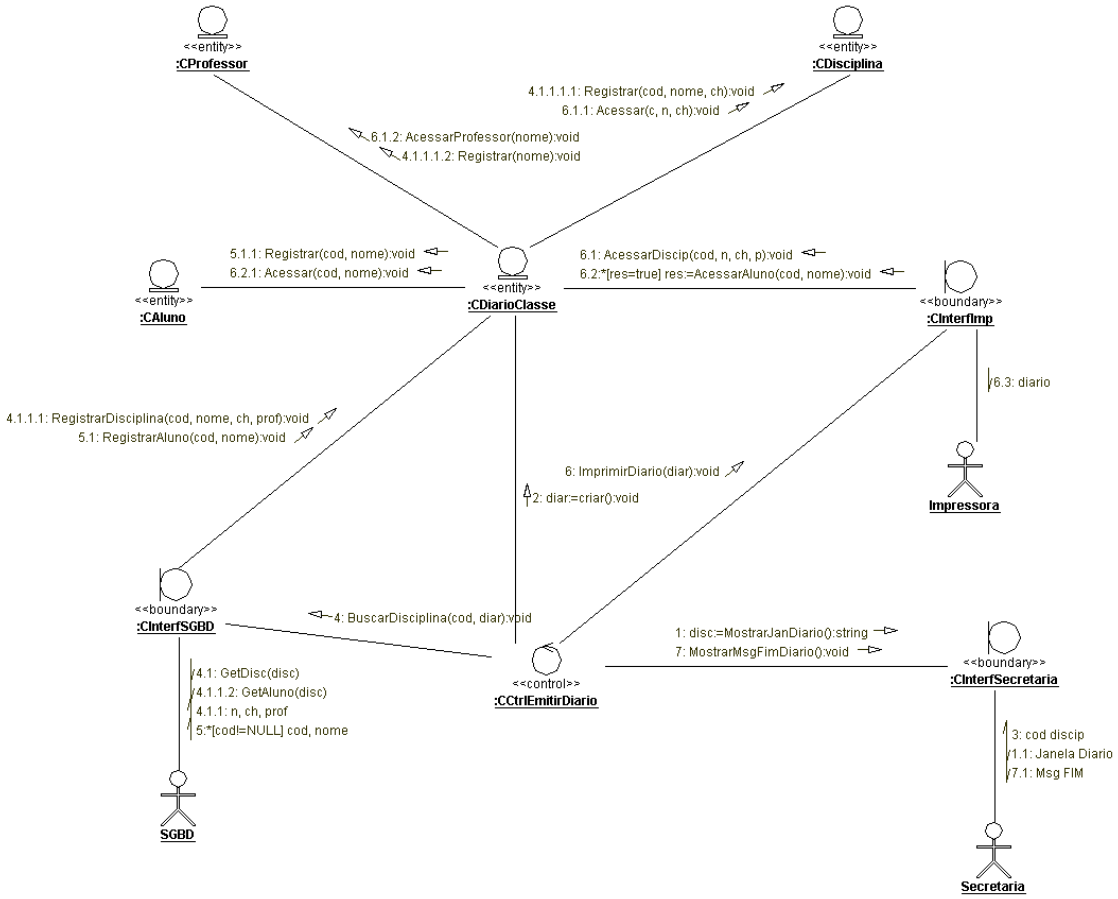

Figura III.17 – Diagrama de colaboração do diagrama de seqüência da figura III.16.

Capítulo IV : Relacionamentos entre Classes

Neste capítulo discute-se os relacionamentos estruturais entre classes. Estes relacionamentos existem em todo sistema orientado a objetos e são eles que permitem a interação entre os objetos para a realização dos casos de uso. Estes relacionamentos precisam ser criteriosamente definidos durante o projeto do *software*. Como será discutido, os relacionamentos necessários podem ser, e em grande parte são, obtidos a partir de análise dos diagramas de colaboração e dos papéis das classes.

A definição das classes e dos relacionamentos irão compor os diagramas de classes do sistema. Este é um dos principais diagramas da UML e dos projetos de *software,* pois eles descrevem o esqueleto do sistema sendo projetado. A partir do diagrama de classes já é possível, por exemplo, a geração (parcial) de código fonte.

Existem, basicamente, três tipos de relacionamentos entre classes : associação, agregação e generalização. As seções seguintes discutem estes tipos de relacionamentos e apresentam a notação UML utilizada para diagramas de classes.

1 Associação entre Classes

A associação é o tipo de relacionamento mais comum e mais importante em um sistema orientado a objetos. Praticamente todo sistema orientado a objetos terá uma ou mais associações entre suas classes. Uma associação é um relacionamento ou ligação entre duas classes permitindo que objetos de uma classe possam comunicar-se com objetos de outra classe.

O objetivo essencial da associação é possibilitar a comunicação entre objetos de duas classes. A associação aplica-se a classes independentes que precisam comunicar-se de forma uni ou bidirecional de maneira a colaborarem para a execução de algum caso de uso. Uma classe pode associar-se a uma ou mais classes para fins de comunicação. Não existe um limite máximo de associações que uma classe pode possuir com outras classes.

1.1 Notação UML para Associações

- **Nome e sentido**

A notação UML para uma associação é um segmento de reta ligando as duas classes. Se a associação for unidirecional, inclui-se uma seta na extremidade de destino da associação. Opcionalmente, mas fortemente sugerido, pode-se incluir um nome na associação. Este nome é colocado sobre o segmento de reta e tem como propósito indicar a natureza da associação. Embora associações sirvam sempre para comunicações, através do seu nome pode-se indicar com mais clareza a natureza ou finalidade da comunicação. O nome, entretanto, serve apenas como elemento de documentação não possuindo uma semântica mais importante. A figura IV.1 ilustra a notação básica para relacionamentos de associação.

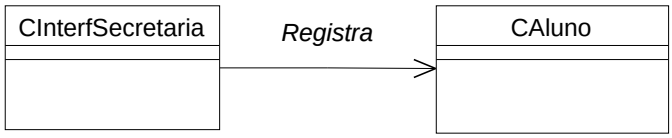

Figura IV.1 – Notação básica para relacionamentos de associação.

No exemplo da figura IV.1, a associação Registra é unidirecional indicando que objetos da classe CInterfSecretaria podem se comunicar com objetos da classe CAluno. Observe que neste caso, a associação não pode ser navegada no sentido contrário, ou seja, objetos da classe CAluno não podem se comunicar com objetos da classe CInterfSecretaria.

- **Papéis das classes**

Pode-se, ainda, incluir uma indicação dos papéis das classes nas associações. Uma especificação de papel indica qual a participação ou atribuição de cada classe na associação. A indicação de papel é feita colocando-se a especificação do papel na forma de um texto nos extremos da associação de forma a indicar, para cada classe, qual o seu papel. Não é comum indicar o nome da associação e os papéis das classes para uma mesma associação. Normalmente, opta-se por uma ou outra especificação. A figura IV.2 ilustra um relacionamento de associação com papéis.

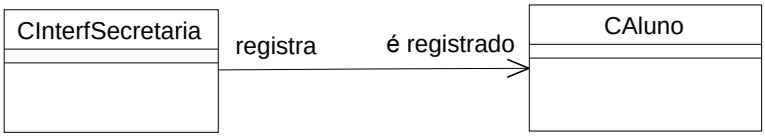

Figura IV.2 – Notação básica para relacionamentos de associação.

- **Cardinalidade**

A cardinalidade é uma indicação que especifica o número de objetos de cada classe envolvidos com a associação. Quando não há uma especificação de cardinalidade, entende-se que a cardinalidade é 1. Isto significa que apenas um objeto de cada classe está envolvido com a associação.

A especificação de cardinalidade é feita em cada extremo da associação e utiliza-se a seguinte notação :

Notação	Significado	Exemplo
Constante numérica	Indica um número fixo de objetos	5
Faixa de valores	Indica um número variável de objetos dentro da faixa de valores especificada	1..4
Presença ou Ausência	Indica nenhum ou um (ou mais) objetos	0..1
Indefinido	Indica um número qualquer de objetos	*

A figura IV.3 ilustra uma associação para a qual indica-se a cardinalidade.

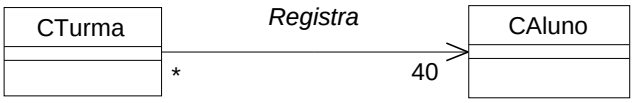

Figura IV.3 – Associação com cardinalidade.

A interpretação da cardinalidade exige alguma atenção. Na verdade existe uma forma correta de leitura da cardinalidade. Deve-se fazer a leitura da cardinalidade de forma distinta para os dois sentidos da associação (mesmo que ela seja unidirecional). Para cada um dos sentidos deve-se esquecer a cardinalidade no extremo de início da leitura e considerar que existe uma associação de UM objeto da classe de início da leitura com tantos objetos quanto estiver especificado pela cardinalidade da classe na outra extremidade da associação. Para o exemplo da figura IV.3, deve-se fazer a seguinte leitura:

"Um objeto da classe Cturma se associa com 40 objetos da classe CAluno, e um objeto da classe CAluno se associa com um número indefinido (inclusive zero) de objetos da classe CTurma".

1.2 Levantamento de Associações

Associações são definidas conforme seja necessário estabelecer um canal para comunicação entre objetos de duas classes. Basicamente, as associações são estabelecidas observando-se as necessidades de comunicação definidas nos diagramas de seqüência e resumidas nos diagramas de colaboração. Para cada par de classes, verifica-se em todos os diagramas de colaboração de todos os casos de uso, se existem mensagens trocadas entre objetos destas classes. Sempre que houverem mensagens trocadas, estabelece-se uma associação uni ou bidirecional conforme o sentido das mensagens. Procedendo-se desta forma para todas as classes do sistema, constrói-se uma rede de comunicação entre as classes.

Embora este procedimento de levantamento permita uma definição bem fundamentada das associações entre classes, é ainda necessário um trabalho de complementação deste processo. Como os diagramas de seqüência e de colaboração retratam cenários dos casos de uso, eles podem não cobrir a totalidade de situações de comunicação que podem se apresentar. Desta forma, é importante estudar o conjunto de associações que foram definidas para sua complementação (inclusão de eventuais associações extras) e para uma boa especificação de nomes e cardinalidades.

A figura IV.4 ilustra o resultado do levantamento de associações considerando o diagrama de colaboração da figura III.16.

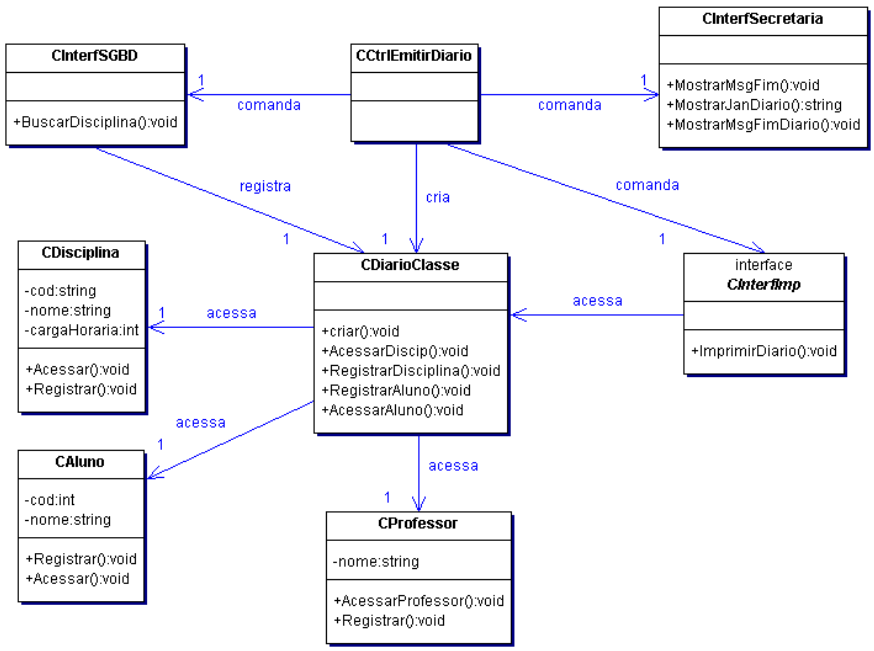

Figura IV.4 – Levantamento de Associações.

2 Agregação entre Classes

A agregação é um relacionamento pertinência entre classes que permite estabelecer a inclusão de objetos de uma classe no interior de objetos de outra classe. A agregação também é dita uma relação “parte-de” já que o objeto agregado passa a constituir ou fazer parte do objeto que agrega. Deve-se observar que não se trata de incluir (ou agregar) uma classe dentro de outra mas de objetos dentro de outros objetos. A relação de agregação permite criar composições de objetos, o que é bastante útil quando se deseja definir hierarquias de dados ou procedimentos.

A agregação fornece também um canal de comunicação entre o objeto que contém e o objeto contido. Note que esta comunicação é unidirecional do objeto que agrega para o objeto agregado. O objeto agregado não conhece, a princípio, o objeto que agrega. Desta forma, ele não pode comunicar-se com o objeto que agrega.

2.1 Notação UML para Agregações

A notação UML para agregações é a de um segmento de reta ligando a classe dos objetos que agregam à classe dos objetos agregados. Na extremidade da classe dos objetos que agregam inclui-se um losângulo, conforme ilustrado na figura IV.5.

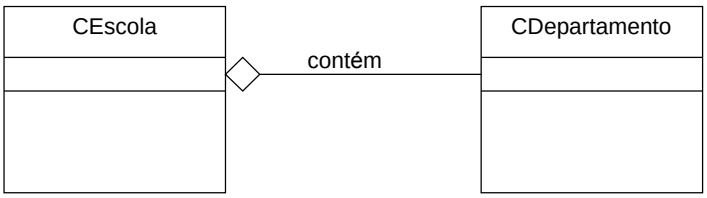

Figura IV.5 – Notação para agregações.

- **Nome e papéis**

Pode-se incluir nomes e papéis nas agregações como ocorre para as associações. Entretanto, como trata-se sempre de uma relação de pertinência, a inclusão de nomes e papéis torna-se desnecessária pois os nomes seriam sempre “inclui”, “contém” ou outro equivalente e os papéis seriam sempre “contém” e “está contido”, ou equivalente.

- **Cardinalidade**

A definicação de cardinalidade tem nas agregações a mesma finalidade do que nas associações. Determina-se através dela o número de objetos envolvidos na relação. Deve-se observar que embora um objeto possa incluir vários outros objetos, um objeto não pode estar contido em mais de um objeto. Desta forma, a cardinalidade no lado da classe dos objetos que agregam será sempre 1, podendo ser suprimida.

A figura IV.6 ilustra um exemplo de agregação com cardinalidades. Neste exemplo, defini-se que um objeto da classe CCarro contém 4 objetos da classe CRoda e 5 objetos da classe CPorta.

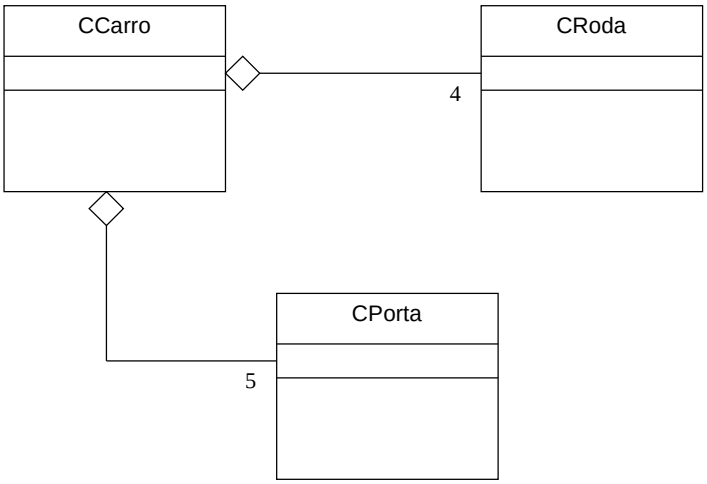

Figura IV.6 – Agregações com cardinalidade.

2.2 Definição de Agregações

O levantamento e definição das agregações em um certo sistema é realizado a partir de diferentes análises. Não existe uma técnica precisa para se definir as agregações em um sistema. De uma forma geral, pode-se utilizar certos raciocínios como sugestão para a definição de agregações, conforme apresentado a seguir.

- **Definição de agregações a partir de decomposições**

  Durante o levantamento e especificação de classes no projeto de um sistema, pode-se perceber que determinada classe tem um conjunto extenso de responsabilidades fazendo com que a classe tenha uma dimensão considerável. Uma tentativa para melhorar a interpretação da classe e, talvez, sua organização seria desmembrar a classe em classes menores. As classes, então, passariam a se comunicar para atender aos serviços originais da classe maior. Esta solução traz como inconveniente o fato de a classe original, que provavelmente tinha uma identidade, se desmembrar perdendo esta identidade. Por exemplo, se tivéssemos a classe CCarro se desmembrando em classes como Crodas, Cmotor e Ccarroceria, a figura e identidade da classe carro se dispersaria.

  Uma alternativa para o desmembramento é a decomposição da classe através de agregações. Neste caso, matém-se a classe original e criam-se classes adicionais que descrevem partes da classe principal. Através das agregações mantém-se a identidade da classe maior indicando que ela é composta por partes que são descritas separadamente.

- **Definição de agregações a partir de composições**

  Um raciocínio para definição de agregações exatamente contrário ao da decomposição é o da composição de objetos. Nesta visão, procura-se identificar conjuntos de objetos que juntos compõem objetos maiores (objetos agregados possuindo uma identidade). Nestas circunstâncias, pode-se criar novas classes definidas sobre relações de agregação com classes que representam suas partes. Deve-se observar que as agregações através de composição precisam ser definidas em função dos casos de uso do sistema pois, embora possível, nem todas as agregações podem interessar a um determinado sistema.

- **Definição de agregações a partir de partes comuns**

Agregações podem também ser obtidas através da identificação de partes comuns a duas ou mais classes. Isto se aplica quando percebe-se dentro de duas ou mais classes um conjunto de atributos e/ou métodos semelhantes. Se estes atributos e/ou métodos juntos possuem alguma identidade, então poderia ser criada uma nova classe extraindo-se a parte comum das classes originais e definindo-se relações de agregação com esta nova classe.

A figura IV.7 apresenta um exemplo de definição de agregações a partir da identificação de partes comuns. Neste exemplo, percebeu-se que os atributos Rua, Numero, Cidade, UF e CEP são os mesmos nas classes CAluno e CProfessor. Como mostra o exemplo, optou-se por criar uma nova classe chamada CEndereco incluindo os atributos comuns citados. Através de relações de agregação as classes CAluno e CProfessor incorporam novamente os atributos de endereço a suas listas de atributos.

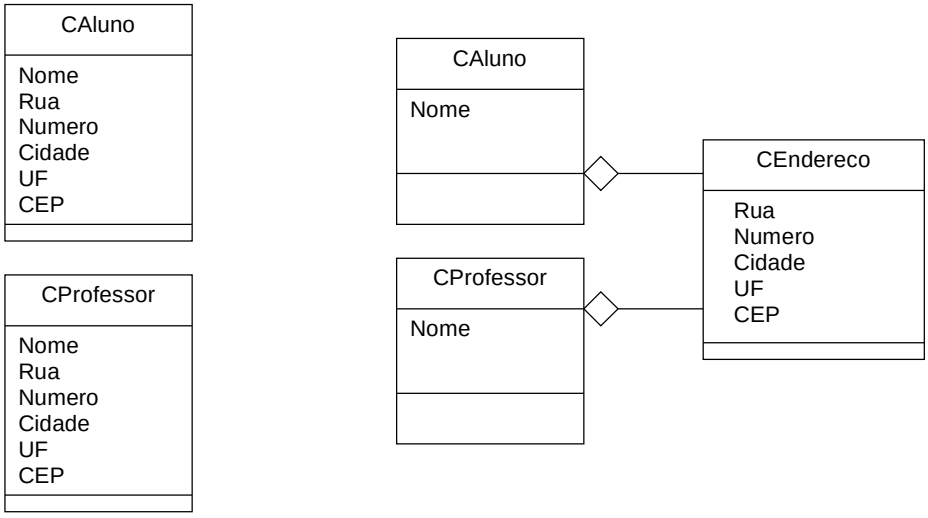

Figura IV.7 – Definição de agregações a partir de partes comuns.

Deve-se observar no exemplo da figura IV.7 que embora a classe CEndereco possua uma agregação com ambas as classes CAluno e CProfessor, isto não significa que um mesmo objeto desta classe esteja contido nas duas classes que agregam. Tratam-se de dois objetos distintos, um agregado à classe CAluno e outro à classe CProfessor.

2.3 Tipos de Agregação

A linguagem UML permite a definição de dois tipos de agregação chamadas agregação por composição e agregação por associação, conforme descrito a seguir.

- **Agregação por Composição**

Uma agregação por composição é uma agregação de fato. Isto significa que realmente é feita a criação ou alocação de um objeto dentro de um outro objeto. A alocação do objeto interno é feita estaticamente pela instanciação do objeto. A figura IV.8 mostra um trecho de código em C++ ilustrando a definição de uma agregação por composição. O objeto `ender` da classe CEndereco está sendo instanciado dentro da classe CAluno. Deve-se observar que na agregação por composição é necessário que o número de objetos agregados seja fixo uma vez que eles são alocados estaticamente.

A notação UML para agregações por composição é um retângulo preenchido como ilustrado na figura IV.8.

```
class CAluno
{
    char nome[30];
    CEndereco ender;

     ...
};
```

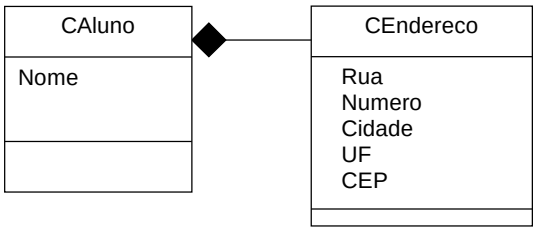

Figura IV.8 – Exemplo de agregação por composição.

- **Agregação por Associação**

A agregação por associação tem a mesma interpretação do que a agregação por composição, ou seja, entende-se que o objeto agregado é um componente do objeto que agrega. A grande diferença está na forma de alocação do objeto agregado. Na agregação por associação a alocação do objeto agregado não ocorre de forma estática no interior da classe do objeto que agrega. A alocação do objeto ocorre de forma estática fora do objeto que agrega ou de forma dinâmica em seu interior. O objeto que agrega guarda apenas o identificador ou endereço do objeto agregado. Por conseqüência, uma agregação por associação não é rigorosamente uma agregação.

As agregações por associação têm este nome pois sua implementação ocorre através de associações. Elas são consideradas agregações quando realiza-se um controle de escopo para os objetos agregados. Desta forma, os objetos agregados deveriam ser criados, acessados e destruídos unicamente pelo objeto que agrega estabelecendo assim laços de agregação. Note que em termos de implementação não existe diferença entre a associação e a agregação por associação.

A agregação por associação é necessária quando deseja-se estabelecer uma agregação envolvendo um número variável de objetos. Como dito anteriormente, a agregação por composição só pode ser definida para um número fixo de objetos.

A figura IV.9 ilustra um trecho de código em C++ e a respectiva representação em UML de uma agregação por associação de um número variável de objetos. Note que, como na associação, a implementação da agregação por associação é feita através de ponteiros em C++.

```
class CTurma
{
    char codigo[8];
    CAluno *aluno[45];

    ...
};
```

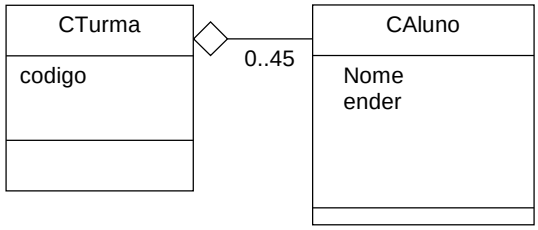

Figura IV.9 – Exemplo de agregação por composição.

3 Generalização/Especialização de Classes

3.1 Notação UML

A generalização ou especialização é um relacionamento que ocorre efetivamente em termos estruturais entre duas classes. Das duas classes envolvidas, uma será considerada uma **classe base** (ou superclasse) e a outra será considerada uma **classe derivada** (ou subclasse). O significado deste relacionamento depende do sentido de leitura da relação. Lendo-se a relação no sentido da classe base para a classe derivada entende-se a relação como uma especialização, ou seja, a classe derivada seria uma classe especial definida a partir da classe base. Lendo-se a relação no sentido da classe derivada para a classe base, entende-se a relação como uma generalização, ou seja, a classe base representa um caso geral da classe derivada.

Independentemente do significado da relação, a implementação da generalização ou especialização ocorre sempre na forma de uma **herança** da classe base para a classe derivada. A herança é um processo através do qual a classe derivada incorpora em seu interior todos os atributos e métodos da classe base. A classe derivada pode incluir atributos e métodos adicionais além daqueles herdados da classe base. Note que a herança é diferente da agregação no sentido em que na agregação incluía-se um objeto dentro do outro preservando-se o objeto que foi incluído. Na herança a classe base não se mantém no interior da classe derivada. Apenas os atributos e métodos são incorporados dentro da classe derivada onde passam a ser considerados como atributos e métodos normais.

A notação UML para generalização ou especialização é um segmento de reta ligando as classes, com um triângulo colocado no extremo da classe base. A figura IV.10 ilustra um relacionamento de generalização entre as classes CAutomovel e CCarro e apresenta o código C++ correspondente. Observe que no código C++, é na especificação da classe derivada (CCarro) que se define a herança. Neste exemplo, um objeto da classe CCarro teria, além dos atributos n_portas e placa, todos aqueles da classe CAutomovel. O mesmo ocorreria para os métodos.

```
class CAutomovel
{   char modelo[30];
    char fabricante[30];
    int  ano;
    char cor;
  public:
    DefineAuto(char,char,char,int);
};

class CCarro::public CAutomovel
{   int n_portas;
    int placa;
  public:
    DefineCar(char,char,char,int,int,int)
};
```

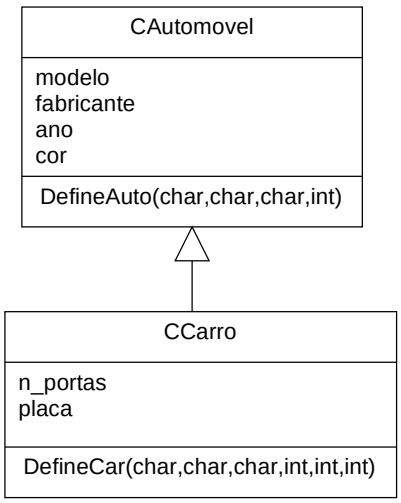

Figura IV.10 – Exemplo de herança.

Observe que para relacionamentos de herança não é necessário definir um nome pois a interpretação do relacionamento é única e por conseqüência o nome torna-se desnecessário. De forma semelhante, não se especifica cardinalidade para relacionamentos de herança pois ela é sempre uma relação de uma única classe para outra classe.

3.2 Herança Múltipla

A herança múltipla ocorre quando uma classe deriva de mais de uma classe base. Mantém-se exatamente os mesmos princípios da herança porém a classe derivada passa a incorporar em seu interior os atributos e métodos de mais de uma classe base. A figura IV.11 ilustra o código e notação UML para uma herança múltipla.

```
class CAutomovel
{   char modelo[30];
    char fabricante[30];
    int  ano;
    char cor;
  public:
    DefineAuto(char,char,char,int);
};

class CBemMovel
{   int   numPatrimonio;
    float preco;
    int   depreciacao;
    int   anoCompra;
  public:
    DefineBem(int, float, int, int);
};

class CCarro::public CAutomovel, public
CBemMovel
{   int n_portas;
    int placa;
  public:
    DefineCar(char,char,char,int,int,int)
};
```

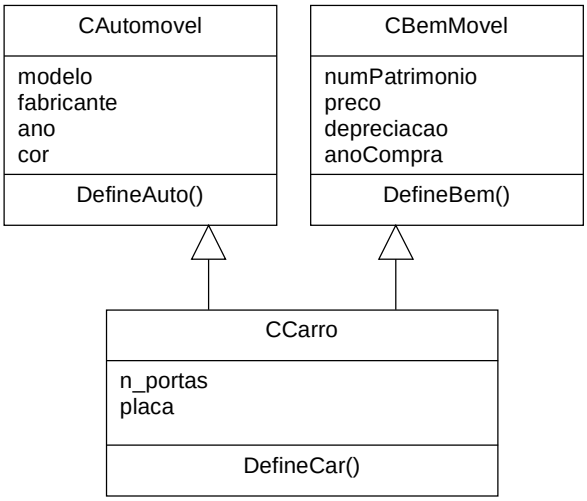

Figura IV.11 – Exemplo de herança múltipla.

O conceito de generalização se perde, em parte, no caso da herança múltipla pois cada classe base não representa mais um caso geral de toda a classe derivada, mas sim um caso geral de uma parte da classe derivada. No exemplo da figura IV.11 não se pode mais afirmar que a classe CAutomovel seja uma generalização completa de CCarro. CAutomovel é, na verdade, uma generalização parcial pois só considera aspectos específicos de CCarro. O conceito de especialização se matém na herança múltipla. No exemplo da figuta IV.11, CCarro é um caso especial de CAutomovel e é também um caso especial de CBemMovel.

3.3 Levantamento de Generalizações

O levantamento da generalizações apropriadas para um determinado sistema envolve um trabalho de análise e de abstração sobre as classes. Existem duas formas principais de se estabelecer generalizações em um sistema, conforme descrito abaixo:

- **Identificação de partes comuns entre classes**

Comparando-se classes, pode-se peceber que duas ou mais possuem partes (atributos e/ou métodos) em comum. Quando estas partes em comum possuem uma dimensão considerável (por exemplo, representam mais da metade da classe) e ao mesmo tempo as partes em comum definem a essência das classes, pode-se criar uma classe base contendo as partes comuns e utilizar herança para integrá-las nas classes derivadas. A classe base definiria, neste caso, uma classe geral para as classes derivadas. Este é exatamente o raciocínio da generalização, tomar classes específica e tentar estabelecer classes mais gerais.

- **Síntese de classes base**

Outra forma de definição de relacionamentos de herança é realizar a concepção das classes a partir de classes abstratas. Estas classes representariam, sempre, soluções gerais. A seguir, através de um raciocínio de especialização, classes mais específicas seriam geradas através da herança. Este procedimento de síntese de classes base é freqüêntemente utilizado quando controem-se bibliotecas de classes. A maior parte das classes nestas bibliotecas são classes gerais utilizadas para criação de classes derivadas.

Na prática, utilizam-se as duas abordagens descritas durante os projetos de *software* orientados a objetos.

4 Conclusão

Neste capítulo fez-se uma apresentação resumida dos relacionamentos entre classes em *software*s orientados a objetos. Um bom conhecimento do significado e aplicação destes relacionamentos é absolutamente necessário para os projetistas de *software*. Os relacionamentos entre classes fazem parte da definição da estrutura do sistema e serão mapeados diretamente em código durante a fase de programação. É, portanto, necessário que todos os relacionamentos estejam corretamente definidos dentro do projeto do sistema.

Os relacionamentos de associação podem, em grande parte, ser obtidos a partir do estudo dos relacionamentos entre objetos feito através de diagramas de seqüência e colaboração. Os relacionamentos de agregação e herança exigem uma reflexão sobre a constituição das classes de forma a que se possa identificar partes comuns e partes destacáveis.

Capítulo 5 : Definição da Dinâmica dos Objetos

Os estudos e especificações feitos até este ponto permitiram definir a estrutura do sistema computacional. Basicamente, definiu-se os componentes do sistema (classes) e seus relacionamentos. Para cada classe definiu-se um conjunto inicial (mas talvez ainda incompleto) de atributos e métodos. Embora estas informações já permitam a codificação (programação) do esqueleto do sistema, elas são ainda insuficientes para a codificação do interior das classes (dos métodos em particular).

De fato, é necessário desenvolver algum estudo sobre o comportamento interno das classes de maneira a permitir a especificação da dinâmica, ou seja, da forma como os objetos de cada classe se comportam. Isto corresponde a uma especificação de como as classes devem ser implementadas.

1 Diagrama de Estados

Segundo a UML, a especificação da dinâmica do sistema deve ser feita através de diagramas de estados. Constrói-se um diagrama de estados descrevendo o comportamento de cada classe e eventuais diagramas complementares para descrever a dinâmica de todo o sistema ou de certos módulos. Deve-se observar que, no caso das classes, o diagrama de estados deve reunir o comportamento de uma classe com todas as suas responsabilidades, ou seja, com o seu comportamento completo em todos os casos de uso. Desta forma, o diagrama de estados de uma classe é uma descrição global do comportamento dos objetos desta classe em todo o sistema.

1.1 Notação Básica

A UML utiliza como notação para diagramas de estados a notação proposta por Harel. Nesta notação, um diagrama de estados é um grafo dirigido cujos nodos representam estados e cujos arcos representam transições entre estados. Estados são representados graficamente por elipses ou retângulos com bordas arredondadas e transições são representadas por segmentos de retas dirigidos (com uma seta em uma das extremidades). Descreve-se a seguir os conceitos de estado e transição de estado.

- **Estado de um Objeto**

  Um estado pode ser entendido como um momento na vida de um objeto. Neste momento (ou estado) o objeto encontra-se em uma certa situação. Assim, ao longo da vida de um objeto, desde sua criação até seu desaparecimento, ele pode passar por vários momentos diferentes: momento em que foi criado, momento em que fez uma inicialização, momento em que fez uma certa solicitação, etc até o momento de seu desaparecimento. Dito de forma análoga, um objeto ao longo de

sua vida passa por um conjunto de diferentes estados. Nos diagramas de estados procura-se descrever este conjunto de estados e seu encadeamento.

Os estados em um diagrama de estados são identificados por um nome. O nome pode ser uma palavra ou um conjunto de palavras que descreva adequadamente a situação representada pelo estado. Com alguma freqüência, os estados são identificados com nomes começados por um verbo no gerúndio ou particípio. Deve-se observar que um mesmo estado pode ser repetido em um diagrama de estados. Neste caso os nomes se repetem em mais de um posição do diagrama.

Existem duas notações gráficas especiais para dois estados particulares existentes em diagramas de estados. Utiliza-se a notação de um círculo preenchido para indicar o estado inicial do diagrama de estados. O estado inicial é um estado de partida do objeto. Pode-se entender que ele representa o momento de criação ou alocação do objeto. Utiliza-se a notação de dois círculos concêntricos para representar o estado final de um objeto. O estado final representa o momento de destruição ou desalocação do objeto.

Exemplos de estados de um objeto :

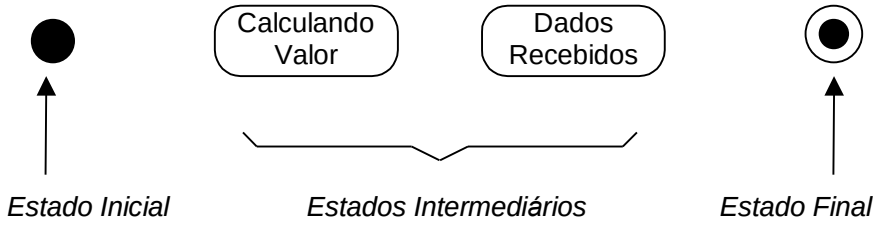

Figura V.1 – Exemplos de estados de um objeto.

- **Transição de Estado**

  Como se pode deduzir do conceito de estado, um objeto não fica permanentemente em um único estado. Os objetos tendem a avançar de um certo estado para algum outro à medida que executem seus processamentos. Este avanço de uma situação (estado) para outra denomina-se uma transição de estado. Assim, em um diagrama de estados descreve-se o conjunto de estados de um objeto e também o conjunto de transições de estados existentes. Observe que a partir de um certo estado do objeto ele só pode evoluir para certos estados (normalmente, não para todos os outros) criando assim caminhos que representam o ou os fluxos de execução daquele objeto.

  Uma transição pode possuir um rótulo com a seguinte sintaxe :

Evento(Argumentos) [Condição] / Ação

  Evento : indica o nome de um sinal, mensagem ou notificação recebida pelo objeto e que torna a transição habilitada podendo, por conseqüência, ocorrer levando o objeto a um novo estado. Exemplos de eventos são: o recebimento de uma mensagem encaminhada pelo sistema operacional, o recebimento de uma notificação (*timer*, interrupção, entrada de dados) gerada pelo sistema operacional e a chamada de uma função feita por outro objeto.

Argumentos : são valores recebidos junto com o evento.

[Condição] : é uma expressão lógica que é avaliada quando o evento, se houver, associado a uma transição ocorrer. Uma transição só ocorre se o evento acontecer e a condição associada for verdadeira.

/ Ação : indica uma ação (cálculo, atribuição, envio de mesagem, etc) que é executada durante a transição de um estado a outro.

A figura V.2 ilustra um diagrama de estados com cinco estados e quatro transições. Observe que as duas transições mais à esquerda e a transição à direita não possuem nenhum rótulo. Por conseqüência, estas transições ocorrem imediantamente após a realização das eventuais ações associadas aos estados imediatamente anteriores. A terceira transição possui como rótulo o nome de um evento chamado “dados”. Este evento representa a chegada de dados de algum canal de entrada (teclado por exemplo) notificada pelo sistema operacional. Neste caso, a transição só ocorrerá quando o evento “dados” acontecer. Enquanto o evento não acontecer, o objeto permanecerá no estado anterior que é “Aguardando Dados”. Note que é comum nomear o estado anterior a uma transição com evento utilizando-se um verbo no gerúndio e nomear o estado seguinte com um verbo no particípio como no exemplo da figura V.2.

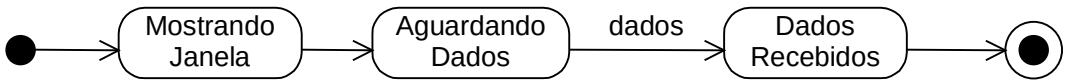

Figura V.2 – Exemplos de transições de estados.

1.2 Fluxo de Estados e Transições

A partir das primitivas Estado e Transição pode-se construir uma grande variedade de diagramas de maior ou menor complexidade. Os encadeamentos de estados e transições podem estabelecer certas construções típicas como descrito a seguir.

- **Seqüências**

  Seqüências são fluxos de estados representados por encadeamentos de um estado e uma transição. O comportamento do objeto segue, por conseqüência, uma série de estados bem determinados. O diagrama na figura V.2 descreve um fluxo seqüencial de estados e transições. As transições podem ou não possuir rótulos com eventos, condições e ações.

- **Bifurcações e Junções**

  Uma bifurcação é representada por duas ou mais transições partindo de um mesmo estado. Nesta situação, tem-se mais de uma possibilidade de continuação do fluxo comportamental do objeto. A partir do estado de bifurcação, o objeto poderia alcançar algum outro estado conforme a transição que ocorrer. Uma junção é representada por duas ou mais transições conduzindo a um mesmo estado. A figura V.3 ilustra um exemplo de diagrama de estados com bifurcação e junção.

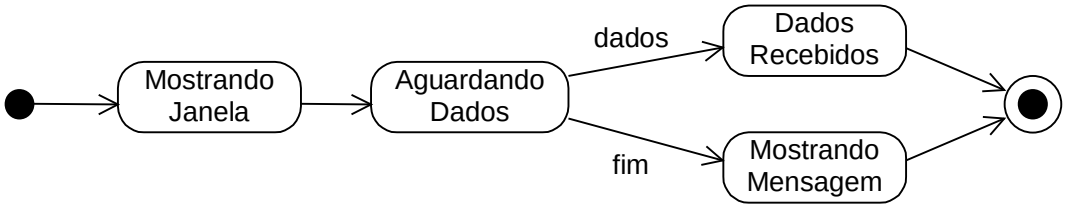

Figura V.3 – Exemplo de bifurcação e junção em diagrama de estado.

Deve-se ter especial atenção na definição das bifurcações de forma a evitar conflitos. Um conflito ocorre quando duas ou mais transições partindo de um mesmo estado estão em habilitadas. Neste caso, existe uma indefinição sobre o comportamento do objeto. Para evitar conflitos, deve-se assegurar que as condições para ocorrência de cada transição são mutuamente exclusivas. A figura V.4 ilustra dois casos de conflito. No diagrama à esquerda, duas transições sem rótulo partem de um mesmo estado indicando claramente um conflito. No diagrama à direita, tem-se condições nas duas transições mas estas condições não são mutuamente exclusivas. Quando o valor da variável x for igual à 10 existirá um conflito.

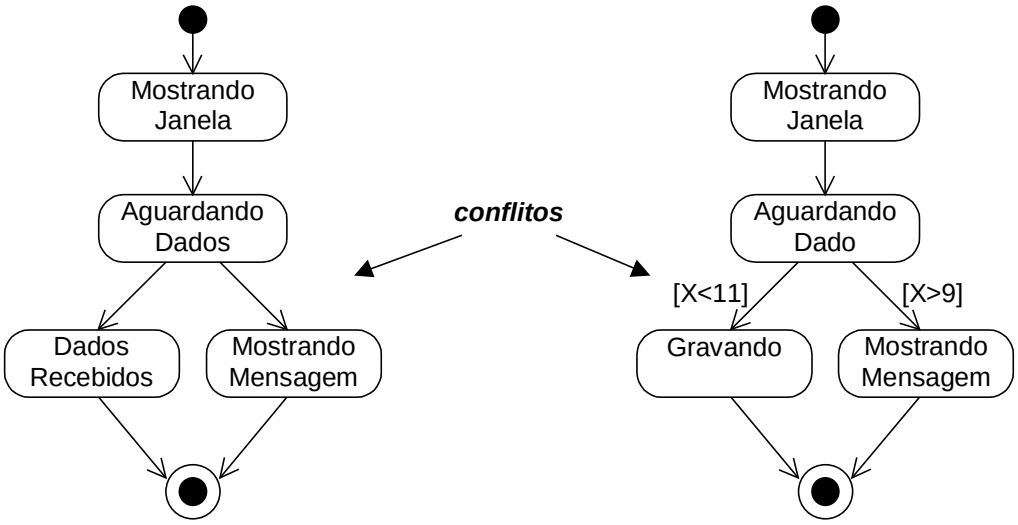

Figura V.4 – Exemplos de conflito entre transições.

- **Repetições**

  Uma repetição ou laço é representado por um encadeamento cíclico de estados e transições contendo, na maioria dos casos, alguma forma de controle sobre a repetição dos ciclos. A figura V.5 ilustra um diagrama de estados com uma repetição. Neste exemplo os três estados serão alcançados várias vezes até que o valor da variável de controle x ultrapasse o valor 10.

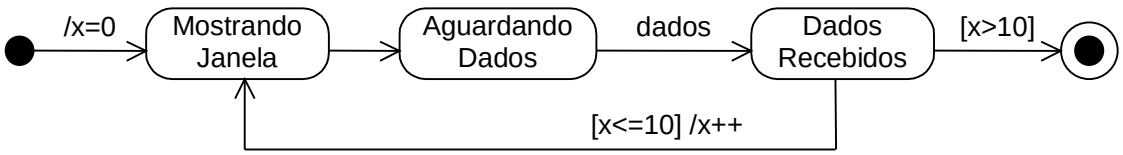

Figura V.5 – Exemplo de repetição em um diagrama de estados.

1.3 Notação Complementar

- **Cláusula de Envio**

  Uma cláusula de envio é uma ação de envio de uma mensagem do objeto que se está modelando para algum outro objeto. O envio da mensagem pode ser síncrono (como no caso de chamadas de funções) ou assíncronos (como no envio de mensagens via sistema de mensagens).

  Em diagramas de estados, indica-se que o objeto está enviando uma mensagem a outro objeto colocando-se uma cláusula de envio como uma ação em uma transição. Como notação para especificação da cláusula de envio coloca-se um acento circunflexo seguido do nome do objeto e do nome da mensagem separados por um símbolo de ponto.

  ```
  ^nome-do-objeto.nome-da-mensagem
  ```

  A figura V.6 mostra um exemplo de diagrama de seqüência e de um diagrama de estados da classe CCtrl com as mensagens enviadas.

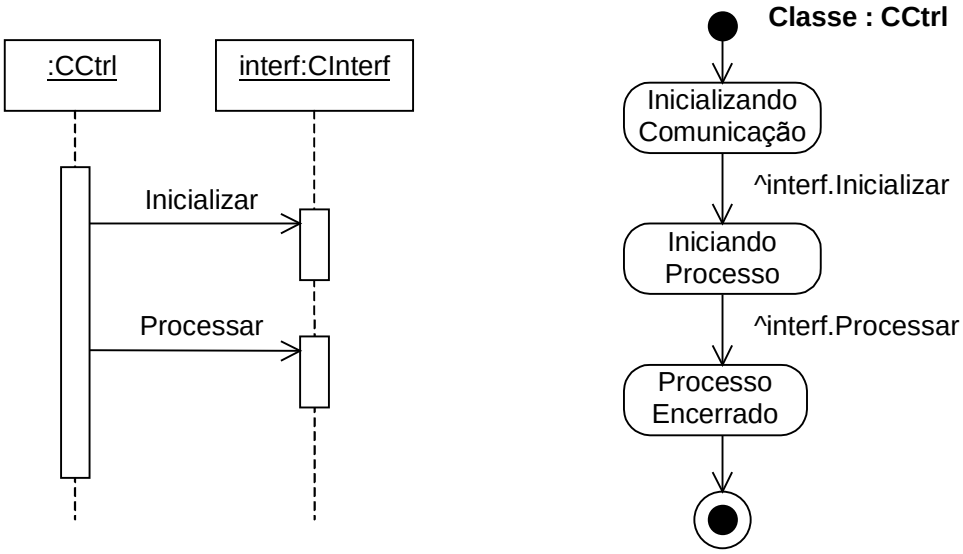

Figura V.6 – Exemplos de cláusulas de envio.

- **Transições Reflexivas**

  Uma transição reflexiva é uma transição que parte de um estado e alcança o mesmo estado de partida. Transições reflexivas representam situações em que ocorre algum evento ou alcança-se alguma condição a partir de um determinado estado produzindo, eventualmente, alguma ação mas sem afetar o estado no qual o objeto se encontra. Transições reflexivas podem ser utilizadas também para indicar que o objeto recebe certas mensagens ou percebe certos eventos sem, entretanto, alterar seu estado comportamental. A figura V.7 ilustra um caso de transição reflexiva.

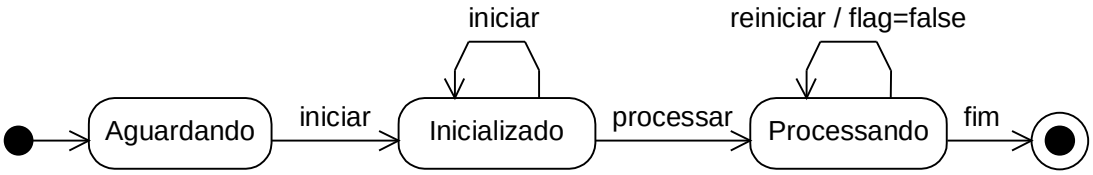

Figura V.7 – Exemplo de transições reflexivas.

- **Ações nos Estados**

  Como visto até este ponto, pode-se especificar ações em um diagrama de estados indicando-as junto às transições de estado. Outra possibilidade é incluir a especificação de ações no interior dos estados. A definição das ações no interior de um estado significa que sempre que aquele estado for alcançado, as ações serão realizadas. Graficamente, divide-se o retângulo de representação de um estado em dois compartimentos: o compartimento de identificação, que contém o nome do estado, e o compartimento das ações com a lista das ações realizadas no interior do estado. Para cada ação existe um prefixo indicando a categoria da ação conforme descrito a seguir.

  Categorias de ações :

  Entrada : uma ação de entrada é uma ação realizada exatamente no momento em que se alcança o estado. Esta ou estas ações são realizadas antes de qualquer outra ação. Na verdade, ações de entrada seriam ações que estariam nas transições que conduzem a certo estado e que, por conseqüência, seriam executadas antes de se alcançar efetivamente o estado. Note que colocando-se uma ação de entrada entende-se que ela será executada independente da transição que conduz ao estado. A figura V.8 mostra um exemplo com dois diagramas de estados equivalentes onde à esquerda especifica-se uma ação nas transições e no outro especifica-se a mesma ação no interior do estado.

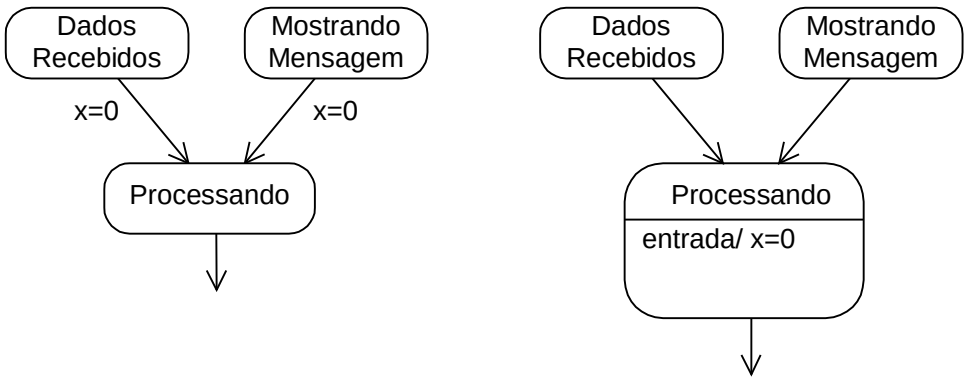

Figura V.8 – Exemplo de definição de ação de entrada.

  Saída : uma ação de saída é uma ação realizada exatamente no momento de abandono de um estado. Ações de saída seriam ações que estariam em todas as transições que partem de um determinado estado. A figura V.9 ilustra a especificação de uma ação de saída.

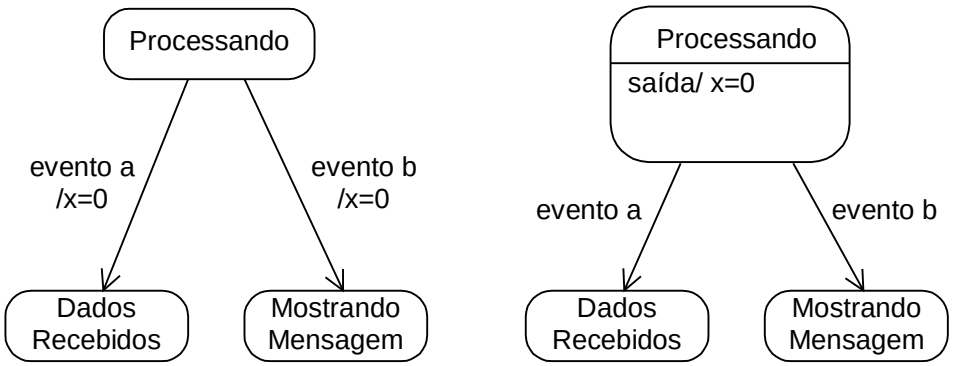

Figura V.9 – Exemplo de definição de ação de saída.

Fazer : o prefixo “fazer” (em inglês, *do*) permite especificar uma atividade não atômica (composta por mais de uma instrução) realizada no interior do estado. Isto significa que quando o objeto alcançar o estado e tiver concluído as eventuais ações de entrada, ele passará a executar a atividade indicada enquanto permanecer neste estado.

Evento : uma ação prefixada com um nome de evento será realizada quando o objeto estiver no estado correspondente e ocorrer o evento indicado. Como o evento é tratado sem mudança de estado pode-se considerar este tipo de especificação equivalente às transições reflexivas com ações. É possível utilizar-se também a palavra reservada *difer* (diferir) no lugar da ação. Neste caso, entende-se que se o evento ocorrer ele será diferido para tratamento posterior. Isto equivaleria a uma “bufferização” do evento para tratamento posterior.

- **Estados Compostos**

  Um estado composto é um estado constituído de um conjunto de subestados. Cada subestado pode, por sua vez, ser também um estado composto constituído por subestados. Utiliza-se o conceito de estado composto para construções hierarquizadas que apresentam nos níveis iniciais, estados maiores (mais abstratos) e em níveis seguintes estados mais específicos (menos abstratos).

  Os subestados são encadeados no interior de um estado composto na forma de um diagrama de estado. Desta forma, tem-se um estado inicial, estados intermediários e um estado final. A figura V.10 ilustra um estado composto mostrado seus subestados interiores. Note que esta ilustração mostra uma visão expandida do diagrama de estados. Em uma visão normal, o estado composto “Processando Entrada de Dados” seria mostrado sem seus subestados.

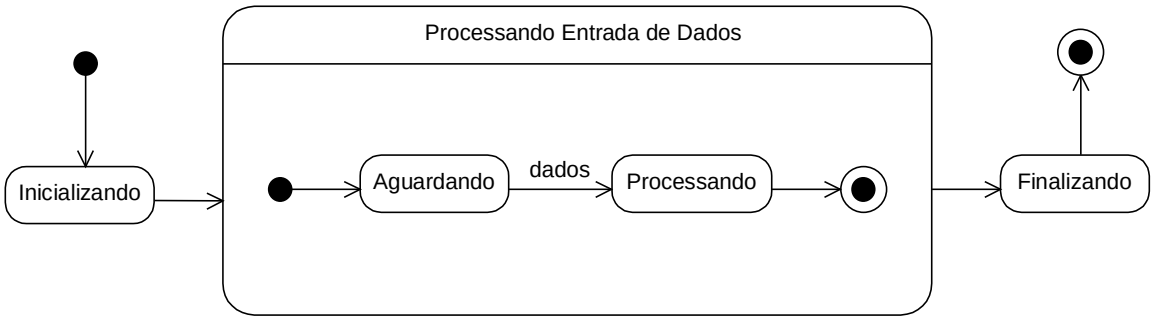

Figura V.10 – Exemplo de estado composto.

1.4 Concorrências

Estados compostos podem descrever em seu interior concorrências. Estas concorrências representam dois ou mais encadeamentos de estados e transições que são percorridos simultaneamente. Em termos de programação, a concorrência dentro de um objeto significa que ele possui mais de um fluxo de controle implementado através de *threads* e utilizando serviços de multitarefa ou multiprocessamento do sistema operacional.

A representação de concorrência se faz através da divisão de um estado composto em regiões que são separadas por linhas tracejadas. Cada região descreve um fluxo concorrente através de um subdiagrama de estados. A figura V.11 ilustra um estado composto com uma concorrência entre dois encadeamentos de estados.

Existem dois pontos de sincronismo explícitos em estados compostos com concorrência. O primeiro é representado pelo estado inicial. Quando um objeto alcança o estado composto, imediatamente abre-se a concorrência alcançando-se igualmente os estados iniciais de todas as concorrências. A partir deste ponto perde-se o sincronismo das concorrências pois cada uma irá evoluir no encadeamento de seus subestados conforme as condições que se apresentarem. O segundo ponto de sincronismo explícito entre as concorrências está no estado final de cada uma. O estado composto só poderá evoluir para estados seguintes quando todas as suas concorrências tiverem alcançado seus estados finais. Dito de outra forma, aguarda-se que todas as concorrências alcancem seus estados finais para que o objeto possa evoluir do estado composto atual para algum próximo estado.

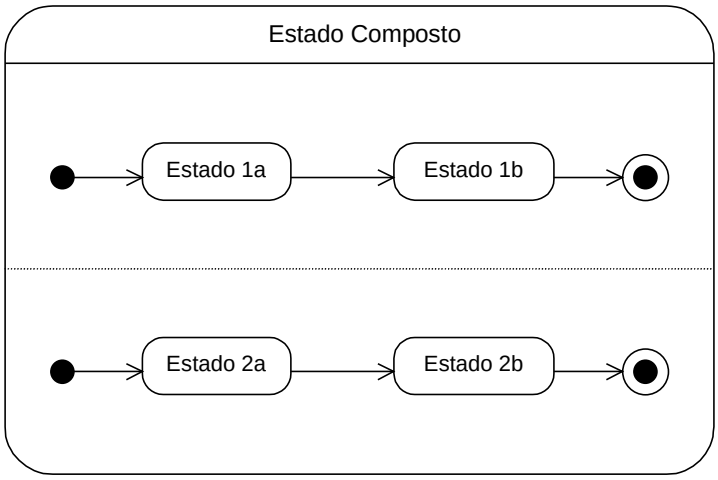

Figura V.11 – Exemplo de concorrência em um estado composto.

Em algumas circunstâncias pode ser desejado estabelecer sincronismos entre os estados intermediários de duas ou mais concorrências. Isto representaria uma dependência de estado entre as concorrências. Uma dependência de estado significa que alguma transição de estado em uma concorrência depende do estado em que se encontra outra concorrência. A notação UML para indicar sincronismo de estados é a de um estado especial de sincronismo representado por um círculo colocado sobre a linha tracejada de divisão de concorrências. Na verdade, o estado de sincronismo não deve ser interpretado

verdadeiramente como um estado mas sim como um contador ou fila agindo como uma condição.

A figura V.12 ilustra a notação para sincronismo entre concorrências. O círculo sobre a linha tracejada indica um estado de sincronismo. O número 1 dentro do estado indica a capacidade de armazenamento de marcas de sincronismo no estado (corresponderia ao limite do contador). No exemplo, sempre que o estado 1a for alcançado e concluir suas ações, uma marca será colocada no estado de sincronismo, ou seja, o contador será incrementado. O estado 1a é um estado especial pois representa uma disjunção (um *fork*) através da qual as duas transições seguinte ocorrem. Sempre que a transição entre os estados 2a e 2b ocorrer, uma marca será retirada do estado de sincronismo, ou seja, o contador será decrementado. Quando o estado de sincronismo já contiver o número máximo de marcas, as transições seguintes ao estado 1a não poderão mais ocorrer até que alguma marca seja retirada do estado de sincronismo. Neste caso, o contador teria alcançado seu limite máximo. A transição entre os estados 2a e 2b não poderá ocorrer se não houverem marcas no estado de sincronismo pois o estado 2b depende do sinal de sincronismo representado por uma marca no estado de sincronismo.

Para representar capacidade infinita para a capacidade do estado de sincronismo utiliza-se o símbolo * no interior do estado. Neste caso, o contador pode ser incrementado infinitamente (pelo menos teoricamente).

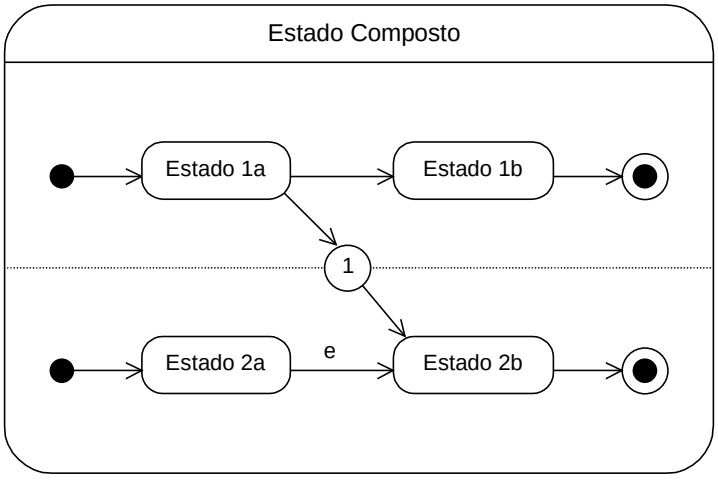

Figura V.12 – Exemplo de sincronismo entre concorrências.

1.5 Exemplos de Diagramas de Estados

Nesta seção serão apresentados três exemplos de diagramas de estados com o propósito de ilustrar a aplicação deste tipo de diagrama. Considera-se nestes exemplos apenas o diagrama de seqüência da figura II.16.

- **Diagrama de estados da classe CDiarioClasse**

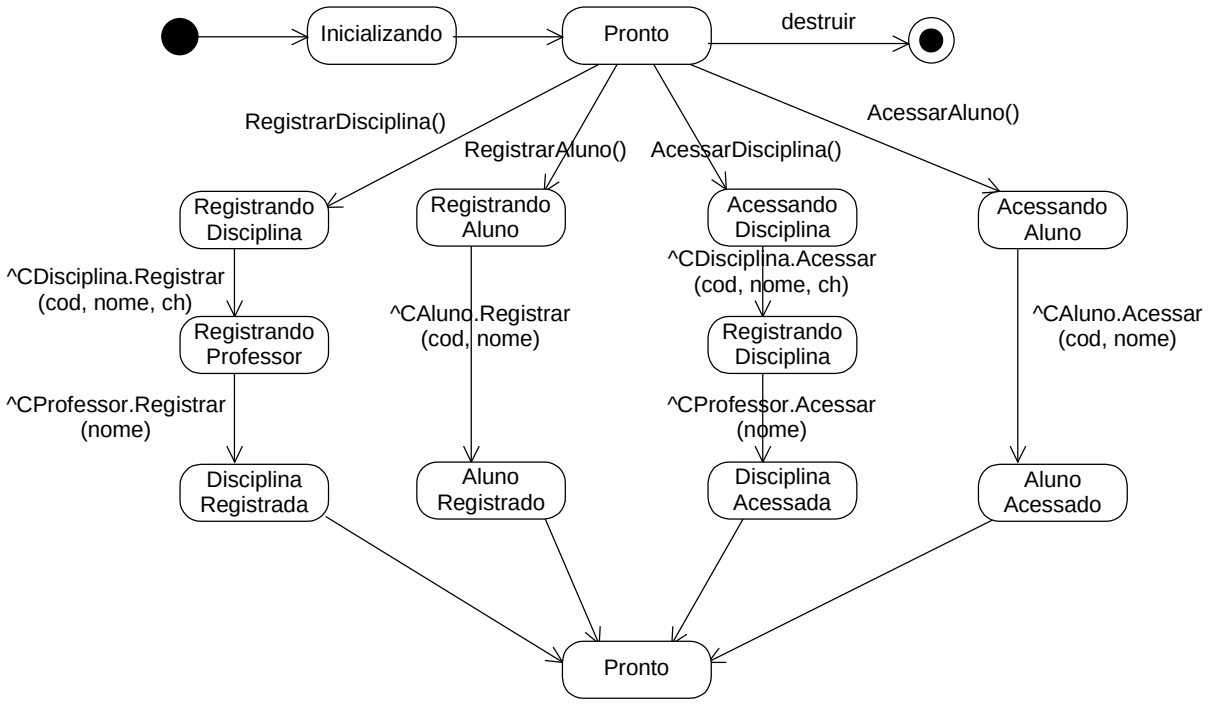

Figura V.13 – Diagrama de estados para a classe CDiarioClasse

- **Diagrama de estados da classe CInterfSGBD**

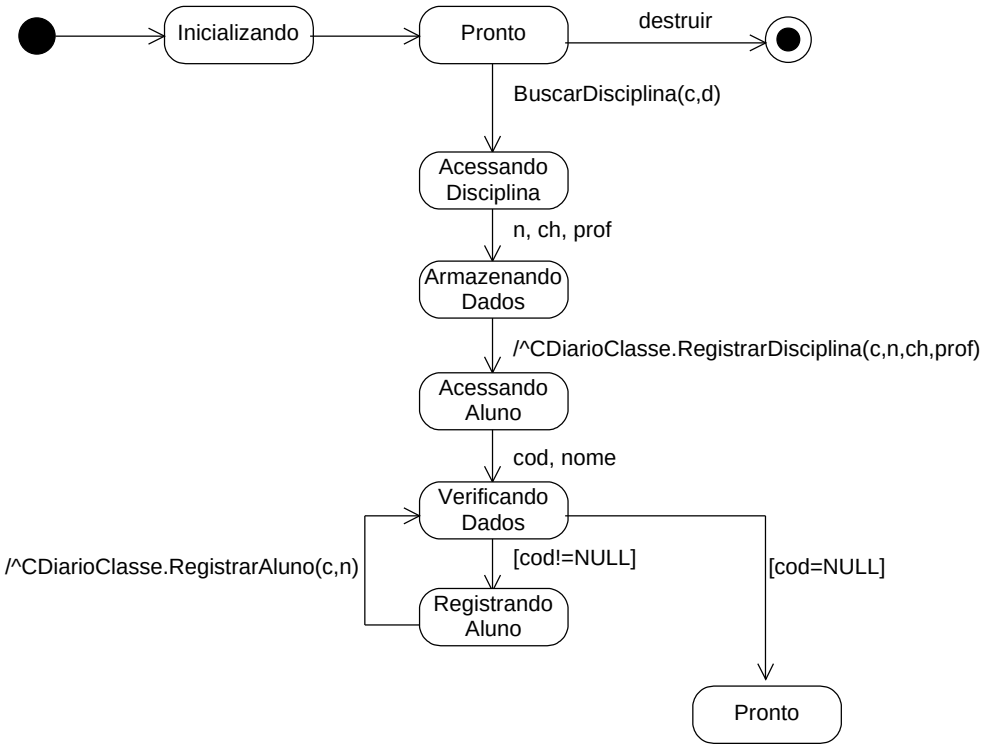

Figura V.14 – Diagrama de estados para a classe CInterfSGBD

Deve-se notar que o diagrama de estados da Figura V.14 está provavelmente incompleto pois só se considerou o caso de uso Emitir Diário de Classe.

- **Diagrama de estados da classe CctrlEmitirDiario**

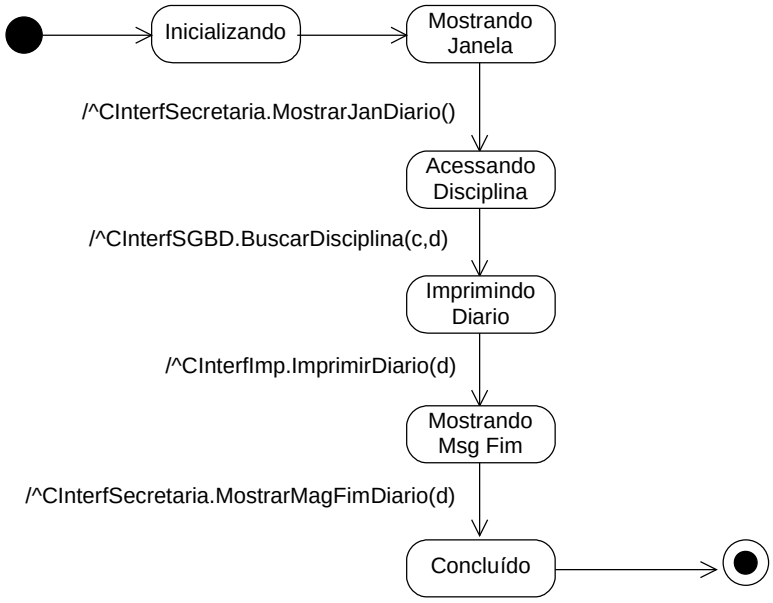

Figura V.15 – Diagrama de estados para a classeCEmitirDiario.

2 Diagramas de Atividades

2.1 Notação Básica

Um diagrama de atividades é um diagrama de estado no qual considera-se que todos ou a grande maioria dos estados representam a execução de ações ou atividades. A notação UML para diagramas de atividades utiliza as mesmas primitivas dos diagramas de estados e inclui algumas notações adicionais.

As novas primitivas incluídas nos diagramas de atividades são:

- **Estado de Bifurcação e de Convergência**

Um estado de bifurcação é um estado do qual partem duas (eventualmente mais) transições. Estados deste tipo são representados em diagramas de atividades utilizando-se uma notação especial na forma de um losango. Observe que para evitar um conflito entre as transições é necessária a colocação de condições ou eventos mutuamente exclusivos nas transições. A figura V.16 ilustra um diagrama de estados com um estado de bifurcação.

A mesma notação utilizada para estados de bifurcação pode ser utilizada para estados de convergência. Um estado de convergência é um estado que possui mais de uma transição de entrada representando, desta forma, um ponto de junção de fluxos alternativos dentro de um diagrama de estados. A figura V.16 ilustra um estado de convergência.

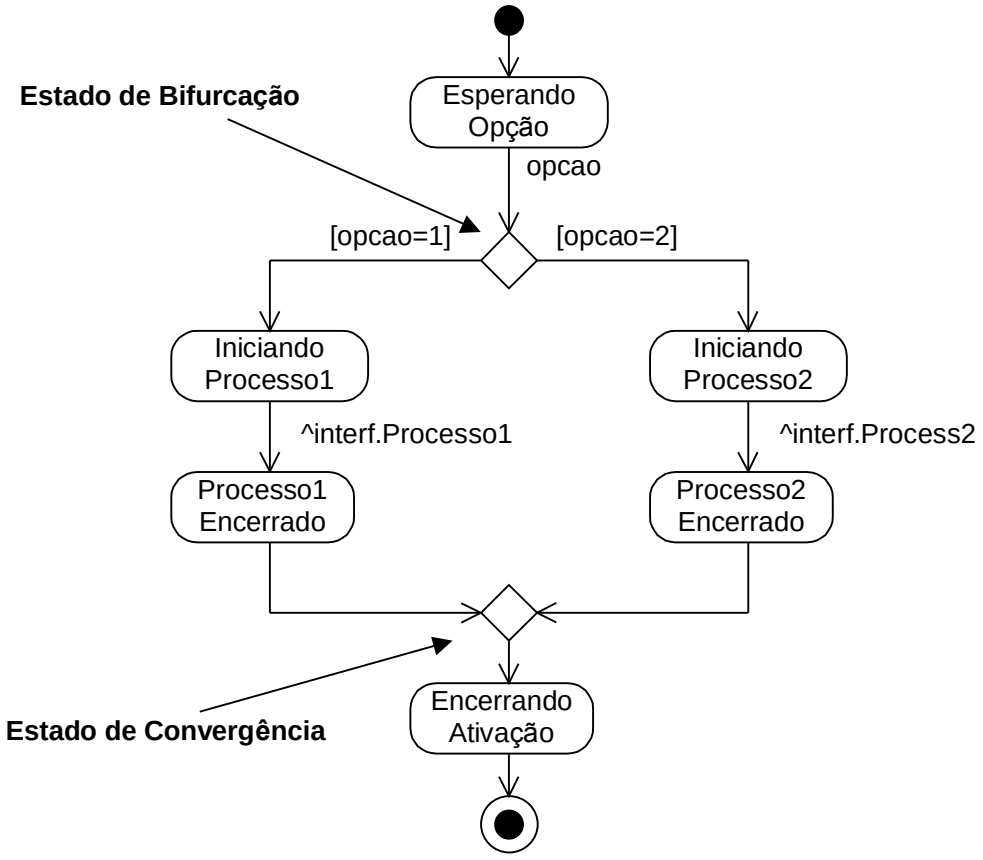

Figura V.16 – Exemplo de estados de bifurcação e de convergência.

- **Sincronismo de Concorrências**

Os diagramas de atividade empregam uma nova notação permitindo a especificação de sincronismo de concorrências. A mesma notação, na forma de uma barra preenchida, é utilizada tanto para abertura de sincronismo quanto para sincronismo de concorrências. Na abertura, entende-se que os fluxos seguintes avançarão concorrentemente. O sincronismo de encerramento de concorrências representa um ponto de aguardo até que todos os fluxos concorrentes alcancem o ponto de sincronismo. A figura V.17 ilustra um diagrama de atividades com concorrência.

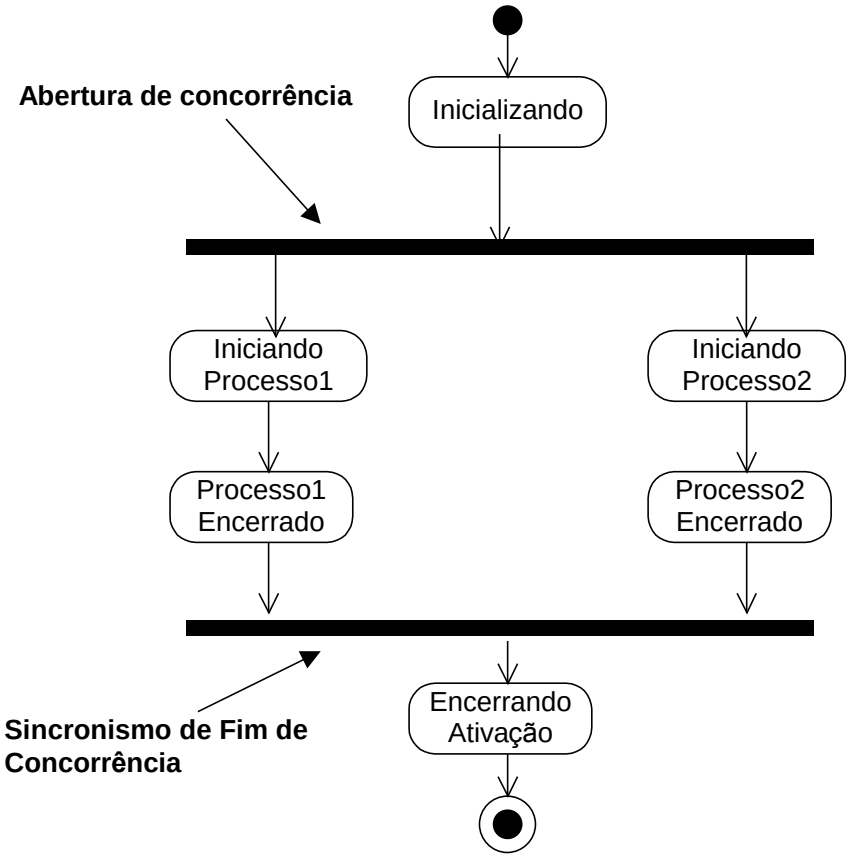

Figura V.17 – Exemplo de abertura e sincronismo de encerramento de concorrências.

2.2 Notação Complementar

- **Recebimento de mensagens ou sinais**

A UML oferece uma notação alternativa para especificação de situações de recebimento de mensagens ou sinais. Utiliza-se um pentágono côncavo com o nome da mensagem ou sinal em seu interior. Pode-se denominar este pentágono como sendo um "estado de recebimento de sinal". Esta notação é utilizada no lugar de apenas uma transição com o nome da mensagem ou sinal. Opcionalmente, pode-se incluir, próximo ao estado de recebimento de sinal, o objeto gerador do sinal ligado ao estado através de um seta tracejada. A figura V.18 apresenta uma ilustração que compara a representação normal de recebimento de sinal e a notação através de estado de recebimento.

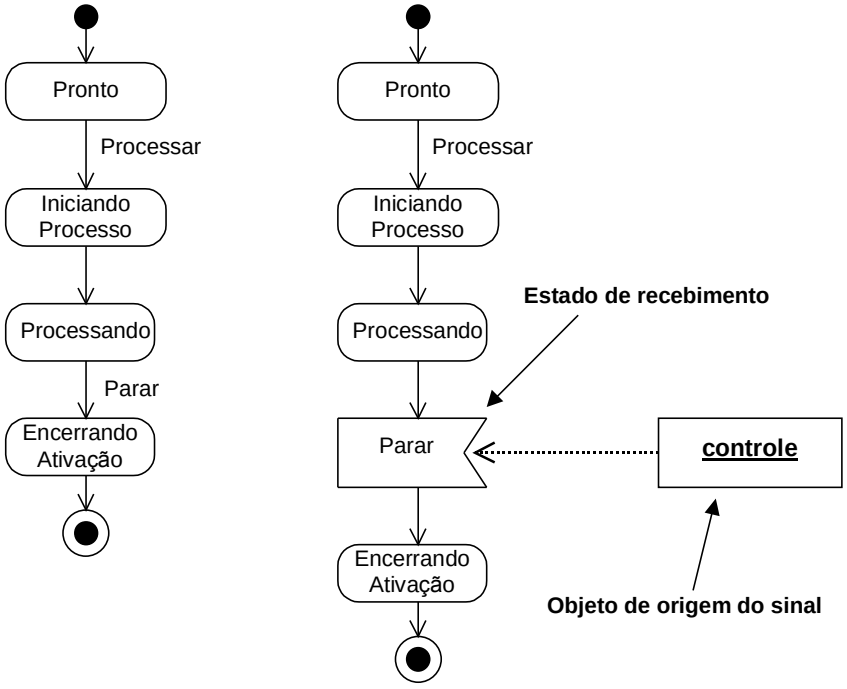

Figura V.18 – Exemplo da notação UML para estado de recebimento de sinal.

- **Envio de mensagens ou sinais**

De forma semelhante ao estado de recebimento de sinal, a UML oferece uma notação especial para representar situações de envio de mensagens ou sinais. Utiliza-se um pentágono convexo com o nome do sinal no interior para representar esta situação. Pode-se entender esta representação como um estado de envio de sinal. Opcionalmente, pode-se especificar o objeto de destino do sinal. A figura V.19 ilustra a notação UML para estado de envio de sinal.

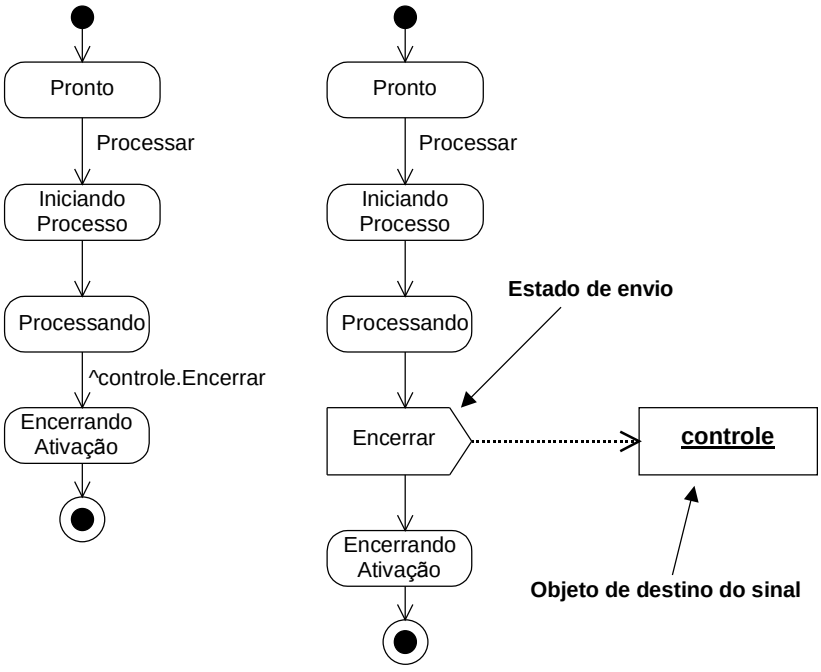

Figura V.19 – Exemplo da notação UML para estado de envio de sinal.

3 Conclusão

Neste capítulo apresentou-se a notação utilizada na UML para a modelagem da dinâmica interna de objetos. Essencialmente, utiliza-se a notação de Harel para diagramas de estados.

A principal utilização dos diagramas de estados é a modelagem individual do comportamento dos objetos de cada classe. O modelo construído permite estudar e especificar o comportamento dos objetos e serve como referência para sua codificação na fase de implantação.

Em muitos aspectos a notação por diagramas de estados se assemelha a notação procedural (como os fluxogramas e diagramas de ação). Os diagramas de estados, entretanto, são mais poderosos pois incluem notações específicas para comunicação entre objetos (como as cláusulas de envio), para concorrências e para níveis de abstração.