UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Iuri Sônego Cardoso

INSERINDO SUPORTE A DECLARAÇÃO DE ASSOCIAÇÕES DA UML 2 EM UMA LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Ciência da Computação
Orientador: Prof. Dr. Raul Sidnei Wazlawick

Florianópolis
2011

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

C268i Cardoso, Iuri Sônego
Inserindo suporte a declaração de associações da UML 2 em uma linguagem de programação orientada a objetos [dissertação] / Iuri Sônego Cardoso ; orientador, Raul Sidnei Wazlawick. – Florianópolis, SC, 2011.
361 p.: il., tabs.

Dissertação (mestrado) – Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da computação. 2. Linguagem de programação (Computadores). 3. UML (Computação). I. Wazlawick, Raul Sidnei. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 681

Iuri Sônego Cardoso

INSERINDO SUPORTE A DECLARAÇÃO DE ASSOCIAÇÕES DA UML 2 EM UMA LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Esta Dissertação foi julgada adequada para obtenção do Título de "Mestre" e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 29 de agosto de 2011.

_____
Prof. Dr. Mario Antonio Ribeiro Dantas
Coordenador do Curso

Banca Examinadora:

_____
Prof. Dr. Raul Sidnei Wazlawick,
Orientador
Universidade Federal de Santa Catarina

_____
Prof. Dr. Álvaro Freitas Moreira,
Universidade Federal do Rio Grande do Sul

_____
Prof. Dr. Olinto José Varela Furtado,
Universidade Federal de Santa Catarina

_____
Prof.ª Dr.ª Patrícia Vilain,
Universidade Federal de Santa Catarina

Aos meus pais José e Arilda, pelo apoio dado ao longo deste trabalho e de toda a minha vida.

AGRADECIMENTOS

A todos que colaboraram de forma direta e indireta para a realização deste trabalho.

A força originadora do universo, seja ela chamada de Deus, Inteligência Suprema, Cosmos, Acaso ou qualquer outro nome, pela minha existência e a existência de tudo que está a minha volta.

Ao Prof. Dr. Raul Sidnei Wazlawick pelo empenho, dedicação e incentivo com que me orientou desde a formulação da proposta de pesquisa até a conclusão deste trabalho. Pela confiança depositada em mim ao me aceitar como seu orientando e pela autoria da idéia de pesquisa que deu origem a este trabalho.

Ao Laboratório de Transportes e Logística (Labtrans/UFSC), que por intermédio do gerente Antônio Venicius dos Santos, me empregou profissionalmente e me deu apoio durante a maior parte do desenvolvimento deste trabalho.

Aos meus pais José dos Santos Cardoso e Arilda Maria Sônego Cardoso, familiares e amigos que colaboraram muito para a realização deste trabalho, através de muito incentivo e compreensão nos momentos difíceis.

“Uma das características distintivas do design de objetos é que nenhum objeto é uma ilha.”

(Kent Beck; Ward Cunningham, 1989)

RESUMO

Diagramas de classe UML são construídos com classes, atributos e associações. Porém, as linguagens de programação usualmente não implementam associações, as quais precisam ser representadas por meio de atributos e métodos, que devem ser construídos pelo programador. Isto gera um *gap* entre o modelo e o código do programa, dificultando a leitura, escrita e a manutenção de código. As soluções propostas em termos de linguagens de programação experimentais, bibliotecas e *code patterns,* apresentam ainda problemas para representar associações e/ou discrepâncias quanto a conceitos e funcionalidades. Este trabalho apresenta a proposta de uma extensão para uma linguagem de programação orientada a objetos que mantém os conceitos e funcionalidades existentes, acrescentando associações com o mesmo significado e expressividade da UML 2. Esta extensão aborda questões como multiplicidade, navegabilidade, visibilidade, *association end ownership* e especialização de associações.

Palavras-chave: Linguagem de Programação. Associações. UML.

ABSTRACT

UML diagrams are built with classes, attributes and associations. However, programming languages usually do not implement associations, which have to be represented by means of attributes and methods that have to be defined by programmers. That produces a gap between model and programming code, making reading, writing and maintaining code a hard task. Proposed solutions such as, experimental programming languages, libraries and code patterns, still present problems for representing associations, and/or inconsistencies regarding concepts and functionalities. This dissertation presents the proposal for an extension for an object-oriented programming language that is consistent with existing concepts and functionalities, adding association with the same meaning and expressiveness of UML 2. This extension approaches issues like multiplicity, navigability, visibility, association end ownership and association specialization.

Keywords: Programming Language. Associations. UML.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Linguagens de modelagem de software como a *Unified Modeling Language* (UML), utilizada no desenvolvimento de sistemas, possuem, entre outras, estruturas de classes e associações para representar entidades do mundo real.

Linguagens de programação orientadas a objetos, utilizadas para o desenvolvimento de sistemas, possuem somente estruturas de classes, necessitando que as estruturas de associações sejam construídas pelo programador. Este usualmente utiliza padrões de codificação que combinam atributos, métodos e classes para mapear associações entre classes.

O uso de padrões de codificação (ou *code patterns*) não oculta do programador, o código de construção das associações, tornando o código do programa mais complexo de ser escrito, lido e compreendido, quando comparado com linguagens experimentais que permitem a declaração de associações, como DSM[1], RelJ[2], entre outras. A complexidade de escrita pode ser resolvida por uma ferramenta de automatização da geração de código, diferentemente da complexidade de leitura e compreensão, em que é necessário agrupar atributos, métodos e classes para compor cada estrutura de associação. Além destas questões, os padrões de código existentes apresentam problemas para representar completamente a semântica de associações da UML 2, quanto a características como: multiplicidade, navegabilidade, visibilidade, *association end ownership* e especialização de associações.

O padrão de codificação proposto por Gessenharter (2009) para resolver estes problemas apresenta problemas quando usado com o *garbage collector*. Um destes problemas, o *problema da referência perdida* (apresentado na Subseção 3.2.2), diz respeito ao uso de *objetos de associação*[3] (*singletons*) para armazenar *links* de associação, permitindo que estes *links* e os objetos *linkados* permaneçam *vivos* (existentes em memória), porém inacessíveis após perderem suas últimas referências externas ao *objeto de associação*. Outro problema, o *problema da destruição de objetos obrigatoriamente associados* (apresentado na Subseção 3.2.3), surge da incerteza de que um objeto realmente é destruído (coletado pelo *garbage collector*) em um determinado ponto do programa.

---

[1] Shah *et al*, 1989.
[2] Bierman; Wren, 2005.
[3] Noble, 1997.

Por outro lado, associações passaram a ser incorporadas em linguagens orientadas a objetos experimentais no fim da década de 1980 através dos trabalhos de Rumbaugh (1987) e Shah *et al.* (1989). Estes trabalhos experimentais objetivaram a melhoria do entendimento do código escrito, bem como a simplificação da escrita do mesmo. Porém, os resultados destes trabalhos, não foram incorporados em linguagens de programação comerciais.

Ao analisar estas e outras propostas de linguagens orientadas a objetos com suporte a associações, como os trabalhos de Albano *et al* (1991), Balzer, Gross e Eugster (2007), Bierman e Wren (2005), e bibliotecas para abstração da construção de associações, como as presentes nos trabalhos de Østerbye (2007) e de Pearce e Noble (2006), constata-se que estas apresentam problemas para representar associações: ou não implementam verificação de limites mínimos de multiplicidade, ou não implementam visibilidade e navegabilidade. Em todos os casos em que há a implementação de verificação do limite mínimo de multiplicidade de forma invariante, há os problemas mencionados com o uso de *garbage collector* (conforme será discutido na Seção 3.2). O trabalho de Balzer, Gross e Eugster (2007) resolve o *problema da referência perdida* porque é possível consultar os *links* de associações através de propriedades dos objetos *linkados*. Porém, isto causa o *problema da visibilidade global* (apresentado na Subseção 3.2.1), impossibilitando encapsulamento por restrição de navegabilidade e visibilidade.

1.1 PROBLEMAS

Com base na avaliação destes trabalhos, enumeram-se os seguintes problemas:

a) Não há nessas abordagens uma forma de representar, em código de linguagem de programação, a semântica das associações da UML, considerando multiplicidade, navegabilidade, visibilidade, *association end ownership* e especialização de associações, sem causar problemas com o uso de *garbage collector* (apresentados na Seção 3.2);
b) Os *code patterns* existentes são as soluções que mais se aproximam da semântica da UML. Porém todos os detalhes da construção de associações no código do programa ficam explícitos, tornando-o complexo na escrita e na compreensão, e retirando o enfoque do programador sobre os problemas de domínio quando este precisa construir tais estruturas;

c) As linguagens de programação e bibliotecas existentes que ocultam a construção de associações não permitem representar características como limite inferior de multiplicidade superior a zero, ou visibilidade e navegabilidade.

Com a finalidade de resolver estes problemas, o presente trabalho apresenta nas próximas seções seus objetivos, limitações, justificativas e estruturação.

1.2 OBJETIVO GERAL

Apresentar como uma linguagem de programação comercial como C#, que possui *garbage collector*, pode ser estendida com a construção nativa de associações e classes de associações com a mesma semântica da UML 2, sem causar os problemas para o uso do *garbage collector* encontrados nos trabalhos correlatos, implementando multiplicidade, visibilidade, navegabilidade, *association end ownership* e especialização de associações. Em outras palavras, mostrar como definir e implementar tal extensão de linguagem sem as limitações observadas nas opções disponíveis na literatura.

1.3 OBJETIVOS ESPECÍFICOS

A extensão da linguagem, resultando na linguagem chamada de Association#, visa:

a) Permitir representar em código de linguagem de programação, a semântica das associações da UML, considerando multiplicidade, navegabilidade, visibilidade, *association end ownership* e especialização de associações, sem causar problemas ao uso do *garbage collector* (resolvendo o problema *a* mencionado na Seção 1.1);
b) Mostrar que com uma linguagem deste tipo é desnecessário o uso de padrões para codificação e reconhecimento de associações (engenharia reversa), resolvendo o problema *b* (mencionado na Seção 1.1);
c) Apresentar uma única abordagem que permite representar características como limite inferior de multiplicidade superior a zero, visibilidade e navegabilidade, resolvendo o problema *c* (mencionado na Seção 1.1);
d) Diminuir a discrepância semântica entre a linguagem de programação e a linguagem de modelagem.

1.4 LIMITAÇÕES

Este trabalho não abrange outras características das associações UML além das mencionadas anteriormente, tais como:

a) Propriedades das extremidades das associações, como ordenação, possibilidade de repetição de *tupla* (*bags*), agregação, composição, etc;
b) Associações qualificadas;
c) Associações n-árias;
d) Redefinição de associações;
e) Subconjunto de associações.

O presente trabalho também se limita a demonstrar como a linguagem orientada a objetos pode ser estendida para obter suporte a declaração de associações, sem detalhar formalmente a análise semântica para esta extensão e sem, ainda, haver implementado um compilador e/ou tradutor.

1.5 JUSTIFICATIVA

Uma linguagem de programação pode ser construída para o propósito de ter suporte a associações da UML, com limites de multiplicidade, *association end ownership*, classes de associação, visibilidade e navegabilidade. Esta abstrai do programador toda a construção destas estruturas, apresentando a ele uma estrutura simples e própria para a representação das mesmas. Retira-se do programador a responsabilidade e a necessidade de construir estas estruturas e portanto, elimina a necessidade do uso de padrões de codificação de associações, permitindo que o programador tenha um enfoque maior nos problemas de domínio.

Os problemas para uso do *garbage collector* com estruturas de associações podem ser resolvidos removendo a responsabilidade de armazenamento das referências para os objetos, do *singleton* que representa a associação (*objeto de associação*) e colocando-a nos próprios objetos participantes da associação, evitando assim o *problema da visibilidade global* (apresentado na Subseção 3.2.1), permitindo a implementação de navegabilidade e visibilidade, e evitando o *problema da referência perdida* (apresentado na Subseção 3.2.2). O *problema da destruição de objetos obrigatoriamente associados* (Subseção 3.2.3) pode ser resolvido por mecanismos que obriguem um objeto a estar consistente enquanto ele está *vivo*.

A extensão de uma linguagem de programação orientada a objetos que utiliza *garbage collector* permite que a nova linguagem herde toda a sintaxe e semântica preexistente, não necessitando definilas, bastando apenas acrescentar a sintaxe e semântica das novas estruturas.

1.6 ESTRUTURA

O presente trabalho inicia no Capítulo 2 com a revisão bibliográfica, em que são abordadas as definições da UML sobre associações, e padrões de construção de associações em linguagens orientadas a objetos.

O Capítulo 3 apresenta o estado da arte, com uma análise detalhada sobre os trabalhos correlatos, as contribuições e os problemas não resolvidos por estes.

A extensão de linguagem de programação que resolve os problemas elencados no estado da arte é apresentada no Capítulo 0 finalizando com as considerações finais deste trabalho no Capítulo 0.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta a revisão bibliográfica deste trabalho de pesquisa, que consiste nas definições da UML sobre associações (Seção 2.1), nos padrões para implementação de associações em linguagens orientadas a objetos (Seção 2.2) e, por último, na importância de associações como estrutura nativa nas linguagens de programação (Seção 2.3).

2.1 DEFINIÇÕES DA UML SOBRE ASSOCIAÇÕES

Esta seção apresenta as definições da UML sobre associações e suas características como multiplicidade, navegabilidade, visibilidade, *association end*, entre outras.

As próximas subseções apresentam um resumo destes conceitos e características, e quando necessário, apresenta críticas e observações sobre os mesmos.

2.1.1 Associação

Uma associação é um relacionamento entre dois ou mais *classifiers*[4] que descreve conexões entre suas instâncias. Associações são a "cola" que une um modelo de sistema. Sem associações, existe apenas um conjunto de classes isoladas (RUMBAUGH; JACOBSON; BOOCH, 2004).

Associações são conjuntos de tuplas. Os valores contidos em tuplas são instâncias das classes conectadas nas extremidades da associação. Estas tuplas, chamadas de *links* de associação, também podem ser entendidas como instâncias de associação. Uma mesma classe pode participar mais de uma vez em uma associação, isto é, uma associação pode associar uma classe a ela mesma. Neste caso, a tupla é composta por mais de um valor da mesma classe (OBJECT MANAGEMENT GROUP, 2010; RUMBAUGH; JACOBSON; BOOCH, 2004).

Associações podem ser binárias, ou n-árias. Uma *associação binária* é uma associação entre duas classes, cada uma em uma extremidade da associação, podendo ambas as extremidades conectarem a mesma classe. Uma *associação n-ária* contém três ou mais extremidades, cada qual conectando uma classe, podendo uma mesma classe estar conectada a mais de uma extremidade (OBJECT

[4] Object Management Group, 2010; Rumbaugh, Jacobson, Booch, 2004.

MANAGEMENT GROUP, 2010; RUMBAUGH; JACOBSON; BOOCH, 2004).

A Figura 1 apresenta um exemplo de diagrama de classe UML em que as classes *Device* e *DeviceController* são associadas pela associação binária *is_controlled_by*. Este exemplo de modelagem significa que dispositivos (instâncias de *Device*) são controlados por controladores de dispositivos (instâncias de *DeviceController*). Outras características desta associação, como nomes de papéis, limites de multiplicidade, entre outros, serão apresentados ainda neste capítulo.

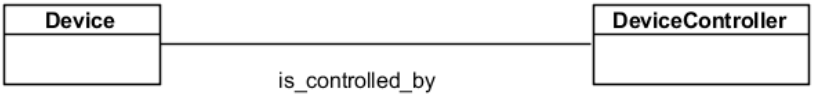

Figura 1. Exemplo de associação binária.

Cada extremidade de uma associação deve receber um nome que pode ser igual ou diferente do nome da classe conectada, devendo sempre ser um nome que identifique o papel da classe na associação. Este nome é comumente chamado de *nome de papel* (OBJECT MANAGEMENT GROUP, 2010; RUMBAUGH; JACOBSON; BOOCH, 2004).

A Figura 2 apresenta o mesmo exemplo da Figura 1 acrescido dos nomes de papéis nas extremidades da associação.

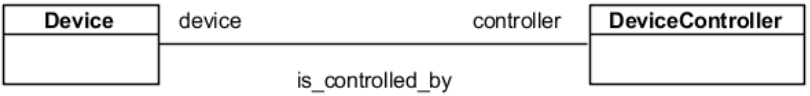

Figura 2. Associação binária com nomes de papéis definidos em cada extremidade da associação.

Extremidades de associação serão mencionadas neste trabalho também pelo seu termo equivalente em inglês: *association end*.

2.1.2 Classe de Associação

Classes de associação são elementos de modelagem que possuem características de associação e de classe. Como uma associação, uma classe de associação conecta duas ou mais classes participantes da associação, ou seja, instancia *links* contendo um valor para cada extremidade da associação. Como uma classe, suas instâncias são objetos que podem possuir atributos e/ou métodos. Estas características

de classe representam propriedades e operações das instâncias da associação, não das instâncias participantes desta. Sendo assim, instâncias de uma classe de associação são ao mesmo tempo *links* e objetos (OBJECT MANAGEMENT GROUP, 2010).

A Figura 3 apresenta o exemplo da Figura 2 acrescido da classe de associação *LastCommandSent*, entre as classes *Device* e *DeviceController*. Esta classe de associação instância o último comando enviado de um controlador de dispositivos (instância de *DeviceController*) para um dispositivo (instância de *Device*).

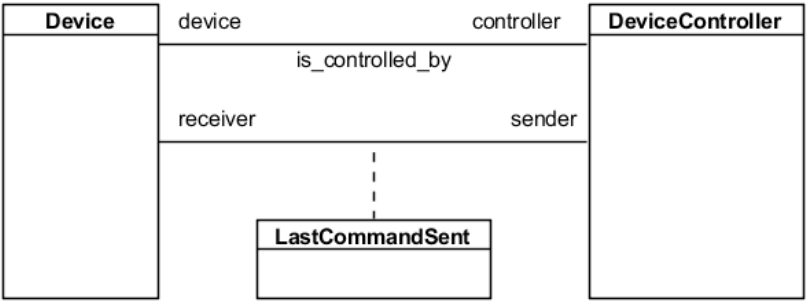

Figura 3.Exemplo da Figura 2 com acréscimo de uma classe de associação.

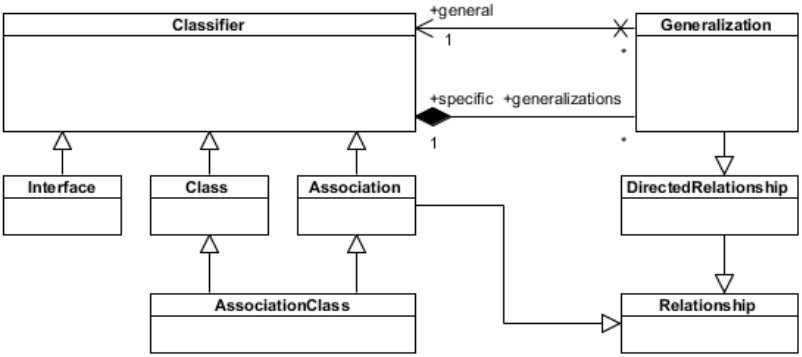

Figura 4. Modelagem parcial do pacote *Kernel* do meta-modelo presente na especificação da UML 2.
Fonte: Object Management Group (2010).

Uma classe de associação possui as semânticas estática e dinâmica, tanto de uma associação quanto de uma classe. Isto é garantido pelo meta-modelo apresentado pela Figura 4, em que

AssociationClass especializa (herda) *Class* e *Association* (OBJECT MANAGEMENT GROUP, 2010).

2.1.3 Link de Associação

Conforme apresentado na Subseção 2.1.1, *link* de associação é uma instância da associação, composta pelas instâncias das classes conectadas nas extremidades da associação. Entretanto, não está claro se *links* de associação são objetos tipados (similar a instâncias de classes) ou apenas uma representação abstrata para o entendimento do conceito de instâncias de associação.

Porém, ao referir-se a instâncias de classes de associação, a UML afirma:

> The ends of the association class that is owns cannot be used to navigate from instances of the association class to the objects on their ends. As association ends, they can be used for navigation between end objects, as in all associations (OBJECT MANAGEMENT GROUP, 2010, p. 47).

Conforme apresentado na Subseção 2.1.2, uma classe de associação possui as semânticas estática e dinâmica de uma associação e também de uma classe. Sendo assim, se deduz que se um *link* de associação pudesse utilizar os *association ends* da associação para navegar deste até as instâncias das classes associadas, obrigatoriamente, uma instância de classe de associação também poderia fazer o mesmo. Porém, conforme mencionado, as instâncias de uma classe de associação não o podem, inferindo então o mesmo para as instâncias (*links*) de uma associação.

Isto acarreta problemas com a suposição de utilizar *links* como objetos dentro de um sistema. Se os *links* não têm acesso aos objetos associados e, no caso de associações, não possuem atributos e métodos, qual a finalidade de utilizá-los como objetos distintos?

Sendo assim, este trabalho considera que o conceito de *link* de associação é uma representação abstrata utilizada apenas para entendimento de associações, não correspondendo estes a objetos discretos.

2.1.4 Association End

A UML não apresenta um conceito formal de *association end* além da modelagem deste como propriedade que pode pertencer tanto a

uma classe participante de uma associação, quanto à própria associação. Porém especifica: *"Each end represents participation of instances of the classifier connected to the end in links of the association."* Em diagramas, são como conectores responsáveis por conectar as instâncias das classes participantes (OBJECT MANAGEMENT GROUP, 2010, p. 38).

Segundo Rumbaugh, Jacobson e Booch (2004), um *association end* mantém referências para um *classifier* alvo. Estruturalmente, um *association end* é uma propriedade que armazena referências para instâncias do *Classifier* (classe, interface, outros) conectado por este.

Ainda segundo a especificação UML 2:

> An end property of an association that is owned by an end class or that is a navigable owned end of the association indicates that the association is navigable from the opposite ends; otherwise, the association is not navigable from the opposite ends" (OBJECT MANAGEMENT GROUP, 2010, p. 38).

Isto indica que:

a) *Association ends* são obrigatoriamente navegáveis quando são propriedades pertencentes a classes;
b) *Association ends* não são obrigatoriamente navegáveis quando pertencentes à associação porque há a necessidade de especificar se são navegáveis ou não (uso do adjetivo *navigable* em *owned end of the association*).

Outra especificação feita pela UML 2, diz: *"an association end owned by a class is also an attribute"* (OBJECT MANAGEMENT GROUP, 2010, p. 45).

Sendo assim, pode-se afirmar que existem três tipos de *association end*:

a) Pertencente à classe participante – o *association end* é uma propriedade da classe participante que a contém, navegável e com possível restrição de visibilidade para as demais classes da mesma forma que um atributo;
b) Pertencente a associação – o association end é uma propriedade da associação, é navegável e não há possibilidade de restrição de visibilidade;
c) Não-navegável – o *association end* é uma propriedade da associação e não é navegável.

Estas definições fazem com que a posse do *association end* e a navegabilidade sejam conceitos parcialmente independentes.

Association ends pertencentes a classes são demarcados em diagramas de classe UML como um ponto opaco sobre a linha da associação no exato ponto onde esta linha toca no retângulo do *classifier* associado (OBJECT MANAGEMENT GROUP, 2010).

A Figura 5 apresenta o exemplo da Figura 3 acrescido da definição de posse dos *association ends*. Neste, *controller* e *receiver* (demarcados pelos pontos opacos sobre a linha da associação) são *association ends* pertencentes a classes, enquanto *device* e *sender* são *association ends* pertencentes a associações. A visibilidade para os *association ends* pertencentes a classes ainda não foram definidas neste exemplo.

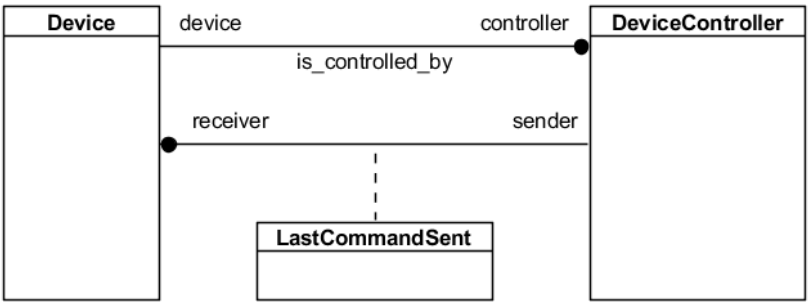

Figura 5. Exemplo da Figura 3 com acréscimo de posse de *association end* e visibilidade

Em associações entre três ou mais *Classifiers*, todos os *association ends* pertencem às associações (OBJECT MANAGEMENT GROUP, 2010).

2.1.5 Navegabilidade dos Association Ends

Segundo Object Management Group (2010) a navegabilidade de uma extremidade de associação é uma característica que determina se as instâncias das demais classes podem ou não acessar instâncias da classe conectada. Esta característica pode variar semanticamente conforme a implementação. Em uma associação, uma extremidade não navegável pode ou não ser acessível pelas demais extremidades. Caso seja, este acesso pode não ser eficiente.

Possivelmente esta possibilidade de navegação em *association ends* não navegáveis seja devido às limitações das ferramentas, conforme afirmado pela Object Management Group (2010, p. 40):

"Note that tools operating on UML models are not prevented from navigating associations from non-navigable ends".

Porém, enfocando a modelagem, desconsiderando questões de implementação como eficiência, este trabalho considerará que extremidades não navegáveis não possibilitam navegação, podendo serem utilizadas para fins similares aos da restrição de visibilidade: encapsulamento.

A Figura 5 apresentou a associação *is_controlled_by* e a classe de associação *LastCommandSent* sem definição de navegabilidade. De acordo com Object Management Group (2010): diagramas em que as associações não possuem indicações de navegabilidade podem também representar associações navegáveis em todos os sentidos.

Os diagramas de classe presentes neste trabalho sempre consideram associações binárias sem definição de navegabilidade como associações navegáveis em ambos os sentidos. Quando um dos *association ends* não for navegável, ambos estarão definidos como navegável e não navegável, conforme o exemplo descrito a seguir.

A Figura 6 apresenta o exemplo da Figura 5 com acréscimo da definição de navegabilidade das associações. Nesta, os *association ends* não navegáveis não possuem mais nome e são marcados com um *X*, enquanto os *association ends* navegáveis (*controller* e *sender*) são marcados com uma seta. Observa-se que o association end que possuía o nome de *receiver* deixou de ser um association end pertencente a classe para ser pertencente a associação. Isto porque association ends não navegáveis sempre são propriedades da associação (conforme Subseção 2.1.4).

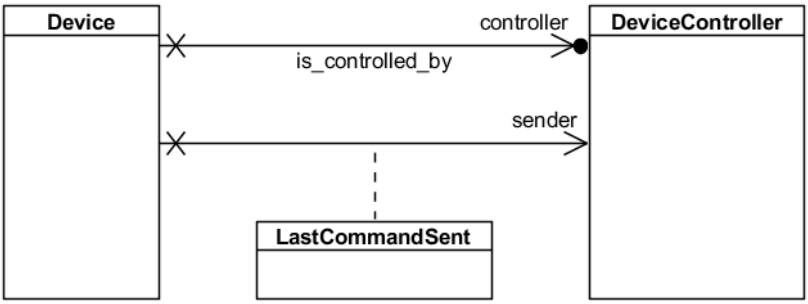

Figura 6. Exemplo da Figura 5 com acréscimo da navegabilidade das associações.

2.1.6 Multiplicidade dos Association Ends

Segundo Object Management Group (2010), a multiplicidade é uma definição de um intervalo inclusivo de inteiros iniciando com um limite inferior e terminando com um (possivelmente infinito) limite superior. A multiplicidade em um conjunto de elemento indica a cardinalidade admissível para instanciação deste elemento.

A multiplicidade em *um association end* determina a quantidade possível de instâncias (da classe conectada àquela extremidade) associadas a cada combinação de instâncias das demais classes conectadas nas demais extremidades. Sendo assim, em associações binárias, a multiplicidade em um *association end* determina a quantidade possível de instâncias da mesma extremidade associadas a cada instância da extremidade oposta (OBJECT MANAGEMENT GROUP, 2010).

A Figura 7 apresenta o exemplo da Figura 6 com acréscimo dos limites de multiplicidade nos *association ends*. Limites de multiplicidade definidos com * (asterisco) representam limite mínimo de zero, sem limite máximo. Limites de multiplicidade *0..1* representam limite mínimo de zero e limite máximo de um. O *association end* de nome *controller* foi renomeado para o seu plural (*controllers*) devido a multiplicidade do mesmo permitir a participação de mais de uma instância.

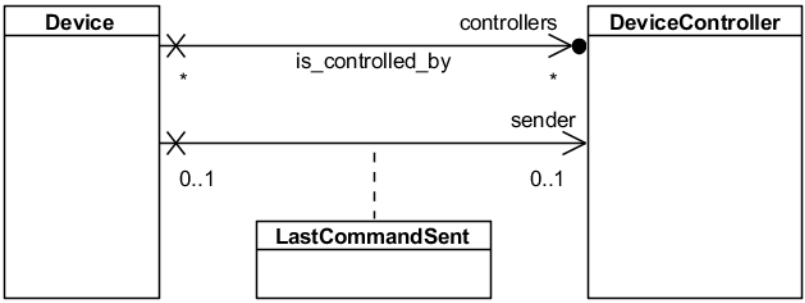

Figura 7. Exemplo da Figura 6 com acréscimo dos limites de multiplicidade.

2.1.7 Visibilidade dos Association Ends

Existem quatro tipos possíveis de visibilidade: *public*, *private*, *protected* e p*ackage* (OBJECT MANAGEMENT GROUP, 2010, p. 142).

Considerando que *namespace* é qualquer classe ou pacote, elementos públicos (*public*) são acessíveis a todos os demais elementos que possuem acesso ao *namespace* que o contém. Elementos privados (*private*) são acessíveis somente pelos elementos contidos no mesmo *namespace*. Elementos protegidos (*protected*) são visíveis somente pelos elementos contidos em um *namespace* o qual é especialização do primeiro. Elementos com visibilidade de pacote (*package*) são elementos visíveis a qualquer elemento dentro do mesmo pacote, independente do *namespace* (OBJECT MANAGEMENT GROUP, 2010).

Quando um *association end* é propriedade de uma classe (no caso, um atributo), este pode ser definido com um tipo de visibilidade. Quando o mesmo é propriedade de uma associação, não há especificação explicita se é possível restringir a sua visibilidade (OBJECT MANAGEMENT GROUP, 2010).

A Figura 8 apresenta o exemplo da Figura 7 com acréscimo da definição de visibilidade do *association end* de nome *controllers* (o único *association end* pertencente a classe). O sinal negativo ("-") que precede o nome de papel indica que a visibilidade é *private*.

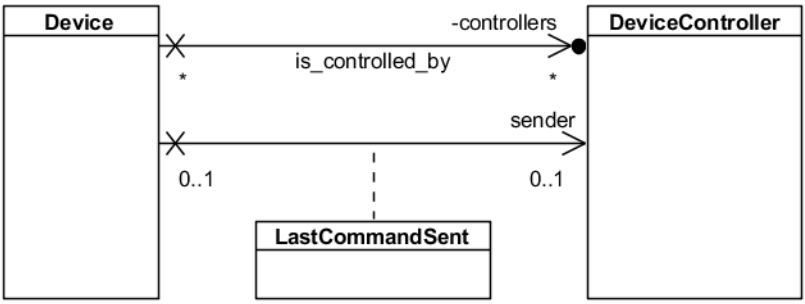

Figura 8. Exemplo da Figura 7 com acréscimo da definição de visibilidade.

2.1.8 Especialização de Associações e Classes de Associação

A especialização/generalização de associações e classes de associação não está claramente definida de forma explícita na UML 2. Porém, sua semântica está implícita na estruturação do meta-modelo, presente na especificação da Object Management Group (2010).

Conforme ilustrado pela Figura 4, as meta-classes *Association* e *Class* especializam *Classifier*. Sendo assim, a semântica de generalização de associações e classes, representada pelas associações

desta com a meta-classe *Generalization*, é a mesma, diferenciando-se apenas por *constraints* definidas em ambos.

A meta-classe *Generalization* faz a ligação entre o *Classifier* especializado (nome de papel *specific*) e o *Classifier* generalizado (nome de papel *general*).

A semântica de generalização entre duas instâncias de *Classifier* é descrita da seguinte forma:

> Where a generalization relates a specific classifier to a general classifier, each instance of the specific classifier is also an instance of the general classifier. Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier (OBJECT MANAGEMENT GROUP, 2010, p. 73).

Assim como na generalização de classes, a generalização de associações especifica que instâncias (*links*) das associações especialistas são também instâncias das associações generalistas. Isto significa que associações especialistas também podem ser vistas como subconjuntos de *links* de suas associações generalistas, apesar de que os *links* da associação especialista podem possuir características mais específicas do que os *links* da associação generalista (RUMBAUGH; JACOBSON; BOOCH, 2004).

Analisando associações como conjuntos, a instanciação de um *link* de associação pode ser considerada como a inserção do mesmo como elemento do conjunto (a associação) e dos superconjunto (a associação generalista da primeira). A destruição de um *link*, de forma análoga, é a remoção do mesmo como elemento do conjunto e dos superconjuntos.

A especialização de associações possui algumas particularidades (OBJECT MANAGEMENT GROUP, 2010):

a) A associação especialista possui o mesmo número de *association ends* da associação generalista;
b) Cada *association end* da associação especialista conecta um tipo (*Classifier*) igual ou especialista do tipo conectado pelo respectivo *association end* da associação generalista.

Pelas particularidades mencionadas e pela semântica de generalização, infere-se que os *association ends* de uma associação especialista podem ser vistos como subconjuntos dos respectivos *association ends* da associação generalista, embora os elementos do

subconjunto possam conter características mais específicas que os elementos do superconjunto. A inclusão automática de um *link*, referente a uma associação especialista, no superconjunto de *links* da associação generalista obriga que a inclusão dos elementos participante em cada *association end* da associação especialista também inclua automaticamente os mesmos elementos nos *association ends* correspondente da associação generalista.

A especialização de classes de associação segue as mesmas regras da especialização de associações e da especialização de classes. Isto é expresso no meta-modelo pela generalização da meta-classe *AssociationClass* pelas meta-classes *Association* e *Class*.

A Figura 9 apresenta um exemplo de generalização/especialização de associações. Neste, todos os *association ends* são propriedades das associações e ambas as associações são navegáveis nos dois sentidos. A associação *is_activate_by* especializa a associação *is_controlled_by* por intermédio do relacionamento de generalização. Cada *link* adicionado/removido na associação *is_activate_by* é também adicionado/removido na associação *is_controlled_by*. Porém, o inverso nem sempre é verdade.

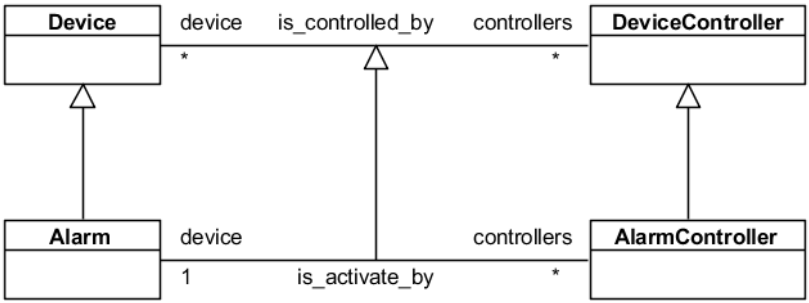

Figura 9. Exemplo de generalização/especialização de associações.

Cada instância de *AlarmController* em que sua referência é adicionada/removida pelo *association end* de nome *controllers*, possuído pela associação *is_activate_by*, é obrigatoriamente adicionada/removida no *association end* de mesmo nome possuído pela associação *is_controlled_by*. O inverso nem sempre é verdade. O mesmo ocorre na outra extremidade da associação, com os *association ends* de nome *device*. Cada instância de *Alarm* adicionada/removida no *association end* de nome *device* pertencente à associação is_*activate_by*,

é adicionada/removida automaticamente no *association end* de mesmo nome da associação *is_controlled_by*.

2.1.9 Interfaces

Segundo o Object Management Group (2010, p. 88), uma interface é um tipo de *classifier* que representa a declaração de um conjunto de características públicas coerentes e obrigações. Uma interface especifica um contrato, sendo que qualquer instância de um *classifier* que realiza a interface deve cumprir esse contrato.

Interfaces são representadas no meta-modelo pela meta-classe *Interface*. Esta, de acordo com a Figura 4, é especialista da meta-classe *Classifier*, assim como *Class* e *Association*. Sendo assim, uma interface pode ter, entre outras coisas, *características* (atributos e operações). Porém, uma restrição obriga que todas as características tenham visibilidade pública (*public*) (OBJECT MANAGEMENT GROUP, 2010).

Interfaces, diferente de classes, não geram instâncias. Para que uma instância implemente uma interface, é necessário que a classe que gerou a instância *realize* a interface por meio do relacionamento de *realização*. Este relacionamento obriga que a classe cumpra com todas as definições da interface, isto é, obriga que a classe implemente todas as características declaradas na interface (OBJECT MANAGEMENT GROUP, 2010).

A participação de uma interface em uma associação acontece da mesma maneira que a participação de uma classe. O *association end* que conecta a interface na associação é do tipo da própria interface, significando que qualquer instância que implemente a interface pode ser referenciada. Quando um *association end* é uma propriedade de uma interface, sua visibilidade é sempre pública devido à restrição mencionada anteriormente (OBJECT MANAGEMENT GROUP, 2010).

A Figura 10 apresenta um exemplo similar ao apresentado pela Figura 9 em que as classes *Device* e *DeviceController* foram substituídas por interfaces de mesmo nome. Sendo assim, os relacionamentos de generalização que existiam entre as classes Alarm e Device, e entre as classes AlarmController e DeviceController, foram substituídos por relacionamentos de realização. Isto é, a classe *Alarm* realiza a interface *Device* e a classe *AlarmController* realiza a interface *DeviceController*.

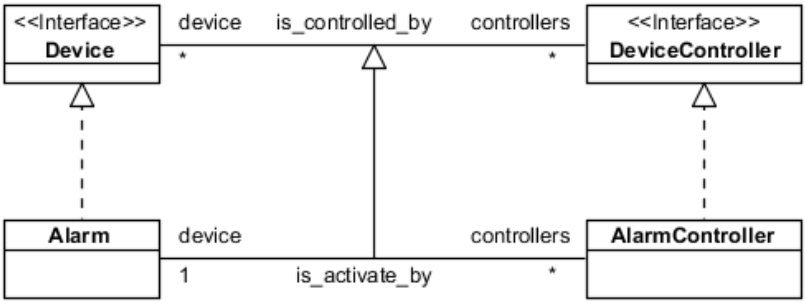

Figura 10. Exemplo de associação entre interfaces com generalização de associação.

2.2 ASSOCIAÇÕES EM PROGRAMAÇÃO ORIENTADA A OBJETOS

Linguagens de programação orientadas a objetos como Java, C#, C++, Object Pascal e outras, não possuem suporte nativo a construção de associações. Os programadores destas linguagens, frequentemente, utilizam padrões de escrita de código para construir associações, combinando atributos, métodos e classes (NOBLE, 1997).

Esta seção apresenta alguns padrões básicos para implementação de associações em linguagens de programação sem suporte nativo a este tipo de estrutura.

2.2.1 Padrões para Codificação de Associações

Noble (1997) reuniu cinco padrões básicos para construção de associações, comumente utilizados pelos programadores, independente da linguagem orientada a objetos utilizada: *associação como atributo* (*relationship as attribute*), *objeto de associação* (*relationship object*), *objeto de coleção* (*collection object*), *active value* e *mutual friends*. Estes apenas permitem a comunicação entre as instâncias das classes participantes, não abordando outras características como visibilidade e *association end ownership*. A multiplicidade (limite superior) e a navegabilidade das associações são características inerentes de cada padrão.

Os padrões *associação como atributo* e *active value* não possuem relevância para este trabalho por não serem aplicados a nenhum dos trabalhos correlatos (apresentados no Capítulo 3), nem ao novo trabalho desenvolvido (apresentado no Capítulo 0). Os demais padrões são apresentados detalhadamente nas próximas subseções.

Para cada padrão, serão apresentados dois diagramas de classe UML e um exemplo de implementação na linguagem Java. Dos dois diagramas, o primeiro será o de alto nível e o segundo de baixo nível. O diagrama de alto nível é a representação UML da associação desconsiderando os recursos da linguagem de programação utilizada na implementação desta. O diagrama em baixo nível é a representação UML para a implementação da associação em uma linguagem orientada a objetos.

2.2.1.1 Objeto de Associação

Associações complexas com qualquer grau de multiplicidade (*um-para-um*, *um-para-muitos* ou *muitos-para-muitos*), navegáveis em ambos os sentidos, e com importância conceitual equivalente às classes em um sistema, podem ser modeladas utilizando o padrão de codificação *objeto de associação*. Este consiste em construir a associação em um objeto externo às classes participantes da associação, que deve conter todo o código da associação, responsabilizando-o pela gerência e manutenção desta. Este objeto possui estruturas internas (atributos, métodos, classes, etc) que referenciam os objetos das classes participantes, associando-os (NOBLE, 1997).

A Figura 11 apresenta dois diagramas de classe UML que representam uma associação construída com o padrão *objeto de associação*. O primeiro (*a*) é o diagrama que representa a referida associação em alto nível. O segundo (*b*) é o diagrama em baixo nível que representa a implementação do padrão mencionado. Observa-se que os limites de multiplicidade em ambos os *association ends* da associação presente no diagrama *a* podem ter quaisquer valores para os limites mínimo e máximo, sendo então representados por sinais de interrogação.

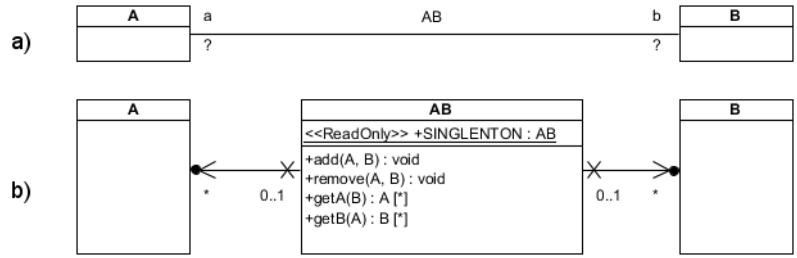

Figura 11. Diagramas de classe para o padrão objeto de associação em alto nível (a) e baixo nível (b).

A classe *AB* instancia um *singleton* que tem capacidade de adicionar e remover tuplas <*A,B*>, assim como retornar os objetos associados a objetos específicos das classes *A* ou *B*. A Figura 12 apresenta a implementação em código Java do padrão *objeto de associação*, de acordo com o modelado pelo diagrama *b* da Figura 11. As estruturas internas foram suprimidas por estas poderem variar de acordo com a necessidade e com a disponibilidade de recursos específicos de cada linguagem de programação.

Código Java

```
1  class A { }
2
3  class B { }
4
5  class AB {
6    public static final AB SINGLETON = new AB();
7    ...
8    public void add(A a, B b) {...}
9    public void remove(A a, B b) {...}
10   public Collection<A> getA(B b) {...}
11   public Collection<B> getB(A a) {...}
12 }
```

Figura 12. Exemplo de implementação em Java do padrão *objeto de associação*.

As vantagens deste padrão em relação aos demais são o desacoplamento do código das associações das classes participantes e aumento da coesão. As classes participantes tornam-se independentes umas das outras, diminuindo o acoplamento. Este padrão tem como desvantagens, maiores custos de implementação, de execução de operações (adição, remoção e consulta) sobre os links de associação, e de memória alocada (NOBLE, 1997).

2.2.1.2 Objeto de Coleção

O padrão *objeto de coleção* é utilizado para implementar associações simples *um-para-muitos*, navegável na direção do único objeto para os muitos objetos associados. Este consiste em criar um atributo na classe em que a multiplicidade de papel é *um*. O atributo refere-se a uma coleção de todos os objetos da classe oposta (em que a multiplicidade de papel é *muitos*) *linkados* ao objeto proprietário da coleção (NOBLE, 1997).

A Figura 13 apresenta os diagramas de classe UML em alto (*a*) e baixo (*b*) níveis para o padrão *objeto de coleção*.

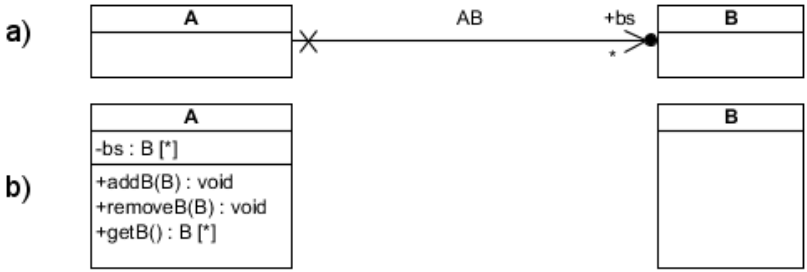

Figura 13. Diagramas UML representando o *pattern objeto de coleção* em alto e baixo níveis.

A Figura 14 apresenta a implementação em código Java da associação entre as classes *A* e *B*, utilizando o padrão *objeto de coleção* modelado pelo diagrama *b* da Figura 13. O atributo *b* (linha 2) da classe *A* contém o objeto de coleção responsável por armazenar as referências para todos os objetos da classe *B* associados à instância de *A*. Os métodos (linhas 4 a 8) presentes na classe *A* são para prover acesso externo ao atributo *b*.

Código Java

```
1  class A {
2    private final Collection<B> bs = new HashSet<B>();
3
4    public void addB(B b) { this.bs.add(b); }
5    public void removeB(B b) { this.bs.remove(b); }
6    public Collection<B> getB() {
7      return Collections.unmodifiableCollection(this.bs);
8  } }
9
10 class B { }
```

Figura 14. Exemplo de implementação em Java do padrão *objeto de coleção*.

As vantagens do padrão *objeto de coleção* em relação ao *objeto de associação* é que o primeiro tem menor custo de implementação e é mais eficiente (o *objeto de coleção* armazena somente os objetos *linkados* ao objeto proprietário deste). A desvantagem é o acoplamento da classe que participa com muitos objetos (no exemplo, a classe *B*) na classe que participa somente com um (no exemplo, a classe *A*) (NOBLE, 1997).

Apesar de ser utilizada para modelar associações *um-para-muitos*, este padrão não garante esta cardinalidade. Se um mesmo objeto da extremidade da associação que participa com *muitos* for referenciado

por mais de um objeto da extremidade que participa com *um*, será então uma associação *muitos-para-muitos*.

2.2.1.3 Mutual Friends

O padrão *mutual friends* é utilizado para implementação de associações simples navegáveis em ambos os sentidos, podendo ter multiplicidade *um-para-um*, *um-para-muitos* ou *muitos-para-muitos* (NOBLE, 1997).

Este padrão especifica que a associação navegável em ambos os sentidos deve ser decomposta em duas associações, cada uma navegável em um dos sentidos. A implementação destas duas associações é feita utilizando-se *associação como atributo* – quando o lado oposto participa no máximo com uma referência – ou *objeto de coleção* – quando o lado oposto participa com mais de uma (ou muitas) referência. Após a implementação das duas associações, implementa-se um mecanismo de sincronização de ambas, tornando-a uma única associação navegável em ambos os sentidos (NOBLE, 1997).

Esta sincronização depende de que um dos participantes seja o *líder* e o outro, o *seguidor*. O *líder* será o responsável por modificar o conjunto de *links* de associação em ambos os lados e o *seguidor* delega ao *líder* esta modificação. Para que o *líder* possa modificar as referências do *seguidor*, é necessário que o *seguidor* disponha ao *líder*, métodos de acesso a suas referências de forma que somente o *líder* tenha acesso. Estes métodos possuem restrições de visibilidade disponíveis nas linguagens de programação (visibilidade de pacote em Java, por exemplo) (NOBLE, 1997).

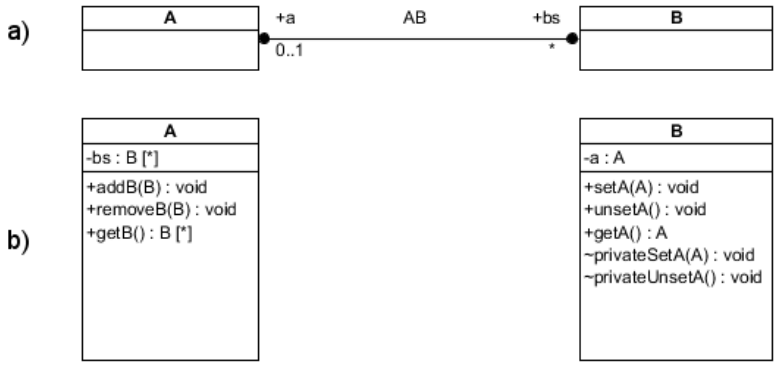

Figura 15. Diagramas UML representando o *pattern mutual friends* em alto e baixo níveis.

A Figura 15 apresenta os diagramas de classe UML em alto nível (*a*) e baixo nível (*b*) para o padrão *mutual friends*.

Código Java

```
1  class A {
2    private final Collection<B> bs = new HashSet<B>();
3    public void addB(B b) throws Exception {
4      b.privateSetA(this);
5      this.bs.add(b);
6    }
7    public void removeB(B b) {
8      b.privateUnsetA();
9      this.bs.remove(b);
10   }
11   public Collection<B> getB() {
12     return Collections.unmodifiableCollection(this.bs);
13 } }
14
15 class B {
16   private A a;
17   public void setA(A a) throws Exception {a.addB(this); }
18   public void unsetA() { a.removeB(this); }
19   public A getA() { return this.a; }
20
21   void privateSetA(A a) throws Exception {
22     if (this.a == null) this.a = a;
23     else if (this.a != a) throw new Exception();
24   }
25   void privateUnsetA() { this.a = null; }
26 }
```

Figura 16. Implementação em Java de uma associação utilizando o padrão *mutual friends*.

A Figura 16 apresenta a implementação em Java de uma associação *um-para-muitos*, navegável em ambos os sentidos, entre as classes *A* e *B*, utilizando o padrão *mutual friends*, modelado pelo diagrama *b* da Figura 15. A classe *A* implementa sua associação com a classe *B* utilizando o padrão *objeto de coleção* (linha 2). A classe *B* utiliza o padrão *associação como atributo* (linha 16) para implementar sua associação com *A*. A classe *A* tem o papel de *líder* e a classe *B*, o papel de *seguidor*. Os métodos de modificação da associação presentes em *A* modificam as referências em ambos os lados da associação. A modificação das referências de um objeto da classe *B* é feita por *A* na chamada dos métodos *privateSetA* e *privateUnsetA* (linhas 4 e 8), restritos ao uso exclusivo da classe *A*. Os métodos públicos de

modificação da associação presentes em *B* (linhas 17 e 18) delegam suas chamadas aos métodos de *A*.

A vantagem do uso de *mutual friends* em relação ao uso de objeto de associação é que o primeiro padrão é mais eficiente. As desvantagens são o acoplamento de ambas as classes envolvidas, cada uma com a outra, e a baixa coesão das classes que necessitam conter o código de gerenciamento e manutenção da associação (NOBLE, 1997).

2.3 IMPORTÂNCIA DAS ASSOCIAÇÕES COMO ESTRUTURA NATIVA

Apesar de ser possível construir associações, utilizando atributos, métodos e classes, a necessidade de construção possui desvantagens significantes quando comparada com a possibilidade de uso de uma estrutura nativa e exclusiva para o mesmo propósito.

Segundo Rumbaugh (1987): relações (associações) são particularmente úteis na modelagem de sistemas grandes que contém muitas classes que interagem entre si porque relações abstraem interações entre classes de forma natural.

Apesar de que associações possam ser construídas pelo programador, este é forçado a especificar detalhes da implementação que são irrelevantes para a lógica da aplicação, podendo cometer erros de implementação que podem gerar inconsistência. Propriedades de uma associação, como a multiplicidade, em vez de serem declaradas, devem ser codificadas manualmente. Mesmo que o código possa ser gerado diretamente dos modelos, por meio de ferramentas, a abstração das associações será perdida na decomposição destas em estruturas utilizadas para a implementação (RUMBAUGH, 1987; BALZER; GROSS; EUGSTER, 2007; BIERMAN; WREN, 2005).

É possível construir classes que implementem, de forma genérica, as operações necessárias para uma associação, sendo estas utilizadas para construir todas as associações de um sistema através do mecanismo de especialização de classes (*herança*). Isto oculta do programador os detalhes da implementação, porém, ainda não eleva os relacionamentos de associação ao mesmo nível semântico dos relacionamentos de generalização/especialização (RUMBAUGH, 1987).

Relacionamentos de generalização, na ausência de uma estrutura específica, também poderiam ser construídos utilizando atributos e métodos. Entretanto, estruturas nativas de generalização estão presentes na maioria das linguagens orientadas a objetos sendo que a experiência demonstra que associações são estruturas mais importantes e mais

comuns que generalizações, na modelagem de sistemas grandes (RUMBAUGH, 1987).

Classes, assim como associações e generalização de classes, são estruturações que também podem ser construídas a partir de estruturas de mais baixo nível. Por exemplo: DSM – uma linguagem orientada a objetos – foi construída em C – uma linguagem procedural (SHAH *et al*, 1989).

Expressar entidades do mundo real como objetos é mais intuitivo do que representá-los por conjuntos de dados que são manipulados por *procedures* e *functions*. De forma análoga, expressar interações entre objetos com associações é mais intuitivo do que expressar estas com atributos, métodos e classes.

3 ESTADO DA ARTE

O presente capítulo, dividido em três seções, inicia na Seção 3.1 com uma análise dos trabalhos correlatos quanto às características presentes e as limitações dos mesmos com respeito à semântica de associações da UML e/ou a presença de outros problemas. Na sequência, a Seção 3.2 apresenta um apontamento dos problemas não solucionados por estes trabalhos, finalizando na Seção 3.3 com uma análise comparativa destes trabalhos quanto as características e limitações.

As duas últimas seções merecem destaque por apresentarem o estado da arte, servindo de base para o entendimento das contribuições da extensão de linguagem proposta por este trabalho de pesquisa (apresentada no Capítulo 0).

3.1 TRABALHOS CORRELATOS

Entre os trabalhos analisados, encontram-se *code patterns* para mapeamento da semântica de associações UML, bibliotecas para abstração da construção de associações e linguagens de programação orientadas a objetos com suporte a associações como construção nativa.

Em alguns destes trabalhos, o termo utilizado para designar o conceito de colaboração entre objetos é *relacionamento* (*relationship*) em vez do utilizado pela UML: *associação* (*association*). Com a finalidade de uniformização, este trabalho sempre se referirá a este conceito como *associação*.

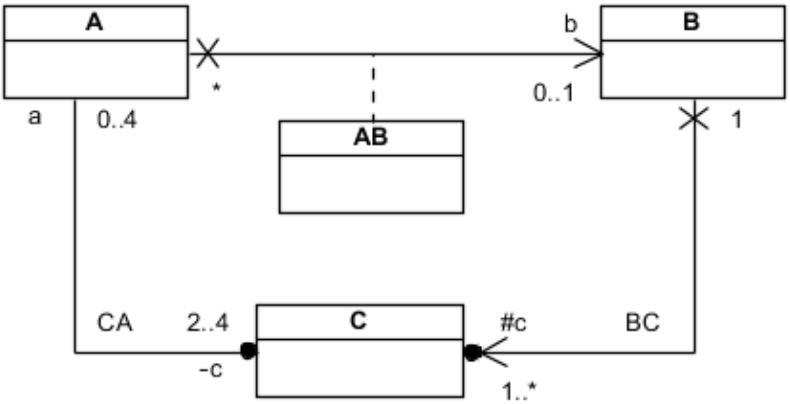

Figura 17. Exemplo de associações entre classes.

Os trabalhos foram analisados quanto às características necessárias para implementação semântica de associações da UML. Estas dizem respeito à capacidade de declaração ou representação de: multiplicidade, navegabilidade, visibilidade, *association end ownership*, classes de associação, especialização de associações e nomes de papel. Com a exceção de especialização de associações, todas as demais características estão presentes no modelo ilustrado pela Figura 17, utilizado para posterior exemplificação de implementação em cada trabalho analisado.

Outra característica analisada diz respeito à capacidade de abstrair a construção de associações sem necessidade de o programador especificar a forma e os detalhes de implementação. Sua relevância foi apresentada na Seção 2.3.

3.1.1 Code Patterns para Construção de Associações

Os *code patterns* aqui apresentados caracterizam-se por implementar associações em código puro de uma linguagem de programação, sem uso de bibliotecas.

3.1.1.1 Code Pattern de Génova et al

Génova, Del Castillo e Llorens (2003) apresentam um *code pattern*, escrito em código Java puro, que tem a finalidade de mapear a semântica de associações da UML. Este estende o *pattern mutual friends* e aborda a verificação dos limites inferior e superior de multiplicidade e características como visibilidade e navegabilidade.

Não há no artigo de Génova, Del Castillo e Llorens (2003) um exemplo de código escrito. Estes se encontraram no trabalho de Del Castillo (2002) o qual apresenta detalhes da ferramenta de geração de código JUMLA. Neste, os *association ends*, referenciados como nomes de papel, são objetos que contêm as estruturas necessárias para acessar e modificar a associação.

A Figura 18 apresenta a implementação mais aproximada possível do modelo da Figura 17 utilizando este *pattern*. Nesta estão representadas as classes *A* (linhas 1 a 18), *B* (linhas 31 a 35) e *C* (linhas 37 a 40); o *association end b* (linha 2) da classe de associação *AB*, o *association end c* (linha 32) da associação *BC* e os *association ends c* (linha 3) e *a* (linha 38) da associação *CA*. Os association ends são instanciados por classes *nested* (*inner*) *AssociationAB* (linhas 5 a 18), *AssociationBC* (linha 34), *AssociationCA* (são duas, uma está presente

nas linhas 20 a 29 e a outra na linha 39) presentes dentro das classes participantes.

Código Java

```
1  class A {
2    public final AssociationAB bRole = new AssociationAB();
3    private final AssociationCA cRole= new AssociationCA();
4
5    public class AssociationAB  {
6      public boolean isBidirectional() { ... }
7      public boolean isMandatory { ... }
8      public boolean isMultiple() { ... }
9      public long getMIN() { ... }
10     public long getMAX() { ... }
11     public boolean isValid() { ... }
12     public long numberOfLinks() { ... }
13     public boolean test(B query_link) { ... }
14     public B get() throws Exception { ... }
15     public boolean add(B new_link) { ... }
16     public int remove() { ... }
17     public int remove(B old_link) { ... }
18   }
19
20   private class AssociationCA {
21     ...
22     private boolean test(C query_link) { ... }
23     private boolean test(Collection query_links) { ... }
24     private Collection get() throws Exception { ... }
25     private boolean add(C new_link) { ... }
26     private int remove() { ... }
27     private int remove(C old_link) { ... }
28     private int remove(Collection old_links) { ... }
29 } }
30
31 class B {
32   protected final AssociationBC cRole =
33     new AssociationBC();
34   protected class AssociationBC  {...}
35 }
36
37 class C {
38   public final AssociationCA aRole = new AssociationCA();
39   public class AssociationCA  { ... }
40 }
```

Figura 18. Exemplo de código escrito no *pattern* utilizado pela ferramenta.

3.1.1.1.1 Limitações em Relação à Semântica da UML

Este *pattern*, quanto à representação do modelo, não aborda ou é limitado quanto a:

a) Implementação de classes de associação (*AB*, na Figura 17);

b) Implementação de especialização de associações;
c) Opção por *association ends* que sejam propriedades da associação (*association end* navegável *c* na associação *CA* e os *association ends* não navegáveis em *AB* e *BC*);
d) Declaração de multiplicidade (e qualquer outra propriedade) em *association ends* não navegáveis (GÉNOVA; DEL CASTILLO; LLORENS, 2002);
e) Mapeamento de associações navegáveis em ambos os sentidos (*associações bidirecionais*) quando pelo menos um dos *association ends* não possui visibilidade pública (GÉNOVA; DEL CASTILLO; LLORENS, 2002).

A opção de que os *association ends* sejam propriedades da associação é impossível porque não existe uma estrutura que represente a associação como *cidadão de primeira ordem*. O código das associações está decomposto nos *association ends*, sendo que estes estão declarados completamente no corpo das classes participantes (GÉNOVA; DEL CASTILLO; LLORENS, 2002).

Association ends não navegáveis não possuem codificação sendo esta a característica que determina que o mesmo não seja navegável. Sendo assim, não é possível que estes contenham propriedades (como a multiplicidade) porque estes não existem no código do programa.

Outra limitação deste *pattern* é a implementação da verificação de consistência. Esta não é uma invariante, mas uma pré-condição na execução dos métodos de acesso às associações (*get*). Isto significa que objetos podem estar em estado de erro quanto às multiplicidades mínimas de suas associações, até que um método de acesso a estas associações seja executado (GÉNOVA; DEL CASTILLO; LLORENS, 2002).

Isto pode causar problemas ao programador quando for depurar erros, pois o local onde ocorreu a exceção não é necessariamente o mesmo onde o problema foi gerado. Por exemplo, um objeto o_1 pode entrar em estado de erro quanto à multiplicidade mínima de uma de suas associações durante a execução de um método m_1. Após, outro método m_2 poderia fazer a chamada de um método de acesso da associação em o_1. Então, a exceção será levantada de dentro do método m_2, no local onde foi chamado o método de acesso e não dentro de m_1, onde o problema foi originado.

3.1.1.2 Code Pattern de Gessenharter

O *code pattern* de Gessenharter (2009) tem por finalidade mapear associações da UML em código puro da linguagem Java. Eles abordam associações binárias com limites inferior e superior de multiplicidade, visibilidade, navegabilidade e *association end ownership*.

Diferentemente do *code pattern* de Génova *et al*, este não utiliza um objeto contendo os métodos de acesso do nome de papel, decompondo os métodos de acesso e modificação da associação no corpo da classe e/ou associação (variando conforme o proprietário do *association end*).

Este *code pattern* utiliza sempre o *pattern objeto de associação* para implementar associações, independente dos *association ends* pertencerem às classes ou à associação. Os *association ends* pertencentes às classes participantes possuem apenas as estruturas de navegação (métodos *add*, *remove*, *get*), delegando suas chamadas ao *objeto de associação*, responsável pelo armazenamento dos *links*. Nos casos em que os *association ends* pertencem às associações (navegáveis ou não), nenhuma estrutura de navegabilidade é necessária no código das classes participantes, estando estas presentes somente no código da classe que implementa a associação (o *objeto de associação*) (GESSENHARTER, 2009).

A Figura 19 apresenta a implementação da classe *A* e a Figura 20 apresenta a implementação das classes *B* (linhas 1 a 13) e *C* (linha 14), todas referentes ao modelo apresentado pela Figura 17.

Código Java

```
1  public class A {
2     /* carregamento de handle */
3     static { CA.fInstance.getHandle(A.class); }
4     private static Object handleCA;
5     public static void takeHandleCA(Object handle)
6        { handleCA = handle; }
7     /*association end da associação CA, referente a C*/
8     private void addC(C c)
9        { CA.fInstance.createLink(c, this); }
10    private void removeC(C c)
11       { CA.fInstance.destroyLink(c, this); }
12    private List<C> getC()
13       {return CA.fInstance.getLinks(this,handleCA);}
14 }
```

Figura 19. Exemplo de código para a classe *A*, utilizando o *pattern* de Gessenharter.

Código Java

```
1  public class B {
2     /* carregamento de handle */
3     static { BC.getHandle(C.class); }
4     private static BC handleBC;
5     public static void takeHandleBC(BC assoc)
6        { handleBC = assoc; }
7     /* association end da associação BC, referente a C*/
8     protected void addC(C c) {handleBC.createLink(this,c);}
9     protected void removeC(C c)
10       { handleBC.destroyLink(this, c); }
11    protected List<C> getC()
12       { return handleBC.getLinks(this); }
13 }
14 public class C { }
```

Figura 20. Exemplo de código para as classes *B* e *C*, utilizando o *pattern* de Gessenharter.

Código Java

```
1  public class AB {
2     /* singleton pattern */
3     public static final AB fInstance = new AB();
4     private AB(){}
5
6     /* gerenciamento da associação */
7     private List<Link> tuples = new LinkedList<Link>();
8     public void createLink(A a, B b) {
9        if (a != null & b != null) {
10          tuples.add(new Link(a,b));   } }
11    public void destroyLink(A a, B b) {
12       for(Link l : tuples) {
13          if (l.a == a && l.b == b) {
14             tuples.remove(l);
15             break; } } }
16    public List<B> getLinks(A a) {
17       List<B> result = new LinkedList<B>();
18       for(Link l : tuples) {
19          if (l.a == a) {
20             result.add(l.b); } }
21       return result; }
22
23    /* implementação dos links */
24    private class Link {
25       private A a;
26       private B b;
27       public Link(A a, B b){ this.a = a;  this.b = b; } }
```

Figura 21. Código para a classe de associação *AB*, utilizando o *pattern* de Gessenharter.

Os *association ends* de nome *c*, referentes à classe *C* e pertencentes às classes *A* e *B*, estão localizados no corpo de ambas as classes às quais pertencem (linhas 8 a 13 da Figura 19 e linhas 8 a 12 da Figura 20).

A Figura 21 apresenta a implementação da classe de associação *AB*, presente no modelo ilustrado pela Figura 17, em formato de associação. Ambos *association ends* são pertencentes à associação, sendo um deles não navegável. Sendo o *association end* referente à classe *A* não navegável, somente o *association end* de nome *b*, referente à classe *B*, possui método de consulta de *links* de associação (método *getLinks* nas linhas 16 a 21). Este, assim como os métodos de criação e destruição de *links* de associação, é acessível publicamente em qualquer ponto do programa que se tenha visibilidade para a classe que mapeia a associação *AB* (GESSENHARTER, 2009).

A classe *nested* chamada *Link* (linhas 24 a 27 da Figura 21) é responsável por instanciar os *links* de associação. Apesar de não ser considerada a implementação de classes de associação neste *pattern*, ela não é impossível. Atributos e métodos poderiam ser adicionados na classe *Link*, e métodos que retornam suas instâncias podem ser construídos.

Semelhantemente, associações n-árias podem ser implementadas acrescentando atributos na classe *Link* para cada nova classe participante. Isto acarretaria a necessidade de novos argumentos no seu construtor e nos métodos da classe de implementação da associação que criam e destroem os links (*createLink* e *destroyLink*). Métodos *getLinks* retornariam coleções de tuplas no lugar de coleções de objetos (GESSENHARTER, 2009).

A classe que implementa a associação é responsável por instanciar um único objeto a ser utilizado como *singleton*. Em alguns casos, como o da Figura 22, este único objeto instanciado também é utilizado como *handle* que tem como finalidade restringir, aos objetos que o detém, o uso de seus métodos públicos (GESSENHARTER, 2009).

O *handle* utilizado no código ilustrado pela Figura 22 é um *handle global*. Isto significa que a instância que o possuir terá acesso completo a todos os métodos públicos da classe o qual ele se refere. Este é caracterizado pelo objeto da classe encapsulado pela mesma (visibilidade *private* na linha 3), passado para as classes que possuem permissão de acesso (método *getHandle* nas linhas 7 a 8) (GESSENHARTER, 2009).

Código Java

```
1  public class BC {
2     /* singleton pattern */
3     private static final BC fInstance = new BC();
4     private BC(){}
5
6     /* handle */
7     public static void getHandle(Class<?> End) {
8        if (End == B.class) {B.takeHandleBC(fInstance);  } }
9
10    /* gerenciamento da associação */
11    ...
12    public List<C> getLinks(B b) {
13       List<C> result = new LinkedList<C>();
14       for(Link l : tuples) {
15          if (l.b == b) { result.add(l.c); } }
16       return result;}
17
18    /* implementação dos links */
19    ...
20 }
```

Figura 22. Exemplo de código para a associação *BC*, utilizando o *pattern* de Gessenharter.

O *handle de método* é o utilizado pelo código ilustrado na Figura 23. Ele é utilizado localmente em cada método de forma que somente os métodos que o requerem são protegidos pelo mesmo. Os demais métodos públicos estão disponíveis sem restrições de uso. Este tipo de *handle* é caracterizado pela disponibilização pública do objeto único (linha 3), pelo *handle* privativo a este (linha 7) em conjunto com o método de instância (não *static*) *getHandle* (linhas 8 a 9). Os métodos protegidos pelo *handle*, como o método *getLinks* (linhas 14 a 22), possuem um parâmetro extra o qual recebem o *handle* para comparação interna (linha 16). Se o *handle* é válido, o método é executado normalmente, caso contrário, não (GESSENHARTER, 2009).

A verificação de limites de multiplicidade, apesar de não ser abordada neste *code pattern*, pode ser implementada de forma similar ao trabalho anterior de *Gessenharter* (2008). Neste, a verificação de multiplicidade é feita pelos métodos que criam e destroem *links* da associação.

A violação temporária dos limites de multiplicidade, necessária em alguns casos como associações em que os limites de multiplicidade inferior e superior são iguais, é mencionada como possível, porém não é

apresentada uma implementação. É proposto o uso de *factory pattern*[5] para o momento de instanciação e de mecanismos similares a transações de banco de dados para o momento de substituição de um *link* por outro, permitindo violação temporária de multiplicidade (GESSENHARTER, 2008).

Código Java

```
 1  public class CA {
 2     /* singleton pattern */
 3     public static final CA fInstance = new CA();
 4     private CA(){}
 5
 6     /* handle */
 7     private Object handle = new Object();
 8     public void getHandle(Class<?> End) {
 9        if (End == A.class) { A.takeHandleCA(this.handle); }
10     }
11
12     /* gerenciamento da associação */
13     ...
14     public List<C> getLinks(A a, Object handle) {
15        List<C> result = null;
16        if (handle == this.handle) {
17           result = new LinkedList<C>();
18           for(Link l : tuples) {
19              if (l.a == a) { result.add(l.c); }
20        } }
21        return result;
22     }
23     public List<A> getLinks(C c) {
24        List<A> result = new LinkedList<A>();
25        for(Link l : tuples) {
26           if (l.c == c) { result.add(l.a); }
27        }
28        return result;   }
29
30     /* implementação dos links */
31     ...
32  }
```

Figura 23. Exemplo de código para a associação *CA*, utilizando o *pattern* de Gessenharter.

Apesar de este *code pattern* resolver o problema da semântica de associações da UML 2, ele apresenta um efeito colateral, descrito neste trabalho como *o problema da referência perdida* (ver Subseção 3.2.2)

[5] Gamma *et al*, 1995.

3.1.2 Bibliotecas para Construção de Associações

Bibliotecas para abstração da construção de associações caracterizam-se por conter estruturas genéricas e parametrizadas para gerenciamento e construção de associações. Estas estruturas necessitam somente ser invocadas ou estendidas para que suas características genéricas sejam acrescentadas ao código de programação, deixando que somente as especificidades e os parâmetros necessitem ser codificados.

As bibliotecas analisadas foram Noiai e RAL, descritas a seguir.

3.1.2.1 RAL: Relationship Aspect Library

Aspectos, na linguagem AspectJ, permitem que estruturas declaradas em seu código, como atributos e métodos, sejam inseridas em instâncias de classes, pelo emprego de *inter-type declarations* (PEARCE; NOBLE, 2006a).

Pearce e Noble (2006b) apresentam *patterns* escritos em AspectJ que permitem que todo o código de uma associação seja removido do código das classes participantes e sejam inseridos em um aspecto, promovendo o desacoplamento da classe em relação à associação, de forma similar ao *pattern objeto de associação,* diferenciando-se deste pelo local do armazenamento dos *links* de associação. Este armazenamento, apesar de ser feito por coleções declaradas no código do aspecto, continuam sendo armazenadas nas instâncias da classe.

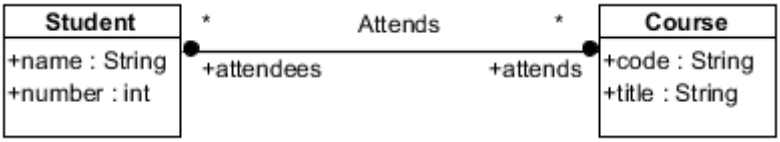

Figura 24. Exemplo de associação *Attends* entre a classe *Student* e *Course*.
Fonte: Diagrama de Classe do exemplo de Pearce e Noble (2006b)

Com a finalidade de ilustrar o uso de programação orientada a aspectos para modelar associações, bem como as vantagens desta sobre a modelagem feita em programação orientada a objetos, a Figura 24 ilustra um modelo de uma associação simples entre duas classes. Este pode ser implementado em código Java puro conforme apresentado pela Figura 25.

Código Java

```
1  class Student {
2    // atributos
3    String name;
4    int number;
5
6    // associação attends
7    HashSet<Course> attends;
8  }
9
10 class Course {
11   // atributos
12   String code;
13   String title;
14
15   // associação attends
16   HashSet<Student> attendees;
17   void enrol(Student s) {
18     this.attendees.add(s);
19     s.attends.add(this);
20   }
21   void withdraw(Student s) {
22     this.attends.remove(s);
23     s.attends.remove(this);
24 } }
```

Figura 25. Exemplo de implementação em código Java puro do modelo ilustrado pela Figura 24.
Fonte: Pearce e Noble (2006b)

A Figura 26 apresenta implementação similar à apresentada pela Figura 25, utilizando recursos *inner-type declarations* da linguagem AspectJ. Nesta, o aspecto *Attends* (linhas 13 a 28) declara dois conjuntos (linhas 15 e 16), um em cada classe participante. Estes, apesar de inseridos no escopo das classes, são visíveis somente dentro do aspecto que os declarou (chamadas nas linhas 19, 20, 23, 24 e 27). Em outras palavras: as classes não têm conhecimento destas estruturas, somente o aspecto. Porém, o aspecto somente tem acesso a estas estruturas por intermédio de instâncias destas classes. Isto cria encapsulamento destas estruturas, significando que estes somente são acessíveis através do aspecto.

Código AspectJ

```
1  class Student {
2    // atributos
3    String name;
4    int number;
5  }
6
7  class Course {
8    // atributos
9    String code;
10   String title;
11 }
12
13 aspect Attends {
14   // inner-type declaration.
15   HashSet<Course> Student.attends;
16   HashSet<Student> Course.attendees;
17
18   public void enrol(Student s, Course c) {
19     c.attendees.add(s);
20     s.attends.add(c);
21   }
22   public void withdraw(Student s, Course c) {
23     c.attends.remove(s);
24     s.attends.remove(c);
25   }
26   public HashSet<Student> getAttendees(Course c) {
27     return c.attendees;
28 } }
```

Figura 26. Implementação AspectJ equivalente a implementação da Figura 25.
Fonte: Pearce e Noble (2006b)

A Figura 27 apresenta um exemplo de execução do modelo ilustrado pela Figura 25 e a Figura 28 o mesmo exemplo de execução, porém referente à implementação ilustrada na Figura 26.

Código Java

```
1  Course comp205 = new Course();
2  Student joe = new Student();
3
4  // cria link <joe,comp205>
5  Comp205.enrol(joe);
6  // lista links <*,comp205>
7  for(Student x : comp205.attendees) {
8    System.out.println(x + "is enrolled"); }
9  // destrói link <joe,comp205>
10 comp205.withdraw(joe);
```

Figura 27. Exemplo de execução do modelo ilustrado pela Figura 25.
Fonte: Adaptação de exemplo encontrado em Pearce e Noble (2006a)

RAL, sigla para *Relationship Aspect Library*, é uma biblioteca para a linguagem AspectJ que utiliza aspectos para implementação de associações. Esta contém aspectos abstratos parametrizados por tipos genéricos, contendo todas as estruturas genéricas necessárias a uma associação (PEARCE; NOBLE, 2006a).

Código AspectJ

```
1  Course comp205 = new Course();
2  Student joe = new Student();
3
4  // cria link <joe,comp205>
5  Attends.aspectOf().enrol(joe,comp205);
6  // lista links <*,comp205>
7  for(Student x : Attends.aspectOf().getAttendees(comp205))
8     { System.out.println(x + "is enrolled"); }
9  // destrói link <joe,comp205>
10 Attends.aspectOf().withdraw(joe,comp205);
```

Figura 28. Exemplo de execução do modelo ilustrado pela Figura 26
Fonte: Adaptação de exemplo encontrado em Pearce e Noble (2006a)

A Figura 29 apresenta uma implementação que utiliza a biblioteca e tem a mesma semântica da associação ilustrada na Figura 26. A palavra reservada *this* empregada nos métodos (linhas 3, 6 e 9), assim como no contexto de uma classe, referencia a instância do aspecto. Esta é única (*singleton*) e pode ser acessada externamente pelo método de aspecto (*static*) *aspectOf* (linhas 5, 7 e 10 da Figura 28).

Código AspectJ

```
1  aspect Attends extends SimpleStaticRel<Student,Course> {
2     public void enrol(Student s, Course c) {
3        this.add(s,c);
4     }
5     public void withdraw(Student s, Course c) {
6        this.remove(s,c);
7     }
8     public HashSet<Student> getAttendees(Course c) {
9        return this.to(c);
10 } }
```

Figura 29. Adaptação do exemplo da Figura 26, utilizando RAL.

O aspecto abstrato *SimpleStaticRel<F,T>*, estendido pelo aspecto concreto *Attends* na Figura 29, implementa os métodos *add* (chamado na linha 3), *remove* (chamado na linha 6), *from* e *to* (chamado na linha 9), responsáveis por adicionar, remover e consultar links de associação em ambos os participantes. Estes métodos são públicos, significando que os métodos *enrol*, *withdraw* e *getAttendees* poderiam ser removidos do

aspecto *Attends*, substituindo as chamadas destes métodos pelos já existentes (PEARCE; NOBLE, 2006a).

A biblioteca possui outro aspecto abstrato que tem por finalidade implementar classes de associação. Este é o aspecto *StaticRel<F,T,P>* em que os tipos genéricos *F* e *T* têm a mesma função que os tipos genéricos homônimos do aspecto *SimpleStaticRel<F,T>*: parametrizar os tipos de ambos os participantes da associação, em que *F* indica o primeiro participante (*from*) e *T* o segundo (*to*). O parâmetro *P* indica a classe que instanciará os pares (tuplas) de instâncias participantes da associação (PEARCE; NOBLE, 2006a).

Os aspectos abstratos apresentados são utilizados para implementar *associações estáticas*. De forma análoga, há classes abstratas para implementação de *associações dinâmicas*. *Associações estáticas* são associações persistentes durante toda a execução do programa enquanto que *associações dinâmicas* são associações temporárias. Esta diferenciação existe por questões de desempenho e custo de memória, e está intimamente relacionado com o local de armazenamento dos links de associação. Nas *associações estáticas*, os *links* são armazenados em coleções presentes nas instâncias participantes da associação de forma que se ambas as instâncias associadas perderem suas referências no programa, ambas serão coletadas em conjunto com o *link* de associação. Nas *associações dinâmicas*, os *links* são armazenados no *objeto de associação*. Este não é um *singleton*, mas um objeto que deve ser instanciado e referenciado somente no interior de um método. Ao perder sua última referência, o objeto que representa a *associação dinâmica* é coletado, causando a coleta de todos os *links* de associação pertencentes a ele. Por não armazenar os *links* nas instâncias participantes, esta abordagem não insere coleções nestas, sendo então implementada em uma classe em vez de um aspecto (PEARCE; NOBLE, 2006a).

Para padronizar as associações de maneira independente da implementação, RAL fornece um conjunto de interfaces para este fim. Associações simples, independente de serem estáticas ou dinâmicas, implementam a interface *SimpleRelationship<F,T>*. Associações em que as tuplas representam instâncias de classes de associação implementam a interface *Relationship<F,T,P>*, sejam estas associações estáticas ou dinâmicas. Ambas as interfaces citadas, assim como a interface necessária para a classe que instancia as tuplas (pares), são apresentadas na Figura 30 (PEARCE; NOBLE, 2006a).

Código Java

```
interface Relationship<F, T, P extends Pair<FROM,TO>> {
   public void add(P pair);
   public void remove(P pair);
   public Set<P> toPairs(T t);
   public Set<P> fromPairs(F f);
   public Set<F> to(T t);
   public Set<T> from(F f);
   ...
}

interface Pair<FROM,TO> {
   public FROM from();
   public TO to();
}

interface SimpleRelationship<F,T>
     extends Relationship<F,T,FixedPair<F,T>> {
   public boolean add(F f, T t);
   public boolean remove(F f, T t);
}
```

Figura 30. Ilustração dos cabeçalhos das interfaces *Relationship*, *Pair* e *SimpleRelationship*.
Fonte: Pearce e Noble (PEARCE; NOBLE, 2006a)

Utilizando estes conceitos, RAL permite implementar associações que não acoplam seu código nas classes participantes, criando-se maior coesão, permitindo reusabilidade das classes (PEARCE; NOBLE, 2006a).

3.1.2.1.1 Limitações em Relação à Semântica da UML

Na biblioteca constam implementações de associações sem limites de multiplicidade e com limite superior de multiplicidade igual a 1. Outras características da UML, como limite inferior de multiplicidade, visibilidade, nomes de papel e especialização de associações, não são abordadas. Associações sempre são navegáveis em ambos os sentidos e os association ends sempre pertencem às associações.

3.1.2.2 NOIAI: No Object Is An Island

Noiai é uma biblioteca para a linguagem C# que contém classes a serem especializadas para a construção de associações binárias. Estas classes possuem os mecanismos genéricos de gerenciamento de

associações de forma que as associações escritas no código de programa são classes especialistas destas (ØSTERBYE, 2007).

Há duas formas de armazenamento dos links de associação: no *singleton* da associação ou nas instâncias participantes. Associações em que o armazenamento de links ocorre no *singleton*, estendem a classe *BaseAssociation<FROM,TO,THIS>* e as que armazenam nas instâncias, a classe *RoleAssociation<FROM,TO,THIS>*. Em ambos os casos, a associação pode ser manipulada (alterada e/ou acessada) pelo *singleton* da associação e/ou pelos objetos de nomes de papel. Objetos representando nomes de papel podem – no caso da implementação *BaseAssociation<...>* – ou devem – no caso da implementação *RoleAssociation<...>* – ser instanciados e armazenados em variáveis de instâncias das classes participantes. Estes também permitem que as instâncias proprietárias naveguem pela associação através dos próprios, que possuem referências para os objetos da classe oposta. Apesar da possibilidade ou obrigatoriedade do uso destes objetos, é impossível restringir a navegação somente a estes (ØSTERBYE, 2007).

Nas classes *BaseAssociation<...>* e *RoleAssociation<...>*, os tipos genéricos, *FROM* e *TO* parametrizam as classes associadas, onde *FROM* é a primeira e *TO* é a segunda. Por ser um relacionamento simétrico, estes nomes servem apenas para haver distinção. O tipo genérico THIS é sempre a própria classe que está especializando *BaseAssociation<...>* ou *RoleAssociation<...>*. Isto é necessário para que as classes generalistas instanciem o *singleton* com as estruturas internas necessárias para a associação declarada e para as associações especialistas desta (chamadas de *sub-associações*) (ØSTERBYE, 2007).

Para mapear classes de associação, a biblioteca conta com mais uma versão das classes *BaseAssociation<...>* e *RoleAssociation<...>*. Estas possuem o mesmo nome, porém contêm um tipo genérico a mais que é o responsável por parametrizar a classe que irá conter os atributos e métodos da classe de associação (ØSTERBYE, 2007).

A Figura 31 apresenta o uso da biblioteca para implementar o modelo ilustrado pela Figura 17. A classe de associação *AB* é mapeada pelas classes *AB* (linhas 12 a 14) e *ABAssoc* (linha 17), sendo a primeira a responsável por instanciar as instâncias da classe de associação, e a segunda responsável por gerenciar a classe de associação. As classes *BC* (linha 18) e *CA* (linha 19) mapeiam e gerenciam as associações de mesmo nome. Os nomes de papeis *c*, presentes nas classes *A* e *B* (linhas 2 e 6, respectivamente) representam os *association ends* pertencentes às classes.

Código C#

```
1  class A {
2    private CA.FromEntitySet c;
3    public A() { this.c = new CA.FromEntitySet(this); }
4  }
5  class B {
6    protected BC.FromEntitySet c;
7    public B() { this.c = new BC.FromEntitySet(this); }
8  }
9
10 class C { }
11
12 class AB {
13   // atributos e métodos;
14 }
15
16 [Cardinality(To = Cardinality.Unique)]
17 class ABAssoc : BaseAssociation<A,B,ABAssoc,AB> {}
18 class BC : BaseAssociation<B,C,BC> {}
19 class CA : BaseAssociation<C,A,CA> {}
```

Figura 31. Codificação do modelo ilustrado pela Figura 6 utilizando a biblioteca NOIAI.

Por não ser possível declarar *association ends* não navegáveis, as associações *AB* e *BC*, originalmente navegáveis em apenas um sentido, foram mapeadas como associações navegáveis em ambos os sentidos. Dos limites de multiplicidade presentes no modelo, os únicos possíveis de serem representados são os limites * e *0..1*. O primeiro caso é o padrão em Noiai, não necessitando de declaração. O segundo é declarado pelo *attribute*[6] *Cardinality* e seus *fields From* e *To* recebendo por atribuição a constante *Cardinality.Unique* (linha 19). Cada *field* indica a cardinalidade do respectivo participante (*From* é o primeiro e *To* o segundo) (ØSTERBYE, 2007).

Noiai também possui suporte a especialização de associações através de extensão da classe *SubAssociation<FROM, TO, THIS, SUPER>*[7] em que os tipos genéricos *FROM*, *TO* e *THIS* possuem a mesma função dos parâmetros homônimos das classes citadas anteriormente. O tipo genérico *SUPER* parametriza a classe que mapeia a associação generalista. O mapeamento de especialização de classes de associação é feito por extensão de outra variante da classe

[6] Microsoft, 2011.

[7] O cabeçalho desta classe não é apresentado no trabalho de Østerbye (2007), apenas sua chamada. O nome do tipo genérico de nome *SUPER* foi atribuído na quarta posição apenas para exemplificação.

SubAssociation<...>, que inclui um tipo genérico a mais para especificar a classe que contém os atributos e métodos (ØSTERBYE, 2007).

3.1.2.2.1 Limitações em Relação à Semântica da UML

Associações construídas com o uso desta biblioteca são sempre navegáveis em ambos os sentidos. O acesso e modificação dos links de associação podem sempre ser feitos pela associação, apesar de que os nomes de papeis possam ser atribuídos às classes participantes. Isto significa que mesmo declarando os *association ends* como propriedades pertencentes às classes, os mesmos podem ser acessados como sendo pertencentes à associação, inutilizando qualquer encapsulamento feito por meio da declaração de restrição de visibilidade dos nomes de papeis.

Os limites de multiplicidade são implementados de forma incompleta. Não há limite inferior de multiplicidade e os limites superiores possíveis sempre são os equivalentes na UML a *1* ou *.

Os nomes de papeis somente são nomeados quando o *association end* é um atributo da classe. Quando não é, não há nomeação de nome de papel.

A especialização de associações e classes de associação não possuem a mesma semântica da UML 2 devido algumas características:

a) É possível declarar, em um *association end* da associação especialista, um limite superior de multiplicidade maior que o limite superior de multiplicidade do respectivo *association end* presente na associação generalista;
b) Um link da associação especialista não é um link da associação generalista, de forma que dois objetos podem estar *linkados* na associação especialista, mas não estarem *linkados* na associação generalista.

3.1.2.2.2 Problemas no Armazenamento de Links da Implementação BaseAssociation

A abordagem *BaseAssociation*<...>, apesar de armazenar todos os *links* de associação no *singleton*, não causa o *problema da visibilidade global* (ver Subseção 3.2.1) porque os *links* de associação são acessíveis somente com a comparação de pelo menos uma referência de uma instância participante. Também não causa o *problema da referência perdida* (ver Subseção 3.2.2) porque os *links* de associação são coletados após ambas as instâncias participantes serem coletadas. Isto é possível porque os *links* de associação são armazenados em dois

dicionários de referências fracas (um para cada lado da associação) (ØSTERBYE, 2007).

Porém, as conseqüências desta abordagem não foram mencionadas pelo autor do trabalho.

> Internally, the linkage objects are not stored in a list, but in *two weak dictionaries* […] both objects of a linkage should be dead before the linkage itself should die. Weak dictionaries build on top of weak references, which allow us to *check if the referenced object is dead*. It is thus possible to check this situation explicitly, and *remove the linkage from the dictionary when both objects are dead* (ØSTERBYE, 2007, p. 73, grifo nosso).

Isto significa que um *link* não mantém referências fortes para os objetos participantes, porque caso contrário, seria impossível os objetos participantes do *link* serem coletados antes do próprio *link* (considerando que o método de remoção de *links* não foi executado). Sendo assim, não há no *singleton* da associação, referências fortes para os objetos associados, significando que um objeto somente se mantém *vivo* se estiver sendo referenciado por uma variável (local, de instância ou de classe).

Esta abordagem é impraticável na implementação de associações porque se espera que a maioria dos objetos sejam referenciados somente por *links* de associação, não por variáveis locais e parâmetros de métodos, nem por atributos. Exemplificando: um objeto *a* é referenciado por uma variável (local em um método, de instância ou de classe) e está *linkado* com um objeto *b*, sendo que *b* está *linkado* com um objeto *c*, *c* está *linkado* com um objeto *d* e assim sucessivamente. Considerando que *links* de associação são referências fortes, não há necessidade de que os objetos *b*, *c* e *d* sejam referenciados por variáveis para permanecerem *vivos*. Caso contrário, se considerar que *links* de associação são referências fracas, todos os objetos necessitam serem referenciados por variáveis para não serem coletados pelo *garbage collector*.

3.1.3 Linguagens de Programação com Suporte a Associações

As linguagens de programação com suporte a associações caracterizam-se por serem linguagens que adotam o paradigma de orientação a objetos e que implementam nativamente uma estrutura de declaração de colaboração entre classes.

As linguagens com suporte a associações analisadas foram: DSM, a linguagem de banco de dados apresentada por Albano, Ghelli e Orsini

(1991), RelJ e um modelo de linguagem de programação apresentado por Balzer, Gross e Eugster (2007). Estas são apresentadas a seguir, nas próximas subseções.

3.1.3.1 DSM: Data Structure Manager

DSM, sigla para *Data Structure Manager*, foi a primeira linguagem de programação orientada a objetos com suporte a associações. É uma implementação do modelo *objeto-relação* apresentado por Rumbaugh (1987), abordando associações binárias e associações qualificadas (SHAH *et al*, 1989). É uma linguagem implementada em C. Não possui *garbage collector* e sendo assim, necessita que o programador especifique a destruição de objetos por meio da chamada do método destrutor das classes (SHAH *et al*, 1989).

Código DSM

```
1  DEFINE a CLASS
2  METHODS
3    add_b(b)
4    get_b(): set OF b
5    ...
6    add_c(c)
7    get_c(): set OF c
8    ...
9  DEFINE b CLASS
10 METHODS
11   add_a(a)
12   get_a(): set OF a
13   ...
14   add_c(c)
15   get_c(): set OF c
16   ...
17 DEFINE c CLASS
18 METHODS
19   add_b(b)
20   get_b(): set OF c
21   ...
22   add_a(a)
23   get_a(): set OF a
24   ...
25 DEFINE ab RELATION
26   a:a *-1 b:b
27 DEFINE bc RELATION
28   b:b 1-* c:c
29 DEFINE ca RELATION
30   c:c *-* a:a
```

Figura 32. Codificação do modelo ilustrado pela Figura 6 utilizando a linguagem DSM.

A Figura 32 apresenta uma implementação escrita em código DSM que mais se aproxima do modelo ilustrado pela Figura 17. As linhas 1 a 8 apresentam o código que define a classe *A*. As linhas 9 a 16 e 17 a 24, apresentam a definição das classes *B* e *C*. As associações *AB*, *BC* e *CA*, com os nomes de papel dos participantes, estão descritas nas linhas 25 a 26, 27 a 28 e 29 a 30 respectivamente.

Os cabeçalhos dos métodos de acesso e modificação do estado das associações estão declarados no corpo das classes. Não há, no trabalho de Shah *et al* (1989), exemplificação de todos os métodos de acesso e modificação do estado das associações, sendo estes então suprimidos do código ilustrado.

A implementação dos métodos de acesso e modificação do estado das associações é adicionada pelo compilador da linguagem, necessitando que os cabeçalhos dos métodos sejam adicionados nas classes pelo programador (SHAH *et al*, 1989).

3.1.3.1.1 Limitações em Relação à Semântica da UML

Apesar de o modelo *objeto-relação* prever limites inferiores de multiplicidade superiores a *0*, DSM implementa apenas limites superiores de multiplicidade que podem ser iguais a *1* ou * (RUMBAUGH, 1987; SHAH *et al*, 1989).

Associações são sempre navegáveis em ambos os sentidos através de métodos inseridos, por intermédio do compilador, nas classes participantes (SHAH *et al*, 1989).Sendo assim, o acesso é sempre realizado pelos métodos das instâncias, o que corresponde ao conceito de *association end* pertencente à classe, na UML 2.

Quanto à declaração de visibilidade, não está claro se os métodos de acesso às associações podem ter sua visibilidade modificada. É mencionado somente o acesso e visibilidade de atributos: *"Field access may be write, or read-only by other classes, or private to the descendant classes only"*. O modelo *objeto-relação* também não menciona nada a respeito. Porém, esta característica é possível de ser implementada no compilador da linguagem (RUMBAUGH, 1987; SHAH *et al*, 1989, p. 192).

Não há, nem no modelo *objeto-relação*, nem na linguagem implementada, abordagem sobre especialização de associações.

3.1.3.2 A Linguagem de Albano et al

Albano, Ghelli e Orsini (1991) apresentam uma linguagem fortemente tipada para banco de dados orientado a objetos em que associações são construções nativas. Utiliza outro conceito para classes, diferente do convencionalmente adotado em linguagens como Java, C++, entre outras. Nesta linguagem, uma classe é um subconjunto de objetos criados pelo tipo a que a classe refere-se. Exemplo: uma classe *c* é um subconjunto do conjunto de objetos do tipo *C*, onde *C* caracteriza e instancia os objetos e *c* inclui e exclui os objetos instanciados.

Similar a classes, associações *n*-árias são subconjuntos de *n*-tuplas instanciadas com as assinaturas a que as associações referem-se. Exemplo: Dada as classes *a* e *b*, referente aos tipos *A* e *B*, e uma associação binária *ab*, referente às tuplas com assinatura $<a_1:A,b_1:B>$, onde a_1 é um rótulo (ou *nome de papel*) para uma instância do tipo *A* e b_1 é um rótulo para uma instância do tipo *B*, infere-se que *ab* é um subconjunto do conjunto de todas as tuplas *<a1:A,b1:B>* instanciadas (ALBANO; GHELLI; ORSINI, 1991).

Uma classe pode ou não conter uma instância do tipo a que a classe se refere, assim como uma associação pode ou não conter uma tupla com a assinatura a qual a associação se refere.

Para que um objeto participe de uma associação, é necessário que ele esteja incluso em sua classe. No exemplo anteriormente citado, um objeto chamado de a_x, do tipo *A*, só pode participar da associação *ab* se anteriormente ele estiver incluso na classe *a*. Este requisito, definido pela *referencial constraint*, é obrigatório também para um objeto b_y do tipo *B*, que deve ser incluído na classe *b* antes de participar de uma tupla contida na associação *ab*. Após a_x e b_y estarem inclusos em suas classes, a tupla $<a_x,b_y>$, criada em qualquer momento anterior com assinatura *<a1:A,b1:B>*, pode ser adicionada na associação *ab* (ALBANO; GHELLI; ORSINI, 1991).

Isto faz das classes *a* e *b* e da associação *ab*, conjuntos. Sobre estes conjuntos, *constraints* com operações sobre *relação* entre conjuntos podem ser definidas.

A associação *ab* tem uma *surjectivity constraint* na classe *a* quando todos os objetos contidos em *a* estão obrigatoriamente em tuplas contidas em *ab*. A associação *ab* tem uma *key constraint* com a classe *b* quando objetos contidos por *b* só podem aparecer em uma única tupla contida por *ab* (ALBANO; GHELLI; ORSINI, 1991).

Estas duas *constraints* têm semântica similar aos limites de multiplicidade *1..** e *0..1*, respectivamente. Quando combinadas, resultam em uma semântica similar ao limite de multiplicidade *1..1*.

Dependency constraints podem ser definidas de forma que a inclusão ou remoção de elementos de um conjunto implica na inclusão ou remoção do mesmo elemento de outro conjunto. Exemplos (ALBANO; GHELLI; ORSINI, 1991):

a) *ab* depende de *a* (*ab owned by a*) – todo objeto removido de *a* causa remoção de todas as tuplas de *ab* onde o objeto removido participa;
b) *b* depende de *ab* (*ab owns b*) – todos os objetos contidos em *b* que tiveram sua última tupla participante removida de *ab*, são removidos de *b*.

Inclusion constraints podem ser utilizadas para dar a semântica de generalização. Supondo a classe a_2 ser especialista da classe *a*, a_2 é um subconjunto de *a* e deve operar com *inclusion constraint* para que todo objeto inserido em a_2 também seja inserido em *a*. Isto só é possível se A_2 for um subtipo de *A* e a_2 ser uma coleção de objetos do tipo A_2. Similarmente, uma associação *ca* pode ser especialista da associação *ab*, quando *ca* é um subconjunto de *ab* e opera com *inclusion constraint* para que todas as tuplas inseridas em *ca*, sejam também inseridas em *ab*. Isto só é permitido se as tuplas de *ca* forem *subtuplas* das tuplas de *ab*. (ALBANO; GHELLI; ORSINI, 1991).

A *surjectivity constraint* combinada com a *referencial constraint* em um mesmo par de conjuntos, causa um problema ao tentar adicionar elementos em um dos dois conjuntos afetados. No exemplo dado, a associação *ab* tem uma *referencial constraint* e uma *surjectivity constraint*, ambos com a classe *a*. Isto significa que, devido à *referencial constraint*, uma tupla contendo referência para o objeto a_x só pode ser adicionada em *ab* depois que a_x for adicionado em *a*. Porém, devido à *surjectivity constraints*, a_x só pode ser adicionado em *a* se *ab* conter pelo menos uma tupla que referencie o objeto a_x. A tentativa de inserir a_x em *a* causa violação da *surjectivity constraint*. A tentativa de inserir $<a_x,b_y>$, onde b_y é do tipo *B*, viola a *referencial constraint* (ALBANO; GHELLI; ORSINI, 1991).

Para solucionar problemas como este, a linguagem conta com um mecanismo de transações que permite violar temporariamente as *constraints*. Assim, dentro de uma destas transações, é possível executar as operações que adicionam a_x a *a* e $<a_x,b_y>$ à *ab*, não importando a ordem de execução destas, sendo estas validadas ao final da transação (ALBANO; GHELLI; ORSINI, 1991).

A Figura 33 apresenta um exemplo de código desta linguagem que mais se aproxima do modelo ilustrado pela Figura 17.

Código Linguagem Albano et al

```
1  let a = new(classOf A)
2  let b = new(classOf B)
3  let c = new(classOf C)
4
5  let ab = new(assocOf
6                  a1:A in a
7                  b1:B in b
8                  key(b1))
9  let bc = new(assocOf
10                 b1:B in b onto b
11                 c1:C in c onto c
12                 key(b1))
13 let ca = new(assocOf
14                 c1:C in c onto c
15                 a1:A in a)
```

Figura 33. Codificação do modelo ilustrado pela Figura 6 utilizando a linguagem de Albano *et al*.

Os conjuntos são instanciados em *run-time*, atribuídos a variáveis globais. As classes *a*, *b* e *c* são instanciadas nas linhas 1, 2 e 3 respectivamente. A classe de associação *ab* e as associações *bc* e *ca* são instanciadas nas linhas 5 a 8, 9 a 12, e 13 a 15, respectivamente. As palavras reservadas *in* (linhas 6, 7, 10, 11, 14 e 15) representam as *referencial constraints* dos valores atribuídos aos campos das tuplas (lado esquerdo) com os respectivos conjunto (lado direto). As palavras reservadas *onto* (linhas 10, 11 e 14) declaram as *surjectivity constraints* de forma similar (ALBANO; GHELLI; ORSINI, 1991).

Não há, no trabalho de Albano, Ghelli e Orsini (1991), exemplo de definição dos tipos de dados. Classes de associação são definidas da mesma forma que associações, diferenciando-se por conter atributos que se referem a classes participantes da associação.

3.1.3.2.1 Limitações em Relação à Semântica da UML

A linguagem não permite representar limites de multiplicidade diferentes de *0..1*, *1..1*, *1..** e *0..**. Não há restrição de navegabilidade e visibilidade. A associação é sempre manipulada por operações do próprio conjunto que a representa, caracterizando *association ends* pertencentes à associação, nunca às classes.

Associações e classes são vistas como conjuntos de visibilidade global. Pesquisas nestes podem ser feitas de forma que é impossível

restringir o acesso a um objeto por intermédio de visibilidade. Estas pesquisas podem ser feitas por condicionais que testam atributos de uma instância de uma classe ou pela comparação da identidade de uma instância com um dos participantes de um *link* de associação (ALBANO; GHELLI; ORSINI, 1991).

Esta característica gera o *problema da visibilidade global*, descrita detalhadamente na Subseção 3.2.1.

As tuplas (elementos das associações) podem conter atributos de forma similar aos objetos (elementos das classes). Porém, diferente destes, não é possível declarar métodos em tuplas (ALBANO; GHELLI; ORSINI, 1991).

Outra limitação diz respeito ao fato de que as *constraints* (responsáveis também pela possível verificação de multiplicidade) não são testadas em instâncias que não estão contidas por suas classes. Isto significa, que existindo uma associação entre as classes *a* e *b* com multiplicidade *[1..1]* na extremidade de *b*, uma instância a_x da classe *a* pode ser instanciada e não associada a nenhuma instância de *b*, desde que a_x não esteja contida em *a*.

3.1.3.3 RelJ

RelJ é uma extensão de um subconjunto da linguagem Java que tem como finalidade resultar em uma linguagem com suporte a associações. Esta possui a construção nativa de associações, com duas abordagens para a verificação do limite superior de multiplicidade: estática e dinâmica (BIERMAN; WREN, 2005).

A abordagem estática só permite declarar limites superiores de multiplicidade iguais a 1 ou *. No caso de limite superior igual a 1, quando uma operação de inclusão de novo *link* de associação violar esta invariante por já haver um *link* anteriormente, este antigo *link* é destruído para dar lugar ao novo *link* que está sendo criado, mantendo a cardinalidade sem levantar exceção (BIERMAN; WREN, 2005).

A abordagem dinâmica permite declarar limites superiores de multiplicidade iguais a qualquer número natural, incluindo o valor indefinido ("*" em UML). Neste caso, quando uma operação de inclusão de *link* de associação violar o limite superior de multiplicidade, uma exceção é levantada (BIERMAN; WREN, 2005).

A Figura 34 apresenta o código RelJ para o modelo ilustrado pela Figura 17. A única diferença entre associações e classes de associação é que a segunda possui atributos e métodos declarados em seu corpo. Isto é aparente pela comparação do código da classe de associação *AB*

(linhas 5 a 7) com as associações *BC* (linha 9) e *CA* (linha 10) (BIERMAN; WREN, 2005).

Código RelJ

```
1  class A { }
2  class B { }
3  class C { }
4
5  relationship AB (many A, one B) {
6    // atributos e métodos
7  }
8
9  relationship BC (one B, many C) { }
10 relationship CA (4 C, 4 A) { }
```

Figura 34. Codificação do modelo ilustrado pela Figura 6 utilizando a linguagem RelJ.

Para representar multiplicidade sem limite superior, utiliza-se a palavra reservada *many* (linhas 5 e 9). Para limite superior igual a 1, utiliza-se a palavra reservada *one* (linhas 5 e 9). Para os demais limites, utiliza-se o número inteiro correspondente (linha 10) (BIERMAN; WREN, 2005).

A Figura 35 apresenta chamadas para as classes e associações codificadas na Figura 34. Operações de adição de associações estão presentes tanto nas associações (linhas 5 e 7) quanto nos *association ends* presentes nas instâncias (linha 6).

Código RelJ

```
1  A a = new A();
2  B b = new B();
3  C c = new C();
4
5  AB ab = AB.add(a,b);    //cria link entre 'a' e 'b'.
6  b.BC += c;              //cria link entre 'b' e 'c'.
7  CA ca = CA.add(c,a);    //cria link entre 'c' e 'a'.
8
9  //lista objetos B linkados a 'a'
10 for(B bi   : a.AB){ print bi; }
11 //lista objetos AB linkados a 'a'
12 for(AB abi : a:AB){ print abi; }
```

Figura 35. Chamada de operações sobre as associações declaradas pelo código ilustrado na Figura 34.

Dado uma instância da primeira classe participante de uma associação, é possível listar:

a) As instâncias da segunda classe participante, associadas a esta (linha 10);
b) As instâncias da associação (ou classe de associação) em que esta participa (linha 12).

RelJ possui o suporte a especialização de classes herdada da linguagem Java e possui especialização de associações com sintaxe similar a esta (BIERMAN; WREN, 2005).

A semântica da especialização de associações é descrita por três invariantes, citadas a seguir.

> Consider a relationship r_2 which extends r_1. For every instance of relationship r_2 between objects o_1 and o_2, there is an instance of r_1, also between o_1 and o_2, to which it delegates requests for r_1's fields (BIERMAN; WREN, 2005, p. 266, grifo nosso).

Esta invariante indica que, de forma idêntica com a especificação UML, toda instância de uma associação especialista é também uma instância da associação generalista (similar a especialização de classes).

A segunda invariante afirma: "*For every relationship r and pair of objects o_1 and o_2, there is at most one instance of r between o_1 and o_2*" (BIERMAN; WREN, 2005, p. 266).

Isto indica que não é possível repetir tuplas de associações, similar as associações da UML 2 em que os *association ends* possuem a propriedade *isUnique*[8] (padrão).

A terceira invariante indica que a semântica dos limites de multiplicidade de forma idêntica a UML 2.

> For a relationship *r*, declared *relationship r (n1, n2)*, where n_1 is annotated with one, there is at most one n_1-instance related through *r* to every n_2-instance. The converse is true where n_2 is annotated with one (BIERMAN; WREN, 2005, p. 280, grifo nosso).

3.1.3.3.1 Limitações em Relação à Semântica da UML

RelJ não aborda limites inferiores de multiplicidade, visibilidade, nomes para os *association ends* (nomes de papel) e *association end ownership*. Associações são navegáveis somente na direção da primeira classe participante para a segunda, não possibilitando o inverso (BIERMAN; WREN, 2005).

[8] Object Management Group, 2010.

3.1.3.4 O Modelo de Balzer et al

Balzer, Gross e Eugster (2007) apresentam um modelo relacional de colaboração de objetos que, de maneira similar a linguagem de Albano *et al* (ver Subseção 3.1.3.2), utiliza:

a) Classes como subconjuntos dos conjuntos de instâncias delas próprias;
b) *Associações binárias* como conjuntos de tuplas de instâncias associadas, ou simplesmente relações binárias entre o primeiro e o segundo participante.

A estes conceitos, este modelo acrescenta *interposição de elementos* e *invariantes*, apresentados nas próximas Subseções.

3.1.3.4.1 Interposição de Elementos

Elementos interpostos são propriedades declaradas no corpo da associação, porém caracterizam uma das duas classes participantes desta. Esta caracterização é referente ao papel desempenhado pela instância da classe participante na associação, não tendo sentido fora deste escopo (BALZER; GROSS; EUGSTER, 2007).

Utilizando o exemplo de um sistema universitário, presente no trabalho de Balzer, Gross e Eugster (2007): um estudante pode, opcionalmente, prestar assistência a um ou mais cursos e um curso pode, opcionalmente, ter assistência de um ou mais estudantes. Para um estudante prestar assistência num curso, é necessário que ele possua uma linguagem de instrução. Entretanto, essa característica não é referente ao curso, nem a prestação de assistência, mas referente ao próprio estudante, que nesta ocasião é chamado de assistente. Porém, apesar desta característica ser do estudante, ela só é necessária para ele no momento em que ele desempenha o papel de assistente, não fazendo sentido em outro escopo. Assim, a interposição de elementos permite que o estudante tenha esta característica somente quando desempenha o papel determinado.

A Figura 36 apresenta a modelagem deste sistema, na linguagem modelada por Balzer, Gross e Eugster (2007). Ela contém as declarações das classes *Student* (linhas 1 a 5) e *Course* (linhas 7 a 9), e da associação entre ambas chamada de *Assists* (linhas 11 a 15). A classe *Student* possui todos os atributos que caracterizam um estudante independentemente de colaboração com outras entidades. A associação *Assists* possui um único atributo (linha 14), declarado com o símbolo

“>” seguido do nome de papel *ta* (*teaching assistant*), significando que este está interposto no referido papel.

Código Modelo Balzer et al

```
1  class Student {
2    String name;
3    int number;
4    int year;
5  }
6
7  class Course {
8    String name;
9  }
10
11 relationship Assists
12 participants (Student ta, Course course) {
13   // atributo interposto no papel ta (teaching assistant)
14   String > ta instructionLanguage;
15 }
```

Figura 36. Código para a associação *Assists* interpondo o atributo *instructionLanguage*.
Fonte: Adaptação de listagens presentes no artigo de Balzer, Gross e Eugster (2007).

Não importando a quantidade de cursos que o estudante preste assistência, ele terá somente uma língua de instrução (atributo interposto *instructionLanguage*). Esta é uma característica do estudante, porém só utilizada quando este assume o papel de assistente (BALZER; GROSS; EUGSTER, 2007).

Quando um estudante freqüenta um curso, ele possui uma nota. Esta nota não caracteriza o estudante, nem o curso, mas a frequência do estudante no determinado curso. É um atributo da frequência (instância da associação) e não das instâncias participantes. A Figura 37 apresenta a modelagem para este caso, na linguagem de Balzer *et al* (BALZER; GROSS; EUGSTER, 2007).

Código Modelo Balzer et al

```
1  relationship Attends
2  participants (Student learner, Course lecture) {
3    // atributo pertencente à associação
4    int mark;
5  }
```

Figura 37. Exemplo de codificação de classe de associação no modelo de Balzer *et al*.
Fonte: Parte de uma ilustração presente no artigo de Balzer, Gross e Eugster (2007).

Interposição de elementos permite a reusabilidade de classes de forma similar aos trabalhos de Pearce e Noble (2006a, 2006b) porque permite que características de uma classe referentes a uma associação sejam declaradas no código da associação. Assim, as classes podem ser reutilizadas em outros modelos onde as associações não estão presentes. Por exemplo: A classe *Student* poderia ser reutilizada em um sistema escolar onde a associação *Assists* não faz sentido (BALZER; GROSS; EUGSTER, 2007).

3.1.3.4.2 Invariantes

O modelo permite declaração de *invariantes* que operam sobre conjuntos (*structural invariants*) e valores (*value-based invariants*). Invariantes que operam sobre conjuntos permitem que funções de conjuntos sejam aplicadas para validação dos mesmos. Invariantes que operam sobre valores permitem a validação de valores de atributos (BALZER; GROSS; EUGSTER, 2007).

A Figura 38 apresenta uma invariante que opera sobre conjuntos. A linha 2 contém a expressão que representa a invariante. Esta indica que o conjunto de tuplas da associação *Attends*, quando interseccionado com o conjunto de tuplas da associação *Assists*, resulta num conjunto vazio (*emptySet*). Isto impede que estudantes prestem assistência a cursos que eles frequentam e vice-versa (BALZER; GROSS; EUGSTER, 2007).

Código Modelo Balzer et al

```
1  invariant (Attends, Assists) attendsAssistsDisjointness:
2    Attends intersection Assists == emptySet;
```

Figura 38. Declaração de uma invariante entre duas associações (*structural inter-relationship invariant*).
Fonte: Balzer, Gross e Eugster (2007).

Exemplificando: se uma instância *john* da classe *Student* está associada a *comp* da classe *Course* por intermédio da associação *Attends,* então *john*, obrigatoriamente, não está associado a *comp* por intermédio da associação *Assists* e vice-versa.

Invariantes como esta que atuam sobre duas ou mais associações são declaradas como *cidadãos de primeira ordem* e são chamadas de *inter-relationship invariants*. Quando atuam somente sobre uma única associação, são declaradas dentro do código desta e chamam-se *intra-relationship invariants* (BALZER; GROSS; EUGSTER, 2007).

A Figura 39 apresenta uma associação da classe *Faculty* com ela própria, contendo a declaração de uma *structural intra-relationship invariant*. A função *surjectiveRelation* (*relação sobrejetora*, localizada na linha 7) verifica se todas as instâncias do segundo participante (papel *substituted*) estão associadas a pelo menos uma instância do primeiro participante (papel *substitute*). A função *asymmetric* (*relação assimétrica*, linha 8) indica que se duas instâncias f_1 e f_2 da classe *Faculty* estão associadas de forma que f_1 é o primeiro participante e f_2 o segundo, então a associação de f_2 como primeiro participante com f_1 como segundo participante é inválida. A função *irreflexive* (*relação irreflexiva*, linha 9) verifica se nenhuma instância de *Faculty* está associada a ela mesma (BALZER; GROSS; EUGSTER, 2007).

Código Modelo Balzer et al

```
1  class Faculty {...}
2
3  relationship Substitutes
4    participants (Faculty substitute, Faculty substituted)
5  {
6    invariant
7      surjectiveRelation(Substitutes) &&
8      asymmetric(Substitutes) &&
9      irreflexive(Substitutes);
10 }
```

Figura 39. Associação com invariantes internas de conjuntos (*structural intra-relationship invariant*).
Fonte: Balzer, Gross e Eugster (2007).

Código Modelo Balzer et al

```
1  relationship WorksFor
2    participants (Student ra, Faculty supervisor) {
3    int > ra grantAmount;
4
5    invariant
6      relation(WorksFor) &&
7      ra.year > 2 &&
8      partialFunction(grantAmount) in N;
9  }
```

Figura 40. Associação contendo uma invariante sobre conjuntos e duas sobre valores.
Fonte: Balzer, Gross e Eugster (2007).

A Figura 40 apresenta duas invariantes que operam sobre valores: A linha 7 formaliza que as instâncias de *Student* só podem estar associadas pela associação *WorksFor* se seu atributo *year* conter valor superior a 2. A linha 8 formaliza que o atributo interposto *grantAmount*

é opcional, através da *função parcial*, e pode conter qualquer número natural.

Classes também podem conter invariantes. Estas têm escopo interno à classe e operam somente em valores de forma similar as *intra-relationship invariants* que operam sobre valores (BALZER; GROSS; EUGSTER, 2007).

Invariantes de associação só são verificadas em instâncias que estejam adicionadas nos conjuntos representados por suas classes. Sendo assim, um objeto recém instanciado e ainda não adicionado a sua classe, não é verificado pelas invariantes de associação (BALZER; GROSS; EUGSTER, 2007).

3.1.3.4.3 Limites de Multiplicidade

Multiplicidade é um subconjunto de *intra-relationship invariants* que operam sobre conjuntos. Associações com limites de multiplicidade *[1..*]* no primeiro participante, podem ser formalizadas com *relação sobrejetora* (Figura 39, linha 7). *Relação total* pode ser utilizada para formalizar limites de multiplicidade *[1..*]* no segundo participante (BALZER; GROSS; EUGSTER, 2007).

Limites de multiplicidade *[0..1]* podem ser formalizados para o primeiro participante através de *relação injetora*, e para o segundo participante através de *relação funcional*. Combinando *relação sobrejetora* com *relação injetora*, obtêm-se multiplicidade *[1..1]* no primeiro participante. O mesmo é possível para o segundo participante, combinando *relação total* com *relação funcional* (BALZER; GROSS; EUGSTER, 2007).

O modelo de Balzer, Gross e Eugster (2007) não indica claramente a existência de uma forma de declaração explícita e direta de multiplicidade. Porém, considerando as estruturas de invariantes apresentas, isto não é complexo de ser implementado nativamente e acrescentado ao modelo.

3.1.3.4.4 Procedimentos Atômicos

Certas operações, quando executadas separadamente, podem provocar violação de invariantes, como por exemplo: violação de multiplicidade mínima quando uma instância recém foi criada. De forma similar a linguagem de Albano *et al*, existe uma forma de agrupar várias operações que possam violar as invariantes, permitindo que ao final do agrupamento, estas sejam testadas para verificação de sua consistência.

Estes agrupamentos são os procedimentos atômicos (BALZER; GROSS; EUGSTER, 2007).

Diferente do mecanismo de transações da linguagem de Albano *et al*, os procedimentos atômicos são implícitos. Cada procedimento da aplicação é um procedimento atômico, não necessitando declará-lo no código de programa (BALZER; GROSS; EUGSTER, 2007).

3.1.3.4.5 Limitações em Relação à Semântica da UML

O modelo, da mesma forma que a linguagem de Albano *et al*, não aborda visibilidade, navegabilidade, *association end ownership,* especialização de associações. Associações e classes são vistas como conjuntos de visibilidade global causando o *problema da visibilidade global* (ver Subseção 3.2.1). *Invariantes* de associações (de forma similar às *constraints* de Albano *et al*) não são testadas em instâncias que não estão contidas por suas classes (BALZER; GROSS; EUGSTER, 2007).

3.2 PROBLEMAS ENCONTRADOS NOS TRABALHOS CORRELATOS

Entre os trabalhos analisados, o principal problema encontrado é a ausência de verificação de limites de multiplicidade, em especial, o limite inferior. Entre os que resolvem este problema, tratando limites inferiores de multiplicidade como invariantes, é possível identificar dois problemas causados por duas situações opostas quanto ao uso de coleções para armazenamento de objetos, contidas em *singletons* (classes e associações): o *problema da visibilidade global* (ver Subseção 3.2.1) e o *problema da referência perdida* (ver Subseção 3.2.2). Estes dois ocorrem de forma exclusiva entre si e sempre em conjunto com um terceiro: o *problema da destruição de objetos obrigatoriamente associados* (ver Subseção 3.2.3).

Trabalho	**Visibilidade Global**	**Referência Perdida**	**Destruição Obj. Obrig. Assoc.**
Linguagem de Albano *et al*	X		X
Modelo de Balzer *et al*	X		X
Pattern de Gessenharter		X	X

Quadro 1. Resultado da análise dos trabalhos quanto à ocorrência dos problemas.

O Quadro 1 apresenta os problemas citados em colunas e os trabalhos que implementam verificação do limite inferior de multiplicidade como invariante, em linhas. Um X no cruzamento entre linha e coluna identifica que o trabalho apresenta o determinado problema.

As subseções a seguir apresentam uma análise detalhada destes problemas.

3.2.1 O Problema da Visibilidade Global

O *problema da visibilidade global* ocorre em coleções globais de objetos que permitem listagem e consulta de elementos através dos valores de seus atributos. Coleções deste tipo permitem o acesso global aos objetos contidos por estas, evitando assim causar o *problema da referência perdida*.

Trabalhos como a linguagem de Albano *et al* (apresentado na Subseção 3.1.3.2) e o modelo de Balzer *et al* (Subseção 3.1.3.4) conceituam classes e associações como conjuntos, e objetos e *links* de associação como elementos destes, permitindo acessar estes elementos de qualquer local do programa. Isto impossibilita a implementação de restrições de acesso, como restrições de navegabilidade e visibilidade, presentes na UML. Estas restrições têm por objetivo encapsulamento de informação, utilizado em padrões de projeto *GRASP*[9].

3.2.2 O Problema da Referência Perdida

Este problema pode ocorrer em programas que utilizam *garbage collector* como gerenciador de memória e coleções restritas como *singletons*. Coleções restritas, em contraste com as coleções que causam o *problema da visibilidade global*, não permitem listagem e nem outro tipo de consulta de elementos que não seja através da comparação de referências dos próprios elementos. Sendo assim, se um objeto *o* está contido em uma coleção restrita *c*, se *c* é um *singleton*, e se todas as demais referências para *o* foram perdidas, não havendo uma forma de recuperar uma destas referências, então *o* estará perdido, porém ainda existente em memória, até o fim da execução do programa.

Trabalhos como o *code pattern* de Gessenharter (apresentado pela Subseção 3.1.1.2) utilizam o *pattern* de *objeto de associação* para o mapeamento de associações. Nestes, um *link* de associação entre *a* e *b*,

[9] Larman, 2000.

armazenado no *singleton* da associação, bem como os objetos *a* e *b*, estarão perdidos em memória se o programa perder todas as demais referências para *a* e para *b*, sem antes remover o *link* de associação de dentro do *singleton*. Isto acarreta perda de memória e postergação da execução dos métodos de destruição destes objetos para o final do programa.

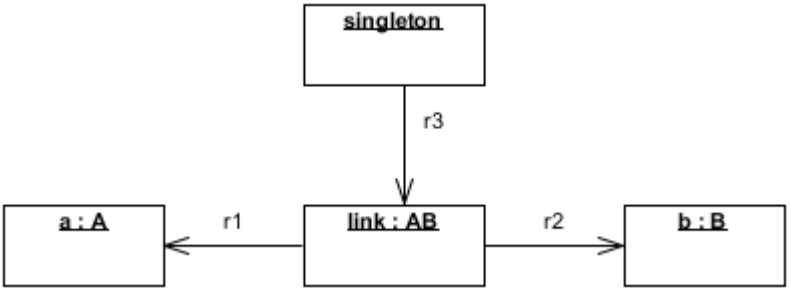

Figura 41. Objetos e suas referências em uma implementação do *pattern objeto de associação*.

O uso de referências fracas poderia resolver este problema, se não causassem outros. Supondo uma associação *AB* entre as classes *A* e *B*, a Figura 41 ilustra os objetos e suas referências em uma implementação do *pattern objeto de associação*. Esta contém a representação dos objetos *a : A* (*a* é instância da classe *A*), *b : B*, *link : AB*, entre *a* e *b*, e o *singleton* (objeto que gerencia a associação). As referências (*r1*, *r2* e *r3*) indicam que o objeto conectado na extremidade que possui a seta é o objeto referenciado pelo objeto conectado na outra extremidade (o objeto que armazena a referência).

Considerando todas as referências como referências fortes, se os objetos *a* e *b* perderem todas as suas referências no programa, mantendo sua *linkagem* um com o outro pelo objeto *link*, então ocorrerá o *problema da referência perdida* para ambos os objetos, incluindo o *link*.

Considerar que *r1* e *r2* são referências fracas possibilita que os objetos a e b sejam coletados quando estes perdem sua última referência no programa. Porém, isto causa o referenciamento fraco entre os objetos *a* e *b*, de modo que a existência do *link* de associação não garante que um possa manter o outro *vivo*. Este é o caso da biblioteca Noiai (apresentado pela Subseção 3.1.2.2.2). Considerar que *r3* é uma referência fraca impedirá que o *link* de associação mantenha-se *vivo*, sendo coletado na primeira interação do *garbage collector*.

Portanto, no uso do *pattern objeto de associação*, o único modo de resolver o *problema da referência perdida* é causando o *problema da visibilidade global*.

3.2.3 O Problema da Destruição de Objetos Obrigatoriamente Associados

Objetos obrigatoriamente associados são objetos que necessitam obrigatoriamente estar associados a pelo menos outro objeto. Em uma associação *AB* entre as classes *A* e *B*, em que é obrigatório que pelo menos uma instância de *B* esteja associada a toda e qualquer instância de *A*, tem-se que as instâncias de *A* são obrigatoriamente associadas a instâncias de *B*.

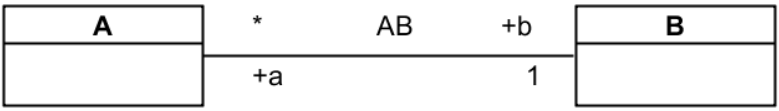

Figura 42. Exemplo de modelo que instancia objetos obrigatoriamente associados (classe A).

Supondo o modelo apresentado pela Figura 42, a_1 e a_2 instâncias de *A*, b_1 uma instância de *B*, a tupla $<a_1,b_1>$ *link* da associação *AB* que faz a *linkagem* entre a_1 e *b1*, tem-se um problema quando, por uma eventual necessidade causada pelas regras de negócio, $<a_1,b_1>$ tenha que ser destruído para a criação do *link* $<a_2,b_1>$ entre a_2 e b_1. Isto é, ocorre um problema quando é necessário substituir o *link* $<a_1,b_1>$ por $<a_2,b_1>$. Neste caso, a_1 estará violando a multiplicidade definida na extremidade de *B* após executar estas operações. Isto poderá ser resolvido criando-se um link entre a_1 e outra instância de *B* recém criada ou destruindo-se a_1.

A segunda opção causa problemas em linguagens que utilizam *garbage collector* porque não há como garantir que a_1 seja coletado e destruído assim que o *link* $<a_1,b_1>$ for destruído. Como a intenção é destruir a_1, de alguma forma deverá ser informado ao mecanismo de verificação de invariantes para tratar a_1 como destruído após a destruição de $<a_1,b_1>$. Porém, o código do programa pode conter operações que removam referências conhecidas para a_1, mas, por erros de programação, pode haver outras referências para esta instância, em outros pontos do programa, desconhecidos ou esquecidos pelo programador. Sendo assim, é possível que a_1, após a destruição de $<a_1,b_1>$, não seja destruído e permaneça em uso pelo programa,

violando as invariantes sem que os mecanismos de validação acusem isto.

No modelo de Balzer et al (ver Subseção 3.1.3.4), supondo existir um *garbage collector* (não há menção sobre a existência deste), a situação descrita está representada pelo código ilustrado na Figura 43. O procedimento de aplicação chamado de *destroyA1* (linhas 5 a 10) executa todas as operações verificando as invariantes somente na sua finalização. Ele destrói o *link* de associação $<a_1,b_1>$ (linha 6), cria o $<a_2,b_1>$ (linha 7), informa aos mecanismos de validação de invariantes que a_1 não será mais monitorado (linha 8) e então tenta destruir a_1, removendo sua (possível) última referência (linha 9).

Código Modelo Balzer et al

```
 1  application Sample {
 2    Object<A> a1, a2;
 3    Object<B> b1;
 4    ...
 5    void destroyA1() {
 6      AB.remove(a1,b1);  //destroi link a1b1.
 7      AB.add(a2,b1);     //cria link a2b1.
 8      A.remove(a1);      //informa que a1 será destruido.
 9      a1 = null;         //tenta destruir a1.
10  } }                    //verifica invariantes.
```

Figura 43. Exemplo de código do modelo de Balzer *et al* para destruição de um objeto

Após a execução do procedimento *destroyA1*, a instância a_1 pode ainda estar sendo referenciada em outro ponto do programa. Esta, apesar de não estar participando em nenhuma associação, ainda permite acesso a seus atributos e métodos.

A linguagem de Albano *et al* utiliza semântica similar ao modelo de Balzer *et al*, diferenciando-se pela sintaxe que é própria da linguagem empregada.

O *code pattern* de Gessenharter não aborda exemplos de violação temporária de limites de multiplicidade, mantendo este problema em aberto.

O *code pattern* de Génova *et al* não trata os limites inferiores de multiplicidade como invariantes, mas como precondições de execução dos métodos de acesso aos *links* de associação. Sendo assim, este problema ocorre não somente na destruição de um objeto obrigatoriamente associado, podendo ocorrer também em qualquer ponto onde a há a remoção de *links* de associação. Porém, diferente das demais abordagens, uma tentativa de consulta de suas associações

levanta uma exceção, alertando sobre o problema, mesmo que em um ponto do programa diferente do ponto onde o problema foi originado.

3.3 ANÁLISE COMPARATIVA DOS TRABALHOS CORRELATOS

Para qualificar os trabalhos correlatos quanto à proximidade com os objetivos deste trabalho, são enumerados abaixo alguns aspectos considerados necessários para atingir estes objetivos:

a) Abstração da construção de associações binárias;
b) Implementação dos mecanismos de verificação do limite inferior de multiplicidade como invariante;
c) Implementação de visibilidade de associações binárias;
d) Implementação de navegabilidade de associações binárias;
e) Possibilidade de escolha do proprietário do *association end*;
f) Possibilidade de especialização de associações;
g) Declaração (ou construção) de classes de associação no mesmo formato de associações;
h) Possibilidade de declaração de nome de papel para cada participante de uma associação.

O Quadro 2 apresenta todos os aspectos em colunas e os trabalhos em linhas. Um X no cruzamento entre linha e coluna identifica que o trabalho possui o determinado aspecto. Algumas observações como suporte incompleto ao aspecto ou limitação, estão apontadas com letras minúsculas e descritas na parte inferior do quadro.

Nenhum dos trabalhos apresenta todas as características. O *code pattern* de Gessenharter é o trabalho que mais se aproxima de uma solução que apresenta todos os aspectos, porém ainda apresenta lacunas. A principal delas é não apresentar o aspecto *a*. Este, em princípio, não pode ser resolvido por um *code pattern* pois este tipo de abordagem significa que todo o código de construção de associações está visível ao programador, sendo necessário escrevê-lo (mesmo que seja por uma ferramenta geradora de código). A construção de classes genéricas para a construção de associações seguindo o *pattern* de Gessenharter, armazenadas em uma biblioteca, poderia ser feita para evitar a necessidade de escrita deste código para cada associação. Entretanto, esta abstração não é completa quando trata-se da construção de *association ends* pertencentes à classes e de *handles* para proteção de métodos.

Assim, para obter uma solução única abordando todos os aspectos mencionados, incluindo o aspecto *a*, uma linguagem de programação é proposta no Capítulo 4.

Trabalho	Abstração de Associações (a)	Multiplicidade Mínima (b)	Visibilidade (c)	Navegabilidade (d)	Association End Ownership (e)	Especialização de Associações (f)	Classe de Associação (g)	Role Name (h)
Linguagem DSM	X	[1]						X
Linguagem de Albano *et al*	X	X [2]				X	X [3]	X
Pattern de Génova *et al*		X [4]	X [4]	X [4]				X
Linguagem RelJ	X					X	X	
Biblioteca RAL	X						X	
Modelo de Balzer *et al*	X	X [2]					X	X
Biblioteca Noiai	X		[5]		[5]	X [5]	X	X [5]
Pattern de Gessenharter		X [6]	X	X	X		X	X

1. DSM não implementa este aspecto apesar de que o modelo *objeto-relação* prevê essa possibilidade sem entrar em detalhes.
2. Limite inferior de multiplicidade é possível de ser implementado utilizando *invariants/constraints*. Porém, estas só são verificadas quando as instâncias estão contidas pelas suas classes.
3. Possui uma estrutura equivalente a classes de associação, porém não é possível acrescentar métodos nesta estrutura.
4. No *code pattern* de Génova *et al*, limite mínimo de multiplicidade não é uma invariante. Visibilidade e navegabilidade apresentam limitações.
5. Noiai permite que os *links* de associação sejam sempre acessados por métodos públicos do próprio *singleton* da associação, inutilizando a declaração de *association ends* nas classes e impedindo restrições de visibilidade. Não há como definir nomes de papel quando acopla nas classes participantes, o objeto de nome de papel. A especialização de associações possui diferenças com a semântica da UML 2.
6. O *pattern* de Gessenharter implementa verificação de limites mínimos de multiplicidade. Porém, a instanciação necessita utilizar o *pattern factory* e não foi apresentada uma forma de violar temporariamente os limites de multiplicidade.

Quadro 2. Resultado da análise dos trabalhos quanto à existência dos aspectos.

4 UMA LINGUAGEM OO COM SUPORTE A ASSOCIAÇÕES

O presente capítulo apresenta a extensão de uma linguagem de programação orientada a objetos, que tem por objetivo acrescentar, como estrutura nativa, associações com a mesma semântica da UML 2, resolvendo os problemas encontrados no estado da arte, abordando multiplicidade, navegabilidade, visibilidade, *association end ownership*, classes de associação e especialização de associações e classes de associação.

Nas próximas seções, são apresentados a proposta da nova linguagem, um exemplo de uso, um teste de desempenho e uma limitação da nova linguagem.

4.1 ASSOCIATION#

A linguagem de programação que foi estendida é C#. Esta linguagem é orientada a objetos, em que classes são tipos que caracterizam os objetos instanciados, e conta com um *garbage collector*.

A nova linguagem, chamada de Association#, possui toda a sintaxe e semântica da linguagem original C# 4.0, preservando o *garbage collector*, acrescentando a sintaxe e semântica das novas estruturas. Estas novas estruturas foram implementadas em forma de classes genéricas (parametrização com uso de *generics*[10]) em C# 4.0 e agrupadas em uma biblioteca de apoio, construída para reduzir a quantidade de código, facilitando a compreensão e o processo de geração de código executável da nova linguagem.

A geração de código executável pode ser feita tanto por uma tradução do código Association# para código da linguagem original, com posterior geração de executável por um compilador C#, quanto por geração direta de código executável através de um compilador para Association#. Em ambos os casos, é necessário adicionar ao código executável do programa, o código executável da biblioteca de apoio. Para melhor compreensão da semântica da nova linguagem, este trabalho apresenta a tradução de código desta para a linguagem original, considerando que o código traduzido pode ser processado por um compilador C# 4.0 compatível com o *framework* .NET 4.0, gerando o código executável.

Nas próximas subseções, são apresentadas: as novas estruturas decorrentes da extensão da linguagem original, a gramática proposta

[10] Microsoft, 2011

para a nova linguagem, a biblioteca de apoio mencionada, e como a nova linguagem resolve os problemas elencados no estado da arte.

4.1.1 As Novas Estruturas

As novas estruturas, inseridas pela extensão da linguagem C#, compreendem: associações binárias, classes de associação e *association ends* (ou *nomes de papéis*). Uma quarta estrutura é necessária para a execução de um conjunto de operações que quando executadas de forma avulsa violam temporariamente a multiplicidade de associações: o *bloco atômico*, uma estrutura similar às transações utilizadas por Albano, Ghelli e Orsini (1991), que será apresentada na Subseção 4.1.1.9.

Para demonstrar as novas estruturas, bem como a exemplificação da tradução desta para C#, serão apresentados exemplos contendo diagramas de classes UML, os respectivos códigos Association# equivalentes a estes e os códigos C# gerados pela tradução dos códigos Association#. Detalhes da tradução para código C#, bem como sobre a biblioteca de apoio, serão apresentados na Subseção 4.1.3

4.1.1.1 Associações Binárias

Em Association#, associações são estruturas que podem ser declaradas em qualquer lugar onde se pode declarar uma classe, desde que as classes participantes tenham visibilidade para estas.

A Figura 44 apresenta um diagrama de classes UML contendo um exemplo de associação binária navegável em ambos os sentidos, com um *association end* pertencente a uma classe (*employer*) e outro pertencente à associação (*employees*).

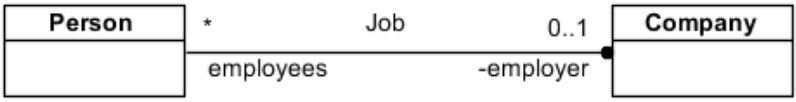

Figura 44. Exemplo de associação navegável em ambos os sentidos, em diagrama de classe UML.

A Figura 45 apresenta o código Association# que corresponde semanticamente ao diagrama de classe da Figura 44. Este apresenta as classes *Person* (linhas 1 a 3) e *Company* (linha 5) associadas pela associação *Job* (linhas 7 a 8). *Person* participa desta associação com o nome de papel *employees* (linha 8) pertencente a associação e sua multiplicidade de papel não possui limites. A classe *Company* participa

com o nome de papel *employer* (linha 2) que pertence a classe *Person*, com limites de multiplicidade de papel entre 0 e 1 (inclusive).

Código Association#

```
1  class Person {
2    private rolename employer[0..1] in Job;
3  }
4
5  class Company { }
6
7  sealed association Job
8    between Person employees[*] and Company;
```

Figura 45. Código *Association#* com um *association end* pertencente à classe e outro a associação.

Os nomes de papel são os nomes dos *association ends* que conectam as classes à associação. Quando este é uma propriedade de uma classe, há a necessidade de indicar qual a associação que está sendo conectada. Isto é feito pela palavra reservada *in*, conforme a linha 2 da Figura 45.

A palavra reservada *sealed*, utilizada na declaração da associação (linha 7), tem a mesma semântica de seu uso na declaração de classes escritas em código C#: não permitir especialização. Assim, a associação *Job* não pode ser especializada. Caso esta palavra reservada seja omitida do código, a associação *Job* passa a ser então passível de ser especializada por outra associação. Estas duas possibilidades também diferenciam o código C# resultante da tradução, conforme será apresentado na Seção 4.1.1.10.

Código C#

```
1  public class Person {
2    public readonly Job.EndForPerson employer;
3    public Person() {
4      this.employer = new Job.EndForPerson(this); } }
5
6  public class Company {
7    public readonly  Job.EndForCompany employees;
8    public Company() {
9      this.employees = new Job.EndForCompany(this); } }
```

Figura 46.Código C# resultante da tradução das classes escritas em Association#, presentes na Figura 45.

A Figura 46 apresenta o código das classes e a Figura 47, o código da associação, ambos resultantes da tradução do código escrito em Association#, apresentado pela Figura 45, para o código C#. Os

association ends, independente de serem propriedades de classe ou de associação no código Association#, são traduzidos para *fields*[11] (linhas 2 e 7 da Figura 46) pertencentes às classes participantes da associação e instanciados nos métodos construtores das instâncias destas classes. Nestes construtores. a instanciação dos *association ends* é feita sempre depois da chamada do construtor da classe *base*[12] e antes de qualquer operação escrita pelo programador no corpo do construtor.

Código C#

```
1  public class Job : Association<Person, Company> {
2    public static readonly Job SINGLETON = new Job();
3
4    private Job() : base(
5      Multiplicity.ZERO_TO_ONE_RANGE,
6      Multiplicity.ZERO_TO_MANY_RANGE,
7      getSourceEnd, getTargetEnd) { }
8    private static EndForPerson
9      getSourceEnd(Person sourceObject) {
10       return sourceObject.employer; }
11   private static EndForCompany
12     getTargetEnd(Company targetObject) {
13       return targetObject.employees; }
14   public class EndForPerson:SourceAssociationEnd {
15     public EndForPerson(Person sourceObject)
16       : base(sourceObject, SINGLETON) { } }
17   public class EndForCompany:TargetAssociationEnd {
18     public EndForCompany(Company targetObject)
19       : base(targetObject, SINGLETON) { } } }
```

Figura 47. Código C# resultante da tradução da associação escrita em Association#, presente na Figura 45.

4.1.1.2 Classes de Associação

A linguagem Association# possibilita também a declaração de classes de associação. Para exemplificar posteriormente a declaração de uma classe de associação, a Figura 48 apresenta o diagrama de classe UML para uma situação em que a associação *Job* da Figura 44 foi substituída por uma classe de associação com o mesmo nome. A esta classe de associação, foram inseridos um atributo (*salary*) e uma operação (*getHoursWorked*).

[11] Microsoft, 2011.
[12] Microsoft, 2011.

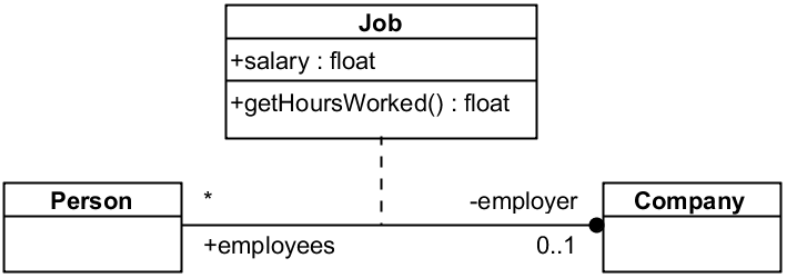

Figura 48. Exemplo de classe de associação em diagrama de classes UML.

Em Association#, classes de associação têm sintaxe similar a associações, diferenciando-se pela inserção da palavra reservada *class* após a palavra reservada *association*, e pela inserção de um corpo na estrutura.

A Figura 49 apresenta o respectivo código Association# para o exemplo da Figura 48. Observa-se a declaração de atributos e operações na classe de associação (linhas 10 e 11) de forma similar a declaração dos mesmos em uma classe.

Código Association#

```
 1  class Person {
 2    private rolename employer[0..1] in Job;
 3  }
 4
 5  class Company { }
 6
 7  sealed association class Job
 8  between Person employees[*] and Company
 9  {
10    public float salary;
11    public float getHoursWorked()  { ... }
12  }
```

Figura 49. Exemplo de código para classe de associação, escrito em Association#.

Instâncias da classe de associação são criadas no momento da criação dos *links* de associações. Isto causa obrigatoriedades como:

a) A classe de associação não pode ter um método construtor de instâncias diferente do construtor sem parâmetros e sem visibilidade declarada;
b) O construtor sem parâmetros da classe de associação não pode ser chamado em ponto algum do código, servindo

apenas para inicializar atributos da instância, quando necessário.

O construtor sem parâmetros da classe de associação é chamado exclusivamente pelas operações que criam *links* de associação (apresentadas na Subseção 4.1.1.7)

O código C#, resultante da tradução do código Association# da Figura 49 é apresentado pela Figura 50, contendo as classes *Person* (linhas 1 a 4) e *Company* (linhas 6 a 10), e a associação *Job* (linhas 12 a 35).

Código C#

```
 1  public class Person {
 2    public readonly Job.Association.EndForPerson employer;
 3    public Person() { this.employer =
 4      new Job.Association.EndForPerson(this); } }
 5
 6  public class Company {
 7    public readonly
 8      Job.Association.EndForCompany employees;
 9    public Company() { this.employees =
10      new Job.Association.EndForCompany(this); } }
11
12  public class Job {
13    public float salary;
14    public float getHoursWorked() {...}
15
16    public class Association :
17      AssociationClass<Person,Company,Job>  {
18        public static readonly Association SINGLETON =
19          new Association();
20        private Association() : base(
21          Multiplicity.ZERO_TO_ONE_RANGE,
22          Multiplicity.ZERO_TO_MANY_RANGE,
23          getSourceEnd, getTargetEnd) { }
24        private static EndForPerson
25          getSourceEnd(Person sourceObject) {
26            return sourceObject.employer; }
27        private static EndForCompany
28          getTargetEnd(Company targetObject)
29            { return targetObject.employees; }
30        public class EndForPerson:SourceAssociationEnd {
31          public EndForPerson(Person sourceObject)
32            : base(sourceObject, SINGLETON) { } }
33        public class EndForCompany:TargetAssociationEnd {
34          public EndForCompany(Company targetObject)
35            : base(targetObject, SINGLETON) { } } } }
```

Figura 50. Código resultante da tradução do código Association# da Figura 49.

4.1.1.3 Association Ends (Role Names)

Em Association#, um *association end* é uma propriedade pertencente a uma classe ou a uma associação. Possui um nome, também chamado de *nome de papel*, e um conjunto de referências para instâncias da *classe atuante do papel* (a classe em que é nomeada pelo nome de papel). Independente do proprietário do *association end* ser uma associação ou uma classe, ele é sempre criado, armazenado e destruído pela *classe oposta ao papel* (a classe oposta à *classe atuante do papel*). Cada instância da *classe oposta ao papel* armazena internamente uma instância do referido *association end*, conforme pode ser observado no código C# da Figura 50 (linhas 4 e 10) e da Figura 46 (linhas 4 e 9).

4.1.1.4 Association End Ownership

As possíveis localizações dos *association ends* no código Association# foram apresentadas na Figura 45 e, similarmente, na Figura 49. O nome de papel *employer* (linha 2 em ambas as figuras) é um *association end* que conecta a classe *Company* (*classe atuante do papel*) à associação *Job*. Por ser um *association end* pertencente à classe e não à associação, ele é uma propriedade da *classe oposta ao papel*, ou seja, a classe *Person*. Assim, *Person* também é a *classe proprietária do association end* de nome *employer*, sendo este, então, uma propriedade da classe *Person*. O nome de papel *employees* (linha 8 em ambas as figuras) é um *association end* navegável pertencente à associação, isto é, *employees* é uma propriedade da associação *Job*. Sua *classe atuante do papel* é *Person* e sua *classe oposta ao papel* é *Company*.

4.1.1.5 Navegabilidade nos Association Ends

A navegabilidade dos *association ends* (nomes de papéis) indica a possibilidade de chamada dos mesmos no código. *Association ends* não navegáveis não podem ser chamados em local algum do código, impossibilitando a navegação do objeto proprietário aos objetos da classe oposta associados.

A Figura 52 apresenta um diagrama de classes como exemplo em que o *association end* que conecta a classe *Person* não é navegável.

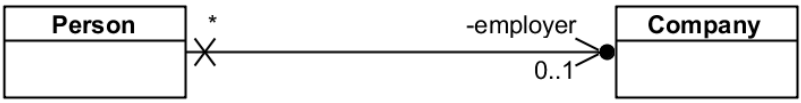

Figura 51. Exemplo em UML para uma associação navegável somente em um dos sentidos

A Figura 52 apresenta o código Association# equivalente ao exemplo em UML apresentado pela Figura 51. Observa-se que o *association end* que tem *Person* como *classe atuante do papel* não tem um nome declarado, caracterizando um *association end* não navegável.

Código Association#

```
1  class Person {
2    private rolename employer[0..1] in Job;
3  }
4
5  class Company { }
6
7  sealed association Job  between Person [*] and Company;
```

Figura 52. Exemplo Association# de associação navegável em apenas um sentido.

Os códigos C#, resultantes de tradução de código Association#, são similares para associações navegáveis em apenas um sentido e para associações navegáveis em ambos os sentidos. A diferença é que em associações navegáveis em apenas um sentido, o *association end* não navegável não tem um nome definido pelo código Association#, mas sim pelo tradutor, podendo ser este aleatório ou então seguir um padrão. Isto porque *association ends* não navegáveis, mesmo não sendo visíveis em código Association#, necessitam ser implementados para que seja possível a verificação dos limites de multiplicidade nos mesmos. Os detalhes sobre verificação dos limites de multiplicidade são abordados na Subseção 4.1.1.8.

Para exemplificar: o código C#, resultante da tradução do código Association# presente na Figura 52, é similar ao código C# composto pela Figura 46 (classes) e pela Figura 47 (associação), diferenciando-se somente pelo nome do atributo *employees* (linha 7 da Figura 46) que, neste caso, teria um nome definido pelo tradutor.

4.1.1.6 Visibilidade nos Association Ends

Association# utiliza os mesmos modificadores de visibilidade presentes em C# (*public*, *private*, *protected* e *internal*), incluindo

package à sua sintaxe e semântica. Com exceção de *internal*, todos os demais modificadores estão presentes na UML e possuem a mesma semântica definida nesta.

Association ends navegáveis pertencentes à associação têm sempre visibilidade pública. *Association ends* pertencentes à classe têm sua visibilidade de acordo com o declarado no código. *Association ends* não navegáveis não são visíveis (nem mesmo para a classe atuante do papel).

4.1.1.7 Executando Operações sobre Associações em Tempo de Execução

Para executar operações em tempo de execução sobre o conjunto de *links* de uma associação, como adicionar, remover e consultar *links*, faz-se necessário executar chamadas a propriedades e/ou operações de um dos *association ends* presentes na associação.

Um *link* entre os objetos a_1 e b_1 é representado em tempo de execução como uma referência de a_1 contida no *association end* armazenado em b_1 e uma referência de b_1 contida no *association end* armazenado em a_1. Cada *association end* armazena essas referências em seu conjunto interno (que é do tipo da *classe atuante do papel*), possibilitando adicionar, remover ou consultar os objetos referenciados.

Utilizando novamente o código da Figura 45, em que o *association end* de nome *employees* é uma propriedade da associação *Job* e o *association end* de nome *employer* é uma propriedade da classe *Person*, a Figura 53 apresenta exemplos, em código Association#, de chamadas para ambos os nomes de papel (*association ends*) declarados neste, adicionando e removendo um *link* de associação entre dois objetos. Estas chamadas estão diferenciadas de acordo com a localização do *association end* (classe ou associação).

Código Association#

```
1  Person p1 = new Person();
2  Company c1 = new Company();
3
4  bool added = p1#employer.Add(c1);
5  bool removed = Job#employees(c1).Remove(p1);
```

Figura 53. Exemplo escrito em Association#, de chamadas aos nomes de papel.

Observa-se que, independente do *association end* utilizado para a operação, esta ocorre em ambos os *association ends* da associação. Assim, a adição de um *link* de associação em um dos *association ends*

causa a adição no outro *association end*, e vice-versa. Similarmente, a remoção em um causa a remoção também no outro.

Association ends pertencentes a classes são chamados conforme exemplificado na linha 4. A linha 5 exemplifica a chamada de *Association ends* pertencentes a associações.

Para haver uma melhor distinção em código, chamadas a atributos e operações são feitas por intermédio do símbolo "." (ponto) enquanto nomes de papéis são chamados por intermédio do símbolo "*#*" (sustenido). Isto é necessário para que não seja permitido chamar nomes de papéis sem haver a chamada para uma de suas operações (descritas adiante, nesta subseção). Assim, não há como atribuir um *association end* em uma variável, mas somente o resultando (retorno) de uma de suas operações.

A Figura 54 apresenta do código ilustrado pela Figura 53, traduzido para código C#.

Código C#

```
1  Person p1 = new Person();
2  Company c1 = new Company();
3
4  bool added = p1.employer.Add(c1);
5  bool removed = c1.employees.Remove(p1);
```

Figura 54. Código C# traduzido do código Association# da Figura 53.

Observa-se que não há mudança alguma na execução de ambas as formas de posse. A diferença é que, no código Association#, um *association end* pertencente à classe é uma propriedade da classe, causando o acoplamento da associação a esta, enquanto o *association end* pertencente à associação é uma propriedade da associação, não causando o acoplamento desta à classe. Maiores detalhes sobre esta discussão são apresentados na Subseção 5.2.3.

Em Association#, os *association ends*, independente de serem propriedades de classes ou associações, possuem um conjunto pré-determinado de operações (métodos) e propriedades:

a) Operações simples de modificação do conjunto de *links* de associação – *Add*, *Remove* e *RemoveAll;*
b) Operações complexas de modificação do conjunto de *links* de associação – as três variações de *Swap*, apresentadas em detalhes na Subseção 4.1.1.13;
c) Operações de consulta de *links* – *Get* e *Contains*;

d) Atributos e operações informativos – *Count* (cardinalidade), *multiplicity* e *IsValid* (se a cardinalidade é válida de acordo com os limites de multiplicidade).

Em classes de associação, os *association ends* possuem, além das operações e propriedades mencionadas, duas operações de consulta de instâncias da classe de associação: *GetAssociationLink* sem parâmetros e com um parâmetro do tipo *OPPOSITE*.

A Figura 55 apresenta vários exemplos de chamadas de operações sobre associações, em código Association#.

Código Association#

```
 1  public class Program {
 2    public static void Main(string[] args) {
 3      Person p1 = new Person();
 4      Company c1 = new Company();
 5      Company c2 = new Company();
 6      Company c3 = new Company();
 7
 8      bool added = p1#employer.Add(c1);
 9      IEnumerable<Company> companies = p1#employer.Get();
10      bool contains = p1#employer.Contains(c1);
11      int cardinality = p1#employer.Count;
12      Multiplicity multiplicity = p1#employer.multiplicity;
13      unit limiteMinimo = multiplicity. minimumBound;
14      uint limiteMaximo = multiplicity. maximumBound;
15      bool removed = p1#employer.Remove(c1);
16
17      Job#employees(c1).Add(p1);
18      Job#employees(c2).Add(p1);
19      Job#employees(c3).Add(p1);
20      p1#employer.RemoveAll();  } }
```

Figura 55. Exemplo de operações sobre a associação codificada pela Figura 45.

A operação *Add* (linhas 8, 17, 18 e 19), recebe como argumento a instância a ser adicionada ao *association end* que executou a operação e retorna um *bool* indicando que a criação do *link* foi bem sucedida. Caso a instância passada como argumento já tenha sido adicionada anteriormente, retorna *false*, não incrementando a cardinalidade. Caso contrário, adiciona, incrementa a cardinalidade e retorna *true*. Em classes de associação, em vez de retornar *true*, retorna a recém criada instância da classe de associação, e em vez de *false*, retorna *null*, indicando que não houve mudança de cardinalidade.

A operação *Remove* (linha 15), recebe como argumento a instância a ser removida do *association end*. Retorna um *bool* indicando se a remoção foi bem sucedida. Caso a instância não esteja sendo referenciada pelo *association end*, retorna *false*, não decrementando a

cardinalidade. Caso contrário, decrementa a cardinalidade e retorna *true*, removendo a referência para a instância passada como argumento. A operação *RemoveAll* (linha 20) remove todas as referências contidas no *association end*.

A operação *Get* (linha 9) retorna um *IEnumerable<PLAYER>*, onde *PLAYER* é substituido pela *classe atuante do papel*, contendo todos os objetos referenciados pelo *association end*. *Contains* (linha 10) retorna um *bool* indicando se o objeto passado como argumento está contido no *association end* (*true* para sim, *false* para não). A propriedade *Count* (linha 11) informa a cardinalidade (quantidade de objetos da *classe atuante do papel* referenciados) e *multiplicity* (linha 12) informa os limites de multiplicidade do *association end*. Detalhes sobre a consulta aos limites de multiplicidade serão abordados na Subseção 4.1.1.8.

A operação *GetAssociationLink* sem argumentos retorna um *IEnumerable<LINK>*, onde *LINK* é substituído pela classe de associação, contendo todos as instâncias desta. A operação de mesmo nome com passagem de um argumento do tipo *OPPOSITE*, onde *OPPOSITE* é a classe atuante do papel, retorna a instância da classe de associação referente ao *link* de associação entre o objeto que armazena o *association end* e o objeto passado como argumento. Caso não exista um *link* entre ambos, retorna *null*.

Na UML, *association ends* não são objetos, mas sim propriedades que permitem uma instância acessar outras que estejam associadas a esta. Sendo assim, não é permitido fazer chamada a um nome de papel sem haver a chamada de uma operação deste (*Add*, R*emove*, G*et* etc). Em Association#, a chamada apenas do nome de papel, sem chamada de operação, faz chamada por padrão à operação *Get*, conforme ilustra a Figura 56. Neste exemplo, a linha 1 apresenta a chamada apenas do nome de papel, que por padrão equivale a chamada da operação *Get*, apresentada na linha 2.

Código Association#

```
1  IEnumerable<Company> employer = p1#employer;
2  IEnumerable<Company> employer = p1#employer.Get();
```

Figura 56. Semântica de chamada de nome de papel sem chamada para uma operação do mesmo.

4.1.1.8 Multiplicidade

No código Association#, em *association ends* navegáveis, a declaração dos limites de multiplicidade é feita logo após a declaração do nome de papel, independente desta ser no corpo da classe ou na associação. A Figura 45 apresenta um exemplo nas linhas 2 e outro na linha 8. Em *association ends* não navegáveis, os limites de multiplicidade são declarados no corpo da associação, logo após o nome da classe atuante do papel, conforme apresentando na linha 7 da Figura 52.

4.1.1.8.1 Verificação dos Limites de Multiplicidade

A verificação dos limites de multiplicidade ocorre sempre quando há modificação na quantidade de elementos nos conjuntos de *links* de associação. Por padrão, as operações de inserção e remoção de *links*, presentes nos *association ends*, sempre operam respeitando os limites de multiplicidade destes (definidos em conjunto com o *association end*). Não é possível executar com êxito estas operações quando elas violam os limites de multiplicidade. Sempre que uma destas tentativas for feita, uma exceção é levantada.

Conforme apresentado na Subseção 4.1.1.1, em código C# resultante de tradução, os *association ends* são objetos instanciados no início dos construtores de instâncias das classes participantes da associação. Os limites de multiplicidade de um *association end* também são testados no momento da instanciação deste. Isto infere que ao instanciar um objeto de uma classe participante de uma ou mais associações, os limites de multiplicidade de seus *association ends* são verificados.

Se a classe oposta possuir limite inferior de multiplicidade maior ou igual a 1 (caracterizando que o objeto recém instanciado é obrigatoriamente associado), uma exceção é levantada imediatamente devido à violação da multiplicidade.

Para este caso, bem como para qualquer outro que seja necessário violar temporariamente os limites de multiplicidade, é necessário desabilitar o mecanismo de verificação, realizar as operações necessárias, e então, reabilitar o mecanismo. No exemplo, as operações necessárias são: criar a instância da classe e associá-la às devidas instâncias da classe oposta.

Ao reabilitar o mecanismo de verificação, necessita-se validar a multiplicidade dos *association ends* que sofreram modificação de

cardinalidade enquanto o mecanismo estava desabilitado. Isto garante que após a reabilitação, os *association ends* não estejam em estado de erro perante seus limites de multiplicidade. Caso estejam, uma exceção é levantada, interrompendo a *thread* corrente.

A desabilitação e posterior reabilitação dos mecanismos de verificação de multiplicidade é feita por *blocos atômicos* (estrutura apresentada na Subseção 4.1.1.9).

Objetos obrigatoriamente associados necessitam de um tratamento diferenciado não somente no momento da instanciação de objetos e execução de operações sobre *links* de associação, mas também na destruição de objetos. A Subseção 4.1.1.14 aborda em detalhes a destruição de objetos obrigatoriamente associados.

4.1.1.9 Bloco Atômico

A estrutura responsável por desabilitar o mecanismo de verificação de multiplicidade, executar as operações, habilitar novamente e validar as operações executadas, é chamada de *bloco atômico*.

O bloco atômico é um bloco de operações demarcado por um corpo (espaço demarcado entre *{* e *}*) que é precedido pela palavra reservada *atomic*. Este só pode ser declarado no corpo de métodos ou no cabeçalho destes.

A Figura 57 apresenta um exemplo em código Association# para a declaração do bloco atômico (linhas 3 a 6) dentro de um método.

Código Association#

```
1  class Program {
2    static final Main(string[] args) {
3      atomic {
4        p1#employer.Remove(c1);
5        p1#employer.Add(c2);
6      }
7      IEnumerable<Company> employer = p1#employer.Get();
8  } }
```

Figura 57. Exemplo de bloco atômico codificado em Association#.

O código C#, resultante da tradução do código Association# apresentado pela Figura 57, está presente na Figura 58. A tradução do bloco atômico com as operações executadas dentro dele está compreendido entre as linhas 3 e 12 (bloco *try...finally...*).

Código C#

```
1  class Program {
2    static final Main(string[] args) {
3      try
4      {
5        AssociationSharpSystem.__DisableSystemConstraints();
6
7        p1.employer.Remove(c1);
8        p1.employer.Add(c2);
9
10     } finally {
11       AssociationSharpSystem.__EnableSystemConstraints();
12     }
13     IEnumerable<Company> employer = p1#employer.Get();
14 } }
```

Figura 58. Tradução do código Association# da Figura 57

Quando declarado no cabeçalho de um método, significa que todo o corpo do método é um bloco atômico.

A Figura 59 apresenta um exemplo em que as duas abordagens são utilizadas. No método de nome *method1* (linhas 2 a 6), é utilizado o bloco atômico dentro do corpo do método. No método de nome *method2* (linhas 8 a 12), o bloco atômico é o próprio corpo do método.

Código Association#

```
1  class A {
2    void method1(){
3      atomic {
4        p1#employer.Remove(c1);
5        p1#employer.Add(c2);
6    } }
7
8    atomic void method2() {
9      p1#employer.Remove(c1);
10     p1#employer.Add(c2);
11     IEnumerable<Company> employer = p1#employer.Get();
12 } }
```

Figura 59. Exemplo de declarações do bloco atômico.

Exceto no caso de métodos construtores, declarar o bloco atômico no cabeçalho ou dentro do corpo de um método, não causa diferenças de execução quando, no segundo caso, o bloco atômico engloba todas as operações que estão contidas no corpo do método. Isto é demonstrado pela Figura 60 que apresenta a tradução para código C# do código Association# ilustrado pela Figura 59. Observa-se que o método de nome *method2* (linhas 10 a 16) é traduzido exatamente como o método de nome *method1* (linhas 2 a 8).

Código C#

```
1  class A {
2    void method1(){
3      try {
4        AssociationSharpSystem.__DisableSystemConstraints();
5        // operações
6      } finally {
7        AssociationSharpSystem.__EnableSystemConstraints();
8    } }
9
10   void method2() {
11     try {
12       AssociationSharpSystem.__DisableSystemConstraints();
13       // operações
14     } finally {
15       AssociationSharpSystem.__EnableSystemConstraints();
16 } } }
```

Figura 60. Código de implementação em C# do exemplo ilustrado pela Figura 59.

No caso de métodos construtores, existe uma diferença entre declarar o bloco atômico no cabeçalho do método e declará-lo no corpo do mesmo abrangendo todas as operações. No primeiro caso, todo o código existente dentro do corpo construtor, incluindo a instanciação dos *association ends* (aparente somente no código C# resultante da tradução), estará dentro do bloco atômico, significando que os limites de multiplicidade destes *association ends* não serão verificados até o fim da execução do bloco. No segundo, somente o código Association# escrito pelo programador no corpo do construtor será afetado pelo bloco atômico, indicando que a instanciação dos *association ends* ocorrerá com a verificação imediata dos limites de multiplicidade.

Para exemplificar esta diferença e a sua aplicabilidade, a Figura 61 apresenta um exemplo para ambos os casos. A declaração do bloco atômico no corpo do método é exemplificada pelo único construtor da classe *A* (linhas 3 a 6), enquanto o construtor parametrizado da classe *B* (linhas 11 a 13) exemplifica a declaração do bloco atômico no cabeçalho.

O construtor parametrizado da classe *B* (linhas 11 a 13) necessita que o bloco atômico seja declarado no cabeçalho do construtor porque uma instância de *B* não pode existir sem estar *linkado* a uma instância de *A*, devido à multiplicidade declarada no *association end* de nome *a* (linha 9). O construtor da classe *A* (linhas 3 a 6) não tem esta necessidade, pois somente necessita executar o bloco atômico no momento em que uma de suas instâncias instanciar um objeto da classe *B* e associá-lo a si próprio (linha 5).

Código Association#

```
1  public class A {
2     public rolename bs[*] in AB;
3     public A() {
4        atomic { // início do bloco atômico
5           this#bs.Add(new B());
6  } } }
7
8  public class B {
9     public rolename a[1] in AB;
10    B() { }
11    public atomic B(A a) { // início do bloco atômico
12       this#a.Add(a);
13 } }
14
15 selead association ABbetween A and B;
```

Figura 61. Exemplo de declarações do bloco atômico em construtores.

Código C#

```
1  class A {
2     readonly AB.EndForA bs;
3     A() {
4        this.bs = new AB.EndForA(this);
5        try { // início do bloco atômico.
6           AssociationSharpSystem.__DisableSystemConstraints();
7
8           this.bs.Add(new B());
9        } finally {
10          AssociationSharpSystem.__EnableSystemConstraints();
11 } } }
12
13 class B {
14    readonly AB.EndForB a;
15    B() { }
16    B(A a) {
17       try { // início do bloco atômico
18          this.a = new AB.EndForB(this);
19          AssociationSharpSystem.__DisableSystemConstraints();
20       this.a.Add(a);
21       } finally {
22          AssociationSharpSystem.__EnableSystemConstraints();
23 } } }
24
25 class AB  : Association<A,B> {...}
```

Figura 62. Código parcial de implementação do exemplo ilustrado pela Figura 61.

A Figura 62 apresenta a implementação do código apresentado pela Figura 61. Observa-se na Figura 62 que o construtor que declara o bloco atômico no corpo (linhas 3 a 11) primeiro instancia o *association*

end de nome *bs* (linha 4) para depois executar o código escrito pelo programador. Neste caso, o bloco atômico (linhas 5 a 11) somente é executado após a instanciação do *association end* de nome *bs*. Considerando que não se está executando um bloco atômico externo ao construtor, os limites de multiplicidade do *association end* de nome *bs* são verificados na sua instanciação. Se o limite inferior de multiplicidade deste fosse maior do que zero, uma exceção seria levantada imediatamente por não haver ainda *links* de associação para o mesmo. Este seria o caso do *association end* de nome *a* (linha 18), pertencente à classe *B*, caso o mesmo não fosse instanciado dentro de um bloco atômico.

Como o *association end* de nome *a* possui uma obrigatoriedade de ter um *link* de associação, ele necessita ser instanciado (linha 18 da Figura 62) dentro um bloco atômico para que seus limites de multiplicidades não sejam testados imediatamente, sendo postergados para o final do bloco.

Em código Association#, um bloco atômico declarado no cabeçalho de um método construtor afeta a instanciação dos *association ends* daquela classe. Em código C# resultante de tradução, estes são instanciados dentro do bloco atômico que engloba todo o corpo do construtor, conforme ilustrado pelas linhas 16 a 23 da Figura 62.

4.1.1.9.1 Aninhamento de Blocos Atômicos

Em Association#, é possível aninhar vários blocos atômicos um dentro de outro. A cada entrada em um bloco atômico, incrementa-se o *nível de profundidade da atomicidade*, sendo este decrementado a cada finalização de bloco. Cada bloco atômico executado é responsável por validar as modificações ocorridas em seu interior, quanto aos conjuntos de referências para instâncias da *classe atuante do papel* armazenadas nos *association ends*. Conjuntos modificados em um bloco atômico mais externo, ou seja, em um nível de profundidade da atomicidade menor que o nível corrente, não são validados pelo bloco atômico corrente, mas sim pelo referido bloco atômico mais externo ao ser finalizado. Isto impede que o bloco atômico mais interno considere inválido um *association end* que violou os limites de multiplicidade no bloco atômico mais externo (e que ainda não foi finalizado).

Código Association#

```
1  // nível de profundidade da atomicidade: 0
2  Person    john = new Person(),
3            paul = new Person();
4  Company   labs = new Person(),
5            inc = new Person();
6  atomic {
7     // nível de profundidade da atomicidade: 1
8     john#employer.Add(labs);
9     atomic {
10       // nível de profundidade da atomicidade: 2
11       paul#employer.Add(labs);
12       john#employer.Add(inc);
13    }
14    // nível de profundidade da atomicidade = 1
15 }
16 // nível de profundidade da atomicidade = 0
```

Figura 63. Exemplo de blocos atômicos em cascata.

Código C#

```
1  // nível de profundidade da atomicidade: 0
2  Person    john = new Person(),
3            paul = new Person();
4  Company   labs = new Person(),
5            inc = new Person();
6  try {
7     AssociationSharpSystem.__DisableSystemConstraints();
8     // nível de profundidade da atomicidade: 1
9     john.employer.Add(labs);
10    try {
11       AssociationSharpSystem.__DisableSystemConstraints();
12       // nível de profundidade da atomicidade: 2
13       paul.employer.Add(labs);
14       john.employer.Add(inc);
15
16    } finally {
17       AssociationSharpSystem.__EnableSystemConstraints();
18       // nível de profundidade da atomicidade = 1
19    }
20 } finally {
21    AssociationSharpSystem.__EnableSystemConstraints();
22    // nível de profundidade da atomicidade = 0
23 }
```

Figura 64. Tradução do código Association# da Figura 63.

A Figura 63 apresenta um exemplo com dois blocos atômicos aninhados. Os comentários presentes no código (linhas 1, 7, 10, 14 e 16) indicam o nível de profundidade da atomicidade daquele ponto em diante. Neste exemplo, os *association ends* pertencentes aos objetos

john e *labs*, modificam pela primeira vez seus conjuntos de *links* de associação dentro do bloco externo (linha 6). Portanto, serão validados somente no final do bloco externo (linha 15), mesmo havendo outras modificações nestes *association ends* no bloco interno (linhas 9 e 13). Os *association ends* dos objetos *paul* e *inc* são modificados somente dentro do bloco atômico interno (linhas 11 e 12) e portanto, são validados no final do bloco interno (linha 13).

A Figura 64 apresenta o código C# resultante da tradução do código Association# presente na Figura 63.

4.1.1.10 Generalização e Especialização de Associações

A generalização e especialização de associações em Association# têm a mesma semântica da generalização e especialização de associações na UML.

Para exemplificar uma generalização e especialização, a Figura 65 apresenta um exemplo em UML, em que *Job* – uma associação entre as classes *Person* e *Company* – é especializada por *Training* – uma associação entre as classes *Student* e *Company*, onde *Student* é uma classe especialista de *Person*.

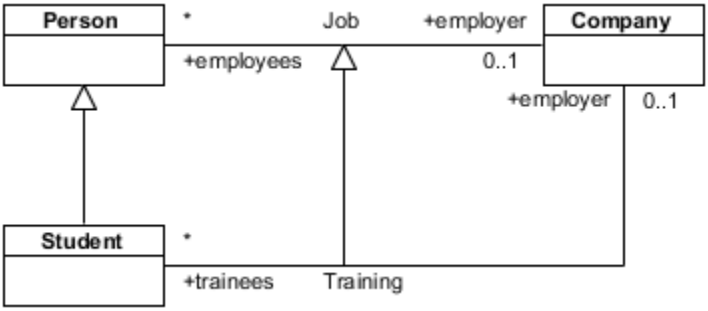

Figura 65. Exemplo UML de generalização de associação.

Conforme exposto na Subseção 4.1.1.1, para que seja possível especializar uma associação é necessário que ela não seja declarada com a palavra reservada *sealed*. Esta condição é similar a condição para a especialização de classes em C#: classes declaradas com *sealed* não podem ser especializadas.

A especialização de associação tem sintaxe análoga à especialização de classe em C#: declara-se uma associação normalmente, adicionando após o nome da associação que está sendo

declarada, o símbolo ":" (dois pontos) seguido do nome da associação que está sendo especializada.

A Figura 66 apresenta o código Association# que representa o modelo ilustrado pela Figura 65. Nela constam as classes *Person* (linha 1) e *Company* (linha 3), a classe *Student* (linha 5) especialista da classe *Person*, a associação generalista *Job* (linhas 7 a 8) e sua associação especialista *Training* (linhas 10 a 11).

Código Association#

```
 1  class Person { }
 2
 3  class Company { }
 4
 5  class Student : Person { }
 6
 7  association Job
 8    between Person employees[*] and Company employer[0..1];
 9
10  association Training : Job
11    between Student trainees[*] and Company employer[0..1];
```

Figura 66. Código Association# para a especialização de associações modelada pela Figura 65.

A Figura 67 apresenta o código das classes *Person*, *Company* e *Student* traduzidos do código Association# da Figura 66 para código C#. A implementação das associações *Job* e *Training* são apresentadas pela Figura 68 (linhas 1 a 20 e 22 a 47, respectivamente).

Código C#

```
 1  public class Person {
 2     public readonly Job.EndForPerson employer;
 3     public Person() {
 4        this.employer = new Job.EndForPerson(this); }  }
 5
 6  public class Company {
 7     public readonly Job.EndForCompany employees;
 8     public readonly Training.EndForCompany trainees;
 9     public Company() {
10        this.employees = new Job.EndForCompany(this);
11        this.trainees =
12          new Training.EndForCompany(this.employees); }  }
13
14  public class Student : Person {
15     public readonly new Training.EndForStudent employer;
16     public Student() {
17        this.employer =
18          new Training.EndForStudent(base.employer); } }
```

Figura 67. Código resultante da tradução das classes da Figura 66.

Código C#

```
public class Job :GeneralizedAssociation<Person,Company> {
   private static readonly Job SINGLETON = new Job();
   private Job() : base(
      Multiplicity.ZERO_TO_ONE_RANGE,
      Multiplicity.ZERO_TO_MANY_RANGE,
      getSourceEnd,getTargetEnd) { }
   private static EndForPerson
      getSourceEnd(Person sourceObject) {
         return sourceObject.employer; }
   private static EndForCompany
      getTargetEnd(Company targetObject) {
         return targetObject.employees; }
   public class EndForPerson
      : SourceGeneralizedAssociationEnd  {
         public EndForPerson(Person sourceObject)
            : base(sourceObject, SINGLETON) { } }
   public class EndForCompany
      : TargetGeneralizedAssociationEnd {
         public EndForCompany(Company targetObject)
            : base(targetObject, SINGLETON) { } } }

public class Training
   :SpecializedAssociation<Student,Company,Person,Company>
{
   private static Training SINGLETON = new Training();
   private Training() : base(
      Multiplicity.ZERO_TO_ONE_RANGE,
      Multiplicity.ZERO_TO_MANY_RANGE,
      sourceEndFinder, targetEndFinder) { }
   public class EndForStudent
      : SpecializedSourceAssociationEnd {
         public EndForStudent(
            GeneralizedAssociationEnd<Person, Company>
               generalizedEnd)
            : base(generalizedEnd, SINGLETON) { } }
   public class EndForCompany
      : SpecializedTargetAssociationEnd {
         public EndForCompany(
            GeneralizedAssociationEnd<Company, Person>
               generalizedEnd)
            : base(generalizedEnd, SINGLETON) { } }
   private static EndForStudent
      sourceEndFinder(Student sourceObject) {
         return sourceObject.employer; }
   private static EndForCompany
      targetEndFinder(Company targetObject) {
         return targetObject.trainees; } }
```

Figura 68. Código resultante da tradução das associações da Figura 66.

Observa-se que o não uso do modificador *sealed* na declaração da associação *Job* fez com que, na tradução para código C#, a classe que a

mapeia especializasse a classe *GeneralizedAssociation<SOURCE, TARGET>* em vez *Association<SOURCE, TARGET>*. A associação *Training*, por ser uma associação que especializa *Job*, independente de utilizar ou não o modificador *selead*, é mapeada em código C# como uma classe que especializa *SpecializedAssociation<SOURCE, TARGET, G_SOURCE, G_TARGET>*. Caso houvesse uma associação que especializasse Training, esta nova associação também seria mapeada em código C# como uma classe que especializa *SpecializedAssociation <SOURCE, TARGET, G_SOURCE, G_TARGET>*, Maiores detalhes sobre estes mapeamentos são apresentados na Subseção 4.1.3.5.

4.1.1.10.1 Generalização e Especialização de Classes de Associação

A generalização/especialização de classes de associação é feita de forma análoga a generalização/especialização de associações, conforme pode ser observado pela Figura 69. Esta apresenta o mesmo exemplo da Figura 66, porém com a diferença de que *Job* e *Training* são classes de associação, em vez de associações.

Código Association#

```
1  class Person { }
2
3  class Company { }
4
5  class Student : Person { }
6
7  association class Job
8    between Person employees[*] and Company employer[0..1]
9  {
10   public float salary;
11   public float getHoursWorked() {...}
12 }
13
14 association class Training : Job
15   between Student trainees[*] and Company employer[0..1]
16 {
17   public float hoursForTraining;
18 }
```

Figura 69. Exemplo de generalização de classes de associação em Association#

A Figura 70 apresenta o código C# resultante da tradução do código Association# presente na Figura 69.

Código C#

```
// classe de associação Job.
public class Job {
   // atributos e métodos.
   public float salary;
   public float getHoursWorked() {...}
   // mecanismos para gerenciar a classe de associação Job.
   public class Association
      : GeneralizedAssociationClass<Person,Company,Job>
        {...} }

// classe de associação Training especialista de Job.
public class Training : Job {
   // atributos e métodos.
   public float hoursForTraining;
   // mecanismos para gerenciar a classe de associação.
   public class Association : SpecializedAssociationClass
      <Student, Company, Training, Person, Company, Job>
      {...} }
```

Figura 70. Implementação de generalização e especialização de classes de associação.

4.1.1.10.2 Condições para Generalização/Especialização de Associações

A especialização de associação requer que, além da ausência da palavra reservada *sealed* na declaração de associação generalista, algumas outras condições sejam atendidas:

a) A primeira classe participante da associação especialista deve ser também a primeira classe participante da associação generalista ou ser especialização desta;
b) A segunda classe participante da associação especialista, similar a primeira classe, deve ser também a segunda classe participante da associação generalista ou ser especialista desta;
c) Os limites superiores de multiplicidade de ambos os participantes da associação especialista não podem exceder os limites superiores de multiplicidade dos respectivos participantes da associação generalista.

As condições *a* e *b* estão relacionadas com a especificação UML. A condição *c* está relacionada com o fato de que toda a inserção nos *association ends* especialistas causam inserção nos *association ends* generalistas, ou seja, todo *link* criado da associação especialista é também um *link* da associação generalista. Isto permite deduzir que é

impossível haver mais *links* da associação especialista do que *links* da associação generalista. O inverso, porém, não é verdadeiro.

Com relação aos limites mínimos de multiplicidade, não existe condição análoga aos limites máximos porque é possível criar *links* da associação generalista entre as instâncias participantes da associação especialista, sem criar *links* desta. Sendo assim, é possível estabelecer limites inferiores de multiplicidade na associação generalista, com grau maior que os respectivos limites inferiores de multiplicidade da associação especialista.

Código Association#

```
 1  association Job
 2    between Person employees[1..*]
 3    and Company employer[0..1];
 4
 5  association Training : Job
 6    between Student trainees[*] and Company employer[0..1];
 7
 8  public class Program {
 9    public void Main(string[] args)
10    {
11      Student s1;
12      Company c1;
13      atomic {
14        s1 = new Student();
15        c1 = new Company();
16        Job#employees(c1).Add(s1)
17  } } }
```

Figura 71. Criação de *link* da associação generalista entre instâncias associadas pela associação especialista.

A Figura 71 ilustra esta situação, onde a associação *Job* (linhas 1 a 3) foi alterada para que o *association end* de nome *employees* (linha 2) tenha o limite inferior de multiplicidade igual a 1, porém o *association end* de nome *trainees* (linha 6) da associação *Training* (linhas 5 a 6), manteve o limite inferior igual a 0. A linha 16 apresenta a operação que cria o *link* da associação *Job* entre as instâncias *s1* e *c1*, sem criar *link* da associação especialista *Training*.

4.1.1.10.3 Executando Operações sobre Associações Generalizadas e Especializadas

A semântica do sistema de generalização e especialização baseia-se nas seguintes regras:

a) Inserção de referência no *association end* especializado causa a inserção da mesma referência no generalizado;
b) Remoção de referência no *association end* especializado causa remoção da mesma referência no generalizado;
c) Remoção de referência no *association end* generalizado, em que a referência se encontra também no especializado, causa remoção em ambos;
d) No caso de Classes de Associação, as regras anteriores se aplicam tanto para os objetos das classes que estão associadas quanto para os objetos da classe de associação;
e) Verificação da multiplicidade no *association end* especializado executa também uma verificação de multiplicidade no generalizado;
f) Instâncias da classe de associação especialista são também instâncias da classe de associação generalista.

4.1.1.11 Associação de Interface e Classes Abstratas

Na linguagem Association#, em uma associação ou classe de associação, a participação de interfaces não é diferente da participação de classes. Para exemplificar esta semelhança, a Figura 72 apresenta um exemplo UML de associação, onde uma interface (*Flying*) participa de uma associação (*FlyingToWing*) e é *implementada*[13] por duas classes concretas (*Bird* e *Bat*), e o respectivo código Association#, apresentado na Figura 73 com uma variação da posse dos *association ends* na Figura 74.

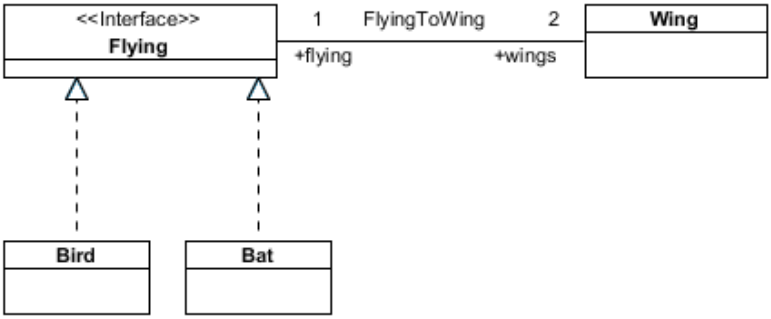

Figura 72. Exemplo de interface participando de uma associação.

[13] Object Management Group, 2010.

Código Association#

```
1  interface Flying { }
2  class Wing { }
3  class Bird : Flying { }
4  class Bat : Flying { }
5
6  association FlyingToWing
7    between Flying flying[1] and Wing wings[2];
```

Figura 73. Participação de interfaces em associações, com *association ends* pertencentes à associação.

Código Association#

```
1  interface Flying {
2    rolename wings[2] in FlyingToWing;
3  }
4
5  class Wing {
6    public rolename flying[1] in FlyingToWing;
7  }
8
9  class Bird : Flying { }
10 class Bat : Flying { }
11
12 association FlyingToWing  between Flying and Wing;
```

Figura 74. Participação de interfaces em associações, com *association ends* pertencentes às interfaces.

Código C#

```
1  public interface Flying {
2    FlyingToWing.EndForFlying wings { get; } }
3  public class Wing {
4    public readonly FlyingToWing.EndForWing flying;
5    public Wing() {
6      this.flying = new FlyingToWing.EndForWing(this); } }
7  public class Bird : Flying {
8    public FlyingToWing.EndForFlying wings
9      { private set; get; }
10   public Bird() {
11     this.wings = new FlyingToWing.EndForFlying(this); } }
12 public class Bat : Flying {
13   public FlyingToWing.EndForFlying wings
14     { private set; get; }
15   public Bat() {
16     this.wings = new FlyingToWing.EndForFlying(this); } }
```

Figura 75. Código resultante da tradução das interfaces e classes da Figura 73.

A Figura 75 apresenta a interface e classes, a Figura 76 a associação, ambas em código C# resultante da tradução do código Association# apresentado pela Figura 73. Os mesmos também podem

ser obtidos pela tradução do código Association# da Figura 74 devido à invariação do código de implementação quanto a posse dos *association ends*.

Código C#

```
1  public class FlyingToWing : Association<Flying,Wing> {
2    private static FlyingToWing SINGLETON =
3      new FlyingToWing();
4    private FlyingToWing() : base(
5      new Multiplicity(2),
6      Multiplicity.ONE_TO_ONE_RANGE,
7      sourceEndFinder, targetEndFinder) { }
8    public class EndForFlying : SourceAssociationEnd {
9      public EndForFlying(Flying source)
10       :base(source, SINGLETON) { }
11   }
12   public class EndForWing : TargetAssociationEnd {
13     public EndForWing(Wing target)
14       :base(target, SINGLETON) { }
15   }
16   private static EndForFlying
17     sourceEndFinder(Flying source){ return source.wings; }
18   private static EndForWing targetEndFinder(Wing target)
19     { return target.flying; }
20 }
```

Figura 76. Código resultante da tradução da associação da Figura 73.

Como pode ser observado na Figura 74, diferentemente de *properties*[14] (atributos) e métodos (operações), nomes de papel declarados em interfaces não necessitam ser declarados novamente nas classes que a especializam. Isto se deve ao fato de que não existe declaração abstrata de nomes de papel, como ocorre com *properties* e métodos, pelo uso da palavra reservada *abstract*. Sendo assim, também não há diferença de codificação de nomes de papel entre classes e classes abstratas.

A necessidade de declaração de nome de papel nas interfaces e classes especialistas de uma interface ocorre somente no caso de conflito de nomes em múltipla herança. Isto ocorre quando um classificador (classe ou interface) especializa dois ou mais classificadores que possuem atributos e/ou nomes de papel com o mesmo nome. Para resolver este problema, utiliza-se a mesma solução existente em C# para solucionar conflitos de métodos com a mesma assinatura e atributos com o mesmo nome: todas as propriedades em conflito provenientes de interfaces podem ser declaradas novamente no corpo do classificador

[14] Microsoft, 2011.

especialista, como *explicit interface implementation*[15]. Desta forma, havendo apenas uma única propriedade declarada sem este recurso, esta passa a ser a propriedade visível para o classificador especialista, sendo as demais visíveis somente quando há conversão da instância para as interfaces por intermédio de *casting*[16].

A única restrição para a declaração de nomes de papel em interfaces é a impossibilidade de utilizar modificadores de visibilidade. Em C# e em UML, toda propriedade (atributos, operações e *association ends*) pertencente a uma interface tem visibilidade pública. Para manter o padrão existente em C#, não é permitido declarar modificador de visibilidade em propriedades de interfaces, assumindo por padrão a visibilidade *public*.

Assim como em C#, em Association# as classes abstratas são codificadas de forma análoga as classes concretas.

4.1.1.12 Associações Reflexivas

Associações reflexivas são associações binárias em que ambas as extremidades estão conectadas à mesma classe. A Figura 77 exemplifica duas variações da associação reflexiva *NodeTree*, em que a classe *Node* é associada com ela própria. No exemplo *a*, *NodeTree* é uma associação navegável em ambos os sentidos e os seus *association ends* são ambos pertencentes à classe *Node* com visibilidade pública. No exemplo *b*, apenas o *association end* de nome *childs* é navegável, sendo este uma propriedade pertencente a associação *NodeTree*.

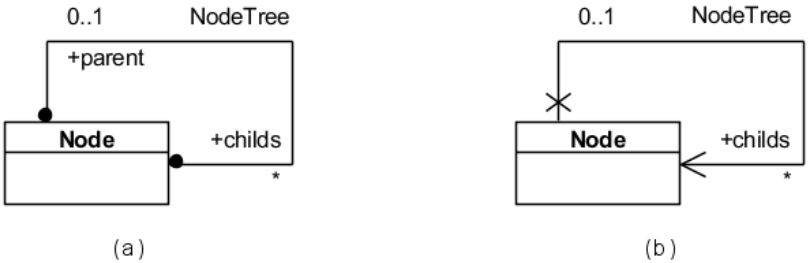

Figura 77. Exemplos de associação reflexiva, variando *association end ownership* e navegabilidade.

[15] Microsoft, 2011.
[16] Ibd.

A Figura 78 apresenta o código Association# para o exemplo *a* da Figura 77.

Código Association#

```
1  class Node {
2    public rolename childs[*] in NodeTree;
3    public rolename parent[0..1] in NodeTree;
4  }
5  association NodeTree between Node and Node;
```

Figura 78. Associação reflexiva com ambos os *association ends* pertencentes à classe.

A Figura 79 apresenta o código Association# para o exemplo *b* da Figura 77.

Código Association#

```
1  class Node { }
2  association NodeTree
3    between   Node childs[*]
4    and       Node[0..1]; //parent
```

Figura 79. Associação reflexiva com um *association end* não navegável e outro pertencente à associação.

A Figura 80 apresenta o código C# resultante da tradução do código Association# apresentado pela Figura 78. O código C# resultante da tradução do código contido pela Figura 79 é similar a este, diferenciando-se pelo nome do field *parent* (linha 3 da Figura 80) e suas chamadas (linha 19 da Figura 80) que recebem um nome aleatório por não haver nome para este no código Association#.

Código C#

```
1  public class Node {
2     public readonly NodeTree.EndForParent childs;
3     public readonly NodeTree.EndForChilds parent;
4     public Node() {
5        this.childs = new NodeTree.EndForParent(this);
6        this.parent = new NodeTree.EndForChilds(this); }  }
7  public class NodeTree : Association<Node,Node> {
8     public static readonly NodeTree SINGLETON =
9        new NodeTree();
10    private NodeTree() : base(
11       Multiplicity.ZERO_TO_MANY_RANGE,
12       Multiplicity.ZERO_TO_ONE_RANGE,
13       getSourceEnd, getTargetEnd) { }
14    private static EndForParent
15       getSourceEnd(Node sourceObject)
16          { return sourceObject.childs; }
17    private static EndForChilds
18       getTargetEnd(Node targetObject)
19          { return targetObject.parent; }
20    public class EndForParent : SourceAssociationEnd {
21       public EndForParent(Node sourceObject)
22          : base(sourceObject, SINGLETON) { } }
23    public class EndForChilds : TargetAssociationEnd {
24       public EndForChilds(Node targetObject)
25          : base(targetObject, SINGLETON) { } } }
```

Figura 80. Código resultante da tradução do código da Figura 78.

4.1.1.13 Operações de Substituição de Links de Associação

Conforme mencionado na Seção 4.1.1.7, os *association ends* possuem três operações de substituição de *links* de associação, chamadas de *Swap*. Estas permitem que um *link* de associação seja substituído por outro em um único passo atômico.

Utilizando o modelo da Figura 44 (associação *Job* entre as classes *Person* e *Company*), supondo os objetos *c1*, *c2* e *c3* instâncias da classe *Company*; *p1* instância da classe *Person*; e <*p1,c1*> um *link* da associação *Job* entre os objetos *p1* e *c1*; então, é possível destruir o *link* <*p1,c1*> para a posterior criação de um dos *links*: <*p1,c2*> ou <*p1,c3*>. Isto seria equivalente a troca de emprego de *p1*, que após a demissão de *c1*, estaria empregando-se em *c2* ou *c3*.

A Figura 81 apresenta o código Association# para que esta troca seja possível. Este executa duas operações atômicas, sendo que cada uma destas verifica a validade da cardinalidade perante a multiplicidade em ambos os *association ends* modificados. Na linha 1, os *association ends* armazenados em *p1* (*employer*) e *c1* (*employees*) são modificados. Na linha 2, há uma modificação no *association end* armazenado em *c2*

(*employees*) e, novamente, uma modificação no *association end* armazenado em *p1*.

Código Association#

```
1  p1#employer.Remove(c1);
2  p1#employer.Add(c2);
```

Figura 81. Exemplo de código para *p1* trocar de emprego, de *c1* para *c2*.

Sendo assim, o *association end* em *p1* sofre duas modificações, enquanto os *association ends* em *c1* e *c2*, apenas uma cada um. Ao final da execução, os *association ends* em *c1* e *c2* tiveram seus fatores de cardinalidade alterados em relação aos fatores de cardinalidade inicial (-1 e +1, respectivamente). Entretanto, o *association end* em *p1* mantém a mesma cardinalidade que tinha antes da execução do mesmo.

Apesar de não sofrer alteração de cardinalidade, o *association end* em *p1* é validado duas vezes. A inserção do código ilustrado pela Figura 81 dentro de um bloco atômico faria com que o *association end* em *p1* fosse verificado somente uma vez, assim como *c1* e *c2*, ao final do bloco. Entretanto, esta única verificação em *p1* poderia ser suprimida, por meio da construção de uma operação atômica, o qual substitui o *link* de associação entre <*p1,c1*> por <*p1,c2*>, validando somente a cardinalidade de *c1* e *c2*.

Em Association#, os *association ends* possuem as operações *Swap*, responsáveis por substituição de *links* de associação em passos atômicos, sem verificação dos *association ends* que não sofrem modificações de cardinalidade.

A Figura 82 apresenta as três operações *Swap* existentes, que serão apresentadas a seguir.

A linha 1 da Figura 82 apresenta a chamada da operação *Swap* que tem a mesma semântica do código ilustrado pela Figura 81, porém executada em um único passo, verificando apenas os association ends armazenados em *c1* e *c2*.

Código Association#

```
1  bool swapped = p1#employer.Swap(c1, c2);
2  swapped = p1#employer.Swap(ref c1, c2);
3  swapped = p1#employer.Swap(c1, p2, c2);
```

Figura 82. Exemplo de chamadas para as operações *Swap*.

Para que esta substituição seja feita é necessário atender as seguintes condições:

a) A instância recebida no primeiro argumento (*c1*) deve estar sendo referenciada pelo *association end* que está executando a operação (*p1#employer*), indicando que existe um *link* de associação entre o objeto que contém o *association end* (*p1*) e o objeto do primeiro argumento (*c1*);
b) A instância recebida no segundo argumento (*c2*) deve não estar sendo referenciada pelo *association end* que está executando a operação (*p1#employer*), indicando que não existe um *link* de associação entre o objeto que contém o *association end* (*p1*) e o objeto do segundo argumento (*c2*);
c) As instâncias recebidas nos argumentos (*c1* e *c2*) não podem violar a multiplicidade (dos *association ends* armazenados nos mesmo) ao final da operação.

Caso as condições *a* e *b* não sejam atendidas, o método retorna *false* em vez de *true*. Caso a condição *c* não seja atendida, uma exceção é levantada.

A linha 2 da Figura 82 apresenta uma variante da operação da linha 1 em que inclui a tentativa de destruição do objeto que terá sua cardinalidade decrementada (primeiro argumento). A necessidade desta abordagem é descrita na Subseção 4.1.1.14.

A linha 3 da Figura 82 apresenta um tipo de substituição de *links* de associação mais complexo que o primeiro (linha 1) e sua variante (linha 2). Esta operação executa uma troca de *links* de associação entre dois pares de objetos *linkados*, de forma similar ao código da Figura 83, onde se supõe *p1* e *p2* instâncias da classe *Person*, *c1* e *c2* instâncias da classe *Company*, e <*p1,c1*> e <*p2,c2*> *links* da associação *Job*. Há então a substituição de <*p1,c1*> e <*p2,c2*> por <*p1,c2*> e <*p2,c1*>. Apesar de esta operação ser mais complexa que as anteriores por ter mais variáveis participando do processo, ela não necessita de verificação de cardinalidade dos quatro *association ends* modificados, pois estes mantêm seus valores iniciais no final da operação.

Código Association#

```
1  p1#employer.Remove(c1);
2  p1#employer.Add(c2);
3  p2#employer.Remove(c2);
4  p2#employer.Add(c1);
```

Figura 83. Exemplo de código para *p1* trocar de emprego com *p2*.

Para que esta operação *Swap* de dupla substituição (troca) seja executada, é necessário atender as seguintes condições:

a) Deve haver um *link* da referida associação, entre o objeto que armazena o *association end* (*p1*) e o objeto recebido pelo primeiro argumento (*c1*);
b) Deve haver um link da referida associação, entre os objetos recebidos pelo segundo e terceiro argumentos;
c) Não deve haver link da referida associação, entre o objeto que armazena o *association end* (*p1*) e o objeto recebido pelo terceiro argumento;
d) Não deve haver link da referida associação, entre os objetos do primeiro e do segundo argumentos.

Caso uma destas condições não seja atendida, a operação retorna *false* em vez de *true*.

As operações *Swap*, além de tornarem desnecessário o uso de blocos atômicos nos casos em que estas podem ser utilizadas, são mais ágeis por evitarem a necessidade de verificação de *association ends* que, apesar de sofrerem modificações no conjunto interno de referências, não sofreram mudanças na cardinalidade no final da execução da operação.

Outro ponto a ser considerado é que as os testes condicionais de execução das operações *Swap* não necessitam ser codificados pelo programador, bastando apenas verificar o retorno da operação. O valor de retorno *true* significa sucesso na execução da operação, *false* significa que a operação não foi executada por não satisfazer uma das condições. Exceção levantada significa violação de limite de multiplicidade.

4.1.1.14 Destruindo Objetos Obrigatoriamente Associados

Por utilizar *garbage collector*, para que a destruição de um objeto ocorra é necessário que o mesmo não esteja sendo referenciado (por uso de referências fortes) em nenhum ponto do programa.

A implementação de associações em Association# utiliza coleções de referências fortes para os objetos *linkados*, nos *association ends*. Isto significa que se dois objetos a_1 e b_1 estão associados (por um *link* de associação), ambos contêm uma referência forte, um para o outro. Se a_1 não está sendo referenciado fortemente em nenhum outro ponto do programa que não seja por b_1, ele depende de que b_1 esteja sendo referenciado fortemente para que ambos não sejam coletados.

Utilizando o exemplo apresentado pela Figura 84, supondo a_1 um objeto da classe *A*, b_1 um objeto da classe *B*, que tanto a_1 quanto b_1, além de *linkados*, estão fortemente referenciados somente por variáveis locais de um método que está sendo executado; se em algum momento

houver a remoção do *link* de associação, o *association end* armazenado em b_1 (*a*) estará com sua regra de multiplicidade violada. Para que isto não ocorra, é necessário que b_1 seja coletado pelo *garbage collector* (ou *linkado* a outro objeto que exerça o mesmo papel de a_1) antes de ser validado pelo mecanismo de verificação de multiplicidade.

Código Association#

```
1  class A { }
2
3  class B { }
4
5  association AB
6      between A a [1]
7          and B b [*];
```

Figura 84. Código Association# para o modelo UML apresentado pela Figura 42.

Há duas formas de executar esta ação:

a) Executar em um bloco atômico as operações de remoção do *link* e remoção de todas as demais referências fortes para b_1. Não havendo referências fortes para b_1, ele e seu *association end* são coletados ao final do bloco atômico, na chamada ao *garbage collector*, antes da verificação de multiplicidade.

b) Remove-se a(s) referência(s) forte(s) para b_1 e, na sequência, a_1 remove o seu *link* com b_1. Não havendo outra referência forte para b_1, ele e seu *association end* são coletados na chamada ao *garbage collector*. Esta chamada é executada durante a verificação de multiplicidade, após esta ter detectado a invalidade da cardinalidade do *association end* de b_1.

Código Association#

```
1  atomic {
2    a1#b.Remove(b1);
3    b1 = null;
4  }
```

Figura 85. Destruição de um objeto obrigatoriamente associado, com o uso de um bloco atômico.

O primeiro caso está ilustrado na Figura 85 e o segundo na Figura 86. Observa-se que o método *Single* (linha 2 da Figura 86) é um método pertencente a interface *IEnumerable* <*T*>, presente no pacote *System.Collections.Generic* da biblioteca padrão da linguagem C#, e sua

função é retornar o único objeto existente na coleção enumerável ou levantar uma exceção caso exista mais de um.

Código Association#

```
1  b1 = null; //removendo referência de b1.
2  a1#b.Remove(a1#b.Get().Single()); //b1 é coletado.
```

Figura 86. Destruição de um objeto obrigatoriamente associado, executado de forma direta.

O segundo caso possui uma variação que executa as duas operações em uma única, utilizando a operação *Remove* do *association end* em que, diferentemente da utilizada na linha 2 da Figura 86, o único argumento é passado por referência (palavra reservada *ref*[17]) e não por cópia. Ao executar esta remoção, se a operação violar os limites de multiplicidade da instância passada como argumento, é atribuído *null* a variável passada como argumento como tentativa de remover a última referência forte a esta. Caso seja realmente a última, a instância que era referenciada por esta variável é coletada pelo *garbage collector*. Caso contrário, a exceção é levantada. A Figura 87 apresenta esta variação da operação *Remove* que recebe o argumento por referência.

Código Association#

```
1  a1#b.Remove(ref b1); //b1 perde ambas as referências.
2                       //b1 == null.
```

Figura 87. Destruição de um objeto obrigatoriamente associado de forma direta em apenas um passo.

Similarmente, a variante do método *Swap* que possui o primeiro argumento declarado como referência (mencionado na Subseção 4.1.1.13) é utilizada com o mesmo propósito: tentar destruir a referência que teve a cardinalidade de seu *association end* decrementada, caso esta seja inválida em relação aos seus limites de multiplicidade.

4.1.2 Gramática

Toda a gramática da linguagem C#[18] pode ser utilizada na linguagem Association#, acrescentando-lhe a sintaxe e semântica das novas estruturas.

Por questões de simplificação, esta subseção apresentará apenas algumas das modificações e extensões realizadas na gramática da

[17] Microsoft, 2011.
[18] Ibd.

linguagem C#. Todas as modificações e extensões estão contidas no Apêndice A.

4.1.2.1 Gramática Léxica

Foram inseridos na gramática léxica as palavras reservadas *association*, *between*, *and*, *rolename* e *atomic*, e o sinal de pontuação ".." (ponto-ponto).

4.1.2.2 Gramática Sintática

Entre as modificações da gramática sintática da linguagem C#, destaca-se a inserção de novas produções para o não-terminal *type-declaration*. Este, antes responsável pela construção de classes, interfaces, entre outras construções de *cidadãos de primeira ordem*, agora também permite a construção de associações e classes de associação.

Para apresentar estas modificações, serão apresentadas algumas partes da nova gramática, em *Backus-Naur Form* (BNF). Os não-terminais estão indicados entre os caracteres < e > (maior e menor). Os terminais estão indicados pelo grifo em negrito e entre pares do caracter " (aspas duplas). Não-terminais e terminais novos, inseridos nesta extensão, estão grifados com sublinhação. *Tokens* opcionais estão demarcados entre *[* e *]* (colchetes).

BNF

```
1  <type-declaration>
2    ::=  <class-declaration>
3    | <struct-declaration>
4    | <interface-declaration>
5    | <enum-declaration>
6    | <delegate-declaration>
7    | <association-declaration>
8    | <associationclass-declaration>
```

Figura 88. Produções para o não-terminal *type-declaration*.

A Figura 88 apresenta as produções novas (linhas 7 e 8) e já existentes (linhas 2 a 6) para o não-terminal *type-declaration*.

Os não-terminais *association-declaration* e *associationclass-declaration* possuem as produções para construção de associações e classes de associação, respectivamente. As produções destes novos não-terminais estão descritas pela Figura 89.

```
BNF
```

```
1  <association-declaration>
2    ::=  [ <association-modifiers> ] "association"
3         <identifier> [ <association-base> ] <between-exp>
4         ";"
5
6  <associationclass-declaration>
7    ::=  [ <association-modifiers> ] "association" "class"
8         <identifier> [ <associationclass-base> ]
9         <between-exp> <associationclass-body>
```

Figura 89. Produções para os não-terminais *association-declaration* e *associationclass-declaration*

A Figura 90 apresenta as produções do não-terminal *class-member-declaration*, responsável pelas construções dos membros (atributos, métodos, classes *nested*, etc) de uma classe. A linha 13 apresenta a nova produção acrescentada na expansão da gramática C#, enquanto as linhas 2 a 12 apresentam as produções que já existiam antes da expansão.

```
BNF
```

```
 1  <class-member-declaration>
 2    ::=  <constant-declaration>
 3    |    <field-declaration>
 4    |    <method-declaration>
 5    |    <property-declaration>
 6    |    <event-declaration>
 7    |    <indexer-declaration>
 8    |    <operator-declaration>
 9    |    <constructor-declaration>
10    |    <destructor-declaration>
11    |    <static-constructor-declaration>
12    |    <type-declaration>
13    |    <rolename-declaration>
```

Figura 90. Produções do não-terminal *class-member-declaration*.

A Figura 91 apresenta a única produção do novo não-terminal *rolename-declaration*. Este é responsável pela construção da declaração de nome de papel pertencente a classes.

```
BNF
```

```
1  <rolename-declaration>
2    ::=  [ <rolename-modifiers> ] "rolename" <identifier>
3         "in" <association-type> ";"
```

Figura 91. Produção do não-terminal *rolename-declaration*.

4.1.3 Biblioteca de Apoio

Esta subseção apresenta o funcionamento e o uso da biblioteca de apoio na construção de associações em código C# proveniente da tradução de código Association#. O código fonte desta encontra-se disponível na mídia digital que acompanha este trabalho e no Apêndice B.

Conforme observado na Subseção 4.1.1, as novas estruturas escritas em código Association# são traduzidas para estruturas em código C# responsáveis por prover a semântica destas novas estruturas.

4.1.3.1 Construindo Associações Binárias

A construção de associações binárias é feita implementando-se uma classe que especializa *Association<SOURCE,TARGET>*, presente no *namespace AssociationSharp.Lang* da biblioteca de apoio. Um exemplo desta construção foi apresentada anteriormente pela Figura 47. Nesta, a classe que representa a associação *Job* tem como finalidade:

a) Configurar suas duas classes *nested* (linhas 14 a 16 e 17 a 19), responsáveis por instanciar os *association ends*;
b) Configurar e instanciar um *singleton* contendo as configurações da associação (linhas 2 a 7);
c) Conter a construção de dois métodos responsáveis por resolver os endereços dos *association ends* (linhas 8 a 10 e 11 a 13) presentes nas instâncias das classes associadas (linhas 2 e 7 da Figura 46).

Os tipos genéricos *SOURCE* e *TARGET*, presentes no cabeçalho da classe *Association<SOURCE,TARGET>*, recebem as classes que serão associadas. Estes nomes não têm relação com a navegabilidade da associação, servindo apenas para fazer distinção entre as duas extremidades da associação. Estes tipos genéricos são utilizados para configurar os tipos utilizados pelas classes internas *SourceAssociationEnd* e *TargetAssociationEnd*, e os métodos *delegates*[19], requisitados pelo construtor (linha 7 da Figura 47), que recebem os endereços dos métodos que resolvem os endereços dos *association ends*.

[19] Microsoft, 2011.

4.1.3.2 Construindo Association Ends

Os *association ends* são construídos estendendo as classes *SourceAssociationEnd* e *TargetAssociationEnd* (linhas 14 a 16 e 17 a 19, da Figura 47), ambas contidas pela classe *Association<SOURCE,TARGET>*, e, posteriormente, instanciando objetos destas classes resultantes (linhas 4 e 9 da Figura 46), armazenando-os em *fields* das instâncias (linhas 2 e 7 da Figura 46).

Independente da localização do *association end* no código Association# ser uma propriedade de classe associada (*employer*, no exemplo) ou ser uma propriedade da associação (no exemplo, *employees*), no código C# resultante de tradução, ele estará sempre localizado na instância da classe. Os *association ends* são instanciados e armazenados, cada um, nas instâncias da *classe aposta ao papel*. Por serem os *association ends* os responsáveis pelo armazenamento de referências para instâncias da *classe atuante do papel*, esta construção assemelha-se ao *pattern mutual friends*. Isto é necessário para que não ocorra o *problema da referência perdida*.

As classes *SourceAssociationEnd* e *TargetAssociationEnd* necessitam ser estendidas, conforme demonstrado pela Figura 47, para que as classes especialistas passem automaticamente o *singleton* da associação por argumento ao construtor das suas respectivas classes generalistas (linhas 16 e 19).

A classe especialista de *SourceAssociationEnd* (no exemplo, *EndForPerson*) instancia *association ends* a serem acoplados às instâncias da classe recebida pelo parâmetro (*generic*[20]) *SOURCE* (no exemplo, *Person*), localizado na classe *Association<SOURCE, TARGET>*. Cada *association end*, instanciado desta maneira, armazena uma referência para a instância de *SOURCE*, recebido como argumento pelo construtor (linhas 15 e 16 da Figura 47), e uma coleção para as referências das instâncias de *TARGET* que forem posteriormente associadas ao objeto de *SOURCE* mencionado. De forma análoga, um *association end* instanciado da classe especialista de *TargetAssociationEnd* referencia a instância de *TARGET* que o contém e todas as instâncias de *SOURCE* associadas.

A Figura 92 apresenta parcialmente o código da classe Association<SOURCE, TARGET> contendo suas duas classes *nested*.

[20] Microsoft, 2011.

Código C#

```
 1  public abstract class Association<SOURCE, TARGET> {
 2    // instance
 3    ...
 4
 5    // types
 6    public class SourceAssociationEnd
 7      : AssociationEnd<SOURCE,TARGET>  {...}
 8    public class TargetAssociationEnd
 9      : AssociationEnd<TARGET,SOURCE>  {...}
10  }
```

Figura 92. Código parcial da classe Association<SOURCE,TARGET> presente na biblioteca de apoio.

Conforme apresentado, *SourceAssociationEnd* e *TargetAssociationEnd* são classes especialistas da classe *AssociationEnd<OWNER,OPPOSITE>* que passam para os parâmetros desta, as classes recebidas pelos parâmetros da classe *Association<SOURCE,TARGET>*. Para a classe *SourceAssociationEnd*, o parâmetro *OWNER* de sua classe generalista recebe a mesma classe recebida em *SOURCE*, e *OPPOSITE* recebe a mesma recebida em *TARGET*. Para a classe *TargetAssociationEnd*, esta parametrização inverte-se. Assim, o parâmetro OWNER representa a *classe oposta ao papel* e o parâmetro OPPOSITE, a *classe atuante do papel*.

A classe *AssociationEnd<OWNER,OPPOSITE>*, apresentada pela Figura 93, possui:

a) Operações simples de modificação do conjunto de *links* de associação (linhas 10 a 13) – *Add*, *Remove* e *RemoveAll*, apresentadas na Subseção 4.1.1.7*;*
b) Operações complexas de modificação do conjunto de links de associação (linhas 14 a 19) – as três variações de *Swap*, apresentadas em detalhes na Seção 4.1.1.14;
c) Operações de consulta de *links* (linhas 20 e 21) – *Get* e *Contains*, apresentadas na Subseção 4.1.1.7;
d) Atributos e operações informativos (linhas 22 a 24) – *Count* (cardinalidade), *multiplicity* (a instância de *AssociationSharp. Lang.Multiplicity* instanciada na associação) e *IsValid* (se a cardinalidade é válida), apresentadas na Subseção 4.1.1.7;
e) Referência interna para a instância da classe proprietária (linha 8);

f) Uma coleção interna do tipo *System.Collections.Generic.HashSet<OPPOSITE>* (classe *base*[21] na linha 2), responsável por armazenar as referências das instâncias da classe parametrizada em *OPPOSITE* que foram associadas ao seu proprietário (classe parametrizada em *OWNER*);
g) As duas referências para os métodos *delegate* (linhas 4 a 7) configurados na associação (linha 7 da Figura 47).

Código C#

```
1  public abstract class AssociationEnd<OWNER, OPPOSITE>
2     : HashSet<OPPOSITE>
3  {
4     internal readonly
5        AssociationEndFinder ownerAssociationEndFinder;
6     internal readonly AssociationEnd<OPPOSITE,OWNER>.
7        AssociationEndFinder oppositeAssociationEndFinder;
8     internal OWNER ownerObject {private set {...} get {...}}
9     protected internal readonly object synchronizer;
10    public bool Add(OPPOSITE opposite) {...}
11    public bool Remove(OPPOSITE opposite) {...}
12    public bool Remove(ref OPPOSITE opposite) {...}
13    public void RemoveAll() {...}
14    public bool
15       Swap(OPPOSITE oldOpposite,OPPOSITE newOpposite) {...}
16    public bool Swap(ref OPPOSITE oldOpposite,
17       OPPOSITE newOpposite) {...}
18    public bool Swap(OPPOSITE thisOpposite,OWNER otherOwner,
19       OPPOSITE otherOpposite) {...}
20    public virtual IEnumerable<OPPOSITE> Get() {...}
21    public bool Contains(OPPOSITE opposite) {...}
22    public int Count { get{...} }
23    public readonly Multiplicity multiplicity;
24    public bool IsValid() {...}
25 }
```

Figura 93. Código parcial da classe *AssociationEnd<OWNER,OPPOSITE>*.

Mesmo que os *association end* tenha limite máximo de multiplicidade igual a 1, este utiliza sempre um HashSet<...> para armazenar referências para instâncias da *classe atuante do papel* por questão de padronização. Maiores detalhes sobre isto são abordados na discussão sobre perda de desempenho (Subseção 5.2.4).

As operações de modificação do conjunto de *links* referenciados pelos *association ends* de uma associação são executados em *thread-safety*. O objeto utilizado como semáforo (linha 8) é sempre o *singleton* que representa a associação (linha 2 da Figura 47).

[21] Microsoft, 2011.

4.1.3.2.1 Navegabilidade e Visibilidade

Navegabilidade e visibilidade, assim como a posse do *association end*, deve ser resolvidas no código Association# pelo compilador da nova linguagem, não necessitando encapsular dados no código C#, evitando assim a necessidade de implementação de *handles*, como no *code pattern* de Gessenharter, ou uso de reflexão.

4.1.3.2.2 Multiplicidade

Conforme mencionado na Subseção 4.1.1.8.1, os *association ends* verificam os limites de multiplicidade a cada operação que modifique o conjunto de links, salvo as exceções de substituição de *links*, mencionada na Subseção 4.1.1.13. Esta verificação é feita pelo objeto da classe *Multiplicity* armazenado no próprio *association end* (linha 20 da Figura 93).

Código C#

```
 1  public class Multiplicity {
 2    public static readonly Multiplicity
 3       ZERO_TO_MANY_RANGE = new Multiplicity(),
 4       ZERO_TO_ONE_RANGE = new Multiplicity(0,1),
 5       ONE_TO_ONE_RANGE = new Multiplicity(1),
 6       ONE_TO_MANY_RANGE = new Multiplicity(1,MANY);
 7    public const uint MANY = uint.MaxValue;
 8    public readonly bool mandatory;
 9    public readonly uint minimumBound;
10    public readonly uint maximumBound;
11    public delegate bool ValidateMethod(int count);
12    public readonly ValidateMethod isValid;
13  }
```

Figura 94. Código parcial da classe *Multiplicity*.

Os objetos da classe *Multiplicity*, apresentada pela Figura 94, contêm:

a) Um valor do tipo *uint* (*unsigned integer*) representando o limite inferior de multiplicidade (linha 9);
b) Um valor inteiro do tipo *uint* para representar o limite superior de multiplicidade (linha 10);
c) Um método *delegate* de validação (definido na linha 11 e armazenado na linha 12) que testa uma cardinalidade recebida por argumento, retornando *true* ou *false* para cardinalidade válida e inválida, respectivamente.

Quando não há limite inferior (limite inferior é igual a zero), somente o limite superior é testado. Quando não há limite superior, somente o limite inferior é testado. Quando não há nenhum dos dois limites, nada é testado e o método *delegate* de validação retorna sempre *true*.

A classe *Multiplicity* contém algumas constantes com multiplicidades já definidas (linhas 2 a 7):

a) *ZERO_TO_MANY_RANGE* – representa a multiplicidade * em UML;
b) *ZERO_TO_ONE_RANGE* – representa a multiplicidade *0..1* em UML;
c) *ONE_TO_ONE_RANGE* – representa a multiplicidade *1* em UML;
d) *ONE_TO_MANY_RANGE* – representa a multiplicidade *1..** em UML;
e) *MANY* – utilizado para instanciar novos objetos *Multiplicity*, representa o limite superior de multiplicidade *.

As linhas 5 e 6 da Figura 47 exemplificam o uso de duas destas constantes. Outros intervalos de limites de multiplicidade podem ser instanciados. Exemplos: *2..5*, *0..4*, *3*, *10..**, etc.

4.1.3.3 Construindo o Bloco Atômico

Um bloco atômico é implementado conforme apresentado pela Figura 58. A classe *AssociationSharpSystem* (presente em *AssociationSharp.Lang*) contém dois métodos *static*, responsáveis por desabilitar e reabilitar o mecanismo de verificação de multiplicidade (*__DisableSystemConstraints* e *__EnableSystem-Constraints*, respectivamente). Todas as operações inseridas dentro do bloco são posicionadas dentro do bloco *try*, logo abaixo da chamada do método que desabilita a verificação. O bloco *try...finally...* é necessário para que independentemente de ocorrer uma exceção ou outro tipo de desvio de fluxo, o método para reabilitação do mecanismo de verificação seja obrigatoriamente executado ao sair do bloco.

Internamente, a classe *AssociationSharpSystem* possui:

a) Lista de referências para todas as classes que instanciam os *association ends* (cada classe instanciou um objeto que implementa a interface *AssociationSharp.Lang. IRegisterAssociationEnd* e adicionou nesta lista);
b) Lista de todos os *singletons* criados pelas associações (ver Figura 47, linha 2);

c) Uma pilha de níveis, cada uma contendo referências fracas para os association *ends* que executaram operações no nível. Níveis são necessários devido à possibilidade de utilizarem-se blocos atômicos dentro de blocos atômicos;

Quando chamado o método para desabilitar o mecanismo de verificação de multiplicidade (método *__DisableSystemConstraints* da classe *AssociationSharpSystem*), todos os *singletons* criados pelas associações são bloqueados pela *thread* corrente, impossibilitando que outras *threads* executem operações nos *association ends* até que o método de habilitação seja chamado. Consequentemente, não há também como outra *thread* executar um bloco atômico até que a *thread* corrente finalize a execução do bloco atômico atual. Todas as classes especialistas de *AssociationEnd<OWNER,OPPOSITE>* são notificadas para que as operações *Add*, *Remove* e *RemoveAll* de suas instâncias (os *association ends*), não façam testes de multiplicidade, mas notifiquem a classe *AssociationSharpSystem* de que elas executaram uma dessas operações e necessitam ser validadas ao habilitar o mecanismo de verificação.

É possível aninhar vários blocos atômicos, um dentro do outro (ver exemplo da Figura 63 e a implementação na Figura 64), o que faz incrementar o *nível de profundidade da atomicidade* a cada chamada do método que desabilita as verificações. Dentro de um bloco atômico, somente a primeira chamada de uma operação que modifique o conjunto interno de referências de um *association end,* faz com que uma referência fraca para ele seja armazenada no *nível de profundidade da atomicidade* corrente. Mesmo que ele seja modificado em níveis mais profundos posteriormente, ele somente será validado no nível que armazenou a referência para ele. A cada chamada feita para o método de habilitação, a execução desempilha um nível, validando os *association ends* referenciados por este. Ao desempilhar o último nível, todos os *singletons* são desbloqueados e as classes de instanciação dos *association ends* são notificadas para validarem cada operação de adição ou remoção que sejam executadas posteriormente.

4.1.3.4 Construindo Classes de Associação

A construção de classes de associação é similar a construção de associações simples. A diferença das duas é de que na implementação de classes de associação utiliza-se uma classe especialista de *AssociationClass<SOURCE,TARGET,LINK>* e é necessário especificar a classe a ser passada para o parâmetro *LINK*, enquanto na

implementação de associações simples, utiliza-se uma classe especialista de *Association<SOURCE,TARGET>*.

A classe passada como parâmetro em *LINK*, no código C#, deve ser uma classe que mapeie todas as propriedades da classe de associação (escrita em Association#), sendo suas instâncias, então, consideradas as instâncias desta classe de associação.

A Figura 50 apresentou a implementação da classe de associação *Job*, substituindo a implementação da associação de mesmo nome apresentada anteriormente a esta, na Figura 47. A antiga classe *Job* (Figura 47) e suas chamadas (linhas 2, 4, 7 e 9 da Figura 46) foram renomeadas para *Association* (linhas 2, 4, 8, 10 e 16), sendo esta classe movida para dentro da nova classe *Job* (linhas 16 a 35), que foi criada contendo os atributos e métodos da classe de associação (linhas 13 a 14). Verifica-se também que a classe generalista da classe *Association* (anteriormente conhecida como *Job*) deixou de ser *Association<SOURCE, TARGET>* (linha 1 da Figura 47) para ser *AssociationClass<SOURCE, TARGET, LINK>* (linha 17).

Observa-se que a classe renomeada para *Association* poderia ter qualquer nome aleatório não conflitante com outras estruturas, pois esta é invisível ao programador no código Association#.

Código C#

```
1  public abstract class AssociationEnd<OWNER, OPPOSITE, LINK>
2     : Dictionary<OPPOSITE, LINK>
3     where LINK : class, new()
4  {
5     ...
6     public IEnumerable<LINK> GetAssociationLink() {...}
7     public LINK GetAssociationLink(OPPOSITE opposite){...}
8  }
```

Figura 95. Código parcial da classe *AssociationEnd<OWNER,OPPOSITE, LINK>*.

Os *association ends* para classes de associação diferenciam-se dos *association ends* para associações em apenas um aspecto: necessitam armazenar, para cada referência da classe oposta, uma referência para a classe recebida pelo parâmetro *LINK*. Para isto, utiliza-se um *System.Collections.Generic.Dictionary<OPPOSITE,LINK>* em vez de um *HashSet<OPPOSITE>*. Dois métodos para retornar os objetos do tipo *LINK* estão inclusos, além dos métodos de consultas previamente mencionados: um para retornar todos os *links* de associação e outro para retornar um *link* de associação específico para um objeto *OPPOSITE* recebido como argumento.

A Figura 95 apresenta o código da classe *AssociationEnd <OWNER, OPPOSITE, LINK>* que difere da classe *AssociationEnd <OWNER, OPPOSITE>* (apresentado na Figura 93).

4.1.3.5 Construindo Associações Generalistas e Especialistas

Estão contidas na biblioteca de apoio, classes para implementação de especialização e generalização de associações, bem como para especialização e generalização de classes de associação. Estas classes são:

a) *GeneralizedAssociation<SOURCE,TARGET>*;
b) *SpecializedAssociation<SOURCE,TARGET,G_SOURCE, G_TARGET>*;
c) *GeneralizedAssociationClass<SOURCE,TARGET,LINK>*;
d) *SpecializedAssociationClass <SOURCE, TARGET, LINK, G_SOURCE, G_TARGET, G_LINK>*.

Implementações de associações generalizadas e classes de associação generalizadas são feitas da mesma forma como descrito anteriormente para associações e classes de associação, necessitando apenas substituir as classes base *Association <SOURCE, TARGET>* por *GeneralizedAssociation <SOURCE, TARGET>* e *AssociationClass <SOURCE, TARGET, LINK>* por *GeneralizedAssociationClass <SOURCE, TARGET, LINK>*. A diferença está em que os *association ends* das classes para implementação de associações generalistas permitem ser conectados a *association ends* provenientes das classes *SpecializedAssociation<...>* e *SpecializedAssociationClass<...>*. Estas são classes para implementação de associações e classes de associação especializadas.

As classes *nested*, responsáveis por instanciarem os *association ends* de ambas as classes generalistas, são *SourceGeneralizedAssociationEnd* e *TargetGeneralizedAssociationEnd*. Para as classes especialistas, estas são *SourceSpecializedAssociationEnd* e *TargetSpecializedAssociationEnd*.

A implementação das associações e classes de associação especializadas não possui grandes diferenças em relação aos demais tipos. As diferenças na estrutura de implementação de associações e classes de associação são somente nos 2 e 3 parâmetros a mais (*G_SOURCE*, *G_TARGET* e *G_LINK*) os quais identificam os tipos generalizados para as classes recebidas nos parâmetros *SOURCE*, *TARGET* e *LINK* (sendo *LINK* e *G_LINK* somente para as classes de associação). A única diferença estrutural nos *association ends*

especializados é que estes recebem como argumento em seus construtores, no lugar do proprietário do mesmo, o *association end* que se está especializando. No caso de especialização de classe de associação, a classe que mapeia (código C#) a classe de associação especialista (do código Association#) é uma classe especialista da classe que mapeia a classe de associação generalista.

A Subseção 4.1.1.10 apresentou um exemplo de generalização e especialização de associações. Neste, a implementação da associação *Job* (linhas 1 a 20 da Figura 68) foi alterada, em relação a implementação do exemplo presente na Figura 44 (apresentada na Figura 47), apenas substituindo a classe base *Association<Person, Company>* (linha 1 da Figura 47) por *GeneralizedAssociation<Person, Company>* (linha 1 da Figura 68).

Neste exemplo de generalização/especialização de associações, em relação ao exemplo de associação, acrescentou-se a associação *Training*. Sua implementação (linhas 22 a 47 da Figura 68) é similar a implementação da associação *Job* (linhas 1 a 20 da Figura 68), tendo como classe base a classe *SpecializedAssociation*. A classe *Company* teve o acréscimo do *association end* de nome *trainees* (linha 8 da Figura 67), proveniente da associação *Training* e que especializa o *association end* de nome *employees* (passado como argumento na linha 12 da Figura 67). Acrescentou-se a classe *Student* (linhas 14 a 18 da Figura 67), especialista da classe *Person* (linhas 1 a 4 da Figura 67), contendo o *association end* de nome *employer* (linha 15 da Figura 67), especialista do *association end* de mesmo nome (passado como argumento na linha 18 da Figura 67), presente na classe base (linha 2 da Figura 67).

Quanto à generalização e especialização de classes de associação, um exemplo foi apresentado na Subseção 4.1.1.10.1. Este é análogo ao exemplo anterior, porém utilizando as classes *GeneralizedAssociationClass<...>* e *SpecializedAssociationClass<…>*

4.1.4 Resolvendo os Problemas do Estado da Arte

A linguagem Association# resolve, em uma única abordagem, os três problemas do estado da arte elencados na Seção 3.2: o *problema da visibilidade global*, o *problema da referência perdida* e o *problema da destruição de objetos obrigatoriamente associados*.

O primeiro problema é resolvido devido a nova linguagem não disponibilizar os *links* de associação por intermédio de consultas à própria associação, sendo necessário sempre consultar uma das instâncias *linkadas* para obter-se os *links* da associação.

O segundo problema é resolvidos através da implementação do *pattern mutual friends*, em que os *association ends* são armazenados nas instâncias das classes participantes da associação.

O terceiro problema é resolvido conforme descrito na Subseção 4.1.1.14.

4.2 EXEMPLO DE USO

Pra exemplificar o uso da linguagem Association#, em especial o uso das estruturas de associação, utilizou-se o exemplo contido no trabalho de Balzer, Gross, Eugster (2007). Este descreve um sistema de uma universidade em que os requisitos são descritos conforme a Figura 96, e a modelagem conforme o diagrama de classes da Figura 97.

R1	The entities of the system are *students*, *courses*, and *faculty members*.
R2	For every student the *name*, a unique registration *number*, and the current *year* of study must be retained. The year of study cannot exceed 10. Courses must indicate their *titles*. Faculty members must list their *names*.
R3	Enrolled students must *attend* courses. When attending a course students can get a *mark* between 1 and 6.
R4	Students can *assist* courses as teaching assistants. Students cannot assist courses they are attending themselves. For every teaching assistant the *language of instruction* must be recorded. For every assisted course a maximal *group size* can be defined, which restricts the number of students that are assisted by a single teaching assistant. In case a maximal group size is prescribed for a course, then it must be guaranteed that the number of students assisted by a single teaching assistant must not exceed the maximal group size defined for that course.
R5	Students can *work for* a faculty member as research assistants, provided that they are at least in their third year. For every research assistant the *grant amount* paid can be retained.
R6	Every course must be *taught* by at least one faculty member.
R7	Every faculty member must name at least one other faculty member as *substitute*. No faculty member can be its own substitute, and two distinct faculty members cannot substitute each other.

Figura 96. Requisitos de um sistema para universidade. Fonte: Balzer, Gross e Eugster (2007).

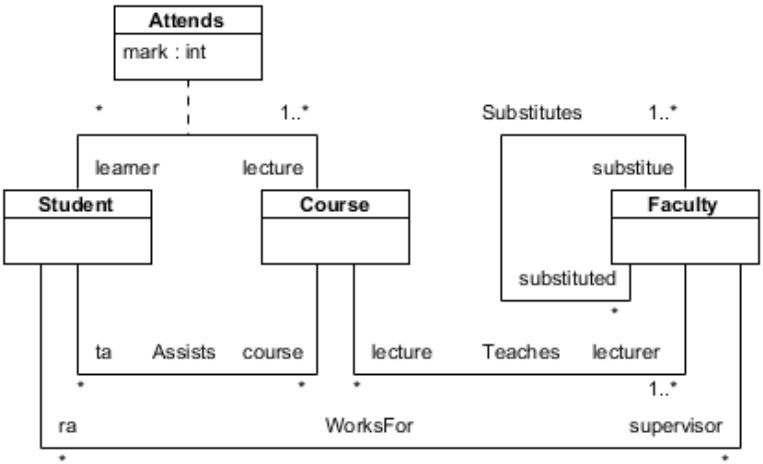

Figura 97. Diagrama de classes simplificado, construído a partir do exemplo de Balzer *et al*

Entretanto, Balzer, Gross e Eugster (2007) modelam este sistema utilizando *interposição de elementos*, que não é suportado pela UML e, consequentemente, pela linguagem Association#. Desta forma, informações como a *linguagem de instrução* (*language of instruction*) e o *tamanho do grupo* (*group size*), ambas descritos em *R4* (Figura 96), bem como o *valor de bolsa concedido* (*grant amount*), descrito em *R5*, devem ser acomodadas diretamente nas classes participantes das associações, ou então, em novas classes, criadas para representar os nomes de papel. A primeira opção, em relação à segunda, é mais simples, porém gera um maior acoplamento e menor coesão. Optou-se pela segunda opção, resultando o diagrama de classe ilustrado pela Figura 98 que não faz uso de *interposição de elementos*. Foram incluídos neste: as classes sinalizadas com o *estereótipo*[22] *<<Role>>*, criadas para representar os nomes de papel que acomodam os *elementos interpostos*; os atributos de cada classe; a visibilidade dos nomes de papel e atributos; e as representações das posses dos *association ends*. A navegabilidade das associações permaneceu bidirecional, conforme já era no modelo da Figura 97.

[22] Object Management Group, 2010.

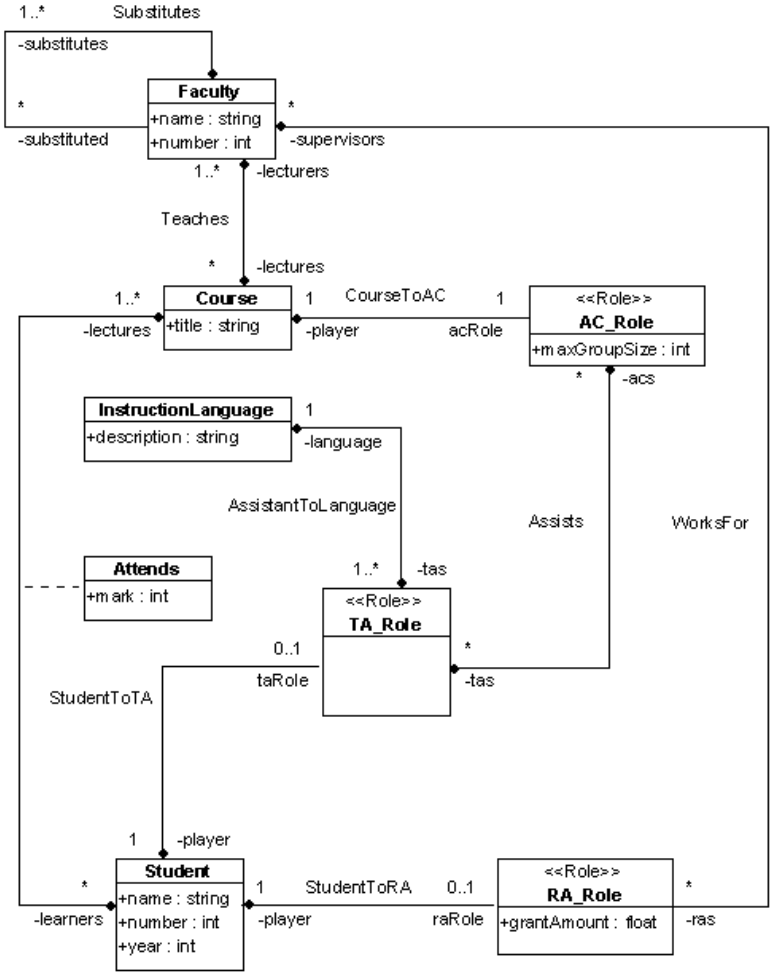

Figura 98. Exemplo de Balzer, Gross e Eugster (2007), modelado em UML.

Verificações de consistência de dados, entre elas *"the year of study cannot exceed 10"* (R2, da Figura 96), *"When attending a course students can get a Mark between 1 and 6"* (R3, da Figura 96), entre outras, modeladas por Balzer, Gross e Eugster (2007) como *invariants*, não serão abordadas por motivos de simplificação. Estas podem ser modeladas em UML por meio de *constraints*, porém só podem ser implementadas em Association# por verificações escritas diretamente

nos métodos das classes, devido a esta linguagem não ter suporte à *constraints*.

Observa-se que os *association ends* pertencentes às associações têm sempre visibilidade pública e, portanto, podem ter sua notação de visibilidade omitida para facilitar a visualização desta diferença. O motivo do uso de ambas as formas de *association end ownership* é explicado adiante nesta seção.

A nomenclatura dos nomes dos *association ends* presentes na Figura 98 foi modificada em relação à Figura 97. *Association ends* que podem referenciar mais de um objeto (limite superior de multiplicidade maior que *1*) são apresentados com nomes no plural.

A informação *linguagem de instrução* poderia ser representada por um simples atributo *string*, similar a representação utilizada por Balzer, Gross e Eugster (2007). Este tipo de abordagem, apesar de funcional, pode apresentar problemas futuramente, caso a informação necessite ser decomposta em subinformações. Por exemplo: a necessidade de informar um dialeto e/ou localidade da *linguagem de instrução*. Por este motivo, criou-se uma classe especifica para esta informação, chamada *InstructionLanguage*, contendo, inicialmente, apenas um atributo *string* para descrição das instâncias (vide Figura 98). Isto só foi possível graças à conversão do nome de papel *ta* para a classe *TA_Role*, permitindo a associação de *TA_Role* com a nova classe *InstructionLanguage*. Esta conversão proporcionou também a possibilidade de mudança de cardinalidade *1..1*, existente entre o elemento interposto *instructionLanguage* e a classe *Student*, para *1..**, possibilitando a reutilização da informação (uma mesma instância de *InstructionLanguage* pode estar associada a mais de uma instância de *TA_Role*).

As classes anotadas pelo estereótipo <<*Role*>>, chamadas aqui de *classes de papel*, apresentam um padrão que consiste em ter sempre uma associação para com a *classe atuante do papel* em que esta está sinalizada pelo *association end* de nome *player*. Toda instância de uma *classe de papel* sempre tem um e apenas um *link* de associação com a *classe atuante* (multiplicidade *1..1*, ou apenas *1*). A classe atuante tem no máximo um *link* de associação com a classe que representa o papel, podendo este ser opcional (os *association ends* de nome *taRole* e *raRole*) ou obrigatório (o *association end* de nome *acRole*).

Observa-se, pela Figura 98, que um aluno (*Student*) não necessariamente tem um papel de *auxiliar de ensino* (*teaching assistant*, ou *TA*, representando pela classe *TA_Role*), ou seja, não necessariamente um aluno atua como um *auxiliar de ensino*. Quando

não atua, não há necessidade de registro da *linguagem de instrução*. Esta não-obrigatoriedade de atuação é representada pela multiplicidade *0..1* na associação entre *Student* e *TA_Role*, na extremidade da segunda classe mencionada. Da mesma forma, um aluno não necessariamente atua como *auxiliar de pesquisa* (*research assistant*, ou *RA*, representado pela classe *RA_Role*). Antes da conversão de nomes de papéis em classes, havia sempre a obrigatoriedade de um aluno ter uma *linguagem de instrução* atribuída a ele (mesmo que o valor deste atributo fosse *null*).

Conforme apresentado na Subseção 2.1.4 (sobre *association ends* na UML) e na Subseção 4.1.1.3 (sobre *association end ownership* na linguagem Association#), *association ends* pertencentes a classes são propriedades das classes opostas, acoplando nestas classes opostas o código das respectivas associações. *Association ends* pertencentes a associações, apesar de não acoplarem as associações no código das classes participantes, possuem a limitação de sempre terem visibilidade pública.

A aplicação de padrões *GRASP*[23] como *baixo acoplamento* e *não fale com estranhos*, estimulam o uso de *association ends* pertencentes a classes com a finalidade de que estas, por meio de restrições de visibilidade dos *association ends*, encapsulem as associações em que participam. Porém, conforme abordado nas Subseções 2.2.1.1 e 3.1.2.1, classes que não possuem acoplamento de associações são independentes destas, possibilitando a reutilização destas classes em outros projetos, sem a necessidade obrigatória de incluir suas associações.

Com a finalidade de exemplificar ambas as abordagens, utilizaram-se na maior parte do projeto modelado na Figura 98, os *association ends* pertencentes a classes, utilizando-se o desacoplamento de associações somente nas associações entre as *classes de papel* e *classes atuantes*, de forma que as *classes atuantes* são independentes das associações e, consequentemente, das respectivas *classes de papel*. Exemplo: na associação *StudentToTA*, o *association end* de nome *taRole*, referente à classe *TA_Role*, é uma propriedade da associação em vez de ser da classe *Student* (isto é sinalizado pela falta do ponto opaco sobre a linha da associação). Assim, *Student* não tem o acoplamento da associação *StudentToTA*. Como consequência desta prática nesta associação e em outras como *StudentToRA* e *CourseToAC*, pode-se reutilizar um subconjunto do modelo apresentado pela Figura 98, conforme apresentado pela Figura 99.

[23] Larman, 2000.

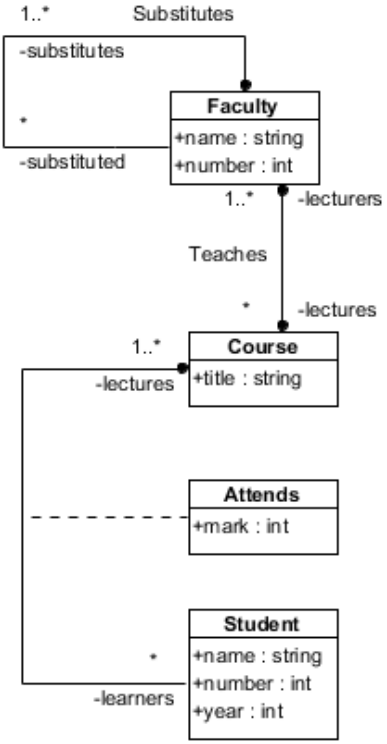

Figura 99. Reutilização de um subconjunto do modelo da Figura 98.

Observa-se que o modelo de linguagem utilizado de Balzer, Gross e Eugster (2007) não permite declaração de *association ends* pertencentes a classes e não permite a chamada de operações de modificação de *links* de associação de dentro dos métodos das classes (permite somente de dentro de *procedures* da aplicação). Isto impossibilita a aplicação dos padrões *GRASP* mencionados.

Por motivos de simplificação, será apresentada apenas a implementação do modelo completo, ilustrado pela Figura 98.

4.2.1 Implementação do Modelo

O modelo foi implementado a partir do diagrama de classes ilustrado pela Figura 98, aplicando-se diretamente no código Association#, os padrões *GRASP* *especialista*, *criador*, *controlador*, *baixo acoplamento* e *não fale com estranhos*. Este último foi aplicado

na maior parte do projeto, excetuando-se pelas associações em que o *association end* utilizado para navegação é pertencente à associação.

O resultado desta implementação foi a camada de domínio para o sistema de universidade exemplificado, em que a aplicação tem acesso somente ao *controlador fachada*[24], ilustrado pela Figura 100 e codificado parcialmente na Figura 101.

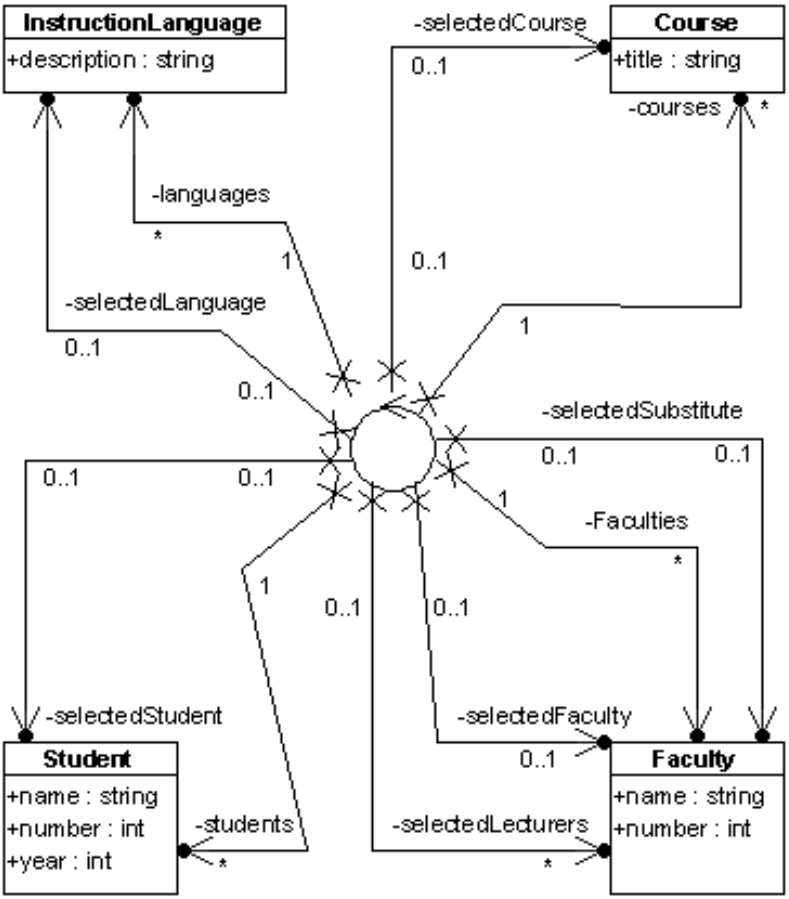

Figura 100. Controlador da camada de domínio e suas associações.

[24] Larman, 2000.

Código Association#

```
1  public class University {
2    public static readonly University INSTANCE =
3      new University();
4    private University() { }
5
6    // Association ends
7    private rolename Faculties [*] in UniversityToFaculty;
8    private rolename Courses [*] in UniversityToCourse;
9    private rolename Students [*] in UniversityToStudent;
10   private rolename SelectedFaculty [0..1] in
11     UniversityToSelectedFaculty;
12   private rolename SelectedSubstitute [0..1] in
13     UniversityToSelectedSubstitute;
14   private rolename SelectedCourse [0..1] in
15     UniversityToSelectedCourse;
16   private rolename SelectedLecturers [*] in
17     UniversityToSelectedLecturers;
18   private rolename SelectedStudent [0..1] in
19     UniversityToSelectedStudent;
20   private rolename SelectedLanguage [0..1] in
21     UniversityToSelectedLanguage;
22   private rolename Languages [*] in UniversityToLanguage;
23
24   // Cadastro de Professores
25   public void RegisterFaculty(string name, int number)
26     {...}
27   public bool SelectFaculty(int number) {...}
28   public void EditSelectedFaculty(string newName,
29     int newNumber){...}
30   public void DeleteSelectedFaculty() {...}
31   public string GetSelectedFacultyName() {...}
32   public int GetSelectedFacultyNumber() {...}
33   public IEnumerable<int> GetFacultiesNumbers() {...}
34
35   // Vínculo de Professores com seus Substitutos
36   public void SelectSubstitute(int substituteNumber) {...}
37   public string GetSelectedSubstituteName() {...}
38   public int GetSelectedSubstituteNumber() {...}
39   public void AddSelectedSubstituteToSelectedFaculty()
40     {...}
41   public void emoveSelectedSubstituteFromSelectedFaculty()
42     {...}
43   public IEnumerable<int>
44     GetSubstitutesNumbersFromSelectedFaculty() {...}
45   ...
46 }
```

Figura 101. Resumo do código Association # da classe controladora *University*.

Para manter o padrão C#/Association# de nomes de métodos, atributos e *association ends*, estes iniciam sempre com a primeira letra maiúscula, na escrita de código.

A Figura 102 apresenta o código simplificado da classe *Faculty*. Nesta, assim como nas demais classes, não há a necessidade de encapsulamento dos atributos e criação de métodos *setAtributo* e *getAtributo*, porque são utilizados *properties* em vez de *fields* para a representação de atributos. Se houver necessidade futura de adicionar controles e/ou operações sobre a atribuição e a recuperação de valores, basta apenas definir um corpo para as operações *set* e *get* das *properties*, da mesma forma que em C#.

Código Association#

```
1  public class Faculty {
2     // atributos
3    public string Name { set; get; }
4    public int Number { set; get; }
5
6    // association ends
7    private rolename Substitutes [*] in Substitutes;
8    private rolename Substituted [*] in Substitutes;
9    private rolename Lectures [*] in Teaches;
10
11   // operações
12   public Faculty(string name, int number) {...}
13   public bool HasSubstitutes() {...}
14   public IEnumerable<int> GetSubstitutesNumbers() {...}
15   public void RemoveAllSubstitutes() {...}
16   public void AddSelectedSubstitute(Faculty substitute)
17      {...}
18   public void RemoveSubstitute(Faculty substitute) {...}
19   public void RemoveAllSubstituted() {...}
20 }
```

Figura 102. Resumo do código Association# da classe *Faculty*.

Observa-se que não há na classe *Faculty*, nenhuma referência à associação *WorksFor*. Isto porque, o *association end* de nome *ras* pertence à associação e não a classe. Houve a necessidade de uma pequena modificação do código desta classe em relação ao modelo: o *association end* de nome *Substitutes* teve sua multiplicidade alterada de *1..** para * porque é esperado que durante a execução do sistema, primeiro cadastrem-se os professores para que depois se determine quem será substituto de quem. Porém, a regra de negócio que todo professor deve ter pelo menos um substituto é mantida através da verificação da existência de substitutos (chamada do método *HasSubstitutes*) ao tentar vincular um professor a um curso. Se o professor não tem substitutos, não é possível suceder tal vínculo.

A Figura 103 apresenta o código simplificado das classes *Course* (linhas 1 a 18) e *Student* (linhas 20 a 39). Similar à classe *Faculty* em

relação à associação *WorksFor*, a classe *Course* não possui referências à associação *CourseToAC* e a classe *Student* não possui referências às associações *StudentToRA* e *StudentToTA*.

Código Association#

```
1  public class Course {
2     // Atributos
3     public string Title { set; get; }
4
5     // Association ends
6     private rolename Learners [*] in Attends;
7     private rolename Lecturers [1..*] in Teaches;
8
9     // Operações
10    public Course(string title,
11       IEnumerable<Faculty> lecturers) {...}
12    public void RemoveAllLecturers() {...}
13    public void AddLecturer(Faculty lecturer) {...}
14    public void RemoveLecturer(int lecturerNumber) {...}
15    public IEnumerable<int> GetLecturersNumbers() {...}
16    public void AddLearner(Student learner) {...}
17    public void RemoveLearner(int studentNumber) {...}
18    public IEnumerable<int> GetLearnersNumbers() {...} }
19
20 public class Student {
21    // atributos
22    public string Name { set; get; }
23    public int Number { set; get; }
24    public int Year { set; get; }
25
26    // association ends
27    private rolename Lectures [1..*] in Attends;
28
29    public Student(string name, int number, int year) {...}
30
31    // operações
32    public void AddLecture(Course course){...}
33    public void RemoveLecture(Course course) {...}
34    public void SetMarkOnLecture(Course course, int mark)
35 {...}
36    public int GetMarkOnLecture(Course course) {...}
37    public IEnumerable<string> GetCoursesTitles() {...}
38    public void RemoveAllLectures() {...}
39    public bool IsAttendingCourse() {...} }
```

Figura 103. Resumo do código Association# das classes *Course* e *Student*.

Por meio do uso do padrão controlador, a aplicação só modifica e acessa informações da camada de domínio pela chamada de métodos da única instância (*singleton*) da classe controladora *University*.

Empregando o *padrão não fale com estranhos*, as operações da classe controladora encadeiam chamadas a métodos de instâncias das classes associadas a ela. Estas, por sua vez, executam a operação em seu próprio escopo e/ou delegam essa chamada a instâncias de outras classes associadas a estas, assim sucessivamente. Por exemplo: o cadastro de um professor – que deve resultar na criação de uma instância de *Faculty* e na posterior associação desta com o *singleton University*, pela associação *UniversityToFaculty* – é feito pela chamada do método *University.RegisterFaculty* (linha 24 da Figura 101). Este é responsável por criar a nova instância e associar ela ao *singleton*.

Um exemplo mais complexo, em que a operação é delegada para além da instância da classe controladora, é: a vinculação de professores substitutos. Está é feita pela prévia seleção do professor a ser substituído (método *University.SelectFaculty*, linha 25 da Figura 101) e do professor a ser o substituto (método *University.SelectSubstitute*, linha 33 da Figura 101), seguido da chamada do método *University. AddSelectedSubstituteToSelectedFaculty* (linha 36 da Figura 101). Este último método, chama o método *Faculty.AddSelectedSubstitute* (linha 16 da Figura 102) da instância selecionada como professor a ser substituído, passando como argumento o professor selecionado como substituto. Então, neste método, a instância do professor a ser substituído adiciona em sua lista de substitutos, a instância do professor recebida como argumento, por meio da chamada do association end de nome *Substitutes* (linha 7 da Figura 102).

O código implementado em Association# para esta camada de domínio, assim como o código C# gerado a partir deste por tradução manual, e o código do teste de unidade deste, estão contidos integralmente no Apêndice C.

4.3 TESTE DE DESEMPENHO

Com a finalidade de medir o desempenho da linguagem Association#, foram feitos testes de comparação de desempenho entre esta e os padrões e bibliotecas apresentados neste trabalho. Foram utilizados para os testes, o *pattern mutual friends*, o *pattern de Gessenharter* e a biblioteca RAL. Dos demais trabalhos correlatos, as linguagens de programação e a biblioteca Noiai não estão disponíveis para testes.

Esta seção apresenta a heterogeneidade de linguagens de programação utilizadas nos testes (Subseção 4.3.1), a implementação destes testes (Subseção 4.3.2) e seus resultados (Subseção 4.3.3).

4.3.1 Heterogeneidade de Linguagens de Programação

Os testes envolvendo os *patterns* foram implementados em C# para que a comparação de desempenho com Association# não envolvesse questões como as diferenças entre as linguagens quanto à otimização de código, implementação das bibliotecas de utilitários (coleções, listas, dicionários etc), a forma de execução do código executável (interpretação ou execução direta), entre outros.

Entretanto, RAL é uma biblioteca implementada em AspectJ/Java, sendo necessário utilizar um compilador AspectJ para gerar código executável a ser interpretado por uma máquina virtual Java comum.

Por haver esta heterogeneidade de linguagens de programação envolvidas, houve a necessidade de estabelecer uma forma de equivalência de desempenho entre programas escritos em C# e em AspectJ/Java, para que a comparação entre Association# e RAL fosse possível. Isto foi feito implementando o *pattern mutual friends* em C# e em Java, atribuindo ao desempenho destes o status de *referência*. Assim, as abordagens implementadas em C# (Association# e a implementação do *pattern de Gessenharter*) têm sua medida de desempenho relativa ao desempenho de *referência* C#, enquanto que RAL tem sua medida de desempenho relativa ao desempenho de *referência* AspectJ/Java. A comparação, então, é feita, não somente entre as medidas absolutas, mas também entre as medidas relativas de desempenho.

4.3.2 Implementação dos Testes

Cada uma das abordagens (Association#, *pattern mutual friends*, *pattern de Gessenharter* e RAL) foi utilizada para implementar uma associação e uma classe de associação, modeladas pela Figura 104 e Figura 105, respectivamente.

Figura 104. Associação AB entre as classes A e B, sem restrições de limites de multiplicidade.

A Figura 106 apresenta a implementação da classe *A* de acordo com a modelagem na Figura 105 (classe de associação), utilizando o *pattern mutual friends*.

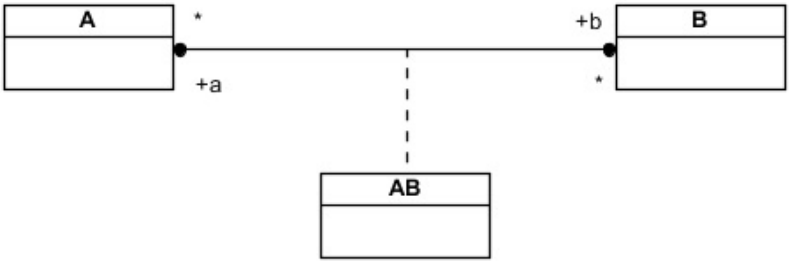

Figura 105. Classe de Associação AB entre as classes A e B, sem restrições de limites de multiplicidade.

Código C#

```
1  public class A {
2     private readonly Dictionary<B, AB> b =
3        new Dictionary<B,AB>();
4     public AB addB(B b) {
5        AB ab = null;
6        if (b != null) { ab = addAB(this, b); }
7        return ab; }
8     public bool removeB(B b) {
9        return b != null && removeAB(this, b); }
10    public IEnumerable<B> getB(){  return this.b.Keys; }
11    public IEnumerable<AB> getAB(){ return this.b.Values; }
12    internal static AB addAB(A a, B b) {
13       AB ab = null;
14       if (!a.b.ContainsKey(b)) {
15          ab = new AB(); a.b.Add(b, ab); b.a.Add(a, ab);
16       }
17       return ab; }
18    internal static bool removeAB(A a, B b) {
19       bool removed;
20       if (removed = b.a.Remove(a)) { a.b.Remove(b); }
21       return removed; }  }
```

Figura 106. Implementação em C# da classe *A* modelada na Figura 104 utilizando o *pattern mutual friends*.

Código C#

```
1  public class B {
2     internal readonly HashSet<A> a = new HashSet<A>();
3     ... }
```

Figura 107. Implementação parcial em C# da classe *B* modelada na Figura 104 utilizando o *pattern mutual friends*.

A Figura 107 apresenta a implementação parcial da classe *B*. Utilizou-se dois dicionários para indexar as instâncias da classe de associação *AB* em função das instâncias da classe oposta, decompondo a

associação *AB* em estruturas das classes participantes *A* e *B*. O código de implementação da associação, seguindo o *pattern mutual friends*, é similar a este, diferenciando-se pelo uso de dois *objetos de coleção* em vez de dicionários, entre outras características.

A Figura 108 apresenta o código escrito em C# que executa operações sobre a classe de associação apresentada pela Figura 106 e Figura 107. O código C# para executar operações sobre a associação é idêntico a este, excluindo as interações sobre a consulta de instâncias da classe de associação (linhas 16 a 18 da Figura 108). Os códigos equivalentes para as implementações em Association# estão nos comentários *in-line* (linhas 8, 13, 17 e 21 da Figura 108). Os códigos que executam operações sobre a implementação no *pattern de Gessenharter* é similar a estes.

Código C#

```
 1  DateTime beginning = DateTime.Now;
 2
 3  A a = new A();
 4  List<B> listB = new List<B>();
 5
 6  for (int i = 0; i < INTERACTIONS; i++) {
 7    B b = new B();
 8    a.addB(b); // a#b.Add(b);
 9    listB.Add(b);
10  }
11
12  for (int i = 0; i < INTERACTIONS; i++) {
13    a.getB(); // a#b.Get();
14  }
15
16  for (int i = 0; i < INTERACTIONS; i++) {
17    a.getAB(); // a#b.GetAssociationLink();
18  }
19
20  for (int i = 0; i < INTERACTIONS; i++) {
21    a.removeB(listB[i]); // a#b.Remove(b);
22  }
23
24  log("Tempo total " + DateTime.Now.Subtract(beginning).
25    Duration().TotalMilliseconds + " ms");
```

Figura 108. Código escrito em C# para o programa que executa operações sobre a classe de associação.

Por utilizar uma biblioteca implementada em AspectJ, o código que executa as operações sobre as implementações em RAL para associação e classe de associação, apresentado pela Figura 109, foi também implementado em AspectJ. Este conjunto (biblioteca e

implementações) foi compilado, resultando em programa Java comum (por tradução do compilador AspectJ).

Código AspectJ

```
long beginning = new
GregorianCalendar().getTimeInMillis();

A a =  new A();
ArrayList<B>  listB = new ArrayList<B>();

for (int i = 0; i < INTERACTIONS; i++) {
   B b = new B();
   a.addB(b);
   listB.add(b);
}

for (int i = 0; i < INTERACTIONS; i++) {
   a.getB();
}

for (int i = 0; i < INTERACTIONS; i++)  {
   a.getAB();
}

for (int i = 0; i < INTERACTIONS; i++)  {
   a.removeB(listB.get(i));
}

log("Tempo total " + MILLISECONDS_FORMAT.format(
   new GregorianCalendar().getTimeInMillis()- beginning)
   + " ms");
```

Figura 109. Código escrito em C# para o programa que executa operações sobre a classe de associação.

Devido à necessidade de heterogeneidade de linguagens, além da implementação em C# para o *pattern de Gessenharter*, incluiu-se a implementação do mesmo também para Java com a finalidade de verificar qual implementação tem melhor desempenho. Os códigos fonte completos destas implementações, para todas as abordagens, encontram-se no Apêndice D.

Os códigos fonte escritos em C# foram compilados pelo compilador *mcs.exe*, presente no *framework* Mono, versão 2.10[25]. O compilador presente na biblioteca *aspectjtools.jar*, pertencente ao pacote AspectJ versão 1.6.11[26], foi utilizado para compilar os códigos fonte escritos em AspectJ. Este compilador foi executado por uma máquina

[25] Mono Project, 2011.
[26] The Eclipse Foundation, 2011.

virtual Java, presente no *framework* Java Standard Edition Development Kit (JDK), versão 1.7.0[27]. O modelo do computador utilizado para compilar e executar os testes é um x86 Pentium Dual-Core T4400 2.2 GHz com 2 GB de memória RAM e sistema operacional *Windows 7 Starter Edition*.

4.3.3 Resultados do Teste

A Tabela 1 apresenta o resultado da medição dos tempos de execução de cada implementação para as operações sobre a associação e a classe de associação. Estes tempos de execução, denominados também de *tempos absolutos de execução*, são apresentados em milissegundos (ms). Uma terceira informação chamada de *variação* apresenta, em percentual, o acréscimo ou decréscimo do tempo de execução das operações sobre a classe de associação em relação ao tempo de execução das operações sobre a associação.

Tabela 1. Medidas de tempo de execução de todas as abordagens testadas.

Implementação	Associação Tempo (ms)	Classe de Associação Tempo (ms)	Variação
Pattern mutual friends (C#)	218	163	-25,23%
Association# (C#)	704	591	-16,05%
Pattern de Gessenharter (C#)	813.537	1.622.924	99,49%
Pattern mutual friends (Java)	273	283	3,66%
RAL (AspectJ/Java)	293	176.014	59.973,04%
Pattern de Gessenharter (Java)	1.083.058	2.154.698	98,95%

Notas: A coluna *variação* contém o acréscimo ou decréscimo do tempo de execução das operações sobre a classe de associação em relação ao tempo de execução das operações sobre a associação.

Utilizando a implementação *pattern mutual friends* em C# como *referência* para todas as implementações em C#, a Tabela 2 apresenta o *tempo relativo de execução* para as implementações em Association# e no *pattern de Gessenharter* (em C#). O *tempo relativo de execução* de uma implementação é calculado pelo percentual do seu *tempo absoluto de execução* em relação ao *tempo absoluto de execução* da implementação utilizada como *referência*. Por exemplo: o *tempo absoluto de execução* das operações sobre a associação implementada em Association# é equivalente a 322,94% (704 ÷ 218 × 100) do *tempo absoluto de execução* das operações sobre a associação implementada

[27] Oracle, 2011.

no *pattern mutual friends* (a *referência*). Sendo assim, o tempo relativo de execução das operações sobre a associação implementada em Association# é 322,94%.

Tabela 2. Cálculo do tempo relativo para as abordagens em C#

Implementação	Associação		Classe de Associação	
	Tempo (ms)	Tempo Relativo	Tempo (ms)	Tempo Relativo
mutual friends (C#)	218	100,00%	163	100,00%
Association# (C#)	704	322,94%	591	362,58%
Gessenharter (C#)	813.537	373.182,11%	1.622.924	995.658,90%

Notas: O Tempo Relativo é o tempo de execução da implementação abordada comparada com o tempo de execução da *referência* (*mutual friends*). Os nomes das implementações estão abreviados por motivos de espaço.

A Tabela 3 apresenta o *tempo relativo de execução* para as implementações em Java, comparando o *pattern mutual friends,* RAL e o *pattern de Gessenharter*, utilizando o primeiro como *referência* para os demais.

Tabela 3. Cálculo do tempo relativo para as abordagens em Java

Implementação	Associação		Classe de Associação	
	Tempo (ms)	Tempo Relativo	Tempo (ms)	Tempo Relativo
mutual friends(Java)	273	100,00%	283	100,00%
RAL (AspectJ/Java)	293	107,33%	176.014	62.195,76%
Gessenharter (Java)	1.083.058	396.724,54%	2.154.698	761.377,39%

Notas: O Tempo Relativo é o tempo de execução da implementação abordada comparada com o tempo de execução da *referência* (*mutual friends*). Os nomes das implementações estão abreviados por motivos de espaço.

Tabela 4. Tempos relativos para todas as implementações em ambas as linguagens de programação.

Implementação	Associação Tempo Relativo	Classe de Associação Tempo Relativo
RAL (AspectJ/Java)	107,33%	62.195,76%
Association# (C#)	322,94%	362,58%
Pattern de Gessenharter (C#)	373.182,11%	995.658,90%
Pattern de Gessenharter (Java)	396.724,54%	761.377,39%

Notas: As abordagens utilizadas como *referência* em ambas as linguagens (C# e Java) foram omitidas por terem seu tempo relativo igual a 100%.

A Tabela 4 apresenta a reunião das medidas de *tempo relativo de execução* de todas as abordagens, exceto as medidas de *tempo relativo de execução* das abordagens utilizadas como *referência*, independente da linguagem de programação utilizada para a implementação.

A Tabela 1 demonstra que a linguagem Association#, assim como a implementação em C# do *pattern mutual friends*, ao contrário do que se esperava, obtiveram menor desempenho sobre a execução de operações na associação quando comparado com a execução de operações sobre a classe de associação.

Observa-se pela Tabela 1 que apesar de RAL possuir o *tempo absoluto de execução* menor que Association# quanto à execução de operações sobre a associação, o mesmo não ocorre na execução de operações sobre a classe de associação. No primeiro caso, ao comparar o *tempo absoluto de execução*, o desempenho de Association# é equivalente a 204,27% (704 ÷ 293 × 100) do desempenho de RAL, porém no segundo caso, o desempenho de Association# é de 0,34% (591 ÷ 176.014 × 100) do desempenho de RAL.

Esta situação repete-se ao comparar o *tempo relativo de execução*. De acordo com a Tabela 1, o desempenho da linguagem Association# na execução da associação é 300,89% (222,94 ÷ 7,33 × 100) do desempenho de RAL, enquanto na execução da classe de associação é de 0,58% (262,58 ÷ 62.095,76 × 100).

Ainda de acordo com a Tabela 1, observa-se que na implementação em RAL, o *tempo absoluto de execução* das operações sobre a classe de associação é 59.973,04% menor do que o *tempo absoluto de execução* das operações sobre a associação. Esta variação é muito grande quando comparado com as demais abordagens, em que a *variação* máxima, em módulo, é de 99,49% (*pattern de Gessenharter* implementado em C#).

Dentre as abordagens analisadas, excluindo as implementações utilizadas como *referências*, a biblioteca RAL possui o melhor desempenho na execução de operações sobre a associação, enquanto a linguagem Association# possui o melhor desempenho na execução de operações sobre a classe de associação. O *pattern de Gessenharter* é o que possui o pior desempenho, variando o *tempo absoluto de execução* entre 813.537 ms e 2.154.698 ms, e o *tempo relativo de execução* entre 373.182,11% e 995.658,90%, em ambas as implementações (Java e C#) para ambos os modelos (associação e classe de associação), conforme apresentado pela Tabela 1 e Tabela 4.

4.3.3.1 Interpretação dos Resultados

Os resultados obtidos pela execução do testes (vide Subseção 4.3.3) são analisados a seguir, levando em consideração a implementação das abordagens utilizadas (Association#, RAL e *pattern de Gessenharter*).

4.3.3.1.1 Association#

Conforme apresentado na Subseção 4.3.3, a linguagem Association#, assim como a implementação em C# do *pattern mutual friends* obtiveram menor desempenho sobre a execução de operações na associação quando comparado com a execução de operações sobre a classe de associação. Possivelmente isto seja devido à forma como foram implementadas as operações de consulta de objetos *linkados* (*Get* em Association# e *GetB* no *pattern mutual friends*) nas associações. Estas operações instanciam, em cada vez que são executadas, um conjunto imutável que armazena e protege o conjunto interno de objetos *linkados*. Isto não acontece nas operações em classes de associação porque o dicionário interno que armazena os pares chave-valor, contendo cada um o objeto *linkado* e a referida instância da classe de associação, possui métodos que já retornam conjuntos imutáveis de chaves e de valores (que provavelmente já existem e não necessitam ser instanciados a cada chamada destes métodos). O mesmo não acontece com o *pattern mutual friends* implementado em Java porque neste é necessário sempre instanciar os conjuntos e coleções imutáveis nas operações mencionadas, tanto para associação quanto para classe de associação.

As perdas de desempenho da linguagem Association# em comparação com o *pattern mutual friends* implementado em C#, apresentadas pela Tabela 2, são justificáveis. A implementação em *mutual friends* apenas sincroniza os *association ends* em ambos os lados da associação, sem condicionais que testam multiplicidade e sem haver sincronização *thread-safety* (uso de blocos *lock*[28]). Já a implementação em Association# sempre executa chamadas ao método de validação dos objetos da classe *Multiplicity*, mesmo que estes representem multiplicidade sem limites inferior e superior. Este tipo de teste é executado em todas as operações, incluindo a de instanciação, exceto as de consultas.

[28] Microsoft, 2011.

Todas as operações dos *association ends* em Association#, exceto as de consulta, utilizam chamadas a métodos *delegates* que podem variar conforme a situação (dentro ou fora de um bloco atômico), diferentemente da implementação no *pattern mutual friends* que utiliza métodos com endereços resolvidos em tempo de compilação.

4.3.3.1.2 Biblioteca RAL

A considerável diferença entre os desempenhos de execução da associação e da classe de associação é devida as diferenças de implementação entre ambas. A associação especializa o aspecto *ral.util.SimpleStaticRel*, enquanto a classe de associação especializa o aspecto *ral.util.StaticRel*. Possivelmente, o aspecto *StaticRel* possua em algum ponto de sua implementação, um ou mais métodos pouco eficientes quando comparados com a implementação *SimpleStaticRel*. No entanto, por não haver uma análise aprofundada do código fonte destes aspectos, não há informação precisa de que pontos da implementação de *StaticRel* causam esta considerável diferença.

4.3.3.1.3 Pattern de Gessenharter

O baixo desempenho do *pattern de Gessenharter*, comparado com as demais abordagens, é devido à forma de armazenamento dos *links* de associação e consequente consulta dos mesmos. Estes são armazenados em uma coleção de tuplas, armazenada no *singleton* da associação, em que, quando há a necessidade de consulta, é feita uma busca sequencial. Nesta busca, todas as tuplas que satisfazem a condição da pesquisa vão sendo armazenadas em uma nova coleção que será retornada como resultado da consulta. Este processo é muito lento quando comparado com os processos equivalentes do *pattern mutual friends* e da linguagem Association#. Nestes últimos, não há pesquisa na implementação da associação, e há apenas uma pesquisa simples em um dicionário no caso da implementação da classe de associação.

4.4 LIMITAÇÃO: O PROBLEMA DO ASSOCIATION END NÃO NAVEGÁVEL

Association ends, mesmo os não navegáveis, mantém referências das instâncias da classe oposta associadas à instância da classe proprietária. Em uma associação navegável somente de *a* para *b*, ocorre um problema quando *a* perde sua última referência no programa, porém

b ainda continua sendo referenciado. *b* não tem visibilidade para *a*, porém seu *association end* mantém um referência para *a* e sendo assim, *a* jamais é coletado pelo *garbage collector* enquanto *b* não for coletado.

Diferentemente do *problema da referência perdida*, *b* não é um *singleton*, podendo ser coletado num momento posterior, antes do fim do programa.

5 CONSIDERAÇÕES FINAIS

O presente capítulo apresenta as contribuições, discussões e conclusões deste trabalho de pesquisa.

5.1 CONTRIBUIÇÕES

Entre as principais contribuições deste trabalho, destacam-se:

a) A proposta de uma linguagem que implementa associações nativamente com a mesma semântica da UML 2, abordando multiplicidade, navegabilidade, visibilidade, *association end ownership* e especialização de associações.
b) A implementação de associações com limite inferior de multiplicidade que soluciona em uma única abordagem os problemas de *visibilidade global*, *referência perdida* e *destruição de objetos obrigatoriamente associados*.

Trabalho	**Abstração de Associações**	**Multiplicidade Mínima**	**Visibilidade**	**Navegabilidade**	**Association End Ownership**	**Especialização de Associações**	**Classe de Associação**	**Role Name**
Linguagem DSM	X							X
Linguagem de Albano *et al*	X	X				X	X	X
Pattern de Génova *et al*		X	X	X				X
Linguagem RelJ	X					X	X	
Biblioteca RAL	X						X	
Modelo de Balzer *et al*	X	X					X	X
Biblioteca Noiai	X					X	X	X
Pattern de Gessenharter		X	X	X	X		X	X
Linguagem Association#	X	X	X	X	X	X	X	X

Quadro 3. Comparação dos trabalhos correlatos com Association#, quanto à existência dos aspectos.

A primeira contribuição é ilustrada pelo Quadro 3 – uma versão simplificada do Quadro 2 acrescida da análise da linguagem Association# (última linha) quanto à presença dos aspectos (cada uma das colunas). Association# possui todos os aspectos analisados.

O Quadro 4 – uma versão do Quadro 1 acrescida da análise da linguagem Association# – ilustra a segunda contribuição: Association# não apresenta nenhum dos três problemas identificados nos trabalhos correlatos que implementam limite inferior de multiplicidade.

Trabalho	Visibilidade Global	Referência Perdida	Destruição Obj. Obrig. Assoc.
Linguagem de Albano *et al*	X		X
Modelo de Balzer *et al*	X		X
Pattern de Gessenharter		X	X
Linguagem Association#			

Quadro 4. Comparação dos trabalhos correlatos com Association#, quanto à ocorrência dos problemas.

Outras contribuições, porém de menor relevância, podem ser enumeradas:

a) Implementação da associação de interfaces;
b) Definição de operações atômicas para substituição de *links* de associação.

5.2 DISCUSSÃO

Esta seção apresenta discussões envolvendo o *problema do association end não navegável*, *interposição de elementos, association end ownership*, a perda de desempenho (comparado com C#) e a possibilidade de estender, com o suporte nativo de associações, outras linguagens orientadas a objetos.

5.2.1 Problema do Association End Não Navegável

Suposições de como solucionar o *problema do association end não navegável* poderiam ser levantadas. Sendo *a* um objeto que está associado ao objeto *b* por uma associação navegável somente no sentido de *a* para *b,* poderia-se propor:

a) Fazer com que *b* nunca referencie objetos da classe oposta, uma vez que eles não são navegáveis;

b) Fazer com que *b* utilize referências fracas para *a* e implementar no método destruidor de *a*, o envio de uma mensagem ao *association end* de *b*, notificando que o mesmo foi coletado e portanto sua referência fraca deve ser removida deste.

A aplicação do exposto na primeira suposição permite que os objetos não mais acessíveis sejam coletados, porém impede a determinação da cardinalidade em *b* e, sendo assim, impede a validação com a sua multiplicidade. Se o *association end* não possui limite de multiplicidade, esta solução pode ser aplicada, caso contrário não.

A aplicação da segunda suposição, apesar de permitir a coleta dos objetos não mais acessíveis pelo programa e de permitir validar a cardinalidade, não funciona corretamente porque a coleta do objeto e execução de seu método destruidor nunca acontecem no exato momento onde a última referência forte foi perdida. Se essa remoção violar a regra de multiplicidade, uma exceção será levantada na *thread* executada pelo *garbage collector,* em qualquer momento posterior a perda da última referência, tornando-se difícil reconhecer onde o comando que violou a regra foi executado.

Uma possível solução para este problema poderia ser a implementação de uma funcionalidade no *garbage collector* em que permita monitoramento diferenciado para objetos específicos. Este monitoramento especial permitiria a coleta de um objeto no exato momento em que este perde a última referência. A execução da operação de coleta seria executada pela *thread* que removeu a última referência, possibilitando assim, que seja identificado o local exato onde o problema ocorreu.

Utilizando esta nova funcionalidade do *garbage collector* em conjunto com a segunda solução proposta, resolveria o problema. Entretanto, estudos precisam ser feitos sobre a possibilidade de implementação desta funcionalidade e sobre o impacto desta no desempenho de execução.

5.2.2 Interposição de elementos

Conforme já mencionado na Seção 4.2, a UML não possui mecanismos que possibilitem a *interposição de elementos.* Consequentemente, a linguagem Association# também não. Em UML e em Association#, nomes de papel (*association ends*) são propriedades e não objetos, não possibilitando estes declarar atributos e métodos

próprios. Associações não possibilitam inserir atributos e operações nas classes participantes.

Código Modelo Balzer et al

```
1  class Student {...}
2  class Course {...}
3  class Faculty {...}
4
5  relationship Attends
6    participants (Student learner, Course lecture) {
7      int mark;
8  }
9
10 relationship Assists
11   participants (Student ta, Course course) {
12     String >ta instructionLanguage;
13 }
14
15 relationship WorksFor
16   participants (Student ra, Faculty supervisor) {
17     int >ra grantAmount;
18 }
```

Figura 110. Exemplo de *interposição de elementos*.
Fonte: Balzer, Gross e Eugster (2007)

Apesar de ser relevante por diminuir o acoplamento entre classes e suas associações – discussão também presente nos trabalhos de Pearce e Noble (2006a, 2006b) – a *interposição de elementos* não é totalmente eficaz e flexível quando comparado com uma solução possível em UML e/ou Association#. A Figura 110 apresenta um exemplo do uso de *interposição de elementos*, presente no trabalho de Balzer, Gross e Eugster (2007). Este é um subconjunto do modelo utilizado na Seção 4.2.

A Figura 111 apresenta um diagrama de classes da UML com a mesma semântica do modelo com *interposição de elementos* presente na Figura 110. Os nomes de papeis *ra* (*research assistant*) e *ta* (*teaching assistant*) foram transformados nas classes *TA_Role* e *RA_Role*, respectivamente (similar a transformação feita na Seção 4.2). Modelados em UML (bem como em Association#), além de possuírem atributos, essas entidades mapeadas como classes podem, dependendo dos requisitos, serem associadas a outras classes. Isto é demonstrado na Seção 4.2 em que este mesmo modelo foi convertido para UML. Estes nomes de papeis convertidos para classes podem ter suas multiplicidades para com a classe *Student*, iguais a *0..1* na extremidade

de *Student*. Este exemplo é posto em prática na Figura 98, em que nem todo *Student* atua como *TA* e/ou um *RA*.

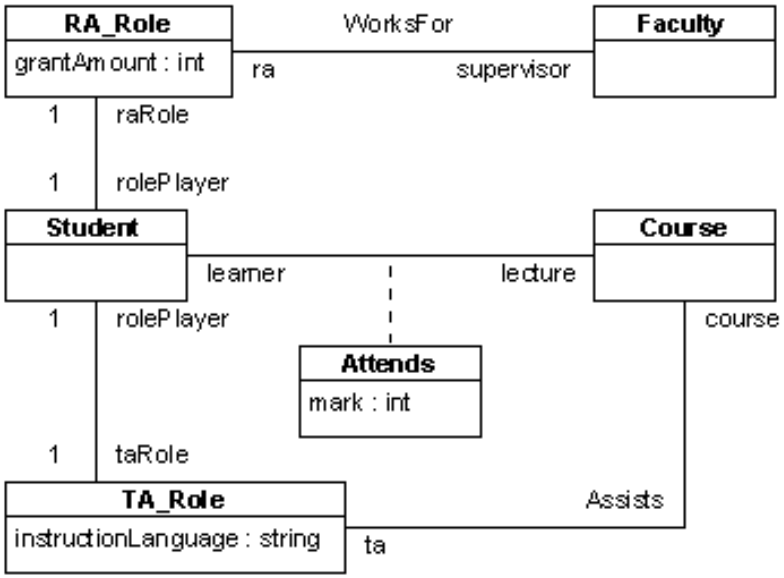

Figura 111. Exemplo de solução em UML para o exemplo de *interposição de elementos*.

Na modelagem de *interposição de elementos*, não há como alterar a cardinalidade dos nomes de papeis com as classes, que é sempre igual a 1, e não há como associar estas estruturas a outras classes.

5.2.3 Association End Ownership

Algumas questões poderiam ser levantadas quanto à necessidade de existir duas opções de posse do *association end* na linguagem Association#, que não implicam em diferenças na execução do programa. As respostas para estas possíveis questões encontram-se no nível de padrões de projeto, e não de execução de programa.

Um *association end* pertencente a uma classe é uma estrutura que faz parte do código da classe e, sendo assim, faz com que esta fique dependente da existência do código da associação referenciada pelo *association end*.

Segundo a discussão presente nos trabalhos de Pearce e Noble (2006a, 2006b), referências à associação no código de uma classe geram

maior acoplamento e menor reusabilidade de código. Entretanto, a ausência do suporte a mecanismos de *interposição de elementos,* obriga a classe acoplar as classes associadas a ela, em que ela tem a responsabilidade de enviar mensagens, quando implementando padrões *GRASP* como *não fale com estranhos*. Nestes, faz-se necessário que cada classe proteja suas associações de forma que somente elas tenham acesso direto às instâncias associadas. Quando instâncias de outras classes que possuem visibilidade a esta, precisam acessar essas associações, elas chamam métodos delegados da classe associada. Desta forma, dificilmente a classe não fará chamadas aos seus nomes de papel, impossibilitando então o não acoplamento das associações às classes. Este encapsulamento também obriga que o nome de papel da classe associada seja declarado com algum tipo de visibilidade que restrinja o acesso às demais (*private*, por exemplo). Esta restrição só é possível quando o *association end* é propriedade da classe, não da associação.

5.2.4 Perda de Desempenho

Críticas ao uso de linguagens com suporte nativo a declaração de associações poderiam ser feitas quanto à questão do desempenho. Ter exposto o código de implementação da associação no código fonte de uma aplicação, permite que este código de implementação possa ser adequado as necessidades específicas de cada associação, como por exemplo: remover a verificação de cardinalidade nos *association ends* que não possuem limite de multiplicidade, entre outros.

Entretanto, essas questões também podem ser resolvidas pelo compilador da linguagem de programação, utilizando otimização de código de acordo com análise estática no código fonte. Para o exemplo mencionado, pode-se implementar *association ends* sem verificação de cardinalidade e utilizá-los nestes casos. Esta verificação seria feita pelo compilador da linguagem que selecionaria os casos em que a possibilidade é aplicável.

Outras características da linguagem Association# que causam perda de desempenho quando comparado com a implementação de associações em código C# (sem abstração), são:

a) Sincronização de ambos os lados de uma associação mesmo quando um dos lados não é navegável e não possui limites de multiplicidade;
b) Chamadas a métodos que não são resolvidos em *compile time* (*delegates*), por *association ends* que não necessitam

diferenciar a execução de operações dentro de um bloco atômico, por não possuírem limites de multiplicidade;

c) Uso de coleções no lugar de referências simples, em associações com limite superior de multiplicidade igual a 1.

A característica *a* é eliminada desde que seja solucionado o *problema do association end não navegável*. A característica *b* pode ser eliminada pela implementação de um *association end* exclusivo que não faz testes de cardinalidade, não necessitando que as operações mudem suas execuções quando dentro de um bloco atômico.

A característica *c* poderia ser eliminada também com otimização de código. Entretanto, esta característica garante padronização da semântica dos *association ends*. Mesmo que o limite superior de multiplicidade seja 1, dentro de um bloco atômico é possível incluir um (ou mais do que um) *link* para posterior exclusão do *link* já existente. A operação *Get* sempre retorna uma coleção, independente de esta ter no máximo um elemento, padronizando o retorno desta operação.

A eliminação deste padrão alteraria não somente a semântica da possibilidade de inclusão de elementos, como também a consulta e a exclusão. Neste caso, não faria sentido a operação *Get* retornar uma coleção de elementos por não haver uma ocasião em que o *association end* possa conter mais de um elemento. Não faria sentido a operação *Remove* necessitar de um argumento que indique qual instância está sendo removida porque só pode haver uma instância referenciada.

Entretanto, utilizar assinaturas diferentes para as operações *Get* e *Remove* implica que, durante uma manutenção de código, se um limite superior de multiplicidade igual a 1 for alterado para um valor maior, todas as chamadas das operações *Get* e *Remove* do *association end* alterado deverão ser modificadas para atenderem a mudança.

5.2.5 Extensão de outras linguagens OO

Outras linguagens, como Java, poderiam ter sido utilizadas tanto para a implementação da biblioteca de apoio como para servir de linguagem base para a extensão. Porém, algumas características ausentes nesta e presentes em C#, como *properties* e possibilidade de passar parâmetros por referência para métodos, são utilizadas na implementação da biblioteca de apoio, na linguagem Association# e na aproximação semântica desta extensão de linguagem com a UML.

O uso da passagem de argumentos como referência é utilizado na destruição de objetos obrigatoriamente associados, sem o uso de um bloco atômico (ver Subseção 4.1.1.14, Figura 87). Esta característica

atua como vantagem no desempenho de execução, não havendo diferenças para a modelagem.

Properties são utilizadas na associação de interfaces, no código C# resultante de tradução (ver Subseção 4.1.1.11, Figura 75) e no código Association#. A ausência destas na implementação de associações (código C#) pode ser contornada substituindo-as por métodos. Entretanto, na codificação de interfaces em Association#, considerando que é impossível declarar variáveis de instância, elas permitem codificar atributos de forma mais intuitiva do que utilizar pares de métodos *set* e *get*.

5.3 CONCLUSÕES

Este trabalho, diferentemente dos trabalhos correlatos, apresentou uma forma de representar em código de linguagem de programação, associações da UML 2, abordando multiplicidade, navegabilidade, *association end ownership* e especialização de associações, resolvendo em uma única abordagem o *problema da destruição de objetos obrigatoriamente associados*, o *problema da visibilidade global* e o *problema da referência perdida*. Isto foi possível através da extensão de uma linguagem de programação orientada a objetos, acrescentando-se suporte nativo a associações, implementando estas (em código C# resultante de tradução) de forma similar ao *pattern mutual friends* composto por dois *objetos de coleção*.

Esta proposta de extensão de linguagem esconde do programador os detalhes de construção de associações, provendo abstração para estas, e resolve os problemas de navegabilidade, visibilidade e *association end ownership* na própria definição da extensão.

O uso desta linguagem elimina a necessidade de estabelecer *code patterns* para construção de associações, facilitando a codificação e o reconhecimento do código escrito, diminuindo a discrepância semântica entre a linguagem de programação e a linguagem de modelagem.

5.3.1 Trabalhos Futuros

Como proposta para trabalhos futuros, sugere-se estudos sobre:

a) Possibilidades de inserção de uma nova funcionalidade no *garbage collector*, que permita a este coletar objetos específicos, no exato momento em que estes perdem a última referência forte, possibilitando resolver o *problema do association end não navegável*;

b) Os conceitos da UML e as possibilidades de implementação de características dos *association ends*, como redefinição, *subsetting*, derivação, ordenação etc;
c) Inserção de associações n-árias, com verificação de limites de multiplicidade, na linguagem Association#;
d) Possibilidade de inserção de invariantes em Association# que sejam codificáveis em OCL;
e) Inserção da semântica de atributos com suas propriedades (multiplicidade, ordenação, etc) na linguagem Association#;
f) Definição da semântica operacional da linguagem Association#;
g) Implementação de um tradutor para a linguagem Association# que gere código C#.

REFERÊNCIAS

ALBANO, Antonio; GHELLI, Giorgio; ORSINI, Renzo. A relationship mechanism for a strongly typed object-oriented database programming language. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES (VLDB), 17., 1991, Barcelona. **Proceedings...** San Francisco: Morgan Kaufmann Publishers Inc, 1991. p. 565-575. ISBN: 1-55860-150-3.

BALZER, Stephanie; GROSS, Tomas R.; EUGSTER, Patrick. A relational model of object collaborations and its use in reasoning about relationships. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP), 21., 2007, Berlin. **Lectures Notes in Computer Science**. Heidelberg: Springer, 2007. v. 4609. p. 323-346. DOI: 10.1007/978-3-540-73589-2_16.

BIERMAN, Gavin M.; WREN, Alisdair. First-class relationships in an object-oriented language. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP), 19., 2005, Glasgow. **Lectures Notes in Computer Science**. Heidelberg: Springer, 2005. v. 3586. p. 262-286. DOI: 10.1007/11531142_12.

DEL CASTILLO, Carlos Ruiz. **Implementación en java de asociaciones binarias uml**. 2002. 165 p. Proyecto Fin de Carrera, Ingeniería Informática (Segundo Ciclo) - Universidad Carlos III de Madrid, Madrid, 2002. Disponível em: <http://www.ie.inf.uc3m.es/ggenova/pfc-Carlos2002.html>. Acesso em: 2 julho 2011.

GAMA, Erich *et al*. **Design patterns:** elements of reusable object-oriented software. Boston: Addison-Wesley, 1995. 395 p. ISBN : 0201633612.

GÉNOVA, Gonzalo; DEL CASTILLO, Carlos Ruiz; LLORENS, Juan. Mapping uml associations into Java code. **Journal of Object Technology**. v. 2, n. 5, p. 135-162, set. 2003. Disponível em: <http://www.jot.fm/issues/issue_2003_09/article4>. Acesso em: 12 outubro 2010. ISSN: 1660-1769

GESSENHARTER, Dominik. Mapping the uml2 semantics of associations to a java code generations model. In: ACM/IEEE INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS (MODELS), 11., 2008, Toulouse. **Lectures Notes in Computer Science**. Heidelberg: Springer, 2008. Volume 5301. p. 813-827. DOI: 10.1007/978-3-540-87875-9_56

_____. Implementing uml associations in java: a slim code pattern for a complex modeling concept. In: WORKSHOP ON RELATIONSHIPS AND ASSOCIATIONS IN OBJECT-ORIENTED LANGUAGES (RAOOL), 2., 2009, Genova. **Proceedings…** New York: ACM, 2009. p. 17-24. DOI: 10.1145/1562100.1562104.

LARMAN, Craig. **Utilizando uml e padrões:** uma introdução à análise e ao projeto orietado a objetos. Tradução de Luiz Augusto Meirelles Salgado. Porto Alegre: Bookman, 2000. 492 p. Título original: Applying uml and patterns: na introduction to object-oriented analysis and design.

MICROSOFT. **C# Programmer's Reference**. Disponível em: <http://msdn.microsoft.com/en-us/library/618ayhy6(v=vs.71).aspx>. Acesso em: 1 julho 2011.

MONO PROJECT. **Mono project:** an cross-platform, open source .net development framework. Disponível em: <http://mono-project.com/>. Acesso em: 1 julho 2011.

NOBLE, James. Basic Relationship Patterns. In: EUROPEAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMMING (EUROPLOP), 2., 1997, Irsee. **Proceedings…** Munich: Siemens AG, 1997.

OBJECT MANAGEMENT GROUP. **Unified modeling language superstructure:** version 2.3 without change bars. Maio 2010. Disponível em: <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>. Acesso em: 15 setembro 2010.

ORACLE. **Oracle technology network for java developers**. Disponível em: <http://www.oracle.com/technetwork/java/>. Acesso em: 10 outubro 2011.

ØSTERBYE, Kasper. Design of a class library for association relationships. In: ACM SIGPLAN SYMPOSIUM ON LIBRARY-CENTRIC SOFTWARE DESIGN (LCSD), 2007, Montreal. **Proceedings…** New York: ACM, 2007. p. 67-75. DOI: 10.1145/1512762.1512769.

PEARCE, David J.; NOBLE, James. Relationship aspects. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD), 5., 2006, Bonn. **Proceedings…** New York: ACM, 2006. p 75-86. DOI: 10.1145/1119655.1119668.

_____;_____. **Relationship aspects patterns**. In: EUROPEAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS (EUROPLOP), 11., 2006, Irsee. Disponível em: <http://homepages.ecs.vuw.ac.nz/~djp/files/PN-EPLOP06.pdf>. Acesso em: 21 junho 2011.

RUMBAUGH, James. Relations as semantic constructs in an object-oriented language. In: ACM CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS (OOPSLA), 2., 1987, Orlando. **Proceedings…** New York: ACM, 1987. p. 466-481. DOI: 10.1145/38765.38850.

_____; JACOBSON, Ivar; BOOCH, Grady. **The unified modeling language reference manual**. 2. ed. Boston: Addison-Wesley, 2004. 550 p. ISBN: 020130998X.

SHAH, Ashwin V. et al. Dsm: an object-relationship modeling language. In: ACM CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS (OOPSLA), 4., 1989, New Orleans. **Proceedings…** New York: ACM, 1989. p. 191-202. ACM, New York (1989). DOI: 10.1145/74877.74898.

THE ECLIPSE FOUNDATION. **The aspect project**: crosscutting objects for better modularity. Disponível em <http://eclipse.org/aspectj/>. Acesso em: 10 outubro 2011.

APÊNDICE A – EXTENSÃO DA GRAMÁTICA C#

A seguir, as inserções e modificações na gramática original (Linguagem C#), para a criação da gramática da Linguagem Association#.

```
<type-declaration>
   ::=  <class-declaration>
   |    <struct-declaration>
   |    <interface-declaration>
   |    <enum-declaration>
   |    <delegate-declaration>
   |    <association-declaration>
   |    <associationclass-declaration>

<association-declaration>
   ::=  [ <association-modifiers> ] "association" <identifier>
        [ <association-base> ] <between-exp> ";"

<association-modifiers>
   ::=  <association-modifier>
   |    <association-modifiers> <association-modifier>

<association-modifier>
   ::=  "public"
   |    "protected"
   |    "internal"
   |    "private"
   |    "sealed"
   |    "package"

<association-base>
   ::=  ":" <association-type>

<association-type>
   ::=  <type-name>

<between-exp>
   ::=  "between" <participant-declaration> "and"
        <participant-declaration>

<participant-declaration>
   ::=  <participant-type> [ <identifier> ]
        <multiplicity-declaration>

<multiplicity-declaration>
   ::=  "[" <multiplicity-range> "]"

<multiplicity-range>
   ::=  "*"
   |    <decimal-digits> ".." "*"
   |    <decimal-digits> ".." <decimal-digits>
   |    <decimal-digits>
```

```
<associationclass-declaration>
   ::=  [ <association-modifiers> ] "association" "class"
        <identifier> [ <associationclass-base> ] <between-exp>
        <associationclass-body>

<associationclass-base>
   ::=  ":" <association-type>
   |    ":" <class-type>
   |    ":" <interface-type-list>
   |    ":" <class-type> "," <interface-type-list>
   |    ":" <association-type> "," <interface-type-list>

<associationclass-body>
   ::=  "{" [ <associationclass-member-declarations> ] "}"

<associationclass-member-declarations>
   ::=  <associationclass-member-declaration>
   |    <associationclass-member-declaration>
        <associationclass-member-declarations>

<associationclass-member-declaration>
   ::=  <constant-declaration>
   |    <field-declaration>
   |    <method-declaration>
   |    <property-declaration>
   |    <event-declaration>
   |    <indexer-declaration>
   |    <operator-declaration>
   |    <destructor-declaration>
   |    <static-constructor-declaration>
   |    <type-declaration>
   |    <rolename-declaration>
   |    <associationclass-constructor-declaration>

<associationclass-constructor-declaration>
   ::=  [ <attributes> ] <identifier> ( ) <constructor-body>

<rolename-declaration>
   ::=  [ <rolename-modifiers> ] "rolename" <identifier> "in"
        <association-type> ";"

<class-member-declaration>
   ::=  <constant-declaration>
   |    <field-declaration>
   |    <method-declaration>
   |    <property-declaration>
   |    <event-declaration>
   |    <indexer-declaration>
   |    <operator-declaration>
   |    <constructor-declaration>
   |    <destructor-declaration>
   |    <static-constructor-declaration>
   |    <type-declaration>
   |    <rolename-declaration>
```

```
<interface-member-declaration>
   ::=  <interface-method-declaration>
   |    <interface-property-declaration>
   |    <interface-event-declaration>
   |    <interface-indexer-declaration>
   |    <interface-rolename-declaration>

<interface-rolename-declaration>
   ::=  "rolename" <identifier> <multiplicity-declaration> "in"
        <association-type> ";"

<embedded-statement>
   ::=  <block>
   |    <empty-statement>
   |    <expression-statement>
   |    <selection-statement>
   |    <iteration-statement>
   |    <jump-statement>
   |    <try-statement>
   |    <checked-statement>
   |    <unchecked-statement>
   |    <lock-statement>
   |    <using-statement>
   |    <atomic-statement>

<atomic-statement>
   ::=  "atomic" <block>

<method-modifier>
   ::=  "new"
   |    "public"
   |    "protected"
   |    "internal"
   |    "private"
   |    "static"
   |    "virtual"
   |    "sealed"
   |    "override"
   |    "abstract"
   |    "extern"
   |    "atomic"
   |    "package"

<constructor-modifier>
   ::=  "public"
   |    "protected"
   |    "internal"
   |    "private"
   |    "extern"
   |    "atomic"
   |    "package"
```

APÊNDICE B – CÓDIGO DA BIBLIOTECA DE APOIO

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Internal/AssociationEnd.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using AssociationSharp.Commons;

namespace AssociationSharp.Lang
{
  public abstract class AssociationEnd<OWNER, OPPOSITE>
    : HashSet<OPPOSITE>, IInvariant
    where OWNER : class
    where OPPOSITE : class
  {
    private static readonly
      RegisterAssociationEnd<OWNER, OPPOSITE> register =
        new RegisterAssociationEnd<OWNER, OPPOSITE>();

    static AssociationEnd()
    {
      AssociationSharpSystem.
        RegisterAssociationEndType(register);
    }

    public delegate AssociationEnd<OWNER, OPPOSITE>
      AssociationEndFinder(OWNER ownerObject);

    internal readonly
      AssociationEndFinder ownerAssociationEndFinder;
    internal readonly
      AssociationEnd<OPPOSITE, OWNER>.AssociationEndFinder
```

```
      oppositeAssociationEndFinder;

  public readonly Multiplicity multiplicity;

  protected internal readonly object synchronizer;

  private WeakReference<OWNER> ownerObjectRef;
  internal OWNER ownerObject
  {
    private set { this.ownerObjectRef =
      new WeakReference<OWNER>(value); }
    get { return this.ownerObjectRef.Target; }
  }

  public AssociationEnd(
      AssociationEndFinder objectAssociationEndFinder,
      AssociationEnd<OPPOSITE, OWNER>.AssociationEndFinder
        oppositeAssociationEndFinder,
      OWNER ownerObject,
      Multiplicity multiplicity,
      object synchronizer
    )
  {
    this.ownerAssociationEndFinder =
      objectAssociationEndFinder;
    this.oppositeAssociationEndFinder =
      oppositeAssociationEndFinder;
    this.ownerObject = ownerObject;
    this.multiplicity = multiplicity;
    this.synchronizer = synchronizer;

    register.mutableInstantiation.Invoke(this);
  }

  internal virtual bool __IsValid(int incrementCount)
  {
    int count = this.Count + incrementCount;

    if (count < 0)
    {
      count = 0;
    };

    return this.multiplicity.isValid(count);
  }

  public bool IsValid()
  {
    return this.multiplicity.isValid(this.Count);
  }

  public void RemoveAll()
  {
    lock (synchronizer)
```

```
    {
      int count = this.Count;
      OPPOSITE[] array = new OPPOSITE[count];
      this.CopyTo(array);

      for (int i = 0; i < count; i++)
      {
        this.Remove(ref array[i]);
      }
    }
  }

  public virtual IEnumerable<OPPOSITE> Get()
  {
    return new UnmodifiedSet<OPPOSITE>(this);
  }

  public new virtual bool Contains(OPPOSITE oppositeObject)
  {
    return base.Contains(oppositeObject);
  }

  public new virtual int Count
  {
    get { return base.Count; }
  }

  internal virtual bool __Add(OPPOSITE oppositeObject)
  {
    return base.Add(oppositeObject);
  }

  internal bool __Remove(OPPOSITE oppositeObject)
  {
    return base.Remove(oppositeObject);
  }

  public new bool Add(OPPOSITE oppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableAdd.Invoke(this,
        oppositeObject);
    }
  }

  public new virtual bool Remove(OPPOSITE oppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableRemove.Invoke(this,
        ref oppositeObject);
    }
  }
```

```
  public bool Remove(ref OPPOSITE oppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableRemove.Invoke(this,
        ref oppositeObject);
    }
  }

  public bool Swap(OPPOSITE oldOppositeObject,
    OPPOSITE newOppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableSimpleSwap.Invoke(this,
        ref oldOppositeObject, newOppositeObject);
    }
  }

  public bool Swap(ref OPPOSITE oldOppositeObject,
    OPPOSITE newOppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableSimpleSwap.Invoke(this,
        ref oldOppositeObject, newOppositeObject);
    }
  }

  public bool Swap(OPPOSITE thisOppositeObject,
    OWNER otherOwnerObject, OPPOSITE otherOppositeObject)
  {
    lock (synchronizer)
    {
      return RegisterAssociationEnd<OWNER, OPPOSITE>.
        DoubleSwap(this,thisOppositeObject,
          otherOwnerObject, otherOppositeObject);
    }
  }
}

public abstract class AssociationEnd<OWNER, OPPOSITE, LINK>
  : Dictionary<OPPOSITE, LINK>, IInvariant
  where OWNER : class
  where OPPOSITE : class
  where LINK : class, new()
{
  public delegate AssociationEnd<OWNER, OPPOSITE, LINK>
    AssociationEndFinder(OWNER ownerObject);

  private static readonly
    RegisterAssociationEnd<OWNER, OPPOSITE, LINK> register =
      new RegisterAssociationEnd<OWNER, OPPOSITE, LINK>();
```

```
static AssociationEnd()
{
  AssociationSharpSystem.
    RegisterAssociationEndType(register);
}

internal readonly object synchronizer;
public readonly Multiplicity multiplicity;

internal readonly AssociationEndFinder
  objectAssociationEndFinder;
internal readonly
  AssociationEnd<OPPOSITE,OWNER,LINK>.AssociationEndFinder
    oppositeAssociationEndFinder;

private WeakReference<OWNER> ownerObjectRef;
internal OWNER ownerObject
{
  private set { this.ownerObjectRef =
    new WeakReference<OWNER>(value); }
  get { return this.ownerObjectRef.Target; }
}

public AssociationEnd(
  AssociationEndFinder objectAssociationEndFinder,
  AssociationEnd<OPPOSITE,OWNER,LINK>.AssociationEndFinder
    oppositeAssociationEndFinder,
  OWNER ownerObject,
  Multiplicity multiplicity,
  object synchronizer)
{
  this.objectAssociationEndFinder =
    objectAssociationEndFinder;
  this.oppositeAssociationEndFinder =
    oppositeAssociationEndFinder;
  this.ownerObject = ownerObject;

  this.multiplicity = multiplicity;
  this.synchronizer = synchronizer;
}

internal LINK createLink(OPPOSITE oppositeObject)
{
  return new LINK();
}

internal virtual bool __IsValid(int incrementCount)
{
  int count = this.Count + incrementCount;

  if (count < 0)
  {
    count = 0;
```

```
    }

    return this.multiplicity.isValid(count);
  }

  public bool IsValid()
  {
    return this.multiplicity.isValid(this.Count);
  }

  public LINK Add(OPPOSITE oppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableAdd.Invoke(this,
        oppositeObject);
    }
  }

  public new virtual bool Remove(OPPOSITE oppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableRemove.Invoke(this,
        ref oppositeObject);
    }
  }

  public bool Remove(ref OPPOSITE oppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableRemove.Invoke(this,
        ref oppositeObject);
    }
  }

  public bool Swap(OPPOSITE oldOppositeObject,
    OPPOSITE newOppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableSimpleSwap.Invoke(this,
        ref oldOppositeObject, newOppositeObject);
    }
  }

  public bool Swap(ref OPPOSITE oldOppositeObject,
    OPPOSITE newOppositeObject)
  {
    lock (synchronizer)
    {
      return register.mutableSimpleSwap.Invoke(this,
        ref oldOppositeObject, newOppositeObject);
```

```
    }
  }

  public bool Swap(OPPOSITE thisOppositeObject,
    OWNER otherOwnerObject, OPPOSITE otherOppositeObject)
  {
    lock (synchronizer)
    {
      return RegisterAssociationEnd<OWNER,OPPOSITE,LINK>.
        DoubleSwap(this, thisOppositeObject,
          otherOwnerObject, otherOppositeObject);
    }
  }

  public virtual IEnumerable<LINK> GetAssociationLink()
  {
    return this.Values;
  }

  internal LINK __Get(OPPOSITE oppositeObject)
  {
    LINK link;

    this.TryGetValue(oppositeObject, out link);

    return link;
  }

  internal virtual bool __Add(OPPOSITE oppositeObject,
    LINK link)
  {
    bool added = true;

    try
    {
      base.Add(oppositeObject, link);
    }
    catch (ArgumentNullException)
    {
      added = false;
    }
    catch (ArgumentException)
    {
      added = false;
    }

    return added;
  }

  internal virtual bool __Remove(OPPOSITE oppositeObject)
  {
    return base.Remove(oppositeObject);
  }
```

```
    public new virtual int Count
    {
      get { return base.Count; }
    }

    public void RemoveAll()
    {
      lock (synchronizer)
      {
        int count = this.Count;
        OPPOSITE[] array = this.Get().ToArray();

        for (int i = 0; i < count; i++)
        {
          this.Remove(ref array[i]);
        }
      }
    }

    public virtual IEnumerable<OPPOSITE> Get()
    {
      return this.Keys;
    }

    public virtual bool Contains(OPPOSITE oppositeObject)
    {
      return base.Keys.Contains(oppositeObject);
    }

    public virtual LINK GetAssociationLink(
      OPPOSITE oppositeObject)
    {
      LINK link;

      this.TryGetValue(oppositeObject, out link);

      return link;
    }
  }
}

/*****************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Internal/GeneralizedAssociationEnd.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
```

```
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 ************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AssociationSharp.Lang
{
  public abstract
    class GeneralizedAssociationEnd<OWNER, OPPOSITE>
      : AssociationEnd<OWNER, OPPOSITE>
    where OWNER : class
    where OPPOSITE : class
  {
    private delegate bool AdditionMethod(
      OPPOSITE oppositeObject);
    private delegate bool RemovalMethod(
      OPPOSITE oppositeObject);
    private delegate bool AssociationQueryMethod(
      OPPOSITE oppositeObject);
    private delegate IEnumerable<OPPOSITE>
      AssociationRetrievalMethod();
    private delegate int CountMethod();

    private readonly List<IEndHandler<OPPOSITE>>
      specializedEnds = new List<IEndHandler<OPPOSITE>>();
    private AdditionMethod additionMethod;
    private RemovalMethod removalMethod;
    private AssociationQueryMethod associationQueryableMethod;
    private AssociationRetrievalMethod
      associationRetrievalMethod;
    private CountMethod countMethod;


    public GeneralizedAssociationEnd(
      AssociationEndFinder objectAssociationEndFinder,
      AssociationEnd<OPPOSITE, OWNER>.AssociationEndFinder
        oppositeAssociationEndFinder,
      OWNER ownerObject,
      Multiplicity multiplicity,
      object synchronizer
    ) : base(objectAssociationEndFinder,
        oppositeAssociationEndFinder,
        ownerObject, multiplicity, synchronizer)
    {
```

```
      this.additionMethod = base.__Add;
      this.removalMethod = base.Remove;
      this.associationQueryableMethod = base.Contains;
      this.associationRetrievalMethod = base.Get;
      this.countMethod = PublicCountMethodWithoutSpecializeds;
    }

    private bool InternalRemoveWithSpecializeds(
      OPPOSITE oppositeObject)
    {
      bool removed = base.Remove(oppositeObject);

      for (int i = 0, count = this.specializedEnds.Count;
        (!removed) && i < count; i++)
      {
        removed =
          this.specializedEnds[i].Remove(oppositeObject);
      }

      return removed;
    }

    public override bool Remove(OPPOSITE oppositeObject)
    {
      return this.removalMethod(oppositeObject);
    }

    internal void registerSpecializedEnd(
      IEndHandler<OPPOSITE> specializedEnd)
    {
      this.specializedEnds.Add(specializedEnd);

      if (this.specializedEnds.Count == 1)
      {
        this.additionMethod = InternalAddWithSpecializeds;
        this.removalMethod = InternalRemoveWithSpecializeds;
        this.associationQueryableMethod =
          InternalContainsWithSpecializeds;
        this.associationRetrievalMethod =
          InternalGetWithSpecializeds;
        this.countMethod =
          InternalCountMethodWithSpecializeds;
      }
    }

    private bool InternalAddWithSpecializeds(
      OPPOSITE oppositeObject)
    {
      bool added = true;

      for (int i = 0, count = this.specializedEnds.Count;
        added && i < count; i++)
      {
        added =
```

```
        !this.specializedEnds[i].Contains(oppositeObject);
    }

    return added && base.__Add(oppositeObject);
  }

  internal override bool __Add(OPPOSITE oppositeObject)
  {
    return this.additionMethod(oppositeObject);
  }

  private int PublicCountMethodWithoutSpecializeds()
  {
    return base.Count;
  }

  private int InternalCountMethodWithSpecializeds()
  {
    int count = base.Count;

    for (int i = 0, f = this.specializedEnds.Count; i < f;
      i++)
    {
      count = count + this.specializedEnds[i].Count;
    }

    return count;
  }

  public override int Count
  {
    get { return this.countMethod(); }
  }

  private IEnumerable<OPPOSITE> InternalGetWithSpecializeds()
  {
    IEnumerable<OPPOSITE> retrieved = base.Get();

    for (int i = 0, count = this.specializedEnds.Count;
      i < count; i++)
    {
      retrieved =
        retrieved.Union(this.specializedEnds[i].Get());
    }

    return retrieved;
  }

  public override IEnumerable<OPPOSITE> Get()
  {
    return this.associationRetrievalMethod();
  }

  private bool InternalContainsWithSpecializeds(
```

```
      OPPOSITE oppositeObject)
    {
      bool contains = base.Contains(oppositeObject);

      for (int i = 0, count = this.specializedEnds.Count;
        (!contains) && i < count; i++)
      {
        contains =
          this.specializedEnds[i].Contains(oppositeObject);
      }

      return contains;
    }

    public override bool Contains(OPPOSITE oppositeObject)
    {
      return this.associationQueryableMethod(oppositeObject);
    }
  }

  public abstract
    class GeneralizedAssociationEnd<OWNER, OPPOSITE, LINK>
      : AssociationEnd<OWNER, OPPOSITE, LINK>
    where OWNER : class
    where OPPOSITE : class
    where LINK : class, new()
  {
    private delegate bool AdditionMethod(
      OPPOSITE oppositeObject, LINK link);
    private delegate bool RemovalMethod(
      OPPOSITE oppositeObject);
    private delegate bool AssociationQueryMethod(
      OPPOSITE oppositeObject);
    private delegate IEnumerable<OPPOSITE>
      AssociationRetrievalMethod();
    private delegate IEnumerable<LINK> LinkRetrievalMethod();
    private delegate LINK LinkQueryMethod(
      OPPOSITE oppositeObject);
    private delegate int CountMethod();

    private readonly List<IEndHandler<OPPOSITE, LINK>>
      specializedEnds =
        new List<IEndHandler<OPPOSITE,LINK>>();
    private AdditionMethod additionMethod;
    private RemovalMethod removalMethod;
    private AssociationQueryMethod associationQueryableMethod;
    private AssociationRetrievalMethod
      associationRetrievalMethod;
    private LinkRetrievalMethod linkRetrievalMethod;
    private LinkQueryMethod linkQueryableMethod;
    private CountMethod countMethod;


    public GeneralizedAssociationEnd(
```

```
      AssociationEndFinder objectAssociationEndFinder,
      AssociationEnd<OPPOSITE,OWNER,LINK>.AssociationEndFinder
        oppositeAssociationEndFinder,
      OWNER ownerObject,
      Multiplicity multiplicity,
      object synchronizer)
        : base(
          objectAssociationEndFinder,
          oppositeAssociationEndFinder,
          ownerObject,
          multiplicity,
          synchronizer)
    {
      this.additionMethod = base.__Add;
      this.removalMethod = base.Remove;
      this.associationQueryableMethod = base.Contains;
      this.associationRetrievalMethod = base.Get;
      this.linkQueryableMethod = base.__Get;
      this.linkRetrievalMethod = base.GetAssociationLink;
      this.countMethod = PublicCountMethodWithoutSpecializeds;
    }

    private bool InternalRemoveWithSpecializeds(
      OPPOSITE oppositeObject)
    {
      bool removed = base.Remove(oppositeObject);

      for (int i = 0, count = this.specializedEnds.Count;
        (!removed) && i < count; i++)
      {
        removed =
          this.specializedEnds[i].Remove(oppositeObject);
      }

      return removed;
    }

    public override bool Remove(OPPOSITE oppositeObject)
    {
      return this.removalMethod(oppositeObject);
    }

    internal void registerSpecializedEnd(
      IEndHandler<OPPOSITE, LINK> specializedEnd)
    {
      this.specializedEnds.Add(specializedEnd);

      if (this.specializedEnds.Count == 1)
      {
        this.additionMethod = InternalAddWithSpecializeds;
        this.removalMethod = InternalRemoveWithSpecializeds;
        this.associationQueryableMethod =
          InternalContainsWithSpecializeds;
        this.associationRetrievalMethod =
```

```
            InternalGetWithSpecializeds;
         this.linkQueryableMethod =
            InternalGetAssociationLinkWithSpecializeds;
         this.linkRetrievalMethod =
            InternalGetAssociationLinkWithSpecializeds;
         this.countMethod =
            InternalCountMethodWithSpecializeds;
      }
   }

   private IEnumerable<LINK>
      InternalGetAssociationLinkWithSpecializeds()
   {
      IEnumerable<LINK> retrieved = base.GetAssociationLink();

      for (int i = 0, count = this.specializedEnds.Count;
         i < count; i++)
      {
         retrieved = retrieved.Union(
            this.specializedEnds[i].GetAssociationLink());
      }

      return retrieved;
   }

   public override IEnumerable<LINK> GetAssociationLink()
   {
      return this.linkRetrievalMethod();
   }

   private LINK InternalGetAssociationLinkWithSpecializeds(
      OPPOSITE oppositeObject)
   {
      LINK link = base.GetAssociationLink(oppositeObject);

      for (int i = 0, count = this.specializedEnds.Count;
         link == null && i < count; i++)
      {
         link = this.specializedEnds[i].GetAssociationLink(
            oppositeObject);
      }

      return link;
   }

   public override LINK GetAssociationLink(
      OPPOSITE oppositeObject)
   {
      return this.linkQueryableMethod(oppositeObject);
   }

   private bool InternalAddWithSpecializeds(
      OPPOSITE oppositeObject, LINK link)
   {
```

```
    bool added = true;

    for (int i = 0, count = this.specializedEnds.Count;
      added && i < count; i++)
    {
      added = !this.specializedEnds[i].Contains(
        oppositeObject);
    }

    return added && base.__Add(oppositeObject, link);
  }

  internal override bool __Add(OPPOSITE oppositeObject,
    LINK link)
  {
    return this.additionMethod(oppositeObject, link);
  }

  private int PublicCountMethodWithoutSpecializeds()
  {
    return base.Count;
  }

  private int InternalCountMethodWithSpecializeds()
  {
    int count = base.Count;

    for (int i = 0, f = this.specializedEnds.Count; i < f;
      i++)
    {
      count = count + this.specializedEnds[i].Count;
    }

    return count;
  }

  public override int Count
  {
    get { return this.countMethod(); }
  }

  private IEnumerable<OPPOSITE> InternalGetWithSpecializeds()
  {
    IEnumerable<OPPOSITE> retrieved = base.Get();

    for (int i = 0, count = this.specializedEnds.Count;
      i < count; i++)
    {
      retrieved = retrieved.Union(
        this.specializedEnds[i].Get());
    }

    return retrieved;
  }
```

```
    public override IEnumerable<OPPOSITE> Get()
    {
      return this.associationRetrievalMethod();
    }

    private bool InternalContainsWithSpecializeds(
      OPPOSITE oppositeObject)
    {
      bool contains = base.Contains(oppositeObject);

      for (int i = 0, count = this.specializedEnds.Count;
        (!contains) && i < count; i++)
      {
        contains = this.specializedEnds[i].Contains(
          oppositeObject);
      }

      return contains;
    }

    public override bool Contains(OPPOSITE oppositeObject)
    {
      return this.associationQueryableMethod(oppositeObject);
    }
  }
}

/*****************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Internal/IEndHandler.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *****************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;

namespace AssociationSharp.Lang
{
  internal interface IEndHandler<G_OPPOSITE>
  {
    bool Remove(G_OPPOSITE oppositeObject);
    bool Contains(G_OPPOSITE oppositeObject);
    IEnumerable<G_OPPOSITE> Get();

    int Count { get; }
  }

  internal interface IEndHandler<G_OPPOSITE, G_LINK>
  {
    bool Remove(G_OPPOSITE oppositeObject);
    bool Contains(G_OPPOSITE oppositeObject);
    IEnumerable<G_OPPOSITE> Get();

    IEnumerable<G_LINK> GetAssociationLink();
    G_LINK GetAssociationLink(G_OPPOSITE oppositeObject);
    int Count { get; }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Internal/IInvariant.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AssociationSharp.Lang
```

```
{
  public interface IInvariant
  {
    bool IsValid();
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Internal/RegisterAssociationEnd.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using AssociationSharp.Lang;
using AssociationSharp.Commons.Assertion;

namespace AssociationSharp.Lang
{
  internal interface IRegisterAssociationEnd
  {
    void Enable();
    void Disable();
  }

  internal sealed class RegisterAssociationEnd<OWNER, OPPOSITE>
: IRegisterAssociationEnd
    where OWNER : class
    where OPPOSITE : class
  {
    internal delegate void InstantiationMethod(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd);
    internal delegate bool AdditionMethod(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd,
```

```
      OPPOSITE oppositeObject);
  internal delegate bool RemovalMethod(
    AssociationEnd<OWNER, OPPOSITE> ownerEnd,
    ref OPPOSITE oppositeObject);
  internal delegate bool SimpleSwapMethod(
    AssociationEnd<OWNER, OPPOSITE> ownerEnd,
    ref OPPOSITE oldOppositeObject,
    OPPOSITE newOppositeObject);

  internal InstantiationMethod mutableInstantiation {
    private set;
    get;
  }
  internal AdditionMethod mutableAdd { private set; get; }
  internal RemovalMethod mutableRemove { private set; get; }
  internal SimpleSwapMethod mutableSimpleSwap
  {
    private set;
    get;
  }

  public void Enable()
  {
    this.mutableInstantiation =
      CheckMandatoryAssociationNow;
    this.mutableAdd = AddAndCheckNow;
    this.mutableRemove = RemoveAndCheckNow;
    this.mutableSimpleSwap = SimpleSwapAndCheckNow;
  }

  public void Disable()
  {
    this.mutableInstantiation =
      CheckMandatoryAssociationLater;
    this.mutableAdd = AddAndCheckLater;
    this.mutableRemove = RemoveAndCheckLater;
    this.mutableSimpleSwap = SimpleSwapAndCheckLater;
  }

  public RegisterAssociationEnd()
  {
    if (AssociationSharpSystem.enableSystemConstraints)
    {
      this.Enable();
    }
    else
    {
      this.Disable();
    }
  }

  internal static void CheckMandatoryAssociationNow(
    AssociationEnd<OWNER, OPPOSITE> ownerEnd)
  {
```

```
      if (ownerEnd.multiplicity.mandatory)
      {
        throw new AssociationSharpMultiplicityException();
      }
    }

    internal static void CheckMandatoryAssociationLater(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd)
    {
      AssociationSharpSystem.PostInvariantToCheck(ownerEnd);
    }

    internal static bool AddAndCheckNow(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd,
      OPPOSITE oppositeObject)
    {
      bool added;

      if (added = ownerEnd.__Add(oppositeObject))
      {
        AssociationEnd<OPPOSITE, OWNER> oppositeEnd;

        if (ownerEnd.IsValid() && (oppositeEnd =
          ownerEnd.oppositeAssociationEndFinder(
            oppositeObject)).__IsValid(1))
        {
          // finish!
          oppositeEnd.__Add(ownerEnd.ownerObject);
        }

        else
        {
          // rollback!
          ownerEnd.__Remove(oppositeObject);

          throw new AssociationSharpMultiplicityException();
        }
      }

      return added;
    }

    internal static bool AddAndCheckLater(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd,
      OPPOSITE oppositeObject)
    {
      bool added = ownerEnd.__Add(oppositeObject);

      if (added)
      {
        AssociationEnd<OPPOSITE, OWNER> oppositeEnd =
          ownerEnd.oppositeAssociationEndFinder(
            oppositeObject);
```

```
        oppositeEnd.__Add(ownerEnd.ownerObject);

        AssociationSharpSystem.PostInvariantToCheck(ownerEnd);
        AssociationSharpSystem.
          PostInvariantToCheck(oppositeEnd);
      }

      return added;
    }

    private static bool RemoveAndCheckNow(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd,
      ref OPPOSITE oppositeObject)
    {
      bool removed;

      if (removed = ownerEnd.__Remove(oppositeObject))
      {
        if (ownerEnd.IsValid())
        {
          AssociationEnd<OPPOSITE, OWNER> oppositeEnd =
            ownerEnd.oppositeAssociationEndFinder(
              oppositeObject);

          if (oppositeEnd.__IsValid(-1))
          {
            oppositeEnd.__Remove(ownerEnd.ownerObject);
          }

          else if (!destroyOppositeObject(
            ref oppositeObject, ref oppositeEnd))
          {
            // rollback!
            ownerEnd.__Add(oppositeObject);

            throw
              new AssociationSharpMultiplicityException();
          }
        }

        else
        {
          // rollback!
          ownerEnd.__Add(oppositeObject);

          throw new AssociationSharpMultiplicityException();
        }
      }

      return removed;
    }

    private static bool RemoveAndCheckLater(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd,
```

```
      ref OPPOSITE oppositeObject)
    {
      bool removed = ownerEnd.__Remove(oppositeObject);

      if (removed)
      {
        AssociationEnd<OPPOSITE, OWNER> oppositeEnd =
          ownerEnd.oppositeAssociationEndFinder(
            oppositeObject);

        oppositeEnd.__Remove(ownerEnd.ownerObject);

        AssociationSharpSystem.PostInvariantToCheck(ownerEnd);
        AssociationSharpSystem.
          PostInvariantToCheck(oppositeEnd);
      }

      return removed;
    }

    private static bool SimpleSwapAndCheckNow(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd,
      ref OPPOSITE oldOppositeObject,
      OPPOSITE newOppositeObject)
    {
      bool swaped;

      if (swaped = ownerEnd.__Remove(oldOppositeObject))
      {
        // oldOppositeObject removed

        if (swaped = ownerEnd.__Add(newOppositeObject))
        {
          // newOppositeObject removed

          AssociationEnd<OPPOSITE, OWNER> newOppositeEnd =
            ownerEnd.oppositeAssociationEndFinder(
              newOppositeObject);
          AssociationEnd<OPPOSITE, OWNER> oldOppositeEnd =
            ownerEnd.oppositeAssociationEndFinder(
              oldOppositeObject);
          OWNER ownerObject = ownerEnd.ownerObject;

          if (newOppositeEnd.__IsValid(+1)
                    &&
                    (oldOppositeEnd.__IsValid(-1)
                         ||
                         destroyOppositeObject(
                           ref oldOppositeObject,
                           ref oldOppositeEnd)
                    )
              )
          {
            // Finalize!
```

```
            if (oldOppositeEnd != null)
            {
              oldOppositeEnd.__Remove(ownerObject);
            }
            newOppositeEnd.__Add(ownerObject);
          }

          else
          {
            // rollback!
            ownerEnd.__Remove(newOppositeObject);
            ownerEnd.__Add(oldOppositeObject);

            throw
              new AssociationSharpMultiplicityException();
          }
        }

        else
        {
          // rollback
          ownerEnd.__Add(oldOppositeObject);
        }
      }

      return swaped;
    }

    private static bool SimpleSwapAndCheckLater(
      AssociationEnd<OWNER, OPPOSITE> ownerEnd,
      ref OPPOSITE oldOppositeObject,
      OPPOSITE newOppositeObject)
    {
      bool swaped;

      if (ownerEnd.__Remove(oldOppositeObject))
      {
        if (ownerEnd.__Add(newOppositeObject))
        {
          // Ok
          swaped = true;
          OWNER ownerObject = ownerEnd.ownerObject;
          AssociationEnd<OPPOSITE, OWNER> oldOppositeEnd =
            ownerEnd.oppositeAssociationEndFinder(
              oldOppositeObject);
          AssociationEnd<OPPOSITE, OWNER> newOppositeEnd =
            ownerEnd.oppositeAssociationEndFinder(
              newOppositeObject);

          oldOppositeEnd.__Remove(ownerObject);
          newOppositeEnd.__Add(ownerObject);

          AssociationSharpSystem.
            PostInvariantToCheck(oldOppositeEnd);
```

```
            AssociationSharpSystem.
              PostInvariantToCheck(newOppositeEnd);
          }
          else
          {
            swaped = false;

            // rollback
            ownerEnd.__Add(oldOppositeObject);
          }
        }

        else
        {
          swaped = false;
        }

        return swaped;
      }

      private static bool destroyOppositeObject<OPPOSITE_END>(
        ref OPPOSITE oppositeObject,
        ref OPPOSITE_END oppositeEnd)
          where OPPOSITE_END : AssociationEnd<OPPOSITE, OWNER>
      {
        // convert strong references to weak references.
        WeakReference<OPPOSITE> oppositeObjectReference =
          WeakReference<OPPOSITE>.convertFromStrongReference(
            ref oppositeObject);
        WeakReference<OPPOSITE_END> oppositeEndReference =
          WeakReference<OPPOSITE_END>.
            convertFromStrongReference(ref oppositeEnd);

        // try to destroy
        System.GC.Collect();

        bool destroyed = !oppositeObjectReference.IsAlive ||
          AssociationSharpSystem.isRemovedForCheck(
            oppositeEndReference.Target);

        if (!destroyed)
        {
          oppositeObject =
            WeakReference<OPPOSITE>.convertToStrongReference(
              ref oppositeObjectReference);
          oppositeEnd = WeakReference<OPPOSITE_END>.
            convertToStrongReference(ref oppositeEndReference);
        }

        return destroyed;
      }

      internal static bool DoubleSwap(
        AssociationEnd<OWNER, OPPOSITE> thisOwnerEnd,
```

```
      OPPOSITE thisOppositeObject,
      OWNER otherOwnerObject,
      OPPOSITE otherOppositeObject)
    {
      bool swaped;

      if (swaped =
        (thisOwnerEnd.ownerObject != otherOwnerObject
          && thisOppositeObject != otherOppositeObject))
      {
        if (swaped = thisOwnerEnd.__Remove(
          thisOppositeObject))
        {
          AssociationEnd<OPPOSITE, OWNER> thisOppositeEnd =
            thisOwnerEnd.oppositeAssociationEndFinder(
              thisOppositeObject);
          AssociationEnd<OWNER, OPPOSITE> otherOwnerEnd =
            thisOppositeEnd.oppositeAssociationEndFinder(
              otherOwnerObject);

          if (swaped = otherOwnerEnd.__Remove(
            otherOppositeObject))
          {
            AssociationEnd<OPPOSITE, OWNER> otherOppositeEnd=
              otherOwnerEnd.oppositeAssociationEndFinder(
                otherOppositeObject);

            thisOppositeEnd.__Remove(
              thisOwnerEnd.ownerObject);
            otherOppositeEnd.__Remove(otherOwnerObject);

            thisOwnerEnd.__Add(otherOppositeObject);
            otherOppositeEnd.__Add(thisOwnerEnd.ownerObject);

            otherOwnerEnd.__Add(thisOppositeObject);
            thisOppositeEnd.__Add(otherOwnerObject);
          }
          else
          {
            // rollback
            thisOwnerEnd.__Add(thisOppositeObject);
          }
        }
      }

      return swaped;
    }
  }

  internal sealed
    class RegisterAssociationEnd<OWNER, OPPOSITE, LINK>
      : IRegisterAssociationEnd
    where OWNER : class
    where OPPOSITE : class
```

```
    where LINK : class, new()
  {
    internal delegate void InstantiationMethod(
      AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd);
    internal delegate LINK AdditionMethod(
      AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
      OPPOSITE oppositeObject);
    internal delegate bool RemovalMethod(
      AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
      ref OPPOSITE oppositeObject);
    internal delegate bool SimpleSwapMethod(
      AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
      ref OPPOSITE oldOppositeObject,
      OPPOSITE newOppositeObject);

    internal InstantiationMethod mutableInstantiation {
      private set; get; }
    internal AdditionMethod mutableAdd { private set; get; }
    internal RemovalMethod mutableRemove { private set; get; }
    internal SimpleSwapMethod mutableSimpleSwap {
      private set; get; }

    public void Enable()
    {
      this.mutableInstantiation =
        CheckMandatoryAssociationNow;
      this.mutableAdd = AddAndCheckNow;
      this.mutableRemove = RemoveAndCheckNow;
      this.mutableSimpleSwap = SimpleSwapAndCheckNow;
    }

    public void Disable()
    {
      this.mutableInstantiation =
        CheckMandatoryAssociationLater;
      this.mutableAdd = AddAndCheckLater;
      this.mutableRemove = RemoveAndCheckLater;
      this.mutableSimpleSwap = SimpleSwapAndCheckLater;
    }

    public RegisterAssociationEnd()
    {
      if (AssociationSharpSystem.enableSystemConstraints)
      {
        this.Enable();
      }
      else
      {
        this.Disable();
      }
    }

    internal static void CheckMandatoryAssociationNow(
      AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd)
```

```
    {
      if (ownerEnd.multiplicity.mandatory)
      {
        throw new AssociationSharpMultiplicityException();
      }
    }

    internal static void CheckMandatoryAssociationLater(
      AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd)
    {
      AssociationSharpSystem.PostInvariantToCheck(ownerEnd);
    }

    internal static LINK AddAndCheckNow(
      AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
      OPPOSITE oppositeObject)
    {
      LINK newLink = ownerEnd.createLink(oppositeObject);

      if (ownerEnd.__Add(oppositeObject, newLink))
      {
        AssociationEnd<OPPOSITE, OWNER, LINK> oppositeEnd;

        if (ownerEnd.IsValid() && (oppositeEnd =
          ownerEnd.oppositeAssociationEndFinder(
            oppositeObject)).__IsValid(1))
        {
          // finish!
          oppositeEnd.__Add(ownerEnd.ownerObject, newLink);
        }

        else
        {
          // rollback!
          ownerEnd.__Remove(oppositeObject);

          throw new AssociationSharpMultiplicityException();
        }
      }
      else
      {
        newLink = null;
      }

      return newLink;
    }

    internal static LINK AddAndCheckLater(
      AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
      OPPOSITE oppositeObject)
    {
      LINK newLink = ownerEnd.createLink(oppositeObject);

      if (ownerEnd.__Add(oppositeObject, newLink))
```

```
    {
      AssociationEnd<OPPOSITE, OWNER, LINK> oppositeEnd =
        ownerEnd.oppositeAssociationEndFinder(
          oppositeObject);

      oppositeEnd.__Add(ownerEnd.ownerObject, newLink);

      AssociationSharpSystem.PostInvariantToCheck(ownerEnd);
      AssociationSharpSystem.
        PostInvariantToCheck(oppositeEnd);
    }
    else
    {
      newLink = null;
    }

    return newLink;
  }

  private static bool RemoveAndCheckNow(
    AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
    ref OPPOSITE oppositeObject)
  {
    LINK link = ownerEnd.__Get(oppositeObject);

    bool removed;

    if (removed = (link != null))
    {
      if (ownerEnd.__IsValid(-1))
      {
        AssociationEnd<OPPOSITE, OWNER, LINK> oppositeEnd =
          ownerEnd.oppositeAssociationEndFinder(
            oppositeObject);

        ownerEnd.__Remove(oppositeObject);

        if (oppositeEnd.__IsValid(-1))
        {
          oppositeEnd.__Remove(ownerEnd.ownerObject);
        }

        else
        {
          if (!destroyOppositeObject(ref oppositeObject,
            ref oppositeEnd, link))
          {
            // rollback!
            ownerEnd.__Add(oppositeObject, link);
            throw
              new AssociationSharpMultiplicityException();
          }
        }
      }
```

```
      else
      {
        throw new AssociationSharpMultiplicityException();
      }
    }

    return removed;
  }

  private static bool RemoveAndCheckLater(
    AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
    ref OPPOSITE oppositeObject)
  {
    LINK link = ownerEnd.__Get(oppositeObject);

    bool removed;

    if (removed = (link != null))
    {
      AssociationEnd<OPPOSITE, OWNER, LINK> oppositeEnd =
        ownerEnd.oppositeAssociationEndFinder(
          oppositeObject);

      ownerEnd.__Remove(oppositeObject);
      oppositeEnd.__Remove(ownerEnd.ownerObject);

      AssociationSharpSystem.PostInvariantToCheck(ownerEnd);
      AssociationSharpSystem.PostInvariantToCheck(
        oppositeEnd);
    }

    return removed;
  }

  private static bool SimpleSwapAndCheckNow(
    AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
    ref OPPOSITE oldOppositeObject,
    OPPOSITE newOppositeObject)
  {
    LINK oldLink = ownerEnd.__Get(oldOppositeObject);

    bool swaped;

    if (swaped = (oldLink != null))
    {
      LINK newLink = ownerEnd.createLink(newOppositeObject);

      if (swaped = ownerEnd.__Add(newOppositeObject,
        newLink))
      {
        // newLink removed
```

```
            AssociationEnd<OPPOSITE,OWNER,LINK> newOppositeEnd=
              ownerEnd.oppositeAssociationEndFinder(
                newOppositeObject);
            AssociationEnd<OPPOSITE,OWNER,LINK> oldOppositeEnd=
              ownerEnd.oppositeAssociationEndFinder(
                oldOppositeObject);

            // oldLink removed
            ownerEnd.__Remove(oldOppositeObject);

            if (newOppositeEnd.__IsValid(+1)
                     &&
                     (oldOppositeEnd.__IsValid(-1)
                          ||
                          destroyOppositeObject(
                            ref oldOppositeObject,
                            ref oldOppositeEnd,
                            oldLink)
                     )
                )
            {
              // Finalize!
              if (oldOppositeEnd != null)
              {
                oldOppositeEnd.__Remove(ownerEnd.ownerObject);
              }

              newOppositeEnd.__Add(ownerEnd.ownerObject,
                newLink);
            }

            else
            {
              // rollback!
              ownerEnd.__Remove(newOppositeObject);
              ownerEnd.__Add(oldOppositeObject, oldLink);

              throw
                new AssociationSharpMultiplicityException();
            }
          }
        }

        return swaped;
      }

      private static bool SimpleSwapAndCheckLater(
        AssociationEnd<OWNER, OPPOSITE, LINK> ownerEnd,
        ref OPPOSITE oldOppositeObject,
        OPPOSITE newOppositeObject)
      {
        LINK oldLink = ownerEnd.__Get(oldOppositeObject);

        bool swaped;
```

```
    if (swaped = (oldLink != null))
    {
      LINK newLink = ownerEnd.createLink(newOppositeObject);

      if (swaped = ownerEnd.__Add(newOppositeObject,
        newLink))
      {
        // Ok
        AssociationEnd<OPPOSITE,OWNER,LINK> oldOppositeEnd=
          ownerEnd.oppositeAssociationEndFinder(
            oldOppositeObject);
        AssociationEnd<OPPOSITE,OWNER,LINK> newOppositeEnd=
          ownerEnd.oppositeAssociationEndFinder(
            newOppositeObject);

        ownerEnd.__Remove(oldOppositeObject);
        oldOppositeEnd.__Remove(ownerEnd.ownerObject);
        newOppositeEnd.__Add(ownerEnd.ownerObject,newLink);

        AssociationSharpSystem.PostInvariantToCheck(
          oldOppositeEnd);
        AssociationSharpSystem.PostInvariantToCheck(
          newOppositeEnd);
      }
    }

    return swaped;
  }

  private static bool destroyOppositeObject<OPPOSITE_END>(
      ref OPPOSITE oppositeObject,
      ref OPPOSITE_END oppositeEnd,
      LINK link)
      where OPPOSITE_END
        : AssociationEnd<OPPOSITE, OWNER, LINK>
  {
    // convert strong references to weak references.
    WeakReference<OPPOSITE> oppositeObjectReference =
      WeakReference<OPPOSITE>.convertFromStrongReference(
        ref oppositeObject);
    WeakReference<OPPOSITE_END> oppositeEndReference =
      WeakReference<OPPOSITE_END>.
        convertFromStrongReference(ref oppositeEnd);

    // try to destroy
    System.GC.Collect();

    bool destroyed = !oppositeObjectReference.IsAlive ||
      AssociationSharpSystem.isRemovedForCheck(
        oppositeEndReference.Target);

    if (!destroyed)
    {
```

```
        oppositeObject =
          WeakReference<OPPOSITE>.convertToStrongReference(
            ref oppositeObjectReference);
        oppositeEnd = WeakReference<OPPOSITE_END>.
          convertToStrongReference(ref oppositeEndReference);
      }

      return destroyed;
    }

    internal static bool DoubleSwap(
      AssociationEnd<OWNER, OPPOSITE, LINK> thisOwnerEnd,
      OPPOSITE thisOppositeObject,
      OWNER otherOwnerObject,
      OPPOSITE otherOppositeObject)
    {
      bool swaped;

      if (swaped =
        (thisOwnerEnd.ownerObject != otherOwnerObject
          && thisOppositeObject != otherOppositeObject))
      {
        LINK oldThisLink = thisOwnerEnd.__Get(
          thisOppositeObject);

        if (swaped = (oldThisLink != null))
        {
          AssociationEnd<OPPOSITE,OWNER,LINK>
            thisOppositeEnd =
              thisOwnerEnd.oppositeAssociationEndFinder(
                thisOppositeObject);
          AssociationEnd<OWNER,OPPOSITE,LINK> otherOwnerEnd =
            thisOppositeEnd.oppositeAssociationEndFinder(
              otherOwnerObject);
          LINK oldOtherLink =
            otherOwnerEnd.__Get(otherOppositeObject);

          if (swaped = (oldOtherLink != null))
          {
            thisOwnerEnd.__Remove(thisOppositeObject);
            otherOwnerEnd.__Remove(otherOppositeObject);

            AssociationEnd<OPPOSITE, OWNER,L INK>
              otherOppositeEnd =
                otherOwnerEnd.oppositeAssociationEndFinder(
                  otherOppositeObject);

            thisOppositeEnd.__Remove(
              thisOwnerEnd.ownerObject);
            otherOppositeEnd.__Remove(
              otherOwnerEnd.ownerObject);

            LINK newThisLink =
              thisOwnerEnd.createLink(otherOppositeObject);
```

```
              LINK newOtherLink =
                 otherOwnerEnd.createLink(thisOppositeObject);

              thisOwnerEnd.__Add(otherOppositeObject,
                 newThisLink);
              otherOppositeEnd.__Add(thisOwnerEnd.ownerObject,
                 newThisLink);
              otherOwnerEnd.__Add(thisOppositeObject,
                 newOtherLink);
              thisOppositeEnd.__Add(otherOwnerEnd.ownerObject,
                 newOtherLink);
           }
        }
     }

     return swaped;
   }
 }
}

/*****************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Internal/SpecializedAssociationEnd.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *****************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AssociationSharp.Lang
{
  public abstract class SpecializedAssociationEnd
    <OWNER, OPPOSITE, G_OWNER, G_OPPOSITE>
      : GeneralizedAssociationEnd<OWNER, OPPOSITE>
    where OWNER : class, G_OWNER
```

```
    where OPPOSITE : class, G_OPPOSITE
    where G_OWNER : class
    where G_OPPOSITE : class
  {
    private readonly
      GeneralizedAssociationEnd<G_OWNER, G_OPPOSITE>
        generalizedEnd;

    public SpecializedAssociationEnd(
        AssociationEndFinder objectAssociationEndFinder,
        AssociationEnd<OPPOSITE, OWNER>.AssociationEndFinder
          oppositeAssociationEndFinder,
        GeneralizedAssociationEnd<G_OWNER, G_OPPOSITE>
          generalizedEnd,
        Multiplicity multiplicity)
      : base(
          objectAssociationEndFinder,
          oppositeAssociationEndFinder,
          generalizedEnd.ownerObject as OWNER,
          multiplicity,
          generalizedEnd.synchronizer)
    {
      this.generalizedEnd = generalizedEnd;
      this.generalizedEnd.registerSpecializedEnd(
        new Handler(this));
    }

    internal override bool __Add(OPPOSITE oppositeObject)
    {
      return !this.generalizedEnd.Contains(oppositeObject) &&
        base.__Add(oppositeObject);
    }

    internal override bool __IsValid(int incrementCount)
    {
      return base.__IsValid(incrementCount) &&
        this.generalizedEnd.__IsValid(incrementCount);
    }

    internal struct Handler : IEndHandler<G_OPPOSITE>
    {
      internal readonly SpecializedAssociationEnd
        <OWNER, OPPOSITE, G_OWNER, G_OPPOSITE> handled;

      public Handler(SpecializedAssociationEnd<OWNER,
        OPPOSITE, G_OWNER, G_OPPOSITE> handled)
        : this()
      {
        this.handled = handled;
      }

      public bool Remove(G_OPPOSITE oppositeObject)
      {
        return oppositeObject is OPPOSITE
```

```
            ? this.handled.Remove(oppositeObject as OPPOSITE)
            : false;
      }

      public bool Contains(G_OPPOSITE oppositeObject)
      {
        return oppositeObject is OPPOSITE &&
          this.handled.Contains(oppositeObject as OPPOSITE);
      }

      public IEnumerable<G_OPPOSITE> Get()
      {
        return this.handled.Get();
      }

      public int Count
      {
        get { return this.handled.Count; }
      }
    }
  }

  public abstract class SpecializedAssociationEnd
    <OWNER, OPPOSITE, LINK, G_OWNER, G_OPPOSITE, G_LINK>
      : GeneralizedAssociationEnd<OWNER, OPPOSITE, LINK>
    where OWNER : class, G_OWNER
    where OPPOSITE : class, G_OPPOSITE
    where LINK : class, G_LINK, new()
    where G_LINK : class, new()
    where G_OWNER : class
    where G_OPPOSITE : class
  {
    private readonly
      GeneralizedAssociationEnd<G_OWNER, G_OPPOSITE, G_LINK>
        generalizedEnd;

    public SpecializedAssociationEnd(
        AssociationEndFinder objectAssociationEndFinder,
        AssociationEnd<OPPOSITE, OWNER, LINK>.
          AssociationEndFinder oppositeAssociationEndFinder,
        GeneralizedAssociationEnd<G_OWNER, G_OPPOSITE, G_LINK>
          generalizedEnd,
        Multiplicity multiplicity)
      : base(
          objectAssociationEndFinder,
          oppositeAssociationEndFinder,
          generalizedEnd.ownerObject as OWNER,
          multiplicity,
          generalizedEnd.synchronizer)
    {
      this.generalizedEnd = generalizedEnd;
      this.generalizedEnd.registerSpecializedEnd(
        new Handler(this));
    }
```

```
internal override bool __Add(OPPOSITE oppositeObject,
  LINK link)
{
  return !this.generalizedEnd.Contains(oppositeObject) &&
    base.__Add(oppositeObject, link);
}

internal override bool __IsValid(int incrementCount)
{
  return base.__IsValid(incrementCount) &&
    this.generalizedEnd.__IsValid(incrementCount);
}

internal struct Handler : IEndHandler<G_OPPOSITE, G_LINK>
{
  internal readonly SpecializedAssociationEnd
    <OWNER, OPPOSITE, LINK, G_OWNER, G_OPPOSITE, G_LINK>
      handled;

  public Handler(SpecializedAssociationEnd
    <OWNER, OPPOSITE, LINK, G_OWNER, G_OPPOSITE, G_LINK>
      handled)
    : this()
  {
    this.handled = handled;
  }

  public bool Remove(G_OPPOSITE oppositeObject)
  {
    return oppositeObject is OPPOSITE
         ? this.handled.Remove(oppositeObject as OPPOSITE)
         : false;
  }

  public bool Contains(G_OPPOSITE oppositeObject)
  {
    return oppositeObject is OPPOSITE &&
      this.handled.Contains(oppositeObject as OPPOSITE);
  }

  public IEnumerable<G_OPPOSITE> Get()
  {
    return this.handled.Get();
  }

  public IEnumerable<G_LINK> GetAssociationLink()
  {
    return this.handled.GetAssociationLink();
  }

  public G_LINK GetAssociationLink(
    G_OPPOSITE oppositeObject)
  {
```

```
          return (oppositeObject is OPPOSITE)
            ? this.handled.__Get(oppositeObject as OPPOSITE)
            : null;
        }

        public int Count
        {
          get { return this.handled.Count; }
        }
      }
    }
}

/*****************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Association.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *****************************************************************/
using System;
using AssociationSharp.Lang;

namespace AssociationSharp.Lang
{
  public abstract class Association<SOURCE, TARGET>
    where SOURCE : class
    where TARGET : class
  {
    // instance
    public Multiplicity sourceMultiplicity {
      private set; get; }
    public Multiplicity targetMultiplicity {
      private set; get; }
    private readonly
      AssociationEnd<SOURCE, TARGET>.AssociationEndFinder
        sourceEndFinder;
    private readonly
```

```
      AssociationEnd<TARGET, SOURCE>.AssociationEndFinder
        targetEndFinder;

    public Association(
        Multiplicity sourceMultiplicity,
        Multiplicity targetMultiplicity,
        AssociationEnd<SOURCE, TARGET>.AssociationEndFinder
          sourceEndFinder,
        AssociationEnd<TARGET, SOURCE>.AssociationEndFinder
          targetEndFinder
        )
    {
      this.sourceMultiplicity = sourceMultiplicity;
      this.targetMultiplicity = targetMultiplicity;
      this.sourceEndFinder = sourceEndFinder;
      this.targetEndFinder = targetEndFinder;

      AssociationSharpSystem.RegisterSynchronizer(this);
    }

    // types

    public class SourceAssociationEnd
      : AssociationEnd<SOURCE, TARGET>
    {
      protected SourceAssociationEnd(
          SOURCE sourceObject,
          Association<SOURCE, TARGET> association)
        : base(
            association.sourceEndFinder,
            association.targetEndFinder,
            sourceObject,
            association.sourceMultiplicity,
            association
        ) { }
    }

    public class TargetAssociationEnd
      : AssociationEnd<TARGET, SOURCE>
    {
      protected TargetAssociationEnd(
          TARGET targetObject,
          Association<SOURCE, TARGET> association)
        : base(
            association.targetEndFinder,
            association.sourceEndFinder,
            targetObject,
            association.targetMultiplicity,
            association
        ) { }
    }
  }

  public abstract class GeneralizedAssociation<SOURCE, TARGET>
```

```
    where SOURCE : class
    where TARGET : class
  {
    // instance
    public readonly Multiplicity sourceMultiplicity;
    public readonly Multiplicity targetMultiplicity;
    private readonly
      AssociationEnd<SOURCE, TARGET>.AssociationEndFinder
        sourceEndFinder;
    private readonly
      AssociationEnd<TARGET, SOURCE>.AssociationEndFinder
        targetEndFinder;

    public GeneralizedAssociation(
        Multiplicity sourceMultiplicity,
        Multiplicity targetMultiplicity,
        AssociationEnd<SOURCE, TARGET>.AssociationEndFinder
          sourceEndFinder,
        AssociationEnd<TARGET, SOURCE>.AssociationEndFinder
          targetEndFinder
        )
    {
      this.sourceMultiplicity = sourceMultiplicity;
      this.targetMultiplicity = targetMultiplicity;
      this.sourceEndFinder = sourceEndFinder;
      this.targetEndFinder = targetEndFinder;
    }

    public class SourceGeneralizedAssociationEnd
      : GeneralizedAssociationEnd<SOURCE, TARGET>
    {
      protected SourceGeneralizedAssociationEnd(
          SOURCE sourceObject,
          GeneralizedAssociation<SOURCE, TARGET> association)
        : base(
            association.sourceEndFinder,
            association.targetEndFinder,
            sourceObject,
            association.sourceMultiplicity,
            association
        ) { }
    }

    public class TargetGeneralizedAssociationEnd
      : GeneralizedAssociationEnd<TARGET, SOURCE>
    {
      protected TargetGeneralizedAssociationEnd(
          TARGET targetObject,
          GeneralizedAssociation<SOURCE, TARGET> association)
        : base(
            association.targetEndFinder,
            association.sourceEndFinder,
            targetObject,
            association.targetMultiplicity,
```

```
            association
        ) { }
    }
}

public abstract class
    SpecializedAssociation<SOURCE, TARGET, G_SOURCE, G_TARGET>
    where SOURCE : class, G_SOURCE
    where TARGET : class, G_TARGET
    where G_SOURCE : class
    where G_TARGET : class
{
    // instance
    public readonly Multiplicity sourceMultiplicity;
    public readonly Multiplicity targetMultiplicity;
    private readonly AssociationEnd<SOURCE, TARGET>.
      AssociationEndFinder sourceEndFinder;
    private readonly AssociationEnd<TARGET, SOURCE>.
      AssociationEndFinder targetEndFinder;

    public SpecializedAssociation(
        Multiplicity sourceMultiplicity,
        Multiplicity targetMultiplicity,
        AssociationEnd<SOURCE, TARGET>.AssociationEndFinder
          sourceEndFinder,
        AssociationEnd<TARGET, SOURCE>.AssociationEndFinder
          targetEndFinder
        )
    {
      this.sourceMultiplicity = sourceMultiplicity;
      this.targetMultiplicity = targetMultiplicity;
      this.sourceEndFinder = sourceEndFinder;
      this.targetEndFinder = targetEndFinder;
    }

    public class SpecializedSourceAssociationEnd
      : SpecializedAssociationEnd
        <SOURCE, TARGET, G_SOURCE, G_TARGET>
    {
      protected SpecializedSourceAssociationEnd(
          GeneralizedAssociationEnd<G_SOURCE, G_TARGET>
            generalizedEnd,
          SpecializedAssociation
            <SOURCE, TARGET, G_SOURCE, G_TARGET> association)
        : base(
            association.sourceEndFinder,
            association.targetEndFinder,
            generalizedEnd,
            association.sourceMultiplicity
        ) { }
    }

    public class SpecializedTargetAssociationEnd
      : SpecializedAssociationEnd
```

```
          <TARGET, SOURCE, G_TARGET, G_SOURCE>
    {
      protected SpecializedTargetAssociationEnd(
        GeneralizedAssociationEnd<G_TARGET, G_SOURCE>
          generalizedEnd,
        SpecializedAssociation
          <SOURCE, TARGET, G_SOURCE, G_TARGET> association)
      : base(
        association.targetEndFinder,
        association.sourceEndFinder,
        generalizedEnd,
        association.targetMultiplicity
      ) { }
    }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/AssociationClass.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using AssociationSharp.Lang;

namespace AssociationSharp.Lang
{
  public abstract class AssociationClass<SOURCE, TARGET, LINK>
    where SOURCE : class
    where TARGET : class
    where LINK : class, new()
  {
    // instance
```

```
    public readonly Multiplicity sourceMultiplicity;
    public readonly Multiplicity targetMultiplicity;
    private readonly
      AssociationEnd<SOURCE,TARGET,LINK>.AssociationEndFinder
        sourceEndFinder;
    private readonly
      AssociationEnd<TARGET,SOURCE,LINK>.AssociationEndFinder
        targetEndFinder;

    public AssociationClass(
      Multiplicity sourceMultiplicity,
      Multiplicity targetMultiplicity,
      AssociationEnd<SOURCE,TARGET,LINK>.AssociationEndFinder
        sourceEndFinder,
      AssociationEnd<TARGET,SOURCE,LINK>.AssociationEndFinder
        targetEndFinder
      )
    {
      this.sourceMultiplicity = sourceMultiplicity;
      this.targetMultiplicity = targetMultiplicity;
      this.sourceEndFinder = sourceEndFinder;
      this.targetEndFinder = targetEndFinder;
    }

    public class SourceAssociationEnd
      : AssociationEnd<SOURCE, TARGET, LINK>
    {
      protected SourceAssociationEnd(
           SOURCE sourceObject,
           AssociationClass<SOURCE, TARGET, LINK> association)
        : base(
             association.sourceEndFinder,
             association.targetEndFinder,
             sourceObject,
             association.sourceMultiplicity,
             association
        ) { }
    }

    public class TargetAssociationEnd
      : AssociationEnd<TARGET, SOURCE, LINK>
    {
      protected TargetAssociationEnd(
           TARGET targetObject,
           AssociationClass<SOURCE, TARGET, LINK> association)
        : base(
             association.targetEndFinder,
             association.sourceEndFinder,
             targetObject,
             association.targetMultiplicity,
             association
        ) { }
    }
  }
```

```
public abstract
   class GeneralizedAssociationClass<SOURCE, TARGET, LINK>
   where SOURCE : class
   where TARGET : class
   where LINK : class, new()
{
   // instance
   public readonly Multiplicity sourceMultiplicity;
   public readonly Multiplicity targetMultiplicity;

   public GeneralizedAssociationClass(
        Multiplicity sourceMultiplicity,
        Multiplicity targetMultiplicity
        )
   {
     this.sourceMultiplicity = sourceMultiplicity;
     this.targetMultiplicity = targetMultiplicity;
   }

   protected abstract SourceGeneralizedAssociationEnd
     getSourceEnd(SOURCE sourceObject);
   protected abstract TargetGeneralizedAssociationEnd
     getTargetEnd(TARGET targetObject);

   public class SourceGeneralizedAssociationEnd
     : GeneralizedAssociationEnd<SOURCE, TARGET, LINK>
   {
     protected SourceGeneralizedAssociationEnd(
        SOURCE sourceObject,
        GeneralizedAssociationClass<SOURCE, TARGET, LINK>
          association)
        : base(
          association.getSourceEnd,
          association.getTargetEnd,
          sourceObject,
          association.sourceMultiplicity,
          association
        ) { }
   }

   public class TargetGeneralizedAssociationEnd
     : GeneralizedAssociationEnd<TARGET, SOURCE, LINK>
   {
     protected TargetGeneralizedAssociationEnd(
        TARGET targetObject,
        GeneralizedAssociationClass<SOURCE, TARGET, LINK>
          association)
        : base(
          association.getTargetEnd,
          association.getSourceEnd,
          targetObject,
          association.targetMultiplicity,
          association) { }
```

```
    }
  }

  public abstract class SpecializedAssociationClass
    <SOURCE, TARGET, LINK, G_SOURCE, G_TARGET, G_LINK>
    where SOURCE : class, G_SOURCE
    where TARGET : class, G_TARGET
    where LINK : class, G_LINK, new()
    where G_SOURCE : class
    where G_TARGET : class
    where G_LINK : class, new()
  {
    // instance
    public readonly Multiplicity sourceMultiplicity;
    public readonly Multiplicity targetMultiplicity;

    public SpecializedAssociationClass(
         Multiplicity sourceMultiplicity,
         Multiplicity targetMultiplicity)
    {
      this.sourceMultiplicity = sourceMultiplicity;
      this.targetMultiplicity = targetMultiplicity;
    }

    protected abstract SpecializedSourceAssociationEnd
      getSourceEnd(SOURCE sourceObject);
    protected abstract SpecializedTargetAssociationEnd
      getTargetEnd(TARGET targetObject);

    public class SpecializedSourceAssociationEnd
      : SpecializedAssociationEnd
        <SOURCE, TARGET, LINK, G_SOURCE, G_TARGET, G_LINK>
    {
      protected SpecializedSourceAssociationEnd(
        GeneralizedAssociationEnd
          <G_SOURCE, G_TARGET, G_LINK> generalizedEnd,
        SpecializedAssociationClass
          <SOURCE, TARGET, LINK, G_SOURCE, G_TARGET, G_LINK>
            association)
      : base(
        association.getSourceEnd,
        association.getTargetEnd,
        generalizedEnd,
        association.sourceMultiplicity) { }
    }

    public class SpecializedTargetAssociationEnd
      : SpecializedAssociationEnd
        <TARGET, SOURCE, LINK, G_TARGET, G_SOURCE, G_LINK>
    {
      protected SpecializedTargetAssociationEnd(
        GeneralizedAssociationEnd<G_TARGET, G_SOURCE, G_LINK>
          generalizedEnd,
        SpecializedAssociationClass
```

```
            <SOURCE, TARGET, LINK, G_SOURCE, G_TARGET, G_LINK>
              association)
       : base(
         association.getTargetEnd,
         association.getSourceEnd,
         generalizedEnd,
         association.targetMultiplicity) { }
     }
   }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/AssociationSharpSystem.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AssociationSharp.Lang
{
  public static class AssociationSharpSystem
  {
    private static readonly HashSet<IRegisterAssociationEnd>
      registers = new HashSet<IRegisterAssociationEnd>();
    private static readonly HashSet<object> synchronizers =
      new HashSet<object>();

    private static readonly Stack<CheckLevel>
      disabledConstraintsLevels = new Stack<CheckLevel>();
    private static readonly HashSet<WeakReference<IInvariant>>
      postedToCheck =
        new HashSet<WeakReference<IInvariant>>(
```

```
        new InvariantReferenceEqualityComparer());
  private static readonly HashSet<WeakReference<IInvariant>>
    removedInvariants =
      new HashSet<WeakReference<IInvariant>>(
        new InvariantReferenceEqualityComparer());

  internal static void RegisterAssociationEndType(
    IRegisterAssociationEnd associationRegister)
  {
    lock (registers)
    {
      registers.Add(associationRegister);
    }
  }

  internal static void RegisterSynchronizer(object synch)
  {
    lock (registers)
    {
      for (int i = 0, f = disabledConstraintsLevels.Count;
        i < f; i++)
      {
        System.Threading.Monitor.Enter(synch);
      }
      synchronizers.Add(synch);
    }
  }

  public static bool enableSystemConstraints
  {
    get { return disabledConstraintsLevels.Count == 0; }
  }

  public static void __EnableSystemConstraints()
  {
    System.GC.Collect();
    try
    {
      CheckInvariants();
    }

    finally
    {
      try
      {
        disabledConstraintsLevels.Pop();

        if (enableSystemConstraints)
        {
          EnableConstraints();
        }
      }
      finally
      {
```

```
            MonitorExit();
          }
        }
      }

      public static void __DisableSystemConstraints()
      {
        try
        {
          MonitorEnter();

          if (enableSystemConstraints)
          {
            try
            {
              DisableConstraints();
            }
            catch (Exception e)
            {
              EnableConstraints();
              throw e;
            }
          }

          disabledConstraintsLevels.Push(CheckLevel.Create());
        }
        catch (Exception e)
        {
          MonitorExit();
          throw e;
        }
      }

      private static void MonitorEnter()
      {
        System.Threading.Monitor.Enter(registers);

        foreach (var synch in synchronizers)
        {
          System.Threading.Monitor.Enter(synch);
        }
      }

      private static void MonitorExit()
      {
        foreach (var synch in synchronizers)
        {
          System.Threading.Monitor.Exit(synch);
        }

        System.Threading.Monitor.Exit(registers);
      }

      private static void EnableConstraints()
```

```
    {
      foreach (IRegisterAssociationEnd register in registers)
      {
        register.Enable();
      }
    }

    private static void DisableConstraints()
    {
      foreach (IRegisterAssociationEnd register in registers)
      {
        register.Disable();
      }
    }

    internal static void PostInvariantToCheck(
      IInvariant invariant)
    {
      WeakReference<IInvariant> invariantRef =
        new WeakReference<IInvariant>(invariant);
      if (postedToCheck.Add(invariantRef))
      {
        disabledConstraintsLevels.Peek().postedToCheck.Add(
          invariantRef);
      }
    }

    internal static void removePostedInvariant(
      IInvariant invariant)
    {
      WeakReference<IInvariant> invariantRef =
        new WeakReference<IInvariant>(invariant);
      if (disabledConstraintsLevels.Peek().postedToCheck.
        Contains(invariantRef))
      {
        postedToCheck.Remove(invariantRef);
        removedInvariants.Add(invariantRef);
      }
    }

    internal static bool isRemovedForCheck(
      IInvariant invariant)
    {
      return removedInvariants.Contains(
        new WeakReference<IInvariant>(invariant));
    }

    private static void CheckInvariants()
    {
      CheckLevel currentLevel =
        disabledConstraintsLevels.Peek();

      try
      {
```

```
        foreach (WeakReference<IInvariant> invariantReference
          in currentLevel.postedToCheck)
        {
          if (!removedInvariants.Contains(
            invariantReference))
          {
            IInvariant invariant = invariantReference.Target;
            if (invariantReference.IsAlive &&
              !invariant.IsValid())
            {
              throw
                new AssociationSharpMultiplicityException();
            }
          }
        }
      }

      finally
      {
        postedToCheck.ExceptWith(currentLevel.postedToCheck);
        removedInvariants.ExceptWith(
          currentLevel.postedToCheck);
        currentLevel.postedToCheck.Clear();
      }
    }

    private struct CheckLevel
    {
      internal readonly HashSet<WeakReference<IInvariant>>
        postedToCheck;

      private CheckLevel(HashSet<WeakReference<IInvariant>>
        postedToCheck)
      {
        this.postedToCheck = postedToCheck;
      }

      internal static CheckLevel Create()
      {
        return new CheckLevel(
          new HashSet<WeakReference<IInvariant>>(
            new InvariantReferenceEqualityComparer()));
      }
    }

    private struct InvariantReferenceEqualityComparer
      : IEqualityComparer<WeakReference<IInvariant>>
    {
      private static readonly int DEFAULT_HASHCODE =
        AssociationSharpSystem.postedToCheck.GetHashCode();

      public bool Equals(WeakReference<IInvariant> invRef1,
        WeakReference<IInvariant> invRef2)
      {
```

```
          return invRef1.Target == invRef2.Target;
       }

       public int GetHashCode(WeakReference<IInvariant> invRef)
       {
          return (invRef.Target != null)
               ? invRef.Target.GetHashCode()
               : DEFAULT_HASHCODE;
       }
     }
   }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Exceptions.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AssociationSharp.Lang
{
  public class AssociationSharpSystemException : Exception
  {
    public AssociationSharpSystemException(string message)
      : base(message) { }
  }

  public class AssociationSharpMultiplicityException
    : AssociationSharpSystemException
  {
    public AssociationSharpMultiplicityException()
      : this(null) { }
```

```
    public AssociationSharpMultiplicityException(
       string message) : base(
          "Mutiplicity constraints violation" +
            (message != null ? ": " + message : ".")) { }
  }
}

/*****************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Lang/Multiplicity.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *****************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections.ObjectModel;

namespace AssociationSharp.Lang
{
  public class Multiplicity
  {
    public delegate bool ValidateMethod(int count);

    public static readonly Multiplicity
         ZERO_TO_MANY_RANGE = new Multiplicity(),
         ZERO_TO_ONE_RANGE = new Multiplicity(0, 1),
         ONE_TO_ONE_RANGE = new Multiplicity(1),
         ONE_TO_MANY_RANGE = new Multiplicity(1, MANY);

    public const uint MANY = uint.MaxValue;

    public readonly bool mandatory;
    public readonly uint minimumBound;
    public readonly uint maximumBound;
    public readonly ValidateMethod isValid;
```

```
    public Multiplicity(uint minimumBound, uint maximumBound)
    {
      this.minimumBound = minimumBound;
      this.maximumBound = maximumBound;
      this.mandatory = (minimumBound > 0);

      if (minimumBound == 0 && maximumBound == MANY)
      {
        this.isValid = unboundedValidate;
      }
      else if (minimumBound == 0)
      {
        this.isValid = upperBoundedValidate;
      }
      else if (maximumBound == MANY)
      {
        this.isValid = lowerBoundedValidate;
      }
      else
      {
        this.isValid = bothBoundedValidate;
      }
    }

    public Multiplicity(uint uniqueBound)
      : this(uniqueBound, uniqueBound) { }

    private Multiplicity() : this(0, MANY) { }

    private bool bothBoundedValidate(int count)
    {
      return count >= minimumBound && count <= maximumBound;
    }

    private bool unboundedValidate(int count)
    {
      return true;
    }

    private bool lowerBoundedValidate(int count)
    {
      return count >= minimumBound;
    }

    private bool upperBoundedValidate(int count)
    {
      return count <= maximumBound;
    }
  }
}

/*******************************************************
 * Association# Language
```

```
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Commons/Assertion/Assert.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 ************************************************************/
using System;
using System.Globalization;

namespace AssociationSharp.Commons.Assertion
{
  public static class Assert
  {
    [System.Diagnostics.Conditional("ASSERT")]
    public static void IsTrue(bool condition, string message)
    {
      if (!condition)
      {
        Fail(message);
      }
    }

    [System.Diagnostics.Conditional("ASSERT")]
    public static void IsTrue(bool condition)
    {
      IsTrue(condition, null);
    }

    [System.Diagnostics.Conditional("ASSERT")]
    public static void IsFalse(bool condition, string message)
    {
      IsTrue(!condition, message);
    }

    [System.Diagnostics.Conditional("ASSERT")]
    public static void IsFalse(bool condition)
    {
      IsTrue(!condition, null);
    }
```

```
    [System.Diagnostics.Conditional("ASSERT")]
    public static void Fail(string message)
    {
      throw new AssertFailedException(message);
    }

    [System.Diagnostics.Conditional("ASSERT")]
    public static void Fail()
    {
      Fail(null);
    }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Commons/Assertion/AssertFailedException.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AssociationSharp.Commons.Assertion
{
  public class AssertFailedException : Exception
  {
    public AssertFailedException() : base() { }

    public AssertFailedException(string message)
       : base(message) { }
  }
}
```

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Commons/UnitTest.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using AssociationSharp.Commons.Assertion;

namespace AssociationSharp.Commons
{
  public static class UnitTest
  {
    public static void test(object unitTest)
    {
      Type type = unitTest.GetType();

      System.Console.Write(
        "Starting test of \"{0}\". Load methods... ", type);
      MethodInfo[] methods =
        type.GetMethods(BindingFlags.Public |
        BindingFlags.Instance |
        BindingFlags.DeclaredOnly);
      int count = methods.GetLength(0);
      System.Console.WriteLine(
        "Done!\nThere is {0:D} methods to be executed",
        count);

      for (int i = 0; i < count; i++)
      {
        try
        {
          MethodInfo method = methods[i];
```

```
            System.Console.Write(
              "\n[{0:D}] Executing method \"{1}\"... ",
              i + 1,
              method.Name);
            methods[i].Invoke(unitTest, new object[0]);
            System.Console.WriteLine("Passed!");
          }

          catch (Exception exception)
          {
            if (exception is TargetInvocationException)
            {
              exception = exception.InnerException;
            }

            if (exception is AssertFailedException)
            {
              System.Console.Write("Failed! ");
              if (exception.Message != null)
              {
                System.Console.Write(
                  "Cause:\"{0}\" ", exception.Message);
              }
              System.Console.WriteLine("on {0}",
                exception.StackTrace);
            }

            else
            {

              System.Console.WriteLine(
                "Error!\n---\nAn exception was thrown:");
              System.Console.WriteLine(
                "{0}:\"{1}\" on\n{2}",
                exception.GetType(),
                exception.Message,
                exception.StackTrace);

              exception = exception.InnerException;

              while (exception != null)
              {
                System.Console.WriteLine("caused by:");
                System.Console.WriteLine("{0}:\"{1}\" on\n{2}",
                  exception.GetType(),
                  exception.Message,
                  exception.StackTrace);
                exception = exception.InnerException;
              }
            }
            System.Console.WriteLine(
              "End of exception stack.\n---");
          }
        }
```

```
      System.Console.WriteLine();
    }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Commons/UnmodifiedSet.cs
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
using System.Text;

namespace AssociationSharp.Commons
{
  public class UnmodifiedSet<T>
    : ISet<T>, ICollection<T>, IEnumerable<T>, IEnumerable
  {
    private readonly ISet<T> set;
    public UnmodifiedSet(ISet<T> set)
    {
      if (set == null)
      {
        throw new ArgumentNullException();
      }
      this.set = set;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
      return this.set.GetEnumerator();
    }
```

```
public IEnumerator<T> GetEnumerator()
{
  return this.set.GetEnumerator();
}

void ISet<T>.UnionWith(IEnumerable<T> enumerable)
{
  throw new NotSupportedException();
}

void ISet<T>.SymmetricExceptWith(IEnumerable<T> other)
{
  throw new NotSupportedException();
}

public bool SetEquals(IEnumerable<T> other)
{
  return this.set.SetEquals(other);
}

public bool Overlaps(IEnumerable<T> other)
{
  return this.set.Overlaps(other);
}

public bool IsSupersetOf(IEnumerable<T> other)
{
  return this.set.IsSupersetOf(other);
}

public bool IsSubsetOf(IEnumerable<T> other)
{
  return this.set.IsSubsetOf(other);
}

public bool IsProperSubsetOf(IEnumerable<T> other)
{
  return this.set.IsProperSubsetOf(other);
}

public bool IsProperSupersetOf(IEnumerable<T> other)
{
  return this.set.IsProperSupersetOf(other);
}

void ISet<T>.IntersectWith(IEnumerable<T> other)
{
  throw new NotSupportedException();
}

void ISet<T>.ExceptWith(IEnumerable<T> other)
{
  throw new NotSupportedException();
```

```
    }

    bool ISet<T>.Add(T element)
    {
      throw new NotSupportedException();
    }

    bool ICollection<T>.Remove(T item)
    {
      throw new NotSupportedException();
    }

    void ICollection<T>.Add(T item)
    {
      throw new NotSupportedException();
    }

    public void CopyTo(T[] array, int arrayIndex)
    {
      this.set.CopyTo(array, arrayIndex);
    }

    public bool Contains(T item)
    {
      return this.set.Contains(item);
    }

    public bool Contains(T value,
      IEqualityComparer<T> comparer)
    {
      return this.set.Contains(value, comparer);
    }

    void ICollection<T>.Clear()
    {
      throw new NotSupportedException();
    }

    public int Count { get { return this.set.Count; } }

    bool ICollection<T>.IsReadOnly { get { return true; } }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: AssociationSharpLib - Library Suport for the
 *          implementation of  associations in C#.
 *
 * Source File: /Commons/WeakReference.cs
 *
```

```
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 ************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace System
{
  public class WeakReference<T> : WeakReference
       where T : class
  {
    public new T Target
    {
      set { base.Target = value; }
      get { return base.Target as T; }
    }

    public WeakReference(T target) : base(target) { }

    public WeakReference(T target, bool trackResurrection)
      : base(target, trackResurrection) { }

    public WeakReference(SerializationInfo info,
      StreamingContext context) : base(info, context) { }

    public static WeakReference<T>
      convertFromStrongReference(ref T strongReference)
    {
      try
      {
        return new WeakReference<T>(strongReference);
      }

      finally
      {
        strongReference = null;
      }
    }

    public static T convertToStrongReference(
      ref WeakReference<T> weakReference)
```

```
    {
      try
      {
        return weakReference.Target;
      }
      finally
      {
        weakReference = null;
      }
    }
  }
}
```

A listagem a seguir descreve o arquivo *makefile* a ser utilizado pelo compilador mcs.exe, através do comando de console: `mcs.exe @makefile`

```
-debug-
-optimize+
-out:../bin/AssociationSharpLib.dll
-platform:anycpu
-sdk:4
-target:library

./Commons/Assertion/Assert.cs
./Commons/Assertion/AssertFailedException.cs
./Commons/UnitTest.cs
./Commons/UnmodifiedSet.cs
./Commons/WeakReference.cs
./Lang/Internal/AssociationEnd.cs
./Lang/Internal/GeneralizedAssociationEnd.cs
./Lang/Internal/IEndHandler.cs
./Lang/Internal/IInvariant.cs
./Lang/Internal/RegisterAssociationEnd.cs
./Lang/Internal/SpecializedAssociationEnd.cs
./Lang/Association.cs
./Lang/AssociationClass.cs
./Lang/AssociationSharpSystem.cs
./Lang/Exceptions.cs
./Lang/Multiplicity.cs
```

APÊNDICE C – CÓDIGO DO EXEMPLO DE USO

O arquivo listado a seguir contém o código Association# para a camada de domínio exemplificada na Seção 4.2.

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: University - A example of using the Association#
 *          language
 *
 * Source File: /DomainLayer.cs
 *
 * Description: Source code of the domain layer for a univesity
 *               system used as use example of using the
 *               Association# language.
 *
 * Language: Association#
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using AssociationSharp.Lang;

namespace UniversitySystem.DomainLayer {
  //
  // Classes principais
  //
  public class Student {
    // atributos
    public string Name { set; get; }
    public int Number { set; get; }
    public int Year { set; get; }

    // association ends
    private rolename Lectures [1..*] in Attends;

    public Student(string name, int number, int year) {
      this.Name = name;
      this.Number = number;
```

```
      this.Year = year;
    }

    // operações
    public void AddLecture(Course course)
    {
      this#Lectures.Add(course);
    }

    public void RemoveLecture(Course course)
    {
      this#Lectures.Remove(course);
    }

    public void SetMarkOnLecture(Course course, int mark)
    {
      this#Lectures.GetAssociationLink(course).Mark = mark;
    }

    public int GetMarkOnLecture(Course course)
    {
      return this#Lectures.GetAssociationLink(course).Mark;
    }

    public IEnumerable<string> GetCoursesTitles()
    {
      return from lecture in this#Lectures.Get()
                select lecture.Title;
    }

    public void RemoveAllLectures()
    {
      this#Lectures.RemoveAll();
    }

    public bool IsAttendingCourse()
    {
      return this#Lectures.Count > 0;
    }
  }

  public class Course {
    // Atributos
    public string Title { set; get; }

    // Association ends
    private rolename Learners [*] in Attends;
    private rolename Lecturers [1..*] in Teaches;

    // Operações
    public Course(string title, IEnumerable<Faculty> lecturers)
    {
      this.Title = title;
      foreach (Faculty lecturer in lecturers) {
```

```
        this#Lecturers.Add(lecturer);
      }
    }

    public void RemoveAllLecturers() {
      this#Lecturers.RemoveAll();
    }

    public void AddLecturer(Faculty lecturer) {
      this#Lecturers.Add(lecturer);
    }

    public void RemoveLecturer(int lecturerNumber) {
      foreach (Faculty lecturer in this#Lecturers.Get()) {
        if (lecturer.Number == lecturerNumber) {
          this#Lecturers.Remove(lecturer);
          break;
        }
      }
    }

    public IEnumerable<int> GetLecturersNumbers() {
      return from lecturer in this#Lecturers.Get()
             select lecturer.Number;
    }

    public void AddLearner(Student learner) {
      this#Learners.Add(learner);
    }

    public void RemoveLearner(int studentNumber) {
      foreach (Student learner in this#Learners.Get()) {
        if (learner.Number == studentNumber) {
          this#Learners.Remove(learner);
          break;
        }
      }
    }

    public IEnumerable<int> GetLearnersNumbers() {
      return from learner in this#Learners.Get()
             select learner.Number;
    }
  }

  public class Faculty {
    // atributos
    public string Name { set; get; }
    public int Number { set; get; }

    // association ends
    private rolename Substitutes [*] in Substitutes;
    private rolename Substituted [*] in Substitutes;
    private rolename Lectures [*] in Teaches;
```

```
  // operações
  public Faculty(string name, int number) {
    this.Name = name;
    this.Number = number;
  }

  public bool HasSubstitutes() {
    return this#Substitutes.Count > 0;
  }

  public IEnumerable<int> GetSubstitutesNumbers() {
    HashSet<int> substitutesNumbers = new HashSet<int>();
    foreach(Faculty substitute in this#Substitutes.Get()) {
      substitutesNumbers.Add(substitute.Number);
    }
    return substitutesNumbers;
  }

  public void RemoveAllSubstitutes() {
    this#Substitutes.RemoveAll();
  }

  public void AddSelectedSubstitute(Faculty substitute) {
    this#Substitutes.Add(substitute);
  }

  public void RemoveSubstitute(Faculty substitute) {
    this#Substitutes.Remove(substitute);
  }

  public void RemoveAllSubstituted() {
    this#Substituted.RemoveAll();
  }
}

//
// Classes secundárias
//
public class InstructionLanguage {
  // atributos
  public string Description { set; get; }

  // association ends
  private rolename TAs [1..*] in AssistantToLanguage;

  // operações
  public InstructionLanguage(
    string instructionLanguageDescription)
  {
    this.Description = instructionLanguageDescription;
  }

  public int GetNumberOfTAs() {
```

```
      return this#TAs.Count;
    }
  }

  public class TA_Role {
    // association ends
    private rolename Player [1] in StudentToTA;
    private rolename Language [1] in AssistantToLanguage;
    private rolename ACs [*] in Assists;

    public atomic TA_Role(InstructionLanguage selectedLanguage)
    {
      this#Language.Add(selectedLanguage);
    }

    public void RemoveInstructionLanguage() {
      this#Language.RemoveAll();
    }

    public string GetInstructionLanguage() {
      return this#Language.Get().Single().Description;
    }

    public void AddAC(AC_Role assistedCourse) {
      this#ACs.Add(assistedCourse);
    }

    public void RemoveAC(AC_Role assistedCourse) {
      this#ACs.Remove(assistedCourse);
    }

    public IEnumerable<string> GetACsTitles() {
      return from ac in this#ACs.Get()
             select ac.GetPlayerTitle();
    }

    public void RemoveAllAC() {
      this#ACs.RemoveAll();
    }

    public int GetStudentNumber() {
      return this#Player.Get().Single().Number;
    }
  }

  public class AC_Role {
    // atributos
    public int MaxGroupSize;

    // association ends
    private rolename TAs [*] in Assists;
    private rolename Player [1] in CourseToAC;

    public AC_Role(int maxGroupSize) {
```

```
      this.MaxGroupSize = maxGroupSize;
    }

    public string GetPlayerTitle() {
      return this#Player.Get().Single().Title;
    }

    public void RemoveAllTA() {
      this#TAs.RemoveAll();
    }

    public IEnumerable<int> GetTAsNumbers() {
      return from ta in this#TAs.Get()
             select ta.GetStudentNumber();
    }
  }

  public class RA_Role {
    // atributos
    public float GrantAmount { set; get; }

    // association ends
    private rolename Player [1] in StudentToRA;
    private rolename Supervisors [*] in WorksFor;

    public RA_Role(float grantAmount) {
      this.GrantAmount = grantAmount;
    }

    public void AddSupervisor(Faculty faculty) {
      this#Supervisors.Add(faculty);
    }

    public void RemoveSupervisor(Faculty faculty) {
      this#Supervisors.Remove(faculty);
    }

    public IEnumerable<int> GetSupervisorsNumbers() {
      return from supervisor in this#Supervisors.Get()
             select supervisor.Number;
    }

    public int GetStudentNumber() {
      return this#Player.Get().Single().Number;
    }

    public void RemoveAllSupervisors() {
      this#Supervisors.RemoveAll();
    }
  }

  //
  // Controladora
  //
```

```
public class University {
  // Singleton
  public static readonly University INSTANCE =
    new University();
  private University() { }

  // Association ends
  private rolename Faculties [*] in UniversityToFaculty;
  private rolename Courses [*] in UniversityToCourse;
  private rolename Students [*] in UniversityToStudent;
  private rolename SelectedFaculty [0..1] in
    UniversityToSelectedFaculty;
  private rolename SelectedSubstitute[0..1] in
    UniversityToSelectedSubstitute;
  private rolename SelectedCourse [0..1] in
    UniversityToSelectedCourse;
  private rolename SelectedLecturers[*] in
    UniversityToSelectedLecturers;
  private rolename SelectedStudent [0..1] in
    UniversityToSelectedStudent;
  private rolename SelectedLanguage[0..1] in
    UniversityToSelectedLanguage;
  private rolename Languages [*] in UniversityToLanguage;

  // Cadastro de Professores
  public void RegisterFaculty(string name, int number) {
    atomic {
      this#Faculties.Add( new Faculty(name, number) );
    }
  }

  public bool SelectFaculty(int number) {
    foreach(Faculty faculty in this#Faculties.Get()) {
      if (faculty.Number == number) {
        this#SelectedFaculty.RemoveAll();
        this#SelectedFaculty.Add(faculty);
        return true;
      }
    }

    return false;
  }

  private Faculty GetSelectedFaculty() {
    return this#SelectedFaculty.Get().Single();
  }

  public void EditSelectedFaculty(string newName,
    int newNumber)
  {
    Faculty selected = GetSelectedFaculty();
    selected.Name = newName;
    selected.Number = newNumber;
  }
```

```
public void DeleteSelectedFaculty() {
  atomic {
    GetSelectedFaculty().RemoveAllSubstitutes();
    GetSelectedFaculty().RemoveAllSubstituted();
    this#Faculties.Remove(GetSelectedFaculty());
    this#SelectedFaculty.RemoveAll();
    this#SelectedSubstitute.RemoveAll();
  }
}

public string GetSelectedFacultyName() {
  return GetSelectedFaculty().Name;
}

public int GetSelectedFacultyNumber() {
  return GetSelectedFaculty().Number;
}

public IEnumerable<int> GetFacultiesNumbers() {
  List<int> numbers = new List<int>();
  foreach(Faculty faculty in this#Faculties.Get()) {
    numbers.Add(faculty.Number);
  }

  return numbers;
}

// Vínculo de Professores com seus Substitutos
public void SelectSubstitute(int substituteNumber) {
  this.SelectedSubstitute.RemoveAll();
  foreach(Faculty faculty in this#Faculties.Get()) {
    if (faculty.Number == substituteNumber) {
      this#SelectedSubstitute.Add(faculty);
      break;
    }
  }
}

public void UnselectSubstitute() {
  this#SelectedSubstitute.RemoveAll();
}

public string GetSelectedSubstituteName() {
  return GetSelectedSubstitute().Name;
}

private Faculty GetSelectedSubstitute() {
  return this#SelectedSubstitute.Get().Single();
}

public int GetSelectedSubstituteNumber() {
  return GetSelectedSubstitute().Number;
}
```

```
public void AddSelectedSubstituteToSelectedFaculty() {
  GetSelectedFaculty().AddSelectedSubstitute(
    GetSelectedSubstitute());
  this.UnselectSubstitute();
}

public void RemoveSelectedSubstituteFromSelectedFaculty() {
  GetSelectedFaculty().RemoveSubstitute(
    GetSelectedSubstitute());
}

public IEnumerable<int>
  GetSubstitutesNumbersFromSelectedFaculty()
{
  return GetSelectedFaculty().GetSubstitutesNumbers();
}

// Cadastro de Cursos
public void ClearSelectedLecturersForNewCourse() {
  this#SelectedLecturers.RemoveAll();
}

public void AddLecturerForNewCourse(int lecturerNumber) {
  foreach(Faculty faculty in this#Faculties.Get()) {
    if (faculty.Number == lecturerNumber) {
      if (faculty.HasSubstitutes()) {
        this#SelectedLecturers.Add(faculty);
      } else {
        throw new Exception();
      }
      break;
    }
  }
}

public void RemoveLecturerFromNewCourse(int lecturerNumber)
{
  foreach(Faculty lecturer in
    this#SelectedLecturers.Get())
  {
    if (lecturer.Number == lecturerNumber) {
      this#SelectedLecturers.Remove(lecturer);
      break;
    }
  }
}

public void RegisterCourse(string title,
  int maxGroupSizePerAssistant)
{
  atomic {
    Course course =
      new Course(title,this#SelectedLecturers.Get());
```

```
      this#Courses.Add( course );
      CourseToAC#ACRole(course).Add(
        new AC_Role(maxGroupSizePerAssistant));
    }
    ClearSelectedLecturersForNewCourse();
  }

  public IEnumerable<string> GetCoursesTitles() {
    return from course in this#Courses.Get()
           select course.Title;
  }

  public void SelectCourse(string courseTitle) {
    this#SelectedCourse.RemoveAll();
    foreach(Course course in this#Courses.Get()) {
      if (course.Title == courseTitle) {
        this#SelectedCourse.Add(course);
        break;
      }
    }
  }

  private Course GetSelectedCourse() {
    return this#SelectedCourse.Get().Single();
  }

  public void EditSelectedCourse(string newCourseTitle) {
    Course selectedCourse = GetSelectedCourse();
    selectedCourse.Title = newCourseTitle;
  }

  public void DeleteSelectedCourse() {
    CourseToAC#ACRole(GetSelectedCourse()).Get().Single().
      RemoveAllTA();
    atomic {
      CourseToAC#ACRole(GetSelectedCourse()).RemoveAll();
      GetSelectedCourse().RemoveAllLecturers();
      this#Courses.Remove(GetSelectedCourse());
      this#SelectedCourse.RemoveAll();
    }
  }

  // Papel de Curso Assistido (AC_Role).
  public void SelectedCourseResetMaxGroupSize(
    int maxGroupSizePerAssistant)
  {
    CourseToAC#ACRole(GetSelectedCourse()).Get().Single().
      MaxGroupSize = maxGroupSizePerAssistant;
  }

  public int GetMaxSizeGroupForSelectedCourse() {
    return CourseToAC#ACRole(GetSelectedCourse()).Get().
      Single().MaxGroupSize;
  }
```

```
// Vínculo de Professores a Cursos
public void AddLecturerForSelectedCourse(
  int lecturerNumber)
{
  foreach(Faculty lecturer in this#Faculties.Get()) {
    if (lecturer.Number == lecturerNumber) {
      GetSelectedCourse().AddLecturer(lecturer);
    }
  }
}

public void RemoveLecturerFromSelectedCourse(
  int lecturerNumber)
{
  CourseToAC#ACRole(GetSelectedCourse()).Get().Single().
    RemoveAllTA();
}

public IEnumerable<int>
  GetLecturersNumbersFromSelectedCourse()
{
  return GetSelectedCourse().GetLecturersNumbers();
}

//Cadastro de Estudantes
public void RegisterLecturerOnSelectedCourse(
  string studentName,
  int studentNumber)
{
  atomic {
    Student learner =
      new Student(studentName,studentNumber,1);
    this#Students.Add( learner );
    GetSelectedCourse().AddLearner( learner );
  }
}

public void SelectStudent(int studentNumber) {
  this#SelectedStudent.RemoveAll();
  foreach(Student student in this.Students.Get()) {
    if (student.Number == studentNumber) {
      this#SelectedStudent.Add(student);
      break;
    }
  }
}

public IEnumerable<int> GetStudentsNumbers() {
  return from student in this#Students.Get()
           select student.Number;
}

private Student GetSelectedStudent() {
  return this#SelectedStudent.Get().Single();
```

```
    }

    public string GetSelectedStudentName() {
      return GetSelectedStudent().Name;
    }

    public int GetSelectedStudentNumber() {
      return GetSelectedStudent().Number;
    }

    public int GetSelectedStudentYear() {
      return GetSelectedStudent().Year;
    }

    public void DeleteSelectedStudent() {
      atomic {
        Student student = GetSelectedStudent();
        student.RemoveAllLectures();
        if (StudentToTA#TARole(student).Count > 0) {
          StudentToTA#TARole(student).Get().Single().
            RemoveAllAC();
          StudentToTA#TARole(student).Get().Single().
            RemoveInstructionLanguage();
          StudentToTA#TARole(student).RemoveAll();
        }
        if (StudentToRA#RARole(student).Count > 0) {
          StudentToRA#RARole(student).Get().Single().
            RemoveAllSupervisors();
          StudentToRA#RARole(student).RemoveAll();
        }
        this#Students.Remove(student);
        student = null;
        this#SelectedStudent.RemoveAll();
      }
    }

    // Vínculo de estudantes a cursos lecionados.
    public void AddLearnerForSelectedCourse(int studentNumber)
    {
      foreach(Student learner in this#Students.Get()) {
        if (learner.Number == studentNumber) {
          GetSelectedCourse().AddLearner( learner );
        }
      }
    }

    public void RemoveLearnerFromSelectedCourse(
      int studentNumber)
    {
      GetSelectedCourse().RemoveLearner(studentNumber);
    }

    // Papel de Assistente (TA_Role).
    public void SelectedStudentCanBeTeachingAssistant(
```

```
    string instructionLanguageDescription)
  {
    InstructionLanguage selectedLanguage = null;
    foreach(InstructionLanguage language in
      this#Languages.Get())
    {
      if (language.Description ==
           instructionLanguageDescription)
      {
        selectedLanguage = language;
        break;
      }
    }

    atomic {
      if (selectedLanguage == null) {
        selectedLanguage =
          new InstructionLanguage(
            instructionLanguageDescription);
        this#Languages.Add(selectedLanguage);
      }
      StudentToTA#TARole(GetSelectedStudent()).Add(
        new TA_Role(selectedLanguage));
    }
  }

  public void SelectedStudentCannotBeTeachingAssistant() {
    TA_Role ta =
      StudentToTA#TARole(GetSelectedStudent()).Get().
        Single();
    string languageDescription =
      ta.GetInstructionLanguage();
    InstructionLanguage language =
      (from lang in this#Languages.Get()
       where lang.Description == languageDescription
       select lang).Single();

    atomic {
      ta.RemoveInstructionLanguage();
      StudentToTA#TARole(GetSelectedStudent()).Remove(ta);
      ta = null;
      if (language.GetNumberOfTAs() == 0) {
        this#Languages.Remove(language);
        language = null;
      }
    }
  }

  public string GetInstructionLanguageOfSelectedStudent() {
    return StudentToTA#TARole(GetSelectedStudent()).Get().
      Single().GetInstructionLanguage();
  }

  // Vínculo de Assistente com Curso
```

```
public void AddSelectedTAToSelectedAssitedCourse() {
  StudentToTA#TARole(GetSelectedStudent()).Get().Single().
    AddAC(CourseToAC#ACRole(
      GetSelectedCourse()).Get().Single());
}

public void RemoveSelectedTAFromSelectedAssitedCourse() {
  StudentToTA#TARole(GetSelectedStudent()).Get().Single()
    .RemoveAC(CourseToAC#ACRole(GetSelectedCourse()).
      Get().Single());
}

public IEnumerable<string> GetSelectedTACoursesTitles() {
  return StudentToTA#TARole(GetSelectedStudent()).Get().
    Single().GetACsTitles();
}

public IEnumerable<int> GetSelectedACAssistantsNumbers() {
  return CourseToAC#ACRole(GetSelectedCourse()).Get().
    Single().GetTAsNumbers();
}

// Vínculo de Aluno com Curso (Attends)
public void AddSelectedLearnerToSelectedLecture() {
  GetSelectedStudent().AddLecture(GetSelectedCourse());
}

public void RemoveSelectedLearnerFromSelectedLecture() {
  atomic {
    GetSelectedStudent().RemoveLecture(
      GetSelectedCourse());
    if (! GetSelectedStudent().IsAttendingCourse()) {
      this.DeleteSelectedStudent();
    }
  }
}

public void SetMarkForSelectedLearnerOnSelectedLecture(
  int mark)
{
  GetSelectedStudent().SetMarkOnLecture(
    GetSelectedCourse(),mark);
}

public int GetMarkForSelectedLearnerOnSelectedLecture() {
  return GetSelectedStudent().GetMarkOnLecture(
    GetSelectedCourse());
}

public IEnumerable<string>
  GetLecturesTitlesOfSelectedLearner()
{
  return GetSelectedStudent().GetCoursesTitles();
}
```

```
public IEnumerable<int>
  GetLearnersNumbersOfSelectedLecture()
{
  return GetSelectedCourse().GetLearnersNumbers();
}

// Estudante com papel de Pesquisador (RA_Role)
public void SelectedStudentIsAResearchAssistant(
  float grantAmount)
{
  atomic {
    StudentToRA#RARole(GetSelectedStudent()).Add(
      new RA_Role(grantAmount));
  }
}

public void SelectedStudentIsNotAResearchAssistant() {
  StudentToRA#RARole(GetSelectedStudent()).RemoveAll();
}

public void SetGrantAmountForSelectedResearchAssistant(
  float newGrantAmount)
{
  StudentToRA#RARole(GetSelectedStudent()).Get().Single().
    GrantAmount = newGrantAmount;
}

public float GetGrantAmountForSelectedResearchAssistant() {
  return StudentToRA#RARole(GetSelectedStudent()).Get().
    Single().GrantAmount;
}

// Vínculo de Pesquisador com Supervisor.
public void
  AddSelectedResearchAssistantToSelectedSupervisor()
{
  StudentToRA#RARole(GetSelectedStudent()).Get().Single().
    AddSupervisor(GetSelectedFaculty());
}

public void
  RemoveSelectedResearchAssistantFromSelectedSupervisor()
{
  StudentToRA#RARole(GetSelectedStudent()).Get().Single().
    RemoveSupervisor(GetSelectedFaculty());
}

public IEnumerable<int>
  GetSupervisorsNumbersFromSelectedResearchAssistant()
{
  return
    StudentToRA#RARole(GetSelectedStudent()).Get().
      Single().GetSupervisorsNumbers();
```

```
    }

    Public IEnumerable<int>
      GetResearchAssistantsNumbersFromSelectedSupervisor()
    {
      return from ra in
             WorksFor#RAs(GetSelectedFaculty()).Get()
             select ra.GetStudentNumber();
    }
  }

  //
  // Associações e Classes de Associação persistentes.
  //
  association UniversityToStudent
    between University [1]
    and Student;

  association UniversityToFaculty
    between University [1]
    and Faculty;

  association UniversityToLanguage
    between University [1]
    and InstructionLanguage;

  association UniversityToCourse
    between University [1]
    and Course;

  association Teaches
    between Faculty
    and Course;

  association AssistantToLanguage
    between TA_Role
    and InstructionLanguage;

  association Assists
    between TA_Role
    and AC_Role;

  association StudentToTA
    between Student
    and TA_Role TARole [0..1];

  association CourseToAC
    between Course
    and AC_Role ACRole [1];

  association class Attends
    between Student
    and Course
  {
```

```
    public int Mark { set; get; }
  }

  association Substitutes
    between Faculty
    and Faculty;

  association StudentToRA
    between Student
    and RA_Role RARole [0..1];

  association WorksFor
    between RA_Role RAs [*]
    and Faculty;

  //
  // Associações e Classes de Associação temporárias
  //
  association UniversityToSelectedStudent
    between University [0..1]
    and Student;

  association UniversityToSelectedFaculty
    between University [0..1]
    and Faculty;

  association UniversityToSelectedLanguage
    between University [0..1]
    and InstructionLanguage;

  association UniversityToSelectedCourse
    between University [0..1]
    and Course;

  association UniversityToSelectedSubstitute
    between University [0..1]
    and Faculty;

  association UniversityToSelectedLecturers
    between University [0..1]
    and Faculty;
}
```

A listagem a seguir apresenta o código C# gerado a partir da tradução manual do código do arquivo DomainLayer.as, listado anteriormente.

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: University - A example of using the Association#
 *          language
 *
 * Source File: /DomainLayer.cs
 *
 * Description: Tranlated code from the source file
 *              DomainLayer.as written in Association#.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using AssociationSharp.Lang;

namespace UniversitySystem.DomainLayer
{
  //
  // Classes principais
  //
  public class Student
  {
    // atributos
    public string Name { set; get; }
    public int Number { set; get; }
    public int Year { set; get; }

    // association ends
    public readonly Attends.Association.EndForByStudent
      Lectures;
    public readonly UniversityToStudent.EndOwnedByStudent
      __AssociationSharp_HideElement_UniversityToStudent;
    public readonly
      UniversityToSelectedStudent.EndOwnedByStudent
__AssociationSharp_HideElement_UniversityToSelectedStudent;
    public readonly StudentToTA.EndOwnedByStudent
      __AssociationSharp_HideElement_TARole;
    public readonly StudentToRA.EndOwnedByStudent
```

```
      __AssociationSharp_HideElement_RARole;

    public Student(string name, int number, int year)
    {
      this.Lectures =
        new Attends.Association.EndForByStudent(this);
      this.__AssociationSharp_HideElement_UniversityToStudent=
        new UniversityToStudent.EndOwnedByStudent(this);
      this.
__AssociationSharp_HideElement_UniversityToSelectedStudent
        = new
        UniversityToSelectedStudent.EndOwnedByStudent(this);
      this.__AssociationSharp_HideElement_TARole =
        new StudentToTA.EndOwnedByStudent(this);
      this.__AssociationSharp_HideElement_RARole =
        new StudentToRA.EndOwnedByStudent(this);

      this.Name = name;
      this.Number = number;
      this.Year = year;
    }

    // operações
    public void AddLecture(Course course)
    {
      this.Lectures.Add(course);
    }

    public void RemoveLecture(Course course)
    {
      this.Lectures.Remove(course);
    }

    public void SetMarkOnLecture(Course course, int mark)
    {
      this.Lectures.GetAssociationLink(course).Mark = mark;
    }

    public int GetMarkOnLecture(Course course)
    {
      return this.Lectures.GetAssociationLink(course).Mark;
    }

    public IEnumerable<string> GetCoursesTitles()
    {
      return from lecture in this.Lectures.Get()
             select lecture.Title;
    }

    public void RemoveAllLectures()
    {
      this.Lectures.RemoveAll();
    }
```

```
    public bool IsAttendingCourse()
    {
      return this.Lectures.Count > 0;
    }
  }

  public class Course
  {
    // Atributos
    public string Title { set; get; }

    // Association ends
    public readonly Attends.Association.EndOwnedByCourse
      Learners;
    public readonly Teaches.EndOwnedByCourse Lecturers;
    public readonly UniversityToCourse.EndOwnedByCourses
      __AssociationSharp_HideElement_UniversityToCourse;
    public readonly UniversityToSelectedCourse.EndOwnedByCourse
__AssociationSharp_HideElement_UniversityToSelectedCourse;
    public readonly CourseToAC.EndOwnedByCourse
      __AssociationSharp_HideElement_ACRole;

    // Operações
    public Course(string title, IEnumerable<Faculty> lecturers)
    {
      this.Learners =
        new Attends.Association.EndOwnedByCourse(this);
      this.Lecturers = new Teaches.EndOwnedByCourse(this);
      this.__AssociationSharp_HideElement_UniversityToCourse =
        new UniversityToCourse.EndOwnedByCourses(this);
      this.
__AssociationSharp_HideElement_UniversityToSelectedCourse =
        new UniversityToSelectedCourse.EndOwnedByCourse(this);
      this.__AssociationSharp_HideElement_ACRole =
        new CourseToAC.EndOwnedByCourse(this);

      this.Title = title;
      foreach (Faculty lecturer in lecturers)
      {
        this.Lecturers.Add(lecturer);
      }
    }

    public void RemoveAllLecturers()
    {
      this.Lecturers.RemoveAll();
    }

    public void AddLecturer(Faculty lecturer)
    {
      this.Lecturers.Add(lecturer);
    }

    public void RemoveLecturer(int lecturerNumber)
```

```
    {
      foreach (Faculty lecturer in this.Lecturers.Get())
      {
        if (lecturer.Number == lecturerNumber)
        {
          this.Lecturers.Remove(lecturer);
          break;
        }
      }
    }

    public IEnumerable<int> GetLecturersNumbers()
    {
      return from lecturer in this.Lecturers.Get()
             select lecturer.Number;

    }

    public void AddLearner(Student learner)
    {
      this.Learners.Add(learner);
    }

    public void RemoveLearner(int studentNumber)
    {
      foreach (Student learner in this.Learners.Get())
      {
        if (learner.Number == studentNumber)
        {
          this.Learners.Remove(learner);
          break;
        }
      }
    }

    public IEnumerable<int> GetLearnersNumbers()
    {
      return from learner in this.Learners.Get()
              select learner.Number;
    }
  }

  public class Faculty
  {
    // atributos
    public string Name { set; get; }
    public int Number { set; get; }

    // association ends
    public readonly Substitutes.EndOwnedBySubstituted
      Substitutes;
    public readonly Substitutes.EndOwnedBySubstitutes
      Substituted;
    public readonly WorksFor.EndOwnedByFaculty RAs;
```

```
    public readonly Teaches.EndOwnedByFaculty Lectures;
    public readonly UniversityToFaculty.EndOwnedByFaculty
      __AssociationSharp_HideElement_UniversityToFaculty;
    public readonly UniversityToSelectedLecturers.
        EndOwnedBySelectedLecturers
__AssociationSharp_HideElement_UniversityToSelectedLecturers;
    public readonly
      UniversityToSelectedFaculty.EndOwnedByFaculty
__AssociationSharp_HideElement_UniversityToSelectedFaculty;
    public readonly UniversityToSelectedSubstitute.
      EndOwnedBySelectedSubstitute
__AssociationSharp_HideElement_UniversityToSelectedSubstitute;

    // operações
    public Faculty(string name, int number)
    {
      this.Substitutes =
        new Substitutes.EndOwnedBySubstituted(this);
      this.Substituted =
        new Substitutes.EndOwnedBySubstitutes(this);
      this.RAs = new WorksFor.EndOwnedByFaculty(this);
      this.Lectures = new Teaches.EndOwnedByFaculty(this);
      this.__AssociationSharp_HideElement_UniversityToFaculty=
        new UniversityToFaculty.EndOwnedByFaculty(this);
      this.
__AssociationSharp_HideElement_UniversityToSelectedLecturers =
        new UniversityToSelectedLecturers.
          EndOwnedBySelectedLecturers(this);
      this.
__AssociationSharp_HideElement_UniversityToSelectedFaculty =
        new UniversityToSelectedFaculty.
          EndOwnedByFaculty(this);
      this.
__AssociationSharp_HideElement_UniversityToSelectedSubstitute =
        new UniversityToSelectedSubstitute.
          EndOwnedBySelectedSubstitute(this);

      this.Name = name;
      this.Number = number;
    }

    public bool HasSubstitutes()
    {
      return this.Substitutes.Count > 0;
    }

    public IEnumerable<int> GetSubstitutesNumbers()
    {
      HashSet<int> substitutesNumbers = new HashSet<int>();
      foreach (Faculty substitute in this.Substitutes.Get())
      {
        substitutesNumbers.Add(substitute.Number);
      }
      return substitutesNumbers;
```

```
    }

    public void RemoveAllSubstitutes()
    {
      this.Substitutes.RemoveAll();
    }

    public void AddSelectedSubstitute(Faculty substitute)
    {
      this.Substitutes.Add(substitute);
    }

    public void RemoveSubstitute(Faculty substitute)
    {
      this.Substitutes.Remove(substitute);
    }

    public void RemoveAllSubstituted()
    {
      this.Substituted.RemoveAll();
    }
  }

  //
  // Classes secundárias
  //
  public class InstructionLanguage
  {
    // atributos
    public string Description { set; get; }

    // association ends
    public readonly AssistantToLanguage.
      EndOwnedByInstructionLanguage TAs;
    public readonly UniversityToLanguage.EndOwnedByLanguage
      __AssociationSharp_HideElement_UniversityToLanguage;
    public readonly
      UniversityToSelectedLanguage.EndOwnedByLanguage
__AssociationSharp_HideElement_UniversityToSelectedLanguage;

    // operações
    public InstructionLanguage(
      string instructionLanguageDescription)
    {
      this.TAs = new AssistantToLanguage.
        EndOwnedByInstructionLanguage(this);
      this.
__AssociationSharp_HideElement_UniversityToSelectedLanguage =
        new UniversityToSelectedLanguage.
          EndOwnedByLanguage(this);
      this.
        __AssociationSharp_HideElement_UniversityToLanguage =
          new UniversityToLanguage.EndOwnedByLanguage(this);
```

```
      this.Description = instructionLanguageDescription;
    }

    public int GetNumberOfTAs()
    {
      return this.TAs.Count;
    }
  }

  public class TA_Role
  {
    // association ends
    public readonly StudentToTA.EndOwnedByTA Player;
    public readonly AssistantToLanguage.EndOwnedByTA Language;
    public readonly Assists.EndOwnedByTA ACs;

    public TA_Role(InstructionLanguage selectedLanguage)
    {
      try
      {
        this.Player = new StudentToTA.EndOwnedByTA(this);
        this.Language =
          new AssistantToLanguage.EndOwnedByTA(this);
        this.ACs = new Assists.EndOwnedByTA(this);

        AssociationSharpSystem.__DisableSystemConstraints();
        this.Language.Add(selectedLanguage);
      }
      finally
      {
        AssociationSharpSystem.__EnableSystemConstraints();
      }
    }

    public void RemoveInstructionLanguage()
    {
      this.Language.RemoveAll();
    }

    public string GetInstructionLanguage()
    {
      return this.Language.Get().Single().Description;
    }

    public void AddAC(AC_Role assistedCourse)
    {
      this.ACs.Add(assistedCourse);
    }

    public void RemoveAC(AC_Role assistedCourse)
    {
      this.ACs.Remove(assistedCourse);
    }
```

```
    public IEnumerable<string> GetACsTitles()
    {
      return from ac in this.ACs.Get()
             select ac.GetPlayerTitle();
    }

    public void RemoveAllAC()
    {
      this.ACs.RemoveAll();
    }

    public int GetStudentNumber()
    {
      return this.Player.Get().Single().Number;
    }
  }

  public class AC_Role
  {
    // atributos
    public int MaxGroupSize;

    // association ends
    public readonly Assists.EndOwnedByAC TAs;
    public readonly CourseToAC.EndOwnedByAC Player;

    public AC_Role(int maxGroupSize)
    {
      this.TAs = new Assists.EndOwnedByAC(this);
      this.Player = new CourseToAC.EndOwnedByAC(this);

      this.MaxGroupSize = maxGroupSize;
    }

    public string GetPlayerTitle()
    {
      return this.Player.Get().Single().Title;
    }

    public void RemoveAllTA()
    {
      this.TAs.RemoveAll();
    }

    public IEnumerable<int> GetTAsNumbers()
    {
      return from ta in this.TAs.Get()
              select ta.GetStudentNumber();
    }
  }

  public class RA_Role
  {
    // atributos
```

```
    public float GrantAmount { set; get; }

    // association ends
    public readonly StudentToRA.EndOwnedByRA Player;
    public readonly WorksFor.EndOwnedByRA Supervisors;

    public RA_Role(float grantAmount)
    {
      this.Player = new StudentToRA.EndOwnedByRA(this);
      this.Supervisors = new WorksFor.EndOwnedByRA(this);

      this.GrantAmount = grantAmount;
    }

    public void AddSupervisor(Faculty faculty)
    {
      this.Supervisors.Add(faculty);
    }

    public void RemoveSupervisor(Faculty faculty)
    {
      this.Supervisors.Remove(faculty);
    }

    public IEnumerable<int> GetSupervisorsNumbers()
    {
      return from supervisor in this.Supervisors.Get()
             select supervisor.Number;
    }

    public int GetStudentNumber()
    {
      return this.Player.Get().Single().Number;
    }

    public void RemoveAllSupervisors()
    {
      this.Supervisors.RemoveAll();
    }
  }

  //
  // Controladora
  //
  public class University
  {
    // Singleton
    public static readonly University INSTANCE =
      new University();
    private University()
    {
      this.Faculties =
        new UniversityToFaculty.EndOwnedByUniversity(this);
      this.Courses =
```

```
            new UniversityToCourse.EndOwnedByUniversity(this);
        this.Students =
            new UniversityToStudent.EndOwnedByUniversity(this);
        this.SelectedFaculty =
            new UniversityToSelectedFaculty.
              EndOwnedByUniversity(this);
        this.SelectedSubstitute = new
            UniversityToSelectedSubstitute.EndOwnedByUniversity(
              this);
        this.SelectedCourse =
            new UniversityToSelectedCourse.EndOwnedByUniversity(
              this);
        this.SelectedLecturers =
            new UniversityToSelectedLecturers.
              EndOwnedByUniversity(this);
        this.SelectedStudent =
            new UniversityToSelectedStudent.EndOwnedByUniversity(
              this);
        this.SelectedLanguage =
            new UniversityToSelectedLanguage.EndOwnedByUniversity(
              this);
        this.Languages =
            new UniversityToLanguage.EndOwnedByUniversity(this);
    }

    // Association ends
    public readonly UniversityToFaculty.EndOwnedByUniversity
      Faculties;
    public readonly UniversityToCourse.EndOwnedByUniversity
      Courses;
    public readonly UniversityToStudent.EndOwnedByUniversity
      Students;
    public readonly
      UniversityToSelectedFaculty.EndOwnedByUniversity
        SelectedFaculty;
    public readonly
      UniversityToSelectedSubstitute.EndOwnedByUniversity
        SelectedSubstitute;
    public readonly
      UniversityToSelectedCourse.EndOwnedByUniversity
        SelectedCourse;
    public readonly
      UniversityToSelectedLecturers.EndOwnedByUniversity
        SelectedLecturers;
    public readonly
      UniversityToSelectedStudent.EndOwnedByUniversity
        SelectedStudent;
    public readonly
      UniversityToSelectedLanguage.EndOwnedByUniversity
        SelectedLanguage;
    public readonly UniversityToLanguage.EndOwnedByUniversity
      Languages;

    // Cadastro de Professores
```

```
public void RegisterFaculty(string name, int number)
{
  try
  {
    AssociationSharpSystem.__DisableSystemConstraints();

    this.Faculties.Add(new Faculty(name, number));

  }
  finally
  {
    AssociationSharpSystem.__EnableSystemConstraints();
  }
}

public bool SelectFaculty(int number)
{
  foreach (Faculty faculty in this.Faculties.Get())
  {
    if (faculty.Number == number)
    {
      this.SelectedFaculty.RemoveAll();
      this.SelectedFaculty.Add(faculty);
      return true;
    }
  }

  return false;
}

private Faculty GetSelectedFaculty()
{
  return this.SelectedFaculty.Get().Single();
}

public void EditSelectedFaculty(string newName,
  int newNumber)
{
  Faculty selected = GetSelectedFaculty();
  selected.Name = newName;
  selected.Number = newNumber;
}

public void DeleteSelectedFaculty()
{
  try
  {
    AssociationSharpSystem.__DisableSystemConstraints();
    GetSelectedFaculty().RemoveAllSubstitutes();
    GetSelectedFaculty().RemoveAllSubstituted();
    this.Faculties.Remove(GetSelectedFaculty());
    this.SelectedFaculty.RemoveAll();
    this.SelectedSubstitute.RemoveAll();
```

```
    }
    finally
    {
      AssociationSharpSystem.__EnableSystemConstraints();
    }
  }

  public string GetSelectedFacultyName()
  {
    return GetSelectedFaculty().Name;
  }

  public int GetSelectedFacultyNumber()
  {
    return GetSelectedFaculty().Number;
  }

  public IEnumerable<int> GetFacultiesNumbers()
  {
    List<int> numbers = new List<int>();
    foreach (Faculty faculty in this.Faculties.Get())
    {
      numbers.Add(faculty.Number);
    }
    return numbers;
  }

  // Vínculo de Professores com seus Substitutos
  public void SelectSubstitute(int substituteNumber)
  {
    this.SelectedSubstitute.RemoveAll();
    foreach (Faculty faculty in this.Faculties.Get())
    {
      if (faculty.Number == substituteNumber)
      {
        this.SelectedSubstitute.Add(faculty);
        break;
      }
    }
  }

  public void UnselectSubstitute()
  {
    this.SelectedSubstitute.RemoveAll();
  }

  public string GetSelectedSubstituteName()
  {
    return GetSelectedSubstitute().Name;
  }

  private Faculty GetSelectedSubstitute()
  {
    return this.SelectedSubstitute.Get().Single();
```

```
    }

    public int GetSelectedSubstituteNumber()
    {
      return GetSelectedSubstitute().Number;
    }

    public void AddSelectedSubstituteToSelectedFaculty()
    {
      GetSelectedFaculty().AddSelectedSubstitute(
        GetSelectedSubstitute());
      this.UnselectSubstitute();
    }

    public void RemoveSelectedSubstituteFromSelectedFaculty()
    {
      GetSelectedFaculty().RemoveSubstitute(
        GetSelectedSubstitute());
    }

    public IEnumerable<int>
      GetSubstitutesNumbersFromSelectedFaculty()
    {
      return GetSelectedFaculty().GetSubstitutesNumbers();
    }

    // Cadastro de Cursos
    public void ClearSelectedLecturersForNewCourse()
    {
      this.SelectedLecturers.RemoveAll();
    }

    public void AddLecturerForNewCourse(int lecturerNumber)
    {
      foreach (Faculty faculty in this.Faculties.Get())
      {
        if (faculty.Number == lecturerNumber)
        {
          if (faculty.HasSubstitutes())
          {
            this.SelectedLecturers.Add(faculty);
          }
          else
          {
            throw new Exception();
          }
          break;
        }
      }
    }

    public void RemoveLecturerFromNewCourse(int lecturerNumber)
    {
      foreach (Faculty lecturer in
```

```
      this.SelectedLecturers.Get())
    {
      if (lecturer.Number == lecturerNumber)
      {
        this.SelectedLecturers.Remove(lecturer);
        break;
      }
    }
  }

  public void RegisterCourse(string title,
    int maxGroupSizePerAssistant)
  {
    try
    {
      AssociationSharpSystem.__DisableSystemConstraints();

      Course course = new Course(title,
        this.SelectedLecturers.Get());
      this.Courses.Add(course);
      course.__AssociationSharp_HideElement_ACRole.Add(
        new AC_Role(maxGroupSizePerAssistant));

    }
    finally
    {
      AssociationSharpSystem.__EnableSystemConstraints();
    }

    ClearSelectedLecturersForNewCourse();
  }

  public IEnumerable<string> GetCoursesTitles()
  {
    return from course in this.Courses.Get()
           select course.Title;
  }

  public void SelectCourse(string courseTitle)
  {
    this.SelectedCourse.RemoveAll();
    foreach (Course course in this.Courses.Get())
    {
      if (course.Title == courseTitle)
      {
        this.SelectedCourse.Add(course);
        break;
      }
    }
  }

  private Course GetSelectedCourse()
  {
    return this.SelectedCourse.Get().Single();
```

```
    }

    public void EditSelectedCourse(string newCourseTitle)
    {
      Course selectedCourse = GetSelectedCourse();
      selectedCourse.Title = newCourseTitle;
    }

    public void DeleteSelectedCourse()
    {
      GetSelectedCourse().
        __AssociationSharp_HideElement_ACRole.Get().Single().
          RemoveAllTA();
      try
      {
        AssociationSharpSystem.__DisableSystemConstraints();

        GetSelectedCourse().
          __AssociationSharp_HideElement_ACRole.RemoveAll();
        GetSelectedCourse().RemoveAllLecturers();
        this.Courses.Remove(GetSelectedCourse());
        this.SelectedCourse.RemoveAll();
      }
      finally
      {
        AssociationSharpSystem.__EnableSystemConstraints();
      }
    }

    // Papel de Curso Assistido (AC_Role).
    public void SelectedCourseResetMaxGroupSize(
      int maxGroupSizePerAssistant)
    {
      GetSelectedCourse().
        __AssociationSharp_HideElement_ACRole.Get().Single().
          MaxGroupSize = maxGroupSizePerAssistant;
    }

    public int GetMaxSizeGroupForSelectedCourse()
    {
      return GetSelectedCourse().
        __AssociationSharp_HideElement_ACRole.Get().Single().
          MaxGroupSize;
    }

    // Vínculo de Professores a Cursos
    public void AddLecturerForSelectedCourse(
      int lecturerNumber)
    {
      foreach (Faculty lecturer in this.Faculties.Get())
      {
        if (lecturer.Number == lecturerNumber)
        {
          GetSelectedCourse().AddLecturer(lecturer);
```

```
      }
    }
  }

  public void RemoveLecturerFromSelectedCourse(
    int lecturerNumber)
  {
    GetSelectedCourse().RemoveLecturer(lecturerNumber);
  }

  public IEnumerable<int>
    GetLecturersNumbersFromSelectedCourse()
  {
    return GetSelectedCourse().GetLecturersNumbers();
  }

  //Cadastro de Estudantes
  public void RegisterLecturerOnSelectedCourse(
    string studentName, int studentNumber)
  {
    try
    {
      AssociationSharpSystem.__DisableSystemConstraints();

      Student learner = new Student(
        studentName, studentNumber, 1);
      this.Students.Add(learner);
      GetSelectedCourse().AddLearner(learner);
    }
    finally
    {
      AssociationSharpSystem.__EnableSystemConstraints();
    }
  }

  public void SelectStudent(int studentNumber)
  {
    this.SelectedStudent.RemoveAll();
    foreach (Student student in this.Students.Get())
    {
      if (student.Number == studentNumber)
      {
        this.SelectedStudent.Add(student);
        break;
      }
    }
  }

  public IEnumerable<int> GetStudentsNumbers()
  {
    return from student in this.Students.Get()
           select student.Number;
  }
```

```
private Student GetSelectedStudent()
{
  return this.SelectedStudent.Get().Single();
}

public string GetSelectedStudentName()
{
  return GetSelectedStudent().Name;
}

public int GetSelectedStudentNumber()
{
  return GetSelectedStudent().Number;
}

public int GetSelectedStudentYear()
{
  return GetSelectedStudent().Year;
}

public void DeleteSelectedStudent()
{
  try
  {
    AssociationSharpSystem.__DisableSystemConstraints();

    Student student = GetSelectedStudent();
    student.RemoveAllLectures();
    if (student.__AssociationSharp_HideElement_TARole.
      Count > 0)
    {
      student.__AssociationSharp_HideElement_TARole.
        Get().Single().RemoveAllAC();
      student.__AssociationSharp_HideElement_TARole.
        Get().Single().RemoveInstructionLanguage();
      student.__AssociationSharp_HideElement_TARole.
        RemoveAll();
    }
    if (student.__AssociationSharp_HideElement_RARole.
      Count > 0)
    {
      student.__AssociationSharp_HideElement_RARole.
        Get().Single().RemoveAllSupervisors();
      student.__AssociationSharp_HideElement_RARole.
        RemoveAll();
    }
    this.Students.Remove(student);
    student = null;
    this.SelectedStudent.RemoveAll();
  }
  finally
  {
    AssociationSharpSystem.__EnableSystemConstraints();
  }
```

```
    }

    // Vínculo de estudantes a cursos lecionados.
    public void AddLearnerForSelectedCourse(int studentNumber)
    {
      foreach (Student learner in this.Students.Get())
      {
        if (learner.Number == studentNumber)
        {
          GetSelectedCourse().AddLearner(learner);
        }
      }
    }

    public void RemoveLearnerFromSelectedCourse(
      int studentNumber)
    {
      GetSelectedCourse().RemoveLearner(studentNumber);
    }

    // Papel de Assistente (TA_Role).
    public void SelectedStudentCanBeTeachingAssistant(
      string instructionLanguageDescription)
    {
      InstructionLanguage selectedLanguage = null;
      foreach (InstructionLanguage language in
        this.Languages.Get())
      {
        if (language.Description ==
          instructionLanguageDescription)
        {
          selectedLanguage = language;
          break;
        }
      }

      try
      {
        AssociationSharpSystem.__DisableSystemConstraints();

        if (selectedLanguage == null)
        {
          selectedLanguage = new InstructionLanguage(
            instructionLanguageDescription);
          this.Languages.Add(selectedLanguage);
        }
        GetSelectedStudent().
          __AssociationSharp_HideElement_TARole.Add(
            new TA_Role(selectedLanguage));
      }
      finally
      {
        AssociationSharpSystem.__EnableSystemConstraints();
      }
```

```
    }

    public void SelectedStudentCannotBeTeachingAssistant()
    {
      TA_Role taRole =
        GetSelectedStudent().
          __AssociationSharp_HideElement_TARole.Get().
            Single();
      string languageDescription =
        taRole.GetInstructionLanguage();
      InstructionLanguage language =
        (from lang in this.Languages.Get()
        where lang.Description == languageDescription
        select lang).Single();

      try
      {
        AssociationSharpSystem.__DisableSystemConstraints();

        taRole.RemoveInstructionLanguage();
        GetSelectedStudent().
          __AssociationSharp_HideElement_TARole.Remove(
            taRole);
        taRole = null;
        if (language.GetNumberOfTAs() == 0)
        {
          this.Languages.Remove(language);
          language = null;
        }
      }
      finally
      {
        AssociationSharpSystem.__EnableSystemConstraints();
      }
    }

    public string GetInstructionLanguageOfSelectedStudent()
    {
      return this.GetSelectedStudent().
        __AssociationSharp_HideElement_TARole.Get().
          Single().GetInstructionLanguage();
    }

    // Vínculo de Assistente com Curso
    public void AddSelectedTAToSelectedAssitedCourse()
    {
      GetSelectedStudent().
        __AssociationSharp_HideElement_TARole.Get().
          Single().AddAC(
            GetSelectedCourse().
              __AssociationSharp_HideElement_ACRole.
                Get().Single());
    }
```

```
public void RemoveSelectedTAFromSelectedAssitedCourse()
{
  GetSelectedStudent().
    __AssociationSharp_HideElement_TARole.Get().Single().
      RemoveAC(GetSelectedCourse().
        __AssociationSharp_HideElement_ACRole.Get().
          Single());
}

public IEnumerable<string> GetSelectedTACoursesTitles()
{
  return GetSelectedStudent().
    __AssociationSharp_HideElement_TARole.Get().Single().
      GetACsTitles();
}

public IEnumerable<int> GetSelectedACAssistantsNumbers()
{
  return GetSelectedCourse().
    __AssociationSharp_HideElement_ACRole.Get().Single().
      GetTAsNumbers();
}

// Vínculo de Aluno com Curso (Attends)
public void AddSelectedLearnerToSelectedLecture()
{
  GetSelectedStudent().AddLecture(GetSelectedCourse());
}

public void RemoveSelectedLearnerFromSelectedLecture()
{
  try
  {
    AssociationSharpSystem.__DisableSystemConstraints();

    GetSelectedStudent().RemoveLecture(
      GetSelectedCourse());
    if (!GetSelectedStudent().IsAttendingCourse())
    {
      this.DeleteSelectedStudent();
    }
  }
  finally
  {
    AssociationSharpSystem.__EnableSystemConstraints();
  }
}

public void SetMarkForSelectedLearnerOnSelectedLecture(
  int mark)
{
  GetSelectedStudent().SetMarkOnLecture(
    GetSelectedCourse(), mark);
}
```

```
public int GetMarkForSelectedLearnerOnSelectedLecture()
{
  return GetSelectedStudent().GetMarkOnLecture(
    GetSelectedCourse());
}

public IEnumerable<string>
  GetLecturesTitlesOfSelectedLearner()
{
  return GetSelectedStudent().GetCoursesTitles();
}

public IEnumerable<int>
  GetLearnersNumbersOfSelectedLecture()
{
  return GetSelectedCourse().GetLearnersNumbers();
}

// Estudante com papel de Pesquisador (RA_Role)
public void SelectedStudentIsAResearchAssistant(
  float grantAmount)
{
  try
  {
    AssociationSharpSystem.__DisableSystemConstraints();

    GetSelectedStudent().
      __AssociationSharp_HideElement_RARole.Add(
        new RA_Role(grantAmount));
  }
  finally
  {
    AssociationSharpSystem.__EnableSystemConstraints();
  }
}

public void SelectedStudentIsNotAResearchAssistant()
{
  GetSelectedStudent().
    __AssociationSharp_HideElement_RARole.RemoveAll();
}

public void SetGrantAmountForSelectedResearchAssistant(
  float newGrantAmount)
{
  GetSelectedStudent().
    __AssociationSharp_HideElement_RARole.Get().Single().
      GrantAmount = newGrantAmount;
}

public float GetGrantAmountForSelectedResearchAssistant()
{
  return GetSelectedStudent().
```

```
        __AssociationSharp_HideElement_RARole.Get().Single().
          GrantAmount;
    }

    // Vínculo de Pesquisador com Supervisor.
    public void
      AddSelectedResearchAssistantToSelectedSupervisor()
    {
      GetSelectedStudent().
        __AssociationSharp_HideElement_RARole.Get().Single().
          AddSupervisor(GetSelectedFaculty());
    }

    public void
      RemoveSelectedResearchAssistantFromSelectedSupervisor()
    {
      GetSelectedStudent().
        __AssociationSharp_HideElement_RARole.Get().Single().
          RemoveSupervisor(GetSelectedFaculty());
    }

    public IEnumerable<int>
      GetSupervisorsNumbersFromSelectedResearchAssistant()
    {
      return GetSelectedStudent().
        __AssociationSharp_HideElement_RARole.Get().Single().
          GetSupervisorsNumbers();
    }

    public IEnumerable<int>
      GetResearchAssistantsNumbersFromSelectedSupervisor()
    {
      return from ra in GetSelectedFaculty().RAs.Get()
             select ra.GetStudentNumber();
    }
  }

  //
  // Associações e Classes de Associação persistentes.
  //
  public class UniversityToStudent
    : Association<University, Student>
  {
    private static UniversityToStudent SINGLETON =
      new UniversityToStudent();

    private UniversityToStudent()
      : base(
          Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ONE_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
```

```
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByStudent : TargetAssociationEnd
    {
      public EndOwnedByStudent(Student target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.Students;
    }

    private static EndOwnedByStudent targetEndFinder(
      Student target)
    {
      return target.
        __AssociationSharp_HideElement_UniversityToStudent;
    }
  }

  public class UniversityToFaculty
    : Association<University, Faculty>
  {
    private static UniversityToFaculty SINGLETON =
      new UniversityToFaculty();

    private UniversityToFaculty()
      : base(
          Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ONE_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByFaculty : TargetAssociationEnd
    {
      public EndOwnedByFaculty(Faculty target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.Faculties;
```

```
    }

    private static EndOwnedByFaculty targetEndFinder(
      Faculty target)
    {
      return target.
        __AssociationSharp_HideElement_UniversityToFaculty;
    }
  }

  public class UniversityToLanguage
    : Association<University, InstructionLanguage>
  {
    private static UniversityToLanguage SINGLETON =
      new UniversityToLanguage();

    private UniversityToLanguage()
      : base(
          Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ONE_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByLanguage : TargetAssociationEnd
    {
      public EndOwnedByLanguage(InstructionLanguage target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.Languages;
    }

    private static EndOwnedByLanguage
      targetEndFinder(InstructionLanguage target)
    {
      return target.
        __AssociationSharp_HideElement_UniversityToLanguage;
    }
  }

  public class UniversityToCourse
    : Association<University, Course>
  {
    private static UniversityToCourse SINGLETON =
      new UniversityToCourse();
```

```
    private UniversityToCourse()
      : base(
          Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ONE_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByCourses : TargetAssociationEnd
    {
      public EndOwnedByCourses(Course target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.Courses;
    }

    private static EndOwnedByCourses targetEndFinder(
      Course target)
    {
      return target.
        __AssociationSharp_HideElement_UniversityToCourse;
    }
  }

  public class Teaches : Association<Faculty, Course>
  {
    private static Teaches SINGLETON = new Teaches();

    private Teaches()
      : base(
          Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ONE_TO_MANY_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByFaculty : SourceAssociationEnd
    {
      public EndOwnedByFaculty(Faculty source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByCourse : TargetAssociationEnd
    {
      public EndOwnedByCourse(Course target)
```

```
      : base(target, SINGLETON) { }
    }

    private static EndOwnedByFaculty sourceEndFinder(
      Faculty source)
    {
      return source.Lectures;
    }

    private static EndOwnedByCourse targetEndFinder(
      Course target)
    {
      return target.Lecturers;
    }
  }

  public class AssistantToLanguage
    : Association<TA_Role, InstructionLanguage>
  {
    private static AssistantToLanguage SINGLETON =
      new AssistantToLanguage();

    private AssistantToLanguage()
      : base(
          Multiplicity.ONE_TO_ONE_RANGE,
          Multiplicity.ZERO_TO_MANY_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByTA : SourceAssociationEnd
    {
      public EndOwnedByTA(TA_Role source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByInstructionLanguage
      : TargetAssociationEnd
    {
      public EndOwnedByInstructionLanguage(
        InstructionLanguage target)
          : base(target, SINGLETON) { }
    }

    private static EndOwnedByTA sourceEndFinder(TA_Role source)
    {
      return source.Language;
    }

    private static EndOwnedByInstructionLanguage
      targetEndFinder(InstructionLanguage target)
    {
      return target.TAs;
    }
  }
```

```
public class Assists : Association<TA_Role, AC_Role>
{
  private static Assists SINGLETON = new Assists();

  private Assists()
    : base(
        Multiplicity.ZERO_TO_MANY_RANGE,
        Multiplicity.ZERO_TO_MANY_RANGE,
        sourceEndFinder,
        targetEndFinder) { }

  public class EndOwnedByTA : SourceAssociationEnd
  {
    public EndOwnedByTA(TA_Role source)
      : base(source, SINGLETON) { }
  }

  public class EndOwnedByAC
    : TargetAssociationEnd
  {
    public EndOwnedByAC(AC_Role target)
      : base(target, SINGLETON) { }
  }

  private static EndOwnedByTA sourceEndFinder(TA_Role source)
  {
    return source.ACs;
  }

  private static EndOwnedByAC targetEndFinder(AC_Role target)
  {
    return target.TAs;
  }
}

public class StudentToTA : Association<Student, TA_Role>
{
  private static StudentToTA SINGLETON = new StudentToTA();

  private StudentToTA()
    : base(
        Multiplicity.ZERO_TO_ONE_RANGE,
        Multiplicity.ONE_TO_ONE_RANGE,
        sourceEndFinder,
        targetEndFinder) { }

  public class EndOwnedByStudent : SourceAssociationEnd
  {
    public EndOwnedByStudent(Student source)
      : base(source, SINGLETON) { }
  }

  public class EndOwnedByTA : TargetAssociationEnd
```

```
    {
      public EndOwnedByTA(TA_Role target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByStudent sourceEndFinder(
      Student source)
    {
      return source.__AssociationSharp_HideElement_TARole;
    }

    private static EndOwnedByTA targetEndFinder(TA_Role target)
    {
      return target.Player;
    }
  }

  public class CourseToAC : Association<Course, AC_Role>
  {
    private static CourseToAC SINGLETON = new CourseToAC();

    private CourseToAC()
      : base(
          Multiplicity.ONE_TO_ONE_RANGE,
          Multiplicity.ONE_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByCourse : SourceAssociationEnd
    {
      public EndOwnedByCourse(Course source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByAC : TargetAssociationEnd
    {
      public EndOwnedByAC(AC_Role target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByCourse sourceEndFinder(
      Course source)
    {
      return source.__AssociationSharp_HideElement_ACRole;
    }

    private static EndOwnedByAC targetEndFinder(AC_Role target)
    {
      return target.Player;
    }
  }

  public class Attends
  {
```

```
    public int Mark { set; get; }

    public class Association
      : AssociationClass<Student, Course, Attends>
    {
      private static readonly Association SINGLETON =
        new Association();

      private Association()
        : base(
          Multiplicity.ONE_TO_MANY_RANGE,
          Multiplicity.ZERO_TO_MANY_RANGE,
          getSourceEnd,
          getTargetEnd) { }

      private static EndForByStudent
        getSourceEnd(Student source)
      {
        return source.Lectures;
      }
      private static EndOwnedByCourse
        getTargetEnd(Course target)
      {
        return target.Learners;
      }

      public class EndForByStudent
        : SourceAssociationEnd
      {
        public EndForByStudent(Student source)
          : base(source, SINGLETON) { }
      }

      public class EndOwnedByCourse : TargetAssociationEnd
      {
        public EndOwnedByCourse(Course target)
          : base(target, SINGLETON) { }
      }
    }
  }

  public class Substitutes : Association<Faculty, Faculty>
  {
    private static Substitutes SINGLETON = new Substitutes();

    private Substitutes()
      : base(
          Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ZERO_TO_MANY_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedBySubstitutes : SourceAssociationEnd
    {
```

```
      public EndOwnedBySubstitutes(Faculty source)
         : base(source, SINGLETON) { }
   }

   public class EndOwnedBySubstituted : TargetAssociationEnd
   {
      public EndOwnedBySubstituted(Faculty target)
         : base(target, SINGLETON) { }
   }

   private static EndOwnedBySubstitutes
      sourceEndFinder(Faculty source)
   {
      return source.Substituted;
   }

   private static EndOwnedBySubstituted
      targetEndFinder(Faculty target)
   {
      return target.Substitutes;
   }
}


public class StudentToRA : Association<Student, RA_Role>
{
   private static StudentToRA SINGLETON = new StudentToRA();

   private StudentToRA()
      : base(
           Multiplicity.ZERO_TO_ONE_RANGE,
           Multiplicity.ONE_TO_ONE_RANGE,
           sourceEndFinder,
           targetEndFinder) { }

   public class EndOwnedByStudent : SourceAssociationEnd
   {
      public EndOwnedByStudent(Student source)
         : base(source, SINGLETON) { }
   }

   public class EndOwnedByRA : TargetAssociationEnd
   {
      public EndOwnedByRA(RA_Role target)
         : base(target, SINGLETON) { }
   }

   private static EndOwnedByStudent sourceEndFinder(
      Student source)
   {
      return source.__AssociationSharp_HideElement_RARole;
   }

   private static EndOwnedByRA targetEndFinder(RA_Role target)
```

```
    {
      return target.Player;
    }
  }

  public class WorksFor : Association<RA_Role, Faculty>
  {
    private static WorksFor SINGLETON = new WorksFor();

    private WorksFor()
      : base(
          Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ZERO_TO_MANY_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByRA : SourceAssociationEnd
    {
      public EndOwnedByRA(RA_Role source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByFaculty : TargetAssociationEnd
    {
      public EndOwnedByFaculty(Faculty target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByRA sourceEndFinder(RA_Role source)
    {
      return source.Supervisors;
    }

    private static EndOwnedByFaculty targetEndFinder(
      Faculty target)
    {
      return target.RAs;
    }
  }

  //
  // Associações e Classes de Associação temporárias
  //
  public class UniversityToSelectedStudent
    : Association<University, Student>
  {
    private static UniversityToSelectedStudent SINGLETON =
      new UniversityToSelectedStudent();

    private UniversityToSelectedStudent()
      : base(
          Multiplicity.ZERO_TO_ONE_RANGE,
          Multiplicity.ZERO_TO_ONE_RANGE,
          sourceEndFinder,
```

```
            targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByStudent : TargetAssociationEnd
    {
      public EndOwnedByStudent(Student target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.SelectedStudent;
    }

    private static EndOwnedByStudent targetEndFinder(
      Student target)
    {
      return target.
__AssociationSharp_HideElement_UniversityToSelectedStudent;
    }
  }

  public class UniversityToSelectedFaculty
    : Association<University, Faculty>
  {
    private static UniversityToSelectedFaculty SINGLETON =
      new UniversityToSelectedFaculty();

    private UniversityToSelectedFaculty()
      : base(
          Multiplicity.ZERO_TO_ONE_RANGE,
          Multiplicity.ZERO_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByFaculty : TargetAssociationEnd
    {
      public EndOwnedByFaculty(Faculty target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
```

```
      sourceEndFinder(University source)
    {
      return source.SelectedFaculty;
    }

    private static EndOwnedByFaculty
      targetEndFinder(Faculty target)
    {
      return target.
__AssociationSharp_HideElement_UniversityToSelectedFaculty;
    }
  }

  public class UniversityToSelectedLanguage
    : Association<University, InstructionLanguage>
  {
    private static UniversityToSelectedLanguage SINGLETON =
      new UniversityToSelectedLanguage();

    private UniversityToSelectedLanguage()
      : base(
          Multiplicity.ZERO_TO_ONE_RANGE,
          Multiplicity.ZERO_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByLanguage : TargetAssociationEnd
    {
      public EndOwnedByLanguage(InstructionLanguage target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.SelectedLanguage;
    }

    private static EndOwnedByLanguage
      targetEndFinder(InstructionLanguage target)
    {
      return target.
__AssociationSharp_HideElement_UniversityToSelectedLanguage;
    }
  }

  public class UniversityToSelectedCourse
    : Association<University, Course>
```

```
  {
    private static UniversityToSelectedCourse SINGLETON =
      new UniversityToSelectedCourse();

    private UniversityToSelectedCourse()
      : base(
          Multiplicity.ZERO_TO_ONE_RANGE,
          Multiplicity.ZERO_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedByCourse : TargetAssociationEnd
    {
      public EndOwnedByCourse(Course target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.SelectedCourse;
    }

    private static EndOwnedByCourse targetEndFinder(
      Course target)
    {
      return target.
__AssociationSharp_HideElement_UniversityToSelectedCourse;
    }
  }

  public class UniversityToSelectedSubstitute
    : Association<University, Faculty>
  {
    private static UniversityToSelectedSubstitute SINGLETON =
      new UniversityToSelectedSubstitute();

    private UniversityToSelectedSubstitute()
      : base(
          Multiplicity.ZERO_TO_ONE_RANGE,
          Multiplicity.ZERO_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
```

```
    }

    public class EndOwnedBySelectedSubstitute
      : TargetAssociationEnd
    {
      public EndOwnedBySelectedSubstitute(Faculty target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.SelectedSubstitute;
    }

    private static EndOwnedBySelectedSubstitute
      targetEndFinder(Faculty target)
    {
      return target.
__AssociationSharp_HideElement_UniversityToSelectedSubstitute;
    }
  }

  public class UniversityToSelectedLecturers
    : Association<University, Faculty>
  {
    private static UniversityToSelectedLecturers SINGLETON =
      new UniversityToSelectedLecturers();

    private UniversityToSelectedLecturers()
      : base(
          Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ZERO_TO_ONE_RANGE,
          sourceEndFinder,
          targetEndFinder) { }

    public class EndOwnedByUniversity : SourceAssociationEnd
    {
      public EndOwnedByUniversity(University source)
        : base(source, SINGLETON) { }
    }

    public class EndOwnedBySelectedLecturers
      : TargetAssociationEnd
    {
      public EndOwnedBySelectedLecturers(Faculty target)
        : base(target, SINGLETON) { }
    }

    private static EndOwnedByUniversity
      sourceEndFinder(University source)
    {
      return source.SelectedLecturers;
    }
```

```
    private static EndOwnedBySelectedLecturers
      targetEndFinder(Faculty target)
    {
      return target.
__AssociationSharp_HideElement_UniversityToSelectedLecturers;
    }
  }
}
```

As listagens a seguir apresentam o programa que executa o teste e o teste de unidade construído para ser utilizado pelo testador de unidades presentes na biblioteca AssociationSharp, no pacote *AssociationSharp.Commons*. No teste de unidade, são executados somente os métodos públicos e de instância.

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: University - A example of using the Association#
 *          language
 *
 * Source File: /Program.cs
 *
 * Description: Application for run a unit test for the domain
 *              layer of university system.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using AssociationSharp.Commons;

namespace UniversitySystem
{
  public class Program
  {
    public static void Main(string[] args)
    {
      UnitTest.test(new Test());
    }
```

```
   }
}
```

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: University - A example of using the Association#
 *          language
 *
 * Source File: /Test.cs
 *
 * Description: Unit Test for the domain layer of university
 *              system
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System.Collections.Generic;
using System.Linq;
using UniversitySystem.DomainLayer;
using AssociationSharp.Commons.Assertion;

namespace UniversitySystem
{
   /*
    * Unit Test
    * Somente os métodos públicos e de instância são executados
pelo teste automatizado.
    */
   class Test
   {
     #region métodos a serem executados automaticamente.

     public void CreateFaculties()
     {
       this.CreateFaculty(FACULTY_JOHN_NAME,
         FACULTY_JOHN_NUMBER);
       this.CreateFaculty(FACULTY_PAUL_NAME,
         FACULTY_PAUL_NUMBER);
       this.CreateFaculty(FACULTY_RAUL_NAME,
         FACULTY_RAUL_NUMBER);
```

```
      this.CreateFaculty(FACULTY_MARY_NAME,
        FACULTY_MARY_NUMBER);
    }

    public void VerififyFaculties()
    {
      this.SelectFaculty(FACULTY_JOHN_NAME);

Assert.IsTrue(University.INSTANCE.GetSelectedFacultyNumber()
    == FACULTY_JOHN_NUMBER);

      this.SelectFaculty(FACULTY_PAUL_NAME);

Assert.IsTrue(University.INSTANCE.GetSelectedFacultyNumber()
    == FACULTY_PAUL_NUMBER);

      this.SelectFaculty(FACULTY_RAUL_NAME);

Assert.IsTrue(University.INSTANCE.GetSelectedFacultyNumber()
    == FACULTY_RAUL_NUMBER);

      this.SelectFaculty(FACULTY_MARY_NAME);

Assert.IsTrue(University.INSTANCE.GetSelectedFacultyNumber()
    == FACULTY_MARY_NUMBER);
    }

    public void AddFacultiesSubstitutes()
    {
      this.AddSubstituteToFaculty(FACULTY_PAUL_NAME,
        FACULTY_JOHN_NAME);
      this.AddSubstituteToFaculty(FACULTY_PAUL_NAME,
        FACULTY_RAUL_NAME);
      this.AddSubstituteToFaculty(FACULTY_JOHN_NAME,
        FACULTY_MARY_NAME);
      this.AddSubstituteToFaculty(FACULTY_MARY_NAME,
        FACULTY_PAUL_NAME);
      this.AddSubstituteToFaculty(FACULTY_MARY_NAME,
        FACULTY_RAUL_NAME);
    }

    public void VerifyFacultiesSubstitutes()
    {
      this.VerifyFacultySubstitutes(FACULTY_PAUL_NAME, new
        string[] { FACULTY_JOHN_NAME, FACULTY_RAUL_NAME });
      this.VerifyFacultySubstitutes(FACULTY_JOHN_NAME, new
        string[] { FACULTY_MARY_NAME });
      this.VerifyFacultySubstitutes(FACULTY_MARY_NAME, new
        string[] { FACULTY_PAUL_NAME, FACULTY_RAUL_NAME });
    }

    public void CreateCourses()
    {
      this.CreateCourse(COURSE_COMPUTATION, new string[] {
```

```
        FACULTY_RAUL_NAME,
        FACULTY_PAUL_NAME,
        FACULTY_MARY_NAME });
      this.ResetMaxGroupSizeForCourse(COURSE_COMPUTATION,
        GROUPSIZE_COMPUTATION);
      this.CreateCourse(COURSE_PHYSICS, new string[] {
        FACULTY_JOHN_NAME, FACULTY_PAUL_NAME,
        FACULTY_MARY_NAME });
      this.ResetMaxGroupSizeForCourse(COURSE_PHYSICS,
        GROUPSIZE_PHYSICS);
      this.CreateCourse(COURSE_MATH, new string[] {
        FACULTY_JOHN_NAME, FACULTY_PAUL_NAME,
        FACULTY_RAUL_NAME });
      this.ResetMaxGroupSizeForCourse(COURSE_MATH,
        GROUPSIZE_MATH);
    }

    public void VerifyCourses()
    {
      this.VerifyCourse(COURSE_COMPUTATION, new string[] {
        FACULTY_RAUL_NAME, FACULTY_PAUL_NAME,
        FACULTY_MARY_NAME });
      this.VerifyMaxGroupSizeForCourse(COURSE_COMPUTATION,
        GROUPSIZE_COMPUTATION);
      this.VerifyCourse(COURSE_PHYSICS, new string[] {
        FACULTY_JOHN_NAME, FACULTY_PAUL_NAME,
        FACULTY_MARY_NAME });
      this.VerifyMaxGroupSizeForCourse(COURSE_PHYSICS,
        GROUPSIZE_PHYSICS);
      this.VerifyCourse(COURSE_MATH, new string[] {
        FACULTY_JOHN_NAME, FACULTY_PAUL_NAME,
        FACULTY_RAUL_NAME });
      this.VerifyMaxGroupSizeForCourse(COURSE_MATH,
        GROUPSIZE_MATH);
    }

    public void CreateStudents()
    {
      this.CreateStudent(COURSE_COMPUTATION,
        STUDENT_IURI_NAME, STUDENT_IURI_NUMBER);
      this.SetInstructionLanguageForTeachingAssistant(
        STUDENT_IURI_NAME, INSTRUCTION_LANGUAGE_PORTUGUESE);
      this.SetStudentToBeAssistantOfCourse(STUDENT_IURI_NAME,
        COURSE_MATH);
      this.SetStudentAsResearchAssistant(STUDENT_IURI_NAME,
        STUDENT_IURI_GRANTAMOUNT);

      this.CreateStudent(COURSE_COMPUTATION,
        STUDENT_GEORGE_NAME, STUDENT_GEORGE_NUMBER);
      this.SetInstructionLanguageForTeachingAssistant(
        STUDENT_GEORGE_NAME, INSTRUCTION_LANGUAGE_ENGLISH);
      this.SetStudentToBeAssistantOfCourse(
        STUDENT_GEORGE_NAME, COURSE_MATH);
      this.SetStudentToBeAssistantOfCourse(
```

```
            STUDENT_GEORGE_NAME, COURSE_PHYSICS);
         this.SetStudentAsResearchAssistant(STUDENT_GEORGE_NAME,
            STUDENT_GEORGE_GRANTAMOUNT);

         this.CreateStudent(COURSE_MATH, STUDENT_JOE_NAME,
            STUDENT_JOE_NUMBER);
         this.SetInstructionLanguageForTeachingAssistant(
            STUDENT_JOE_NAME, INSTRUCTION_LANGUAGE_GERMAN);
         this.SetStudentToBeAssistantOfCourse(STUDENT_JOE_NAME,
            COURSE_COMPUTATION);
         this.SetStudentAsResearchAssistant(STUDENT_JOE_NAME,
            STUDENT_JOE_GRANTAMOUNT);

         this.CreateStudent(COURSE_PHYSICS, STUDENT_MARTIN_NAME,
            STUDENT_MARTIN_NUMBER);
         this.SetInstructionLanguageForTeachingAssistant(
            STUDENT_MARTIN_NAME, INSTRUCTION_LANGUAGE_ENGLISH);
         this.SetStudentToBeAssistantOfCourse(
            STUDENT_MARTIN_NAME, COURSE_COMPUTATION);
         this.SetStudentAsResearchAssistant(STUDENT_MARTIN_NAME,
            STUDENT_MARTIN_GRANTAMOUNT);
      }

      public void VerifyStudents()
      {
         this.VerifyStudent(STUDENT_IURI_NAME,
            STUDENT_IURI_NUMBER);
         this.VerifyInstructionLanguageForTeachingAssistant(
            STUDENT_IURI_NAME, INSTRUCTION_LANGUAGE_PORTUGUESE);
         this.VerifyIfStudentIsAssistantOfCourses(
            STUDENT_IURI_NAME, new string[] { COURSE_MATH });
         this.VerifyResearchAssistantGrantAmout(
            STUDENT_IURI_NAME, STUDENT_IURI_GRANTAMOUNT);

         this.VerifyStudent(STUDENT_GEORGE_NAME,
            STUDENT_GEORGE_NUMBER);
         this.VerifyInstructionLanguageForTeachingAssistant(
            STUDENT_GEORGE_NAME, INSTRUCTION_LANGUAGE_ENGLISH);
         this.VerifyIfStudentIsAssistantOfCourses(
            STUDENT_GEORGE_NAME, new string[] {
               COURSE_MATH, COURSE_PHYSICS });
         this.VerifyResearchAssistantGrantAmout(
            STUDENT_GEORGE_NAME, STUDENT_GEORGE_GRANTAMOUNT);

         this.VerifyStudent(STUDENT_JOE_NAME,
            STUDENT_JOE_NUMBER);
         this.VerifyInstructionLanguageForTeachingAssistant(
            STUDENT_JOE_NAME, INSTRUCTION_LANGUAGE_GERMAN);
         this.VerifyIfStudentIsAssistantOfCourses(
            STUDENT_JOE_NAME, new string[] {
               COURSE_COMPUTATION });
         this.VerifyResearchAssistantGrantAmout(STUDENT_JOE_NAME,
            STUDENT_JOE_GRANTAMOUNT);
```

```
    this.VerifyStudent(STUDENT_MARTIN_NAME,
      STUDENT_MARTIN_NUMBER);
    this.VerifyInstructionLanguageForTeachingAssistant(
      STUDENT_MARTIN_NAME, INSTRUCTION_LANGUAGE_ENGLISH);
    this.VerifyIfStudentIsAssistantOfCourses(
      STUDENT_MARTIN_NAME, new string[] {
        COURSE_COMPUTATION });
    this.VerifyResearchAssistantGrantAmout(
      STUDENT_MARTIN_NAME, STUDENT_MARTIN_GRANTAMOUNT);
  }

  public void StudentsAttendsOthersCourses()
  {
    this.StudentAttendsOtherCourse(STUDENT_IURI_NAME,
      COURSE_PHYSICS);
    this.StudentAttendsOtherCourse(STUDENT_JOE_NAME,
      COURSE_PHYSICS);
  }

  public void VerifyAttends()
  {
    //computation
    this.VerifyLearnersOfLecture(COURSE_COMPUTATION,
      new string[] { STUDENT_IURI_NAME, STUDENT_GEORGE_NAME
      });

    //math
    this.VerifyLearnersOfLecture(COURSE_MATH, new string[] {
      STUDENT_JOE_NAME });

    //physics
    this.VerifyLearnersOfLecture(COURSE_PHYSICS,
      new string[] { STUDENT_MARTIN_NAME, STUDENT_IURI_NAME,
        STUDENT_JOE_NAME });
  }

  public void SetAttendsMarks()
  {
    this.SetAttendsMark(STUDENT_IURI_NAME,
      COURSE_COMPUTATION, MARK_IURI_COMPUTATION);
    this.SetAttendsMark(STUDENT_IURI_NAME, COURSE_PHYSICS,
      MARK_IURI_PHYSICS);

    this.SetAttendsMark(STUDENT_GEORGE_NAME,
      COURSE_COMPUTATION, MARK_GEORGE_COMPUTATION);

    this.SetAttendsMark(STUDENT_JOE_NAME, COURSE_MATH,
      MARK_JOE_MATH);
    this.SetAttendsMark(STUDENT_JOE_NAME, COURSE_PHYSICS,
      MARK_JOE_PHYSICS);

    this.SetAttendsMark(STUDENT_MARTIN_NAME, COURSE_PHYSICS,
      MARK_MARTIN_PHYSICS);
  }
```

```
public void VerifyAttendsMarks()
{
  Assert.IsTrue(this.GetAttendsMark(STUDENT_IURI_NAME,
    COURSE_COMPUTATION) == MARK_IURI_COMPUTATION);
  Assert.IsTrue(this.GetAttendsMark(STUDENT_IURI_NAME,
    COURSE_PHYSICS) == MARK_IURI_PHYSICS);

  Assert.IsTrue(this.GetAttendsMark(STUDENT_GEORGE_NAME,
    COURSE_COMPUTATION) == MARK_GEORGE_COMPUTATION);

  Assert.IsTrue(this.GetAttendsMark(STUDENT_JOE_NAME,
    COURSE_MATH) == MARK_JOE_MATH);
  Assert.IsTrue(this.GetAttendsMark(STUDENT_JOE_NAME,
    COURSE_PHYSICS) == MARK_JOE_PHYSICS);

  Assert.IsTrue(this.GetAttendsMark(STUDENT_MARTIN_NAME,
    COURSE_PHYSICS) == MARK_MARTIN_PHYSICS);
}

public void StudentsAreResearchAssistants()
{
  this.StudentIsResearchAssistant(STUDENT_IURI_NAME,
    FACULTY_RAUL_NAME);
  this.StudentIsResearchAssistant(STUDENT_IURI_NAME,
    FACULTY_MARY_NAME);
  this.StudentIsResearchAssistant(STUDENT_GEORGE_NAME,
    FACULTY_MARY_NAME);
  this.StudentIsResearchAssistant(STUDENT_JOE_NAME,
    FACULTY_JOHN_NAME);
  this.StudentIsResearchAssistant(STUDENT_MARTIN_NAME,
    FACULTY_PAUL_NAME);
}

public void VerifyRAsWorksForSupervisors()
{
  this.VerifyIfStudentIsResearchAssistant(
    STUDENT_IURI_NAME, new string[] {
      FACULTY_RAUL_NAME, FACULTY_MARY_NAME });
  this.VerifyIfStudentIsResearchAssistant(
    STUDENT_GEORGE_NAME, new string[] {
      FACULTY_MARY_NAME });
  this.VerifyIfStudentIsResearchAssistant(
    STUDENT_JOE_NAME, new string[] { FACULTY_JOHN_NAME });
  this.VerifyIfStudentIsResearchAssistant(
    STUDENT_MARTIN_NAME, new string[] {
      FACULTY_PAUL_NAME });
}

public void StudentsNotWorksForFaculties()
{
  this.StudentNotWorksForFaculties(STUDENT_IURI_NAME,
    FACULTY_RAUL_NAME);
  this.StudentNotWorksForFaculties(STUDENT_IURI_NAME,
```

```
        FACULTY_MARY_NAME);
      this.StudentNotWorksForFaculties(STUDENT_GEORGE_NAME,
        FACULTY_MARY_NAME);
      this.StudentNotWorksForFaculties(STUDENT_JOE_NAME,
        FACULTY_JOHN_NAME);
      this.StudentNotWorksForFaculties(STUDENT_MARTIN_NAME,
        FACULTY_PAUL_NAME);
    }

    public void StudentsAreNotResearchAssistants()
    {
      this.StudentIsNotResearchAssistant(STUDENT_IURI_NAME);
      this.StudentIsNotResearchAssistant(STUDENT_GEORGE_NAME);
      this.StudentIsNotResearchAssistant(STUDENT_JOE_NAME);
      this.StudentIsNotResearchAssistant(STUDENT_MARTIN_NAME);
    }

    public void StudentsNotAssistsCourses()
    {
      this.StudentNotAssistsCourse(STUDENT_IURI_NAME,
        COURSE_MATH);
      this.StudentNotAssistsCourse(STUDENT_GEORGE_NAME,
        COURSE_MATH);
      this.StudentNotAssistsCourse(STUDENT_GEORGE_NAME,
        COURSE_PHYSICS);
      this.StudentNotAssistsCourse(STUDENT_JOE_NAME,
        COURSE_COMPUTATION);
      this.StudentNotAssistsCourse(STUDENT_MARTIN_NAME,
        COURSE_COMPUTATION);
    }

    public void StudentsAreNotTeachingAssistants()
    {
      this.StudentIsNotTeachingAssistant(STUDENT_IURI_NAME);
      this.StudentIsNotTeachingAssistant(STUDENT_GEORGE_NAME);
      this.StudentIsNotTeachingAssistant(STUDENT_JOE_NAME);
      this.StudentIsNotTeachingAssistant(STUDENT_MARTIN_NAME);
    }

    public void RemoveStudents()
    {
      //computation
      this.RemoveLearnersOfLecture(COURSE_COMPUTATION,
        new string[] {
          STUDENT_IURI_NAME, STUDENT_GEORGE_NAME });

      //math
      this.RemoveLearnersOfLecture(COURSE_MATH, new string[] {
        STUDENT_JOE_NAME });

      //physics
      this.RemoveLearnersOfLecture(COURSE_PHYSICS,
        new string[] { STUDENT_MARTIN_NAME, STUDENT_IURI_NAME,
          STUDENT_JOE_NAME });
```

```
    }

    public void RemoveCourses()
    {
      this.RemoveCourse(COURSE_COMPUTATION);
      this.RemoveCourse(COURSE_MATH);
      this.RemoveCourse(COURSE_PHYSICS);
    }

    public void RemoveFaculties()
    {
      this.RemoveFaculty(FACULTY_JOHN_NAME);
      this.RemoveFaculty(FACULTY_MARY_NAME);
      this.RemoveFaculty(FACULTY_PAUL_NAME);
      this.RemoveFaculty(FACULTY_RAUL_NAME);
    }

    #endregion métodos a serem executados automaticamente.

    #region Constantes
    private const string FACULTY_JOHN_NAME = "John";
    private const int FACULTY_JOHN_NUMBER = 201101001;
    private const string FACULTY_PAUL_NAME = "Paul";
    private const int FACULTY_PAUL_NUMBER = 201101002;
    private const string FACULTY_RAUL_NAME = "Raul";
    private const int FACULTY_RAUL_NUMBER = 201101003;
    private const string FACULTY_MARY_NAME = "Mary";
    private const int FACULTY_MARY_NUMBER = 201101004;

    private const string COURSE_COMPUTATION = "Computation";
    private const string COURSE_MATH = "Math";
    private const string COURSE_PHYSICS = "Physics";

    private const string STUDENT_IURI_NAME = "Iuri";
    private const int STUDENT_IURI_NUMBER = 2011010001;
    private const float STUDENT_IURI_GRANTAMOUNT = 1000;
    private const string STUDENT_JOE_NAME = "Joe";
    private const int STUDENT_JOE_NUMBER = 2011010002;
    private const float STUDENT_JOE_GRANTAMOUNT = 1000;
    private const string STUDENT_GEORGE_NAME = "George";
    private const int STUDENT_GEORGE_NUMBER = 2011010003;
    private const float STUDENT_GEORGE_GRANTAMOUNT = 1000;
    private const string STUDENT_MARTIN_NAME = "Martin";
    private const int STUDENT_MARTIN_NUMBER = 2011010004;
    private const float STUDENT_MARTIN_GRANTAMOUNT = 1000;

    private const string INSTRUCTION_LANGUAGE_PORTUGUESE =
      "Portuguese";
    private const string INSTRUCTION_LANGUAGE_ENGLISH =
      "English";
    private const string INSTRUCTION_LANGUAGE_GERMAN =
      "German";

    private const int MARK_IURI_COMPUTATION = 8;
```

```
    private const int MARK_IURI_PHYSICS = 7;
    private const int MARK_GEORGE_COMPUTATION = 9;
    private const int MARK_JOE_MATH = 8;
    private const int MARK_JOE_PHYSICS = 6;
    private const int MARK_MARTIN_PHYSICS = 7;

    private const int GROUPSIZE_COMPUTATION = 15;
    private const int GROUPSIZE_MATH = 25;
    private const int GROUPSIZE_PHYSICS = 20;

    #endregion Constantes

    #region métodos auxiliares

    private void CreateFaculty(string name, int number)
    {
      University.INSTANCE.RegisterFaculty(name, number);
    }

    private void SelectFaculty(string facultyName)
    {
      IEnumerable<int> facultiesNumbers =
        University.INSTANCE.GetFacultiesNumbers();

      bool find = false;
      foreach (int facultyNumber in facultiesNumbers)
      {
        University.INSTANCE.SelectFaculty(facultyNumber);
        if (University.INSTANCE.GetSelectedFacultyName() ==
          facultyName)
        {
          University.INSTANCE.SelectFaculty(facultyNumber);
          find = true;
          break;
        }
      }

      Assert.IsTrue(find);
    }

    private void AddSubstituteToFaculty(string substituteName,
      string facultyName)
    {
      this.SelectFaculty(facultyName);

      IEnumerable<int> facultiesNumbers =
        University.INSTANCE.GetFacultiesNumbers();

      bool find = false;
      foreach (int facultyNumber in facultiesNumbers)
      {
        University.INSTANCE.SelectSubstitute(facultyNumber);
        if (University.INSTANCE.GetSelectedSubstituteName() ==
          substituteName)
```

```
        {
          find = true;
          break;
        }
      }

      Assert.IsTrue(find);

      University.INSTANCE.
        AddSelectedSubstituteToSelectedFaculty();
    }

    private void VerifyFacultySubstitutes(string facultyName,
      IEnumerable<string> substitutesNames)
    {
      this.SelectFaculty(facultyName);
      HashSet<int> substitutesNumbers =
        new HashSet<int>(University.INSTANCE.
          GetSubstitutesNumbersFromSelectedFaculty());

      foreach (int number in substitutesNumbers)
      {
        University.INSTANCE.SelectSubstitute(number);
        if (substitutesNames.Contains(
          University.INSTANCE.GetSelectedSubstituteName()))
        {
          Assert.IsTrue(substitutesNumbers.Remove(number));
        }
      }

      Assert.IsTrue(substitutesNumbers.Count == 0);
    }

    private void CreateCourse(string courseTitle,
      IEnumerable<string> lecturersNames)
    {
      foreach (string lectureName in lecturersNames)
      {
        this.SelectFaculty(lectureName);
        University.INSTANCE.AddLecturerForNewCourse(
          University.INSTANCE.GetSelectedFacultyNumber());
      }

      University.INSTANCE.RegisterCourse(courseTitle, 0);
    }

    private void SelectCourse(string courseTitle)
    {
      bool find = false;
      foreach (string title in
        University.INSTANCE.GetCoursesTitles())
      {
        if (title == courseTitle)
        {
```

```
            find = true;
            University.INSTANCE.SelectCourse(title);
            break;
          }
        }

        Assert.IsTrue(find);
      }

      private void ResetMaxGroupSizeForCourse(string courseTitle,
        int maxGroupSize)
      {
        University.INSTANCE.SelectCourse(courseTitle);
        University.INSTANCE.SelectedCourseResetMaxGroupSize(
          maxGroupSize);
      }

      private void VerifyCourse(string courseTitle,
        IEnumerable<string> lecturersNames)
      {
        this.SelectCourse(courseTitle);
        HashSet<int> lecturersNumbers =
        new HashSet<int>(University.INSTANCE.
          GetLecturersNumbersFromSelectedCourse());

        foreach (string name in lecturersNames)
        {
          this.SelectFaculty(name);
          Assert.IsTrue(
            University.INSTANCE.GetSelectedFacultyName()
              == name);
          Assert.IsTrue(lecturersNumbers.Remove(
            University.INSTANCE.GetSelectedFacultyNumber()));
        }

        Assert.IsTrue(lecturersNumbers.Count == 0);
      }

      private void VerifyMaxGroupSizeForCourse(
        string courseTitle, int maxGroupSize)
      {
        this.SelectCourse(courseTitle);
        Assert.IsTrue(
          University.INSTANCE.GetMaxSizeGroupForSelectedCourse()
            == maxGroupSize);
      }

      private void CreateStudent(string courseTitle,
        string studentName, int studentNumber)
      {
        University.INSTANCE.SelectCourse(courseTitle);
        University.INSTANCE.RegisterLecturerOnSelectedCourse(
          studentName, studentNumber);
      }
```

```
private void SelectStudent(string studentName)
{
  bool find = false;
  foreach (int number in
    University.INSTANCE.GetStudentsNumbers())
  {
    University.INSTANCE.SelectStudent(number);
    if (University.INSTANCE.GetSelectedStudentName()
      == studentName)
    {
      find = true;
      break;
    }
  }

  Assert.IsTrue(find);
}

private void SetInstructionLanguageForTeachingAssistant(
  string studentName, string instructionLanguage)
{
  this.SelectStudent(studentName);
  University.INSTANCE.
    SelectedStudentCanBeTeachingAssistant(
      instructionLanguage);
}

private void SetStudentToBeAssistantOfCourse(
  string studentName, string courseTitle)
{
  this.SelectStudent(studentName);
  this.SelectCourse(courseTitle);

  University.INSTANCE.
    AddSelectedTAToSelectedAssitedCourse();
}

private void SetStudentAsResearchAssistant(
  string studentName, float grantAmount)
{
  this.SelectStudent(studentName);
  University.INSTANCE.SelectedStudentIsAResearchAssistant(
    grantAmount);
}

private void SetStudentToBeAssistantOfFaculty(
  string studentName, string facultyName)
{
  this.SelectStudent(studentName);
  this.SelectFaculty(facultyName);
  University.INSTANCE.
    AddSelectedResearchAssistantToSelectedSupervisor();
}
```

```
private void VerifyIfStudentIsAssistantOfCourses(
   string studentName, IEnumerable<string> coursesTitles)
{
   this.SelectStudent(studentName);
   HashSet<string> acTitles =
      new HashSet<string>(University.INSTANCE.
        GetSelectedTACoursesTitles());

   foreach (string title in coursesTitles)
   {
      Assert.IsTrue(acTitles.Remove(title));
   }

   Assert.IsTrue(acTitles.Count == 0);
}

private void VerifyInstructionLanguageForTeachingAssistant(
   string studentName, string instructionLanguage)
{
   this.SelectStudent(studentName);
   Assert.IsTrue(University.INSTANCE.
      GetInstructionLanguageOfSelectedStudent() ==
        instructionLanguage);
}

private void VerifyStudent(string studentName,
   int studentNumber)
{
   this.SelectFaculty(studentName);
   Assert.IsTrue(
      University.INSTANCE.GetSelectedFacultyNumber() ==
        studentNumber);
}

private void VerifyResearchAssistantGrantAmout(
   string studentName, float studentGrantAmount)
{
   this.SelectStudent(studentName);
   Assert.IsTrue(University.INSTANCE.
      GetGrantAmountForSelectedResearchAssistant() ==
        studentGrantAmount);
}

private void StudentAttendsOtherCourse(
   string studentName, string otherCourseTitle)
{
   this.SelectCourse(otherCourseTitle);
   this.SelectStudent(studentName);
   University.INSTANCE.
      AddSelectedLearnerToSelectedLecture();
}

private void VerifyLearnersOfLecture(string courseTitle,
```

```
      IEnumerable<string> learnersNames)
    {
      this.SelectCourse(courseTitle);
      HashSet<int> learnersNumbers = new HashSet<int>(
        University.INSTANCE.
          GetLearnersNumbersOfSelectedLecture());

      foreach (string name in learnersNames)
      {
        this.SelectStudent(name);
        Assert.IsTrue(learnersNumbers.Remove(
          University.INSTANCE.GetSelectedStudentNumber()));
      }

      Assert.IsTrue(learnersNumbers.Count == 0);
    }

    private void SetAttendsMark(string studentName,
      string courseTitle, int mark)
    {
      this.SelectCourse(courseTitle);
      this.SelectStudent(studentName);
      University.INSTANCE.
        SetMarkForSelectedLearnerOnSelectedLecture(mark);
    }

    private int GetAttendsMark(string studentName,
      string courseTitle)
    {
      this.SelectCourse(courseTitle);
      this.SelectStudent(studentName);
      return University.INSTANCE.
        GetMarkForSelectedLearnerOnSelectedLecture();
    }

    private void StudentIsResearchAssistant(string studentName,
      string facultyName)
    {
      this.SelectStudent(studentName);
      this.SelectFaculty(facultyName);
      University.INSTANCE.
        AddSelectedResearchAssistantToSelectedSupervisor();
    }

    private void VerifyIfStudentIsResearchAssistant(
      string studentName,
      IEnumerable<string> supervisorsNames)
    {
      this.SelectStudent(studentName);
      HashSet<int> supervisorsNumbers =
        new HashSet<int>(University.INSTANCE.
        GetSupervisorsNumbersFromSelectedResearchAssistant());

      foreach (string supervidorName in supervisorsNames)
```

```
      {
        this.SelectFaculty(supervidorName);
        Assert.IsTrue(supervisorsNumbers.Remove(
          University.INSTANCE.GetSelectedFacultyNumber()));
      }

      Assert.IsTrue(supervisorsNumbers.Count == 0);
    }

    private void StudentNotWorksForFaculties(
      string studentName, string facultyName)
    {
      this.SelectStudent(studentName);
      this.SelectFaculty(facultyName);
      University.INSTANCE.
RemoveSelectedResearchAssistantFromSelectedSupervisor();
    }

    private void RemoveLearnersOfLecture(string lectureTitle,
      IEnumerable<string> learnersNames)
    {
      this.SelectCourse(lectureTitle);
      foreach (string learnerName in learnersNames)
      {
        this.SelectStudent(learnerName);
        University.INSTANCE.
          RemoveSelectedLearnerFromSelectedLecture();
      }
    }

    private void StudentIsNotResearchAssistant(
      string studentName)
    {
      this.SelectStudent(studentName);
      University.INSTANCE.
        SelectedStudentIsNotAResearchAssistant();
    }

    private void StudentIsNotTeachingAssistant(
      string studentName)
    {
      this.SelectStudent(studentName);
      University.INSTANCE.
        SelectedStudentCannotBeTeachingAssistant();
    }

    private void StudentNotAssistsCourse(
      string studentName, string courseTitle)
    {
      this.SelectStudent(studentName);
      this.SelectCourse(courseTitle);
      University.INSTANCE.
        RemoveSelectedTAFromSelectedAssitedCourse();
    }
```

```
    private void RemoveCourse(string courseTitle)
    {
      this.SelectCourse(courseTitle);
      University.INSTANCE.DeleteSelectedCourse();
    }

    private void RemoveFaculty(string facultyName)
    {
      this.SelectFaculty(facultyName);
      University.INSTANCE.DeleteSelectedFaculty();
    }

    #endregion métodos auxiliares
  }
}
```

A listagem a seguir descreve o arquivo makefile a ser utilizado pelo compilador mcs.exe, através do comando de console: `mcs.exe @makefile`

```
-debug+
-optimize+
-out:../bin/Univeristy.exe
-platform:anycpu
-sdk:4
-target:exe
-reference:../bin/AssociationSharpLib.dll

./Program.cs
./DomainLayer.cs
./Test.cs
```

APÊNDICE D – CÓDIGO DOS TESTES DE DESEMPENHO

As duas próximas listagens representam os arquivos fontes Association# dos dois programas de teste. O primeiro é para teste de associações básicas e o segundo para teste de classes de associação.

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *          Support.
 *
 * Source File: /C#/BasicModelAS.as
 *
 * Description: A test of association implemented in
 *                 Association#.
 *
 * Language: Association#.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
namespace
  PerformanceTest.ManyToManyBidirectional.AssociationSharp
{
  public class A
  {
    public rolename b [*] in AB;
  }

  public class B
  {
    public rolename a [*] in AB;
  }

  public association AB
    between A
    and  B;
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
```

```
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *          Support.
 *
 * Source File: /C#/AssociationClassModelAS.as
 *
 * Description: A test of association classes implemented in
 *              Association#.
 *
 * Language: Association#.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 ************************************************************/
namespace PerformanceTest.AssociationClass.
  ManyToManyBidirectional.AssociationSharp
{
  public class A
  {
    public rolename b [*] in AB;
  }

  public class B
  {
    public rolename a [*] in AB;
  }

  public association class AB
    between A
    and  B { }
}
```

As próximas duas listagens apresentam os códigos C# resultantes da tradução dos códigos Association# presentes nas duas listagens anteriores, acrescidos das implementações dos pares de associação e classe de associação, utilizando o *pattern mutual friends* e o *pattern de Gessenharter* para cada par.

```
/************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
```

```
 *          Support.
 *
 * Source File: /C#/BasicModelAS.cs
 *
 * Description: Translated code from the source file
 *              BasicModelAS.as written in Association#.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 ************************************************************/
using AssociationSharp.Lang;

namespace
  PerformanceTest.ManyToManyBidirectional.AssociationSharp
{

  public class A
  {
    public readonly AB.AEnd b;
    public A()
    {
      this.b = new AB.AEnd(this);
    }
  }

  public class B
  {
    public readonly AB.BEnd a;
    public B()
    {
      this.a = new AB.BEnd(this);
    }
  }

  public class AB : Association<A, B>
  {
    public static readonly AB association = new AB();

    private AB() : base(Multiplicity.ZERO_TO_MANY_RANGE,
      Multiplicity.ZERO_TO_MANY_RANGE,
      getSourceEnd, getTargetEnd) { }

    public class AEnd : SourceAssociationEnd
    {
```

```
      public AEnd(A sourceObject) : base(sourceObject,
        association) { }
    }

    public class BEnd : TargetAssociationEnd
    {
      public BEnd(B targetObject) : base(targetObject,
        association) { }
    }

    private static AEnd getSourceEnd(A sourceObject)
    {
      return sourceObject.b;
    }

    private static BEnd getTargetEnd(B targetObject)
    {
      return targetObject.a;
    }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *          Support.
 *
 * Source File: /C#/AssociationClassModelAS.cs
 *
 * Description: Translated code from the source file
 *              AssociationClassModelAS.as written in
 *              Association#.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using AssociationSharp.Lang;

namespace
PerformanceTest.AssociationClass.ManyToManyBidirectional.Associ
ationSharp
```

```
{
  public class A
  {
    public readonly AB.AtoB.SourceAssociationEnd b;
    public A()
    {
      this.b = new AB.AtoB.AEnd(this);
    }
  }

  public class B
  {
    public readonly AB.AtoB.TargetAssociationEnd a;
    public B()
    {
      this.a = new AB.AtoB.BEnd(this);
    }
  }

  public class AB
  {
    public class AtoB : AssociationClass<A, B, AB>
    {
      public static readonly AtoB association = new AtoB();

      private AtoB()
        : base(Multiplicity.ZERO_TO_MANY_RANGE,
          Multiplicity.ZERO_TO_MANY_RANGE,
          getSourceEnd, getTargetEnd) { }

      public class AEnd : SourceAssociationEnd
      {
        public AEnd(A sourceObject) : base(sourceObject,
          association) { }
      }

      public class BEnd : TargetAssociationEnd
      {
        public BEnd(B targetObject) : base(targetObject,
          association) { }
      }

      private static AssociationClass<A,B,AB>.
        SourceAssociationEnd getSourceEnd(A sourceObject)
      {
        return sourceObject.b;
      }

      private static AssociationClass<A,B,AB>.
        TargetAssociationEnd getTargetEnd(B targetObject)
      {
        return targetObject.a;
      }
    }
```

```
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *          Support.
 *
 * Source File: /C#/BasicModelCS.cs
 *
 * Description: A test of association implemented in mutual
 *              friends.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System.Collections.Generic;
using AssociationSharp.Commons;

namespace PerformanceTest.ManyToManyBidirectional.CSharp
{
  public class A
  {
    private readonly HashSet<B> b = new HashSet<B>();

    public bool addB(B b)
    {
      return b != null && addAB(this, b);
    }

    public bool removeB(B b)
    {
      return b != null && removeAB(this, b);
    }

    public IEnumerable<B> getB()
    {
      return new UnmodifiedSet<B>(this.b);
    }

    internal static bool addAB(A a, B b)
```

```
    {
      bool added;
      if (added = (a.b.Add(b)))
      {
        b.a.Add(a);
      }

      return added;
    }

    internal static bool removeAB(A a, B b)
    {
      bool removed;

      if (removed = b.a.Remove(a))
      {
        a.b.Remove(b);
      }
      return removed;
    }
  }

  public class B
  {
    internal readonly HashSet<A> a = new HashSet<A>();

    public bool addA(A a)
    {
      return a != null && A.addAB(a, this);
    }

    public bool removeA(A a)
    {
      return a != null && A.removeAB(a, this);
    }

    public IEnumerable<A> getA()
    {
      return new UnmodifiedSet<A>(this.a);
    }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *          Support.
 *
 * Source File: /C#/AssociationClassModelCS.cs
 *
 * Description: A test of association class implemented in
```

```
 *               mutual friends.
 *
 * Description: Generated code from the source file
 *               AssociationClassModelCS.as written in
 *               Association#.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 ************************************************************/
using System.Collections.Generic;

namespace
  PerformanceTest.AssociationClass.ManyToManyBidirectional.
    CSharp
{
  public class A
  {
    private readonly Dictionary<B, AB> b =
      new Dictionary<B, AB>();

    public AB addB(B b)
    {
      AB ab = null;

      if (b != null)
      {
        ab = addAB(this, b);
      }

      return ab;
    }

    public bool removeB(B b)
    {
      return b != null && removeAB(this, b);
    }

    public IEnumerable<B> getB()
    {
      return this.b.Keys;
    }

    public IEnumerable<AB> getAB()
    {
```

```
      return this.b.Values;
    }

    internal static AB addAB(A a, B b)
    {
      AB ab = null;
      if (!a.b.ContainsKey(b))
      {
        ab = new AB();
        a.b.Add(b, ab);
        b.a.Add(a, ab);
      }

      return ab;
    }

    internal static bool removeAB(A a, B b)
    {
      bool removed;

      if (removed = b.a.Remove(a))
      {
        a.b.Remove(b);
      }
      return removed;
    }
  }

  public class B
  {
    internal readonly Dictionary<A, AB> a =
      new Dictionary<A, AB>();

    public AB addA(A a)
    {
      AB ab = null;

      if (a != null)
      {
        ab = A.addAB(a, this);
      }

      return ab;
    }

    public bool removeA(A a)
    {
      return a != null && A.removeAB(a, this);
    }

    public IEnumerable<A> getA()
    {
      return this.a.Keys;
    }
```

```
    public IEnumerable<AB> getAB()
    {
      return this.a.Values;
    }
  }

  public class AB
  {
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *          Support.
 *
 * Source File: /C#/BasicModelGe.cs
 *
 * Description: A test of association implemented in the code
 *              pattern of Gessenharter.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System.Collections.Generic;
using System;

namespace PerformanceTest.ManyToManyBidirectional.Gessenharter
{

  public class A
  {
    /* association implementation code */
    static A()
    {
      AB.getHandle(typeof(A));
    }
    private static AB fAB;
    internal static void takeHandle(AB assoc)
    {
```

```
      fAB = assoc;
    }

    /* members of owned end */
    public void addB(B b)
    {
      fAB.createLink(this, b);
    }
    public void removeB(B b)
    {
      fAB.destroyLink(this, b);
    }
    public LinkedList<B> getB()
    {
      return fAB.getLinks(this);
    }
  }

  public class B
  {

    /* association implementation code */
    static B()
    {
      AB.getHandle(typeof(B));
    }
    private static AB fAB;
    internal static void takeHandle(AB assoc)
    {
      fAB = assoc;
    }

    /* members of owned end */
    public void addA(A a)
    {
      fAB.createLink(a, this);
    }
    public void removeA(A a)
    {
      fAB.destroyLink(a, this);
    }
    public LinkedList<A> getA()
    {
      return fAB.getLinks(this);
    }
  }

  public class AB
  {
    /* singleton pattern*/
    private static readonly AB fInstance = new AB();
    private AB() { }

    /* providing handles */
```

```
public static void getHandle(System.Type type)
{
  if (type == typeof(A))
  {
    A.takeHandle(fInstance);
  }
  else if (type == typeof(B))
  {
    B.takeHandle(fInstance);
  }
}

/* methodos for managing the association */
private readonly LinkedList<Link> tuples =
  new LinkedList<Link>();

public void createLink(A a, B b)
{
  if (a != null && b != null)
  {
    this.tuples.AddLast(new Link(a, b));
  }
  else
  {
    throw new Exception();
  }
}

public void destroyLink(A a, B b)
{
  foreach (Link l in this.tuples)
  {
    if (l.a == a && l.b == b)
    {
      this.tuples.Remove(l);
      break;
    }
  }
}

public LinkedList<A> getLinks(B b)
{
  LinkedList<A> result = new LinkedList<A>();
  foreach (Link l in this.tuples)
  {
    if (l.b == b)
    {
      result.AddLast(l.a);
    }
  }
  return result;
}

public LinkedList<B> getLinks(A a)
```

```
    {
      LinkedList<B> result = new LinkedList<B>();
      foreach (Link l in this.tuples)
      {
        if (l.a == a)
        {
          result.AddLast(l.b);
        }
      }
      return result;
    }

    /* implementation of links */
    private class Link : Object
    {
      public readonly A a;
      public readonly B b;

      public Link(A a, B b)
      {
        this.a = a;
        this.b = b;
      }
    }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *          Support.
 *
 * Source File: /C#/AssociationClassModelGe.cs
 *
 * Description: A test of association class implemented in
 *              the code pattern of Gessenharter.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 *************************************************************/
using System.Collections.Generic;
```

```
using System;

namespace
  PerformanceTest.AssociationClass.ManyToManyBidirectional.
    Gessenharter
{

  public class A
  {

    /* association implementation code */
    static A()
    {
      AB.getHandle(typeof(A));
    }
    private static AB fAB;
    internal static void takeHandle(AB assoc)
    {
      fAB = assoc;
    }

    /* members of owned end */
    public void addB(B b)
    {
      fAB.createLink(this, b);
    }
    public void removeB(B b)
    {
      fAB.destroyLink(this, b);
    }
    public LinkedList<B> getB()
    {
      return fAB.getLinks(this);
    }
    public LinkedList<AB.Object> getAB()
    {
      return fAB.getObjects(this);
    }
  }

  public class B
  {

    /* association implementation code */
    static B()
    {
      AB.getHandle(typeof(B));
    }
    private static AB fAB;
    internal static void takeHandle(AB assoc)
    {
      fAB = assoc;
    }
```

```
    /* members of owned end */
    public void addA(A a)
    {
      fAB.createLink(a, this);
    }
    public void removeA(A a)
    {
      fAB.destroyLink(a, this);
    }
    public LinkedList<A> getA()
    {
      return fAB.getLinks(this);
    }
    public LinkedList<AB.Object> getAB()
    {
      return fAB.getObjects(this);
    }
  }

  public class AB
  {
    /* singleton pattern*/
    private static readonly AB fInstance = new AB();
    private AB() { }

    /* providing handles */
    public static void getHandle(System.Type type)
    {
      if (type == typeof(A))
      {
        A.takeHandle(fInstance);
      }
      else if (type == typeof(B))
      {
        B.takeHandle(fInstance);
      }
    }

    /* methodos for managing the association */
    private readonly LinkedList<Link> tuples =
      new LinkedList<Link>();

    public void createLink(A a, B b)
    {
      if (a != null && b != null)
      {
        this.tuples.AddLast(new Link(a, b));
      }
      else
      {
        throw new Exception();
      }
    }
```

```
public void destroyLink(A a, B b)
{
  foreach (Link l in this.tuples)
  {
    if (l.a == a && l.b == b)
    {
      this.tuples.Remove(l);
      break;
    }
  }
}

public LinkedList<A> getLinks(B b)
{
  LinkedList<A> result = new LinkedList<A>();
  foreach (Link l in this.tuples)
  {
    if (l.b == b)
    {
      result.AddLast(l.a);
    }
  }
  return result;
}

public LinkedList<B> getLinks(A a)
{
  LinkedList<B> result = new LinkedList<B>();

  foreach (Link l in this.tuples)
  {
    if (l.a == a)
    {
      result.AddLast(l.b);
    }
  }

  return result;
}

public LinkedList<Object> getObjects(B b)
{
  LinkedList<Object> result = new LinkedList<Object>();

  foreach (Link l in this.tuples)
  {
    if (l.b == b)
    {
      result.AddLast(l);
    }
  }

  return result;
}
```

```
    public LinkedList<Object> getObjects(A a)
    {
      LinkedList<Object> result = new LinkedList<Object>();

      foreach (Link l in this.tuples)
      {
        if (l.a == a)
        {
          result.AddLast(l);
        }
      }

      return result;
    }

    /* implementation of links */
    private class Link : Object
    {
      public readonly A a;
      public readonly B b;

      public Link(A a, B b)
      {
        this.a = a;
        this.b = b;
      }
    }

    /* implementation of association class */
    public class Object
    {
      /* attributes and methods */
    }
  }
}
```

A próxima listagem contém o código da aplicação que executa os testes nas 6 implementações listadas anteriormente neste apêndice.

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *          Support.
 *
 * Source File: /C#/Program.cs
 *
 * Description: The application that runs the tests in C# code.
 *
 * Language: C# 4 - .NET Framework 4 Client Profile
```

```
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *            Wazlawick 2011.  Permission to copy, use, modify,
 *            sell and distribute this software is granted
 *            provided this copyright notice appears in  all
 *            copies. This software is provided "as is" without
 *            expressor implied warranty, and with no claim as
 *            to its suitability for any purpose.
 ************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;

namespace Performance
{
  class Program
  {
    const int INTERACTIONS = 100000;

    private static readonly StreamWriter logger =
      new StreamWriter(@".\Performance.log", true);

    static void Main(string[] args)
    {
      if (args.Count() >= 1)
      {
        logger.AutoFlush = true;

        switch (int.Parse(args[0]))
        {
          case 1: MFAssociation(); break;
          case 2: MFAssociationClass(); break;
          case 3: ASharpAssociation(); break;
          case 4: ASharpAssociationClass(); break;
          case 5: GeAssociation(); break;
          case 6: GeAssociationClass(); break;
          default: WrongOption(args); break;
        }
      }
      else
      {
        WrongOption(args);
      }
    }

    private static void WrongOption(object option)
    {
      Console.WriteLine("Opção inválida! {0}", option);
    }
```

```
    static void log(string text)
    {
      logger.WriteLine("{0:f} - {1}", DateTime.Now, text);
      Console.WriteLine("{0:f} - {1}", DateTime.Now, text);
    }

    private static void ASharpAssociation()
    {
      log("Iniciando 'Association# Association'...");

      DateTime beginning = DateTime.Now;


  PerformanceTest.ManyToManyBidirectional.AssociationSharp.A a
    = new PerformanceTest.ManyToManyBidirectional.
      AssociationSharp.A();
  List<PerformanceTest.ManyToManyBidirectional.
    AssociationSharp.B> listB =
      new List<PerformanceTest.ManyToManyBidirectional.
        AssociationSharp.B>();

      for (int i = 0; i < INTERACTIONS; i++)
      {
        PerformanceTest.ManyToManyBidirectional.
          AssociationSharp.B b =
            new PerformanceTest.ManyToManyBidirectional.
              AssociationSharp.B();
        a.b.Add(b);
        listB.Add(b);
      }

      for (int i = 0; i < INTERACTIONS; i++)
      {
        a.b.Get();
      }

      for (int i = 0; i < INTERACTIONS; i++)
      {
        a.b.Remove(listB[i]);
      }

      log("Tempo total " +
        DateTime.Now.Subtract(beginning).Duration().
        TotalMilliseconds + " ms");
    }

    private static void MFAssociation()
    {
      log("Iniciando 'Mutual Friends Association'...");

      DateTime beginning = DateTime.Now;

      PerformanceTest.ManyToManyBidirectional.CSharp.A a = new
        PerformanceTest.ManyToManyBidirectional.CSharp.A();
```

```
      List<PerformanceTest.ManyToManyBidirectional.CSharp.B>
        listB = new List<PerformanceTest.
          ManyToManyBidirectional.CSharp.B>();

      for (int i = 0; i < INTERACTIONS; i++)
      {
        PerformanceTest.ManyToManyBidirectional.CSharp.B b =
          new PerformanceTest.ManyToManyBidirectional.
            CSharp.B();
        a.addB(b);
        listB.Add(b);
      }

      for (int i = 0; i < INTERACTIONS; i++)
      {
        a.getB();
      }

      for (int i = 0; i < INTERACTIONS; i++)
      {
        a.removeB(listB[i]);
      }

      log("Tempo total " +
        DateTime.Now.Subtract(beginning).Duration().
        TotalMilliseconds + " ms");
    }

    private static void GeAssociation()
    {
      log("Iniciando 'Gessenharter (C#) Association'...");

      DateTime beginning = DateTime.Now;

      PerformanceTest.ManyToManyBidirectional.Gessenharter.A a
        = new PerformanceTest.ManyToManyBidirectional.
          Gessenharter.A();
      List<PerformanceTest.ManyToManyBidirectional.
        Gessenharter.B> listB = new List<PerformanceTest.
          ManyToManyBidirectional.Gessenharter.B>();

      for (int i = 0; i < INTERACTIONS; i++)
      {
        PerformanceTest.ManyToManyBidirectional.Gessenharter.B
          b = new PerformanceTest.ManyToManyBidirectional.
            Gessenharter.B();
        a.addB(b);
        listB.Add(b);
      }

      for (int i = 0; i < INTERACTIONS; i++)
      {
        a.getB();
      }
```

```
    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.removeB(listB[i]);
    }

    log("Tempo total " + DateTime.Now.Subtract(beginning).
      Duration().TotalMilliseconds + " ms");
  }

  private static void ASharpAssociationClass()
  {
    log("Iniciando 'Association# AssociationClass'...");

    DateTime beginning = DateTime.Now;

    PerformanceTest.AssociationClass.
      ManyToManyBidirectional.AssociationSharp.A a = new
        PerformanceTest.AssociationClass.
          ManyToManyBidirectional.AssociationSharp.A();
    List<PerformanceTest.AssociationClass.
      ManyToManyBidirectional.AssociationSharp.B> listB =
        new List<PerformanceTest.AssociationClass.
          ManyToManyBidirectional.AssociationSharp.B>();

    for (int i = 0; i < INTERACTIONS; i++)
    {
      PerformanceTest.AssociationClass.
        ManyToManyBidirectional.AssociationSharp.B b = new
          PerformanceTest.AssociationClass.
            ManyToManyBidirectional.AssociationSharp.B();
      a.b.Add(b);
      listB.Add(b);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.b.Get();
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.b.GetAssociationLink();
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.b.Remove(listB[i]);
    }

    log("Tempo total " +
      DateTime.Now.Subtract(beginning).Duration().
      TotalMilliseconds + " ms");
  }
```

```
private static void MFAssociationClass()
{
  log("Iniciando 'Mutual Friends AssociationClass'...");

  DateTime beginning = DateTime.Now;

  PerformanceTest.AssociationClass.
    ManyToManyBidirectional.CSharp.A a = new
      PerformanceTest.AssociationClass.
        ManyToManyBidirectional.CSharp.A();
  List<PerformanceTest.AssociationClass.
    ManyToManyBidirectional.CSharp.B> listB = new
      List<PerformanceTest.AssociationClass.
        ManyToManyBidirectional.CSharp.B>();

  for (int i = 0; i < INTERACTIONS; i++)
  {
    PerformanceTest.AssociationClass.
      ManyToManyBidirectional.CSharp.B b = new
        PerformanceTest.AssociationClass.
          ManyToManyBidirectional.CSharp.B();
    a.addB(b);
    listB.Add(b);
  }

  for (int i = 0; i < INTERACTIONS; i++)
  {
    a.getB();
  }

  for (int i = 0; i < INTERACTIONS; i++)
  {
    a.getAB();
  }

  for (int i = 0; i < INTERACTIONS; i++)
  {
    a.removeB(listB[i]);
  }

  log("Tempo total " +
    DateTime.Now.Subtract(beginning).Duration().
    TotalMilliseconds + " ms");
}

private static void GeAssociationClass()
{
  log(
    "Iniciando 'Gessenharter (C#) AssociationClass'...");

  DateTime beginning = DateTime.Now;

  PerformanceTest.AssociationClass.
```

```
          ManyToManyBidirectional.Gessenharter.A a = new
            PerformanceTest.AssociationClass.
              ManyToManyBidirectional.Gessenharter.A();
       List<PerformanceTest.AssociationClass.
          ManyToManyBidirectional.Gessenharter.B> listB = new
            List<PerformanceTest.AssociationClass.
              ManyToManyBidirectional.Gessenharter.B>();

       for (int i = 0; i < INTERACTIONS; i++)
       {
          PerformanceTest.AssociationClass.
            ManyToManyBidirectional.Gessenharter.B b = new
              PerformanceTest.AssociationClass.
                ManyToManyBidirectional.Gessenharter.B();
          a.addB(b);
          listB.Add(b);
       }

       for (int i = 0; i < INTERACTIONS; i++)
       {
          a.getB();
       }

       for (int i = 0; i < INTERACTIONS; i++)
       {
          a.getAB();
       }

       for (int i = 0; i < INTERACTIONS; i++)
       {
          a.removeB(listB[i]);
       }

       log("Tempo total " + DateTime.Now.Subtract(beginning).
          Duration().TotalMilliseconds + " ms");
    }
  }
}
```

A listagem a seguir descreve o arquivo makefile a ser utilizado pelo compilador mcs.exe, através do comando de console: `mcs.exe @makefile`

```
-debug-
-optimize+
-out:../../bin/Performance.exe
-platform:anycpu
-sdk:4
-target:exe
-reference:../../bin/AssociationSharpLib.dll

./Program.cs
./BasicModelCS.cs
./BasicModelAS.cs
```

```
./BasicModelGe.cs
./AssociationClassModelCS.cs
./AssociationClassModelAS.cs
./AssociationClassModelGe.cs
```

As próximas listagens apresentam as implementações em AspectJ/Java dos pares de associação e classe de associação, utilizando o *pattern mutual friends*, o *pattern de Gessenharter* e a biblioteca RAL.

```
/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          gessenharter/association/A.java
 *
 * Description: The class A implemented in the code pattern of
 *          Gessenharter.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/

package performance.gessenharter.association;

import java.util.*;

public class A {
   /* association implementation code */
   static {
     AB.getHandle(A.class);
   }
   private static AB fAB;
   static void takeHandle(AB assoc)
   {
     fAB = assoc;
   }

   /* members of owned end */
   public void addB(B b) {
```

```
    fAB.createLink(this,b);
  }
  public void removeB(B b) {
    fAB.destroyLink(this,b);
  }
  public LinkedList<B> getB() {
    return fAB.getLinks(this);
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          gessenharter/association/AB.java
 *
 * Description: The association AB implemented in the code
 *          pattern of Gessenharter.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/

package performance.gessenharter.association;

import java.util.*;

public class AB {
  /* singleton pattern*/
  private static final AB fInstance = new AB();
  private AB() { }

  /* providing handles */
  public static void getHandle(Class<?> type) {
    if (type == A.class) {
      A.takeHandle(fInstance);
    }
    else if (type == B.class) {
      B.takeHandle(fInstance);
```

```
    }
  }

  /* methodos for managing the association */
  private final LinkedList<Link> tuples =
    new LinkedList<Link>();

  public void createLink(A a, B b) {
    if (a != null && b != null) {
      this.tuples.add(new Link(a,b));
    }
  }

  public void destroyLink(A a, B b) {
    for (Link l : this.tuples) {
      if (l.a == a && l.b == b) {
        this.tuples.remove(l);
        break;
      }
    }
  }

  public LinkedList<A> getLinks(B b) {
    LinkedList<A> result = new LinkedList<A>();
    for(Link l : this.tuples) {
      if (l.b == b) {
        result.add(l.a);
      }
    }
    return result;
  }

  public LinkedList<B> getLinks(A a)
  {
    LinkedList<B> result = new LinkedList<B>();
    for (Link l : this.tuples)
    {
      if (l.a == a)
      {
        result.add(l.b);
      }
    }
    return result;
  }

  /* implementation of links */
  private class Link extends Object {
    public final A a;
    public final B b;

    private Link(A a, B b) {
      this.a = a;
      this.b = b;
    }
```

```
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          gessenharter/association/B.java
 *
 * Description: The class B implemented in the code pattern of
 *          Gessenharter.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/

package performance.gessenharter.association;

import java.util.*;

public class B {
  /* association implementation code */
  static {
    AB.getHandle(B.class);
  }
  private static AB fAB;
  static void takeHandle(AB assoc)
  {
    fAB = assoc;
  }

  /* members of owned end */
  public void addA(A a)
  {
    fAB.createLink(a, this);
  }
  public void removeA(A a)
  {
    fAB.destroyLink(a, this);
```

```
  }
  public LinkedList<A> getA()
  {
    return fAB.getLinks(this);
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          gessenharter/associationclass/A.java
 *
 * Description: The class A implemented in the code pattern of
 *          Gessenharter.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/
package performance.gessenharter.associationclass;

import java.util.*;

public class A {
  /* association implementation code */
  static {
    AB.getHandle(A.class);
  }
  private static AB fAB;
  static void takeHandle(AB assoc)
  {
    fAB = assoc;
  }

  /* members of owned end */
  public void addB(B b) {
    fAB.createLink(this,b);
  }
  public void removeB(B b) {
```

```
    fAB.destroyLink(this,b);
  }
  public LinkedList<B> getB() {
    return fAB.getLinks(this);
  }

  public LinkedList<AB.Link> getAB() {
    return fAB.getObjects(this);
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          gessenharter/associationclass/AB.java
 *
 * Description: The association class AB implemented in the
 *          code pattern of Gessenharter.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/
package performance.gessenharter.associationclass;

import java.util.*;

public class AB {
  /* singleton pattern*/
  private static final AB fInstance = new AB();
  private AB() { }

  /* providing handles */
  public static void getHandle(Class<?> type) {
    if (type == A.class) {
      A.takeHandle(fInstance);
    }
    else if (type == B.class) {
      B.takeHandle(fInstance);
```

```
    }
  }

  /* methodos for managing the association */
  private final LinkedList<Link> tuples =
    new LinkedList<Link>();

  public void createLink(A a, B b) {
    if (a != null && b != null) {
      this.tuples.add(new Link(a,b));
    }
  }

  public void destroyLink(A a, B b) {
    for (Link l : this.tuples) {
      if (l.a == a && l.b == b) {
        this.tuples.remove(l);
        break;
      }
    }
  }

  public LinkedList<A> getLinks(B b) {
    LinkedList<A> result = new LinkedList<A>();
    for(Link l : this.tuples) {
      if (l.b == b) {
        result.add(l.a);
      }
    }
    return result;
  }

  public LinkedList<B> getLinks(A a)
  {
    LinkedList<B> result = new LinkedList<B>();
    for (Link l : this.tuples)
    {
      if (l.a == a)
      {
        result.add(l.b);
      }
    }
    return result;
  }

  public LinkedList<AB.Link> getObjects(A a) {
    LinkedList<AB.Link> result = new LinkedList<AB.Link>();
    for (Link l : this.tuples)
    {
      if (l.a == a)
      {
        result.add(l);
      }
    }
```

```
    return result;
  }

  public LinkedList<AB.Link> getObjects(B b) {
    LinkedList<AB.Link> result = new LinkedList<AB.Link>();
    for (Link l : this.tuples)
    {
      if (l.b == b)
      {
        result.add(l);
      }
    }
    return result;
  }

  /* implementation of links */
  public class Link extends Object {
    public final A a;
    public final B b;

    private Link(A a, B b) {
      this.a = a;
      this.b = b;
    }
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          gessenharter/associationclass/B.java
 *
 * Description: The class B implemented in the code pattern of
 *          Gessenharter.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/
```

```
package performance.gessenharter.associationclass;

import java.util.*;

public class B {
   /* association implementation code */
   static {
     AB.getHandle(B.class);
   }
   private static AB fAB;
   static void takeHandle(AB assoc)
   {
     fAB = assoc;
   }

   /* members of owned end */
   public void addA(A a)
   {
     fAB.createLink(a, this);
   }
   public void removeA(A a)
   {
     fAB.destroyLink(a, this);
   }
   public LinkedList<A> getA()
   {
     return fAB.getLinks(this);
   }

   public LinkedList<AB.Link> getAB() {
     return fAB.getObjects(this);
   }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *      Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          mutualfriends/association/A.java
 *
 * Description: The class A implemented in pattern mutual
 *          friends.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
```

```
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/
package performance.mutualfriends.association;

import java.util.*;

public class A
{
  private final HashSet<B> b = new HashSet<B>();

  public boolean addB(B b)
  {
    return b != null && addAB(this, b);
  }

  public boolean removeB(B b)
  {
    return b != null && removeAB(this, b);
  }

  public Set<B> getB()
  {
    return Collections.unmodifiableSet(this.b);
  }

  static boolean addAB(A a, B b)
  {
    boolean added;
    if (added = (a.b.add(b)))
    {
      b.a.add(a);
    }

    return added;
  }

  static boolean removeAB(A a, B b)
  {
    boolean removed;

    if (removed = b.a.remove(a))
    {
      a.b.remove(b);
    }
    return removed;
  }
}

/*************************************************************
```

```
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          mutualfriends/associationclass/B.java
 *
 * Description: The class B implemented in pattern mutual
 *          friends.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/
package performance.mutualfriends.association;

import java.util.*;

public class B
{
   final HashSet<A> a = new HashSet<A>();

   public boolean addA(A a)
   {
     return a != null && A.addAB(a, this);
   }

   public boolean removeA(A a)
   {
     return a != null && A.removeAB(a, this);
   }

   public Set<A> getB()
   {
     return Collections.unmodifiableSet(this.a);
   }
}

/************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
```

```
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          mutualfriends/associationclass/A.java
 *
 * Description: The class A implemented in pattern mutual
 *          friends.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/
package performance.mutualfriends.associationclass;

import java.util.*;

public class A
{
  private final HashMap<B, AB> b = new HashMap<B, AB>();

  public AB addB(B b)
  {
    AB ab = null;

    if (b != null)
    {
      ab = addAB(this, b);
    }

    return ab;
  }

  public boolean removeB(B b)
  {
    return b != null && removeAB(this, b);
  }

  public Set<B> getB()
  {
    return Collections.unmodifiableSet(this.b.keySet());
  }

  public Collection<AB> getAB() {
```

```
    return Collections.unmodifiableCollection(this.b.values());
  }

  final static AB addAB(A a, B b)
  {
    AB ab = null;
    if (!a.b.containsKey(b))
    {
      ab = new AB();
      a.b.put(b, ab);
      b.a.put(a, ab);
    }

    return ab;
  }

  static boolean removeAB(A a, B b)
  {
    boolean removed;

    if (removed = (b.a.remove(a) != null))
    {
      a.b.remove(b);
    }
    return removed;
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          mutualfriends/associationclass/AB.java
 *
 * Description: The association class AB implemented in pattern
 *          mutual friends.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
```

```
 ************************************************************/
package performance.mutualfriends.associationclass;

public class AB {

}

/************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *      Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          mutualfriends/associationclass/B.java
 *
 * Description: The class B implemented in pattern mutual
 *          friends.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/
package performance.mutualfriends.associationclass;

import java.util.*;

public class B
{
  final HashMap<A, AB> a = new HashMap<A, AB>();

  public AB addA(A a)
  {
    AB ab = null;

    if (a != null)
    {
      ab = A.addAB(a, this);
    }

    return ab;
  }
```

```
  public boolean removeA(A a)
  {
    return a != null && A.removeAB(a, this);
  }

  public Set<A> getB()
  {
    return Collections.unmodifiableSet(this.a.keySet());
  }

  public Collection<AB> getAB() {
    return Collections.unmodifiableCollection(this.a.values());
  }
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          ral/association/A.java
 *
 * Description: The class A implemented in RAL.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/
package performance.ral.association;

public class A {

}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
```

```
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          ral/association/AB.aj
 *
 * Description: The association AB implemented in RAL.
 *
 * Language: AspectJ.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/
package performance.ral.association;

import ral.util.SimpleStaticRel;

public aspect AB extends
   SimpleStaticRel<performance.ral.association.A,
     performance.ral.association.B>
{

}

/************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          ral/association/B.java
 *
 * Description: The class B implemented in RAL.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
```

```
 *       to its suitability for any purpose.
 ************************************************************/
package performance.ral.association;

public class B {

}

/************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          ral/associationclass/A.java
 *
 * Description: The class A implemented in RAL.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/
package performance.ral.associationclass;

public class A {

}

/************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          ral/associationclass/AB.aj
 *
 * Description: The association class AB implemented in RAL.
 *
 * Language: AspectJ.
```

```
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/
package performance.ral.associationclass;

import ral.util.StaticRel;

public aspect AB extends
   StaticRel<performance.ral.associationclass.A,
     performance.ral.associationclass.B,
     performance.ral.associationclass.Link>
{

}

/************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          ral/associationclass/B.java
 *
 * Description: The class B implemented in RAL.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/
package performance.ral.associationclass;

public class B {
```

```
}

/*************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *          ral/associationclass/Link.java
 *
 * Description: The link of association class AB implemented in
 *          RAL.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 *************************************************************/
package performance.ral.associationclass;

import ral.lang.*;
import ral.util.*;

public class Link extends AbstractPair<A, B> {

   public Link(A x, B y) {
     super(x, y);
   }

   @Override
   public Pair<A, B> clone(A f, B t) {
     return new Link(f,t);
   }
}
```

As próximas listagens apresentam o código fonte da aplicação Java e do aspecto principal responsável por executar os testes. Este aspecto principal é necessário para que seja possível fazer chamada as associações em RAL (codificadas em aspectos).

```
/*************************************************************
```

```
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *              Application.java
 *
 * Description: The application that runs the tests in
 *              AspectJ code.
 *
 * Language: Java.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/

package performance;

public final class Application {
  public static void main(String[] args) {
    // execution advice on aspect MainAspect.
  }
}

/************************************************************
 * Association# Language
 * A C# 4 language extension for declaring UML 2 associations.
 * http://www.inf.ufsc.br/~raul/associationsharp/
 *
 * Project: Performance - Performance Test of the Library
 *     Support.
 *
 * Source File: /AspectJ/PerformenaceAspectJ/src/performance/
 *              MainAspect.aj
 *
 * Description: The main aspect, required to run a program
 *              coded in aspects.
 *
 * Language: AspectJ.
 *
 * Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
 * Date: 05/05/2011
 *
```

```
 * Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
 *       Wazlawick 2011. Permission to copy, use, modify,
 *       sell and distribute this software is granted
 *       provided this copyright notice appears in  all
 *       copies. This software is provided "as is" without
 *       express or implied warranty, and with no claim as
 *       to its suitability for any purpose.
 ************************************************************/
package performance;

import java.io.FileWriter;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.GregorianCalendar;

import performance.ral.associationclass.Link;

final aspect MainAspect {

  // execute on application starts.
  before(String[] args) : execution(public static void
    Application.main(String[]))&& args(args)
  {
    MainAspect.main(args);
  }

  private static final FileWriter logger;
  private static final int INTERACTIONS = 100000;
  private static final SimpleDateFormat DATE_FORMAT =
    new SimpleDateFormat("dd/MM/yyyy HH:mm:ss:SS");
  private static final NumberFormat MILLISECONDS_FORMAT =
    new DecimalFormat("######0.0000");

  static {
    try {
      logger = new FileWriter(".\\Performance.log",true);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }

  public static void main(String[] args) {
    if (args.length >= 1)
    {
      switch (Integer.parseInt(args[0]))
      {
        case 1: MFAssociation(); break;
        case 2: MFAssociationClass(); break;
        case 3: RALAssociation(); break;
        case 4: RALAssociationClass(); break;
```

```
        case 5: GeAssociation(); break;
        case 6: GeAssociationClass(); break;
        default: WrongOption(args); break;
      }
    }
    else
    {
      WrongOption(args);
    }
  }

  private static void MFAssociation() {
    log("Iniciando 'Mutual Friends (Java) Association'...");

    long beginning = new GregorianCalendar().getTimeInMillis();

    performance.mutualfriends.association.A a =
      new performance.mutualfriends.association.A();
    ArrayList<performance.mutualfriends.association.B> listB =
      new ArrayList<performance.mutualfriends.association.
        B>();

    for (int i = 0; i < INTERACTIONS; i++)
    {
      performance.mutualfriends.association.B b =
        new performance.mutualfriends.association.B();
      a.addB(b);
      listB.add(b);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.getB();
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.removeB(listB.get(i));
    }

    log("Tempo total " + MILLISECONDS_FORMAT.format(new
      GregorianCalendar().getTimeInMillis() - beginning)+
      " ms");
  }

  private static void MFAssociationClass() {
    log(
    "Iniciando 'Mutual Friends (Java) AssociationClass'...");

    long beginning = new GregorianCalendar().getTimeInMillis();

    performance.mutualfriends.associationclass.A a =
      new performance.mutualfriends.associationclass.A();
    ArrayList<performance.mutualfriends.associationclass.B>
```

```
      listB = new ArrayList<performance.mutualfriends.
        associationclass.B>();

    for (int i = 0; i < INTERACTIONS; i++)
    {
      performance.mutualfriends.associationclass.B b =
        new performance.mutualfriends.associationclass.B();
      a.addB(b);
      listB.add(b);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.getB();
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.getAB();
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.removeB(listB.get(i));
    }

    log("Tempo total " + MILLISECONDS_FORMAT.format(
      new GregorianCalendar().getTimeInMillis() - beginning)
      + " ms");
  }

  private static void RALAssociation() {
    log("Iniciando 'RAL Association'...");

    long beginning = new GregorianCalendar().getTimeInMillis();
    performance.ral.association.AB ab =
      performance.ral.association.AB.aspectOf();

    performance.ral.association.A a =
      new performance.ral.association.A();
    ArrayList<performance.ral.association.B> listB =
      new ArrayList<performance.ral.association.B>();

    for (int i = 0; i < INTERACTIONS; i++)
    {
      performance.ral.association.B b =
        new performance.ral.association.B();
      ab.add(a,b);
      listB.add(b);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      ab.from(a);
```

```
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      ab.remove(a,listB.get(i));
    }

    log("Tempo total " + MILLISECONDS_FORMAT.format(new
      GregorianCalendar().getTimeInMillis() - beginning) +
      " ms");
  }

  private static void RALAssociationClass() {
    log("Iniciando 'RAL AssociationClass'...");

    long beginning = new GregorianCalendar().getTimeInMillis();
    performance.ral.associationclass.AB ab =
      performance.ral.associationclass.AB.aspectOf();


    performance.ral.associationclass.A a =
      new performance.ral.associationclass.A();
    ArrayList<performance.ral.associationclass.B> listB =
      new ArrayList<performance.ral.associationclass.B>();

    for (int i = 0; i < INTERACTIONS; i++)
    {
      performance.ral.associationclass.B b =
        new performance.ral.associationclass.B();
      ab.add(new Link(a,b));
      listB.add(b);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      ab.from(a);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      ab.fromPairs(a);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      ab.remove(a,listB.get(i));
    }

    log("Tempo total " + MILLISECONDS_FORMAT.format(new
      GregorianCalendar().getTimeInMillis() - beginning) +
      " ms");
  }

  private static void GeAssociation() {
```

```
    log("Iniciando 'Gessenharter (Java) Association'...");

    long beginning = new GregorianCalendar().getTimeInMillis();

    performance.gessenharter.association.A a =
      new performance.gessenharter.association.A();
    ArrayList<performance.gessenharter.association.B> listB =
      new ArrayList<performance.gessenharter.association.B>();

    for (int i = 0; i < INTERACTIONS; i++)
    {
      performance.gessenharter.association.B b =
        new performance.gessenharter.association.B();
      a.addB(b);
      listB.add(b);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.getB();
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.removeB(listB.get(i));
    }

    log("Tempo total " + MILLISECONDS_FORMAT.format(new
      GregorianCalendar().getTimeInMillis() - beginning) +
      " ms");
  }

  private static void GeAssociationClass() {
    log("Iniciando 'Gessenharter (Java) AssociationClass'...");

    long beginning = new GregorianCalendar().getTimeInMillis();

    performance.gessenharter.associationclass.A a =
      new performance.gessenharter.associationclass.A();
    ArrayList<performance.gessenharter.associationclass.B>
      listB = new ArrayList<performance.gessenharter.
        associationclass.B>();

    for (int i = 0; i < INTERACTIONS; i++)
    {
      performance.gessenharter.associationclass.B b =
        new performance.gessenharter.associationclass.B();
      a.addB(b);
      listB.add(b);
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.getB();
```

```
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.getAB();
    }

    for (int i = 0; i < INTERACTIONS; i++)
    {
      a.removeB(listB.get(i));
    }

    log("Tempo total " + MILLISECONDS_FORMAT.format(new
      GregorianCalendar().getTimeInMillis() - beginning)
      + " ms");
  }

  private static void log(String text) {
    String string = DATE_FORMAT.format(new Date())+" - "+text;
    try {
      logger.write(string+"\r\n");
      logger.flush();
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
    System.out.println(string);
  }

  private static void WrongOption(String[] options) {
    System.out.print("Opção inválida:");
    for(String option : options) {
      System.out.print(" "+option);
    }
    System.out.println();
  }
}
```

A próxima listagem apresenta o script utilizado pelo ANT para compilar e gerar o JAR executável com todos os fontes AspectJ/Java listados.

```
<!--
 Association# Language
 A C# 4 language extension for declaring UML 2 associations.
 http://www.inf.ufsc.br/~raul/associationsharp/

   Project: Performance - Performance Test of the Library
       Support.

   Source File: /AspectJ/PerformenaceAspectJ/build.xml

   Description: The ANT Script to build Performance Test for
            AspectJ.
```

```
   Language: ANT Script.

   Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
   Date: 05/05/2011

   Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
              Wazlawick 2011. Permission to copy, use, modify,
              sell and distribute this software is granted
              provided this copyright notice appears in  all
              copies. This software is provided "as is" without
              express or implied warranty, and with no claim as
              to its suitability for any purpose.
-->

<project default="create_run_jar">
  <target name="init.tasks"
depends="init.vars,aspectjtools.jar.available,aspectjrt.jar.ava
ilable,ral.jar.available"
          unless="tasks.init"
  >
    <taskdef
resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.proper
ties"
    >
      <classpath>
        <pathelement path="${aspectjtools.jar}" />
        <pathelement path="${ral.jar}" />
      </classpath>
    </taskdef>
    <property name="tasks.init" value="true" />
  </target>

  <target name="clean" depends="init">
    <delete>
      <fileset dir="./bin" includes="**/*.class" />
      <fileset file="PerformanceJava.jar" />
    </delete>
  </target>

  <target name="init" depends="init.vars,init.tasks" />

  <target name="init.vars">
    <property environment="env" />
    <property name="aspectj.lib.dir"
              location="${env.ASPECTJ_HOME}/lib"
    />
    <property name="aspectjrt.jar"
              location="${aspectj.lib.dir}/aspectjrt.jar"
    />
    <property name="aspectjtools.jar"
              location="${aspectj.lib.dir}/aspectjtools.jar"
    />
    <property name="ral.jar"
              location="../../../bin/lib/ral.jar"
```

```
    />
    <available file="${aspectjtools.jar}"
               property="aspectjtools.jar.available"
    />
    <available file="${aspectjrt.jar}"
               property="aspectjrt.jar.available"
    />
    <available file="${ral.jar}"
               property="ral.jar.available"
    />
  </target>

  <target name="aspectjrt.jar.available"
          depends="init.vars"
          unless="aspectjrt.jar.available"
  >
    <fail message="Aspectjrt not found at ${aspectjrt.jar}" />
  </target>

  <target name="aspectjtools.jar.available"
          depends="init.vars"
          unless="aspectjtools.jar.available"
  >
    <fail message="Aspectjtools not found at
${aspectjtools.jar}"
    />
  </target>
  <target name="ral.jar.available"
          depends="init.vars"
          unless="ral.jar.available"
  >
    <fail message="RAL not found at ${ral.jar}" />
  </target>

  <target name="build" depends="clean">
    <iajc source="1.5"
          sourceroots="./src/performance"
          destdir="./bin"
    >
      <classpath>
        <pathelement path="${aspectjtools.jar}" />
        <pathelement path="${ral.jar}" />
      </classpath>
    </iajc>
  </target>

  <target name="create_run_jar" depends="build">
    <jar destfile="../../../bin/PerformanceJava.jar">
      <manifest>
        <attribute name="Main-Class"
                   value="performance.Application"
        />
        <attribute name="Class-Path"
                   value=". lib/aspectjrt.jar lib/ral.jar"
```

```
        />
      </manifest>
      <fileset dir="./bin" />
    </jar>
    <copy file="${aspectjrt.jar}" todir="../../../bin/lib" />
  </target>
</project>
```

A próxima listagem apresenta o arquivo de lote *Performance.bat*, responsável por executar os doze testes (seis em C# e seis em Java).

```
@REM ===========================================================
@REM Association# Language
@REM A C# 4 language extension for declaring UML 2 associations
@REM http://www.inf.ufsc.br/~raul/associationsharp/

@REM Batch File: <root>/bin/Performance.bat

@REM Description: The ANT Script to build Performance Test for
@REM AspectJ.

@REM Language: ANT Script.

@REM Author: Iuri Sônego Cardoso - iuricardoso@gmail.com
@REM Date: 05/05/2011

@REM Copyright: Copyright Iuri Sônego Cardoso and Raul Sidnei
@REM          Wazlawick 2011. Permission to copy, use, modify,
@REM          sell and distribute this software is granted
@REM          provided this copyright notice appears in  all
@REM          copies. This software is provided "as is" without
@REM          express or implied warranty, and with no claim as
@REM          to its suitability for any purpose.
@REM ===========================================================
Performance.exe 1
Performance.exe 2
Performance.exe 3
Performance.exe 4
Performance.exe 5
Performance.exe 6
java -jar PerformanceJava.jar 1
java -jar PerformanceJava.jar 2
java -jar PerformanceJava.jar 3
java -jar PerformanceJava.jar 4
java -jar PerformanceJava.jar 5
java -jar PerformanceJava.jar 6
start notepad.exe .\performance.log
```