Universidade Federal de Pernambuco
Graduação em Ciência da Computação
Centro de Informática

Trabalho de Graduação

INTEGRANDO UML E MÉTODOS FORMAIS

Rafael Magalhães Borges

Orientador: Alexandre Cabral Mota
Co-orientador: Augusto Cézar Alves Sampaio

Recife
Setembro de 2004

Eu queria ver no escuro do mundo
Aonde está o que você quer
Pra me transformar no que te agrada
No que me faça ver

Quais são as cores e as coisas pra te prender?
Eu tive um sonho ruim e acordei chorando
Por isso eu te liguei

Será que você ainda pensa em mim
Será que você ainda pensa

Às vezes te odeio por quase um segundo
Depois te amo mais
Teus pêlos, teu rosto, teu gosto, tudo
Tudo que não me deixa em paz

Quais são as cores e as coisas pra te prender?
Eu tive um sonho ruim e acordei chorando
Por isso eu te liguei

Será que você ainda pensa em mim
Será que você ainda pensa

—HERBERT VIANNA (Quase um Segundo)

A meus pais
A Vanessa

AGRADECIMENTOS

"Make it a habit to tell people thank you. To express your appreciation, sincerely and without the expectation of anything in return. Truly appreciate those around you, and you'll soon find many others around you. Truly appreciate life, and you'll find that you have more of it".

—RALPH MARSTON

A meus pais, agentes proporcionadores da minha formação.

A Vanessa, minha namorada, pela enorme compreensão.

A meus orientadores e, acima de tudo, amigos, Alexandre e Augusto, pelos importantes e duradouros ensinamentos.

Aos meus amigos de curso, em especial, Rafael, Alessandro, Alexandra, Cynthia, Filipe, Ivan, Júlio, Marcus, Mauro, Paulo e Pedro, pelo companheirismo.

Ao grupo PET, sob o tutorado do prof. Fernando, pela experiência de ensino, pesquisa e extensão.

Aos companheiros do ForMULa, especialmente Allan, Joabe e Rodrigo, pelas enriquecedoras discussões.

Ao prof. Ruy e ao prof. Ayrton (do CEFET), por fomentarem meu gosto pela Matemática e pela Lógica.

Aos meus mestres no CIn, pelo conhecimento adquirido.

E a todos que contribuíram, direta ou indiretamente, para a conclusão deste trabalho, meus sinceros agradecimentos.

RESUMO

UML é uma linguagem bastante difundida na indústria e nas universidades. Entretanto, sua semântica ainda é informal e possui ambigüidades. Por outro lado, *OhCircus* é uma linguagem de especificação formal que unifica as teorias de Z, CSP, cálculo de refinamentos e orientação a objetos. Neste trabalho propomos como contribuição inicial a *integração* entre diagramas de classes UML e especificações *OhCircus*, através da tradução da primeira via a segunda. Em particular, nossa abordagem utiliza, além do mapeamento sintático, o conceito de classe *modelo* para capturar, naturalmente, associações e restrições globais. Além disso, alcançamos uma outra contribuição quando utilizamos nosso próprio trabalho para provar o refinamento de associações em atributos, um ponto importante entre a caracterização do nível de abstração entre modelos UML, desde análise, passando por projeto, até implementação.

Palavras-chave: UML, *OhCircus*, integração, mapeamento, diagrama de classes, refinamento

ABSTRACT

UML is a widespread language in industry and academia. However its semantics is still informal and has ambiguities. On the other hand, *OhCircus* is a formal specification language which unifies Z, CSP, the refinement calculus and object-oriented theories. In this work, we are concerned with their *integration*, translating UML class diagrams into *OhCircus* specifications. In particular, our approach uses, beyond the syntactic mapping, the concept of class *model* to capture, naturally, associations and global constraints. Finally, we use our own work to prove the refinement of associations as attributes, a much desired result linking analysis to design and implementation.

Keywords: UML, *OhCircus*, integration, mapping, class diagram, refinement

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

If this is true, building software will always be hard.
There is inherently no silver bullet.

—FRED BROOKS, JR.

O desenvolvimento de software enfrenta, desde a década de 1960, o desafio de produzir software segundo os mesmos referenciais das demais engenharias. Porém, diversos projetos são cancelados e outros tantos estouram custos e prazos sem atingir uma qualidade aceitável. Este problema se agrava quando falamos dos sistemas críticos, que envolvem elevadas somas em dinheiro ou vidas humanas.

Em junho de 1996, o foguete do projeto *Ariane 5* explodiu no ar 40 segundos após seu lançamento. A investigação detectou um erro de especificação e projeto, onde as conseqüências de desabilitar a proteção do sistema não foram devidamente avaliadas. Esta falha custou 7 bilhões de dólares e 10 anos à agência espacial européia [JM97].

Entre 1985 e 1987, o Therac-25, um dispositivo para tratamento oncológico, aplicou doses radioativas letais em seis pacientes. Investigações apontam o seu software como a causa do problema, uma vez que ele era o maior responsável pela segurança do sistema (o Therac-25, ao contrário dos demais membros de sua família, não possuía mecanismos de proteção duplicados e independentes). Negligências durante os testes unitários permitiram que erros no software passassem despercebidos [LT93].

As mais diversas causas são atribuídas a esta crise. Alguns trabalhos [Gib94, Bro95, Bro87] mais pragmáticos sugerem, por exemplo, a instabilidade dos requisitos. Um outro [Hoa84], mais filosófico, compara os desenvolvedores atuais aos artesãos pré-industriais: ambos produzem seus artefatos utilizando técnicas baseadas no empirismo, desconhecendo a ciência por trás dos seus ofícios. Dijkstra vai além: sugere não só o embasamento, mas também a verificação formal dos programas [Dij72]. Porém é importante ressaltar que todos concordam que produzir software de qualidade (onde um dos maiores desafios é a corretude) é essencial.

Diversas foram as "balas de prata" propostas para resolver este desafio: a introdução de linguagens de programação de alto-nível, a aplicação de

análise estruturada e métodos de projeto, o uso de ferramentas para auxiliar o desenvolvimento (ferramentas CASE) etc. Entretanto, Brooks [Bro87] afirma que essas tecnologias, apesar de trazerem algum progresso, resolvem apenas a complexidade *acidental*[1] do software. Hoje, esta crise é tomada como crônica[2].

1.1 ENGENHARIA DE SOFTWARE

Todo software possui um *ciclo de vida* associado, dividido em três macro fases: o *processo de desenvolvimento*, responsável por construir o software, a *operação*, comprometida com a utilização da aplicação pelos usuários, e finalmente, a *manutenção*, interessada em promover atualizações no software. É fácil perceber que a primeira fase é a mais importante de todo o ciclo; se esta não for realizada com um mínimo de sucesso, o software será pouco ou mal utilizado, e sua manutenção será extremamente difícil.

A Engenharia de Software é justamente a disciplina que tenta abordar o problema de desenvolver software sistematicamente, melhorando a previsão de custos e prazos e construindo-o com qualidade aceitável, proporcional ao investimento. O uso de métodos, ferramentas, documentações e métricas adequados contribuem para o sucesso nesta empreitada.

Porém, neste trabalho, procuramos ressaltar a importância de utilizar uma notação adequada durante o processo de desenvolvimento. Em particular, a *Unified Modeling Language* (UML) merece destaque: uma notação proposta pela OMG que unifica diversas outras existentes. Suas principais características incluem expressividade, para modelar sistemas de propósito geral, abrangência, para ser utilizada em todas as etapas do processo, e simplicidade, para o entendimento dos usuário leigos, dos clientes e dos desenvolvedores. Hoje, graças a essas características, UML é o padrão de mercado.

É importante ressaltar que, apesar de toda essa sistemática, a Engenharia de Software oferece apenas *direções* de como produzir software. Infelizmente, a experiência com técnicas informais mostra que isso não oferece garantias, especialmente quando falamos de sistemas grandes, complexos e críticos.

Métodos Formais

Métodos Formais surgiram como uma possível solução para a Engenharia de Software e sua crise. Estabelecer a Matemática como alicerce do desenvol-

[1] Complexidade acidental contempla aspectos de produtividade: usar uma linguagem de programação de alto-nível agiliza a produção de código, mas o software não fica menos complicado por isso.

[2] Esta afirmação apenas estabelece que o software é, inerentemente, complexo; não há uma técnica miraculosa capaz de tornar seu desenvolvimento simples.

vimento de software dá rigor científico a diversos conceitos outrora informais; é agora possível verificar formalmente a corretude dos programas através de provas.

A idéia por trás do desenvolvimento formal é simples: especificar sistemas utilizando estruturas matemáticas e abstraindo detalhes operacionais permite um maior entendimento do problema, enquanto as provas de propriedades destes modelos passam a ser mais intuitivas e relativamente fáceis. Depois, a substituição dessa especificação por outras mais concretas, com estruturas de dados cada vez mais computacionais e menos abstratas, garante as propriedades do modelo original [WD96]. Por fim, a aplicação de transformações (ou leis) matemáticas leva-nos à derivação de uma implementação que, por construção, está correta [Mor94, Dij97].

Em particular, a prova de propriedades teria ajudado bastante o *Ariane 5*. Se fosse verificada a ausência de *deadlocks* no foguete, ele poderia não ter explodido. Melhor ainda se uma ferramenta como o FDR [Gol01] estivesse disponível: a detecção da seqüência de eventos que culminaria no *deadlock* seria menos complicada.

O exemplo do CICS [WD96, Mot97] ilustra muito bem o emprego de métodos formais na indústria. O CICS é uma família de softwares da IBM para gerenciamento e processamento de transações. Ele permite o acesso simultâneo de milhões de usuários do mundo inteiro e deve trabalhar continuamente, sem interrupções. Devido à complexidade do sistema, o uso de métodos formais (em particular, Z [Spi92]) foi imprescindível para reestruturar todo o sistema, garantindo-lhe a confiança almejada. Apesar do esforço durante a formalização, as vantagens tornaram-se evidentes nas fases subseqüentes e em seus respectivos custos.

Contudo, mesmo com casos de sucesso como este, o do TCAS II [Hei96] e outros [BH95], Métodos Formais ainda não são utilizados em larga escala na indústria. Como destacam [CW96, Som02], isso se deve, dentre outros fatores, à barreira imposta pela sua forte notação matemática. Porém, é importante frisar que nenhum método foi universalmente aceito até hoje.

1.2 ESTADO DA ARTE

O estado da arte na área de Métodos Formais busca resolver as principais deficiências encontradas (em particular, a da forte notação matemática) quanto ao seu uso industrial, tornando-se assim mais acessíveis. Duas alternativas têm sido as mais exploradas atualmente: a simplificação da linguagem formal, o que inclui até a adição de elementos gráficos, como Alloy [Jac02], ou a utilização de uma outra linguagem, geralmente informal, mapeando suas construções para outra formal, como os diversos trabalhos que exploram UML

e Z ou UML e Object-Z [MBM03, RBR03, Zep02, BHH+97, KC00].

Apesar dessa primeira alternativa possuir seus méritos [GB03, Ghe04], acreditamos que a segunda é mais promissora. Podemos assim utilizar, como intermediárias, as linguagens com as quais o desenvolvedor já está habituado e mapeá-las numa linguagem formal poderosa, concebida sem maiores restrições conceituais, como executabilidade (embora sejam necessárias algumas para a utilização prática).

Como notação informal, a UML [OMG03a, OMG03c] merece destaque especial. Ela utiliza elementos gráficos para representar as diversas entidades do software assim como seus relacionamentos. E graças a sua aparente simplicidade e facilidade de entendimento, tornou-se padrão de mercado. Porém, pode expressar ambigüidades e é insuficiente para representar até propriedades mais simples [OMG03b].

E na direção de mapear uma notação informal em uma formal, os trabalhos mais influentes na área são os do grupo *Precise* UML [FEL97, EC97]. Eles traduzem as meta-construções de UML para esquemas em Z[3] e os diagramas representam valores desses esquemas (em outros termos, oferecem uma semântica denotacional utilizando Z). Sob a nossa visão, essa abordagem não é interessante, pois trata UML em um nível semântico diferente daquele da linguagem formal.

Descartada esta possibilidade, resta-nos apenas trabalhar com mapeamentos onde a notação formal esteja no mesmo nível semântico de UML. Nossa escolha envolveu alguns critérios bastante simples: obter uma representação mais intuitiva de construções como classes e possibilitar a utilização de refinamento. Utilizar Z, por exemplo, seria inadequado por necessitar expressar de uma forma não muito intuitiva algumas construções inerentes ao modelo orientado a objetos, como classes e herança. Já Object-Z [Smi00], em algumas situações, não permite refinamento passo-a-passo[4]. Mais ainda: Object-Z permite que uma subclasse não seja sequer um subtipo da superclasse[5] [CSW03].

Portanto, dados esses critérios, optamos por *OhCircus* [CSW03], uma linguagem que integra conceitos bem estabelecidos na comunidade formal: a linguagem baseada em modelos Z [Spi92], a álgebra de processos CSP [RHB97] e o cálculo de refinamentos [Mor94], além dos conceitos de orientação a objetos,

[3]Esquemas são semelhantes a registros em linguagens de programação, exceto pela possibilidade de incluir restrições sobre os valores.

[4]Refinamento passo-a-passo (*stepwise refinement*) estabelece que a substituição de um componente de especificação por outro (que seja um refinamento seu) no mesmo contexto preserva o comportamento.

[5]Object-Z permite que uma subclasse renomeie, redefina ou cancele operações da superclasse.

provendo uma linguagem unificada para classes e processos. Vale destacar que algumas idéias foram obtidas a partir de UML-RT[6] [Lyo98, SR98], o que a torna ainda mais apropriada para o mapeamento almejado.

Atualmente, essa integração entre notações formais e de mercado é uma área de pesquisa bastante ativa, permitindo que as práticas da Engenharia de Software sejam provadas formalmente, enquanto os resultados dos Métodos Formais são aplicados na indústria. Em particular, *hidden formal methods* procuram disponibilizar estes resultados para o desenvolvedor sem que ele perceba o uso de Métodos Formais.

1.3 OBJETIVOS

O objetivo deste trabalho é capturar os principais elementos que constituem os diagramas de classes anotados de UML e mapeá-los em especificações *OhCircus*. Este é um tópico de pesquisa bastante ativo [BHH+97, LB98, Pai99, KC00, Zep02, FG03, RRS04, LCA04], mas nossa abordagem difere das demais porque utiliza uma linguagem projetada para acomodar várias construções de UML(-RT) e procura preservar a estrutura do diagrama de classes. Vale destacar que a semântica de *OhCircus* ainda está incompleta; porém, como as vantagens de utilizá-la superam as desvantagens, esperamos que esse trabalho seja uma contribuição para a evolução da própria linguagem.

Como discutido anteriormente, UML é por si só insuficiente para capturar todas as propriedades relevantes de um sistema [OMG03b], sendo também necessárias anotações (invariantes, pré- e pós-condições *etc*). Por outro lado, a linguagem proposta pela OMG para suprir este papel, OCL [OMG03b], é limitada: apenas especifica restrições (semelhantes a asserções) e ainda não possui semântica formal bem definida. Neste trabalho, consideramos algumas características e construções dessa linguagem, mas *OhCircus* será utilizada nas anotações.

Além de tratar dos elementos individuais do diagrama de classes (as próprias classes), também preservamos toda a sua estrutura, como relacionamentos, invariantes globais e aspectos dinâmicos do sistema (via *history model* [Smi92, RJB99]). Isto é realizado através de uma (meta-)classe, sintaticamente equivalente às demais, que captura tal estrutura (classe *modelo*). A principal motivação para isso é explorar refinamento em UML [SMR03].

Por fim, diversos trabalhos [BHH+97, Eva98, LB98], inclusive [OMG03c], tomam como verdade a noção de equivalência entre associações e atributos. Através da nossa abordagem, propomos utilizar as associações numa

[6] UML-RT é uma extensão de UML que contempla concorrência.

visão abstrata do diagrama de classes, sendo eliminadas ao longo do refinamento, introduzindo atributos nas classes que participam desta associação. O objetivo principal é, além do próprio resultado, dar uma intuição de (e possivelmente consolidar) o mapeamento e a noção de classe modelo.

1.4 VISÃO GERAL

Estruturamos este trabalho da seguinte maneira: no próximo capítulo, descrevemos a notação UML. Discutimos cada um dos conceitos abordados ao longo do trabalho e procuramos estabelecer claramente a semântica de cada construção, embora informalmente, baseada nos documentos de referência da OMG [OMG03a, OMG03c]. No capítulo 3, apresentamos *OhCircus*, a linguagem formal orientada a objetos que será utilizada como alicerce de nosso trabalho.

A primeira contribuição deste trabalho será introduzida no capítulo 4, onde apresentamos o mapeamento de um diagrama de classes em UML para uma especificação *OhCircus*, discutindo seus respectivos méritos. É também nesse capítulo que descrevemos o conceito de classe *modelo*, a (meta-) classe que preserva a estrutura e propriedades do sistema.

A segunda contribuição aparece no capítulo 5, onde procuramos tratar dos aspectos de refinamento em diagramas UML. Em particular, procuramos abordar um dos refinamentos mais utilizados pela indústria: o mapeamento de associações em atributos.

Por fim, no capítulo 6, apresentamos as nossas conclusões sobre o trabalho. Também serão discutidos trabalhos futuros relacionados.

É importante ressaltar também que este trabalho será permeado por um grande exemplo. Ele será introduzido no capítulo 2, onde sua semântica será discutida. Uma possível representação dele será apresentada no capítulo 3. Por último, no capítulo 4, procuramos construir sua classe modelo e respectivo mapeamento em *OhCircus*.

CAPÍTULO 2

UML

A *Unified Modeling Language* (UML) é a linguagem proposta pela OMG (*Object Management Group*) que reúne as principais características das notações dos métodos Booch [Boo91], OMT [RBP+91] e OOSE [JCJv92]. Tornou-se o padrão de mercado por ser intuitiva e fácil de usar.

UML é amplamente utilizada para modelar sistemas. Um modelo de UML representa a descrição do conjunto de objetos que fazem parte da aplicação e de interações as quais eles estão sujeitos ao longo do tempo. Cada um desses conjuntos constitui uma *configuração* do sistema, e a coleção de todas as configurações possíveis denota a *semântica* do modelo. Assim, um modelo pode ser visto como a *intensão* do sistema.

UML é dotada de vários diagramas que permitem expressar os aspectos estáticos e dinâmicos de uma aplicação. Aspectos estáticos estão relacionados à estrutura do sistema e independem do tempo. O objetivo é descrever as entidades de um sistema e como elas *sempre* vão interagir. Por outro lado, os aspectos dinâmicos dizem respeito à evolução da aplicação. O foco é contemplar a criação e destruição de objetos e interações ao longo do tempo; em outros termos, as *transformações* às quais o estado global do sistema está sujeito.

A Figura 2.1 introduz um sistema bancário bastante simplificado. Nele, estabelecemos que pessoas possuem contas. Pessoas e contas estão representadas, respectivamente, pelas entidades *Person* e *Account*. Já a propriedade de posse é modelada através do relacionamento *owns*. O diagrama simboliza a *estrutura* dessa aplicação utilizando UML.

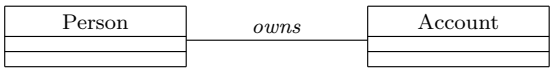

Figura 2.1. Aspectos estáticos.

Os diagramas da Figura 2.2 exemplificam configurações válidas para a aplicação bancária do exemplo anterior. No primeiro, *charles* possui a conta *a123*, enquanto no segundo *charles* e *alice* são titulares da mesma conta (uma conta conjunta). O conjunto de todas as configurações válidas expressa a semântica deste modelo.

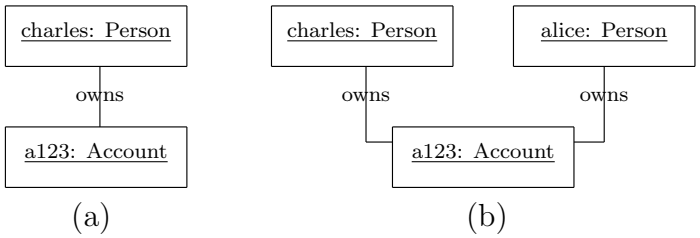

Figura 2.2. Aspectos dinâmicos.

Considerando que este sistema possui as operações (ou transformações) *CadastrarPessoa*, que adiciona uma pessoa ao nosso sistema, e *IncluirTitular*, que inclui um novo titular a uma conta, aplicar *CadastrarPessoa(alice)* seguida de *IncluirTitular(alice, a123)* ao estado simbolizado pelo diagrama da Figura 2.2 (a) transforma-o no da Figura 2.2 (b). A sucessão de estados a qual um sistema pode ser submetido denota os seus aspectos dinâmicos.

Neste capítulo, apresentamos as principais construções que descrevem os aspectos estáticos de uma aplicação. Inicialmente, discutimos os níveis de abstração de UML e como eles denotam uma semântica. Posteriormente, descrevemos a visão estática através de diagramas de classes e sua extensão, os diagramas de objetos. Nas seções seguintes, introduzimos classificadores e relacionamentos. Por fim, sugerimos uma semântica para os diagramas de classes.

Este capítulo é baseado nas especificações de UML 2.0 [OMG03a, OMG03c], UML 1.5 [OMG03d] e no *Manual de Referência* UML [RJB99].

Níveis de abstração

UML é apenas uma das quatro camadas da arquitetura de metamodelagem. Esta arquitetura é interessante porque busca lidar com a complexidade inerente a grandes linguagens. É importante observar o relacionamento entre cada uma dessas camadas, onde as mais abstratas oferecem uma infraestrutura para as mais concretas. A relação entre os níveis de abstração pode ser observada na Figura 2.3.

Meta-metamodelo (M3). O meta-metamodelo foi proposto pela OMG para lidar com a complexidade de UML. Em particular, esta abordagem permitiu estabelecer um alicerce sobre o qual possíveis extensões de UML e até mesmo novos metamodelos pudessem ser construídos. O MOF (*Meta Object Facility*) é quem provê os blocos básicos para UML. Por exemplo, todos os construtores de entidades de UML são agrupados em um única construção de MOF, a

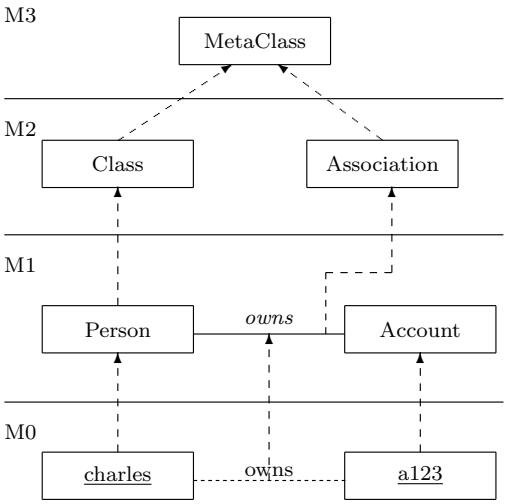

Figura 2.3. Relação entre os níveis de abstração

MetaClass.

Metamodelo (M2). Um metamodelo é uma instância de um meta-metamodelo. De fato, UML é um metamodelo por ser uma instância do MOF. UML define a linguagem com a qual podemos construir modelos. Conceitos, como pessoas ou contas, são agrupados numa única construção, *Class*. Já os relacionamentos entre os conceitos são agrupados em *Association*. Ambos introduzem novas entidades no modelo e por isso são instâncias da *MetaClass*.

Modelo (M1). Utilizamos um modelo para expressar entidades presentes no domínio semântico da aplicação; de fato, ele representa uma abstração do sistema. Nesta camada, definimos as regras, ou a *intensão*, do nosso sistema. Por exemplo, a Figura 2.1 representa o nosso modelo de aplicação bancária, com suas entidades e relacionamentos.

Objetos (M0). Os objetos e suas interações representam os componentes básicos de qualquer sistema. Estas entidades são descritas no modelo, mas representam a informação propriamente dita (a instância) na aplicação. Os diagramas da Figura 2.2 exibem os objetos *charles*, *alice* e *a123* e suas interações *owns*.

Esta relação entre os níveis de abstração estabelece que o conjunto de todas as possíveis instâncias válidas da linguagem introduzida por um nível pode ser visto como a sua semântica. Por exemplo, na Figura 2.3, podemos

ver o conjunto de todas as possíveis situações onde os objetos de *Person* e *Account* interagem adequadamente (M0) como a semântica do modelo proposto em M1. Todas as possíveis situações onde classes e associações são utilizados corretamente para construir modelos (M1) podem ser vistas como a semântica de UML (M2). De fato, UML é uma das possíveis configurações válidas das entidades de MOF (M3).

2.1 DIAGRAMA DE CLASSES

Os diagramas de classes são os diagramas mais utilizados nos projetos de desenvolvimento de software. Eles modelam os conceitos do domínio da aplicação e os aspectos estruturais do sistema utilizando classificadores e relacionamentos como seus blocos básicos. São também chamados de *visão estática* por representar informações que independem do tempo.

O exemplo a seguir foi adaptado daqueles de [OMG03b, CSW03] e será utilizado ao longo de todo este trabalho. É um exemplo de sistema bancário que possui alguns detalhes omitidos e incrementado com relações de emprego. Embora pareça estranho, também adicionamos a este modelo uma associação de matrimônio, para contemplar mais alguns aspectos de UML. Apesar de não ser exaustivo, ele ilustra as principais construções presentes em diagramas de classes.

Na Figura 2.4, podemos visualizar o diagrama de classes referente a este sistema bancário. Este diagrama introduz, dentre outras, as entidades *Person*, *Bank* e *Account*. Cada uma dessas entidades representa um conjunto de valores e por isso são chamadas de classificadores. Porém, tais entidades não existem sozinhas; elas necessitam interagir. Assim, alguns relacionamentos são introduzidos entre as entidades, como a associação *owns* entre os classificadores *Person* e *Account*. Outro exemplo interessante é o relacionamento entre duas *instâncias* (ou valores) de *Person* através de *marriage*.

Diagrama de objetos. Os diagramas de objetos exibem as instâncias das entidades de um diagrama de classes particular. Eles representam o estado do sistema em um dado momento e devem satisfazer às restrições impostas pelo modelo.

O diagrama de objetos simbolizado na Figura 2.5 aponta uma configuração possível (com alguns detalhes omitidos) do diagrama de classes do estudo de caso. Neste nível, estamos tratando das *instâncias* das entidades introduzidas pelo diagrama anterior. Por exemplo, *alice* e *charles* são *objetos* (ou instâncias) de *Person*, enquanto o *link* (denominado *marriage*) entre *alice* e *charles* representa um exemplar do relacionamento *marriage* entre duas entidades *Person*.

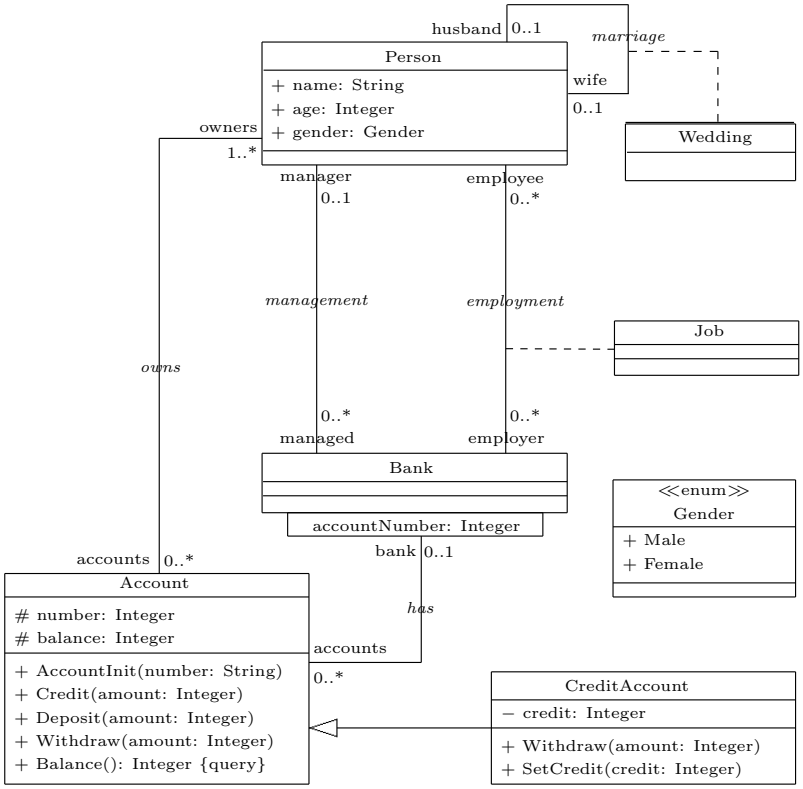

Figura 2.4. Diagrama de classes do estudo de caso.

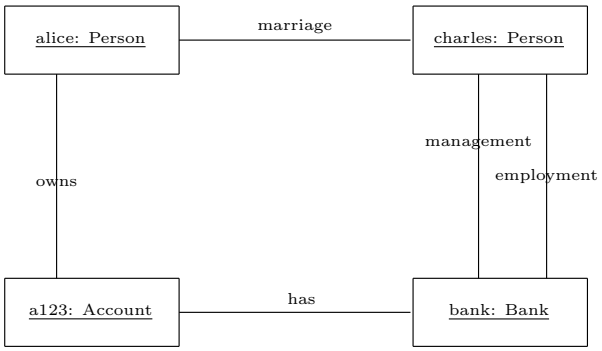

Figura 2.5. Um possível diagrama de objetos do estudo de caso.

Maiores detalhes sobre cada uma dessas construções serão apresentados em suas respectivas seções.

2.2 CLASSIFICADORES

Classifier é uma meta-classe (de MOF) que agrupa todas as construções que classificam valores. Além de introduzir entidades, os classificadores também possuem *membros*, que declaram uma característica comportamental ou estrutural. Suas principais subclasses são *Class* e *DataType*. As demais subclasses (como *Interface*) não serão contempladas aqui.

A notação empregada para introduzir um classificador é um retângulo. Este retângulo possui diversos compartimentos nos quais seus membros são especificados. O primeiro compartimento é reservado ao nome do classificador. Outros compartimentos podem ser introduzidos por suas instâncias.

Classes

Classes são os elementos-chave em um diagrama de classes. Elas representam um conceito dentro do sistema e introduzem tipos no modelo. Descrevem a estrutura e o comportamento de um conjunto de objetos através de atributos, associações e métodos. Na Figura 2.6 podemos observar a classe *Account*.

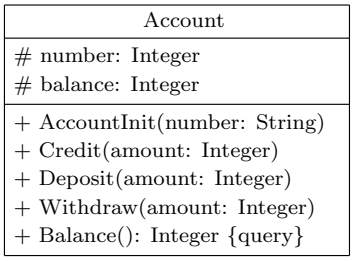

Figura 2.6. Classe *Account*.

Os atributos descrevem valores que os objetos de uma classe contêm. Todo atributo possui um tipo associado e pode, de acordo com sua multiplicidade, conter um, mais de um ou até mesmo nenhum valor. Também pode ser especificado um valor inicial. A classe *Account* da Figura 2.6 possui dois atributos: *number* e *balance*. Cada um deles contém um único valor do tipo *Integer*.

Os métodos de uma classe representam a implementação de operações. Eles incorporam uma transformação no estado do objeto a partir do qual foram invocados. Possuem uma lista de parâmetros e um tipo de retorno. Um

método pode ser definido (através de tags) como *query*, se não efetua modificações no estado do objeto, ou como *constructor* (utilizando estereótipos) se representa um construtor. Na Figura 2.6, a classe *Account* possui cinco métodos. *Credit*, *Deposit* e *Withdraw* realizam as respectivas operações de crédito, depósito e saque, recebendo o valor como parâmetro e modificando o saldo adequadamente. *Balance* simplesmente retorna o saldo atual, sem modificar o estado da conta (método *query*). Por fim, *AccountInit* possui o estereótipo *constructor* (embora não esteja explícito na figura), indicando que este método só será executado durante a criação de um objeto, provendo seu estado inicial.

Em UML também é possível *especificar* os métodos: descrever as condições sob as quais o método vai executar com sucesso (pré-condições), bem como o estado do sistema após a sua execução (pós-condições). Especificações podem ser dadas tanto em linguagem natural quanto em algum formalismo particular. No nosso caso, utilizamos *OhCircus* [CSW03].

Visibilidade é uma propriedade comum a atributos e métodos (os membros de uma classe) que estabelece a acessibilidade do membro em relação às demais entidades. Membros privados (simbolizados por −) são visíveis apenas no escopo da própria classe. Os membros protegidos (apresentados com #) são visíveis no escopo das subclasses (além de no da própria classe). Já os públicos (introduzidos com +) são acessíveis por qualquer entidade do modelo. No exemplo da Figura 2.6, os atributos da classe *Account* são protegidos, o que indica que podem ser acessados por qualquer uma de suas subclasses, enquanto seus métodos são públicos, sendo visíveis em todo o modelo.

Objetos. Objetos são instâncias de classes. Cada objeto possui identidade e, portanto, é distinguível dos demais. O estado de um objeto é representado pelos valores de seus atributos e das associações das quais ele participa. Seu comportamento é dado através dos métodos; estes podem ser invocados, provocando (ou não) alterações no estado do objeto associado.

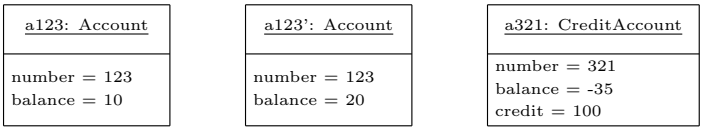

Figura 2.7. Objetos de *Account* e *CreditAccount*.

Três objetos podem ser observados na Figura 2.7. Os dois primeiros são

do tipo *Account* e o terceiro do tipo *CreditAccount*. Ainda mais: o objeto *a123'* pode ser o mesmo objeto *a123* em um estado posterior dado pela chamada de método `a123.creditar(10)`.

Tipos de Dados

Os tipos de dados (também chamados *tipos primitivos*) representam valores que são livres de efeitos colaterais e não possuem identidade. Dois valores que possuem a mesma representação são indistinguíveis. Geralmente representam conceitos de um domínio matemático e seus valores são imutáveis. É importante notar que o valor armazenado por um atributo pode ser atualizado, mas o valor em si não. Em UML, os tipos numéricos, *strings* e booleanos são os primitivos pré-definidos.

Enumerações. As enumerações representam tipos de dados definidos pelo usuário. O usuário pode especificar cada um dos valores (distintos) deste tipo. Quaisquer métodos definidos para enumerações devem ser rotulados com a tag *query* (garantindo que os valores não podem ser modificados; eles são livres de efeitos colaterais).

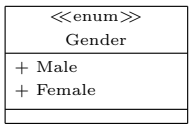

Figura 2.8. Enumeração *Gender*.

No exemplo da Figura 2.8, a enumeração *Gender* é introduzida: ela possui apenas dois valores, *Male* e *Female*.

2.3 RELACIONAMENTOS

Os relacionamentos representam conexões semânticas entre elementos do modelo. UML oferece diversos mecanismos para expressar estas ligações, sendo os principais as associações e as generalizações. As associações caracterizam as relações estruturais entre as instâncias enquanto a generalização cria uma taxonomia entre as entidades.

Associações

Uma associação estabelece uma relação estrutural entre dois classificadores. As associações podem ter um nome e possuem dois extremos (*association*

ends), ocupados por classes, para os quais podem ser designados papéis[1].

Em UML também é possível representar associações com mais de dois extremos. Elas são pouco comuns e não possuem uma semântica tão simples quanto as binárias. Por isso, preferimos não considerá-las.

Na Figura 2.9, a associação cujo nome é *owns* possui dois extremos. O primeiro contém o classificador *Person*, cujo papel é o de *owners*, e o segundo *Account*, cujo papel é *accounts*. Semanticamente, *owns* relaciona as instâncias de *Person* e *Account*.

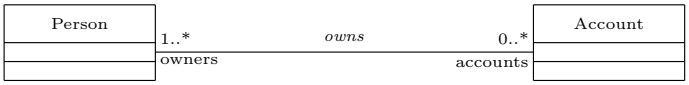

Figura 2.9. Associação *owns*.

A multiplicidade estabelece a quantidade de entidades de um extremo que estão relacionadas com uma única do outro. Pode ser especificado um intervalo para a cardinalidade deste conjunto. Se considerarmos ainda o relacionamento *owns*, temos que, para cada *Person*, poderão estar associadas 0 ou mais (*0..**) instâncias de *Account*; e para cada instância de *Account*, é possível ter 1 ou mais de *Person*. Esta última restrição obriga que toda conta possua pelo menos um dono.

A navegabilidade estabelece o conceito de visibilidade para associações. A entidade em um extremo é visto pela entidade oposta[2] se a associação entre elas for navegável; caso contrário, tal entidade não poderá expressar nada sobre as instâncias as quais ela está associada. A navegabilidade não é contemplada em nosso exemplo.

Links. Um link expressa uma instância (uma tupla) de uma associação. Ele sempre conecta duas instâncias do modelo. Na Figura 2.10, dois links de *owns* estão expressos: os que relacionam *alice* e *charles* à conta *a123*. A multiplicidade é respeitada: cada pessoa está associada a 0 ou mais contas, enquanto cada conta a 1 ou mais pessoas. Se existisse uma conta que não estivesse relacionada a qualquer pessoa, então estaria caracterizada uma configuração inválida do nosso sistema.

Classes de associação. Classes de associação são associações que possuem atributos e métodos. Semanticamente, é uma única construção que se comporta como associação e classe ao mesmo tempo, apesar de estar representada

[1] Papéis representam o comportamento de um elemento que participa de um contexto.
[2] Entidade oposta é a entidade que está conectada ao outro extremo da associação.

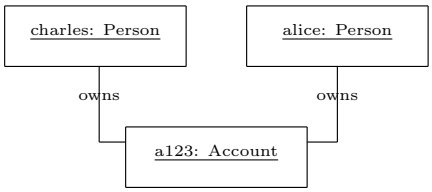

Figura 2.10. Links *owns*.

através de dois elementos no modelo. Na Figura 2.11, *Job* é uma classe de associação que forma com *employment* uma única entidade.

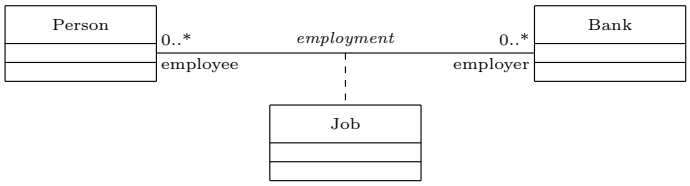

Figura 2.11. Associação *employment*.

Associações recursivas. Associações recursivas são aquelas onde ambos os extremos da associação estão conectadas ao mesmo classificador. Na Figura 2.12, a associação *marriage* é recursiva: ela conecta duas instâncias (possivelmente iguais) de *Person*.

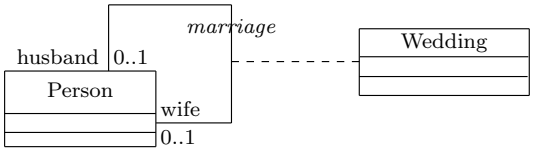

Figura 2.12. Associação *marriage*.

As associações recursivas apresentam ambigüidade apenas quando incluem uma classe de associação: suas instâncias são indistinguíveis quando referenciadas. Se tomarmos o exemplo da Figura 2.13, não é possível descobrir qual classe de associação *Wedding* (*w1* ou *w2*) o usuário deseja referenciar, exceto quando ele explicita o papel (*husband* ou *wife*) desempenhado pelo objeto *someone* no link.

Associações qualificadas. As associações qualificadas são associações em cujo um dos extremos possui um ou mais atributos qualificadores. Estes atributos servem para referenciar individualmente os objetos aos quais uma instância particular está associada; estes atributos *particionam* a associação.

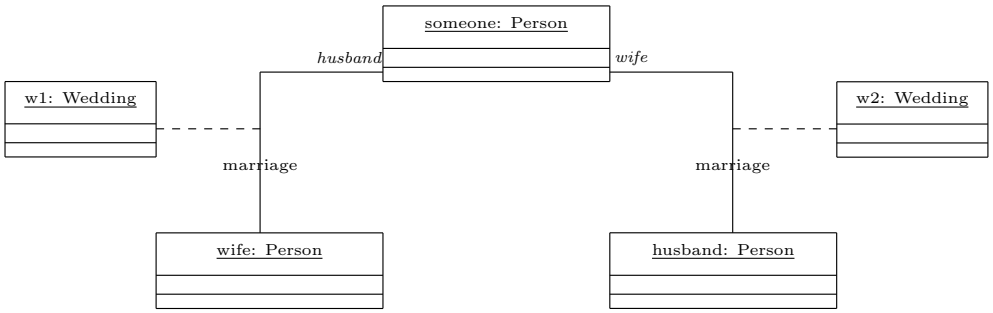

Figura 2.13. Links *marriage*.

A associação *has* da Figura 2.14 possui um atributo qualificador: o *accountNumber*. Este atributo identifica, individualmente, as contas que um banco possui.

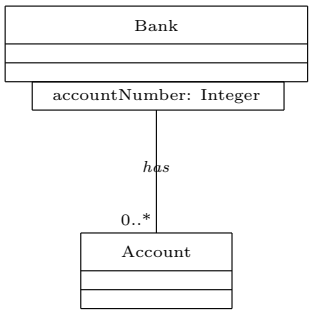

Figura 2.14. Associação *has*.

Estas três últimas particularidades (classes de associação, associações recursivas e associações qualificadas) estão sendo consideradas porque oferecem complicações adicionais ao mapeamento. É importante representá-las adequadamente para permitir uma escrita mais concisa das restrições, principalmente quando estas forem expressas em OCL[3] [OMG03b].

Generalizações

A generalização captura aspectos de herança entre uma classe mais geral (superclasse) e uma mais específica (subclasse). De fato, todos os elementos presentes na superclasse são herdados pela subclasse. Esta relação também estabelece que toda instância da subclasse é também uma da superclasse e pode ser utilizada onde esta última seria (*princípio de Liskov* [LW93]). Vale salientar que estamos interessados apenas em generalizações simples, onde

[3]Uma das extensões deste mapeamento pode contemplar OCL.

classes só possuem uma única superclasse.

Na Figura 2.15, podemos observar uma relação de generalização entre a *CreditAccount* e *Account*. Todos os membros de *Account* foram herdados por *CreditAccount*. Graças à generalização, contas que possuem crédito podem participar das associações *has* e *owns*, sendo utilizadas como contas convencionais, sem qualquer distinção.

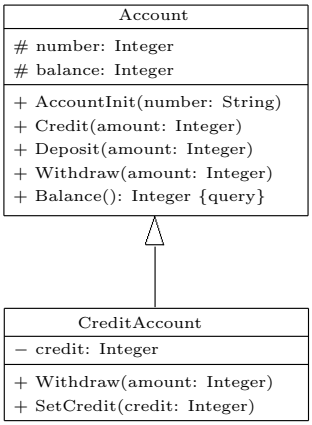

Figura 2.15. Relação de generalização entre *Account* e *CreditAccount*.

2.4 EXTENSIBILIDADE

UML foi projetada para ser uma linguagem extensível. Para isso, oferece três construções: as restrições (*constraints*), os estereótipos (*stereotypes*) e os rótulos (*tagged values*).

Declarações textuais explícitas que denotam alguma propriedade particular a ser satisfeita pelas entidades são consideradas restrições[4]. Quando determinamos que o saldo de uma conta deve ser sempre superior ou igual a 0, estamos introduzindo uma restrição no modelo. Antes, contas com saldo arbitrário satisfaziam-lhe; com a restrição, apenas aquelas cujo saldo não seja negativo o fazem.

O segundo tipo de extensão encontrado em UML são os estereótipos. Estes permitem que uma semântica diferente seja associada a uma construção já existente. Ao estereotipar o classificador *Gender* com *enumeration*, por exemplo, fica estabelecido que este agora possui um significado diferente do de uma classe como *Person*.

Por fim, as tags representam valores que podem ser associados a um elemento do modelo, introduzindo, implicitamente, uma restrição (estamos

[4]Restrições são expressas no modelo através de {...}.

ignorando rótulos meramente informativos). Por exemplo, o método *Balance* da classe *Account* é rotulado como *query*: este método, em particular, não modifica o estado do objeto (métodos podem modificá-lo irrestritamente).

2.5 SEMÂNTICA

Um diagrama de objetos *satisfaz* um diagrama de classes se todos os objetos e links são instâncias das entidades e respeitam as restrições introduzidas pelo modelo. Em outros termos, se o diagrama de objetos representa uma *configuração válida* do diagrama de classes.

Se considerarmos o diagrama de classes da Figura 2.9, os diagramas de objetos da Figura 2.2 satisfazem o modelo: *alice* e *charles* são instâncias de *Person*, *a123* de *Account* e não há qualquer restrição violada. Porém, se tomarmos um diagrama de objeto que contém apenas uma conta, estaríamos violando o modelo, pois toda conta deve estar associada a pelo menos uma pessoa (multiplicidade $1..*$).

Assim, conforme sugerido ao longo deste texto, a semântica de um diagrama de classes é o conjunto de todos os diagramas de objetos que o satisfazem. Essa definição também nos permite extrair uma relação de equivalência: dois diagramas de classes são *equivalentes* se possuem a mesma representação semântica.

CAPÍTULO 3

OHCIRCUS

Métodos Formais é a área da computação que procura prover um interpretação formal para os diversos aspectos de um programa, como tipos de dados, concorrência, comunicação, implementabilidade etc. Esta disciplina já é de longa data, e tem origem nos trabalhos de Dijkstra [Dij72].

Diversos foram os formalismos propostos para contemplar estes aspectos, sendo Z [Spi92] e CSP [RHB97] dois dos mais utilizados na indústria.

Z é uma linguagem baseada em modelos para especificar sistemas imperativos. Utilizando teoria dos conjuntos e lógica de primeira ordem, introduzimos esquemas para representar o estado e suas diversas operações. Dessa especificação, é possível verificar formalmente as propriedades, garantindo sua corretude. Posteriormente, podemos realizar passos de refinamento, onde modelos cada vez mais próximos de uma implementação preservam as propriedades do original.

CSP é uma álgebra de processos. Processos são definidos através dos eventos que podem (ou não) comunicar com o ambiente. Esses eventos têm uma ordem de ocorrência, e podem ser oferecidos como um escolha ou aleatoriamente ao ambiente. Além disso, os processos podem ser combinados utilizando os operadores de paralelismo, como sincronização ou *interleave*. CSP permite estudar concorrência e comunicação entre processos, e segue um desenvolvimento semelhante ao de Z, baseado em provas de propriedades e refinamento.

Estudar estes aspectos individualmente é importante, mas atualmente procura-se unificar os diversos formalismos e verificar a influência de uns sobre os outros. Em particular, as várias tentativas de integrar Z e álgebras de processos (CSP-Z, CSP-OZ [Fis98]) procuram contemplar estado e aspectos de comunicação de sistemas concorrentes numa linguagem unificada, aproveitando as teorias e ferramentas existentes. Um formalismo semelhante é *Circus* [WC02]; entretanto, propõe e formaliza um *cálculo de refinamentos* para esta abordagem. Além disso, procura ser familiar àqueles que conhecem Z e CSP e possibilita o reaproveitamento de ferramentas existentes, como FDR [Gol01] e Z/EVES [Saa99].

OhCircus é uma extensão de *Circus* que adiciona, além de processos, classes, herança, ligação dinâmica e outros recursos dos paradigmas concorrente

e orientado a objeto. E, embora sua semântica ainda não esteja completa, as vantagens de utilizá-la superam este problema.

Neste capítulo vamos apresentar a parte orientada a objetos de *OhCircus*, uma vez que concorrência foge ao escopo deste trabalho. Será através do exemplo de um sistema bancário, bastante semelhante àquele de [CSW03] e da Figura 2.4, explorando classes, herança e associações.

Neste capítulo seguimos o estilo de Z, onde os esquemas são introduzidos antes de seus comentários.

3.1 CLASSES

Da mesma forma que *Circus* e Z, um programa em *OhCircus* é uma seqüência de páragrafos. Neles é possível definir processos e classes, embora neste trabalho estejamos apenas interessados nas características orientadas a objetos da linguagem. Para ilustrar estes elementos, a próxima declaração introduz a classe *Account*, que modela contas bancárias.

class *Account* $\widehat=$ **begin**

Uma classe em *OhCircus* é bastante semelhante a uma especificação em Z. Atributos, construtor e métodos também são introduzidos através de parágrafos, geralmente sob a forma de esquemas.

state *AccountState*

protected $number : \mathbb{Z}$
protected $balance : \mathbb{Z}$

A cláusula **state** destaca o esquema que define o estado de uma classe. Este esquema é semelhante ao de Z, embora suas declarações de variáveis introduzam atributos na classe. Essas declarações também podem incluir qualificadores. Se nada é dito, o atributo é tido como **private**, mas também podemos declará-los como **protected** ou **public**, todos com o mesmo significado dos de
UML. Embora sejam tornados explícitos, modificadores de visibilidade são de interesse apenas de linguagens de implementação (como Java [GJSB00]). *OhCircus* (e as demais linguagens de especificação) não oferecem qualquer restrição de acesso aos atributos.

O esquema de estado *AccountState* declara dois atributos protegidos: *number* e *balance*. O primeiro denota o número de uma conta, enquanto o segundo, seu saldo. Uma vez qualificados como **protected**, as subclasses

de *Account* são capazes de também manipular estes atributos.

initial *AccountInit*

$$AccountState'$$
$$number? : \mathbb{Z}$$

$$number' = number?$$
$$balance' = 0$$

A cláusula **initial** introduz um construtor. Ele não pode ser invocado diretamente; apenas através de uma expressão **new**. Para *Account*, ele toma o número de um conta como entrada e inicializa o saldo com 0.

public *Deposit*

$$\Delta AccountState$$
$$amount? : \mathbb{N}$$

$$balance' = balance + amount?$$
$$number' = number$$

public *Withdraw*

$$\Delta AccountState$$
$$amount? : \mathbb{N}$$

$$amount? \leq balance$$
$$balance' = balance - amount?$$
$$number' = number$$

logical *GetNumber*

$$\Xi AccountState$$
$$n! : \mathbb{N}$$

$$n! = number$$

Métodos são diferenciados de outros parágrafos através do uso de qualificações como **private**, **protected**, **public** ou **logical**. As três primeiras estão diretamente relacionadas à visibilidade do método, novamente semelhantes a

UML. Já os métodos lógicos são apenas artefatos de especificação, sendo úteis, por exemplo, para calcular uma expressão complexa, mas não necessitam ser implementados.

Analogamente a Z, os métodos de uma classe *OhCircus* interagem com o estado, modificando-o (Δ) ou não (Ξ). Os métodos *Deposit* e *Withdraw* modificam o estado da conta incrementando ou decrementando respectivamente o saldo com o valor do parâmetro de entrada. Em particular, o método *Withdraw* só é executado quando há saldo suficiente. Por fim, o método *GetNumber* é declarado como lógico e não modifica o estado da conta.

end

Toda declaração de classe é finalizada com um **end**.

3.2 HERANÇA

Em *OhCircus*, classes podem herdar de uma única superclasse (herança simples). Isto é declarado através da cláusula **extends** que, se omitida, declara herdar implicitamente da classe especial **object**. Na próxima declaração, *CreditAccount* herda de *Account*.

class *CreditAccount* $\widehat{=}$ **extends** *Account* **begin**

A intenção é construir uma conta que, além de registrar o saldo de um cliente, oferece-o também crédito.

state *CreditAccountState*

private $credit : \mathbb{Z}$

$balance + credit \geq 0$

public *Withdraw*

$\Delta CreditAccount$
$amount? : \mathbb{N}$

$amount? \leq balance + credit$
$balance' = balance - amount?$
$number' = number$

Se um método é redefinido, não existe *inclusão* do método da superclasse na nova especificação. Porém, é necessário que a nova definição preserve o comportamento original. Em outro termos, é necessário que a nova especificação seja um refinamento do método original.

A operação *Withdraw* de *CreditAccount* precisa comportar a nova situação expressa por contas que possuem crédito: não é necessário possuir saldo suficiente; apenas que o saque não ultrapasse o limite. Note o enfraquecimento da pré-condição e como ela representa um refinamento da original.

public *Deposit*

$\Delta CreditAccountState$
$amount? : \mathbb{N}$

$balance' = balance + amount?$
$number' = number$
$credit' = credit$

Os métodos que não são redefinidos são implicitamente herdados pela nova classe, com a ressalva de que eles, obrigatoriamente, não modificam os componentes introduzidos pelo novo estado. Por exemplo, o método *Deposit* da classe *Account* não foi redefinido e, por isso, está disponível para *CreditAccount* da forma acima (observe a inclusão do predicado $credit' = credit$).

public *SetCredit*

$\Delta CreditAccountState$
$\Xi AccountState$
$credit? : \mathbb{N}$

$credit' = credit?$

Por fim, *SetCredit* representa a definição de um novo método. O destaque

vai para a notação utilizada ($\Xi\, AccountState$) para declarar que este método não altera os atributos da superclasse.

end

3.3 ASSOCIAÇÕES

Para representar a aplicação bancária em si, é necessário também especificar a classe *Bank*. É ela quem vai relacionar contas e clientes, e conter as operações de abertura e fechamento de contas, inclusão e exclusão de titulares e associação entre titulares e contas. Fazendo analogia ao padrão de projeto *Façade* [GHJV95], esta classe representa a fachada do sistema. Particularmente, apresentaremos apenas seu estado, que abre precedentes para discussões posteriores.

class $Bank \mathrel{\widehat=}$ **begin**

state *Bank*

$$
\begin{array}{l}
accounts : \mathbb{P}\, Account \\
persons : \mathbb{P}\, Person \\
owns : Account \leftrightarrow Person \\
\hline
\mathrm{dom}\, owns \subseteq accounts \land \mathrm{ran}\, owns \subseteq persons \\
\forall\, a : accounts \bullet \#owns(\!|\, \{a\} \,|\!) \in \mathbb{N}_1 \\
\\
\forall\, a_1, a_2 : accounts \bullet a_1.number = a_2.number \Leftrightarrow a_1 = a_2
\end{array}
$$

Os atributos de banco incluem um conjunto de contas e outro de clientes cadastrados, além de uma relação entre contas e clientes. O invariante estabelece que apenas as contas e clientes cadastrados podem participar da relação, e que toda conta necessita estar associada a pelo menos um titular. Além disso, garante que duas contas são iguais quando seus números são iguais (identidade das contas).

As vantagens de utilizar esta abordagem (com relações), ao invés de incluir atributos nas classes é facilitar o reuso. Se já existem, por exemplo, especificações verificadas destas classes, a simples introdução do atributo pode provocar alterações em toda a classe e conseqüente revalidação. Note que *OhCircus* não possui um construtor equivalente à associação de UML.

end

CAPÍTULO 4

INTEGRANDO UML E OHCIRCUS

O diagrama de classes representa todo um sistema. Nele estão caracterizadas as principais entidades do modelo, seus respectivos relacionamentos e possíveis restrições globais. Essa coleção de informações independe do tempo e por isso é chamada de visão estática.

Em *OhCircus*, a "visão estática" é capturada diretamente através do seu conjunto de classes. Como visto anteriormente, elas possuem invariantes, atributos e métodos, como as de UML. Contudo, a interação entre duas classes é capturada apenas através de atributos, uma vez que não existe o conceito de *associação*. Por exemplo, recorde que no capítulo anterior, a classe *Bank* servia de elo entre *Account* e *Person*. Agora, imagine se *Bank* fosse removida; então, a relação entre contas e pessoas deveria ser capturada através de atributos e invariantes. As classes *Account* e *Person* teriam, respectivamente, um conjunto de pessoas e de contas, além de um invariante para garantir a consistência dessa associação.

Outro detalhe importante é que em *OhCircus* não é possível estabelecer invariantes globais. As classes só podem fazer restrições locais, sobre os seus próprios atributos. Recorde da página 26 que o invariante da classe *Bank*, embora pareça o contrário, é local à própria classe. Outras classes poderiam definir diferentes restrições.

A nossa solução para estes problemas é introduzir uma classe chamada *Model*, responsável por capturar toda a estrutura de um diagrama de classes: o conjunto de instâncias das classes, os relacionamentos, os invariantes globais e até mesmo os aspectos dinâmicos. Nós acreditamos que esta abordagem oferece uma visão mais abstrata dos diagramas de classes quando comparada a outras [Eva98, LB98, BHH+97], que consideram somente a representação das classes, capturando associações diretamente através de atributos e ignorando restrições globais.

Note que a classe *Model* não faz parte do diagrama de classes em si, mas sim da nossa interpretação sobre o mesmo: ela representa uma meta-classe.

Tomando como referência nossa discussão acima sobre como capturar a associação entre contas e pessoas, ilustramos a seguir uma classe *Model* que realizaria este objetivo.

class *Model* $\widehat{=}$ **begin**

state *ModelState*

$$persons : \mathbb{P}\, Person$$
$$accounts : \mathbb{P}\, Account$$
$$owns : Person \leftrightarrow Account$$

$$owns \in persons \leftrightarrow accounts$$
$$\forall\, a : accounts \bullet \#(owns^{\sim}(\!|\, \{a\} \,|\!)) \in \mathbb{N}_1$$

end

Seus componentes *accounts* e *persons* representam as instâncias das classes *Account* e *Person* respectivamente, enquanto *owns* captura a associação entre elas. Note também como o invariante relaciona os domínios da relação com os conjuntos das instâncias (só pessoas e contas que estiverem nos respectivos conjuntos *persons* e *accounts* podem participar do relacionamento *owns*) e as respectivas multiplicidades (toda conta em *accounts* se relaciona com pelo menos uma pessoa através de *owns*). Esta classe representa o diagrama de objetos da Figura 4.1.

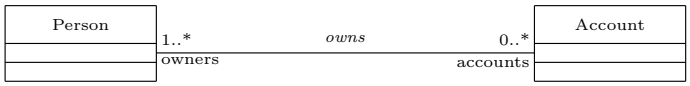

Figura 4.1. Diagrama de classes simples.

Observe como a classe *Model* retrata, de fato, a semântica do modelo tal qual discutimos na seção 2.5. Cada possível valor dessa classe reproduz uma configuração válida do diagrama de classes. Em outros termos, cada instância de *Model* reflete um diagrama de objetos. Por exemplo, a valoração $(\!|\, persons \rightsquigarrow \{charles, alice\},\ accounts \rightsquigarrow \{a123\},\ owns \rightsquigarrow \{(charles, a123), (alice, a123)\} \,|\!)$ representa o diagrama da Figura 4.2.

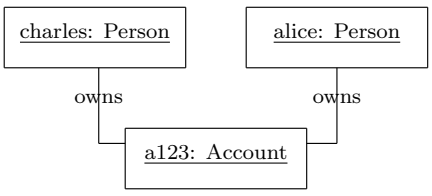

Figura 4.2. Diagrama de objetos que representa uma valoração válida.

Embora não seja o foco principal deste trabalho, é importante observar também como a classe *Model* traduz adequadamente a noção de aspectos

dinâmicos: suas instâncias evoluem através da aplicação de operações, produzindo *seqüências* de estados válidos. Esta noção é bastante semelhante ao *history model* de Object-Z [Smi92], e pode ser futuramente explorada.

4.1 CLASSIFICADORES

Nesta seção apresentaremos o mapeamento dos elementos *classificadores* de UML: as classes e os tipos de dados. Ao final, ainda discutimos a importância da identidade em UML e a correspondência disso em *OhCircus*.

Classes

Uma vez que possuem as mesmas construções, como atributos e métodos, as classes de UML são facilmente mapeadas em *OhCircus*, embora algumas restrições devam ser impostas. Por exemplo, na Figura 4.3, temos a classe *Account* mapeada em uma classe de *OhCircus*.

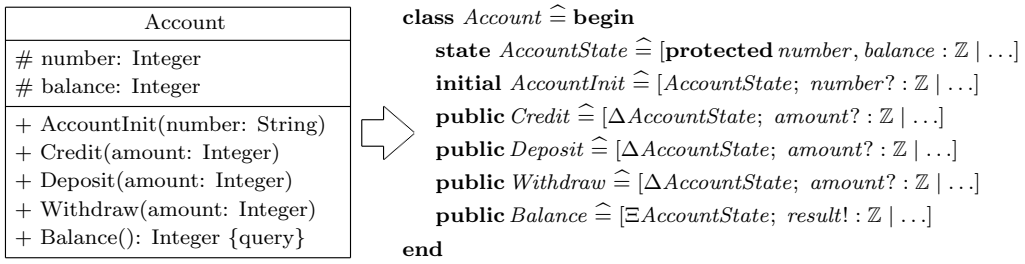

Figura 4.3. Mapeando a classe *Account*.

Os atributos de uma classe UML são mapeados em variáveis do esquema de estado (**state** *schema*) de uma classe *OhCircus*, juntamente com a visibilidade e seu tipo. Note que os atributos da classe *Account*, *number* e *balance*, foram diretamente mapeados em variáveis do esquema *AccountState*. A única complicação é a *multiplicidade* do atributo (ver Tabela 4.1).

Multiplicidade	Mapeamento
0..1	O atributo é mapeado em uma variável do esquema de estado.
1	O atributo é mapeado em variável cujo valor é diferente de **null**.
$m..n$	O atributo é mapeado em um conjunto cuja multiplicidade pertence ao intervalo $m..n$.

Tabela 4.1. Mapeamento da multiplicidade de atributos.

Se a multiplicidade máxima é igual a 1, o atributo é traduzido diretamente em variável. Ainda mais, se a multiplicidade mínima não for 0, obriga-o a ser diferente de **null**. Caso a multiplicidade máxima seja maior que 1, o atributo é representado como um conjunto cujo tamanho deve pertencer ao intervalo especificado pela multiplicidade. O valor padrão é 1.

O método estereotipado como *constructor* é mapeado num esquema inicial (**initial** *schema*). É importante observar que, devido a uma restrição de *OhCircus*, só pode haver um único *construtor* em classes UML. O método *AccountInit* (estereotipado como *constructor*, embora não esteja visível na figura) é mapeado no esquema **initial** *AccountInit*, responsável pela inicialização da classe *Account*.

Os demais métodos são mapeados em esquemas de operações. Estes esquemas modificam o estado (Δ-esquemas), exceto quando rotulados *query*; neste caso, são mapeados em Ξ-esquemas. Os parâmetros de entrada dos métodos são mapeados em variáveis de entrada (decoradas com ?), enquanto o resultado do método, se existente, é mapeado na variável *result*!. Todos essas traduções podem ser observadas na Figura 2.6.

state *ModelState*

$$accounts : \mathbb{P}\, Account$$
$$persons : \mathbb{P}\, Person$$
$$\ldots$$

Como sugerido anteriormente, cada classe introduz em *Model* um conjunto de instâncias, diretamente representado através de conjuntos-potência. Acima, podemos ver os conjuntos de instâncias das classes *Account* e *Person*.

Vale ressaltar que, se os tipos dos atributos de uma classe são também classes, então eles introduzem invariantes na classe *Model*. Estes invariantes expressam que os valores destes atributos devem estar contidos em seus respectivos conjuntos de instâncias.

Tipos de Dados

UML possui quatro tipos de dados básicos: `Integer`, `UnlimitedNatural`, `Boolean` e `String`. Em *OhCircus*, nós temos a correspondência direta de $\mathbb{Z}$ para `Integer`, $\mathbb{N}$ para `UnlimitedNatural` e $\mathbb{B}$ para `Boolean`. Vale ressaltar que `Integer` e `UnlimitedNatural` são ambos infinitos, como consta na especificação de UML [OMG03c]. Entretanto, o tipo `String` não possui equivalente e nossa abordagem é representá-lo por uma seqüência de elementos

em algum conjunto de caracteres (e.g. ASCII ou UNICODE, que são naturalmente definidos utilizando enumerações). Assim, várias de suas operações como `concat`, `size` e `substring` tornam-se diretamente disponíveis.

Enumerações. As enumerações são traduzidas em *free types* de *OhCircus*. Cada um dos valores da enumeração tornam-se construtores nulários (ou constantes) prefixados pelo nome da enumeração. Isso se faz necessário porque os nomes introduzidos por enumerações em UML são locais à classe e necessitam de qualificação; já os introduzidos pelos *free types* de *OhCircus* são globais. Na Figura 4.4, podemos observar a enumeração *Gender* sendo mapeada no *free type* *Gender*.

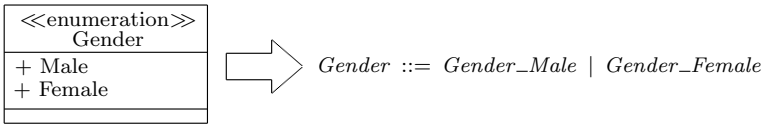

Figura 4.4. Mapeando a enumeração *Gender*.

É importante observar que, uma vez que os tipos de dados não possuem identidade, eles não introduzem um "conjunto de todas as instâncias" na classe *Model*; sua representação é direta.

Identidade

Todos os objetos de UML possuem identidade: eles são unicamente identificados e distintos uns dos outros no sistema. Por outro lado, valores que não possuem esta propriedade, como os tipos de dados, são diferenciados através de sua estrutura. Se os objetos de UML pudessem abdicar da identidade, eles seriam então distinguidos pelos *valores* de seus atributos. Vale ressaltar que esta característica é inerente aos objetos. Estas situações são possíveis e bastante comuns em C++ [Str00], na qual objetos podem ser construídos por valor ou por referência.

Como temos visto ao longo deste texto, contas e bancos estão relacionados através da associação *has*. Portanto, dois bancos diferentes possuem conjuntos de contas distintos. Porém, nada impede que contas possuam o mesmo número (embora estejam em bancos diferentes) e o mesmo saldo. Em UML, ainda assim estas contas são diferentes, graças às suas identidades. Entretanto, em linguagens que possuem semântica de cópia (como Z, *OhCircus* e ROOL [BS00]), os objetos dependem de sua estrutura. Neste caso, os objetos são idênticos e estão sendo compartilhados pelos bancos.

Este problema pode ser uma complicação para o nosso mapeamento: uma vez que estas contas pertencem ao mesmo conjunto (o conjunto de todas as contas do sistema), e conjuntos ignoram a multiplicidade de seus elementos, estas contas, outrora distintas por sua identidade, tornam-se uma só. Elas serão realmente compartilhadas pelos bancos, e a partir de então, quaisquer modificações feitas em uma refletem na outra. Porém, semântica de referência é um problema em aberto, e foge ao escopo deste trabalho. Cabe ao projetista controlar explicitamente a identidade dos objetos e se nenhuma medida for tomada para contemplar este entrave, o mapeamento pode tomar uma representação semântica diferente da almejada.

state *AccountState*

$$IDAccount : \mathbb{N}$$
$$\ldots$$

state *ModelState*

$$accounts : \mathbb{P}\, Account$$

$$\forall\, a_1, a_2 : accounts \bullet a_1.IDAccount \neq a_2.IDAccount \Leftrightarrow a_1 \neq a_2$$

Este interessante paliativo introduz o atributo *IDAccount* na classe *Account*, enquanto um invariante global estabelece que duas contas são distintas quando suas *identidades* são diferentes. Isto resolve nosso problema, embora deva ser utilizado em todas as classes onde não é possível depender apenas de sua estrutura.

4.2 RELACIONAMENTOS

Os mapeamentos dos principais relacionamentos entre classificadores (herança e associação) são discutido nesta seção. Em particular, cada possível elemento que pode ser adicionado a uma associação é contemplado individualmente.

Generalização

Para declarar uma subclasse em *OhCircus*, apenas incluímos a cláusula **extends** seguida do nome da superclasse. A Figura 4.5 ilustra este relacionamento entre *CreditAccount* e *Account* em *OhCircus*. Além disso, *OhCircus* introduz uma restrição em UML: subclasses só podem herdar de uma única superclasse.

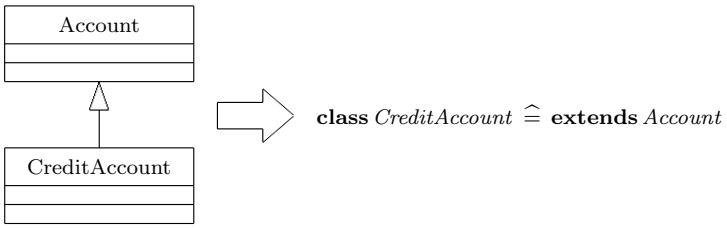

Figura 4.5. Mapeando herança entre as classes *CreditAccount* e *Account*.

state *ModelState*

$accounts : \mathbb{P}\ Account$

$creditAccounts : \mathbb{P}\ CreditAccount$

$creditAccounts \subseteq accounts$

A classe *CreditAccount*, como esperado, introduz em *Model* o conjunto de instâncias *creditAccounts*. Além disso, um invariante garante que os elementos de *creditAccounts* também são valores de *accounts*.

Associações

Associações são uma interessante construção a ser capturada, visto que não estão diretamente disponíveis em linguagens orientada a objetos como *OhCircus*. A abordagem mais comum é representá-las diretamente através de atributos [Eva98, LB98, Cla99], embora acreditemos que não seja a mais natural por existir uma diferença conceitual entre ambas (vide o Capítulo 2).

Uma vez que introduzem entidades no modelo, associações devem ser capturadas globalmente pela classe *Model*. Os papéis desempenhados pelas classes tornam-se atributos e um invariante conecta todos estes itens. Tais atributos são apenas *açúcares sintáticos*, pois a consistência da associação é mantida por uma relação na classe *Model*. Contudo, eles se fazem necessários porque a semântica de UML permite que uma classe fale dos membros aos quais ela está associada. Por fim, são também estabelecidas restrições para contemplar a multiplicidade.

Figura 4.6. Mapeando a associação *management*.

state *ModelState*

$$
\begin{array}{l}
persons : \mathbb{P}\, Person \\
banks : \mathbb{P}\, Bank \\
management : Person \leftrightarrow Bank \\
\hline
management \in persons \leftrightarrow banks \\
\forall\, b : banks \bullet \#(management^{\sim}\ (\!|\ \{b\}\ |\!)) \in 0..1 \\
\forall\, p : persons \bullet p.managed = management\ (\!|\ \{p\}\ |\!) \\
\forall\, b : banks \bullet b.manager = management^{\sim}\ (\!|\ \{b\}\ |\!)
\end{array}
$$

state *PersonState*

$$
managed : \mathbb{P}\, Bank
$$

state *BankState*

$$
manager : \mathbb{P}\, Person
$$

Considere a associação *management* entre *Person* e *Bank* da Figura 4.6. A associação torna-se um atributo da classe *Model*, como pode ser visto anteriormente. O invariante conecta os domínios da associação com os conjuntos de instâncias e as devidas multiplicidades. Já os papéis *manager* e *managed* tornam-se atributos de, respectivamente, *Bank* e *Person*. Por fim, o invariante de *Model* ainda estabelece como os atributos são interpretados em termos da relação original[1].

Classes de associação. Apesar de representar uma única entidade que desempenha papel de associação e classe ao mesmo tempo, a classe de associação será capturada através de um mapeamento separado. A "classe" desse relacionamento é mapeada usando os passos descritos na Seção 4.1, enquanto a "associação" é capturada através de um atributo modificado na classe *Model*. Dado que a classe de associação é unicamente determinada pelo par relacionado, a associação é mapeada numa função[2]. Ainda mais: no sistema não podem existir instâncias dessa "classe" que não pertençam ao relacionamento. Por último, em virtude de UML, os classificadores relacionados ganham novos atributos para acessar as classes de associação às quais elas estão relacionadas.

[1] A imagem relacional de um conjunto de objetos ($(\!|\ \_\ |\!)$) é o conjunto de objetos associados ao primeiro através de uma relação.

[2] Perceba que o domínio dessa função representa a relação original.

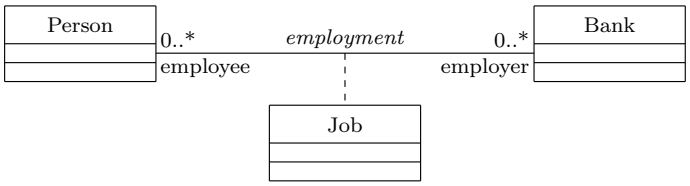

Figura 4.7. Mapeando a associação *employment* e a classe *Job*.

state *ModelState*

$persons : \mathbb{P}\, Person$
$banks : \mathbb{P}\, Bank$
$jobs : \mathbb{P}\, Job$
$employment : (Person \times Bank) \pfun Job$

$employment \in (persons \times banks) \pfun jobs$
$\operatorname{ran} employment = jobs$

$\forall\, p : persons \bullet p.employer = (\operatorname{dom} employment) (\!|\, \{p\} \,|\!)$
$\forall\, b : banks \bullet b.employee = (\operatorname{dom} employment)^{\sim} (\!|\, \{b\} \,|\!)$

$\forall\, p : persons \bullet p.job = employment (\!|\, \{p\} \times banks \,|\!)$
$\forall\, b : banks \bullet b.job = employment (\!|\, persons \times \{b\} \,|\!)$

state *PersonState*

$employer : \mathbb{P}\, Bank$
$job : \mathbb{P}\, Job$

state *BankState*

$employee : \mathbb{P}\, Person$
$job : \mathbb{P}\, Job$

Na Figura 4.7, temos a associação *employment*, que relaciona *Person* e *Bank*. Cada par nesta relação determina um *Job* e, por isso, *employment* é uma função cuja imagem é igual a *jobs*. Para os papéis da associação, o mapeamento é semelhante ao discutido na seção anterior, exceto pela inclusão dos atributos *job* e respectivos invariantes.

Associações recursivas. As associações recursivas conectam instâncias da mesma classe. Entretanto, quando esta inclui uma classe de associação, dá origem a uma ambigüidade de navegação, sendo necessário especificar a partir

de qual papel ela será efetuada (vide Seção 2.3). Tomando a associação *marriage* da Figura 4.8, a navegação `Person.wedding` é ambígua, necessitando do respectivo papel (`Person.wedding[wife]` ou `Person.wedding[husband]`).

A solução dessa ambigüidade é bastante simples: como os classificadores relacionados vão receber dois atributos que dizem respeito à classe de associação, introduzimos um sufixo em cada um, com o respectivo papel.

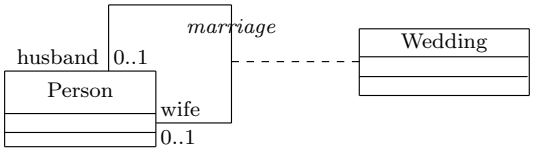

Figura 4.8. Mapeando a associação recursiva *marriage*.

state *ModelState*

$$
\begin{array}{l}
persons : \mathbb{P}\ Person \\
weddings : \mathbb{P}\ Wedding \\
marriage : (Person \times Person) \pfun Wedding \\
\hline
marriage \in (persons \times persons) \pfun weddings \\
\mathrm{ran}\ marriage = weddings \\
\\
\forall\, p : persons \bullet \\
\quad \#((\mathrm{dom}\ marriage) \lhd\!\!\rhd \{p\}) \in 0..1 \\
\quad \#((\mathrm{dom}\ marriage)^{\sim} \lhd\!\!\rhd \{p\}) \in 0..1 \\
\\
\forall\, p : persons \bullet \\
\quad p.husband = (\mathrm{dom}\ marriage) \lhd\!\!\rhd \{p\} \\
\quad p.wife = (\mathrm{dom}\ marriage)^{\sim} \lhd\!\!\rhd \{p\} \\
\quad p.wedding\_husband = marriage \lhd\!\!\rhd \{p\} \times persons \\
\quad p.wedding\_wife = marriage \lhd\!\!\rhd persons \times \{p\}
\end{array}
$$

state *PersonState*

$$
\begin{array}{l}
husband : \mathbb{P}\ Person \\
wife : \mathbb{P}\ Person \\
\\
wedding\_husband : \mathbb{P}\ Wedding \\
wedding\_wife : \mathbb{P}\ Wedding
\end{array}
$$

Note que a classe *Person* recebeu dois atributos relacionados à classe de associação *Wedding*: *wedding_wife* e *wedding_husband*. As demais construções continuam semelhantes à tradução da seção anterior.

Associações qualificadas. Como dito anteriormente, os qualificadores identificam unicamente objetos numa coleção que pode ser navegada através de uma associação. O classificador que possui tal modificador recebe um atributo que, ao invés de um simples conjunto, é uma *função* do par ordenado dos tipos dos qualificadores para o tipo do conjunto-resultado de uma navegação[3]. Observe que a associação não é alterada em virtude disso.

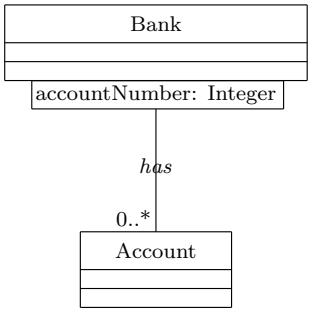

Figura 4.9. Mapeando a associação qualificada *has*.

state *ModelState*

$$banks : \mathbb{P}\, Bank$$
$$accounts : \mathbb{P}\, Account$$
$$has : Bank \leftrightarrow Account$$

$$\ldots$$
$$\forall\, b : banks \bullet \operatorname{ran} b.accounts = has \,(\!|\, \{b\} \,|\!)$$

state *BankState*

$$accounts : \mathbb{Z} \rightarrowtail Account$$

A classe *Bank* qualifica a associação *has* com um inteiro na Figura 4.9, e por isso recebe uma função de $\mathbb{Z}$ para *Account*, cuja imagem é a imagem relacional de *Bank* em *has*.

[3] A imagem desta função ainda é a imagem relacional da instância.

CAPÍTULO 5

REFINAMENTO

A relação de refinamento expressa um conceito bastante comum em Engenharia de Software: um componente "melhor" pode ser utilizado em lugar de outro, sem modificar as propriedades de um sistema. Por exemplo, descrever mais precisamente as estruturas de dados ou explicitar como alguns cálculos serão realizados representam possíveis melhorias. De modo geral, refinar significa adicionar informações a um modelo, resolvendo escolhas de projeto que ainda estejam em aberto ou contemplando situações ainda inexploradas. É importante destacar que estes aperfeiçoamentos podem ser feitos gradativamente, produzindo modelos cada vez mais próximos da implementação.

O RUP [Kru00] é um exemplo de processo de desenvolvimento que utiliza, mesmo que informalmente, o conceito de refinamento. Inicialmente, é construído um modelo de análise, cujo maior interesse é satisfazer os requisitos do cliente. Depois, diversos passos de projeto são aplicados a este modelo, apurando-o cada vez mais. Ao final, um modelo de implementação é produzido. Embora bastante interessante, este processo está sujeito a falhas, visto que é informal e pouco rigoroso.

Por outro lado, a comunidade de Métodos Formais caracteriza precisamente o que é uma relação de refinamento. Enfraquecer as pré-condições para contemplar mais situações e fortalecer a pós-condição para resolver o não-determinismo representam as melhorias. E para garantir a corretude deste procedimento, existem as obrigações de prova.

Graças ao mapeamento do capítulo anterior, é possível agora explorar refinamento nos moldes formais: o refinamento de modelos UML pode ser garantido pelo refinamento de dados em *OhCircus*.

O exemplo deste capítulo lida com um antigo debate da comunidade de orientação a objetos: a representação de associações como atributos [Gén01, GBHS97, Rum96, Tan95, Vel94]. Nosso papel é dar um respaldo formal a esta abordagem, transformando modelos que mesclam atributos e associações em modelos que só possuem atributos, tornando-os mais próximos da implementação em Java, por exemplo. A nossa abordagem é propor uma formalização em Z para obter o suporte do Z/EVES [Saa03], sendo o resultado facilmente estendido para *OhCircus*, em virtude do alicerce comum para refinamento.

Na próxima seção apresentamos os conceitos de refinamento de dados. Na

seguinte descrevemos os modelos de interesse em Z: o abstrato, que contém duas classes e uma associação, e o concreto, que contém apenas atributos. Na última seção, discutimos o invariante de acoplamento que conecta estas especificações e oferecemos uma intuição da prova deste refinamento.

5.1 REFINAMENTO DE DADOS

Para nossos propósitos, um tipo de dado (por exemplo, uma classe) é um conjunto de estados possíveis e uma lista de operações que representam relações sobre estes estados (por exemplo, atributos e métodos). A representação desse estado é irrelevante, haja vista que ela é encapsulada pelas operações.

Nesta seção, desejamos mostrar quais são as condições necessárias para que um tipo de dado A seja refinado por um outro C, o segundo possuindo uma lista de operações correspondente a do primeiro.

As linguagens Z e *OhCircus* abordam este problema estabelecendo condições para que uma relação *Retrieve* (chamada invariante de acoplamento) entre os estados de A e C seja uma *simulação* [WD96, CSW03]; qualquer aplicação de uma operação em C pode ser simulada pela operação equivalente em A. Assim, duas obrigações de prova surgem para garantir que uma operação COP sobre o estado concreto seja um refinamento de uma operação abstrata correspondente, AOP, sobre o estado abstrato. Estas obrigações são para todas as operações e estão sumarizadas nos próximos dois teoremas.

Teorema da Aplicabilidade

$$\forall\, AState, CState \bullet \\ \quad \text{pre}\ AOP \land Retrieve \Rightarrow \text{pre}\ COP$$

O teorema da aplicabilidade estabelece que, para todos os estados $AState$ em que a operação AOP é aplicável, os possíveis estados $CState$ correspondentes satisfazem a pré-condição de COP. Note que este teorema permite que a operação COP considere outros estados para os quais a operação AOP, correspondentemente, não esteja definida.

Teorema da Corretude

$$\forall\, AState, CState, CState' \bullet \\ \quad \text{pre}\ AOP \land Retrieve \land COP \Rightarrow (\exists\, AState' \bullet AOP \land Retrieve')$$

O segundo teorema, o da Corretude, pode ser melhor entendido através da Figura 5.1. Em todo estado *AState* (e *CState*s correspondentes) onde a pré-condição de *AOP* é satisfeita, aplicar a operação *COP* leva ao estado *CState'*, correlativo àquele da aplicação de *AOP*. Perceba que a operação *COP* pode restringir os possíveis estados de saída (relacionados aos de *AOP*) resolvendo o não-determinismo.

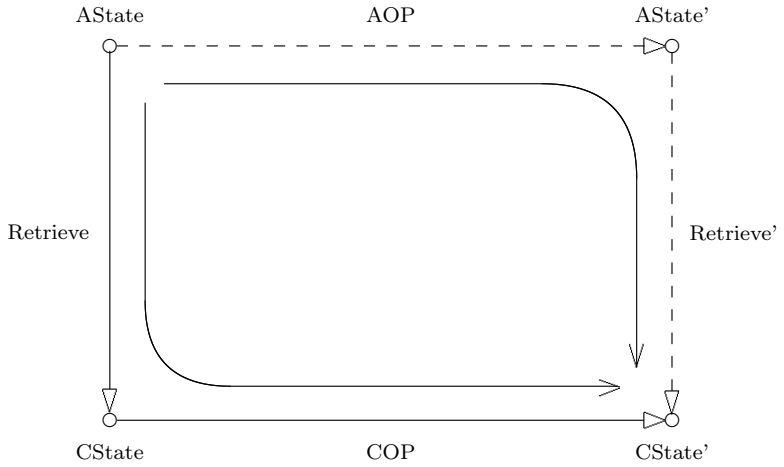

Figura 5.1. Teorema da corretude.

A operação de inicialização também fica obrigada a cumprir ambos os teoremas. Contudo, uma vez que sempre é aplicável (sua pré-condição é true), o primeiro teorema é trivialmente satisfeito, enquanto o segundo é simplificado como se segue:

Teorema da Inicialização

$$\forall\, CState' \bullet CInit \Rightarrow (\exists\, AState' \bullet AInit \land Retrieve')$$

Para qualquer inicialização do tipo de dados C, existe uma inicialização correspondente para o tipo de dados A.

5.2 MODELOS

O auxílio de um provador de teoremas é muito importante para garantir a corretude de uma demonstração. Entretanto, *OhCircus* ainda não possui tal ferramenta. Por outro lado, *OhCircus* compartilha a mesma teoria de refinamentos de Z, e este possui o suporte do Z/EVES [Saa03]. Por isso, preferimos adotar Z para a confecção do nosso modelo.

No decorrer desta seção, apresentaremos os dois diagramas de classes referentes aos modelos abstrato e concreto. Em nosso mapeamento, cada um desses diagramas de classes introduz uma classe *Model* na especificação. Estas classes vão servir como estados das especificações e suas (meta-) operações é quem vão ser refinadas.

As operações identificadas para a classe *Model* alteram o conjunto de instâncias e associações de um diagrama através da adição e remoção de elementos, numa abordagem semelhante àquela de [LB98]. Entretanto, como a representação dos conjuntos de instâncias e suas respectivas operações não mudam de um diagrama para outro, é trivial provar o seu refinamento. Portanto, estamos interessados apenas nas operações de adicionar e remover um par de uma associação.

$$[A, B]$$

Inicialmente, desejamos estabelecer que a estrutura de A e B são arbitrárias. Ao abstrair a estrutura das classes, tornamos a formalização mais geral.

Modelo abstrato

O modelo abstrato é bastante simples. Ele contém duas classes e uma associação entre elas. Para ser o mais abrangente possível, nenhuma restrição será imposta às classes ou à associação. A Figura 5.2 apresenta este diagrama.

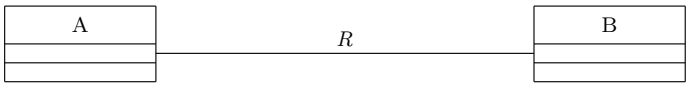

Figura 5.2. Diagrama de classes abstrato.

ModelR

$$iA : \mathbb{P}\, A$$
$$iB : \mathbb{P}\, B$$
$$R : A \leftrightarrow B$$

$$\mathrm{dom}\, R \subseteq iA \land \mathrm{ran}\, R \subseteq iB$$

Este esquema representa a classe modelo do diagrama anterior, com algumas pequenas mudanças: iA e iB representam os conjuntos de instâncias de A e B respectivamente, enquanto R representa a relação entre eles. Note a ausência dos papéis da associação; eles serão introduzidos posteriormente.

$$
\begin{array}{|l}
\hline
AddR \\
\Delta ModelR \\
a? : A \\
b? : B \\
\hline
a? \in iA \land b? \in iB \\
(a?, b?) \notin R \\
\\
R' = R \cup \{(a?, b?)\} \\
iA = iA' \land iB = iB' \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
RemR \\
\Delta ModelR \\
a? : A \\
b? : B \\
\hline
a? \in iA \land b? \in iB \\
(a?, b?) \in R \\
\\
R' = R \setminus \{(a?, b?)\} \\
iA = iA' \land iB = iB' \\
\hline
\end{array}
$$

As operações *AddR* e *RemR* representam as duas possíveis formas de interação dos objetos com a associação: é possível adicionar ou remover *links*. No modelo abstrato, isto é representado através da união e subtração de pares de uma relação. Note a relação de pertinência entre os parâmetros e os conjuntos de instâncias.

Modelo concreto

O modelo concreto modifica a representação da associação, agora capturada por meio de atributos e invariantes. A Figura 5.3 apresenta este diagrama.

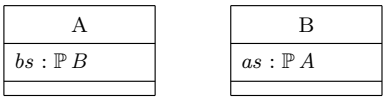

Figura 5.3. Diagrama de classes concreto.

ModelA

$$
\begin{array}{l}
iA : \mathbb{P}\, A \\
iB : \mathbb{P}\, B \\
as : B \pfun \mathbb{P}\, A \\
bs : A \pfun \mathbb{P}\, B \\
\hline
\mathrm{dom}\, as = iB \land \mathrm{ran}\, as \subseteq \mathbb{P}\, iA \\
\mathrm{dom}\, bs = iA \land \mathrm{ran}\, bs \subseteq \mathbb{P}\, iB \\
\forall\, a : iA;\ b : iB \bullet b \in bs\ a \Leftrightarrow a \in as\ b
\end{array}
$$

O modelo concreto introduz "atributos" nas classes A e B através das funções as e bs. Esta é única representação possível, dado que Z não permite esquemas mutuamente recursivos. Mas note a equivalência de representações entre $a.bs$ e $bs\ a$. Novamente, estes "atributos" devem estar relacionados aos conjuntos de instâncias de A e B. A última linha do invariante estabelece a consistência da associação via atributos: se o par (a, b) pertence à associação, então $a \in b.as$ se e somente se $b \in a.bs$.

AddA

$$
\begin{array}{l}
\Delta ModelA \\
a? : A \\
b? : B \\
\hline
a? \in iA \land b? \in iB \\
a? \notin as\ b? \land b? \notin bs\ a? \\
\\
bs' = bs \oplus \{a? \mapsto (bs\ a? \cup \{b?\})\} \\
as' = as \oplus \{b? \mapsto (as\ b? \cup \{a?\})\} \\
iA' = iA \land iB' = iB
\end{array}
$$

RemA

$$
\begin{array}{l}
\Delta ModelA \\
a? : A \\
b? : B \\
\hline
a? \in iA \land b? \in iB \\
b? \in bs\ a? \land a? \in as\ b? \\
\\
bs' = bs \oplus \{a? \mapsto (bs\ a? \setminus \{b?\})\} \\
as' = as \oplus \{b? \mapsto (as\ b? \setminus \{a?\})\} \\
iA' = iA \land iB' = iB
\end{array}
$$

As operações que alteram a associação foram modificadas para comportar a nova representação de dados. Só aquelas instâncias para as quais o *link* está sendo adicionado (ou removido) é que têm seus "atributos" atualizados. Os demais elementos permanecem inalterados.

5.3 PROVA DO REFINAMENTO

Para mostrar que o modelo concreto refina o abstrato, é necessário estabelecer o invariante de acoplamento que relaciona ambos os estados e provar que ele representa uma relação de simulação, através da demonstração dos teoremas da aplicabilidade e corretude para todas as operações.

Retrieve

$$ModelR$$
$$ModelA$$

$$\forall\, a : iA \bullet bs\ a = R\ (\!|\ \{a\}\ |\!)$$
$$\forall\, b : iB \bullet as\ b = R^{\sim}\ (\!|\ \{b\}\ |\!)$$

O nosso *Retrieve* é justamente aquilo que propomos para o mapeamento dos papéis de uma associação como atributos: o conjunto de elementos aos quais uma instância está associada via uma relação. Esta relação está em conformidade também com aquilo estabelecido na especificação de UML [OMG03c]

Demonstração

A prova do refinamento foi realizada utilizando o Z/EVES 2.3 [Saa03] e seus passos podem ser vistos no Capítulo B. Todavia, nesta seção vamos dar uma intuição da corretude da mesma. Em particular, vamos focar no teorema da corretude para a operação *Add*. As outras demonstrações exigem um raciocínio semelhante.

Teorema 5.1

$$\forall\, ModelR;\ ModelA;\ ModelA' \bullet$$
$$\text{pre}\ AddR \land Retrieve \land AddA$$
$$\Rightarrow (\exists\, ModelR' \bullet AddR \land Retrieve')$$

Como apresentado anteriormente, este teorema define que aplicar a operação abstrata leva a um estado semelhante ao da execução da concreta (*modulo Retrieve*).

Os passos manuais são bastante diretos: expandir os esquemas, aplicar a *one point rule* três vezes, efetuar uma pequena manipulação de predicados (como aplicar a lei *domCup*, da *toolkit* matemática do Z/EVES [Saa03]) e eliminar as proposições triviais (as proposições que estavam presentes tanto no antecedente quanto no conseqüente). Isso nos deixou, essencialmente, com o seguinte teorema, onde as variáveis livres estão quantificadas universalmente:

$$
\begin{array}{l}
(\forall\, a : iA \bullet bs\ a = R\ (\!|\ \{a\}\ |\!)) \\
\quad \land\ (\forall\, b : iB \bullet as\ b = R^{\sim}\ (\!|\ \{b\}\ |\!)) \\
\quad \land\ as' = as \oplus \{b? \mapsto (as\ b? \cup \{a?\})\} \\
\quad \land\ bs' = bs \oplus \{a? \mapsto (bs\ a? \cup \{b?\})\} \\
\Rightarrow (\forall\, a : iA \bullet bs'\ a = R'\ (\!|\ \{a\}\ |\!)) \\
\quad \land\ (\forall\, b : iB \bullet as'\ b = R'^{\sim}\ (\!|\ \{b\}\ |\!))
\end{array}
$$

Reescrevendo os objetivos de prova baseado na equivalência

$$(\forall\, x : A \cup B \bullet P(x)) \Leftrightarrow (\forall\, x : A \bullet P(x)) \land (\forall\, x : B \bullet P(x)) :$$

$$
\begin{array}{l}
\ldots \\
\Rightarrow (\forall\, a : iA \setminus \{a?\} \bullet bs'\ a = R'\ (\!|\ \{a\}\ |\!)) \\
\quad \land\ (\forall\, a : \{a?\} \bullet bs'\ a = R'\ (\!|\ \{a\}\ |\!)) \\
\quad \land\ (\forall\, b : iB \setminus \{b?\} \bullet as'\ b = R'^{\sim}\ (\!|\ \{b\}\ |\!)) \\
\quad \land\ (\forall\, b : \{b?\} \bullet as'\ b = R'^{\sim}\ (\!|\ \{b\}\ |\!))
\end{array}
$$

Eliminando o quantificador universal para um único elemento:

$$
\begin{array}{l}
\ldots \\
\Rightarrow (\forall\, a : iA \setminus \{a?\} \bullet bs'\ a = R'\ (\!|\ \{a\}\ |\!)) \\
\quad \land\ bs'\ a? = R'\ (\!|\ \{a?\}\ |\!) \\
\quad \land\ (\forall\, b : iB \setminus \{b?\} \bullet as'\ b = R'^{\sim}\ (\!|\ \{b\}\ |\!)) \\
\quad \land\ as'\ b? = R'^{\sim}\ (\!|\ \{b?\}\ |\!))
\end{array}
$$

Por fim é fácil concluir que a primeira linha dos objetivos de prova representa exatamente $\forall\, a : iA \setminus \{a?\} \bullet R\ (\!|\ \{a\}\ |\!) = R'\ (\!|\ \{a\}\ |\!)$, enquanto a segunda linha mostra a atualização do parâmetro de entrada: $bs'\ a? = \{a? \mapsto (bs\ a? \cup \{b?\})\}$. O raciocínio é análogo para os outros casos.

CAPÍTULO 6

CONCLUSÕES

Neste trabalho consideramos a formalização de UML usando a linguagem de especificação formal *OhCircus*. Apesar de não ser exaustiva, nós abordamos as mais importantes construções de UML relativas à modelagem estática.

Inicialmente, apresentamos as principais construções de ambas as linguagens. Pudemos discutir a (interpretação da) semântica proposta pela OMG para UML, analisando-a através de um exemplo. Em seguida, introduzimos os principais elementos de *OhCircus*.

Uma das contribuições deste trabalho é a tradução dos diagramas de classes de UML para *OhCircus*. Preferimos abordar as linguagens como se estivessem no mesmo nível semântico, mapeando sintaticamente o maior número de elementos de UML em construções de *OhCircus*. Acreditamos que esta alternativa seja mais promissora, uma vez que *OhCircus* é uma linguagem orientada a objetos, não havendo necessidade de (re)definir conceitos como classes e herança.

Contudo, UML define alguns elementos que não estão disponíveis em *OhCircus*. A proposta de uma classe *modelo* surgiu então como uma interessante adição a este mapeamento. Trazer a "semântica" de um modelo orientado a objetos (i.e. seus conjuntos de objetos, interações e restrições) para a própria especificação mostrou-se valioso, visto que agora é possível capturar naturalmente associações, invariantes globais e até mesmo aspectos dinâmicos.

A outra contribuição desta monografia é a análise de refinamento em UML. Em particular, a classe-modelo permitiu que explorássemos a mesma teoria de refinamentos de Z, onde há um único estado (global) e operações que atuam sobre ele; graças a isso, tivemos o suporte de ferramentas. O estudo de caso foi a tão discutida representação de associações como atributos, e nossa contribuição foi mostrar que a abordagem com atributos é um refinamento da primeira.

6.1 TRABALHOS FUTUROS

Diversos são os trabalhos futuros que este trabalho pode dar origem; a grande parte deles está diretamente relacionada à extensão deste mapeamento, contemplando aspectos ainda não explorados de UML. Outra parte está relacionada à utilização deste mapeamento como base formal para a

construção de transformações de modelos.

- A extensão mais imediata deste trabalho é contemplar os demais elementos estáticos que não foram capturados, como classes abstratas, interfaces, alguns tipos de modificadores de associações etc. Ainda nesta linha, estudar a possibilidade de utilizar OCL, uma linguagem para expressar restrições, para anotar os modelos UML.

- Também cabe aqui verificar a possibilidade de "inverter" o mapeamento, examinando como levar uma especificação nos moldes deste trabalho (e até mesmo arbitrária) para UML. Isso será importante quando houver um suporte ferramental para *OhCircus*, onde o modelo poderá ser analisado formalmente, mas seus resultados serem expressos em termos de UML.

- O estudo dos aspectos dinâmicos de UML através da classe-modelo é um trabalho relacionado bastante interessante: analisar quais as *seqüências* de instâncias do diagrama são válidas e a possibilidade de prover invariantes dinâmicos são apenas alguns dos itens que devem ser abordados.

- Concorrência em UML através de *OhCircus* já vem particularmente sendo explorada através do *profile* de *real-time* [SMR03], com resultados bastante promissores.

- Trabalhos relacionados a refinamentos também são importantes. A demonstração formal, baseada neste mapeamento, da validade dos padrões de projeto é um exemplo de importante contribuição à área. Outros refinamentos podem ser explorados, como a transformação de associações bidirecionais em unidirecionais, a inclusão ou remoção de uma classe do modelo etc, propondo, por exemplo, um conjunto de leis de transformação de modelos para UML.

6.2 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos as nossas principais contribuições e possíveis trabalhos relacionados a este. É importante destacar que a principal aplicação desta integração é justamente oferecer uma base formal a UML, permitindo a análise de seus modelos e a demonstração de técnicas ou resultados outrora aplicados informalmente.

A originalidade (até então conhecida) desta abordagem, onde unimos diversos elementos presentes isoladamente em outras propostas, e também nossa contribuição para respaldar o uso de associações e sua representação como atributos são os pontos-chaves deste trabalho.

As vantagens de aplicar Métodos Formais foram bastante clarificadas por este estudo de caso. A representação de associações em atributos é bastante debatida, uma vez que não tem uma semântica bem definida. O uso de Métodos Formais permitiu estabelecermos uma base formal onde ambas as representações puderam ser comparadas e, conseqüentemente, validadas através de provas.

O uso do Z/EVES foi fundamental para a demonstração. O respaldo de uma ferramenta já consolidada é muito importante para dar maior credibilidade ao trabalho. Infelizmente, seu uso não é muito amigável, sendo necessário um pouco de prática para utilizá-lo de maneira eficiente.

APÊNDICE A

MAPEAMENTO COMPLETO

$[Char]$

$String == \text{seq}\ Char$

$Gender ::= Gender\_Male \mid Gender\_Female$

class $Model \mathrel{\widehat=}$ **begin**

state *ModelState*

$$
\begin{array}{l}
accounts : \mathbb{P}\, Account \\
banks : \mathbb{P}\, Bank \\
creditAccounts : \mathbb{P}\, CreditAccount \\
jobs : \mathbb{P}\, Job \\
persons : \mathbb{P}\, Person \\
weddings : \mathbb{P}\, Wedding \\
\\
employment : (Person \times Bank) \pfun Job \\
has : Bank \rel Account \\
management : Person \rel Bank \\
marriage : (Person \times Person) \pfun Wedding \\
owns : Person \rel Account \\
\hline
creditAccounts \subseteq accounts \\
\\
employment \in (persons \times banks) \pfun jobs \\
has \in banks \rel accounts \\
management \in persons \rel banks \\
marriage \in (persons \times persons) \pfun weddings \\
owns \in persons \rel accounts \\
\mathrm{ran}\, employment = jobs \\
\mathrm{ran}\, marriage = weddings \\
\\
\forall\, a : accounts \bullet \\
\quad a.bank = has^{\sim}(\!|\, \{a\} \,|\!) \land \\
\quad a.owners = owns^{\sim}(\!|\, \{a\} \,|\!) \land \\
\quad \#(has^{\sim}(\!|\, \{a\} \,|\!)) \in 0..1 \land \\
\quad \#(owns^{\sim}(\!|\, \{a\} \,|\!)) \in \mathbb{N}_1 \\
\\
\forall\, b : banks \bullet \\
\quad \mathrm{ran}\, b.accounts = has(\!|\, \{b\} \,|\!) \land \\
\quad b.employee = (\mathrm{dom}\, employment)^{\sim}(\!|\, \{b\} \,|\!) \land \\
\quad b.manager = management^{\sim}(\!|\, \{b\} \,|\!) \land \\
\quad b.job = employment(\!|\, persons \times \{b\} \,|\!) \land \\
\quad \#(management^{\sim}(\!|\, \{b\} \,|\!)) \in 0..1 \\
\\
\forall\, p : persons \bullet \\
\quad p.accounts = owns(\!|\, \{p\} \,|\!) \land \\
\quad p.employer = (\mathrm{dom}\, employment)(\!|\, \{p\} \,|\!) \land \\
\quad p.husband = (\mathrm{dom}\, marriage)(\!|\, \{p\} \,|\!) \land \\
\quad p.job = employment(\!|\, \{p\} \times banks \,|\!) \land \\
\quad p.managed = management(\!|\, \{p\} \,|\!) \land \\
\quad p.wedding\_husband = marriage(\!|\, \{p\} \times persons \,|\!) \land \\
\quad p.wedding\_wife = marriage(\!|\, persons \times \{p\} \,|\!) \land \\
\quad p.wife = (\mathrm{dom}\, marriage)^{\sim}(\!|\, \{p\} \,|\!) \land \\
\quad \#((\mathrm{dom}\, marriage)(\!|\, \{p\} \,|\!)) \in 0..1 \land \\
\quad \#((\mathrm{dom}\, marriage)^{\sim}(\!|\, \{p\} \,|\!)) \in 0..1
\end{array}
$$

end

class *Person* $\widehat{=}$ **begin**

state *PersonState*

$name : String$
$age : \mathbb{Z}$
$gender : Gender$

$accounts : \mathbb{P}\ Account$
$managed : \mathbb{P}\ Bank$
$employer : \mathbb{P}\ Bank$
$job : \mathbb{P}\ Job$
$husband : \mathbb{P}\ Person$
$wedding\_husband : \mathbb{P}\ Wedding$
$wedding\_wife : \mathbb{P}\ Wedding$
$wife : \mathbb{P}\ Person$

end

class *Bank* $\widehat{=}$ **begin**

state *BankState*

$accounts : \mathbb{Z} \rightarrowtail Account$
$employee : \mathbb{P}\ Person$
$manager : \mathbb{P}\ Person$
$job : \mathbb{P}\ Job$

end

class *Account* $\widehat{=}$ **begin**

state *AccountState*

$$\begin{array}{l} \textbf{protected}\ number : \mathbb{Z} \\ \textbf{protected}\ balance : \mathbb{Z} \end{array}$$

initial *AccountInit*

$$\begin{array}{l} AccountState' \\ number? : \mathbb{Z} \\ \hline number' = number? \\ balance' = 0 \end{array}$$

public *Deposit*

$$\begin{array}{l} \Delta AccountState \\ amount? : \mathbb{N} \\ \hline balance' = balance + amount? \\ number' = number \end{array}$$

public *Withdraw*

$$\begin{array}{l} \Delta AccountState \\ amount? : \mathbb{N} \\ \hline amount? \leq balance \\ balance' = balance - amount? \\ number' = number \end{array}$$

logical *GetNumber*

$$\begin{array}{l} \Xi AccountState \\ n! : \mathbb{N} \\ \hline n! = number \end{array}$$

end

class *CreditAccount* $\widehat{=}$ **extends** *Account* **begin**

state *CreditAccountState*

$$\begin{array}{|l}
\textbf{private } credit : \mathbb{Z} \\
\hline
balance + credit \geq 0 \\
\hline
\end{array}$$

public *Withdraw*

$$\begin{array}{|l}
\Delta CreditAccount \\
amount? : \mathbb{N} \\
\hline
amount? \leq balance + credit \\
balance' = balance - amount? \\
number' = number \\
\hline
\end{array}$$

public *Deposit*

$$\begin{array}{|l}
\Delta CreditAccountState \\
amount? : \mathbb{N} \\
\hline
balance' = balance + amount? \\
number' = number \\
credit' = credit \\
\hline
\end{array}$$

public *SetCredit*

$$\begin{array}{|l}
\Delta CreditAccountState \\
\Xi AccountState \\
credit? : \mathbb{N} \\
\hline
credit' = credit? \\
\hline
\end{array}$$

end

APÊNDICE B

PROVA DO REFINAMENTO

Neste capítulo, apresentamos as provas de corretude do refinamento através de *scripts de prova* do Z/EVES 2.3 [Saa03].

B.1 ADD – APLICABILIDADE

theorem AddAppl
$\forall\, ModelR;\ ModelA \bullet$
pre $AddR \land Retrieve \Rightarrow$ pre $AddA$

proof[*AddAppl*]
prove by reduce;
apply cupSubsetLeft to expression $\{a?\} \cup \mathrm{dom}\ bs$;
apply cupSubsetLeft to expression $\{b?\} \cup \mathrm{dom}\ as$;
prove by reduce;
instantiate $b\_\_1 == b?$;
instantiate $a\_\_1 == a?$;
prove by reduce;
apply inImage to predicate $a? \in R^{\sim} (\!|\ \{b?\}\ |\!)$;
apply inImage to predicate $b? \in R (\!|\ \{a?\}\ |\!)$;
prove by reduce;
apply oplusDef to expression $bs \oplus \{(a?, (\{b?\} \cup (R (\!|\ \{a?\}\ |\!))))\}$;
apply oplusDef to expression $as \oplus \{(b?, (\{a?\} \cup (R^{\sim} (\!|\ \{b?\}\ |\!))))\}$;
prove by reduce;
use ranSubset$[B, \mathbb{P}\ A][S := as,\ R := \{b?\} \lhd as]$;
use ranSubset$[A, \mathbb{P}\ B][S := bs,\ R := \{a?\} \lhd bs]$;
prove by reduce;
apply applyCupLeft to expression $(\{a?\} \lhd bs \cup$
$\{(a?, (\{b?\} \cup (R (\!|\ \{a?\}\ |\!))))\})\ a$;
apply applyCupLeft to expression $(\{b?\} \lhd as \cup$
$\{(b?, (\{a?\} \cup (R^{\sim} (\!|\ \{b?\}\ |\!))))\})\ b$;
instantiate $a\_\_0 == a,\ b\_\_0 == b$;
prove by reduce;
■

B.2 ADD – CORRETUDE

theorem distributeImageOverCup $[A, B]$
$\forall\, Q, R : A \leftrightarrow B;\ S : \mathbb{P}\, A \bullet (Q \cup R)(\!|\ S\ |\!) = (Q(\!|\ S\ |\!)) \cup (R(\!|\ S\ |\!))$

proof[*distributeImageOverCup*]
use imageDef$[A,\ B][R := Q\ \cup\ [$**local** $A\ \times$ **local** $B]\ R]$;
prove by reduce;
use imageDef$[A,\ B][R := Q]$;
use imageDef$[A,\ B]$;
prove by reduce;
■

theorem AddCorr
$\forall\, ModelR;\ ModelA;\ ModelA' \bullet$
pre $AddR \wedge Retrieve \wedge AddA \Rightarrow$
$(\exists\, ModelR' \bullet AddR \wedge Retrieve')$

proof[*AddCorr*]
prove by reduce;
cases;
apply oplusDef to expression $bs \oplus \{(a?,\ (bs\ a? \cup \{b?\}))\}$;
prove by reduce;
apply applyCupRight to expression $(\{(a?,\ bs\ a? \cup \{b?\})\} \cup \{a?\} \lhd bs)\ a$;
prove by reduce;
instantiate $a\_\_2 == a?$;
prove by reduce;
use distributeImageOverCup$[A,\ B][Q := \{(a?,\ b?)\},\ S := \{a\}]$;
prove by reduce;
instantiate $a\_\_2 == a$;
prove by reduce;
next;
apply oplusDef to expression $as \oplus \{(b?,\ (as\ b? \cup \{a?\}))\}$;
prove by reduce;
apply applyCupRight to expression $(\{(b?,\ as\ b? \cup \{a?\})\} \cup \{b?\} \lhd as)\ b$;
prove by reduce;
instantiate $b\_\_2 == b?$;
prove by reduce;
use distributeImageOverCup$[B,\ A][Q := \{(b?,\ a?)\},\ R := R^{\sim},\ S := \{b\}]$;
instantiate $b\_\_2 == b$;
prove by reduce;
next;
■

B.3 REM – APLICABILIDADE

theorem RemAppl
$\forall\ ModelR;\ ModelA \bullet$
pre $RemR \land Retrieve \Rightarrow$ pre $RemA$

proof[*RemAppl*]
prove by reduce;
apply cupSubsetLeft to expression $\{a?\} \cup \text{dom } bs$;
apply cupSubsetLeft to expression $\{b?\} \cup \text{dom } as$;
prove by reduce;
instantiate $b\_\_1 == b?$;
instantiate $a\_\_1 == a?$;
prove by reduce;
apply inImage to predicate $a? \in R \sim (\!| \{b?\} |\!)$;
apply inImage to predicate $b? \in R (\!| \{a?\} |\!)$;
prove by reduce;
apply oplusDef to expression $as \oplus \{(b?, ((R \sim (\!| \{b?\} |\!)) \setminus \{a?\}))\}$;
apply oplusDef to expression $bs \oplus \{(a?, ((R (\!| \{a?\} |\!)) \setminus \{b?\}))\}$;
prove by reduce;
use ranSubset$[B, \mathbb{P} A][S := as, R := \{b?\} \lhd as]$;
use ranSubset$[A, \mathbb{P} B][S := bs, R := \{a?\} \lhd bs]$;
prove by reduce;
apply applyCupLeft to expression $(\{a?\} \lhd bs \cup \{(a?, ((R (\!| \{a?\} |\!)) \setminus \{b?\}))\})\ a$;
apply applyCupLeft to expression $(\{b?\} \lhd as \cup \{(b?, ((R \sim (\!| \{b?\} |\!)) \setminus \{a?\}))\})\ b$;
instantiate $a\_\_0 == a,\ b\_\_0 == b$;
prove by reduce;
■

B.4 REM – CORRETUDE

theorem distributeDresOverSetminus $[A, B]$
$\forall Q, R : A \leftrightarrow B;\ S : \mathbb{P} A \bullet S \lhd (Q \setminus R) = (S \lhd Q) \setminus (S \lhd R)$

proof[*distributeDresOverSetminus*]
apply extensionality to predicate $S \lhd [\textbf{local } A, \textbf{local } B]\ (Q \setminus [(\textbf{local } A \times \textbf{local } B)]\ R) = S \lhd [\textbf{local } A, \textbf{local } B]\ Q \setminus [\textbf{local } A \times \textbf{local } B]\ S \lhd [\textbf{local } A, \textbf{local } B]\ R$;
prove by reduce;
■

theorem distributeUnitImageOverSetminus $[A, B]$
$\forall R : A \leftrightarrow B;\ a : A;\ b : B \bullet (R(\!| \{a\} |\!)) \setminus \{b\} = (R \setminus \{(a, b)\})(\!| \{a\} |\!)$

proof[*distributeUnitImageOverSetminus*]
use imageDef$[A,\ B][R := R \setminus [\mathbf{local}\ A \times \mathbf{local}\ B]\ \{(a,\ b)\},\ S := \{a\}]$;
use imageDef$[A,\ B][S := \{a\}]$;
prove by reduce;
use distributeDresOverSetminus$[A,\ B][Q := R,\ R := \{(a,\ b)\},\ S := \{a\}]$;
prove by reduce;
apply extensionality to predicate $\mathrm{ran}\ [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (\{a\} \lhd [\mathbf{local}\ A,\ \mathbf{local}\ B]\ R) \setminus [\mathbf{local}\ B]\ \{b\} = \mathrm{ran}\ [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (\{a\} \lhd [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (R \setminus [(\mathbf{local}\ A \times \mathbf{local}\ B)]\ \{(a,\ b)\}))$;
prove by reduce;
cases;
apply inRan to predicate $x \in \mathrm{ran}\ [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (\{a\} \lhd [\mathbf{local}\ A,\ \mathbf{local}\ B]\ R)$;
prove by reduce;
apply inRan to predicate $x \in \mathrm{ran}\ [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (\{a\} \lhd [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (R \setminus [(\mathbf{local}\ A \times \mathbf{local}\ B)]\ \{(a,\ b)\}))$;
prove by reduce;
next;
apply inRan to predicate $y \in \mathrm{ran}\ [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (\{a\} \lhd [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (R \setminus [(\mathbf{local}\ A \times \mathbf{local}\ B)]\ \{(a,\ b)\}))$;
prove by reduce;
apply inRan to predicate $y \in \mathrm{ran}\ [\mathbf{local}\ A,\ \mathbf{local}\ B]\ (\{a\} \lhd [\mathbf{local}\ A,\ \mathbf{local}\ B]\ R)$;
prove by reduce;
next;
■

theorem nullSetminusImage $[A, B]$
$\forall R : A \leftrightarrow B;\ a, a' : A;\ b : B \mid a \neq a' \bullet (R(\!|\ \{a\}\ |\!)) = (R \setminus \{(a', b)\})(\!|\ \{a\}\ |\!)$

proof[*nullSetminusImage*]
use imageDef$[A,\ B][R := R \setminus [\mathbf{local}\ A \times \mathbf{local}\ B]\ \{(a', b)\},\ S := \{a\}]$;
use imageDef$[A,\ B][S := \{a\}]$;
prove by reduce;
use distributeDresOverSetminus$[A,\ B][Q := R,\ R := \{(a',\ b)\},\ S := \{a\}]$;
prove by reduce;
■

theorem RemCorr
$\forall\, ModelR;\ ModelA;\ ModelA' \bullet$
pre $RemR \land Retrieve \land RemA \Rightarrow$
$(\exists\, ModelR' \bullet RemR \land Retrieve')$

proof[*RemCorr*]
prove by reduce;
cases;
apply oplusDef to expression $bs \oplus \{(a?,\ (bs\ a? \setminus \{b?\}))\}$;
prove by reduce;
apply applyCupRight to expression $(\{(a?,\ bs\ a? \setminus \{b?\})\} \cup \{a?\} \lhd bs)\ a$;
prove by reduce;
instantiate $a\_\_2 == a?$;
instantiate $a\_\_2 == a$;
use distributeUnitImageOverSetminus$[A,\ B][a := a?,\ b := b?]$;
use nullSetminusImage$[A,\ B][a' := a?,\ b := b?]$;
prove by reduce;
next;
apply oplusDef to expression $as \oplus \{(b?,\ (as\ b? \setminus \{a?\}))\}$;
prove by reduce;
apply applyCupRight to expression $(\{(b?,\ as\ b? \setminus \{a?\})\} \cup \{b?\} \lhd as)\ b$;
prove by reduce;
instantiate $b\_\_2 == b?$;
instantiate $b\_\_2 == b$;
use distributeUnitImageOverSetminus$[B,\ A][R := R^{\sim},\ a := b?,\ b := a?]$;
use nullSetminusImage$[B,\ A][R := R^{\sim},\ a := b,\ a' := b?,\ b := a?]$;
prove by reduce;
next;
■

REFERÊNCIAS BIBLIOGRÁFICAS

[BH95] J. Bowen and M. Hinchey. *Applications of Formal Methods*. Prentice Hall PTR, 1995.

[BHH+97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In *Proceedings of ECOOP'97*. Springer Verlag, LNCS, 1997.

[Boo91] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin–Cummings, Redwood City, Calif., 1st edition, 1991.

[Bro87] F. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer Magazine*, 20(4):10–19, 1987.

[Bro95] F. Brooks, Jr. *The Mythical Man-Month (Anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[BS00] P. Borba and A. Sampaio. The basic laws of rool: An object-oriented language, September 2000.

[Cla99] T. Clark. Type checking UML static diagrams. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 503–517. Springer, 1999.

[CSW03] A. Cavalcanti, A. Sampaio, and J. Woodcock. A Unified Language of Classes and Processes. In *St Eve: State-Oriented vs. Event-Oriented Thinking in Requirements Analysis, Formal Specification and Software Engineering*, Satellite Workshop at FM'03, unknown 2003.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 1996.

[Dij72] E. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Dij97] E. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1997.

[EC97] Andy Evans and Tony Clark. Foundations of the Unified Modeling Language. In *Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, K, 23-24 September 1997*, 1997.

[Eva98] A. Evans. Reasoning with UML Class Diagrams. In *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*. IEEE, 1998.

[FEL97] Robert B. France, Andy Evans, and Kevin Lano. The UML as a formal modeling notation. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.

[FG03] Ana María Funes and Chris George. Formalizing UML class diagrams, 2003.

[Fis98] Clemens Fischer. How to combine z with process algebra. In *Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation*, pages 5–23. Springer-Verlag, 1998.

[GB03] R. Gheyi and P. Borba. Refactoring alloy specifications. In *Sixth Brazilian Workshop on Formal Methods*, pages 166–181, Campina Grande, Brazil, October 2003.

[GBHS97] I. Graham, J. Bischof, and B. Henderson-Sellers. Associations considered a bad thing. *Journal of Object Oriented Programming*, 9(9):41–48, February 1997.

[Ghe04] R. Gheyi. Basic laws of object modeling. Master's thesis, Federal University of Pernambuco, 2004.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gib94] W. Gibbs. Software's Chronic Crisis. *Scientific American (International ed.)*, 271(3):86–95, 1994.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[Gén01] Gonzalo Génova. Semantics of navigability in uml associations. Technical Report UC3M-TR-CS-2001-06, Computer Science Department, Carlos III University of Madrid, November 2001.

[Gol01] M. Goldsmith. *FDR: User Manual and Tutorial, version 2.77*. Formal Systems (Europe) Ltd, 2001.

[Hei96] M. Heimdahl. Experiences and lessons from the analysis of tcas ii. *SIGSOFT Softw. Eng. Notes*, 21(3):79–83, 1996.

[Hoa84] C. Hoare. Programming: Sorcery or Science? *IEEE Software*, 1(2):5–12, 15–16, April 1984.

[Jac02] D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Technical report, Software Design Group, MIT Lab for Computer Science, 2002.

[JCJv92] I. Jacobson, M. Christerson, P. Jonsonn, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[JM97] J.-M. Jézéquel and B. Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(2):129–130, January 1997.

[KC00] S. Kim and D. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. *Lecture Notes in Computer Science*, 1878:2–21, 2000.

[Kru00] P. Kruchten. *An Introduction to the Rational Unified Process*. Addison-Wesley, 2000.

[LB98] K. Lano and J. Bicarregui. UML Refinement and Abstraction Transformations. *ROOM 2 Workshop, Bradford University*, 1998.

[LCA04] Kevin Lano, David Clark, and Kelly Androutsopoulos. Uml to b: Formal verification of object-oriented models. In *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 187–206. Springer, 2004.

[LT93] N. Leveson and C. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, June 1993.

[LW93] B. Liskov and J. Wing. Family Values: A Behavioral Notion of Subtyping. Technical Report MIT/LCS/TR-562b, MIT, 1993.

[Lyo98] A. Lyons. UML for Real-Time Overview. Whitepaper, Objec-Time Limited, April 1998.

[MBM03] P. Moura, R. Borges, and A. Mota. Experimenting Formal Methods through UML. Submitted to WMF'2003, July 2003.

[Mor94] C. Morgan. *Programming from Specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., 1994.

[Mot97] Alexandre Cabral Mota. Formalização e anáise do saci-1 em csp-z. Master's thesis, Depto. de Informática, Universidade Federal de Pernambuco, September 1997.

[OMG03a] OMG. UML 2 Infrastructure Final Adopted Specifcation. Whitepaper, Object Management Group, September 2003.

[OMG03b] OMG. UML 2 OCL Final Adopted Specification. Whitepaper, Object Management Group, October 2003.

[OMG03c] OMG. UML 2 Superstructure Final Adopted specification. Whitepaper, Object Management Group, August 2003.

[OMG03d] OMG. Unified modeling language. Specification v1.5, Object Management Group, March 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[Pai99] R. Paige. Integrating a program design calculus and a subset of UML. *The Computer Journal*, 42(2), 1999.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice Hall, 1991.

[RBR03] D. Roe, K. Broda, and A. Russo. Mapping UML Models incorporating OCL Constraints into Object-Z. Technical Report 2003/9, Imperial College London, 2003.

[RHB97] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., 1999.

[RRS04] Franklin Ramalho, Jacques Robin, and Ulrich Schiel. Concurrent transaction frame logic formal semantics for uml activity and class diagrams. *Electronic Notes in Theoretical Computer Science*, 95:83–109, 2004.

[Rum96] J. Rumbaugh. A search for values: Attributes and associations. *Journal of Object Oriented Programming*, 9(3):6–8, June 1996.

[Saa99] M. Saaltink. The Z/EVES 2.0 User's Guide. Technical Report TR-99-5493-06a, ORA Canada, One Nicholas Street, Suite 1208 - Ottawa, Ontario K1N 7B7 - CANADA, October 1999.

[Saa03] M. Saaltink. The Z/EVES 2.2 Mathematical Toolkit. Technical Report TR-03-5493-05c, ORA Canada, June 2003.

[Smi92] G. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, October 1992.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[SMR03] A. Sampaio, A. Mota, and R. Ramos. Class and Capsule Refinement for UML-RT. In *WMF 2003: 6th Workshop on Formal Methods, Brazil*, pages 16–34, 2003. Extended version to appear in Electronic Notes in Theoretical Computer Science, Elsevier, 2004.

[Som02] I. Sommerville. *Software Engineering*. Addison-Wesley, 2002.

[Spi92] M. Spivey. *The Z Notation*. Prentice-Hall, 1992.

[SR98] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Whitepaper, Rational Software Corp., March 1998.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[Tan95] C. Tanzer. Remarks on object-oriented modeling of associations. *Journal of Object Oriented Programming*, 7(9):43–46, February 1995.

[Vel94] A. V. Velho. Attribute: A semantic and seamless construct. In B. Magnusson et al., editors, *TOOLS 13*. Prentice Hall, 1994.

[WC02] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

[Zep02] P. Zeppo. From UML to B Specifications. Master's thesis, Dept. of Computer Science, King's College London, 2002.