UM ESTUDO SOBRE A ENGENHARIA DE IDA E VOLTA ENTRE UML E JAVA

LUIS PAULO ALVES MAGALHÃES

UM ESTUDO SOBRE A ENGENHARIA DE IDA E VOLTA ENTRE UML E JAVA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Rodolfo Sérgio Ferreira de Resende
Coorientador: Antonio Maria Pereira de Resende

Belo Horizonte
Julho de 2011

Magalhães, Luis Paulo Alves.

M188e Um estudo sobre a engenharia de ida e volta entre UML e Java / Luis Paulo Alves Magalhães. — Belo Horizonte, 2011.
xviii, 122 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Rodolfo Sérgio Ferreira de Resende.
Coorientador: Antonio Maria Pereira de Resende.

1. Computação - Teses. 2. Engenharia de software - Teses. 3. UML (Linguagem de modelagem padrão) – Teses. 4. Java (Linguagem de programação de computador) – Teses. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Um estudo sobre a engenharia de ida e volta entre UML e java

LUIS PAULO ALVES MAGALHÃES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. RODOLFO SÉRGIO FERREIRA DE RESENDE - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ANTÔNIO MARIA PEREIRA DE RESENDE - Co-orientador
Departamento de Ciência da Computação - UFLA

PROF. CLARINDO ISAÍAS P. DA SILVA E PÁDUA
Departamento de Ciência da Computação - UFMG

PROF. HEITOR AUGUSTUS XAVIER COSTA
Departamento de Ciência da Computação - UFLA

Belo Horizonte, 29 de julho de 2011.

Resumo

No desenvolvimento de software, os modelos, dentre outros artefatos, podem facilitar o entendimento do software. Manter o código e os modelos consistentes entre si não é uma tarefa simples. Combinada com um processo iterativo e com as ferramentas adequadas, a engenharia de ida e volta permite que o código e o modelo permaneçam sincronizados. A UML tornou-se a representação gráfica mais presente em projetos de sistema de software orientado a objeto e a linguagem Java tornou-se uma das linguagens de programação mais utilizadas atualmente. Vários trabalhos no início dos anos 2000 discutiram a questão de navegar de UML para Java e de Java para UML, no contexto da teoria e das ferramentas CASE. Apesar da crescente popularidade, há pouca avaliação relatada sobre o uso do desenvolvimento baseado em UML. As duas tecnologias, UML e Java, evoluíram de lá pra cá e muitos trabalhos se tornaram obsoletos. As ferramentas CASE devem ser expostas uma avaliação adequada a fim de determinar se elas são eficazes de ajudar os usuários em sua meta. Este trabalho procurou avançar a discussão sobre o estado da arte da questão da engenharia de ida e volta entre as novas características da UML e as novas características da plataforma Java. Analisamos a transcrição do modelo para o código e vice-versa, e também a interação da ferramenta com o usuário (desenvolvedor de software) durante o mapeamento de UML para Java e vice-versa. A capacidade de transcrição das ferramentas de UML para Java não teve grandes mudanças, considerando trabalhos dos últimos anos. A interação da ferramenta com o usuário pode ser melhorada.

Palavras-chave: Engenharia de Ida e Volta, Engenharia Reversa, Ferramentas Case, Mapeamento Modelo-Código.

Abstract

In software development, models, among other artifacts, can facilitate the understanding of the software. To Keep the code and models consistent with each other is not a simple task. Combined with an iterative process and with the right tools, forward and backward engineering allows code and model to stay synchronized. The UML has become the standard for graphical representation of object-oriented software systems design and the Java language has become one of the most widely used programming languages. Several studies in the early 2000 discussed the issue of how to translate from UML for Java and from Java to UML in the context of theory and CASE tools. The two languages, UML and Java, have evolved and this opens space for new studies. CASE tools should be exposed to a proper assessment to determine whether they are effective in helping users in their goals. This dissertation discusses some aspects related to the round trip engineering between UML and the Java language. The capability of tools transcription of UML for Java did not have major changes, considering the work of recent years. The interaction with the user of the tool can be improved.

Keywords: Round-Trip Engineering, Reverse Engineering, Case Tools, Model-Code Mapping

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

1.1 Contextualização e Motivação

No desenvolvimento de software, os modelos, dentre outros artefatos, podem facilitar o entendimento. De acordo com Booch e outros [Booch et al., 2005], modelos são desenvolvidos para facilitar o entendimento do sistema que se está desenvolvendo, uma vez que permitem trabalhar em um nível mais alto de abstração. Abstração é uma habilidade humana fundamental que permite o projetista lidar com a complexidade [Khaled, 2009].

Modelagem consiste em construir uma abstração de um sistema, concentrando-se em aspectos de maior interesse e ignorando detalhes irrelevantes. Geralmente, a modelagem se concentra na construção de um modelo que seja simples o suficiente para que uma pessoa possa compreendê-lo completamente [Bruegge & Dutoit, 2009]. A modelagem é um meio para lidar com a complexidade por permitir, de forma incremental, refinar os modelos simples para outros mais detalhados que estão próximos da realidade.

Conforme Booch e outros [Booch et al., 2005], com a modelagem, quatro objetivos são alcançados:

- Os modelos ajudam a visualizar o sistema como ele é ou como se deseja que seja;
- Os modelos permitem especificar a estrutura e o comportamento de um sistema;
- Os modelos proporcionam um guia para a construção do sistema;
- Os modelos permitem documentar as decisões tomadas.

A utilização de modelos no desenvolvimento de software pode ter outras finalidades além de facilitar o entendimento do sistema ou a comunicação entre os envolvidos na construção do sistema. Modelos abstratos de sistemas de software são criados e sistematicamente transformados para implementações concretas [France & Rumpe, 2007]. É o estado da arte fornecer a modelagem e a geração de código a partir dos modelos [Gessenharter, 2008].

De acordo com France e Rumpe [France & Rumpe, 2007], o tempo que se gasta construindo os modelos pode ser compensado com o tempo que seria gasto na construção de várias linhas de código, uma vez que este pode ser gerado sistematicamente a partir dos modelos. Por esta razão, a abordagem de Desenvolvimento Dirigido por Modelos (*Model-Driven Development*) tem crescido nos últimos anos. Segundo France e Rumpe [France & Rumpe, 2007], trabalhos atuais sobre metodologias de Engenharia Dirigida por Modelos (Model-Driven Engineering) têm se concentrado na produção de implementação e desenvolvimento de artefatos a partir de modelos detalhados.

Modelos são criados para auxiliar em propósitos específicos, por exemplo, para apresentar para uma pessoa uma descrição entendível de algum aspecto de um sistema ou para apresentar a informação de forma que possa ser mecanicamente analisada [France & Rumpe, 2007]. Os modelos são construídos em nível mais alto de abstração e podem ser transformados em código automaticamente. A probabilidade de ocorrerem erros humanos durante a modelagem é menor que durante a implementação [Ferdinand et al., 2008].

Em um ambiente de desenvolvimento de software centrado em modelo, modelos se tornam "cidadãos de primeira classe" no processo de desenvolvimento [Hettel et al., 2008]. Na fase de análise do desenvolvimento do software, modelos são um dos primeiros artefatos do sistema de software, criados com base nos artefatos de requisitos. Através da técnica de engenharia à frente (*forward engineering*), os modelos podem ser transformados em código. Porém, os modelos também podem ser gerados em uma situação diferente. Na fase de manutenção, quando o software se encontra em grau de maturidade mais avançado, os modelos podem ser recuperados. Através da técnica de engenharia reversa (*reverse engineering*), os modelos podem ser gerados a partir do código [Larman, 2008].

Uma vez que facilitar o entendimento do sistema é um dos principais benefícios da utilização de modelos, é indispensável que eles sejam consistentes entre si, ou seja, que eles estejam sincronizados. É importante observar que é desejável que essa consistência exista ao longo de todo o desenvolvimento de software para maximizar a utilização dos modelos e o desenvolvedor usufrua de seus benefícios. À medida que o software é desenvolvido, mudanças ocorrem e os modelos ficam desatualizados. A

engenharia reversa torna-se uma técnica a ser aplicada após cada iteração. Segundo Buss e Henshaw [Buss & Henshaw, 2010], a engenharia reversa é uma disciplina da engenharia de software que inverte o tradicional ciclo de vida do software.

De acordo com Antkiewicz e Czarnecki [Antkiewicz & Czarnecki, 2006], a engenharia de ida e volta (*round-trip engineering*) consiste na aplicação da engenharia à frente e da engenharia reversa. O objetivo da engenharia de ida e volta é minimizar a distância entre diferentes representações do sistema. Ela fornece a técnica que permite que os desenvolvedores circulem livremente entre o código fonte e os modelos, apoiando a abordagem de desenvolvimento iterativo [Angyal et al., 2006]. Depois de sincronizar o modelo com o código revisado, desenvolvedores podem escolher a melhor maneira de trabalhar: modificar mais o código ou modificar o modelo. Segundo Wagelaar [Wagelaar, 2008], a engenharia de ida e volta é útil em situações em que o modelo não fornece uma visão completa do software.

Manter o código e os modelos consistentes entre si não é uma tarefa simples. Não é incomum que alguns modelos sejam deixados de lado [Kollman et al., 2002]. Segundo Pressman [Pressman, 2006], a manutenção de software pode representar mais de 60% do tempo de desenvolvimento de um software. Os quesitos inteligibilidade e manutenibilidade estão intimamente ligados. Em outras palavras, a facilidade de dar manutenção em um sistema de software está ligada à facilidade de entender o sistema de software. É difícil realizar manutenção em um software sem antes entendê-lo. A compreensão da funcionalidade e do comportamento de um sistema de software é fundamental e facilitada por meio de sua abstração, representada nos modelos.

Tendo em vista os benefícios da utilização dos modelos no desenvolvimento de software, percebe-se a importância de mantê-los sempre atualizados, isto é, sempre refletindo o código fonte atual. Através da combinação das técnicas de engenharia à frente e engenharia reversa, a engenharia de ida e volta garante código e modelo consistentes. Combinada com um processo iterativo e com as ferramentas adequadas, a engenharia de ida e volta permite que o código e o modelo permaneçam sincronizados.

Conforme Hidaka e outros [Hidaka et al., 2009], o modelo de transformação bidirecional - engenharia à frente MODELO=>CÓDIGO e engenharia reversa CÓDIGO=>MODELO - desempenha um papel importante na manutenção da coerência entre os dois modelos, e tem muitas aplicações potenciais no desenvolvimento de software.

A linguagem de representação UML (*Unified Modeling Language*) [UML, 2011] tornou-se a representação gráfica mais presente em projetos de sistema de software orientado a objeto [Labiche, 2009] e a linguagem Java tornou-se uma das linguagens de programação mais utilizadas atualmente, sendo classificada em 1º lugar no TIBCO

desde 2005 [TIBCO, 2011]. Por este motivo, o nosso trabalho foi realizado utilizando essas duas tecnologias.

Kollmann e outros [Kollman et al., 2002] afirmam que, devido às diferenças nos conceitos de desenho e níveis de implementação, não existe uma forma formalmente aceita para a interpretação da extração de diagramas a partir do código fonte. Um exemplo da dificuldade de realizar engenharia reversa é que existem conceitos no modelo de origem que não têm uma correspondência no modelo de destino e vice-versa. Além disso, pode haver vários modelos fonte que estão sendo mapeados para um único e mesmo modelo [Hettel et al., 2008]. Outros trabalhos ainda afirmam que tanto a especificação de Superestrutura da UML 2 [Superstructure, 2011] quanto a geração de código devem ser melhorados com relação a determinados aspectos [Gessenharter, 2008].

Vários trabalhos no início dos anos 2000 discutiram a questão de navegar de UML para Java e de Java para UML, no contexto da teoria e das ferramentas CASE [Kollman et al., 2002] [Harrison et al., 2000] [Génova et al., 2003] [Briand et al., 2006] [Kollmann & Gogolla, 2001] [Gogolla & Kollmann, 2000] [Akehurst et al., 2007].

No entanto, conforme Anda e outros [[Anda et al., 2006] apud [Dzidek et al., 2008]], apesar da crescente popularidade, há pouca avaliação relatada sobre o uso do desenvolvimento baseado em UML. As duas tecnologias, UML e Java, evoluíram de lá pra cá e muitos trabalhos se tornaram obsoletos.

Segundo Sensalire [Sensalire et al., 2009], deve-se expor as ferramentas CASE a uma avaliação adequada a fim de determinar se elas são eficazes para ajudar os usuários em sua meta. Apesar da afirmação, percebe-se a falta de uma orientação geral sobre como a interação entre ferramenta e usuário/desenvolvedor deve ser realizada, e muitos dos desenvolvedores de ferramentas efetuam pouca ou nenhuma avaliação.

1.2 Objetivos e Metodologia

O presente trabalho teve por objetivo estudar o estado da arte da engenharia de ida e volta, abordando alguns aspectos da relação entre UML e Java que são mais aderentes à evolução destas duas linguagens e das ferramentas de suporte. A abordagem do trabalho foi estudar a engenharia de ida e volta do ponto de vista das ferramentas. É um trabalho com caráter predominante de registrar a evolução das duas linguagens e das ferramentas. Além de analisar a consistência entre modelo e código, analisamos a informação na interação da ferramenta com o usuário (desenvolvedor de software), isto é, como é feito o mapeamento de UML para Java e vice-versa. Foram elaborados estudos de caso para analisar a consistência entre modelo e código e a interação da

ferramenta com o usuário durante a engenharia de ida e volta. Foram estabelecidos critérios para sistematizar a análise.

É importante destacar que analisamos se existem informações nas interações, quando elas ocorrem e para quais aspectos as ferramentas permitem opções de mapeamento. A avaliação de dimensões tais como qualidade da interação, comunicabilidade, apreensibilidade são importantes mas não fizeram parte do escopo do trabalho para efeito de simplificação.

A importância dos modelos e a forma como eles estão sendo considerados no contexto do desenvolvimento de software nos últimos anos foram estudadas. No período de março de 2010 a março de 2011, foram pesquisadas e analisadas superficialmente, utilizando exemplos simples, diversas ferramentas CASE, entre livres e proprietárias. Pela complexidade da análise, foram avaliadas apenas 3 ferramentas utilizando os critérios estabelecidos no capítulo 3, a saber: Astah* [Astah*, 2011], Rational Software Architect [RSA, 2011] e Enterprise Architect [EA, 2011].

Existem vários trabalhos, em diferentes contextos, que abordam a engenharia à frente e a engenharia reversa. No primeiro caso, a engenharia à frente de UML para Java é analisada isoladamente; no segundo caso, a engenharia reversa de Java para UML é analisada isoladamente; e, no terceiro caso, a combinação da engenharia à frente e da engenharia reversa, que caracteriza a engenharia de ida e volta, é analisada. Cabe ressaltar que há diferença entre a engenharia à frente do primeiro caso e do terceiro caso. No primeiro caso, a engenharia à frente é extensa, o objetivo é explorar ao máximo aspectos da ida de UML para Java; enquanto que, no terceiro caso, a engenharia à frente não é extensa. De modo análogo, a engenharia reversa do segundo caso é extensa e a do terceiro caso não é extensa.

Este trabalho não investiu de forma direta nas manifestações dos dois primeiros casos, mais extensas e abrangentes, como por exemplo: traduzir vários modelos com diferentes diagramas em um sistema; considerar um produto de software e transcrevê-lo em diversos modelos e diagramas da UML. Este trabalho estudou como pequenos trechos de modelo são tratados, no contexto da engenharia de ida e volta.

Este trabalho pode ser do interesse de alguém que esteja interessado em escolher uma ferramenta com suporte à Engenharia de Ida e Volta entre UML e Java, pois discute alguns dos aspectos dessa questão. Tambem pode ser do interesse de desenhistas e programadores com relação a como navegar entre UML e Java.

Este trabalho nao incluiu aspectos de MDE (Model-Driven Engineering).

1.3 Estrutura do Trabalho

Este trabalho está organizado da seguinte forma.

O capítulo 2 aborda os temas principais deste trabalho: engenharia à frente, engenharia reversa e engenharia de ida e volta; ferramentas CASE; interação entre ferramenta e usuário; mapeamento modelo=>código; e, por fim, apresenta alguns trabalhos relacionados.

O capítulo 3 apresenta o planejamento e a execução das verificações do mapeamento UML=>Java e Java=>UML. São apresentados os critérios para a seleção das ferramentas, os critérios para avaliação das ferramentas, os estudos de caso, a execução dos estudos de caso e a análise dos resultados.

O capítulo 4 apresenta os aspectos prático-operacionais. São apresentadas algumas dificuldades encontradas durante o desenvolvimento do trabalho e as soluções adotadas.

O capítulo 5 conclui o trabalho apresentando considerações finais, ressaltando as contribuições e as limitações. São discutidos ainda alguns trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

2.1 Engenharia de Ida e Volta

No ambiente de desenvolvimento de software dirigido por modelo, varios modelos são utilizados para descrever um sistema de software em diferentes níveis de abstração e em diferentes perspectivas [Hettel et al., 2008]. Os modelos são construídos com o objetivo de delimitar o problema que se está estruturando, restringindo o foco a poucos aspectos por vez [Booch et al., 2005].

De acordo com Booch e outros [Booch et al., 2005], a modelagem é uma parte central de todas as atividades que levam à implantação de um bom software. Modelos são construídos para: i) comunicar a estrutura e o comportamento desejados do sistema; ii) visualizar e controlar a arquitetura do sistema; iii) compreender melhor o sistema que se está elaborando; iv) expor oportunidades de simplificação e reaproveitamento; e v) gerenciar riscos.

Atualmente, o desenvolvimento com abordagens baseadas em modelos é explorado, pois os modelos não são apenas mais próximos do pensamento humano, mas também ajudam na comunicação entre os desenvolvedores. Várias metodologias de desenvolvimento baseado em modelo têm sido desenvolvidas [Angyal et al., 2006]. A saber:

1. DSM (*Domain-Specific Modeling* - Modelagem de Domínio Específico) - é uma maneira de desenhar e desenvolver sistemas em alto nível de abstração, focalizando de perto o domínio do problema. É, geralmente, aplicada em conjunto com a programação generativa. Devido à geração automática de código, esta metodologia pode melhorar expressivamente a qualidade do código fonte e aumentar a produtividade [DSM, 2011];

2. MDA (*Model-Driven Architecture* - Arquitetura Dirigida por Modelo) - é uma abordagem de desenvolvimento adotada e divulgada, dentre outros, pelo consórcio OMG (*Object Management Group*) [OMG, 2011] e que aumenta o papel de modelos no processo de desenvolvimento. Ela usa modelos com diferentes níveis de abstração para desenhar, modificar e manter sistemas de software. MDA foi proposta para a engenharia à frente, em que modelos abstratos são criados pelos desenvolvedores [MDA, 2011];

3. MIC (*Model-Integrated Computing* - Computação Integrada a Modelo) - é uma técnica que transforma modelos de domínio específico em código executável. MIC provê um *framework* para produção de software usando ambientes de metamodelagem e interpretadores de modelo. MIC apoia a criação de ambientes de modelagem flexível e ajuda a rastrear as mudanças dos modelos [MIC, 2011];

4. MBD (*Model-Based Development* - Desenvolvimento Baseado em Modelo) - é um método cada vez mais aplicado na produção de artefatos de software, enfatiza o uso de modelos em todas as fases de desenvolvimento do sistema. Modelos são utilizados para descrever todos os artefatos do sistema, isto é, interfaces, interações e propriedades de todos os componentes que compõem o sistema. Estes modelos podem ser manipulados de diferentes maneiras para analisar o sistema e, em alguns casos, gerar a implementação completa do sistema [Stürmer et al., 2006].

São irrefutáveis os benefícios da utilização de modelo no desenvolvimento de software. Entretanto, existe uma dificuldade. Tais modelos fazem mais sentido quando refletem o código fonte, ou seja, quando estão sincronizados com o código fonte. E manter modelo e código consistentes não é uma tarefa simples. Não é incomum que alguns modelos sejam deixados de lado.

Segundo Hettel e outros [Hettel et al., 2008], devido à necessidade de manutenção ou mudança de requisitos, o código fonte é alterado ou estendido. Se não houver algum tipo de amarração, o modelo não reflete mais o código fonte modificado. Para evitar alterações inconsistentes, o código fonte tem sido refletido de volta para o modelo do sistema, conforme a Figura 2.1. Este processo é conhecido como engenharia de ida e volta (*RTE – Round-Trip Engineering*).

Uma das etapas do desenvolvimento de software que mais dependem dos modelos é a atividade de manutenção. Para dar manutenção em um sistema é preciso entendê-lo. Conforme Pressman [Pressman, 2006], manutenibilidade e inteligibilidade estão intimamente ligadas. Em outras palavras, a capacidade de dar manutenção está estreitamente relacionada à capacidade de compreender o sistema. É razoável dar aten-

ção especial a essa relação, uma vez que a manutenção pode representar mais de 60% do tempo do desenvolvimento de um software.

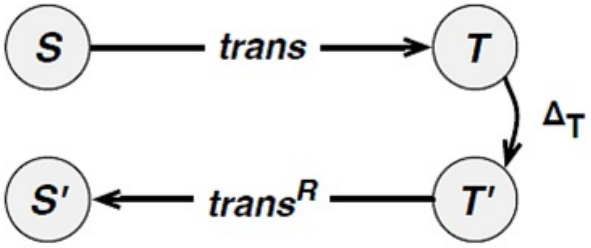

Figura 2.1. Alterações feitas no destino T devem ser refletidas de volta para a origem S. Portanto, algum tipo de transformação reversa transR é necessária [Hettel et al., 2008].

A engenharia de ida e volta é uma técnica que se propõe a contornar esse problema. Ela permite a sincronização entre código e modelo: quaisquer alterações do código fonte são sincronizadas de volta para o modelo e vice-versa, de forma que o código fonte permanece consistente com o modelo [Angyal et al., 2006]. Seguindo esse raciocínio, Angyal e outros [Angyal et al., 2006] também afirmam que a engenharia de ida e volta melhora a qualidade do software e a eficácia do desenvolvimento.

De acordo com Sendall e Küster [Sendall & Küster, 2004] e Angyal e outros [Angyal et al., 2006], a engenharia de ida e volta envolve a sincronização de modelos e a manutenção dos mesmos compatíveis, permitindo assim que o engenheiro de software possa trabalhar com diferentes representações. Desta forma, depois de sincronizar o modelo com o código revisado, desenvolvedores podem escolher a melhor maneira para trabalhar: alterar o código ou alterar o modelo.

Outra vantagem da engenharia de ida e volta é apoiar o desenvolvimento de software dividido em iterações, em vez de fases sequenciais [Angyal et al., 2006]. Uma iteração envolve todas as fases, e uma parte do desenho é implementada como um arquivo executável em uma iteração. As vantagens do Desenvolvimento Iterativo e Incremental (IID – *Iterative and Incremental Development*) são os riscos mais baixos, os erros podem ser detectados mais cedo e a facilidade de testar novas funções. A engenharia de ida e volta automatiza a sincronização do código fonte e dos modelos entre as iterações.

Segundo Sendall e Küster [Sendall & Küster, 2004], a engenharia de ida e volta está intimamente relacionada com as tradicionais disciplinas de engenharia de software: engenharia à frente (mapeamento de modelo para código), engenharia reversa (mapeamento de código para modelo) e reengenharia (compreensão do software existente e modificação do mesmo). A engenharia de ida e volta é, muitas vezes, erroneamente

definida como simplesmente suporte para a engenharia à frente e para a engenharia reversa. Na verdade, a característica da engenharia de ida e volta fundamental que a distingue da engenharia à frente e engenharia reversa é a capacidade de sincronizar os artefatos existentes, que evoluem gradualmente pela atualização de forma simultânea de cada artefato para refletir as alterações feitas para outros artefatos. Segundo Angyal e outros [Angyal et al., 2008], a engenharia de ida e volta tem foco na sincronização.

Outra característica da engenharia de ida e volta é a possibilidade de atualização automática dos artefatos em resposta às inconsistências detectadas automaticamente. Nesse sentido, é diferente da engenharia reversa e da engenharia à frente, que podem ser tanto manuais (tradicional) quanto automáticas (através da geração automática ou análise dos artefatos). A atualização automática pode ser automática ou sob demanda. Na engenharia de ida e volta automática, todos os artefatos relacionados são atualizados imediatamente após cada alteração feita em um deles. Na engenharia de ida e volta sob demanda, os autores dos artefatos podem evoluir simultaneamente os artefatos (mesmo em um sistema distribuído) e, em algum momento, verificar se existem inconsistências e escolher propagar algumas delas e conciliar os potenciais conflitos [Antkiewicz & Czarnecki, 2006].

Antkiewicz e Czarnecki [Antkiewicz & Czarnecki, 2006] apresentam uma abordagem à qual se referem como engenharia de ida e volta ágil. A abordagem oferece apoio sob demanda em vez de sincronização automática. Os artefatos a serem sincronizados podem ser editados de forma independente pelos desenvolvedores em seus espaços de trabalho local e a reconciliação (*reconciliation*) das diferenças pode ser feita de forma iterativa. Além disso, a abordagem ágil assume que o modelo pode ser completamente recuperado a partir do código utilizando análise estática.

No entanto, dependendo do contexto, existem alguns obstáculos. De acordo com Hettel e outros [Hettel et al., 2008], a dificuldade com a engenharia de ida e volta é o fato muitas vezes negligenciado de que as transformações, em geral, não são nem totais nem injetoras. Em outras palavras, existem conceitos no modelo origem que não têm um correspondente no modelo destino e vice-versa. Além disso, pode haver vários modelos sendo mapeados para um único e mesmo modelo destino.

2.1.1 Engenharia à Frente

Segundo Pressman [Pressman, 2006], a engenharia à frente é a atividade que parte de uma abstração de alto nível de implementação lógica independente para a implementação física do software. Em termos práticos, a engenharia à frente parte de um modelo desenvolvido para a geração do código fonte. Desta forma, com o código fonte sendo

resultado direto do modelo, tem-se a garantia de modelo e código fonte consistentes.

De forma geral, a engenharia à frente envolve a geração de um ou mais artefatos de software que estão mais próximos em forma e nível de detalhe para o sistema final a ser implantado, em comparação com os artefatos de entrada. A Figura 2.2 ilustra um contexto em que a engenharia à frente e a engenharia reversa são aplicadas.

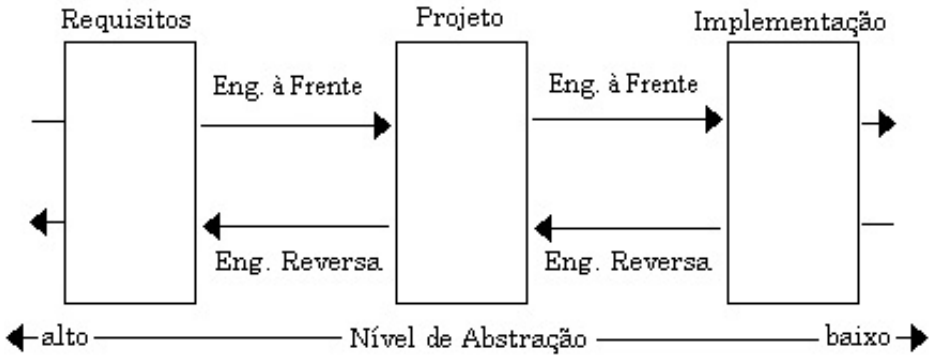

Figura 2.2. Relação entre Nível de Abstração, Engenharia à Frente e Engenharia Reversa. [Harandi & Ning, 1990]

De acordo com Sommerville [Sommerville, 2007], a engenharia à frente é o processo de desenvolvimento convencional de software, que inicia com uma especificação do sistema e envolve o projeto e a implementação de um novo sistema. Em outras palavras, é o processo tradicional de passagem de abstrações de alto nível e projetos lógicos para a implementação do sistema.

2.1.2 Engenharia Reversa

A engenharia reversa é um ramo da engenharia de software responsável por possibilitar a recuperação de informações perdidas ao longo do desenvolvimento do software. De acordo com Chikofsky e Cross [Chikofsky & Cross II, 1990], a engenharia reversa pode ser definida como o processo de análise para identificar seus componentes e seus inter-relacionamentos e criar suas representações em outra forma ou em nível mais alto de abstração.

A engenharia reversa tem sido vista tradicionalmente como um processo de duas etapas: extração e abstração de informação. A extração de informação analisa os artefatos do sistema objeto para coletar dados brutos, enquanto a abstração cria documentos e visões orientadas ao usuário [Canfora & Di Penta, 2007].

Pressman [Pressman, 2006] define a engenharia reversa como a atividade de reverter um software às suas definições mais abstratas de desenvolvimento - modelo -

com o objetivo de compreender como funciona e como não funciona para poder ser modificado de acordo com as necessidades apontadas pela reengenharia. Em termos práticos, a engenharia reversa faz o inverso da engenharia à frente, parte do código fonte para a geração do modelo. Deste modo, tem-se a garantia de código fonte e modelo consistentes.

Segundo Canfora e Di Penta [Canfora & Di Penta, 2007], "o padrão IEEE-1219 [IEEE, 1993] considera a engenharia reversa como uma importante tecnologia de apoio para lidar com sistemas que possuem o código fonte como a única representação de confiança. Os objetivos da engenharia reversa são múltiplos, por exemplo: i) lidar com a complexidade, gerando visões alternativas; ii) recuperar informações perdidas; iii) detectar efeitos secundários, sintetizando abstrações superiores; e iv) facilitar a reutilização".

De outra perspectiva, Canfora e Di Penta [Canfora & Di Penta, 2007] afirmam que a engenharia reversa pode ter dois grandes objetivos: redocumentação e recuperação de projeto. A redocumentação visa à produção e à revisão de visões alternativas de um determinado artefato, com o mesmo nível de abstração, por exemplo, bela escrita de código fonte ou visualização de Grafos de Fluxo de Controle (CFGs – *Control Flow Graphs*), enquanto a recuperação de projeto visa à recriação de abstrações de desenho a partir do código fonte, documentação existente, conhecimento de especialistas e qualquer outra fonte de informação.

A engenharia reversa de um código fonte pode gerar novamente o modelo de um sistema graficamente em nível mais alto de abstração. De forma mais genérica, a engenharia reversa é aplicada para gerar um modelo mais abstrato de artefato de software [Angyal et al., 2006]. A Figura 2.3 ilustra a engenharia à frente e a engenharia reversa.

Em casos eficientes e ótimos, a engenharia à frente e a engenharia reversa realizam apenas transformações incrementais, de forma que partes do modelo sejam modificadas, em vez de todo o modelo. Muller e outros [Müller et al., 2000] destacaram a ideia da exploração de informações extraídas pela engenharia reversa no processo de desenvolvimento à frente, ou seja, tornando algumas informações ou artefatos - por exemplo, visões arquiteturais, diagramas de projeto, ligações de rastreamento - disponíveis para os desenvolvedores através do uso de técnicas de engenharia reversa.

Segundo Canfora e Di Penta [Canfora & Di Penta, 2007], a maturidade de hoje de várias peças da tecnologia de engenharia reversa e a disponibilidade dos ambientes de desenvolvimento extensível permitem a possibilidade de continuar realizando engenharia reversa enquanto um sistema de software está sendo desenvolvido. Esta ideia tem sido aplicada em outras áreas da engenharia de software: por exemplo, Saff

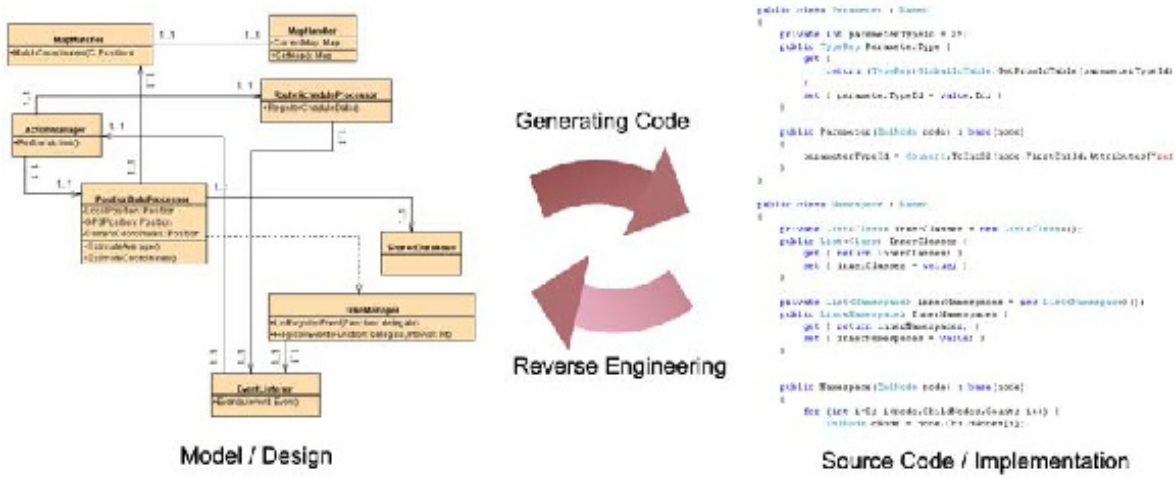

Figura 2.3. Engenharia à Frente e Engenharia Reversa [Angyal et al., 2006].

e Ernst [Saff & Ernst, 2003] propuseram a ideia de teste contínuo, isto é, de executar repetidamente suítes de teste durante o desenvolvimento ou manutenção de uma parte da funcionalidade para melhorar a eficiência dos testes.

No contexto da engenharia reversa, isso implicaria diferentes benefícios. Primeiro, através da extração e constante atualização de visões de alto nível (por exemplo, visões arquiteturais ou diagramas de desenho, como os diagramas de classe, diagramas de sequência, diagramas de estado) a partir do código fonte em desenvolvimento, seria possível fornecer ao desenvolvedor uma imagem mais clara do sistema sendo criado. Segundo, quando artefatos em diferentes níveis de abstração estão disponíveis, uma verificação de consistência permanente entre eles pode ajudar a reduzir os erros de desenvolvimento, por exemplo, checando se o código é compatível com o projeto ou está em conformidade com as pré e pós-condições. Terceiro, uma análise contínua do código em desenvolvimento e de outros artefatos pode ser utilizada para fornecer aos desenvolvedores com conhecimentos úteis, por exemplo, sugerindo o uso de um componente particular ou melhorar a qualidade do código fonte, por exemplo, através de melhoria do nível dos comentários, por meio do aumento de sua compreensão [Canfora & Di Penta, 2007].

Abordagens de engenharia reversa visam maior inteligibilidade de sistemas de software, e envolve documentação atualizada e modelos consistentes. A engenharia de ida e volta como um todo minimiza a distância entre as diferentes representações do sistema. As ferramentas CASE têm papel fundamental para possibilitar, de fato, aos desenvolvedores trabalharem com modelo UML e código fonte.

2.1.3 Ferramentas de CASE

A sigla CASE significa "*Computer-Aided Software Engineering*" - Engenharia de Software Auxiliada por Computador. Neste texto vamos tratar o termo CASE como se fosse também um adjetivo e vamos referir a "ferramentas CASE" significando "ferramenta de CASE". Um dos benefícios oferecidos pelas ferramentas CASE é orientar e disciplinar o processo de desenvolvimento de software. Ferramenta CASE é uma classificação que abrange aplicações baseadas em computadores que auxiliam atividades de engenharia de software, desde análise de requisitos e modelagem até programação e testes [Issa et al., 2007].

Algumas vantagens oferecidas pelas ferramentas CASE, direta ou indiretamente, são: i) qualidade do produto final; ii) produtividade; iii) minimização do tempo para a tomada de decisão; iv) menor quantidade de código de programação; v) melhoria e redução de custos na manutenção. Em contrapartida, suas desvantagens podem ser verificadas com a incompatibilidade de ferramentas e o tempo de treinamento para utilização da ferramenta.

De acordo com Canfora e Di Penta [Canfora & Di Penta, 2007], um dos principais desafios da análise de programa é lidar com a alta dinamicidade. Muitas linguagens de programação amplamente utilizadas permitem a alta dinamicidade, que constitui um poderoso mecanismo de desenvolvimento, mas torna a análise mais difícil. Por exemplo, linguagens como Java introduzem o conceito de reflexão e a capacidade de carregar classes em tempo de execução. A análise dinâmica é, entretanto, necessária como um complemento necessário para a análise estática.

No contexto da linguagem UML, que será abordada na seção 2.2, conforme Angyal e outros [Angyal et al., 2006], a maioria das ferramentas CASE realiza engenharia reversa e produz diagramas de classe, mas há falta de ferramentas de apoio para extrair outros diagramas.

2.2 UML

Esta seção tem como objetivo contextualizar a Linguagem de Modelagem Unificada (UML - *Unified Modeling Language*) e justificar a sua importância na modelagem de software, além de apresentar o estado em que ela se encontra atualmente e mostrar alguns exemplos.

2.2.1 Importância

Dobing e Parsons [Dobing & Parsons, 2006] afirmam que a frequência da utilização de elementos da UML varia consideravelmente: diagramas de classe, diagramas de sequência e diagramas de casos de uso são utilizados mais frequentemente, enquanto diagramas de colaboração são menos utilizados. A utilização do diagrama de classe é superior à de qualquer outro diagrama. A principal utilidade da UML no desenvolvimento de software é o entendimento. Existem outros benefícios que são consequência, como a facilitação da comunicação entre desenvolvedores e comunicação entre desenvolvedor e cliente. Segundo Dobing e Parsons [Dobing & Parsons, 2006], ao contrário do que afirma a literatura popular, os desenvolvedores parecem acreditar que os diagramas da UML podem ser entendidos pelos clientes: os clientes estão mais envolvidos com narrativas de casos de uso e diagramas de atividade, mas estão mais envolvidos com os demais componentes do que eles esperavam.

O trabalho realizado por Dobing e Parsons [Dobing & Parsons, 2006] relata que 73% dos entrevistados utilizavam o diagrama de classes em dois terços ou mais de seus projetos. Já os casos de uso eram utilizados em 44%. Apenas 3% dos entrevistados disseram que os diagramas de classes nunca foram utilizados em seus projetos e 25% disseram o mesmo sobre os diagramas de colaboração. Os entrevistados com mais experiência em UML relataram que seus projetos utilizavam mais elementos, sugerindo que o nível de utilização dos componentes da UML pode aumentar à medida que os profissionais adquirem experiência.

2.2.2 Estado Atual

A UML amadureceu significativamente desde as primeiras versões. Várias pequenas revisões (UML 1.3, 1.4 e 1.5) ajustaram deficiências e defeitos da primeira versão da UML. A versão UML 2.0 foi aprovada pela OMG em 2005 [UML, 2011].

Na época da elaboração deste trabalho, a UML estava na versão 2.4 e sua evolução tem acontecido continuamente. Para o nosso trabalho, foi considerada a versão liberada em janeiro de 2011. Por ser o padrão de fato para a representação gráfica do desenvolvimento de sistemas de software orientado a objeto, a análise crítica da Linguagem de Modelagem Unificada tem sido o foco de muitos esforços de pesquisa nos últimos anos. A UML tem apoiado a modelagem de dados, a modelagem de negócios, a modelagem de objetos e a modelagem de componentes [UML, 2011].

2.2.3 Aplicação

Um diagrama é a representação gráfica de um conjunto de elementos, geralmente representado como grafos de vértices (itens) e arcos (relacionamentos). São desenhados para permitir a visualização de um sistema sob diferentes perspectivas; nesse sentido, um diagrama constitui uma projeção de um determinado sistema [Booch et al., 2005].

Segundo Angyal e outros [Angyal et al., 2008], a UML é demasiadamente genérica para apoiar eficientemente a modelagem de domínio específico bem delimitado, isto é, a modelagem de todo o sistema em detalhes. Diagramas da UML não necessariamente elevam muito o nível de abstração, são apenas a representação visual de classes fonte. As ferramentas de modelagem da UML geram esqueletos do código fonte a partir de diagramas, e uma vez que os esqueletos necessitam ser preenchidos manualmente, eles também oferecem facilidades de sincronização, conhecida como ida e volta entre os diagramas e o código fonte.

Conforme Larman [Larman, 2008], "todas as ferramentas da UML afirmam apoiar muitos dos aspectos do desenvolvimento, mas muitas deixam a desejar. Isso acontece porque muitas das ferramentas trabalham apenas com os modelos estáticos: elas podem gerar diagramas de classe a partir do código, mas não podem gerar diagramas de interação. Ou para a engenharia à frente, elas podem gerar a definição de classe básica (por exemplo, Java) a partir de um diagrama de classe, mas não os corpos de métodos a partir de diagramas de interação".

No entanto, o código não é apenas declaração de variáveis, ele corresponde também a comportamento. Por exemplo, suponha que se deseja entender a estrutura básica de fluxo de chamada de uma aplicação ou framework existente. Se a ferramenta pode gerar um diagrama de sequência a partir do código, se torna muito mais fácil seguir a lógica de fluxo de chamada do sistema para aprender suas colaborações básicas [Larman, 2008].

2.2.4 Diagramas

Os diagramas podem ser classificados e subdivididos em diagramas estruturais e diagramas comportamentais. Os diagramas estruturais da UML estão voltados para os aspectos estáticos de um sistema, que podem ser considerados como a representação da estrutura relativamente estável do sistema. Já os diagramas comportamentais da UML estão voltados para os aspectos dinâmicos do sistema, que podem ser considerados como a representação de suas partes que sofrem alterações. A versão 2.4 do documento de Especificação da UML [Superstructure, 2011] discute 14 tipos de diagramas. A Figura 2.4 dá uma visão geral dos diagramas UML.

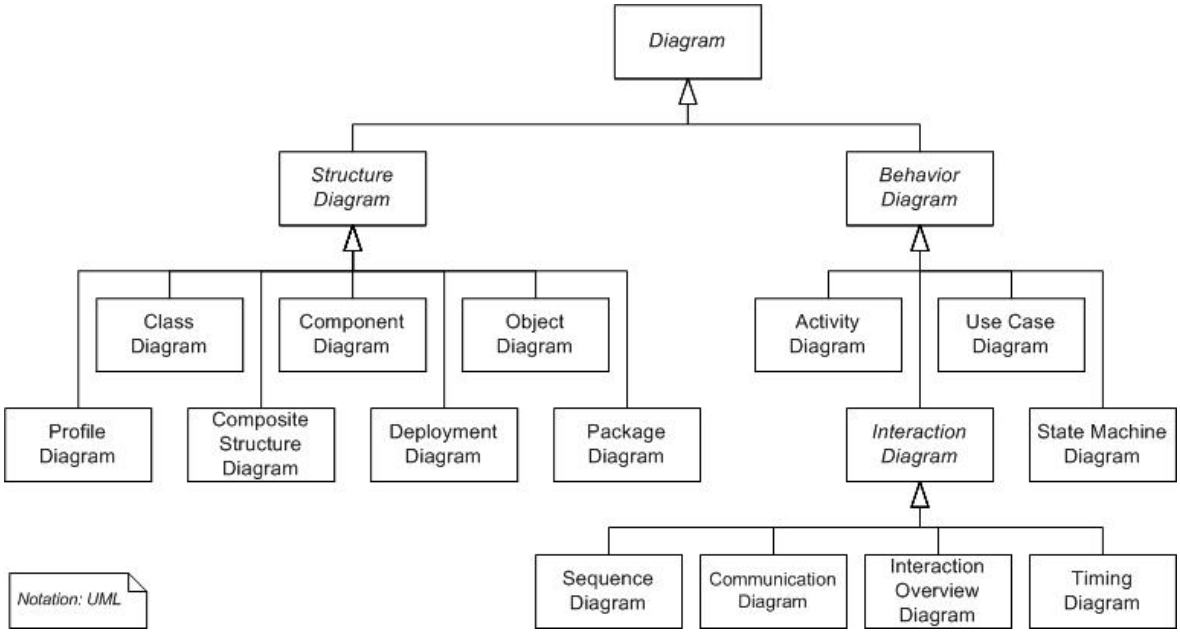

Figura 2.4. Hierarquia dos diagramas UML [UML, 2011].

2.3 Mapeamento Modelo-código

O mapeamento entre modelo e código fonte de um software (mapeamento Modelo-Código) não é uma tarefa simples. Existem restrições que tornam complexa a correspondência de elementos no modelo com elementos do código fonte. De acordo com Buarque [Buarque, 2009], existem problemas semânticos, complexidades e ambigüidades inerentes aos modelos que tornam o processo de mapeamento de modelos uma tarefa árdua e propensa a falhas.

Segundo France e outros [France et al., 2006], além de toda a complexidade, a UML carece de precisão semântica, pois muitos dos seus elementos (primitivas) têm diferentes interpretações e varia conforme o entendimento do projetista. Isso gera ambigüidades. Pastor e Molina [Pastor & Molina, 2007] afirmam que a maioria dos métodos baseados em UML tem elementos como generalização, associação e agregação tão ambíguos e dependentes da interpretação do projetista que o resultado em termos do projeto de software é imprevisível. Isso porque os relacionamentos de classes têm mais semântica do que o proposto por esses métodos. Assim, um modelo conceitual só será preciso se, e somente se, esses relacionamentos estiverem claramente definidos.

Segundo Kollmann e outros [Kollman et al., 2002], não existe formalização para a representação gráfica dos modelos gerados pela engenharia reversa de sistemas de software orientado a objeto. As diversas ferramentas geralmente adotam extensões não formalizadas da UML e, como resultado, é difícil, ou mesmo impossível, garantir que a

semântica do modelo não seja ambígua quando se trabalha com diferentes ferramentas ao mesmo tempo.

Por exemplo, como diferenciar, em nível de código, um relacionamento de agregação com um relacionamento de composição? Existem elementos visuais do diagrama de classes UML que permitem fazer a diferenciação. Entretanto, na linguagem Java, o relacionamento de agregação e o relacionamento de composição podem estar representados da mesma forma, como é ilustrado na Figura 2.5 e na Figura 2.6.

Desta forma, percebe-se que o mapeamento modelo-código correto é fundamental para o sucesso da engenharia de ida e volta. Se não existir correspondência consistente, as ferramentas que automatizam o processo de engenharia à frente e engenharia reversa não terão resultados consistentes, informações serão perdidas a cada iteração.

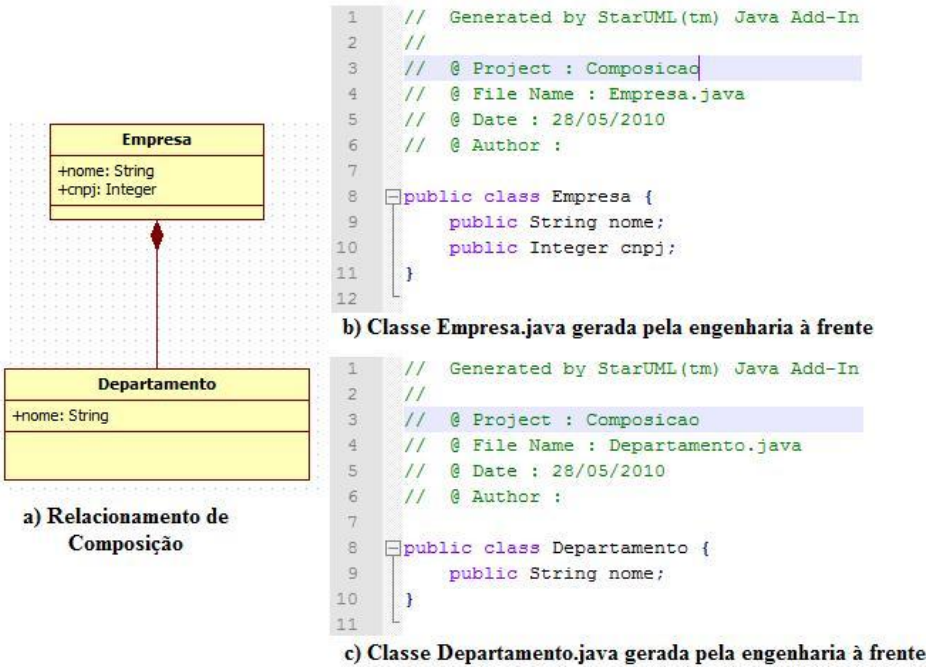

Figura 2.5. Engenharia à frente de diagrama de classe realizado pela ferramenta StarUML: Relacionamento de Composição.

2.4 Trabalhos Relacionados

Kollmann e outros [Kollman et al., 2002] realizaram um trabalho comparativo de quatro ferramentas. A primeira comparação é feita entre duas versões das ferramentas comerciais Rational Rose e Together e a segunda comparação é feita entre duas versões das ferramentas acadêmicas IDEA e FUJABA. As quatro ferramentas foram utilizadas para fazer a engenharia reversa de um sistema denominado Java Mathaino. A análise

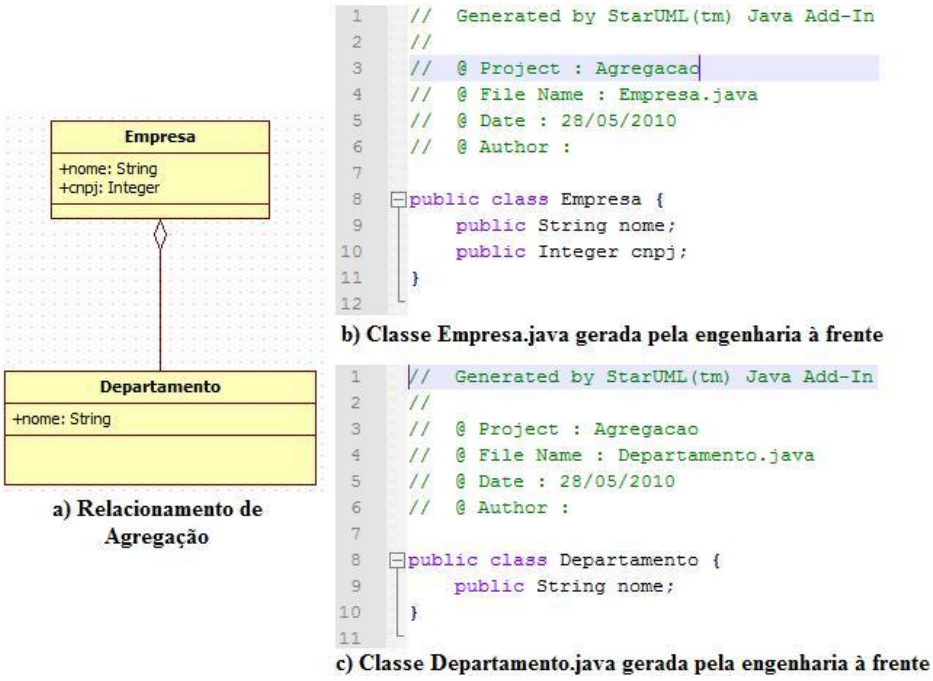

Figura 2.6. Engenharia à frente de diagrama de classe realizado pela ferramenta StarUML: Relacionamento de Agregação.

comparativa das versões das ferramentas examinou algumas propriedades do diagrama de classe da UML, a saber: i) Número de Classes; ii) Número de Associações; iii) Tipos de Associações; iv) Manipulação de Interfaces; v) Manipulação de Classes Java Collection; vi) Multiplicidades; vii) Nomes de Papéis; viii) Classes Internas; ix) e Detalhes de Compartimento de Classe.

Algumas ferramentas ofereciam perfis utilitários para medir um conjunto pré-definido de métricas, mas diferiam significativamente entre si, tornando difícil comparar os resultados de maneira uniforme. Além disso, nenhuma das ferramentas foi capaz de deduzir os elementos semanticamente equivalentes entre os vários modelos UML e comparar seus estados internos.

Os resultados das versões das ferramentas examinadas foram comparados em duas diferentes categorias: conceitos básicos e avançados. Conceitos básicos referem-se ao núcleo da UML, como classes e associações, e exige que os resultados sejam uma representação válida do modelo básico de software (isto é, nenhum elemento central tenha sido omitido ou imprecisamente representado). A segunda categoria avaliou a capacidade das ferramentas gerarem uma representação mais abstrata, em vez de uma visão de simples implementação. Isso inclui estratégias para recuperação de desenho e reconhecimento de fatos que não são imediatamente visíveis a partir do código fonte.

Todas as ferramentas, nas versões examinadas, obtiveram sucesso nos conceitos

básicos. Nos conceitos avançados, verificou-se que a capacidade de engenharia reversa das ferramentas industriais, Rational Rose e Together, não vai muito além dos recursos básicos de UML. Para as ferramentas acadêmicas, IDEA e FUJABA, representações mais abstratas e reconhecimento de características avançadas foram claramente o domínio: elas conseguiram reconhecer todas as características do conjunto de propriedades totalmente ou pelo menos até certo ponto, enquanto as ferramentas industriais não abordaram várias delas (por exemplo, multiplicidades, associações inversas e resolução de *container* não foram abordadas pelas ferramentas industriais).

Bellay e Gall [Bellay & Gall, 1997] apresentam um trabalho detalhado de comparação de ferramentas CASE de engenharia reversa. Eles avaliaram quatro ferramentas de engenharia reversa que analisam código fonte C, a saber: Refine/C, Imagix4D, Sniff+ e Rigi; investigaram as capacidades destas ferramentas, nas versões examinadas, através da aplicação delas em um sistema de software comercial incorporado como um estudo de caso; identificaram benefícios e deficiências das quatro ferramentas e avaliaram sua aplicabilidade, sua usabilidade e sua extensibilidade. O enfoque principal foi sobre os recursos da ferramenta para gerar relatórios gráficos, como árvores de chamadas, dados e gráficos de controle de fluxo.

Eles afirmam que a avaliação da tecnologia de software depende do entendimento de: 1) como a tecnologia avaliada difere de outra tecnologia; e 2) como essas diferenças atendem às necessidades dos contextos específicos de uso. O contexto de uso foi a recuperação de informações arquiteturais de sistemas de software embarcados. Foram escolhidos quatro representantes significativamente diferentes de ferramentas de engenharia reversa para cobrir uma ampla gama de ferramentas. Ao final das análises, eles afirmam que nenhuma das ferramentas, naquelas versões, poderia ser declarada como melhor na avaliação, uma vez que elas eram bastante diferentes, com diferentes vantagens e desvantagens. Todas elas forneciam boa capacidade de engenharia reversa em contextos de uso diferentes. Como resultado mais geral, a ferramenta de avaliação mostrou que o desempenho e a capacidade de uma ferramenta de engenharia reversa são dependentes do estudo de caso e seu domínio de aplicação, bem como a finalidade da análise.

Khaled [Khaled, 2009] apresenta um trabalho de comparação de ferramentas segundo alguns critérios, a saber: documentação HTML, apoio total aos diagramas da UML, engenharia de ida e volta, integração com ferramentas de modelagem de dados, exportação de diagramas e robustez. As ferramentas comparadas foram Rational Rose, ArgoUML, MagicDraw e Enterprise Architect. Foram feitas breves considerações sobre a avaliação de cada uma das versões das ferramentas. Algumas descrições eram simplesmente do tipo "a ferramenta X é capaz de gerar a documentação HTML". Ao final,

foram apresentadas algumas conclusões do tipo "a ferramenta X é a melhor escolha para a atividade Y". O objetivo do trabalho foi auxiliar um desenvolvedor na escolha da ferramenta mais adequada para determinada atividade.

Arcelli e outros [Arcelli et al., 2005] realizaram um trabalho que consiste na comparação de ferramentas de engenharia reversa baseada em decomposição de mecanismos de desenho. Segundo eles, a utilidade dos mecanismos de desenho em engenharia à frente é bem conhecida e diversas ferramentas proveem suporte para sua aplicação no desenvolvimento de sistemas. No entanto, o papel dos mecanismos de desenho em engenharia reversa ainda é argumentado, principalmente devido à sua definição informal, que leva a várias implementações possíveis de cada mecanismo.

Um dos aspectos mais discutidos relacionados aos mecanismos de desenho é a necessidade de sua formalização de acordo com os inconvenientes que podem apresentar. A formalização leva à identificação dos chamados sub-mecanismos, que são elementos recorrentes fundamentais pelos quais os mecanismos de desenho são compostos. No trabalho, eles analisaram o papel dos sub-mecanismos utilizados em duas ferramentas de engenharia reversa: FUJABA e SPQR. A atenção esteve voltada para como os mecanismos eram sub-explorados para definir e detectar mecanismos de desenho.

Neste contexto, há uma forte necessidade de formalizar mecanismos de desenho. Inevitavelmente, a formalização leva à identificação de elementos recorrentes regulares. Considerando os mecanismos de composições de elementos mais simples, pode-se reduzir significativamente a criação de variantes em engenharia à frente, ao mesmo tempo em que aumenta a possibilidade de identificar padrões de aplicação em engenharia reversa.

Angyal e outros [Angyal et al., 2006] realizaram um trabalho que analisa a importância do desenvolvimento baseado em modelo e dá uma visão geral do estado da arte de métodos e ferramentas de engenharia reversa. Além disso, considera a engenharia de ida e volta como uma abordagem que pode ser utilizada para alcançar melhor qualidade de software, uma vez que as mudanças que afetam o desenho não são feitas no código, mas no modelo.

O trabalho apresenta a modelagem visual de sistemas de software como uma representação do comportamento desejável do sistema, em alto nível de abstração, que pode ser uma forma eficaz de tornar o processo de desenvolvimento de software mais eficiente. O problema de inconsistência do modelo com o código, ao longo do processo de desenvolvimento, também é analisado. Para ajudar os desenvolvedores a alcançar novamente a sincronia entre o modelo e código, as ferramentas de engenharia reversa foram criadas.

São abordadas algumas metodologias de desenvolvimento baseado em modelo,

a saber: i) DSM (Domain-Specific Modeling - Modelagem de Domínio Específico); ii) MDA (Model-Driven Architecture - Arquitetura Orientada a Modelo); iii) (Model-Integrated Computing - Computação Integrada a Modelo); e iv) MBD (Model-Based Development - Desenvolvimento Baseado em Modelo).

Ferramentas e abordagens mais relevantes são apresentadas, subdivididas de acordo com seu foco principal: i) Ferramentas CASE (Rational XDE, Together, Eclipse GMT, Fujaba e MetaEdit++); Parsers e Ferramentas de Código Fonte (Columbus, CPP2XMI, Rigi, SHriMP, GUPRO e JavaCC); Reconhecimento de Padrões de Projeto (CrocoPat, Columbus, PideJ, SPOOL e PINOT).

O trabalho permite verificar que ainda há deficiências quanto à realização de engenharia de ida e volta de forma plena. A maioria das ferramentas CASE não é capaz de trabalhar com outros diagramas de forma satisfatória, a não ser com o diagrama de classes. Fazendo correlação com o trabalho de Kollmann e outros [Kollman et al., 2002], que diz que não existe ainda um esquema padrão para a representação de modelos resultantes da engenharia reversa de um sistema, torna-se relevante a avaliação de versões recentes das ferramentas observando, principalmente, novos elementos que foram inseridos a partir da UML 2.0

Ali [Ali, 2005] realizou um trabalho que aborda a importância da engenharia reversa de software como uma disciplina da engenharia de software. Segundo ele, a atração da atenção dos estudantes não tem sido voltada para essa questão. Em vez de dar manutenção nos sistemas existentes, o desenvolvimento de novos softwares sempre tem tido prioridade. Porém, com a chegada da internet e da tecnologia cliente-servidor, muitas organizações desejam adaptar seus sistemas existentes. Desta forma, a tendência tem se voltado um pouco para a evolução de software e manutenção. E, agora, mais ainda, precisa-se de engenheiros de software que possam trabalhar eficazmente com os sistemas legados.

Segundo Ali [Ali, 2005], utilizando as três abordagens para a análise de sistema empregando engenharia reversa (Análise de Caixa Branca - *White Box*; Análise de Caixa Preta - *Black Box*; e Análise de Caixa Cinza - *Gray Box*), estes são alguns métodos geralmente utilizados pelos engenheiros de software: i) rastreamento de entrada; ii) estudar as diferentes versões do sistema; iii) cobertura de código) iv) acesso kernel; v) extração de informações através de dados que tenham ficado em *buffers* compartilhados; e vi) estudo de APIs.

O trabalho busca convencer os acadêmicos que estão envolvidos no projeto de currículo de engenharia de software das universidades, universitários e também aqueles que também podem fazer este curso na universidade. Existem alguns resultados: injetar confiança em seus estudantes de engenharia para trabalhar em vários problemas reais do

mundo é uma abordagem desenvolvida por Barsotti [[Barsotti, 2003] apud [Ali, 2005]].

O trabalho de Ali [Ali, 2005] aborda a importância da engenharia reversa de forma geral. Foi um trabalho breve, com levantamento de questões teóricas e de estatísticas de pesquisas realizadas sobre a importância da engenharia reversa no meio acadêmico e na indústria. Percebe-se que a engenharia reversa é uma atividade que exerce significativa influência no meio onde é empregada e está em ascensão.

Outro trabalho relacionado foi realizado na Universidade de Missouri-Rolla [Robert. B. Stone, 2000], introduzindo técnicas de engenharia reversa e estimulando estudantes a estudarem produtos do mundo real. E os resultados foram encorajadores: 77% dos estudantes consideraram que a introdução da metodologia de engenharia reversa reforçou conceitos ensinados durante as aulas. E, ainda, 82% queriam que ela fosse incorporada em cursos futuros, especialmente em cursos de projeto.

Através da engenharia reversa, estudantes de engenharia de software podem se beneficiar:

- A melhor e mais profunda compreensão é a primeira e principal vantagem do ensino de conceitos de engenharia reversa. Conhecimento valioso é adquirido com a funcionalidade de hardware e software;

- A indústria de software é, talvez, a indústria de mais rápida mutação. Avança por si própria e amplia suas práticas para outros campos da ciência e engenharia;

- Uma vez que a engenharia reversa de um sistema é uma tarefa difícil, o seu aprendizado auxilia no aprendizado de outras habildades;

- Muitas universidades ao redor do mundo oferecem engenharia reversa em um nível avançado e elas estão conduzindo pesquisa.

O trabalho realizado por Canfora e Di Penta [Canfora & Di Penta, 2007] apresenta uma pesquisa realizada na área de engenharia reversa de software, discute histórias de sucesso e principais realizações, e fornece um roteiro para possíveis desenvolvimentos futuros à luz das tendências emergentes em engenharia de software.

Eles apresentaram as principais realizações da engenharia reversa durante a última década, organizadas em três principais áreas: análise de programa, recuperação de projeto e visualização. Em seguida, utilizando as mesmas três áreas, identificaram e discutiram questões que pudessem vir a ser desafios da engenharia reversa nos anos seguintes. Além disso, identificaram desafios da engenharia reversa que derivam de paradigmas emergentes de computação, como a computação orientada a serviço e

a computação autônoma. Por fim, trataram a questão da facilitação da adoção da engenharia reversa.

Segundo Canfora e Di Penta [Canfora & Di Penta, 2007], apesar da maturidade da engenharia reversa, e do fato de inúmeros trabalhos de engenharia reversa parecerem temporariamente resolver problemas cruciais e atenderem importantes necessidades industriais, sua adoção na indústria ainda é limitada. Eles apresentam algumas direções para avanço:

- Educação de engenharia reversa: o ensino de engenharia reversa como uma parte integral do processo de projeto de software – e não apenas como técnicas para lidar com mudanças – aumentará a consciência do papel da engenharia reversa, contribuindo assim com redução das barreiras; companhias devem investir em engenharia reversa ou pelo menos adotar práticas de engenharia reversa;

- Evidências empíricas:: especialmente durante os últimos anos, a maioria das pesquisas em engenharia reversa tem sido validada empiricamente com estudos visando medir o desempenho de uma técnica ou comparando-a com outras já existentes;

- Aumento da maturidade e interoperabilidade das ferramentas: isso deve ser questionado, pode ou não ser o papel dos pesquisadores. Entretanto, a avaliabilidade de ferramentas de engenharia reversa maduras e sua interoperabilidade irão favorecer a sua utilização e, consequentemente, a sua adoção.

Tonella e outros [Tonella et al., 2007] realizaram estudos empíricos em engenharia reversa. Segundo eles, a engenharia reversa, que surgiu com o objetivo de modernizar os sistemas legados, normalmente escritos em linguagens de programação antigas, tem estendido sua aplicabilidade a praticamente todo o tipo de sistema de software. Além disso, os métodos originalmente concebidos para recuperar uma visão esquemática de alto nível do sistema destino têm sido alargados a vários outros problemas enfrentados pelos programadores quando precisam entender e modificar o software existente. A posição dos autores é que a próxima fase de desenvolvimento para esta disciplina seja necessariamente baseada na avaliação de métodos empíricos. Na verdade, essa avaliação é necessária para adquirir conhecimento sobre os efeitos reais da aplicação de uma determinada abordagem, bem como para convencer os usuários finais dos custos e benefícios. A contribuição do trabalho deles para o estado da arte é um roteiro para a pesquisa futura no campo, que inclui: clareamento do escopo de investigação, definição de uma taxonomia de referência e adoção de um framework comum para a execução de experimentos.

Hettel e outros [Hettel et al., 2008] realizaram um trabalho na área de sincronização de modelo, apresentando uma definição formal de engenharia de ida e volta e tratando da questão da semântica das alterações no contexto de transformações parciais ou não injetoras. De acordo com eles, no ambiente de desenvolvimento de software centrado em modelo, vários modelos diferentes são utilizados para descreverem um sistema de software em diferentes camadas de abstração e em diferentes perspectivas. Seguindo a visão MDA (*Model Driven Architecture*), a transformação de modelo é utilizada para apoiar o refinamento gradual de modelos abstratos em modelos mais concretos.

A definição formal da sincronização de modelo em si e a transformação de ida e volta são apresentadas utilizando modelos matemáticos. Hettel e outros [Hettel et al., 2008] consideram que a contribuição original de seu trabalho é a definição formal de engenharia de ida e volta, no contexto de transformação do modelo. A definição proposta vai além das abordagens existentes, uma vez que envolve transformações parciais e não injetoras, que foram mostradas serem mais realistas que as transformações injetoras requeridas pelas abordagens existentes para RTE ou sincronização de modelo.

Hidaka e outros [Hidaka et al., 2009] também afirmam que a transformação de modelo bidirecional desempenha um papel importante na manutenção da consistência entre dois modelos, e tem muitas aplicações potenciais no desenvolvimento de software, incluindo sincronização de modelo, engenharia de ida e volta, evolução de software, desenvolvimento de software de múltiplas visões e engenharia reversa. Entretanto, a semântica bidirecional não clara, o método de bidirecionalização de domínio específico e a falta de um *framework* semântico de desenvolvimento são problemas conhecidos que impedem a transformação de ser utilizada.

Capítulo 3

Planejamento e Execução dos Estudos de Caso

Este capítulo apresenta a avaliação de três ferramentas CASE - Astah*, Rational Software Architect e Enterprise Architect - realizada através da análise dos mapeamentos de alguns modelos da UML para a linguagem Java e vice-versa. A seção 3.1 apresenta os critérios utilizados para selecionar as ferramentas que foram analisadas. A seção 3.2 apresenta os critérios utilizados para avaliar as ferramentas. A seção 3.3 faz uma discussão sobre Perfis da UML. A seção 3.4 descreve e discute os estudos de caso elaborados a partir da versão 2.4 do documento de especificação de superestrutura da UML [Superstructure, 2011], publicado em janeiro de 2011. Ao fim desta seção, obtém-se o conjunto de ferramentas a serem testadas neste trabalho de pesquisa e os estudos de caso aplicáveis. A seção 3.5 apresenta a execução dos estudos de caso em cada uma das ferramentas e algumas considerações em relação aos resultados apresentados.

3.1 Critérios para Seleção das Ferramentas CASE

A seleção das ferramentas CASE baseou-se nas seguintes premissas: i) relevância traduzida em participação no mercado e interesse (nos fóruns eletrônicos, nos trabalhos científicos, entre outros) das pessoas; e ii) seleção de 3 ferramentas para avaliação.

Com foco nestas premissas, estabeleceram-se os seguintes critérios, apresentados na Tabela 3.1, para a seleção de ferramentas.

Aplicando-se o critério C1, relacionam-se as ferramentas CASE utilizadas no mercado, atualmente. Aplicou-se as expressões “ferramentas de engenharia de ida e volta” e “round-trip engineering tools”, “ferramentas de engenharia reversa” e “reverse engineering tools” e “ferramentas UML” e “UML tools” nos principais sistemas de busca

Tabela 3.1. Critérios para seleção das ferramentas CASE

Critério	Descrição
C1	A busca das ferramentas deve ser feita em máquinas de busca poulares (por exemplo no Google e no Bing), aplicando-se as sentenças "ferramentas de engenharia de ida e volta" e "round-trip engineering tools", "ferramentas de engenharia reversa" e "reverse engineering tools"; "ferramentas UML" e "UML tools", tomando em consideração cerca de 50 elos de navegação (links).
C2	A ferramenta deve oferecer suporte a linguagem UML e a linguagem Java.
C3	A ferramenta deve possuir disponibilidade para ser copiada a partir do sítio da organização responsável por ela.
C4	A ferramenta deve oferecer licença gratuita ou para teste temporário, considerando-se que não foi colocada como parte do trabalho a obtenção de recursos para aquisição de licenças.
C5	A ferramenta deve não ter sido descontinuada, e oferecer suporte à UML 2.
C6	A ferramenta deve possuir fórum de discussões e dúvidas ativo, colhendo-se informações relacionadas à sua existência, número de mensagens trocadas, número de membros e visualizações, para se avaliar seu alcance no mercado. A ferramenta deve possuir lista de discussão recente e suporte.

da atualidade, Google e Bing. Para cada sistema de busca, e para cada expressão, visitaram-se 50 *links*. Considerando 6 expressões, então visitaram-se 300 *links* que, multiplicados por 2 sistemas de busca, totalizam 600 *links* visitados.

O critério C2 consiste em restringir a pesquisa para ferramentas CASE que suportam a linguagem de modelagem UML e a linguagem de programação Java. Com o escopo menor, é possível realizar um trabalho mais detalhado. A UML é a linguagem mais popular para a representação gráfica de sistemas de software orientados a objeto e a linguagem Java representa uma das linguagens mais utilizadas atualmente, sendo classificada em 1º lugar no TIBCO desde 2005 [TIBCO, 2011]. Por esse motivo, este trabalho foi realizado utilizando essas duas tecnologias.

O critério C3 consiste em restringir a avaliação para apenas ferramentas CASE disponíveis para cópia. Para não dependermos de envio de CDs ou outros esquemas que poderiam atrasar os trabalhos, optamos por avaliar ferramentas que estivessem disponíveis na Internet para cópia.

O critério C4 consiste em restringir a avaliação para apenas ferramentas CASE livres, gratuitas ou pagas que ofereçam licença de teste. A licença deve ser extensa o suficiente para poder ser feita uma avaliação.

O critério C5 consiste em descartar as ferramentas CASE descontinuadas. Há ferramentas que pararam de ser desenvolvidas e possuem apenas versões antigas. Neste caso a análise de tais ferramentas só é justificada em função de interesses mais específicos do que os deste trabalho, não acompanhando a evolução da linguagem. Desta forma, o trabalho deu preferência às ferramentas CASE que estão em constante evolução e possuem suporte à UML 2.

O critério C6 consiste em restringir a avaliação para ferramentas CASE que possuam suporte, fóruns, listas de discussão. Isto ajudou o autor desta pesquisa a sanar dúvidas durante o teste das ferramentas e na avaliação da popularidade da ferramenta. Acredita-se que quanto mais pessoas se apresentam no fórum, maior sua abrangência no mercado. Esses ambientes e seus números evidenciam maior ou menor aceitação e utilização da ferramenta pelos usuários ou visitantes.

Dentre as ferramentas encontradas, após a aplicação dos critérios de seleção, selecionamos: Astah*, Rational Software Architect e Enterprise Architect. A aplicação dos critérios de seleção das ferramentas CASE é relatada com mais detalhes na seção 4.1.

A seção 3.2 discute os critérios que foram utilizados para avaliar as ferramentas selecionadas. O objetivo foi verificar as seguintes características: i) as ferramentas são consistentes na transcrição do modelo para o código e vice-versa; ii) as ferramentas oferecem recursos que facilitam o usuário entender e/ou influenciar o processo de mapeamento de UML para Java e vice-versa de forma clara. Por exemplo: os usuários sabem, em alto nível, o caminho traçado pela ferramenta para realizar as tarefas? Quando o mapeamento não é um para um, a ferramenta comunica e interage permitindo que o usuário selecione opções ou forneça uma opção ou a ferramenta tem sempre uma escolha padrão? A ferramenta é flexível, isto é, permite ao usuário inserir um tipo de atributo que não esteja pré-definido pela ferramenta? A ferramenta exibe algum tipo de mensagem, antes ou depois, quando a operação contém erro? Um objetivo próximo a estes descritos é avaliação da comunicabilidade das ferramentas, mas em função da complexidade de tal análise esta dimensão ficou fora do escopo do trabalho. Os critérios para avaliação de ferramentas CASE são analisados na próxima seção.

3.2 Critérios para Avaliação de Ferramentas CASE

Segundo Sensalire [Sensalire et al., 2009], deve-se expor as ferramentas a uma avaliação adequada a fim de determinar se elas são eficazes para ajudar os usuários em sua meta. Apesar da afirmação, percebe-se a falta de uma orientação geral sobre como a

interação entre ferramenta e usuário (desenvolvedor) deve ser realizada, e muitos dos desenvolvedores de ferramentas efetuam pouca ou nenhuma avaliação.

A falta desta orientação geral foi percebida após a realização de consultas nos principais sistemas de busca de trabalhos científicos como Scopus, Springer, IEEExplore, ACM Digital Library, Portal de Periódicos CAPES. Não foram encontrados trabalhos alinhados com nossas metas. Sabe-se de trabalhos como os de Clarisse Sieckenius [de Souza, 2005] [de Souza et al., 2010], mas o foco é a questão da IHC - Interação Humano-Computador -, enquanto que o nosso foco são as questões técnicas de opções de mapeamento UML para Java e Java para UML. Além de analisar a transcrição de UML para Java e vice-versa, nossa avaliação consistiu em verificar se existem ou não opções de mapeamento e quais são as opções, caso existam. A avaliação de dimensões tais como qualidade da interação, comunicabilidade e apreensibilidade são importantes, mas não fizeram parte do escopo do trabalho para efeito de simplificação.

Decidimos, então, desenvolver critérios próprios para avaliação das ferramentas CASE, conforme descritos na Tabela 3.2, cujo objetivo foi avaliar a transcrição do modelo para o código e a interação entre usuário e ferramenta CASE durante o processo de mapeamento de ida e volta entre as linguagens UML e Java. Neste trabalho, os critérios definidos atendem a dois momentos distintos do processo de mapeamento entre UML e Java, ou engenharia de ida e volta.

O primeiro momento trata da engenharia de IDA, responsável por mapear modelos da UML para a linguagem Java. Neste primeiro momento, os modelos dos estudos de caso em UML foram mapeados para Java, utilizando as ferramentas CASE selecionadas para análise. O código gerado e a forma como ele foi gerado foram avaliados de acordo com os critérios da engenharia de ida, elaborados neste trabalho.

O segundo momento trata da engenharia de VOLTA, responsável por mapear código em Java para modelos da UML, cujo processo também é conhecido como engenharia reversa. Pretendíamos elaborar outros estudos de caso com estruturas Java e analisar separadamente o mapeamento de Java para UML. No entanto, em razão da complexidade da avaliação, não foi possível e este mapeamento foi analisado apenas utilizando o código gerado pela engenharia de ida. Desta forma, a engenharia de volta não foi analisada de forma separada, apesar de algumas considerações sobre ela terem sido feitas juntamente com as considerações sobre a engenharia de ida. Deste modo, utilizam-se os critérios para a mapeamento de UML para Java, sendo que para cada critério é composto pelos seguintes itens: i) a identificação (ID); ii) a descrição; iii) o método de conferência; e iv) o sistema de avaliação.

O ID representa o código e o nome do critério.

A Descrição apresenta detalhadamente o objetivo do critério.

O Método de Conferência descreve como é a verificação do desempenho da ferramenta em cada estudo de caso, de modo que seja possível analisar sistematicamente a confiabilidade do mapemaneto.

O Sistema de Avaliação descreve como as ferramentas serão classificadas, conforme seu desempenho. Uma pontuação é atribuída de acordo com uma escala likert [Likert, 1932]. Por exemplo: (0) não atende; (1) atende parcialmente; e (2) atende totalmente. As considerações e a pontuação foram feitas pelo autor do trabalho. O objetivo desta pontuação é caracterizar e descrever cada ferramenta segundo cada um dos critérios.

De acordo com Sensalire e outros [Sensalire et al., 2009], apesar de haver a falta de um guia geral para auxiliar na avaliação das ferramentas, cada pessoa que realiza uma avaliação, no entanto, tem experiências que, se compartilhadas, podem orientar avaliadores futuros. A Figura 3.1 mostra o ciclo de melhoria de uma ferramenta CASE [Sensalire et al., 2009].

Figura 3.1. Ciclo de melhoria da ferramenta CASE [Traduzido: [Sensalire et al., 2009]].

A Tabela 3.2 apresenta a relação dos critérios de avaliação das ferramentas CASE.

O objetivo do critério C1 é verificar se o significado do Modelo da UML é preservado no Modelo da linguagem Java. Há algum tipo de inconsistência no código gerado? Todos os detalhes do modelo são verificados no código ou há informação perdida? O modelo produzido pela engenharia de volta corresponde ao modelo inicial? A utilização do critério C1 pode ajudar a identificar problemas do tipo "inconsistências" que

Tabela 3.2. Relação dos critérios de avaliação das ferramentas CASE.

CRITÉRIOS
ID: C1 - Confiabilidade do Código Gerado. **Descrição:** Este critério propicia verificar se o código gerado corresponde ao modelo elaborado pelo usuário. **Método de Conferência:** Inspeção de código gerado em Java a partir da observação e entendimento do modelo desenhado em UML. **Sistema de Avaliação:** Aplicação de escala likert, adotando-se: 0 – Código inconsistente, 1 – Código parcialmente consistente, 2 – Código consistente.
ID: C2 - Capacidade de Resolução de Ambiguidades de Modelo. **Descrição:** Este critério propicia verificar se a ferramenta é capaz de resolver automaticamente ou semi-automaticamente ambiguidades durante o mapeamento de UML para Java. **Método de Conferência:** Inspeção se há interfaces de diálogo com o usuário, concedendo-lhe direito de escolha durante o processo de mapeamento de UML para Java, ou se há área de configuração do mapeamento desejado. **Sistema de Avaliação:** Aplicação de escala likert, adotando-se: 0 – Não resolve, 1 – Resolve parcialmente, 2 – Resolve.
ID: C3 - Consistência Interna da Ferramenta. **Descrição:** Este critério propicia verificar se a ferramenta é capaz de, a partir de um modelo UML desenhado pelo usuário, gerar código Java e fazer a engenharia reversa, construindo um modelo UML, compatível com o modelo inicial elaborado pelo modelador. **Método de Conferência:** Inspeção de modelo gerado observando detalhes (tipos de atributos, tipos de relacionamentos entre classes, multiplicidade, extremidades de associação, sentença de propriedades, entre outros) existentes no modelo inicial. **Sistema de Avaliação:** Aplicação de escala likert, adotando-se: 0 – Inconsistente, 1 –Parcialmente Consistente, 2 – Consistente.
ID: C4 - Flexibilidade e Robustez da Ferramenta. **Descrição:** Este critério propicia verificar se a ferramenta permite ser detalhado em nível de modelo tanto quanto pode ser detalhado em nível de código utilizando a importação de bibliotecas. **Método de Conferência:** Inspeção se é possível importar/habilitar bibliotecas que permitam elaborar um modelo robusto e detalhado, modelar todo tipo de relacionamento, de modo a diminuir a implementação manual de código. Inspeção se os tipos utilizados no modelo estão presentes nos pré-definidos, nas bibliotecas ou no modelo. **Sistema de Avaliação:** Aplicação de escala likert, adotando-se: 0 – Não Robusta, 1 –Parcialmente Robusta, 2 – Robusta.

comprometem a qualidade do software. Para facilitar a verificação da correspondência entre modelo e código, procurou-se definir um "código esperado" para os modelos ava-

liados de cada estudo de caso apresentados na seção 3.4, mas durante as pesquisas não foram encontrados relatos de boas práticas para geração de código Java a partir de um determinado elemento da UML. Desta forma, serão verificadas se os elementos Java gerados a partir dos modelos são consistentes com a linguagem Java e se representam, de fato, as estruturas do modelo.

O objetivo do critério C2 é verificar como a ferramenta trata a engenharia à frente e a engenharia reversa quando efetivamente há defeitos. Sabe-se que a relação entre modelo e código não é sempre um para um [Hettel et al., 2008]. Isto é, um modelo nem sempre terá apenas uma forma única de representação em nível de código e vice-versa. Por exemplo, considere um modelo em que há uma associação do tipo 1 para N entre as classes A e B. Há várias formas de transcrever esta parte do modelo. Podem ser usadas coleções ou podemos usar arranjos, por exemplo. A ferramenta disponibiliza algum tipo de interface que permite ao usuário escolher alguma das opções ou existe uma solução padrão? Existe alguma área de configuração em que o usuário possa escolher a opção de geração que seja mais adequada ao seu interesse? Enfim, de alguma forma, o usuário pode intervir no resultado apresentado pela ferramenta?

O objetivo do critério C3 é verificar se a ferramenta cria, perde ou transforma alguma informação no processo de engenharia à frente ou na engenharia reversa. Observe que o critério C1 verifica consistência somente na engenharia à frente. Dado um modelo elaborado, será gerado pela ferramenta o respectivo código. Em seguida, sem realizar nenhum tipo de alteração no que foi gerado pela ferramenta, o código gerado será a entrada para a ferramenta gerar de volta o modelo. Teoricamente, este modelo gerado deveria ser idêntico ao modelo inicial.

O objetivo do critério C4 é verificar se a ferramenta trata de forma adequada as especificações de pacotes, bibliotecas, classes e interfaces. Por exemplo, na criação de um diagrama de classe, os tipos das classes, atributos e métodos são pré-definidos nas ferramentas. Considerando a realidade de muitas linguagens de programação possuírem diversas bibliotecas com vários tipos de atributos e métodos, para uma ferramenta que dá suporte a mais de uma linguagem, não é razoável prever todos os tipos possíveis. Desta forma, ou a ferramenta se limita a ter um mínimo de opções ou permite a importação de biblioteca relacionada a determinada linguagem e, consequentemente, a possibilidade de declarar no modelo tipos dessa biblioteca.

Por exemplo, na biblioteca “java.lang.Math” existe um método chamado “sqrt (double a)”, com características pré-definidas. Se o desenvolvedor desejasse declarar um método deste tipo no modelo, não seria interessante implementar tudo do início. Ao contrário, é desejável que a ferramenta permita algum tipo de importação e utilização desta biblioteca no modelo.

3.3 Perfis da UML

A UML fornece um conjunto de mecanismos de extensão - estereótipos, valores etiquetados (*tagged values*) e restrições - para especialização dos seus elementos, permitindo a personalização de extensões da UML para aplicações de domínio específico. Essas personalizações são conjuntos de extensões da UML agrupadas em perfis da UML (*UML profiles*) [Möller et al., 2008]. Um perfil da UML define uma maneira específica de utilização da UML. Por exemplo, "*Java profile*" define uma maneira de modelar código fonte Java em UML [OMG, 2011].

Conforme Mueller e outros [Mueller et al., 2006], os perfis permitem personalizar a UML de forma que qualquer sistema poderia, em teoria, ser modelado em qualquer nível de detalhe. Um perfil é feito por um conjunto de estereótipos, valores etiquetados e restrições para definir como a sintaxe e a semântica do modelo são estendidos para uma terminologia de domínio específico.

Como os perfis podem estender um meta-modelo, eles são derivados da definição MOF (*Meta Object Facility*) e fornecem extensibilidade apenas o suficiente para criar uma perspectiva, evitando a complexidade da definição de um meta-modelo novo [Mueller et al., 2006].

O perfil deve ser capaz de especializar a semântica dos elementos do meta-modelo da UML. Por exemplo, em um modelo com o perfil "*Java model*", a generalização de classes deve ser capaz de restringir a herança simples sem ter que atribuir explicitamente um estereótipo «classe Java» para toda instância de classe. Existem várias razões pelas quais se pode querer personalizar um meta-modelo. Uma delas é que as informações adicionadas podem ser utilizadas na transformação de um modelo para outro modelo ou código (como a definição de regras de mapeamento entre o modelo e o código Java) [Superstructure, 2011].

Deveria existir um UML Profile para modelos de desenho ou para modelos de implementação utilizando a linguagem Java ou utilizando a plataforma J2EE - assim como existe para CORBA, SysML, SoaML, entre outros. Não foi encontrado no Catálogo de Especificações de Perfis da UML (*Catalog of UML Profiles Specifications*) [OMG, 2011] um "UML Profile for Java". O documento "*Metamodel and UML Profile for Java and EJB Specifications*", de fevereiro de 2004, disponível no sítio da OMG [OMG, 2011], faz referência a um *Java Profile*, onde são mapeados conceitos de meta-modelo Java para elementos do perfil. No entanto, não foram encontradas atualizações desde aquela época, o que permite dizer que as mudanças a partir da UML 2 não foram consideradas.

Consequentemente, sem a referência de um "*UML Profile for Java*", os desen-

volvedores das ferramentas são levados a construir seu próprio perfil. Com o objetivo de verificar o perfil que cada uma das três ferramentas selecionadas utiliza, enviamos uma mensagem para o suporte de cada ferramenta perguntando qual "*UML Profile for Java*" era utilizado pela ferramenta.

A Astah* deu a seguinte resposta: "atualmente, não oferecemos qualquer 'UML Profile' que possa ser baixado e aplicado na Astah*". No entanto, a edição professional tem a capacidade de personalizar o UML profile, permitindo ao usuário: i) especificar '*TaggedValues*' e importá-los; ii) especificar conjunto de ícones personalizados para os Estereótipos; e iii) definir valores específicos Java (atributos, métodos, «enum», final, entre outros).".

A RSA respondeu dizendo informações sobre os produtos da IBM estão disponíveis no sítio dela. No sítio foi encontrado um material que mostra como aplicar um perfil. Na RSA, o perfil de UML para Java contém vários estereótipos que podem ser aplicados para validar elementos no modelo de origem e estereótipos que controlam como a transformação gera o código Java.

Não recebemos resposta do pessoal de suporte da Enterprise Architect. No entanto, analisando a ferramenta verificou-se ser possível especificar os estereótipos (o nome do estereótipo, onde ele é aplicado e comentários sobre ele), os valores etiquetados e as restrições. O estereótipo "instantiate", por exemplo - utilizado na seção 3.4.1 -, pode ser definido no perfil.

3.4 Estudos de Caso

Foram elaborados 6 estudos de caso para a verificação da forma como as ferramentas CASE realizam a transcrição de UML para Java e vice-versa, e como elas interagem com o usuário ao realizar a engenharia de ida e volta (engenharia à frente e engenharia reversa).

De acordo com Gessenharter [Gessenharter, 2008], o suporte à modelagem UML por meio de diagramas de classe e a geração do código a partir dele são importantes tendências atuais das ferramentas. Mas, enquanto desenhar diagramas, geralmente, é bem suportado, a geração de código é de alcance limitado. Classes associativas, multiplicidades, agregação e composição não são corretamente ou não são processadas pela maioria dos geradores de código. Segundo Szlenk [Szlenk, 2008], uma razão pode ser o fato da semântica não ser formalmente definida na especificação da UML. Como resultado, as associações são normalmente transformadas em código com as mesmas propriedades das classes associadas ou conjuntos tipados correspondentes. Assim, é

difícil determinar como uma dada mudança em um modelo influencia o seu significado e, por exemplo, se uma determinada transformação de modelo preserva a semântica do modelo ou não.

A Arquitetura Dirigida por Modelos (*Model Driven Architecture* - MDA) da OMG [OMG, 2011] é uma abordagem para o processo de Desenvolvimento Dirigido por Modelos (*Model Driven Development* - MDD) utilizando a UML [UML, 2011]. MDD concentra-se em modelos, em que o código é gerado a partir dele. Apenas a qualidade de um modelo deve influenciar a qualidade do código, isto é, um modelo altamente detalhado que abrange todos os aspectos de uma aplicação deve resultar em código gerado automaticamente, sem necessidade de adaptações. Esta ambição requer boas ferramentas de geração de código. A maioria dos geradores de código produz um código que não cobre todos os elementos do modelo de entrada [Gessenharter, 2008].

Apesar de haver controvérsias, uma vez que, por exemplo, Bertrand Meyer [Meyer, 1997] considera que a associação é um elemento "estranho" à orientação a objetos, de acordo com Diskin e outros [Diskin et al., 2008], a associação entre classes é um construto centrado na modelagem OO. Entretanto, a semântica precisa de associações não foi definida, e apenas os tipos mais básicos são implementados nas modernas ferramentas de engenharia à frente e reversa.

Esta situação é problemática: os modelos são o resultado da análise e das fases de concepção e devem obedecer a todos os requisitos de um sistema e, portanto, o código gerado deve abranger toda a semântica do modelo. Caso contrário, a conformidade do código com o modelo continua a ser de responsabilidade exclusiva do desenvolvedor. Isto implica em mais linhas de código escrito à mão e propenso a erros [Gessenharter, 2008].

Gessenharter [Gessenharter, 2008] afirma que, além disso, a verificação do modelo é invalidada. O código gerado deve possuir todas as restrições que sempre devem ser realizadas, como é o caso em um SGBD relacional, onde a definição de chaves estrangeiras permite tambem definir a rejeição da remoção de uma tupla se isso resultar em uma violação das restrições do nível conceitual.

Os estudos de caso elaborados basearam-se em 3 (três) diagramas da UML. O diagrama de classe, por ser o diagrama mais utilizado da UML [Dobing & Parsons, 2006] e uma das mais populares apresentações visuais de desenho de software [Sharif & Maletic, 2009]; o diagrama de estrutura composta, por ter relação direta com o diagrama de classe e ser um novo recurso da UML 2.x; e o diagrama de sequência, por ser o principal diagrama de interação [Superstructure, 2011] e um dos principais diagramas comportamentais. O diagrama de sequência também possui novos recursos que surgiram a partir da UML 2.x como os fragmentos combinados e os

usos de interação.

O objetivo não é exaurir todas as possibilidades de modelagem, mas:

1. apontar detalhes importantes não implementados pelas ferramentas;
2. identificar elementos não suportados pelas ferramentas;
3. analisar casos de mapeamento incorreto de UML para Java;
4. analisar casos de mapeamento incorreto de Java para UML; e
5. verificar se a ferramenta interage com o usuário durante a engenharia de ida e volta.

DIAGRAMA DE CLASSE De acordo com a versão 2.4 do documento de especificação de Superestrutura da UML [Superstructure, 2011], publicado em janeiro de 2011, os elementos de modelo (*model elements*) mais comuns do diagrama de classe são:

- Associação;
- Agregação;
- Classe;
- Composição;
- Dependência;
- Generalização;
- Interface;
- Realização de Interface;
- Realização.

DIAGRAMA DE ESTRUTURA COMPOSTA De acordo com o documento de especificação de Superestrutura da UML [Superstructure, 2011], os elementos mais comuns do diagrama de estrutura composta são:

- Parte;
- Porta;

- Colaboração;
- Uso de Colaboração;
- Conector;
- Papel de ligação (*role binding*).

DIAGRAMA DE SEQUÊNCIA De acordo com o documento de especificação de Superestrutura da UML [Superstructure, 2011], dois dos vários elementos utilizados no diagrama de sequência são:

- Fragmentos de Interação;
- Usos de Interação.

A maioria das figuras que contêm os modelos utilizados nos estudos de caso foi retirada do documento de especificação de Superestrutura da UML [Superstructure, 2011]. As figuras utilizadas no estudo de caso 3 foram escolhidas segundo o critério da simplicidade para facilitar os trabalhos. As figuras que se referem ao estudo de caso 6 foram retiradas do trabalho de Micskei e Waeselynck [Micskei & Waeselynck, 2010].

O documento de especificação de Superestrutura da UML [Superstructure, 2011] possui algumas instruções de estilo (*Style Guidelines*) que identificam as convenções de notação recomendada pela especificação. Se aplicadas de forma consistente, a compreensão e a comunicação são facilitadas. Por exemplo, existe uma orientação de estilo que sugere que os nomes das classes devem estar em negrito e outra que sugere que os nomes das classes abstratas sejam escritos em itálico.

No entanto, não foi encontrado na versão 2.4 do documento de especificação de Superestrutura da UML [Superstructure, 2011] nenhum tipo de convenção de notação que mostre qual a forma correta de codificar qualquer estrutura dos modelos da UML para a linguagem Java. Pretendíamos apresentar, considerando a versão 1.6 da linguagem Java, apresentar "códigos esperados" resultantes do mapeamento dos modelos de cada estudo de caso para a linguagem Java, a fim de facilitar a verificação e validação dos resultados apresentados pelas ferramentas. Entretanto, como durante as pesquisas não foram encontrados relatos de boas práticas para geração de código Java a partir de um determinado elemento da UML, optamos por realizar a verificação analisando se as estruturas Java geradas são construções válidas da plataforma Java e se correspondem ou não a interpretações que podem ser entendidas.

3.4.1 ESTUDO DE CASO 1 - Estereótipos

Os estereótipos estendem o vocabulário da UML permitindo a criação de novos tipos de elementos semelhantes aos já existentes, porém específicos para o problema. Eles permitem aos usuários definir a semântica de notação, alargando assim a linguagem [Sharif & Maletic, 2009]. Um perfil da UML é o elemento que contém estereótipos, valores etiquetados e restrições que podem ser utilizados para estender o meta-modelo da UML [Ziadi et al., 2003]. O modelo da Figura 3.2 foi utilizado neste estudo de caso. Esta modelagem é utilizada em um dos documentos da UML e o objetivo aqui não é interpretar o significado, mas sim verificar o tratamento dado aos estereótipos pelas ferramentas. Em função disso não será discutido o perfil da UML correspondente a este estereótipo.

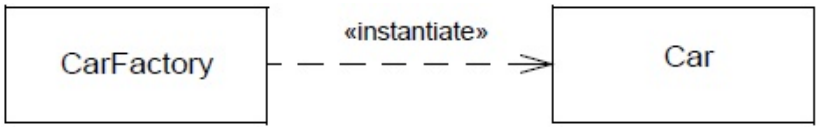

Figura 3.2. Estereótipo «instantiate» [Superstructure, 2011].

3.4.2 ESTUDO DE CASO 2 Associações: Associação Simples, Agregação e Composição; Extremidades de associação.

Segundo Milicev [Milicev, 2007], infelizmente, apesar do conceito de associação ser muito antigo e herdado de outras técnicas de modelagem de sucesso, ainda não existe um entendimento totalmente sem ambiguidade sobre ele. Esse conceito essencialmente simples foi significativamente melhorado para aumentar a expressividade da linguagem. Infelizmente, essas melhorias introduziram muitas novas ambiguidades na interpretação de todo o conceito.

Conceitualmente e visualmente, os relacionamentos de associação simples, agregação e composição têm diferenças. Porém, sua distinção no código não é clara, se é que é feita. Essa distinção geralmente não é feita no código pelas ferramentas. Na linguagem Java, esses três tipos de relacionamentos possuem código similar. Gessenharter [Gessenharter, 2008] afirma que classes de associação, multiplicidades, agregação e composição não são corretamente ou não são processadas pela maioria dos geradores de código. As ferramentas geralmente não consideram a diferença entre uma associação simples, uma agregação e uma composição.

Linguagens de programação orientadas a objeto não contêm sintaxe ou semântica para expressar associações diretamente. Desta forma, associações da UML têm

sido implementadas por uma combinação adequada de classes, atributos e métodos [Génova et al., 2003]. Conforme Gessenharter [Gessenharter, 2009], é difícil implementar relacionamentos por dois principais motivos: eles fornecem uma semântica complexa para entidades relacionadas e não são "construtos de primeira classe" nas linguagens de programação modernas. O desafio de implementar relacionamentos em código é resolver a semântica de elementos abstratos do modelo e transformá-las em referências ou ponteiros da linguagem destino.

Uma extremidade de associação pode ser adornada com um nome de papel (*role name*), uma multiplicidade, uma sentença de propriedades (*property string*), uma navegabilidade, um modificador de visibilidade, entre outros [Superstructure, 2011]. Uma associação descreve um conjunto de tuplas cujos valores se referem a instâncias tipadas. Uma instância de uma associação é chamada de ligação.

A Figura 3.3 mostra os tipos de extremidades de associação no relacionamento de associação simples (navegável, não navegável e não especificado), mapeia todas as combinações de extremidades. AB é navegável para os dois lados. CD não é navegável para nenhum lado. EF não possui navegabilidade especificada. GH é não navegável para G e é navegável para H. IJ não possui navegabilidade especificada para I e é navegável para J.

A Figura 3.4 mostra uma associação ternária. A Figura 3.5 mostra uma agregação com a extremidade de associação de A com navegabilidade não especificada e de B não navegável. O ponto ao lado das classes indica que a extremidade de associação ao lado de da classe A é uma propriedade de B e a extremidade de associação ao lado de B é propriedade de A. A Figura 3.6 mostra uma composição com extremidades de associação navegáveis e nomeadas.

3.4.2.1 Associação simples, Agregação e Composição

Segundo Gessenharter [Gessenharter, 2008], a situação da geração de código em relação às associações é surpreendente. As ferramentas avaliadas não consideram a diferença entre associações "simples", agregações ou composições.

Akehurst e outros [Akehurst et al., 2007] consideram útil o uso de referências, mas o descarta porque este mecanismo certamente não impede o acesso a partes excluídas. Eles propõem uma ideia simples aplicável: ligações são sempre acessíveis, diretamente pelas instâncias referenciadas ou indiretamente pela associação. Se o todo de uma composição é excluído, os valores de ligações que referenciam suas partes devem ser definidos como nulos se as ligações não são uma instância da própria composição. Se nem o todo nem suas partes são referenciados por uma propriedade de outra classe,

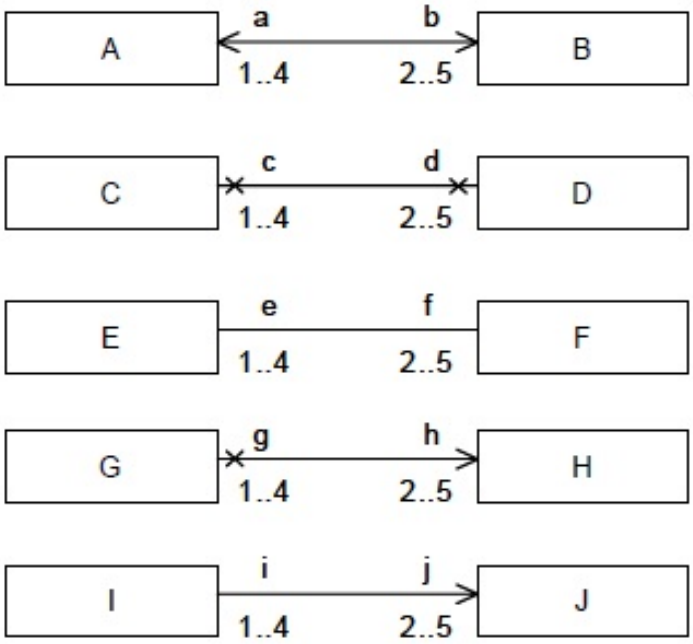

Figura 3.3. Tipos de extremidades de associação no relacionamento de associação simples [Superstructure, 2011].

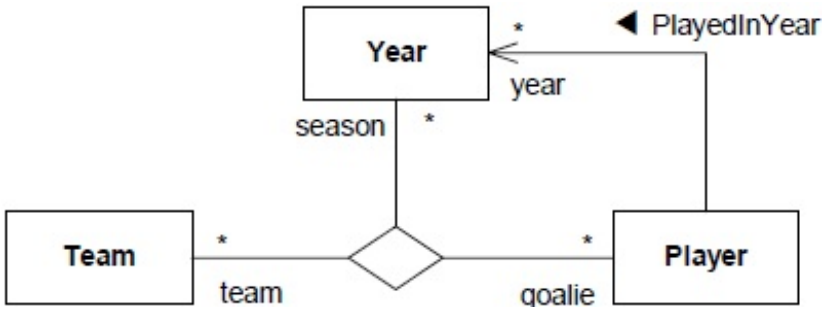

Figura 3.4. Associação binária e ternária [Superstructure, 2011].

Figura 3.5. Agregação [Superstructure, 2011].

ambos permanecem ligados, mas inacessíveis a partir de qualquer outro lugar e podem ser removidos pelo coletor de lixo. Um problema não resolvido é que a destruição das ligações pode violar multiplicidades das classes associadas.

Os modelos das figuras 3.3, 3.4, 3.5 e 3.6 são mostrados para abordar com exemplos os tipos de associação. Nosso objetivo não foi estudar todas as possibilidades e

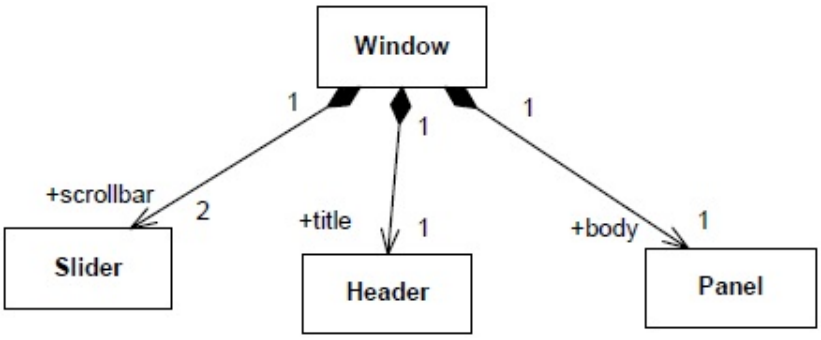

Figura 3.6. Relacionamento de Composição [Superstructure, 2011].

este estudo de caso utilizou apenas o modelo da Figura 3.6.

3.4.2.2 Adornos da extremidade de associação

De acordo com Akehurst e outros [Akehurst et al., 2007], o conceito de associação não existe em linguagens de programação orientadas a objeto como Java. Consequentemente, a fim de gerar o código de um modelo da UML é necessário elaborar um mapeamento para a linguagem de programação OO escolhida.

Os limites superiores das multiplicidades são rudimentarmente implementados. Se ele for maior que 1, o atributo que representa a extremidade de associação é do tipo coleção. Assim, apenas dois estados são identificados, um limite superior que é igual a 1 ou um de maior valor. As ferramentas, geralmente, não rejeitam uma nova ligação se o limite superior é violado [Gessenharter, 2008].

Uma propriedade (*property*) é um elemento da UML que pode ser considerado sob dois pontos de vista: i) no meta-modelo, em que ela é um atributo (*structural feature*); ou ii) é um valor com nome que denota um aspecto de um elemento, por exemplo, uma classe, uma associação, entre outros [Superstructure, 2011]. Algumas propriedades são predefinidas e outras podem ser definidas pelo usuário. As propriedades definem características de elementos da modelagem.

As propriedades podem ser apresentadas de várias formas. A forma mais usual é a sentença de propriedade (*property string*). O formato geral é um conjunto de sentenças da forma "nome=valor", por exemplo "abc, def=xyz". A primeira propriedade, "abc", implica uma propriedade booleana e é uma abreviação de "abc=true". A segunda propriedade, "def", tem valor "xyz". A especificação da UML define 0, 1 ou mais propriedades para cada um dos elementos da modelagem [Superstructure, 2011]. A sentença de propriedade "unique" para uma extremidade de associação representa, portanto, que "unique=true".

As sentenças de propriedade são colocadas entre chaves e servem para dar detalhes da associação. Por exemplo, a propriedade "subset <nome da propriedade>" serve para mostrar que a extremidade é um subconjunto da propriedade chamada <nome da propriedade> [Superstructure, 2011]; a propriedade "union" serve para mostrar que a extremidade é obtida como sendo a união de seus subconjuntos; a propriedade "ordered" serve para mostrar que a extremidade representa um conjunto ordenado; a propriedade "unique" serve para mostrar que a extremidade representa uma coleção que não permite que os mesmos elementos apareçam mais de uma vez. As propriedades das extremidades de associação são pouco apoiadas pelas ferramentas [Gessenharter, 2008].

A Figura 3.7 mostra um modelo com nomes da extremidade de associação, sentenças de propriedade e multiplicidade. Se existe a indicação de navegabilidade da classe Customer para a classe Purchase, então a extremidade purchase é uma propriedade da classe Customer.

O trabalho realizado por Gessenharter [Gessenharter, 2008] verificou o estado da arte da geração de código a partir do diagrama de classe da UML avaliando 13 ferramentas - entre elas Astah*, RSA e Enterprise Architect, avaliadas neste trabalho. Todas as ferramentas avaliadas geraram código para as associações utilizando atributos nas classes. No caso de uma associação binária, cada classe foi gerada com um atributo que pode armazenar uma referência para a classe oposta. O trabalho verificou que na maioria das ferramentas os limites superiores das multiplicidades são rudimentarmente implementados. Além disso, a ausência da representação da propriedade navegabilidade no código leva a associações que são navegáveis em ambos os sentidos. Esta suposição é problemática porque as associações não navegáveis podem ser úteis com classes de associação e, portanto, devem ser autorizadas e traduzidas para o código [Gessenharter, 2008].

O modelo da Figura 3.7 foi utilizado neste estudo de caso.

Figura 3.7. Adornos da extremidade de associação: navegabilidade, multiplicidade, sentença de propriedades e nomes da extremidade de associação [Superstructure, 2011].

3.4.3 ESTUDO DE CASO 3 – Classes concretas, Classes Abstratas e Interfaces: *extends* e *implements*.

O conceito de herança é bem estudado na literatura e caracteriza as linguagens de programação orientadas a objeto [Chirilă et al., 2010] [Parkinson & Bierman, 2008] [Smans et al., 2009]. A linguagem Java define: i) a relação de herança (*extends*) entre classes; ii) a relação de herança entre interfaces (*extends*); e iii) a relação de implementação (*implements*) entre classe e interface. A relacao *extends* é definida utilizando conceitos de herança simples e não utiliza conceitos de herança múltipla [Lewis & Loftus, 2007]. No entanto, Java suporta herança múltipla entre interfaces, isto é, uma interface I1 pode herdar as características de uma interface I2 e de uma interface I3. Apesar de herança ser considerada no âmbito da relação *extends*, de acordo com Chirila e outros [Chirilă et al., 2010], os aspectos de herança devem ser considerados tanto em relação a *extends* como em relação a *implements*.

Segundo Tempero e outros [Tempero et al., 2008], existem diferentes tipos de vértices para distinguir diferentes tipos de "tipos" (*types*), isto é, classes (C), interfaces (I), enums (E), annotations (A) e exceptions (Ex). Podemos distinguir as classes e interfaces por elas terem relações bastante diferentes entre si e desempenharem papéis diferentes em uma hierarquia de herança. Distinguimos *enums* e *annotations* porque, embora sejam implementadas, respectivamente, como classes especializadas e interfaces, suas funções são diferenciadas.

A Figura 3.8 mostra um relacionamento de herança simples e a Figura 3.9 mostra um relacionamento de herança múltipla. Um problema relacionado à falta de implementação de herança múltipla em Java, segundo Warth e outros [Warth et al., 2006], é a necessidade de duplicação de uma quantidade significativa de código em determinados momentos. Kegel e Steimann [Kegel & Steimann, 2008] ressaltam os aspectos positivos e negativos da herança em programas orientados a objetos. Como aspecto positivo permite o reuso de implementação com o mínimo de esforço e como aspecto negativo estabelece forte acoplamento entre as classes e tende a inchar as interfaces das subclasses com membros desnecessários. Esse aspecto negativo é particularmente um problema em linguagens como Java, cuja noção de subclasse mistura os conceitos de herança e subtipagem (*typing*) de modo que a primeira não pode ser desfrutada sem o último.

1. Exemplo de herança simples entre classes: C1 *extends* C2

2. Exemplo de herança múltipla entre interfaces: I1 *extends* I2, I3

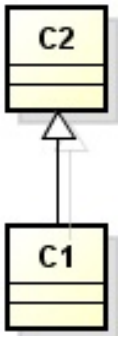

Figura 3.8. Herança Simples entre classes.

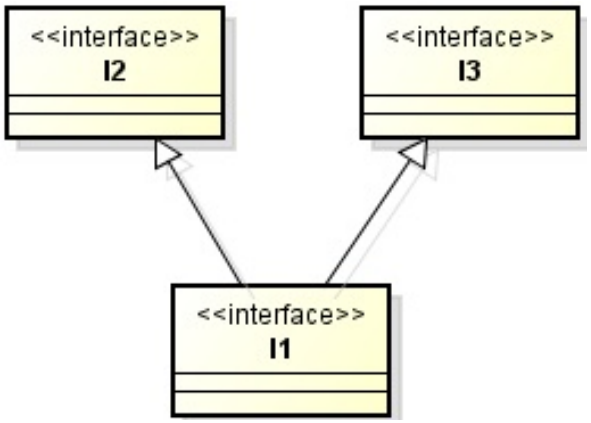

Figura 3.9. Herança Múltipla entre interfaces.

Os trabalhos científicos pesquisados continham apenas modelos simples, explorando apenas o relacionamento *extends*. Para facilitar os trabalhos, foi usado o critério simplicidade para escolher o modelo utilizado no estudo de caso. A Figura 3.10 mostra um modelo com classes concretas, classes abstratas e interfaces, e os relacionamentos *extends* e *implements*.

A Figura 3.10 mostra um modelo em que a classe concreta Pessoa implementa a interface IPessoa; a classe concreta MGEndereço estende a classe abstrata Endereço que, por sua vez, implementa a interface IEndereço. O modelo da Figura 3.10 foi utilizado neste estudo de caso.

3.4.4 ESTUDO DE CASO 4 - Classes, Interfaces e Anotações

O diagrama de classe é um dos um dos diagramas mais utilizados da UML e presente na análise e desenho de software [Artale et al., 2010]. Conceitos de modelagem de classe são os conceitos mais utilizados em UML. Alguns projetos de desenvolvimento criam apenas modelos contendo apenas diagramas de classe [France et al., 2006].

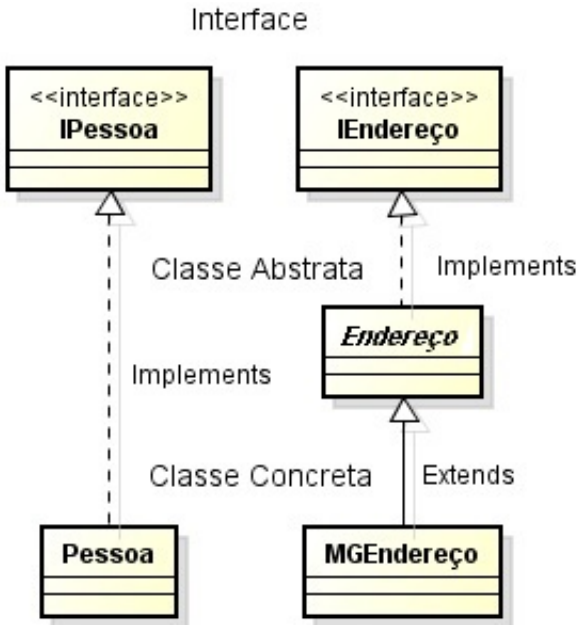

Figura 3.10. Classes concretas, Classes abstratas e Interfaces.

Dobing e Parsons [Dobing & Parsons, 2006] realizaram um trabalho experimental em que foram entrevistados 171 analistas que utilizavam a UML e outros 11 que utilizavam componentes da UML como parte de outra metodologia OO. Os entrevistados relataram ter se envolvido em uma média de 27 projetos (cerca de 6,2 utilizando a UML), ao longo de uma carreira de 15 anos em tecnologia da informação. O trabalho realizado por Dobing e Parsons [Dobing & Parsons, 2006] relata que 73% dos entrevistados utilizavam o diagrama de classes em dois terços ou mais de seus projetos.

Kollmann e outros [Kollman et al., 2002] realizaram um trabalho que analisou o estado da arte das ferramentas de engenharia reversa em relação ao mapeamento de Java para diagramas de classe da UML. Alguns elementos do diagrama de classe analisados foram: i) número de classes; ii) números de associações; iii) tipos de associações; iv) multiplicidades; entre outros. Alguns desses elementos foram utilizados como métricas por Manso e outros [Manso et al., 2003] para analisar a complexidade estrutural do diagrama de classe. Com o objetivo de analisar ferramentas de engenharia de ida e volta e estender o trabalho de Kollmann e outros [Kollman et al., 2002], este estudo de caso tem como objetivo analisar como é feito o mapeamento de alguns elementos da UML para a linguagem Java, entre eles as anotações (*annotations*) padronizadas de classes.

Foram analisados como esses conceitos são considerados durante o mapeamento de UML para Java. A Figura 3.11 mostra um diagrama de classes com classes (abstratas e não abstratas), atributos (estáticos e não estáticos), métodos (estáticos e não estáticos)

e interfaces. Durante e após esse processo de mapeamento, foram feitas verificações e considerações sobre a interação da ferramenta com o usuário e a consistência dos artefatos de saída da ferramenta.

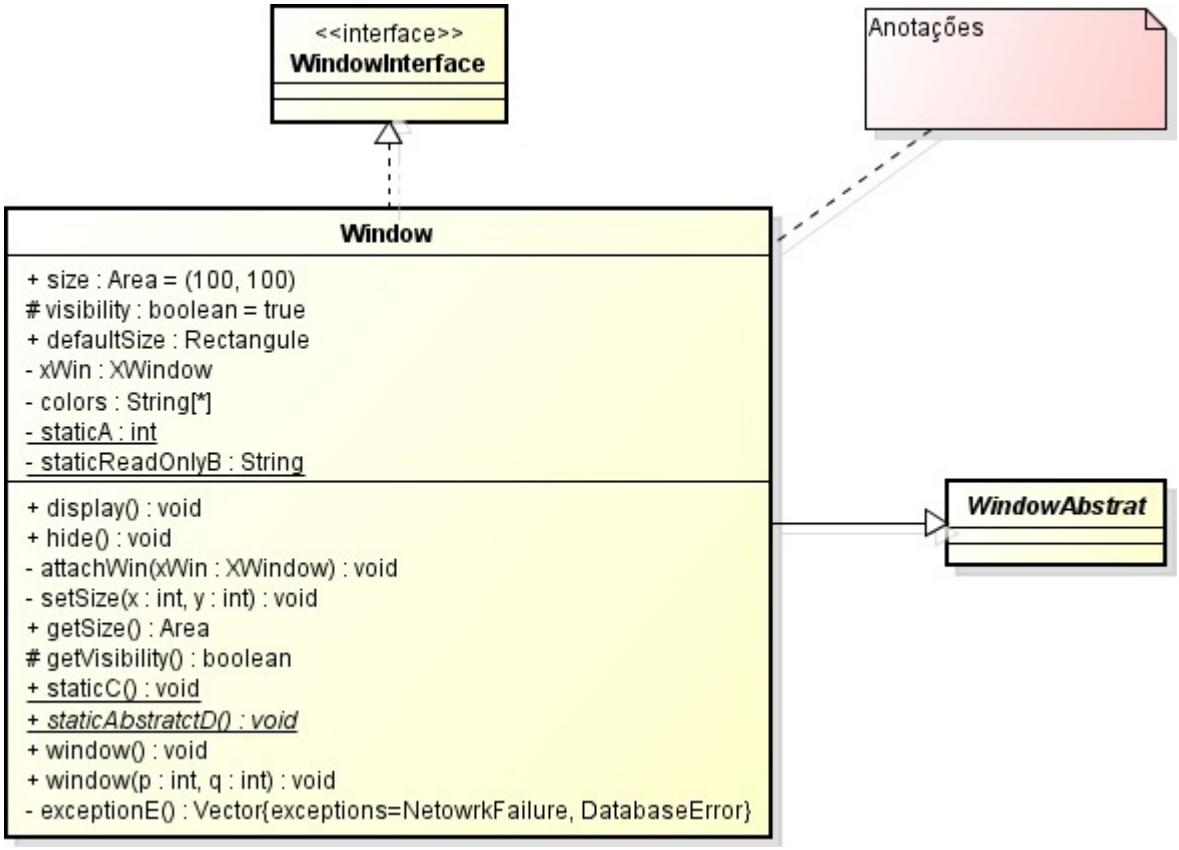

Figura 3.11. Diagrama de Classe [Superstructure, 2011] (Adaptado).

3.4.5 ESTUDO DE CASO 5 - Estrutura Composta

A função do diagrama de estrutura composta é estender a capacidade de modelagem da UML, além das classes e relacionamentos e é, principalmente, auxiliar a modelagem de estruturas internas de classes com um conceito mais bem definido de decomposição [Oliver & Luukala, 2006]. Segundo France e outros [France et al., 2006], um diagrama de estrutura composta descreve a estrutura interna de um classificador estruturado. Um classificador estruturado pode ser associado com portas que representam pontos de interações com o classificador.

Não foram encontrados trabalhos científicos que abordam a relação entre UML e Java no que se refere ao diagrama de estrutura composta. Não foram encontrados

relatos de boas práticas para geração de código Java a partir do diagrama de estrutura composta.

3.4.5.1 Estrutura composta I

A Figura 3.12 mostra um diagrama de classe, à esquerda, que corresponde em certo sentido ao diagrama de estrutura composta, à direita. O diagrama de estrutura composta complementa o diagrama de classes e mostra detalhes da estrutura interna de uma classe. O fato de a multiplicidade entre as classes Wheel e Engine serem N:N no diagrama de classes e 2:1 no diagrama de estrutura composta não significa que há inconsistência. Várias objetos da classes Wheel pode se relacionar com vários objetos da classe Engine. Mas, no contexto da classe Car, o classificador estruturado Car possui duas partes: uma coleção de dois elementos do tipo Wheel, chamado de rear (duas rodas traseiras) e um elemento do tipo Engine (motor). Desta forma, pode-se entender que um objeto do tipo carro possui duas rodas traseiras que são tracionadas por um motor através de um eixo. É importante observar que o diagrama de estrutura composta considera os elementos em nível de objetos, isto é, em instâncias de classe.

Outro aspecto importante que se pode verificar no diagrama de estrutura composta é a definição do relacionamento do classificador estruturado, o "todo", com as partes. A linha que contorna a parte Wheel é contínua, enquanto a linha que contorna a parte Engine é tracejada. A linha contínua representa o relacionamento de composição, já a linha tracejada representa o relacionamento de associação simples. O diagrama de estrutura composta da Figura 3.12 foi utilizado neste estudo de caso.

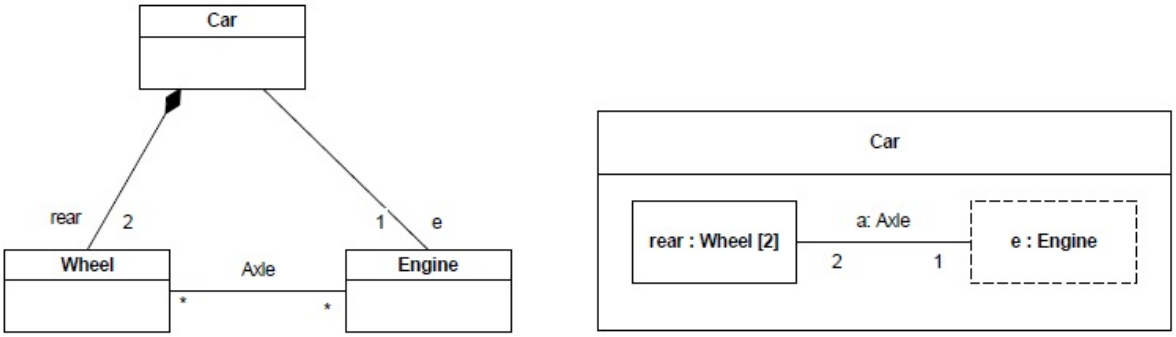

Figura 3.12. Um diagrama de classe correspondente a um diagrama de estrutura composta. [Superstructure, 2011]

3.4.5.2 Estrutura Composta II

A Figura 3.13 e a Figura 3.14 mostram que, num diagrama de estrutura composta, é possível fazer especializações (vários tipos de Wheel) com mais facilidade que num diagrama de classe. A declaração de quatro objetos do mesmo tipo, assim como a definição de seus relacionamentos é uma tarefa mais simples no diagrama de estrutura composta que no diagrama de classe.

Porém, se o diagrama de estrutura composta não puder ser utilizado como entrada para o mapeamento de UML para Java, ou seja, se não puder ser feito o mapeamento desse diagrama para o código, os detalhes particulares que podem ser verificados no diagrama ficarão como informações perdidas após o processo de mapeamento. O modelo da Figura 3.13 foi utilizado neste estudo de caso.

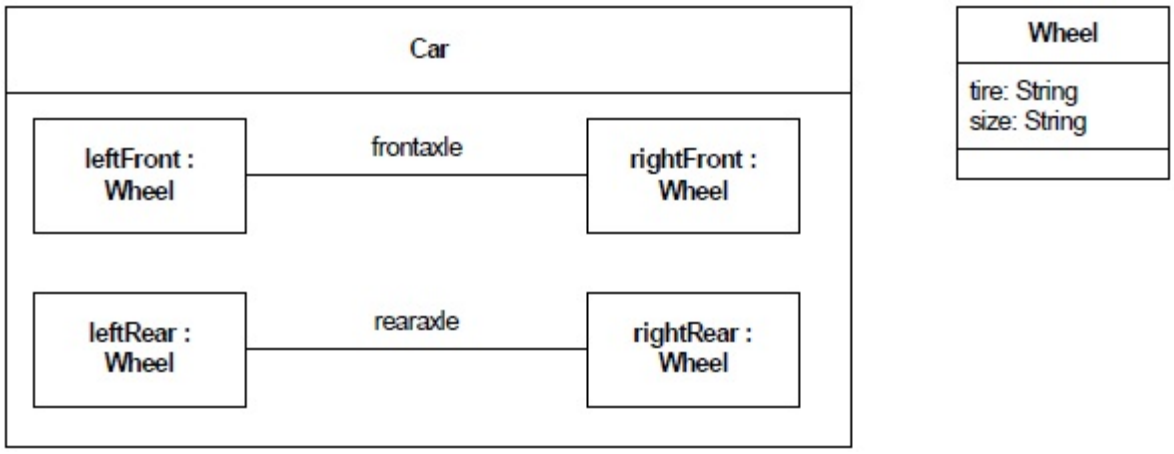

Figura 3.13. Diagrama de estrutura composta da classe Car com a parte Wheel (i).

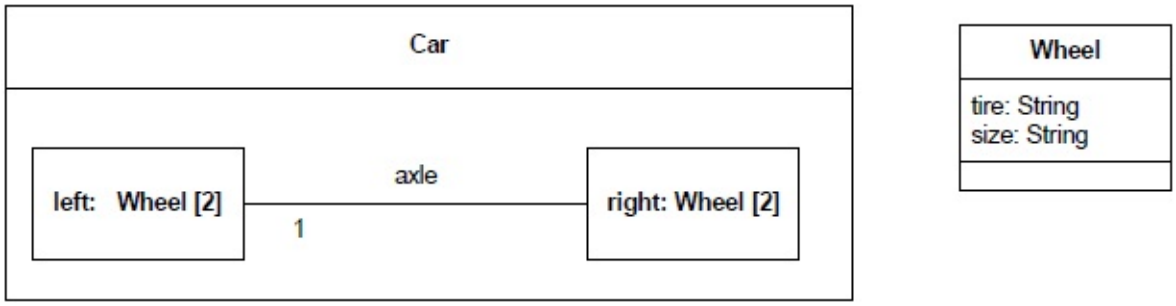

Figura 3.14. Diagrama de estrutura composta da classe Car com a parte Wheel (ii).

3.4.6 ESTUDO DE CASO 6 - Fragmento Combinado

Os diagramas de comportamento da UML incluem muitos conceitos que não são apresentados nas linguagens de programação mais populares, como C++ e Java, por exemplo: eventos, estados, histórico de estados (*history states*), entre outros. Isto significa que não existe um mapeamento um-para-um entre um *statechart* - composto por estados, eventos e transições - e sua implementação." [Jakimi & Elkoutbi, 2009].

A partir da UML 2.0, o diagrama de sequência sofreu algumas alterações. A versão 2.0 da UML modificou significativamente o Diagrama de Sequência e a expressividade da linguagem foi consideravelmente aumentada [Micskei & Waeselynck, 2010].

Não foram encontrados trabalhos científicos que abordam a relação entre UML e Java no que se refere aos fragmentos combinados. Não foram encontrados relatos de boas práticas para geração de código Java a partir destes elementos do diagrama de sequência.

3.4.6.1 Fragmento Combinado I

A Figura 3.15 mostra um diagrama sequência que possui o fragmento combinado *alt*. Este diagrama foi utilizado neste estudo de caso. O fragmento combinado *alt* representa uma estrutura condicional. As operações que fazem parte do primeiro compartimento só são executados se a condição for satisfeita. Do contrário, apenas os métodos do segundo compartimento serão executados.

3.4.6.2 Fragmento Combinado II

A Figura 3.16 mostra um diagrama de sequência que possui o fragmento combinado *loop*. O objetivo dos fragmentos combinados é resumir um conjunto de informações numa estrutura mais simples. O termo *loop* indica iteração, assim como *alt* representa uma estrutura condicional. Porém, há casos em que essas estruturas feitas para auxiliar no entendimento o tornam confuso, como pode ser verificado na Figura 3.16. Dentro das limitações do fragmento combinado *loop* estão o método m2 completamente e o método m1 parcialmente. Nessas condições, surge a questão: ambos os métodos m1 e m2 são "afetados" pelo fragmento combinado *loop* ou apenas o método m2?

3.5 Aplicação dos Estudos de Caso

Esta seção apresenta os resultados obtidos a partir da aplicação dos estudos de caso apresentados na seção 3.4, utilizando as três ferramentas selecionadas - Astah*, Ra-

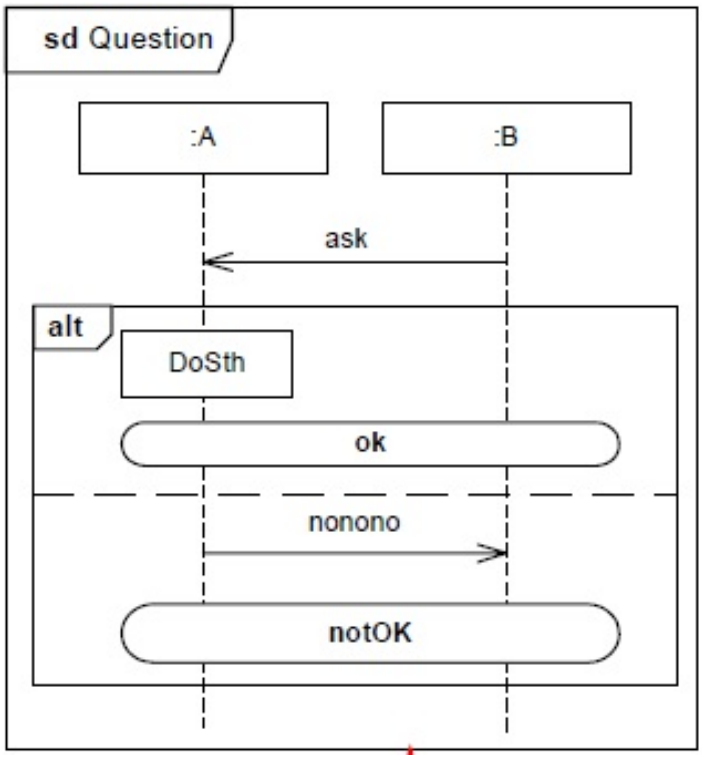

Figura 3.15. Diagrama de sequência com o fragmento combinado *alt*.

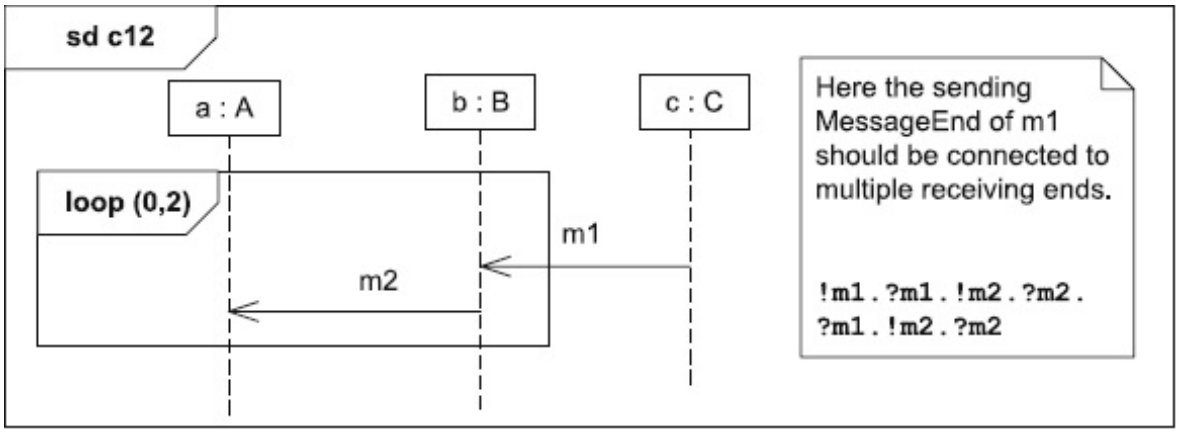

Figura 3.16. Diagrama de sequência com o fragmento combinado *loop* gerando ambiguidade.

tional Software Architect e Enterprise Architect. As subseções desta seção estão em conformidade com as subseções da seção 3.4.

Os comentários do código gerado que possuíam apenas informações como data, autor, versão do projeto e afins, foram retirados. Apenas as informações relevantes e pertinentes à análise foram preservadas na apresentação do código. Os códigos gerados pelas ferramentas referente a cada estudo de caso são apresentados.

3.5.1 ESTUDO DE CASO 1 - Estereótipos

A Figura 3.17 mostra o modelo elaborado na Astah* para o estudo de caso 1. O relacionamento de dependência entre a classe CarFactory e a classe Car é estereotipado. O código gerado pela Astah* é mostrado na Figura 3.18. Foram mapeadas apenas as classes. A informação conceitual do estereótipo e do relacionamento de dependência não foi mapeada.

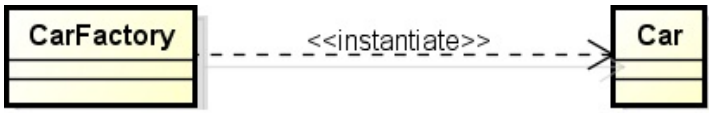

Figura 3.17. Modelo elaborado na ferramenta Astah* para o estudo de caso 1.

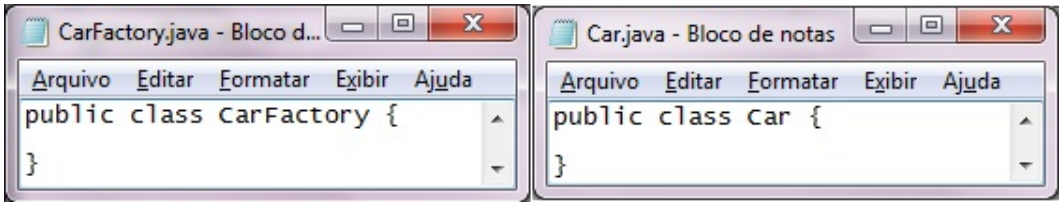

Figura 3.18. Código gerado pela Astah* para o modelo da Figura 3.17.

Considerando os critérios estabelecidos neste capítulo, Astah* teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Parcialmente consistente (1). Astah* gerou as duas classes conforme apresentado no modelo. Porém, não fez referência no código ao estereótipo e ao relacionamento entre a classe CarFactory e a classe Car.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Não se aplica (0). Não existem estruturas no modelo que podem ser mapeadas de mais de uma forma.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). Após realizar a engenharia reversa, apenas as classes foram recuperadas. O relacionamento de dependência e o estereótipo não foram recuperados, uma vez que não foram mapeados na engenharia à frente.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). O modelo foi elaborado conforme o modelo do estudo de caso. No entanto, a ferramenta não utiliza todas as informações constantes no modelo ao fazer o mapeamento do modelo para o código.

A Figura 3.19 mostra o modelo elaborado na RSA. A simplicidade do modelo facilita a geração do código. Porém, o estereótipo do relacionamento de dependência

entre a classe CarFactory e a classe Car não foi mapeado no código. A Figura 3.20 mostra o código gerado pela RSA.

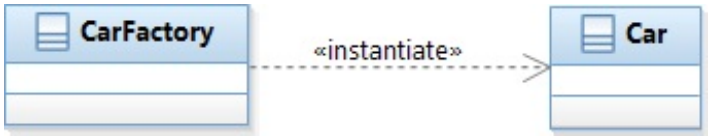

Figura 3.19. Modelo elaborado na ferramenta RSA para o estudo de caso 1.

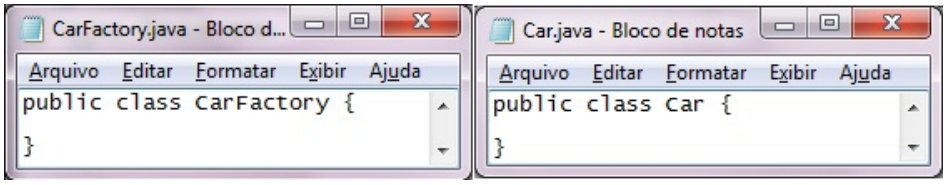

Figura 3.20. Código gerado pela RSA a partir do modelo da Figura 3.19.

Considerando os critérios estabelecidos neste capítulo, RSA teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Parcialmente consistente (1). O mapeamento realizado pela RSA produziu as duas classes, conforme apresentado no modelo. Porém, o estereótipo e o relacionamento de dependência não foram mapeados.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Não se aplica (0). Não existem estruturas no modelo que podem ser mapeadas de mais de uma forma.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). Apenas as duas classes foram recuperadas pela engenharia reversa. O relacionamento de dependência e o estereótipo não foram recuperados, uma vez que não foram mapeados na engenharia à frente.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). O modelo foi elaborado conforme o modelo do estudo de caso. No entanto, a ferramenta não utiliza todas as informações constantes no modelo ao fazer o mapeamento do modelo para o código.

A Figura 3.21 mostra o modelo elaborado na EA. Não houve dificuldade para a elaboração do modelo. Porém, EA não mapeou para o código o estereótipo nem relacionamento de dependência entre a classe CarFactory e a classe Car. A Figura 3.22 mostra o código gerado pela EA.

Considerando os critérios estabelecidos neste capítulo, EA teve o seguinte desempenho:

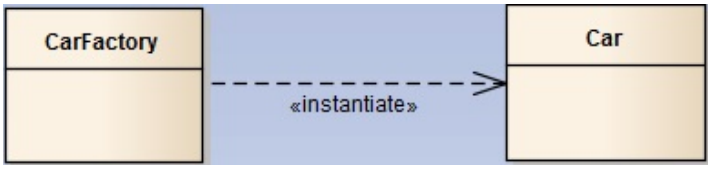

Figura 3.21. Modelo elaborado na ferramenta EA para o estudo de caso 1.

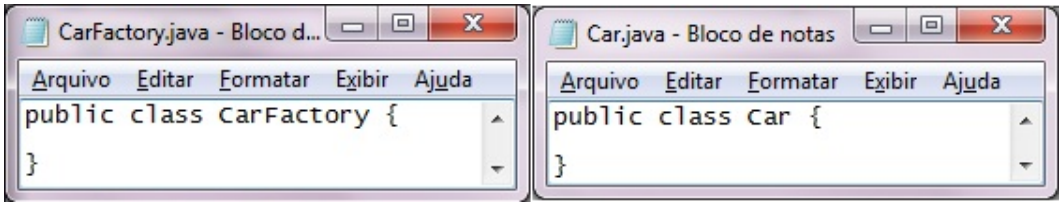

Figura 3.22. Código gerado pela RSA a partir do modelo da Figura 3.21.

C1 – Confiabilidade do Código Gerado: Parcialmente consistente (1). EA gerou as duas classes conforme apresentado no modelo. No entanto, não fez referência no código ao estereótipo e ao relacionamento entre a classe CarFactory e a classe Car.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Não se aplica (0). Não existem estruturas no modelo que podem ser mapeadas de mais de uma forma.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). A engenharia reversa recuperou apenas as classes. Como o relacionamento de dependência e o estereótipo foram mapeados na engenharia à frente, não foram recuperados.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). O modelo foi elaborado conforme o modelo do estudo de caso. No entanto, a ferramenta não utiliza todas as informações constantes no modelo ao fazer o mapeamento do modelo para o código.

3.5.2 ESTUDO DE CASO 2 - Associações: Associação Simples, Agregação e Composição; Extremidades de associação

3.5.2.1 Associação simples, Agregação e Composição

A Figura 3.23 mostra o modelo elaborado na ferramenta Astah*. O que há de destaque neste estudo de caso é a multiplicidade que existe na extremidade de associação da classe Slider. A Figura 3.24 mostra o código criado pela Astah*. Este exemplo permite verificar que a opção *default* da Astah* para representar um conjunto de dados do mesmo tipo é a criação de uma Collection. Não foi solicitado nenhum tipo de inter-

venção do usuário para a escolha da estrutura mais adequada para a representação da multiplicidade. Poderia ter criado um List, um Array ou um Vector, por exemplo, mas foi criada uma Collection. A Astah* também não foi precisa na definição do tamanho da estrutura de dados. A multiplicidade da classe Slider é 2 e não infinita, como foi mapeado no código.

É possível perceber que os nomes das extremidades de associação (*scrollbar* na extremidade da classe Slider; *title* na extremidade da classe Header; e *body* na extremidade da classe Panel) foram utilizados pela ferramenta. O atributo do tipo Slider declarado na classe Window recebeu o nome de "*scrollbar*"; o atributo do tipo Header declarado na classe Window recebeu o nome de "*title*"; o atributo do tipo Panel declarado na classe Window recebeu o nome de "*body*".

Entretanto, a informação visual e conceitual dos relacionamentos de composição foi perdida. Observando o modelo, é possível afirmar que o relacionamento entre a classe Window e as outras classes é de composição. Observando o código, não é possível afirmar o mesmo. O relacionamento de composição é generalizado para o de associação simples. Não foi exibido nenhum tipo de mensagem informando isso ao usuário.

A partir do código gerado, foi realizada a engenharia reversa. Todas as classes foram recuperadas, assim como os nomes nas extremidades de associação. Entretanto, os relacionamentos entre as classes foram mapeados como associações simples, em vez de composição. Como a multiplicidade na extremidade de associação da classe Slider foi mapeada para o código como N, a engenharia de volta mapeou a multiplicidade como "*", em vez de 2.

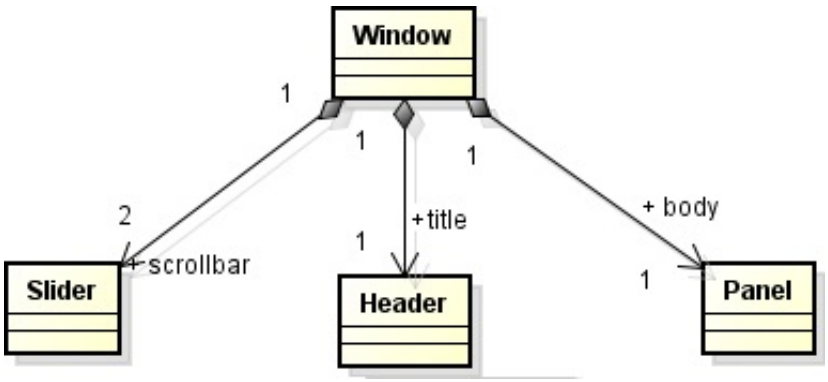

Figura 3.23. Modelo elaborado na ferramenta Astah* para o estudo de caso 2(i).

A Figura 3.25 mostra o modelo elaborado na RSA referente ao modelo do estudo de caso 2. A Figura 3.26 mostra o código gerado. O relacionamento de composição foi informação perdida, uma vez que RSA o mapeou como associação simples. A multipli-

Figura 3.24. Códigos gerados pela Astah* para o modelo da Figura 3.23.

cidade da extremidade de associação da classe Slider foi considerada no mapeamento. Foi gerada uma coleção de Slider de tamanho 2 utilizando arranjos. Os arranjos são a estrutura padrão utilizada para representar coleções com limite superior definido. Apesar de haver outras formas de mapeamento da multiplicidade, RSA não forneceu essas opções ao usuário.

A engenharia reversa, utilizando o código gerado, produziu um modelo correspondente ao modelo inicial. As multiplicidades e os nomes nas extremidades de associação foram gerados corretamente. No entanto, o relacionamento de composição foi informação perdida, a engenharia reversa gerou relacionamentos de associação simples.

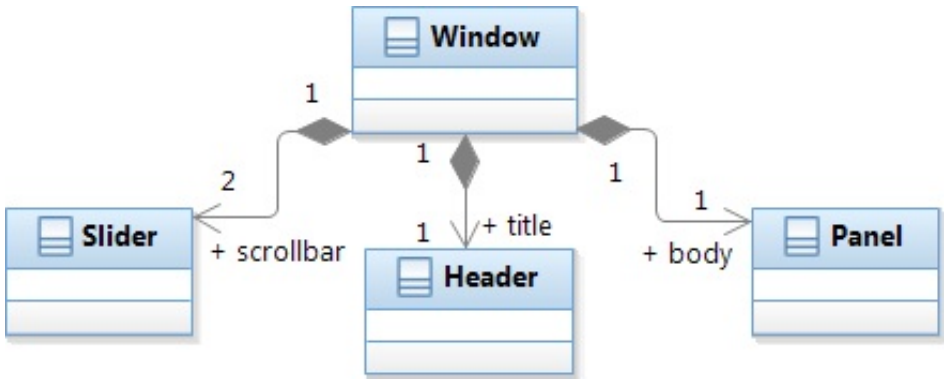

Figura 3.25. Modelo elaborado na ferramenta RSA para o estudo de caso 2(i).

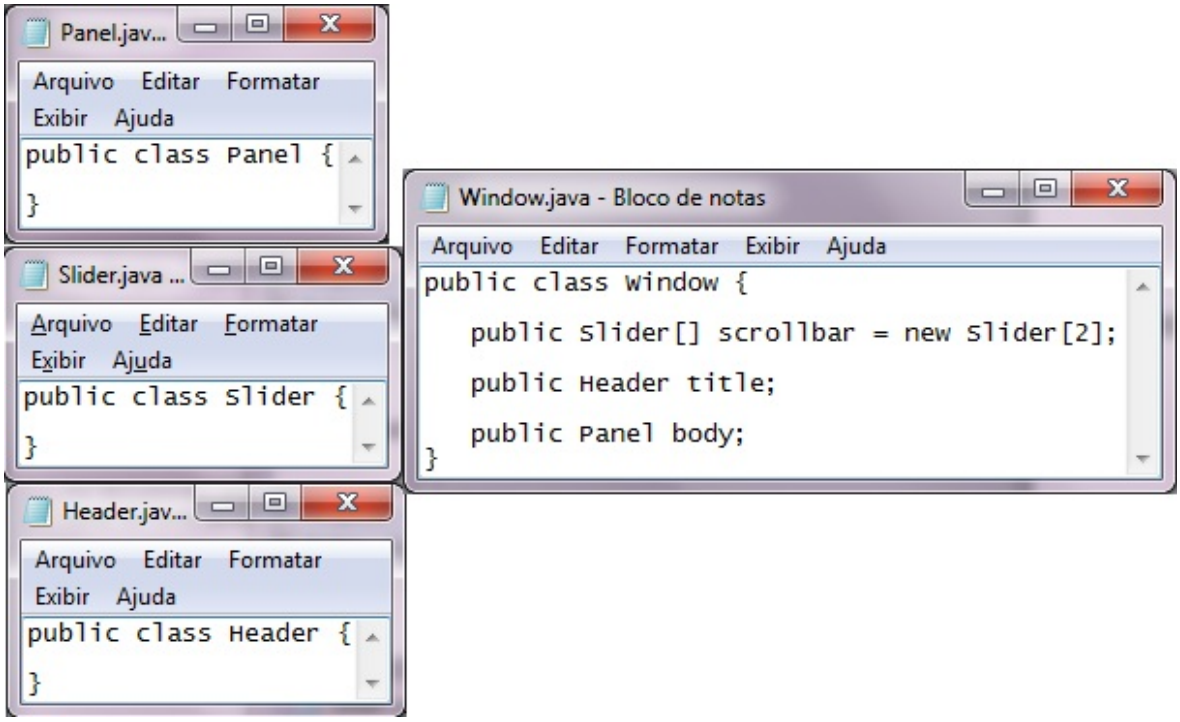

Figura 3.26. Códigos Gerados Pela RSA Para o Modelo da Figura 3.25

A Figura 3.27 mostra o modelo elaborado na EA referente ao modelo do estudo de caso 2. A Figura 3.28 mostra o código gerado pela EA a partir do modelo da Figura 3.27. É possível observar que não há no código qualquer elemento que faça menção ao relacionamento de composição. Além disso, a multiplicidade definida na extremidade de associação da classe Slider foi ignorada. Esperava-se o mapeamento de uma coleção do tipo Slider. A navegabilidade foi mapeada corretamente, foi criado um atributo referente a cada classe que está associada com a classe Window. Os nomes dos atributos foram os mesmos dos nomes das extremidades de associação

A engenharia reversa recuperou as classes, os relacionamentos e os nomes das extremidades de associação. Os nomes das extremidades de associação, assim como a navegabilidade dos relacionamentos foram recuperados. No entanto, a multiplicidade o elemento que caracteriza o relacionamento de composição foram informações perdidas.

3.5.2.2 Adornos da extremidade de associação

A Figura 3.29 mostra o modelo elaborado na Astah* para o estudo de caso 2. A Figura 3.30 mostra o código gerado pela Astah*. Os códigos das classes Purchase e Account foram gerados conforme o esperado. O código da classe Customer foi gerado com dois atributos, observando a multiplicidade. No entanto, a Astah* não especificou o

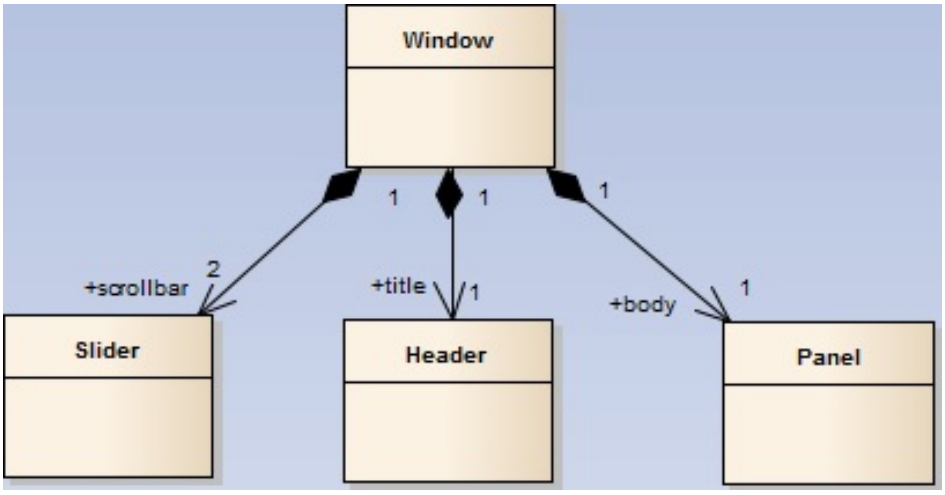

Figura 3.27. Modelo elaborado na ferramenta EA referente ao estudo de caso 2(i).

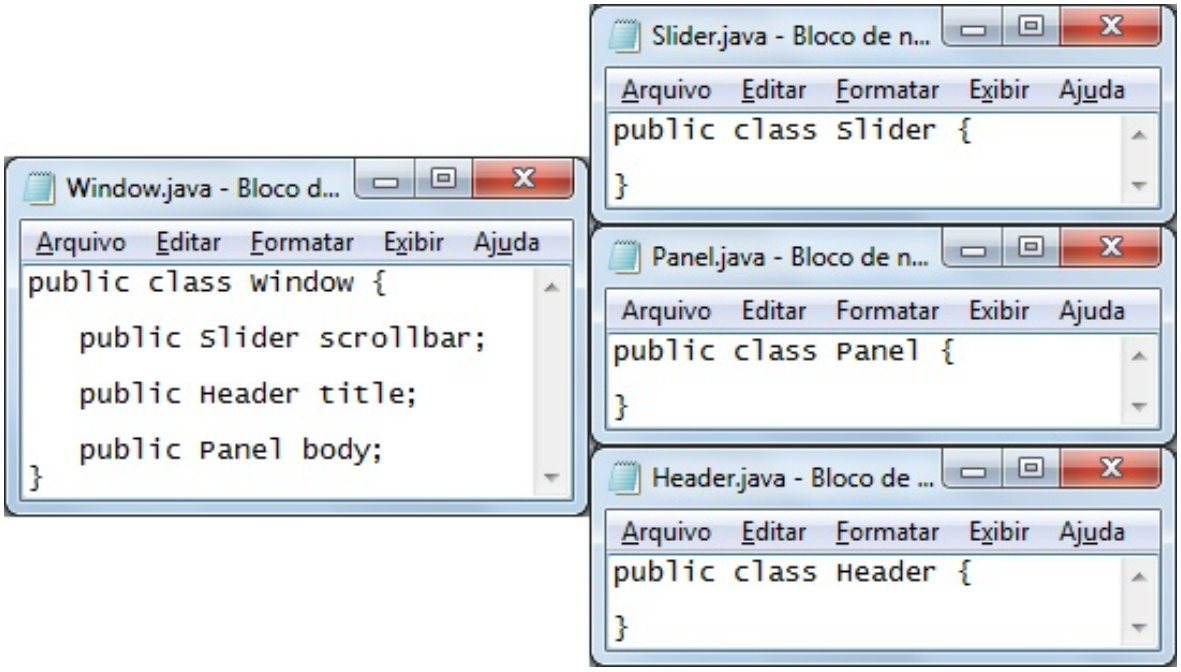

Figura 3.28. Código gerado pela EA a partir do modelo da Figura 3.27.

tamanho da coleção do tipo Account. Diferentemente da coleção do tipo Purchase, que não tem limite superior definido, a coleção de Account é limitada em 5 no modelo. Nos dois casos, a estrutura Collection foi utilizada, por *default*, para representar a coleção. Não foi fornecido ao usuário nenhum tipo de opção de mapeamento quanto à escolha da estrutura mais adequada para representar a coleção.

A multiplicidade não foi ignorada, mas o mapeamento não observou os detalhes especificados no modelo. As sentenças de propriedade "ordered, unique e unique" não

foram consideradas no mapeamento. Como pode ser visto no código gerado, não foi feita nenhuma referência a estas estruturas.

A engenharia reversa recuperou as classes e os nomes das extremidades de associação, conforme o modelo. No entanto, a multiplicidade da extremidade de associação da classe Account foi recuperada sem especificar o limite superior estabelecido no modelo. A navegabilidade, que existia apenas na direção de Customer para Purchase e de Customer para Account, após a engenharia reversa existe em todos as direções. Além disso, as sentenças de propriedade foram informações perdidas.

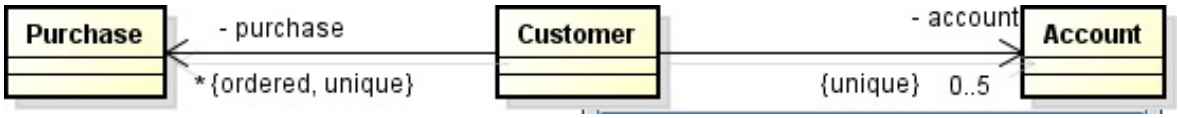

Figura 3.29. Modelo elaborado na ferramenta Astah* referente ao estudo de casoe 2(ii).

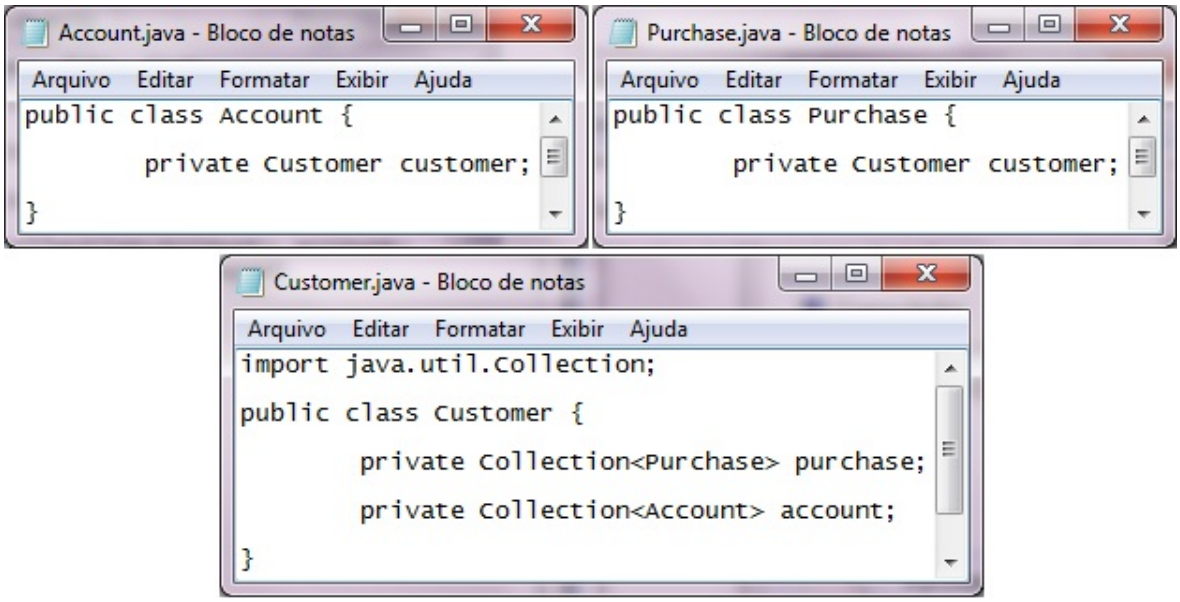

Figura 3.30. Código gerado pela EA a partir do modelo da Figura 3.29.

A Figura 3.31 mostra o modelo elaborado na ferramenta RSA para o estudo de caso 2. Não foi possível inserir as sentenças de propriedade "ordered, unique e unique". A Figura 3.32 mostra o código gerado pela RSA. O relacionamento da classe Customer com a classe Purchase é de 1:N. RSA criou, por defatult, na classe Customer uma coleção do tipo Purchase, utilizando a estrutura Set. Para representar a multiplicidade entre a classe Customer e a classe Account, RSA criou um arranjo com 5 posições . É possível inferir que a RSA utiliza diferentes tipos de estrutura de acordo com a

multiplicidade. Quando os limites inferior e/ou superior estão definidos, a estrutura "[]" é utilizada; quando o não estão definidos, a estrutura Set é utilizada. Porém, não foi oferecido nenhum tipo de opção de mapeamento.

As classes Account e Purchase foram criadas com seus corpos vazios, observando a navegabilidade definida no modelo. Como não foi possível inserir as sentenças de propriedade "unique e ordered, unique", a análise do mapeamento foi prejudicada. A engenharia reversa gerou um modelo com todas as características definidas no modelo inicial.

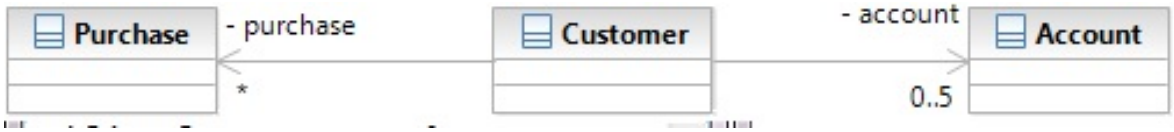

Figura 3.31. Modelo elaborado na ferramenta RSA referente ao estudo de caso 2(ii).

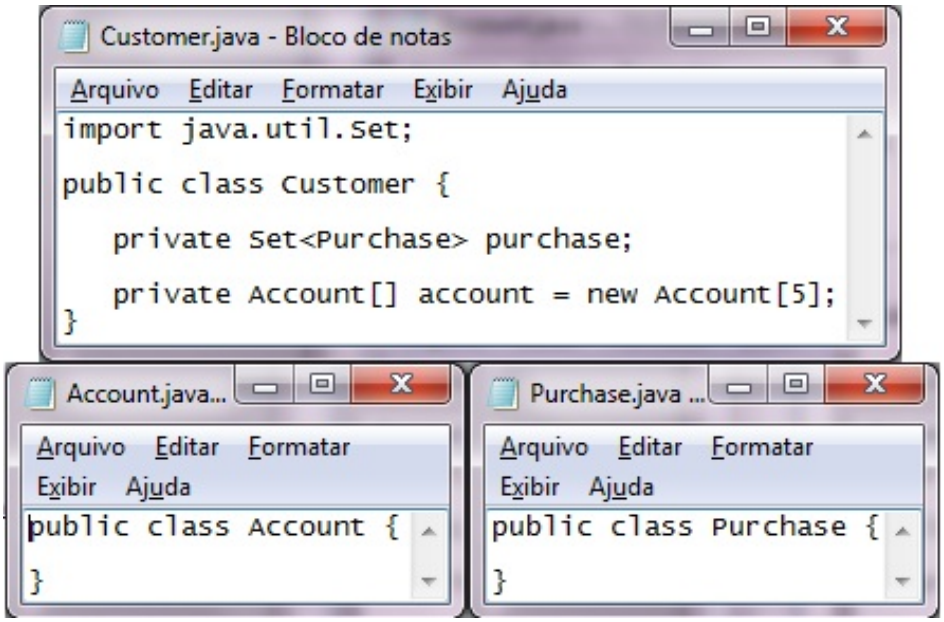

Figura 3.32. Código gerado pela EA a partir do modelo da Figura 3.31.

A Figura 3.33 mostra o modelo elaborado na ferramenta EA para o estudo de caso 2. Não foi possível inserir todas as sentenças de propriedade, encontramos apenas "ordered". A Figura 3.34 mostra o código gerado pela EA a partir do modelo da Figura 3.33. A classe Customer possui atributos que fazem referência à classe Account e à classe Purchase, observando a navegabilidade especificada no modelo. No entanto, a multiplicidade nas extremidades de associações destas classes foi ignorada nos dois casos. Em vez de um atributo simples, deveria ter sido gerada uma coleção.

As extremidades de associação da classe Customer não possuem navegabilidade especificada. A EA gerou um atributo na classe Account para fazer referência à classe Customer, mas não fez o mesmo na classe Purchase. A sentença de propriedade na extremidade de associação da classe Account não foi mapeada.

A engenharia reversa gerou um modelo com os principais elementos presentes no modelo inicial, mas houve informação perdida. Por não terem sido mapeadas na engenharia à frente, as multiplicidades não puderam ser recuperadas. Da mesma forma, a sentença de propriedade "ordered" não foi recuperada. Em virtude do atributo mapeado na classe Account para fazer referência à classe Customer, a navegabilidade foi recuperada como navegável nas duas direções.

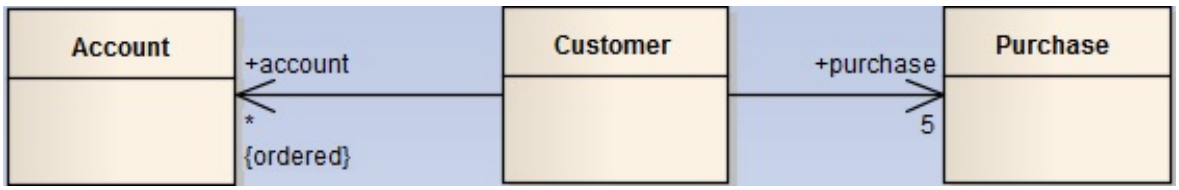

Figura 3.33. Modelo elaborado na ferramenta EA referente ao estudo de caso 2(ii).

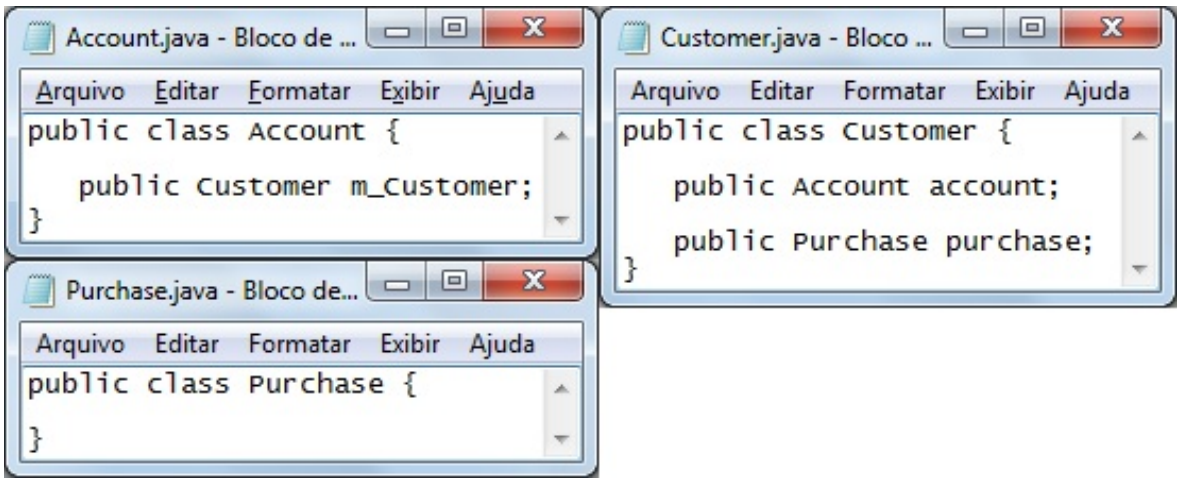

Figura 3.34. Código gerado pela EA a partir do modelo da Figura 3.33.

Considerando os critérios estabelecidos neste capítulo, Astah* teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Parcialmente consistente (1). As classes e os relacionamentos foram recuperados corretamente. No entanto, nas duas situações apresentadas, o mapeamento da multiplicidade não observou o limite superior especificado no modelo.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Resolve parcialmente (1). As multiplicidades observadas nas duas situações permitem mais de um tipo de mapeamento. Astah* optou, por *default*, por representar a coleção com a estrutura Set. A navegabilidade definida como não especificada é mapeada para o código como navegável, embora o usuário não seja consultado se é a opção mais adequada.

C3 – Consistência Interna da Ferramenta: Parcialmente Consistente (1). Todas as classes e relacionamentos foram recuperados pela engenharia reversa. Todavia, as informações conceituais – relacionamento de composição e as sentenças de propriedades - foram perdidas durante o processo de ida e volta. Além disso, os limites superiores especificados no modelo também foram generalizados como ilimitados. A navegabilidade dos relacionamentos do modelo gerado também não correspondeu à do modelo inicial.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). Foi possível definir todos os detalhes do modelo conforme o estudo de caso 2. No entanto, a multiplicidade e a navegabilidade foram mapeadas sem observar detalhes claros do modelo.

Considerando os critérios estabelecidos neste capítulo, RSA teve o seguinte desempenho:

C1 – Confiabilidade do Modelo Gerado: Consistente (2). Todas as classes foram geradas, assim como os relacionamentos, a navegabilidade, a multiplicidade e os nomes nas extremidades de associação. Como não foi possível inserir as sentenças de propriedade “unique e ordered, unique”, parte da análise foi prejudicada.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Resolve parcialmente (1). As multiplicidades observadas nas duas situações permitem mais de um tipo de mapeamento. RSA utilizou duas diferentes estruturas Java para mapear uma coleção com limite superior definido e outra com limite superior ilimitado. No entanto, não foi fornecido nenhum tipo de opção de mapeamento ao usuário.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). O modelo gerado a partir da engenharia reversa recuperou todas as informações presentes no modelo inicial, com exceção do relacionamento de composição, que foi recuperado como um relacionamento de associação simples. Mas, como se trata de informação conceitual e a linguagem Java não define uma estrutura que o diferencie da associação simples, isso não é considerado um erro. A multiplicidade foi recuperada corretamente, assim como os nomes das extremidades de associação.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente Robusta (1). Com exceção do relacionamento de composição, que é informação conceitual, a RSA preservou todas as informações do modelo, tanto no mapeamento de UML para Java quanto

de Java para UML. No entanto, a análise foi prejudicada em parte porque não foi possível inserir as sentenças de propriedade no modelo.

Considerando os critérios estabelecidos neste capítulo, a EA teve o seguinte desempenho:

C1 – Confiabilidade do Modelo Gerado: Parcialmente consistente (1). As classes e os atributos foram gerados conforme as especificações do modelo. No entanto, a multiplicidade na extremidade de associação da classe Slider não foi considerada no mapeamento.

C2 – Capacidade de Resolução de Ambiguidades: Não se aplica (0). A parte do código que permite verificar a capacidade de resolver ambigüidades é a coleção do tipo Slider, que pode ser representado de várias maneiras. A EA sequer gerou essa coleção. Ela ignorou a multiplicidade, o que é diferente de não ser capaz de resolver a ambiguidade.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). O modelo gerado a partir da engenharia reversa recuperou todas as informações presentes no modelo inicial, com exceção da multiplicidade. As informações conceituais - relacionamento de composição e sentença de propriedades - não foram mapeadas para o código. Portanto, também não foram mapeadas de volta para o modelo.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). A multiplicidade, que é um aspecto básico e importante do diagrama de classes, foi ignorada no mapeamento. Não foi possível inserir todas as sentenças de propriedades no modelo.

3.5.3 ESTUDO DE CASO 3 - Classes concretas, Classes Abstratas e Interfaces: *extends* e *implements*

A Figura 3.35, a Figura 3.36 e a Figura 3.37 mostram os modelos elaborados nas ferramentas Astah*, RSA e EA, respectivamente, referente ao estudo de caso 3. Os modelos foram elaborados com facilidade. Os códigos gerados pelas ferramentas foram idênticos. A Figura 3.38 mostra o código.

As classes, as interfaces e os relacionamentos de realização (*realization*) e herança (*extends*) foram mapeados corretamente. O relacionamento de realização é representado em Java pela palavra reservada *implements*. A Figura 38 mostra as classes Endereco e Pessoa implementando as interfaces IEndereco e IPessoa, respectivamente. A classe MGEndereco estende a classe Endereco.

A partir do código gerado, a engenharia reversa recuperou todos os elementos constantes no modelo inicial. Todas as ferramentas obtiveram desempenho similar.

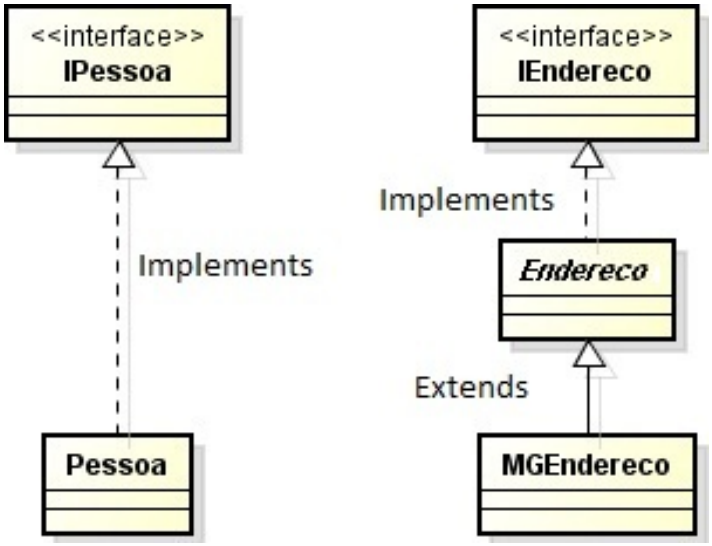

Figura 3.35. Modelo elaborado na ferramenta Astah* referente ao estudo de caso 3.

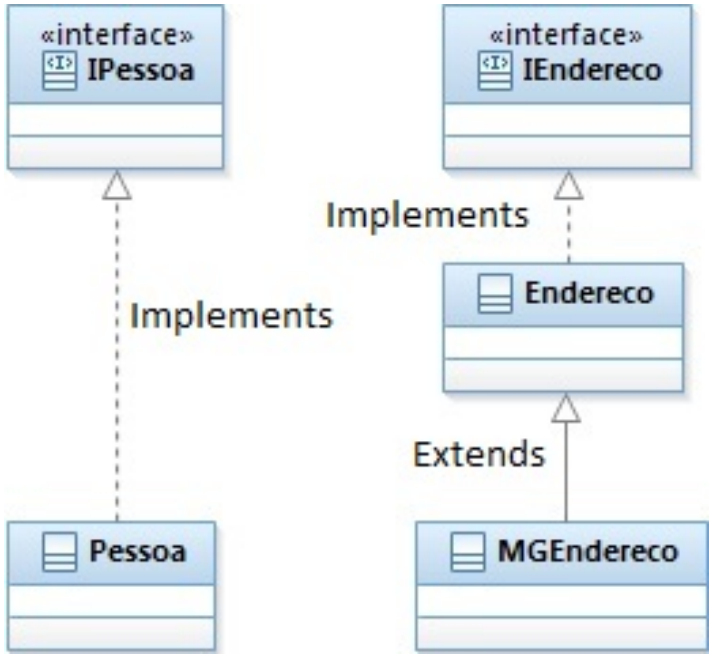

Figura 3.36. Modelo elaborado na ferramenta RSA referente ao estudo de caso 3.

Considerando os critérios estabelecidos neste capítulo, as ferramentas obtiveram o seguinte desempenho:

C1 – Confiabilidade do Modelo Gerado: Consistente (2). Todos os elementos do modelo foram mapeados corretamente para o código: as classes, as interfaces e os relacionamentos de realização e de herança.

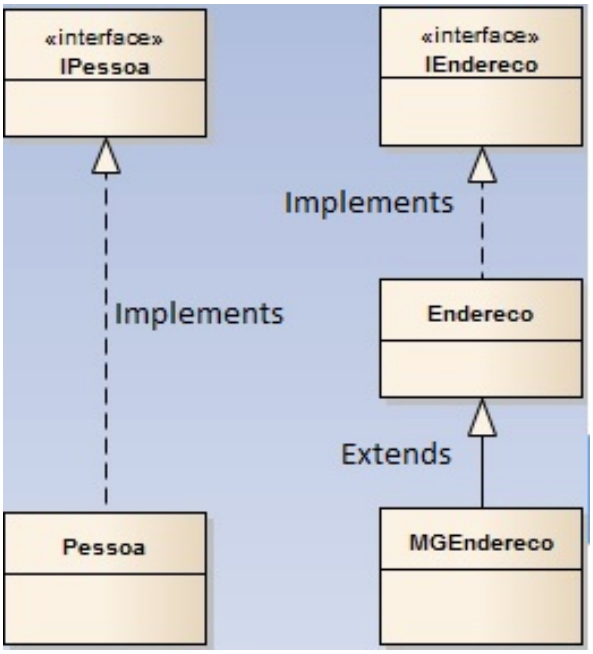

Figura 3.37. Modelo elaborado na ferramenta EA referente ao estudo de caso 3.

C2 – Capacidade de Resolução de Ambiguidades: Não se aplica (0). Os elementos do modelo possuem relação de um para um com os elementos do código, isto é, existe apenas uma forma de mapeamento. Este estudo de caso não explorou a capacidade da ferramenta resolver ambiguidades.

C3 – Consistência Interna da Ferramenta: Consistente (2). O modelo gerado pela engenharia reversa a partir do código mapeado corresponde exatamente ao modelo inicial.

C4 – Flexibilidade e Robustez da Ferramenta: Robusta (2). As ferramentas permitem a inserção do relacionamento *extends* entre classes e *implements* entre classe e interface, conforme a linguagem Java. No entanto, não permitem a inserção do relacionamento realizes entre classes, mas apenas entre classe e interface.

3.5.4 ESTUDO DE CASO 4 - Classes, Interfaces e Anotações

A Figura 3.39 mostra o modelo elaborado na Astah* referente ao estudo de caso 4. A Figura 3.40 mostra o código da classe Window gerado após a engenharia de ida. Além da classe Window, foram criadas, com o corpo vazio, as classes Area, Rectangle, WindowAbstract, WindowInterface e XWindow.

As classes Area, Rectangle e XWindow foram criadas pela Astah* pelo fato de serem declarados atributos desses tipos e eles não estarem pré-definidos. A Figura 3.41

Figura 3.38. Código gerado pelas ferramentas referente ao modelo das Figuras 3.35, 3.36 e 3.37.

mostra a mensagem exibida ao usuário no momento da declaração do atributo size, do tipo Area. É perguntado se o usuário deseja criar esse novo tipo no modelo corrente. A vantagem dessa solicitação é ter a certeza de que só serão declaradas variáveis de tipos pré-definidos. A Figura 3.42 mostra alguns tipos pré-definidos nas ferramentas.

Porém, para o contexto em que se considere a importação de bibliotecas da linguagem Java, a criação de novos tipos não é interessante. Desta forma, Astah* também permite importar bibliotecas de forma que seja possível declarar atributos de outros tipos além dos pré-definidos.

Os tipos dos atributos, métodos, retornos e parâmetros foram mapeados conforme o modelo, além da visibilidade. O atributo colors, que é uma coleção de String, foi mapeado utilizando arranjos - "[ ]" -, observando a multiplicidade definida no modelo. No entanto, Astah* gerou código com erro de sintaxe para o atributo size. A instanciação

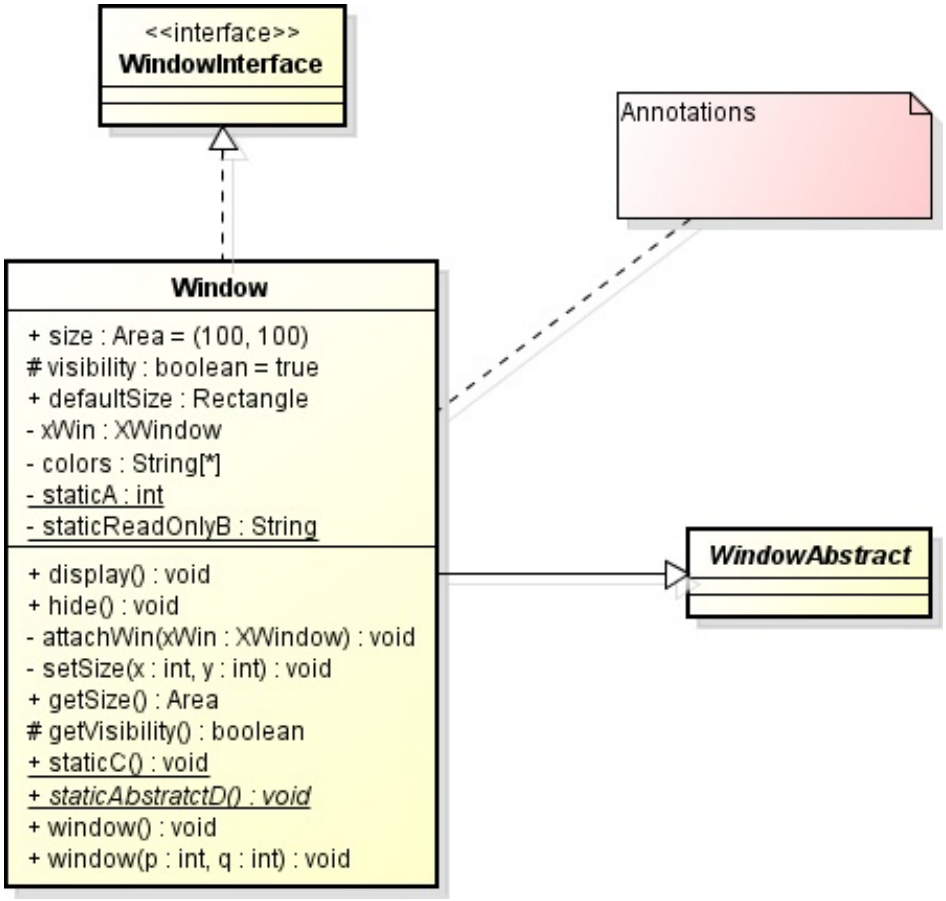

Figura 3.39. Modelo elaborado na ferramenta Astah* referente ao estudo de caso 4.

correta para a declaração seria "public Area area = new Area(100, 100)".

O elemento anotação não foi mapeado para o código. Com exceção da declaração do atributo size e do atributo visibility, o código gerado pela ferramenta foi consistente com o modelo. A interface WindowInterface e a classe abstrata WindowAbstract foram mapeadas corretamente. Para manter a consistência, Astah* criou as classes referentes aos tipos não pré-definidos: Area, Rectangle e XWindow.

Após obter o código gerado, foi feito o processo contrário. Como havia erro de sintaxe no código da classe Window, Astah* exibiu uma mensagem de erro, conforme a Figura 3.43. O modelo não foi gerado de volta, caracterizando inconsistência interna da ferramenta, uma vez que produziu um artefato e não foi capaz de interpretá-lo.

Considerando os critérios estabelecidos neste capítulo, Astah* teve o seguinte desempenho:

C1 - Confiabilidade do Código Gerado: Código parcialmente consistente (1). Com exceção do erro de sintaxe da declaração dos atributos size e visibility, o código gerado

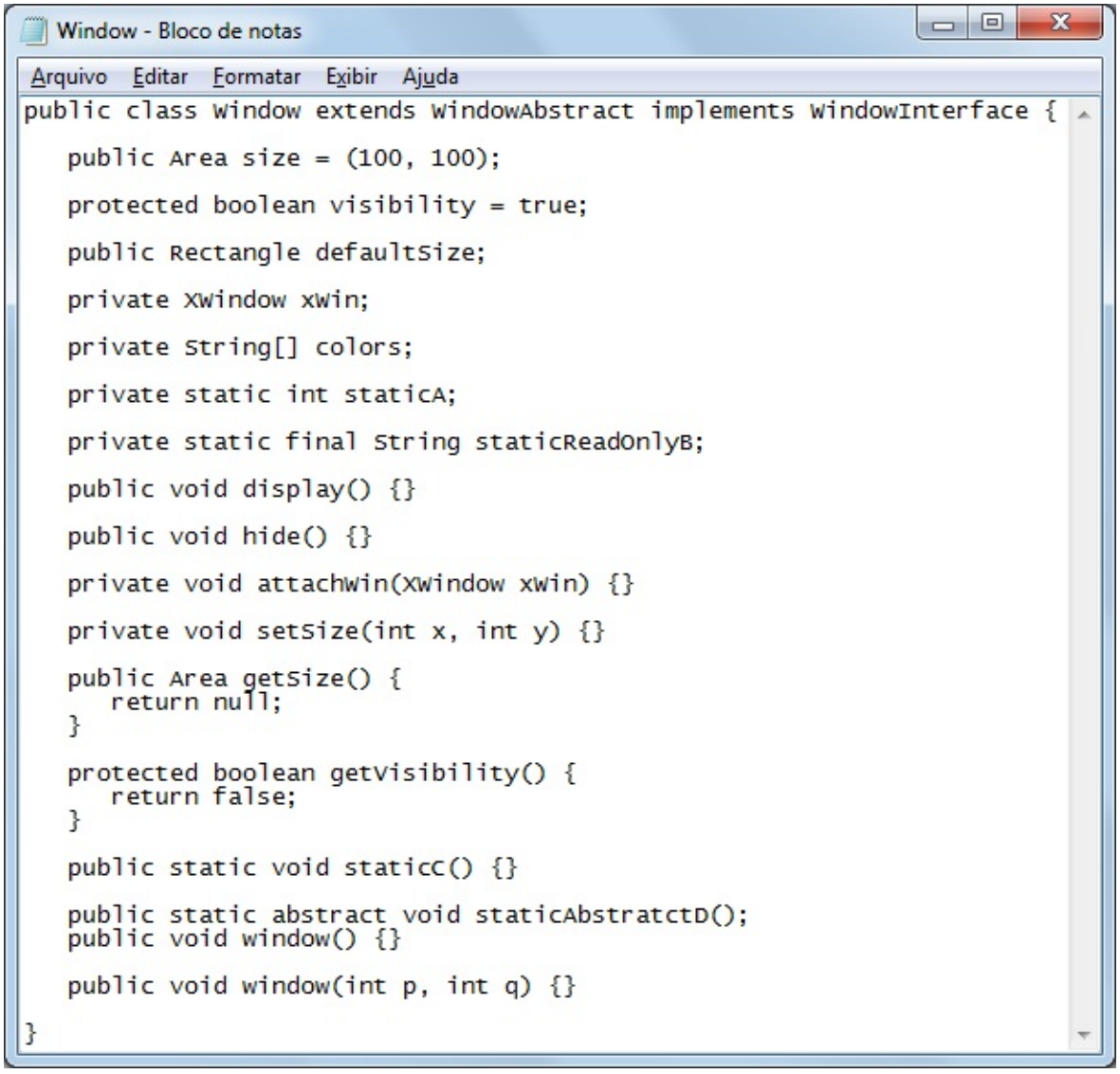

Figura 3.40. Classe Window gerada pela ferramenta Astah*.

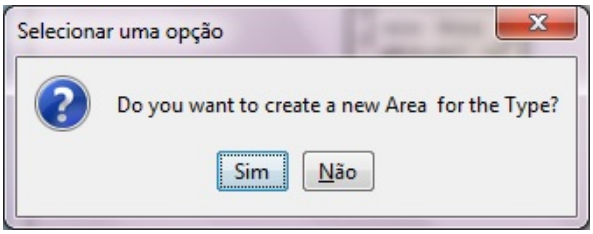

Figura 3.41. Mensagem enviada ao usuário perguntando sobre a criação do novo tipo Area.

é consistente com o modelo especificado.

C2 - Capacidade de Resolução de Ambiguidades de Modelo: Resolve parcialmente

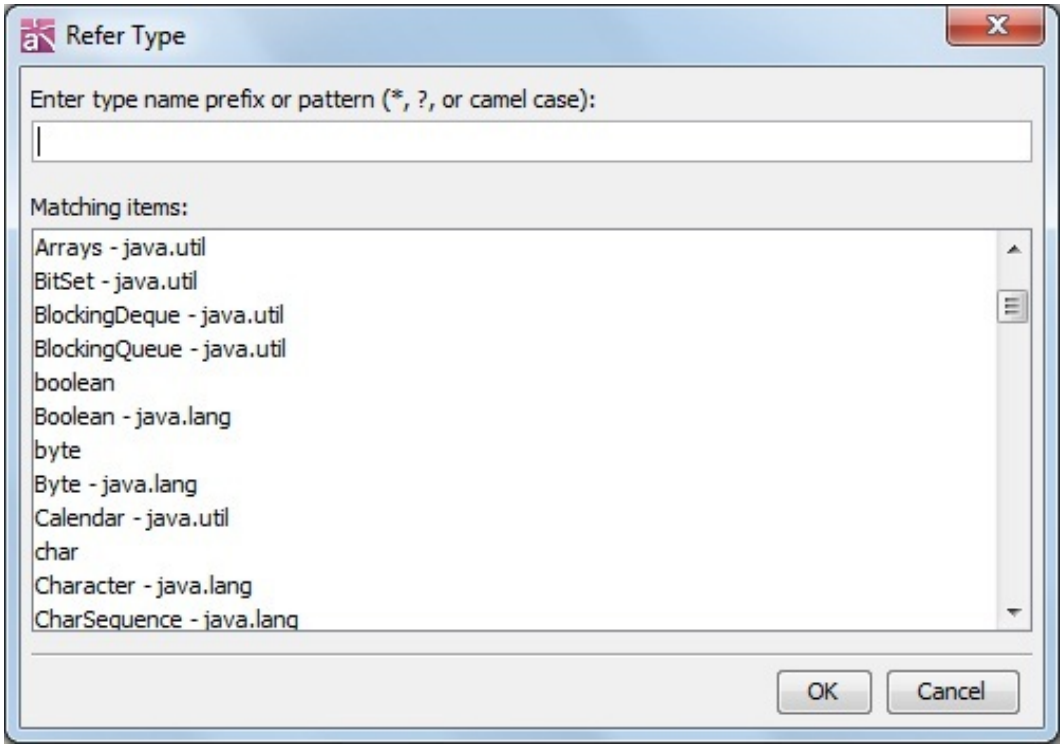

Figura 3.42. Tipos pré-definidos pela ferramenta Astah*.

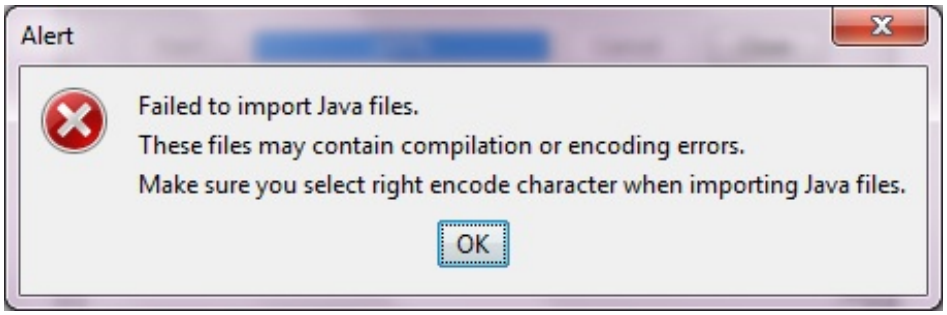

Figura 3.43. Erro na engenharia reversa do estudo de caso 4.

(1). O atributo colors permite mais de um tipo de mapeamento. A coleção poderia ter sido representada por Set, List ou Vector, por exemplo. Não houve interação com o usuário, mas pode-se verificar que a opção *default* da ferramenta para representar uma coleção é utilizar arranjo. Existe uma área de configuração em que pode ser alterada a opção *default*.

C3 - Consistência Interna da Ferramenta: Inconsistente (0). Ocorreu erro no processo de engenharia reversa e não foi possível gerar o modelo a partir do código. Observa-se, neste caso, que a ferramenta gerou código e foi incapaz de mapear de volta para o modelo este código que ela mesma gerou. Com a correção manual da sintaxe da

inicialização do atributo size, Astah* foi capaz de gerar o modelo, que correspondeu ao modelo inicial.

C4 - Flexibilidade e Robustez da Ferramenta: Parcialmente Robusta (1). Para declarar um atributo do tipo Area, foi necessário criar a respectiva classe no modelo. Um ponto positivo da ferramenta foi restringir a declaração de atributos a tipos pré-definidos ou tipos existentes nas bibliotecas importadas. Um ponto negativo foi a ferramenta ter gerado código com erro de sintaxe.

A Figura 3.44 mostra o modelo elaborado na RSA conforme estabelecido no estudo de caso 4. Alguns detalhes, como parâmetros e tipos dos parâmetros, são omitidos no modelo para sintetizar o que é exibido, mas são propriamente definidos nas propriedades. Por exemplo, apesar de existirem dois métodos window(), o primeiro não possui parâmetros e o segundo possui dois parâmetros A Figura 3.45 mostra o código gerado pela ferramenta.

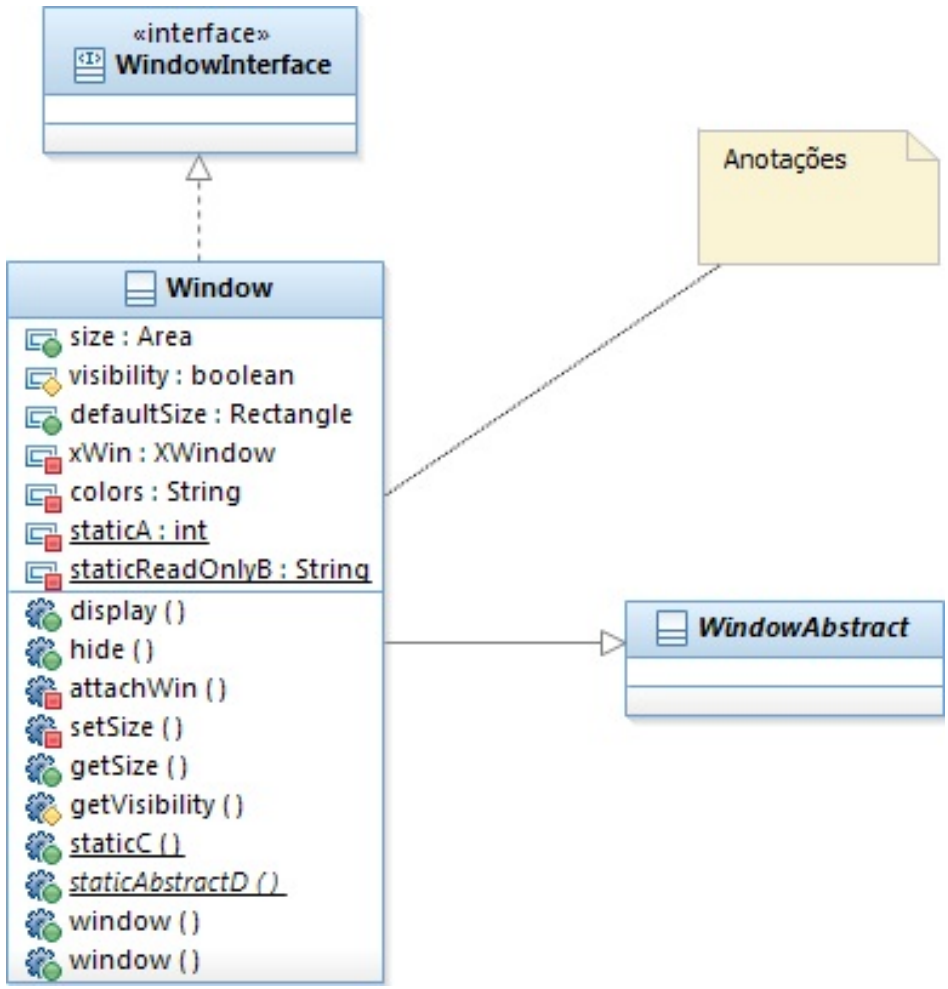

Figura 3.44. Modelo elaborado na ferramenta RSA referente ao estudo de caso 4.

O código gerado é parcialmente consistente com o modelo. Os tipos dos atributos, dos métodos, dos parâmetros e dos métodos foram preservados, assim como a visibilidade. A interface WindowInterface e a classe abstrata WindowAbstract foram mapeadas corretamente.

Os aspectos "somente de leitura" e "abstrato" são representados na linguagem Java, respectivamente, com as palavras reservadas final e abstract. O atributo staticReadOnlyB, estático e somente de leitura, não teve este aspecto mapeado para o código. De modo análogo, o método staticAbstract, estático e abstrato, não teve este aspecto mapeado para o código. O elemento anotação não foi mapeado para o código. O atributo size foi mapeado para o código sem inicialização. Por outro lado, o atributo visibility foi inicializado corretamente.

Não é permitido declarar atributo de tipo não existente, como foi o caso do tipo XWindow. Nenhuma informação foi exibida ao usuário, apenas não foi permitida a declaração deste tipo. Então, criou-se a classe do tipo XWindow no próprio modelo. A partir de então, pode-se declarar atributo deste novo tipo. É um ponto positivo porque garante que, para que sejam utilizados, todos os, de fato, tipos existam.

O atributo colors foi mapeado utilizando a estrutura Set da biblioteca java.util da linguagem Java para representar a coleção. A coleção poderia ser representada de outras formas. No entanto, RSA não ofereceu opções de mapeamento. A RSA tem a opção de habilitar ou importar as bibliotecas da linguagem Java. Em uma área de configuração, como é mostrado na Figura 3.46, é possível habilitar a consulta de tipos de classes em bibliotecas referenciadas. Assim, podem-se instanciar objetos dos tipos existentes na biblioteca.

A RSA gera o código de um modelo de forma indireta, isto é, através de uma transformação. Nessa transformação, é possível habilitar ou desabilitar algumas funções. Desse modo, o usuário pode interagir com a ferramenta para influenciar no código gerado. A Figura 3.47 mostra a transformação criada na RSA para gerar o código referente ao modelo estabelecido no estudo de caso 4.

Considerando os critérios estabelecidos neste capítulo, RSA teve o seguinte desempenho:

C1 - Confiabilidade do Código Gerado: Parcialmente consistente (1). O aspecto "somente de leitura" do atributo staticReadOnlyB e aspecto "abstrato" do método "staticAbstractD" não foram mapeados. Além disso, o atributo size não foi inicializado com os valores especificados no modelo.

C2 - Capacidade de Resolução de Ambiguidades de Modelo: Resolve parcialmente (1). A coleção do atributo colors permite mais de um tipo de mapeamento. A RSA utiliza a estrutura Set, por *default*, para representar a multiplicidade com limite

Window - Bloco de notas

Arquivo Editar Formatar Exibir Ajuda

```
import java.util.Set;

public class Window extends WindowAbstract implements WindowInterface {

   public Area size;

   protected boolean visibility = new Boolean(true);

   public Rectangle defaultSize;

   private XWindow xWin;

   private Set<String> colors;

   private static int staticA;

   private static String staticReadOnlyB;

   public void display() {}

   public void hide() {}

   private void attachWin(XWindow xWin) {}

   private void setSize(int x, int y) {}

   public Area getSize() {
      return null;
   }

   protected boolean getVisibility() {
      return false;
   }

   public static void staticC() {}

   public static void staticAbstractD() {}

   public void window() {}

   public void window(int x, int y) {}
}
```

Figura 3.45. Código gerado pela RSA a partir do modelo da Figura 3.44.

superior indefinido. RSA não interagiu com o usuário.

C3 - Consistência Interna da Ferramenta: Parcialmente Consistente (1). Após o mapeamento de Java para UML, todos os atributos foram recuperados, assim como seus tipos e suas visibilidades. No entanto, o aspecto "somente de leitura" do atributo staticReadOnlyB e o aspecto "abstrato" do método "staticAbstractD" foram informações perdidas após a ida e a volta.

C4 - Flexibilidade e Robustez da Ferramenta: Parcialmente Robusta (1). A RSA permite criar classes manualmente e também habilitar a utilização de classes de bibliotecas específicas. Todavia, o par ordenado (100, 100) definido como valor inicial

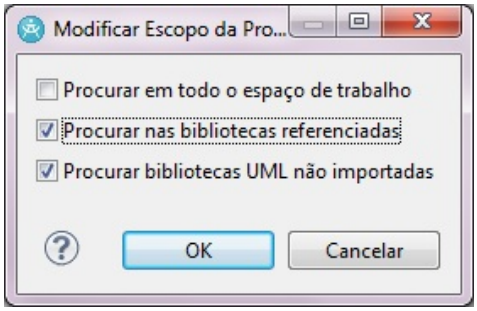

Figura 3.46. Área de configuração que habilita a utilização de classes de bibliotecas específicas.

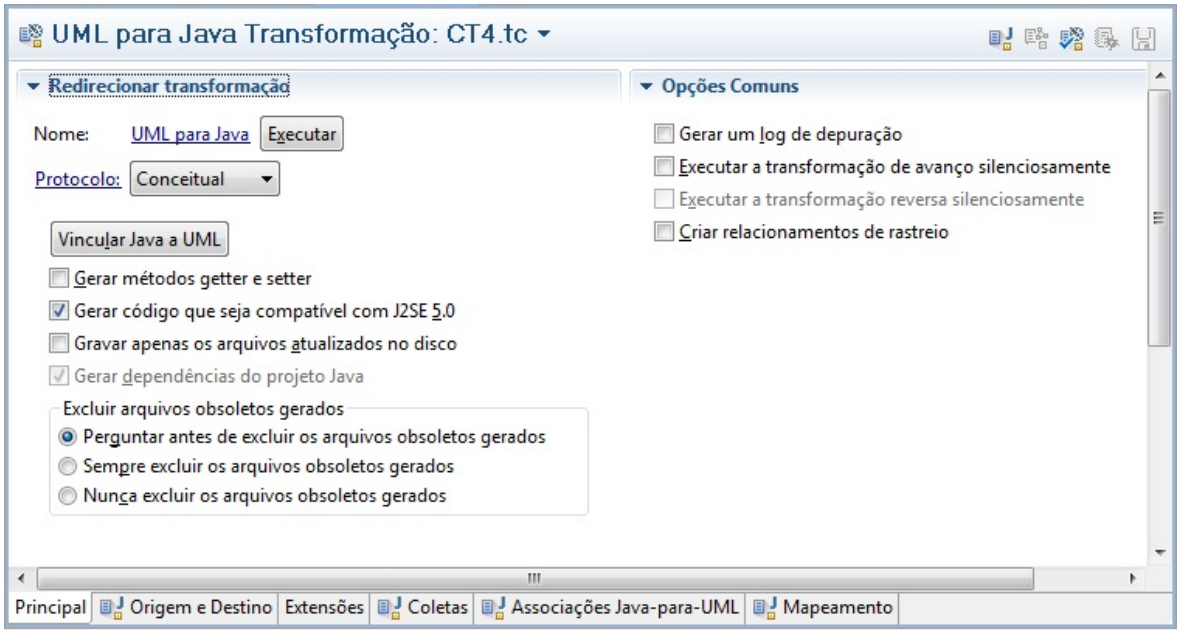

Figura 3.47. Transformação criada na RSA para gerar o código do modelo do estudo de caso 4.

para o atributo size não foi mapeado.

A Figura 3.48 mostra o modelo elaborado na Enterprise Architect. Os atributos e os métodos são ordenados a cada inserção, por isso a disposição deles não é a mesma do modelo do estudo de caso. A ferramenta esconde os nomes dos parâmetros, deixando explícitos apenas os tipos, como pode ser visto nos métodos attacth e setSize, mas não interfere no mapeamento. O código gerado pela EA é mostrado na Figura 3.49.

O código gerado é parcialmente consistente com o modelo. RSA gerou código com erro de sintaxe para o atributo do tipo Area. A instanciação correta para a declaração seria "public Area area = new Area(100, 100)". Outra falha foi ignorar a multiplicidade do atributo colors, do tipo String. O modelo mostra uma coleção do tipo String e o

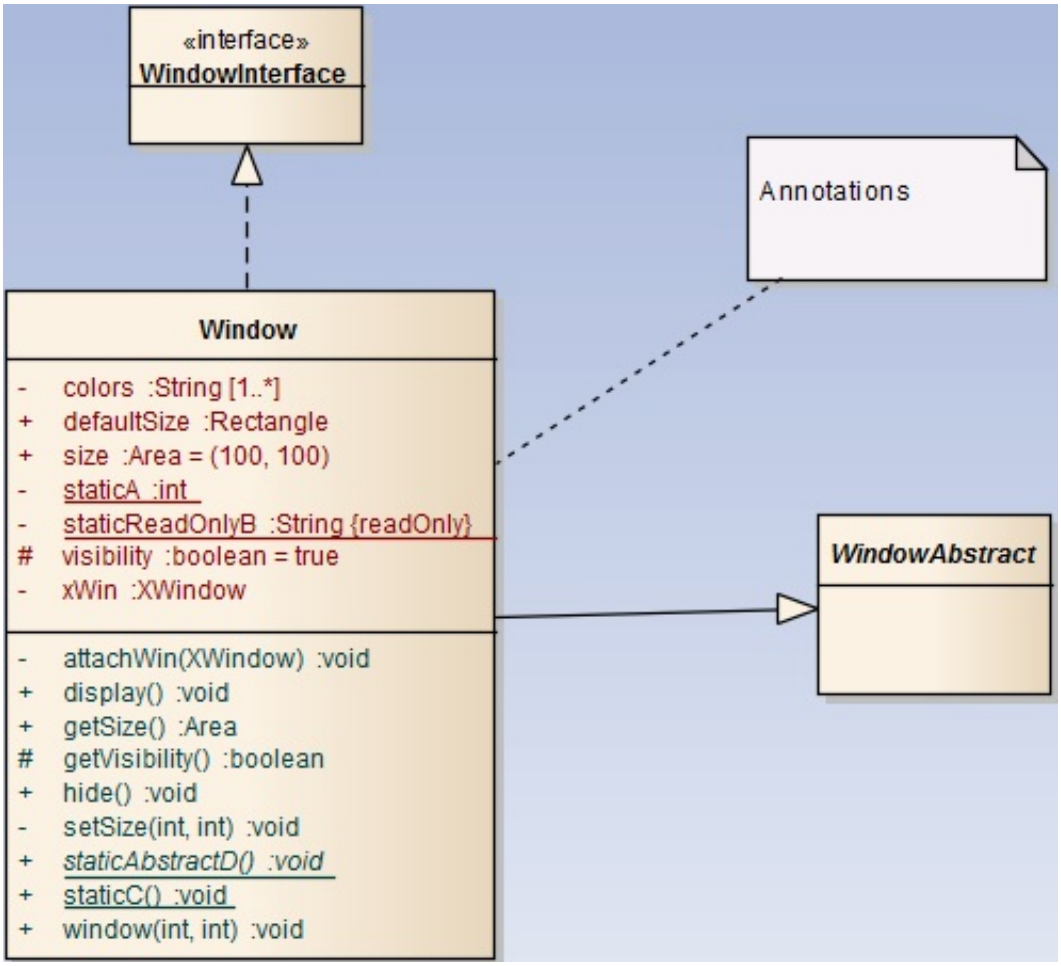

Figura 3.48. Modelo elaborado na ferramenta EA referente ao estudo de caso 4.

código mostra um atributo simples. No entanto, os outros atributos e métodos foram mapeados corretamente, assim como a interface WindowInterface e a classe abstrata WindowAbstract.

A EA possui tipos pré-definidos e permite também importar bibliotecas. A Figura 3.50 mostra a interface que permite ao usuário inserir a biblioteca. A ferramenta não impede que seja declarado atributo ou método de um tipo que não está pré-definido ou não pertence a nenhuma biblioteca. Desta forma, EA permite utilizar tipos não existentes no modelo corrente, levando à inconsistência.

A Figura 3.51 mostra a interface exibida pela EA para a geração de código. São mapeadas apenas a classe Window, a classe abstrata WindowAbstract e a interface WindowInterface. Como EA não impede a declaração de tipos além dos pré-definidos e das bibliotecas, foram declarados atributos dos tipos Area, Rectangle e XWindow, mas não foi solicitada a criação de elementos de classes, no modelo, referentes a esses

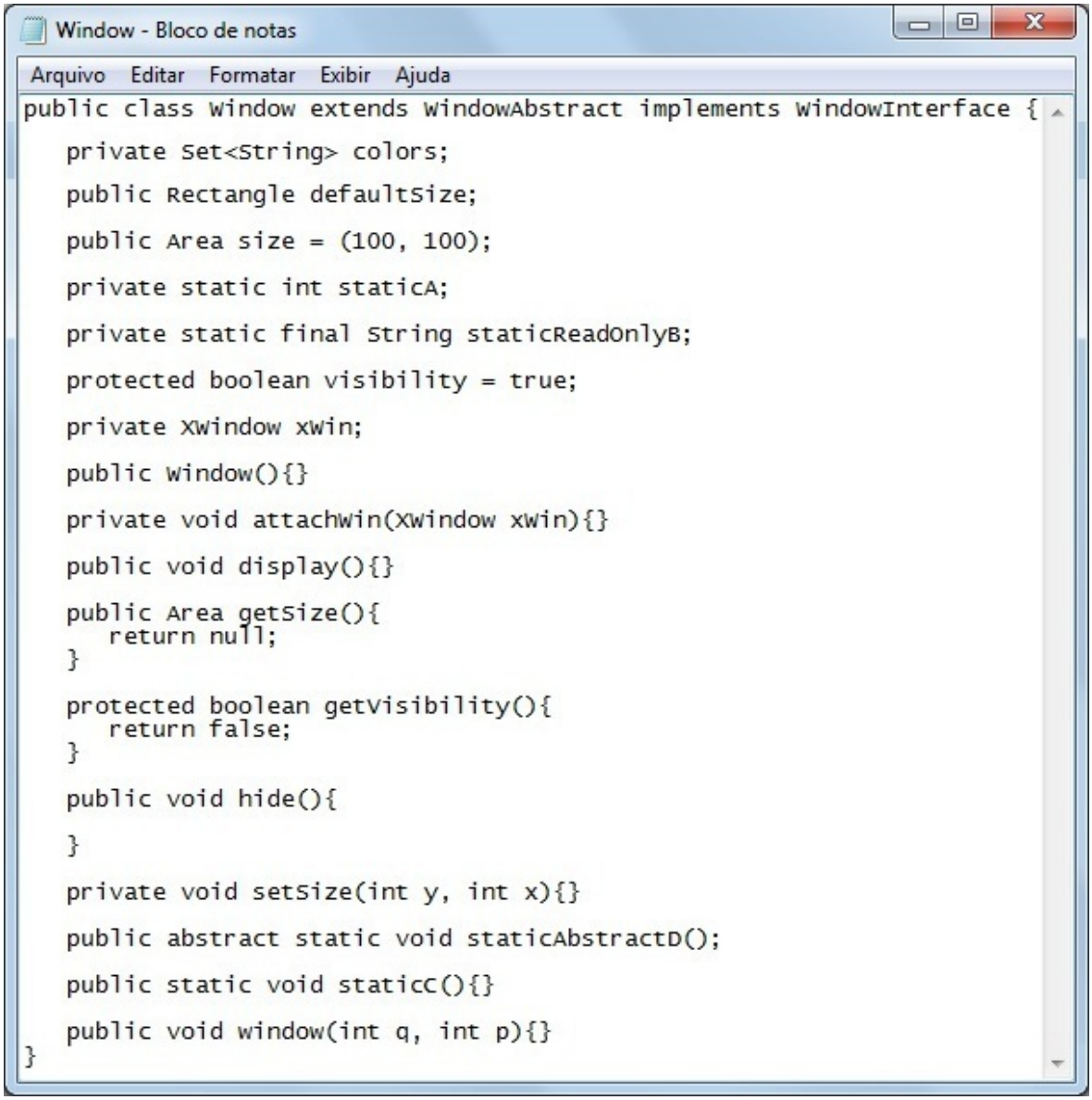

Figura 3.49. Código gerado pela EA a partir do modelo da Figura 3.48.

tipos. Desta forma, os atributos são de tipos que não existem no contexto do modelo.

A Figura 3.52 mostra uma mensagem exibida durante o mapeamento do modelo para o código. Apesar de verificar e exibir a mensagem de erro, é gerado o código com erro referente à inicialização do atributo size. Em seguida, após o mapeamento contrário, do código para modelo, o modelo é gerado conforme o modelo inicial, embora seja exibida uma mensagem de erro, referente à inicialização do atributo size.

Considerando os critérios estabelecidos neste capítulo, EA teve o seguinte desempenho:

C1 - Confiabilidade do Código Gerado: Parcialmente consistente (1). Os atri-

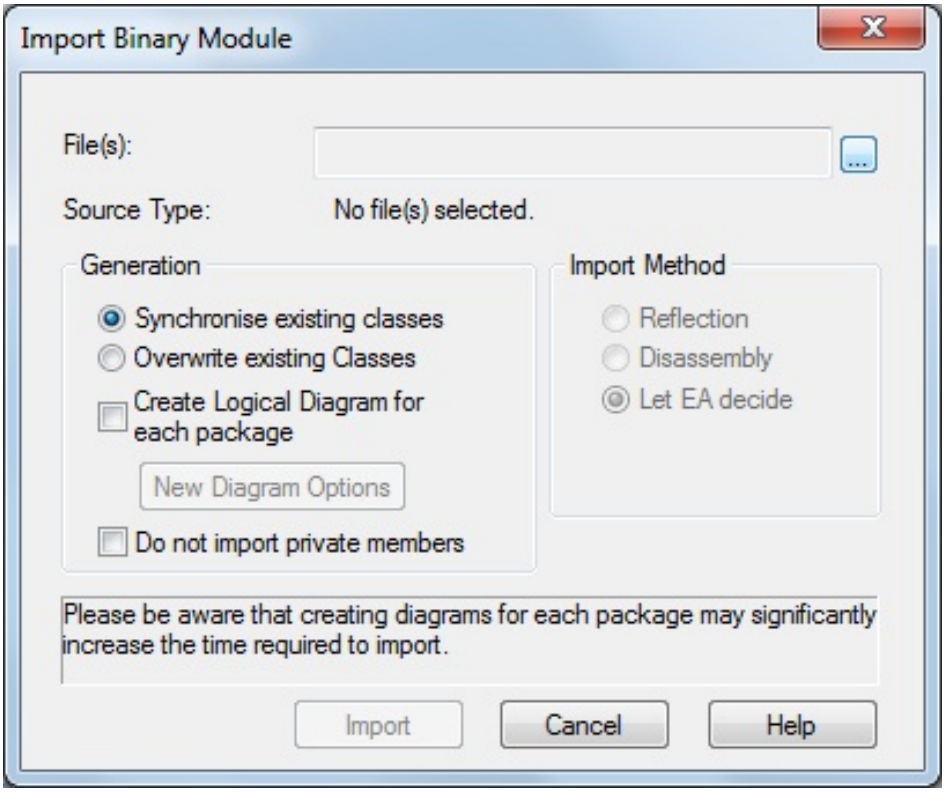

Figura 3.50. Interface para importar bibliotecas.

butos, métodos, parâmetros e retornos foram gerados corretamente, respeitando seus tipos e visibilidades. No entanto, houve erro de sintaxe na inicialização do atributo size e a multiplicidade do atributo colors foi ignorada.

C2 - Capacidade de Resolução de Ambiguidades de Modelo: Não se aplica (0). A parte do código que permite verificar a capacidade de resolver ambiguidades é a coleção do atributo colors, que pode ser representado de várias maneiras. EA sequer gerou essa coleção. Ela ignorou a multiplicidade, o que é diferente de não ser capaz de resolver a ambiguidade.

C3 - Consistência Interna da Ferramenta: Parcialmente consistente (1). Apesar de gerar código com erro de sintaxe, EA conseguiu gerar um modelo similar ao modelo inicial fazendo o mapeamento contrário. Todos os atributos, métodos e parâmetros, assim como seus tipos e visibilidades, foram recuperados. O atributo size foi representado no modelo da mesma forma que no modelo inicial. Todavia, o atributo colors não foi mapeado como uma coleção de String e sim como um atributo simples.

C4 - Flexibilidade e Robustez da Ferramenta: Parcialmente Robusta (1). A EA permite criar classes manualmente e também utilizar classes de bibliotecas importadas. Todavia, permite a declaração de tipos não existentes no contexto do modelo; gera

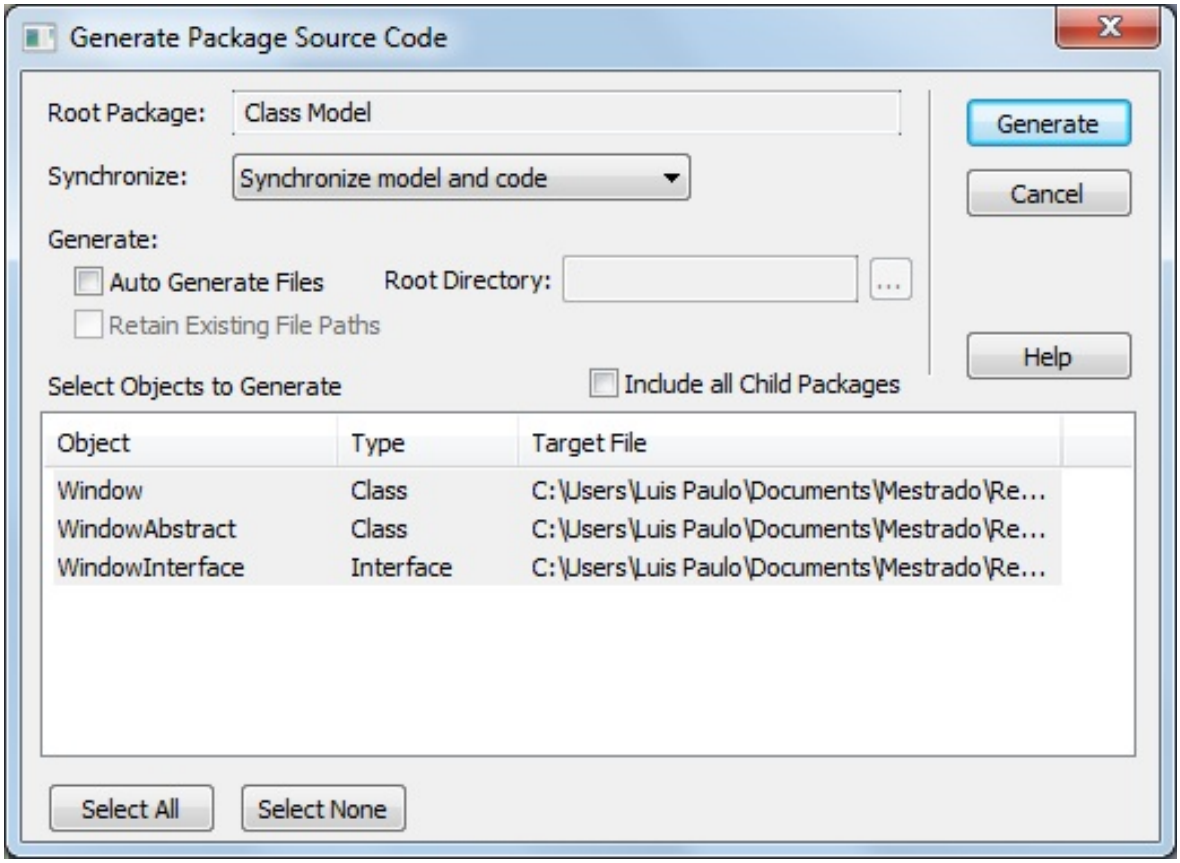

Figura 3.51. Interface para importar bibliotecas.

código com erro de sintaxe, como aconteceu com a inicialização do atributo size; e desconsidera a multiplicidade no mapeamento, como aconteceu com o atributo colors.

3.5.5 ESTUDO DE CASO 5 - Estrutura Composta

3.5.5.1 Estrutura Composta I

A Figura 3.53 mostra o modelo elaborado na ferramenta Astah* para o estudo de caso 5 (i). Astah* fornece a opção de criar o diagrama de estrutura composta a partir do diagrama de classe. Após criar o diagrama de estrutura composta desta forma, a classe Car foi arrastada para dentro do diagrama de estrutura composta. A ferramenta exibiu a mensagem mostrada na Figura 3.54, perguntando ao usuário se Car seria uma classe estruturada ou uma classe. Após confirmar, o classificador estruturado Car foi criado com apenas a parte *rear*, do tipo Wheel, como pode ser visto na Figura 3.55, indicando que a instância de Wheel está contida na classe Car. Em seguida, a parte *e*, do tipo Engine, e o conector foram inseridos no classificador estruturado Car.

Após criar o diagrama de estrutura composta a partir do diagrama de classe,

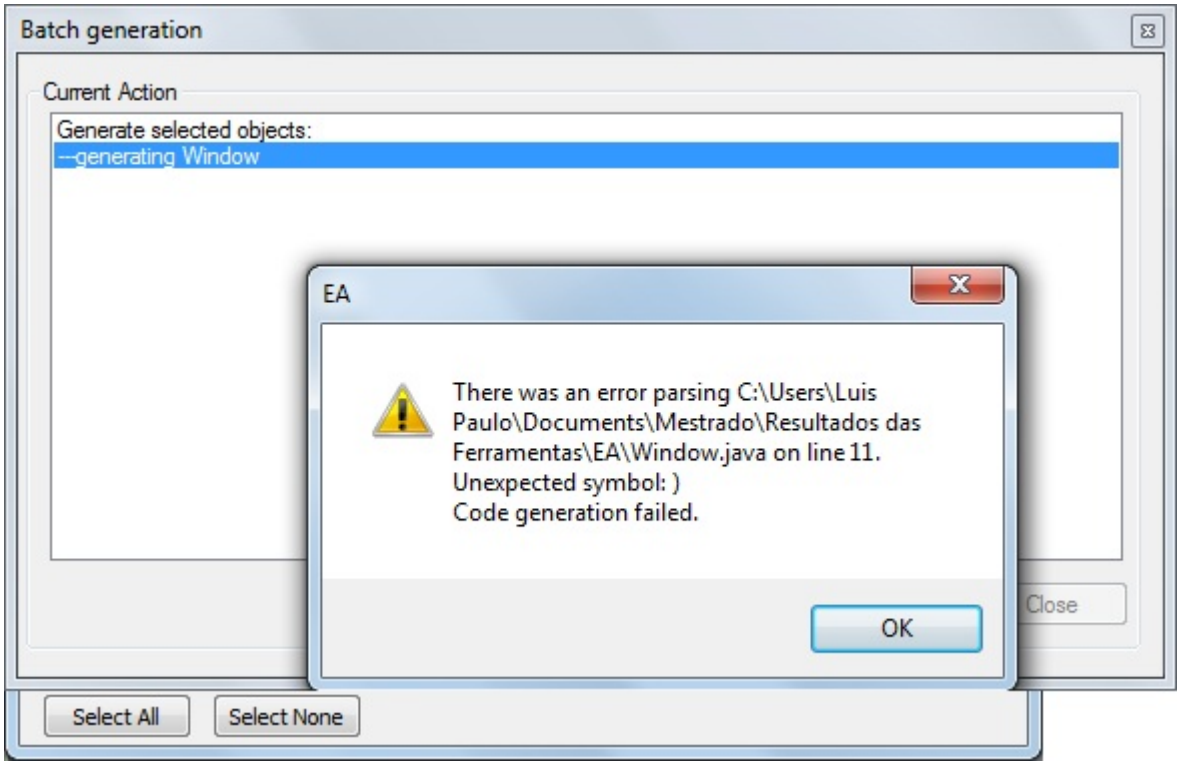

Figura 3.52. Erro ocorrido na geração de código.

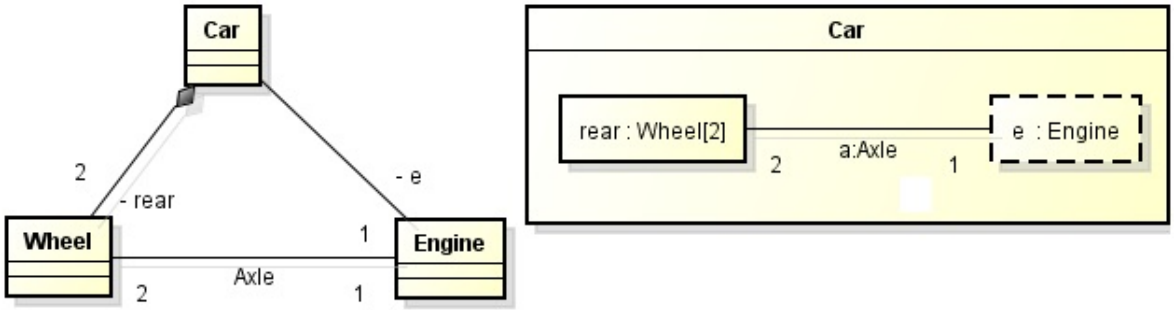

Figura 3.53. Modelo elaborado na ferramenta Astah* para o estudo de caso 5(i).

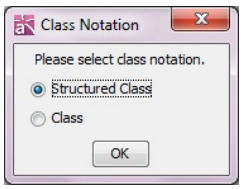

Figura 3.54. Mensagem exibida ao usuário solicitando uma escolha.

foram feitos ajustes para que ele ficasse conforme o diagrama de estrutura composta do estudo de caso. Em virtude disso, o diagrama de classe foi levemente alterado

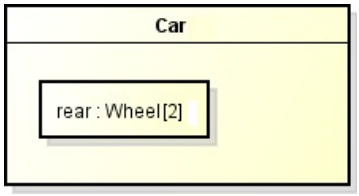

Figura 3.55. Classificador estruturado Car criado a partir do diagrama de classes.

automaticamente pela Astah*. Como o objetivo desse estudo de caso é verificar a geração do código a partir do diagrama de estrutura composta, optou-se por deixá-lo coerente com o modelo do estudo de caso, embora as multiplicidades da associação entre a classe Wheel e Engine tenha sido alterada.

Astah* não gera o código a partir de um diagrama, mas sim a partir das classes do modelo. Desta forma, uma vez que o diagrama de estrutura composta é composto pelo classificador estruturado Car e pelas partes Wheel e Engine, o mapeamento produziu o código das classes Car, Wheel e Engine, como pode ser visto na Figura 3.56.

As multiplicidades foram consideradas pela Astah* no mapeamento. No entanto, não foi observada a especificação do limite superior. A estrutura Collection foi utilizada, por *default*, para representar a coleção. Não foi fornecida nenhuma opção de mapeamento para o usuário, embora a multiplicidade pudesse ter sido mapeada utilizando outra estrutura.

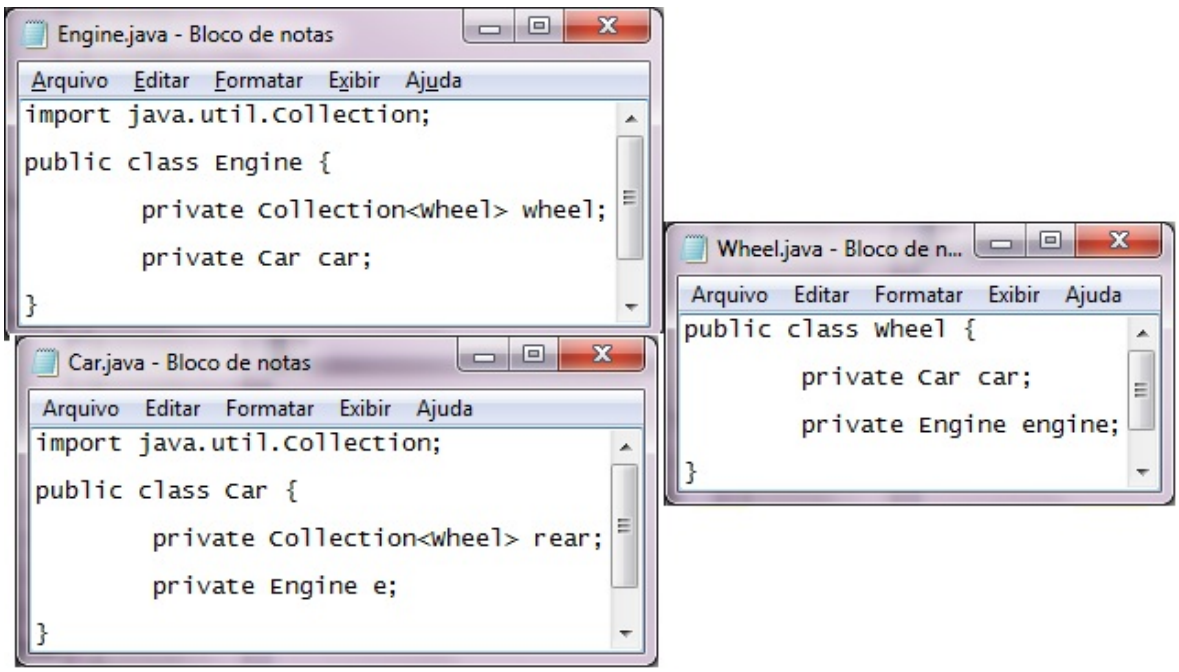

Figura 3.56. Código das classes Car, Engine e Wheel.

A Figura 3.57 mostra o modelo elaborado na RSA para o estudo de caso 5 (i). O diagrama de estrutura composta, à direita, foi criado a partir do diagrama de classe, à esquerda. Após clicar com o botão direito sobre a classe Car, foi selecionada a opção de incluir um diagrama de estrutura composta, como pode ser visto na Figura 3.58.

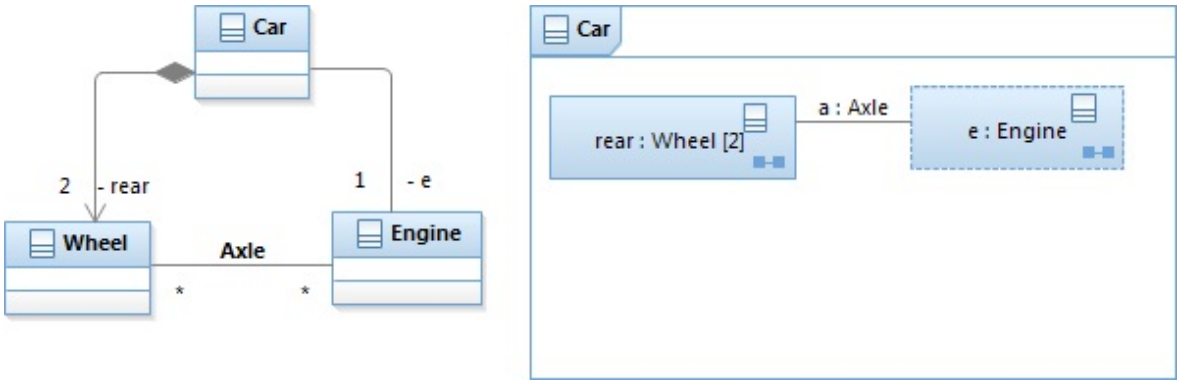

Figura 3.57. Modelo elaborado na ferramenta RSA para o estudo de caso 5(i).

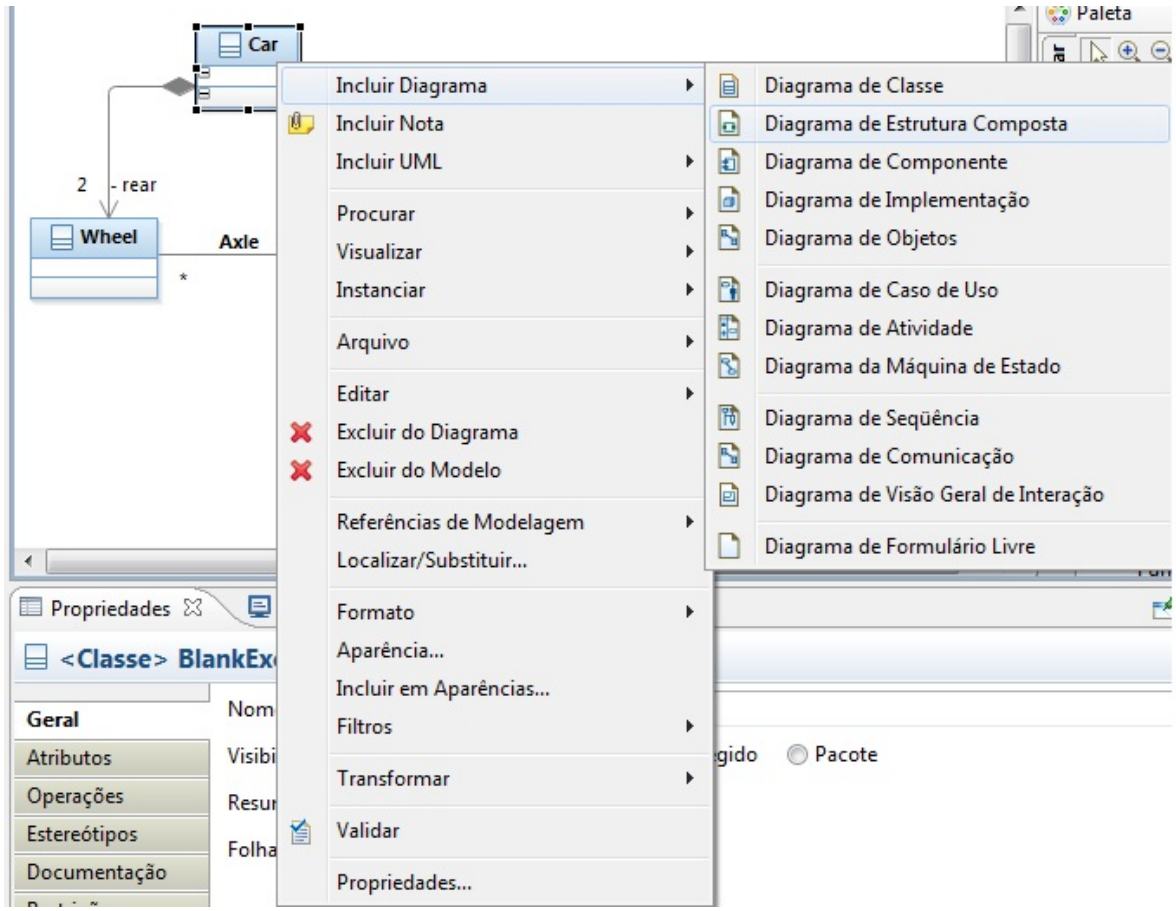

Figura 3.58. Criação do diagrama de estrutura composta a partir da classe Car.

Após a geração do diagrama de estrutura composta a partir do diagrama de classe, apenas o conector "a: Axle" não foi mapeado, tendo sido posteriormente inserido manualmente. Após inseri-lo, não foi possível especificar a multiplicidade 2:1 da classe Wheel com a classe Engine no diagrama de estrutura composta por meio do conector "a: Axle". Visualmente, isso não compromete o modelo, uma vez que o arranjo com limite superior 2, na parte rear do tipo Wheel, representa duas instâncias desse tipo relacionadas com uma instância do tipo Engine.

Após a engenharia à frente, as multiplicidades nas extremidades de associação de Wheel e Engine foram mapeadas sem especificar limite superior, conforme o diagrama de classes. Isso permite verificar que o diagrama de estrutura composta representa uma instância mais detalhada dos objetos definidos no diagrama de classe. O código é gerado a partir das classes do modelo.

A Figura 3.59 mostra o código gerado. Pode-se observar que a classe Car reflete o que está definido no diagrama de estrutura composta e no diagrama de classes. RSA mapeou a multiplicidade especificada no diagrama de classe, gerando uma coleção utilizando a estrutura Set. Na classe Car, foram gerados um atributo do tipo Engine e um arranjo com duas posições do tipo Wheel. Apesar de mapear multiplicidades para o código com estruturas diferentes, RSA não interagiu com o usuário fornecendo as opções de mapeamento.

Figura 3.59. Código gerado pela RSA a partir do modelo da Figura 3.57.

A Figura 3.60 mostra um diagrama de estrutura composta criado diretamente, sem a utilização do diagrama de classes. Inicialmente, foi criado o classificador estruturado Car2. Em seguida, inseriu-se uma parte. Foi exibida a mensagem mostrada na Figura 3.61, solicitando ao usuário o que ele deseja fazer. Foi escolhida a opção "Criar Classe", criou-se a classe Wheel2 e criou-se a parte rear, do tipo Wheel2. O mesmo foi feito para a parte e, do tipo Engine2. Após ajustar os outros detalhes, o código gerado foi similar ao código gerado a partir do modelo da Figura 3.57. Não foi identificado nenhum tipo de relacionamento entre as classes Wheel2 e Engine2. É possível dizer que, na RSA, os relacionamentos entre as partes de um classificador estruturado só existem se eles existirem no diagrama de classe.

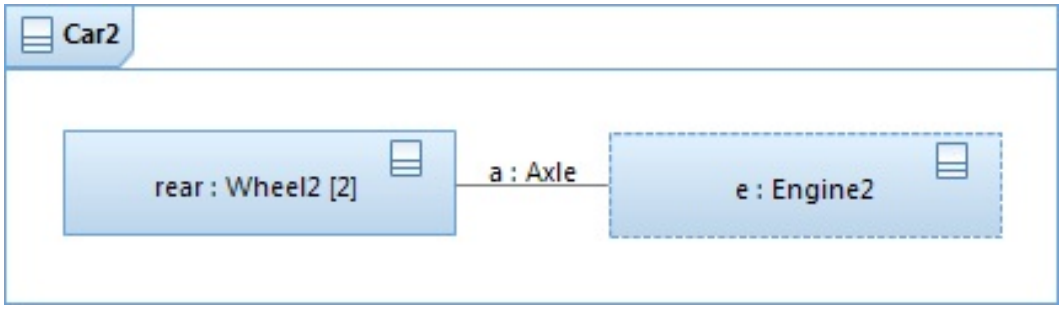

Figura 3.60. Diagrama de estrutura composta criado diretamente.

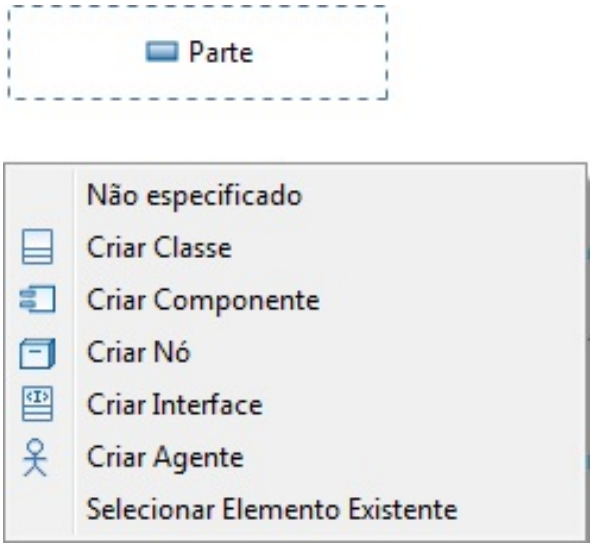

Figura 3.61. Mensagem exibida ao usuário solicitando o que usuário deseja fazer.

A Figura 3.62 mostra o diagrama de estrutura composta elaborado na ferramenta EA para o estudo de caso 5 (i). Inicialmente, foi criado o classificador estrutura Car e

sua respectiva classe no modelo. Em seguida, ao inserir a parte rear, a EA não solicitou a criação de uma classe ou a especificação do tipo de rear. O mesmo aconteceu com a parte e. Isso gerou inconsistência no modelo, uma vez que as partes rear e e deveriam se referir a classes no diagrama de classe. Desta forma, após a engenharia à frente, não foram mapeadas. O mapeamento do diagrama de estrutura composta para o modelo produziu apenas a classe Car, sem corpo, como pode ser visto na Figura 3.63.

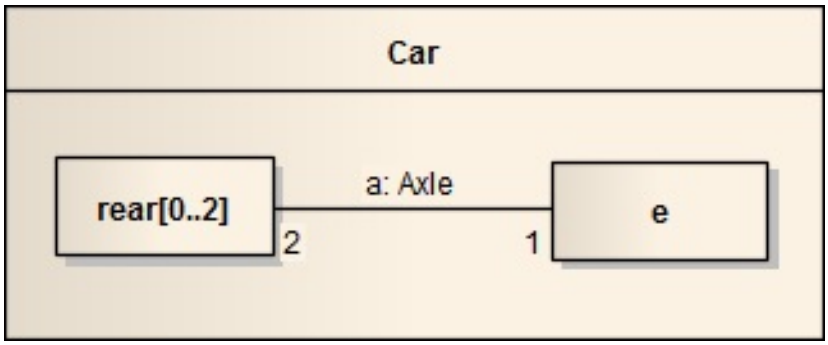

Figura 3.62. Modelo elaborado na ferramenta EA referente ao estudo de caso 5(i).

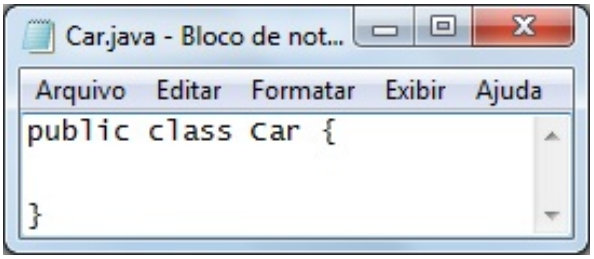

Figura 3.63. Código gerado pela EA a partir do modelo da Figura 3.62.

EA não gerou automaticamente as classes nem solicitou a criação das mesmas, permitindo a inconsistência no modelo. Para contornar o problema, as classes foram inseridas no modelo manualmente. A Figura 3.64 mostra o código gerado pela EA a partir do novo modelo. Foram geradas as três classes. A classe Car contém o atributo rear do tipo Wheel pelo fato de a linha contínua da parte rear indicar que a parte está contida na instância de Car. As multiplicidades não foram observadas.

O classificador estruturado Car possui as duas partes na sua estrutura interna, mas seu respectivo código faz referência apenas à parte Wheel. A parte e, do tipo Engine, se relaciona com dois objetos da parte rear, do tipo Wheel, mas como a navegabilidade não é especificada, apenas a classe Wheel tem um atributo para se referir à classe Engine.

A engenharia reversa produziu um modelo com classes com as características do código. Algumas informações foram perdidas. Após recuperar o modelo com as classes, o diagrama de estrutura criado, a partir das classes do modelo, não continha a multiplicidade e o conector especificados no modelo inicial.

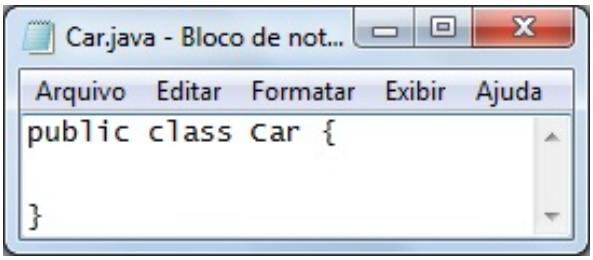

Figura 3.64. Código gerado pela EA a partir do modelo da Figura 3.62 após ajuste.

3.5.5.2 Estrutura Composta II

A Figura 3.65 mostra o diagrama de estrutura composta elaborado na ferramenta Astah* para o estudo de caso 5 (ii). A Figura 3.66 mostra o código gerado para as classes Car e Wheel. Como foi mencionado no estudo de caso anterior, a ferramenta Astah* não gera um código a partir de um diagrama, mas sim a partir das classes do modelo.

A classe Car foi gerada sem nenhum atributo. As partes declaradas no classificador estruturado mostradas no modelo não foram declaradas no código da classe Car. Uma vez que as partes fazem parte da estrutura interna do classificador estrutura, e são contornadas por linha contínua, a classe Car deveria fazer referência às partes. A classe Wheel foi gerada corretamente, com os dois atributos do tipo String. Isso permite deduzir que se o diagrama de estrutura composta for usado para complementar o diagrama de classe, o mapeamento do modelo para o código é mais confiável. Isto é, não é recomendado gerar o código a partir de um diagrama de estrutura composta diretamente na Astah*, pois algumas estruturas não são consideradas no mapeamento.

A Figura 3.67 mostra o modelo elaborado na ferramenta RSA referente ao estudo de caso 5 (ii). A classe Wheel foi criada com os atributos tire e size do tipo String, conforme especificado no estudo de caso. A Figura 3.68 mostra o código gerado. Foram gerados quatro atributos do tipo Wheel na classe Car. A classe Wheel foi gerada com os dois atributos do tipo String. O código reflete o no modelo.

A Figura 3.69 mostra a interação que a RSA faz com o usuário durante o processo de inserção de uma parte no classificador estruturado Car. É perguntado ao usuário se

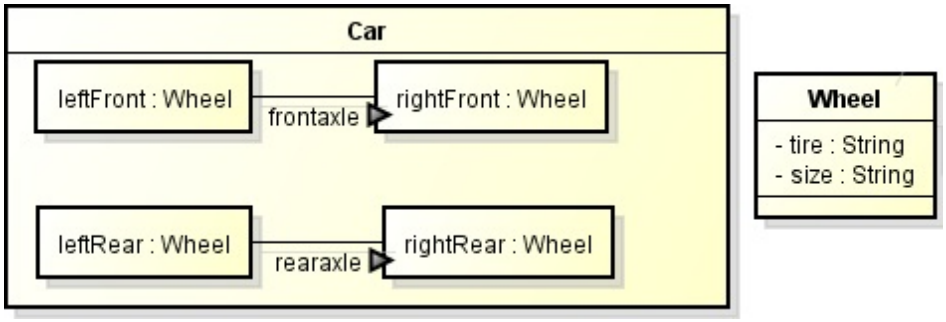

Figura 3.65. Modelo elaborado na ferramenta Astah* referente ao estudo de caso 5(ii).

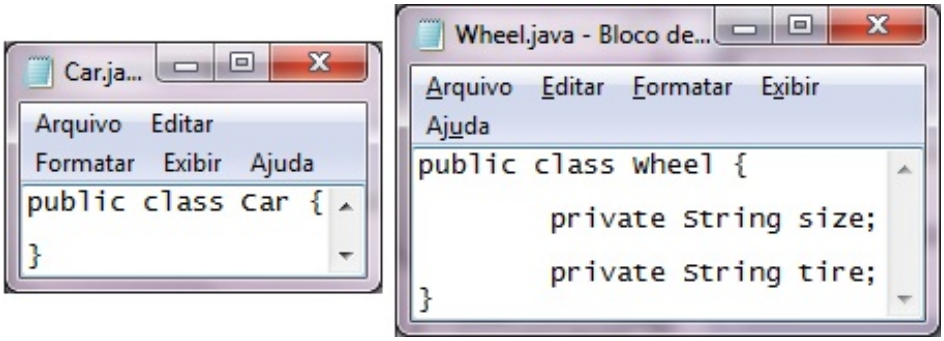

Figura 3.66. Código gerado pela Astah* a partir do modelo da Figura 3.65.

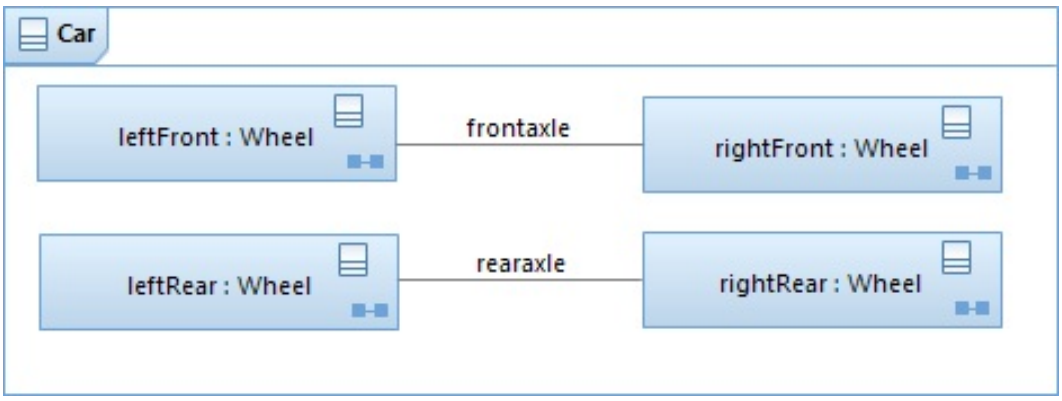

Figura 3.67. Modelo elaborado na ferramenta RSA referente ao estudo de caso 5(ii).

ele deseja criar uma classe ou selecionar um elemento existente, por exemplo.

A Figura 3.70 mostra a interação que RSA faz com o usuário para mostrar as alterações que estão sendo feitas do código para modelo. Nesse caso, não há nenhuma alteração, o modelo e o código estão consistentes. Como foi mencionado anteriormente, RSA não gera código a partir de um diagrama, mas sim a partir do modelo de classes. O modelo é construído durante a criação dos diagramas.

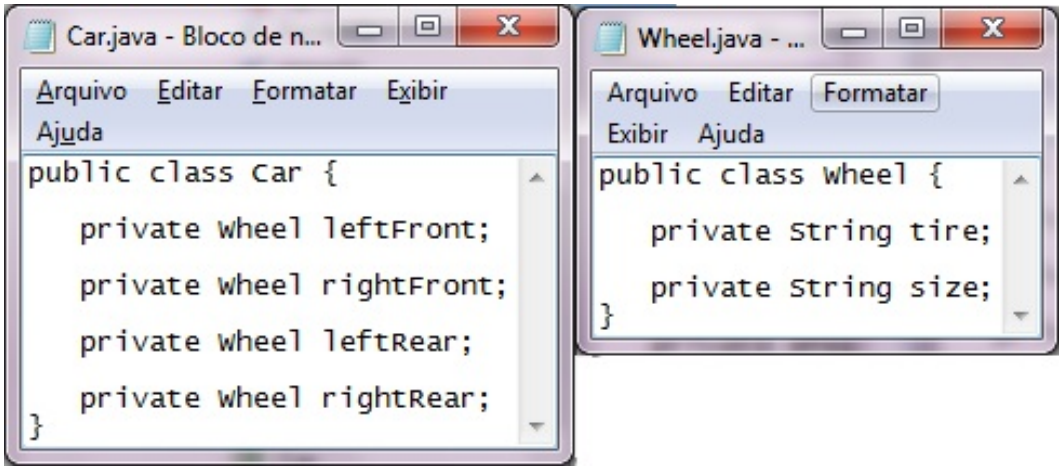

Figura 3.68. Código gerado pela RSA a partir do modelo da Figura 3.67.

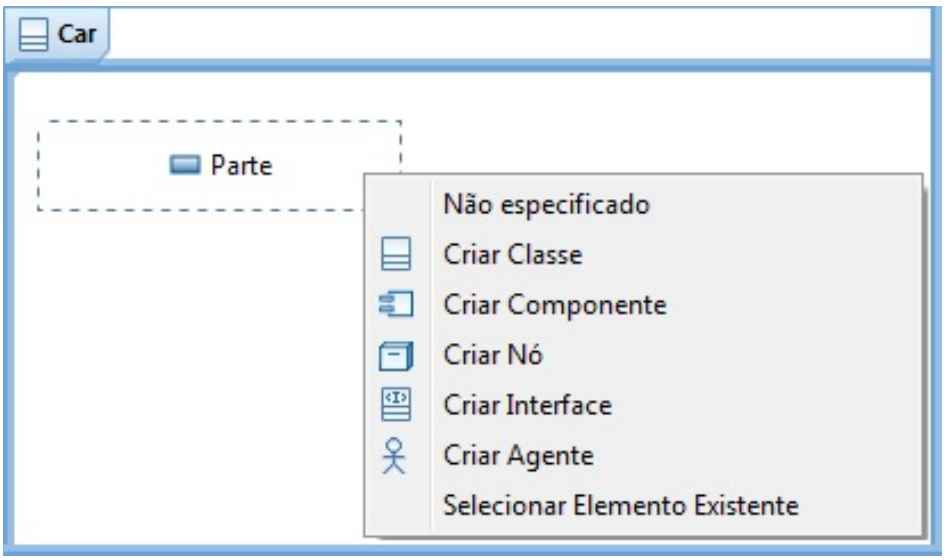

Figura 3.69. Interação do usuário com a ferramenta para a inserção de uma parte num classificador estruturado.

A Figura 3.71 mostra o diagrama de estrutura composta elaborado na ferramenta EA para o estudo de caso 5 (ii). A Figura 3.72 mostra o código gerado a partir do modelo da Figura 3.71. Pode-se observar na classe Car que os elementos que fazem parte da estrutura interna do classificador estruturado Car não foram mapeados. Consequentemente, a engenharia reversa não conseguiu recuperar tais informações ao fazer o mapeamento contrário.

Considerando os critérios estabelecidos neste capítulo, Astah* teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Parcialmente consistente (1). No primeiro

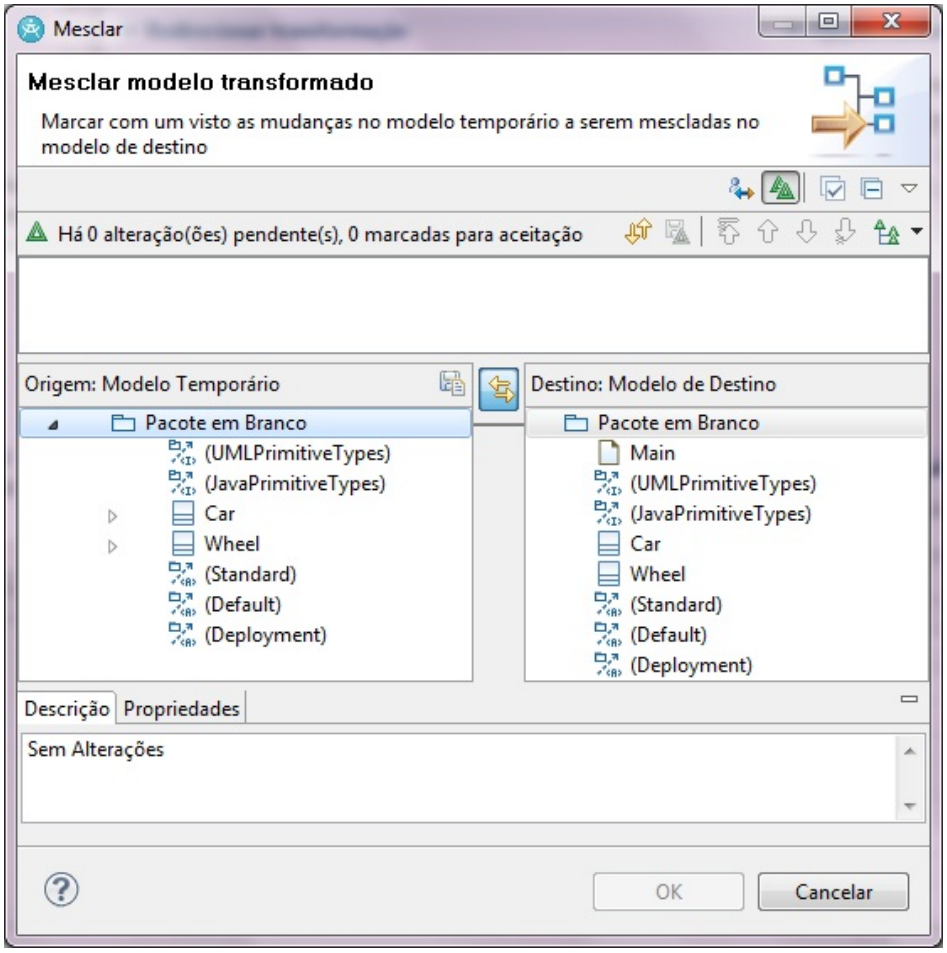

Figura 3.70. Interface utilizada pela RSA para mostrar ao usuário a transformação de modelo.

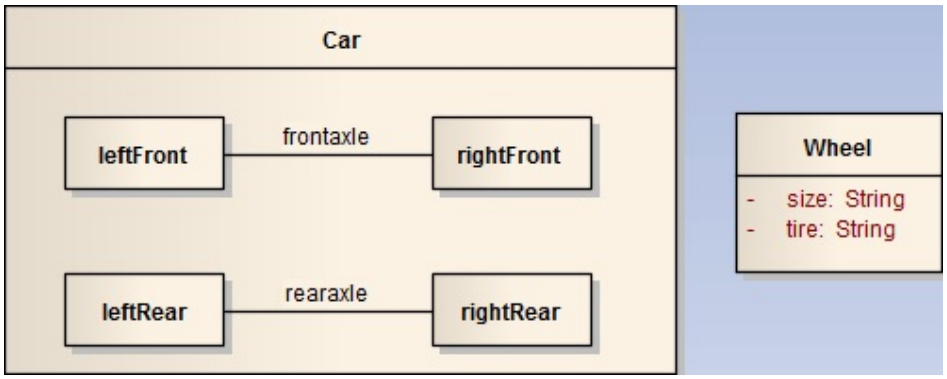

Figura 3.71. Modelo elaborado na ferramenta EA referente ao estudo de caso 5(ii).

modelo, o limite superior da multiplicidade não é observado. Um relacionamento de 1:2 e 1:N é mapeado da mesma forma. No segundo modelo As classes foram criadas. No

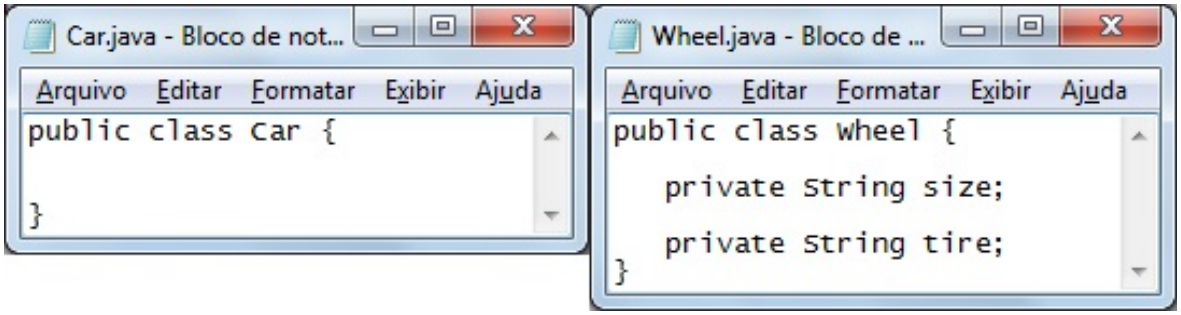

Figura 3.72. Código gerado pela EA a partir do modelo da Figura 3.71.

entanto, o código da classe Car foi gerado sem nenhum atributo. As partes inseridas no classificador estruturado Car, instâncias relacionadas com instância de Car, não foram mapeadas para o código.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Resolve parcialmente (1). A multiplicidade observada no primeiro modelo permite mais de um tipo de mapeamento. Astah* utilizou a estrutura Collection para representar a multiplicidade, mas não forneceu nenhuma opção de mapeamento para o usuário.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). No primeiro modelo, as multiplicidades e os nomes das extremidades de associação não foram recuperados, apesar das classes e das associações terem sido recuperadas. No segundo modelo, a engenharia reversa gerou o modelo com as classes Car e Wheel. A classe Wheel foi recuperada corretamente. No entanto, o classificador estruturado Car não teve as partes recuperadas, uma vez que não haviam sido mapeadas pela engenharia de ida.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). A Astah* permite inserir uma parte no classificador estruturado, mas caso o diagrama de estrutura composta não esteja associado a um diagrama de classes, algumas estruturas não são consideradas no mapeamento.

Considerando os critérios estabelecidos neste capítulo, RSA teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Consistente (2). Em relação ao primeiro e ao segundo modelo, o código gerado é consistente. As multiplicidades foram mapeadas corretamente, assim como os nomes das extremidades de associação. Algumas informações, como os nomes dos conectores - frontaxle e rearaxle -, não foram mapeadas, mas se tratam de informações conceituais.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Resolve parcialmente (1). A multiplicidade observada no primeiro modelo permite mais de um tipo

de mapeamento. RSA utilizou a estrutura arranjos e a estrutura Set para representar a multiplicidade, mas não forneceu nenhuma opção de mapeamento para o usuário.

C3 – Consistência Interna da Ferramenta: Consistente (2). A engenharia reversa foi capaz de recuperar os elementos presentes no modelo inicial, uma vez que o mapeamento do modelo para o código considerou tais elementos. Os nomes dos conectores frontaxle e rearaxle não foram considerados; no entanto, são informações conceituais, assim como o relacionamento de composição.

C4 – Flexibilidade e Robustez da Ferramenta: Robusta (2). O diagrama de estrutura composta é gerado de forma consistente a partir do diagrama de classe. RSA faz engenharia à frente e engenharia reversa por meio de transformações de modelo. O que é mapeado, na verdade, são as classes do modelo e não os diagramas propriamente ditos. Ao fazer uma transformação, RSA permite verificar e selecionar as alterações que estarão ocorrendo.

Considerando os critérios estabelecidos neste capítulo, EA teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Parcialmente consistente (1). As classes correspondentes ao classificador estruturado e às partes foram criadas. No entanto, o código da classe Car foi gerado sem nenhum atributo. No primeiro modelo, a multiplicidade foi ignorada. No segundo modelo, as partes inseridas no classificador estruturado Car, instâncias relacionadas com a instância Car, não foram mapeadas para o código.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Não se aplica (0). A multiplicidade entre as partes rear e e foi mapeada para o código. Desta forma, não foi possível verificar a capacidade de EA resolver ambiguidades.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). No primeiro modelo, as classes foram recuperadas, mas como foi perdida informação após engenharia à frente, o modelo recuperado não correspondeu totalmente ao modelo inicial. No segundo modelo, a engenharia reversa gerou o modelo com as classes Car e Wheel. A classe Wheel foi recuperada corretamente. No entanto, o classificador estruturado Car não teve as partes recuperadas, uma vez que não haviam sido mapeadas durante a engenharia à frente.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). A EA permite inserir uma parte no classificador estruturado, mas não a considera no mapeamento. A EA permitiu a inconsistência entre o diagrama de estrutura composta e o modelo. Foi possível criar partes no diagrama de estrutura composta sem que se definissem seus tipos.

3.5.6 ESTUDO DE CASO 6 - Fragmento Combinado

3.5.6.1 Fragmento Combinado I

A Figura 3.73 mostra o diagrama de sequência elaborado na ferramenta Astah* para o estudo de caso 6 (i). A Figura 3.74 mostra o código gerado a partir do modelo da Figura 3.73. O fragmento combinado *alt* foi inserido sem nenhum tipo de restrição, isto é, com qualquer disposição, mesmo que causasse ambiguidade.

Foram geradas apenas as classes com seus corpos vazios. Os mensagens listadas no diagrama de sequência foram ignorados pela ferramenta. Da mesma forma, o fragmento combinado *alt*, que representa uma estrutura condicional, foi ignorado. Isso reforça a ideia de que a Astah* gera o código a partir do que está definido nas classes do modelo e não de um diagrama.

Uma chamada de método no diagrama de sequência é implementada na linguagem Java como um método ou operação. A chamada assíncrona da classe B para a classe A, no modelo, deveria ter sido mapeada na classe A como um método, no código. A classe B chamou um método da classe A. No entanto, nem as chamadas de métodos nem o fragmento combinado *alt* foram mapeados para o código. A continuação (continuation) também não foi mapeada.

A Figura 3.75 mostra o diagrama de sequência elaborado na RSA referente ao estudo de caso 6 (i). A Figura 3.76 e a Figura 3.77 mostram a forma como a ferramenta RSA permite a inserção de um fragmento combinado no diagrama de sequência. Escolhe-se o fragmento combinado e, no diagrama de sequência, mostra-se a área que ele irá cobrir. Após delimitar a área que será encoberta pelo fragmento combinado, é exibida uma tela perguntando ao usuário quais linhas de vida estarão encobertas, de fato, pelo fragmento combinado. A Figura 3.78 mostra o código gerado pela RSA a partir do modelo da Figura 3.75. O fragmento combinado e as continuações (continuations) não foram mapeadas.

A Figura 3.79 mostra o diagrama de sequência elaborado na ferramenta EA para o estudo de caso 6 (i). A versão de teste utilizada no trabalho não disponibiliza a opção de engenharia à frente quando se está trabalhando com diagramas de sequência. Dessa forma, não foi possível analisar o mapeamento feito pela ferramenta EA para este estudo de caso.

3.5.6.2 Fragmento Combinado II

A Figura 3.80 mostra o diagrama de sequência elaborado na Astah* referente ao estudo de caso 6 (ii). O código gerado pela Astah* a partir do modelo da Figura 3.80 pode

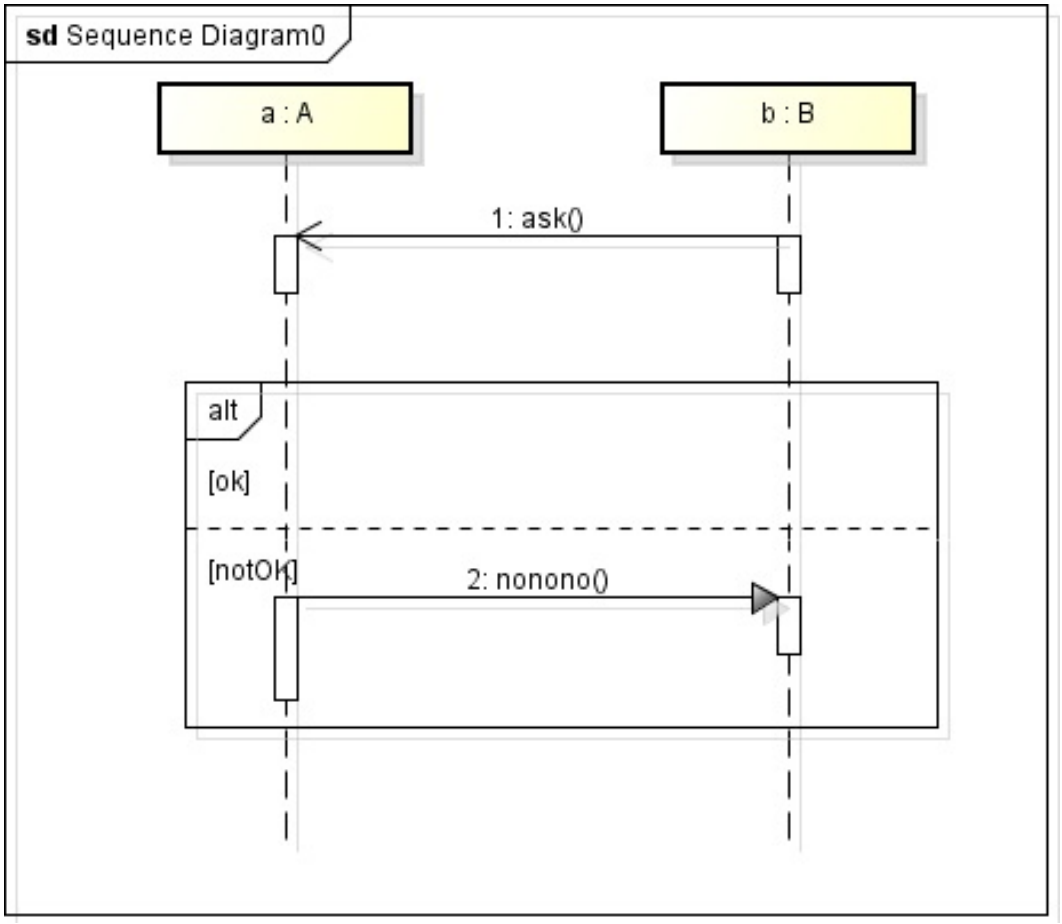

Figura 3.73. Modelo elaborado na ferramenta Astah* referente ao estudo de caso 6(i).

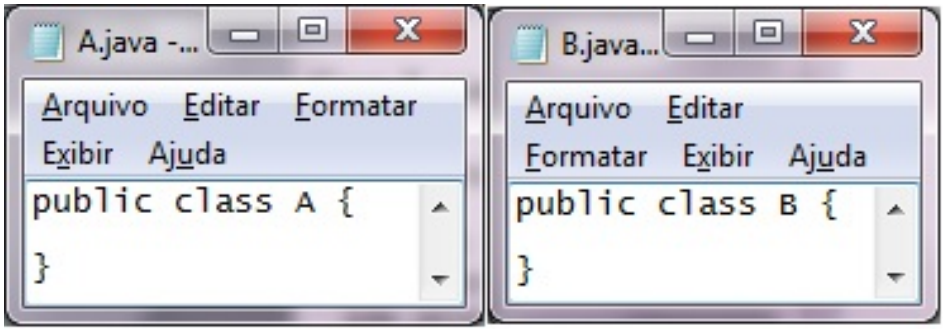

Figura 3.74. Código gerado pela Astah* a partir do modelo da Figura 3.73.

ser visto na Figura 3.81. Da mesma forma que no estudo de caso anterior, Astah* gerou apenas as classes vazias. As chamadas de métodos e o fragmento combinado *loop* foram ignorados. O código foi criado sem nenhum tipo de referência a esses dois tipos de elementos.

Um dos objetivos desse estudo de caso é verificar a capacidade de resolução de

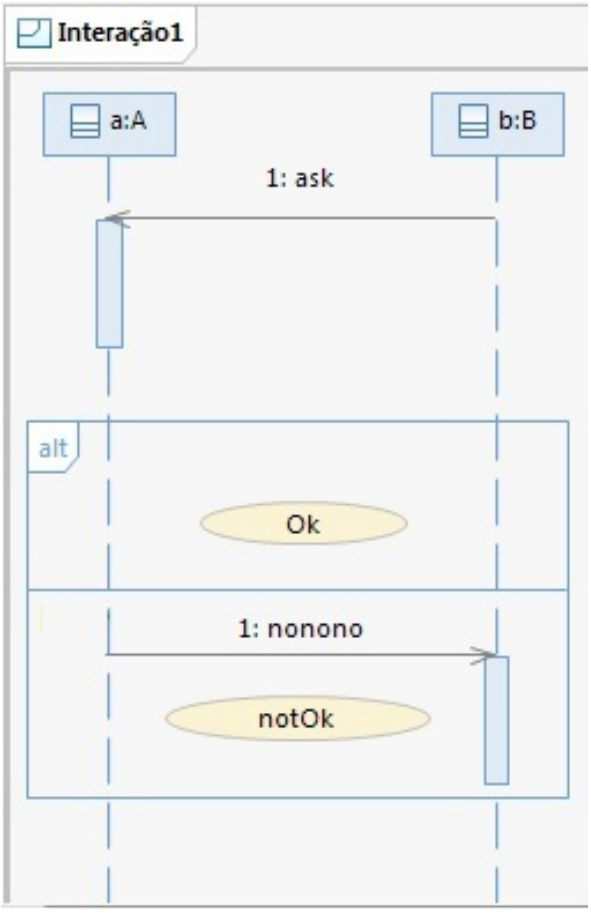

Figura 3.75. Modelo elaborado na ferramenta RSA referente ao estudo de caso 6(i).

ambiguidade da ferramenta. O fragmento combinado *loop* foi a estrutura analisada. Entretanto, se nem as chamadas das linhas de vida foram mapeados para o código, quanto mais o fragmento combinado.

A Figura 3.82 mostra um diagrama de sequência no momento em que está sendo definida a área de cobertura de um fragmento combinado. Como se pode observar, a área encobre a linha de vida A e a linha de vida B, com a chamada m2 completamente no seu interior e a chamada m1 parcialmente.

A disposição deste fragmento combinado gera ambiguidade. Apenas a mensagem m2 seria influenciada, uma vez que o fragmento combinado a encobre totalmente, ou a mensagem m1 também seria influenciada, uma vez que a operação faz parte da linha de vida B? Para evitar ambiguidade, RSA exibe uma tela perguntando ao usuário quais linhas de vida estarão encobertas, de fato, pelo fragmento combinado, como pode ser visto na Figura 3.83. Dessa forma, pode-se deduzir que RSA considera a cobertura das mensagens e não das operações existentes nas linhas de vida.

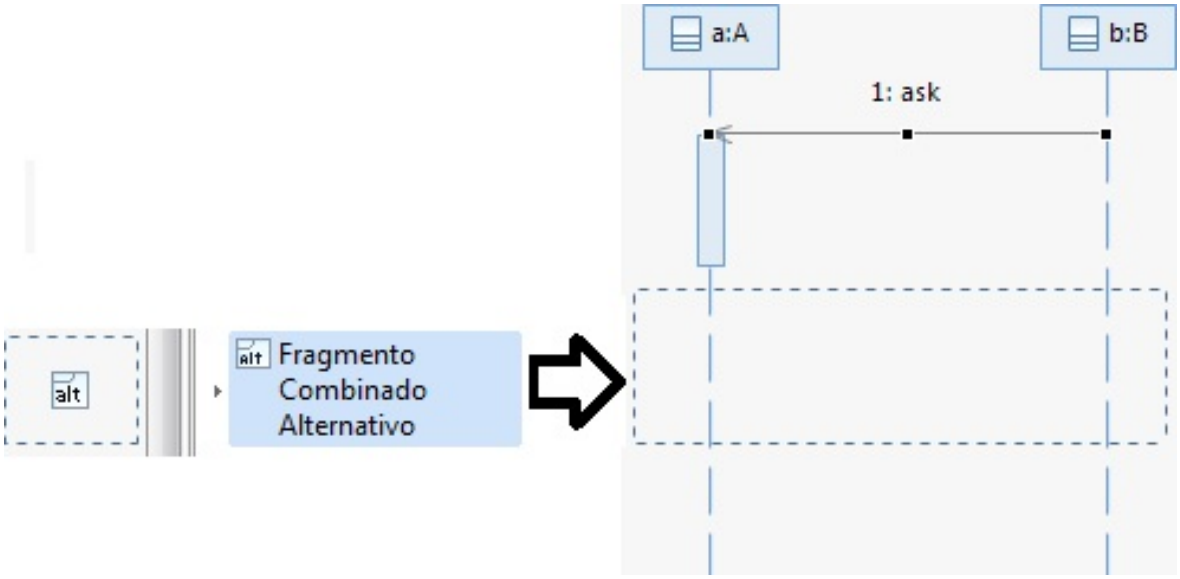

Figura 3.76. Fragmento combinado inserido no diagrama de sequência na ferramenta RSA.

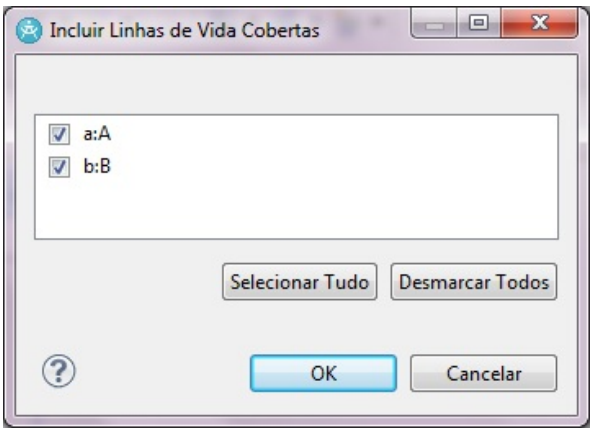

Figura 3.77. Interface utilizada pela RSA permitindo ao usuário definir a área de cobertura do fragmento combinado.

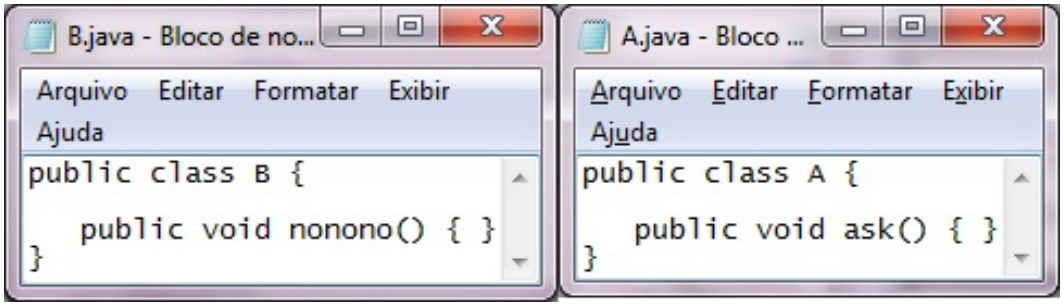

Figura 3.78. Código gerado pela RSA a partir do modelo da Figura 3.75.

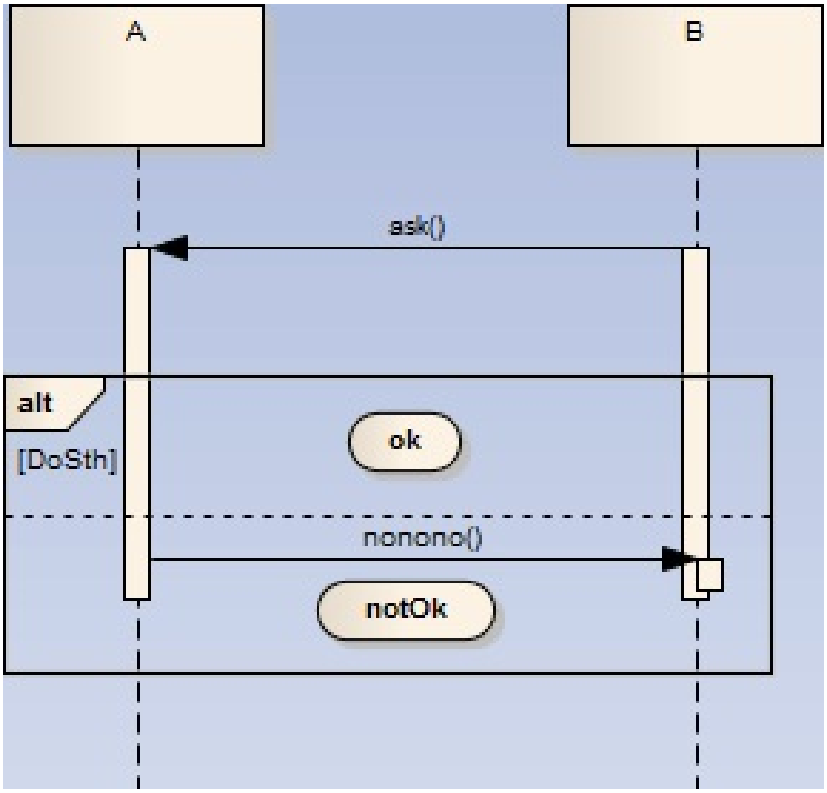

Figura 3.79. Modelo elaborado na ferramenta EA referente ao estudo de caso 6(i).

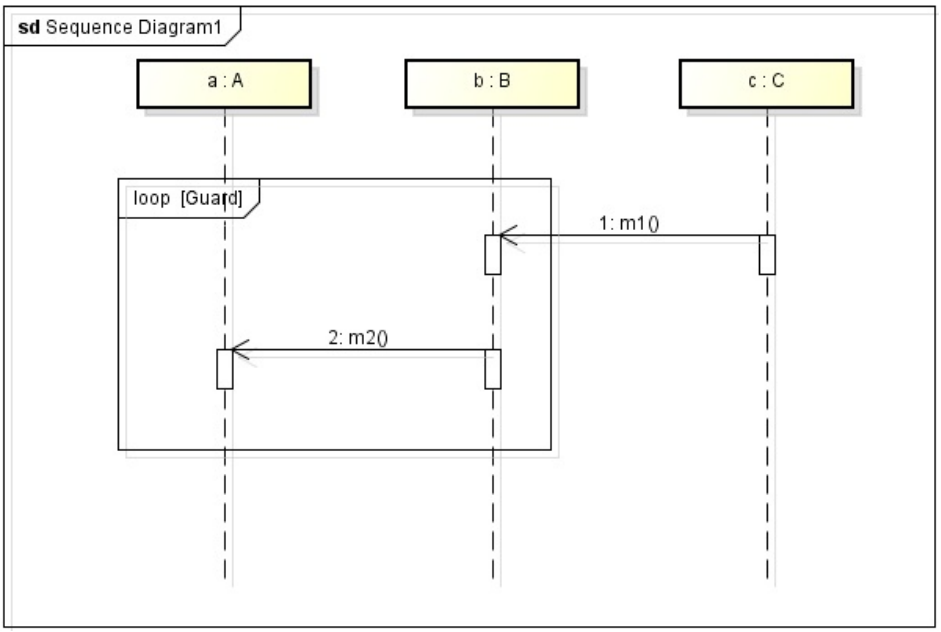

Figura 3.80. Modelo elaborado na ferramenta Astah* referente ao estudo de caso 6 (ii).

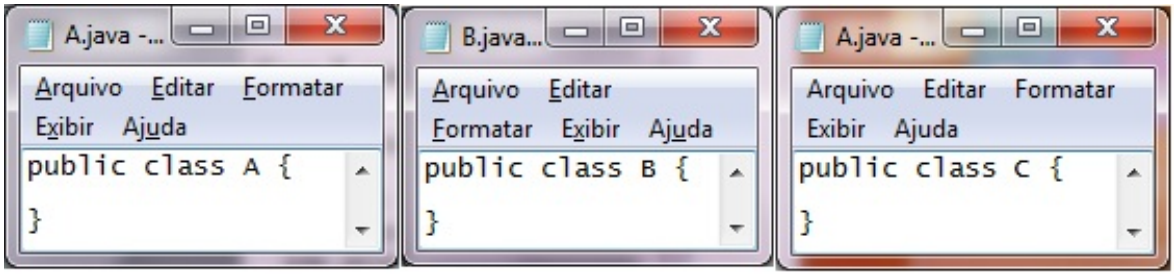

Figura 3.81. Código gerado pela Astah* a partir do modelo da Figura 3.80.

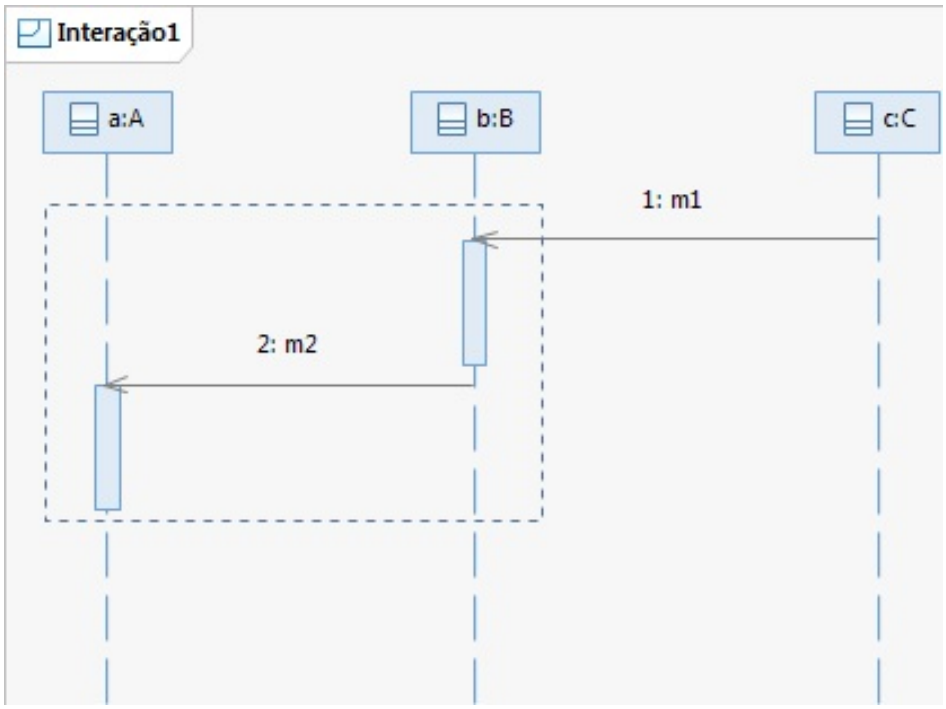

Figura 3.82. Modelo elaborado na ferramenta RSA referente ao estudo de caso 6 (ii).

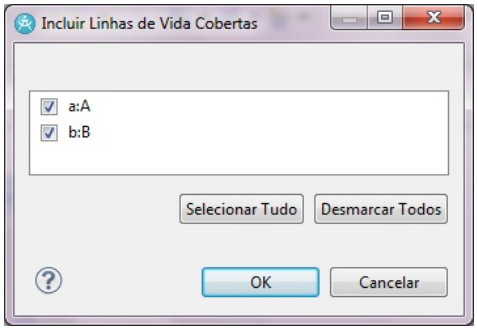

Figura 3.83. Interface utilizada pela RSA permitindo ao usuário definir a área de cobertura do fragmento combinado.

Porém, ocorre algo inesperado. O fragmento combinado é inserido no diagrama de sequência, mas as mensagens m1 e m2 são levadas para baixo, conforme Figura 3.84.

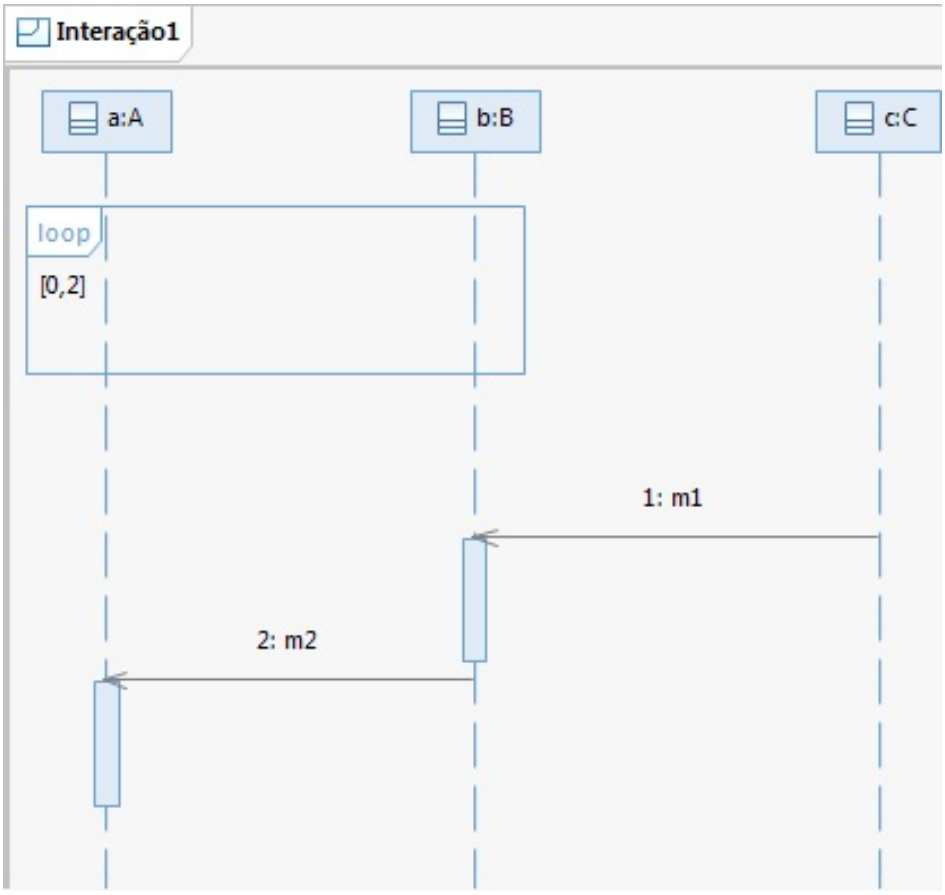

Figura 3.84. Diagrama de sequência após a inserção do fragmento combinado *loop*.

Em seguida, para tentar ajustar o diagrama de sequência conforme o modelo do estudo de caso, tentou-se inserir uma chamada assíncrona da classe B para a classe, como havia sido feito anteriormente. Uma tela foi exibida perguntando se era desejado criar um novo método ou utilizar um método existente, dentre eles o m2, como pode ser visto na Figura 3.85.

Após os ajustes, tentando adequar o diagrama de sequência ao modelo do estudo de caso, chegou-se na estrutura mostrada na Figura 3.86. Para que as duas chamadas fossem influenciadas pelo fragmento combinado *loop*, a área de cobertura teve que abranger completamente as chamadas. Ao contrário do que se tinha deduzido antes, para definir a área de cobertura e evitar ambiguidades, RSA não considera as operações das linhas de vida, mas sim as mensagens.

A Figura 3.87 mostra o código gerado a partir do diagrama de sequência da

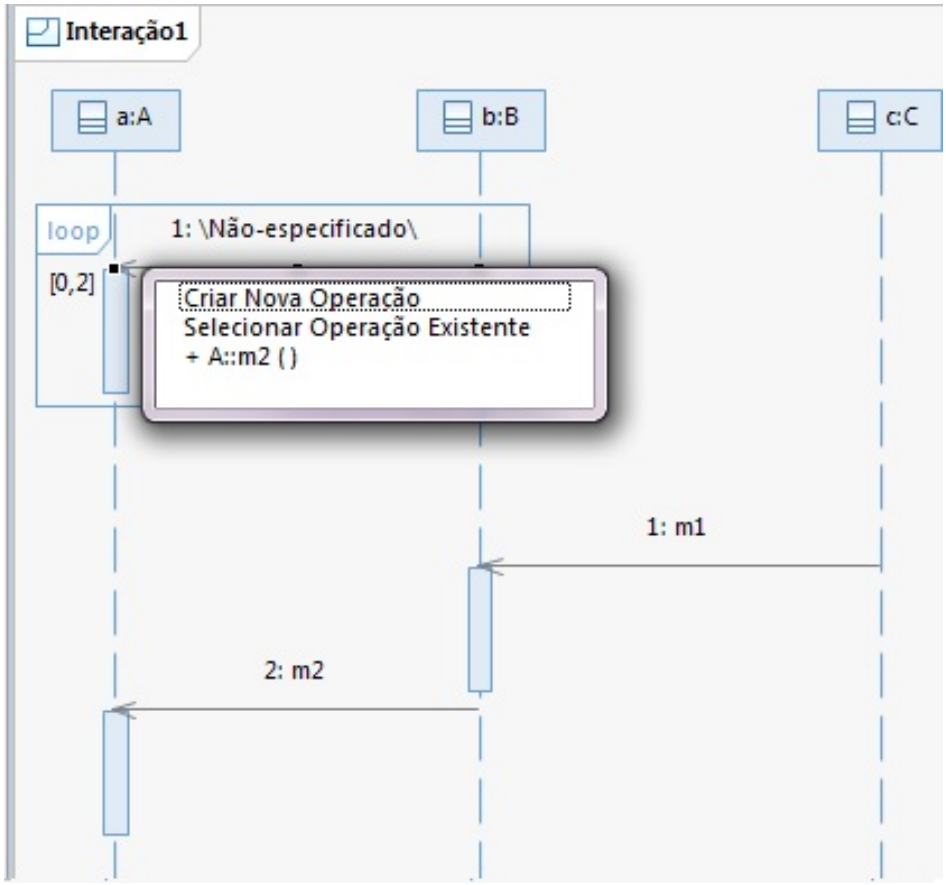

Figura 3.85. Inserção de uma chamada assíncrona dentro do fragmento combinado *loop*.

Figura 3.86. Apenas as mensagens foram mapeadas. O fragmento combinado e as continuações não foram mapeadas.

A Figura 3.88 mostra o diagrama de sequência elaborado na ferramenta EA para o estudo de caso 6(ii). A versão de teste utilizada no trabalho não disponibiliza a opção de engenharia à frente quando se está trabalhando com diagramas de sequência. Dessa forma, não foi possível analisar o mapeamento feito pela ferramenta EA para este estudo de caso.

Considerando os critérios estabelecidos neste capítulo, Astah* teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Parcialmente consistente (1). As classes do diagrama de sequência foram criadas corretamente, mas os detalhes do diagrama de sequência foram todos ignorados, as chamadas de método e os fragmentos combinados.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Não resolve (0). A ambiguidade que pode ser considerada nesse estudo de caso é a disposição do fragmento combinado. Astah* permite qualquer disposição do fragmento, inclusive a que causa

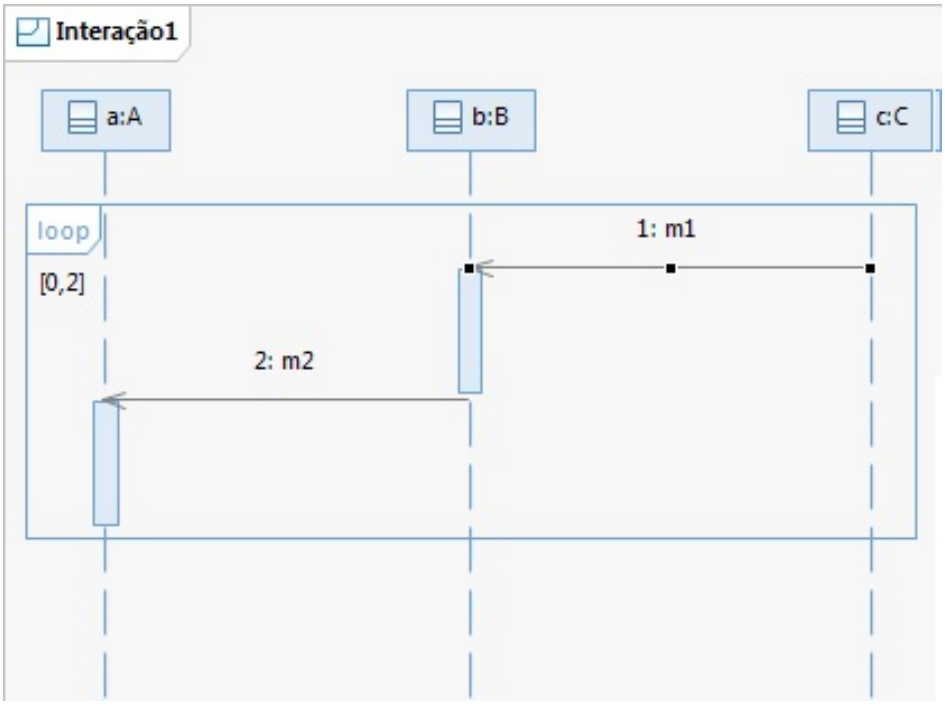

Figura 3.86. Diagrama de sequência final, após ajustes.

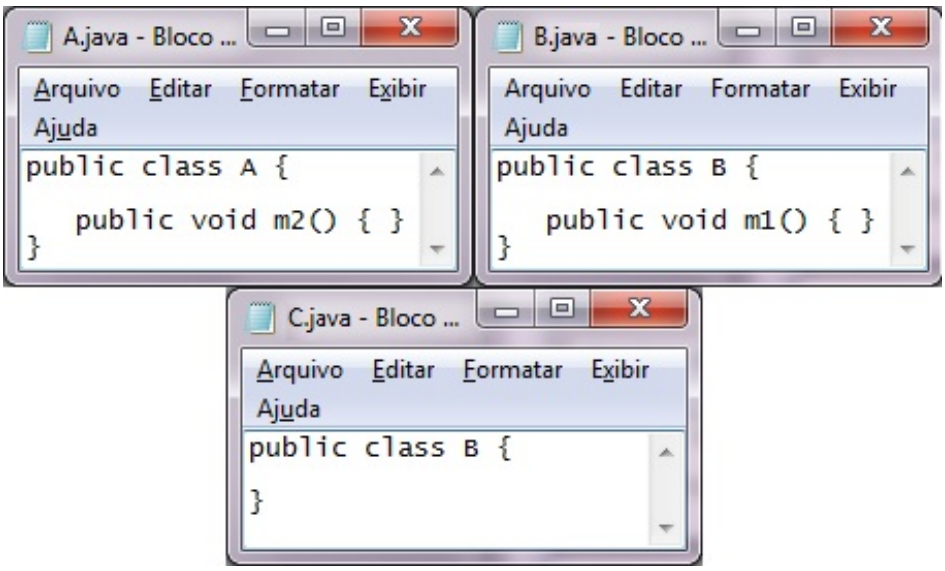

Figura 3.87. Código gerado a patir do diagrama de sequência da Figura 3.86 pela RSA.

ambiguidade.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). Nenhum os elementos do diagrama de sequência foram mapeados, com exceção das linhas de vida, que geraram as classes. Nem as chamadas de métodos foram mapeados. A

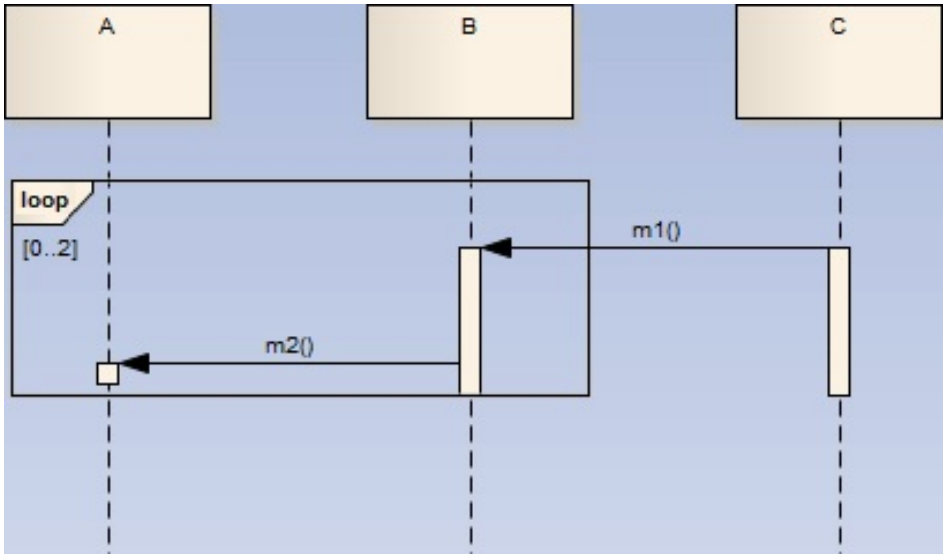

Figura 3.88. Modelo elaborado na ferramenta EA referente ao estudo de caso 6(ii).

engenharia reversa recuperou apenas as linhas de vida.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). A EA permitiu inserir os elementos no diagrama de sequência, mas eles não foram utilizadas no mapeamento do modelo para o código.

Considerando os critérios estabelecidos neste capítulo, RSA teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Parcialmente consistente (1). O código corresponde ao modelo. As classes foram criadas e seus métodos também. Mas os fragmentos combinados não foram mapeados.

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Resolve (2). A tela exibida ao usuário solicitando a seleção das linhas de vida que serão encobertas pelo fragmento combinado evita que haja ambiguidade no modelo. Apenas as disposições coerentes são permitidas pela RSA.

C3 – Consistência Interna da Ferramenta: Parcialmente consistente (1). Os fragmentos combinados não foram mapeados para o código. Consequentemente, não foram recuperaddos para o modelo. Desta forma, o modelo produzido pela engenharia reversa não possui as mesmas características do modelo inicial.

C4 – Flexibilidade e Robustez da Ferramenta: Parcialmente robusta (1). A RSA permitiu inserir os elementos no modelo. Através de mensagens e caixas de diálogo ao usuário, garantiu o modelo consistente. No entanto, não mapeou o fragmento combinado para o código.

Considerando os critérios estabelecidos neste capítulo, a RSA teve o seguinte desempenho:

C1 – Confiabilidade do Código Gerado: Não se aplica (0).

C2 – Capacidade de Resolução de Ambiguidades do Modelo: Não se aplica (0).

C3 – Consistência Interna da Ferramenta: Não se aplica (0).

C4 – Flexibilidade e Robustez da Ferramenta: Não se aplica (0).

A Tabela 3.3 mostra os resultados obtidos após a aplicação dos estudos de caso. Apesar de nortear e sistematizar a avaliação, os critérios estabelecidos e a atribuição de valores devem ser interpretados com cuidado. O objetivo foi analisar alguns poucos aspectos; foi verificado, em relação aos aspectos analisados, se a ferramenta atende, atende parcialmente ou não atende; não foi verificado o quão bem a ferramenta atende. Considerando os aspectos analisados, a ferramenta RSA nos deu a impressão de ser melhor atualizada. Existe sincronização entre os diagramas. Por exemplo, a inserção de uma chamada de método em um diagrama de sequência implica na inserção de um método na respectiva classe do modelo; a inserção de um relacionamento de composição em um diagrama de classe implica na inserção de uma parte com linha contínua no diagrama de estrutura composta.

As três ferramentas oferecem a opção de sincronização entre modelo e código. Neste caso, apenas as alterações são atualizadas para o código ou para o modelo. Se o modelo possui mais informações que o código - como o relacionamento de composição e as sentenças de propriedade -, a sincronização após alterações no código mantém as informações "exclusivas" do modelo. Desta forma, uma vez que o modelo tenha sido criado, a sincronização de modelo e código a cada iteração possibilita contornar problemas como a perda de informação conceitual ao longo do desenvolvimento.

De forma geral, as três ferramentas interagem pouco com o usuário. A maioria das interações são mensagens informativas de erro ou de sucesso após uma operação. Quando o mapeamento não é um para um, isto é, um elemento do modelo pode ser mapeado de mais de uma forma para o código, nenhuma ferramenta oferece alternativas de mapeamento. Mas existem situações com interação. A RSA evitou situações de ambiguidade, por exemplo, quando interagiu com o usuário para inserir corretamente o fragmento combinado no diagrama de sequência, conforme discutido na seção 3.5.6.2.

As estruturas básicas do diagrama de classe, como classes, atributos, métodos e relacionamentos, foram mapeadas corretamente em todas as ferramentas. No entanto, elementos como multiplicidade e navegabilidade tiveram mapeamentos divergentes. EA generalizou todas as multiplicidades como 1; Astah* considerou as diferentes multiplicidades, mas não observou os limites superiores definidos; RSA observou os limites definidos.

Tabela 3.3. Tabela sumário, contendo as ferramentas, critérios e valores atribuídos

ESTUDO DE CASO 1				**ESTUDO DE CASO 4**			
CRITÉRIOS	ASTAH	RSA	EA	CRITÉRIOS	ASTAH	RSA	EA
C1	1	1	1	C1	1	1	1
C2	0	0	0	C2	1	1	0
C3	1	1	1	C3	0	1	1
C4	1	1	1	C4	1	1	1
ESTUDO DE CASO 2				**ESTUDO DE CASO 5**			
CRITÉRIOS	ASTAH	RSA	EA	CRITÉRIOS	ASTAH	RSA	EA
C1	1	2	1	C1	1	2	1
C2	1	1	0	C2	1	1	0
C3	1	1	1	C3	2	2	1
C4	1	1	1	C4	1	2	1
ESTUDO DE CASO 3				**ESTUDO DE CASO 6**			
CRITÉRIOS	ASTAH	RSA	EA	CRITÉRIOS	ASTAH	RSA	EA
C1	2	2	2	C1	1	1	0
C2	0	0	0	C2	0	2	0
C3	2	2	2	C3	1	1	0
C4	2	2	2	C4	1	1	0

Capítulo 4

Aspectos Prático-Operacionais

Este capítulo apresenta alguns aspectos prático e operacionais. O processo de seleção das ferramentas é detalhado na seção 4.1. Além disso, são relatadas, na seção 4.2, as dificuldades encontradas e as soluções adotadas com relação: i) ao *download* e à instalação das ferramentas; ii) às dúvidas quanto à elaboração dos modelos nas ferramentas, como, por exemplo, a inserção de um relacionamento de associação ternária entre classes; iii) à questões sobre recursos e soluções das ferramentas, como, por exemplo, a possibilidade de se importar bibliotecas para instanciar no modelo o maior número de tipos de uma linguagem de programação específica; entre outros.

4.1 Aplicação dos Critérios de Seleção das Ferramentas

Após aplicar o critério C1 e o critério C2 encontraram-se as ferramentas apresentadas na Tabela 4.1.

Em seguida, aplicou-se os critérios C3, C4 e C5, disponibilidade de cópia pela internet, tipo de licença e ferramenta em desenvolvimento e suporte à UML 2, respectivamente, obtendo-se o seguinte resultado, apresentado na Tabela 4.2:

- A ferramenta Rational TAU não possui versão gratuita ou licença para teste, além de a última versão ter sido lançada em fevereiro de 2009;

- As informações encontradas sobre as ferramentas ArgoUML, BOUML, FUJABA, StarUML, Rational TAU, Together e Umbrello permitem afirmar que elas não estão em desenvolvimento;

Tabela 4.1. Ferramentas que suportam UML e Java encontradas no Google e no Bing.

ID	FERRAMENTA
1	ArgoUML
2	Artisan Studio
3	Astah*
4	BOUML
5	Eclipse (plugin MyEclipse)
6	EclipseUML
7	Enterprise Architect
8	FUJABA
9	MagicDraw UML
10	MicroGOLD with Class
11	NetBeans
12	Poseidon UML
13	Power Designer
14	Rational Software Architect - RSA
15	Rational Rhasody
16	Rational TAU
17	StarUML
18	Together
19	Umbrello
20	Umodel
21	Visual Paradigm

- Não foram encontradas versões das ferramentas ArgoUML, Artisan Studio, FUJABA, MicroGOLD with Class, NetBeans e PowerDesigner que suportassem a UML 2;

Considerando que a ferramenta TAU G2 não possui licença para teste, decidiu-se, então, descartá-la por não ser possível testá-la. As ferramentas ArgoUML, Artisan Studio, BOUML, FUJABA, MicroGOLD with Class, NetBeans, PowerDesigner, StarUML, TAU G2, Together e Umbrello não estão mais em desenvolvimento e/ou não possuem versões que suportam a UML2. Desta forma, decidiu-se descartá-las. Após estas considerações, as ferramentas elencadas até o critério C5 são listadas na Tabela 4.3:

A partir da Tabela 4.3, aplicou-se o critério C6, que consiste em verificar se há fórum de discussão ativo e a popularidade da ferramenta. Após a busca de informações nos fóruns das ferramentas buscando preencher a Tabela 4.4, algumas informações não foram encontradas. Foram enviadas mensagens para o suporte das ferramentas listadas na Tabela 4.3 para levantar informação. Alguns suportes não responderam, outros afir-

Tabela 4.2. Disponibilidade das ferramentas CASE para *download* e suas licenças.

ID	FERRAMENTA	*DOWNLOAD*	TIPO DE LICENÇA	EM DESEVOLVIMENTO	UML 2
1	ArgoUML	SIM	Gratuita	NÃO	NÃO
2	Artisan Studio	SIM	30 dias	SIM	NÃO
3	Astah*	SIM	20 dias + 30 dias	SIM	SIM
4	BOUML	SIM	Gratuita	NÃO	SIM
5	Eclipse (MyEclipse)	SIM	30 dias	SIM	SIM
6	EclipseUML	SIM	30 dias	SIM	SIM
7	Enterprise Architect	SIM	30 dias	SIM	SIM
8	FUJABA	SIM	Gratuita	NÃO	NÃO
9	MagicDraw UML	SIM	Demo	SIM	SIM
10	MicroGOLD with Class	SIM	30 dias	SIM	NÃO
11	NetBeans	SIM	Gratuita	SIM	NÃO
12	Poseidon UML	SIM	30 dias	SIM	SIM
13	Power Designer	SIM	30 dias	SIM	NÃO
14	RSA	SIM	30 dias	SIM	SIM
15	Rational Rhapsody	SIM	30 dias	SIM	SIM
16	Rational TAU	SIM	Paga	NÃO	SIM
17	StarUML	SIM	Gratuita	NÃO	SIM
18	Together	SIM	30 dias	NÃO	SIM
19	Umbrello	SIM	-	NÃO	SIM
20	Umodel	SIM	30 dias	SIM	SIM
21	Visual Paradigm	SIM	10 dias + 20 dias	SIM	SIM

Tabela 4.3. Ferramentas após a aplicação do critério C3, C4 e C5.

ID	FERRAMENTA
1	Astah*
2	Eclipse (plugin MyEclipse)
3	EclipseUML
4	Enterprise Architect
5	MagicDraw UML
6	Poseidon UML
7	Rational Rhapsody
8	Rational Software Architect – RSA
9	Umodel
10	Visual Paradigm

maram não ter informação além das discussões no fórum, outros afirmaram não poder revelar informação a quem não fosse cliente da empresa. As informações encontradas ao fim de todas as pesquisas e consultas, realizadas no período de janeiro a fevereiro de 2011, são apresentadas na Tabela 4.4. A Tabela 4.5 apresenta informações, no campo OBSERVAÇÕES, que especificam as respostas das mensagens enviadas para o suporte e/ou fórum das ferramentas. A Tabela 4.6 mostra a conceituação das ferramentas no fórum da UML [UML, 2011] segundo alguns aspectos, a saber: usabilidade, desenho, simulação (executabilidade) e conformidade com as especificações. Todas as ferramentas listadas e avaliadas no fórum da UML estão presentes nesta pesquisa. Porém, nem

todas as ferramentas desta pesquisa estão presentes no fórum da UML.

Tabela 4.4. Aplicação do critério C6: Informações sobre o fórum.

ID	FERRAMENTAS	MENSAGENS	MEMBROS	VISUALIZAÇÕES
1	Astah*	789	451.082	21.771
2	Eclipse (myEclipse)	21.205	Não encontrado	Não encontrado
3	EclipseUML	571	2.044	Não encontrado
4	Enterprise Architect	75.672	19.241	Não encontrado
5	MagicDraw	1.152	324	Não encontrado
6	Poseidon UML	6.830	1.674	Não encontrado
7	Rhapsody	1.968	Não encontrado	70.512
8	RSA	18.338	Não encontrado	124.783
9	UModel	8.814	11.615	Não encontrado
10	Visual Paradigm	8.854	Não encontrado	Não encontrado

Tabela 4.5. Resposta aos *e-mails* enviados para o suporte das ferramentas CASE.

ID	FERRAMENTAS	OBSERVAÇÕES
1	Astah*	Respondeu. Explicação da discrepância de informações do fórum.
2	Eclipse (myEclipse)	Não respondeu.
3	EclipseUML	Respondeu. Não revela informações além do que está no fórum.
4	Enterprise Architect	Respondeu. Nenhuma informação além das constantes no fórum.
5	MagicDraw	Respondeu. Nenhuma informação além das constantes no fórum.
6	Poseidon UML	Não respondeu.
7	Rhapsody	Não respondeu.
8	RSA	Respondeu. Nenhuma informação além das constantes no fórum.
9	UModel	Não respondeu.
10	Visual Paradigm	Respondeu. Não pode revelar informações a quem não é cliente.

Tabela 4.6. Conceituação das ferramentas no fórum da UML [UML, 2011]

ID	FERRAMENTAS	Usabilidade	Desenho	Simulação/ Executabilidade	Conformidade com as especificações	TOTAL
1	Astah*	-	-	-	-	-
2	Eclipse (myEclipse)	-	-	-	-	-
3	EclipseUML	-	-	-	-	-
4	Enterprise Architect	3	4	1	3	11
5	MagicDraw	3	4	2	4	13
6	Poseidon UML	3	3	1	1	8
7	Rhapsody	2	3	3	4	12
8	RSA	3	3	2	4	12
9	UModel	3	4	1	2	10
10	Visual Paradigm	3	3	1	4	11

Considerando o número de membros, observa-se que a Astah* é a mais popular, seguida pela Enterprise Architect e Umodel, respectivamente. Observando-se apenas o número de mensagens, a Enterprise Architect é a mais popular; a Astah* não parece

ser tão expressiva, em vista da discrepância entre o número de mensagens e o número de membros.

Uma mensagem foi enviada ao suporte da ferramenta Astah* para se obter uma explicação para o baixo número de mensagens no fórum da Astah*, se comparado com o número de membros. Foi retornado que cerca de 38% dos membros são residentes no Japão e eles utilizam outro fórum em japonês. Além disso, muitos dos membros não possuem o inglês como sua primeira língua. Eles também oferecem suporte técnico por correio eletrônico para os clientes que possuem licenças válidas, uma vez que a maioria deles entra em contato diretamente via correio em vez de utilizar fórum. Essas são algumas razões que eles assumem para justificar o baixo número de mensagens no fórum.

É importante ressaltar que o número de mensagens informado no fórum das ferramentas está relacionado a toda funcionalidade da ferramenta. Isto é, são consideradas todas as mensagens que dizem respeito a qualquer tipo de recurso que a ferramenta oferece, seja relacionada à UML ou não. Se a ferramenta suporta o desenvolvimento Java, sem envolver UML, e o fórum aborda questões simplesmente de Java, as mensagens relacionadas a esta questão são consideradas. Desta forma, o número de mensagens por si só não é suficiente para dizer se a ferramenta é mais ou menos popular no que diz respeito à engenharia de ida e volta. O número de mensagens pode dizer o quanto uma ferramenta é discutida pelos usuários.

O número de mensagens, o número de membros e o número de visualizações são parâmetros que ajudam inferir o quanto a ferramenta é aceita ou o quanto as pessoas se interessam pela ferramenta. Eles foram utilizados para se escolher as três ferramentas consideradas mais adequadas para tornar o trabalho mais representativo.

O número de mensagens é a informação que está disponível nos fóruns de todas as ferramentas. A ferramenta Enterprise Architect possui o número de mensagens muito superior ao das outras ferramentas. A ferramenta Enterprise Architect também possui um número relativamente alto de membros se comparado ao das outras ferramentas. Por esses motivos, ela foi selecionada. Além dela, Astah* também foi selecionada pelo altíssimo número de membros. A terceira ferramenta selecionada foi a RSA. A RSA é uma ferramenta robusta muito utilizada no mercado e nos trabalhos científicos.

O conceito atribuído às ferramentas no fórum da UML, como pode ser visto na Tabela 4.6, mostra a Magic Draw com pequena vantagem em relação às outras ferramentas. Nossa intenção era utilizar neste trabalho a ferramenta mais bem conceituada no fórum da UML. No entanto, a versão da ferramenta Magic Draw disponível para teste não realiza engenharia à frente ou reversa, o que compromete sua avaliação neste trabalho. A Tabela 4.7 mostra as três ferramentas selecionadas que foram avaliadas.

Desejava-se avaliar também ferramentas gratuitas. Todavia, os recursos oferecidos por estas estavam bem aquém do que era esperado das ferramentas.

Tabela 4.7. Ferramentas Selecionadas

ID	FERRAMENTA
1	Astah*
2	Enterprise Architect
3	Rational Software Architect – RSA

4.2 Cópia, Instalação e Utilização das Ferramentas

A ferramenta RSA e Enterprise Architect possuem licença de teste para apenas 30 dias, enquanto que a Astah* possui licença para 40 dias, extensível por mais 50 dias. Considerando o tempo para aprender a usar a ferramenta e o amadurecimento e refinamento deste trabalho, 30 dias não foram suficientes para avaliar as ferramentas. Foi necessário instalá-las em outras duas máquinas, em momentos diferentes, para completar a avaliação.

As versões das três ferramentas utilizadas neste trabalho foram: i) Astah* - versão 6.3, Professional, liberada em novembro de 2010; ii) Enterprise Architect - versão 8.0, Professional, liberada em outubro de 2010; e iii) Rational Software Architect - versão 8.0, liberada em agosto de 2010.

Algumas dificuldades no momento da elaboração dos modelos dos estudos de caso, em cada ferramenta, foram: i) a disposição diferente, em cada ferramenta, dos elementos da UML e dos recursos da ferramenta; ii) os diferentes procedimentos para realizar a engenharia de ida e volta.

A inserção de elementos básicos no modelo, como classes, atributos e associações, é simples e pode ser feita de maneira similar em cada ferramenta. Existe uma barra, ao lado ou acima do diagrama, com elementos da UML. No entanto, para inserir outros elementos como, por exemplo, sentença de propriedades, em cada ferramenta se faz de uma forma diferente.

Enquanto que, para realizar o mapeamento de UML para Java na RSA, é necessário criar uma transformação de UML para Java (Transformation UML to Java) e definir alguns parâmetros, na Astah* e na Enterprise Architect bastava selecionar os modelos e clicar em um botão ou em um item de menu "gerar código". Apesar de não ser necessário criar uma transformação nestas ferramentas, é possível definir alguns parâmetros como, por exemplo, a estrutura Java para qual será mapeada uma coleção.

Verificou-se que a associação n-ária não é suportada pelas ferramentas. RSA e Astah* não possuem o elemento associação ternária dentre os elementos da UML. Após contato com o suporte destas ferramentas, Astah* afirmou que futuramente pretende adicionar esse elemento à ferramenta. Enterprise Architect, apesar de permitir a representação de associações, não as mapeia para o código; no modelo, três classes são visualmente conectadas, mas após o mapeamento nenhuma estrutura no código considera a associação ternária.

A utilização de bibliotecas de uma linguagem específica para instanciar elementos no modelo UML é possível nas três ferramentas – neste trabalho, importamos bibliotecas da linguagem Java. Após não identificarmos como realizar essa importação na ferramenta Astah*, enviamos mensagem ao suporte e, após seguir as orientações recebidas, conseguimos realizar a operação com sucesso.

As dúvidas de utilização foram resolvidas através de consultas aos fóruns e aos materiais disponibilizados nos sítios das ferramentas. Além disso, uma série de mensagens foi enviada aos suportes das ferramentas para responder questões específicas relacionadas: i) à licença; ii) a recursos da ferramenta; iii) a como se realizar determinada operação; iv) tipo de perfil da UML utilizado pela ferramenta; entre outros.

Capítulo 5

Conclusão

5.1 Considerações Finais

Neste trabalho foi estudada a questão da engenharia de ida e volta entre a linguagem UML e a linguagem Java. Inicialmente, a intenção era realizar um trabalho similar ao de Kollmann e outros [Kollman et al., 2002] e verificar a capacidade das ferramentas CASE atuais realizarem a engenharia de ida e volta, considerando o diagrama de classes e a linguagem Java, utilizando um sistema real.

Como a UML 2 possui outros elementos importantes que não foram analisados naquele trabalho, foram adicionados alguns deles na análise deste trabalho. Aquele trabalho avaliou a transcrição do código para o modelo. Este trabalho avaliou a transcrição em ambos os sentidos e também questões técnicas de mapeamento do modelo para o código e do código para o modelo. Pela complexidade da análise, o trabalho se limitou a avaliar apenas três ferramentas. Além disso, este trabalho não objetivou exaurir todas as possibilidades de mapeamento entre as duas linguagens, mas apresentar alguns casos considerados relevantes.

Alguns trabalhos sobre interação entre ferramenta e usuário (desenvolvedor) relacionados à engenharia de ida e volta, isto é, ao mapeamento modelo=>código e código=>modelo, focaram em aspectos de implementação [Akehurst et al., 2007] [Gessenharter, 2008] [Gessenharter, 2009] [Diskin et al., 2008]. Por outro lado, outros trabalhos focaram na usabilidade da ferramenta de forma geral, cuidando da questão de IHC - Interação Humano-Computador [de Souza, 2005] [de Souza et al., 2010]. No entanto, não conhecemos trabalhos que tratam de questões técnicas de opções de mapeamento entre UML e Java. Além de analisar a transcrição de UML para Java e vice-versa, este trabalho se preocupou em avaliar se existem ou não opções de mapeamento. Cabe ressaltar que dimensões como qualidade da interação, comunicabilidade,

apreensibilidade são importantes, mas não fizeram parte do escopo do trabalho para efeito de simplificação.

Verificamos a engenharia de ida e volta implementada pelas ferramentas. Através das técnicas de engenharia à frente e engenharia reversa, as ferramentas possibilitam a sincronização de modelo e código. Aspectos que não são mapeados do modelo para o código - informações conceituais - permanecem inalterados no modelo após sincronizar o código alterado com o modelo. Existem alguns aspectos do modelo que não são mapeados da maneira esperada e podem comprometer a qualidade do código.

Observamos que a interação da ferramenta com o usuário é basicamente restrita ao envio de mensagens de erro ou de sucesso ao usuário após cada operação. Há casos em que ocorrem erros e não é exibido nenhum tipo de mensagem informativa. Diante de um mapeamento com mais de uma opção, não foi solicitado o auxílio do usuário. As ferramentas possuem uma opção *default*, para cada mapeamento, que pode ser modificada numa área de configuração.

A capacidade de transcrição das ferramentas de UML para Java não teve grandes mudanças, considerando trabalhos dos últimos anos [Akehurst et al., 2007] [Gessenharter, 2008]. As associações continuam sendo mapeadas sem diferenciar composição, agregação e associações simples. Nem todos os adornos das extremidades de associação são mapeados para o código. Os elementos que surgiram a partir da UML 2 - como o diagrama de estrutura composta e os fragmentos combinados do diagrama de sequência - são suportados pelas ferramentas. No entanto, o mapeamento dessas estruturas para o código ainda é precário.

5.2 Trabalhos Futuros

Em busca de maior aprimoramento sobre estudos de avaliação da interação das ferramentas CASE com o usuário (desenvolvedor), outros aspectos da relação UML e Java podem analisados. A ênfase deste trabalho foi o mapeamento de UML para Java, isto é, engenharia à frente. Desta forma, consideramos interessante estender o trabalho dando ênfase ao mapeamento de Java para UML.

A relação da UML com outras linguagens de programação orientadas a objeto, como C++, pode ser do interesse destes desenvolvedores.

Outro trabalho futuro seria analisar outras dimensões da usabilidade, como qualidade da interação, comunicabilidade, apreeensibilidade, entre outros.

Apenas três ferramentas foram avaliadas. Para melhor representação da realidade, seria interessante a avaliação de um número maior de ferramentas. Além disso,

consideramos importante que essa avaliação fosse feita por um grupo de desenvolvedores que não tivessem contato anterior com ferramentas específicas para avaliar aspectos de facilidade de aprendizado, dentre outros, e por um grupo de desenvolvedores com experiência nas ferramentas para avaliar aspectos como, por exemplo, precisão e consistência. Neste trabalho foram analisados apenas alguns aspectos. Acreditamos que uma maior abrangência trará maiores contribuições.

Referências Bibliográficas

[Akehurst et al., 2007] Akehurst, D.; Howells, G. & Maier, K. M. (2007). Implementing associations: UML 2.0 to Java 5. *Software and Systems Modeling*, 6(1):3--35.

[Ali, 2005] Ali, M. R. (2005). Why teach reverse engineering? *SIGSOFT Softw. Eng. Notes*, 30:1--4.

[Anda et al., 2006] Anda, B.; Hansen, K.; Gullesen, I. & Thorsen, H. K. (2006). Experiences from introducing UML-based development in a large safety-critical project. *Empirical Softw. Engg.*, 11:555--581.

[Angyal et al., 2006] Angyal, L.; Lengyel, L. & Charaf, H. (2006). An Overview of the State-of-The-Art Reverse Engineering Techniques. *7th International Symposium of Hungarian Researchers on Computational Intelligence.*

[Angyal et al., 2008] Angyal, L.; Lengyel, L. & Charaf, H. (2008). A synchronizing technique for syntactic model-code round-trip engineering. Em *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pp. 463--472.

[Antkiewicz & Czarnecki, 2006] Antkiewicz, M. & Czarnecki, K. (2006). Framework-specific modeling languages with round-trip engineering. Em *MoDELS'06*, pp. 692--706.

[Arcelli et al., 2005] Arcelli, F.; Masiero, S.; Raibulet, C. & Tisato, F. (2005). A comparison of reverse engineering tools based on design pattern decomposition. Em *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, pp. 262--269.

[Artale et al., 2010] Artale, A.; Calvanese, D. & Ibáñez García, A. (2010). Full satisfiability of UML class diagrams. Em *Proceedings of the 29th international conference on Conceptual modeling*, ER'10, pp. 317--331, Berlin, Heidelberg. Springer-Verlag.

[Astah*, 2011] Astah* (2011). Astah* professional versão 6.3. Disponível em `http://astah.change-vision.com/en/product/astah-professional.html`, acessado em 07/2011.

[Barsotti, 2003] Barsotti, D. (2003). Giving back in a big way. Disponível em `http://www.progressiveengineer.com`, acessado em 07/2011.

[Bellay & Gall, 1997] Bellay, B. & Gall, H. (1997). A comparison of four reverse engineering tools. Em *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pp. 2--11, Washington, DC, USA. IEEE Computer Society.

[Booch et al., 2005] Booch, G.; Rumbaugh, J. & Jacobson, I. (2005). *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional.

[Briand et al., 2006] Briand, L. C.; Labiche, Y. & Leduc, J. (2006). Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Trans. Softw. Eng.*, 32:642--663.

[Bruegge & Dutoit, 2009] Bruegge, B. & Dutoit, A. H. (2009). *Object-Oriented Software Engineering Using* UML*, Patterns, and* J*ava*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edição.

[Buarque, 2009] Buarque, A. S. M. (2009). Desenvolvimento de software dirigido por modelos: Um foco em engenharia de requisitos. Dissertação de Mestrado. Universidade Federal de Pernambuco.

[Buss & Henshaw, 2010] Buss, E. & Henshaw, J. (2010). A software reverse engineering experience. Em *CASCON First Decade High Impact Papers*, CASCON '10, pp. 42--60, New York, NY, USA. ACM.

[Canfora & Di Penta, 2007] Canfora, G. & Di Penta, M. (2007). New frontiers of reverse engineering. Em *2007 Future of Software Engineering*, FOSE '07, pp. 326--341, Washington, DC, USA. IEEE Computer Society.

[Chikofsky & Cross II, 1990] Chikofsky, E. J. & Cross II, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7:13--17.

[Chirilă et al., 2010] Chirilă, C.-B.; Sakkinen, M.; Lahire, P. & Jurca, I. (2010). Reverse inheritance in statically typed object-oriented programming languages. Em *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '10, pp. 5:1--5:5, New York, NY, USA. ACM.

[de Souza, 2005] de Souza, C. S. (2005). *The Semiotic Engineering of Human-Computer Interaction (Acting with Technology)*. The MIT Press.

[de Souza et al., 2010] de Souza, C. S.; Leitão, C. F.; Prates, R. O.; Amélia Bim, S. & da Silva, E. J. (2010). Can inspection methods generate valid new knowledge in HCI? the case of semiotic inspection. *Int. J. Hum.-Comput. Stud.*, 68:22--40.

[Diskin et al., 2008] Diskin, Z.; Easterbrook, S. M. & Dingel, J. (2008). Engineering associations: From models to code and back through semantics. Em *Technology of Object-Oriented Languages and Systems*, pp. 336--355.

[Dobing & Parsons, 2006] Dobing, B. & Parsons, J. (2006). How UML is used. *Commun. ACM*, 49:109--113.

[DSM, 2011] DSM (2011). Domain-specific modeling. Disponível em `http://www.dsmforum.org` acessado em 07/2011.

[Dzidek et al., 2008] Dzidek, W. J.; Arisholm, E. & Briand, L. C. (2008). A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Trans. Softw. Eng.*, 34:407--432.

[EA, 2011] EA (2011). Enterprise Architect. versão 8.0, professional. Disponível em `http://www.sparxsystems.com.au/`, acessado em 07/2011.

[Ferdinand et al., 2008] Ferdinand, C.; Heckmann, R.; Wolff, H.-J.; Renz, C.; Parshin, O. & Wilhelm, R. (2008). Model-driven development of reliable automotive services. capítulo Towards Model-Driven Development of Hard Real-Time Systems, pp. 145--160. Springer-Verlag, Berlin, Heidelberg.

[France & Rumpe, 2007] France, R. & Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. Em *2007 Future of Software Engineering*, FOSE '07, pp. 37--54, Washington, DC, USA. IEEE Computer Society.

[France et al., 2006] France, R. B.; Ghosh, S.; Dinh-Trong, T. & Solberg, A. (2006). Model-driven development using UML 2.0: Promises and pitfalls. *Computer*, 39:59--66.

[Génova et al., 2003] Génova, G.; Del Castillo, C. R. & Llorens, J. (2003). Mapping UML Associations into Java Code. *JOURNAL OF OBJECT TECHNOLOGY*, 2(5):135--162.

[Gessenharter, 2008] Gessenharter, D. (2008). Mapping the UML2 semantics of associations to a Java code generation model. Em *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pp. 813--827, Berlin, Heidelberg. Springer-Verlag.

[Gessenharter, 2009] Gessenharter, D. (2009). Implementing UML associations in Java: a slim code pattern for a complex modeling concept. Em *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*, RAOOL '09, pp. 17--24, New York, NY, USA. ACM.

[Gogolla & Kollmann, 2000] Gogolla, M. & Kollmann, R. (2000). Re-documentation of Java with UML class diagrams. Em *Proc. 7th Reengineering Forum, Reengineering Week 2000*, pp. 41--48.

[Harandi & Ning, 1990] Harandi, M. T. & Ning, J. Q. (1990). Knowledge-based program analysis. *IEEE Softw.*, 7:74--81.

[Harrison et al., 2000] Harrison, W.; Barton, C. & Raghavachari, M. (2000). Mapping UML designs to Java. Em *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pp. 178--187, New York, NY, USA. ACM.

[Hettel et al., 2008] Hettel, T.; Lawley, M. & Raymond, K. (2008). Model synchronisation: Definitions for round-trip engineering. Em *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, ICMT '08, pp. 31--45, Berlin, Heidelberg. Springer-Verlag.

[Hidaka et al., 2009] Hidaka, S.; Hu, Z.; Kato, H. & Nakano, K. (2009). A compositional approach to bidirectional model transformation. Em *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pp. 235--238.

[IEEE, 1993] IEEE (1993). Std. 1219 - IEEE standard for software maintenance.

[Issa et al., 2007] Issa, L.; Pádua, C. I. P. S.; Resende, R. F.; Viveiros, S. & de Alcântara dos Santos Neto, P. (2007). Desenvolvimento de interface com usuário dirigida por modelos com geração automática de código. Em *CIbSE*, pp. 341--354.

[Jakimi & Elkoutbi, 2009] Jakimi, A. & Elkoutbi, M. (2009). Automatic code generation from UML statechart. *International Journal of Engineering*, 1(2):165--168.

[Kegel & Steimann, 2008] Kegel, H. & Steimann, F. (2008). Systematically refactoring inheritance to delegation in Java. Em *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pp. 431--440, New York, NY, USA. ACM.

[Khaled, 2009] Khaled, L. (2009). A comparison between UML tools. Em *Environmental and Computer Science, 2009. ICECS '09. Second International Conference on*, pp. 111--114.

[Kollman et al., 2002] Kollman, R.; Selonen, P.; Stroulia, E.; Systä, T. & Zundorf, A. (2002). A study on the current state of the art in tool-supported UML-based static reverse engineering. Em *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pp. 22--32, Washington, DC, USA. IEEE Computer Society.

[Kollmann & Gogolla, 2001] Kollmann, R. & Gogolla, M. (2001). Application of UML associations and their adornments in design recovery. Em *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pp. 81--90, Washington, DC, USA. IEEE Computer Society.

[Labiche, 2009] Labiche, Y. (2009). Models in software engineering. capítulo The UML Is More Than Boxes and Lines, pp. 375--386. Springer-Verlag, Berlin, Heidelberg.

[Larman, 2008] Larman, C. (2008). *Utilizando* UML *e Padrões 3ed.* BOOKMAN COMPANHIA ED.

[Lewis & Loftus, 2007] Lewis, J. & Loftus, W. (2007). J*ava software solutions: foundations of program design.* Pearson/Addison-Wesley.

[Likert, 1932] Likert, R. (1932). A technique for the measurement of attitudes.

[Manso et al., 2003] Manso, M. E.; Genero, M. & Piattini, M. (2003). No-redundant metrics for UML class diagram structural complexity. Em *Proceedings of the 15th international conference on Advanced information systems engineering*, CAiSE'03, pp. 127--142, Berlin, Heidelberg. Springer-Verlag.

[MDA, 2011] MDA (2011). Model driven architecture. Disponível em `http://www.omg.org/mda/specs.htm` acessado em 07/2011.

[Meyer, 1997] Meyer, B. (1997). UML: The positive spin. Disponível em `http://archive.eiffel.com/doc/manuals/technology/bmarticles/uml/page.html` acessado em 07/2011.

[MIC, 2011] MIC (2011). Model-integrated computing. Institute for Software Integrated Systems. Disponível em `http://www.isis.vanderbilt.edu/research/MIC` acessado em 07/2011.

[Micskei & Waeselynck, 2010] Micskei, Z. & Waeselynck, H. (2010). The many meanings of UML 2 Sequence Diagrams: a survey. *Software and Systems Modeling*.

[Milicev, 2007] Milicev, D. (2007). On the semantics of associations and association ends in UML. *IEEE Trans. Softw. Eng.*, 33:238--251.

[Möller et al., 2008] Möller, M.; Olderog, E.-R.; Rasch, H. & Wehrheim, H. (2008). Integrating a formal method into a software engineering process with UML and Java. *Form. Asp. Comput.*, 20:161--204.

[Mueller et al., 2006] Mueller, W.; Rosti, A.; Bocchio, S.; Riccobene, E.; Scandurra, P.; Dehaene, W. & Vanderperren, Y. (2006). UML for ESL design: basic principles, tools, and applications. Em *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pp. 73--80, New York, NY, USA. ACM.

[Müller et al., 2000] Müller, H. A.; Jahnke, J. H.; Smith, D. B.; Storey, M.-A.; Tilley, S. R. & Wong, K. (2000). Reverse engineering: a roadmap. Em *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pp. 47--60, New York, NY, USA. ACM.

[Oliver & Luukala, 2006] Oliver, I. & Luukala, V. (2006). On UML's Composite Structure Diagram. Em *Fifth Workshop on System Analysis and Modelling*, Kaiserslautern, Germany.

[OMG, 2011] OMG (2011). Object Management Group. Disponível em `http://www.omg.org/` acessado em 07/2011.

[Parkinson & Bierman, 2008] Parkinson, M. J. & Bierman, G. M. (2008). Separation logic, abstraction and inheritance. Em *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, POPL '08, pp. 75--86, New York, NY, USA. ACM.

[Pastor & Molina, 2007] Pastor, O. & Molina, J. C. (2007). *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Pressman, 2006] Pressman, R. S. (2006). Engenharia de software. Editora McGraw-Hill. ISBN 8586804576, 6° edição.

[Robert. B. Stone, 2000] Robert. B. Stone, D. A. M. (2000). The touchy -feely side of engineering education: Bringing hands-on experience to classroom. ASEE Midwest Section Conference. Boston, EUA.

[RSA, 2011] RSA (2011). Rational Software Architect. versão 8.0. Disponível em `http://www-01.ibm.com/software/awdtools/architect/swarchitect`, acessado em 07/2011.

[Saff & Ernst, 2003] Saff, D. & Ernst, M. D. (2003). Reducing wasted development time via continuous testing. Em *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pp. 281--292, Washington, DC, USA. IEEE Computer Society.

[Sendall & Küster, 2004] Sendall, S. & Küster, J. M. (2004). Taming Model Round-Trip Engineering. Vancouver, Canada.

[Sensalire et al., 2009] Sensalire, M.; Ogao, P. & Telea, A. (2009). Evaluation of software visualization tools: Lessons learned. 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. p19-26.

[Sharif & Maletic, 2009] Sharif, B. & Maletic, J. (2009). An empirical study on the comprehension of stereotyped UML class diagram layouts. Em *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pp. 268--272.

[Smans et al., 2009] Smans, J.; Jacobs, B. & Piessens, F. (2009). Implicit dynamic frames: Combining dynamic frames and separation logic. Em *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pp. 148--172, Berlin, Heidelberg. Springer-Verlag.

[Sommerville, 2007] Sommerville, I. (2007). *Software Engineering*. International computer science series. Addison-Wesley.

[Stürmer et al., 2006] Stürmer, I.; Conrad, M.; Fey, I. & Dörr, H. (2006). Experiences with model and autocode reviews in model-based software development. Em *Proceedings of the 2006 international workshop on Software engineering for automotive systems*, SEAS '06, pp. 45--52, New York, NY, USA. ACM.

[Superstructure, 2011] Superstructure (2011). Superstructure - OMG Unified Modeling Language. versão 2.4. Disponível em: `http://www.omg.org/spec/UML/2.4/Superstructure/PDF/`.

[Szlenk, 2008] Szlenk, M. (2008). Balancing agility and formalism in Software Engineering. capítulo UML Static Models in Formal Approach, pp. 129--142. Springer-Verlag, Berlin, Heidelberg.

[Tempero et al., 2008] Tempero, E.; Noble, J. & Melton, H. (2008). How do Java programs use inheritance? an empirical study of inheritance in Java software. Em *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pp. 667--691, Berlin, Heidelberg. Springer-Verlag.

[TIBCO, 2011] TIBCO (2011). TIBCO: The power of now. Disponível em `http://www.tibco.com.br/`, acessado em 07/2011.

[Tonella et al., 2007] Tonella, P.; Torchiano, M.; Du Bois, B. & Systä, T. (2007). Empirical studies in reverse engineering: state of the art and future trends. *Empirical Softw. Eng.*, 12:551--571.

[UML, 2011] UML (2011). Unified Modeling OMGlanguage. Disponível em `http://www.uml.org/`, acessado em 07/2011.

[Wagelaar, 2008] Wagelaar, D. (2008). Challenges in bootstrapping a model-driven way of software development. In: ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering, pp. 25-30.

[Warth et al., 2006] Warth, A.; Stanojević, M. & Millstein, T. (2006). Statically scoped object adaptation with expanders. Em *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pp. 37--56, New York, NY, USA. ACM.

[Ziadi et al., 2003] Ziadi, T.; Hélouët, L. & marc Jézéquel, J. (2003). Towards a UML profile for software product lines. Em *In PFE*, pp. 129--139. Springer.