UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

2007.2

Geração do Projeto da Arquitetura no RUP a partir da Análise de Casos de Uso

Trabalho de Graduação

Aluno: Bruno Rodrigues de Castro Ribeiro – brcr@cin.ufpe.br

Orientador: Augusto César Alves Sampaio – acas@cin.ufpe.br

Co-orientador: Rafael Machado Duarte – rmd@cin.ufpe.br

Recife, 24 de Janeiro de 2008.

Universidade Federal de Pernambuco

Centro de Informática
Graduação em Ciência da Computação
2007.2

Bruno Rodrigues de Castro Ribeiro

Geração do Projeto da Arquitetura no RUP a partir da Análise de Casos de Uso

Trabalho de Graduação

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador – Augusto César Alves Sampaio
Co-orientador – Rafael Machado Duarte

Recife, 24 de Janeiro de 2008.

Resumo

Esse trabalho visa melhorar a eficiência do processo de análise e projeto existente no *RUP* (*Rational Unified Process*) provendo uma ferramenta capaz de oferecer suporte à integração de modelos que é realizada dentro desse processo. Para tanto, a ferramenta de software aqui implementada realiza a integração automática de modelos resultantes da análise de casos de uso originando um novo modelo que representa a arquitetura do sistema. Esta integração baseia-se em um conjunto pré-definido de regras de casamento de padrões e de integração com a finalidade de detectar os elementos que necessitam ser mesclados e definir como deve ser feita essa composição.

Além disso, a ferramenta apresenta políticas de solução de conflitos, segundo as quais eventuais conflitos decorrentes do processo de integração são devidamente tratados. Todo o processo executado pela aplicação possui a finalidade de gerar um novo diagrama de classes contendo um novo modelo integrado a partir das entradas. O último deve representar uma visão unificada das entidades que foram modeladas nos diagramas de classe das realizações dos casos de uso fornecidos.

No discorrer das seções desse documento são detalhados o funcionamento do processo de integração e as decisões tomadas ao longo do projeto de desenvolvimento da ferramenta, atualmente implementada como uma aplicação *stand-alone* e com a possibilidade de criação de versões em *plug-in* para as diversas ferramentas de modelagem de software utilizadas no mercado.

Palavras-chaves: integração de modelos, análise e projeto, *RUP*, processo de integração, metamodelos, metaclasses, regras, ferramenta, modeladem.

Abstract

This work aims to improve the efficiency of the process of Analysis and Design, which is a part of RUP (Rational Unified Process), by providing a tool capable of supporting the integration of models that is held within that process. Thus, the software tool implemented here makes the automatic integration of the resulting models from the use cases analysis creating a new model that represents the system's architecture. This integration is based on a set of pre-defined rules of matching and merging with the objective of detecting the elements that need to be merged and define how this composition should be done.

Moreover, the tool has policies for resolving conflicts, according to which any arising conflicts from the integration process are adequately treated. All this process is executed by the application with the purpose of generating a resulting class diagram containing a new integrated model origined by the entries. The latter must represent a unified vision of the entities that were modeled in the provided class diagrams that are a part of the use cases realizations.

Throughout the sections of this document are detailed how the integration process works and the decisions taken during the design and development of the tool, currently implemented as a stand-alone application and with the possibility of creating versions of plug-ins for the various software modeling tools used in the market.

Keywords: model merging, analysis and design, RUP, integration process, metamodels, metaclasses, rules, tool, modeling.

Dedicatória

Dedico esse trabalho a toda a minha família, principalmente, aos meus pais.

Agradecimentos

A meu pai e minha mãe, pelo amor, pelos exemplos, pelo apoio, pela compreensão e pela infra-estrutura que me deram, permitindo que minha instrução fosse sempre o objetivo maior a ser alcançado.

A todos os professores do Centro de Informática que através de aulas me apresentaram as várias áreas do meu curso, fornecendo uma base sólida de conhecimentos.

A todos que participaram de algum trabalho de faculdade comigo. Aos que trabalharam, pela experiência de trabalho em grupo e de convivência. Aos que não, pela noção de como lidar com isso, pois no mundo profissional também existirão casos como esses.

Ao orientador deste trabalho, pelo apoio e pelo interesse.

A Rafael Duarte, pelo suporte técnico, pelo tempo e atenção cedidos e pela indicação de listas de discussão relevantes.

A Ana Paula, Marcela e Daniele pela presença ao longo do curso em boa parte dos trabalhos (essas trabalharam) e pela amizade.

Ao Citi pela participação em dois projetos que consistiram minhas primeiras experiências de mercado.

A minha namorada Maria Gabriela, pelo carinho, pela compreensão, pelo incentivo e principalmente por acreditar que eu estava, em pleno fim de semana, trabalhando no CIn.

Sumário

Índice de Figuras

Índice de Tabelas

1. Introdução

Devido ao nível crescente de exigência do mercado por soluções de software de alta qualidade e baixos tempo e custo de desenvolvimento, as empresas de software, estão continuamente revendo seus processos de desenvolvimento de soluções. O objetivo principal dessa revisão é aprimorar cada vez mais a relação entre qualidade, custo e tempo dos produtos e serviços oferecidos.

Olhando por esse prisma, qualquer ferramenta, técnica ou conhecimento, que influencie algum dos três pontos citados acima, pode ser decisivo para manter ou melhorar a posição de uma empresa no mercado. Mercado este que a cada dia torna-se mais competitivo e inflacionado por novas organizações com grande capacidade de impor uma forte concorrência apesar da pouca experiência e conhecimento adquiridos.

No tocante ao desenvolvimento de software, muitos pontos podem ser estudados e avaliados para a melhoria de uma dessas três dimensões. O foco deste trabalho é o processo de análise e projeto. Apesar de crucial, muitas vezes os resultados deste processo não são visíveis a curto prazo pelos clientes das empresas de software, gerando uma impressão indesejada de desperdício de recursos.

Partindo do exposto, este trabalho visa automatizar a integração dos diagramas de classe originados da modelagem de casos de uso isolados, gerando o diagrama de classes da arquitetura do sistema. Apesar dessa tarefa apresentar uma sistematização, ainda se trata de um passo realizado manualmente, onde parte considerável do esforço empregado consiste apenas na inclusão, remoção e fusão de elementos dos modelos iniciais no modelo de arquitetura. Desse modo, um tempo precioso é gasto com operações perfeitamente automatizáveis, ao invés de ser utilizado na tomada de decisões de projeto importantes para a definição da arquitetura. Em se tratando de projetos de grande porte, a integração aqui proposta dos resultados da análise de casos de uso possibilitará um grande ganho de produtividade, agilizando o processo de análise e projeto. Apesar do foco ser a geração do projeto de arquitetura, esse trabalho também serve de apoio para a integração de modelos de forma mais ampla.

A seção 2 tem por finalidade apresentar o contexto que envolve a concepção da aplicação de software tratada por esse trabalho. Nele é abordado o meio que impulsionou as pesquisas para a criação e o aprimoramento das técnicas e processos de desenvolvimento de software. É mostrada uma visão geral do *RUP* [4], *framework* de processos concebido nessa época e alvo desse trabalho. Posteriormente, a disciplina de Análise e Projeto, do *framework* citado, é detalhada, situando as atividades que serão auxiliadas pelo uso da ferramenta desenvolvida. Na seção 3, são discutidos alguns dos trabalhos e das ferramentas que estão voltados para esse mesmo objetivo mostrando as diferenças entre eles.

Na seção 4, o modo como o processo de integração foi idealizado e fragmentado é descrito, detalhando o objetivo de cada parte do processo e discutindo os conceitos e pontos importantes do mesmo. Já na seção 5, são apresentadas as tecnologias adotadas, bem como os pontos e justificativas de utilização de cada uma. Posteriormente, no estudo de caso descrito

na seção 6, é mostrado um exemplo no qual a ferramenta criada foi empregada ilustrando as entradas informadas e as saídas obtidas.

Por fim, são discutidas as dificuldades encontradas na implementação, levantando-se pontos de possível extensão e trabalhos futuros para a melhoria do processo de integração, da compatibilidade com outras ferramentas de modelagem e integração da disciplina de análise e projeto e das funcionalidades oferecidas pela aplicação.

2. Contexto

Desde o surgimento dos primeiros computadores, aplicações de software têm sido criadas com diversos objetivos. Dentre eles, podem-se citar a realização de cálculos complexos em tempo hábil, a automatização de atividades repetitivas e o compartilhamento de dados e informações em geral. Inicialmente, as restrições impostas pela tecnologia existente e o início do desenvolvimento da área da informática eram obstáculos grandes a serem vencidos. Portanto, as primeiras aplicações desenvolvidas não apresentavam um grande nível de complexidade, tão pouco de tamanho.

Porém, com o passar do tempo esse contexto foi rapidamente modificado, devido à quantidade subseqüente de pesquisas e evoluções tecnológicas resultantes das primeiras. O poder computacional e a capacidade de armazenamento de informações dos computadores que eram fabricados pela indústria cresceram assustadoramente. O ramo de comunicação também sofreu grandes mudanças com o desenvolvimento de novas formas de transmissão de dados entre computadores, desde conexões via cabo entre apenas dois terminais ao advento da Internet móvel. A criação de novos protocolos de comunicação contribuiu muito para o crescimento desse ramo, trazendo consigo a possibilidade de transmissão de imagens, arquivos de vídeo e vídeo em tempo real.

Toda essa evolução no cenário onde o desenvolvimento de software está inserido causou grandes impactos na forma de produção, na complexidade das tarefas que uma aplicação deveria realizar e principalmente no tamanho da mesma. Os processos de desenvolvimento que eram utilizados já não atendiam satisfatoriamente as condições impostas pelo novo contexto. A partir desse ponto, foram impulsionadas pesquisas no setor de criação de ferramentas e de proposição de práticas, técnicas e metodologias que pudessem prover um suporte mais adequado ao processo de produção e manutenção de aplicações de software.

Na área de metodologias de desenvolvimento de software as pesquisas citadas e outras subseqüentes deram origem a dois grandes grupos bem definidos: o das metodologias ágeis e o das metodologias tradicionais. No primeiro, podem ser elencados o *Scrum* (1986) [16], o *Crystal Clear* [17] e o *Extreme Programming* (1996) [18]. Metodologias desse grupo tem como caracteríticas iterações mais curtas em seu ciclo de vida, e definições de processo mais abertas e com estapas mais resumidas. O segundo grupo, no entanto, constituido por metodologias como, o *RUP* (1998), o Modelo Espiral (1988) [19] e o *Catalysis* [20] apresentam iterações, geralmente mais longas e processos mais sistematizados para endereçar as questões do desenvolvimento de software. Como metodologias desse grupo possuem um maior enfoque na análise e projeto de sistemas e apresentando uma maior sistematização de seus processo, o

trabalho realizado se adequa melhor a elas.

2.1 Visão Geral do *RUP*

Bastante utilizado até hoje em projetos de desenvolvimento de software, o *RUP* é um *framework* iterativo de processos de desenvolvimento de software que foi criado pela *Rational Software Corporation*, atualmente parte integrante da *IBM*. Um dos pontos mais interessantes do *RUP* é que, conforme dito, ele não se trata de um único processo concreto e pré-definido, e sim de um *framework* adaptativo de processos. Sob essa perspectiva, ele foi concebido com a finalidade de ser adaptado pelas empresas de desenvolvimento e equipes de projeto. Adaptações que devem visar exclusivamente à seleção dos elementos apropriados as necessidades dessas entidades. Tais elementos são escolhidos a partir do *framework* de processos para a composição de fluxos de atividades pertinentes ao trabalho a ser realizado.

O Processo Unificado, desde o começo, foi projetado para incluir tanto um processo genérico de domínio publico quanto uma especificação mais detalhada que poderia ser vendida como um produto comercial bem acabado. Enquanto produto de processo de software o *RUP* apresenta uma base de conhecimento integrada através de *hyperlinks* contendo amostras de artefatos e descrições detalhadas dos diversos tipos de atividades das quais é composto.

Os desenvolvedores e criadores desse *framework* de processos se concentraram em diagnosticar as várias características presentes em diversos tipos diferentes de projetos de software que não obtiveram o êxito esperado. Por meio dessa abordagem, os pesquisadores tentaram identificar as principais causas desses fracassos. Uma vez identificadas, foram pesquisadas soluções para esses sintomas nos processos de engenharia de software existentes na época. Abaixo é apresentada uma lista representativa dos motivos de fracasso dos projetos pesquisados:

- Gerenciamento de requisitos ambíguo e impreciso
- Comunicação ambígua e imprecisa
- Arquitetura não funciona adequadamente sob estresse
- Complexidade extrema
- Inconsistências não detectadas nos requisitos, no projeto e na implementação.
- Quantidade insuficiente de testes
- Avaliação subjetiva do estado do projeto
- Falha no combate aos riscos
- Propagação de mudanças não controlada

- Automação insuficiente

A falha de um projeto é causada por uma combinação de vários fatores, embora cada projeto falhe de maneira única. O resultado do trabalho realizado pelos pesquisadores foi um sistema das melhores práticas de software ao qual eles atribuíram o nome de *Rational Unified Process*. O Processo foi concebido pelas mesmas técnicas que a equipe utilizava para projetar software, razão pela qual ele possui internamente um modelo orientado a objetos em *UML* (*Unified Modeling Language*) [10].

O ciclo de vida do *RUP* é uma implementação do modelo espiral. Ela foi criada pela montagem dos elementos constituintes em seqüências semi-ordenadas. Conseqüentemente, o ciclo de vida do *RUP* está representado como uma *WBS* (*Work Breakdown Structure*), possibilitando desse modo a personalização a fim de atender as necessidades específicas de um projeto. O ciclo de vida do *RUP* organiza as tarefas em fases e iterações. Um projeto tem quatro fases: Concepção, Elaboração, Construção, Transição.

Na fase inicial de Concepção, é estabelecido o escopo do projeto e suas fronteiras, determinando os principais casos de uso do sistema. Esses casos de uso devem ser elaborados com a precisão necessária para se proceder estimativas de prazos e custos. As estimativas devem ser globais para o projeto como um todo e detalhadas para a fase seguinte. Assim, a ênfase nesta etapa recai sobre o planejamento e, por conseguinte, é necessário levantar requisitos do sistema e preliminarmente analisá-los. Ao término dessa fase, são examinados os objetivos do projeto para se decidir sobre a continuidade do desenvolvimento.

O propósito da fase de Elaboração é analisar de maneira mais refinada o domínio do problema, estabelecer uma arquitetura de fundação sólida, desenvolver um plano de projeto para o sistema a ser construído e eliminar os elementos de projeto que oferecem maior risco. Embora o processo deva sempre acomodar alterações, as atividades da fase de elaboração asseguram que os requisitos, a arquitetura e os planos estão suficientemente estáveis e que os riscos estão suficientemente mitigados, de modo a se poderem prever com precisão os custos e prazos para a conclusão do desenvolvimento.

Durante a fase de construção, um produto completo é desenvolvido de maneira iterativa e incremental, a fim de estar pronto para a transição à comunidade usuária. Nesta fase, o foco principal vai para o desenvolvimento de componentes e outras características do sistema a ser projetado. Esta é a fase na qual a maior parte da codificação ocorre. Em grandes projetos, várias iterações de construção devem ser desenvolvidas em um esforço para dividir os casos de uso em segmentos gerenciáveis, com a finalidade de produzir protótipos demonstráveis.

Na fase de transição, o software é disponibilizado à comunidade usuária. Após o produto ter sido colocado em uso, naturalmente surgem novas considerações que vão demandar a construção de novas versões para permitir ajustes do sistema, corrigir problemas ou concluir algumas características que foram postergadas. É importante realçar que dentro de cada fase, um conjunto de iterações, envolvendo planejamento, levantamento de requisitos, análise, projeto e implementação e testes, é realizado. Contudo, de uma iteração para outra e

de uma fase para a próxima, a ênfase sobre as várias atividades muda, como pode ser visto na próxima figura.

Na fase de concepção, o foco principal recai sobre o entendimento dos requisitos e a determinação do escopo do projeto (planejamento e levantamento de requisitos). Na fase de elaboração, o enfoque está na captura e modelagem dos requisitos (levantamento de requisitos e análise), ainda que algum trabalho relativo às atividades de projeto e implementação seja realizado para prototipar a arquitetura, com o objetivo de evitar certos riscos técnicos. Na fase de construção, o enfoque concentra-se no projeto e na implementação, visando evoluir e rechear o protótipo inicial, até obter o primeiro produto operacional. Finalmente, a fase de transição concentra-se nos testes, visando garantir que o sistema possui o nível adequado de qualidade. Além disso, usuários devem ser treinados, características ajustadas e elementos esquecidos adicionados.

O *RUP* é baseado em um conjunto de blocos, ou elementos constituintes, descrevendo o que deve ser produzido, as competências necessárias e as instruções passo a passo descrevendo como objetivos específicos de desenvolvimento são atingidos. Os principais blocos, ou elementos constituintes, são: os papéis (quem), produtos de trabalho (o que) e as tarefas (como). Dentro de cada iteração, as tarefas são classificadas em nove disciplinas: Modelagem de Negócios, Requisitos, Análise e Projeto, Implementação, Teste, Implantação, Configuração e Gerenciamento de Mudanças, Gerenciamento de Projetos e Ambiente. O interesse desse trabalho é aprimorar o suporte ferramental a uma atividade específica da disciplina de Análise e Projeto do *RUP*, portanto apenas esta disciplina será detalhada para fins de contextualização.

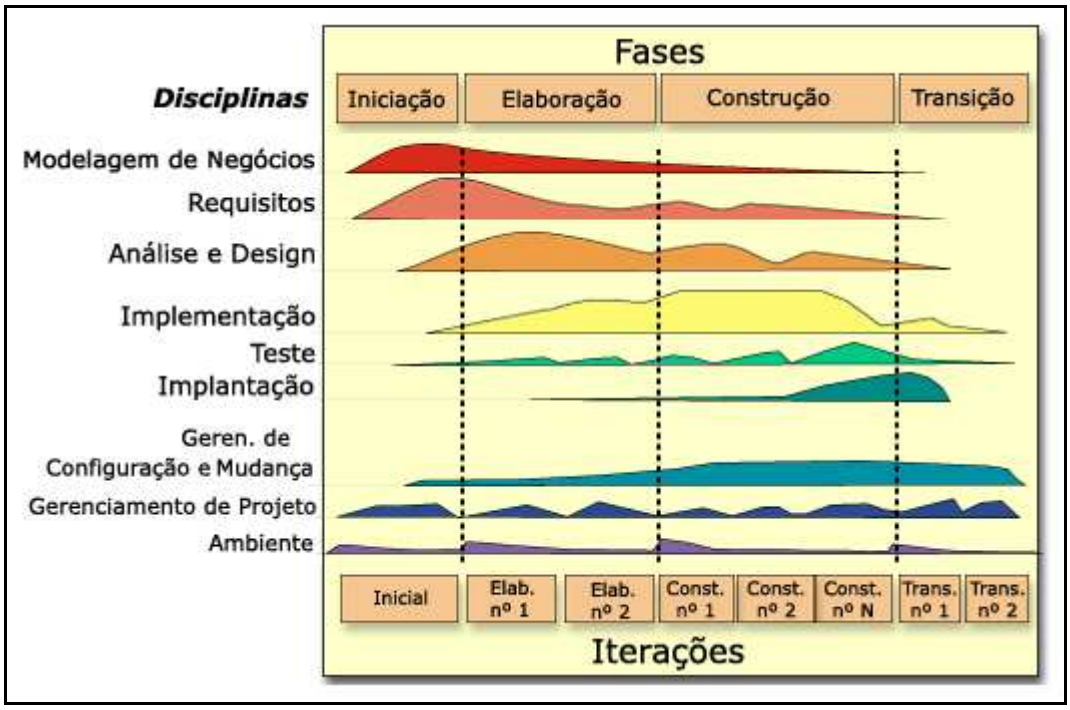

Figura 1 – Gráfico que relaciona as fases e disciplinas do *RUP* entre si.

2.2 Disciplina de Análise e Projeto

No *Rational Unified Process*, análise e projeto são combinados em uma única disciplina, embora sejam duas técnicas separadas. A investigação e a pesquisa de assuntos (análise) levam a novas descobertas e a uma melhor compreensão do domínio, e isto fornece razões e argumentos para apoiar um sistema de TI como uma solução para um problema (projeto). As fronteiras entre essas duas técnicas, freqüentemente, não são estabelecidas de forma precisa, não sendo incomum para uma mesma pessoa preencher ambos os papéis de Analista de Sistemas e de Projetista de Sistemas. No *RUP*, no entanto, o papel de Analista de Sistema desempenha um papel mais central na disciplina Requisitos do que o Projetista de Sistemas. O Analista de Software realiza a maior parte das tarefas de análise, enquanto que o Projetista de Software é responsável pelos aspectos de design.

A disciplina de Análise e Projeto desempenha um papel fundamental ao longo de todo o processo de engenharia de software e desde a fase de concepção até a fase de construção do RUP. No início do ciclo de vida, a disciplina de Análise e Projeto está preocupada com o estabelecimento de uma visão factível para o sistema e com a avaliação de técnicas adequadas para a construção da solução. O grau de risco envolvido é o responsável por determinar o nível do trabalho de análise e projeto que deve ser realizado durante a fase concepção. Desenvolver uma arquitetura inicial para o sistema é o objetivo dessa disciplina no começo da fase de elaboração, ao passo que mais tarde nessa mesma fase, o principal objetivo torna-se o refinamento da arquitetura e sua comparação com soluções anteriores equivalentes. A avaliação, a validação e o projeto de arquitetura recaem também no âmbito da disciplina de Análise e Projeto, o que enfatiza o papel desta disciplina e fato do *RUP* ser um *framework* de processos centrado em arquitetura. Tal abordagem permite às equipes de projeto concentrarem-se na qualidade da arquitetura e da solução, a fim de garantir que as necessidades sejam atendidas de forma adequada, ponto que é um dos princípios fundamentais do *RUP*.

Assim como no *RUP* de maneira geral, os modelos desempenham um papel muito importante na disciplina de Análise e Projeto. Os modelos nos permitem analisar o comportamento e a estrutura do sistema e proporcionar formas eficazes de comunicar os resultados aos interessados. O desenvolvimento guiado por modelos e a arquitetura guiada por modelos enfatizam o papel de modelos como os elementos fundamentais para a implementação de sistemas. Isso permite um aumento do nível de abstração no qual o desenvolvimento humano trabalha.

A análise de modelos evolui para o projeto de modelos e o projeto de modelos evolui ao longo de todo o ciclo de vida do projeto. Não existem regras precisas que permitam definir o momento em que se pode começar a produzir elementos de implementação e integrá-los em versões interessantes do sistema. *RUP* oferece uma série de maneiras de proceder desde o projeto até o código, as quais incluem "*Sketch e Code*" e "*Round-Trip Engineering (RTE) with Single Evolving Design Model*". Na disciplina de Análise e Projeto, o Modelo de Projeto constitui o principal artefato gerado. A próxima figura ilustra o detalhamento do fluxo pertencente à disciplina de Análise e Projeto.

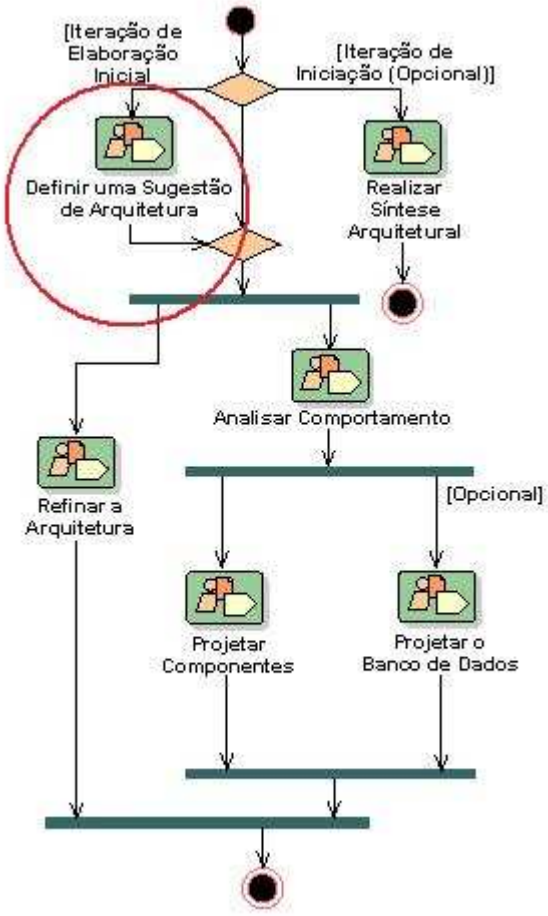

Figura 2 – Fluxo detalhado da disciplina de Análise e Projeto.

Na figura acima, o detalhe de fluxo relevante para esse trabalho é o de definição de uma arquitetura candidata que aparece circulado em vermelho. Seus principais objetivos são:

- Criar um esboço inicial da arquitetura do sistema.
- Definir um conjunto inicial de elementos significativos em termos de arquitetura que servirá como base para a análise.
- Definir um conjunto inicial de mecanismos de análise.
- Definir a disposição em camadas e organização iniciais do sistema.
- Definir as realizações de caso de uso que serão tratadas na iteração atual.
- Atualizar as realizações de caso de uso com as interações das classes de análise.

Nesse detalhamento de fluxo, a arquitetura atual é inicialmente revisada na atividade de Análise Arquitetural, onde são selecionados os casos de uso significativos em termos de arquitetura. Uma vez escolhidos, os casos de uso são submetidos à atividade de Análise de

Caso de Uso. Nela realizam-se, para cada caso de uso, tarefas como a identificação de classes de análise e distribuição de responsabilidades, visando à geração do modelo de análise e documentos relacionados que consistem na realização do caso de uso respectivo. Depois que cada caso de uso é analisado, a arquitetura é atualizada, conforme necessário, de modo que reflita as adaptações exigidas para acomodar o novo comportamento do sistema e tratar dos possíveis problemas arquiteturais identificados.

É justamente nesse ponto, onde as realizações dos casos de uso são unificadas para criar a arquitetura do sistema (figura 3), que a ferramenta aqui proposta visa oferecer suporte, realizando a parte mecanizada e repetitiva dessa tarefa e permitindo que o arquiteto de software concentre-se nos pontos mais complexos dessa operação.

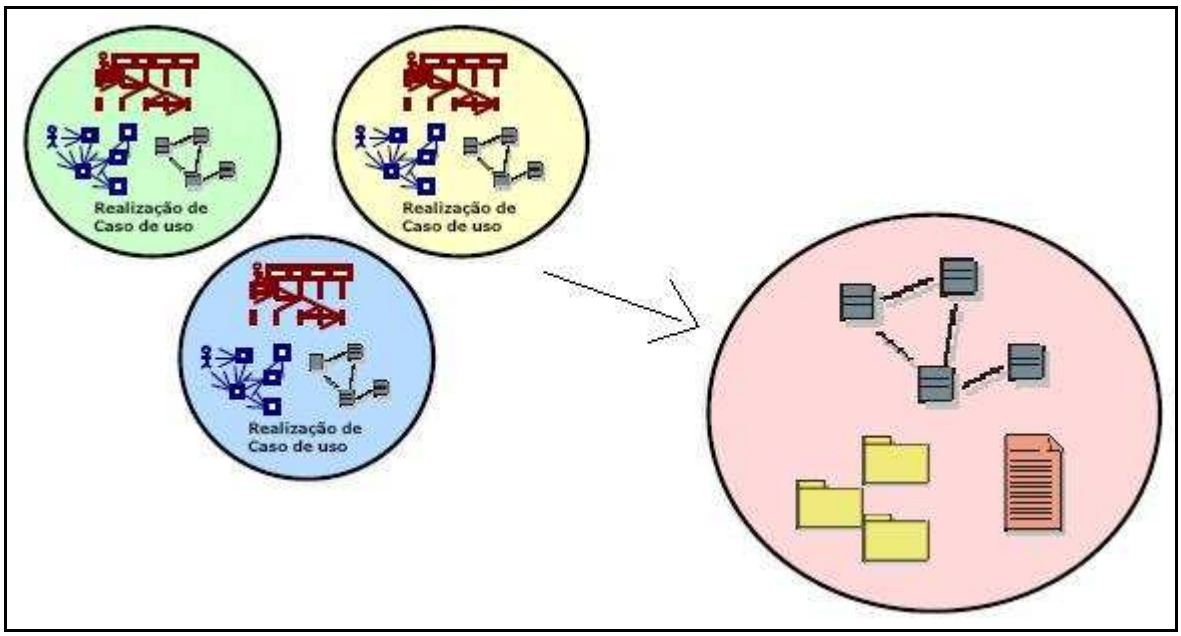

Figura 3 – Unificação das realizações dos casos de uso numa arquitetura de sistema.

3. Ferramentas e Trabalhos Relacionados

Com o objetivo de oferecer suporte e facilitar a execução de atividades pertencentes a processos de desenvolvimento de software como o *RUP*, inúmeras ferramentas são desenvolvidas a cada dia e outras tantas, já bem estabelecidas no mercado, aperfeiçoam-se com a finalidade de cada vez mais atender às necessidades do contexto para o qual foram desenvolvidas. Não apenas a quantidade, mas também a variedade de disciplinas que essas ferramentas suportam num processo de desenvolvimento de software é muito grande. Concentrando-se na disciplina de análise e projeto do *RUP*, conforme explicado na seção anterior, é possível notar uma gama de aplicações de modelagem de software que tratam das necessidades presentes nessa disciplina do processo.

Contudo, apesar de muitas dessas ferramentas de modelagem apresentarem um conjunto de funcionalidades bastante pertinentes no auxílio de grande parte das tarefas do fluxo de análise e projeto, quase nenhuma delas endereçam a questão da integração de

modelos, ponto bastante presente em tarefas do fluxo citado.

Uma das ferramentas que oferecem tal suporte é o *Rational Rose Real Time*, desenvolvido pela *Rational Software Corporation*. Trata-se de uma ferramenta de modelagem que auxilia o processo de produção de softwares ao longo do seu ciclo de vida utilizando o padrão *UML* para a representação dos modelos que suporta. Como um de seus componentes possui um integrador de modelos que permite ao usuário comparar e integrar modelos no formato *.rtmdl, é possível comparar os elementos de até sete modelos em paralelo, descobrir as divergências entre eles e integrá-los em um modelo resultante. Esse módulo integrador disponibiliza tanto uma interface gráfica (Figura 4) como uma interface de linha de comando e tem duas formas de integração uma manual e outra automática.

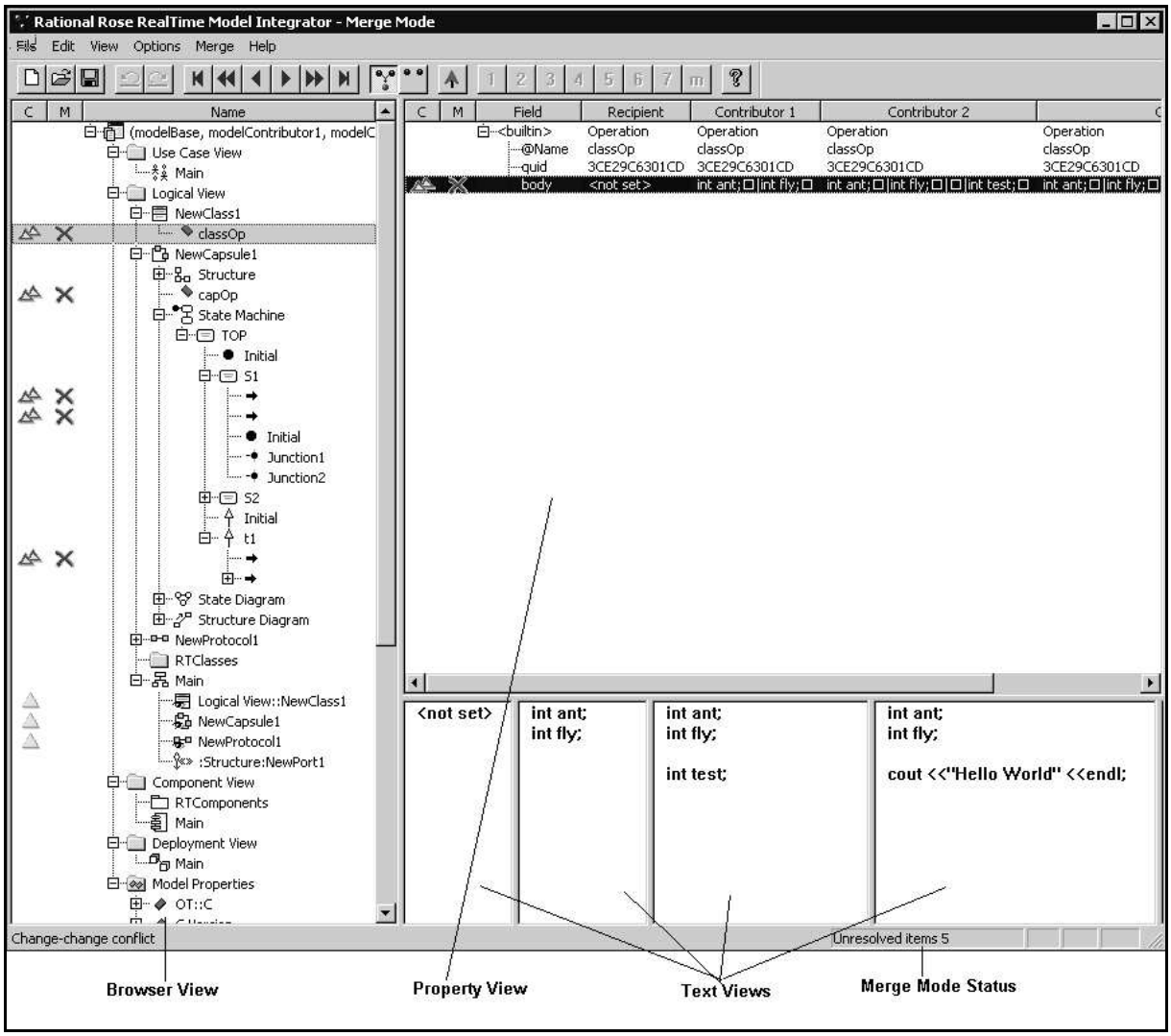

Figura 4 – *Screenshot* da tela de *Merge Mode* do integrador de modelos do Rose.

Na primeira, o integrador compara e identifica todas as diferenças existentes entre os modelos informados e o usuário decide de que maneira será resolvida cada uma dessas desigualdades. No modo automático, os modelos de entrada são comparados e um modelo resultante é gerado. Este apresenta o resultado da integração dos elementos dos modelos que

não apresentaram conflitos. Os pontos conflitantes são assinalados e o usuário deve decidir qual a forma mais adequada para a solução dos mesmos.

Pelo fato do integrador não fazer uso do mesmo nível semântico encontrado em *UML*, ele não apresenta certas funcionalidades necessárias que possibilitam uma maior eficácia e precisão do processo de integração. Apesar de ser capaz de integrar vários modelos e ser dotado de uma interface gráfica que auxilia o usuário a localizar os pontos de conflito, o algoritmo de integração de modelos utilizado pelo integrador de modelos do Rose não trabalha com nenhuma política de solução de conflitos. Dessa forma, a responsabilidade pela resolução dos conflitos originados pelo processo de integração recai sobre o usuário.

Além disso, dado que o Rose RT é uma ferramenta proprietária, ele não disponibiliza uma maneira prática de adaptação do algoritmo empregado no processo de integração. Isso impede que o usuário possa definir as regras segundo as quais os elementos de dois ou mais modelos serão integrados, ponto bastante interessante em algumas situações. É válido salientar também, que o componente integrador funciona baseado em identificadores gerados automaticamente para cada elemento pertencente ao modelo, porém esses só possuem o caráter de unicidade dentro do próprio modelo. Esse fato, apesar de não influenciar a integração de duas versões diferentes do mesmo modelo, é responsável por uma considerável queda de eficiência em se tratando da junção de dois modelos diferentes possuindo alguns elementos em comum.

Analisando os pontos apresentados sobre o componente de integração pertencente ao Rose RT, pode-se concluir que mesmo com todo o suporte oferecido por ele à junção de dois ou mais modelos, ele não atende às necessidades existentes na integração de modelos para a geração do projeto de arquitetura. As facilidades oferecidas pelo integrador são úteis e interessantes ao se considerar um contexto de gerência de modelos e controle de versões dos mesmos. Nessa situação um grupo de usuários age sobre um conjunto de modelos realizando atividades de correção ou extensão, gerando novas versões dos mesmos arquivos de modelos armazenados em um repositório comum.

Um trabalho bastante interessante e pertinente à área de integração de modelos, pode ser encontrado em [8]. Trata-se de um artigo sobre um *framework* para integração de modelos baseado nos relacionamentos existentes entre os modelos a serem fundidos. Nele há um grande foco na definição de maneira explicita dos relacionamentos entre os modelos com a finalidade de prover as informações necessárias para o processo de integração realizado pelo *framework* descrito no artigo. A estrutura geral desse *framework* está ilustrada na figura abaixo.

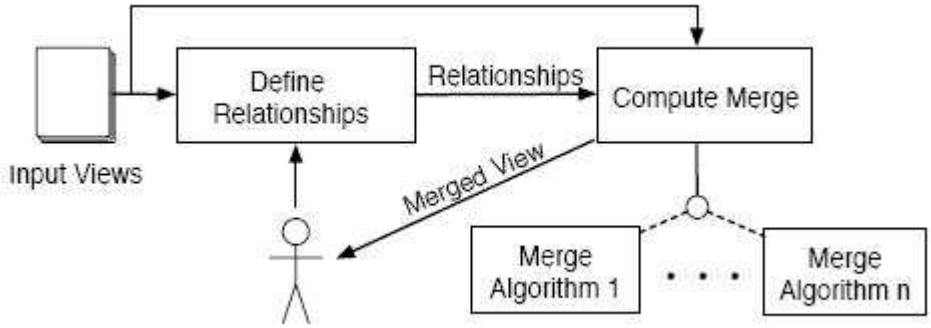

Figura 5 – Visão geral do *framework* de integração baseado nas relações entre modelos.

Seu funcionamento ocorre da seguinte forma: inicialmente, um conjunto de modelos é escolhido para ser integrado. Os usuários podem definir diferentes tipos de relações entre eles; e para cada tipo de relação, pode-se explorar maneiras alternativas de mapear os modelos entre si. Esses modelos juntamente com um conjunto selecionado de relacionamentos são então usados para computar uma integração, por meio da aplicação de um algoritmo de integração adequado. O resultado obtido é então apresentado aos usuários para uma análise mais aprofundada, a qual pode levar à descoberta de novos relacionamentos entre modelos ou a nulidade de alguns dos mapeamentos existentes. Os usuários poderão então dar inicio a uma nova iteração através da revisão das relações estabelecidas e da execução das atividades subseqüentes.

Os modelos utilizados nesse trabalho para ilustrar sua integração apresentam proporções muito pequenas, tornando-se relativamente fácil identificar as relações existentes entre eles por meio da realização manual de mapeamentos. Para grandes modelos, no entanto, esses relacionamentos são mais difíceis de serem identificados manualmente. Isso requer o desenvolvimento de técnicas adequadas de casamento entre elementos de modelos diferentes para encontrar correspondências entre eles.

Portanto, mesmo apresentando uma abordagem mais consistente que a utilizada pelo módulo integrador do Rose RT, discutido anteriormente, esse *framework* baseado em relacionamentos entre os modelos ainda apresenta uma grande necessidade de envolvimento do usuário ao longo do processo de integração. Tomando como referência o contexto atual do mercado de software, a obrigatoriedade de definição de todos os relacionamentos entre os modelos a serem integrados durante o processo de integração constitui um ponto crítico. Esse fato é facilmente constatado devido a crescente demanda de realização de projetos de desenvolvimento de software por diferentes setores do mercado e para os mais variados fins.

Aliado a isso, pode-se acrescentar o aumento do escopo desses projetos em função do surgimento de novas tecnologias e da facilidade de aquisição das mesmas, influenciando de forma considerável o tamanho e a complexidade dos softwares entregues na conclusão desses projetos. Tendo em vista essa situação, o tempo gasto e a dificuldade encontrada na definição dos relacionamentos entre modelos de um projeto de grande porte, certamente, prejudicará a parcela de tempo e esforço ganhos pelo emprego do *framework* citado.

A ferramenta proposta nesse trabalho apresenta um processo de integração de modelos desenvolvido usando uma linguagem criada para trabalhar com modelos e metamodelos. Por se tratar de uma linguagem voltada para o contexto em questão, a possibilidade de extensão do algoritmo de integração, para incorporar gradualmente o processamento de metadados, tornou-se mais fácil. Tais metadados seriam relativos ao comportamento dos elementos contidos nos modelos de entrada. A partir dessas informações, seria viável uma comparação e junção de elementos que tomasse em consideração as características comportamentais de cada componente dos modelos. Com isso, a qualidade dos resultados obtidos do processo de integração poderia ser aumentada substancialmente, reduzindo ainda mais os possíveis ajustes finais por parte do usuário da ferramenta.

Outra vantagem apresentada é a utilização de regras de casamento e de integração agrupadas e definidas em arquivos. Devido ao carregamento das regras no momento da execução a partir dos arquivos, podem-se selecionar quais regras utilizadas pela presença ou ausência delas nesses arquivos. Dessa maneira, a ferramenta dispõe de um modo prático de alterar o funcionamento do algoritmo de integração sem a necessidade de manipulação de código fonte. Menos importante do ponto de vista de integração de modelos, porém bastante útil é o fato do software desenvolvido suportar mais de um formato de arquivo, tanto para a entrada quanto para a saída. Sendo um dos formatos suportados expresso em *XMI* (*XML Metadata Interchange*) [3], a ferramenta consegue disponibilizar um arquivo de padrão aberto e bem conhecido pelo mercado, inclusive tratando-se de um formato suportado por várias ferramentas de modelagem na operação de exportar de modelos.

4. Definição do Processo de Integração

O processo que consiste na junção de dois modelos distintos originando como resultado um novo modelo, de maneira a não haver repetição de elementos ou estruturas representantes de um mesmo conceito, constitui a atividade complexa denominada de processo de integração. Nele está presente uma série de tarefas cuja total automatização representa um objetivo não trivial a ser alcançado. Os principais componentes desse conjunto de atividades podem ser listados da seguinte forma:

- Estabelecimento de correspondências entre elementos.
- Validação de conformidade dos elementos correspondentes.
- Verificação da necessidade de aplicação de transformações.
- Verificação da aplicabilidade de transformações a elementos.

Os dois primeiros pontos da lista acima são tratados ao longo da avaliação de dois elementos de modelos diferentes ao final da qual se define a existência ou não de uma correspondência e em se havendo ambos devem ser integrados em um único elemento no modelo final. Os itens restantes ocorrem, efetivamente, durante a integração dos elementos que já foram avaliados na porção inicial do processo e definem as transformações a serem empregadas a cada par de elementos equivalentes.

Em vista disso, a definição do funcionamento desse processo precisa ser realizada com bastante cautela e focando questões como a modularização, a adaptabilidade e a estensibilidade. A presença de tais características favorece a manutenção e a configuração adequada do processo com a finalidade de torná-lo apto a ser empregado em diversos cenários e, portanto mais interessante.

4.1 Estratégia de Integração

Como a implementação da ferramenta discutida nesse trabalho utilizou um *framework* para composição de modelos desenvolvido em *KerMeta* [21] a estratégia de integração empregada foi a mesma já existente no *framework* usado. Nela o processo de integração se divide em duas grandes etapas. A primeira, chamada de Etapa de Casamento, fica responsável pela parte inicial do processo, realizando avaliações e estabelecendo relações de correspondência entre elementos. Já a Etapa de Interação cuida da seleção das transformações que serão aplicadas a cada elemento e de sua efetiva aplicação, montando a partir dos resultados parciais o modelo resultante.

Juntamente com a fragmentação do processo em duas etapas, o processamento de modelos é realizado aos pares para evitar o aumento da complexidade da operação de integração, pois o crescimento do número de modelos implicaria maiores dificuldades para definir as correspondências entre os elementos envolvidos no processamento. Apesar da limitação, resultados obtidos a partir desse tipo de situação são passíveis de serem alcançados

através de iterações consecutivas, onde dois modelos do conjunto alvo seriam informados na primeira iteração e as demais apresentariam como entradas o resultado da integração anterior e um novo modelo do conjunto alvo respectivamente.

Assim sendo, o primeiro passo do processo é a leitura dos modelos de entrada, que é realizado pelo módulo carregador de modelos. Este tem como função ler arquivos no formato *.km (contendo o modelo representado segundo o padrão *XMI*) e obtêm deles os pacotes *KerMeta* correspondentes. A partir desse ponto, os pacotes são então usados como entrada para o módulo integrador. Em seguida o integrador comunica-se com o verificador de casamento com a finalidade de saber quais elementos apresentam correspondências entre si, para que então ele possa decidir quais desses elementos serão integrados ou copiados para o modelo resultante sem sofrer alterações.

De modo a ser capaz de verificar a correspondência entre os elementos informados pelo integrador, o verificador de casamento baseia-se nas regras de correspondência geradas a partir de arquivos, pelo carregador de regras. Para realizar a integração, as regras de integração também são carregadas pelo carregador de regras da mesma forma que as de casamento. Nesse ponto o carregador de regras realiza um papel importante para a flexibilidade do processo de integração, promovendo o desacoplamento das regras e, por conseguinte facilitando a sua substituição.

4.2 Etapa de casamento

Nesta etapa ocorre o processo de estabelecimento de correspondências. O último é baseado em um conjunto de regras de correspondência que definem quando elementos pertencentes aos modelos de entrada serão considerados o mesmo e, portanto, integrados. As regras de casamento são escritas em arquivos de texto, de acordo com uma sintaxe muito simples. Elas consistem em fórmulas booleanas usando predicados pré-definidos relacionados com as propriedades das metaclasses envolvidas.

Quando o processo de verificação de correspondência é realizado, um motor denominado *Sintaks* é usado para converter os arquivos de texto contendo as regras em um modelo. Essa transformação exige um arquivo do tipo *.sts, que é um arquivo de mapeamento responsável por estabelecer uma ponte entre o arquivo de texto (expresso em gramática concreta) e o modelo (expresso em gramática abstrata); e um metamodelo para o modelo resultante. Os modelos representam as fórmulas booleanas que são avaliadas pelo verificador de correspondências para verificar se os elementos se correspondem ou não.

Para evitar a criação de modelos inválidos, o verificador de casamentos é dotado de um mecanismo de validação. Quando dois elementos apresentam o mesmo nome, porém não são coincidentes, a integração é considerada inválida e o processo é prontamente interrompido. Esse recurso evita a criação de um modelo contendo elementos com o mesmo identificador. Tal mecanismo apesar de simples, permite uma economia de tempo considerável analisando-se um cenário onde cada modelo de entrada apresenta uma grande quantidade de classes representadas nele.

A maioria dos predicados de correspondência útil já está presente no *framework* que foi utilizado para o desenvolvimento do processo de integração. Entretanto, também é possível adicionar alguns novos predicados caso necessário. Para isso, é preciso apenas adicionar o predicado desejado a gramática do motor *Sintaks* e fornecer o código para avaliá-lo no metamodelo de regras de correspondência.

4.2.1 Regras de casamento

As regras de casamento são aquelas utilizadas para determinar se dois elementos de modelos diferentes deveriam ser integrados em um único elemento. Existem regras específicas para cada uma das metaclasses: Classe, Propriedade, Operação. A seguir são apresentados alguns exemplos de regras disponíveis separadas por metaclasse:

- Regras de Classe:
  - rule r1 bothAbstract
  - rule r2 conformingSuperclasses
- Regras de Propriedade:
  - rule r1 and(sameType,containedMultiplicity)
- Regras de Operação:
  - rule r1 bothAbstract
  - rule r2 sameReturnType
  - rule r3 sameParameters

Cada uma dessas regras avalia as características relativas aos seus predicados de dois elementos da metaclasse a qual essa regra pertence. A primeira regra, nesse caso, avalia elementos que sejam classes, verificando se ambos são classes abstratas. Já a regra de propriedade citada, avalia os atributos de duas classes de acordo com seus tipo e multiplicidade. As últimas três regras, por sua vez, avaliam as operações das classes verificando se as primeiras são abstratas, apresentam o mesmo tipo de retorno ou os mesmos parâmetros respectivamente. O resultado dessas avaliações é repassado ao integrador para a seleção da regra de integração a ser aplicada em cada um desses casos.

4.3 Etapa de integração

A literatura relacionada à integração de modelos apresenta várias formas diferentes de abordagem para a etapa de integração propriamente dita. Dentre essas abordagens é possível mencionar algoritmos de integração baseados em grafos e processos interativos de junção de modelos. Contudo, a maioria das soluções propostas apresenta alguma desvantagem, como a impossibilidade de adaptação ou extensão da lógica presente nos algoritmos empregados. Na etapa de integração aqui descrita, o embasamento do processo de integração em arquivos de regras resolve esse problema.

Iniciada quando o algoritmo de integração recebe como entrada dois modelos (em última análise pacotes *KerMeta*), ela é a etapa na qual ocorre a aplicação das transformações necessárias para a integração dos elementos. Cada modelo é submetido à etapa de casamento que verifica a correspondência entre as classes modeladas neles. Aquelas que possuem o mesmo nome e correspondência são integradas, conseqüentemente os seus membros internos também são integrados, isto é, propriedades e operações que compõem as classes. Por sua vez, aquelas classes que não apresentam uma correspondência de nomenclatura são copiadas diretamente para o modelo resultante (integração por união).

Os detalhes mais aprofundados dessa etapa bem como do processo em sua totalidade serão detalhados posteriormente na seção de implementação, onde uma figura ilustra a visão geral de forma mais clara. O mesmo processo é usado também para integrar os componentes de cada elemento, ou seja, as propriedades e operações dos mesmos.

4.3.1 Regras de integração

As regras de integração são utilizadas para definir o resultado da integração entre dois elementos que foram definidos como sendo correspondentes pela avaliação do verificador de casamentos. Existem regras de integração específicas para cada uma das seguintes metaclasses:

- Classes
- Propriedades
- Operações

Além delas, foram concebidas mais três regras gerais para a etapa de integração dos modelos. Elas são usadas para definir qual será o resultado da integração de elementos correspondentes dos dois modelos. Os valores de todas as propriedades presentes nas metaclasses dos elementos integrados são definidos por essas regras, a saber:

- *Keep left*: definições presentes no primeiro modelo sobrepõem-se as presentes no segundo modelo informado.
- *Keep right*: definições presentes no segundo modelo sobrepõem-se as presentes no primeiro modelo informado.
- *Custom*: cada elemento será integrado em conformidade com as regras específicas que regem a sua categoria.

À primeira vista, regras como manter o que está representado no primeiro modelo ou no segundo parecem ser um pouco ingênuas, mas, na verdade, elas podem ser realmente úteis. Sabe-se que às vezes um modelo deve sobrepor-se a outro modelo, e escolhendo uma simples regra, como uma dessas, nestes casos pode poupar certo tempo que seria gasto em configurações refinadas do algoritmo de integração sem a real necessidade para tanto.

As regras de integração são carregadas a partir de arquivos *XMI*, que são convertidos a partir de arquivos texto utilizando o motor *Sintaks*, em um processo muito semelhante ao que foi realizado para o carregamento da regras de casamento. Se a regra escolhida for a regra

Custom, então, uma regra para cada uma das propriedades da metaclasse deve ser especificada. Algumas das possíveis regras personalizadas são mostradas na Tabela 1. O processo de integração inclui também os corpos das operações, mas neste momento, existe apenas uma regra personalizada, que compõe sequencialmente ambos.

Tabela 1 – Regras de integração.

Classes	Propriedades	Operações
keepConcreteClass	*keepMoreGenericType*	*keepBiggerParameterList*
keepSuperClassUnion	*keepBiggerMultiplicity*	*keepExceptionsUnion*
	keepReference	

4.3.2 Políticas de solução de conflitos

Durante um processo de integração de modelos, quando dois artefatos de software (cada um dos quais pertencendo a um modelo diferente) são supostamente equivalentes, um deles deve ser ignorado. Suas diferenças sintáticas causam, no entanto, dúvidas sobre qual deverá ser a estrutura sintática apresentada no elemento resultante da integração dos dois primeiros. É nesse ponto que a política de solução de conflito é empregada no processo de junção de ambos.

Uma política de solução de conflitos consiste, nesse caso, numa maneira de conferir prioridade a um dos modelos integrados em função de um ou mais fatores. Desse modo, quando o processo de integração encontra situações onde a definição do elemento resultante a ser salvo é dúbia, a relação de prioridade estabelecida entre os modelos pela política de solução de conflitos seleciona o devido elemento.

A implementação atual da ferramenta emprega uma política muito simples. Essa política é orientada pela ordem de informação dos modelos ao integrador combinada a seleção de regras carregadas pelo módulo carregador de regras. De fato, as regras utilizadas aqui são as mesmas três regras de integração que apresentam escopo geral de funcionamento. Elas foram aproveitadas na composição da política de solução de conflitos porque são capazes de definir uma ordem de preferência em relação aos modelos de entrada.

Um efeito colateral positivo dessa abordagem consiste na inclusão das características de adaptação e extensão à definição da política de solução de conflitos adotada pela aplicação. Assim, também é permitido ajustar a resolução de conflitos conforme as necessidades do cenário encontrado, aumentando assim a robustez da ferramenta.

5. Implementação

A ferramenta de software que esse trabalho desenvolveu tem como objetivo aprimorar uma etapa do processo de análise e projeto da metodologia *RUP* (*Rational Unified Processs*). Dado que o *RUP* é uma metodologia de desenvolvimento de software genérica e, portanto desvinculada de qualquer ferramenta existente no mercado, optou-se pela criação de uma ferramenta, inicialmente, *stand-alone*.

Outra questão importante, considerada durante a implementação, é o padrão utilizado para descrever as entradas recebidas pela aplicação e as saídas geradas por ela. Novamente, devido ao contexto genérico que envolve a aplicação fez-se uso do *XMI* (*XML Metadata Interchange*) como padrão, visto que, ele está bem estabelecido no mercado.

Contudo, a fim de tornar a ferramenta mais interessante e apta a futuramente gerar um *plug-in* para uma aplicação de modelagem de software, o formato *.rtmdl também é suportado como entrada e como saída da ferramenta. Esse formato foi escolhido por ser o padrão empregado pelo *Rational Rose Real Time* que é uma ferramenta de modelagem de software bastante conhecida e utilizada no mercado.

A ferramenta desenvolvida aqui, conforme citado na seção 4, utilizou-se de um *framework* existente de composição de modelos implementado em *KerMeta*, linguagem que será abordada em um dos tópicos dessa seção. Esse *framework* trabalha com arquivos de entrada e de saída representados em *XMI*, sendo capaz de integrá-los por meio da aplicação de regras de casamento e de integração realizadas por dois módulos distintos: o verificador de casamentos e o integrador. Dessa forma, o trabalho realizado consistiu na criação de um módulo capaz de:

- Converter e desconverter arquivos de *.rtmdl para *.km (padrão utilizado pelo *framework*).
- Acionar a rotina do *framework* que integra modelos no formato *.km, informando os modelos de entrada convertidos.
- Oferecer uma interface gráfica simples.

Para que pudesse ser realizada a conversão dos tipos contidos em arquivos *.rtmdl em tipos válidos para a linguagem *KerMeta*, um arquivo com definições de tipos foi adicionado ao *framework*, conforme será detalhado no tópico 5.3.1. Para tanto, a ferramenta foi desenvolvida segundo a arquitetura presente na figura 6. Nela o pacote modelMerging contém o *framework* utilizado, sendo este acessado pelo "Invocador", que utiliza as bibliotecas em "lib" e o arquivo de configuração contido em "conf" para realizar essa tarefa. Caso necessário o conversor e o desconversor são chamados com a finalidade de processar algum arquivo de entrada ou de saída. Ambos usam as classes agrupadas no pacote "estruturaModelo" para criar em memória um objeto representante do modelo que é processado. As classes contidas nesse pacote são detalhadas, ainda nessa seção, pela figura 9.

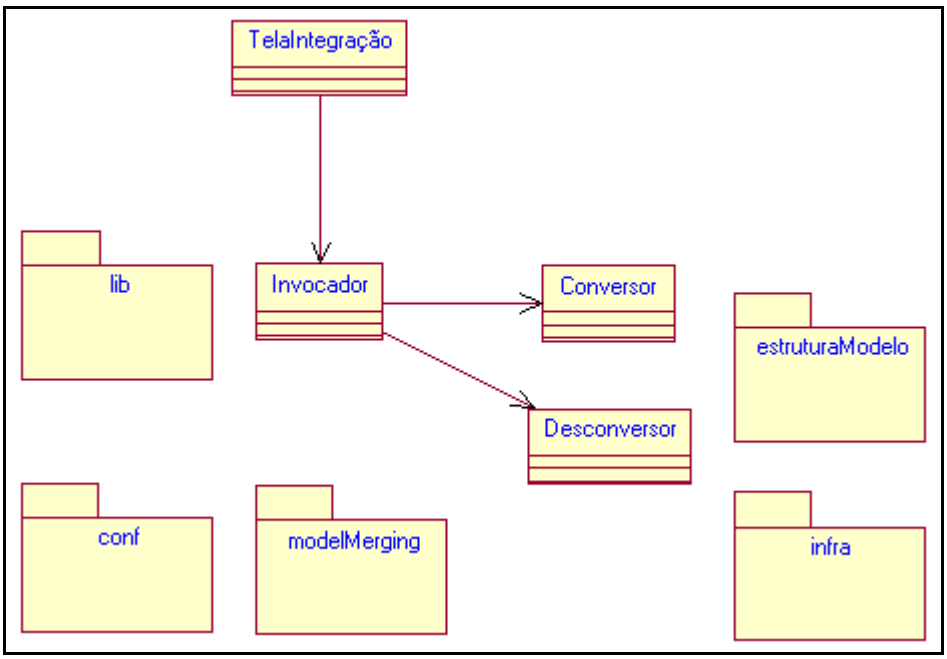

Figura 6 – Visão geral da arquitetura da ferramenta.

Com isso, a realização de operações de integração é facilitada pelo provimento de uma interface gráfica simples, pela possibilidade de se trabalhar com modelos de entrada gerados através de ferramenta (nesse casso, o Rose RT), ao invés de codificados e pela representação gráfica que as saídas apresentam, permitindo uma melhor visualização dos resultados da integração. Além disso, a escolha pelo formato *.rtmdl possibilita a integração com o Rose RT, que oferece suporte ao fluxo de análise e projeto, mas não possui uma funcionalidade de integração como a oferecida pela ferramenta aqui exposta. Para realizar a implementação dessa ferramenta foram empregadas três tecnologias distintas: as linguagens de programação *Java* e *KerMeta* e o padrão *XMI*. A seguir, serão discutidos os pontos, as formas e os motivos de utilização de cada uma dessas tecnologias.

5.1 *XMI* (*XML Metadata Interchange*)

O *XMI* é um padrão da *OMG* (*Object Management Group*) [9] para troca de informações. Por ser baseado em *XML*, ele apresenta o mesmo caráter de estruturação que facilita seu processamento por parte de aplicações. Além disso, a flexibilidade para a definição de novas *tags* é bastante útil para possíveis adições de novas informações tanto nos arquivos de entrada como nos arquivos de saída da ferramenta tratada nesse trabalho.

É válido lembrar que atualmente o uso mais comum desse padrão é na troca facilitada de metadados entre ferramentas de modelagem (baseadas no *UML*, também da *OMG*). Esse fato é importante, pois além de propiciar futuras integrações com diversas das ferramentas de modelagem utilizadas no mercado, também permite a inclusão de informações sobre o comportamento das entidades e componentes modelados. A presença de tais metadados pode ser explorada pelo processo de integração na melhoria dos resultados, pois as entidades a serem integradas teriam seus comportamentos considerados durante esse processo.

Na ferramenta desenvolvida neste trabalho o padrão *XMI* é empregado nos arquivos de entrada e saída no formato *.km (formato adotado como padrão da aplicação criada). Seu principal objetivo é prover uma forma estruturada de representação para as informações contidas nos diagramas a serem integrados. A figura abaixo mostra o trecho de um arquivo de entrada no formato citado.

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:kermeta_language_behavior="http://kermeta/kermeta.ecore//language/beh
         xmlns:kermeta_language_structure="http://kermeta/kermeta.ecore//language/st
<kermeta_language_structure:Package tag="/1 /2" name="Transito">
    <ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition" name=
       <ownedAttribute name="pessoa" type="/0/Transportar/pessoa/@containedType.0" i
          <containedType xsi:type="kermeta_language_structure:Class">
             <typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
                             href="platform:/plugin/fr.irisa.triskell.kermeta/lib/fr
          </containedType>
       </ownedAttribute>
       <ownedAttribute name="onibus" type="/0/Transportar/onibus/@containedType.0" i
          <containedType xsi:type="kermeta_language_structure:Class">
             <typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
                             href="platform:/plugin/fr.irisa.triskell.kermeta/lib/fr
          </containedType>
       </ownedAttribute>
       <ownedOperation name="pegarOnibus" type="/0/Transportar/pegarOnibus/@containe
          <containedType xsi:type="kermeta_language_structure:Class">
             <typeDefinition xsi:type="kermeta_language_structure:ClassDefinition" h
          </containedType>
          <ownedParameter name="pessoa" type="/0/Transportar/pegarOnibus/@ownedParam
             <containedType xsi:type="kermeta_language_structure:Class">
                <typeDefinition xsi:type="kermeta_language_structure:ClassDefinitior
                                href="platform:/plugin/fr.irisa.triskell.kermeta/lib
             </containedType>
          </ownedParameter>
       </ownedOperation>
    </ownedTypeDefinition>
             .
             .
             .
</kermeta_language_structure:Package>
<kermeta_language_structure:Tag name="mainClass" value="root_package::Main" object="
<kermeta_language_structure:Tag name="mainOperation" value="main" object="/0"/>
</xmi:XMI>
```

Figura 7 – Arquivo de entrada Transito.km.

O arquivo acima apresenta sete *tags* básicas que são utilizadas na representação das entidades presentes num diagrama de classes expresso no formato *.km. São elas:

1. *<kermeta_language_structure:Package>* : representa o diagrama de classe em si (no exemplo acima, Transito). Encerra as representações de todos os elementos modelados no diagrama. Visto que, cada arquivo *.km representa um diagrama de classes, só pode haver uma *tag* desse tipo por arquivo.

2. <*ownedTypeDefinition*> : representa uma classe modelada no diagrama. Encerra as representações de seus atributos e operações.
3. <*ownedAttribute*> : representa um atributo de uma classe.
4. <*ownedOperation*> : representa uma operação de uma classe. Contém as representações de seus parâmetros de entrada.
5. <*ownedParameter*> : representa uma entrada de uma operação.
6. <*containedType*> : *tag* utilizada para definir tipos de elementos em geral.
7. <*typeDefinition*> : *tag* auxiliar empregada em conjunto com a anterior que permite referenciar tipos pré-definidos (geralmente primitivos).

Assim como os arquivos padrões de entrada, os de saída possuem o mesmo formato e estrutura e, portanto não serão exemplificados aqui.

5.2 Java

Linguagem de programação desenvolvida e mantida pela *Sun*, *Java* foi escolhida como uma das linguagens para a implementação desse trabalho. Isso ocorreu devido a três pontos importantes:

- Suprir a deficiência de interface gráfica de *KerMeta*.
- Possibilidade de integração com *KerMeta*.
- Facilidade oferecida no processamento de arquivos.

5.2.1 Interface Gráfica

Como a linguagem *KerMeta* não oferece suporte ao desenvolvimento de interfaces gráficas, essa porção da aplicação foi desenvolvida em *Java*, por razões que serão detalhadas no próximo sub-tópico. Por simplicidade a interface da ferramenta é constituída por uma tela com três campos e o botão "Gerar", conforme a figura 8.

Os três campos servem para informar os caminhos absolutos dos arquivos que serão integrados e do arquivo resultante do processo de integração. Todos esses campos admitem arquivos nos formatos *.km e *.rtmdl. Vale salientar que a ordem dos arquivos de entrada pode alterar o resultado obtido, porque em caso de conflito o processo de integração realiza ações diferentes baseadas na prioridade de cada modelo. Na configuração atual o primeiro modelo informado tem prioridade maior em situações de conflito.

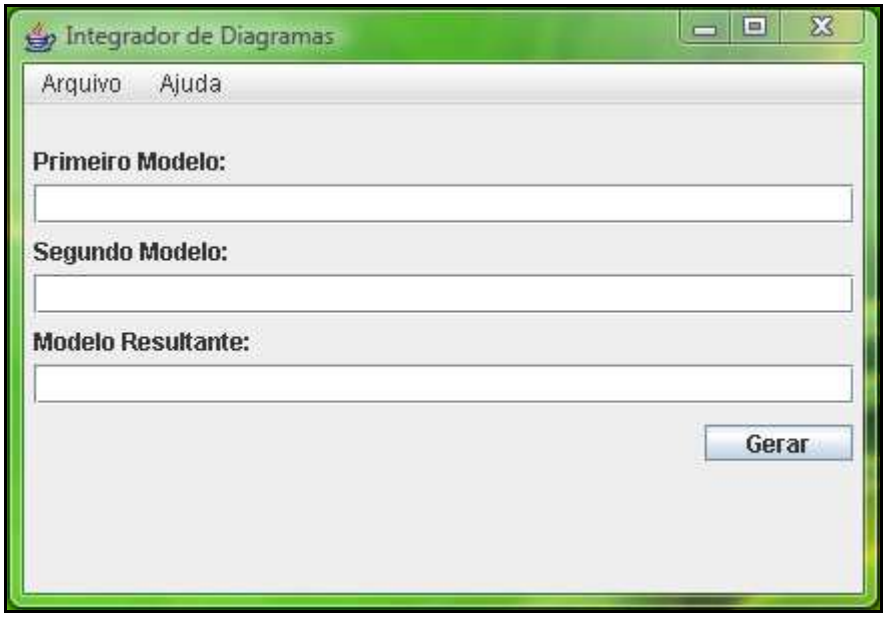

Figura 8 – *Screenshot* da ferramenta de integração de diagramas.

5.2.2 Integração com KerMeta

Por se tratar de uma linguagem recente, existem algumas questões que ainda não foram endereçadas durante a definição de *KerMeta*. Dentre elas, uma das mais críticas é a ausência de suporte a interfaces gráficas, problema citado na seção anterior. Isso representa um grande obstáculo para o desenvolvimento de aplicações interessantes codificadas inteiramente em *KerMeta*.

Outro ponto que agrava a situação é o fato dessa linguagem ainda não dispor de uma gama significativa de possibilidades de integração com códigos escritos em outras linguagens. Há, dessa forma, uma dificuldade de suprir as deficiências apresentadas dessa linguagem pela conciliação com outra tecnologia.

A fim de resolver esse problema utilizou-se a linguagem *Java*. Apesar de *KerMeta* possuir uma palavra reservada em sua sintaxe para chamar operações estáticas em *Java*, não foi possível fazer uso desse recurso. Sua utilização foi inviabilizada, pois o cenário idealizado necessitava que uma operação *KerMeta* fosse invocada de um contexto *Java* e não o contrário. A abordagem adotada foi estabelecer a integração através de chamadas diretas ao interpretador de *KerMeta*. Dessa forma, no momento de executar a operação em *KerMeta* que efetivamente processa os modelos iniciais gerando o modelo integrado, a parte *Java* da aplicação informa ao interpretador os caminhos dos arquivos de entrada e a função a ser invocada.

5.2.3 Processamento de Arquivos de Entrada e de Saída

Dado que *Java* é uma linguagem mais estável que *KerMeta* e possui uma documentação

bastante detalhada, os processos de conversão de arquivos de entrada em arquivos *.km e desconversão do arquivo *.km resultante para o formato desejado do arquivo de saída foram implementados nessa linguagem. Tais processos apenas são executados quando os formatos dos arquivos de entrada ou de saída diferem do padrão. Caso contrário, os arquivos são apenas copiados para os devidos diretórios.

Com o objetivo de facilitar a realização de manutenções dessa parte da ferramenta, tanto para inclusão de novas informações para processamento, quanto para estender o suporte a novos formatos de arquivo, os processos de conversão e de desconversão foram divididos em duas etapas.

O primeiro processo e responsável por converter um arquivo de entrada em um arquivo no formato *.km cujo caminho é posteriormente informado a rotina de integração escrita em *KerMeta*. Em sua etapa inicial, o arquivo de entrada é lido e as informações relacionadas às estruturas do diagrama de classes que ele representa são carregadas em um objeto do tipo "Modelo". Relacionado a esse objeto existem outros objetos que correspondem cada um a um tipo de elemento encontrado em um diagrama de classes (Ex: classe, atributo, operação). Esses relacionamentos são estruturados segundo o modelo abaixo.

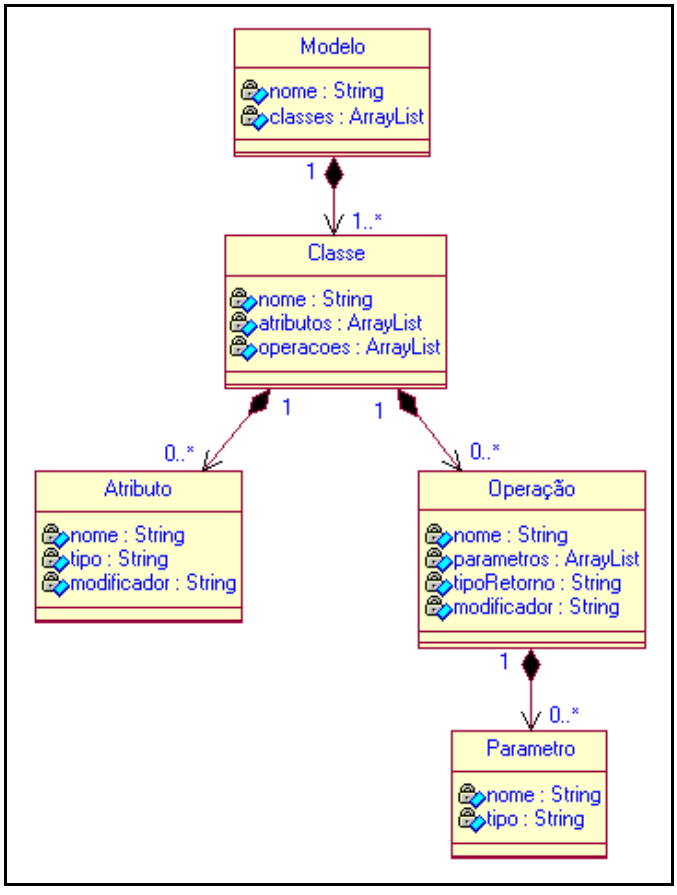

Figura 9 – Modelagem da classe Modelo e classes relacionadas.

Na figura 10, pode-se observar um trecho de uma entrada *.rtmdl e a estrutura correspondente criada em memória ao final da primeira etapa do processo de conversão.

```
                        quid "3619364703E4")))
root_category (object Class_Category "Logical View"
        quid "3619364703C5"
        global TRUE
        exportControl "Public"
        logical_models (list unit_reference_list
                (object Class "Gui"
                        quid "47024CA800CA"
                        class_attributes (list class_attribute_list
                                (object ClassAttribute "fachada"
                                        quid "47024E3C0262"
                                        type "Fachada"))
                        operations (list Operations
                                (object Operation "cadastarEntidade"
                                        quid "47024E5B00F4"
                                        parameters (list Parameters
                                                (object Parameter "novo"
                                                        quid "47024E9C01CE"
                                                        initv "null"
                                                        type "Entidade"))
                                        result "boolean"))
                        assocsOwned (list assocsOwnedList
                                (object Assoc "$UNNAMED$0"
                                        quid "47024E2B0195"
```

modelo	ModeloRTMDL (id=43)
classes	ArrayList<E> (id=45)
elementData	Object[10] (id=54)
[0]	Classe (id=63)
atributos	ArrayList<E> (id=66)
elementData	Object[10] (id=67)
[0]	Atributo (id=68)
modificador	"Private"
nome	"fachada"
tipo	"Fachada"
modCount	1
size	1
nome	"Gui"
operacoes	ArrayList<E> (id=65)
elementData	Object[10] (id=73)
[0]	Operacao (id=74)
modificador	"Public"
nome	"cadastarEntidade"
parametros	ArrayList<E> (id=77)
elementData	Object[10] (id=80)
[0]	Parametro (id=81)
nome	"novo"
tipo	"Entidade"
modCount	1
size	1
tipoRetorno	"boolean"
modCount	1
size	1
[1]	Classe (id=56)
[2]	Classe (id=58)

Figura 10 – Trecho de arquivo *rtmdl e estrutura correspondente gerada.

Uma vez carregado um objeto do tipo "Modelo" com as informações do arquivo de entrada, começa a segunda parte do processo de conversão. Nela o modelo é processado de modo que os dados contidos nele são expressos através das *tags* discutidas na seção 5.1 e escritos em um arquivo numa pasta temporária. O arquivo criado desse modo é aquele cujo caminho será informado à rotina de integração, dando prosseguimento a esse processo. A figura a seguir ilustra o resultado da segunda etapa sobre a estrutura apresentada na figura anterior.

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://ww

<kermeta_language_structure:Package tag="/1 /2" name="Test">

  <ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition" name=
    <ownedAttribute name="fachada" type="/0/Gui/fachada/@containedType.0" isOrdere
      <containedType xsi:type="kermeta_language_structure:Class" typeDefinition="/
    </ownedAttribute>

    <ownedOperation name="cadastarEntidade" type="/0/Gui/cadastarEntidade/@contain
      <containedType xsi:type="kermeta_language_structure:Class">
        <typeDefinition xsi:type="kermeta_language_structure:ClassDefinition" href
      </containedType>
      <ownedParameter name="novo" type="/0/Gui/cadastarEntidade/@ownedParameter.0/
        <containedType xsi:type="kermeta_language_structure:Class" typeDefinition=
      </ownedParameter>
    </ownedOperation>

  </ownedTypeDefinition>
        ⋮
```

Figura 11 – Arquivo *.km temporário relativo à classe "Gui".

No processo de desconversão a mesma lógica e aplicada. A diferença é que ele transforma o arquivo *.km resultante da integração em um arquivo com nome e formato iguais aos definidos pelo usuário no campo "Modelo Resultante" (figura 8, seção 5.2.1). Por isso, o objeto "Modelo" passa a ser preenchido a partir de um arquivo *.km na etapa 1 e tem seus dados expressos segundo os padrões do formato do arquivo de saída.

5.3 KerMeta

KerMeta (abreviatura de "*Kernel Metamodeling*") [1] é uma linguagem *open-source*, interpretada e de domínio específico dedicada à engenharia de metamodelos. Sua criação foi iniciada em 2005 por Franck Fleurey. Uma de suas grandes vantagens é o fato dela atender a lacuna deixada por *MOF* (*Meta-Object Facility*) que define apenas a estrutura dos metamodelos, pois, ela possui um modo de especificar semântica estática (à maneira de *OCL*) e semântica dinâmica pelo uso de semântica operacional na operação dos metamodelos.

Essa linguagem utiliza conceitos de outras como *MOF*, *OCL* (*Object Constraint Language*) e *QVT* (*Query / Views / Transformations*) [11] e por ser recente, encontra-se na versão

0.5.1. *KerMeta* foi concebida a fim de:

- Escrever programas que também são modelos.
- Escrever transformações de modelos (programas que transformam um modelo em outro).
- Escrever restrições sobre modelos e executá-los.

O objetivo dessa abordagem orientada a modelos é incluir acima do nível de objeto um novo nível de abstração. Dessa maneira, pode-se olhar um dado sistema como um conjunto de conceitos que forma explicitamente um todo coerente, que pode ser chamado de modelo. As principais características da linguagem *KerMeta* são:

- Imperatividade
- Orientação a objetos
- Orientação a modelos
- Suporte a funções e expressões lambda de primeira classe
- Atribuição estática de tipos

5.3.1 Definição e Mapeamento de Tipos

Em um cenário de integração entre dois diagramas expressos em arquivos *.km (formato padrão da aplicação) não há necessidade de uma conversão de tipos porque as definições de tipos referenciam entidades descritas no arquivo ou tipos pré-definidos de *KerMeta*. Porém, considerando-se os demais formatos suportados, nota-se que esse casamento de tipos não ocorrerá, tornando impossível a integração.

Logo foi preciso definir um mapeamento entre os tipos pré-definidos de cada formato de arquivo suportado e os tipos padrão de *KerMeta*. Para tanto, a documentação da linguagem foi analisada com a meta de identificar os tipos padrão da mesma e estabelecer correspondências entre estes e os tipos padrão de cada formato de arquivo. Contudo, não foi possível encontrar um mapeamento único entre esses conjuntos de tipos que permitisse a conversão e desconversão de formatos sem perda ou troca de tipos.

A solução para esse problema foi criar o arquivo *.km mostrado na próxima figura. Esse arquivo funciona como uma fonte única de tipos definidos que pode ser referenciada pelos arquivos convertidos de outros formatos. Com essa estratégia pôde-se gerar um mapeamento único cujos tipos são compreendidos pela rotina de integração por estarem definidos em um arquivo *.km referenciado pelo arquivo de entrada.

```
nl version="1.0" encoding="ASCII"?>
i:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:kermeta_language_structure="http://kermeta/kermeta.ecore//language/str
kermeta_language_structure:Package name="temp">
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="String"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="Character"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="Double"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="Float"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="Byte"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="Short"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="Integer"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="Long"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="Boolean"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="UnknownJavaObject
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="NotModeledType"/>
 <ownedTypeDefinition
     xsi:type="kermeta_language_structure:ClassDefinition" name="VoidType"/>
/kermeta_language_structure:Package>
mi:XMI>
```

Figura 12 – Arquivo *.km com a definição dos tipos básicos.

Na tabela seguinte consta o mapeamento estabelecido entre os tipos padrão encontrados em arquivos no formato *.rtmdl.

Tabela 2 – Mapeamento entre tipos do Rose RT e tipos definidos em *KerMeta*.

Tipos do Rose RT	Tipos Definidos em KerMeta
String	String
Char	Character
Double	Double
Float	Float
Byte	Byte
Short	Short
Int	Integer
Long	Long

Tipos do Rose RT	Tipos Definidos em KerMeta
Boolean	Boolean
Tipos não especificados no diagrama	UnknownJavaObject
Tipos especificados que não são pré-definidos nem foram modelados	NotModeledType
Void	VoidType

5.3.2 Processo de Integração de Modelos

O processo de integração tem início efetivamente quando, após as devidas conversões dos arquivos de entrada, a parte *Java* da ferramenta registra os metamodelos das regas de casamento e de integração no repositório *EMF* (*Eclipse Modeling Framework*) e invoca (via interpretador) a função de integração informando os caminhos dos arquivos a serem considerados. A partir desse ponto, essa função inicializa uma instância do repositório *EMF* a fim de ter acesso aos metamodelos de regas registrados pelo código *Java* e carrega ambos os modelos através dos caminhos informados.

Logo em seguida uma instância do integrador é criada dando origem por conseqüência a uma instância do objeto que verifica se há casamento entre os elementos dos modelos. Esse objeto auxiliar informa o integrador quando dois elementos devem ser integrados, para que então o integrador analise e execute a ação de integração correspondente.

Tanto o integrador quanto o avaliador de casamentos realizam suas funções baseados em conjuntos de regras pré-definidas. Esses grupos de regras são denominados regras de integração e regras de casamento. Cada um desses grupos apresenta suas regras separadas em três arquivos de texto. O primeiro contém regras relativas a classes, o segundo apresenta regras relativas a propriedades e o último traz regras relativas a operações. Durante a inicialização do integrador e do verificador de casamento um carregador de regras é criado como mostra a próxima figura.

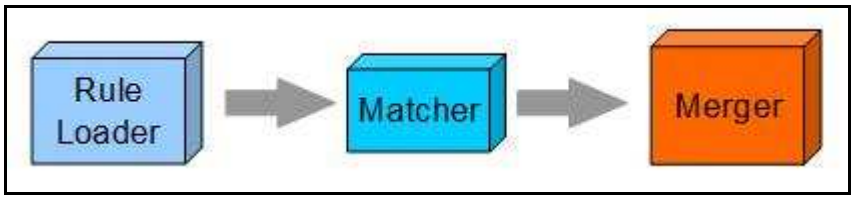

Figura 13 – Dependências entre os principais módulos do sistema.

Por sua vez o ultimo lê os arquivos texto de regras e, baseado num arquivo de mapeamento destes nos metamodelos de regras do repositório *EMF*, converte-os em arquivos *.xmi. Uma vez carregados as regras contidas nesses arquivos são reconhecidas como objetos *KerMeta* que são utilizados pelo integrador e pelo verificador de casamentos para realizar a integração e checar o casamento entre dois elementos respectivamente. A figura abaixo ilustra o papel do carregador de regras.

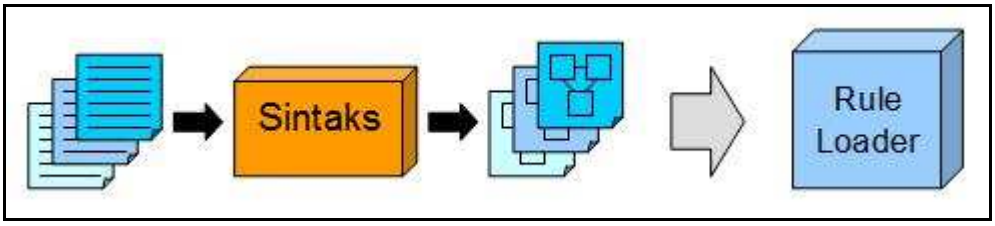

Figura 14 – Processo de carregamento de regras.

Terminada a inicialização do integrador e do verificador de casamento, é efetuada uma chamada à operação de integração de modelos passando os dois modelos de entrada que já foram carregados. Essa operação percorre os elementos do modelo, consultando o verificador para saber se houve casamento e aplicando a transformação definida pelo objeto *KerMeta* correspondente, que foi originado a partir de uma regra de integração. O resultado desse processamento é uma nova instância de modelo posteriormente salva em um arquivo *.km, encerrando o processo de integração. A figura a seguir mostra uma visão geral do processo.

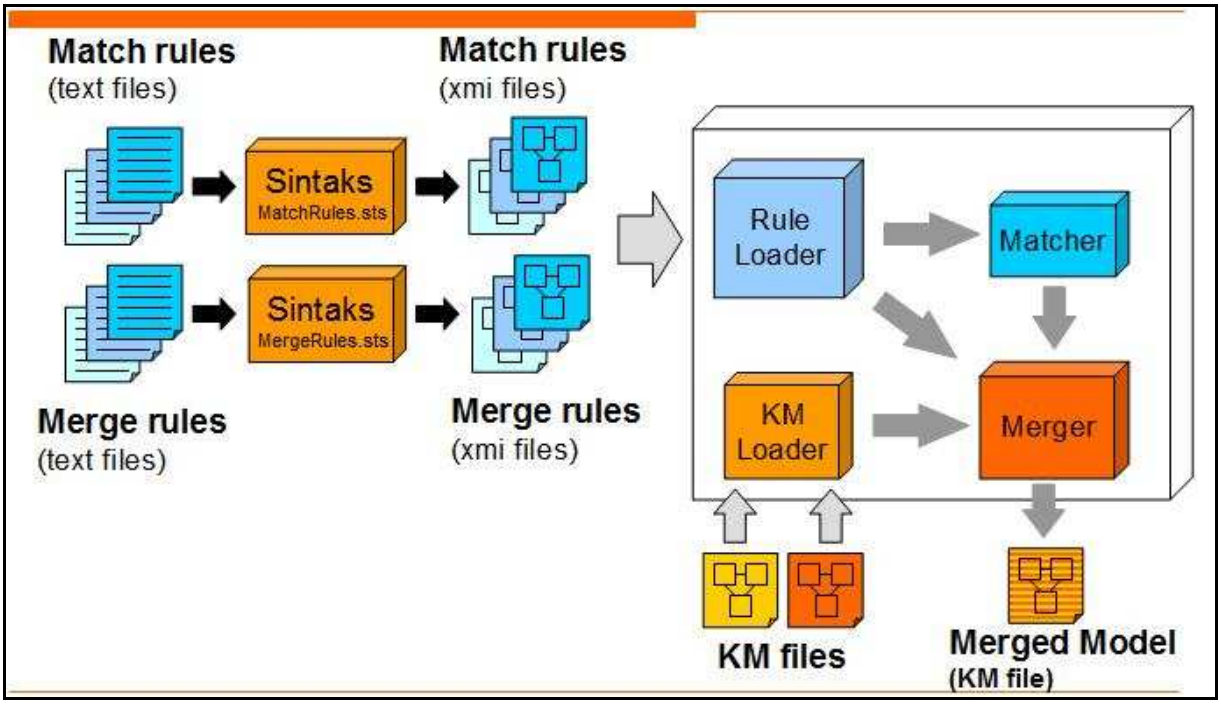

Figura 15 – Visão geral do processo de integração.

6. Estudo de Caso

Com o objetivo de ilustrar a utilização da ferramenta criada, um cenário de desenvolvimento de software é descrito a seguir. Nele são selecionados dois modelos originados dos casos de uso que deverão ser integrados durante a etapa de projeto de arquitetura. Esses modelos são, então, integrados por meio da ferramenta e os resultados parciais e final são mostrados.

Como cenário empregado será utilizado o exemplo de desenvolvimento do sistema *QIB* (*Qualiti Internet Bank*) presente em [15]. Nele um sistema bancário de internet é especificado, analisado e projetado. O sistema apresenta várias funcionalidades relativas à algumas das operações comumente oferecidas pelos diversos sites bancários existentes. Essas funcionalidades são exibidas no diagrama de casos de uso abaixo.

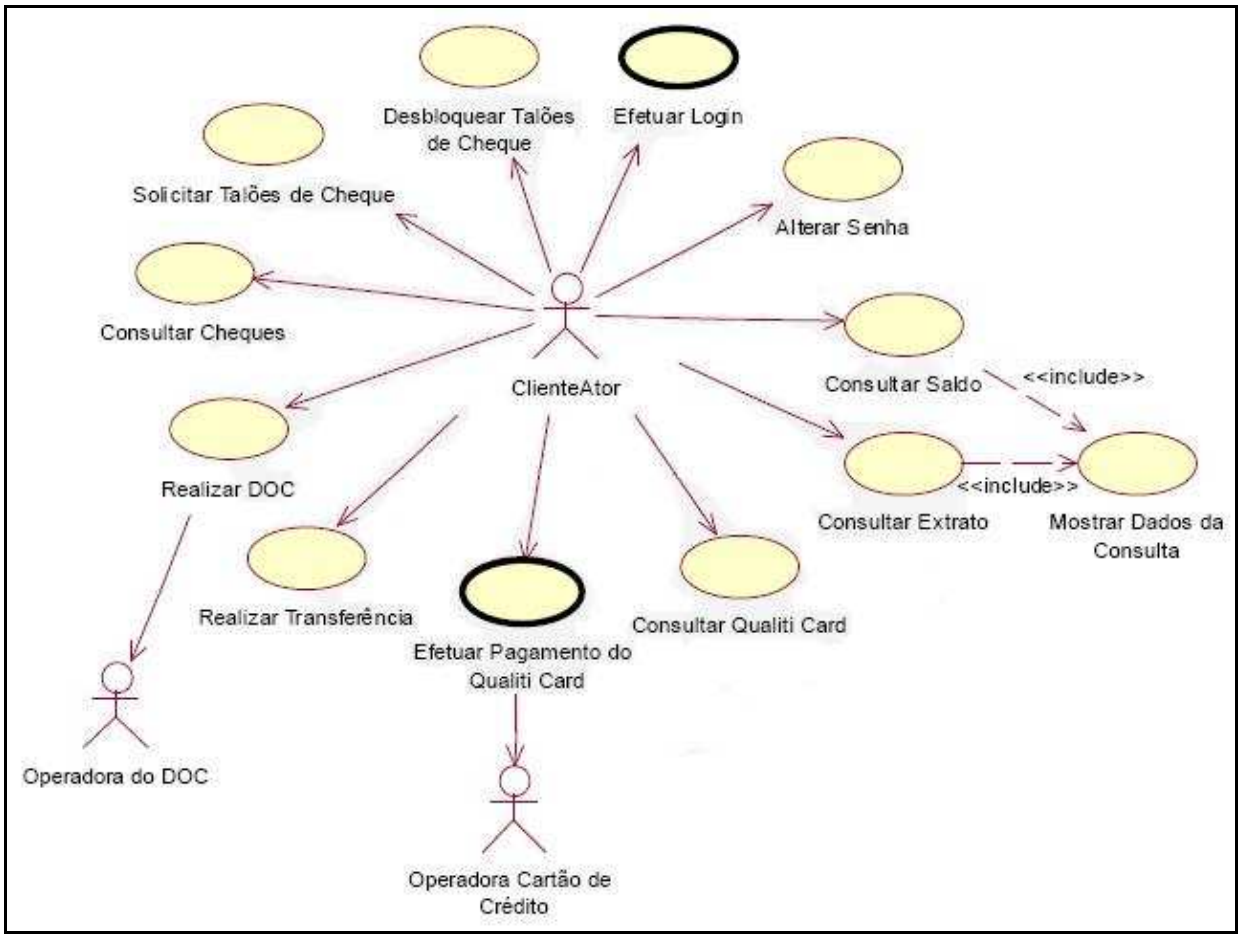

Figura 16 – Diagrama de casos de uso do sistema *QIB*.

Dentre esses casos de uso, serão empregados nesse estudo de caso apenas os de "Efetuar Login" e de "Efetuar Pagamento do Qualiti Card", ambos em destaque no diagrama mostrado acima. Segundo o processo descrito no *RUP* é necessária a existência de uma tabela de mapeamento entre as classes de análise e os elementos de projeto ao se projetar a arquitetura a partir da análise dos casos de uso. Nessa tabela estão definidos a inclusão de

novos elementos e a permanência, exclusão ou desdobramento de elementos já modelados pela análise. Essas definições representam decisões de projeto como, a centralização do acesso a funcionalidades do sistema, a adoção da arquitetura em camadas e a independência do modo de armazenamento de dados. Contudo, a ferramenta proposta ainda não suporta a definição da tabela citada, que representa um dos pontos de trabalhos futuros. Por isso, com a finalidade de ilustrar apropriadamente o processo de integração, essas decisões de projeto que deveriam estar especificadas pelos mapeamentos da tabela serão incorporadas automaticamente nos modelos gerados pela análise de casos de uso. Essas decisões são:

- Inclusão da classe "Fachada" para a centralização do acesso às funcionalidades.
- Inclusão de repositórios de suas interfaces separando e isolando camadas de dados e de negócios.
- Conversão do cadastro de pagamento em cadastro de transação, para futuro reuso por outros casos de uso do sistema.

Partindo dessa premissa, o primeiro caso de uso selecionado trata a questão da autenticação de um usuário em um sistema *web*, através de um login e uma senha correspondente, conforme descrito no fluxo de atividades da próxima tabela.

Tabela 3 – Fluxo de eventos do caso de uso de "Efetuar Login".

Fluxo de Eventos Principal
1. O cliente informa login e senha. 2. O sistema verifica se o login e a senha são válidos. 3. O sistema registra o início de uma sessão de uso.
Fluxo Secundário
1. No passo 2, se o login ou a senha forem inválidos, o sistema exibe uma mensagem e volta ao passo 1.

O resultado do processo de análise sobre o fluxo apresentado somado às alterações descritas anteriormente resulta no modelo apresentado na figura 17. O arquivo que o contém corresponde à uma das entradas que são informadas para a ferramenta integradora.

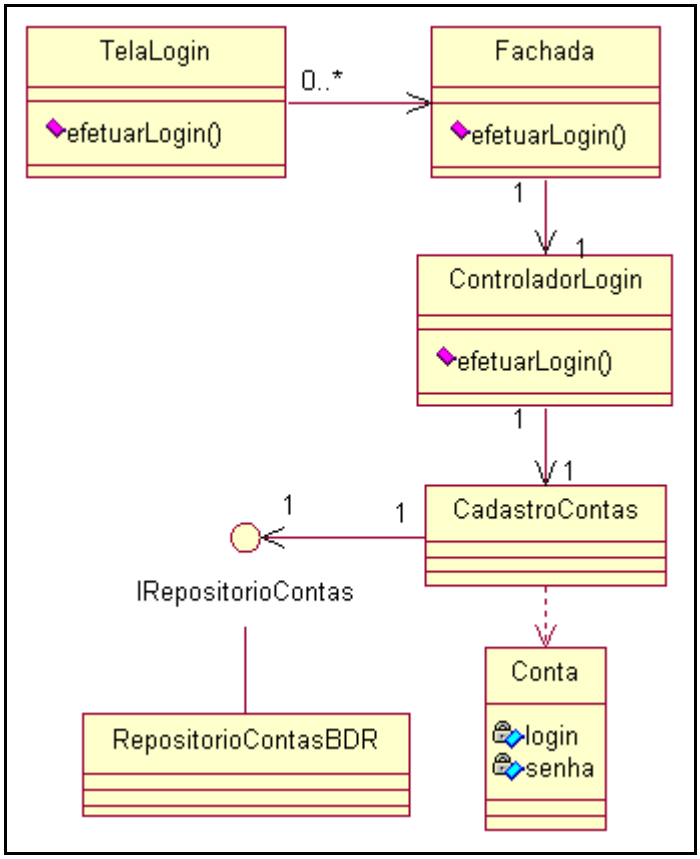

Figura 17 – Modelagem do caso de uso de "Efetuar Login".

Já o caso de uso de "Efetuar Pagamento do Qualiti Card" é responsável por realizar o pagamento do Qualiti Card junto à operadora de cartão de crédito. Ele assume que cada cliente possui apenas um cartão do qual é titular e que este cartão está vinculado exclusivamente a uma operadora. Seu fluxo está descrito na tabela abaixo.

Tabela 4 – Fluxo de eventos do caso de uso de "Efetuar Pagamento do Qualiti Card".

Fluxo de Eventos Principal
1. O cliente informa os dados necessários para efetuar o pagamento: o código de barras da fatura que deseja efetuar o pagamento e o valor que deseja pagar. 2. O sistema recupera a conta bancária do cliente logado. 3. O sistema verifica se o saldo da conta é suficiente para realizar o pagamento. 4. O sistema debita da conta do cliente. 5. O sistema envia o pagamento à operadora de cartão de crédito. 6. O sistema registra a transação de pagamento e emite um comprovante da mesma para o usuário. A transação registrada contém os dados da conta do cliente, o código de barras da fatura, data, hora e valor do pagamento.

Fluxo Secundário
1. No passo 3, se o saldo disponível na conta do cliente for menor que o valor do pagamento, o sistema informa que o saldo é insuficiente e retorna para o passo 1. 2. No passo 5, se a operadora de cartão de crédito estiver inativa, o sistema exibe uma mensagem e cancela a operação. 3. Em qualquer momento o usuário pode cancelar a operação.

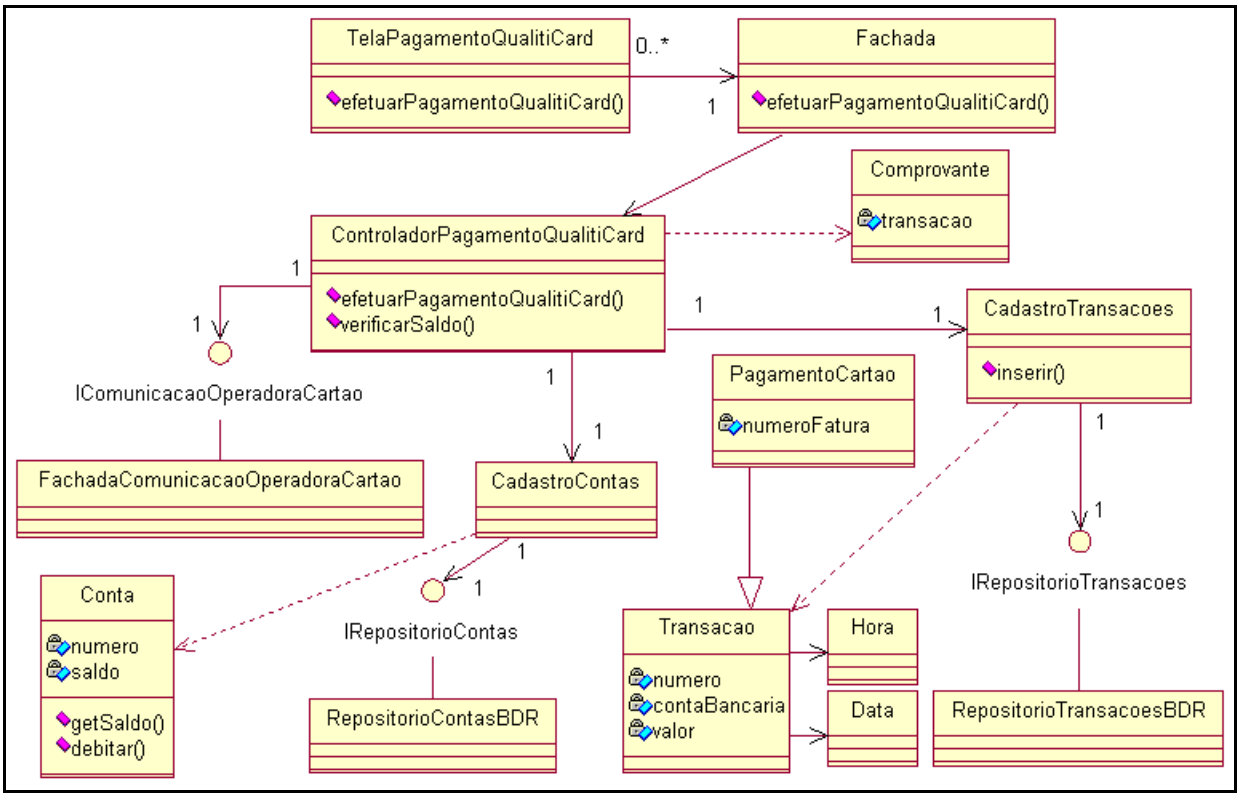

Figura 18 – Modelagem do caso de uso de "Efetuar Pagamento do Qualiti Card".

O resultado do processo de análise sobre o último fluxo novamente incorporando as decisões de projeto descritas resulta no modelo apresentado na figura 18. O arquivo que o contém corresponde a outra entrada que é informada para a ferramenta integradora. De posse dos arquivos de ambos os modelos a ferramenta é iniciada, abrindo a tela principal (figura 8, seção 5). Os caminhos desses arquivos são informados juntamente com o caminho para salvamento do arquivo de saída. Nesse exemplo, assumi-se os nomes dos arquivos como sendo "login.rtmdl" e "pagarCartao.rtmdl" respectivamente. Visto que os dois não estão no formato padrão da ferramenta, eles são convertidos para esse formato (Anexos A e B respectivamente) e salvos em um diretório temporário, conforme descrito no tópico 5.2.3.

O processo de integração lê esses arquivos e os processa, dando origem ao modelo integrado em formato *.km (Anexo C). Assumindo que o nome do arquivo de saída informado é "arquiteturaQIB.rtmdl" é preciso submeter o resultado da integração ao processo de desconversão, também discutido no tópico 5.2.3. O modelo resultante em formato *.rtmdl é

ilustrado na figura a seguir.

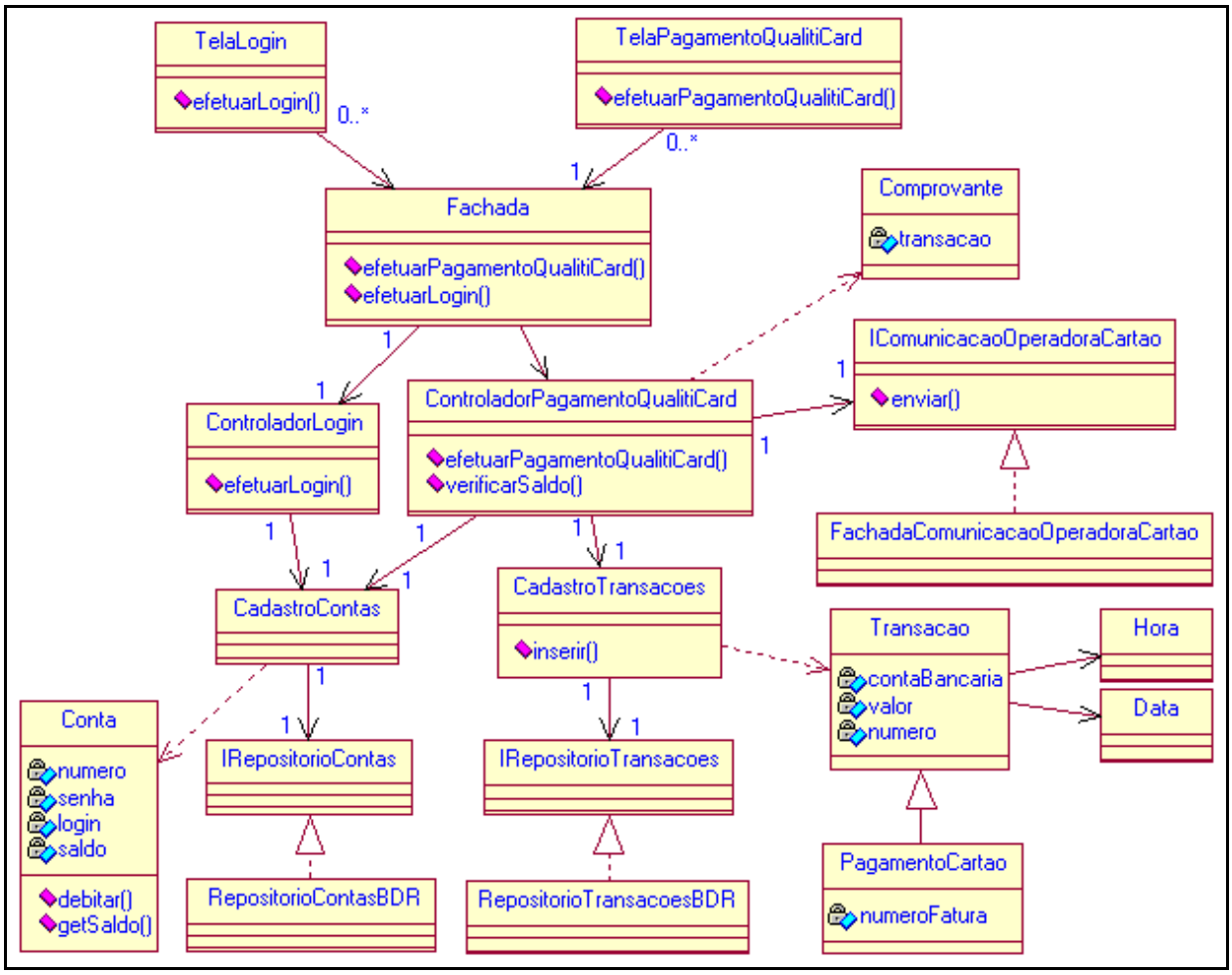

Figura 19 – Modelo resultante do processo de integração.

Como o processo de integração não lida com a apresentação gráfica dos elementos, estes precisaram ser reorganizados espacialmente no Rose RT, a fim de ficarem dispostos como mostra a figura. Nota-se aqui a integração por cópia no caso das telas e a mesclagem de elementos no caso das classes “Fachada” e “Conta”. Contudo, a integração do cadastro de conta nesse caso não é interessante, pois ele representa conceitos diferentes nos modelos de entrada, problema endereçado pela tabela de mapeamento proposta em trabalhos futuros.

7. Conclusões

Dado que os modelos foram promovidos a artefatos primários no desenvolvimento de software, a procura por técnicas, ferramentas ou metodologias que facilitem as atividades de gestão de modelos têm apresentado um crescimento contínuo. Atualmente, as operações de gerenciamento como a validação e a transformação de modelos têm sido extensivamente estudadas, enquanto outros aspectos da Engenharia de Modelos, talvez de importância equivalente, permanecem significativamente subdesenvolvidos. Dentre esses aspectos, um dos menos desenvolvidos é o da integração de modelos.

Atualmente, existem muitos padrões de Engenharia de Modelos e outros novos ainda estão surgindo, a maioria dos quais são gerenciados pela *OMG* (*Object Management Group*). O *OCL* (*Object Constraint Language*) é a linguagem padrão para expressar restrições sobre modelos e metamodelos. O padrão *QVT* (*Queries-Views-Transformations*) tem como principal foco as transformações e as relações no contexto de um modelo para outro. No que diz respeito à questão de transformações de modelo para texto (contexto da geração código), propostas de uma linguagem apropriada foram solicitadas pela *OMG* [12]. Finalmente, em relação a transformações de texto para modelo (como é o caso da engenharia reversa), o ASTM (*Abstract Syntax Tree Metamodel*) [13] e o KDM (*Knowledge Discovery Metamodel*) [14], foram propostos e estão atualmente em fase de padronização. Apesar de toda essa gama de padrões existentes, nenhum deles endereça o ponto da integração de modelos de modo genérico nem de modo específico.

Alternando para o cenário das ferramentas que oferecem suporte a diversas atividades durante todo o ciclo de vida de um projeto de desenvolvimento de software, a questão da integração de modelos também é geralmente negligenciada. As poucas aplicações que apresentam algum módulo com funcionalidades de integração, possuem esse componente focado em outros fins que não a geração de um projeto de arquitetura pela integração de modelos resultantes da análise de casos de uso.

O grande obstáculo nesse ramo, ainda em desenvolvimento, da Engenharia de Modelos referente à integração de modelos é o caráter heurístico que permeia o processo de definição de equivalências entre elementos constituintes de modelos distintos. O trabalho de remover a participação humana nessa etapa do processo pode acarretar perda na qualidade dos resultados obtidos, caso não seja tratada adequadamente. Outro fato importante é o nível de flexibilização permitido pelo algoritmo de integração utilizado no processo. Esse ponto adquire suma importância no sentido de que cada integração de um dado conjunto de modelos encerra suas próprias peculiaridades que devem lidadas conforme o necessário, evitando assim a geração de modelos resultantes inválidos.

Esse trabalho contribuiu com o desenvolvimento do setor de integração de modelos pertencente a área da Engenharia de Modelos, promovendo uma melhoria na eficiência de realização do fluxo de análise e projeto do *RUP*, através da criação de uma ferramenta para a integração automatizada de modelos. Esta deve ser capaz de oferecer suporte a atividade de geração do projeto da arquitetura, presente no fluxo citado. Não obstante, ele também pode

ser tomado como um esforço inicial no desenvolvimento de uma ferramenta que automatize pontos, ainda não abordados por ferramentas existentes, da disciplina de análise e projeto do *RUP* almejando sempre o crescimento da eficiência desse processo.

7.1 Trabalhos futuros

Tendo em vista, o estado atual de desenvolvimento da a área na qual a ferramenta proposta foi implementada e o fato de se tratar de uma primeira versão de uma aplicação que lida com um contexto complexo, existe um variado conjunto de melhorias a serem incluídas em aperfeiçoamentos a serem realizados. Estes se estendem desde a inclusão e do aproveitamento de mais informações acerca do comportamento dos elementos contidos nos modelos de entrada, até a implementação de melhorias de usabilidade da interface gráfica da ferramenta.

Conforme discutido nas seções anteriores desse trabalho, a integração de um dado conjunto de modelos pode apresentar uma variedade de características próprias incapazes de serem encontradas da mesma forma em outros cenários de integração. Isso ocorre devido ao tipo de semântica que está associada aos modelos. Por não se tratar de uma semântica formalmente definida, ele resulta por deixar mais de uma forma de interpretação possível, dificultando um processamento automático exato. Uma forma de facilitar o tratamento dessa questão é a inclusão de uma tabela de mapeamento entre elementos de modelos diferentes como entrada para o processo de integração.

Essa tabela seria informada pelo usuário da ferramenta de integração, juntamente com os caminhos onde estão localizados os modelos de entrada e o diretório no qual será salvo o resultado do processo de integração. Sua função seria auxiliar o trabalho realizado pelo verificador de correspondências entre elementos. Atualmente, o verificador define as correspondências através de regras booleanas pré-estabelecidas que se baseiam, prioritariamente, em termos de nomenclatura e estruturação dos elementos avaliados, estando sujeitas a avaliações incorretas devido a uma ambigüidade semântica. Nesse ponto os mapeamentos contidos na tabela, informada pelo usuário, possuiriam prioridade em relação aos resultados das avaliações realizadas pelas expressões booleanas na definição das correspondências entre elementos.

Contudo, é importante ressaltar que diferentemente do trabalho exposto na seção de ferramentas [8], o conjunto de mapeamentos entre modelos que compõe a tabela proposta aqui não apresenta um caráter obrigatório. O processo de integração executado pela aplicação desenvolvida possui a capacidade de estabelecer correspondências entre elementos por meio da avaliação de expressões, tornando opcional a informação da tabela citada. Além disso, não há a necessidade do conjunto de relacionamentos, descritos na tabela de mapeamento, conter todas as relações entre os modelos. Na realidade a intenção é que apenas os elementos causadores de ambigüidade sejam contemplados por essas relações, visto que as regras de casamento se responsabilizariam pela correspondência dos elementos não mapeados.

Ainda sobre o processo de integração, um foco importante para os esforços futuros é a extensão dos conjuntos de regras de casamento e de integração que são utilizadas hoje no

processo. Principalmente no tocante a integração de corpos de operações novas regras devem ser criadas com a finalidade de permitir possibilidades mais interessantes de configuração do algoritmo de integração, conforme foi discutido na seção 4. Outro ponto chave de aperfeiçoamento, que também deve ser explorado a fundo, consiste na implementação de políticas de solução de conflitos mais robustas que as empregadas nessa primeira versão da ferramenta. Uma possibilidade que deve ser seriamente considerada é a análise do comportamento e dos relacionamentos dos elementos conflitantes a fim de se estabelecer uma predileção mais embasada no contexto dos modelos integrados.

Voltando-se para o ponto de vista da usabilidade da aplicação desenvolvida a principal questão a ser endereçada, inicialmente, está relacionada a seleção dos conjuntos de regras que serão considerados durante a integração dos modelos. Devido à flexibilidade presente no *framework KerMeta* utilizado na junção dos modelos, existe a possibilidade de alteração ou substituição de regras pertencentes a um meta-modelo comum. Essa característica é conferida ao *framework* através da forma de funcionamento do módulo carregador de regras, que permite o desacoplamento das mesmas.

Explorando essa capacidade, planeja-se realizar uma extensão da atual interface gráfica da aplicação com o objetivo de incluir uma seção onde o usuário teria a opção de selecionar, dentre as regas existentes, um subconjunto de regras de casamento e de integração que seriam consideradas durante o processamento dos modelos pela ferramenta. Assim, o usuário possuiria à sua disposição um mecanismo prático de configuração dos algoritmos de casamento e integração empregados no processo, capacitando-o a adaptá-los a diversas situações de acordo com sua necessidade.

Analisando-se agora o contexto do mercado de desenvolvimento de software, nota-se que a criação de uma ferramenta de integração de modelos com a capacidade de suportar adequadamente a atividade de geração de projetos de arquitetura seria de grande valia. Contudo, uma vez que essa ferramenta tem por meta a redução do esforço desprendido na realização da atividade citada, seria interessante sua disponibilização de maneira integrada a programas existentes de modelagem de software ou de suporte a ciclos de vida de desenvolvimento de softwares, como o Rose RT e outros do gênero.

Pensando nisso, outro objetivo futuro consiste na criação de versões sob a forma de *plug-ins* para os programas mais utilizados pelo mercado pertencentes a área de atuação mencionada. Assim, podem-se evitar alguns inconvenientes de adoção da ferramenta de integração que seriam enfrentados com o uso da versão *stand-alone*, dentre os quais estão a inclusão de mais um programa para a realização de atividades do fluxo de análise e projeto e a quebra do fluxo de trabalho dentro de uma mesma ferramenta. Essa forma de apresentação contribui para o aumento do grau de usabilidade e redução do tempo gasto pelo usuário, concentrando a execução de tarefas comuns numa única ferramenta.

Também com a pretensão de aumentar o grau de integração do resultado desse trabalho com as soluções de modelagem existentes no mercado, pretende-se estender o suporte da aplicação concebida a novos formatos de arquivos empregados por programas de modelagem de software. Apesar dessa não consistir numa alteração profunda, ela possibilita o emprego da solução em um novo cenário de uso, no qual dois modelos criados através de programas distintos poderão ser integrados. Aliado a isso, o formato do modelo resultante

obtido dessa operação pode ser escolhido pelo usuário como sendo o mesmo de um dos modelos de entrada ou qualquer outro suportado, melhorando a portabilidade dos resultados.

Referências Bibliográficas

[1] – Triskell team. KerMeta Home Page. Disponível em: <http://www.kermeta.org>. Acesso em: 20/09/2007.

[2] – Misael, S. S. Uma Proposta para a Integração de Modelos de Padrões de Software com Ferramentas de Apoio ao Desenvolvimento de Sistemas. Disponível em: <www.mcc.ufc.br/disser/MisaelSilva.pdf>. Acesso em: 24/09/2007.

[3] – Object Management Group. MOF 2.0/XMI Mapping Specification, v2.1. Disponível em: <http://www.omg.org/docs/formal/05-09-01.pdf>. Acesso em: 27/09/2007.

[4] – Philippe Kruchten. The Rational Unified Process: An Introduction (2nd Edition). Acesso em: 19/11/2007.

[5] – Ricardo, A. F. Processo Unificado e RUP (Rational Unified Process). Disponível em: <http://paginas.terra.com.br/negocios/processos2002/processo_unificado_e_rup.htm>. Acesso em: 21/12/2007.

[6] – ABNT. Associação Brasileira de Normas Técnicas. Disponível em: <http://www.abnt.org.br/>. Acesso em: 14/01/2007.

[7] – Wikipedia. IBM Rational Unified Process. Disponível em: <http://en.wikipedia.org/wiki/Rup#History_of_RUP>. Acesso em: 27/09/2007.

[8] – Mehrdad S., Shiva N., Steve E. e Marsha C. Department of Computer Science, University of Toronto, Canada. A Relationship-Driven Framework for Model Merging. Disponível em: <http://www.cs.toronto.edu/~sme/papers/2007/Sabetzadeh-MiSE2007.pdf>. Acesso em: 27/10/2007.

[9] – Object Management Group. Site Oficial. Disponível em: <http://www.omg.org>. Acesso em: 05/10/2007.

[10] – James Rumbaugh, Ivar Jacobson & Grady Booch . The Unified Modeling Language Reference Manual. Acesso em: 07/01/2007.

[11] – Object Management Group. MOF QVT Final Adopted Specification.

Disponível em: <http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf>. Acesso em: 19/12/2007.

[12] – Object Management Group. MOF Model to Text Transformation Language Request

For Proposals (RFP). Disponível em: <http://www.omg.org/cgi-bin/doc?ad/04-04-07.pdf>. Acesso em: 27/12/2007.

[13] – Object Management Group. Abstract Syntax Tree Metamodel, Request For Proposals

(RFP). Disponível em: <http://www.omg.org/cgi-bin/doc?admtf/05-02-02.pdf>. Acesso em: 30/12/2007.

[14] – Object Management Group. Knowledge Discovery Metamodel, Request For Proposals

(RFP). Disponível em: <http://www.omg.org/cgi-bin/doc?lt/03-11-04.pdf>. Acesso em: 30/12/2007.

[15] – Qualiti Software Processes. Analisar Caso de Uso. Disponível em: <http://www.cin.ufpe.br/~if718/transparencias/pdf/analiseCasosDeUsoRT.zip>. Acesso em: 17/01/2008.

[16] – Ken Schwaber & Mike Beedle. Agile. Software Development with Scrum. Acesso em: 22/01/2008.

[17] – Alistair Cockburn. Crystal Clear: A Human-Powered Methodology for Small Teams. Acesso em: 22/01/2008.

[18] – Kent Beck & Martin Fowler. Planning Extreme Programming. Acesso em: 22/01/2008.

[19] – Barry Boehm. A Spiral Model of Software Development and Enhancement. Acesso em: 22/01/2008.

[20] – D'Souza D. & Alan W. The Catalysis Approach. Acesso em: 23/01/2008.

[21] – Rafael Duarte & Olivier Barais. A Flexible Framework for Model Composition in KerMeta. Acesso em: 09/01/2008.

Anexos

Anexo A – Arquivo temporário "login.km"

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:kermeta_language_behavior="http://kermeta/kermeta.ecore//language/behavior"
xmlns:kermeta_language_structure="http://kermeta/kermeta.ecore//language/structure">

<kermeta_language_structure:Package tag="/1 /2" name="Test">

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="CadastroContas">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="Conta">
<ownedAttribute name="login" type="/0/Conta/login/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

<ownedAttribute name="senha" type="/0/Conta/senha/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="IRepositorioContas">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="RepositorioContasBDR">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="TelaLogin">
<ownedOperation name="efetuarLogin" type="/0/TelaLogin/efetuarLogin/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>
```

```
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="ControladorLogin">
<ownedOperation name="efetuarLogin"
type="/0/ControladorLogin/efetuarLogin/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="Fachada">
<ownedOperation name="efetuarLogin" type="/0/Fachada/efetuarLogin/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

</ownedTypeDefinition>

</kermeta_language_structure:Package>

<!-- default -->
<kermeta_language_structure:Tag name="mainClass" value="root_package::Main"
object="/0"/>
<kermeta_language_structure:Tag name="mainOperation" value="main" object="/0"/>

</xmi:XMI>
```

Anexo B – Arquivo temporário "pagarCartao.km"

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:kermeta_language_behavior="http://kermeta/kermeta.ecore//language/behavior"
xmlns:kermeta_language_structure="http://kermeta/kermeta.ecore//language/structure">

<kermeta_language_structure:Package tag="/1 /2" name="Test">

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="IRepositorioTransacoes">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="RepositorioTransacoesBDR">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="Comprovante">
<ownedAttribute name="transacao" type="/0/Comprovante/transacao/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="CadastroTransacoes">
<ownedOperation name="inserir" type="/0/CadastroTransacoes/inserir/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="PagamentoCartao">
<ownedAttribute name="numeroFatura"
type="/0/PagamentoCartao/numeroFatura/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="Transacao">
<ownedAttribute name="numero" type="/0/Transacao/numero/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
```

```
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

<ownedAttribute name="contaBancaria"
type="/0/Transacao/contaBancaria/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

<ownedAttribute name="valor" type="/0/Transacao/valor/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition" name="Data">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition" name="Hora">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="CadastroContas">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="Conta">
<ownedAttribute name="numero" type="/0/Conta/numero/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

<ownedAttribute name="saldo" type="/0/Conta/saldo/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedAttribute>

<ownedOperation name="getSaldo" type="/0/Conta/getSaldo/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

<ownedOperation name="debitar" type="/0/Conta/debitar/@containedType.0">
```

<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="IRepositorioContas">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="RepositorioContasBDR">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="ControladorPagamentoQualitiCard">
<ownedOperation name="efetuarPagamentoQualitiCard"
type="/0/ControladorPagamentoQualitiCard/efetuarPagamentoQualitiCard/@containedType.0
">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

<ownedOperation name="verificarSaldo"
type="/0/ControladorPagamentoQualitiCard/verificarSaldo/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="Fachada">
<ownedOperation name="efetuarPagamentoQualitiCard"
type="/0/Fachada/efetuarPagamentoQualitiCard/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="TelaPagamentoQualitiCard">
<ownedOperation name="efetuarPagamentoQualitiCard"
type="/0/TelaPagamentoQualitiCard/efetuarPagamentoQualitiCard/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>

```
</containedType>
</ownedOperation>

</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="FachadaComunicacaoOperadoraCartao">
</ownedTypeDefinition>

<ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="IComunicacaoOperadoraCartao">
<ownedOperation name="enviar"
type="/0/IComunicacaoOperadoraCartao/enviar/@containedType.0">
<containedType xsi:type="kermeta_language_structure:Class">
<typeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
href="TypeClasses.km#/0/UnknownJavaObject"/>
</containedType>
</ownedOperation>

</ownedTypeDefinition>

</kermeta_language_structure:Package>

<!-- default -->
<kermeta_language_structure:Tag name="mainClass" value="root_package::Main"
object="/0"/>
<kermeta_language_structure:Tag name="mainOperation" value="main" object="/0"/>

</xmi:XMI>
```

Anexo C – Arquivo com o modelo integrado em formato *.km

```
<?xml version="1.0" encoding="ASCII"?>
<kermeta_language_structure:Package xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:kermeta_language_structure="http://www.kermeta.org/kermeta//language/structure"
xmlns:kermeta_language_structure_1="http://kermeta/kermeta.ecore//language/structure"
name="Test">
  <ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="CadastroContas"/>
  <ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="Conta">
    <ownedAttribute name="numero"
type="//@ownedTypeDefinition.1/@ownedAttribute.0/@containedType.0">
      <containedType xsi:type="kermeta_language_structure:Class">
        <typeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
href="../temp/TypeClasses.km#//UnknownJavaObject"/>
      </containedType>
    </ownedAttribute>
    <ownedAttribute name="senha"
type="//@ownedTypeDefinition.1/@ownedAttribute.1/@containedType.0">
      <containedType xsi:type="kermeta_language_structure:Class">
        <typeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
href="../temp/TypeClasses.km#//UnknownJavaObject"/>
      </containedType>
    </ownedAttribute>
    <ownedAttribute name="login"
type="//@ownedTypeDefinition.1/@ownedAttribute.2/@containedType.0">
      <containedType xsi:type="kermeta_language_structure:Class">
        <typeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
href="../temp/TypeClasses.km#//UnknownJavaObject"/>
      </containedType>
    </ownedAttribute>
    <ownedAttribute name="saldo"
type="//@ownedTypeDefinition.1/@ownedAttribute.3/@containedType.0">
      <containedType xsi:type="kermeta_language_structure:Class">
        <typeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
href="../temp/TypeClasses.km#//UnknownJavaObject"/>
      </containedType>
    </ownedAttribute>
    <ownedOperation name="debitar">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedOperation>
    <ownedOperation name="getSaldo">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedOperation>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="IRepositorioContas"/>
  <ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="RepositorioContasBDR"/>
  <ownedTypeDefinition xsi:type="kermeta_language_structure:ClassDefinition"
name="Fachada">
    <ownedOperation name="efetuarPagamentoQualitiCard">
```

```
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedOperation>
    <ownedOperation name="efetuarLogin">
      <type href="../temp/login.km#/0/Fachada/efetuarLogin/@containedType.0"/>
    </ownedOperation>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="IRepositorioTransacoes"/>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="RepositorioTransacoesBDR"/>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="Comprovante">
    <ownedAttribute name="transacao">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedAttribute>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="CadastroTransacoes">
    <ownedOperation name="inserir">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedOperation>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="PagamentoCartao">
    <ownedAttribute name="numeroFatura">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedAttribute>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="Transacao">
    <ownedAttribute name="contaBancaria">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedAttribute>
    <ownedAttribute name="valor">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedAttribute>
    <ownedAttribute name="numero">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedAttribute>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="Data"/>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="Hora"/>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="ControladorPagamentoQualitiCard">
    <ownedOperation name="efetuarPagamentoQualitiCard">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedOperation>
    <ownedOperation name="verificarSaldo">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedOperation>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="TelaPagamentoQualitiCard">
    <ownedOperation name="efetuarPagamentoQualitiCard">
```

```
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedOperation>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="FachadaComunicacaoOperadoraCartao"/>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="IComunicacaoOperadoraCartao">
    <ownedOperation name="enviar">
      <type href="../temp/pagarCartao.km#/0/Conta/saldo/@containedType.0"/>
    </ownedOperation>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="TelaLogin">
    <ownedOperation name="efetuarLogin">
      <type href="../temp/login.km#/0/Fachada/efetuarLogin/@containedType.0"/>
    </ownedOperation>
  </ownedTypeDefinition>
  <ownedTypeDefinition xsi:type="kermeta_language_structure_1:ClassDefinition"
name="ControladorLogin">
    <ownedOperation name="efetuarLogin">
      <type href="../temp/login.km#/0/Fachada/efetuarLogin/@containedType.0"/>
    </ownedOperation>
  </ownedTypeDefinition>
</kermeta_language_structure:Package>
```

Assinaturas

Augusto César Alves Sampaio

(Orientador)

Bruno Rodrigues de Castro Ribeiro

(Aluno)