UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Síntese de Projeto Arquitetural a partir de Realizações de Casos de Uso

Trabalho de Graduação

Aluno: Fernando Valente Kakimoto (fvk@cin.ufpe.br).
Orientador: Augusto Cezar Alves Sampaio (acas@cin.ufpe.br).

Recife, 28 de Novembro de 2008.

Resumo

Uma característica intrínseca a sistemas de software é a complexidade de seu desenvolvimento, resultando na necessidade de estudá-los e analisá-los sob diferentes perspectivas. Uma delas é a visão de projeto arquitetural, a qual abrange características que dão suporte, tanto estrutural quanto comportamental, às funcionalidades de um sistema.

No RUP (Rational Unified Process), a visão de projeto arquitetural é definida pela disciplina de Análise e Projeto, onde é comum a criação de diagramas UML. Entretanto, as ferramentas comerciais existentes não oferecem suporte automatizado para várias atividades do processo, o que contribui para a grande repetição de suas tarefas.

Assim, o trabalho se propõe a analisar ferramentas de modelagem existentes e propor automações de forma a possibilitar a criação de modelos de projeto a partir de modelos abstratos de análise.

Palavras chave: RUP, análise e projeto, projeto arquitetural, modelagem, automatização, modelo, diagrama UML.

Abstract

An intrinsic feature of software systems is their development complexity, resulting in the need of studying and analyzing them from different perspectives. One of them is the architectural design vision, which includes features that support, both structural and behavioral, system functionalities.

In RUP (Rational Unified Process), the architectural design vision is defined by the Analysis and Design discipline, where the creation of UML diagrams is very common. Meanwhile, the existing commercial tools do not support automated for various process activities, contributing to the high repetition of their duties.

Thus, this work aims to analyze existing modeling tools and propose automations able to allow the creation of design models from abstract analysis models.

Keywords: RUP, analysis and design, design architecture, modeling, automation, models, and UML diagrams.

Agradecimentos

A Deus,
por me dar a saúde necessária para sair em busca de meus objetivos.

Aos meus pais,
pelas lições de vida, sempre me guiando pelos caminhos da ética e da dignidade.

Ao meu irmão,
pelo exemplo de luta e perseverança.

À Maria, minha namorada,
pelo apoio e conselhos, além da divisão dos problemas e conquistas.

Aos professores,
por passar da melhor forma seu conhecimento adquirido em anos de estudo.

Ao meu orientador, Professor Augusto Sampaio,
pela disponibilidade e significativa contribuição dada neste trabalho.

Aos colegas de turma,
que assim como eu estiveram esses quatro anos na luta por um ensino superior de qualidade.

Ao Grupo de Pesquisa em Redes e Telecomunicações, ao Grupo de TV Digital do C.E.S.A.R e à Inove Informática,
pela colaboração em minha experiência como profissional.

A todas as pessoas que, de alguma forma, contribuíram na construção deste trabalho.

Sumário

Índice de Figuras

1. Introdução

Na computação, desenvolvimento de software consiste em definir os requisitos, modelar e implementar um sistema computacional. O desenvolvimento de sistemas computacionais é uma tarefa árdua, envolvendo uma série de atividades realizadas de forma organizada por um conjunto de pessoas em um determinado período de tempo.

Atualmente, sistemas de software são desenvolvidos em uma variedade de plataformas: desktop, web, *mobile*, cujas características são específicas de cada domínio. Porém, uma característica intrínseca a todo sistema computacional é a complexidade envolvida na sua criação e evolução. Por essa razão, investe-se cada vez mais tempo e recursos financeiros em técnicas de levantamento de requisitos, análise e projeto de um sistema, antes de iniciar a sua implementação propriamente dita.

Particularmente, a análise e o projeto de um sistema podem ser desenvolvidos sob diferentes perspectivas ou visões. Uma delas é a visão de projeto arquitetural, a qual abrange as características do sistema que dão suporte, tanto estrutural quanto comportamental, às suas funcionalidades.

No RUP (Rational Unified Process) [3], o projeto arquitetural é definido pela disciplina de Análise e Projeto através da criação de modelos responsáveis, principalmente, pelo gerenciamento da complexidade do sistema em desenvolvimento. São várias as razões para a utilização de modelos, entre elas:

- Promover a difusão de informações relativas ao sistema entre os indivíduos envolvidos em sua construção.
- Reduzir os custos do desenvolvimento, já que a detecção e correção de erros são realizadas ainda na construção do modelo.
- Prever o comportamento do sistema através da análise e possível simulação de seus modelos.

Devido à importância da criação de modelos no processo de desenvolvimento de software, sentiu-se necessidade de uma linguagem padrão para a modelagem de sistemas, resultando na criação da UML (Unified Modeling Language) [4]. Aliado à definição da linguagem, foram construídas diversas ferramentas gráficas para suportar a criação de modelos UML, entre as quais se pode citar: Borland Together [12], JUDE [10], Rational Rose [8], Rational Software Architect [15], StarUML [11], entre outras.

As ferramentas de modelagem desenvolvidas, apesar de algumas particularidades, têm como característica comum o suporte à elaboração de um artefato de software que represente a arquitetura final de um sistema. Entretanto, até a sua definição, muitas são as tarefas desempenhadas pela equipe responsável. Por isso, um dos objetivos das ferramentas de modelagem é proporcionar facilidades no trabalho de seus usuários e na construção de diagramas UML.

Apesar de todos os esforços realizados nesse sentido, as ferramentas comerciais existentes não oferecem suporte automatizado para várias atividades do processo, o que acaba contribuindo para a grande repetição de suas tarefas e para a pouca produtividade na criação e manutenção de um modelo. Conseqüentemente, a definição do projeto arquitetural de um sistema acaba, muitas vezes, sendo descontinuada e até abolida do processo utilizado, falha que pode resultar em problemas de implementação, manutenção e extensão.

Em particular, o produto da atividade Realizar Casos de Uso da disciplina de Análise e Projeto do RUP é um conjunto de diagramas (seqüência, colaboração e classes) que modelam cada caso de uso isoladamente, enquanto que o Projeto da Arquitetural é elaborado como uma combinação e consolidação das visões individuais dos casos de uso. Apesar de o RUP oferecer uma sistemática que guie esta transição, as ferramentas comerciais como o Rose, entre outras, não oferecem suporte mecanizado para automatizá-la.

Dessa forma, é de fundamental importância a automação das funcionalidades das ferramentas de modelagem existentes, sendo este o objetivo central deste trabalho. Com isso, é possível agilizar a criação de projetos arquiteturais e evitar que essa atividade, essencial para o desenvolvimento de um projeto de qualidade, caia no desuso.

Para atingir este objetivo, o trabalho apresenta no capítulo 2 o Rational Unified Process, principal processo de desenvolvimento de software atualmente, sua disciplina de Análise e Projeto e a atividade Projetar Arquitetura, contextualizando alguns dos problemas envolvidos em sua realização. Em seguida, o capítulo 3 faz um estudo sobre as principais ferramentas de modelagem utilizadas no mercado, bem como um breve paralelo entre cada uma delas.

No capítulo 4, é apresentada a solução a ser desenvolvida pelo trabalho, buscando resolver os problemas da criação de um projeto arquitetural. O foco é na proposição de transformações automatizadas que apóiem a transição da análise de casos de uso para o projeto arquitetural. As transformações incluem a aplicação de padrões de

projeto, a decomposição e merge de classes e o mapeamento de classes em subsistemas. Já no capítulo 5, descreve-se a implementação da solução introduzida no capítulo anterior, como um *plug-in* na ferramenta Rational Software Architect. O capítulo 6 demonstra a utilização do resultado do trabalho em um exemplo real de projeto arquitetural. Finalmente, o capítulo 7 faz uma retrospectiva de todo o trabalho apresentado, além de introduzir alguns trabalhos relacionados e os próximos passos a serem dados, dando seqüência ao trabalho desenvolvido.

2. Rational Unified Process

Na tentativa de lidar com a complexidade do desenvolvimento de sistemas computacionais e de minimizar os problemas envolvidos na sua construção, surgiram os processos de desenvolvimento de software. Um processo de desenvolvimento compreende todas as atividades necessárias para definir, desenvolver, testar e manter um produto de software [1].

Basicamente, um processo de desenvolvimento envolve quatro regras [2]:

1. Prover um guia para ordenar as atividades da equipe desenvolvedora.
2. Especificar quais artefatos devem ser criados e quando devem ser criados.
3. Direcionar as tarefas individuais, bem como as tarefas da equipe como um todo.
4. Oferecer um critério para monitorar e medir os produtos e atividades do projeto.

Sem um processo de software, a equipe desenvolvedora irá trabalhar de maneira *ad hoc*, confiando o sucesso do projeto na disciplina e na competência pessoal de cada membro da equipe. Em contrapartida, organizações que empregam um processo bem definido e documentado são capazes de desenvolver sistemas complexos de forma repetida, previsível e relativamente impessoal, garantindo uma melhoria a cada novo projeto e um aumento da eficiência e produtividade de toda a organização.

O *Rational Unified Process* [3] é um processo de engenharia de software proprietário, desenvolvido pela Rational Software e adquirido posteriormente pela IBM, que fornece uma abordagem disciplinada para a atribuição de tarefas e responsabilidades dentro de uma organização. Seu objetivo principal é garantir a produção de um software de qualidade que satisfaça as necessidades de seus usuários finais a partir de um determinado cronograma e orçamento [3].

O RUP captura muitas das boas práticas do desenvolvimento de software, sendo adequado para um vasto número de projetos e organizações. Entre elas, pode-se citar as práticas de: desenvolvimento iterativo de software, gerenciamento de requisitos, utilização de arquiteturas baseadas em componentes, criação de modelos visuais do software, verificação da qualidade do software e controle de mudanças do software.

A abordagem iterativa utilizada pelo RUP é geralmente mais efetiva do que a abordagem linear, ou cascata, devido a diversos fatores. Primeiro, tal abordagem leva

em consideração a mudança dos requisitos ao longo do projeto, já que tal fato é uma realidade constante no mundo atual. Segundo, a abordagem iterativa antecipa a mitigação dos riscos devido à progressiva integração dos elementos do projeto, em vez de integrá-los apenas ao final do desenvolvimento. Finalmente, ela facilita o reuso de componentes, pois é mais fácil identificar partes reusáveis na medida em que elas vão sendo criadas do que identificá-las apenas no início do projeto.

Quanto ao gerenciamento de requisitos, o RUP utiliza uma abordagem sistemática de elicitar, organizar, comunicar e gerenciar as mudanças nos requisitos de um sistema. Os benefícios dessa atividade incluem um melhor controle das complexidades do projeto, aumento da qualidade do software e satisfação do cliente, redução dos custos e possíveis atrasos na entrega do produto e melhoria na comunicação da equipe.

Outra boa prática de desenvolvimento utilizada pelo RUP é a definição de uma arquitetura baseada em componentes, o que possibilita o projeto, a implementação e o teste individual de cada componente criado. Tal fato resulta em uma integração gradual dos componentes formadores do sistema, bem como em uma maior possibilidade de reuso das suas partes, formando, assim, a base de reuso dentro de uma organização.

Finalmente, o RUP utiliza-se de uma linguagem gráfica, *The Unified Modeling Language* [4], para visualização, especificação e construção de modelos de projeto, os quais ajudam a entender e modelar tanto um problema encontrado quanto a sua solução, tornando-se uma simplificação da realidade. O RUP é um guia para o uso efetivo da UML na modelagem do sistema, descrevendo quais modelos serão necessários, a razão de utilizá-los e como construí-los.

2.1 Visão Geral do RUP

O processo iterativo do RUP é dividido em fases, conforme mostra a Figura 2-1. Entretanto, diferentemente da abordagem em cascata, suas fases não constituem uma seqüência de requisitos, análise, projeto, implementação e integração. Ao invés disso, as fases são ortogonais, sendo cada uma delas concluídas por um *milestone* principal. Em cada uma das quatro fases podem ser feitas mais de uma iteração, com esse número variando de acordo com o tamanho do projeto desenvolvido.

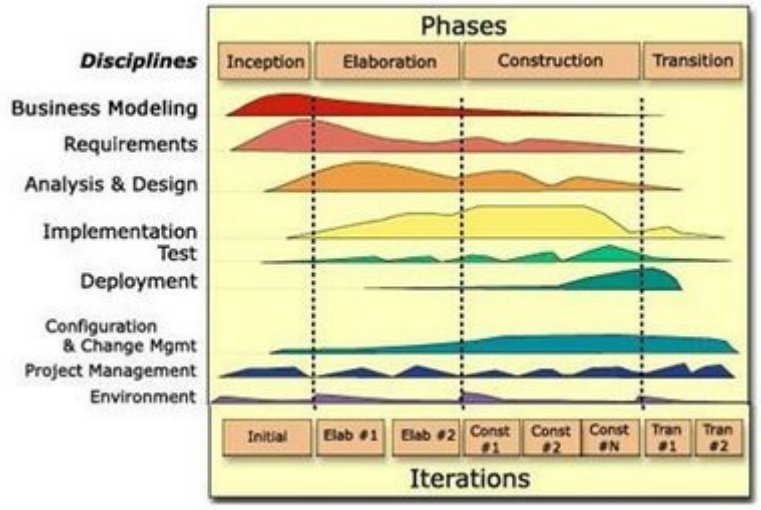

Figura 2-1 As fases do processo iterativo do RUP.

A primeira fase, a fase de Concepção, tem como principal objetivo convergir as opiniões de todos os *steakholders* do projeto sobre o seu escopo e suas fronteiras, incluindo critérios de aceitação e o que deve, e o que não deve fazer parte do produto final. Entre suas atividades estão: formular o escopo do projeto, planejar e preparar o modelo de negócio, estimar potenciais riscos e sintetizar uma possível arquitetura do sistema. Logo, ao fim desta fase o projeto já deverá ter um escopo definido, os primeiros casos de uso criados e um esboço da arquitetura prototipada.

Em seguida, na fase de Elaboração, a equipe do projeto deverá analisar os problemas do domínio em questão, estabelecer uma arquitetura sólida para o sistema, desenvolver o plano de projeto e eliminar os elementos de maior risco. Assim, o final da fase de elaboração marca a definição de uma arquitetura estável e a mitigação dos maiores riscos apontados na fase anterior, junto com um protótipo executável que confirme tal fato.

Já na fase de Construção, todos os demais componentes e funcionalidades não incluídas durante a fase de Elaboração são desenvolvidos e integrados ao produto. Nesta fase, é dada ênfase ao gerenciamento de recursos e ao controle das operações a fim de aperfeiçoar os custos, o cronograma e a qualidade do projeto. Ao seu final, o produto desenvolvido deverá estar estável e maduro o suficiente para ser implantado na comunidade de usuários.

Por fim, a fase de Transição é responsável por disponibilizar o software na comunidade de usuários. Esta fase compreende várias iterações, incluindo a disponibilização de versões beta, o conserto de falhas existentes e a implementação de

melhorias, sendo concluída quando a versão de implantação atingir o conjunto completo de funcionalidades acordadas pelos *steakholders* no início do projeto.

Assim como alguns outros processos de desenvolvimento, o RUP é capaz de descrever ‘quem’ está fazendo ‘o que’, ‘como’ e ‘quando’ através de quatro elementos primários: papéis, atividades, artefatos e fluxo de atividades. Os papéis definem as responsabilidades de cada indivíduo dentro da equipe, as atividades expressam o comportamento que cada papel realiza, os artefatos correspondem ao produto criado por cada papel e um fluxo de atividade é caracterizado por uma seqüência de atividades que resultam em um valor observável e mostram interações entre os papéis [3].

Para organizar as diversas atividades desempenhadas pelos vários papéis existentes, as atividades são agrupadas dentro de nove disciplinas. São elas: Modelagem de Negócio, Requisitos, Análise e Projeto, Implementação, Teste, Implantação, Gerenciamento de Configuração e Mudança, Gerenciamento de Projeto e Ambiente.

O trabalho em questão busca contribuir com uma maior automação da atividade Projetar Arquitetura da disciplina de Análise e Projeto. Por essa razão, apenas a disciplina citada será detalhada a seguir.

2.2 Disciplina de Análise e Projeto

A disciplina de Análise e Projeto desempenha um papel fundamental ao longo de todo o ciclo de vida do processo, perdurando desde a fase de Concepção até a fase de Construção. Seu propósito é traduzir os requisitos do sistema em uma especificação capaz de descrever a sua implementação [3]. Para isso, é preciso entender os requisitos do projeto, estabelecer uma arquitetura robusta de forma a tornar fácil o entendimento do sistema e a sua construção, e ajustar a arquitetura desenvolvida de forma a atender os requisitos não-funcionais, tais como, *performance*, robustez, escalabilidade e testabilidade.

Para cumprir com seu propósito, a disciplina utiliza-se dos papéis do arquiteto, responsável por comandar e coordenar atividades técnicas e artefatos durante o projeto, e do projetista de software, cujo trabalho consiste em definir como as classes existentes se ajustam ao ambiente de desenvolvimento através da análise e projeto de casos de uso e subsistemas. Pode-se incluir opcionalmente um projetista de bando de dados, responsável por propor um modelo objeto-relacional que atenda às necessidades de persistência do projeto, e revisores, especialistas que revisam os principais artefatos produzidos pela disciplina.

A Figura 2-2 ilustra o fluxo e os detalhes de fluxo da disciplina de Análise e Projeto definido pelo RUP.

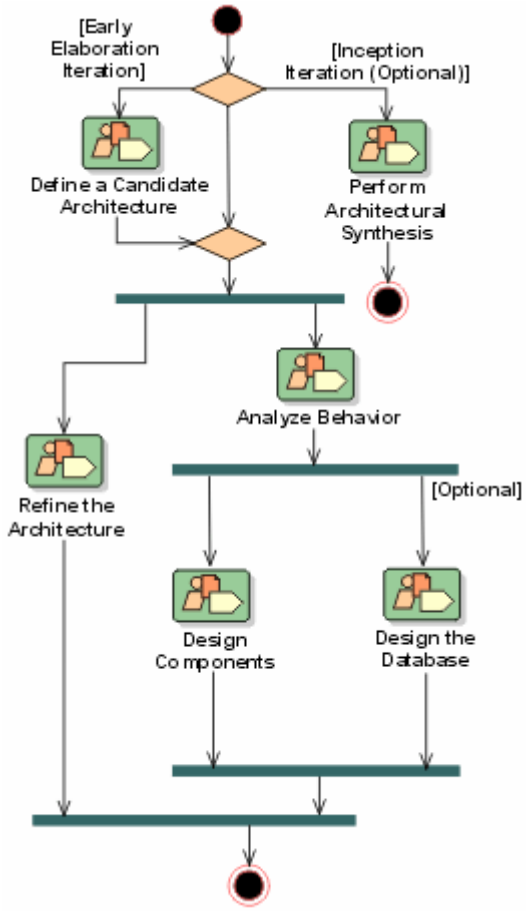

Figura 2-2 Fluxo da disciplina de Análise e Projeto

Na fase de Concepção, a disciplina está preocupada em estabelecer se o sistema é viável do ponto de vista técnico através da análise das tecnologias envolvidas na sua construção. Portanto, durante o detalhe de fluxo Realizar Síntese Arquitetural busca-se mostrar a existência de uma solução que satisfaça os requisitos acordados entre os *steakholders* do projeto.

Dado que o sistema é viável do ponto de vista técnico, a disciplina foca na criação de uma arquitetura inicial para o projeto, a fim de prover um ponto de partida para o trabalho de análise. Assim, durante a fase de Elaboração, o detalhe de fluxo Definir uma Arquitetura tem por responsabilidade criar um esqueleto da arquitetura e identificar classes de análise para cada caso de uso especificado.

Além dos casos de uso, é de fundamental importância mapear a análise do comportamento requerido pelo sistema. Por esta razão, realiza-se em seguida o detalhe

de fluxo Analisar Comportamento, cuja função é criar classes de análise que expressam o comportamento do sistema e determinar como tais classes se ajustam à arquitetura.

Com a análise concluída, é necessário refinar o modelo criado de forma a prover uma transição natural entre as atividades de análise e atividades de projeto. Assim, realiza-se o detalhe de fluxo Refinar a Arquitetura, buscando identificar os elementos de projeto a partir dos elementos de análise existentes, bem como maximizar o reuso de componentes e elementos de projeto disponíveis.

Por fim, realiza-se o detalhe de fluxo Projetar Componentes, produzindo um conjunto de componentes que provêm o comportamento apropriado para satisfazer os requisitos do sistema. Para tal, soma-se o trabalho de refinar a definição dos elementos de projeto e atualizar a realização dos casos de uso baseado nos novos elementos identificados. Caso o sistema incluia um banco de dados, o detalhe de fluxo Projetar Bando de Dados acontece em paralelo.

Após a conclusão do fluxo da disciplinada de Análise e Projeto, obtém-se todos os componentes a serem implementados no futuro e necessários para a conclusão e o sucesso do projeto.

2.3 Atividade Projetar Arquitetura

Apesar de constituir uma única disciplina, a Análise e Projeto pode ser claramente dividida em duas fases. A primeira constitui a fase de análise, a qual se baseia nos casos de uso para mapear os requisitos em um conjunto de classes e subsistemas. A segunda, a fase de projeto, utiliza os resultados obtidos na fase anterior e os adapta em função dos requisitos não-funcionais e do ambiente de desenvolvimento do sistema.

O detalhe de fluxo Refinar Arquitetura consiste no divisor de águas entre as duas fases, pois é durante a sua realização que os elementos de análise mapeados na arquitetura inicial são definitivamente projetados. Para isso, o detalhe de fluxo citado é subdividido em cinco atividades: Identificar Mecanismos de Projeto, Identificar Elementos de Projeto, Identificar Reuso, Projetar Cápsulas e Projetar Distribuição.

A atividade Identificar Elementos de Projeto, ou simplesmente Projetar Arquitetura, merece destaque entre as demais. Isso porque a segunda atividade do detalhe de fluxo Refinar Arquitetura é responsável por analisar as interações das classes de análise, identificar elementos de projeto e introduzir padrões [6] à arquitetura,

correspondendo, assim, à grande parte do tempo desprendido durante o detalhe de fluxo em questão.

Ao término da atividade Projetar Arquitetura, classes de análise evoluem para um conjunto de diferentes elementos de projeto, podendo ser [5]:

- Classes de projeto: representam, em vez de uma única, um conjunto de responsabilidades.
- Subsistemas: representam um conjunto de classes relacionadas e independentes de outros subsistemas.
- Interfaces: representam declarações abstratas de responsabilidades providas por uma classe ou subsistema.

Durante esta atividade fica evidente a importância da utilização de modelos visuais, denominados diagramas, para a definição do projeto arquitetural. O modelo será fundamental para apresentar os elementos necessários para a construção do sistema, expressar o seu comportamento e prever possíveis erros de implementação.

Entretanto, com o aumento da complexidade dos sistemas de software atuais, a criação do projeto arquitetural, bem como a sua manutenção ao longo de todo o processo de engenharia de software, tornou-se uma atividade extremamente longa e repetitiva, resultando muitas vezes no seu abandono. Logo, automatizar a atividade Projetar Arquitetura vem a ser fundamental para aumentar a produtividade da equipe de desenvolvimento e evitar a sua descontinuação durante o processo.

A Figura 2-3 e Figura 2-4 ilustram o trabalho realizado pela atividade Projetar Arquitetura. Com um exemplo relativamente simples, já é possível perceber a quantidade de operações necessárias no sentido de refinar um modelo de análise para um de projeto. Assim, o trabalho proposto foca em automatizar operações ilustradas pelo exemplo, tais como o *merge, split* e generalização de classes, o projeto de subsistemas e a introdução dos padrões *Singleton* [6], *Persistent Data Collection* [7] e *Facade* [6].

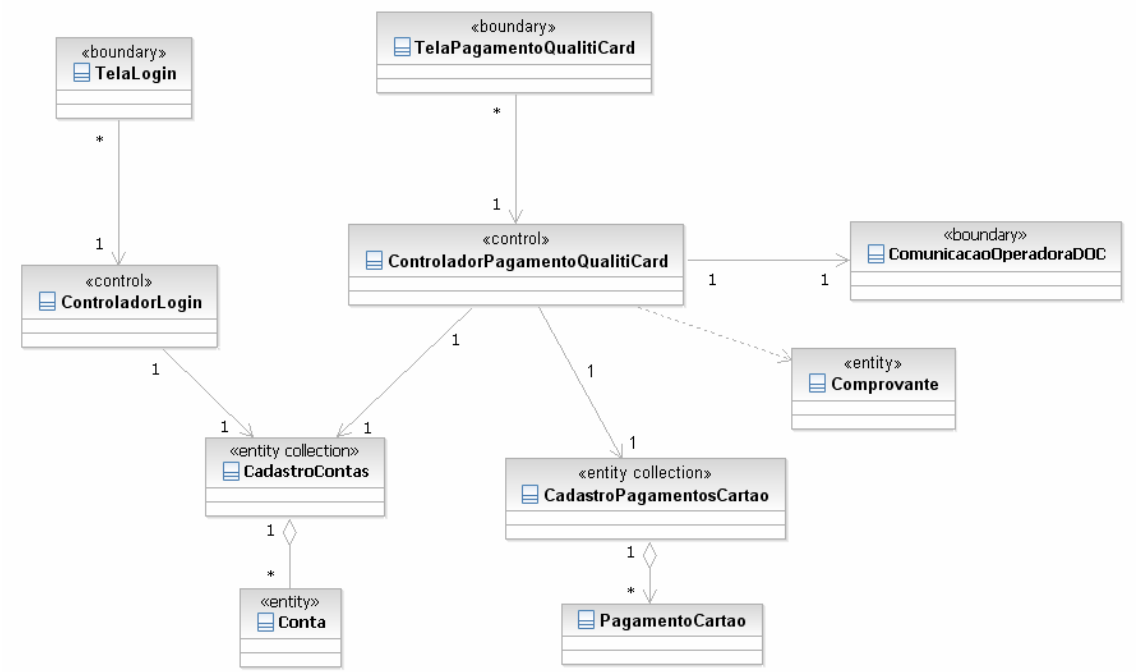

Figura 2-3 Modelo de análise

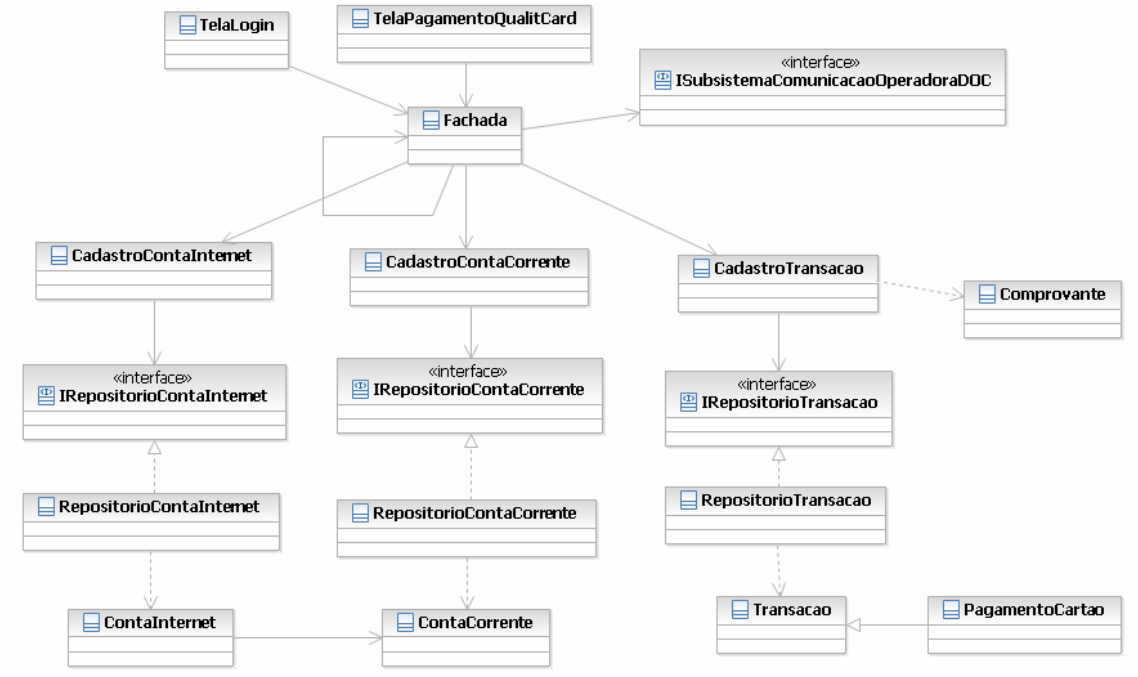

Figura 2-4 Modelo de projeto

3. Ferramentas de Modelagem UML

Ferramentas de modelagem UML se mostram essenciais na criação de modelos e no desenvolvimento da atividade Projetar Arquitetura da disciplina de Análise e Projeto do RUP. Entretanto, apesar do conjunto de funcionalidades oferecidas pelas ferramentas disponíveis no mercado e dos esforços realizados no sentido de oferecer um suporte automatizado na manutenção de modelos, tal atividade permanece de forma sistemática, sem um apoio automatizado para todo o processo. Com o objetivo de agilizar essa fase do processo, foram analisadas algumas das ferramentas de modelagem utilizadas nos dias atuais.

3.1 Rational Rose Enterprise

Umas das ferramentas de maior sucesso, o Rational Rose Enterprise pode ser definido como uma ferramenta UML de projeto de software orientado a objetos, voltado para a sua modelagem visual e construção de componentes. A ferramenta criada pela Rational, e posteriormente adquirida pela IBM, provê uma linguagem de modelagem comum que permite a criação rápida de software de qualidade, seja ele em Ada, ANSI C++, C++, CORBA, Java™, J2EE™, Visual C++® and Visual Basic® [8].

O Rose oferece suporte à UML, análise e geração de código, visualização e modelagem de aplicações Web e de Banco de Dados, além de se integrar perfeitamente com outras ferramentas da IBM, como o IBM Rational ClearCase®. Por essa razão, o Rose constitui uma das ferramentas de modelagem mais completas, justificando o seu alto custo de aquisição.

Apesar de todas as suas virtudes, no quesito de automação arquitetural o Rose deixa a desejar. Operações bastante exigidas nessa atividade, como *split*, merge e generalização de classes, não são suportadas, acontecendo o mesmo com a introdução de padrões arquiteturais [6]. Entretanto, caso o usuário opte por estender funcionalidades da ferramenta, a mesma foi projetada para atender tal necessidade.

Para isso, o Rational Rose Enterprise dispõe de uma interface de extensão responsável por interagir com o modelo desenvolvido e permitir operações de consulta e modificação do mesmo. Dessa forma, torna-se possível adicionar à ferramenta novas funcionalidades e, conseqüentemente, atribuir automações à atividade Projetar Arquitetura. A Rose Extensibility Interface [9] provê procedimentos para customização

e extensão de menus, automatização de funções da ferramenta, execução de funções do Rose através de outra aplicação e criação de *add-ins*.

Para suportar a extensibilidade descrita, o Rose é constituído por componentes definidos em sua própria interface, como mostra a Figura 3-1.

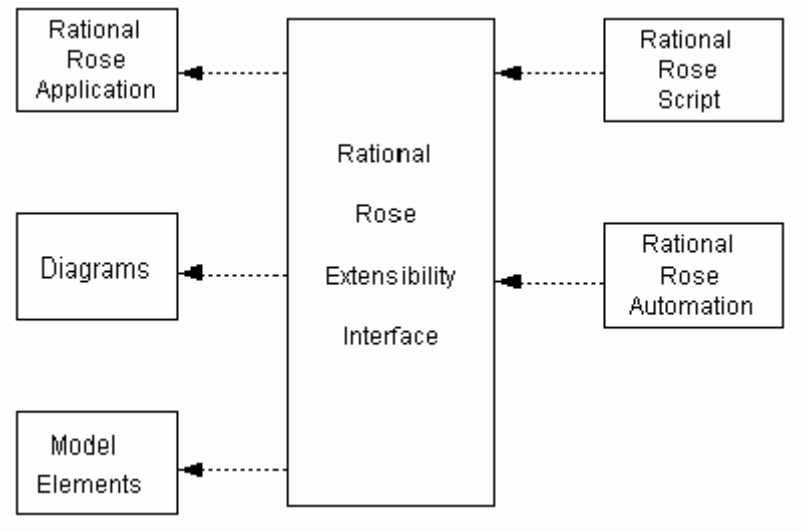

Figura 3-1 Rose Extensibilty Interface

Infelizmente, apesar da baixa complexidade envolvida na criação de *add-ins* para a ferramenta, seu processo de desenvolvimento utiliza uma linguagem ultrapassada e descontinuada, o Visual Basic. A mesma foi substituída pelo VB.NET e a sua integração com o novo .NET framework da Microsoft passou a ser um impedimento na extensão do Rose. Algumas tentativas no sentido de criar extensões foram realizadas, tendo sucesso apenas aquelas que se utilizavam da antiga linguagem Visual Basic, constituindo um ponto negativo para a utilização da ferramenta no projeto proposto.

3.2 JUDE/ Community

Baseada no conceito de "Usável desde a sua Instalação", a JUDE/Community é uma das mais poderosas ferramentas de modelagem UML grátis disponíveis. Sendo rica em funcionalidades, ela oferece edição e impressão de diagramas UML 2.0, importação/exportação de código-fonte Java, criação de *Javadoc* a partir de diagrama de classes e disponibilização de gráficos [10]. Por ser isenta de custos e bastante funcional, a ferramenta tem se mostrado uma boa opção, principalmente, para a comunidade acadêmica e empresas de pequeno porte.

Criada pela Change Vision, a JUDE/Community foi desenvolvida para a modelagem de sistemas Java, disponibilizando a seus usuários classes e interfaces dos

pacotes java.lang e java.util. Assim, é possível agregar aos modelos algumas das funcionalidades de Java 5, tais como classes genéricas, listas, pilhas, árvores, dicionários, entre outras. Todavia, ao optar apenas pela modelagem de sistemas Java, a ferramenta analisada restringiu significativamente seu mercado, fazendo com que equipes de sistemas desenvolvidos em outras linguagens utilizassem ferramentas alternativas.

A Figura 3-2 ilustra o ambiente oferecido pela ferramenta JUDE/Community.

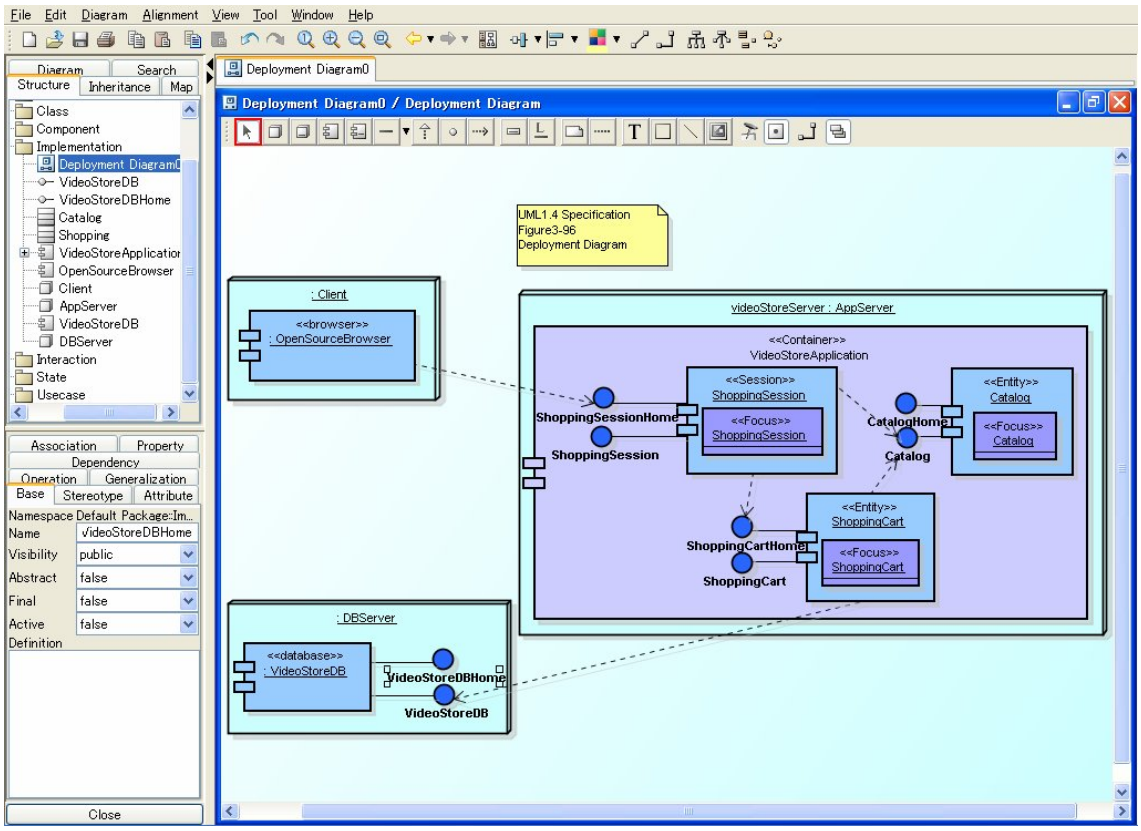

Figura 3-2 Ambiente JUDE/Community

Mesmo sendo bastante poderosa, a JUDE/Community esbarra nas mesmas ineficiências do Rational Rose. Na etapa de projeto do sistema, a ferramenta não apresenta nenhum tipo de automação no sentido de criar classes de projeto, projetar subsistemas ou introduzir padrões. Todas essas atividades, repetidas constantemente ao decorrer do detalhe de fluxo Refinar Arquitetura, são feitas manualmente, refletindo uma natural redução da produtividade da equipe.

Uma solução seria a extensão de suas funcionalidades a fim de adicionar-lhe as automações necessárias. Apesar de a ferramenta disponibilizar uma API que permite o acesso ao modelo e a seus elementos, a JUDE/Community não oferece suporte à criação de *add-ins*, ou seja, não é possível estendê-la. A JUDE API provê apenas a edição de

modelos através de outras aplicações de software, fora do seu ambiente de desenvolvimento, tornando inviável a criação de automações para a ferramenta em questão.

3.3 StarUML

StarUML é um projeto *open-source* para o desenvolvimento rápido, flexível e extensível de modelos UML [11]. Seu objetivo é prover uma ferramenta de modelagem de software aliada a uma plataforma de substituição de ferramentas comerciais, tais como Rational Rose, Together, entre outras. Apesar de ser desenvolvido quase que em sua totalidade na linguagem Delphi, StarUML é um projeto multi-linguagem, não estando preso a uma linguagem de programação específica e podendo ser desenvolvido nas linguagens C/C++, Java, Visual Basic, Delphi, JScript, VBScript, C#, VB.NET.

Criada em 1996, a ferramenta estudada tem como características o suporte a diagramas UML 2.0 e à tecnologia MDA (*Model Driven Architecture*), geração e engenharia reversa de código Java, C++ e C#, extensão de diagramas, geração de documentos *Microsoft Office* e suporte a padrões de projeto GOF e EJB. Além das funcionalidades providas, a ferramenta oferece a oportunidade de extensão através de uma API aberta, cadastro de eventos e uma arquitetura baseada em *plug-ins*.

A figura 3-3 ilustra o ambiente de desenvolvimento do projeto StarUML.

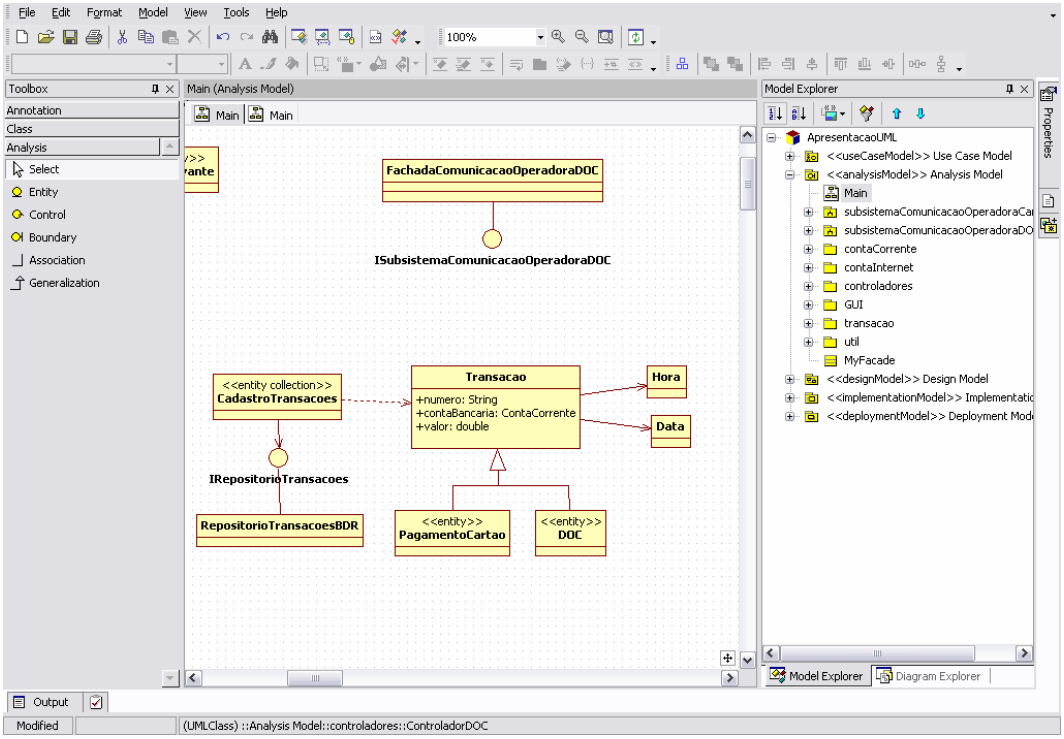

Figura 3-3 Ambiente StarUML

Apesar de constituir um projeto *open-source*, a ferramenta StartUML se mostra bastante completa, atendendo a necessidades básicas de criação de diagramas de classes, bem como a necessidades mais avançadas como a criação de diagramas de composição e *deployment*, a compatibilidade com outras ferramentas de modelagem e a introdução de padrões ao modelo. Tal fato pode ser comprovado pelos estudos de caso que a utilizam, envolvendo instituições como a *Electronics and Telecommunications Research Institute*, *SK Telecom e Asian Medial Center* [11].

Porém, assim como as demais ferramentas analisadas, a StarUML se mostra pouco eficiente no sentido de automação do projeto arquitetural. Isso porque as principais funcionalidades requeridas por tal atividade não são contempladas de forma automatizada. Assim, apesar do suporte diferenciado a padrões provido pela ferramenta, a atividade Projetar Arquitetura permanece sistemática e responsável por grande parte do tempo desprendido pelo detalhe de fluxo Refinar Arquitetura da disciplina de Análise e Projeto do RUP.

Na tentativa de estender a ferramenta, composta por módulos independentes, o trabalho esbarrou na linguagem Delphi, utilizada na criação de *add-ins*. Aliado ao aprendizado de uma nova linguagem, o processo de desenvolvimento de *add-ins* para a StarUML envolve a criação de esquemas XML, o registro dos esquemas criados e o cadastramento de eventos ligados ao modelo. Assim, a tarefa de estender a ferramenta se mostrou um enorme esforço e inviável para o tempo de execução do trabalho a ser apresentado.

3.4 Borland Together

Outra ferramenta bastante conhecida e utilizada é o Borland Together, uma plataforma de modelagem visual desenhada para suportar arquitetos, desenvolvedores, designers de UML, analistas de processos de negócios e modeladores de dados na criação acelerada de aplicações de software de alta qualidade [12]. Seja na mudança de processos de negócios, criação de novas aplicações ou extração de informação de design de sistemas existentes, o Together dá a todos os participantes uma compreensão comum das importantes decisões sobre o design da arquitetura do software.

A ferramenta provê, entre outras funcionalidades, o suporte multi-linguagem (Java, C++ e CORBA/IDL), a criação de modelos UML 2.0 e de processos de negócio, a revisão dos modelos e código-fonte através de auditorias e métricas, a definição e implementação de *Domain Specific Languages* [13] e o gerenciamento da

rastreabilidade entre o código-fonte da aplicação e seus requisitos. Diferentemente das ferramentas estudadas até o momento, o Together utiliza a plataforma Eclipse [14] como seu ambiente de desenvolvimento e realiza uma abordagem focada nas atividades da equipe do projeto como um todo, envolvendo desenvolvedores, arquitetos de software, projetista de banco de dados e gerentes de qualidade e projeto.

A seguir, a Figura 3-4 ilustra o ambiente do Together.

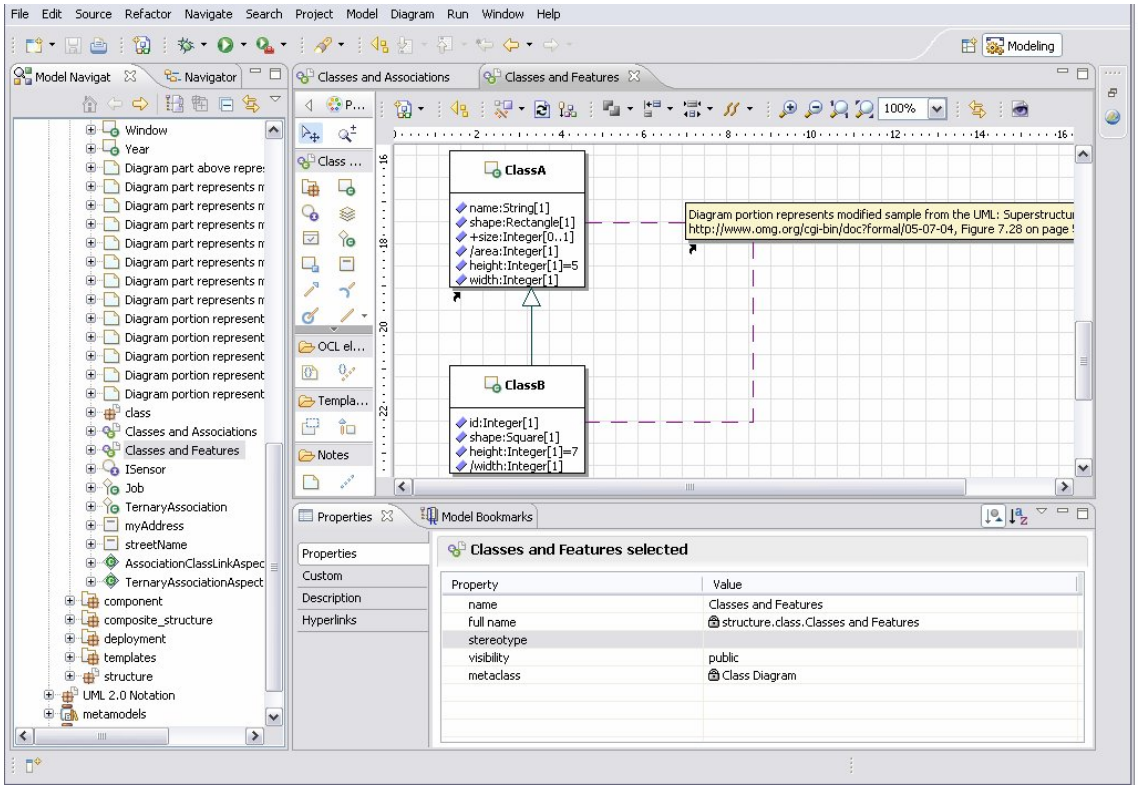

Figura 3-4 Ambiente Borland Together

A ferramenta apresenta uma integração diferenciada entre os modelos UML e o código-fonte gerado, já que o ambiente utilizado para a modelagem do sistema é o mesmo utilizado para a sua codificação. Assim, na medida em que classes são criadas para integrar os diversos diagramas suportados, seu código pode ser visualizado em abas paralelas, incluindo seus respectivos métodos e atributos.

O Borland Together oferece também um suporte bastante interessante a padrões de projeto. A ferramenta possui um repositório de padrões GOF possíveis de serem integrados ao modelo UML construído. Entretanto, a abordagem escolhida não foca na automação de um projeto de arquitetural, ou seja, a funcionalidade de adição de padrões altera o modelo de forma parcial, necessitando de alterações manuais para satisfazer a criação de um modelo de projeto a partir de um modelo de análise.

Essa ineficiência pode ser notada na introdução do padrão *Facade*, que leva em consideração apenas os subsistemas da fachada a ser criada, deixando de lado seus respectivos clientes. A Figura 3-5 e a Figura 3-6 demonstram um exemplo antes e depois da introdução automática do padrão *Facade* pelo Together. Uma implementação satisfatória deveria substituir as relações existentes entre os subsistemas e seus clientes pelas relações entre clientes e fachada e entre a fachada e os subsistemas.

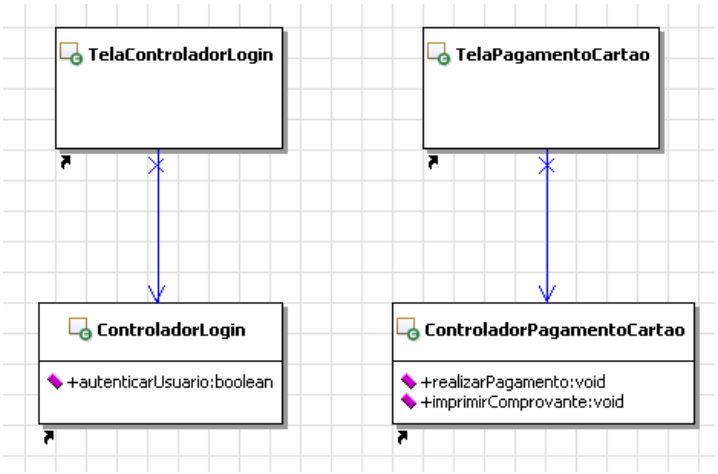

Figura 3-5 Modelo UML isento de padrão

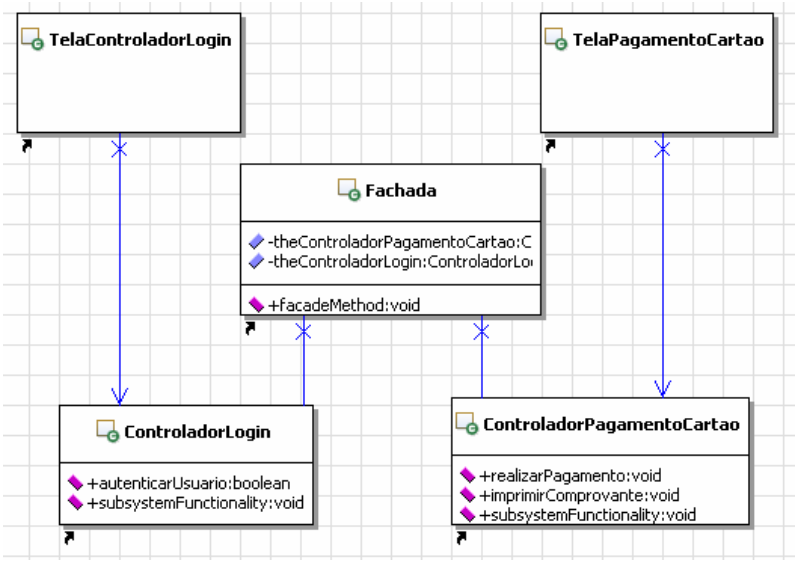

Figura 3-6 Introdução do padrão *Facade* pelo Together

Além da ineficiência apresentada na introdução de padrões, o mesmo pode ser percebido com as funcionalidades de projetar subsistemas e de realizar *merge*, *split* e generalização de classes. Tais automações seriam possíveis apenas com a criação de *plug-ins*, contudo o Together não disponibiliza uma API que inclua extensões de

funcionalidades aos modelos UML existentes, resultando no abandono dos estudos em cima da ferramenta.

3.5 Rational Software Architect

A última ferramenta de modelagem analisada, o Rational Software Architect, integra a família do software Rational: uma plataforma para governar o desenvolvimento de software e sistemas em um mundo "*on demand*" [15]. Para concluir com seu objetivo, a família Rational foi dividida em categorias, sendo cada uma responsável por determinada fase no ciclo de vida de um projeto. As categorias são:

- Análise, modelagem, design e construção: auxiliam na criação e construção de aplicativos gerados por modelo, arquiteturas resilientes para SOA, especificações de programação, processos de dados e negócios e regras de negócios.
- Gerenciamento de requisitos: ferramentas integradas para o gerenciamento de requisitos, desenvolvimento de casos de uso, modelagem de negócios e modelagem de dados.
- Gerenciamento de configuração, mudança e release: gerenciamento de ativos de ciclo de vida, incluindo-se a automação de processos, controle de mudanças, gerenciamento de build, rastreabilidade e relatórios.
- Linguagens de programação tradicional e compiladores: ferramentas baseadas nas linguagens 3GL e 4GL/RAD e ambiente de desenvolvimento unificado.
- Gerenciamento de processos, projetos e portfólios: ferramentas integradas para gerenciamento de requisitos, modelos e teste, para implementar um processo de desenvolvimento e avaliar e emitir relatórios de progresso.
- Suítes de desenvolvimento de software: oferecem suporte ao processo de desenvolvimento e implementação do software.
- Gerenciamento de qualidade de software: ferramentas para todas as dimensões de qualidade de software: funcionalidade, confiabilidade e desempenho no desenvolvimento e produção.

O Architect, que em sua versão mais recente foi renomeado para Rational Software Architect for WebShere Software, integra a categoria de Análise, modelagem, design e construção, fornecendo suporte integrado para o desenvolvimento orientado à modelo com UML e criação de aplicativos e serviços bem arquitetados.

Assim como o Borland Together, a ferramenta analisada optou por estender o ambiente de desenvolvimento do Eclipse, aproximando definitivamente o projeto de um sistema de sua implementação. Entre suas funcionalidades estão o suporte a UML 2.0 e à modelagem de linguagem específicas (DSLs), integração com o IBM WebShere Business Modeler, introdução e criação de padrões de projeto, gerenciamento de riscos, rastreabilidade dos requisitos e geração de código. Dessa forma, o Architect se apresenta como uma ferramenta focada não só na modelagem visual do sistema, mas também em atividades de gerenciamento, desenvolvimento, qualidade e testes.

A Figura 3-7 demonstra o ambiente de desenvolvimento do Rational Software Architect.

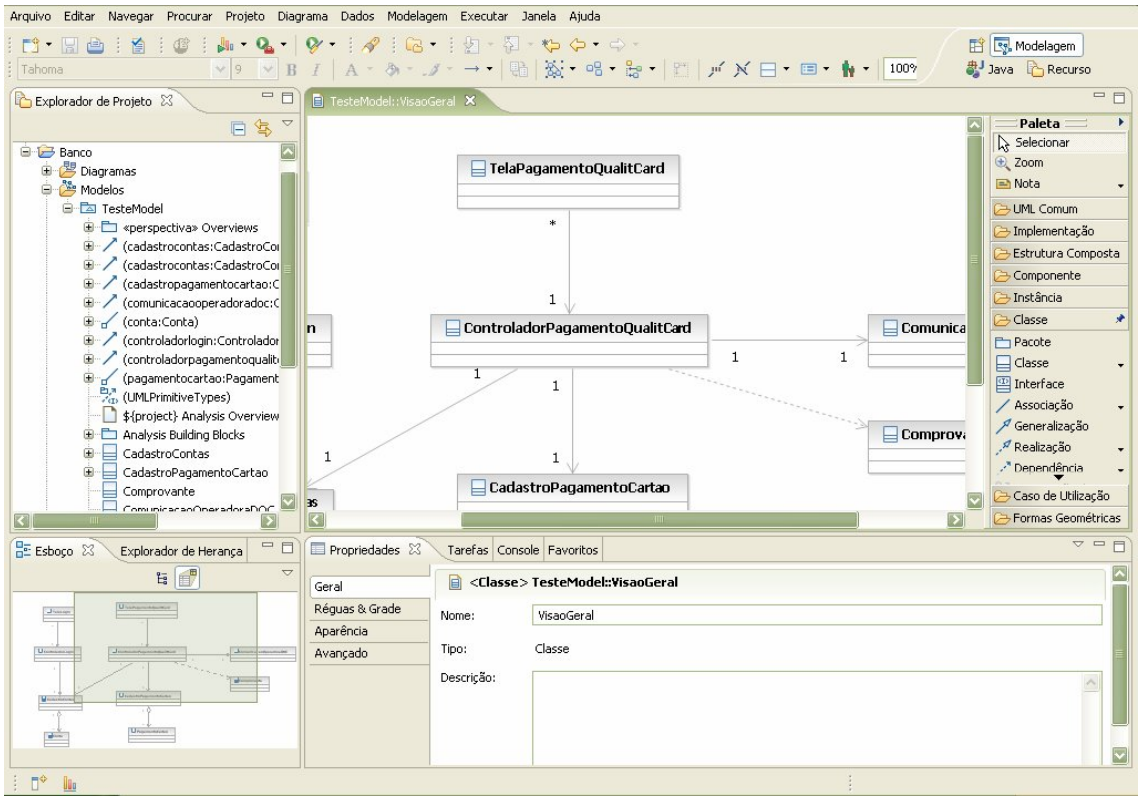

Figura 3-7 Ambiente Rational Software Architect

Seguindo a linha das demais ferramentas estudadas, o Architect não disponibiliza automações na construção do projeto de um sistema. Mesmo possuindo uma vasta quantidade de funcionalidades, atalhos diversos e a opção de introdução e criação de padrões arquiteturais, a abordagem escolhida pela ferramenta não agiliza de forma satisfatória a transição de um modelo de análise para um de projeto. Ou seja, atividades como as de projetar subsistemas e refatorar classes de análise não são disponibilizadas, enquanto que a introdução de padrões é realizada de forma parcial, semelhantemente aos exemplos apresentados pela Figura 3-5 e Figura 3-6.

Contudo, um diferencial do Architect consiste na extensão de suas funcionalidades. A ferramenta criou o conceito de *pluglets*: programas Java, desenvolvidos no próprio ambiente do Eclipse, voltados exclusivamente para estender funcionalidades relacionadas aos modelos e diagramas UML. Através de *pluglets* é possível consultar informações dos modelos, criar novas classes, interfaces e relacionamentos, bem como criar e alterar diagramas, sejam eles de classe, seqüência ou colaboração.

Devido a este diferencial, o Rational Software Architect se mostrou a ferramenta ideal para o desenvolvimento do trabalho proposto, cujo objetivo é a extensão da ferramenta de modelagem citada para incorporar funções de automação arquitetural, de forma a agilizar a atividade Projetar Arquitetura e evitar a sua descontinuação no processo de engenharia de software do RUP.

4. Solução Proposta

Dada a importância de se automatizar a atividade Projetar Arquitetura, discutida na seção 2.3, e o estudo das ferramentas de modelagem UML realizado no capítulo anterior, o trabalho em questão propõe uma solução que consiste na extensão da ferramenta Rational Software Architect, a fim de lhe acrescentar funcionalidades focadas na criação de projetos arquiteturais de forma automatizada. Para isso, o trabalho considera as funcionalidades de *refactoring* e generalização de classes de análise, projeto de subsistemas e introdução de padrões de projeto ao modelo.

No *refactoring* de classes de análise, são automatizadas as funções de *split* e *merge* das classes existentes. Esta funcionalidade será responsável pela agregação e/ou separação dos membros (atributos, métodos, relacionamentos) de uma classe, bem como pela resolução de possíveis conflitos envolvidos nessas operações. Dessa forma, evita-se que o projetista de software realize manualmente as operações citadas, resultando em uma maior facilidade e rapidez na criação de um projeto arquitetural. A Figura 4-1 e a Figura 4-2 ilustram o *refactoring* de classes.

No primeiro caso, as classes Fachada, ControladorLogin e ControladorPagamentoCartao são combinadas e resultam em única classe Fachada. Ou seja, a funcionalidade dos dois controladores é adsorvida pela classe Fachada. No segundo exemplo, a classe Conta é divida em uma classe que representa uma conta de acesso (ContaInternet) e outra que representa uma conta bancária (ContaCorrente); similarmente, o cadastro de contas é separado em dois cadastros.

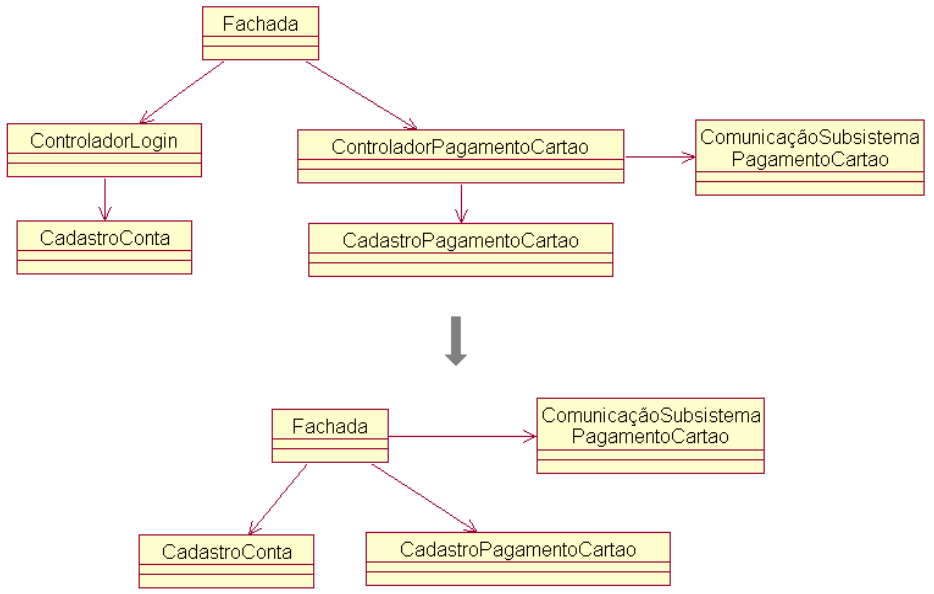

Figura 4-1 Marge de classes

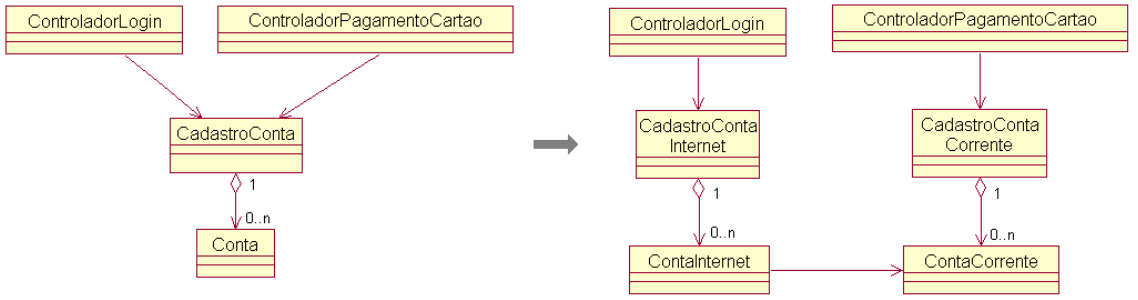

Figura 4-2 Split de classes

Com os exemplos acima, é possível notar as várias operações a serem desempenhadas durante o processo de *refactoring*, tais como a criação de novas classes, a remoção de classes existentes e a passagem de métodos, atributos e relacionamentos de uma classe para a outra. Com o intuito de amenizar tal ineficiência do processo, a extensão criada pela solução proposta permite a automação dessas operações, podendo-se obter o mesmo resultado de forma muito mais ágil.

Na generalização de classes, busca-se automatizar a introdução de superclasses ao modelo. Apesar de ser uma atividade simples, a criação de generalizações envolve, em alguns casos, uma série de passos sistematizados, como a criação da superclasse, seus relacionamentos e seus membros. Com esta automação, os passos envolvidos neste processo serão reduzidos a uma simples interação com a ferramenta, resultando na criação de generalizações com maior rapidez. A Figura 4-3 ilustra a generalização de classes.

Neste exemplo, é introduzida uma classe abstrata para representar as transações realizadas pelo cliente da aplicação bancária e o PagamentoCartao passa a ser um dos tipos de transação.

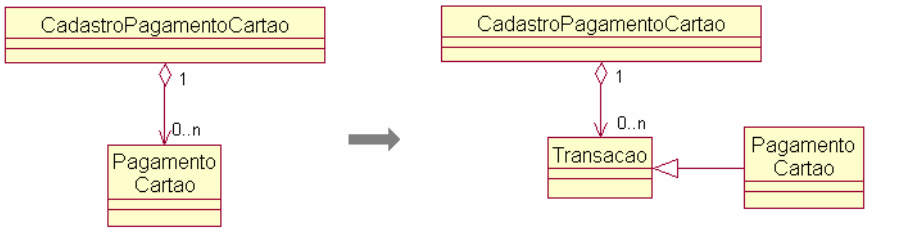

Figura 4-3 Generalização de classes

Outra automação é realizada na atividade de projetar subsistemas. Nela, as operações de criação do pacote, da interface e da fachada do subsistema, bem como a interação desses elementos no modelo são de responsabilidade da solução proposta. Ou seja, não só os novos elementos serão criados, como também incorporados ao modelo

com seus respectivos relacionamentos, buscando reduzir de forma significativa o número de interações entre projetista e diagramas. A Figura 4-4 ilustra o projeto de subsistemas, com a classe ComunicacaoSubsistemaPagamentoCartao se transformando em um subsistema, o qual é representado no diagrama pela interface e por uma classe fachada que implementa a interface.

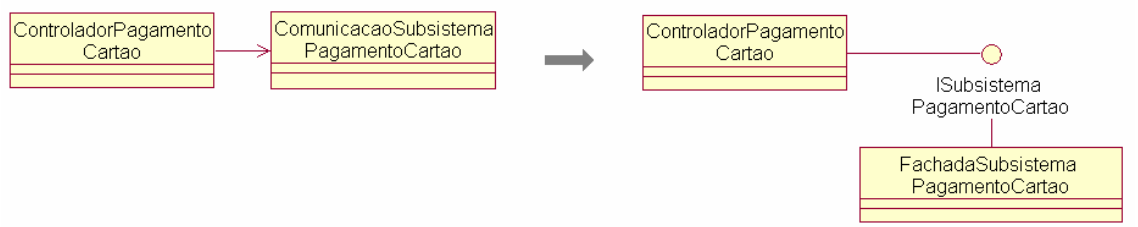

Figura 4-4 Projeto de subsistemas

Finalmente, a solução proposta busca automatizar o processo de introdução dos padrões *Singleton*, *Facade* e *Persistent Data Collection* através da criação das classes e/ou relacionamentos envolvidos em cada um dos padrões citados. Para o padrão *Singleton*, a solução adiciona ao diagrama uma associação na classe participante, o atributo privado estático e o método público estático, que retorna a instância da classe, conforme mostra a Figura 4-5.

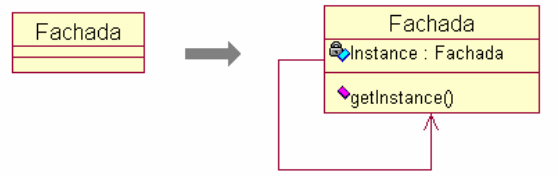

Figura 4-5 Padrão *Singleton*

Para o padrão *Facade*, a solução será responsável por acrescentar uma fachada ao modelo, respeitando os relacionamentos entre seus clientes e subsistemas, bem como incorporando as operações dos subsistemas à fachada recém criada. A Figura 4-6 ilustra a introdução do padrão *Facade*.

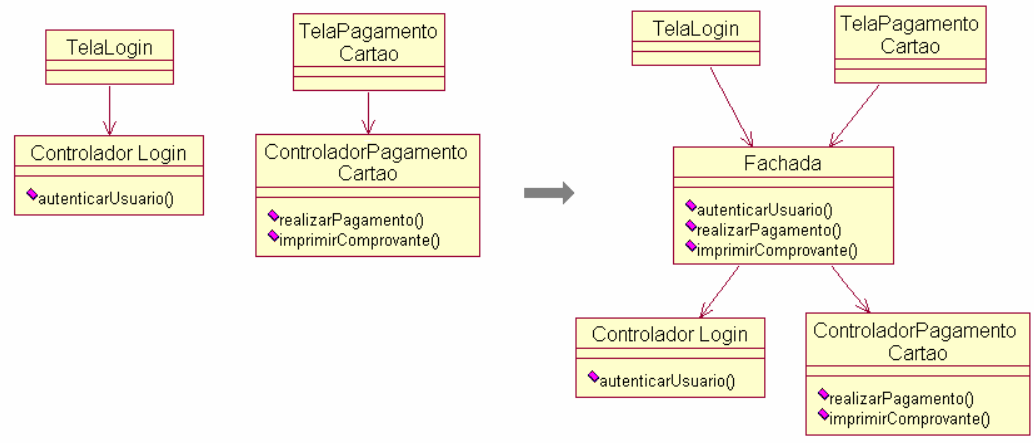

Figura 4-6 Padrão *Facade*

Por fim, no padrão *Persistent Data Collection* a solução adiciona ao diagrama as classes e interfaces envolvidas no padrão citado, garantindo a consistência do modelo. Em outras palavras, responsabilidades do padrão já incorporadas não são adicionadas e relacionamentos pré-existentes não são substituídos. A Figura 4-7 exemplifica a introdução do padrão PDC.

O diagrama à esquerda da figura, antes da transformação, apresenta uma visão de análise onde o relacionamento entre o cadastro e a entidade Conta é de agregação. No projeto, com a decisão de abstrair uma possível forma de persistência (por exemplo, em um banco de dados relacional) da visão de negócio, é criada uma interface entre o cadastro e o repositório propriamente dito, conforme mostra o diagrama à direita da figura. Como o repositório não armazena efetivamente objetos do tipo Conta, mas apenas converte objetos para uma representação a ser armazenada, e vice-versa, o relacionamento com a entidade passa a ser de dependência.

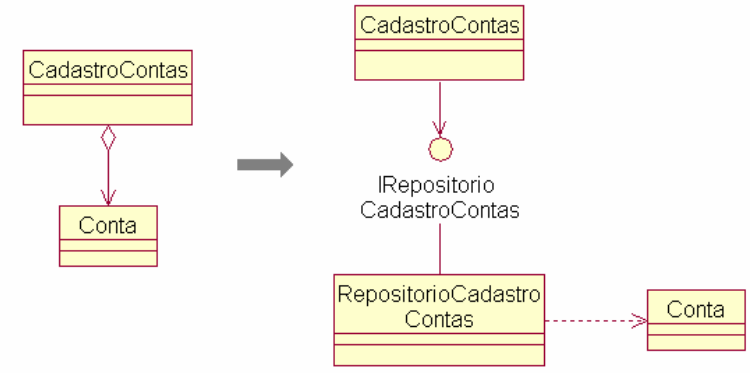

Figura 4-7 Padrão PDC

Portanto, apesar de a atividade Projetar Arquitetura envolver diversas outras transformações, o trabalho em questão optou por focar apenas nestas apresentadas devido a restrições de tempo. Porém, conforme discutido no Capítulo 6, com essas

poucas automações criadas, já é possível perceber uma maior facilidade no desenvolvimento do projeto de um sistema, bem como um notável ganho de tempo na sua construção.

Desta forma, atinge-se o principal objetivo do trabalho: agilizar o processo envolvido na criação de um arquitetura de um sistema e auxiliar a manutenção do modelo criado durante todo o ciclo de vida do projeto.

5. Implementação

Para o desenvolvimento das automações propostas, o trabalho em questão utilizou o conceito de *pluglets*, criado pela IBM em sua nova plataforma Rational, e o framework de modelagem gráfica GMF, utilizado para a criação de editores gráficos. Assim, o desenvolvimento de *pluglets* permitiu criar extensões ao Software Architect, enquanto que a API de referência do GMF proporcionou a interação com os modelos e diagramas UML criados na ferramenta.

Com o objetivo de esclarecer a implementação do trabalho em questão, as seções a seguir detalham a criação de *pluglets* e a utilização do GMF.

5.1 Pluglets

Pluglets são pequenas aplicações Java utilizadas para o desenvolvimento de extensões ao ambiente de desenvolvimento da plataforma Rational [16]. Por constituir uma aplicação Java comum, *pluglets* podem ser desenvolvidos e testados na própria instância do ambiente de produção da linguagem citada. Mais do que isso, *pluglets* possuem acesso a API's de *plug-ins* do Eclipse, permitindo a interação com recursos do próprio Rational Software Architect e evitando-se a criação de um *plug-in* proprietário [17].

Para a sua criação, a própria plataforma Rational oferece um *wizard* que disponibiliza a opção de desenvolver projetos *pluglets* e suas classes. De forma semelhante a demais aplicações Java, um *pluglet* deve conter uma classe principal que implemente um ponto de entrada, além de estender a classe *com.ibm.xtools.pluglets.Pluglet*. A Figura 5-1 ilustra o exemplo de um *pluglet* que exibe um "*Hello, World!*".

No trecho de código ilustrado, cria-se apenas umas classe `HelloWorld`, que estende a classe `Pluglet`, caracterizando o desenvolvimento de um *plug-in* para a plataforma Rational. Em seguida, é exibida uma caixa de diálogo com a mensagem "`Hello, world!`".

```
package pluglet;

import com.ibm.xtools.pluglets.Pluglet;

public class HelloWorld extends Pluglet {

    public void plugletmain(String[] args) {
        inform("Hello, world!");
    }

}
```

Figura 5-1 Exemplo *pluglet*

Após a criação de um *pluglet*, é possível executa-lo. Para tal, o usuário deve utilizar os menus da própria plataforma, navegando pelas opções Run -> Internal Tools -> Pluglet, sendo possível visualizar, dentro do próprio ambiente de produção, as funcionalidades criadas. A partir deste ponto, o desenvolvedor está livre para desenvolver a extensão desejada, bastando apenas utilizar as técnicas de orientação a objetos e, então, criar suas aplicações Java em forma de *pluglets*.

5.2 Graphical Modeling Framework

O projeto GMF provê uma ponte entre o Eclipse Modeling Framework [23] e o Graphical Editing Framework [24], criando uma infra-estrutura para o desenvolvimento de editores gráficos baseados em ambas as tecnologias [18]. Apesar de constituir um framework para o desenvolvimento de *plug-ins* gráficos para o Eclipse, o que não é o foco do trabalho apresentado, o GMF foi utilizado na criação de algumas das ferramentas de modelagem citadas no capítulo 4, entre elas o Borland Together e o Rational Software Architect. Por essa razão, o conhecimento de sua documentação foi essencial para a interação com os modelos e diagramas UML do RSA e, conseqüentemente, para o desenvolvimento da solução proposta pelo trabalho.

A documentação do projeto GMF é bastante extensa. Todavia, o trabalho focou apenas em um número restrito de classes e interfaces, estando entre as mais utilizadas as contidas no pacote *org.eclipse.gmf.runtime.notation*:

- View: representação de um objeto visão, que corresponde a abstrações centrais na notação de modelos, sendo implementadas por outras interfaces do pacote.
- Diagram: abstração de um diagrama. Através dessa interface é possível consultar os nós de determinado diagrama, bem como inserir novos e remover nós e ligações já existentes.

- Node: abstração de um nó. Com essa interface, é possível ter acesso ao elemento que compõe o nó (classe, interface, pacote), seus relacionamentos e o diagrama em que está inserido.
- Edge: abstração de uma ligação, representando a conexão entre dois outros elementos no diagrama. Por meio dessa interface, é possível ter acesso a informações sobre o relacionamento (associação, dependência, realização) de dois nós e a responsabilidade de cada nó na relação (origem, destino).

Com o conjunto das interfaces descritas já é possível interagir com os diagramas UML do RSA de forma bastante satisfatória. Isso porque a interface *Diagram* nos permite ter acesso aos elementos contidos nos diagramas, a interface *Node* disponibiliza métodos para capturar as propriedades dos nós do diagrama escolhido e a interface *Edge* provê as funcionalidades necessárias para recuperar os relacionamentos de determinado nó do diagrama.

A documentação do projeto GMF, por sua vez, utiliza funcionalidades providas pela API de *plug-ins* UML2 do Eclipse [19]. Com esta, é possível identificar abstrações essenciais para a manipulação de modelos, tais como, classes, pacotes, associações, dependências, entre outras. Assim, é possível interagir com o modelo por meio de sua própria API, bem como através de métodos existentes na documentação do GMF.

Com a utilização de ambos os projetos, é possível identificar duas diferentes perspectivas de implementação. A primeira consiste na visão de modelos, disponível por meio da API UML2 do Eclipse. Nessa perspectiva, é possível manipular modelos de forma a criar novos componentes e recuperar componentes já existentes. Assim, componentes como classes, pacotes e até mesmo diagramas são dependentes de um modelo pré-existente.

A segunda perspectiva é a visão de diagramas, provida pelo projeto GMF. Nessa visão, os diagramas são manipulados através da criação e remoção de seus elementos, onde cada elemento do diagrama encapsula um elemento do modelo. Ou seja, um nó pode encapsular uma classe, interface ou um pacote e uma ligação pode encapsular uma associação, uma dependência ou uma realização. A Figura 5-2 ilustra as duas perspectivas analisadas.

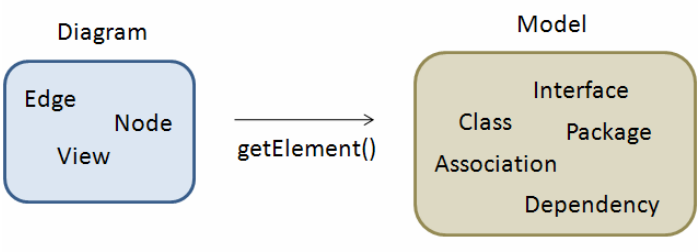

Figura 5-2 Perspectivas Diagrama x Modelo

Utilizando-se as duas APIs em conjunto, foi possível realizar alterações tanto no nível de diagramas, não influenciando no modelo propriamente dito, quanto alterações no modelo em si, afetando diretamente os seus componentes.

5.3 APAR

Feitas as explicações necessárias para o entendimento da criação de extensões no RSA, é preciso detalhar a solução utilizada no desenvolvimento das automações propostas pelo trabalho. Denominada APAR, a solução consiste de um conjunto de *pluglets*, cada um atacando uma das propostas discutidas no Capítulo 4. Os *pluglets* estão agrupados em um mesmo projeto, cuja divisão de pacotes é ilustrada pela Figura 5-3.

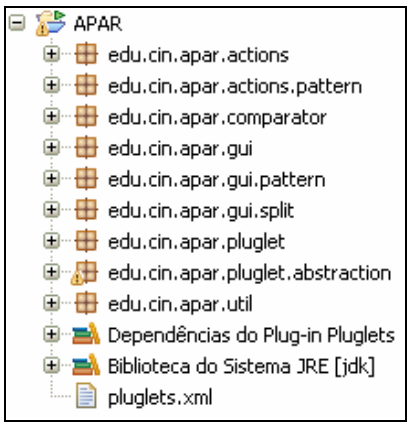

Figura 5-3 Divisão dos pacotes APAR

Os pacotes do APAR podem ser descritos da seguinte forma:

- *edu.cin.apar.pluglet*: contém todos os *pluglets* criados pela solução.
- *edu.cin.apar.pluglet.abstractions*: contém abstrações que encapsulam funcionalidades das APIs utilizadas durante o desenvolvimento.
- *edu.cin.apar.gui.**: contém a interface gráfica de cada *pluglet*.
- *edu.cin.apar.actions.**: contém as funcionalidades de automação de cada *pluglet*.

- *edu.cin.apar.comparator*: contém os comparadores para os objetos das APIs utilizadas.
- *edu.cin.apar.util*: inclui classes utilitárias para a implementação das automações.

Assim, apesar de constituírem automações independentes, os *pluglets* compartilham classes em comum tais como as utilitárias e as classes de abstrações, criando um conjunto de funcionalidades básicas a serem reusadas em outros *pluglets* que venham a ser desenvolvidos no futuro.

Para um melhor entendimento do APAR, cada *pluglet* é detalhado separadamente.

5.3.1 Criação de Generalizações

Para o desenvolvimento da criação de generalizações, o trabalho criou um *pluglet* que irá interagir com o diagrama UML de forma a criar uma superclasse que receberá as relações destinadas à sua subclasse, como mostrado no capítulo 4. Para isso, o *pluglet* desenvolvido é constituído pelas classes `CreateGeneralizationPluglet`, que corresponde à sua classe principal, `CreateGenaralizationShell`, que corresponde à sua interface gráfica, e `CreateGeneralizationAction`, responsável pela funcionalidade de automação propriamente dita.

A classe principal, que estende *com.ibm.xtools.pluglets.Pluglet*, é um pouco mais complexa que o exemplo mostrado na Figura 5-1. Sua responsabilidade consiste em capturar o nó selecionado pelo usuário, o qual corresponderá à subclasse, e inicializar a interface gráfica do *pluglet*. Caso nenhum nó seja selecionado, uma mensagem de alerta é exibida. A implementação da classe principal da automação pode ser vista na Figura 5-4.

```
package edu.cin.apar.pluglet;

import java.util.Iterator;

public class CreateGeneralizationPluglet extends Pluglet {

    public void plugletmain(String[] args) {
        try {
            String undoLabel = "Create Diagrams";
            TransactionalEditingDomain editDomain = UMLModeler.getEditingDomain();
            editDomain.getCommandStack().execute(new RecordingCommand(editDomain, undoLabel) {

                protected void doExecute() {

                    final List elements = UMLModeler.getUMLUIHelper().getSelectedElements();

                    if (elements.size() == 0) {
                        Shell shell = Display.getCurrent().getActiveShell();
                        MessageDialog.openInformation(shell, "Info", "Choose one class node to design.");
                    }

                    for (Iterator iter = elements.iterator(); iter.hasNext();) {
                        logObject(iter.next(), "");
                    }
                }
            });
        } catch (Exception e) {
            out.println("The operation was interrupted");
        }
    }
```

Figura 5-4 Classe principal

O pedaço de código acima é padrão para os *pluglets* baseados no modelo *Model Enumeration* do RSA. Portanto, todos os demais *pluglets* da solução terão suas classes principais semelhantes ao código ilustrado. Primeiramente, utiliza-se o método getSelectedElementes para capturar os elementos selecionados do diagrama, no caso a classe cuja generalização pretende-se criar. Em seguida, invoca-se o método logObject para cada elemento selecionado, onde é inicializada a interface gráfica do *pluglet*.

A interface gráfica do *pluglet* em questão é implementada pela classe CreateGeneralizationShell, a qual é inicializada conforme a Figura 5-5.

```
private void logObject(Object object, String indent) {
    if (object instanceof Node) {
        Node node = (Node) object;
        EObject eObject = node.getElement();

        if (eObject instanceof Classifier) {
            new CreateGeneralizationShell(node);
        }

    } else {
        Shell shell = new Shell();
        MessageDialog.openInformation(shell, "Info", "Choose a class.");
    }
}
```

Figura 5-5 Inicialização da interface gráfica

Desenvolvida em SWT [20], a interface gráfica permite ao usuário apenas escolher o nome da generalização a ser criada, podendo inclusive ser umas das classes

já existentes no modelo. A Figura 5-6 apresenta a GUI desenvolvida para o *pluglet* de criar generalizações.

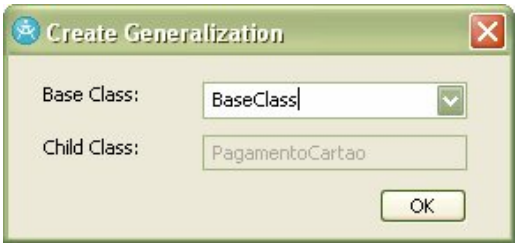

Figura 5-6 Tela Criar Generalização

Finalmente, a classe CreateGeneralizationAction realiza as operações necessárias para o desenvolvimento da automação de criar generalização. Portanto, criam-se a superclasse e a relação de generalização correspondente, bem como se transferem para a superclasse todas as ligações destinadas ao nó da subclasse. A implementação de tais operações pode ser vista na Figura 5-7.

Na figura, é possível notar a invocação dos métodos createClassElement, responsável pela criação da superclasse, createClassNode, responsável por incluir o nó correspondente à superclasse no diagrama, createGeneralization, cuja implementação cria a relação de generalização entre a superclasse e a classe filha, e createEdge, que insere a ligação entre ambas as classes no diagrama. Por fim, transfere-se todas as ligações destinadas à classe filha para a sua superclasse recém criada através do método transferTargetAssociations.

```
Class _class = modelPluglet.createClassElement(baseClassName);
Node baseNode = diagramPluglet.createClassNode(modelPluglet.getModel(), baseClassName);

Classifier childClassifier = (Classifier) childNode.getElement();
Relationship r = childClassifier.createGeneralization(_class);
diagramHelper.createEdge(childNode, baseNode, r);

Operations.transferTargetAssociations(diagramPluglet, childNode, baseNode);
```

Figura 5-7 Código de criar generalização

5.3.2 Introdução de Padrões

A introdução de padrões de projeto a diagramas UML foi implementada por um único *pluglet*, identificado por PatternPluglet. Semelhantemente ao *pluglet* anterior, a sua classe principal realiza as operações ilustradas na Figura 5-4. Inicialmente, sua diferença com relação aos demais *pluglets* está na inicialização de sua interface gráfica correspondente. Assim, ao invés de chamar CreateGeneralizationShell, o *pluglet* faz referência à classe ApplyPatternShell.

Ao ser carregada, a interface oferece ao usuário a opção de definir qual padrão de projeto aplicar: *Singleton*, *Facade* ou *Persistent Data Collection*. Escolhida a opção, o usuário será direcionado para outra tela onde então é possível definir as responsabilidades de cada classe no padrão selecionado.

Para o padrão *Singleton*, a única responsabilidade a ser configurada é a própria classe de única instância. Portanto, o usuário precisa informar apenas o nome da classe a receber o padrão, podendo ser tanto uma classe nova quanto uma já existente no modelo. A seguir, é mostrada a interface do padrão *Singleton*.

Figura 5-8 GUI *Singleton*

Depois de informar a nome da classe a receber o padrão, o *pluglet* delega as funcionalidades de criar o padrão *Singleton* para a classe `ApplySingletonAction`. Sua responsabilidade consiste em criar a classe definida pelo usuário no modelo, criar o atributo e método estáticos e definir o relacionamento da classe para si mesma. A Figura 5-9 ilustra o código utilizado para realizar tais operações.

De forma semelhante ao *pluglet* anterior, as duas primeiras linhas criam a classe e o nó onde o padrão será aplicado. Caso a classe já exista, o método `createClassElement` retorna a instância da classe existente, o mesmo acontecendo com o método `createClassNode`. Em seguida, o método `createAttribute` e `createOperation` são responsáveis por criar o atributo e método estático, respectivamente. Finalmente, cria-se a associação da classe *singleton* para ela mesma através do método `createOneToOneAssociation`.

```
Class _class = modelUtil.createClassElement(singleton);
Node classNode = diagramUtil.createClassNode(model, singleton);

ClassPluglet pcu = new ClassPluglet(_class);

Property property = pcu.createAttribute("Instance", _class);
property.setVisibility(VisibilityKind.PRIVATE_LITERAL);

Operation operation = pcu.createOperation("getInstance()", new BasicEList(), _class);
operation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
operation.setIsStatic(true);

if(!diagramUtil.hasAssociationBetweenNodes(classNode, classNode)){
    String end1Name = _class.getName().toLowerCase();
    Relationship r = Operations.createOneToOneAssociation(_class, _class, null, end1Name);
    diagramHelper.createEdge(classNode, classNode, r);
}
```

Figura 5-9 Implementação *Singleton*

No padrão *Facade*, o usuário define, através da interface gráfica disponibilizada, a classe fachada e um conjunto de pares cliente-subsistema. Cada cliente corresponderá à origem de uma relação com a fachada, enquanto que cada subsistema corresponderá ao destino. Assim, é possível substituir a relação existente entre um cliente e um subsistema por dois novos relacionamentos: cliente-fachada e fachada-subsistema. A imagem a seguir demonstra a GUI criada para a introdução do padrão *Facade*.

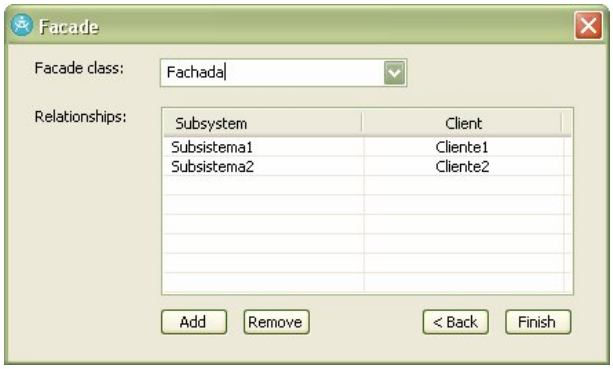

Figura 5-10 GUI *Facade*

Para a sua implementação, a classe ApplyFacadeAction cria a classe fachada indicada pelo usuário, substitui os relacionamentos entre os cliente e seus respectivos subsistemas por relações, já explicadas, com a fachada e copia as assinaturas dos métodos do subsistema para a fachada recém criada. Por motivos de simplificação, apenas a parte mais relevante do código será mostrada, correspondendo à substituição dos relacionamentos entre clientes e subsistemas.

O trecho de código verifica, por razões de consistência, se o subsistema e seu cliente realmente se relacionam, conforme indicado na interface gráfica do *pluglet*. Para isso, o código utiliza o método searchEdgeBetweenNodes. Em caso positivo, o código invoca os métodos createOneToOneAssociation, para criar a relação entre a fachada e o

subsistema, e createOneToManyAssociation, para a relação entre o cliente do subsistema e a fachada. Com o objetivo de evitar repetições e resolver conflitos, antes de criar tais relações, faz-se a verificação se as classes envolvidas na criação do padrão já não estão relacionadas de outra forma qualquer.

```
Edge edge = diagramUtil.searchEdgeBetweenNodes(clientNode, subsystemNode);
if (edge != null) {

    if (!diagramUtil.hasAssociationBetweenNodes(facadeNode, subsystemNode)) {
        String end1Name = subsystem.toLowerCase();
        Relationship r = Operations.createOneToOneAssociation(
                facadeClass, subsystemClass, edge.getElement(), end1Name);
        diagramHelper.createEdge(facadeNode, subsystemNode, r);
    }

    if (!diagramUtil.hasAssociationBetweenNodes(clientNode, facadeNode)) {
        String end1Name = facade.toLowerCase();
        Relationship r = Operations.createOneToManyAssociation(
                clientClass, facadeClass, edge.getElement(), end1Name);
        diagramHelper.createEdge(clientNode, facadeNode, r);
    }
} else {
    throw new Exception("There is no relationship between " + client
            + " and " + subsystem + " in this diagram.");
}
```

Figura 5-11 Implementação *Facade*

Finalmente, para introduzir o padrão *Persistent Data Collection*, o usuário deverá selecionar a classe básica no diagrama do RSA e, ao rodar o *pluglet*, informar as responsabilidades envolvidas no padrão. Caso o usuário deseje inserir o padrão em mais de uma classe básica, o *pluglet* disponibiliza a opção "*Select classes"*, onde o usuário poderá selecionar o conjunto de classes básicas que receberão o padrão PDC.

Entretanto, por motivos de usabilidade, ao utilizar esta opção o usuário não informa as classes e interfaces que serão utilizadas pelo PDC. As responsabilidades envolvidas são criadas com nomes padrão, baseadas no nome da classe básica escolhida utilizando-se seguinte critério:

Tabela 5-1 Responsabilidades padrão PDC

Responsabilidade	Nome
BusinessBasic	$Basic (definida pelo usuário)
PersistentDataCollection	$Basic + "Repository"
IBusinessData	"I" + $Basic + "Repository"
BusinessCollection	$Basic + "Business"
Facade	"Facade"

Caso o usuário não opte pela opção "*Select classes*", ele deverá delegar as responsabilidades do padrão PDC às diferentes classes e interfaces conforme mostra a Figura 5-12.

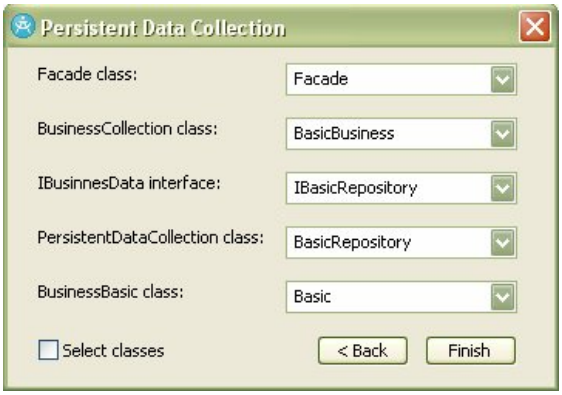

Figura 5-12 GUI *Persistent Data Collection*

Definidas as responsabilidades, a classe ApplyPDCAction é então chamada para adicionar ao modelo todas as classes e interfaces envolvidas no padrão, bem como as relações de associação, realização e dependência. A implementação dessas operações pode ser vista, em parte, na imagem abaixo. Na Figura 5-13 estão ilustradas apenas a criação e interação entre a fachada e a classe de negócios.

O código apresentado cria as classes e os nós responsáveis pela fachada e o cadastro através dos métodos createClassElement e createClassNode, respectivamente. Em seguida, cria-se a relação de associação entre ambos os elementos através do método createOneToOneAssociation e a ligação de ambos os nós por meio do método createEdge.

```
/* Create facade */
String facadeName = map.get(Constants.PDC_FACADE_LABEL);
Class facadeClass = modelUtil.createClassElement(facadeName);
Node facadeNode = diagramUtil.createClassNode(model, facadeName);
pcu = new ClassPluglet(facadeClass);
pcu.createOperation("systemService", parameters);


/* Create business */
String businessName = map.get(Constants.PDC_BUSINESS_LABEL);
Class businessClass = modelUtil.createClassElement(businessName);
Node businessNode = diagramUtil.createClassNode(model, businessName);
pcu = new ClassPluglet(businessClass);
pcu.createOperation("specificSystemService", parameters);

/* Create facadeNode -> businessNode association */
if (!diagramUtil.hasAssociationBetweenNodes(facadeNode, businessNode)) {
    String end1Name = businessName.toLowerCase();
    Relationship r = Operations.createOneToOneAssociation(
            facadeClass, businessClass, null, end1Name);
    diagramHelper.createEdge(facadeNode, businessNode, r);
}
```

Figura 5-13 Implementação Persistent Data Collection

5.3.3 Merge de Classes

O merge de classes é implementado pelo *pluglet* MergeClassPluglet. Assim como os demais, esse *pluglet* é constituído pela classe principal, por sua GUI correspondente e pela classe que realiza as operações da automação (*action*). Primeiramente, o usuário deverá selecionar no diagrama do RSA quais as classes sofrerão o merge e, em seguida, executar o *pluglet*. Dessa forma, uma nova janela será aberta, solicitando o nome da classe resultante do merge, como mostra a Figura 5-14.

Figura 5-14 Tela merge de classes.

A implementação utilizada pela classe MergeClassAction abordou o problema de uma forma bastante simples: cada classe envolvida no merge terá seus atributos, métodos e relacionamentos transferidos para a classe destino. Para a resolução de possíveis conflitos, a solução optou por sempre priorizar o dado mais antigo. Assim, a classe destino irá ignorar as repetições de métodos, atributos e relacionamentos com outros membros. A implementação da solução pode ser vista na Figura 5-15.

Primeiramente, invoca-se o método populateNodeExistingAssociations, criando-se uma lista dos relacionamentos da classe resultante do *merge*. Dessa forma, evita-se a repetição de associações entre duas classes. Na seqüência, itera-se sobre cada nó participante, verificando se o nó iterado não corresponde ao nó resultante. Em caso negativo, transfere-se para o nó resultante do *merge* todos os membros (atributos, métodos e ligações) do nó iterado através dos métodos inheritClassifierAttribute, inheritClassifierOperations e mergeNodeAssociation, respectivamente.

```
Class _class = modelUtil.createClassElement(targetClassName);
Node newNode = diagramUtil.createClassNode(modelUtil.getModel(), targetClassName);

List<EdgePluglet> associations = populateNodeExistingAssociations(newNode);
ClassPluglet classUtil = new ClassPluglet(_class, associations);

for (Node currentNode : nodes.values()) {
    Classifier e = (Classifier) currentNode.getElement();
    if (!e.getName().equals(_class.getName())) {
        classUtil.mergeNodeAssociations(diagramUtil, currentNode, nodes);
        classUtil.inheritClassifierAttributes(e);
        classUtil.inheritClassifierOperations(e);

        diagramHelper.destroyView(currentNode);
    }
}
```

Figura 5-15 Implementação merge de classes

5.3.4 Projeto de Subsistemas

O projeto de subsistemas, diferentemente dos demais *pluglets*, envolveu uma preocupação a mais: a criação de pacotes ao modelo. De acordo com a disciplina de Análise e Projeto, os elementos necessários à implementação de um subsistema devem ficar encapsulados em um pacote à parte.

Entretanto, mesmo com essa peculiaridade, o desenvolvimento do DesignSubsystemPluglet não fugiu aos padrões da implementação utilizados até o momento. Ou seja, o *pluglet* continuou sendo formado pela classe principal, sua GUI e seu *action*.

Na GUI, o usuário informa os nomes do subsistema, de sua interface e de sua fachada. Em seguida, o trabalho de criação dos elementos necessários para a automação é feito pela classe DesignSubsystemAction. A GUI do *pluglet* de projetar subsistemas é mostrada pela Figura 5-16.

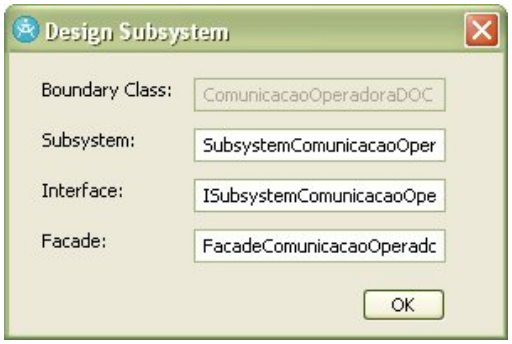

Figura 5-16 Tela Projetar Subsistemas

Para a implementação da automação, foram feitas as operações de criação dos elementos envolvidos e da relação de realização entre a fachada e a interface do

subsistema. Adicionalmente, os métodos da classe de análise inicial são transferidos para a interface e a fachada criadas no projeto de subsistemas. O desenvolvimento das operações descritas é ilustrado na imagem apresentada na Figura 5-17.

O código ilustra a criação de um pacote através do método createPackageElement, da interface através do método createOwnedInterface e da fachada através do método createOwnedClass. Em seguida, cria-se a realização da interface pela fachada invocando-se o método createInterfaceRealization. Por fim, as operações do subsistema são herdados utilizando-se os métodos inheritNodeAssociations, inheritClassifierAttributes e inheritClassifierOperations.

```
Package subsystem = modelUtil.createPackageElement(subsystemName);

Interface _interface = subsystem.createOwnedInterface(iSubsystemName);
diagramUtil.createInterfaceNode(model, iSubsystemName);

Class facade = subsystem.createOwnedClass(facadeName, false);
facade.createInterfaceRealization("facadeRealization", _interface);

Classifier classifier = (Classifier) node.getElement();

InterfacePluglet interfaceUtil = new InterfacePluglet(_interface);
interfaceUtil.inheritNodeAssociations(diagramUtil, node);
interfaceUtil.inheritClassifierAttributes(classifier);
interfaceUtil.inheritClassifierOperations(classifier);

ClassPluglet classUtil = new ClassPluglet(facade);
classUtil.inheritClassifierAttributes(classifier);
classUtil.inheritClassifierOperations(classifier);
```

Figura 5-17 Implementação Projeto de Subsistemas

5.3.5 Separação de Classes

O último *pluglet* desenvolvido pela solução foi o SplitClassPluglet, responsável por realizar o *split* de uma classe existente no modelo. Para isso, o usuário deverá escolher quantas e quais serão classes resultantes do *split*, bem como quais os métodos, atributos e relacionamentos que cada classe resultante irá herdar. Assim, é possível definir classes com diferentes membros para uma mesma operação de *split*. A tela do *pluglet* é ilustrada pela Figura 5-18.

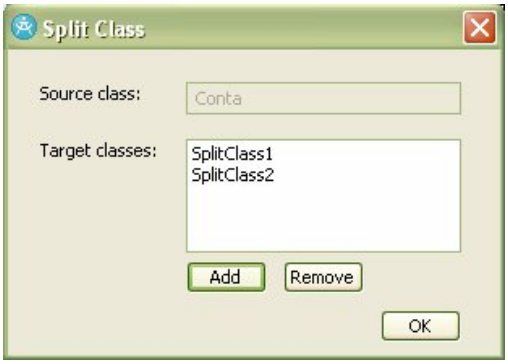

Figura 5-18 Tela separação de classes

Com uma abordagem semelhante ao merge, a separação de classes cria as classes definidas pelo usuário, adicionado seus respectivos membros a serem herdados. A resolução de possíveis conflitos envolvidos nessas operações é feita respeitando sempre o dado mais antigo, não inserir membros repetidos à classe destino. A implementação da separação de classes pode ser vista na Figura 5-19.

```
for (SplitElement element : elements) {
    String splitName = element.getClassName();
    Class _class = modelUtil.createClassElement(splitName);
    Node _node = diagramUtil.createClassNode(modelUtil.getModel(), splitName);

    ClassPluglet pcu = new ClassPluglet(_class);
    inheritMembers(element, _class, _node, pcu);
}
```

Figura 5-19 Implementação separação de classes

Assim, conclui-se o desenvolvimento das automações propostas por meio dos cinco *pluglets* descritos. Como dito anteriormente, além da criação dos *pluglets*, o trabalho teve a oportunidade de desenvolver um conjunto de classes úteis para futuros trabalhos envolvendo a plataforma Rational da IBM e os projetos GMF e UML2 do Eclipse.

6. Estudo de Caso

Depois de demonstrar o desenvolvimento da solução proposta, o trabalho em questão tem por objetivo ilustrar a sua aplicabilidade em um exemplo real de projeto arquitetural. Para isso, este capítulo descreve o passo a passo do projeto de um sistema bancário utilizado pelo curso de Análise e Projeto [21] oferecido pelo Centro de Informática da Universidade Federal de Pernambuco.

O QIB consiste em um sistema simples, cujas funcionalidades abrangem consulta de saldo, autenticação de usuário, realização de DOC, entre outras. Para um melhor entendimento do sistema, seu diagrama de casos de uso é ilustrado abaixo.

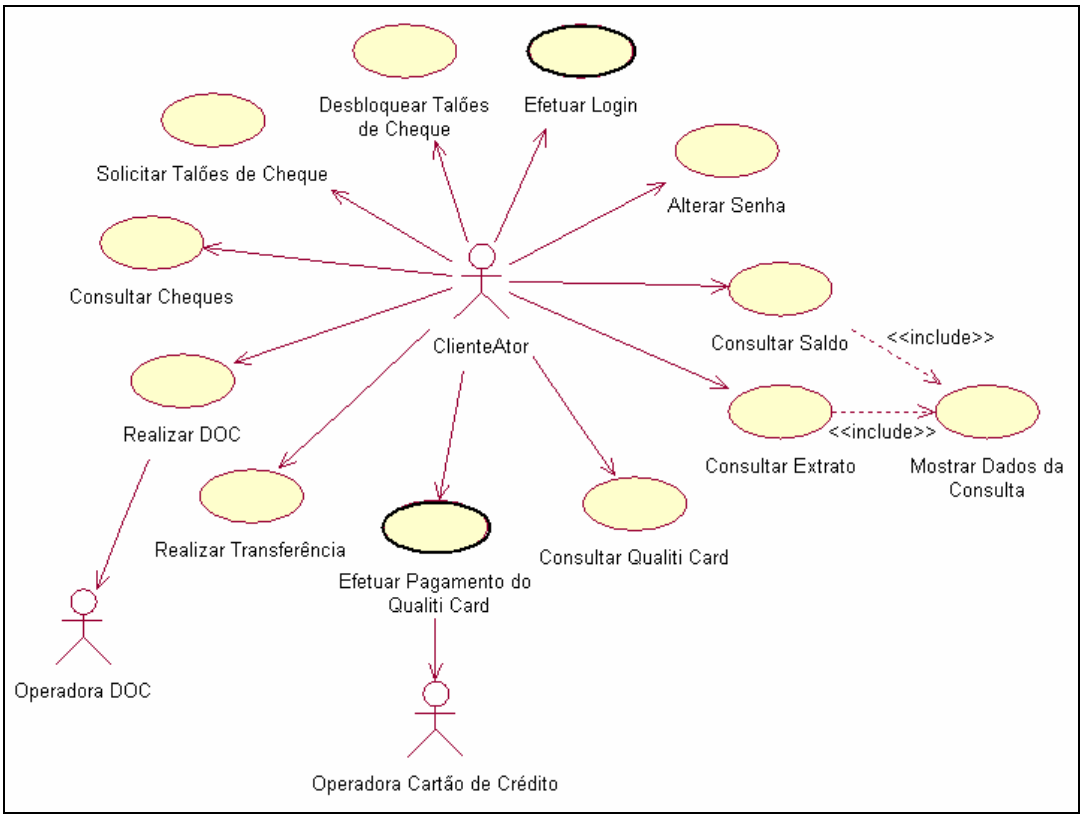

Figura 6-1 Diagrama de casos de uso do QIB

Para demonstrar a utilização da extensão criada pelo trabalho, serão utilizados os casos de uso Efetuar Login e Efetuar Pagamento do Qualit Card. Assim, através de técnicas conhecidas [22] para a identificação de classes de análise, é possível chegar ao diagrama que reflete a análise de ambos os casos de uso, representado na Figura 6-2.

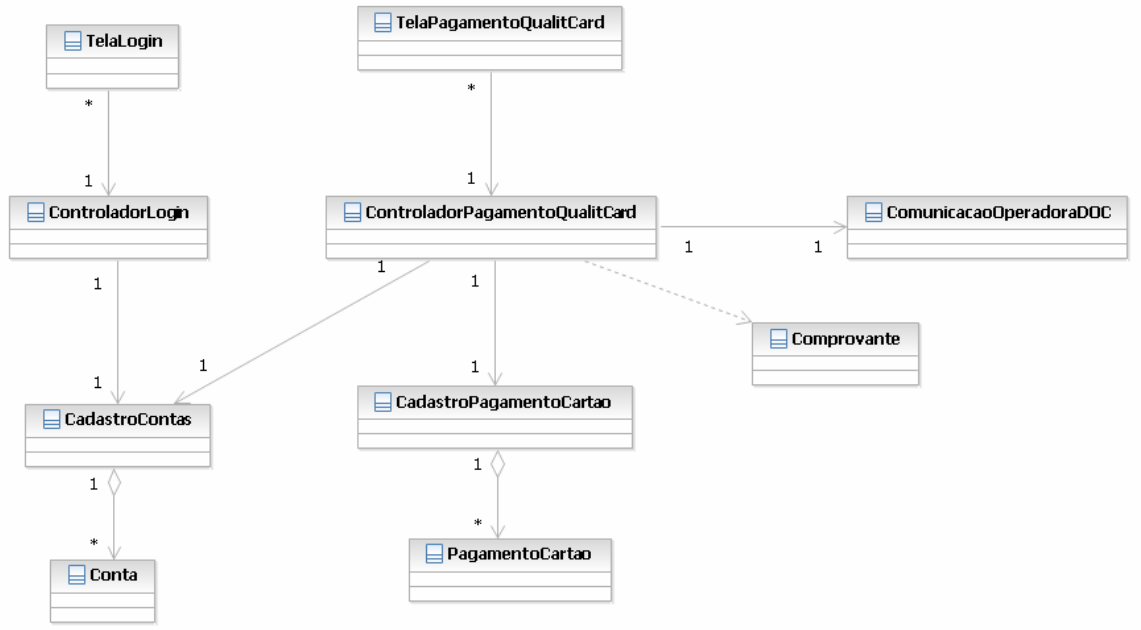

Figura 6-2 Modelo de análise do QIB

Nesse diagrama, é possível notar a existência das classes de fronteira TelaLogin, TelaPagamentoQualitCard e ComunicaçãoOperadoraDoc, das classes de controle ControladorLogin e ControladorPagamentoQualitCard, das classes de cadastro CadastroContas e CadastroPagamentoCartao e, finalmente, das classes básicas Conta, PagamentoCartao e Comprovante. Portanto, percebe-se que mesmo com um modelo reduzido, as responsabilidades inicialmente identificadas formam uma porção consistente e funcional do sistema QIB.

Após a criação do modelo de análise para ambos os casos de uso, segue-se a realização de projeto de sua arquitetura, quando são identificados os elementos de projeto, as oportunidades de reuso e padrões de projeto que serão incorporados ao sistema. A fim de documentar tais alterações é criada a tabela de mapeamento, responsável por correlacionar os elementos de análise em elementos de projeto. A tabela de mapeamento do QIB pode ser vista na Figura 6-3.

Classes de Análise	Elementos de Projeto
	Fachada TelaMenu Data Hora
Conta	ContaInternet ContaCorrente
CadastroContas	CadastroContasInternet IRepositorioContasInternet RepositorioContasInternetBDR CadastroContasCorrente IRepositorioContasCorrente RepositorioContasCorrenteBDR
CadastroPagamentosCartao	CadastroPagamentosCartao IRepositorioPagamentosCartao RepositorioPagamentosCartaoBDR
ComunicacaoOperadoraCartao	SubsistemaComunicacaoOperadoraCartao ISubsistemaComunicacaoOperadoraCartao FachadaComunicacaoOperadoraCartao
As demais classes são mapeadas diretamente em elementos de projeto	

Figura 6-3 Tabela de mapeamento QIB

Com a tabela acima é possível identificar todas as mudanças a serem realizadas ao modelo de análise do sistema. Entretanto, ao invés de realizá-las manualmente nas ferramentas de modelagem, as alterações serão realizadas utilizando as funcionalidades de automação criadas pelo trabalho em questão.

Inicialmente, a primeira alteração a ser feita no modelo é a introdução do padrão *Facade* a fim de centralizar as funcionalidades oferecidas pelo sistema em uma única classe. Dessa forma, é preciso modificar o modelo para que as classes de fronteira tenham acesso à camada de negócios mediante a interceptação de uma fachada. Para isso, utiliza-se o *pluglet* descrito na seção 5.3.2.

Com a utilização do `PatternPluglet`, é preciso selecionar o padrão *Facade* entre as opções apresentas e adicionar cada relação cliente-subsistema que será substituída pela introdução da fachada. Após a configuração das relações e a execução do *pluglet*, o diagrama do modelo resultante é atualizado conforme a Figura 6-4.

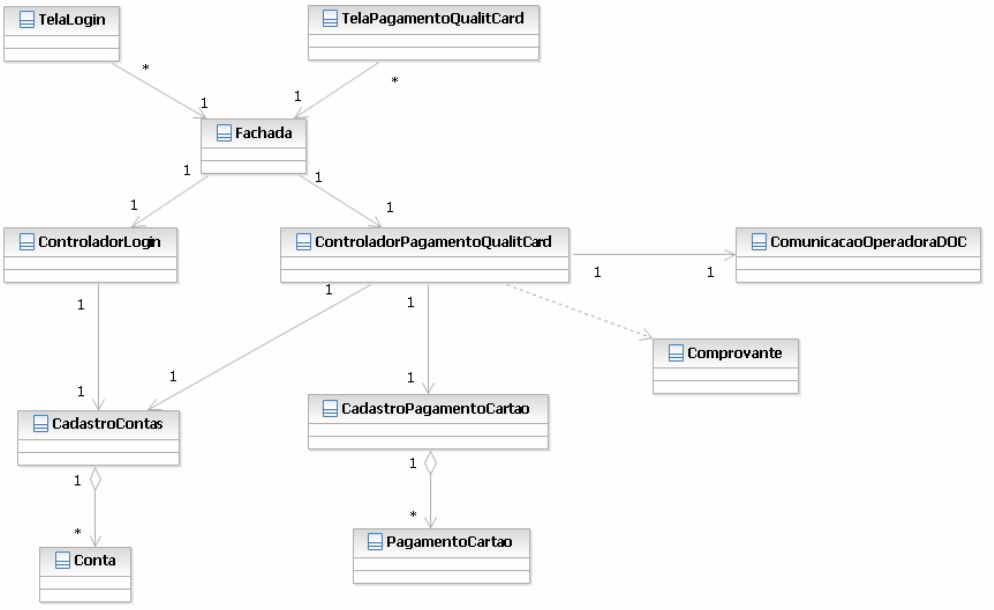

Figura 6-4 Aplicação da fachada

Em seguida, nota-se a necessidade de um ponto global de acesso à Fachada, bem como da existência de uma única instância dessa classe. Para tal, a classe Fachada deve assumir o papel de um *singleton*, sendo instanciada através de um método estático o qual retorna sempre a mesma instância da classe para todas as suas requisições.

Tal comportamento pode ser vinculado ao modelo facilmente através da utilização, novamente, do `PatternPluglet`, devendo-se selecionar o padrão *Singleton* e em seguida informar a classe que receberá esse papel. A Figura 6-5 mostra o diagrama UML após a introdução do padrão *Singleton* na Fachada.

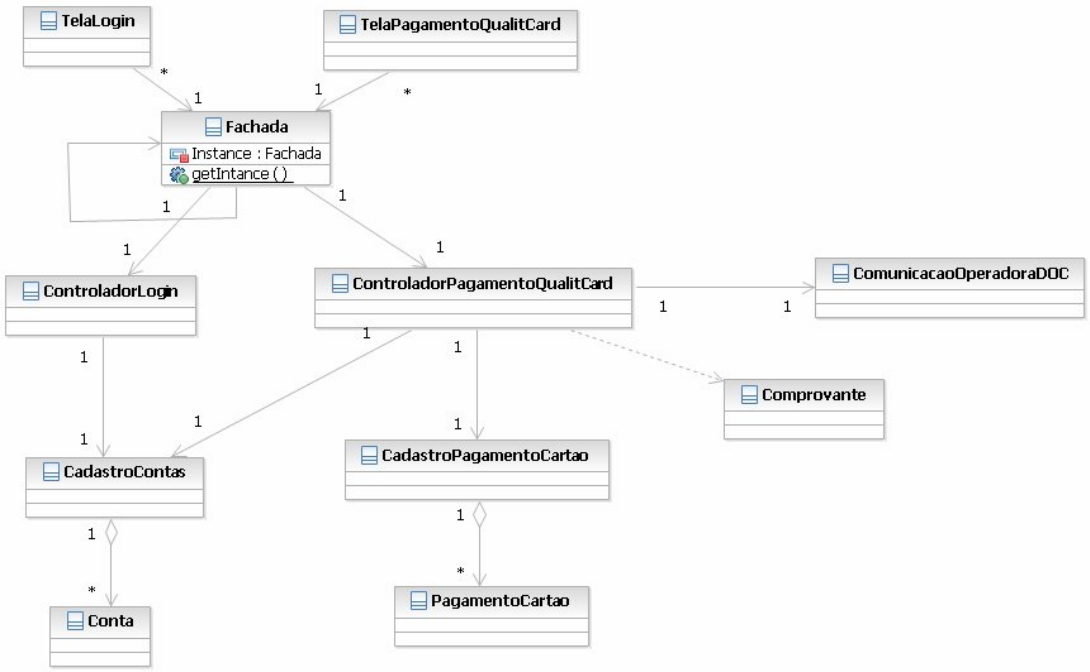

Figura 6-5 Introdução do padrão *Singleton*

Concluída a introdução da Fachada e do padrão *Singleton*, faz-se necessário projetar um subsistema de comunicação a partir da classe de fronteira de entre o QIB e a operadora DOC: ComunicacaoOperadoraDOC. No projeto de subsistemas, a classe de fronteira é substituída por uma abstração mais completa do subsistema projetado, constituída pelo subsistema propriamente dito, uma interface contendo o contrato utilizado para a comunicação externa, e a fachada que implementa a interface.

Para o projeto de subsistemas, o trabalho em questão desenvolveu o DesignSubsistemPluglet, responsável tanto por criar as novas responsabilidades no modelo, quanto alterar seu diagrama UML correspondente. Ao inicializar o *pluglet*, é preciso apenas informar o nome das responsabilidades do subsistema. Assim, incorpora-se ao modelo o pacote correspondente ao subsistema, sua fachada e sua interface, enquanto que no diagrama a classe de fronteira é substituída pela interface do subsistema criado. As alterações estão ilustradas nas Figuras 6-6 e 6-7.

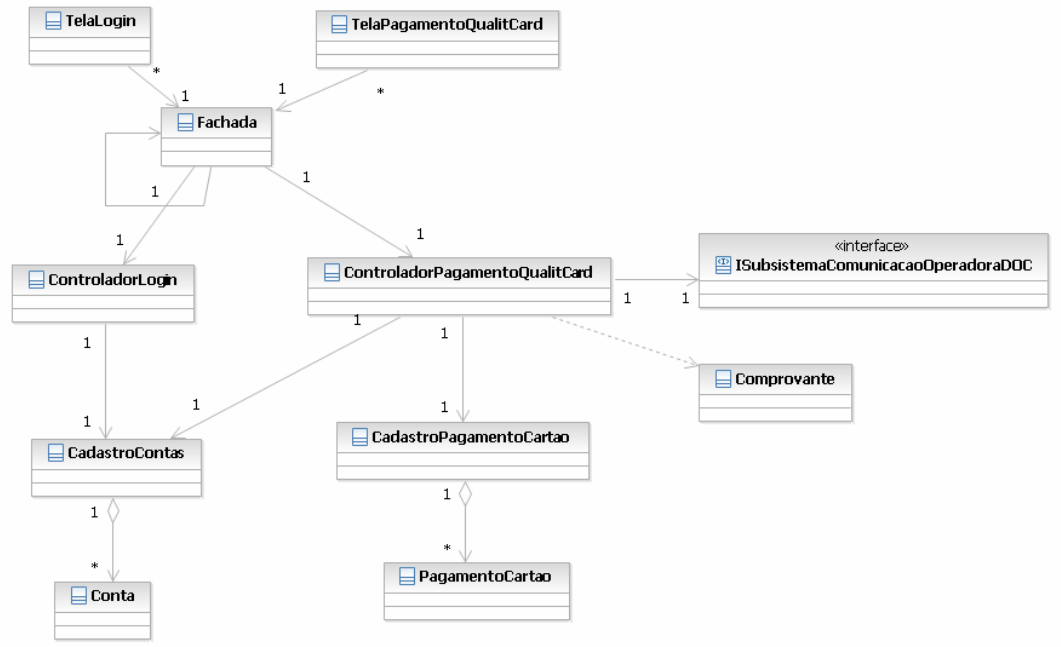

Figura 6-6 Projeto de subsistema

Em seguida, percebe-se a necessidade de decompor a classe Conta em duas novas classes: ContaCorrente e ContaInternet. A conta corrente corresponde a uma conta de banco padrão, contendo um número, uma agência e um saldo, enquanto que a conta internet é utilizada apenas para acesso ao *home banking*, contendo usuário, uma senha e uma conta corrente associada.

Para modelar esse cenário, é preciso separar as classes CadastroContas e Conta em quatro novas classes: CadastroContaCorrente, CadastroContaInternet,

ContaCorrente e ContaInternet. Ao invés de realizar a separação manualmente, preocupando-se com relacionamentos, métodos e atributos, é possível automatizá-la através do SplitClassPluglet. Basta selecionar a classes que sofrerá o *split*, as classes resultantes do *split* e os respectivos membros a serem herdados, deixando o resto do trabalho para a automação.

A única alteração manual a ser feita é a inclusão da relação de associação entre ContaInternet e ContaCorrente. Por essa relação não estar vinculada à operação de *split* da classe Conta, a mesma não é provida pela automação. O *split* do cadastro e da classe básica estão ilustrados, respectivamente, na Figura 6-8 e na Figura 6-9.

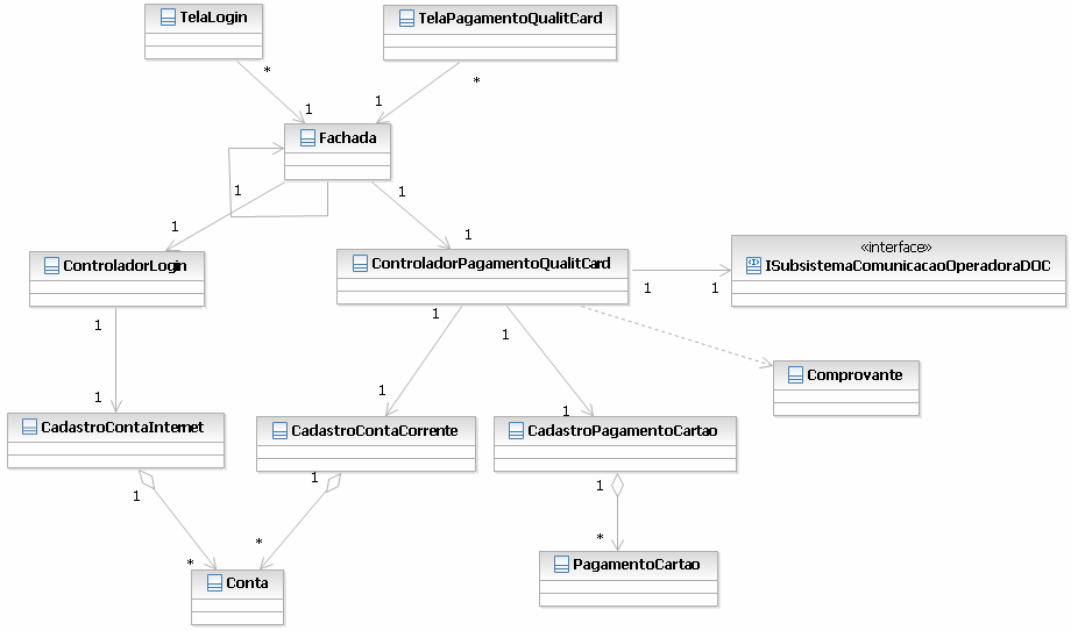

Figura 6-7 Split CadastroContas

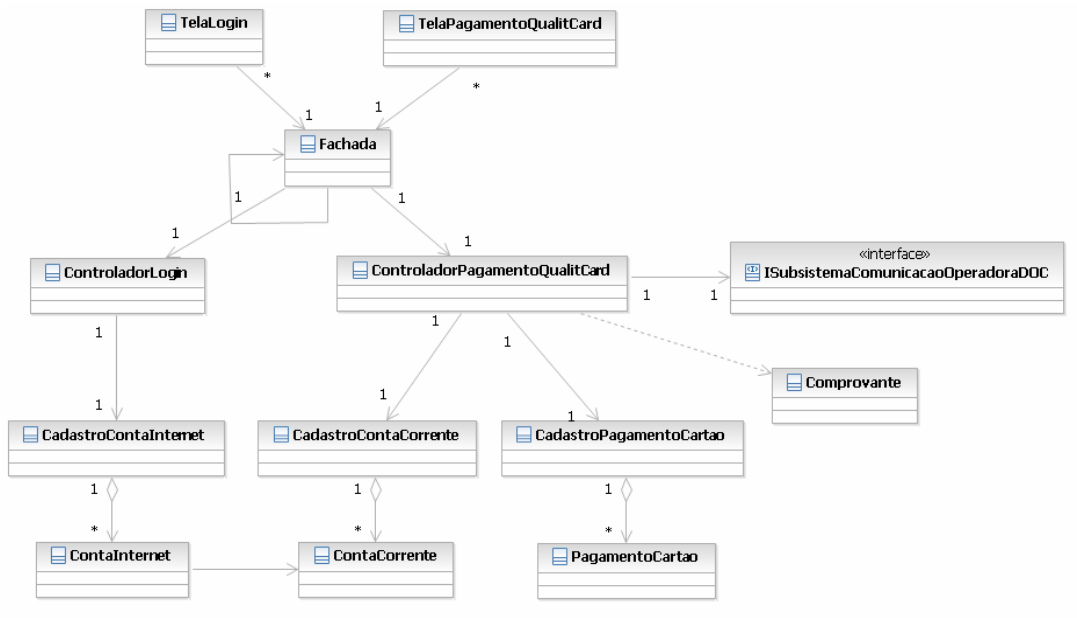

Figura 6-8 Split Conta

Terminado o *split*, a fase de projeto identificou a necessidade de se criar uma generalização para a classe `PagamentoCartao`. A generalização é responsável por abstrair qualquer tipo de operação bancária, sendo o pagamento do cartão apenas uma restrição às operações possíveis de serem realizadas.

A tarefa de criar generalizações é automatizada pelo `CreateGeneralizationPluglet`. Como a classe `PagamentoCartao` já se relaciona com a classe `CadastroPagamentoCartao`, é aconselhável criar uma nova classe de cadastro responsável pela agregação das transações, excluir do diagrama o cadastro existente e, só então, criar a classe Transação.

Entretanto, caso o usuário preferir, é possível antecipar a criação da generalização `Transação`, fazendo com que a mesma herde para si os relacionamentos destinados à sua subclasse, no caso `PagamentoCartao`, faltando apenas renomear a classe `CadastroPagamentoCartao` para `CadastroTransacoes`. Seja qual for a opção escolhida, as alterações resultarão em um diagrama idêntico ao da Figura 6-10.

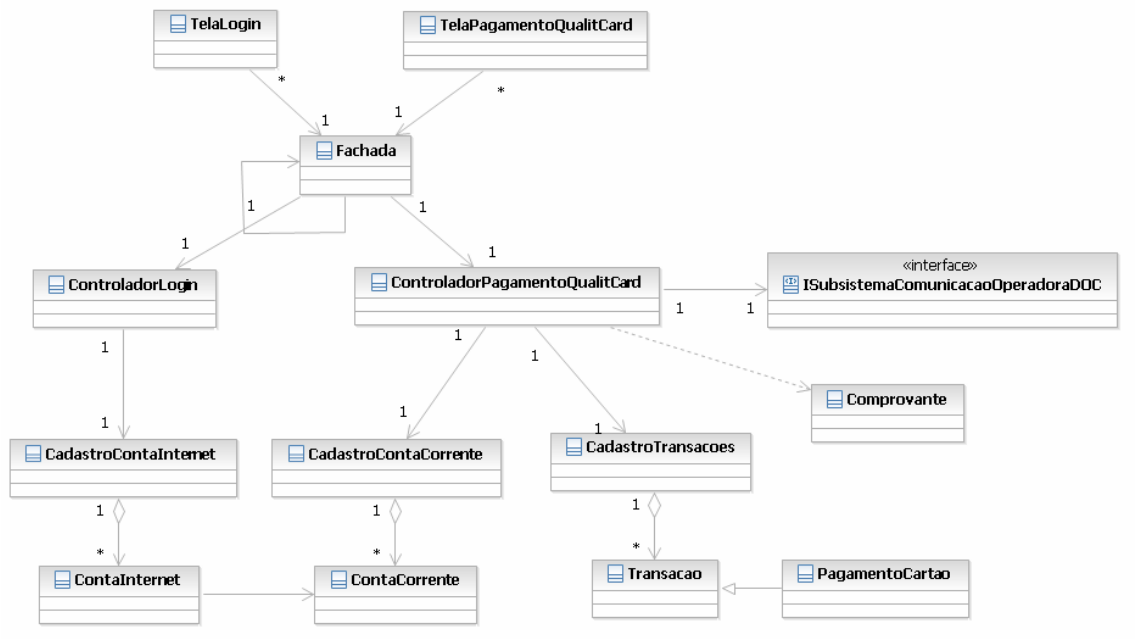

Figura 6-9 Generalização Transação

Com a realização do *split* no cadastro de contas e a generalização das operações bancárias, criou-se um modelo cujos elementos possuem responsabilidades coesas e bem definidas. Porém, mesmo centralizando o ponto de acesso às funcionalidades do sistema, a fachada se mostra como um fluxo de controle simples, apenas repassando as requisições recebidas aos controladores.

Com o objetivo de melhorar a lógica do sistema e diminuir o número de chamadas feitas até a efetivação de determinada operação, é sensata a realização do merge entre controladores e a fachada, centralizando, assim, o ponto de acesso ao sistema, bem como as regras de negócio associadas a cada funcionalidade. Para isso, o MergeClassPluglet se mostra bastante funcional na realização de merges, bastando ao usuário apenas selecionar no diagrama as classes que sofrerão o merge, iniciar o *pluglet* e definir a classe destino. Neste cenário, seriam selecionados ambos os controladores como classes a sofrerem o merge, e a Fachada como classe destino.

O resultado da operação de merge pode ser visto na Figura 6-11. Por motivos de simplificação os métodos e atributos dos controladores foram omitidos, porém todos são transferidos para a fachada que em caso de conflito, prioriza sempre o dado mais antigo.

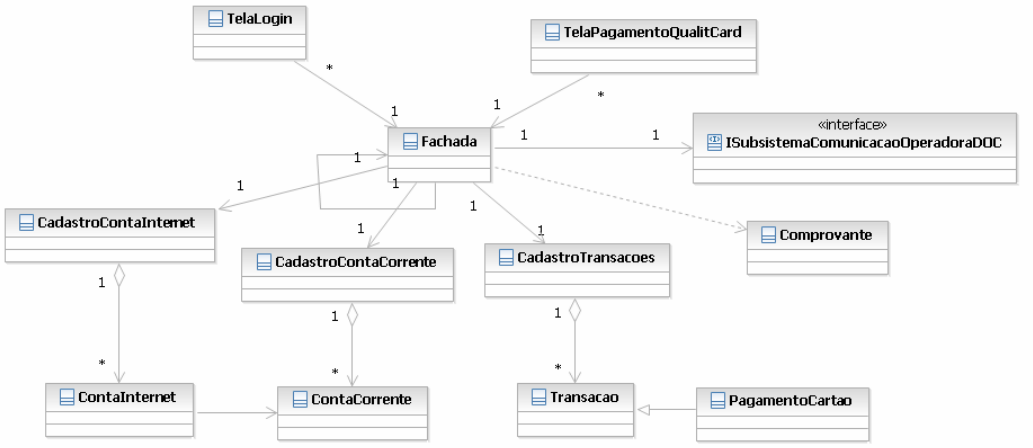

Figura 6-10 Merge dos controladores com a fachada

Para finalizar o projeto arquitetural do QIB resta ainda a introdução do padrão *Persistent Data Collection* a todas as coleções a serem persistidas. O padrão defende que as relações entre os cadastros e suas agregações devem ser substituídas por classes de persistência, responsáveis por armazenar os dados.

Para aplicar tal padrão de forma automatizada a todas as coleções existentes, o PatternPluglet oferece a opção de introduzir o PDC a diversas classes simultaneamente. Portanto, o usuário deverá apenas iniciar o *pluglet*, selecionar o PDC entre as opções de padrões existentes, escolher a opção "*Select classes*" e informar as coleções que receberão o padrão. O resultado pode ser visto na Figura 6-12.

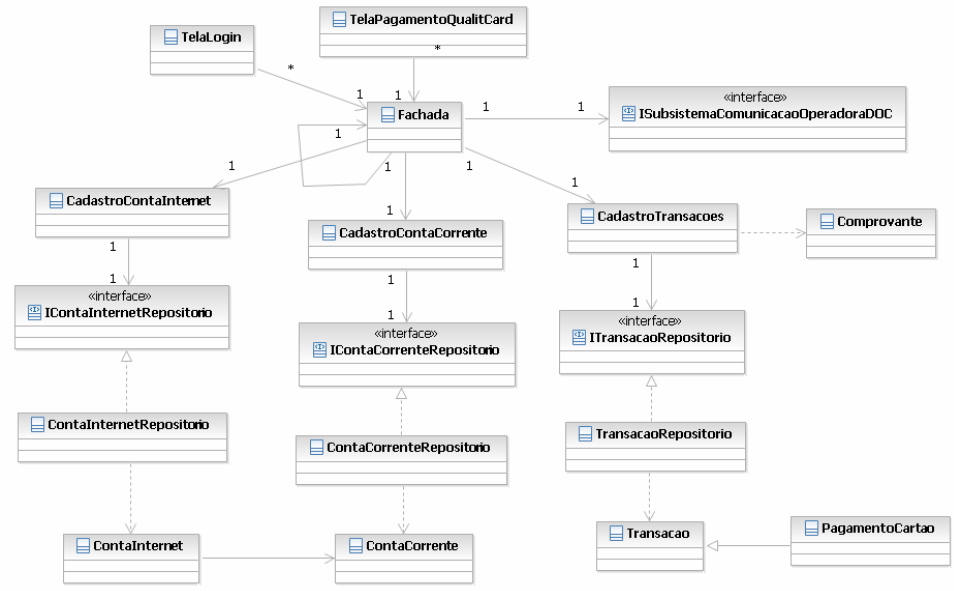

Figura 6-11 Introdução do padrão PDC

Desta forma, conclui-se o projeto arquitetural dos casos de uso Efetuar Login e Efetuar Pagamento do Qualit Card com pouquíssimas intervenções manuais ao diagrama. Foi possível perceber que a grande maioria das alterações feitas no modelo se

deu por meio dos *pluglets* desenvolvidos durante o trabalho apresentado, aumentando de forma significativa a agilidade na criação do projeto arquitetural, bem como diminuindo o tempo para a sua realização, erros potenciais resultantes de atualizações manuais e a sistematização do processo envolvido nessa atividade.

7. Conclusões

Através da revisão da disciplina de Análise e Projeto do RUP foi possível identificar que a realização da atividade Projetar Arquitetura continua sendo uma tarefa lenta e repetitiva. Apesar do esforço feito pelas ferramentas de modelagem disponíveis no mercado, que gerenciam e aceleram o processo, muitas das atividades desenvolvidas são sistemáticas, transparecendo diversas oportunidades de melhoria.

Além da criação de novas funcionalidades, uma melhoria bastante significativa pode ser alcançada através da automação das funcionalidades já existentes. Assim, busca-se diminuir o número de interações entre usuário e ferramenta, atingindo-se o mesmo resultado de forma automatizada e, portanto, mais rápida e produtiva.

Para isso, é preciso que a ferramenta de modelagem utilizada permita a extensão de suas funcionalidades, de forma a incorporar novos *plugins* à sua plataforma. Após um comparativo realizado entre algumas das principais ferramentas atuais, o Rational Software Architect se mostrou bastante convidativo para a criação de extensões, sendo então a ferramenta alvo do trabalho.

A partir de um estudo detalhado da ferramenta, foi possível analisar os passos a serem automatizados, buscando-se aprimorar as tarefas mais comumente realizadas em um projeto arquitetural. Assim, o trabalho focou em automatizar a decomposição e composição de classes, a criação de generalizações, o projeto de subsistemas e a introdução dos padrões *Facade*, *Singleton* e *Persistent Data Collection*.

Para criar as extensões citadas, o trabalho utilizou o conceito de *pluglets*, que constituem *plugins* específicos para a nova plataforma Rational, e o *Graphical Modeling Framework* (GMF), conjunto de APIs que possibilitam a criação de editores gráficos para o Eclipse. Dessa forma, a criação de *pluglets* permitiu estender o RSA, enquanto que a o GMF proporcionou a interação com os modelos e diagramas UML criados na ferramenta.

Finalmente, com a implementação das automações concluída, apresentou-se um estudo de caso onde foi possível demonstrar a sua utilização em um caso real de projeto arquitetural. Nesse exemplo, foi possível perceber a redução significativa na quantidade de interações com a ferramenta de modelagem e, conseqüentemente, um ganho real de produtividade. Com isso, evita-se que a atividade Projetar Arquitetura, bem como a manutenção do modelo durante o ciclo de vida de um projeto, caia no desuso, criando um gargalo no processo de desenvolvimento utilizado.

7.1 Trabalhos Relacionados

A sistemática envolvida na realização da atividade Projetar Arquitetura é de conhecimento geral, ocasionando o desenvolvimento de várias pesquisas que buscam agilizar o processo envolvido nessa atividade. Algumas delas podem ser encontradas em [25] e [26].

Na primeira, é feito um estudo sobre a disciplina de Análise e Projeto seguida da criação de automações para algumas de suas atividades, entre elas a inferência de relacionamentos e operações, refinamento do diagrama de pacotes e o mapeamento de classes. De forma similar ao trabalho apresentado, a pesquisa analisada se propõe a automatizar a atividade Projetar Arquitetura.

Entretanto, o trabalho apresentado em [25] se restringe a criar o artefato de mapeamento das classes de análise em elementos de projeto na ferramenta CASE e utilizá-lo para automatizar parcialmente o processo de atualização dos diagramas das realizações de casos de uso. O trabalho se compromete também a inferir as dependências entre os pacotes presentes no diagrama, automatizando a sua criação.

Na segunda pesquisa analisada, o IIMPaR, o foco é exclusivamente na integração de modelos de padrões de software para o Rational Rose. O trabalho desenvolveu, então, um *plug-in* para o Rose visando a inclusão de um repositório de padrões à ferramenta, permtindo aplicá-los automaticamente ao modelo de um sistema.

É possível notar a semelhança com o trabalho apresentado, que também provê a introdução de padrões de projetos ao modelo de forma automatizada. Contudo, o IIMPaR não adiciona nenhuma outra automação à disciplina de Análise e Projeto, não constituindo, dessa forma, uma ferramenta de automatização do projeto arquitetural tão abrangente como o trabalho aqui apresentado.

Outros trabalhos relacionados podem ser encontrados em algumas das próprias ferramentas de modelagem estudas no Capítulo 3, entre elas a StarUML, o Borland Together e o RSA. Todas as ferramentas citadas já incluem em seu conjunto de funcionalidades a introdução de padrões de projeto ao modelo, constituindo um avanço em relação a ferramentas mais antigas como o Rational Rose. Entretanto, como citado no trabalho apresentado, a abordagem escolhida por essas ferramentas para a introdução de padrões não foca em automatizar a atividade Projetar Arquitetura, fazendo com que o projeto de um sistema de software necessite de constantes intervenções manuais.

7.2 Trabalhos Futuros

Apesar de o trabalho apresentado representar uma melhoria na realização da atividade Projetar Arquitetura do RUP, o mesmo focou em apenas algumas de suas tarefas. Dessa forma, oportunidades de automação ainda podem ser encontradas na criação de um projeto arquitetural, bem como nas demais atividades da disciplina de Análise e Projeto.

Uma delas seria automatizar a introdução de padrões de projetos não contemplados pela solução, tomando como base o conjunto de padrões GoF [6]. Assim, padrões como o *Abstract Factory, Adapter* e *Observer* poderiam ser incorporados ao modelo de um sistema de forma mais rápida e satisfatória, como já acontece com os padrões *Facade, Singleton* e PDC.

Além dos padrões, poderiam ser contemplados estudos de caso mais complexos, envolvendo outras tarefas da atividade Projetar Arquitetura. Partindo-se dessa idéia, a solução poderia, por exemplo, permitir a criação automatizada do diagrama de pacotes a partir do modelo existente.

Como o trabalho não realizou nenhuma espécie de análise para verificar, de fato, o ganho de produtividade ao utilizar as automações criadas, seria interessante o desenvolvimento de experimentos objetivando medir a usabilidade e produtividade de um desenvolvedor usando a ferramenta. Através desse experimento, seria possível identificar melhorias nas automações, aprimorando ainda mais a criação de um projeto arquitetural.

Finalmente, o trabalho concentrou esforços em automatizar apenas a atividade Projetar Arquitetura, porém a disciplina de Análise e Projeto envolve várias outras atividades passíveis de automação. Portanto, pesquisas futuras poderiam se concentrar em atividades como a geração e manutenção da tabela de mapeamento a partir do projeto arquitetural, já que é preciso manter a tabela sempre consistente com o modelo, geração do projeto de casos de uso a partir do mapeamento, refletindo as alterações do projeto arquitetural na realização de casos de uso e, por fim, a geração de unidades de concorrência, incluindo a introdução de cápsulas e protocolos.

Referências

[1]. Bezerra , E., "*Princípios de Análise e Projeto de Sistemas com UML"*, Editora Campus, 2000.

[2]. Grady Booch, "*Object Solutions – Managing the Object Oriented Project"*, Addison-Wesley, 1995.

[3]. P. Kruchten, "*The Rational Unified Process: An Introduction*", Addison-Wesley, 2000.

[4]. Booch, G., Rumbaugh, J. e Jacobson, I., "*The Unified Modeling language User Guide"*, Addison-Wesley, 1999.

[5] Analysis & Design: Workflow. Disponível em: <http://rup.hops-fp6.org/process/workflow/ana_desi/wfd_and.htm>. Acessado em 06/10/2008.

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

[7] Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: The persistent data collections pattern. In *First Latin American Conference on Pattern Languages of Programming*, Rio de Janeiro, Brazil, 3th-5th October 2001.

[8] Rational Rose Enterprise. Disponível em: <http://www-01.ibm.com/software/awdtools/developer/rose/enterprise/>. Acessado em 21/10/2008.

[9] Rational Software Corporation: Using the Rose Extensibility Interface. Rational Software Corporation (2001).

[10] JUDE/Community. Disponível em <http://jude.change-vision.com/jude-web/product/community.html>. Acessado em 22/10/2008.

[11] StarUML - The Open Source UML/MDA Platform. Disponível em: <http://staruml.sourceforge.net/en/>. Acessado em: 25/10/2008.

[12] Borland Together - Modelagem Visual para o Design de Arquitetura do Software. Disponível em: <http://www.borland.com/br/products/together/>. Acessado em 27/10/2008.

[13] Diomidis Spinellis, "*Notable design patterns for domain specific languages*". Journal of Systems and Software, 56(1):91–99, February 2001. Disponível em <http://www.spinellis.gr/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html>. Acessado em 29/10/2008.

[14] Eclipse Modeling Tools. Disponível em <http://www.eclipse.org/downloads/ >. Acessado em 29/10/2008.

[15] Rational Software. Disponível em <http://www-01.ibm.com/software/br/rational/>. Acessado em 29/10/2008.

[16] Extending the modeling environment using pluglets. Disponível em <http://publib.boulder.ibm.com/infocenter/rsmhelp/v7r0m0/index.jsp?topic=/com.ibm.xtools.ras.pluglets.ui.doc/topics/t_extendraswithpluglets.html >. Acessado em 03/11/2008.

[17] Using Pluglets in IBM Rational Software Architect. Disponível em <http://www.ibm.com/developerworks/rational/library/06/1114_kelsey/index.html>. Acessado em 03/11/2008.

[18] The Eclipse Graphical Modeling Framework. Disponível em <http://www.eclipse.org/modeling/gmf/>. Acessado em 04/11/2008.

[19] UML2 Developer Guide. Disponível em <http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.uml2.doc/references/javadoc/org/eclipse/uml2/uml/package-tree.html>. Acessado em 05/11/2008.

[20] SWT: The Standard Widget Toolkit. Disponível em <http://www.eclipse.org/swt/>. Acessado em 06/11/2008.

[21] Disciplina de Análise e Projeto. Disponível em <http://www.cin.ufpe.br/~if708>. Acessado em 10/11/2008.

[22] Projetar Arquitetura: classes de projetos, subsistemas e recuso. Disponível em: <http://www.cin.ufpe.br/~if718> Acesso em: 08/11/2008

[23] Eclipse Modeling Framework Project (EMF). Disponível em <http://www.eclipse.org/modeling/emf/>. Acessado em 04/11/2008

[24] Graphical Editing Framework. Disponível em <http://www.eclipse.org/gef/>. Acessado em 04/11/2008.

[25] Márcio Bezerra, "*Automação de Atividades de Análise e Projeto no RUP*". Monografia – Centro de Informática, Universidade Federal de Pernambuco. 2007

[26] ANDRADE, R. M. C., SANTOS, M. S., NOGUEIRA, R., ROCHA, L. S., MENDONÇA, N. C. IIMPaR: Uma Interface de Integração de Modelos de Padrões de Software para o Rose. In: XII Sessão de Ferramentas, evento integrante do XIX Simpósio Brasileiro de Engenharia de Software (SBES'05), 2005, Uberlândia - MG, Brazil. Anais da XII Sessão de Ferramentas, SBES'05, 2005.

Prof. Ph.D. Augusto Cezar Alves Sampaio

Orientador

Fernando Valente Kakimoto

Aluno