Usando Padrões para Reestruturação de uma Aplicação Legada

Thiago L. V. L. Santos, Rodrigo T. Ramos e Börje F. F. Karlsson[1]

[1] Centro de Informática
Universidade Federal de Pernambuco (UFPE)
Recife - PE, Brasil

[tlvls,rtr,bffk]@cin.ufpe.br

Abstract. *This paper describes the experience of restructuring a real tool for cell phone testing. New system requirements, have lead us to the problem of restructuring the application in order to fulfill the required changes. This effort involved the use of coding, design and architectural patterns. The focus of this paper is in the application of design patterns to remodel the legacy application permitting easy software maintainance and a high degree of reusability, minimizing future tool developers efforts.*

Resumo. *Este artigo descreve a experiência adquirida na reestruturação de uma ferramenta real para realização de testes em celulares. Devido a mudanças de requisitos do software legado, detectamos que uma reestruturação, envolvendo padrões de codificação, padrões de projeto e padrões arquiteturais, seria necessária para permitir a execução das mudanças solicitadas. Neste artigo enfocamos a aplicação dos padrões de projeto para reconstrução da aplicação legada de forma que ela se torne de mais fácil manutenção e reusabilidade, permitindo assim minimizar os esforços necessários aos futuros desenvolvedores da ferramenta.*

1. Introdução

A adição de funcionalidades ou características a sistemas legados é de grande relevância em muitas organizações e constitui um fator importante no desenvolvimento de software. A modernização de softwares legados pode ter diversos objetivos como: controlar gastos com manutenção, melhorar performance, ou se adequar a mudanças de plataforma ou requisitos.

Segundo a definição da ANSI (ANSI/IEEE Std 729-1983), manutenção de software significa ”modificação de um produto de software depois de sua conclusão para corrigir falhas, melhorar performance ou outros atributos, ou adaptar o produto para mudanças de ambiente”.

Uma razão adicional apresentada em [Pressman, 2002] para a manutenção preventiva de software é a melhoria de legibilidade, capacidade de manutenção e de extensões futuras. O que vai na mesma direção de [Basili, 1990], que diz que manutenção pode ser considerada como desenvolvimento orientado a reuso.

Tais mudanças podem ser muito complexas devido a algumas características de sistemas legados, como falta de documentação, não aplicação ou má aplicação de processo, problemas de projeto, ou problemas com tecnologias anteriores. Devido a estes

Financiado pela Motorola Industrial Ltda., convênio de cooperação técnico-científica MOT C006/02.

problemas, sistemas legados frequentemente necessitam de reestruturação ou até mesmo reengenharia [Pressman, 2002].

Geralmente, é preferível optar por pequenas reestruturações do código, já que grandes mudanças podem trazer resultados negativos não esperados, sem garantias de que o novo sistema vá funcionar com a mesma corretude do sistema legado.

Podemos ver um sistema como uma coleção de padrões que interagem entre si [McNatt and Bieman, 2001]. Padrões podem ser vistos em diferentes níveis de abstração: arquiteturais [Buschman et al., 1996], de projeto [Gamma et al., 1995],ou de codificação [Sun, 1999], onde remodelagens podem ser aplicadas.

Na reestruturação de código legado pode ser util introduzir padrões de projeto para adicionar clareza e facilitar evoluções futuras, além de servir como guia e ajudar a evitar tomadas de decisões erradas.

Embora o processo de projeto de uma aplicação seja iterativo, e a sua reorganização seja uma situação frequente, métodos e ferramentas se concentram na criação de novos sistemas.

Uma das poucas abordagens propostas para reestruturação de sistemas legados é o COREM [Gall et al., 1995] que consiste de quatro fases: (i) Recuperação do *design*, onde modelos de projeto e relacionamento entre os objetos são recuperados, geralmente através de engenharia reversa do sistema; (ii) Modelagem da aplicação [Rumbaugh et al., 1991], baseada na análise de requisitos do sistema; (iii) Mapeamento de objetos [Gall and Weidl, 1998], que faz a associação entre os modelos; e (iv) Adaptação do código fonte, onde são realizadas adaptações no código para a reutilização do código legado. Nossa abordagem segue passos bastante similares.

Na seção 2 apresentaremos a ferramenta de testes legada e a motivação para o trabalho, na seção 3 comentaremos nossas ações para entedermos o *design* da aplicação e os pontos de reestruturação encontrados. Na seção 4 comentamos a reestruturação propriamente dita, a escolha da arquitetura, aplicação de padrões de projeto e mostramos o novo modelo da aplicação. Por fim comentaremos algumas conclusões referentes ao esforço executado.

2. Aplicação legada

Este trabalho é baseado na reestruturação de uma ferramenta utilizada pelo Centro de Informática, em conjunto com Motorola, para realização de testes das *features* relacionadas a *messaging*. A objetivo principal da ferramenta é o envio de mensagens EMS (Extended Message Service [3GPP, 2002]) através de um celular conectado a uma porta serial do computador. Como entrada da aplicação são utilizados arquivos XML [W3C, 1998] representando mensagens EMS, que podem ser criados, editados e visualizados na própria ferramenta. O visualizador é utilizado para analisar e depurar graficamente o XML, podendo exibir os vários tipos de elementos da mensagem, como texto, sons, figuras e animações, além dos valores do cabeçalho da mensagem.

Esta aplicação foi desenvolvida utilizando a linguagem Java [Gosling et al., 2000], sendo composta por aproximadamente 110 classes. Apesar da linguagem de desenvolvimento ser Orientada a Objetos, o que propiciaria uma modelagem OO, alguns trechos de código seguem um método de programação semelhante ao de linguagens estruturadas [Watt, 1994]. Requisições de extensão e manutenção da ferramenta revelaram graves problemas de modelagem e codificação, que foram o ponto de partida para este trabalho. Exemplos são: o grande acoplamentos entre todos as entidades que compõem o pro-

grama, principalmente as que tratavam da parte gráfica; dependências do protocolo de comunicação com o celular; dificuldade de extensão para suportar novos protocolos de envio de mensagens; controle falho do acesso às portas seriais; falta de estruturação dos pacotes e nomenclaturas deficientes.

Devido aos problemas estruturais encontrados, o objetivo desta reestruturação é a construção de um software, que satisfaça todos os requisitos impostos ao sistema legado, composto de componentes cujas responsabilidades sejam claramente identificáveis, para facilitar a extensão, reuso e manutenção do software.

3. Ações preliminares

Nesta seção descrevemos as principais medidas adotadas para entender os detalhes da aplicação legada e identificar como padrões [Sun, 1999, Gamma et al., 1995, Buschman et al., 1996] poderiam nos ajudar na sua reestruturação.

3.1. Recuperação do *design* da aplicação

Para tornar possível a reestruturação da aplicação legada precisamos entender a aplicação [Gall et al., 1995, Zimmer, 1994] descrita superficialmente na seção 2. Nesta etapa, buscamos recuperar os detalhes sobre o seu funcionamento, principalmente através documentação do sistema, se ela existir. Nesta busca, encontramos uma documentação incompleta e insuficiente, sendo necessária a utilização de outros métodos investigativos como: (i) um novo levantamento de requisitos baseado tanto na utilização da ferramenta, quanto em conversas com os *stakeholders*; (ii) realização de engenharia reversa para descoberta dos principais relacionamentos entre as entidades do sistema; e (iii) leitura de código para descoberta das práticas de programação dos desenvolvedores envolvidos no projeto. Durante o levantamento de requisitos pudemos entender melhor o que a aplicação se propõem a fazer (seção 2), durante o processo de engenharia reversa identificamos alguns elementos-chaves da aplicação (Figura 1, em notação UML [Booch et al., 1998]), e durante a análise de código verificamos algumas práticas de programação suscetíveis a falhas, como o não tratamento de exceções.

Neste diagrama podemos observar: entidades básicas ($ShortMessage$, $InformationElement$, $TextElement$), classes que realizam acesso a dispositivos seriais ($SimpleRead$, $SerialConfFrame$, $SerialSendFrame$), classes que tratam de regras de negócio do sistema ($ParentFrame$), e classes de interface gráfica ($ParentFrame$, $XmlEditor$, $XmlViewer$) fortemente acopladas ao restante do sistema. Este diagrama ajuda a evidenciar algumas práticas problemáticas:

1. Não houve um preocupação em definir o procedimento correto de particionamento das classes em pacotes;
2. As classes e pacotes são altamente inter-conectados (acoplados), não havendo separação das responsabilidades de cada um;
3. Falta de definições das interfaces internas e externas do sistema;
4. O sistema não possui um padrão de arquitetura bem definido, o que facilitaria a descoberta dos componentes-chaves;

O resultado desta análise foi o ponto de partida para que pudéssemos definir a reestruturação da aplicação.

3.2. Pontos de reestruturação

Com base nas informações sobre o *design* da aplicação, procuramos encontrar os principais problemas que poderiam ser resolvidos através da aplicação de padrões de codificação, de projeto e de arquitetura. Encontramos:

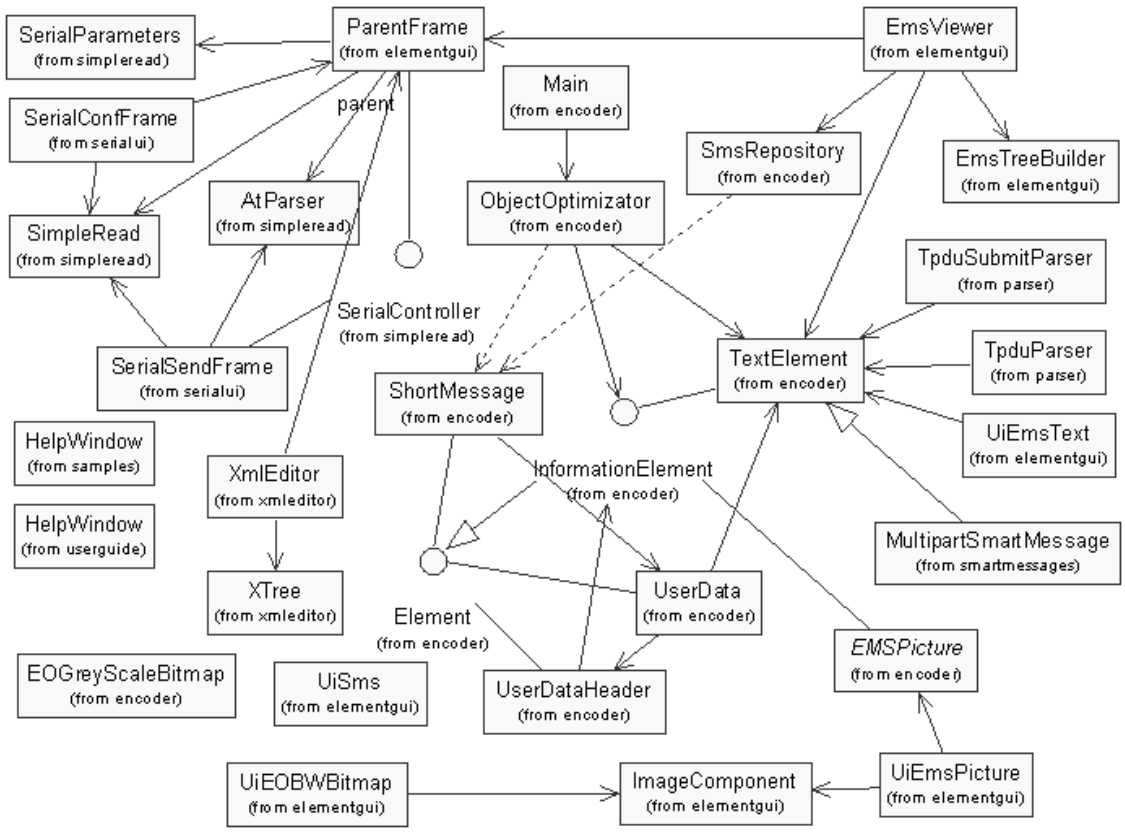

Figura 1: Diagrama de classes simplificado do sistema legado

Problemas de código

- Falta de uniformidade no padrão de codificação Java utilizado. Classes definidas por diferentes desenvolvedores com diferentes padrões de nomenclatura de pacotes, classes, métodos e atributos, e práticas de codificação, dificultando o processo de manutenção;
- O não tratamento explicito de exceções. Por exemplo: O uso indiscriminado de blocos `try {<código da aplicação>} catch(Exception e){}`, impossibilitando a identificação de erros ocorridos durante a utilização da ferramenta;
- A ausência de definição das responsabilidades de cada pacote e classe. Não identificação das interfaces que cada classe deveria fornecer, mais especificamente, a preocupação com a definição dos serviços que cada objeto deveria implementar, forte indício de uma implementação *ad hoc*;
- Redundância de código. Classes em pacotes diferentes que apresentam as mesmas funcionalidade, por exemplo, a classe $HelpWindow$ definida nos pacotes *samples* e *userguide*, cujo objetivo é a visualização da ajuda da aplicação.

Soluções propostas

- Utilização de uma instância de padrão de codificação para uniformizar a linguagem e práticas dos diferentes desenvolvedores, por exemplo o Java Coding Standards da Sun [Sun, 1999];
- Aplicação adequada de conceitos de orientação a objetos como: *information hidding*, herança e interfaces. Neste caso, seriam necessárias definições de *guidelines* para orientar os desenvolvedores sobre as boas práticas de modelagem e implementação de sistemas, entre boas práticas estão: a utilização de padrões de

projeto conhecidos da comunidade de engenharia software, e engenharia de software baseada em componentes [Pressman, 2002].

Problemas relacionados a padrões de projeto

- Na aplicação legada houve pouca preocupação em utilizar padrões de projeto, o que facilitaria o trabalho de manutenção e reutilização do código. A classe $EmsTreeBuilder$ mostrada na Figura 1 é um dos poucos exemplos da utilização de padrões. Em uma análise inicial fomos capazes de identificar, pelo menos, a possibilidade de utilização de: *Singletons* para controlar o acesso as portas seriais, *Abstract Factories* no processo de leitura e criação das representações gráfica, e em memória, das mensagens. A aplicação de padrões será descrita na seção 4.1.

Solução proposta

- Leitura das principais referência sobre padrões de projeto, em especial o *Design Patterns* da gangue dos quatro, para verificação da aplicabilidade dos padrões descritos.
- Aprendizado dos conceitos-chaves de componentização de software para a aplicação prática de tais conceitos [Pressman, 2002].

Problemas arquiteturais

- Metodologias como RUP [Jacobson et al., 1999] descrevem a arquitetura como um ponto-chave para construção de sistemas. Ela deve ser descrita, validada e estabilizada antes do início da implementação. Além disso, deve servir para identificar a possibilidade de utilização de componentes já existentes. No sistema legado, nenhum deste pontos foi coberto, não havendo uma arquitetura bem definida, nem sequer a definição de componentes.

Solução proposta

- Um abordagem possível seria a utilização de uma arquitetura baseada em camadas [Buschman et al., 1996], o que permitiria o isolamento de: comunicação com portas seriais, lógica de envio de mensagens, e camada de interação com usuário. Facilitando o processo de componentização e definição clara de responsabilidades de cada grupo de classes. Isto permitiria uma maior adaptabilidade a diferentes contextos, por exemplo, a execução do programa usando interfaces gráficas, ou ainda através de linhas de comando, que foi uma mudança solicitada nos requisitos da aplicação após sua implementação, e que não pode ser implementada devido ao alto acoplamento da interface gráfica aos elementos que tratam das regras de envio de mensagens.

4. Reestruturação

O processo de reestruturação foi baseado na abordagem de orientação a problema [Zimmer, 1994]. Neste tipo de abordagem, o aprendizado adquirido sobre o domínio da aplicação legada serve como entrada para a construção de um sistema novo onde as erros encontrados tendem a não ser repetidos. O primeiro erro que tentamos evitar foi a não-existência de uma arquitetura bem definida. Para resolver este problema, elaboramos a arquitetura em camadas mostrada na Figura 2.

Observamos que o sistema anterior era basicamente monolítico, sem separação clara das funcionalidades desempenhadas por cada elemento que constitui a aplicação, e

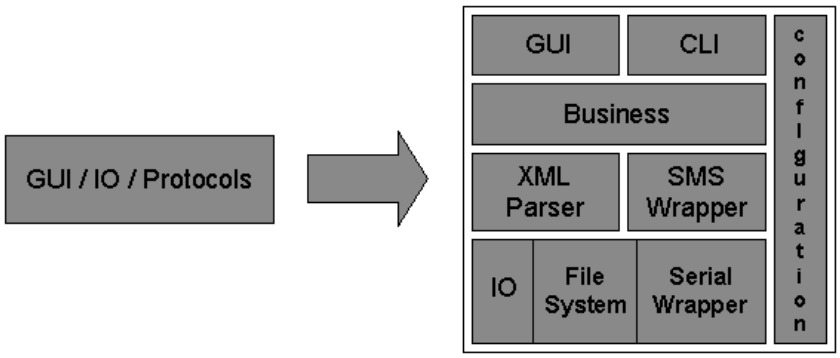

Figura 2: Nova arquitetura da aplicação

Componente	Função
GUI:	**G**raphical **U**ser **I**nterface - Define a camada que trata de toda a necessidade de representação gráfica das mensagens manipuladas pela aplicação, e interação com o usuário, que poderá ser tanto o engenheiro de testes responsável pela criação dos testes, quanto o engenheiro de testes responsável pela execução;
CLI:	**C**ommand **L**ine **I**nterface - Define mecanismos de interação com a ferramenta através de linha de comando;
Business:	Permite que aplicações tenha uma interface comum de acesso aos serviços de leitura, escrita e envio de mensagens, obedecendo as restrições impostas pelas regras de negócio;
XML Parser:	Responsável pela leitura e escrita de arquivos no formato XML específico utilizado pela aplicação;
SMS Wrapper:	Realiza as conversões entre os tipos de objetos usados pela GUI, pelo leitor XML, pelo Serial Wapper, e pelo File System, realizando conversão entre os objetos (ou protocolos) aceitos por cada componente. Alguns dos protocolos tratados são SMS e ESM da 3GPP , NSM - Nokia Smart Message [Nokia, 2000], e Multimedia Messaging Service [OMA, 2002];
IO:	Representa a camada de acesso aos dispositivos orientados a caracteres, como o sistema de arquivos e as portas seriais;
File System:	Define a interface entre a aplicação e um sistema de arquivos;
Serial Wrapper:	A função deste componente é esconder qual protocolo de baixo nível é utilizado pelo computador para a troca de mensagens com o celular, pois além do protocolo de codificação do SMS, cada celular pode apresentar um protocolo de comunicação serial diferente, por exemplo, Text/PDU/Block Mode [3GPP, 2000] definidos pela 3GPP;
Configuration:	Este componente é responsável por toda a parametrização do software, desde a camada de IO até as interfaces gráficas.

Tabela 1: Componentes da nova arquitetura

reformulamos para um modelo em camadas, com separação clara de funcionalidades dos componentes.

Além da arquitetura, detectamos também erros oriundos de decisões de projeto não documentadas, como a disparidade entre o formato do arquivo gerado para gravação no sistema de arquivos e o enviado pela porta serial, erro que corrigimos na nossa reestruturação. Observamos também um número de classes com definições de grande quantidade de parâmetros *hardcoded* , dificultando a parametrização do software,

ou simplesmente gerando a necessidade de recompilar todo o sistema quando pequenas mudanças forem solicitadas, necessidade que poderia ser facilmente evitada caso a configuração do software fosse mais flexível.

Na próxima seção identificaremos as oportunidades de uso dos padrões de projeto para remodelagem do sistema, baseado na arquitetura descrita acima.

4.1. Aplicações de padrões de projeto

Padrões de projeto descrevem práticas comuns utilizadas durante o desenvolvimento de sistemas computacionais para corrigir erros de modelagem de software ou para agregar qualidade a estes sistemas. Os padrões escolhidos foram retirados do catálogo de padrões contido em [Gamma et al., 1995], que é uma coletânea bastante abrangente e amplamente difundida na comunidade. [Gamma et al., 1995] também mostra abordagem para a descoberta e uso de padrões, alguns dos quais utilizamos neste trabalho.

Baseado no *design* recuperado e nas características previamente identificadas no estudo do sistema legado (seção 3.2), encontramos diversos erros de modelagem e decisão de projeto que poderiam ser atacados através da boa utilização de padrões de projeto. A experiência adquirida foi de grande importância durante a transição do modelo análise para o modelo de projeto da solução proposta.

Assim como no catálogo de padrões [Gamma et al., 1995], dividimos os padrões de acordo com os três grupos: Estruturais, Comportamentais e Criacionais. A seguir mostraremos quais padrões de cada grupo foram utilizados para implementações dentro da nova arquitetura:

Estruturais :

Facade : padrão usado para a criação de uma classe única que representa todo o sistema e centraliza a comunicação entre as camadas de interface com o usuário e a camada de regras de negócios do sistema. A ausência deste padrão impossibilitou a criação de um a CLI para a aplicação existente. A nova arquitetura já prevê esta facilidade;

Decorator : utilizado para gerar abstrações sobre: o tipo de escrita e leitura de IO nas implementações dos componentes File System e Serial Wrapper; os processos de conversão entre protocolos para o envio de mensagens usados pelo celular no SMS Wraper;

Adapter : usado no tratamento de eventos lançados pela camada de interface gráfica do usuário. Usamos um adapter para cada tipo de evento lançado, evitando a construção de máquinas de estados complexas para o tratamento de eventos, prática encontrada na aplicação legada. Isto aumenta a facilidade de manutenção do código e a sua legibilidade;

Composite : como SMS Wraper lida com diversas extensões do protocolo SMS (que muitas vezes utilizam mais de um pacote SMS para compor uma mensagem) é importante a utilização deste padrão para abstrair a multiplicidade dos pacotes de uma mensagem. Alguns exemplos de extensões do protocolo SMS são: o EMS, que adiciona suporte a som e imagens nas mensagens SMS e o MMS (Multimidia Messaging) que será o padrão para envio de mensagens com componentes multimídia em redes celulares 3G.

Comportamentais :

Strategy : devido a necessidade da aplicação suportar o envio de mensagens utilizando diversos modelos de telefones celulares, cada um com suporte a diferentes protocolos de comunicação serial com terminais remotos, utilizamos este padrão para implementar as diferentes heurísticas dos protocolos possíveis e controlar a comunicação com o celular;

Visitor : utilizado para varredura das estruturas hierárquicas que representam as mensagem. Durante estas varreduras poderemos converter os tipos da mensagens, construir representações gráficas, ou simplesmente imprimir a estrutura.

Criacionais :

Builder : será usado tanto na criação de cada um dos *Information Element* [3GPP, 2002] extraídos durante o processamento de uma mensagem SMS, quanto para criação dos elementos que representam graficamente uma mensagem;

Abstract Factory : Quando lemos um arquivo XML, ou binário precisamos realizar um processo de conversão entre o formato de representação do arquivo e o formato que a aplicação utiliza em memória. Cada tipo de mensagem (EMS, MMS, NSM,etc) possui um formato específico de escrita e leitura, sendo este padrão utilizado para definir os diferentes tipos de leitores destes tipos de mensagens;

Singleton : para controlar o acesso concorrente às portas seriais e assegurar a consistência na comunicação com os telefones celulares, utilizamos *Singletons* para gerenciar cada uma das portas;

Factory method : usado para abstrair o subtipo de um *Information Element*, por exemplo os elementos *imelody* e *pre-definded sound*, ambos são objetos de som da mensagem.

De fato, observamos que diversos padrões descritos na literatura poderiam ser aplicados para beneficiar a aplicação em questão. Sendo assim, utilizamos esses padrões na elaboração do novo modelo da aplicação. O resultado este processo é descrito na próxima seção.

4.2. Nova modelagem da aplicação

Aqui, apresentamos o novo modelo de classes simplificado com as modificações necessárias para inclusão dos padrões encontrados. À nova aplicação resultante da reestruturação demos o nome de Phoenix.

No diagrama da Figura 3, podemos identificar os subconjuntos de classes responsáveis pela implementação de cada um dos padrões. A $PhoenixFacade$ é responsável pelas regras de negócio implementando o padrão *Facade*. Ao utilizar a $PhoenixFacade$, a interface gráfica ($PhoenixGUI$) torna-se independente da camada de negócio, assim como a classe a responsável pela interface textual ($PhoenixCLI$).

Todos os protocolos que estendem o SMS (no diagrama representados pelas classes EMS, MMS e NSM) são na verdade composições de mensagens SMS (emprego do padrão *Composite*), que precisam de leitores específicos como $EMSFactory$, $MMSFactoy$ e $NSMFactory$ para serem construídos (*Abstract Factory*).

A visualização destas estruturas é construída (*Builder*) através de uma varredura (*Visitor*) de toda a estrutura hierárquica de uma mensagem e sua conversão para uma representação gráfica. Estas duas tarefas são executadas pelas classes $TreeBuilder$ e $NodeNavigator$ respectivamente, sendo uma instância da $TreeView$ o resultado desta varredura. Neste processo de varredura o $TreeBuilder$ pode requisitar que cada componente da mensagem forneça uma representação gráfica exclusiva de cada tipo (por exemplo, uma image, um label ou um som) através de um método (*TemplateMethod*) apropriado. Todos os tratamentos de eventos gerados pelas interfaces devem ser realizados através de *Adapters*, seguindo a abordagem padrão de tratamento de eventos de Java.

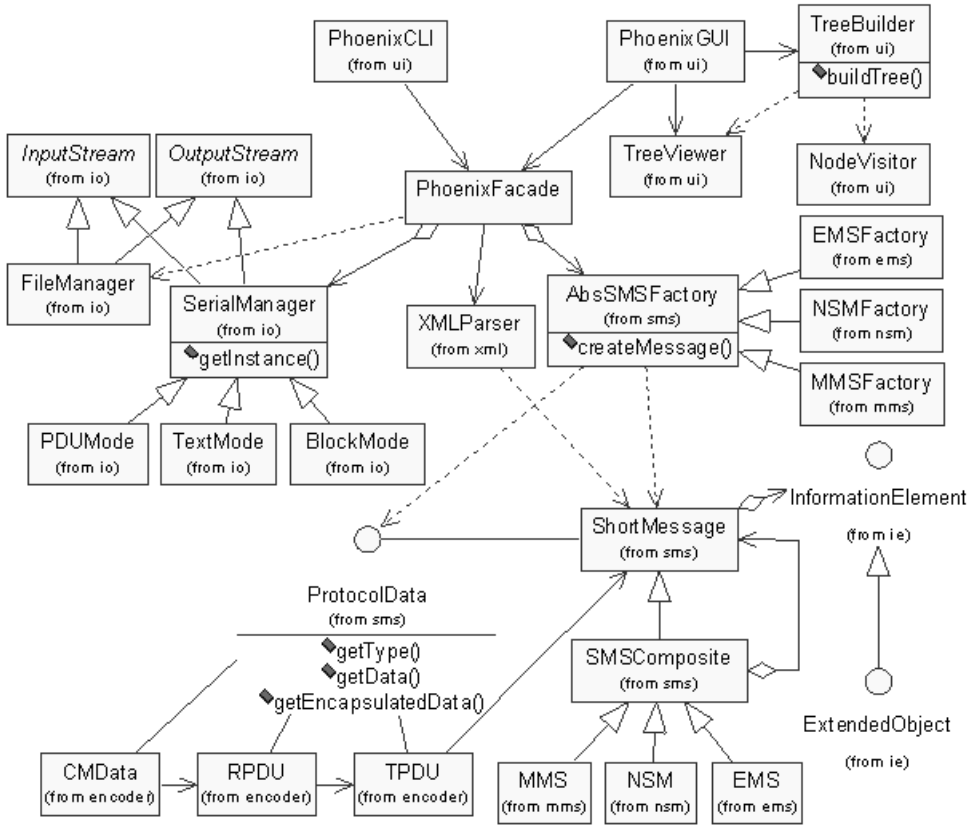

Figura 3: Diagrama de classes simplificado da modelagem atual do sistema

O $SerialManager$ (implementado como um *Singleton*) é responsável pelo gerenciamento das portas seriais e pela delegação dos diferentes tipos da comunicação com as interfaces seriais (*Strategy*). Tanto o $SerialManager$ quanto o $FileManager$ estendem classes do pacote utilitário `java.io`, com o objetivo de obter todos as facilidades (*Decorators*) deste pacote. Outro exemplo da utilização de *Decorators* é o encapsulamento estabelecido entre as classes $ShortMessage$, $TPDU$, $RPDU$ e $CMData$. A $ShortMessage$ que representa o conteúdo da mensagem é encapsulada por um $TPDU$, o $TPDU$ que contém meta-informações sobre a mensagem é encapsulada por um $RPDU$, e assim por diante.

É possível observar que o modelo é mais simples que o anterior (Figura 1), sendo mais fácil identificar as responsabilidades de cada um dos componentes e os relacionamentos entre cada um deles.

Nota-se também a existência de algumas associações entre os papeis das classes nas duas modelagem, sendo possível reutilizar procedimentos e algoritmos nas classes do novo sistema (terceira e quarta fase do COREM [Gall et al., 1995]). Tal procedimento tem como objetivo o reaproveitamento do conhecimento especializado contido nessas classes, partindo-se do principio que o sistema foi testado e validado durante todo o seu tempo de vida.

5. Conclusão

Através do uso de padrões de projeto conseguimos deixar a estrutura da aplicação mais legível e modularizada. Diminuindo o esforço de futuros desenvolvedores no processo

de manutenção e extensão da aplicação. Com as reestruturações realizadas conseguimos propiciar a realização das mudanças requisitadas a ferramenta.

A aplicação de um método de reestruturação, permitiu através de ações simples, a descoberta de conhecimento do sistema legado e a associação de seus elementos à nova modelagem. Ajudando bastante a tomada de decisões arquiteturais durante o desenvolvimento do novo sistema. Esperamos que esta experiência, incluindo o processo de reestruturação, sirva de apoio para outras iniciativas nesta área.

Achamos um bom tema para estudo futuro, a leitura de *antpatterns* e de como esses poderiam ser aplicados neste contexto.

Referências

3GPP (2000). Equipment (DTE-DCE) interface for Short Message Service (SMS) and Cell Broadcast (CBS). Technical Report 27005-310, 3rd Generation Partnership Project. Disponível em http://www.3gpp.org/ftp/Specs/.

3GPP (2002). Technical realization of the Short Message Service (SMS). Technical Report 23040-601, 3rd Generation Partnership Project. Disponível em http://www.3gpp.org/ftp/Specs/.

Basili, V. R. (1990). Viewing maintenace as reuse-oriented software development. IEEE Software 7(1), pp 19-25.

Booch, G., Jacobson, I., Rumbaugh, J., and Rumbaugh, J. (1998). *The Unified Modeling Language User Guide*. Addison-Wesley, 1st edition.

Buschman, F., Meunier, R., Rohnert, H., Sommerland, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons.

Gall, H., Klosch, R., and Mittermeir, R. (1995). Architectural transformation of legacy systems. Gall, H. C., Klosch, R. R., Mittermeir, R. T., Architectural transformation of legacy systems, Proceedings of the ICSE-17 Workshop on Program Transformation for Software Evolution, Seattle, EUA.

Gall, H. and Weidl, J. (1998). Binding object models to source code: an approach to object-oriented re-architecting. Proceedings of the 22nd Computer Software Applications Conference, Viena, Austria.

Gamma, E., Helm, R., Johnson, R., , and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2000). *The Java^{TM} Language Specification*. Addison-Wesley, 2nd edition.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley, 1st edition.

McNatt, W. B. and Bieman, J. M. (2001). Coupling of design patterns: Common practices and their benefits. 25th Annual International Computer Software and Applications Conference (COMPSAC'01), Chicago, EUA, pp 574.

Nokia (2000). Smart messaging specification. Technical Report 3.0.0, Nokia Mobile Phones Ltd. Disponível em http://www.forum.nokia.com/smsforum/main/1,6566,1_2_5_1,00.html.

OMA (2002). Multimedia messaging service specification. Technical Report 1.1, Open Mobile Alliance. Disponível em

http://www.openmobilealliance.org/omacopyrightNEW.asp?doc=OMAMMS-v1_1-20021104C.zip.

Pressman, R. S. (2002). *Engenharia de Software*. McGraw-Hill, 5th edition.

Rumbaugh, J., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Designs*. Prentice-Hall.

Sun (1999). Code conventions for the javatm programming language. Disponível em http://java.sun.com/docs/codeconv/.

W3C (1998). XML (eXtensible Markup Language). Disponível em http://www.w3.org/XML.

Watt, D. A. (1994). *Programming Language Concepts and Paradigms*. Prentice Hall International.

Zimmer, W. (1994). Experiences using Design Patterns to Reorganize an Object-Oriented Application. W. Zimmer, Experiences using Design Patterns to Reorganize an Object-Oriented Application, Position Paper for the Pattern Workshop, The 8th European Conference on Object-Oriented Programming, Bologna, Itália.