UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Definição e Validação de Refatoramentos de Software em Larga Escala

Glaucimar da Silva Aguiar

Campina Grande – PB
Maio – 2002

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE PÓS GRADUAÇÃO EM INFORMÁTICA

Definição e Validação de Refatoramentos de Software em Larga Escala

Glaucimar da Silva Aguiar

Dissertação submetida à Coordenação de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal da Paraíba como requisito parcial para a obtenção do grau de Mestre em Ciências (MSc).

Orientador: Jacques Philippe Sauvé

Campina Grande – PB
Maio - 2002

UFPb - BIBLIOTECA - CAMPUS II
766 23.08.2002

AGUIAR, Glaucimar da Silva

A282D

Definição e Validação de Refatoramentos de Software em Larga Escala

Dissertação (mestrado), Universidade federal de Campina Grande, Centro de Ciências e tecnologia, Coordenação de Pós-graduação em Informática, Campina Grande – Pb, Julho de 2002.

186 p. Il.

Orientador: Jacques Philippe Sauvé

Palavras-chaves:

1. Engenharia de Software
2. Refatoramentos de Software
3. Manutenção de Sistemas

CDU – 519.683

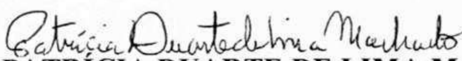
"DEFINIÇÃO E VALIDAÇÃO DE REFATORAMENTOS DE SOFTWARE EM LARGA ESCALA"

GLAUCIMAR DA SILVA AGUIAR

DISSERTAÇÃO APROVADA EM 12.07.2002

PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Orientador

PROFª FRANCILENE PROCÓPIO GARCIA , D.Sc
Examinadora



PROFª PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D
Examinadora

PROFª ROSÂNGELA APARECIDA D. PENTEADO, Drª
Examinadora

CAMPINA GRANDE – PB

À Neide, Humberto, Gilmara, Édna, Solon

À Thilly

Agradecimentos

Ao meu orientador, Jacques, pela confiança de que poderíamos realizar um bom trabalho.

A todos da Light Infocon, por permitirem e apoiarem este trabalho. Especialmente a Vladimir, Alessandro e Risalva pela paciência e disponibilidade em ajudar nos momentos de dificuldade.

Às minhas amigas, Eliane e Ladjane, que estiveram sempre ao meu lado como fonte de força e estímulo.

À Zane, pelo apoio e amizade na reta final desta caminhada.

À Aninha, pela amizade e auxílio sempre constantes.

À Fátima, pelo carinho e confiança.

Aos meus amigos e amigas, pelo apoio e imensa torcida.

Aos membros da Banca Examinadora pela atenção e ricas sugestões.

Aos professores e colegas do DSC, pelos valorosos conhecimentos compartilhados durante nossa convivência.

Sumário

1 Introdução 1
1.1 Refatoramentos de software 2
1.2 Objetivos 4
1.3 Estrutura da dissertação 4
2 A evolução de sistemas de software com refatoramento 6
2.1 A evolução de sistemas de software 6
2.2 As atividades de manutenção 6
2.3 Refatoramento de software 7
2.3.1 Definição 7
2.3.2 Requisitos para o refatoramento 8
2.3.3 Quando refatorar? 8
2.3.4 O catálogo de refatoramentos 10
2.3.5 Formato 11
2.3.6 Exemplo 11
3 O projeto de refatoramento 15
3.1 A Metodologia 15
3.2 O estudo de caso 16
3.2.1 Preparação 16
3.2.2 Estudo inicial do código do sistema 18
3.2.3 Criação de testes 19
3.2.4 Definição e aplicação dos refatoramentos 22
4 Refatoramentos de software em larga escala 23
4.1 Definição 23
4.2 Motivação 23
4.3 Modelo de apresentação 24
4.3.1 Nome 24
4.3.2 Fundamentos 24
4.3.3 Sumário 24
4.3.4 Condições de aplicação 25
4.3.5 Mecanismo 25
4.3.6 Conseqüências 25
4.3.7 Exemplo 25
4.4 Classificação 25
4.5 Apresentação dos refatoramentos 27
5 Refatoramentos que diminuem acoplamento 28
5.1 Eliminação de dependências cíclicas 28
5.1.1 Fundamentos 28
5.1.2 Sumário 29
5.1.3 Condições de aplicação 30

5.1.4 Mecanismo ... 31
5.1.5 Conseqüências ... 31
5.1.6 Exemplo ... 32
5.2 Diminuição de acoplamento através de tipos de dados ... 40
5.2.1 Sumário ... 40
5.2.2 Condições de aplicação ... 41
5.2.3 Mecanismo ... 42
5.2.4 Consequências ... 43
5.2.5 Exemplo ... 43
5.3 Isolamento de dependência de plataforma ... 47
5.3.1 Introdução ... 47
5.3.2 Fundamentos ... 48
5.3.3 Problemas ... 49
5.3.4 Sumário ... 50
5.3.5 Condições de aplicação ... 51
5.3.6 Mecanismo ... 52
5.3.7 Conseqüências ... 54
5.3.8 Exemplo ... 54
6 Refatoramentos que aumentam a coesão ... 77
6.1 Divisão de classe de fronteira ... 77
6.1.1 Fundamentos ... 77
6.1.2 Sumário ... 78
6.1.3 Condições de aplicação ... 80
6.1.4 Mecanismo ... 80
6.1.5 Conseqüências ... 82
6.1.6 Exemplo ... 82
6.2 Divisão de módulos grandes ... 97
6.2.1 Sumário ... 97
6.2.2 Condições de aplicação ... 97
6.2.3 Mecanismo ... 98
6.2.4 Conseqüências ... 99
6.2.5 Exemplo ... 100
6.3 Junção de módulos com funcionalidade incompleta ... 103
6.3.1 Sumário ... 103
6.3.2 Condições de aplicação ... 103
6.3.3 Mecanismo ... 103
6.3.4 Conseqüências ... 104
6.3.5 Exemplo ... 104
7 Refatoramentos que introduzem abstrações ... 107
7.1 Modelagem de conceitos da solução através de classe ... 107

7.1.1 Sumário ... 107
7.1.2 Condições de aplicação ... 107
7.1.3 Mecanismo ... 108
7.1.4 Conseqüências ... 109
7.1.5 Exemplo ... 109
7.2 Separação de persistência ... 120
7.2.1 Fundamentos ... 120
7.2.2 Sumário ... 121
7.2.3 Condições de aplicação ... 122
7.2.4 Mecanismo ... 122
7.2.5 Conseqüências ... 123
7.2.6 Exemplo ... 123
8 Refatoramentos que introduzem padrões de projeto ... 135
8.1 Introdução de *Observer* ... 135
8.1.1 Fundamentos ... 135
8.1.2 Sumário ... 136
8.1.3 Condições de aplicação ... 136
8.1.4 Mecanismo ... 136
8.1.5 Conseqüências ... 137
8.1.6 Exemplo ... 138
8.2 Introdução de *Singleton* ... 145
8.2.1 Fundamentos ... 145
8.2.2 Sumário ... 145
8.2.3 Condições de aplicação ... 145
8.2.4 Mecanismo ... 146
8.2.5 Conseqüências ... 147
8.2.6 Exemplo ... 147
8.3 Introdução de *Singleton* de vários contextos ... 151
8.3.1 Fundamentos ... 151
8.3.2 Sumário ... 151
8.3.3 Condições de aplicação ... 152
8.3.4 Mecanismo ... 152
8.3.5 Conseqüências ... 152
8.3.6 Exemplo ... 153
9 Validação dos refatoramentos ... 156
9.1 Modelo de avaliação ... 156
9.2 Modelo conceitual ... 156
9.2.1 Métricas ... 157
9.2.2 Categorias ... 158
9.3 Avaliação do projeto de refatoramento ... 158

9.3.1 Categoria: acoplamento ... 158
9.3.2 Categoria: coesão ... 163
9.3.3 Categoria: complexidade ... 166
9.3.4 Categoria: polimorfismo ... 172
9.4 Conclusões ... 173
10 Conclusões ... 175
10.1 Problemas e observações ... 175
10.2 A validação dos refatoramentos ... 177
10.3 Sumário das contribuições ... 177
10.4 Trabalho futuro ... 178
11 Apêndice A - Identificação dos grandes pacotes do LightBase Server ... 179
11.1 A identificação dos módulos ... 179
11.2 Lbs ... 179
11.3 LbwServ ... 180
11.4 AppManager ... 180
11.5 IPWorks ... 180
11.6 LiParser ... 180
11.7 Slot ... 180
11.8 Sort ... 180
11.9 Comparator ... 181
11.10 LT ... 181
11.11 HtmlTools ... 181
11.12 CtawLib ... 181
11.13 Compress ... 181
11.14 Alfc ... 182
11.15 LbStart ... 182
11.16 LiSvc ... 182
11.17 RpcStuff ... 182
11.18 Li ... 182
11.19 LiFile ... 182
11.20 Crypt ... 182
12 Referências bibliográficas ... 184

Lista de Figuras

Figura 1.1 - Curva custo da correção de erros de Boehm [Boehm, 1981]. 2
Figura 1.2 - Curva do custo da implantação de mudanças [Beck, 1999]. 2
Figura 5.1 - Ilustração de uma árvore de dependências. 29
Figura 5.2 – Exemplos de dependências cíclicas. 30
Figura 5.3 - Grafo de dependências entre os módulos do exemplo. 32
Figura 5.4 - Origem da dependência entre os módulos *lbs* e *li*. 33
Figura 5.5 - Dependências entre os módulos *lbs* e *li* após a recolocação das classes LBSC_List e LBSC_Node. 34
Figura 5.6 - Refatoramento *Diminuição de acoplamento através de tipos de dados*. 41
Figura 5.7 - Grafo de dependências entre os módulos do servidor LightBase. 44
Figura 5.8 - Detalhamento das conexões entre os módulos *slot* e *lbs*. 46
Figura 5.9 - Relação entre os módulos *slot* e *lbs* após a aplicação do refatoramento. 47
Figura 5.10 - Refatoramento *Isolamento de dependência de plataforma*. 51
Figura 5.11 - Projeto do sistema – linhas de execução (*threads*). 62
Figura 5.12 - Projeto do sistema - arquivos de inicialização. 73
Figura 6.1 - Padrão de projeto *Fachada*. 78
Figura 6.2 - Refatoramento *Divisão de classe de fronteira*. 79
Figura 6.3 - Relação entre as classes `Base`, `Sessao` e `Campo` do exemplo. 83
Figura 6.4 - Ilustração do refatoramento *Divisão de módulos grandes*. 97
Figura 6.5 - Dependências existente entre os módulos criados a partir de *li*. 102
Figura 6.6 - Dependências dos módulos *Time*, *Lists* e *Security*. 105
Figura 6.7 - Dependências após unificação dos módulos *Time* e *Lists*. 106
Figura 7.1 - Ilustração do padrão *Bridge*. 121
Figura 7.2 - Ilustração do refatoramento *Separação de persistência*. 121
Figura 7.3 - Estratégia de persistência adotada pela classe Base antes do refatoramento. 126
Figura 7.4 - Estratégia de persistência utilizada pela classe Base após a aplicação do refatoramento. 126
Figura 8.1 - Representação resumida do padrão *Observer*. 135
Figura 8.2 – Idéia básica do refatoramento *Introdução de Observer*. 136
Figura 8.3 - Conceitos envolvidos no reprocessamento de uma base de dados e interessados no progresso da operação. 139
Figura 8.4 -Diagrama de classes para o reprocessamento de bases. 139
Figura 8.5 - Diagrama de seqüência simplificado referente à operação de reprocessamento. 139
Figura 8.6 - Diagrama de classes após a aplicação do refatoramento. 141
Figura 8.7 - Padrão de projeto *Singleton*. 145
Figura 8.8 - Refatoramento *Introdução de Singleton de vários contextos*. 151
Figura 9.1 - Histograma mostrando a distribuição do número de classes do sistema em função dos valores da métrica CBO antes e depois do projeto de refatoramento. 159
Figura 9.2 - Histograma da distribuição do número de classes do sistema em função dos valores da métrica DIT antes e depois do projeto de refatoramento. 160

Figura 9.3 - Histograma da distribuição do número de classe em função dos valores da métrica DCC............. 161
Figura 9.4 - Gráfico com as médias das métricas de acoplamento. 162
Figura 9.5 - Histograma da distribuição do número de classes em função dos valores da métrica LCOM. 164
Figura 9.6 - Porcentagem dos valores da métrica LCOM. 164
Figura 9.7 - Valores da métrica CIS antes e depois do projeto de refatoramento. 165
Figura 9.8 - Valores da métrica MCCABE_Avg antes e depois do refatoramento. 166
Figura 9.9 - Valores da métrica MCCABE_Max antes e depois do refatoramento. 167
Figura 9.10 - Valores da métrica WMC antes e depois do refatoramento. 168
Figura 9.11 - Valores de WMC antes e depois do refatoramento. 169
Figura 9.12 - Valores da métrica RFC antes e depois do refatoramento. 170
Figura 9.13 - Gráfico com a média das métricas de complexidade antes e depois do refatoramento................. 171
Figura 9.14 - Valores da métrica NOP antes e depois do refatoramento. 172

Lista de Tabelas

Tabela 3.1 - Metodologia criada para o desenvolvimento do projeto de refatoramento deste trabalho. 16
Tabela 3.2 - Lista dos requisitos do projeto de refatoramento do serivdor LightBase. 17
Tabela 3.3 - Lista das restrições do projeto de refatoramento do estudo de caso. 17
Tabela 3.4 - Resumo das informações obtidas na atividade de estudo inicial do código do sistema. 19
Tabela 3.5 - Sumário da estratégia de testes adotada. 21
Tabela 4.1 - Classificação dos refatoramentos propostos. 26
Tabela 5.1 - Recursos do sistema operacional utilizados pelo sistema. 55
Tabela 5.2 - Símbolos externos identificados pelo programa de análise de arquivos-objeto. 75
Tabela 6.1- Lista dos módulos do sistema exemplo. 101
Tabela 6.2 – Classes e funcionalidades de *li*. 101
Tabela 6.3 - Agrupamento de classes em funcionalidades. 102
Tabela 6.4 - Identificação de módulos e funcionalidades. 104
Tabela 9.1 - As métricas, e seus significados, utilizadas na análise dos resultados. 157
Tabela 9.2 - Grupos de métricas e seus componentes. 158
Tabela 9.3 - Valores das médias de acoplamento. 163
Tabela 9.4- Valores das médias das métricas de Complexidade antes e depois do refatoramento. 171

Resumo

Esta dissertação descreve uma maneira de lidar com os problemas causados pela perda de estrutura de sistemas de software com milhares de linhas de código que foram produzidos sem o uso de técnicas de refatoramento e um bom conjunto de testes. O objetivo desta pesquisa é organizar a reestruturação de sistemas de software de grande porte através da definição de refatoramentos apropriados. Primeiramente, um conjunto de refatoramentos é definido com base do projeto de reestruturação de um sistema de software real. Para avaliar os resultados da aplicação destes refatoramentos, apresenta-se um modelo de avaliação que é aplicado ao sistema usado como estudo de caso.

Abstract

This dissertation describes how to deal with problems caused by the loss of structure in software systems with thousands of line of code that were produced without the use of refactoring techniques and a good test suite. The purpose of this research is to organize the restructuring of large software systems by defining appropriate refactoring techniques. First, it defines several large refactorings and their application to a large scale real software system. In order to evaluate the benefits of this collection of refactorings, an evaluation model is presented and applied to the system used as case study.

1 INTRODUÇÃO

Uma característica central na evolução de grandes sistemas de software é que mudanças – necessárias para adicionar novas funcionalidades, acomodar novo hardware, e corrigir erros – tornam-se mais difíceis com o tempo [Eick, 2001]. A evolução do sistema ao longo dos anos pode levar a sistemas desnecessariamente complexos e inflexíveis. É difícil prever os efeitos de uma mudança, assim, a evolução e manutenção destes sistemas podem tornar-se progressivamente mais dispendiosas.

As tarefas executadas durante a fase de manutenção do software podem ser divididas em três categorias: corretiva, adaptativa e de aperfeiçoamento [Pfleeger, 1998]. A primeira delas, a corretiva, refere-se à correção de erros. A segunda, adaptativa, refere-se às mudanças realizadas para adaptar o sistema a uma nova necessidade, a um novo hardware, a um novo banco de dados, etc. A terceira categoria, de aperfeiçoamento, engloba as mudanças realizadas com o objetivo de melhorar algum aspecto do sistema [Eick, 2001]. Apesar da possibilidade de perdas na estrutura, as mudanças são necessárias por serem responsáveis pela capacidade de agregar valor ao sistema [Eick, 2001]. Esta perda de estrutura é também conhecida como Entropia do Software, segundo a qual o sistema tende a se degradar à medida que evolui e acomoda mudanças.

Para ilustrar o custo de alterações no sistema, a Figura 1.1 representa a curva "custo de correção de erros" de Boehm [Boehm, 1981]. Segundo esta curva, o custo de correção de um erro aumenta exponencialmente [Beck, 1999] à medida que o projeto prossegue através das fases de análise, projeto, codificação, teste e produção.

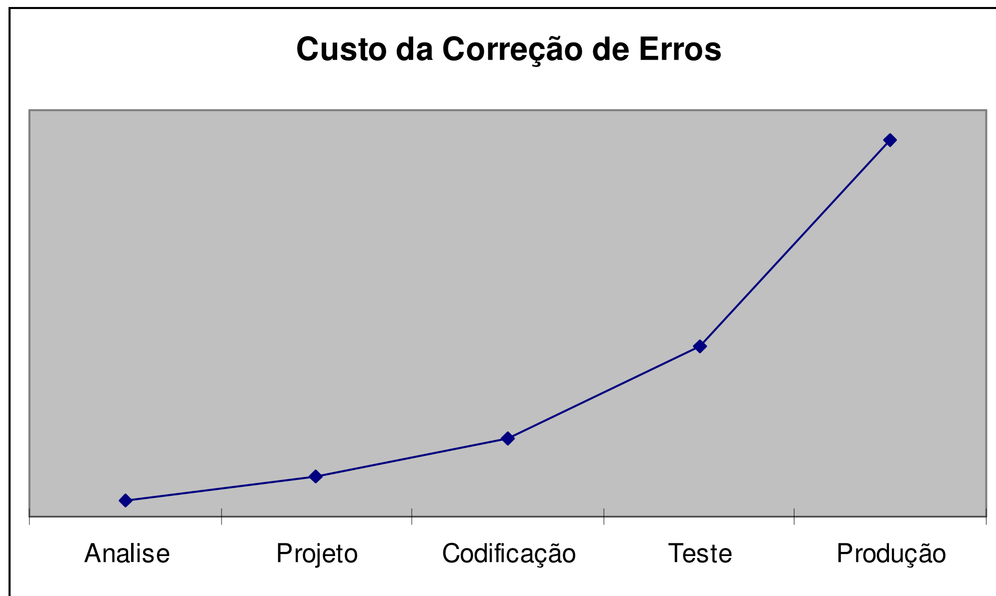


Figura 1.1 - Curva custo da correção de erros de Boehm [Boehm, 1981].

1.1 Refatoramentos de software

Refatoramentos são mudanças efetuadas no código de um sistema para aumentar a possibilidade de reuso de seus componentes e também como forma de torná-lo mais fácil de manter e entender, ou seja, é uma forma de melhorar a estrutura do sistema. Entende-se, deste modo, que refatoramentos podem contribuir para a redução dos custos ilustrados no gráfico da Figura 1.1.

A promessa técnica de XP (Extreme Programming) proposta por Kent Beck é exatamente a redução dos custos associados a mudanças [Beck, 1999]. A sugestão é que a curva do custo das alterações do sistema ao longo do tempo se pareça com a Figura 1.2. Uma das técnicas utilizadas por XP e que compõe a lista de razões para esta mudança nos custos de alteração do sistema é refatoramento.

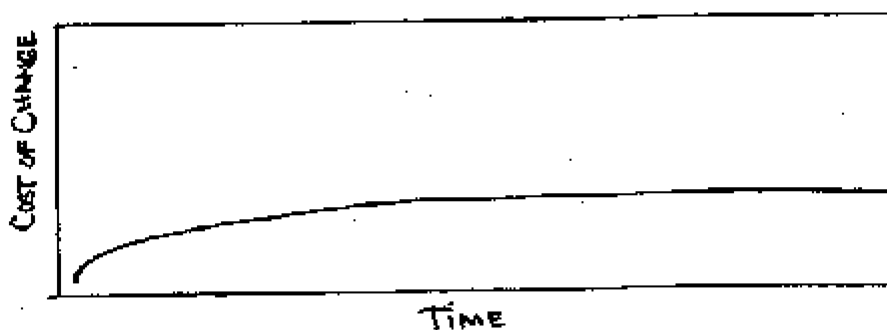


Figura 1.2 - Curva do custo da implantação de mudanças [Beck, 1999].

O uso freqüente da técnica ajuda a diminuir os efeitos causados pela perda de estrutura do sistema e permite, ainda, que o projeto seja modificado para melhor acomodar e se adaptar às novas necessidades [Opdyke, 1992].

Os refatoramentos existentes e propostos por Fowler [Fowler, 1999] resolvem problemas de compreensão, de reuso de classes, entre outros. No entanto, eles se mantêm no nível das classes do sistema, dificultando sua aplicação na reestruturação de um sistema como um todo. Integração e interdependências entre módulos do sistema, acoplamento deste sistema com o ambiente hospedeiro são exemplos de situações não tratadas no catálogo apresentado por Fowler [Fowler, 1999].

Apesar da valorosa contribuição destas técnicas no combate à Entropia do Software, é inadequado ou, no mínimo, incompleto, o uso das mesmas técnicas em projetos de reestruturação de sistemas de grande porte. Poucos dos refatoramentos propostos em [Fowler, 1999] envolvem a manipulação de várias classes, caso típico na reestruturação de sistemas deste tipo. Em projetos de reestruturação é possível encontrar situações em que seja necessário manipular módulos inteiros, organizar a relação entre módulos do sistema e examinar a relação entre várias classes ao mesmo tempo.

A reengenharia de sistemas legados – aqueles que têm valor e que devem continuar evoluindo e acomodando mudanças – é amplamente reconhecida como um dos mais significativos desafios enfrentados pelos engenheiros de software [Stevens, 1998]. Este problema pode afetar empresas seriamente, pois uma falha na reengenharia pode comprometer a tentativa de a empresa continuar competitiva. Os negócios atualmente precisam adaptar-se rapidamente às mudanças no ambiente para obter vantagens de novas oportunidades.

Sistemas modernos construídos em linguagens orientadas a objetos ou através do uso de componentes podem tornar-se difícil de manter e modificar devido aos anos de modificações que podem obscurecer sua estrutura [Stevens, 1998]. Assim, a dificuldade de evolução dos sistemas de software é um problema que afeta muitas organizações e muitos sistemas, sendo eles construídos sob o paradigma orientado a objetos, utilizando componentes ou sob o paradigma funcional.

Na comunidade de engenharia de software orientada a objetos, padrões de projeto [Gamma, 1994] têm sido adotados como uma maneira efetiva de transmitir experiências sobre o projeto de software [Ducasse, 1999]. Fowler [Fowler, 1999] utiliza a idéia de padrões para transmitir conhecimentos adquiridos no refatoramento de sistemas de software. A proposta

desta dissertação é reutilizar a idéia de padrões para transmitir experiências adquiridas no refatoramento de sistemas de grande porte.

1.2 Objetivos

O objetivo deste trabalho é propor meios de lidar com os problemas gerados pela Entropia de Software em sistemas de grande porte. Para tanto, apresentam-se estratégias de ataque a alguns destes problemas identificados com base no projeto de reestruturação de um sistema em particular. A apresentação destas estratégias é feita através da reutilização da idéia de refatoramentos proposta em [Fowler, 1999].

Os refatoramentos apresentados neste trabalho foram identificados com base nas necessidades do estudo de caso utilizado e não têm a pretensão de abordar todas as necessidades de sistemas de grande porte. Eles referem-se especialmente a sistemas orientados a objetos. Também não é objetivo deste trabalho propor uma metodologia formal para projetos de reestruturação de sistemas. O escopo do trabalho restringe-se a sistemas orientados a objetos e aos problemas enfrentados no projeto de refatoramento do estudo de caso que podem ser úteis na reestruturação de outros sistemas de software.

Refatoramentos de larga escala diferem de padrões de projeto na sua ênfase. Padrões de projeto apresentam soluções para um problema de projeto, enquanto refatoramentos propõem modificações em um código que não é mais apropriado e também, descreve meios para implementar estas modificações [Ducasse, 1999].

Refatoramentos de larga escala também são diferentes de *Anti-Patterns* [Brown, 1998] pelo enfoque. Os antipadrões focam na prevenção dos erros [Ducasse, 1999] que levam ao surgimento de antipadrões e não no processo de transformação de uma solução adotada em uma outra refatorada e aparentemente mais adequada.

1.3 Estrutura da dissertação

Esta dissertação está dividida em dez capítulos, sendo o primeiro deles a presente introdução. Os capítulos dois e três abordam a teoria envolvida no tema tratado neste trabalho: refatoramentos de software em larga escala. O quarto capítulo apresenta a metodologia abordada e aspectos importantes encontrados durante o desenvolvimento do trabalho. Os refatoramentos identificados estão distribuídos entre os capítulos cinco, seis, sete e oito. O nono capítulo traz uma análise das métricas do sistema antes e depois da aplicação destes

refatoramentos e que deve servir de subsídio para a validação dos resultados alcançados. O décimo capítulo traz a conclusão do trabalho.

2 A EVOLUÇÃO DE SISTEMAS DE SOFTWARE COM REFATORAMENTO

O objetivo deste capítulo é apresentar o conceito de refatoramento de software. Primeiramente é traçado um breve perfil da evolução dos sistemas de software e das atividades de manutenção conduzidas durante esta evolução. O objetivo nesta etapa é mostrar o cenário comum do surgimento de sistemas entrópicos já que eles atestam, com muita fidelidade, a utilidade das técnicas de refatoramento.

2.1 A evolução de sistemas de software

A fase de desenvolvimento de um sistema de software é concluída quando o sistema está operacional, ou seja, sendo utilizado por usuários no ambiente real de produção. Qualquer atividade realizada para modificar o sistema após esta etapa é considerada **manutenção** [Pfleeger, 1998]. A fase de manutenção é muito importante porque lida com as pressões de mudança sofridas pelo software ao longo do tempo [Eick, 2001]. Os requisitos mudam, novas funcionalidades são requeridas, erros surgem e devem ser reparados, o ambiente de hardware e software no qual o produto se insere também muda. Enfim, várias são as razões que levam um produto de software a evoluir e continuar evoluindo após sua concepção.

A observação de sistemas grandes dá a indicação de como um produto de software evolui com o tempo, de como se dá esta fase evolucionária. Lehman (1980) observou o comportamento de sistemas à medida que eles evoluem e resumiu suas observações em cinco leis. A idéia principal desta evolução é que mudanças se tornam cada vez mais difíceis com o tempo, pois a complexidade aumenta e o programa perde sua estrutura, a menos que algo seja feito para reduzir estes efeitos [Eick, 2001]. Este é o retrato do surgimento de sistemas entrópicos.

2.2 As atividades de manutenção

A razão para que mais e mais atenção seja dada a esta etapa do ciclo de vida de sistemas de software é que a maior parte do esforço de um projeto está na fase de manutenção [Roberts, 1999]. Alguns estudos estimam que a relação de esforço no ciclo de vida de um sistema de software obedeça à regra 80-20, onde 80% do esforço total gasto no produto é

utilizado na manutenção e 20% no desenvolvimento [Pfleeger, 1998]. Estes custos são altos devido à perda de estrutura que dificulta as tarefas de manutenção, tornando-as mais lentas e caras. Esta perda de estrutura do software é mais conhecida na Engenharia de Software como Entropia do Software. Sistemas tornam-se entrópicos quando perdem sua estrutura devido às freqüentes mudanças implementadas.

2.3 Refatoramento de software

Refatoramentos são mudanças feitas na estrutura interna do software para torná-lo mais compreensível e baixar o custo de sua modificação, sem que o seu comportamento externo sofra qualquer alteração [Opdyke, 1992]. É uma forma de melhorar o projeto do software depois de ter o produto pronto e funcionando. Esta não é uma técnica recente; ela tem sido usada com outros nomes por programadores de diversas linguagens de programação [Opdyke, 1992]. Bons programadores sempre “limpam seu código” ou o “enxugam”. Refatoramento de software consiste em um conjunto de regras bem planejadas para efetuar, com segurança, mudanças no código com o objetivo único de torná-lo mais compreensível, legível, flexível e, conseqüentemente, fácil de manter. Observe que a expressão *com segurança* foi usada para destacar uma preocupação do refatoramento que é manter o código funcionando. A expressão *objetivo único* foi usada para salientar que, ao refatorar, não se adiciona funcionalidade: as mudanças servem apenas para reorganizar o código.

O uso freqüente da técnica pode diminuir os efeitos indesejados causados pela perda de estrutura do software e permite, ainda, que o projeto seja modificado para melhor acomodar e se adaptar às novas necessidades. As regras de refatoramento podem ser utilizadas em qualquer fase do processo de desenvolvimento, mas principalmente na fase de manutenção [Opdyke, 1992].

A razão principal para o uso da técnica é que raramente os programadores conseguem criar as melhores abstrações da primeira vez [Fowler, 1999]. E sabe-se também, que boas abstrações são a chave principal do reuso. E, como foi mostrado, o reuso está relacionado com a facilidade de manutenção, pois quanto mais extensível for o software, mais fácil será adicionar novas funcionalidades e mais rápida será a manutenção.

2.3.1 Definição

O objetivo do refatoramento é realizar mudanças no código pronto, e funcionando, para torná-lo mais legível e compreensível ou mais extensível, adaptável. Para isto, se apóia em

técnicas bem planejadas com o intuito de diminuir o risco de introdução de erros no sistema. Assim, o refatoramento pode ser entendido como um conjunto de técnicas bem planejadas que tem o objetivo de melhorar a capacidade do sistema de se adaptar a novas necessidades e acomodar novas funcionalidades.

A seguir apresentam-se as definições fornecidas por Martin Fowler em [Fowler, 1999].

“Refatoramento (substantivo): uma mudança feita na estrutura interna do software para torná-lo mais fácil de entender e mais barato de modificar. O comportamento deve ser mantido inalterado”.

“Refatorar (verbo): reestruturar um software através da aplicação de uma série de refatoramentos sem mudar o seu comportamento”.

2.3.2 Requisitos para o refatoramento

Existem duas regras essenciais que devem ser seguidas para que qualquer refatoramento possa ser corretamente aplicado. É fortemente recomendado que nenhuma delas sejam desprezadas:

- não se deve adicionar funcionalidade e realizar refatoramento ao mesmo tempo; são duas etapas completamente distintas;
- é preciso dispor de testes automáticos.

Estas regras são fundamentais para que se obtenham os resultados esperados com a aplicação das técnicas de refatoramento. Refatoramento é aplicado sobre código que funciona. A adição de funcionalidade é uma etapa completamente distinta do refatoramento, que deve ser feita antes ou depois ou, ainda, antes e depois, mas jamais durante [Fowler, 1999]. A razão básica é que não será possível certificar-se que as mudanças efetuadas pelo refatoramento foram eficazes.

Para modificar código que funciona, é preciso ter a possibilidade de verificar se algum problema foi introduzido. Para isto, são necessários alguns testes que realizem esta verificação, pois quando a menor mudança for realizada no software, os testes podem fornecer uma margem de confiabilidade de que erros não foram inseridos no sistema.

2.3.3 Quando refatorar?

Outro aspecto importante da técnica de refatoramento é o momento certo para sua aplicação. No entanto, talvez a pergunta mais adequada fosse: quando um código deve ser modificado para melhorar sua legibilidade, flexibilidade, facilidade de manutenção? [Fowler,

1999] Muito embora sejam várias e provavelmente válidas as respostas a esta pergunta, listam-se a seguir algumas situações que exemplificam claramente o momento certo para o refatoramento do código.

Quando "parece difícil" adicionar uma nova funcionalidade

Sempre que é necessário adicionar uma nova característica a um software e esta adição leva à repetição de código em várias partes do sistema, deve-se refatorar, ou reorganizar, o código existente antes de implementar a nova funcionalidade [Fowler, 1999]. É certo que mais tempo e esforço será preciso para organizar esta parte do sistema que se encontra espalhada, porém a adição desta nova função e de outras, que por ventura venham a ser necessárias, serão mais rápidas e fáceis. Além disto, diminui-se o risco de introdução de erros no sistema já que um único trecho de código desempenhará a mesma atividade e não vários como anteriormente.

Quando não é possível compreender o código

Para realizar alguma modificação ou atualização em um software, é preciso compreendê-lo para que a alteração não quebre a estrutura do código ou introduza comportamentos inesperados. Porém, quando não for possível compreender um trecho de código, ou for difícil identificar sua responsabilidade ou a lógica utilizada para executá-la, esta situação indica claramente que as técnicas de refatoramento podem, e devem, ser utilizadas para melhorar a legibilidade do código e evitar a mesma dificuldade nas próximas alterações.

Após alguma alteração ou produção de um novo trecho de código

Na fase de programação dificilmente o programador está preocupado em quem irá manter o código que ele produz. A sua preocupação maior é ter sua tarefa completada em tempo hábil e de acordo com os cronogramas. Este mesmo programador dificilmente irá pensar na legibilidade daquilo que está produzindo. Somado a isto, tem-se o fato de que é muito raro que um programador acerte a melhor e mais legível forma de resolver determinado problema na primeira tentativa. Por estas razões, é fácil concluir que é de fundamental importância que ao produzir um novo trecho de código ou ao alterar um código existente, o programador seja levado também a realizar a fase de refatoramento do código. É mais fácil enxergar problemas de projeto ou de estrutura depois que o software está pronto e funcionando [Opdyke, 1999].

2.3.4 O catálogo de refatoramentos

Em [Fowler, 1999] os refatoramentos propostos consistem no resultado de anos de experiência do autor trabalhando profissionalmente na área. Eles estão organizados em um catálogo que o autor propõe ser o início de um catálogo que deve ser incrementado e melhorado com as contribuições de outras pessoas que passem a usar a técnica, ou que já a utilizam.

O catálogo está organizado de acordo com o problema que os refatoramentos tentam resolver. As classificações criadas pelo autor são:

- Compondo métodos. Refatoramentos que visam organizar o código adequadamente através da reorganização dos métodos existentes.
- Movendo funcionalidades entre os objetos. As decisões em projeto de software orientado a objetos são, em sua maioria, decisões acerca de atribuição de responsabilidade [Fowler, 1999]. Esta classe é composta por aqueles refatoramentos que reorganizam a atribuição de responsabilidades dentro do sistema.
- Organizando dados. Os refatoramentos nesta classe têm o objetivo de tornar o trabalho com dados mais fácil [Fowler, 1999] através da transformação de estruturas de dados em classes, da substituição de valores por classes ou ainda modificações mais simples como a substituição de números mágicos por constantes.
- Simplificando expressões condicionais. Lógica condicional pode tornar-se complicada, assim os refatoramentos nesta classe têm o objetivo de propor formas para simplificar esta lógica, seja através da simplificação das expressões lógicas, da sua divisão ou através de polimorfismo.
- Tornando as invocações a métodos mais simples. A chave principal destes refatoramentos é tornar as interfaces das classes mais simples e fáceis de entender e usar.
- Tratando generalizações. A maioria dos refatoramentos nesta classe trata com o movimento de métodos em uma hierarquia de herança. Os demais lidam com a adição de classes à hierarquia ou a sua eliminação, quando a herança não for a melhor maneira para tratar uma dada situação.
- Refatoramentos em larga escala. Esta classe apresenta refatoramento que detém um escopo de atuação maior que os refatoramentos das classes anteriores. Eles

lidam com um número mais elevado de classes e são mais difíceis de catalogar e exemplificar por que a situação muda muito com estes refatoramentos. Outro ponto importante de ressaltar é que estes refatoramentos podem levar muito tempo. Os refatoramentos anteriores levam, no máximo, uma ou duas horas para serem executados, alguns dos refatoramentos tratados nesta classe levaram meses ou anos para serem executados [Fowler, 1999].

2.3.5 Formato

Um refatoramento é composto das seguintes partes [Fowler, 1999]:

- um **nome**, esta parte é importante para facilitar a comunicação entre as pessoas que discutam o tema;
- um pequeno **resumo** descrevendo a situação na qual este refatoramento é necessário e o que ele propõe;
- a **motivação** descreve as razões que fazem este refatoramento útil e necessário e as circunstâncias sob as quais a sua aplicação é desencorajada;
- o **mecanismo** descreve os passos que devem ser seguidos para que o refatoramento possa ser aplicado de forma mais segura;
- os **exemplos** apresentam um uso simples deste refatoramento com o objetivo de mostrar como ele funciona.

2.3.6 Exemplo

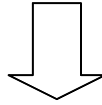
O objetivo aqui é fornecer uma visão prática da técnica apresentada. A seguir, apresenta-se o refatoramento *Replace Array with Object* tal como é apresentado no catálogo produzido por Martin Fowler [Fowler, 1999]. Este é um exemplo simples e não tem a pretensão de ilustrar todas as vantagens que podem ser obtidas com o uso constante da técnica.

Replace Array with Object

Existe um array no qual alguns elementos têm significados diferentes.

Substitua o array por um objeto que tem um atributo para cada elemento.

```
String [] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```



```
Performance row = new Performance();
Row.SetName ("LiverPool");
Row.SetWins ("15");
```

Motivação

Arrays são estruturas comuns de organização de dados. Entretanto, eles devem ser usados apenas para conter uma coleção de objetos similares em alguma ordem. É possível, porém, encontrar arrays utilizados para conter vários tipos de informações diferentes. Convenções do tipo "o primeiro elemento do array é o nome da pessoa" são difíceis de lembrar. Já com o uso de um objeto, pode-se utilizar nomes de atributos e métodos para obter estas informações, assim, não é preciso lembrar de nomes ou esperar que os comentários no código estejam atualizados. É possível também encapsular a informação e usar outros reftoramentos para adicionar comportamento a esta informação.

Mecanismo

Crie uma nova classe para representar a informação contida no array. Adicione o array como um atributo público desta classe.

Mude todos os clientes do array para usar a nova classe.

Compile e teste.

Um por um, adicione métodos de acesso (recuperação e atribuição) para cada elemento do array. Mude os clientes para usar os métodos de acesso. Compile e teste após cada mudança.

Quando todo o acesso ao array estiver substituído por métodos, torne o array privado.

Compile.

Para cada elemento do array, crie um atributo na classe e mude o código cliente para usar o atributo.

Compile e teste após cada elemento ter sido modificado.

Quando todos os elementos forem substituídos por atributos, apague o array.

Exemplo

Este exemplo inicia-se com um array que é usado para armazenar o nome, a quantidade de vitórias e perdas de um time esportivo:

```
        String [] row = new String[3];
```

Este array poderia ser usado com código parecido com o seguinte:

```
        row[0] = "Liverpool";
        row[1] = "15";

        String name = row[0];
        int wins = Integer.parseInt(row[1]);
```

Para transformar este array em um objeto, inicia-se com a criação da classe:

```
        class Performance {}
```

O próximo passo é a criação de um atributo público para a classe (esta não é uma boa escolha, mas será melhorada em seguida).

```
        public String[] _data = new String [3];
```

O passo a seguir sugere o encontro dos pontos de criação e de acesso ao array para substituição do trecho de código. Os trechos nos quais o array é criado devem ser substituídos por:

```
        Performance row = new Performance();
```

Já os trechos que fazem uso do array devem ser substituídos por:

```
        row._data [0] = "Liverpool";
        row._data [1] = "15";

        String name = row._data[0];
        Int wins = Integer.parseint ( row._data[1]);
```

Um por um, métodos de acesso às informações de vem ser adicionados. Por exemplo, o nome:

```
class Performance...

        public String getName() {
                return _data[0];
        }

        public void setName(String arg) {
                _data[0] = arg;
        }
```

Depois, deve-se alterar o código cliente para utilizar os métodos de acesso à informação:

```
        row.setName("Liverpool")
        row._data [1] = "15"

        String name = row.getName();
        int wins = Integer.parseInt(row._data[1]);
```

Com o segundo elemento, deve-se fazer o mesmo. Para tornar a transformação mais simples, pode-se encapsular o tipo de dado:

```
class Performance...

	public int getWins() {
		return Integer.parseInt(_data[1]);
	}

	public void setWins(String arg) {
		_data[1] = arg;
	}
```

....

No código cliente, tem-se:

```
row.setName("Liverpool");
	row.setWins("15");

	String name = row.getName();
	int wins = row.getWins();
```

Com a criação de métodos de acesso a todos os elementos, pode-se transformar o array em um atributo privado.

```
	private String[] _data = new String[3];
```

A parte mais importante deste refatoramento, a mudança de interface, está agora completa. É também útil, entretanto, substituir o array internamente. Pode-se fazer isso através da adição de um atributo para cada elemento do array e mudar os métodos de acesso para usá-los.

```
	class Performance...

	public String getName() {
		return _name;
	}

	public void setName(String arg) {
		_name = arg;
	}
	private String _name;
```

Isto deve ser repetido para cada elemento do array. Quando todas as mudanças estiverem implementadas, o array pode ser apagado.

3 O PROJETO DE REFATORAMENTO

Este trabalho foi desenvolvido com base no projeto de refatoramento de um sistema de software real. O sistema utilizado como estudo de caso deste trabalho é um sistema comercial de recuperação textual que precisa passar por um projeto de revitalização para estender sua vida útil.

3.1 A Metodologia

Para iniciar um projeto de refatoramento é preciso que sejam identificados alguns passos a serem seguidos uma vez que uma metodologia formal para tratamento deste tipo de projeto ainda não é consenso. A Tabela 3.1 apresenta a metodologia criada e seguida para o desenvolvimento do projeto de refatoramento do servidor LightBase.

Etapa	Descrição
Preparação	Identificação dos requisitos de negócio objetivados pela empresa numa tentativa de maximizar as chances de sucesso do projeto.
Estudo inicial do código do sistema	Estudo do sistema para obtenção do conhecimento necessário às tarefas de refatoramento.
Criação de testes	Elaboração de uma estratégia e confecção de uma bateria adequada de testes que suportem as mudanças proporcionadas pelos refatoramentos.
Definição e aplicação dos refatoramentos	Com base no conhecimento adquirido na etapa de Estudo inicial do código do sistema, os problemas encontrados no software devem ser analisados, soluções devem ser propostas e a mais adequada deve ser aplicada ao sistema. Estes passos originam o refatoramento. Esta etapa deve ser aplicada exaustivamente durante o desenvolvimento do

projeto.

Tabela 3.1 - Metodologia criada para o desenvolvimento do projeto de refatoramento deste trabalho.

3.2 O estudo de caso

O estudo de caso consiste de um sistema de recuperação textual que foi desenvolvido ao longo de três anos e está no mercado há mais de seis anos. Esta ferramenta está disponível em versão cliente-servidor e o projeto de refatoramento considera o módulo servidor. É um sistema orientado a objetos e composto por, aproximadamente, 300.000 linhas de código C++ e chama-se LightBase Server [LightBase].

A seguir apresentam-se os resultados obtidos com o projeto de refatoramento do servidor LightBase em cada uma das etapas da metodologia descrita anteriormente.

3.2.1 Preparação

Um projeto de refatoramento em larga escala é feito, primordialmente, para obter algum benefício de negócio e não por pureza técnica. Deste modo, é necessário definir claramente os requisitos de negócio da empresa para maximizar as chances de sucesso do projeto [Aguiar, 2001].

O principal objetivo de negócio da empresa com o projeto de refatoramento utilizado como estudo de caso deste trabalho é o transporte do sistema para uma plataforma de software diferente daquela para a qual o sistema foi inicialmente projetado (do Windows para Unix). Assim, antes de iniciar o projeto, os requisitos e restrições da empresa foram identificados e priorizados numa tentativa de maximizar as chances de sucesso do projeto de refatoramento, as tabelas a seguir apresentam este levantamento.

Os requisitos de negócio identificados para este projeto estão listados na Tabela 3.2, ordenados de acordo com sua prioridade.

Requisito	Classificação	Descrição
Realizar o porte do servidor para Linux, visando a possibilidade de portes para outras plataformas UNIX.	Obrigatório	
Os clientes binários atuais devem continuar funcionando.	Obrigatório	

Adaptar o LightBase Server para as funcionalidades ODBC.	Desejável	
Tratar UNICODE tanto nos índices como nos dados.	Desejável	Unicode é um padrão mundial para codificação de caracteres que usa 16 bits para a representação de um caractere.
Aumentar o espaço de endereçamento das bases.	Desejável	Atualmente o tamanho máximo de uma base que o LightBase pode suportar é 2 Gigabytes, deseja-se aumentar este número.

Tabela 3.2 - Lista dos requisitos do projeto de refatoramento do serivdor LightBase.

Na Tabela 3.3 estão listados alguns pontos importantes que devem ser considerados quando do desenvolvimento deste projeto de refatoramento.

Restrição	Descrição
Transporte do módulo CGI e do LBCOM.	O transporte do servidor para uma plataforma UNIX deve considerar um porte futuro de clientes. Neste projeto deve ser considerada a possibilidade de que futuramente o cliente também estará em plataforma Unix.
Mudança ou melhora do motor de indexação.	O projeto deve considerar o fato de que o motor de indexação do produto poderá ser modificado ou melhorado. Isto pode influenciar o desenvolvimento do projeto de refatoramento nos módulos que tratam de indexação.

Tabela 3.3 - Lista das restrições do projeto de refatoramento do estudo de caso.

Não é aconselhável fazer adaptações funcionais ou melhoramentos no sistema durante o refatoramento [Sneed, 1995]. No caso de existirem melhoramentos e novas funcionalidades a serem introduzidas, elas devem ser realizadas em um projeto após o projeto de refatoramento.

No estudo de caso deste trabalho, os requisitos presentes na Tabela 3.2 como adaptação do servidor para as funcionalidades ODBC ou o aumento do espaço de endereçamento das

bases, não serão contemplados por consistirem de adição de funcionalidades ou melhoramento do sistema.

O objetivo de negócio do estudo de caso é a migração do sistema para outro sistema operacional, porém o projeto de refatoramento, que representa uma reestruturação técnica deve ser realizado para permitir que o objetivo principal seja atendido. Isto significa que alguns refatoramentos devem ser criados e implementados com o intuito de possibilitar a migração entre as plataformas de software.

Poucas metodologias de migração de sistemas estão disponíveis e uma abordagem genérica ainda não é consenso. As abordagens existentes são de muito alto nível ou ainda não foram aplicadas na prática [Bisbal, 1999]. Em [Bisbal, 1999] afirma-se que os poucos projetos do tipo migração publicados na literatura descrevem soluções *ad hoc*.

Sobre metodologias de reestruturação de sistemas, encontra-se um cenário parecido com o descrito anteriormente. A metodologia de reestruturação de sistemas mais amplamente pesquisada e conhecida, "*cold turkey*", propõe a substituição do sistema antigo por um novo com as mesmas funcionalidades [Stevens, 1998], consiste, efetivamente, no desenvolvimento do sistema vislumbrando, desde o início, todos os requisitos satisfeitos no sistema. No entanto, esta abordagem promove um aumento considerável no risco da reestruturação por propor mudanças radicais em um único passo, especialmente quando se trata de sistemas grandes [Stevens, 1998].

Deste modo, a abordagem utilizada no projeto de migração e refatoramento neste trabalho segue uma metodologia *ad hoc* e direcionada à resolução do problema enfrentado.

3.2.2 Estudo inicial do código do sistema

Antes que o código de um software possa ser refatorado é preciso adquirir um bom conhecimento do sistema. Esta etapa da metodologia objetiva fornecer um maior entendimento do produto. A principal dificuldade enfrentada nesta etapa foi a falta de documentação. O sistema dispunha de uma documentação desatualizada e incompleta. A documentação existente versava essencialmente sobre a organização, em disco, dos objetos de persistência do sistema e seu mapeamento em objetos do negócio. Deste modo, a documentação existente do sistema revelou-se incompleta e insuficiente para as necessidades do projeto.

Esta realidade foi amenizada pela experiência obtida pela autora durante um ano em que realizou as atividades de manutenção do sistema antes do projeto de refatoramento. Porém, o conhecimento acumulado não foi suficiente para compreender o sistema por inteiro. Deste

modo, atividades de engenharia reversa foram desempenhadas para possibilitar o progresso do projeto. Estas atividades englobaram a criação de diagramas de classe de alguns módulos do projeto como forma de melhorar a compreensão dos mesmos. Isto foi feito com o auxílio da ferramenta Rational Rose [Rational].

A primeira atividade realizada com intuito de aumentar a compreensão do sistema foi identificar os módulos do sistema e sua funcionalidade. O resultado pode ser encontrado no Apêndice A, porém a Tabela 3.4 apresenta um resumo das informações obtidas. Em conjunto com esta identificação, foi criado o gráfico de dependências do sistema (que também pode ser encontrado no Apêndice A). Nele, todos os módulos e a interação entre eles foram apresentados. O exame do código bem como as entrevistas com analistas, programadores e integrantes mais experientes da equipe de manutenção do sistema foram decisivas para o sucesso deste levantamento.

Informação	Valor
Número de módulos	19
Número máximo de classes em um módulo	395
Número mínimo de classes em um módulo	1

Tabela 3.4 - Resumo das informações obtidas na atividade de estudo inicial do código do sistema.

A atividade seguinte restringiu-se aos módulos identificados. A engenharia reversa dos módulos principais foi executada à medida que foi realizado o estudo individual de cada um. Esta atividade resultou na criação do diagrama de classes de alguns módulos e na identificação das responsabilidades de cada classe contida neles.

Não foi feita a engenharia reversa completa do sistema porque isto poderia ser responsável por um esforço grande que talvez não proporcionasse os ganhos almejados. O objetivo da realização da engenharia reversa foi obter um entendimento mínimo que possibilitasse a identificação dos problemas principais do software para que o projeto de refatoramento pudesse prosseguir.

3.2.3 Criação de testes

Segundo a definição de refatoramento, as técnicas para implementá-lo devem ser descritas de forma a minimizar as chances de introdução de erros no sistema. Além disso, um dos pré-requisitos para a aplicação de tais técnicas é uma forma de garantir, com uma margem de confiabilidade, que o sistema continua funcionando como antes das mudanças. Esta forma

é a presença de uma boa bateria de testes que cubra, especialmente, os trechos de código afetados. Sendo assim, uma boa bateria de testes do sistema que será refatorado é um requisito para o projeto de refatoramento.

O servidor LightBase dispunha de poucos testes automáticos antes do início do projeto de refatoramento. A cada nova versão, o software passava por uma etapa de testes *ad-hoc* realizada por pessoas que não tinham experiência em tal função, uma forma ineficaz de verificar a presença de erros e problemas no sistema.

A estratégia de testes utilizada pela empresa revelou-se inadequada e pouco representativa, assim, o projeto de refatoramento não deveria confiar nesta estratégia visto que projetos deste tipo apresentam alto risco de inserir novos erros no sistema. Este foi outro obstáculo encontrado durante o planejamento do projeto de refatoramento. O planejamento também de uma estratégia de testes e de construção dos testes automáticos tornou-se necessário antes que o projeto pudesse prosseguir.

O primeiro objetivo com a criação dos testes é elaborar uma estratégia de testes para minimizar os riscos de introdução de problemas com as modificações feitas pelo refatoramento. Ou seja, certificar-se de que, após a execução do projeto e de acordo com os testes, o sistema continua funcionando como antes, já que não são previstas correções de erros. De acordo com o objetivo, testes de regressão são adequados às necessidades. Teste de regressão é o teste aplicado a novas versões para verificar se o sistema processa as mesmas funções da mesma maneira que as versões anteriores [Pfleeger, 1998].

A estratégia escolhida para testar o sistema foi utilizar testes funcionais e testes de sistema. Testes funcionais examinam a funcionalidade total do produto. Testes de sistema envolvem o exame de todo o sistema computacional. Os testes de unidade foram descartados, pois as mudanças propostas pelos refatoramentos, provavelmente, mudariam a estrutura de classes do sistema.

Os testes de sistema construídos para este projeto incluem testes de estresse e de performance numa tentativa de validar o desempenho e o comportamento do sistema em situações de atividade intensa.

A implementação e execução dos testes contaram com o apoio de uma terceira pessoa especialmente destinada a este fim. A Tabela 3.5 apresenta um sumário da estratégia de testes adotada para o projeto de refatoramento.

Objetivo	Diminuir o risco de introdução de erros no sistema.
Estratégia	Teste de regressão.
Tipos de teste	Testes funcionais e de sistema.

Tabela 3.5 - Sumário da estratégia de testes adotada.

Os testes funcionais têm o objetivo de validar o funcionamento do sistema através do exercício de suas funcionalidades. A construção dos testes funcionais para a validação do sistema foi realizada com o apoio de uma ferramenta de apoio a testes chamada VBUnit [VBUnit]. Esta ferramenta teve origem em outra ferramenta construída para testes de unidade de classes Java chamada JavaUnit [Gamma, 1999], [Beck, 1998]. Posteriormente o seu projeto foi reutilizado para a construção de ferramentas de testes para outras linguagens de programação.

O VBUnit facilita a construção de testes de unidade para componentes e/ou aplicativos desenvolvidos no Visual Basic. O LightBase dispõe de um componente de software chamado LbCom que funciona como cliente do servidor LightBase. O VBUnit foi utilizado juntamente com o LbCom para a construção dos testes funcionais automáticos para o sistema.

Uma ferramenta de testes desenvolvida em VisualBasic e também utilizando o componente LbCom foi construída para os testes de sistema. O objetivo principal desta ferramenta é estressar o servidor com atualização de informações nas bases de dados para verificar a consistência da informação após as atualizações concorrentes promovidas pela ferramenta.

Outra ferramenta também construída com o objetivo de apoiar a validação das modificações realizadas no servidor foi chamada de LogGraf. Esta ferramenta gera gráficos a partir do log de atividades do servidor LightBase. Através dos gráficos gerados por LogGraf é possível acompanhar a atividade de cada uma das linhas de execução ativas no servidor no momento da geração do log. Ssaber quais delas executavam, no momento de uma possível falha e quais métodos cada uma delas invocou durante sua execução, bem como, no momento da falha. Esta ferramenta facilita a análise da informação contida no log principalmente em casos de problemas gerados por acesso a informações compartilhadas entre linhas de execução que são difíceis de identificar.

O componente LbCom possibilita acesso à criação e modificação de bases e à manipulação das informações nelas contidas. Porém, o servidor LightBase tem uma lista de funcionalidades mais ampla e que não é coberta pelo componente como, por exemplo, a administração destas bases e a gerência de usuários. Para estes casos, testes de funcionalidade

especiais foram criados. A criação destes testes contou com o apoio da ferramenta CppUnit [CppUnit], que é a correspondente ao VBUnit e JavaUnit, para a linguagem C++. Estes testes utilizam diretamente a interface do servidor para a execução destes serviços.

Apesar da dificuldade envolvida na atribuição de um valor para a cobertura obtida pela bateria de testes criada, estima-se que este conjunto cubra menos de 50% do total das funcionalidades fornecidas pelo sistema. Uma cobertura mais ampla das funcionalidades do sistema inviabilizaria a execução do projeto devido às restrições de tempo.

A definição das funcionalidades a serem testadas foi baseada na interface do sistema que o componente LBCOM permitia acesso e na identificação das funcionalidades que estariam mais susceptíveis a mudanças dada a amplitude de sua complexidade dentro do sistema. Sendo assim, a análise de riscos foi a principal técnica utilizada para definir as funcionalidades que seriam contempladas na criação dos testes.

3.2.4 Definição e aplicação dos refatoramentos

Esta atividade consiste na definição e aplicação de refatoramentos identificados com base nas necessidades do software. Esta etapa é desenvolvida com base nas informações obtidas na etapa de *Estudo inicial do código do sistema*. Com um problema estrutural identificado e sua solução proposta, passa-se à implementação desta solução com o cuidado de não alterar o comportamento externo do software. A implementação de cada solução é acompanhada pela avaliação de suas conseqüências e benefícios.

A partir das mudanças efetuadas durante esta fase do projeto, os refatoramentos são definidos e a análise de seus efeitos é efetuada. O próximo capítulo traz uma discussão acerca de refatoramento de software em larga escala e apresenta o formato seguido para sua construção.

4 REFATORAMENTOS DE SOFTWARE EM LARGA ESCALA

O objetivo deste capítulo é descrever os refatoramentos de software em larga escala. Esta descrição engloba a sua definição, o padrão criado para sua apresentação e também a classificação utilizada para sua apresentação.

4.1 Definição

Refatoramentos em larga escala são técnicas de reestruturação de sistemas de grande porte que tenham sua estrutura comprometida. Utilizam o formato definido em [Fowler, 1999] para apresentar a técnica de reorganização de sistemas.

Algumas alterações no formato do refatoramento foram necessárias para acomodar o novo enfoque dado às transformações que é o aumento da abrangência da aplicação das mesmas. Os refatoramentos de larga escala, possivelmente, envolvem um número de classes mais elevado do que os definidos por Fowler.

Para manter a coerência com a definição de refatoramentos, o refatoramento em larga escala obedece às mesmas regras apresentadas anteriormente para a realização de refatoramentos menores: não adicionar funcionalidade e realizar refatoramento ao mesmo tempo; e, dispor de testes automáticos. Deste modo, o risco de introdução de erros no sistema é minimizado, já que existe uma maneira de verificar se o sistema funciona da mesma forma que antes, com uma margem de confiabilidade que depende, primordialmente, dos testes utilizados nesta verificação.

4.2 Motivação

Os refatoramentos propostos em [Fowler, 1999] resolvem problemas de entendimento. No entanto o escopo de sua atuação pode ser insuficiente para resultar em melhorias expressivas em casos de sistemas grandes e com problemas estruturais. A necessidade de refatoramentos de grande porte é sentida quando é preciso realizar mudanças na estrutura de um sistema e estas envolvem várias classes ou quando as mudanças devem resolver problemas que exigem uma visão do sistemática do software como a interação entre os componentes do sistema e sua interface.

4.3 Modelo de apresentação

Para a apresentação dos refatoramentos em larga escala, utiliza-se o formato definido a seguir. A razão que motiva a definição de um formato é facilitar a comunicação e a produção de outros refatoramentos além dos propostos aqui. Este formato é uma adaptação do padrão de refatoramento proposto por Fowler [Fowler, 1999] de modo a contemplar as necessiades de um refatoramento de larga escala.

4.3.1 Nome

Para facilitar a comunicação entre os envolvidos no projeto de refatoramento de um software, cada refatoramento definido será nomeado.

4.3.2 Fundamentos

Alguns refatoramentos envolvem conhecimentos que não estão limitados aos problemas de projeto ou à solução proposta para solucioná-los. É possível que a solução a um problema de projeto seja a aplicação de um padrão de projeto, por exemplo. Isto exige a compreensão do padrão de projeto bem como dos malefícios e benefícios adquiridos com sua aplicação. Assim, este item do formato de refatoramento tem por objetivo prover informações necessárias à compreensão do problema e/ou solução abordados. Deste modo, não é fundamental ao entendimento do padrão desde que o leitor detenha os conhecimentos adequados.

É possível que este item seja omitido na apresentação de alguns refatoramentos caso os temas abordados por ele já tenham sido apresentados ou o refatoramento não exija conhecimentos específicos.

4.3.3 Sumário

Neste item apresenta-se um resumo dos problemas e soluções abordados pelo refatoramento. O objetivo desta seção é fornecer um guia de referência para facilitar a compreensão do refatoramento sem que seja necessária a leitura do texto integral do refatoramento.

4.3.4 Condições de aplicação

Nesta seção são listados os pré-requisitos para a aplicação do refatoramento. Os problemas que identificam a possível necessidade de aplicação deste refatoramento são apresentados aqui.

4.3.5 Mecanismo

Este item fornece a seqüência de passos que devem ser realizados para resolver o problema identificado e aplicar a solução proposta. Esta seqüência objetiva minimizar os riscos de introdução de erros no sistema através da execução de cautelosas etapas de mudanças no sistema.

4.3.6 Conseqüências

A análise dos benefícios e malefícios obtidos com a adoção da solução proposta é apresentada nesta seção. Os problemas foram apresentados anteriormente, uma solução foi proposta e nesta seção são demonstradas as razões que tornam a solução proposta adequada à resolução do problema identificado. Porém, esta análise não apresenta apenas os benefícios obtidos, ou como os problemas são minimizados, resolvidos. É importante listar também os aspectos negativos da solução para que os efeitos do refatoramento sejam conhecidos previamente e, desta forma, uma decisão mais consciente quanto à sua aplicação ou não possa ser tomada.

4.3.7 Exemplo

Nesta seção apresenta-se um sistema com o problema descrito pelo refatoramento bem como a aplicação do mecanismo proposto para aplicar a solução proposta. O objetivo é exemplificar a aplicação do refatoramento seguindo os passos propostos.

4.4 Classificação

Com o objetivo de tornar mais fácil a compreensão dos refatoramentos propostos, eles são divididos em categorias. Embora existam várias formas de categorizá-los, duas são destacadas aqui. A primeira forma de classificação divide os refatoramentos em dois grandes grupos: o grupo daqueles que mantêm seu escopo de atuação dentro de um único módulo e o grupo daqueles que têm um escopo de abrangência maior, englobando não apenas um

módulo, mas, principalmente, a inter-relação entre módulos do sistema. Ou seja, é possível dividir os refatoramentos em **intermódulos** e **intramódulos**.

A outra maneira de classificação destacada aqui consiste na identificação da natureza do problema que o refatoramento se propõe a resolver ou na solução proposta pelo refatoramento. Neste caso, encontra-se um número maior de grupos e possibilidades, são eles:

- refatoramentos que resolvem problemas de acoplamento;
- refatoramentos que resolvem problemas de coesão;
- refatoramentos para introdução de abstração;
- refatoramentos para introdução de padrões de projeto.

A Tabela 4.1 mostra como se distribuem os refatoramentos entre as classes definidas anteriormente.

		Escopo	
		Intermódulo	**Intramódulo**
Propósito	**Problemas de acoplamento**	• Diminuição de acoplamento através de tipos abstratos de dados. • Eliminação de dependências cíclicas.	• Isolamento de dependência de plataforma.
	Problemas de coesão	• Junção de módulos com funcionalidade incompleta. • Divisão de grandes módulos.	• Divisão de classe de fronteira.
	Introdução de abstração		• Modelagem de conceitos da solução através de classe. • Separação de persistência.
	Introdução de padrões de projeto	• Introdução de *Singleton* de vários contextos.	• Introdução de *Observer*. • Introdução de *Singleton*.

Tabela 4.1 - Classificação dos refatoramentos propostos.

A primeira classificação reflete o escopo do refatoramento; já a segunda reflete o propósito do refatoramento, ou seja, o tipo de problema que ele resolve.

É fácil perceber que a divisão dos refatoramentos de acordo com o problema ou com a solução proposta representa um elemento mais eficiente na sua classificação uma vez que promove uma distribuição mais uniforme entre os grupos. Deste modo, a segunda classificação será a utilizada como ferramenta de categorização dos refatoramentos propostos neste trabalho.

4.5 Apresentação dos refatoramentos

Os refatoramentos aqui propostos são apresentados de acordo com a sua classificação e não representam a ordem de aplicação no estudo de caso. A aplicação destes refatoramentos é cíclica, ou seja, um conjunto de refatoramentos pode ser aplicado várias vezes seguidas, ou alternadas. A aplicação de um refatoramento se dá com base no problema encontrado e não obedece uma ordem de aplicação.

5 REFATORAMENTOS QUE DIMINUEM ACOPLAMENTO

Este capítulo apresenta os refatoramentos em larga escala que tentam diminuir o acoplamento existente entre os componentes do sistema.

Acoplamento é a medida de quão fortemente uma classe está conectada a, tem conhecimento de ou depende de outras classes [Larman, 1997]. Classes com altos índices de acoplamento são indesejáveis por várias razões, entre elas [Larman, 1997]:

- mudanças nas classes relacionadas podem implicar em mudanças locais à classe;
- é mais difícil entender a classe isoladamente;
- é mais difícil reutilizar a classe porque implica no reuso de todas as outras classes das quais ela depende.

Um exemplo de forte acoplamento é a herança de implementação, uma subclasse é muito dependente de sua superclasse.

A diminuição do acoplamento deve ser considerada e almejada durante toda a fase de projeto do sistema. As atividades de manutenção também devem ser norteadas pela conservação do projeto e pela não inclusão de acoplamento desnecessário no sistema. Porém, muitas vezes as alterações requeridas impõem a acomodação de funcionalidades que não foram previstas e o sistema evolui de maneira que a estrutura inicial não permite. Algumas destas mudanças podem provocar a introdução de acoplamento desnecessário, bem como outras características que degradem a estrutura geral do sistema.

Os refatoramentos que endereçam o acoplamento como problema têm o objetivo de sugerir alternativas às conexões introduzidas e apresentar estruturas mais flexíveis de cooperação entre as classes e/ou módulos de um sistema.

5.1 Eliminação de dependências cíclicas

5.1.1 Fundamentos

Grafo de dependências

O grafo de dependências de um sistema representa o sistema em termos de módulos e da cooperação entre eles. Cada módulo consiste em um conjunto de classes que respondem por um subconjunto das responsabilidades gerais do sistema, ou por uma atividade necessária para que outras atividades possam ser realizadas.

Idealmente, um grafo de dependências se parece com uma árvore, quando então pode ser chamado de árvore de dependências. A idéia de árvore indica que os módulos presentes em um nível da hierarquia conhecem apenas aqueles módulos existentes no nível imediatamente inferior. Ou seja, um nível requer serviços do nível inferior e fornece serviços ao nível imediatamente superior. Com esta organização, tem-se um grafo como mostrado na Figura 5.1.

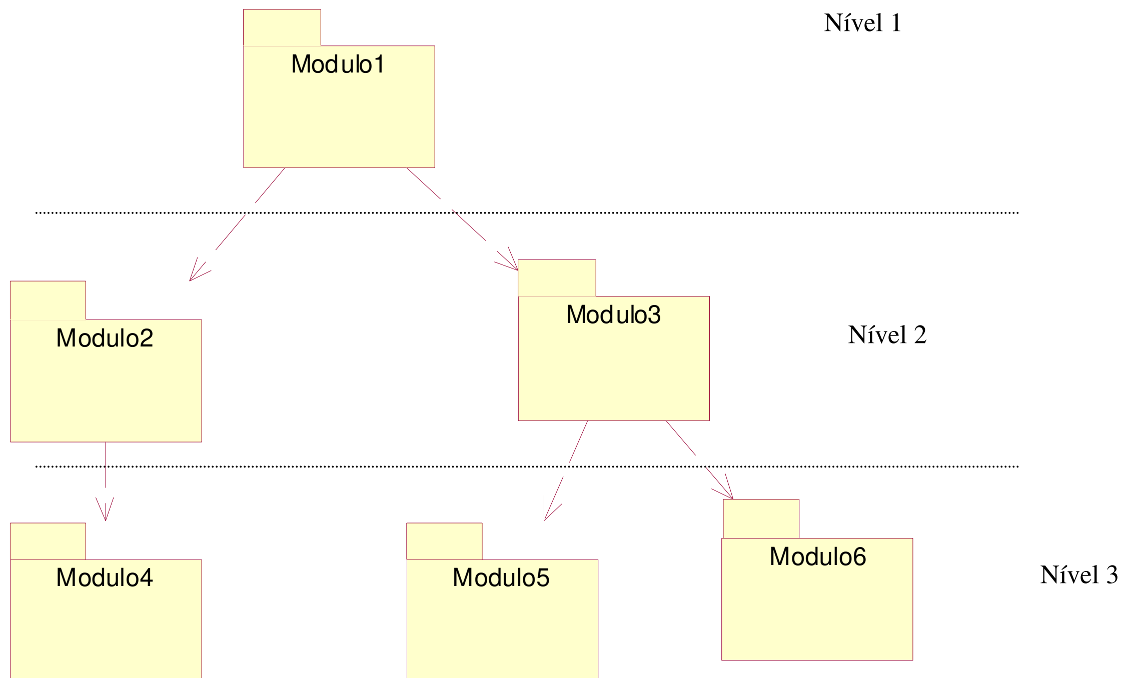


Figura 5.1 - Ilustração de uma árvore de dependências.

Os grafos de dependências normalmente não obedecem a este formato. Porém, ao lidar com dependências entre módulos, este provavelmente será o objetivo: transformar o grafo de dependências do sistema o mais próximo possível de uma árvore de dependências.

5.1.2 Sumário

Em alguns sistemas é possível encontrar dependências cíclicas envolvendo os módulos que o compõem. Este refatoramento fornece meios para a eliminação destas dependências como forma de aumentar a possibilidade de reuso dos componentes.

A Figura 5.2 apresenta alguns exmplos de dependências cíclicas que podem ser encontradas em sistemas de software.

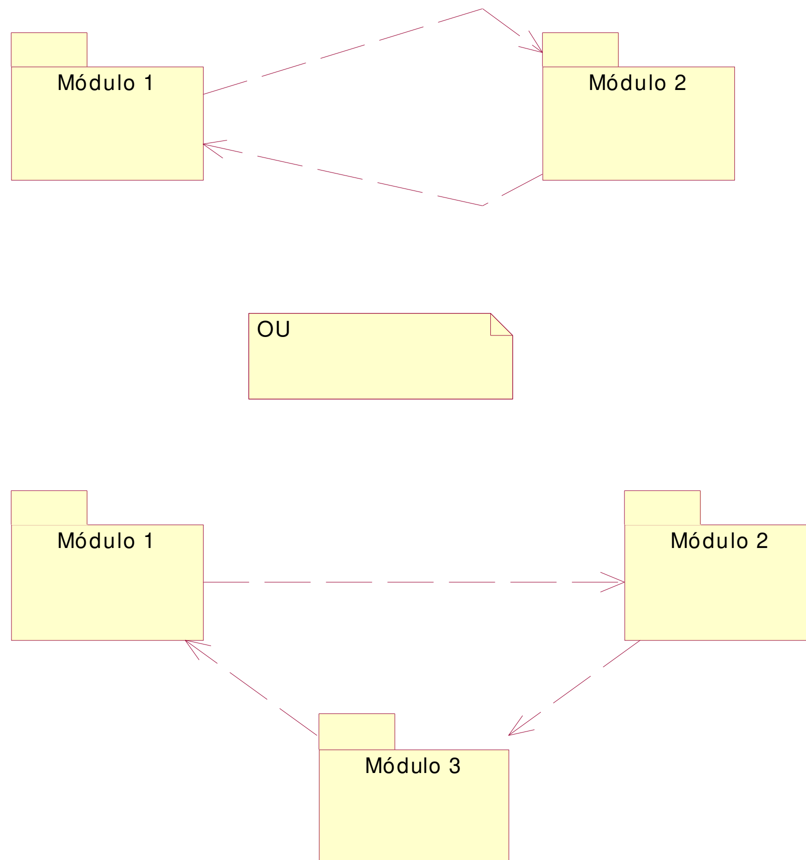


Figura 5.2 – Exemplos de dependências cíclicas.

5.1.3 Condições de aplicação

A condição de aplicação é clara: a existência de dependências cíclicas entre os módulos do sistema. É possível que haja situação em que este tipo de dependência é necessário; porém na maioria dos casos ele pode e deve ser evitado. A razão pela qual é desejável evitar este tipo de dependência é a diminuição da possibilidade de reuso dos módulos envolvidos na relação. Uma vez que, para utilizar as funcionalidades providas por um módulo, é preciso utilizar os demais.

Outra forte razão para que este tipo de dependência seja evitado é que o forte acoplamento entre os módulos dificulta também a manutenção, já que mudanças realizadas em um dos módulos podem afetar os demais.

Em projetos que objetivam transportar o sistema de software para um ambiente diferente daquele para o qual foi inicialmente projetado, a existência de dependências cíclicas

entre módulos do sistema pode constituir um obstáculo por obrigar que este transporte seja realizado para todos os envolvidos na relação ao mesmo tempo. Ou seja, os módulos envolvidos na dependência cíclica devem ser transportados todos ao mesmo tempo, pois não há como testar um dos módulos sem que os outros estejam funcionando. Esta situação favorece o aparecimento de uma inconsistência: a validação de um módulo que usa serviços de outro que não foi validado porque usa serviços deste que está sendo validado.

5.1.4 Mecanismo

1 Criar o grafo de dependência do sistema.
2 Identificar as dependências cíclicas.
3 Verificar, para cada uma delas a real necessidade de existência da dependência cíclica.
4 Caso não seja necessária a dependência cíclica, é preciso desfazê-la.
    4.1 Em alguns casos, é necessária apenas a recolocação de classes, ou seja, transferir uma classe (ou classes) de um módulo para outro.
    4.2 É possível existirem casos em que seja necessária a quebra do módulo em outros menores, cada um responsável por uma determinada funcionalidade. Neste caso, o refatoramento *Divisão de grandes módulos* deve ser aplicado.
5 Atualizar os demais módulos que fazem referências aos envolvidos na etapa anterior para que reflitam a nova realidade.
6 Após a eliminação das dependências, os módulos envolvidos no processo devem ser compilados e testados.
7 Compilar e testar todo o sistema.

5.1.5 Conseqüências

Como conseqüência da aplicação deste refatoramento pode-se citar o aumento na possibilidade de reuso dos módulos que antes estavam envolvidos na dependência cíclica. É possível que após a execução do mecanismo proposto, estes módulos estejam mais coesos, e, conseqüentemente, mais fáceis de reutilizar.

Além do citado anteriormente, há um grande ganho também na facilidade de manutenção dos módulos já que modificações em um deles afetam um menor número de dependentes.

No entanto, existe o risco da multiplicação do número de módulos do sistema, o que pode ser também prejudicial à compreensão do mesmo. É preciso que os módulos criados

sejam coesos e sejam responsáveis por funcionalidades que realmente justifiquem a sua existência para que o sistema não seja povoado por módulos pequenos com muitas dependências e que não reflitam uma funcionalidade completa.

Além das observações anteriores, é importante lembrar que, através da execução deste refatoramento, o transporte dos módulos do sistema é facilitado uma vez que não há mais a inconsistência gerada pela existência de dependência cíclica no sistema.

5.1.6 Exemplo

O exemplo de aplicação deste refatoramento começa com a análise geral da integração dos módulos que compõem o estudo de caso como um todo. A Figura 5.3 apresenta o grafo de dependências do sistema completo.

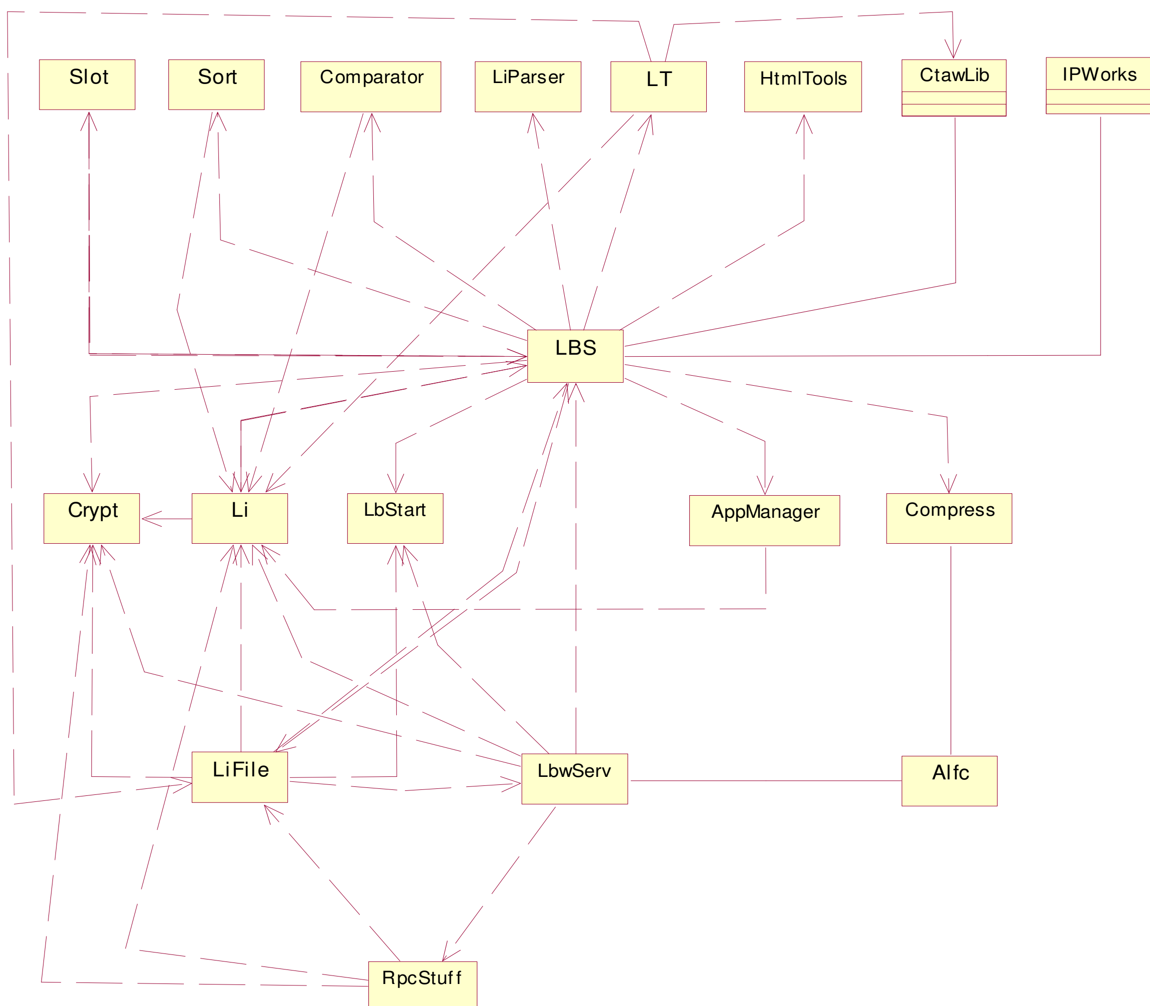


Figura 5.3 - Grafo de dependências entre os módulos do exemplo.

Passos 1 e 2

A primeira etapa deste refatoramento é a construção do grafo de dependências. A Figura 5.3 apresenta o grafo de dependências do estudo de caso.

Através do grafo, é fácil identificar as dependências cíclicas presentes no sistema. As relações entre os módulos *LiFile*, *LbwServ*, *RpcStuff* consistem em um exemplo de dependência cíclica, outro exemplo é composto pelas relações entre *LiFile* e *Lbs*. Esta é a grande vantagem da criação do grafo de dependências: a facilidade na identificação das dependências do sistema.

Passos 3 e 4

De acordo com o grafo da Figura 5.3, existem no servidor LightBase várias dependências cíclicas que devem ser eliminadas devido às razões citadas anteriormente.

Após o exame de cada uma destas dependências, percebeu-se que algumas eram impostas por códigos não utilizados, ou seja, "código morto" e por esta razão foram eliminadas. Outras, porém, exigiram uma simples recolocação de classes, o que significa mover as classes entre os módulos. Por exemplo, o LBS depende de LI e vice-versa (veja Figura 5.4). Ao investigar a origem desta dependência, descobre-se que o LI utiliza uma única classe do LBS chamada `LBSC_List`. Esta classe, apresentada a seguir, depende apenas da classe `LBSC_Node`, que não apresenta referência a outras classes.

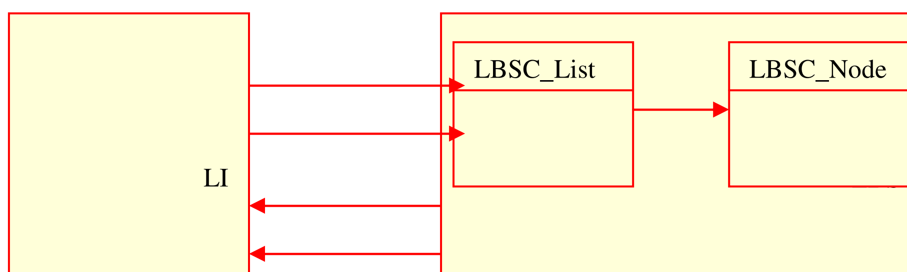


Figura 5.4 - Origem da dependência entre os módulos *lbs* e *li*.

Ao investigar a classe `LBSC_List` nota-se que ela representa uma lista de objetos `LBSC_Node`. Porém, não é privilégio do módulo ***lbs*** utilizar listas de objetos, qualquer outro objeto do sistema pode usar esta estrutura de dados. A função do ***lbs*** no sistema é implementar a lógica do servidor LightBase apenas. A definição de estruturas de dados não deveria fazer parte de suas atribuições. O módulo ***li*** é um módulo que já contém outras definições de tipos básicos de dados, deste modo, a transferência das classes para o módulo ***li*** poderia proporcionar um aumento na coesão do módulo ***lbs*** sem prejuízos à coesão do módulo ***li***.

É preciso verificar também se algum outro módulo referencia as classes que se tornaram candidatas à transferência, uma vez que isto pode provocar a introdução de novas relações entre módulos do sistema. Esta verificação pode ser realizada automaticamente pela ferramenta de desenvolvimento utilizada, no desenvolvimento deste projeto esta atividade foi realizada com o auxílio do Microsoft Visual C++ 6.0 [VC++6.0]. No exemplo aqui tratado, não existem outros módulos, exceto o próprio LBS, que utilizem as classes `LBSC_List` e `LBSC_Node`. Assim, a solução para a dependência cíclica é simplesmente a transferência das classes do ***lbs*** para ***li***, vide Figura 5.5.

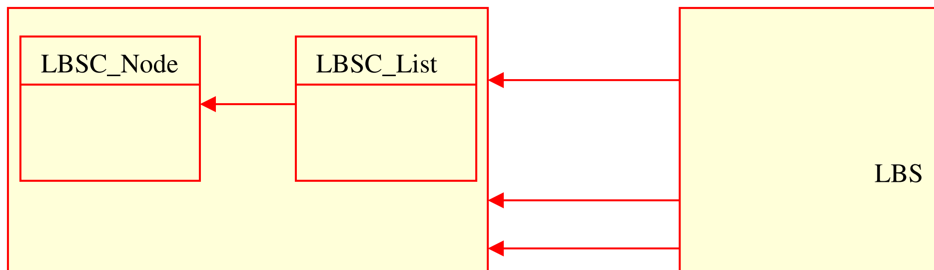


Figura 5.5 - Dependências entre os módulos *lbs* e *li* após a recolocação das classes LBSC_List e LBSC_Node.

```
class LBSC_Node {
	private:
		friend		class	LBSC_List;
		void			*pvData;
		LBSC_Node		*pcnNextNode;
		LBSC_Node		*pcnPreviousNode;

	public:
				LBSC_Node( void *pvData );
				~LBSC_Node( void );
		void		AddNext( LBSC_Node *pcnNextNode );
		void		AddPrevious( LBSC_Node *pcnPreviousNode );
};

LBSC_Node::LBSC_Node( void *pvData )
{
	pcnNextNode = NULL;
	pcnPreviousNode = NULL;
	this->pvData = pvData;
}

LBSC_Node::~LBSC_Node( void )
{
	if( pvData ){
		delete pvData;
	}
	if( pcnPreviousNode != NULL ){
		pcnPreviousNode->pcnNextNode = pcnNextNode;
	}
	if( pcnNextNode != NULL ){
		pcnNextNode->pcnPreviousNode = pcnPreviousNode;
	}
}
```

```
void
LBSC_Node::AddNext( LBSC_Node *pcnNewNode )
{
	pcnNewNode->pcnNextNode = this->pcnNextNode;
	if( this->pcnNextNode != NULL ){
		this->pcnNextNode->pcnPreviousNode = pcnNewNode;
	}
	this->pcnNextNode = pcnNewNode;
	this->pcnNextNode->pcnPreviousNode = this;
}

void
LBSC_Node::AddPrevious( LBSC_Node *pcnNewNode )
{
	pcnNewNode->pcnPreviousNode = this->pcnPreviousNode;
	if( this->pcnPreviousNode != NULL ){
		this->pcnPreviousNode->pcnNextNode = pcnNewNode;
	}
	this->pcnPreviousNode = pcnNewNode;
	this->pcnPreviousNode->pcnNextNode = this;
}

class LBSC_List
{
	protected:
		LBSC_Node	*pclCurrent;
		LBSC_Node	*pclFirst;
		LBSC_Node	*pclLast;
		int		iNumElem;
		int		iCurrElem;

	public:
				LBSC_List();
				~LBSC_List();
		virtual			int Compare( void *, void *, int ) = 0;
		virtual			void *Duplicate( void * ) = 0;
		int		Add( void *, int );
		void		*Update( void * );
		int		Del();
		void		*Next();
		void		*Previous();
		void		*First();
		void		*Last();
		void		*Current();
		void		*Nth( int );
		int		DelAll();
		int		NumElem();
		void		*operator [ ]( int );
		int		GetCurrentIndex();
		void		*Search( void *, int, int = 0 );

		void		SetCurrObj( void * );
};

LBSC_List::LBSC_List()
{
	pclCurrent = NULL;
	pclFirst = NULL;
	pclLast = NULL;
	iNumElem = 0;
	iCurrElem = -1;
}

LBSC_List::~LBSC_List( void )
{
	DelAll();
}
```

```
int
LBSC_List::Add( void *pvNewData, int iInsertType )
{
	LBSC_Node	*pcnTemp;

	if( pvNewData == NULL ){
		return( LBSE_BADARG );
	}
	if( (pcnTemp = new LBSC_Node( pvNewData )) == NULL ){
		return( LBSE_NOMEMORY );
	}
	if( pclCurrent == NULL ){
		pclCurrent = pclFirst = pclLast = pcnTemp;
		iCurrElem = 0;
		iNumElem = 1;
		return( OK );
	}
	switch( iInsertType ){
	case TAIL:
			Last();
	case AFTER:
			pclCurrent->AddNext( pcnTemp );
			if( pclCurrent == pclLast ){
				pclLast = pcnTemp;
			}
			iCurrElem++;
			break;
	case HEAD:
			First();
	case BEFORE:
			pclCurrent->AddPrevious( pcnTemp );
			if( pclCurrent == pclFirst ){
				pclFirst = pcnTemp;
			}
			break;
	case ASC_ORDER:
	case DESC_ORDER:
			First();
			while( (iInsertType == DESC_ORDER ?
			   Compare( pclCurrent->pvData, pcnTemp->pvData, 0 ) > 0 :
Compare( pclCurrent->pvData, pcnTemp->pvData, 0) < 0) ){
				if( pclCurrent == pclLast ){
					pclCurrent->AddNext( pcnTemp );
					pclLast = pcnTemp;
					break;
				} else {
					Next();
				}

			}
			if( pclLast != pcnTemp ){
				pclCurrent->AddPrevious( pcnTemp );
				if( pclCurrent == pclFirst ){
					pclFirst = pcnTemp;
				}
			}
			break;
	default:
			delete pcnTemp;
			return( LBSE_BADARG );
	}
	pclCurrent = pcnTemp;
	++iNumElem;
	return( OK );
}

int
LBSC_List::Del()
```

```
{
        LBSC_Node     *pcnTemp;

        pcnTemp = pclCurrent;
        if( pclCurrent == NULL ){
                return( LBSE_EMPTYLIST );
        }
        if( pclCurrent == pclFirst ){
                pclFirst = pclCurrent->pcnNextNode;
        }
        if( pclCurrent == pclLast ){
                pclLast = pclCurrent->pcnPreviousNode;
                --iCurrElem;
        }
        if( pclCurrent->pcnNextNode != NULL ){
                pclCurrent = pclCurrent->pcnNextNode;
        } else {
                pclCurrent = pclCurrent->pcnPreviousNode;
        }
        delete pcnTemp;
        --iNumElem;
        return( OK );
}

int
LBSC_List::DelAll()
{
        while( Del() != LBSE_EMPTYLIST ){
                ;
        }
        return( OK );
}

void *
LBSC_List::Next()
{
        if( !pclCurrent ){
                return( NULL );
        }
        if( pclCurrent->pcnNextNode == NULL ){
                return( NULL );
        }
        pclCurrent = pclCurrent->pcnNextNode;
        ++iCurrElem;
        return( pclCurrent->pvData );
}

void *
LBSC_List::Previous()
{
        if( !pclCurrent ){
                return( NULL );
        }
        if( pclCurrent->pcnPreviousNode == NULL ){
                return( NULL );
        }
        pclCurrent = pclCurrent->pcnPreviousNode;
        --iCurrElem;
        return( pclCurrent->pvData );
}

void *
LBSC_List::First()
{
        if( pclFirst != NULL ){
                pclCurrent = pclFirst;
                iCurrElem = 0;
                return( pclCurrent->pvData );
```

```
        }
        return( NULL );
}

void *
LBSC_List::Last()
{
        if( pclLast != NULL ){
                pclCurrent = pclLast;
                iCurrElem = iNumElem - 1;
                return( pclCurrent->pvData );
        }
        return( NULL );
}

void *
LBSC_List::Current()
{
        if( pclCurrent != NULL ){
                return( pclCurrent->pvData );
        }
        return( NULL );
}

void *
LBSC_List::Nth( int iIndex )
{
        void    *pvResult;

        if( pclCurrent == NULL ){
                return( NULL );
        }
        if( iIndex < 0 || iIndex >= iNumElem ){
                return( NULL );
        }
        if( iIndex < iCurrElem ){
                pvResult = First();
        } else {
                if( iIndex == iNumElem - 1 ){
                        return( Last() );
                } else {
                        pvResult = Current();
                }
        }
        while( iCurrElem < iIndex && (pvResult = Next()) != NULL );
        return( pvResult );
}

void *
LBSC_List::Search( void *pvSearchData, int iSearchType, int iSearchArg )
{
        void            *pvAux;
        LBSC_Node               *pcnTemp;
        int             iResultCompare;

        pcnTemp = pclCurrent;
        if( pcnTemp == NULL ){
                return( NULL );
        }
        pvAux = pclCurrent->pvData;

        while( (iSearchType & SEARCH_LESS) && (iSearchType & SEARCH_GREAT) &&
(iResultCompare = Compare( pvAux, pvSearchData, iSearchArg )) == 0 &&(pvAux =
Next()) != NULL ){
                        ;
        }
        if((iSearchType&SEARCH_LESS)&&(iSearchType & SEARCH_GREAT)){
                if( pvAux == NULL ){
```

```
                    pclCurrent = pcnTemp;
            }
            return( pvAux );
    }

    while( (iResultCompare = Compare( pvAux, pvSearchData, iSearchArg )) < 0 &&
(pvAux = Next()) != NULL ){
            ;
    }
    if( (iSearchType & SEARCH_EQUAL) && iResultCompare == 0 ){
            return( pvAux );
    }
    if( iSearchType & SEARCH_LESS ){
            if( pvAux == NULL ){
                    return( pclCurrent->pvData );
            } else {                                // Element found
                    if( pclCurrent->pcnPreviousNode == NULL ){
                            pclCurrent = pcnTemp;
                            return( NULL );
                    } else {
                            return( Previous() );
                    }
            }
    }
    if( iSearchType & SEARCH_GREAT ){
            if( pvAux == NULL ){
                    pclCurrent = pcnTemp;
                    return( NULL );
            } else if( iResultCompare > 0 ){
                    return( pclCurrent->pvData );
            } else {
                    while( (iResultCompare = Compare( pvAux, pvSearchData,
iSearchArg )) == 0 && (pvAux = Next()) != NULL ){
                            ;
                    }
                    if( pvAux == NULL ){
                            pclCurrent = pcnTemp;
                    }
                    return( pvAux );
            }
    }
    pclCurrent = pcnTemp;
    return( NULL );
}

void*
LBSC_List::Update( void * pvNewData )
{
    if( pclCurrent == NULL ){
            return( NULL );
    }
    void* pvReturn = pclCurrent->pvData;

    pclCurrent->pvData = pvNewData;

    return pvReturn;
}

int
LBSC_List::NumElem()
{
    return( iNumElem );
}

void *
LBSC_List::operator [ ] ( int iPos )
{
    return Nth( iPos );
```

```
}

int
LBSC_List::GetCurrentIndex()
{
	return( iCurrElem );
}

void
LBSC_List::SetCurrObj( void *pvNewVal )
{
	if( pclCurrent ){
		pclCurrent->pvData = pvNewVal;
	}
}
```

Com isso, elimina-se a dependência de ***li*** com o ***lbs*** e, conseqüentemente, a dependência recíproca entre os dois. A decisão tomada para transferir as classes entre os módulos foi simplificada pelo fato de já existir a dependência do ***lbs*** em relação a ***li***. Caso não existisse esta dependência, a transferência das classes implicaria na adição de mais uma relação entre módulos do sistema. Neste caso, poderia ser válida a investigação de outras alternativas para a solução ao problema.

Como foi visto anteriormente, não existem outros módulos que façam referências às classes transferidas exceto ***lbs*** e ***li***, assim, para concluir a execução do refatoramento, deve-se compilar, gerar e testar o sistema. Os testes executados aqui devem ser aqueles construídos com o objetivo de validar o projeto de refatoramento, neste estudo de caso, os testes funcionais e de sistema construídos para este fim foram executados e analisados. Caso algum outro módulo fizesse referência às classes afetadas pelas modificações, a atualização destas referências deveria ser realizada para permitir a geração destes módulos.

5.2 Diminuição de acoplamento através de tipos de dados

5.2.1 Sumário

Este refatoramento propõe a diminuição do acoplamento entre módulos do sistema através do uso de tipos abstratos de dados (TADs). A Figura 5.6 apresenta um resumo deste refatoramento.

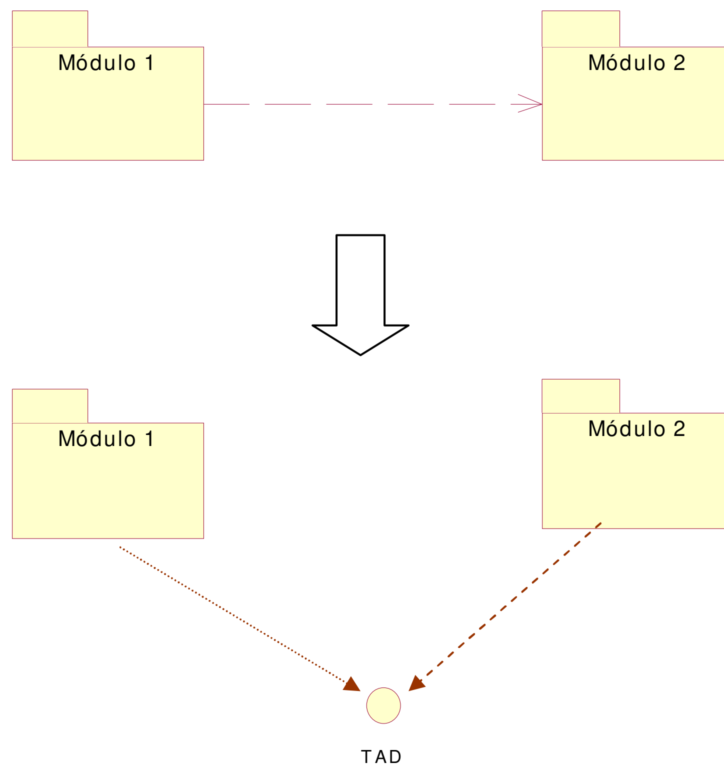


Figura 5.6 - Refatoramento *Diminuição de acoplamento através de tipos de dados*.

5.2.2 Condições de aplicação

Um sistema grande geralmente é dividido em módulos, ou seja, conjuntos de classes. Esses conjuntos desempenham alguma atividade dentro do sistema. Por exemplo, em um sistema bancário há, provavelmente, um módulo responsável pela criptografia dos dados, outro pela comunicação através da rede, entre outros.

É comum que alguns destes módulos cooperem entre si para desempenhar suas atividades. Ou seja, é comum que um módulo tenha visibilidade de outro. Isto pode ser feito através de herança, agregação, ou simples associações entre as classes contidas nos módulos. Estas situações podem provocar o forte acoplamento entre os dois módulos, tornando difícil que um deles seja modificado sem afetar o outro.

Em sistemas de software que comumente sofrem pressões de mudanças esta não é uma situação desejável, pois manter o sistema pode levar mais tempo do que deveria. Esta situação consiste de um cenário candidato à aplicação do refatoramento *Diminuição de acoplamento através de tipos de dados*.

É difícil especificar de forma precisa como identificar a necessidade de criação de um TAD, no entanto, uma boa indicação de onde um tipo abstrato de dado deve ser utilizado é o uso de polimorfismo. Ou seja, em trechos onde o polimorfismo é utilizado ou pretende ser urilizado, deve-se criar um TAD.

5.2.3 Mecanismo

1 Identificar os módulos do sistema.

  1.1 O ideal é que este passo seja automatizado. É indicado o uso de ferramentas que auxiliem a atividade de engenharia reversa do código para identificar os módulos do sistema.

2 Construir grafo de dependências entre os módulos.

3 Eliminar as dependências que, semanticamente, não fazem sentido.

  3.1 Algumas dependências entre os módulos são, semanticamente, equivocadas. Por exemplo, em um sistema bancário, não faz sentido que o módulo responsável pela criptografia da informação dependa do módulo responsável pela apresentação do sistema.

4 Para cada dependência verificar a real necessidade de existência e eliminar aquelas que podem ser eliminadas.

  4.1 Algumas dependências podem ter sido geradas desnecessariamente devido à inexperiência dos mantenedores do sistema, ao desconhecimento dos conceitos de orientação a objetos ou mesmo devido à realização de atividades de manutenção de maneira despreocupada com a evolução geral da arquitetura do sistema.

5 Para as dependências que restarem, identificar as classes, de cada módulo, que fazem parte da relação.

6 Criar um tipo abstrato de dado representando a funcionalidade que causa a dependência entre os módulos.

  6.1 Esta é uma etapa muito importante do refatoramento, uma vez que este TAD deve refletir aquilo que o módulo precisa tornar externo, ou seja, de conhecimento de seus clientes para que este possa fornecer-lhes seus serviços.

7 Fazer o módulo dependente referenciar apenas o TAD criado, e não mais a classe, como antes.

8 Compilar e testar o sistema.

5.2.4 Consequências

Com a aplicação deste refatoramento, diminui-se o acoplamento entre os módulos, uma vez que eles não estão mais conectados através de uma implementação (classe), mas sim através de um comportamento (TAD). Deste modo, qualquer classe que implemente aquele comportamento pode substituir a original.

Esta nova situação aumenta a possibilidade de reuso dos módulos. Além disso, facilita a manutenção já que modificações em um módulo não precisam atingir o outro, pois um módulo está conectado ao outro apenas através do TAD que provavelmente permanecerá inalterado.

Por outro lado, este refatoramento introduz o risco da multiplicação do número de TADs contidos no sistema, o que pode também dificultar o entendimento. Por esta razão é preciso definir com cuidado a interface externa do módulo, ou seja, aquilo que deve ser exposto por dele para que não sejam criados tipos desnecessários ou que diminuam o nível de encapsulamento do módulo. E além deste cuidado, a dica apresentada nas *Condições de aplicação* sobre como identificar a necessidade da criação de um TAD pode ajudar a evitar este efeito indesejado.

5.2.5 Exemplo

O exemplo de aplicação deste refatoramento considera um sistema completo e foi estudo de caso abordado neste trabalho.

Passos 1 e 2

O primeiro passo do refatoramento é identificar os módulos do sistema, onde “módulo” significa um agrupamento de classes responsável por uma ou mais funcionalidades. Os módulos do sistema foram identificados através de entrevista com pessoas que fizeram parte do desenvolvimento do software e que participam das atividades de manutenção do mesmo. A impossibilidade de realização destas entrevistas pode implicar na necessidade de investigação do código fonte para extrair as informaçõs necessárias à construção do grafo. As ligações entre estes módulos foram identificadas através da análise do código fonte. A partir destas informações, o grafo de dependências apresentado na Figura 5.7 foi criado.

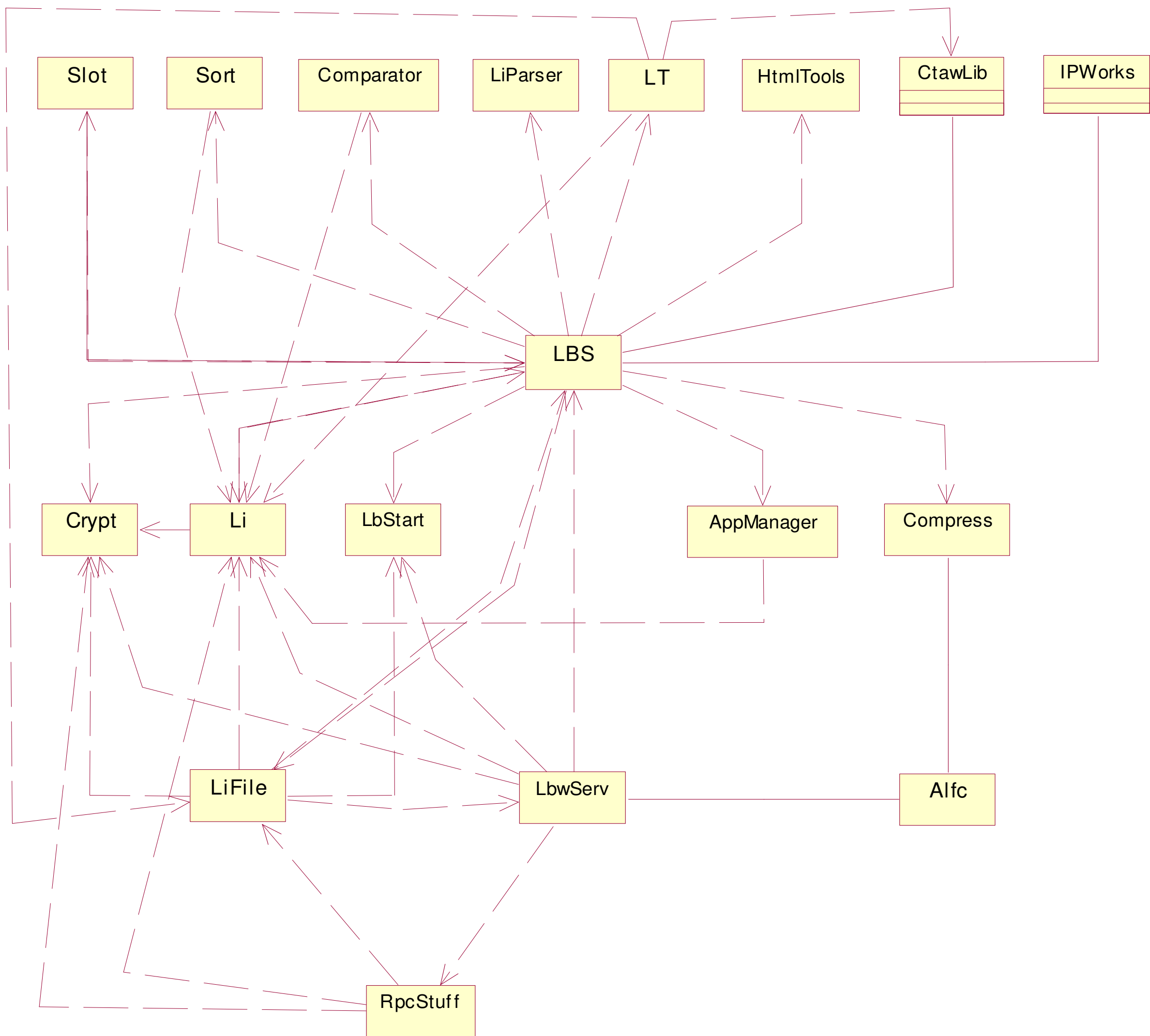


Figura 5.7 - Grafo de dependências entre os módulos do servidor LightBase.

Passos 3 e 4

Esta etapa do refatoramento objetiva melhorar o projeto do sistema. A eliminação de dependências que não fazem sentido ajuda a manter a integridade do sistema com os conceitos de modelagem OO e com o mundo real. Quanto mais próximo do mundo real estiver o sistema, mais fácil será entendê-lo.

Através da análise da funcionalidade dos módulos e do grafo de dependências da Figura 5.7 não é clara a razão para existir a dependência entre ***lbs*** e ***lbstart***. O ***lbs*** é o módulo principal, responsável pelas funcionalidades do sistema. Já o módulo ***lbstart*** é responsável pela autenticação da cópia servidor, ou seja, quando o servidor inicia, a chave de ativação e o número de série da cópia do produto são verificados e validados pelo módulo.

O exame da ligação entre os dois módulos revela que o ***lbs*** apenas utiliza o ***lbstart*** no momento de inicialização do servidor. Sendo o ***lbs*** o módulo responsável pela inteligência da aplicação, não faz sentido que ele mantenha uma ligação com um módulo responsável por validações e verificações da licença da cópia do produto. Assim, esta ligação deve ser eliminada e a responsabilidade por esta verificação deve ser transferida para o módulo responsável pela inicialização do sistema: o ***lbwserv***. Este é o responsável pelas inicializações necessárias para que o servidor execute adequadamente.

A relação entre os módulos ***lbs*** e ***lbstart*** é apenas um exemplo de relações que, semanticamente, não fazem sentido; no entanto, outras relações deste tipo podem ser encontradas na Figura 5.7.

A análise de cada conexão para verificar se o código que impõe aquela dependência é realmente utilizado ou se a relação é necessária ou pode ser substituída é muito importante na simplificação do código do sistema. Através desta atividade, "código morto" é eliminado do sistema. Este tipo de código não é utilizado, mas é deixado no sistema por esquecimento ou por outras razões, podendo ser responsável pelo aumento da complexidade do sistema, dificuldade na identificação de problemas e nas atividades de depuração.

Passos 5, 6 e 7

Para cada uma das dependências que restarem, identificar as classes de cada módulo que fazem parte da relação e criar TADs para diminuir o acoplamento entre os módulos.

No grafo da Figura 5.7, restam várias ligações entre os módulos, duas delas são analisadas a seguir.

Algumas classes neste exemplo representam abstrações de estruturas persistentes do sistema. Algumas destas estruturas detêm informações especiais que são armazenadas em estruturas chamadas *slots*. O módulo ***slot*** é responsável pelo armazenamento e pela recuperação destas informações especiais. O módulo ***lbs*** contém algumas classes que representam abstrações de estruturas persistentes e que têm informações especiais armazenadas em slots. Esta é a razão da dependência existente entre os módulos ***lbs*** e ***slot***.

No entanto, ainda resta uma dependência que parte de ***slot*** para o módulo ***lbs***. Isto ocorre porque, de acordo com a atual implementação, a classe `slot` precisa obter informações da classe persistente, por exemplo, tamanho do slot. Isto obriga que uma nova classe de tratamento de slot seja criada para cada classe persistente que necessite de informações armazenadas em slots. Estas dependências constituem um exemplo de dependência cíclica que não foi eliminada através da aplicação do refatoramento *Eliminação de dependências*

cíclicas. A razão para isto é que a dependência é realmente necessária e aquele refatoramento não propõe alternativas à modelagem apresentada. A forma de resolver esta dependência é a aplicação do refatoramento *Diminuição de acoplamento através de tipos de dados*.

O diagrama de classes da Figura 5.8 apresenta a relação existente entre as classes do módulo ***slot*** e do ***lbs***.

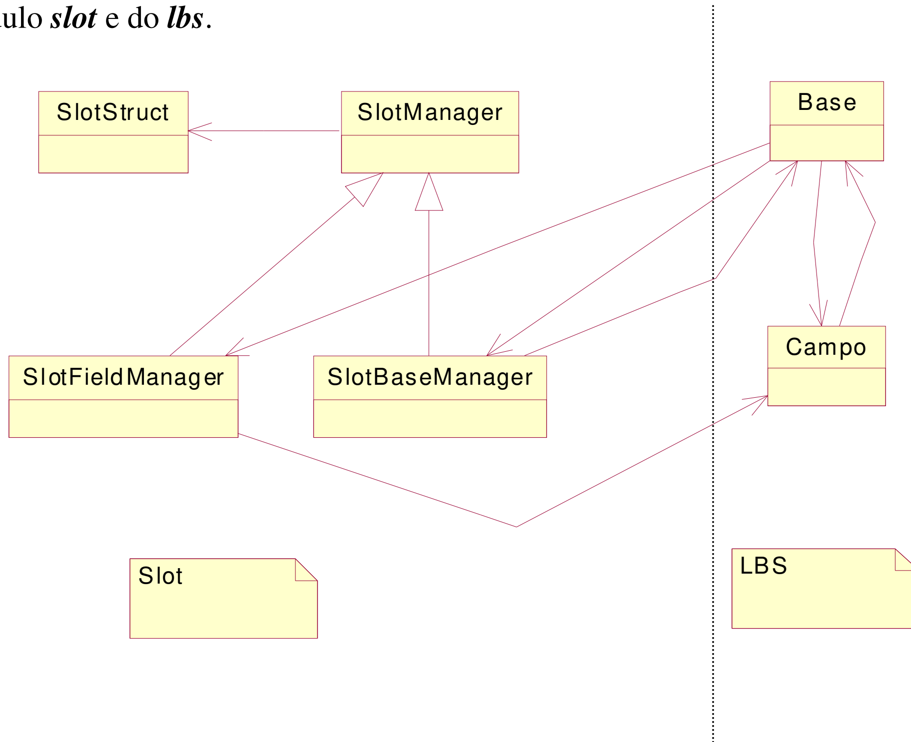


Figura 5.8 - Detalhamento das conexões entre os módulos *slot* e *lbs*.

Esta solução apresenta vários problemas como a interdependência entre classes que dificulta o reuso e teste, bem como a interdependência entre os módulos do sistema. Além disso, pode-se verificar que a classe `Base` possivelmente apresenta autos índices de acoplamento e baixos índices de coesão. No entanto, a relação entre os dois módulos é a principal parte a ser considerada.

O refatoramento *Diminuição de acoplamento através de tipos de dados* propõe a criação de TADs para diminuir o acoplamento entre os módulos do sistema. Para o caso apresentado na Figura 5.8, o tipo `SlotObject` foi criado para representar a abstração de uma estrutura genérica que contem informações especiais armazenadas em slots. Abstrações que necessitem fazer uso desta facilidade devem prover algumas informações básicas que definem, assim, sua interface.

A Figura 5.9 apresenta o novo diagrama de classes após a introdução do TAD `SlotObject`.

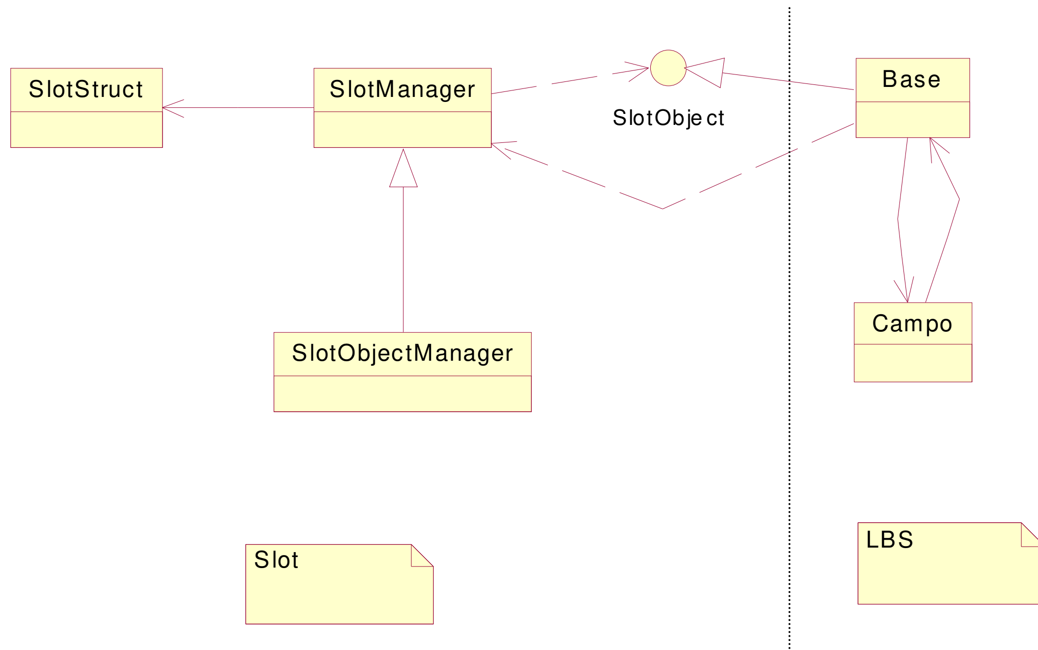


Figura 5.9 - Relação entre os módulos *slot* e *lbs* após a aplicação do refatoramento.

Deste modo, o módulo ***slot*** está acoplado não mais à classe `Base` ou `Campo` ou a qualquer outra implementação, mas sim, a qualquer classe que implemente a interface `SlotObject`. Já o módulo ***lbs*** está acoplado a qualquer classe que implemente a interface de tratamento de slots `SlotManager` e à interface `SlotObject` através de herança.

Para concluir, deve-se compilar, gerar e executar os testes funcionais e de sistema construídos com o intuito de verificar a corretude das modificações efetuadas.

5.3 Isolamento de dependência de plataforma

5.3.1 Introdução

A portabilidade é considerada uma característica desejável em muitos tipos de produtos de software, pois pode aumentar o valor do produto através da extensão de seu ciclo de vida útil e também da expansão do número de instalações possíveis.

Estudos recentes tentam adicionar aos processos de desenvolvimento de software a preocupação com esta característica que pode ser pensada como uma forma de reuso. Pesquisas nesta área caracterizam o reuso mais amplamente como sendo qualquer caso no qual um artefato associado com um sistema de software pode ser usado em mais de uma situação [Mooney, 1997]. Com isso, é possível entender portabilidade como o reuso completo do produto em várias plataformas operacionais.

Este refatoramento concentra-se em uma abordagem para a realização do transporte de um produto completo, não vislumbrando a introdução de novas técnicas para aumentar a portabilidade em processos de software. Antes que o refatoramento seja apresentado, apresentam-se alguns conceitos importantes no domínio do problema.

5.3.2 Fundamentos

Portabilidade[1]

O conceito mais importante a ser apresentado é o de *portabilidade*. "Transportar, ou portar, é o ato de produzir uma versão executável de uma unidade de software ou sistema em um novo ambiente, baseado em uma versão existente".[Mooney, 1997]. Assim, portabilidade é a capacidade que o sistema tem de ser transportado para outra plataforma. Por exemplo, caso o compilador da linguagem seja substituído por outro, as funções do sistema não deveriam ser afetadas [Pfleeger, 1998].

Ambiente/Plataforma

Refere-se ao conjunto completo de elementos que interagem com o software. Isto pode incluir o processador, o sistema operacional, compilador, os dispositivos de entrada/saída, bibliotecas, rede, entre outros.

Portabilidade Binária/Código

Os tipos de portabilidade que podem ser considerados para software são: binária e de código. A primeira consiste no porte do código executável e a segunda no porte da representação da linguagem fonte, ou seja, dos arquivos-fonte. Está claro que a portabilidade binária tem vantagens em relação à segunda, porém isto só é possível em ambientes muito similares. Portabilidade de código assume a disponibilidade do código-fonte; no entanto permite adaptar a unidade de software a um maior número de ambientes.

Este refatoramento concentra o interesse no transporte do código-fonte de um sistema de software completo.

Recursos do sistema operacional

O que está sendo chamado de recurso do sistema operacional engloba os serviços que podem ser obtidos do sistema operacional como linhas de execução (*threads*), tratamento de

---

[1] O termo amplamente utilizado em inglês é *port* ou *porting*.

região crítica, arquivos do tipo ini. Tais serviços são tipicamente obtidos através de chamadas ao sistema (*system calls*).

Recursos de bibliotecas externas

Também é comum que os sistemas de software utilizem bibliotecas distribuídas junto com o compilador ou com o ambiente de desenvolvimento. Esta também é uma fonte de problemas quando se deseja transportar um software, pois é possível que as bibliotecas não estejam disponíveis nas duas plataformas. Assim, *recursos de bibliotecas externas* devem ser entendidos como funções, definições, tipos, entre outros, que são definidos em bibliotecas e que não fazem parte do padrão da linguagem.

5.3.3 Problemas

Os principais problemas inerentes ao porte de um software podem ser originados das incompatibilidades entre os compiladores, das diferenças de hardware e sistemas operacionais. Os problemas endereçados aqui se concentram essencialmente nos problemas gerados pelas incompatibilidades de compiladores e diferenças de sistemas operacionais. Os problemas gerados pela diferença entre hardware (por exemplo, ordem de bytes na memória) não são tratados.

- Possíveis problemas gerados pela incompatibilidade do compilador:
    - ❖ as opções de compilação podem ser diferentes de um compilador para outro, sobretudo em sistemas operacionais diferentes e pouco semelhantes;
    - ❖ sintaxe, pois um compilador pode oferecer funcionalidades extras, que não pertençam ao padrão da linguagem podendo ser rejeitadas pelo compilador destino;
- Problemas gerados pelas diferenças entre sistemas operacionais.

Este é, provavelmente, a maior fonte de problemas quando se trata de transporte de software, pois em sistemas operacionais diferentes, as *APIs* (*Application Programming Interface*) disponíveis para acessá-los podem também divergir em alguns aspectos, como:

- ❖ pequenas diferenças em recursos padrão;
- ❖ recursos semelhantes, mas diferentes nos dois sistemas;

Por exemplo, linhas de execução (*threads*) estão presentes nos dois sistemas, no entanto, a API disponível para acessar este recurso no Windows é diferente do Unix.

- ❖ recursos presentes em um dos sistemas e ausentes no outro.

No *Windows* existe um recurso chamado arquivo de inicialização; há um formato especial para armazenamento de parâmetros de inicialização das aplicações. Para facilitar o uso, o *Windows* dispõe de uma API especializada na recuperação de parâmetros presentes neste tipo de arquivo. Porém este é um conceito que não existe no Linux.

5.3.4 Sumário

Este refatoramento mostra-se necessário quando os projetos de software apresentam diversos módulos utilizando recursos e serviços do sistema operacional. Esta situação pode ser indesejável quando o software precisa ser transportado para outro ambiente hospedeiro. ***Isolamento de dependência de plataforma*** sugere uma abordagem de ataque ao problema de como isolar as diferenças entre os dois ambientes. O principal objetivo deste refatoramento é criar uma camada de portabilidade que esconda os serviços específicos da plataforma e que forneça ao software uma interface abstrata de acesso ao ambiente.

A Figura 5.10 e a Figura 5.11 apresentam o resumo das modificações propostas pelo refatoramento.

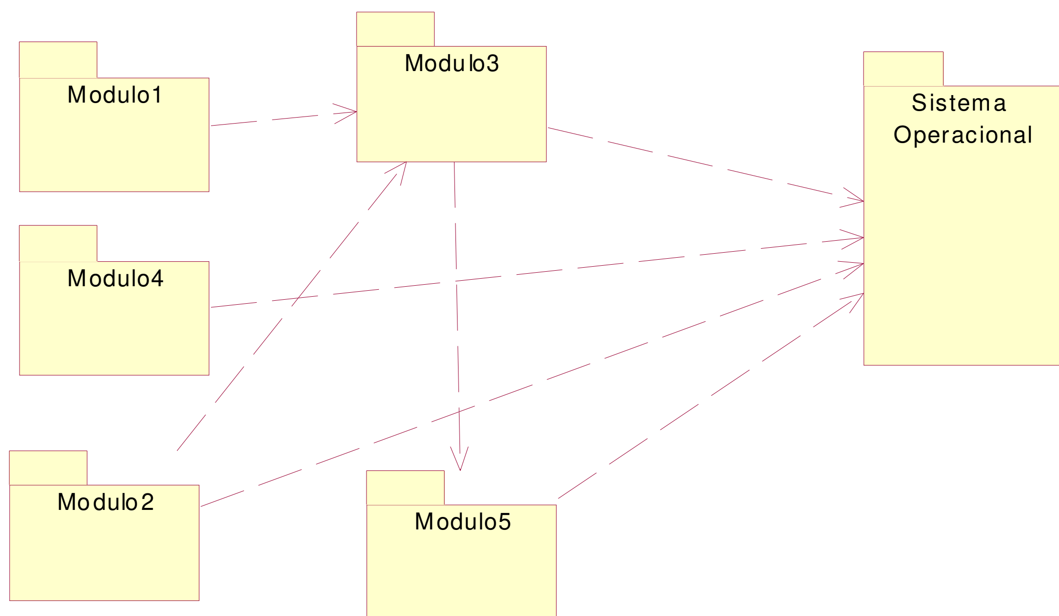


Figura 5.10 - Situação do sistema antes do refatoramento *Isolamento de dependência de plataforma.*

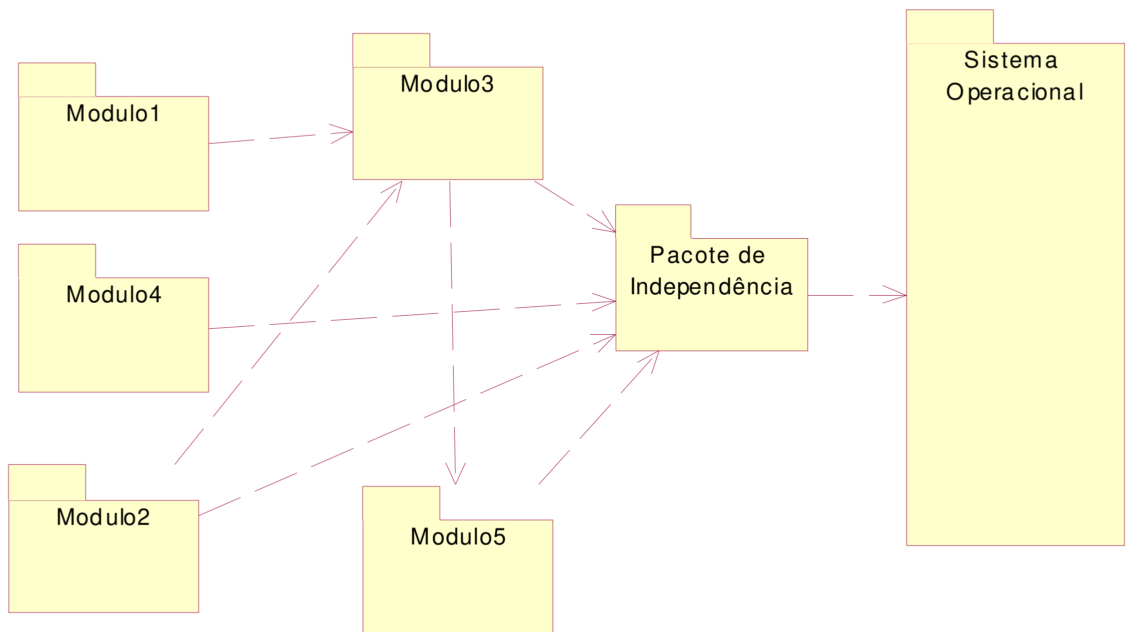


Figura 5.11 – Sistema após o refatoramento Isolamento de dependência de plataforma.

5.3.5 Condições de aplicação

A abordagem proposta deve ser utilizada quando se deseja isolar os trechos do código que dependem da plataforma na qual o sistema está inserido. Não é raro encontrar projetos de software que direcionam o seu desenvolvimento para um ambiente de hardware e software específico, o que leva a um forte acoplamento entre o ambiente inicialmente definido e o produto de software final. No entanto, também não são raras as situações nas quais ocorrem necessidades de mudanças no ambiente hospedeiro do sistema.

Quando se deseja transportar um produto de software de um ambiente para outro, é necessário desacoplá-lo do ambiente hospedeiro achando e destruindo, quando possível, a ligação entre os dois. Em pacotes de software construídos sem a previsão de uma possível migração para outra plataforma, é comum encontrar por todo o código fonte, invocações ao sistema operacional, utilização de bibliotecas não proprietárias, entre outras ligações que podem dificultar o transporte.

É importante ressaltar que este refatoramento não considera os problemas inerentes ao transporte de software entre plataformas de hardware diferentes, apenas atacando problemas no transporte do sistema entre ambientes de software distintos.

5.3.6 Mecanismo

1 Identificar os recursos do sistema operacional utilizados pelo programa.

Nesta atividade, procuram-se aqueles recursos que podem interferir no funcionamento do software. Alguns exemplos são: RPC (Remote Procedure Call), Threads, MMF (Memory Mapped File), segmento compartilhado.

2 Identificar a funcionalidade do software que faz uso de cada recurso.

Tanto esta atividade como a anterior podem ser realizadas através de entrevista com especialistas no sistema ou através da investigação do próprio código do sistema.

3 Identificar exatamente o que é utilizado de cada recurso.

Nesta atividade deve-se listar a interface de acesso ao recurso que é realmente utilizada pelo sistema para que seja possível verificar a necessidade de uso do recurso e a possibilidade de sua substituição.

4 Verificar a necessidade de uso do recurso.

Nesta atividade deve-se verificar se a funcionalidade do software que faz uso de um recurso pode ser reproduzida com outra abordagem. Caso isto seja possível é preferível substituir a solução que implementa a funcionalidade em questão. Com isto, reduz-se o número de recursos a serem isolados. É importante ressaltar que para decidir pela substituição da solução utilizada, outras variáveis devem ser consideradas, para que não haja perda na funcionalidade ou no desempenho do sistema, por exemplo.

5 Para cada recurso identificado como necessário, é preciso:

5.1 Definir e criar o tipo abstrato de dado para uso do recurso.

Esta é uma etapa muito importante e crucial no refatoramento. A definição desta interface deve considerar as informações coletadas no passos 2 e 3. Esta interface permanece constante (assim, reduz o impacto de mudanças na aplicação) mesmo que a implementação da interface mude a medida que o sistema é transportado de ambiente para ambiente [Bass, 1998].

5.2 Implementar o recurso nas duas plataformas.

Com o tipo abstrato de dados definido, é preciso implementar as classes que respondem pela funcionalidade do recurso e que obedecem aos tipos abstratos definidos no item 5.1.

5.3 Planejar, projetar, implementar e executar os testes que validem as classes implementadas nas duas plataformas.

5.4 Refatorar o trecho de código, em todos os trechos do sistema que utilizam o recurso, de modo a utilizar o pacote de independência.

5.5 Testar o sistema.

6 Identificar recursos de bibliotecas externas (por exemplo, Microsoft Foundation Classes (mfc)) e também a freqüência com que estes recursos aparecem.

6.1 O ideal nesta etapa é automatizar a identificação destes recursos. Isto torna a identificação mais confiável por ser menos susceptível a erros. Além da maior precisão da automatização comparado ao exame do código pelo programador. Por exemplo, em sistemas codificados em linguagem C/C++, uma alternativa para a automatização seria a construção de um programa que analisasse os arquivos objeto (*.obj*) gerados por um programa e identificasse os símbolos não encontrados. Símbolos são as variáveis, funções ou tipos de dados utilizados em um arquivo fonte. Nos arquivos-objeto gerados pelo compilador, estão presentes tabelas de símbolos que contêm todos os símbolos utilizados no arquivo fonte origem. Esta tabela também contém a informação sobre a presença ou não do símbolo no arquivo fonte, ou seja, na tabela os símbolos estão classificados em externos e internos. Um símbolo externo é aquele que não foi definido no arquivo fonte que deu origem ao arquivo objeto. Sendo assim, um programa que analise todos os arquivos objeto de um sistema e cruze as informações contidas nestas tabelas de símbolos poderia identificar os símbolos usados no sistema e que pertencem a bibliotecas externas.

6.2 É importante lembrar de eliminar da lista aqueles recursos que estão disponíveis nas duas plataformas.

7 Para cada recurso das bibliotecas externas, verificar a real necessidade de sua utilização.

7.1 Para analisar a possibilidade de substituição dos símbolos identificados no passo anterior, deve-se analisar cada trecho de código onde o recurso é utilizado e para cada um verificar a possibilidade de troca por outra abordagem. Alguns símbolos podem ser amplamente utilizados no sistema. Nestes casos, o exame do código pode ser mais dispendioso do que a implementação do recurso na plataforma destino. Por isso, é importante identificar a freqüência de uso do recurso.

8 Os recursos identificados como necessários devem ser mapeados para a plataforma destino.

8.1 Uma possível maneira de realizar este mapeamento é criar bibliotecas de utilitários que forneçam a implementação de cada recurso das bibliotecas externas utilizado no sistema.

8.2 É possível que exista a mesma funcionalidade nas duas plataformas, porém com sintaxe diferente. Assim, antes de decidir pela implementação do recurso, é aconselhável fazer uma pesquisa pela existência da funcionalidade na plataforma destino.

9 Planejar, projetar, implementar e executar os testes que validem as bibliotecas criadas.

10 Compilar, gerar e testar o sistema.

5.3.7 Conseqüências

Com a aplicação deste refatoramento, consegue-se um grau satisfatório de transportabilidade do software, uma vez que os recursos do sistema operacional estão encapsulados em classes e estas, por sua vez, escondidas atrás de tipos abstratos de dados. Para transportar o sistema para outro sistema operacional, o código cliente dos recursos presentes no pacote de independência permanece inalterado e escrevem-se novas classes que implementem os tipos definidos para cada recurso.

Este aumento na portabilidade pode ser visto como o desacoplamento entre o software e o sistema operacional e ainda como um aumento no nível de reuso do sistema.

Em contraposição, a aplicação do refatoramento *Isolamento de dependência de plataforma* pode causar impacto negativo no desempenho do sistema. A introdução de mais uma camada de software pode provocar lentidão. Deste modo, as vantagens na aplicação deste refatoramento dependerão do contexto no qual está inserido o sistema de software em questão.

5.3.8 Exemplo

É preciso lembrar que o exemplo utilizado neste trabalho vislumbra o transporte de um sistema, originalmente desenvolvido para a plataforma Windows, para a plataforma Linux.

Passos 1 e 2

Na Tabela 5.1, estão listados os recursos do sistema operacional utilizados no código do servidor LightBase bem como a funcionalidade do sistema que necessita destes recursos.

Recurso	Funcionalidade associada
Arquivos Mapeados na Memória (MMF).	Otimizar a ordenação fazendo melhor uso

	da memória e permitir uma boa velocidade mesmo em grandes objetos.
Arquivos de inicialização (.ini).	Configuração do sistema.
Linhas de execução (Threads).	Possibilidade de ambiente multi-usuário.
Semáforos, mutex, região crítica, eventos.	Controle de concorrência.
Chamada remota de procedimento (RPC-DCE).	Comunicação, via rede, com os clientes.

Tabela 5.1 - Recursos do sistema operacional utilizados pelo sistema.

Passo 3

Após a identificação dos recursos utilizados pelo sistema, deve-se identificar o que de cada recurso é utilizado, ou seja, deve-se identificar a API do recurso que é realmente utilizada.

As tabelas a seguir apresentam o que de cada recurso identificado anteriormente é utilizado no sistema.

Arquivos mapeados em memória (MMF)

CreateFileMapping	Cria um objeto de mapeamento de arquivo com ou sem nome para o arquivo especificado.
MapViewOfFile	Mapeia uma visão do arquivo para o espaço de endereçamento do processo que a chamou. O mapeamento de arquivo torna uma parte específica do arquivo visível ao processo.
UnmapViewOfFile	Retira a visibilidade de um processo a um arquivo mapeado.

Arquivos de inicialização

GetPrivateProfileString	Recupera uma cadeia de caracteres da sessão especificada em um arquivo de inicialização.
GetPrivateProfileInt	Recupera um inteiro associado com uma chave na sessão especificada de um dado arquivo de inicialização.
WritePrivateProfileString	Copia uma cadeia de caracteres na sessão especificada do arquivo de inicialização dado.
WritePrivateProfileSection	Substitui as chaves e valores em uma determinada sessão do arquivo de inicialização.

Linhas de execução (threads)

CreateThread	Cria uma linha de execução para executar no espaço de endereçamento do processo que a criou.
_beginthread	Cria uma linha de execução.

Semáforos, mutex, região crítica, eventos

WaitForSingleObject	Espera pela mudança de estado em um objeto de sincronização como semáforo, *mutex*, evento.
WaitForMultipleObjects	Espera pela mudança de estado em vários objetos de sincronização como semáforo, *mutex*, evento.
CreateMutex	Cria um objeto *mutex* com ou sem nome.
OpenMutex	Retorna um identificador de um objeto *mutex* existente que é identificado através do seu nome.
ReleaseMutex	Realiza uma operação de incremento no *mutex*.
InitializeCriticalSection	Inicia um objeto região crítica. O objeto região crítica deve ser criado e logo após inicializado para que possa ser utilizado na sincronização de linhas de execução, ou processos.
DeleteCriticalSection	Libera todos os recursos usados pelo objeto região crítica.
EnterCriticalSection	Espera até conseguir entrar na região crítica especificada.
LeaveCriticalSection	Sai da região crítica especificada.
CreateEvent	Cria um objeto evento com ou sem nome.
OpenEvent	Retorna um identificador de um evento existente.
SetEvent	Muda o estado do evento especificado para sinalizado.
ResetEvent	Força o estado do evento indicado para não-sinalizado.

Chamada remota de procedimento (RPC-DCE)

O modelo RPC (*Remote Procedure Call*) descreve como processos em diferentes nodos de uma rede podem comunicar-se e coordenar atividades [Barkley, 1993]. O paradigma RPC é baseado na invocação de procedimentos de uma linguagem de programação com a diferença que o procedimento pode ser executado em outro espaço de endereçamento, por exemplo, outra máquina. Existem várias representações do modelo original: *Open Network Computing* (ONC) RPC da Sun Microsystems [SUN,1990], [SUN, 1991]; *Distributed Computing Environment* (DCE), *Open Software Foundation* (OSF) [DCE, 1991]; e a especificação da Organização Internacional de Padronização (ISO) [ISO, 1991].

A representação utilizada no sistema é RPC-DCE.

RpcMgmtStopServerListening	Faz com que a aplicação, remota ou local, pare de atender, ou receber, chamadas de RPC.
RpcServerListen	Faz com que a aplicação fique bloqueada escutando as chamadas de RPC.
RpcServerRegisterIf	Registra uma interface na biblioteca *run-time* de RPC.
RpcSmEnableAllocate	Estabelece o ambiente gerente de memória.
RpcSmGetThreadHandle	Retorna o identificador de uma linha de execução, ou NULL para o ambiente gerente de memória.
RpcServerUnregisterIf	Exclui uma interface da biblioteca *run-time* do RPC.
RpcSmDisableAllocate	Libera recursos e memória do ambiente gerente de memória.
RpcServerUseProtseqEp	Informa a biblioteca *run-time* de RPC para usar protocolo e porta especificados para receber chamadas remotas a procedimento (RPC).
RpcSmEnableAllocate	Estabelece o ambiente gerente de memória.

Passo 4

Analisando a necessidade de usar os recursos especificados, identificou-se inicialmente que o uso de arquivos mapeados na memória (MMF) poderia ser eliminado. A primeira razão é a baixa freqüência com que ordenação de arquivos grandes ocorre no sistema. A segunda, e mais forte razão, é que existem algoritmos de ordenação que poderiam contribuir muito mais com o desempenho do sistema nesta atividade do que o uso deste recurso. Deste modo, o uso de arquivos mapeados em memória pode ser eliminado e assim o será.

Passo 5

Esta etapa do refatoramento consiste na identificação dos TADs e na implementação das classes do sistema. Por questões de simplicidade e clareza, apenas o tratamento de três dos recursos anteriormente identificados serão mostrados e analisados aqui.

Linhas de execução

1 Definição e criação de TADs

Para definir o tipo abstrato de dados que representará uma linha de execução, analisa-se primeiramente a interface do recurso que é verdadeiramente utilizada e já identificada anteriormente. De acordo com ela, a única utilidade do recurso é a criação de linhas de

execução, outras possibilidades como, por exemplo, atribuições de prioridade, não são necessárias.

Deste modo, deve-se concentrar apenas na criação de linhas de execução. Em C++ sempre que uma nova linha de execução precisa ser criada, deve-se criar uma nova função com o código inerente à funcionalidade. Isso pode povoar o sistema com funções. Mas o sistema é orientado a objetos, deste modo, precisa-se de uma maneira para encapsular esta necessidade que é inerente à linguagem e não às linhas de execução.

Diante destas observações, apresenta-se a seguir uma possível modelagem de classes de acesso a linhas de execução.

```
//Interface de uma classe que pode ser executada.
class Runnable
{
public:
	virtual void run(void) = 0{};
	Runnable(){};
	virtual ~Runnable(){};

};

//Classe que representa um thread.

/** Esta classe executa em um thread separado qualquer instância de uma classe que
herde de Runnable. Consiste de uma abstração de mais alto nível do que
SystemThread. **/

class Thread : public Runnable
{
public:

	Thread( Runnable * rtarget );

	/**Inicia uma nova linha de execução e nela executa o run do target. Se
target for nulo, não faz nada.
	**/
	void run(void);

	/** Seta o objeto a ser executável por esta linha de execução.**/
	void init ( Runnable * target );

	/**Espera pela execução do thread.
		@milliseconds - número de milésimos de segundos que deve esperar.**/
	int wait( long milliseconds );

	Thread();
	virtual ~Thread();

private:
	//Objeto a ser executado.
	Runnable * target;

	//Identifica a linha de execução criada pelo sistema.
	void* hThread;
};
```

Com esta implementação, onde se identifica a existência do padrão de projeto *Command* [Gamma, 1994], não é mais necessário criar funções e qualquer classe que

implemente a interface `Runnable` pode ter seu método `run` executado em uma linha de execução diferente. A implementação da classe `Thread` pode confirmar estas afirmações.

```
unsigned long threadFunc( void * params);

Thread::Thread()
{
	target = NULL;
	hThread = NULL;
}

Thread::~Thread()
{
	if ( hThread != NULL ){
		(SystemThread::getInstance())->Destroy( hThread );
	}
}

int Thread::wait(long milliseconds)
{
	return (SystemThread::getInstance())->Wait( hThread, milliseconds );
}

void Thread::init(Runnable * target)
{
	this->target = target;
}

void Thread::run()
{
	if( !target ){
		return;
	}
	hThread = (SystemThread::getInstance())->Create( &threadFunc, (void *)target
);
}

Thread::Thread(Runnable * rtarget)
{
	init( rtarget );
}

unsigned long threadFunc( void * params )
{
	((Runnable *)params)->run();
	return 0;
}
```

Esta constitui apenas a primeira etapa do mapeamento do recurso. Ao observar com cuidado a classe anterior, identifica-se uma classe que ainda não foi tratada: `SystemThread`. Esta classe esconde as diferenças de sistema operacional da classe `Thread`. Independente da plataforma onde o sistema executa, a classe `Thread` funciona, desde que a classe `SystemThread` lhe forneça o acesso correto ao sistema operacional. Assim, a classe que realmente esconde as diferenças entre as plataformas é `SystemThread` mostrada a seguir.

```
//Representa um thread do sistema operacional.
/** Define a interface de comunicação com o SO para o uso de thread.**/

class SystemThread
{
public:
```

```
       virtual void Destroy( void * hThread ) = 0;
       virtual int Wait( void * hThread, long milliseconds ) = 0;
       virtual void * Create( LPTHREAD_START_ROUTINE lpFunc, void* params ) = 0;
       static SystemThread * getInstance();
       virtual ~SystemThread();

private:
       static SystemThread * pthread;
protected:
       SystemThread();
       static SystemThread * newSysThread(void);
};
```

Como é possível notar, esta classe é implementada como um *Singleton* [Gamma, 1994]. A classe será instanciada de acordo com o sistema operacional no qual foi compilada. Esta é a interface de acesso ao ambiente hospedeiro que permite a comunicação com o mesmo. Os métodos virtuais são deixados para que as subclasses implementem. Para cada sistema operacional de destino uma nova subclasse será criada e ela saberá como se comunicar com o sistema operacional. Com isto, a classe `Thread` não precisa mudar a cada nova plataforma de destino, apenas uma nova subclasse de `SystemThread` será adicionada ao sistema e, conseqüentemente, o código cliente da classe `Thread` também permanecerá inalterado. A implementação de `SystemThread` está mostrada a seguir.

```
SystemThread* SystemThread::pthread = NULL;

SystemThread::SystemThread()
{

}

SystemThread::~SystemThread()
{

}

SystemThread * SystemThread::getInstance()
{
       if ( pthread == NULL ){
              pthread = newSysThread();
       }
       return pthread;
}

void * SystemThread::Create(LPTHREAD_START_ROUTINE lpFunc, void* params)
{
       return 0;
}

int SystemThread::Wait( void * hThread, long milliseconds )
{
       return 0;
}

void SystemThread::Destroy(void *hThread)
{

}
```

```
SystemThread * SystemThread::newSysThread()
{
#ifdef _WINDOWS
	return new SystemThreadWin();
#else
	return new SystemThreadLin();
#endif
}
```

2 Implementação do recurso nas duas plataformas

Novamente é possível notar duas novas classes `SystemThreadWin` e `SystemThreadLin`. A primeira contém a implementação de `SystemThread` para o *Windows* e a segunda para o *Linux*. A seguir estas classes estão também listadas.

```
class SystemThreadWin : public SystemThread
{
public:
	SystemThreadWin( void );
	void Destroy( void * hThread );
	int Wait( void * hThread, long milliseconds );
	void * Create( LPTHREAD_START_ROUTINE lpFunc, void* params );
	~SystemThreadWin( void );
};

SystemThreadWin::SystemThreadWin( void )
{
}

SystemThreadWin::~SystemThreadWin()
{
}

void SystemThreadWin::Destroy( void * hThread )
{
	if ( hThread ){
		wait( 0, hThread );
		::CloseHandle(hThread);
		hThread = NULL;
	}
}

int SystemThreadWin::Wait(void * hThread, long milliseconds)
{
	return ::WaitForSingleObject( hThread, milliseconds );
}

void * SystemThreadWin::Create(LPTHREAD_START_ROUTINE lpFunc, void* params )
{
	DWORD ThreadId = 0;
	return ::CreateThread(NULL, 0, lpFunc, params, 0, &ThreadId);
}

class SystemThreadLin : public SystemThread
{
private:
	pthread_cond_t cond;
	pthread_mutex_t mut;
public:
	SystemThreadLin( void );
	void Destroy( void * hThread );
	int Wait( void * hThread, long milliseconds );
	void * Create( LPTHREAD_START_ROUTINE lpFunc, void* params );
	~SystemThreadLin( void );
};
```

```
SystemThreadLin::SystemThreadLin( void )
{
}

SystemThreadLin::~SystemThreadLin()
{
}

void SystemThreadLin::Destroy( void * hThread )
{
	if ( hThread ){
		delete hThread;
		hThread = NULL;
	}
}

int SystemThreadLin::Wait( void * hThread, long milliseconds )
{
	if ( pthread_join(*((pthread_t *)hThread), NULL ) ){
		return WAIT_FAILED;
	}
	return WAIT_OBJECT_0;
}

void * SystemThreadLin::Create( LPTHREAD_START_ROUTINE lpFunc, void* params )
{
	pthread_t * hThread = new pthread_t;
	pthread_attr_t attr;
	pthread_attr_init(&attr);
	pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
	pthread_create( hThread, &attr, (void* (*) (void *))lpFunc, params);
	return (HANDLE)hThread;
}
```

Deste modo, o projeto desta parte do sistema está mostrado na Figura 5.12.

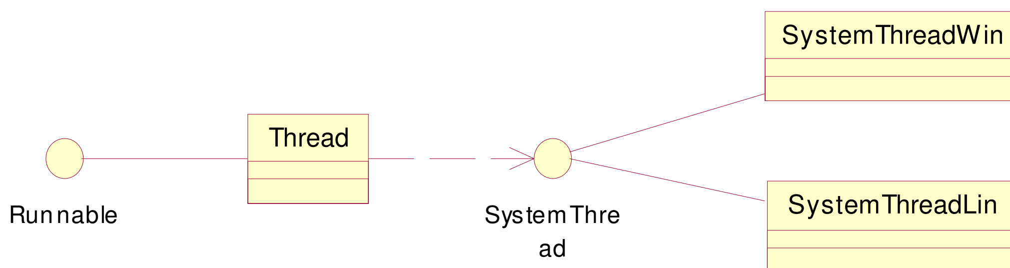


Figura 5.12 - Projeto do sistema – linhas de execução (*threads*).

3 Planejar, projetar, implementar e executar os testes que validem as classes implementadas nas duas plataformas.

Esta etapa consiste no planejamento, projeto, implementação, confecção e execução dos testes de unidade para as novas classes. Os testes automáticos devem ser confeccionados e executados com o auxílio de uma ferramenta automática de testes.

4 Refatorar o trecho de código, em todos os trechos do sistema que utilizam o recurso, de modo a utilizar o pacote de independência.

Após a certificação de que as classes funcionam bem, nas duas plataformas, é chegado o momento de fazer com que o código que usa linhas de execução no sistema inteiro faça referência às novas classes. Segue um exemplo.

```
…
DWORD dwThreadId = 0;
sLockInfo *psLock = new sLockInfo;

if ( psLock ) {
psLock->pBase = this;
       psLock->hEvent = hRet;
       psLock->uTmOut = iLockTimeOut;
       // criar thread para esperar o timer expirar
       (UINT)CreateThread( NULL,
                     0,
                     (LPTHREAD_START_ROUTINE) LockTimerProc,
                     (LPVOID)psLock,
                     0,
                     &dwThreadId );
}
return 0;
…
```

Neste trecho de código, extraído do módulo central do sistema, uma linha de execução é criada para executar a função `LockTimerProc`. Para que funcione utilizando as novas classes de tratamento de linhas de execução, ele deve ser refatorado.

Inicia-se a adaptação do código reescrevendo a parte responsável pela criação da linha de execução.

```
…
sLockInfo *psLock = new sLockInfo;

if ( psLock ) {
psLock->pBase = this;
       psLock->hEvent = hRet;
       psLock->uTmOut = iLockTimeOut;
       // criar thread para esperar o timer expirar
       Thread * pThread = new Thread ( new LockTimerObj( psLock ) );
       pThread->run();
}
return 0;
…
```

No entanto, ainda não é possível compilar o sistema, porque a classe `LockTimerObj` ainda não foi criada. Esta classe contem o código da antiga função `LockTimerProc` no método `run` e implementa a interface `Runnable`.

```
class LockTimerObj : public Runnable {
public:
       LockTimerObj();
       ~LockTimerObj();
       void run(void);
}
```

Após a criação da classe, o código que antes pertencia à função `LockTimerProc` deve ser transformado no código do método `run` da classe `LockTimerObj`. Esta nova classe representa um objeto de contagem de tempo.

5 Testar o sistema

Nesta atividade, o teste do sistema completo deve ser efetuado. O teste executado aqui também deve ser automático e deve envolver testes de funcionalidade. O objetivo é certificar-se de que o sistema apresenta as mesmas respostas aos testes que apresentava antes das modificações.

Arquivos de inicialização

Os arquivo de inicialização são utilizados no sistema como forma de armazenar informações sobre a aplicação e que precisam ser recuperadas em tempo de execução e também que podem ser modificadas pelo usuário.

No código do estudo de caso, este recurso é amplamente utilizado em todo o sistema. O seu uso é feito diretamente através da API do Windows.

A solução adotada para o caso dos arquivos de inicialização foi implementar na plataforma destino uma API de acesso a este tipo de arquivos. Deste modo, a primeira atividade deve ser a criação dos tipos abstratos de dados que representarão e darão acesso ao recurso.

6 Descrição e criação dos TADs.

Os arquivos de incialização são amplamente utilizados no sistema, no entanto, a utilidade principal deste recurso é configurar a aplicação, provendo informações que podem ser modificadas pelo usuário e que são mantidas em arquivos para facilitar sua manipulação. Assim, este recurso é utilizado como repositório de informações do ambiente no qual o sistema executa. Deste modo, um tipo abstrato de dado que represente esta funcionalidade necessita de dois métodos, um para recuperar e outro para salvar informações. O código a seguir apresenta o TAD `IEnvironment` e uma classe `Environment` que é utilizada para salvar e recuperar informações do ambiente da aplicação.

```
class IEnvironment
{
protected:
	IEnvironment(void);
	public:
	/**
		Seta uma variavel no ambiente;
		@key - chave, ou seja, nome da variavel
		@newValue - valor a ser atribuido aa variavel.
	**/
	virtual void setVar( const char * key, const char * newValue ) = 0;
	/**
		Recupera uma informacao no ambiente.
		@key - chave, ou seja, nome da variavel
		@value - buffer que ira armazenar o valor da variavel.
	**/
	virtual void getVar( const char * key, char * value ) = 0;
	virtual ~IEnvironment(void);
};
```

```
class Environment
{
protected:
	Environment(void);
	static IEnvironment* newEnv();
public:
	static IEnvironment* getInstance( void );
virtual ~Environment(void);
private:
	static IEnvironment * pcEnv;
};

IEnvironment* Environment::pcEnv = NULL;

Environment::Environment()
{
}

Environment::~Environment()
{

}

IEnvironment* Environment::getInstance()
{
	if ( pcEnv == NULL ){
		pcEnv = newEnv();
	}
	return pcEnv;
}

IEnvironment* Environment::newEnv()
{
	return new IniEnvironment();
}
```

Mais uma vez o padrão *Singleton* [Gamma, 1994] aparece no projeto do sistema. Além de o ambiente de uma aplicação ser único e, assim, ser representado por uma única instância, o uso deste padrão permite a instanciação da classe de acordo com o sistema operacional no qual o código é compilado. A outra razão para uso do padrão é que a classe é utilizada em vários pontos do sistema.

O código apresentado anteriormente ainda não finaliza a modelagem de arquivos de inicialização. É possível perceber a presença de uma classe ainda não mostrada: `IniEnvironment`. Esta classe está apresentada a seguir.

```
class IniEnvironment : public IEnvironment
{
public:
	/**
		Construtor. Inicializa o nome do arquivo ini.
	**/
	IniEnvironment();
	~IniEnvironment();
	/**
	Seta o valor de key com newValue.
	Procura no arquivo ini de mesmo nome da aplicação na sessão
	APPSETTINGS.
```

```
        **/
        void setVar( const char * key, const char * newValue );
        /**
        Recupera o valor de key e armazena em value.
        Procura no arquivo ini de mesmo nome da aplicação na sessão APPSETTINGS.
        **/
        void getVar( const char * key, char * value );

private:
        /**
                Nome completo do arquivo ini.
                Lei de formacao: <caminho_da_app><nome_do_exec>.
        **/
        char * szIniFile;

protected:
        /**
                Retorna o nome completo do arquivo ini.
        **/
        const char * getIniFile();
};

#define SECTION             "APPSETTINGS"
#define INIEXT              ".ini"
#define MAX_VALUE_SIZE      200

IniEnvironment::IniEnvironment()
{
        char szAux[250];
        strcpy( szAux, (Application::getInstance())->getPath());
        strcat( szAux, (Application::getInstance())->getExeName());
        strcat( szAux, INIEXT );
        szIniFile = strdup ( szAux );
}

IniEnvironment::~IniEnvironment()
{
        if ( szIniFile ){
                free ( szIniFile );
        }

}

void IniEnvironment::setVar( const char * key, const char * newValue )
{
        if ( key && newValue ){
                (IniFile::getInstance())->WritePrivateProfileString( SECTION, key,
newValue, getIniFile() );
        }

}
void IniEnvironment::getVar( const char * key, char * value )
{
        if ( value ){
                (IniFile::getInstance())->GetPrivateProfileString( SECTION, key, "",
value, MAX_VALUE_SIZE, getIniFile() );
        }
}

const char * IniEnvironment::getIniFile()
{
        return szIniFile;
}
```

A seguir encontra-se o código da classe `IniFile`.

```
//Abstracao de arquivos ini.
/**
	Esta classe fornece uma interface de acesso a arquivos ini.
	Obedece a mesma api do windows, mas deve ser usada tanto no Windows
	como no linux pois a instancia criada eh a da classe que implementa
	esta interface no so onde o sistema foi compilado.
**/
class IniFile
{
private:
	static IniFile * pclIni;
protected:
	IniFile();
private:
	/**
		Cria uma instancia de uma classe que obedece a interface IniFile.
		Aqui, a classe instanciada será aquela que implementa esta interface
no SO onde o sistema foi compilado.
	**/
	static IniFile * criaIni();
public:
	/**
		Retorna uma instancia de uma classe que obedece aa interface IniFile.
	**/
	static IniFile * getInstance(void);
	/**
		Recupera o valor de uma dada chave em um arquivo ini e retorna uma
string.
		@lpAppName - sessao do arquivo ini onde deve ser procurada a chave.
		@lpKeyName - chave a ser recuperada.
		@lpDefault - valor default da chave.
		@lpReturnedString - buffer onde deve ser armazenado o valor da chave.
		@nSize - tamanho do buffer.
		@lpFileName - caminho e nome do arquivo ini.
		@return - numero de caracteres lidos.
	**/
	virtual DWORD GetPrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName,
LPCTSTR lpDefault, LPTSTR lpReturnedString, DWORD nSize, LPCTSTR lpFileName ){
return 0;};
	/**
		Recupera o valor de uma dada chave em um arquivo ini e retorna um
long.
		@lpAppName - sessao do arquivo ini onde deve ser procurada a chave.
		@lpKeyName - chave a ser recuperada.
		@nDefault - valor default da chave.
		@lpFileName - caminho e nome do arquivo ini.
		@return - o valor da chave.
	**/
	virtual UINT GetPrivateProfileInt( LPCTSTR lpAppName, LPCTSTR lpKeyName, INT
nDefault, LPCTSTR lpFileName) { return 0; };

	/**
		Grava um novo valor em uma dada chave de um arquivo ini.
		@lpAppName - sessao do arquivo ini onde deve ser procurada a chave.
		@lpKeyName - chave a ser recuperada.
		@lpString - novo valor a ser usado.
		@lpFileName - caminho e nome do arquivo ini.
		@return - TRUE em caso de sucesso e FALSE caso contrario.
	**/
	virtual BOOL WritePrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName,
LPCTSTR lpString, LPCTSTR lpFileName ){ return 0; };
	/**
		Grava uma sessao inteira em um arquivo ini.
		@lpAppName - sessao do arquivo ini onde deve ser procurada a chave.
		@lpString - a sessao a ser gravada.
		@lpFileName - caminho e nome do arquivo ini.
		@return - TRUE em caso de sucesso e FALSE caso contrario.
	**/
```

```
	virtual BOOL WritePrivateProfileSection( LPCTSTR lpAppName, LPCTSTR lpString,
LPCTSTR lpFileName ){ return 0; };
	virtual ~IniFile();
};

IniFile *	IniFile::pclIni = NULL;


IniFile::IniFile()
{
}

IniFile::~IniFile()
{
	if ( pclIni ){
		delete pclIni;
	}
}

IniFile *  IniFile::criaIni()
{
#ifdef _WINDOWS
	return IniWin::criaIniWin();
#else
	return IniLinux::criaIniLinux();
#endif
}

IniFile * IniFile::getInstance(void)
{
	if ( pclIni == NULL ){
		pclIni = IniFile::criaIni();
	}
	return pclIni;
}
```

Esta classe obedece à mesma interface do *Windows* para tratamento de arquivos de inicialização. Um Singleton [Gamma, 1994] aparece para possibilitar a instanciação correta da classe. As classes `IniWin` e `IniLinux` estão listada a seguir.

```
using namespace std;

//change this if you expect to have huge lines in your INI files...
#define MAX_INI_LINE 500

//estes typedefs existem apenas para facilitar a compreensao do codigo
typedef map<string, string> INISection;
typedef map<string, INISection> INIFile;


class IniLinux : public IniFile {
public:
	IniLinux(){ };
	~IniLinux(){ };
	DWORD GetPrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName, LPCTSTR
lpDefault, LPTSTR lpReturnedString, DWORD nSize, LPCTSTR lpFileName );
	UINT GetPrivateProfileInt( LPCTSTR lpAppName, LPCTSTR lpKeyName, INT
nDefault, LPCTSTR lpFileName);
	BOOL WritePrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName, LPCTSTR
lpString, LPCTSTR lpFileName );
	BOOL WritePrivateProfileSection( LPCTSTR lpAppName, LPCTSTR lpString, LPCTSTR
lpFileName );
	static IniFile * criaIniLinux( void );
```

```
private:
	BOOL breakString(const char * completeStr, char * first, char * second, char
separator);
	string GetIniSetting(INIFile &theINI, const char *section, const char *key,
const char *defaultval="");
	void PutIniSetting(INIFile &theINI, const char *section, const char
*key=NULL, const char *value="");
	void SaveIni(INIFile &theINI, const char *filename);
	INIFile LoadIni(const char *filename);
};

IniFile *  IniLinux::criaIniLinux( void )
{
	return new IniLinux;
}

BOOL IniLinux::WritePrivateProfileSection( LPCTSTR lpAppName, LPCTSTR lpString,
LPCTSTR lpFileName )
{
	INIFile ini = LoadIni(lpFileName);
	if ( ini.size() == 0 ){
		return FALSE;
	}
	char first [ MAX_INI_LINE ], second [ MAX_INI_LINE ];
	char * aux = (char *)lpString;
	if ( strlen(aux) >= MAX_INI_LINE ){
		return FALSE;
	}
	while( breakString((const char *) aux, first, second, '=' ) ){
		PutIniSetting( ini, lpAppName, first, second );
		aux = aux +(strlen(aux) + 1);
		if ( strlen(aux) >= MAX_INI_LINE ){
			return FALSE;
		}
	}
	SaveIni(ini, lpFileName);
	return TRUE;
}
BOOL IniLinux::breakString( const char * completeStr, char * first, char * second,
char separator)
{
	string line(completeStr);
	int iPos = line.find(separator);
	if ( iPos <= 0 ){
		return FALSE;
	}
	strncpy( first, completeStr, iPos );
	first[ iPos ] = '\0';
	strcpy( second, completeStr+(iPos+1) );
	return TRUE;
}

BOOL IniLinux::WritePrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName,
LPCTSTR lpString, LPCTSTR lpFileName )
{
	INIFile ini = LoadIni(lpFileName);
	if ( ini.size() == 0 ){
		return FALSE;
	}
	PutIniSetting(ini, lpAppName, lpKeyName, lpString);
	SaveIni(ini, lpFileName);
	return TRUE;
}

UINT IniLinux::GetPrivateProfileInt(LPCTSTR lpAppName, LPCTSTR lpKeyName, INT
nDefault, LPCTSTR lpFileName)
{
	INIFile ini = LoadIni(lpFileName);
```

```
        if ( ini.size() == 0 ){
                return nDefault;
        }
        string Val = GetIniSetting (ini, lpAppName, lpKeyName, "");
        if ( Val.size() <= 0 ){
                return nDefault;
        }
        return atoi( Val.data() );
}

DWORD IniLinux::GetPrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName,
LPCTSTR lpDefault, LPTSTR lpReturnedString, DWORD nSize, LPCTSTR lpFileName )
{
        INIFile ini = LoadIni(lpFileName);
        if ( ini.size() == 0 ){
                strcpy(lpReturnedString, lpDefault );
                return 0;
        }
        string strVal = GetIniSetting (ini, lpAppName, lpKeyName, lpDefault);
        if ( strVal.size() > 0 && strVal.size() <= (unsigned long)nSize ){
                strcpy( lpReturnedString, strVal.c_str() );
        } else {
                strcpy( lpReturnedString, lpDefault );
        }
        return strlen( lpReturnedString );
}

string IniLinux::GetIniSetting(INIFile &theINI, const char *section, const char
*key, const char *defaultval)
{
        string result(defaultval);

        INIFile::iterator INISection = theINI.find(string(section));
        INISection::iterator apair = INISection->second.find(string(key));
        if(apair != INISection->second.end())
                result = apair->second;
        return result;
}

void IniLinux::PutIniSetting(INIFile &theINI, const char *section, const char *key,
const char *value)
{
        INIFile::iterator iniSection;
        INISection::iterator apair;

        if((iniSection = theINI.find(section)) == theINI.end())
        {
                // no such section?  Then add one..
                INISection newsection;
                if(key)
                        newsection.insert(pair<string, string> (key, value) );
                theINI.insert( pair<string, INISection> (section, newsection) );
        }
        else if(key)
        {       // found section, make sure key isn't in there already,
                // if it is, just drop and re-add
                apair = iniSection->second.find(key);
                if(apair != iniSection->second.end())
                        iniSection->second.erase(apair);
                iniSection->second.insert( pair<string, string> (key, value) );
        }
}

INIFile IniLinux::LoadIni(const char *filename)
{
        INIFile theINI;
        char  *value, *temp;
        string section;
```

```
	char buffer[MAX_INI_LINE+1];

	if(FILE *file = fopen(filename, "r"))
	{
		while(!feof(file))
		{
			buffer[0] = buffer[MAX_INI_LINE] = '\0';
			fgets(buffer, MAX_INI_LINE, file);
			if((temp = strchr(buffer, '\n')))
				*temp = '\0';		// cut off at newline
			if((buffer[0] == '[') && (temp = strrchr(buffer, ']')))
			{	// if line is like -->	[section name]
				*temp = '\0';	// chop off the trailing ']';
				section = &buffer[1];
				PutIniSetting(theINI, &buffer[1]);	// start new
section
			}
			else if(buffer[0] && (value = strchr(buffer, '=')))
			{
				*value++ = '\0'; // assign whatever follows = sign to
value, chop at "="
				PutIniSetting(theINI, section.c_str(), buffer, value);
	// and add both sides to INISection
			}
			else if(buffer[0])
				PutIniSetting(theINI, section.c_str(), buffer, "");	//
must be a comment or something
		}
		fclose(file);
	}
	return theINI;
}

void IniLinux::SaveIni(INIFile &theINI, const char *filename)
{
	FILE *file = fopen(filename, "w");
	if(!file)
		return;

	// just iterate the hashes and values and dump them to a file.
	INIFile::iterator section= theINI.begin();
	while(section != theINI.end())
	{
		if(section->first > "")
			fprintf(file, "\n[%s]\n", section->first.c_str());
		INISection::iterator pair = section->second.begin();

		while(pair != section->second.end())
		{
			fprintf(file, (pair->second > "") ? "%s=%s\n" : "%s\n", pair-
>first.c_str(), pair->second.c_str());
			pair++;
		}
		section++;
	}
	fclose(file);
}


class IniWin : public IniFile
{
private:
	IniWin();
public:
	static IniWin * criaIniWin();
	~IniWin();
	/**
		Invoca a funcao <code>GetPrivateProfileString</code> do windows.
```

```
        **/
        DWORD GetPrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName, LPCTSTR
lpDefault, LPTSTR lpReturnedString, DWORD nSize, LPCTSTR lpFileName );
        /**
                Invoca a funcao <code>GetPrivateProfileInt</code> do windows.
        **/
        UINT GetPrivateProfileInt( LPCTSTR lpAppName, LPCTSTR lpKeyName, INT
nDefault, LPCTSTR lpFileName);
        /**
                Invoca a funcao <code>WritePrivateProfileString</code> do windows.
        **/
        BOOL WritePrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName, LPCTSTR
lpString, LPCTSTR lpFileName );
        /**
                Invoca a funcao <code>WritePrivateProfileSection</code> do windows.
        **/
        BOOL WritePrivateProfileSection( LPCTSTR lpAppName, LPCTSTR lpString, LPCTSTR
lpFileName );
};

IniWin::IniWin()
{
}

IniWin::~IniWin()
{
}

IniWin * IniWin::criaIniWin()
{
        return new IniWin();
}

DWORD IniWin::GetPrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName,
LPCTSTR lpDefault, LPTSTR lpReturnedString, DWORD nSize, LPCTSTR lpFileName )
{
        return ::GetPrivateProfileString( lpAppName, lpKeyName, lpDefault,
lpReturnedString, nSize, lpFileName );
}

UINT IniWin::GetPrivateProfileInt( LPCTSTR lpAppName, LPCTSTR lpKeyName, INT
nDefault, LPCTSTR lpFileName)
{
        return ::GetPrivateProfileInt( lpAppName, lpKeyName, nDefault, lpFileName );
}

BOOL IniWin::WritePrivateProfileString( LPCTSTR lpAppName, LPCTSTR lpKeyName,
LPCTSTR lpString, LPCTSTR lpFileName )
{
        return ::WritePrivateProfileString( lpAppName, lpKeyName, lpString,
lpFileName );
}

BOOL IniWin::WritePrivateProfileSection( LPCTSTR lpAppName, LPCTSTR lpString,
LPCTSTR lpFileName )
{
        return ::WritePrivateProfileSection( lpAppName, lpString, lpFileName );
}
```

Deste modo, o projeto do sistema neste trecho corresponde ao mostrado na Figura 5.13.

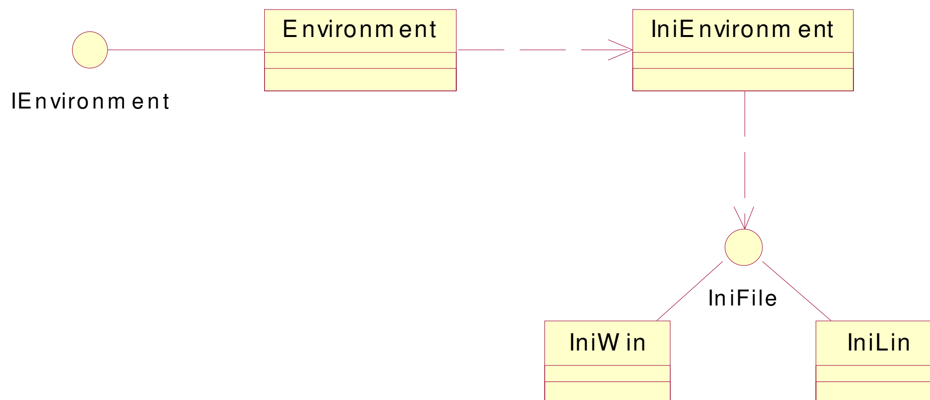


Figura 5.13 - Projeto do sistema - arquivos de inicialização.

Uma vantagem obtida com esta modelagem é a flexibilidade. Para modificar o ambiente de repositório, por exemplo, utilizar o *Registry* do Windows ao invés de arquivos ini, a mudança é mais simples. É suficiente criar uma nova classe que implemente o TAD `IEnvironment` e que utilize o novo meio de armazenamento. Ou seja, o sistema estará conectado ao TAD `IEnvironment` que armazena e recupera informações de configuração do sistema, independente de como estas informações estão armazenadas. Assim, caso a forma de armazenamento mude, o sistema não precisa ser alterado, apenas a classe `Environment` deverá instanciar a nova classe que representa o novo meio de armazenamento das informações de configuração do sistema.

Os demais passos de criação e execução dos testes de classe e do sistema devem ser repetidos como no caso anterior.

Um exemplo de trecho de código a ser refatorado é mostrado a seguir.

```
…
// Recupera o tempo de espera para nova tentativa de lock de registro
{
	C_BaseCritSect	cCS2( this, CRITSECT2 );
	this->iLockTimeSleep = GetPrivateProfileInt( LBSINILBSSECTION,
LOCKTIMESLEEPSTR, LOCKTIMESLEEP, LBSINIFILE );

// Recupera o time-out de lock de registro
	this->iLockTimeOut = GetPrivateProfileInt( LBSINILBSSECTION, LOCKTIMEOUTSTR,
LOCKTIMEOUT, LBSINIFILE );
}
…
```

Neste trecho, a função `GetPrivateProfileInt` da *API* do *Windows* de acesso a arquivos de inicialização é invocada. É preciso reorganizar o código para que não mais referencie tal função e sim as novas classes criadas para recuperação de informações necessárias à aplicação.

Assim, o mesmo código referenciando as novas classes aparece logo a seguir.

```
…
```

```
// Recupera o tempo de espera para nova tentativa de lock de registro
{
	C_BaseCritSect	cCS2( this, CRITSECT2 );
IEnvironment * pEnv = Environment::getInstance();

char szTime[ MAX_VALUE ];
	pEnv->getVar(LOCKTIMESLEEP, szTime );
	this->iLockTimeSleep = atoi( szTime );

// Recupera o time-out de lock de registro
	pEnv->getVar(LOCKTIMEOUT, szTime );
	this->iLockTimeOut = atoi ( szTime );
}
…
```

Passo 6

Nesta etapa devem ser identificados os símbolos utilizados no programa e que são externos a ele. Por exemplo, bibliotecas do ambiente e bibliotecas proprietárias. Isto é importante porque indica a utilização de bibliotecas que podem não estar disponíveis na plataforma.

Os símbolos externos identificados no servidor LightBase estão listados na Tabela 5.2 a seguir.

Símbolo	**Funcionalidade**	**Freqüência**
strupr	Converte os caracteres de uma cadeia para maiúsculas.	82
_strlwr	Converte os caracteres de uma cadeia para minúsculas.	54
Sleep	Coloca o processo para dormir por um determinado tempo.	45
stricmp	Compara duas cadeias de caracteres sem considerar maiúsculas e minúsculas.	294
strdup	Duplica a cadeia de caracteres.	195
itoa	Converte um número inteiro em uma cadeia de caracteres.	38

GetLastError	Retorna o último erro ocorrido após alguma operação do sistema.	19
SetLastError	Atribui um novo valor ao erro do sistema.	15
stat	Retorna informação acerca do estado de um arquivo.	4
_S_IFMT Tipo do arquivo _S_IFDIR Diretório _S_IREAD Permissão de leitura. _S_IWRITE Permissão de escrita. _S_IEXEC Permissão de execução.	Constantes utilizadas para indicar o tipo do arquivo na estrutura passada à função anterior.	24

Tabela 5.2 - Símbolos externos identificados pelo programa de análise de arquivos-objeto.

Os símbolos apresentados na tabela consistem de um subconjunto do total de 33 símbolos identificados com o objetivo de facilitar a compreensão do exemplo de refatoramento. É importante anotar o número de vezes que o símbolo é utilizado no sistema.

Passo 7

Para concluir algo sobre a utilidade, ou não, dos símbolos identificados, é preciso analisar o código no qual o símbolo está sendo utilizado. Veja a seguir a análise dos trechos de código aonde aparecem alguns dos símbolos listados na Tabela 5.2.

O exame dos símbolos que são amplamente utilizados pode implicar no gasto de muito esforço e por isto deve ser evitado. De acordo com a Tabela 5.2, *strupr*, *sleep* e *stricmp* são utilizados no sistema em várias ocasiões e assim, são exemplos de casos onde é preferível implementar comportamento semelhante na plataforma destino a examinar a eventual exclusão de seu uso.

Passo 8

Todos os recursos presentes na Tabela 5.2 devem permanecer no sistema e, conseqüentemente, mapeados para a plataforma destino. Estes recursos podem não estar disponíveis na plataforma destino e, neste caso, devem ser implementados, ou podem estar

disponíveis de forma diferente, ou seja, com uma interface de acesso diferente. Neste caso os recursos podem ser mapeados através do uso da idéia do padrão *Adapter* [Gamma, 1994].

A análise de cada um dos recursos encontra-se a seguir.

- Sleep

A função sleep está também disponível na plataforma Linux que é o destino do sistema, porém, apresenta uma simples diferença de sintaxe. Para resolver esta diferença, cria-se uma biblioteca chamada ***Util*** que contém o mapeamento dos recursos. No caso de `sleep`, este mapeamento consiste da linha a seguir.

```
#define Sleep(x) sleep(x)
```

- Strupr

Este recurso não está presente no Linux e por isso deve ser implementado. Assim, a função será implementada e adicionada à biblioteca ***Util***.

```
char * _strupr ( char *string )
{
       int i = 0;
       while(string[i] != NULL){
              string[i] = toupper (string[i]);
              i++;
       }
       return string;
}
```

- Stricmp

Este recurso também está presente no Linux, porém com um nome diferente: `stricasecomp`. Neste caso, a linha seguinte também é suficiente para o mapeamento do recurso.

```
#define stricmp stricasecomp
```

Passo 9 e 10

Testes automáticos de unidade devem ser implementados e executados para a validação das classes da biblioteca e testes funcionais e de regressão devem ser executados para validar o sistema.

6 REFATORAMENTOS QUE AUMENTAM A COESÃO

Neste capítulo apresentam-se os refatoramentos que contribuem para o aumento dos índices de coesão de um componente de software, seja este componente uma classe ou um conjunto delas (um módulo).

Coesão existe quando todos os elementos de um componente (classe ou módulo, por exemplo) trabalham juntos para fornecer algum comportamento conhecido [Booch, 1994]. Esta é uma característica desejável no desenvolvimento de sistemas, pois ajuda a manter o código simples e independente. Deve, então, ser considerada durante todas as decisões de projeto tomadas no ciclo de vida de um sistema.

Baixos índices de coesão podem indicar que o componente responde por muitas responsabilidades e, possivelmente, estas responsabilidades estão pouco relacionadas umas com as outras. Ou ainda, pode significar que o componente não responde por uma funcionalidade definida e apenas contribui para a realização de alguma atividade. Esta situação, além das desvantagens citadas em seguida, pode implicar na aparição de acoplamento excessivo entre os componentes.

Um componente que detém baixos índices de coesão pode apresentar os seguintes problemas [Larman, 1997]:

- difícil compreensão;
- dificuldade no reuso;
- dificuldade de manutenção;
- mudanças no sistema constantemente afetam o componente.

6.1 Divisão de classe de fronteira

6.1.1 Fundamentos

***Padrão* Façade**

Para compreender este refatoramento, é preciso conhecer o padrão de projeto chamado *Façade*. Ele provê uma interface única de acesso para um conjunto de classes em um subsistema [Gamma, 1994]. Deve ser usado quando se deseja fornecer uma interface simples de acesso a um subsistema complexo ou quando se deseja dividir o sistema em camadas.

Os participantes do padrão são: a classe que representará o ponto de entrada do sistema (a fachada), e as classes clientes deste subsistema. Estas classes podem ter visibilidade tanto da fachada como das demais classes do sistema, dependendo da sua necessidade. A Figura 6.1 ilustra o projeto proposto pelo padrão.

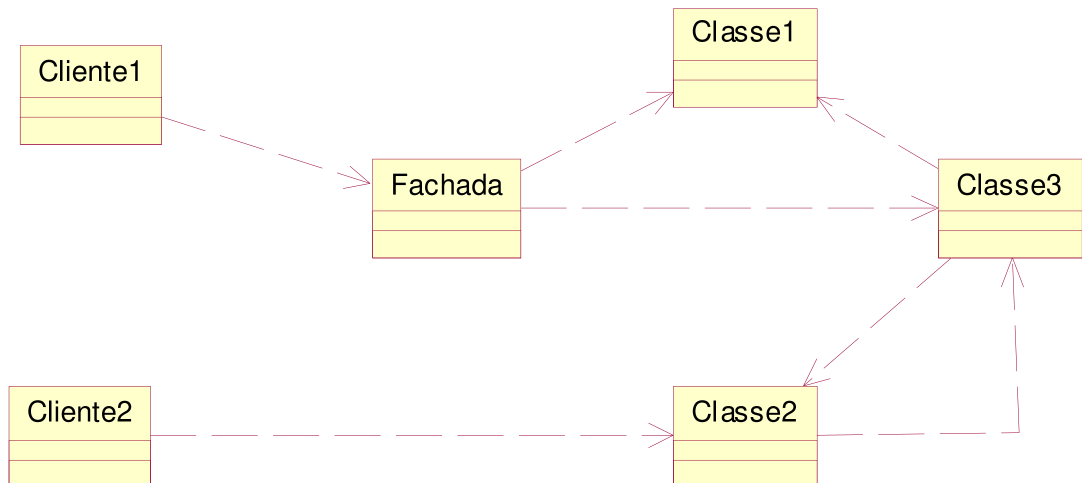


Figura 6.1 - Padrão de projeto *Fachada*.

A classe `Fachada` fornece uma interface mais simples de acesso às funcionalidades fornecidas pelo subsistema que é composto pelas `Classe1`, `Classe2` e `Classe3`.

God Class

Classes pouco coesas que contêm dezenas de métodos e respondem por uma grande parcela da responsabilidade total de um sistema são conhecidas como *God Classes*. Essas classes raramente representam boas abstrações. Estão, geralmente, povoadas por métodos e atributos não relacionados [Brown, 1998], freqüentemente abrangendo níveis variados de abstração. Como conseqüência da sua falta de coesão, elas são difíceis de manter e atualizar, pois qualquer mudança pode afetar muitas funcionalidades do sistema, já que estão concentradas em uma única classe.

6.1.2 Sumário

Este refatoramento modifica uma classe de fronteira[2], com características de uma *God Class*, de um subsistema permitindo que a sua interface externa permaneça a mesma. A

[2] Classes de fronteira são as classes que permitem acesso a um módulo. Elas são visíveis fora do módulo ao qual pertencem.

necessidade de manter a interface externa pode ser justificada, por exemplo, pela impossibilidade de modificar todo o código cliente da classe.

A classe é alterada, mas, com o uso do padrão *Façade*, estas alterações são visíveis apenas internamente ao módulo no qual está a classe e que pode ser alterado.

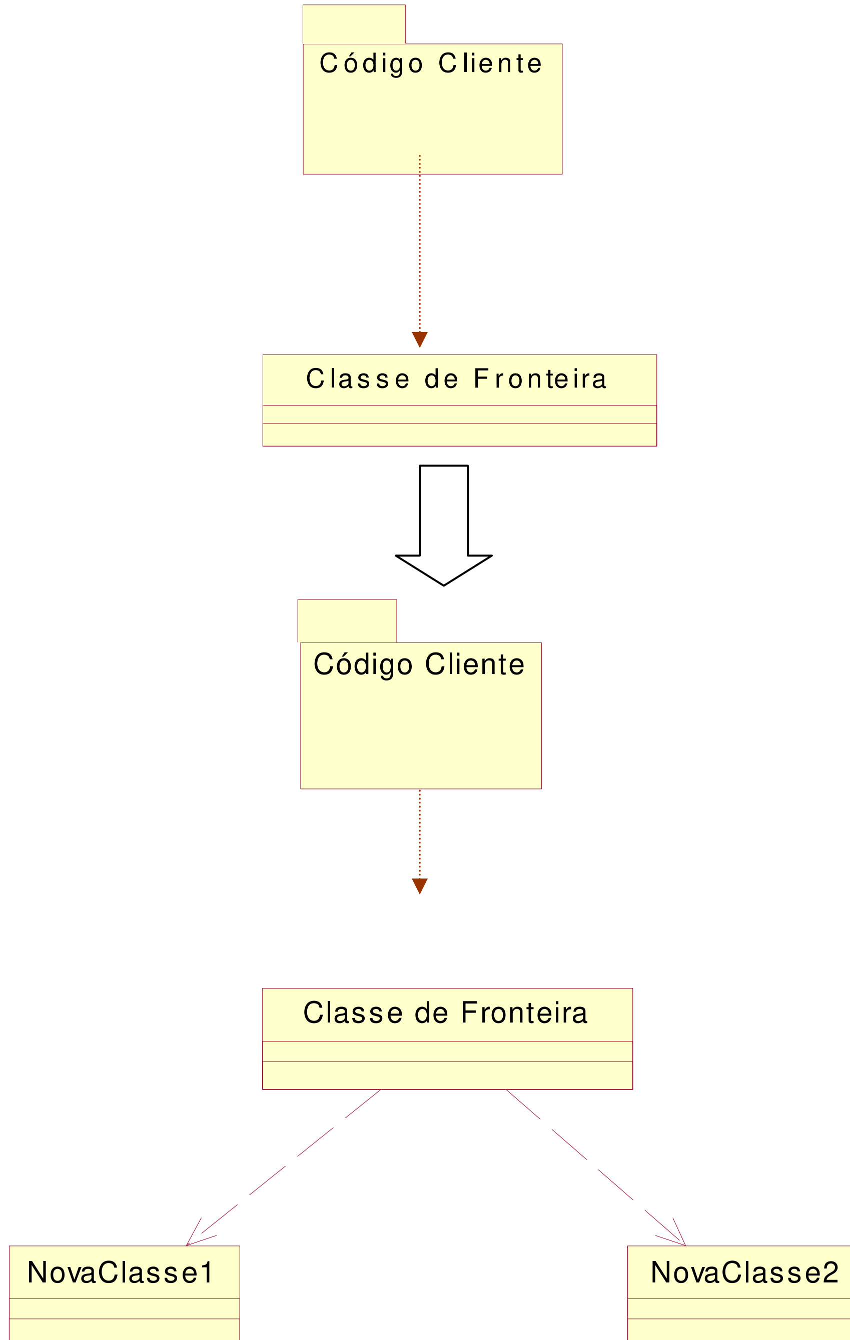


Figura 6.2 - Refatoramento *Divisão de classe de fronteira*.

6.1.3 Condições de aplicação

Este refatoramento deve ser aplicado quando há uma classe de fronteira que apresenta as características de uma *God Class*. Além disto, o código cliente deste subsistema não pode, ou não deve, ser alterado. As razões que impossibilitam a alteração deste código podem ser várias, por exemplo, o subsistema sendo alterado é distribuído e aplicações são construídas fazendo uso dele; deve-se manter a compatibilidade com versões anteriores; ou ainda, o código cliente está distribuído entre várias aplicações que inviabilizam a alteração.

6.1.4 Mecanismo

1 Identificar a funcionalidade a ser extraída da *God Class*.

2 Identificar os métodos inerentes à funcionalidade.

3 Criar nova classe que será a nova responsável pela funcionalidade identificada.

4 Certificar-se de que os métodos identificados anteriormente são realmente utilizados e não contêm código inútil. Caso contenham código inútil, ou seja, código que nunca é executado, este deve ser eliminado. Para verificar a utilidade de um método, alguns cuidados devem ser observados:

  4.1 Caso o método seja público, ou seja, visível interma e externamente ao módulo, muito cuidado deve ser tomado ao eliminá-lo. É possível excluí-lo sem problemas da classe fachada se todo o código cliente, apesar de não poder ser modificado, puder ser examinado. Desta forma pode-se ter certeza de que o método não é utilizado. Mas caso não se tenha acesso a todas as aplicações clientes deste subsistema, não se pode afirmar que ele não é utilizado, conseqüentemente, este método não deve ser eliminado.

  4.2 Caso o método seja visível apenas internamente ao subsistema ou mesmo apenas internamente à classe, ele pode ser eliminado com mais tranqüilidade já que o código cliente pode ser examinado por completo. Assim, o método não utilizado pode ser eliminado da classe fachada.

5 Adicionar à nova classe os métodos realmente utilizados que foram identificados anteriormente.

6 A adição dos métodos deve ser acompanhada da adição do comportamento deles que deve ser extraída da *God Class*. Em alguns casos esta extração de comportamento será direta; porém pode ser preciso fazer alguns ajustes ao código para que ele se adapte à nova classe.

7 Identificar atributos da classe de fronteira que sejam inerentes à funcionalidade sendo extraída.

8 Criar estes atributos na classe nova, se houver.

9 Caso os atributos não sejam mais utilizados dentro da classe de fronteira, eles podem ser eliminados.

10 Os mesmos cuidados tomados para a eliminação de métodos não utilizados devem ser tomados aqui.

11 Planejar, projetar, implementar e executar os testes da nova classe que validem a sua funcionalidade.

Os passos apresentados representam a extração de uma nova classe a partir da antiga. Eles devem ser aplicados exaustivamente até que a classe origem tenha sido completamente dissociada em classes coesas e representativas de boas abstrações. Os passos seguintes indicam como transformar a classe antiga em uma fachada.

1 Fazer a classe fachada (antiga *God Class*) referenciar a(s) nova(s) classe(s).

    1.1 Caso existam muitas referências à classe e estas não sejam feitas através de interfaces e *Factory Methods* [Gamma, 1994], o nome da classe deve ser mantido, pois não é desejável modificar todo o código cliente. Porém se este não for o caso, a substituição da nomenclatura em favor da clareza e da melhor representação de sua funcionalidade deve ser considerada.

2 É importante identificar claramente o relacionamento entre a classe fachada e a classe recentemente criada, pode se tratar de agregação, ou uma simples associação.

    2.1 Se a nova classe representar uma funcionalidade que não é usada freqüentemente e representa um processamento isolado e independente, o relacionamento entre as duas representa provavelmente uma associação. Neste caso, um objeto da classe nova será instanciado apenas durante a execução do método da classe fachada e logo após seu processamento, será destruído.

    2.2 Caso contrário, ou seja, a funcionalidade extraída seja necessária para o desempenho das demais atividades da classe fachada, a relação entre elas pode ser uma agregação. Neste caso, identifica-se uma agregação, onde a classe fachada contém uma referência para a nova classe.

3 Compilar e testar a classe fachada cobrindo todas as funcionalidades do sistema para as quais ela, agora, permite acesso.

4 Compilar e testar todo o sistema.

6.1.5 Conseqüências

A introdução de mais uma camada de software é, mais uma vez, motivo de possível queda de desempenho do sistema. No entanto, os benefícios obtidos com a aplicação do refatoramento são muito expressivos.

Este refatoramento permite grandes mudanças no projeto de um módulo central do sistema, ou seja, do qual várias outras aplicações dependam e que sofre pressões de mudança constantes. Possibilita também que estas alterações sejam realizadas sem que seja necessária a modificação de todo o código cliente. Isto pode ser entendido como um reflexo do baixo acoplamento entre o subsistema e seus clientes.

Muito do código existente na classe e não mais utilizado, o que é freqüente em uma *God Class*, será eliminado com este refatoramento já que as responsabilidades da classe serão divididas entre novas classes. Isto permitirá a identificação de código desnecessário, e, conseqüentemente a sua eliminação. Esta é mais uma vantagem do refatoramento: eliminar "código morto", ou seja, código que não é utilizado e apenas dificulta a compreensão do sistema.

A elevação dos índices de coesão das classes do sistema pode ser facilmente observada quando se examina o mecanismo deste refatoramento e percebe-se que uma classe é subdividida em outras com funcionalidades específicas. O processamento pelo qual uma única classe era responsável foi distribuído por outras classes, cada uma respondendo por uma destas responsabilidades. A classe fachada não é responsável direta por nenhuma funcionalidade do sistema, ela consiste apenas de uma camada de acesso às funcionalidades. Deste modo, não é considerada uma *God Class* uma vez que não realiza processamento, apenas delega a responsabilidade de realizá-los.

Apesar de ser um refatoramento que modifica o projeto interno de módulos do sistema, o risco de sua aplicação é diminuído pelo uso do padrão *Façade* para tratar as classes de fronteira e, assim, diminuir a propagação das modificações efetuadas.

6.1.6 Exemplo

A Figura 6.3 apresenta a seguir parte do gráfico de classes do módulo servidor do sistema que é objeto de estudo neste trabalho.

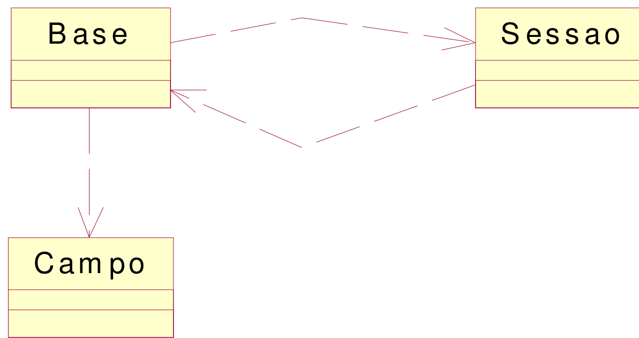


Figura 6.3 - Relação entre as classes `Base`, `Sessao` e `Campo` do exemplo.

A classe `Base` soma um total aproximado de 365 métodos e `Sessao`, 135 métodos. São exemplos claros de *God Classes* e que funcionam como porta de acesso ao módulo, ou seja, são classes de fronteira. Deste modo, identifica-se um exemplo claro de situação que sugere a aplicação do refatoramento *Divisão de classe de fronteira*. Não é possível a simples divisão das classes e criação de outras, pois estas são classes de fronteira do módulo e este visível aos clientes do sistema. A modificação desta estrutura de classes implicaria na invalidade de todo o código cliente do sistema. Como o projeto de refatoramento não abrange o código cliente, esta suposição é inadequada.

Passo 1

Considere a classe `Sessao`. O primeiro passo proposto pelo refatoramento é identificar uma funcionalidade a ser extraída. A listagem apresentada a seguir consiste da definição da classe `Sessao`.

```
class Sessao :      public LBSC_Error, public LBSC_OpInfo,
{
        friend class  LBSC_LicControl;
        friend class  LBSC_OLSort;
        friend class  Base;
        friend DWORD  LockTimerProc( LPVOID );
        friend void   CALLBACK LicTimerProc( HWND, UINT, UINT, DWORD );
        friend int    RecoverLicence();
        friend DWORD  ThreadFunc( LPVOID );
        friend struct _sLoginInfo;

        private:
                LBSC_BaseList       lbscblBaseList;
                LBSC_User                   *plbscuUser;
                C_Date              cdOpenSessionDate;
                C_Time              ctOpenSessionTime;
                LBSC_Ticket                 *plbsctTicket;
                BOOL                        bIsLogged;
                BOOL                        bIsMono;
                BOOL                        bSLogin;
                FullSTR                     strUDBLogged;
                TLoginInfo                  *pLoginInfo;
                int                         iSecurityFlags;
                FullSTR                     strAppName;
```

```
             FullSTR                     strAppType;
      static int                   iNumLic;
      static int                   iUserCacheSize;
      static TUserCacheInfo        *pUserCache;
      static int GetFNameOnProfile( char *, char * );
      static int InsertBasesInControlFile(C_RecordFile &, char * );
      int    FindUser( Base *, const char *, const char * );
      int    FindUserEx(Base *, const char *, const char * );
      void   Init();
      int    WriteImpExpFileHeader( C_File *, char *, BOOL, int );
      int    ImportBase(LBSC_Ticket *, C_File *, BOOL, char *, char *, char * );
      int    RenameToOrigFiles( char *, char * );
      int    CopyDir( char *, char *, BOOL = FALSE );
      int    PutBaseInUDB( char *, BYTE, char *, char * );
      int    DelBaseOfUDB( char *, char *, char * );
      BOOL   CheckFieldInCurrentKey( char *, Base & );
      long   GetUserType(Base *, char * );
      const LBSC_Ticket*  SLogin( TNetInfo & );
      int    BufferToFile( C_File *, C_File *, long );
      int    PutPrivateSlotInfo( const char *);
      int    GetPrivateSlotInfo( const char *, char * );
      int    GetPrivateSlotSize( const char * );
      const char *        MakePrivateSlotInfo();
      int    SizeofPrivateSlot();
      char *GetUDBName( const char * );
      int    GetBaseType( const char * );
      int    RenameDirFiles( char * );
      int    LockSessList();
      int    ReleaseSessList();
      Sessao *GetFirstSession();
      Sessao *GetNextSession();
      Sessao *GetPreviousSession();
      Sessao *GetLastSession();
      int           LockLicList();
      int           ReleaseLicList();
      TLicInfo      *GetFirstLic();
      TLicInfo      *GetNextLic();
      TLicInfo      *GetPreviousLic();
      TLicInfo      *GetLastLic();

      int ReprocBase(const LBSC_Ticket *, Base *, BOOL = TRUE );
      int GetBaseLongName( C_RecordFile &, char *, char * );
int IncorporateBase3( const LBSC_Ticket *, char *, char *, char *, char * );
      int IncorporateBase4( const LBSC_Ticket *, char *, char *, char *, char * );

      int ReorganizeUDB( const LBSC_Ticket *, Base * );
     int ConvertGroupInfo(Base *pOrigBase,Base *pNewBase=NULL);
     int ConvertBaseTo(LBSC_Ticket *pTicket,Base* pBase,int iVersionTo);
     void RemoveIndexSystem( char *, char * );
     static int FindUserInCache( const char *szUserName, const char *szPassword);
     void DelUserFromCache( const char *szUserName, const char *szPassword, const
char *szUDB );
     void    DelUDBFromCache( const char *szUDB );
     void    AddUserToCache( const char *szUserName, const char *szPassword, const
char *szUDB );
     static  void         LoadUserCacheFromBases();
     static  BOOL         LoadUserCacheFromFile();
     void ChangeUserCachePwd( const char *szName, const char *szPwd, const char
*szUDB, const char *szNewPwd );

      public:
             Sessao ();
             ~Sessao ();
      static char                   *getAppDrive( char * );
      static Sessao              *New( const char *=NULL );
      static void                      Delete(Sessao * );
      static void                      LoadUserCache();
      static void                      SaveUserCache();
```

```
	static char  *WhatUDBsForUser( const char *szUserName, const char
*szPassword, const char *szServer );
	static char  *WhatServers();
	static LBSC_AllBasesList         *WhatBases( int );
	static     LBSC_ServerBasesList  *WhatBasesOnServer( int, const char *, char
* = NULL );
	static int CreateDefaultUserBase(char*,char*, char*, char*, char*);
	static int RebuildControl( char * );
	static char *GetDefUserBaseName( char * );
	static int   DeleteDefaultUserBase( char *, char *, BOOL = FALSE );
	static void  Delete( void * );
	static void  Delete( char * );
	static void  Delete( int * );
	static void  Delete( TField * );
	static void  Delete( TUDBRecord * );
	static void Delete( LBSC_AllBasesList * );
	static void Delete( LBSC_ServerBasesList * );
	static void Delete( LBSC_PermList * );
	static void Delete( LBSC_ACLList * );
	static void Delete( LBSC_AppUserInfoList * );
	static void Delete( LBSC_AppNetInfoList * );
	static void Delete( LBSC_AppSessionNetInfoList * );
	static void Delete( LBSC_AppBaseNetInfoList * );
	static void Delete( TBaseInfo * );
	static void Delete( TSlotCache * );
	static void Delete( TSlotCacheInfo * );
	static void Delete(Sessao ** );
	static void Delete( TACL_Lists * );
	static void Delete( LBSC_MaintBase * );
	static int KillSession( char *, char *, char *, char *, char * );
	static Sessao **GetSessions(char*, char*, char*, char*, char* );
	static char *GetBaseLongName( char *, char *, char * );
	static void StBreakSecurity( int, const char * );
	static char *GetGroups( const char *, const char *, const char * );
	static const char *GetReinstallPath();

	int DeleteUDB( const LBSC_Ticket *, char *, char *, BOOL = FALSE );
	char *WhatServersForUser( char * );
	const LBSC_Ticket *Login( TNetInfo & );
	int Logout();
	BOOL IsLogged();
	BOOL IsMono();
	LBSC_AllBasesList   *WhatBasesForUser( int, char * );
	LBSC_ServerBasesList *WhatBasesForUserOnServer( int, const char *, const char
* );
	int CreateBase( const LBSC_Ticket *, char *, char*, char *, char *,
	BYTE, BOOL, char *, TField *, int, int, int = 0 );
	int CreateBase( const LBSC_Ticket *, char * );
	int OpenBase( const LBSC_Ticket *, char *, char *, BOOL, BOOL, BOOL, Base **
);
	int CloseBase( const LBSC_Ticket *, Base * );
	int DeleteBase( const LBSC_Ticket *, char * );
	int ClearBase( const LBSC_Ticket *, char * );
	const char *GetUserName();
	int Export(LBSC_Ticket *, LBSC_ExportBaseList *, char *, char *, BOOL );
int Import(LBSC_Ticket*, char*, char*, char*, char*, char*);
	int ReorganizeBase( const LBSC_Ticket *, Base * );
int ReorganizeUDB( const LBSC_Ticket *, char * );
	int ReorganizeDefaultUserBase( const LBSC_Ticket * );
	int TicketIsOk( const LBSC_Ticket * );
	int ExportBaseFormat(const LBSC_Ticket*, Base *, char * );
	char* WhatUDBLogged();
	char *WhatUDB( char *, const char * = DEFAULTSERVER );
	int CreateUDB( const LBSC_Ticket *, char*, char*, char *);
	char *GetGroups(const LBSC_Ticket*, char*, BOOL = FALSE );
	int AddUser( const LBSC_Ticket *, char *, TUDBRecord * );
	int DelUser( const LBSC_Ticket *, char *, char * );
	int UpdateUserPassword(LBSC_Ticket*, char*, char*, char*, char* );
```

```
	int UpdateUserType( const LBSC_Ticket *, char *, char *, long );
	long GetUserType( const LBSC_Ticket *, char *, char * );
	int UpdateUserDescription( LBSC_Ticket *, char *, char *, char * );
	int UpdateUserAddress(LBSC_Ticket*, char*, char*, char* );
	int UpdateUserPhone( const LBSC_Ticket *, char *, char *, char * );
	char *GetUserDescription( const LBSC_Ticket *, char *, char * );
	char *GetUserAddress(const LBSC_Ticket*, char *, char * );
	char *GetUserPhone( const LBSC_Ticket *, char *, char * );
	int AddGroupToUser( const LBSC_Ticket *, char *, char *, char * );
	int DelGroupFromUser(LBSC_Ticket*, char*, char*, char * );
	char *GetUsers( const LBSC_Ticket *, char * );
	int GetUsersData( LBSC_Ticket *, char *, TUDBRecord **, char *** );
	int LeaveBase( const LBSC_Ticket *, char * );
	int IncorporateBase(LBSC_Ticket*, char*, char*, char*, char* );
	int ChangeUDB(LBSC_Ticket*, char*, char*, char*, char * );
	int RenameBase( const LBSC_Ticket *, char *, char * );
	BOOL IsUDBOwner( const LBSC_Ticket *, char * );
	int ValidUser( const LBSC_Ticket *, const char *, const char * );
int Compare(Sessao *, int );
	Sessao *Duplicate( void );
void KillServer( char * );
	int    SetServerParam( const LBSC_Ticket *, int, int );
	static void GetServerParam( char *, int *, int * );
	static int SetClientParam( int, int, int, int, char ** );
	static void GetClientParam( int *, int *, int *, int *, char *** );
	LBSC_AppUserInfoList *GetUsersInfo( const LBSC_Ticket * );
	int UnLockRecord( const LBSC_Ticket *, long );
	int KillSessions( const LBSC_Ticket *, char *, char *, long );
	int KillServer( const LBSC_Ticket *, char * );
void BreakSecurity( int );
void SetAppInfo( const char *szAppName, const char *szAppType );
	const char *GetAppName();
	const char *GetAppType();
	void CancelOperation();
	void GetOpInfo( char *, float * );
int SetBaseForMaintenance(LBSC_Ticket* pTicketPar, const char *szBaseName, const
char* szUserName, char* szWarningMsg, BOOL bSet );
	char *GetDefaultWarningMsg();
	char *GetBaseWarningMsg( const char *szBaseName );
	char *GeMaintBaseUserName( const char *szBaseName );
	LBSC_MaintBase *GetReservedMaintBases( LBSC_Ticket *pTicketPar );
	BOOL NeedReprocOnConvert(LBSC_Ticket *pTicket, char *szBaseName, char
*szBasePassword );
	int ConvertBase( LBSC_Ticket *pTicket, char *szBaseName, char *szBasePassword
);
};
```

Através do exame desta definição, percebe-se que esta classe apresenta muitos problemas. Acumula inúmeras funcionalidades, é classe de fronteira, é de difícil compreensão, entre outros. No entanto é possível agrupar os métodos por funcionalidade. Por exemplo, UDB é um termo que aparece constantemente nos nomes dos métodos da classe. Deste modo, pode-se supor que estes métodos contribuem para uma mesma funcionalidade, ou para um conjunto de funcionalidades relacionadas.

UDB, neste caso, é a abreviação de *User database*. Este é um conceito do domínio do sistema em questão e representa a estrutura utilizada para armazenamento de cadastro de usuários e bases de dados. Apesar desta importância aparente, não existe no sistema uma abstração que o represente. Muito embora este armazenamento seja feito em bases, que são representadas pela classe `Base`, existem peculiaridades de uma UDB que não são tratadas

nesta classe. O seu tratamento está distribuído pelos métodos da classe `Sessao`. Assim, esta é uma funcionalidade que pode ser extraída da classe em questão.

Passo 2

A seguir estão listados os métodos que estão diretamente relacionados com o tratamento de UDBs no sistema. A identificação dos métodos relacionados à funcionalidade foi feita através do exame do funcionamento de cada um deles.

```
private:
int    FindUser( Base *, const char *, const char * );
int    FindUserEx(Base *, const char *, const char * );
int    PutBaseInUDB( char *, BYTE, char *, char * );
int    DelBaseOfUDB( char *, char *, char * );
char *GetUDBName( const char * );
int ReorganizeUDB( const LBSC_Ticket *, Base * );
public:
static char  *WhatUDBsForUser( const char *szUserName, const char *szPassword,
const char *szServer );
static int CreateDefaultUserBase(char*,char*, char*, char*, char*);
static char *GetDefUserBaseName( char * );
static int   DeleteDefaultUserBase( char*, char *, BOOL = FALSE );
int DeleteUDB( const LBSC_Ticket*, char*, char*, BOOL = FALSE );
int ReorganizeUDB( const LBSC_Ticket *, char * );
int ReorganizeDefaultUserBase( const LBSC_Ticket * );
int CreateUDB( const LBSC_Ticket *, char *, char *, char *);
int AddUser( const LBSC_Ticket *, char *, TUDBRecord * );
int DelUser( const LBSC_Ticket *, char *, char * );
char *GetUsers( const LBSC_Ticket *, char * );
int GetUsersData(LBSC_Ticket *, char*, TUDBRecord**, char *** );
int ValidUser(const LBSC_Ticket *, const char *, const char * );
LBSC_AppUserInfoList *GetUsersInfo( const LBSC_Ticket * );
```

Passos 3, 4 e 5

A classe criada para representar a abstração do repositório de usuários e bases de dados está listada a seguir.

```
class UDB : public Base {
protected:
       BOOL CheckFieldInCurrentKey( char * );
public:
       UDB( const LBSC_Ticket *, const char *, Sessao *, BOOL = FALSE, BOOL = TRUE,
BOOL = FALSE );
       ~UDB( void ){};
       int FindUser( const LBSC_Ticket *pTicket, const char *szUserName, const char
*szUserPassword );
       int DelUser( const LBSC_Ticket *pTicket, char *szUsers );
       int addUser( LBSC_Ticket * pTicket, TUDBRecord *ptUDBRecord);
       int addGroupToUser( const LBSC_Ticket *pTicket, char *szUserName, char
*szGroups );
       int delGroupFromUser( const LBSC_Ticket *pTicket, char * szUserName, char *
szGroups );
       int verifyUserPermission(LBSC_Ticket* pclTicket, const char *szUserName);
       int addBase( const LBSC_Ticket * plbsctTicket, char *szBaseName, BYTE
bBaseType, char *szUserName );
       int delBase( const LBSC_Ticket * plbsctTicket, char *szBaseName, char
*szUserName );
       int getUsers(LBSC_Ticket * pTicket, char ** szReturn );
       int getGroups( LBSC_Ticket * plbscTicketPar, char * szUserName, char **
szReturn );
       static void getStruct( TField * pStruct );
```

```
};
```

Uma UDB é, na verdade uma base de dados com campos pré-definidos. Deste modo, a classe `UDB` mantém as mesmas funcionalidades de `Base`, porém com um tratamento de mais alto nível para os seus clientes, já que uma UDB é entendida como um repositório de usuários e bases, e bases são repositórios de informações do usuário.

Alguns métodos da lista anterior não aparecem na classe `UDB` simplesmente por representarem uma facilidade fornecida pela classe `Sessao` e que realmente não pertencem à classe `UDB`, como por exemplo, o método `WhatUDBsForUser`.

```
static char   *WhatUDBsForUser( const char *szUserName, const char *szPassword,
const char *szServer );
```

Passo 6, 7 e 8

Não foi identificado nenhum atributo da classe `Sessao` correspondente ao tratamento de `UDB`.

O método da classe `Sessao` que deu origem ao método `addBase` foi `PutBaseInUDB` listado a seguir.

```
int
Sessao::PutBaseInUDB( char *szBaseName, BYTE bBaseType, char *szUserBaseName, char
*szUserName )
{
	if( !szBaseName || !szUserBaseName || !szUserName ){
		return( LBSE_BADARG );
	}
	if( szBaseName[0] == '\0' || szUserBaseName[0] == '\0' ){
		return( LBSE_BADARG );
	}
	strupr( szBaseName );
	strupr( szUserBaseName );
	strupr( szUserName );

	Base   lUserBase( ptTicket, szUserBaseName, this, FALSE, FALSE );
	if( lUserBase.BaseObjOk() == FALSE ){
	return ( LBSE_UBNOTOK );
	}
	if( lUserBase.GetBaseType() != USER_BASE ){
		return ( LBSE_INVALIDUSERBASE );
	}
	int iRet = FindUser( &lUserBase, szUserName, USR_PASS );
	if((iRet!=LBS_OK ) && ( iRet != LBSE_INVALIDPASSWORD ) ){
		return ( LBSE_USERNOTFOUND );
	}
	if(lUserBase.GetFRepByVal(USERBASEACCESSBASES, szBaseName) >= 0 ){
		return ( LBS_OK );
	}
	iRet = lUserBase.PutFRep( UBACCESSBASES, szBaseName );
	if(iRet == LBS_OK ){
		iRet = lUserBase.PutFRep( UBACCESSBASES, bBaseType );
		if( iRet== LBS_OK ){
			if( (iRet = lUserBase.LockRecord(ptTicket)) == LBS_OK){
				iRet = lUserBase.UpdateRecord( ptTicket );
				lUserBase.ReleaseRecord( ptTicket );
			}
		}
```

```
        }
        return( iRet );
}
```

Ao ser transportado para a classe UDB, o método passou a ser chamado addBase, ou seja, adiciona uma base de dados a UDB.

```
int
Udb::addBase(LBSC_Ticket * ptTicket, char *szBaseName, BYTE bBaseType, char
*szUserName )
{
        int iRet = FindUser( ptTicket, szUserName, USR_PASS );
        if((iRet != LBS_OK) && ( iRet != LBSE_INVALIDPASSWORD ) ){
                return( LBSE_USERNOTFOUND );
        }
        if( GetFRepByVal( UBACCESSBASES, szBaseName ) >= 0 ){
                return( LBS_OK );
        }
        iRet = PutFRep( UBACCESSBASES, szBaseName );
        if(iRet == LBS_OK ){
                iRet = PutFRep( UBACCESSBASES, bBaseType );
                if(iRet == LBS_OK ){
                        if( (iRet = LockRecord( ptTicket )) == LBS_OK ){
                                iRet = UpdateRecord( ptTicket );
                                ReleaseRecord( ptTicket );
                        }
                }
        }
        RETURN ( iRet );
}
```

A abertura da base de usuários bem como todas as devidas verificações encontradas no início do método PutBaseInUDB não aparecem no método addBase de UDB pois foram transportadas para o método que inicia as informações da classe. Os métodos que antes eram invocados da classe Base continuam sendo, porém através de herança já que a classe UDB é subclasse de Base.

O transporte do método PutBaseInUDB para a nova classe, UDB, neste caso foi facilitado principalmente pelo fato de não utilizar nenhum atributo da classe Sessao e não conter nenhuma referência a métodos que não cooperassem com a funcionalidade da classe UDB. Este é mais um indicativo de que o método estava na classe errada, uma vez que utiliza poucas informações (atributos) e serviços (métodos) da classe na qual está inserido.

Passo 9

A criação de testes automáticos e sua execução devem ser efetuadas para testar a nova classe com o objetivo de verificar seu comportamento.

Funcionalidade 1: Bases recentemente utilizadas

Passo 1

Outra funcionalidade identificada que integra a lista das responsabilidades assumidas pela classe `Sessao` é a de manter uma lista das bases de dados usadas recentemente.

Passo 2

A seguir estão listados os métodos que participam, ou cooperam, com esta funcionalidade.

```
static int    FindUserInCache( const char *szUserName, const char *szPassword);
void   DelUserFromCache( const char *szUserName, const char *szPassword, const char
*szUDB );
void   DelUDBFromCache( const char *szUDB );
void   AddUserToCache( const char *szUserName, const char *szPassword, const char
*szUDB );
static void LoadUserCacheFromBases();
static BOOL LoadUserCacheFromFile();
void   ChangeUserCachePwd( const char *szName, const char *szPwd, const char *szUDB,
const char *szNewPwd );
static void LoadUserCache();
static void SaveUserCache();
```

Passos 3, 4 e 5

A seguir está listada a classe `BaseCache`. Esta classe representa uma lista de classes recentemente utilizadas.

```
class BaseCache
{
public:
       ~BaseCache ();
       BaseCache();
       int    FindUser( const char * szUserName, const char * szPassword );
       void  LoadUserCache( void );
       void  DelUserFromCache( const char *szUserName, const char *szPassword, const
char *szUDB );
       void  DelUDBFromCache( const char *szUDB );
       void  AddUserToCache( const char *szUserName, const char *szPassword, const
char *szUDB );
       void  ChangeUserCachePwd( const char *szName, const char *szPwd, const char
*szUDB, const char *szNewPwd );
       void  SaveUserCache();

       void  LoadUserCacheFromBases( void );
       BOOL  LoadUserCacheFromFile( void );
       char* WhatUDBsForUser( const char *szUserName, const char *szPassword );


};
```

Passos 6, 7 e 8

Através da análise do código dos métodos que compõem a funcionalidade identificada, percebe-se que os atributos da classe `Sessao` encontrados no quadro a seguir são utilizados pelos métodos listados e só por eles, assim, devem ser transportados para a classe `BaseCache`.

```
int                 iUserCacheSize;
TUserCacheInfo      *pUserCache;
```

O método listado a seguir será transportado para a classe `BaseCache` após algumas alterações.

```
void
Sessao::DelUDBFromCache( const char *szUDB )
{
	C_CacheCritSect cCS0( this, CRITSECT0 );
	if ( !szUDB || !pUserCache || iUserCacheSize <= 0 ) {
		return;
	}
	if ( iUserCacheSize > 1 ) {
		int iUser = 0;
		//contar o numero de usuarios que irao sair da cache
		for ( int i=0; i < iUserCacheSize; i++ ) {
			if ( stricmp( pUserCache[i].szUDB, szUDB ) == 0 ){
				iUser++;
			}
		}
		//Alocar espaco para a nova cache
		TUserCacheInfo *pNewUserCache = new TUserCacheInfo[ iUserCacheSize -
iUser ];

		//Copiar para a nova cache os usuarios que nao sao da UDB a ser
deletada
		int j;
		for ( j = 0, i=0; i < iUserCacheSize ; i++ ) {
			if ( stricmp( pUserCache[i].szUDB, szUDB ) ){
				pNewUserCache[ j ] = pUserCache[i];
				j++;
			}
		}
		//Decrementar o contador da cachê
		iUserCacheSize -= iUser;

		//destruir a cache atual
		delete [] pUserCache;
		pUserCache = pNewUserCache;
		qsort( pUserCache, iUserCacheSize, sizeof(TUserCacheInfo),
CompareCacheInfo );
	} else {
		//Zerar a cache se nao sobrou mais nenhum usuario nela.
		delete pUserCache;
		iUserCacheSize--;
		pUserCache = NULL;
	}
}
```

Ao transportar para a nova classe, o método apresentado assume o seguinte formato.

```
void
Cache::DelUDBFromCache( const char *szUDB )
{
	C_CacheCritSect cCS0( this, CRITSECT0 );
	if ( !szUDB || !pUserCache || iUserCacheSize <= 0 ) {
		return;
	}
	if ( iUserCacheSize > 1 ) {
		int iUser = 0;
		//contar o numero de usuarios que irao sair da cache
		int i;
		for ( i = 0; i < iUserCacheSize; i++ ) {
			if ( stricmp( pUserCache[i].szUDB, szUDB ) == 0 ){
				iUser++;
```

```
                    }
            }
            //Alocar espaco para a nova cache
            TUserCacheInfo *pNewUserCache = new TUserCacheInfo[ iUserCacheSize -
iUser ];

            //Copiar para a nova cache os usuarios que nao sao da UDB a ser
deletada
            int j;
            for ( j = 0, i=0; i < iUserCacheSize ; i++ ) {
                  if ( stricmp( pUserCache[i].szUDB, szUDB ) ){
                        pNewUserCache[ j ] = pUserCache[i];
                        j++;
                  }
            }
            //Decrementar o contador da cache
            iUserCacheSize -= iUser;

            //destruir a cache atual
            delete [] pUserCache;
            pUserCache = pNewUserCache;
            qsort( pUserCache, iUserCacheSize, sizeof(TUserCacheInfo),
CompareCacheInfo );
      } else {
            //Zerar a cache se nao sobrou mais nenhum usuario nela.
            delete pUserCache;
            iUserCacheSize--;
            pUserCache = NULL;
      }
}
```

Não há alterações no código durante o transporte do método. Os atributos referenciados são aqueles também transportados para a nova classe, deste modo, não há problemas na transferência do método entre as classes.

As etapas descritas anteriormente devem ser repetidas para as demais funcionalidades acumuladas pela *God Class* até que a classe `Sessao` consista apenas de uma classe fachada, que funciona apenas como facilitadora de acesso ao módulo como um todo.

Passo 10, 11 e 12

Com todas a classes criadas e testadas, pode-se partir para a adaptação da classe `Sessao`. É preciso fazer com que ela não mais implemente as funcionalidades, mas sim, delegue responsabilidade para as classes correspondentes.

Mantendo a atenção no caso da UDB, investigam-se algumas modificações feitas nos métodos da classe `Sessao` para que façam referência à nova classe.

```
int
Sessao::AddUser( const LBSC_Ticket *pTicket, char *szUDBName, TUDBRecord
*ptUDBRecord )
{
      if( bIsLogged == FALSE ){
            return( LBSE_USERNOTLOGGED );
      }
      if( _bInvalidLicence ){
            return( LBSE_INVALIDLIC );
      }
      if( TicketIsOk( pTicket ) != 0 ){
            return( LBSE_TICKETNOTOK );
```

```
        }
        if( !ptUDBRecord || !szUDBName ){
                return( LBSE_BADARG );
        }

        Base *bUserBase = new Base(pTicket, szUDBName, this, FALSE, FALSE);
        if( !bUserBase->BaseObjOk() ){
                return( LBSE_INVALIDUSERBASE );
        }
        if(bUserBase->LB1.tcrHead.szOwnerName!= plbscuUser->GetUserName()){
                if(GetUserType(bUserBase, plbscuUser->GetUserName())!= MASTER_USER){
                        return ( LBSE_USERNOTOWNERBASE );
                }
        }
        int i = 0;
        int iRet = LBS_OK;
        while( !ptUDBRecord[ i ].IsZero() ){
                if( bUserBase->Locate( pTicket, 1, ENTIRETREE, strupr( (char*)
ptUDBRecord[ i ].szUserName ), EQUAL_KEY ) != LBS_OK ){
                        bUserBase->ClearRecord();
                        LBSC_Field    *pf = (*bUserBase)[USERBASEUSERNAME];
                        if( !pf ){
                                delete bUserBase;
                                return ( LBSE_BADUSERBASE );
                        }
                        (*pf) = strupr( (char*) ptUDBRecord[ i ].szUserName );
                        pf = (*bUserBase)[ USERBASEUSERPASSWORD ];
                        if( !pf ){
                                delete bUserBase;
                                return ( LBSE_BADUSERBASE );
                        }
                        (*pf)=strupr( (char*)ptUDBRecord[ i ].szUserPassword );
                        pf = (*bUserBase)[ USERBASEUSERTYPE ];
                        if( !pf ){
                                delete bUserBase;
                                return ( LBSE_BADUSERBASE );
                        }
                        (*pf) = ptUDBRecord[ i ].lUserType;
                        pf = (*bUserBase)[ USERBASEUSERDESCRIPTION ];
                        if( !pf ){
                                delete bUserBase;
                                return( LBSE_BADUSERBASE );
                        }
                        (*pf) = (char*) ptUDBRecord[ i ].szUserDescription;
                        pf = (*bUserBase)[ USERBASEUSERADDRESS ];
                        if( !pf ){
                                delete bUserBase;
                                return ( LBSE_BADUSERBASE );
                        }
                        (*pf) = (char*) ptUDBRecord[ i ].szUserAddress;
                        pf = (*bUserBase)[ USERBASEUSERPHONE ];
                        if( !pf ){
                                delete bUserBase;
                                return ( LBSE_BADUSERBASE );
                        }
                        (*pf) = (char*) ptUDBRecord[ i ].szUserPhone;
                        pf = (*bUserBase)[ USERBASEGROUPLIST ];
                        if( !pf ){
                                delete bUserBase;
                                return ( LBSE_BADUSERBASE );
                        }
                        (*pf) = strupr( (char*) ptUDBRecord[ i ].szUserGroup );
                        pf = (*bUserBase)[ USERBASEUSERCREATEDATE ];
                        if( !pf ){
                                delete bUserBase;
                                return ( LBSE_BADUSERBASE );
                        }
                        (*pf) = C_Date();
```

```
                    pf = (*bUserBase)[ USERBASEUSERUPDATEDATE ];
                    if( !pf ){
                          delete bUserBase;
                          return ( LBSE_BADUSERBASE );
                    }
                    (*pf) = C_Date();
                    iRet = bUserBase->AppendRecord( pTicket );

                    if ( iRet == LBS_OK ) {
                          AddUserToCache( ptUDBRecord[ i ].szUserName, ptUDBRecord[
i ].szUserPassword, szUDBName );
                    } else if ( ptUDBRecord[ 1 ].IsZero() ) {
                          delete bUserBase;
                          return ( iRet );
                    }

              } else {
                    if( ptUDBRecord[ 1 ].IsZero() ){
                          delete bUserBase;
                          return ( LBSE_USERALREADYEXIST );
                    }
              }
              i++;
       }
       delete bUserBase;
       RETURN( LBS_OK );
}

int
Sessao::DelUser( const LBSC_Ticket *pTicket, char *szUDBName, char *szUsers )
{
       if( bIsLogged == FALSE ){
              return( LBSE_USERNOTLOGGED );
       }
       if( _bInvalidLicence ){
              return ( LBSE_INVALIDLIC );
       }

       if( TicketIsOk( pTicket ) != 0 ){
              return ( LBSE_TICKETNOTOK );
       }
       if( !szUsers || !szUDBName ){
              return ( LBSE_BADARG );
       }

       Base bUserBase( pTicket, szUDBName, this, FALSE, FALSE );
       if( !bUserBase.BaseObjOk() ){
              return( LBSE_INVALIDUSERBASE );
       }
       if( bUserBase.LB1.tcrHead.szOwnerName != plbscuUser->GetUserName() ){
              if( GetUserType( &bUserBase, (char*) plbscuUser->GetUserName() ) !=
MASTER_USER ){
                    return( LBSE_USERNOTOWNERBASE );
              }
       }
       if( strstr( strupr( szUsers ), strupr( (char*) plbscuUser->GetUserName() ) )
){
              return( LBSE_USERLOGGED );
       }

       LBSC_Field    *pfName = bUserBase[ USERBASEUSERNAME ];
       if( pfName ){
              C_StrTok      cStrTok;
              char *szToken = cStrTok.StrTok( szUsers, " " );
              while( szToken ){
                    if( bUserBase.Locate( pTicket, pfName->GetId(), ENTIRETREE,
strupr( szToken ) ) == LBS_OK ){
                          if( bUserBase.LockRecord( pTicket ) == LBS_OK ){
```

```
                                  char szName[USERNAMESIZE + 1];
                                  bUserBase.GetFRep(USERBASEUSERNAME, 0, szName );
                                  char szPwd[PASSWORDSIZE + 1];
                                  bUserBase.GetFRep(UBUSERPASSWORD, 0, szPwd);

                                  bUserBase.DeleteRecord( pTicket );
                                  bUserBase.ReleaseRecord( pTicket );

                                  DelUserFromCache( szName, szPwd, szUDBName);
                           }
                    }
                    szToken = cStrTok.StrTok( NULL, " " );
             }
             return( LBS_OK );
      }
      return( LBSE_BADUSERBASE );
}
```

Os métodos anteriores pertencem à classe `Sessao` antes do refatoramento; após as alterações o código é o seguinte.

```
int
Sessao::AddUser( const LBSC_Ticket *pTicket, char *szUDBName, TUDBRecord
*ptUDBRecord )
{
      if( bIsLogged == FALSE ){
             return ( LBSE_USERNOTLOGGED );
      }
      if( _bInvalidLicence ){
             return ( LBSE_INVALIDLIC );
      }

      if( TicketIsOk( pTicket ) != 0 ){
             return ( LBSE_TICKETNOTOK );
      }
      if( !ptUDBRecord || !szUDBName ){
             return ( LBSE_BADARG );
      }

      UDB * bUserBase = new UDB(pTicket, szUDBName, this, FALSE, FALSE );
      if( !bUserBase->BaseObjOk() ){
             return( LBSE_INVALIDUSERBASE );
      }


      if ( bUserBase->verifyUserPermission ( pTicket, plbscuUser->GetUserName() )
!= LBS_OK ){
             delete bUserBase;
             return ( LBSE_USERNOTOWNERBASE );
      }

      int iRet = bUserBase->addUser( pTicket, ptUDBRecord );
      delete bUserBase;
      return ( iRet );
}

int
Sessao::DelUser( const LBSC_Ticket *pTicket, char *szUDBName, char *szUsers )
{
      if( bIsLogged == FALSE ){
             return ( LBSE_USERNOTLOGGED );
      }
      if( _bInvalidLicence ){
             return ( LBSE_INVALIDLIC );
```

```
	}

	if( TicketIsOk( pTicket ) != 0 ){
		return ( LBSE_TICKETNOTOK );
	}
	if( !szUsers || !szUDBName ){
		return ( LBSE_BADARG );
	}

	Udb bUserBase( pTicket, szUDBName, this, FALSE, FALSE );
	if( !bUserBase.BaseObjOk() ){
		return ( LBSE_INVALIDUSERBASE );
	}
	if ( bUserBase.verifyUserPermission ( pTicket, plbscuUser->GetUserName() ) !=
LBS_OK ){
		return ( LBSE_USERNOTOWNERBASE );
	}
	if( strstr( strupr( szUsers ), strupr( (char*) plbscuUser->GetUserName() ) )
){
		return ( LBSE_USERLOGGED );
	}

	int iRet = bUserBase.DelUser( pTicket, szUsers );
	RETURN ( iRet );
}
```

Após o refatoramento, apenas as validações dos parâmetros são realizadas na classe fachada, a funcionalidade em si é implementada pela classe `UDB` e assim `Sessao` invoca esta classe para que realize o trabalho por ela.

Sobre a criação da classe `BaseCache`, algumas alterações na classe `Sessao` são apresentadas a seguir.

O método público `LoadUserCache` da classe `Sessao` referente à funcionalidade implementada pela classe `BaseCache`, está listado a seguir antes e depois do refatoramento.

```
void
Sessao::LoadUserCache()
{
	if ( !LoadUserCacheFromFile() ) {
		LoadUserCacheFromBases();
	}
}
```

```
void
Sessao::LoadUserCache()
{
	if ( pCache == NULL ){
		return;
	}
	pCache->LoadUserCache();
}
```

A variável `pCache` é um apontador para um objeto do tipo `BaseCache`, ela deve ser declarada como uma variável de instância da classe `Sessao` e inicializada no construtor da classe.

Com o término das transformações na classe de fachada, os testes desta e do sistema completo devem ser efetuados.

As alterações promovidas nas classes `Sessao` e `Base` possivelmente não irão diminuir o seu número de métodos, mas sim o número de responsabilidades pelas quais elas respondem. As classes funcionam como uma interface mais simples de acesso ao módulo, mas não são responsáveis por quaisquer funcionalidades. Deste modo, a estrutura interna do sistema pode ser alterada, mas estas alterações não afetarão os clientes destas classes, já que sua interface deve ser mantida inalterada.

6.2 Divisão de módulos grandes

6.2.1 Sumário

Um módulo que não tem funcionalidade definida é dividido em outros mais coesos. Possivelmente o módulo responde por muitas tarefas que mantêm pouca relação umas com as outras. Aumentando, assim, a sua complexidade. Na Figura 6.4 encontra-se a ilustração deste refatoramento.

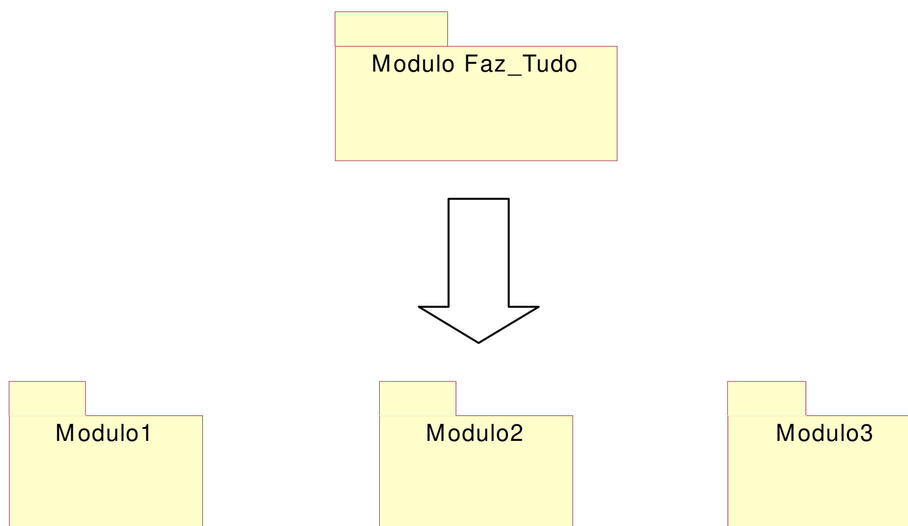


Figura 6.4 - Ilustração do refatoramento *Divisão de módulos grandes*.

6.2.2 Condições de aplicação

Em sistemas grandes é comum existirem módulos grandes e pouco coesos que podem dificultar a compreensão do sistema e, conseqüentemente, sua manutenção. Este tipo de situação geralmente reflete a existência de *God Modules*. Estes módulos geralmente são

complexos e funcionam como depósito das responsabilidades que não se sabe a quem atribuir. Refletem, na verdade, uma fraqueza no projeto do sistema por consistirem daquilo que não se encaixa em nenhum dos módulos do sistema. Como conseqüência, são compostos por um conjunto de classes que não mantêm relação nenhuma uma com a outra ou mantêm pouca relação.

Em situações nas quais não é possível listar as responsabilidades de um módulo, é provável que exista um *God Modules*. Este módulo tende a crescer, pois à medida que novas funcionalidades são requisitadas, a sua implementação pode requerer novos comportamentos que não são suportados pelo projeto do sistema, e isto pode levar a incrementos no módulo.

6.2.3 Mecanismo

1 Identificar módulo *God Module*.

2 Verificar a existência de código morto no módulo, ou seja, aquele código que não é utilizado no sistema.

Isto pode ser feito através do exame do código. Por exemplo, um método de uma classe é morto se ele não é referenciado em lugar algum no sistema ou se é utilizado em um trecho de código que não é executado. Assim como um classe é morta se não é referenciada ou é referenciada por um código que não é executado dentro do sistema.

2.1 Caso exista, eliminá-lo.

2.2 Compilar e testar o sistema.

3 Identificar as principais responsabilidades implementadas no módulo.

3.1 O objetivo desta etapa é identificar, quando possível, as funcionalidades desempenhadas pelo módulo através do exame do código fonte.

4 Para cada funcionalidade identificada, verificar a existência de um módulo que possa englobar esta funcionalidade.

4.1 Aqui, o objetivo é verificar se já existe algum módulo no sistema que responde por aquela funcionalidade, ou parte dela, que possa acomodar as classes, ou classe, presentes no *God Module*.

4.2 Caso exista, a classe, ou o conjunto de classes, deve ser movida para o módulo correspondente e todas as referências a ela devem ser atualizadas.

4.3 Caso contrário, criar um novo módulo que acomode somente a funcionalidade identificada. Logo após, as referências devem ser atualizadas.

5 É provável encontrar neste tipo de módulo, classes que, sozinhas, não respondem por funcionalidade alguma. Neste caso, deve-se identificar as classes que cooperam com esta classe e aquelas com as quais ela coopera.

6 Mover a classe para o módulo onde existam mais classes dependentes dela ou que ela dependa.

7 Atualizar as referências às classes movidas entre módulos.

8 Compilar e testar os módulos atingidos pelas mudanças e também todo o sistema.

6.2.4 Conseqüências

Com módulos coesos e com responsabilidades bem definidas, o sistema se torna mais fácil de manter, entender e até mesmo testar. A inserção de novas funcionalidades se torna, também, mais fácil e rápida.

A nova divisão de funcionalidades evita que mudanças no ambiente externo afetem o módulo e vice-versa, desde que a interface de acesso ao mesmo permaneça inalterada [Bass, 1998]. Deste modo, o sistema aumenta a sua capacidade de acomodar mudanças, e se adaptar a novas realidades.

A conseqüência da aplicação deste refatoramento é exatamente esta: aumentar a consistência dos módulos que compõem o sistema. Ainda observa-se que código morto, que não é mais utilizado, também é eliminado. Isto consiste de um grande ganho, uma vez que este código pode dificultar a compreensão do sistema.

No entanto, é necessário cautela ao aplicar este refatoramento. Existe o risco de uma multiplicação do número de módulos dentro do sistema, caso existam muitos *God Modules*. É preciso que as funcionalidades identificadas justifiquem a criação de um módulo separado. Um mesmo módulo pode responder por funcionalidades relacionadas, porém diferentes. A chave aqui é tomar cuidado na criação de novos módulos. Uma vez que muitos módulos dentro do sistema podem também dificultar a compreensão e significar um aumento também no número de dependência entre os módulos existentes.

Outra conseqüência que ainda pode ser gerada pelo grande número de módulos é a diminuição de performance, uma vez que ela depende, parcialmente, do número de iterações entre módulos que é necessária [Bass, 1998].

6.2.5 Exemplo

Uma forma de identificar módulos grandes e pouco coesos é listar as responsabilidades dos módulos do sistema. A Tabela 6.1 apresenta a lista de módulos do exemplo e suas respectivas responsabilidades.

Módulo	Responsabilidade
lbs	Todas as atividades do servidor: manipulação de bases, controle de usuários, controle de concorrência, indexação/desindexação, entre outras.
lbwServ	Receber requisições dos clientes. Repassá-las aos módulos que desempenham a atividade, e, finalmente, responder ao cliente.
appManager	Internacionalização do produto. É utilizado por todos os módulos que necessitam apresentar alguma mensagem para o usuário.
iPWorks	Comunicação através do protocolo HTTP.
liParser	Contagem das palavras de um texto e sua divisão em frases e parágrafos.
slot	Algumas classes representam abstrações de estruturas persistentes do sistema, algumas destas estruturas detêm informações especiais que são armazenadas em estruturas chamadas slots. O módulo Slot é responsável pelo armazenamento e pela recuperação destas informações especiais.
sort	Ordenação de listas de dados.
comparator	Compara duas chaves que podem ser ou não compostas.
lt	Manipulação do sistema de índices. As árvores binárias (B-Trees) são implementadas por CtawLib que nada conhece do significado da informação armazenada. É responsável, assim, pelo sistema de índices.
htmlTools	Ferramentas para manipulação de páginas html.
ctawLib	Árvores binárias (B-Trees) dos índices.
compress	Compressão dos dados.
lbstart	Personalização das cópias servidoras; verificação da validade de chaves de ativação, números de série.
liSvc	Permite que o servidor execute como serviço no Windows NT e 2000.
rpcStuff	Tratamento dos dados enviados e recebidos pela rede. Aqui os dados são criptografados e decriptografados depois de serem recebidos, ou antes,

	de serem enviados.
li	Esta é uma lib genérica. Contém várias funcionalidades utilitárias.
liFile	Leitura e escrita dos arquivos da base estão concentradas aqui. Inclui conceitos como cabeçalho, registro, etc.
crypt	Responsável pela criptografia dos dados.

Tabela 6.1- Lista dos módulos do sistema exemplo.

Com esta lista, identificam-se alguns casos de módulos que abrangem responsabilidades em excesso. No entanto, ***li*** caracteriza-se fortemente como um *God Module* pois não é possível, sequer, listar suas responsabilidades.

Com o módulo identificado, deve-se passar à investigação do mesmo para verificar a existência de código inútil. Para realizar esta tarefa, optou-se por investigar a existência de referências às classes componentes do módulo. Como todas as classes eram referenciadas, não foi detectada a existência de código inútil.

Através da análise das classes que compõem o módulo ***li***, listam-se na Tabela 6.2 as suas principais funcionalidades.

Classe	Funcionalidade
Acl	Controla o acesso dos usuários aos objetos.
AclList	Lista de Acls.
C_Buffer	Abstração de um depósito de bytes.
C_StrTok	Divide um texto em palavras.
Ticket	Responsável pela autenticação dos dados que são enviados e recebidos pelo servidor.
Perm	Gerencia as permissões dos objetos do servidor.
PermList	Lista de Perms.
Node	Elemento de uma lista.
Ttime	Abstração de hora.
Tdate	Abstração de data.
List	Implementa uma lista nodos.

Tabela 6.2 – Classes e funcionalidades de *li*.

De acordo com as funcionalidades listadas, pode-se agrupar as classes em grupos de funcionalidades como na Tabela 6.3.

Classes	Funcionalidade
Acl, AclList, Perm, PermList, Ticket	Gerência de permissões, controles de acesso e autenticação.
Ttime, Tdate	Gerência de data e hora.
List, Node, C_Buffer, C_StrTok	Tratamento de listas e cadeias de caracteres.

Tabela 6.3 - Agrupamento de classes em funcionalidades.

Não há na lista de módulos apresentada na Tabela 6.1 nenhum que responda pelas funcionalidades listadas na Tabela 6.3. Deste modo, de acordo com as sugestões do refatoramento, novos módulos devem ser criados que respondam por tais responsabilidades.

De acordo com o número de funcionalidades identificadas, três novos módulos devem ser criados. Os nomes escolhidos devem refletir a funcionalidade fornecida. Para o primeiro grupo, aquele que gerencia permissões e autenticação, ***Security***, para o segundo que gerencia datas e horas, ***Time*** e para o terceiro, gerência de listas e cadeias de caracteres, ***Lists***.

É possível que algumas classes não se enquadrem em nenhuma das categorias identificadas. Para estes casos, lugares apropriados devem ser procurados considerando o seu uso e as dependências pré-existentes entre os módulos.

Após a criação dos módulos, estabelecem-se as relações ilustradas na Figura 6.5.

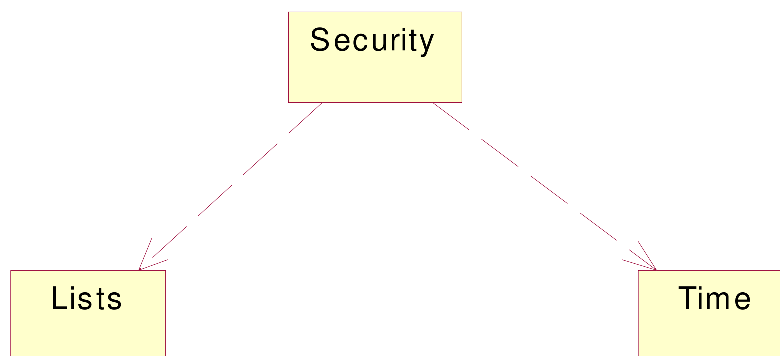


Figura 6.5 - Dependências existente entre os módulos criados a partir de *li*.

Os módulos que referenciavam ***li*** devem ser reajustados para fazer referência aos novos módulos criados a partir de sua dissociação. Para assegurar uma margem de confiança na correção das mudanças efetuadas, o sistema deve ser compilado, gerado e testado.

6.3 Junção de módulos com funcionalidade incompleta

6.3.1 Sumário

Promove o agrupamento de módulos pequenos em módulos maiores, aumentando, conseqüentemente, a coesão destes componentes por representarem, enfim, funcionalidades mais significativas e próximas daquelas do sistema.

6.3.2 Condições de aplicação

Objetiva aumentar a coesão do sistema através da eliminação de módulos que não respondem por funcionalidades completas. A aplicação deste refatoramento requer a existência de módulos que não representam funcionalidades completas e, assim, são difíceis de reutilizar em outras situações, pois provavelmente, mantêm relações de dependência com outros módulos.

6.3.3 Mecanismo

1 Identificar módulos que respondem por funcionalidades incompletas e pouco significativas no domínio do sistema.

2 Listar as atribuições destes módulos e identificar a funcionalidade do sistema com a qual cooperam.

3 Identificar os módulos com os quais mantêm dependência.

4 Decidir como destruir este módulo através de sua unificação com outros.

    4.1 Algumas observações devem ser consideradas nesta decisão. É desejável que a junção seja feita com um módulo com o qual já se mantenha algum tipo de relacionamento. Porém, também é possível que o módulo não tenha dependentes. Neste caso, se existirem módulos com responsabilidades relacionadas, mesmo que não cooperem entre si, podem ser unificados para promover uma limpeza no projeto do sistema. É possível ainda que haja casos em que seja necessária a criação de um novo módulo a partir de dois ou mais. Todavia, o importante é observar qual atitude contribuirá mais para o aumento da coesão do sistema.

5 Com a escolha da estratégia de unificação realizada, deve-se passar a efetivação desta unificação.

5.1 A junção dos módulos pode implicar na cópia de arquivos, modificação de outros arquivos, entre outros, porém o que realmente merece maior atenção são as alterações nos demais módulos do sistema que dependem do módulo destruído.

6 Com as alterações realizadas e as referências atualizadas, o módulo deve ser compilado, gerado e testado.

7 Compilar, gerar e testar o sistema completo.

6.3.4 Conseqüências

Algumas conseqüências já foram citadas ao longo da apresentação deste refatoramento, porém vale ressaltar que com módulos mais coesos, a reutilização deles em outras situações fica facilitada. Além disso, a diminuição no número de módulos presentes no sistema pode facilitar a sua compreensão. Outra grande vantagem é a possível diminuição no número de ligações entre os módulos do sistema. Quanto mais coeso é um componente, menos ele irá precisar da colaboração de outros. Isto significa que o número de conexões presentes não sistema poderá ser diminuído.

6.3.5 Exemplo

O exemplo da aplicação deste refatoramento complementa o exemplo apresentado no refatoramento anterior (*Divisão de módulos grandes*). Naquele exemplo, um módulo foi dividido em três outros: ***Security***, ***Time***, ***Lists***. Na Tabela 6.4 listam-se estes módulos bem como suas respectivas responsabilidades dentro do sistema.

Módulo	**Funcionalidade(s)**
Security	Gerência de permissões, controles de acesso e autenticação.
Time	Tratamento de data e hora.
Lists	Tratamento de listas e cadeias de caracteres.

Tabela 6.4 - Identificação de módulos e funcionalidades.

O módulo ***Time*** responde pela funcionalidade de tratamento de data e hora e tem duas classes para este fim. O módulo ***Lists*** responde pelo tratamento de listas e cadeias de caracteres e é composto por quatro classes. Tanto data, hora como listas e cadeias de caracteres são tipos básicos de dados tratados no sistema ao qual os módulos pertencem que é

um banco de dados de recuperação textual. Deste modo, pode-se concluir que data, hora, listas e cadeias de caracteres são tipos básicos de dados úteis no tratamento das informações armazenadas pelo sistema. Assim, pode-se afirmar que o módulo ***Time*** é responsável pelo tratamento de uma parte dos tipos básicos de dados tratados pelo sistema e o módulo ***Lists*** é responsável por outra parte destes tipos. Conseqüentemente, nenhum dos dois é responsável pelo tratamento completo dos tipos básicos de dados no sistema, o que sugere que nenhum dos módulos responde por uma funcionalidade completa.

Para esclarecer a ambigüidade na definição e diferenciação das funcionalidades destes dois módulos, deve-se supor a introdução de um novo tipo de dado básico, por exemplo, o inteiro de 64 bits. A primeira decisão a ser tomada seria em que módulo adicionar a abstração deste tipo, já que dois módulos são responsáveis pelo tratamento dos tipos de dados básicos no sistema. Deste modo, sugere-se a criação de um novo módulo, ***BasicDataTypes***, a partir da unificação dos dois, ***Time*** e ***Lists*** que será responsável pelo tratamento dos tipos de dados básicos utilizados pelo sistema.

Apesar de o módulo ***Security*** ser um módulo também pequeno, sua funcionalidade é bem definida no sistema: segurança. O módulo é responsável pela segurança das informações. Não há relação entre esta funcionalidade e aquelas atribuídas a ***Time*** e ***Lists***. Deste modo, não deve haver dúvidas de que este módulo não deve fazer parte da unificação proposta anteriormente.

O próximo passo é a identificação das relações que estes módulos mantêm com outros. Através do gráfico de dependências do sistema, é possível extrair o gráfico apresentado na Figura 6.6.

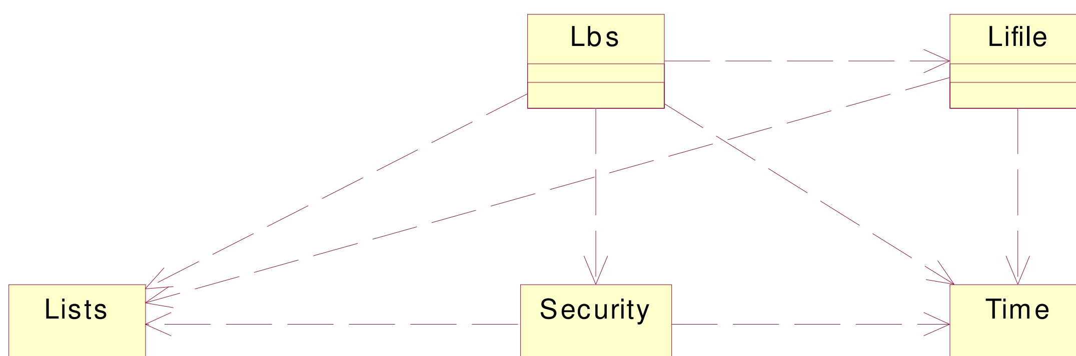


Figura 6.6 - Dependências dos módulos *Time*, *Lists* e *Security*.

Observando as dependências apresentadas pelos módulos ***Time*** e ***Lists***, verifica-se que a lista de módulos que dependem de ambos é igual. Ou seja, os módulos que dependem de um,

também dependem do outro. Esta observação reforça ainda mais a necessidade de unificação dos módulos uma vez que reforça a idéia de que ambos respondem por funcionalidades complementares. Os módulos que utilizam os tipos de dados definidos em um módulo também utilizam os tipos definidos no outro, deste modo, a unificação de ambos não implicará na introdução de dependências extras.

A solução sugerida para a unificação dos módulos foi reforçada pelo exame das dependências no sistema. Deste modo, a Figura 6.7 apresenta o gráfico de dependências após a unificação.

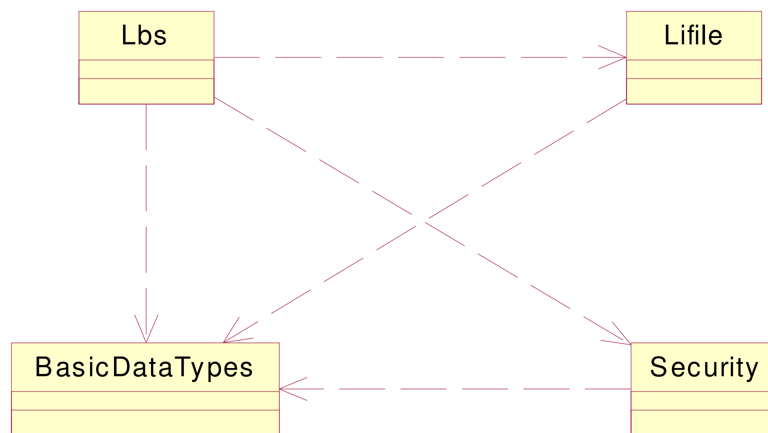


Figura 6.7 - Dependências após unificação dos módulos *Time* e *Lists*.

Os passos seguintes consistem da atualização das dependências e do teste do módulo e, posteriormente, do sistema como um todo.

7 REFATORAMENTOS QUE INTRODUZEM ABSTRAÇÕES

Uma das características de um bom projeto de sistema orientado a objetos é a presença de boas abstrações. Boas abstrações significam entidades que representem a lógica da aplicação com responsabilidades bem definidas dentro do sistema. No entanto, esta é uma das partes mais difíceis da modelagem de sistemas OO: decompor o sistema em classes e atribuir responsabilidades a essas classes. A tarefa é difícil porque engloba muitos fatores como encapsulamento, dependência, flexibilidade, performance, reuso, capacidade de adaptação, entre outros [Gamma, 1994].

Construir um bom projeto de software, assim, não é uma tarefa fácil. Um bom projeto geralmente é resultado de muitas iterações e dificilmente é atingido na primeira tentativa. É possível que algumas abstrações importantes sejam esquecidas ou não identificadas e, ainda, que as atividades de manutenção sejam realizadas sem preocupação com a evolução do projeto do sistema o que pode comprometer sua qualidade.

Os refatoramentos propostos neste capítulo têm o objetivo de nortear a introdução de abstrações que, por qualquer uma das razões citadas anteriormente, não foram adicionadas ao projeto do sistema, comprometendo assim, sua qualidade.

7.1 Modelagem de conceitos da solução através de classe

7.1.1 Sumário

A proposta deste refatoramento é introduzir a abstração de um conceito do domínio da solução que não é tratada adequadamente no sistema. Ao longo dos anos de manutenção, é possível que as correções e adições de funcionalidade implementadas introduzam novos conceitos no sistema e estes não sejam modelados adequadamente. Deste modo, este refatoramento propõe a transformação deste conceito em uma abstração bem encapsulada e detentora de uma API que supra as necessidades de seu código cliente e que representem bem a sua funcionalidade no sistema.

7.1.2 Condições de aplicação

Em alguns sistemas, existem conceitos importantes da solução que não são adequadamente tratados, muitas vezes não são tratados como classe ou classes. É possível que

o uso deste conceito esteja espalhado por todo o sistema e sua funcionalidade reimplementada várias vezes ou implementada erroneamente em classes com outras atribuições. O primeiro caso pode indicar a duplicação de código que dificulta as atividades de manutenção e mesmo a compreensão do sistema. A segunda pode comprometer a coesão da classe na qual o conceito está implementado e promover associações semanticamente inadequadas entre as classes do sistema.

As duas soluções apresentadas podem implicar em inúmeros problemas como a dificuldade de sincronização no caso de sistemas com suporte à concorrência, deixando no código cliente do recurso a responsabilidade pelo controle de inconsistência devido ao acesso concorrente. A inexistência de uma classe impossibilita o encapsulamento das informações deixando públicos aspectos inerentes ao recurso ou a sua implementação.

Estes problemas podem implicar na dificuldade de introdução de novas funcionalidades do sistema devido às fraquezas apresentadas pelo seu projeto.

7.1.3 Mecanismo

1 Criar classe que representará a abstração do conceito.

2 Adicionar os atributos que são inerentes à abstração.

   2.1 Por exemplo, o recurso em questão pode conter uma lista de dados ou uma versão, isto é importante quando a forma de representação muda e é necessário manter este controle para fazer atualização de versões antigas.

3 Adicionar os métodos de acesso ao recurso.

   3.1 Estes métodos devem ser identificados através da análise do código que utiliza o recurso; é através dele que será possível identificar a API que será necessária para que o sistema utilize o recurso da forma mais adequada.

4 Caso o sistema seja multi processado, o que é muito provável em sistemas de grande porte, é importante dispensar atenção especial à questão da concorrência.

   4.1 Nesta etapa, deve-se implementar a estratégia de controle de acesso concorrente ao recurso, caso este seja compartilhado entre várias linhas de execução.

5 Planejar, projetar, implementar e executar testes que exercitem a nova classe.

6 Modificar o código cliente para que faça referência à nova classe.

   6.1 Sugere-se que esta modificação seja feita uma por vez, testando cada caso para minimizar a possibilidade de introduzir problemas no código.

7 Compilar e testar todo o sistema.

7.1.4 Conseqüências

Uma das conseqüências da aplicação deste refatoramento é o desacoplamento do código cliente com a implementação do recurso. A eliminação de código duplicado, ou o aumento da coesão da classe que mantinha a implementação do recurso são outras conseqüências.

Outra vantagem significativa da aplicação deste refatoramento é a questão de facilidade de compreensão do código. Com uma boa abstração e bem encapsulada, além de limpo, o código se torna fácil de entender e manter.

Caso a implementação do conceito mude, os clientes não sofrerão alteração, uma vez que a forma de acesso ao recurso deve continuar a mesma.

O código duplicado é reduzido, ou até mesmo, eliminado, pois a abstração deve fornecer a API necessária de acesso ao recurso e estará concentrada na implementação da classe, não mais distribuída no código.

A maior dificuldade na aplicação deste padrão consiste na definição da interface, pois é necessário estudar todos os pontos onde o recurso é utilizado para contemplar o maior número possível de possibilidades e necessidades de acesso. Além disso, como o recurso não está modelado por uma classe, a identificação dos pontos do sistema que fazem uso deste recurso pode ser trabalhosa pricncipalmente nos casos em que o recurso é amplamente utilizado.

7.1.5 Exemplo

No servidor LightBase, existe um arquivo que é responsável pela organização das bases de dados. Neste arquivo, existe a lista com todas as bases presentes no sistema bem como as UDBs a que pertencem. A função deste arquivo é armazenar as informações de uma base que são necessárias no momento de seu uso, mas que o usuário não precisa informar. Este arquivo é amplamente utilizado no módulo ***lbs***; no entanto, não é modelado por uma abstração. O acesso ao arquivo é feito através da interface utilizada para acesso a arquivos comuns. Este é um exemplo claro de um conceito da solução que deve ser mais bem representado no sistema.

A seguir é possível verificar como é feito o acesso ao arquivo.

```
...
// Acessar lbs.ini para pegar o arquivo de controle
if( GetFNameOnProfile( szDirBase, szControlFile ) != LBS_OK ){
       ERETURN( LBSE_LBSINIERROR );
}
sprintf( szFullControlFile, "%s\\%s", szDirBase, szControlFile );
C_SessCritSect      cCS4( this, CRITSECT4 );
// verificar versao do arquivo de controle
if(GetControlFileVersion(szFullControlFile)!= CRYPTO_MAGICNUMBER){
       ERETURN( LBSE_BADCTRLFILE );
}
C_RecordFile cfBasesFile(szFullControlFile, CRYPTO_MAGICNUMBER, 0,
```

```
sizeof( TBasesFile ), "HEADKEY", "RECKEY", NULL, SH_DENYRW, TRUE);
if( cfBasesFile.IsOpen() ){
	cfBasesFile.R_Seek( 0 );
	do{
		if( cfBasesFile.R_Read( &tbfAux ) != OK ){
			break;
		}
		if( stricmp( (char*) tbfAux.szBaseName, (char*) szOldName ) == 0 ){
			break;
		}
	} while( cfBasesFile.R_SeekNext() == OK );
	if( stricmp( (char*) tbfAux.szBaseName, (char*) szOldName ) == 0 ){
		lBFilePos = cfBasesFile.R_CurPos();
	}
}
...
```

O primeiro passo é identificar a interface do recurso, no caso, um arquivo. Examinando o uso do arquivo que serve como um repositório de estruturas identificadoras de bases, identificou-se a necessidade de criar a classe a seguir.

```
//! Representa o arquivo de controle do Lbs
/*!Representação do arquivo de controle do lbs.
	Esta classe faz operações de leitura/escrita no arquivo.
*/
class LBSC_ControlFile
{
public:
	LBSC_ControlFile( void );
	int GetControlFileVersion( char *szFileName );
	int ConvertControlFileTo(int iVersion);
	int estaIncorporada( TBasesFile &tbfAux, char *szBase, char *szUdb );
	int Rebuild( );
	int RemoveBase( char *szBaseName, const char * szUDBName );
	int findEntry (TBasesFile &tbfAux, char *Base, char *Udb, long *pos);
	int addEntry( TBasesFile &tbfAux, long pos );
	static int GetName( char *szFilePath, char *szFileName );
};
```

No entanto, a definição da classe ainda não está completa. Existe apenas um arquivo de controle no sistema e o acesso a este arquivo está espalhado por todo o sistema, o que sugere que um controle de concorrência deve ser feito para manter a consistência da informação. Deste modo, optou-se por modelar a classe como um *Singleton* [Gamma, 1994]. O acesso ao arquivo será controlado através de região crítica. Veja a seguir a alteração feita sobre a classe.

```
class LBSC_ControlFile
{
private:
	MsgMapper	*_pcInter;
	static LBSC_ControlFile * pclCtrlFile;
	LBSC_ControlFile( void );

public:
	int GetControlFileVersion( char *szFileName );
	int ConvertControlFileTo(int iVersion);
	int estaIncorporada( TBasesFile &tbfAux, char *szBase, char *szUdb );
	int Rebuild( );
	int RemoveBase( char *szBaseName, const char * szUDBName );
	int findEntry (TBasesFile &tbfAux, char *Base, char *Udb, long *pos);
```

```
        int addEntry( TBasesFile &tbfAux, long pos );

        static int GetName( char *szFilePath, char *szFileName );
        static LBSC_ControlFile * getInstance( void );

        ~LBSC_ControlFile( void );
};
```

O próximo passo é a identificação dos atributos da classe. Como foi dito anteriormente, o acesso ao arquivo deve ser controlado por região crítica; deste modo, uma referência a uma classe que provenha tal funcionalidade deve ser adicionada à classe. A linha a seguir foi acrescentada à lista de atributos privados da classe.

```
CriticSect * critic;
```

A próxima etapa do refatoramento é adicionar o código à classe criada.

```
#define COMPONENTENIVEL          3
#define BASENIVEL                2
#define UDBNIVEL                        1

LBSC_ControlFile * LBSC_ControlFile::pclCtrlFile = NULL;

LBSC_ControlFile::LBSC_ControlFile()
{
        critic = (CriticSectCreator::getInstance())->newCriticSect();
        if ( critic ){
                critic->InitializeCriticalSection();
        }
        _pcInter = MsgMapper::getMapper( "LBS" );

}

LBSC_ControlFile::~LBSC_ControlFile()
{
        if ( critic ){
                critic->DeleteCriticalSection();
                delete critic;
        }

}

LBSC_ControlFile* LBSC_ControlFile::getInstance()
{
        if ( pclCtrlFile == NULL ){
                pclCtrlFile = new LBSC_ControlFile();
        }
        return pclCtrlFile;
}


int LBSC_ControlFile::Rebuild( )
{
        char            szFullAux[ FULLNAMESIZE ];
        char            szAux[ FILENAMESIZE ];

        if( GetName( szFullAux, szAux ) != LBS_OK ){
                SetLastError( (DWORD) (LBSE_LBSINIERROR) );
                return( LBSE_LBSINIERROR );
        }

        char            szFNameAux[ FULLNAMESIZE ];
```

```
        (SO::getInstance())->mountPath(szFNameAux, szFullAux, szAux);
        C_RecordFile cfBasesFile( szFNameAux, CURRENT_CTRLFILE_MAGICNUMBER, 0,
sizeof( TBasesFile ), "HEADKEY",  "RECKEY", "w", SH_DENYRW, TRUE );
        if( !cfBasesFile.IsOpen() ){
                SetLastError( (DWORD) (LBSE_NOBASESFILE) );
                return( LBSE_NOBASESFILE );
        }
        cfBasesFile.R_Seek( 0 );

        int iRet = InsertBases( cfBasesFile, szFullAux );

        cfBasesFile.Close();
        SetLastError( (DWORD) (iRet) );
        return( iRet );
}

int LBSC_ControlFile::ConvertControlFileTo(int iVersion)
{
        if ( iVersion == CTRLFILE_HIERARQUIA_DIR_BASE ){
                OldTBasesFile tbfOld;
                char                szOldCtrlFile[ FULLNAMESIZE ];
                char                szOldPath[ FULLNAMESIZE ];
                char                szOldFName[ FILENAMESIZE ];
                TBasesFile          tbfNew;
                char                szNewCtrlFile[ FULLNAMESIZE ];
                const char*         ctrlAuxName = "AUX1";
                const char*         errLogFile = "err.log";
                char *              szMsg;
                LBSC_Log            convLog( errLogFile );

                if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *) "CONVLOG_INICIO" ))
!= NULL ){
                        convLog.print( szMsg );
                        delete szMsg;
                }
                if( GetName( szOldPath, szOldFName ) != LBS_OK ){
                        if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *)
"CONVLOG_EINI" )) != NULL ){
                                convLog.print( szMsg );
                                delete szMsg;
                        }
                        SetLastError((LBSE_LBSINIERROR) );
                        return ( int ) NULL;
                }
                (SO::getInstance())->mountPath( szOldCtrlFile, szOldPath, szOldFName
);
                C_RecordFile cfOldBasesFile( szOldCtrlFile,
CRYPTO_CTRLFILE_MAGICNUMBER, 0, sizeof( OldTBasesFile ), HEADKEY",  "RECKEY", "r",
SH_DENYNO, TRUE );
                if( !cfOldBasesFile.IsOpen() ){
                        if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *)
"CONVLOG_EOPEN_ARQ_CONTROLE" )) != NULL ){
                                convLog.print( szMsg );
                                delete szMsg;
                        }
                SetLastError( (LBSE_NOBASESFILE) );
                        return ( int ) NULL;
                }
                cfOldBasesFile.R_Seek( 0 );

                (SO::getInstance())->mountPath( szNewCtrlFile, szOldPath, ctrlAuxName
);
                C_RecordFile cfNewBasesFile( szNewCtrlFile,
CURRENT_CTRLFILE_MAGICNUMBER, 0, sizeof( TBasesFile ), "HEADKEY", "RECKEY", "w",
SH_DENYNO, TRUE );
                if ( !cfNewBasesFile.IsOpen() )   {
```

```
                    if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *)
"CONVLOG_EOPEN_ARQ_CONTROLE" )) != NULL ){
                          convLog.print( szMsg );
                          delete szMsg;
                    }
                    SetLastError( (LBSE_NOBASESFILE) );
                    return LBSE_ERROR;
             }
             cfNewBasesFile.R_Seek( 0 );
             do{
                    if( cfOldBasesFile.R_Read( &tbfOld ) != OK ){
                          break;
                    }
                    if ( tbfOld.bRecDeleted ){
                          continue;
                    }
                    copia ( tbfNew, tbfOld );
                    cfNewBasesFile.R_Append( &tbfNew );
                    char   szBase[ FULLNAMESIZE ];
                    if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *)
"CONVLOG_WRITING_BASE" )) != NULL ){
                          sprintf( szBase, "%s %s - %s", szMsg, (const char
*)tbfOld.szUserBaseName, (const char *)tbfOld.szBaseName );
                          convLog.print( szBase );
                          delete szMsg;
                    }
                    if ( tbfNew.bBaseType != USER_BASE ){
                          char   szOldBPath[ FULLNAMESIZE ];
                          char   szNewBPath[ FULLNAMESIZE ];
                          char pathSeparator = (SO::getInstance())-
>getPathSeparator();
                          sprintf(szNewBPath, "%s%c%s%c%s", szOldPath ,
pathSeparator, (const char *)tbfOld.szUserBaseName, pathSeparator, (const char
*)tbfOld.szBaseName );
                          sprintf(szOldBPath, "%s%c%s", szOldPath, pathSeparator,
(const char *)tbfOld.szBaseName );
                          if ( copiaBase( szOldBPath, szNewBPath ) != LBS_OK ){
                                 if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *)
"CONVLOG_ECOPY_BASE" )) != NULL ){
                                        sprintf( szBase, "%s %s - %s", szMsg,
(const char *)tbfOld.szUserBaseName, (const char *)tbfOld.szBaseName );
                                        convLog.print( szBase );
                                        delete szMsg;
                                 }
                          } else {
                                 if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *)
"CONVLOG_COPYING_BASE" )) != NULL ){
                                        sprintf( szBase, "%s %s - %s", szMsg,
(const char *)tbfOld.szUserBaseName, (const char *)tbfOld.szBaseName );
                                        convLog.print( szBase );
                                        delete szMsg;
                                 }
                          }
                    }
             } while( cfOldBasesFile.R_SeekNext() == OK );
             cfNewBasesFile.Close();
             cfOldBasesFile.Close();
             if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *)
"CONVLOG_CLOSE_ARQ_CONTROLE" )) != NULL ){
                    convLog.print( szMsg );
                    delete szMsg;
             }
             remove( szOldCtrlFile );
             rename( szNewCtrlFile, szOldCtrlFile );
             if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *)
"CONVLOG_REMOVE_ARQ_CONTROLE" )) != NULL ){
                    convLog.print( szMsg );
                    delete szMsg;
```

```
            }
            if ( (szMsg = _pcInter->GetGenMsgsAppVar( (char *) "CONVLOG_END" )) !=
NULL ){
                  convLog.print( szMsg );
                  delete szMsg;
            }
            return LBS_OK;
      }
      return LBSE_ERROR;
}

void LBSC_ControlFile::copia( TBasesFile & tbfNew, OldTBasesFile tbfOld)
{
      tbfNew.szBaseName = tbfOld.szBaseName;
      tbfNew.szUserBaseName = tbfOld.szUserBaseName;
      tbfNew.szBaseLongName = tbfOld.szBaseLongName;
      tbfNew.bBaseType = tbfOld.bBaseType;
      tbfNew.bRecDeleted = tbfOld.bRecDeleted;
      tbfNew.szBasePath = "";
}

int LBSC_ControlFile::copiaBase( char * szOld, const char * szNew)
{
      return ( MoveFile(szOld, szNew) != 0 ? LBS_OK : LBSE_ERROR);
}

int LBSC_ControlFile::GetName(char *szFilePath, char *szFileName)
{
      PRINTLOG( _clLBSLog, ("LBSC_ControlFile::GetName") );
      (IniFile::getInstance())->GetPrivateProfileString( LBSINILBSSECTION,
DIR_BASE_KEY, LBSINIDEFMSG, szFilePath, FULLNAMESIZE, LBSINIFILE );
      (IniFile::getInstance())->GetPrivateProfileString( LBSINILBSSECTION,
LBSINIBASESFILENAME, LBSINIDEFMSG, szFileName, FILENAMESIZE, LBSINIFILE );

      if( ( strcmp( LBSINIDEFMSG, szFilePath ) == 0 ) ||
          ( strcmp( LBSINIDEFMSG, szFileName ) == 0 ) ){
            SetLastError( (DWORD) (LBSE_LBSINIERROR) );
            return( LBSE_LBSINIERROR );
      }
      (SO::getInstance())->putDriveOnPath( szFilePath );
      SetLastError( LBS_OK);
      return( LBS_OK );
}

int LBSC_ControlFile::GetControlFileVersion( char *szFileName )
{
      // se o arquivo nao existir, entao retornar versao OK
      if( !C_File::Exist( szFileName ) ){
            return( CURRENT_CTRLFILE_MAGICNUMBER );
      }

      C_File cfBasesFile( szFileName, "rb", SH_DENYRW, TRUE );
      if( !cfBasesFile.IsOpen() ){
            return( LBSE_NOTOPEN );
      }
      cfBasesFile.Seek( 0, SEEK_SET );

      BYTE   bMagicNum = 0;

      cfBasesFile.Read( &bMagicNum, sizeof( bMagicNum ) );

      return( (int) bMagicNum );
}

int LBSC_ControlFile::InsertBases( C_RecordFile &cfBasesFile, char *szSourceDir )
{
      int iRet;
      if ( !critic ){
```

```
                return LBSE_ERROR;
        }
        critic->EnterCriticalSection();
        iRet = PutDirInFile( cfBasesFile, szSourceDir, UDBNIVEL );
        critic->LeaveCriticalSection();

        return iRet;
}

int LBSC_ControlFile::PutDirInFile( C_RecordFile &cfBasesFile, char *szSourceDir,
int nivel )
{
        if( !szSourceDir || !cfBasesFile.IsOpen() ){
                return( LBSE_BADARG );
        }

        // vamos procurar as bases no diretorio
        char            szPath [MAXPATH ];
        strcpy( szPath, szSourceDir );
        File * Newdir = (SO::getInstance())->getFile( szPath );

        if( Newdir == INVALID_HANDLE_VALUE ||!Newdir->isDirectory() ){
                return( LBSE_ERROR );
        }

        for ( int i = 0; i < Newdir->elements(); i++ ){
                File * file = Newdir->getElement( i );
                if ( file != NULL && file->isDirectory() ){
                        char    szDrive[ MAXDRIVE ];
                        char    szDir[ MAXDIR ];
                        char    szName[ MAXFILE ];
                        char    szExt[ MAXEXT ];
                        char    szBaseDir[ FILENAMESIZE ];
                        char    udb[ MAXDIR ];
                        char    LB1Name[ FILENAMESIZE ];
                        BYTE bBaseType;

                        LBSC_Base::GetDirBase( szBaseDir );
                        _splitpath(szSourceDir,szDrive,szDir,szName,szExt);

                        switch ( nivel ){
                        case COMPONENTENIVEL:
                                (SO::getInstance())->mountPath(szFullBaseName, szName,
"");
                                break;
                        case UDBNIVEL:
                        case BASENIVEL:
                                strcpy(szFullBaseName, file->getFileName());
                                break;
                        }

                        sprintf( LB1Name, "%s%c%s%c%s%s", szSourceDir,
(SO::getInstance())->getPathSeparator(),file->getFileName(), (SO::getInstance())-
>getPathSeparator(),file->getFileName(), CONTROLFILEEXT );
                        C_LB1 clb1;
                        clb1.Open( LB1Name, "r" );
                        if ( clb1.IsOpen() ){
                                TControlRecHead tcRHead;
                                clb1.R_ReadHead( &tcRHead );
                                strcpy( udb, tcRHead.szUserBase );
                                bBaseType = tcRHead.bBaseType;
                        }

                        // vamos inserir o nome no arquivo de controle.
                        TBasesFile    tbfAux;
                        tbfAux.szBaseName = szFullBaseName;
                        strupr( (char*) tbfAux.szBaseName );
                        tbfAux.szBaseLongName = (char*) tbfAux.szBaseName;
```

```
                    tbfAux.szUserBaseName = udb;
                    strupr( (char*) tbfAux.szUserBaseName );
                    tbfAux.bRecDeleted = FALSE;
                    tbfAux.bBaseType = bBaseType;
                    cfBasesFile.R_Append( &tbfAux );

                    (SO::getInstance())->mountPath( szSourceName, szSourceDir,
file->getFileName() );
                    PutDirInFile(cfBasesFile,szSourceName,nivel + 1 );
            }
    }

    FindClose( Newdir );
    return( LBS_OK );
}

int LBSC_ControlFile::estaIncorporada( TBasesFile &tbfAux, const char *szBase,
const char *szUdb )
{
    long iRet = findEntry( tbfAux, szBase, szUdb, NULL );
    if ( iRet == LBS_OK && tbfAux.bRecDeleted == TRUE ){
            iRet = LBSE_BASENOTFOUND;
    }
    return (int) iRet;
}

int LBSC_ControlFile::findEntry( TBasesFile &tbfAux, const char *szBase, const char
*szUdb, long * pos )
{
    int             iRet;
    char                    szFullAux[ FULLNAMESIZE ];
    char                    szAux[ FILENAMESIZE ];

    if( GetName( szFullAux, szAux ) != LBS_OK ){
            return( LBSE_LBSINIERROR );
    }

    char            szFNameAux[ FULLNAMESIZE ];

    (SO::getInstance())->mountPath(szFNameAux, szFullAux, szAux);

    if ( !critic ){
            return LBSE_ERROR;
    }

    critic->EnterCriticalSection();

    if(GetControlFileVersion(szFNameAux)!=CURRENT_CTRLFILE_MAGICNUMBER){
            iRet = LBSE_BADCTRLFILE;
            critic->LeaveCriticalSection();
            return iRet;
    }

    C_RecordFile cfBasesFile(szFNameAux, CURRENT_CTRLFILE_MAGICNUMBER,0,sizeof(
TBasesFile ), "HEADKEY", "RECKEY", "r", SH_DENYRW, TRUE );
    if( !cfBasesFile.IsOpen() ){
            // probelmas na abertura do arquivo
            iRet = LBSE_NOBASESFILE;
            critic->LeaveCriticalSection();
            return iRet;
    }
    cfBasesFile.R_Seek( 0 );

    while( cfBasesFile.R_Read( &tbfAux ) == OK ){
            // Pesquisar se existe a BASE
            if(stricmp((char*)tbfAux.szBaseName,(char*)szBase)==0 &&
stricmp((char*)tbfAux.szUserBaseName, szUdb) == 0){
                    iRet = LBS_OK;
```

```
                    if ( pos ){
                          *pos = cfBasesFile.R_CurPos();
                    }
                    break;
              }
              if( cfBasesFile.R_SeekNext() != OK ){
                    iRet = LBSE_BASENOTFOUND;
                    break;
              }
        }
        cfBasesFile.Close();
        critic->LeaveCriticalSection();
        return iRet;
}

int LBSC_ControlFile::addEntry( TBasesFile &tbfAux, long pos )
{
        int iRet = LBS_OK;
        char                szFullAux[ FULLNAMESIZE ];
        char                szAux[ FILENAMESIZE ];

        if( GetName( szFullAux, szAux ) != LBS_OK ){
              return( LBSE_LBSINIERROR );
        }

        char          szFNameAux[ FULLNAMESIZE ];

        (SO::getInstance())->mountPath(szFNameAux, szFullAux, szAux);
        if(GetControlFileVersion(szFNameAux)!=CURRENT_CTRLFILE_MAGICNUMBER){
              return LBSE_BADCTRLFILE;
        }

        C_RecordFile cfBasesFile( szFNameAux, CURRENT_CTRLFILE_MAGICNUMBER, 0,
sizeof( TBasesFile ), "HEADKEY", "RECKEY", NULL, SH_DENYRW, TRUE );
        if( !cfBasesFile.IsOpen() ){
              return LBSE_NOBASESFILE;
        }
        cfBasesFile.R_Seek( 0 );
        if( pos == -1 ){
              cfBasesFile.R_Append( &tbfAux );
        } else {
              cfBasesFile.R_Seek( pos );
              cfBasesFile.R_Write( &tbfAux );
        }
        cfBasesFile.Close();
        return iRet;
}

int
LBSC_ControlFile:: RemoveBase( const char *szBaseName, const char * szUDBName )
{

        // finalmente, vamos remover a base do arquivo de controle
        TBasesFile    tbfAux;
        char          szFNameAux[ FULLNAMESIZE ];
        char          szFullAux[ FULLNAMESIZE ];
        char          szAux[ FILENAMESIZE ];
        long lPos = -1;

        int iRet = findEntry( tbfAux, szBaseName, szUDBName, &lPos );

        if ( iRet != LBS_OK ){
              return ( iRet );
        }

        if ( tbfAux.bRecDeleted == TRUE ){
              return ( LBS_OK );
        }
```

```
        tbfAux.bRecDeleted = TRUE;
        addEntry ( tbfAux, lPos );
        return( LBS_OK );
}
```

Alguns métodos foram adicionados à interface identificada previamente devido à implementação utilizada. A versão final da classe `LBSC_ControlFile` está apresentada a seguir.

```
class LBSC_ControlFile
{
private:
        CriticSect * critic;
        static LBSC_ControlFile * pclCtrlFile;
protected:
        int copiaBase( const char * szOld, const char * szNew);
        void copia( TBasesFile & tbfNew, OldTBasesFile tbfOld);
        int PutDirInFile( C_RecordFile &cfBasesFile, char *szSourceDir, int nivel );
        int InsertBases( C_RecordFile &cfBasesFile, char *szSourceDir );
        LBSC_ControlFile( void );

public:
        int GetControlFileVersion( char *szFileName );
        int ConvertControlFileTo(int iVersion);
        int estaIncorporada( TBasesFile &tbfAux, char *szBase, char *szUdb );
        int Rebuild( );
        int RemoveBase( char *szBaseName, const char * szUDBName );
        int findEntry( TBasesFile &tbfAux, char *Base, char *Udb, long *pos);
        int addEntry( TBasesFile &tbfAux, long pos );

        static int GetName( char *szFilePath, char *szFileName );
        static LBSC_ControlFile * getInstance( void );

        ~LBSC_ControlFile( void );
};
```

Para concluir esta etapa de implementação da classe, deve-se construir testes de unidade para a classe e executá-los corrigindo os problemas se e quando surgirem.

Na etapa seguinte do refatoramento, é necessário modificar os trechos de código que fazem referência diretamente ao arquivo para que utilizem a nova classe criada. A seguir um exemplo.

```
…
        // finalmente, vamos remover a base do arquivo de controle.
        TBasesFile     tbfAux;
        char           szFNameAux[ FULLNAMESIZE ];
        char           szFullAux[ FULLNAMESIZE ];
        char           szAux[ FILENAMESIZE ];

        // Acessar lbs.ini para pegar o arquivo de controle
        if( GetFNameOnProfile( szFullAux, szAux ) != LBS_OK ){
               ERETURN( LBSE_LBSINIERROR );
        }
        sprintf( szFNameAux, "%s%s%s", szFullAux, "\\", szAux );

        C_SessCritSect       cCS4( this, CRITSECT4 );

        if(GetControlFileVersion(szFNameAux) != CRYPTO_MAGICNUMBER){
               ERETURN( LBSE_BADCTRLFILE );
        }
```

```
	C_RecordFile cfBasesFile( szFNameAux, CRYPTO_MAGICNUMBER, 0, sizeof(
TBasesFile ), "HEADKEY", "RECKEY", NULL, SH_DENYRW, TRUE);
	if( !cfBasesFile.IsOpen() ){
		ERETURN( LBSE_NOBASESFILE );
	}
	cfBasesFile.R_Seek( 0 );
	do{
		if( cfBasesFile.R_Read( &tbfAux ) != OK ){
			break;
		}
		if( stricmp( tbfAux.szBaseName, szBaseName ) == 0 ){
			break;
		}
	} while( cfBasesFile.R_SeekNext() == OK );
	if( stricmp( tbfAux.szBaseName, szBaseName ) == 0 ){
		tbfAux.bRecDeleted = TRUE;
		cfBasesFile.R_Write( &tbfAux );
	}
	cfBasesFile.Close();
…
```

O trecho de código apresentado elimina uma entrada no arquivo de controle, ou seja, uma base. Deve-se modificá-lo para que utilize a classe `LBSC_ControlFile` e não mais faça acesso direto ao arquivo. O código após as alterações está mostrado a seguir.

```
int iRet = pclCtrlFile->RemoveBase( szBaseName, strUDBLogged );
```

O próximo método apresentado, pertencente à classe `Base`, também sofreu alterações após a introdução da classe `LBSC_ControlFile`.

```
int Base::ModifyBaseName( char *szNewBaseName )
{
	TBasesFile	tbfAux;
	char		szAux[ FILENAMESIZE ];
	char		szFullAux[ FULLNAMESIZE ];
	char		szFNameAux[ FULLNAMESIZE ];

	// pegar o dir_base
	if(Sessao::GetFNameOnProfile( szFullAux, szAux ) != LBS_OK ){
		ERETURN( LBSE_LBSINIERROR );
	}

	(SO::getInstance())->mountPath( szFNameAux, szFullAux, szAux);

	C_SessCritSect		cCS4( plbscsOwnerSession, CRITSECT4 );

	// verificar versao do arquivo de controle
	if(GetControlFileVersion(szFNameAux) != CRYPTO_MAGICNUMBER){
		ERETURN( LBSE_BADCTRLFILE );
	}
	C_RecordFile cfBasesFile( szFNameAux, CRYPTO_MAGICNUMBER, 0, sizeof(
TBasesFile ), "HEADKEY", "RECKEY", NULL, SH_DENYRW, TRUE );
	if( !cfBasesFile.IsOpen() ){
		ERETURN( LBSE_NOBASESFILE );
	}
	cfBasesFile.R_Seek( 0 );
	do{
		if( cfBasesFile.R_Read( &tbfAux ) != OK ){
			break;
		}
		if( stricmp(tbfAux.szBaseName, szBaseName ) == 0 ){
			break;
```

```
                }
        } while( cfBasesFile.R_SeekNext() == OK );
        if( stricmp( tbfAux.szBaseName, (char*) szBaseName ) == 0 ){
                // achamos a base
                tbfAux.szBaseLongName = szNewBaseName;
                cfBasesFile.R_Write( &tbfAux );
                ERETURN( LBS_OK );
        }
        ERETURN( LBSE_ERROR );
}
```

Após o refatoramento, o método foi modificado para o seguinte.

```
int Base::ModifyBaseName( char *szNewBaseName )
{
        TBasesFile    tbfAux;
        char * udbname = plbscsOwnerSession->WhatUDBLogged();
        long lBFilePos = -1;
        C_SessCritSect       cCS4( plbscsOwnerSession, CRITSECT4 );
        LBSC_ControlFile* pclCtrF = LBSC_ControlFile::getInstance();
        if (pclCtrF == NULL ){
                ERETURN ( LBSE_NOMEMORY );
        }
        int iRet =pclCtrF->findEntry(tbfAux, szBase, udbname, &lBFP);
        if ( udbname ){
                delete udbname;
        }
        if ( iRet != LBS_OK ){
                ERETURN( iRet );
        }
        tbfAux.szBaseLongName = szNewBaseName;
        pclCtrF ->addEntry( tbfAux, lBFilePos );
        ERETURN( LBS_OK );
}
```

Com a adequação de todo o código à nova abstração, o sistema deve ser compilado, gerado e testado por inteiro.

7.2 Separação de persistência

7.2.1 Fundamentos

Padrão **Bridge**

Promove a diminuição do acoplamento entre uma abstração e sua implementação, assim os dois podem variar independentemente [Gamma, 1994]. Quando uma abstração pode ser implementada de várias maneiras diferentes é comum usar herança para acomodar todas as implementações. No entanto, esta solução promove um forte acoplamento entre a abstração e a implementação, o que torna a solução difícil de manter, estender e reusar.

O padrão, ilustrado na Figura 7.1 *Bridge* [Gamma, 1994] se aplica a este tipo de problema e propõe que uma abstração e sua implementação sejam colocadas em hierarquias de classes diferentes [Gamma,1994].

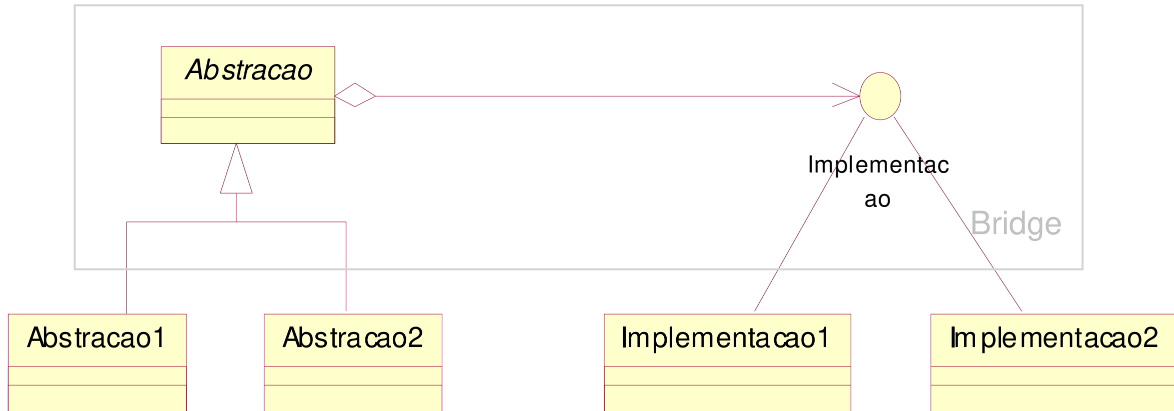


Figura 7.1 - Ilustração do padrão *Bridge*.

Todas as operações das subclasses de `Abstracao` consistem na invocação de métodos abstratos do TAD `Implementacao`.

7.2.2 Sumário

Existem objetos do domínio de um problema que precisam manter algum tipo de persistência. Em alguns casos, porém, estes conceitos não estão separados na modelagem do sistema. Este refatoramento separa a parte persistente de uma classe daquilo que não é. A Figura 7.2 apresenta um resumo do refatoramento.

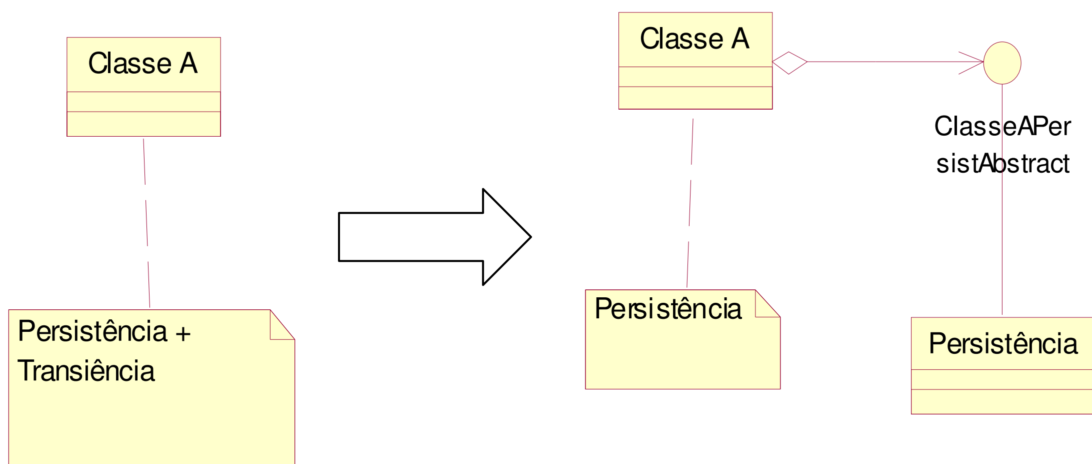


Figura 7.2 - Ilustração do refatoramento *Separação de persistência*.

7.2.3 Condições de aplicação

Alguns conceitos do domínio de um dado problema devem manter a persistência entre execuções do sistema, seja esta persistência feita em arquivos, bancos de dados, ou qualquer outro tipo de repositório de informações. Porém, muito freqüentemente encontra-se esta situação modelada por uma abstração que engloba tanto a implementação da lógica e do comportamento transiente do objeto, como a forma de persistência da abstração.

Esta modelagem pode causar problemas quanto à manutenção do sistema, por favorecer o acoplamento entre a implementação do comportamento e as informações persistentes.

A situação anterior pode revelar uma modelagem pouco extensível e que não favorece o reuso da classe por não permitir que existam objetos do mesmo tipo, que tenham o mesmo comportamento, mas com a implementação da persistência diferente. Por exemplo, duas instâncias da mesma classe, onde uma delas tem sua persistência em arquivo e outra em um banco e dados.

Assim, este refatoramento é aplicável principalmente nos casos em que se deseja evitar uma ligação permanente entre uma abstração e o mecanismo de persistência; nos casos em que a abstração e o mecanismo de persistência possam ser extensíveis através de herança; onde mudanças na implementação da persistência não sejam refletidas, ou mesmo conhecidas, pelo código cliente da abstração.

7.2.4 Mecanismo

1 Identificar uma classe que engloba tanto a abstração quanto a persistência de um conceito.

2 Definir e criar os TADs que representarão a persistência da abstração.

3 Criar uma subclasse do TAD de persistência e adicionar a esta classe a implementação dos métodos.

    3.1 Esta será a classe que realmente implementará a persistência. Ela irá fornecer a persistência de um conceito através da interface do TAD definido anteriormente.

4 Planejar, projetar, implementar e executar testes que exercitem a nova classe.

5 Fazer com que a classe antiga referencie o TAD de persistência sempre que precisar das funcionalidades persistentes de sua abstração.

6 Adicionar à classe antiga uma referência ao TAD de persistência.

7 Compilar e testar o sistema.

7.2.5 Conseqüências

O desacoplamento entre a lógica de uma abstração e sua persistência é o principal ganho obtido com a aplicação deste refatoramento. É possível substituir a estratégia de persistência utilizada e manter o código da lógica da abstração inalterado.

Ainda com esta estrutura é possível estender tanto a lógica da abstração, como a persistência independentemente, sem que uma interfira na outra. Caso uma nova informação seja adicionada, as duas partes da abstração devem mudar, porém a separação permite uma atualização incremental, em etapas [Bass, 1998].

Ainda sobre as vantagens do uso desta modelagem, é importante citar o fato de que ela possibilita que classes diferentes utilizem a mesma classe para implementar sua persistência e que instâncias da mesma classe possam implementar sua persistência de forma diferente e ainda, serem tratadas de maneira idêntica pelo código cliente, sem distinção.

7.2.6 Exemplo

Passo 1

A situação usada para ilustrar a aplicação deste refatoramento consiste de uma classe chamada `Base` que é a abstração de um conceito persistente no domínio do sistema. Esta classe representa um repositório de informações. Cada `Base` criada é armazenada em arquivos com funções e estruturas definidas. A parte persistente e a parte transiente de uma base estão representadas em conjunto pela classe `Base`. Caso fosse necessário modificar a estratégia de persistência da classe, a parte transiente sofreria alterações já que ambas estão implementadas na classe `Base`, parte dela está listada a seguir.

```
class Base {

private:
BOOL                       bHeaderLocked;
LBSC_IndexSystem    *plbscisIndexSystem;
int                        iLockTimeOut;
long                       lLockPos;
BOOL                       bBaseObjOk;
BOOL                       bStruct;
BOOL                       bSelfDelete;
BOOL                       bFullAccess;
BOOL                       bSelfReorganize;
CL_StringN <FILENAMESIZE>  szBaseName;
CL_StringN <FULLNAMESIZE>  szBaseLongName;
CL_StringN <PATHNAMESIZE>  szBasePath;
CL_StringN <FULLNAMESIZE>  szCompleteBaseName;
C_Date                            cdCreateDate;
C_Date                            cdLastModifyDate;
LBSC_Session                      *plbscsOwnerSession;
C_LB1                             cfBaseControlFile;
C_LB3                             cfBaseStructFile;
```

```
C_LB4                          cfBaseRepetitionFile;
void                                   *pvUserData;
int                                    iNavigationState;
int                                    iSearchType;

int                                    iIndexTree;
BOOL                                   bStopIndex;

BOOL   BaseStruct();
int    Delete();
int    RenameFile( C_File *, C_File *, char *, char * );
int    OpenAllFiles( BOOL bCheckVersion = TRUE );
int    CloseAllFiles();
int    RemoveDir( char *, BOOL );
int    UpdateStructFile();
int    DelFieldOnFile( UINT );
int    AppendLastFieldOnFile();

// Metodos para indexacao/desindexacao
int    Index( const LBSC_Ticket *, int, BOOL = FALSE );
int    Unindex( const LBSC_Ticket *, int iType );
int    ChangeIndex( const LBSC_Ticket *, int, BOOL, BOOL ;
int    IndexLT( LBSC_IndexSystem *, int, long, UINT, int, LTC_PARSER * );
int    IndexLTGoWord( LBSC_IndexSystem *, int, long, UINT, int, LTC_PARSER * );
int    UnindexLT( LBSC_IndexSystem *, int, long, UINT, int, LTC_PARSER * );
int    WriteRecordInLogFile( const LBSC_Ticket * );
int    SaveIndexStatus( const LBSC_Ticket * );
int    LoadIndexStatus( const LBSC_Ticket * );
// Metodos de importacao/exportacao
int    ExportLB3( C_File * );
int    ExportLB1( const LBSC_Ticket *, C_File *, BOOL );
int    Export( const LBSC_Ticket *, C_File *, BOOL );

// obtem a versao do arquivo LB1 de uma base
static int    GetBaseVersion( char * );


// metodos para tratamento de lock
int    MakeLockFileName( char *, BOOL = TRUE );
BOOL   ExistLockFile( BOOL = TRUE );
int    WriteOnLockFile( TLockInfo *, BOOL = TRUE );
int    ReadFromLockFile( TLockInfo *, BOOL = TRUE );

// tratamento de lock do header de LB1
int    LockLB1Header();
int    ReleaseLB1Header();

// metodo para liberar espaco em LB2 e LB4
int    FreeSpaceInLB24( long, BOOL );

// deleta os arquivos do sistema de indices
int    RemoveIndexSystem( char * = NULL );

// metodos para verificacao de consistencia do header de LB1
int    CheckLB1Header();
int    GetLB1BakStruct( TControlRecHead * );
int    PutLB1BakStruct( TControlRecHead * );

// metodo para copia um C_File para outro
int    CopyFile( C_File *, C_File * );

public:
// Construtores e destrutores
                    LBSC_Base( );
                    ~LBSC_Base();

LBSC_Base     *Duplicate( void );
int    ModifyBaseOwnerName( LBSC_Ticket *, char *, char * );
```

```
int    ModifyPassword( const LBSC_Ticket *, char *, char * );
int    ModifyRecordPassword(LBSC_Ticket *, char *, char * );
int    BlockReExport( const LBSC_Ticket * );
BOOL   IsExportable();
char   *GetUserBase();
char   *GetOwnerName();
int    GetFullBaseName( char * );
int    GetBaseName( char * );
int    GetBaseLongName( char * );
BOOL   IsEncrypt();
BOOL   IsFullAccess();
BYTE   GetBaseType();
int    ModifyBaseType( BYTE );
int    ModifyBaseName( char *szNewBaseName );
BOOL   IndexSystemOk();
BOOL   IsExclusive();
BOOL   IsReadOnly();
TBaseInfo           *GetBaseInfo( const LBSC_Ticket * );
int    ClearBase( const LBSC_Ticket * );
LBSC_Session  *GetOwnerSession();
int    GetBaseTimeInfo( struct stat * );

...

};
```

Esta não é a classe completa; no entanto, é suficiente para perceber que ela engloba tanto o comportamento persistente como o transiente. O tratamento dos arquivos está descrito e caracterizado nos métodos privados da classe (`CopyFile`, `CheckLB1Header`, `GetLB1BakStruct`, `PutLB1BakStruct`, por exemplo). Os métodos públicos caracterizam o comportamento transiente relativo ao estado do objeto depois de criado.

Passos 2 e 3

Para definir um TAD que represente a persistência da abstração, é preciso observar como a persistência é acessada dentro da classe. Os métodos que permitem acesso à persistência da classe são, em sua maioria, métodos privados, deste modo, podem ser modificados ou excluídos sem que o código cliente da classe seja atingido.

Ao tentar extrair uma interface para a persistência da classe `Base`, percebe-se que uma única interface não é adequada. Uma base é formada por registros. Estes registros são compostos de campos, que por sua vez podem ter várias repetições, ou seja, valores.

Assim, uma única interface para responder pela persistência de todos estes objetos pode não representar uma boa escolha. É possível que seja criada uma classe excessivamente complexa. E também, com uma classe modelando cada conceito, adiciona-se a possibilidade de que elas variem independentemente.

Para representar a persistência da classe `Base` optou-se por utilizar o padrão *Composite* [Gamma, 1994]. A Figura 7.3 representa o diagrama de classes para a persistência da classe `Base`.

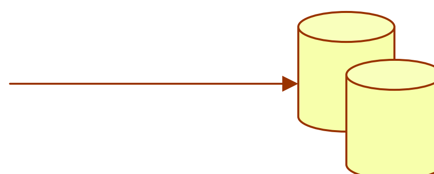


Figura 7.3 - Estratégia de persistência adotada pela classe Base antes do refatoramento.

Na Figura 7.3, percebe-se que a classe `Base` acessa a classe `C_File` que acessa o disco para recuperar as informações de que necessita. No entanto, a classe `C_File` nada mais é do que uma interface de mais alto nível de acesso a arquivos. Deste modo, a classe `Base` sabe em quais arquivos estão as informações necessárias ou como os arquivos estão organizados. O fato de utilizar a classe `C_File` significa apenas que `Base` não está acoplada às rotinas de acesso a arquivos do sistema operacional ou da linguagem.

A Figura 7.4 representa a estratégia de persistência que será adotada pela classe `Base` para separar a sua persistência de sua lógica.

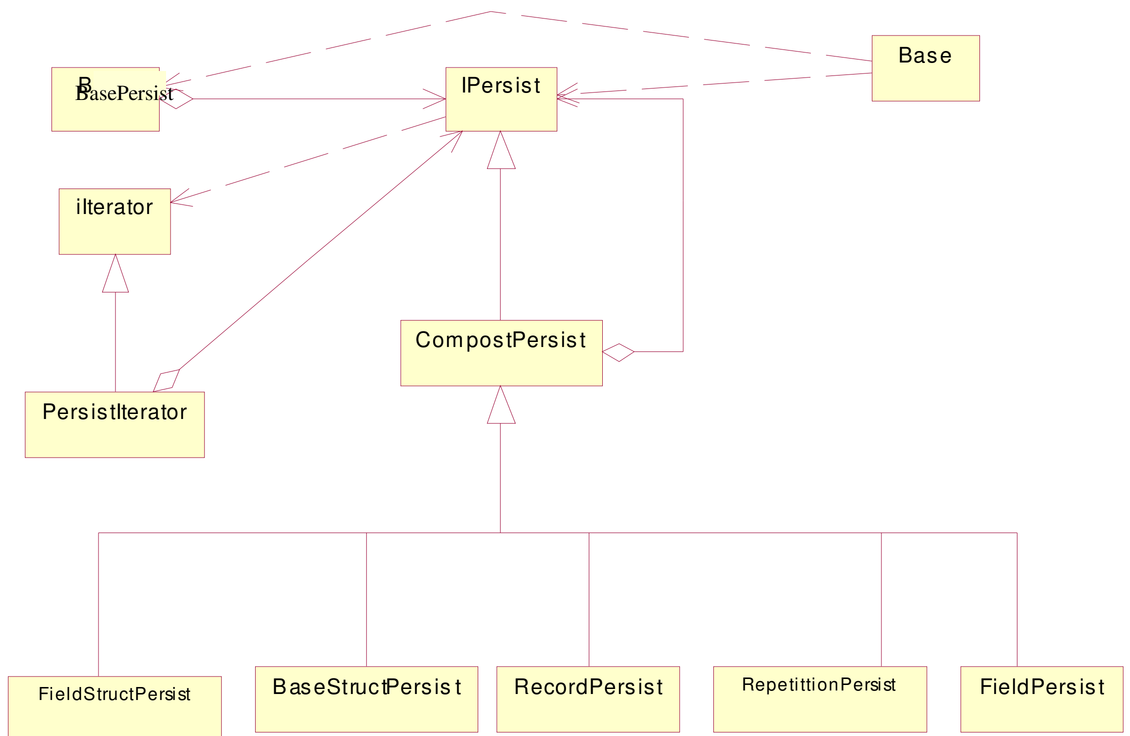


Figura 7.4 - Estratégia de persistência utilizada pela classe Base após a aplicação do refatoramento.

```
class iIterator {
public:
	virtual int getNumElements();
	virtual void* getFirst();
	virtual void* getNext();
	virtual void* getPrevious();
	virtual void* getLast();
	virtual void* getInPos( int iPos );
```

```
        virtual int add( void* newRec );
        virtual int del( void* newRec );
};

class IPersist{
public:
        IPersist();
        virtual ~IPersist();
        virtual iIterator* getLeaves();
        virtual int getState();
        virtual int save();
        virtual int modifyAtribute( const char *szAtribute, const char *szOldValue,
const char* szNewValue );
        virtual const char* getAttribute( const char * szAttribute );
};

class PersistIterator : public iIterator{
private:
        IPersist* elements;
public:
        PersistIterator(IPersist * list);
        ~PersistIterator();
        int getNumElements();
        IPersist* getFirst();
        IPersist* getNext();
        IPersist* getPrevious();
        IPersist* getLast();
        IPersist* getInPos( int iPos );
        int add( IPersist* newRec );
        int del( IPersist* newRec );
        int save();
};
class CompostPersist : public IPersist {
private:
        IPersist* elements;
public:
        CompostPersist (IPersist * list);
        ~CompostPersist ();
        PersistIterator* getLeaves();
        virtual int getState();
        virtual int save();
        virtual int modifyAtribute( const char *szAtribute, const char *szOldValue,
const char* szNewValue );
        virtual const char* getAttribute(const char* szAttribute);
};

class BaseStructPersist : CompostPersist{
private:
        IPersist* pFields;
public:
        BaseStructPersist ();
        ~ BaseStructPersist ();
        PersistIterator* getFields();
        int getStruct();
        int saveStruct();
        int modifyAtribute( const char *szAtribute, const char *szOldValue, const
char* szNewValue );
        const char* getAttribute( const char * szAttribute );
};

class BasePersist{
private:
        IPersist* pRecords;
public:
        BasePersist();
        ~BasePersist();
        int Initialize();
        int Finish();
```

```
        int Export();
        int Recover();
        int save();
        PersistIterator* getRecords();
};

class RecordPersist : public CompostPersist {
private:
        IPersist* pFields;
public:
        RecordPersist();
        ~RecordPersist();
        PersistIterator* getLeaves();
        int getState();
        int save();
        int modifyAtribute( const char *szAtribute, const char *szOldValue, const
char* szNewValue );
        const char* getAttribute( const char * szAttribute );
};

class FieldStructPersist : public CompostPersist {
private:
        IPersist* pChildren;
public:
        FieldStructPersist();
        ~FieldStructPersist();
        PersistIterator* getLeaves();
        int save();
        int getState();
        int modifyAtribute( const char *szAtribute, const char *szOldValue, const
char* szNewValue );
        const char* getAttribute( const char * szAttribute );
};

class FieldPersist : public CompostPersist {
private:
        IPersist* pRepetittions;
public:
        FieldPersist();
        ~FieldPersist();
        PersistIterator* getLeaves();
        int save();
        int getState();
        int modifyAtribute( const char *szAtribute, const char *szOldValue, const
char* szNewValue );
        const char* getAttribute( const char * szAttribute );
};

class RepetittionPersist : public CompostPersist {
public:
        RepetittionPersist();
        ~ RepetittionPersist();
        PersistIterator* getLeaves();
        void * getValue();
        int save();
        int getState();
        int modifyAtribute( const char *szAtribute, const char *szOldValue, const
char* szNewValue );
        const char * getAttribute( const char * szAttribute );
};
```

Passos 4, 5, 6 e 7

Após a implementação das classes que compõem o esquema de persistência, testes de unidade devem ser construídos de modo a testar as classes implementadas. Com isto

terminado, deve-se passar a próxima etapa que é o refatoramento das classes do sistema para fazê-las referenciar a nova estrutura de persistência.

O método apresentado a seguir é um exemplo de código que faz referência à antiga estrutura de persistência implementada pelo sistema.

```
int Base::ReadRecord( const LBSC_Ticket *lbscTicket )
{

…

if( LB1.tcrHead.lNumRecords == 0 ){
	// reler o header de LB1
	if( LB1.R_ReadHead() == OK ){
		if( LB1.tcrHead.lNumRecords == 0 ){
			ERETURN( LBSE_NORECORD );
		}
		ERETURN( FirstFRecord( lbscTicket ) );
	}
	ERETURN( LBSE_HEADERROR );
}
if( LB1.R_Read( &tcrControlRec ) != OK ){
	if( LB1.R_Read( &tcrControlRec, sizeof( unsigned long ), sizeof(
tcrControlRec ) - sizeof( unsigned long ) ) != OK ){
		ERETURN( LBSE_RECORDLOCKED );
	}
	tcrControlRec.ulCreateDate = 0;
} else {
	ERETURN( LBSE_NORECORD );
}

lbscrCurrRecord.bStatus = tcrControlRec.bRecStatus;
if( tcrControlRec.bRecStatus & REC_EXCLUDED ){
	ERETURN( LBSE_EXCLUDEDREC );
}
if( !tcrControlRec.lContentPos ){
	ERETURN( LBSE_ZERORECORD );
}

lbscrCurrRecord.szOwnerName = tcrControlRec.szOwnerName;
lbscrCurrRecord.szLastUpdateUserName = tcrControlRec.szLastModifyUserName;
lbscrCurrRecord.SetCreateDate( tcrControlRec.ulCreateDate );
lbscrCurrRecord.SetModifyDate( tcrControlRec.ulModifyDate );
lbscrCurrRecord.SetStatus( tcrControlRec.bRecStatus );
ClearRecord();
lbscrCurrRecord.ulAccess = tcrControlRec.ulAccess;

if( !LB3.IsOpen() ){
	ERETURN( LBSE_BASENOTOPEN );
}
if ( tcrContentRecPsw.iNrFields > LB3.R_FileSize() ){
	ERETURN( LBSE_WRONGRECORD );
}
for( int i = 0; i < tcrContentRecPsw.iNrFields; i++ ){
if( LB4.Seek( tcrControlRec.lContentPos + sizeof( TContentRecPsw ) + (i * sizeof(
TContentRec )), SEEK_SET ) != OK ){
		ERETURN( LBSE_WRONGRECORD );
	}
	if(LB4.Read(&tcrContentRec, sizeof(tcrContentRec)) != OK){
		ERETURN( LBSE_WRONGRECORD );
	}
	if(lbscrCurrRecord.Id(tcrContentRec.uiFieldId) == NULL){
		continue;
	}
	LBSC_Field *pcfField = lbscrCurrRecord( tcrContentRec.uiFieldId );
	if( pcfField ){
```

```
              pcfField->lRepetFilePos = tcrContentRec.lFirstRepetition;
              int iNumRep = tcrContentRec.iNumberOfRepetition;
              long lSpace = tcrContentRec.lFirstRepetition;
              while( lSpace && iNumRep ){
                     if( LB4.Seek( lSpace, SEEK_SET ) != OK ){
                            ERETURN( LBSE_WRONGRECORD );
                     }
                            if( LB4.Read( &trrRepetitionRec,
                                   sizeof( trrRepetitionRec ) ) != OK ){
                                   ERETURN( LBSE_WRONGRECORD );
                            }
                            BOOL bRepExist = TRUE;
                            BOOL bDataHole = FALSE;
                            if( trrRepetitionRec.lRepetitionSize == 0
||trrRepetitionRec.lRepetitionSize == DATA_HOLE_SIZE ){
                                   bDataHole = (trrRepetitionRec.lRepetitionSize ==
DATA_HOLE_SIZE);
                                   trrRepetitionRec.lRepetitionSize=1;
                                   bRepExist = FALSE;
                            }
...

       LBSC_Data     *pd = (*pcfField) [ 0 ];
if( pd ){
     switch( pcfField->GetType() ){
              case TEXT_FIELD:
              pd->ModifyData( szBuffer, trrRepetitionRec.lRepetitionSize );
              break;
              case BINARY_FIELD:
              if( !bRepExist ){
              char   szBin[ 2 * sizeof( int ) ];
                     memset( szBin, 0, 2 * sizeof( int ) );
                     pd->ModifyData( szBin );
              } else {
                     pd->ModifyData( szBuffer );
              }
              break;
              case REFERENCED_FIELD:
              if( !bRepExist ){
                     char   szBin[ sizeof( LBSC_Reference ) ];
                     memset( szBin, 0, sizeof( LBSC_Reference ) );
                     pd->ModifyData( szBin );
              } else {
              if(trrRepetitionRec.lRepetitionSize < sizeof( LBSC_Reference ) ){
                     char   szBin[ sizeof( LBSC_Reference ) ];
                            memset( szBin, 0, sizeof(LBSC_Reference));
                            memcpy( szBin, szBuffer, trrRepetitionRec.lRepetitionSize
);

                            pd->ModifyData( szBin );
                     } else {
                            pd->ModifyData( szBuffer );
                     }
              }
              break;
              default:
              pd->ModifyData( szBuffer );
       }
       }
…
}
```

Ao examinar o método anterior, percebe-se que o acesso às informações presentes no disco é feito diretamente à estrutura de arquivos. Isto provoca um forte acoplamento entre a lógica do sistema e a forma de armazenamento da informação. Esta é uma característica indesejada uma vez que pode implicar em maior esforço para manter o sistema. Por exemplo,

quando for necessário modificar a estrutura de armazenamento das informações, a lógica da aplicação será atingida, ou seja, esta necessidade causará reflexos em todo o sistema já que a lógica conhece bem a estrutura de armazenamento das informações.

Com a nova estrutura de acesso à informação implementada, deve-se modificar o código anterior de modo que ele referencie a nova estrutura de acesso à informação.

A primeira modificação a ser feita é introduzir uma referência à classe persistente na classe `Base`.

```
BasePersist * pBasePersist;
IPersist*     CurrRecord; //registro corrente
```

O método em estudo, `ReadRecord`, lê o próximo registro da base e o torna o registro corrente, ou seja, qualquer alteração feita em um registro será feito neste lido.

```
int Base::ReadRecord( const LBSC_Ticket *lbscTicket )
{
//validações
…
PersistIterator* pIt = pBasePersist->getRecords();
       CurrRecord = pIt->getNext();
return (CurrRecord != NULL);
}
```

O código que anteriormente estava contido neste método está distribuído no novo conjunto de classes responsáveis pela persistência. No código antigo, todo o tratamento de registros, campos e repetições estava contido no método `ReadRecord`. Ele era responsável por recuperar a informação persistente de todas estas entidades já que elas não estavam representadas por abstrações no sistema. Com o novo esquema, cada um destes conceitos é representado por uma classe que detém a responsabilidade específica de cuidar de sua persistência. Deste modo, a classe `Base`, que consiste do comportamento lógico do conceito "Base de Dados" não mais está acoplada ao esquema de persistência implementado pelo sistema. Caso a estrutura de arquivos utilizados como meio de persistência mude, esta mudança não ultrapassará os limites de `BasePersist`.

O método a seguir é outro exemplo de trecho de código que será modificado para que referencie a nova abstração criada para representar a persistência das informações do sistema. Este método é responsável por apagar o registro corrente da base de dados.

```
int Base::DeleteRecord( const LBSC_Ticket *lbscTicket )
{
       int    iRet;

       if( !LB1.IsOpen() ){
              ERETURN( LBSE_BASENOTOPEN );      // Base fechada
       }
       if( bBaseObjOk == FALSE ){
              ERETURN( LBSE_OBJNOTOK );
       }
       // Checar permissoes de ACL
       if( VerifyPermission( plbscsOwnerSession->GetUserName(), CurrRecNum(
lbscTicket ), USERRECORD, ACL_DEL ) != LBS_OK ){
```

```
            ERETURN( LBSE_NOPERMISSION );
        }
        if( lbscrCurrRecord.IsUndefined() ){
            ERETURN( LBSE_UNDEFINEDRECORD );
        }
        C_BaseCritSect      cCS3( this, CRITSECT3 );
        C_BaseCritSect      cCS5( this, CRITSECT5 );
        C_BaseCritSect      cCS6( this, CRITSECT6 );
        C_BaseCritSect      cCS8( this, CRITSECT8 );
        C_BaseCritSect      cCS9( this, CRITSECT9 );

        // a regiao critica abaixo protege toda a operacao de
        // delecao do registro
        C_BaseCritSect      cCS13( this, CRITSECT13 );

        LBSC_LB1_LockHead   cLB1_LockHead( LB1 );

        if( !cLB1_LockHead.IsLocked() ){
            ERETURN( LBSE_HEADERROR );
        }

        long lCurRecNum = LB1.R_CurPos();
        TControlRec  tcrControlRec;
        if( LB1.R_Read( &tcrControlRec ) != OK ){
            ERETURN( LBSE_ERROR );
        }
        if( tcrControlRec.bRecStatus & REC_EXCLUDED ){
            lbscrCurrRecord.SetIsUndefined( TRUE );
            lbscrCurrRecord.bStatus |= REC_EXCLUDED;
            ERETURN( LBSE_EXCLUDEDREC );
        }
        if( lbscrCurrRecord.Delete( &LB1 ) == LBS_OK ){
            LB1.Flush();
            int     iNumFields = GetNumberOfFields();
            for( int i = 0; i < iNumFields; i ++ ){
                LBSC_Field    *pf = lbscrCurrRecord[ i ];

                if( pf ){
                    pf->UpdateFieldFlag( TRUE );
                }
            }

            FreeSpaceInLB24(tcrControlRec.lContentPos, FALSE );

            Unindex( lbscTicket, PARTIAL_INDEX );

            bBaseUpdated = TRUE;

            --LB1.tcrHead.lNumRecords;
            ++LB1.tcrHead.lDeletedRecNum;

            DelRecFromOcList( lCurRecNum );

            ReleaseRecord( lbscTicket );

            iRet = LBS_OK;

            if( LB1.tcrHead.lNumRecords == 0 ){
                LB1.tcrHead.lFirstRecActivePos = LB1.tcrHead.lLastRecActivePos
= -1;
                lbscrCurrRecord.Clear();
                lbscrCurrRecord.SetIsUndefined( TRUE );
            } else {
                if( LB1.R_CurPos() == LB1.tcrHead.lFirstRecActivePos ){
                    iRet = NextRecord( lbscTicket );
                    while( LB1.R_Seek( ++LB1.tcrHead.lFirstRecActivePos ) ==
LBS_OK ){
                        if(LB1.R_Read(&tcrControlRec)!=OK){
```

```
                                EReturn( LBSE_FATAL );
                        }
                        if( !(tcrControlRec.bRecStatus & REC_EXCLUDED) ){
                                break;
                        }
                }
                if( LB1.tcrHead.lFirstRecActivePos >
LB1.tcrHead.lLastRecActivePos ){
                        LB1.tcrHead.lFirstRecActivePos =
LB1.tcrHead.lLastRecActivePos = -1;
                        lbscrCurrRecord.Clear();
                        lbscrCurrRecord.SetIsUndefined(TRUE);
                }
        } else {
                if( LB1.R_CurPos() == LB1.tcrHead.lLastRecActivePos ){
                        iRet = PreviousRecord( lbscTicket );
                        while(LB1.R_Seek(--LB1.tcrHead.lLastRecActivePos)
== LBS_OK ){
                                if( LB1.R_Read( &tcrControlRec ) != OK ){
                                        EReturn( LBSE_FATAL );
                                }
                                if(!(tcrControlRec.bRecStatus
&REC_EXCLUDED) ){
                                        break;
                                }
                        }
                        if(LB1.tcrHead.lLastRecActivePos == -1){
                                LB1.tcrHead.lFirstRecActivePos =
LB1.tcrHead.lLastRecActivePos = -1;
                                lbscrCurrRecord.Clear();
                                lbscrCurrRecord.SetIsUndefined( TRUE );
                        }
                } else {
                        if( (iRet = NextRecord( lbscTicket )) != LBS_OK ){
                                iRet = PreviousRecord(lbscTicket);
                        }
                }
        }
        if( ( LB1.tcrHead.lDeletedRecNum ==
LB1.tcrHead.lDeletedRecNumReorg ) && bSelfReorganize ){
                plbscsOwnerSession->ReorganizeBase( lbscTicket, this );
        }
    }
    if( pcOLList && !(FISIC_OC_LIST) ){
        LBSC_Expr *pExpr = pcOLList->Current();
        if( pExpr && pExpr->lSlot <= 0 ){
                lbscrCurrRecord.Clear();
                lbscrCurrRecord.SetIsUndefined( TRUE );
                iRet = LBS_OK;
        }
    }
    LB1.tcrHead.ulLastModifyDate = C_Date().Hash();
    LB1.tcrHead.szLastModifyUserName = plbscsOwnerSession->GetUserName();

    if((LB1.tcrHead.bBaseType==BASE_PUBLIC_REC_PRIVATE
)||(LB1.tcrHead.bBaseType == BASE_PRIVATE_REC_PRIVATE ) ){
        char *szName = plbscsOwnerSession->GetUserName();
        int iResult;
        lbscrCurrRecord.szOwnerName = szName;
        iResult = DelACLPerm( szName, lCurRecNum, NULL, USERRECORD );
        iResult = DelACLPerm( szName, lCurRecNum, NULL, GROUPRECORD );
    }

    EReturn( iRet );
  }
  EReturn( LBSE_ERROR );
}
```

Mais uma vez a lógica envolvida em apagar um registro do sistema está acoplada com a remoção física deste registro. A seguir o código do método após o refatoramento para referenciar a nova abstração de persistência.

```
int Base::DeleteRecord( const LBSC_Ticket *lbscTicket )
{
	int	iRet;

	if (!pBasePersist){
		ERETURN( LBSE_BASENOTOPEN );	// Base fechada
	}
	if( bBaseObjOk == FALSE ){
		ERETURN( LBSE_OBJNOTOK );
	}
	// Checar permissoes de ACL
	if( VerifyPermission( plbscsOwnerSession->GetUserName(), CurrRecNum(
lbscTicket ), USERRECORD, ACL_DEL ) != LBS_OK ){
		ERETURN( LBSE_NOPERMISSION );
	}
	if(!CurrRecord){
		ERETURN( LBSE_UNDEFINEDRECORD );
	}

	PersistIterator* pIt = pBasePersist->getRecords();

	iRet = pIt->del ( CurrRecord );

return iRet;
}
```

Assim, todos os trechos de código que acessem diretamente a informação no sistema devem ser modificados de modo a referenciar o novo conjunto de classe que representa a estrutura de persistência. Quando todos os trechos estiverem atualizados, o teste completo do sistema deve ser efetuado.

8 REFATORAMENTOS QUE INTRODUZEM PADRÕES DE PROJETO

Neste capítulo os refatoramentos que introduzem alguns padrões de projeto definidos em [Gamma, 1994] são apresentados. Uma das vantagens do uso de técnicas de refatoramento é a possibilidade de introduzir padrões de projeto mesmo depois de o projeto estar completo. Os refatoramentos apresentados aqui introduzem padrões de projeto em um nível que abrange o envolvimento de mais de uma classe.

8.1 Introdução de *Observer*

8.1.1 Fundamentos

O padrão de projeto chamado *Observer* [Gamma, 1994] define uma dependência um-para-muitos de modo que quando o estado de um objeto muda, todos os interessados são notificados da mudança. Ou seja, os objetos interessados em outro são avisados sobre as mudanças no seu estado.

Os constituintes básicos do padrão são dois: o observador e o observável. Um observável pode ser, e geralmente é, observado por vários observadores. A Figura 8.1 apresenta a estrutura do padrão.

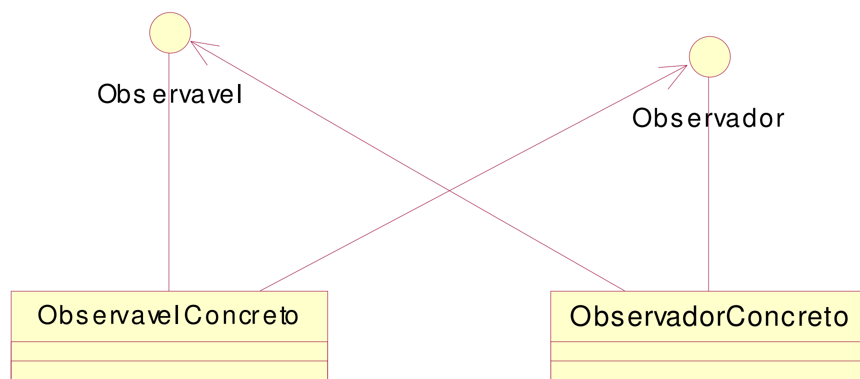


Figura 8.1 - Representação resumida do padrão *Observer*

A grande vantagem no uso deste padrão é o fraco acoplamento entre as classes constituintes da relação. Deste modo, elas podem variar independentemente e não afetarão umas às outras. O padrão é utilizado quando uma mudança a um objeto requer mudanças em

outros, no entanto os objetos que devem ser avisados sobre mudanças não são previamente conhecidos. Ou ainda, quando o objeto deve avisar a outros de suas alterações sem fazer suposições sobre eles.

8.1.2 Sumário

Uma classe necessita saber o progresso de um processamento realizado por um objeto de uma outra classe. Para modelar esta situação, introduz-se o padrão O*bserver* [Gamma, 1994].

A Figura 8.2 apresenta a proposta deste refatoramento.

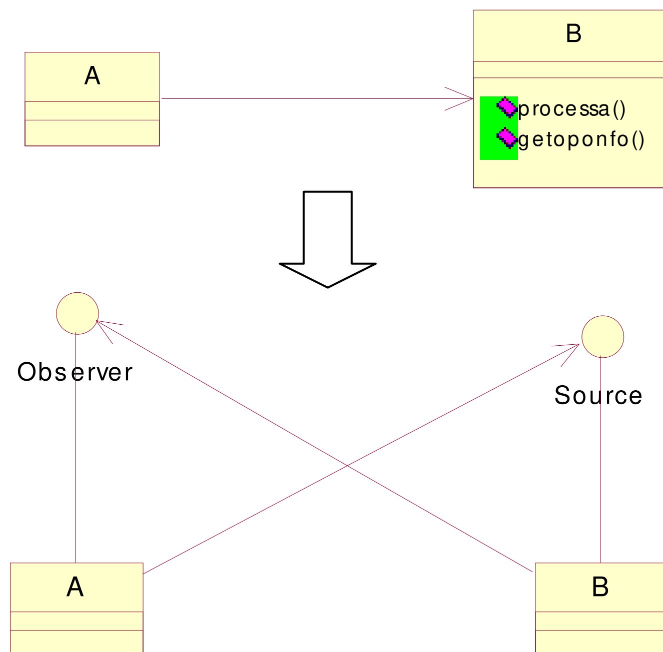


Figura 8.2 – Idéia básica do refatoramento *Introdução de Observer*.

8.1.3 Condições de aplicação

Uma das situações onde o padrão de projeto *Observer* [Gamma, 1994] pode ser utilizado é quando um objeto precisa publicar alguns eventos que podem ser por ele gerados e não conhece os objetos interessados nestes eventos. No entanto, este refatoramento considera especialmente o caso de um objeto realizar algum processamento e outras classes necessitarem obter informações sobre o progresso deste processamento.

8.1.4 Mecanismo

1 Criar classe `EventListener` interessada nos eventos gerados por uma outra classe.

1.1 A classe `EventListener` deve representar a classe que é avisada quando o evento ocorrer.

2 Criar a classe `StatusChangeEventSource` que gera eventos e informa aos seus interessados.

2.1 Esta classe (`StatusChangeEventSource`) deve representar uma classe que gera o evento de mudança de estado.

2.2 Implementar a classe.

3 Planejar, projetar, implementar e executar os testes que validem a nova classe.

4 Fazer a classe que gera informações (aquela que deve ser observada, a origem dos eventos) herdar da classe `StatusChangeEventSource`.

5 Quando ocorrer uma mudança no estado da classe observável, fazer com que ela invoque o método `StatusChange`.

6 Fazer a classe observadora implementar a interface `StatusChangeEventListener` e implementar o método `Update ( StatusChangeEvent *)` da forma mais adequada à classe.

7 Compilar e testar as classes envolvidas na modificação.

8 Repetir os dois passos anteriores para todas as classes interessadas em um dado evento.

9 Compilar e testar o sistema.

Os nomes das classes propostos são apenas sugestões e podem ser modificados para adequar-se à situação na qual o refatoramento será aplicado.

A própria estrutura para implementação do padrão *Observer* pode ser adaptada para prover maior flexibilidade como, por exemplo, a geração de vários tipos de eventos diferentes. Esta estrutura consiste apenas de uma sugestão, o importante é a seqüência de ações utilizadas para realizar a modificação no código do sistema.

8.1.5 Conseqüências

Com a aplicação deste refatoramento, observa-se que o nível de acoplamento entre as classes envolvidas diminui. As classes estão acopladas a interfaces e não a implementações. Além do benefício de mais de um objeto poder acompanhar o progresso do processamento realizado por uma classe, sem que ela precise conhecer suas interfaces ou seus comportamentos.

A flexibilidade do sistema também aumenta uma vez que a estrutura está pronta para inserção de novas operações que necessitem prover informações a outras classes.

É importante perceber que, em algumas situações, a aplicação do refatoramento pode implicar no aumento de complexidade do sistema. Em situações nas quais as classes envolvidas, tanto a classe origem dos eventos quanto a classe interessadas nos eventos já fizerem parte de uma hierarquia de classes é possível que seja introduzida herança múltipla no sistema o que pode provocar o aumento da complexidade destas classes. A herança múltipla dificulta o entendimento da classe por indicar que a subclasse herda comportamento de várias outras classes e, além de provocar um aumento de complexidade, implica na dificuldade de reuso da classe uma vez que esta não pode ser reutilizada separadamente.

8.1.6 Exemplo

Algumas operações realizadas no servidor LightBase envolvem a reorganização de muitos dados, ou a geração completa dos índices do sistema e por isto podem ser demoradas e durar desde de alguns minutos a algumas horas, dependendo do volume de dados que serão processados. Nestes casos, é preciso que o progresso das operações seja apresentado ao usuário para que ele compreenda o tempo gasto e acompanhe a evolução do processo. Um exemplo destas operações é o reprocessamento das bases de dados. Esta operação recria a base e os índices de dados. Nesta situação é preciso que o objeto que realiza o processamento avise aos interessados sobre as mudanças de seu estado, mas provavelmente ele não precisa conhecer estes interessados que podem ser elementos de interface ou outros objetos do sistema que não contribuem pra o desenvolvimento de sua atividade.

Assim, o exemplo apresentado aqui é a introdução do padrão *Observer* no reprocessamento das bases de dados. Onde `Base` é a classe observada e as demais são as interessadas na mudança de seu comportamento.

A Figura 8.3 apresenta os principais conceitos envolvidos no reprocessamento de uma base de dados. A Figura 8.4 apresenta as principais classes envolvidas nesta tarefa. A Figura 8.5 apresenta o diagrama de seqüência do reprocessamento de bases para fornecer mais informação sobre esta funcionalidade do sistema.

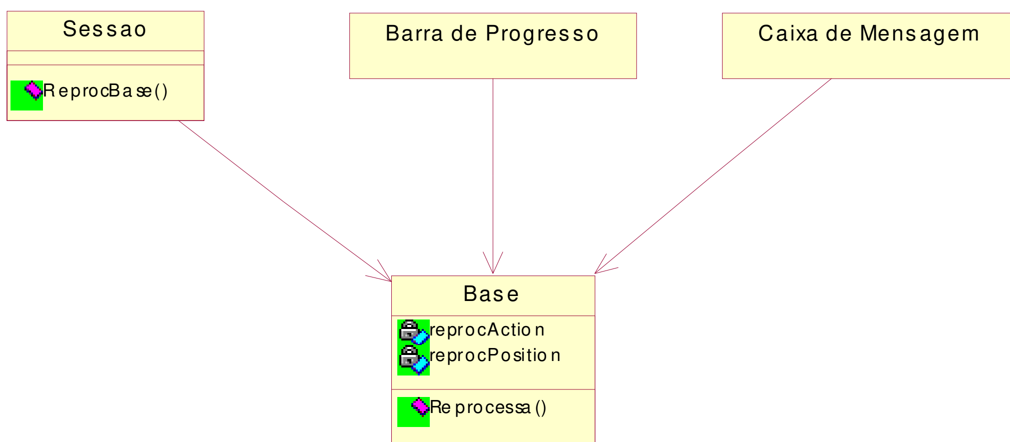


Figura 8.3 - Conceitos envolvidos no reprocessamento de uma base de dados e interessados no progresso da operação.

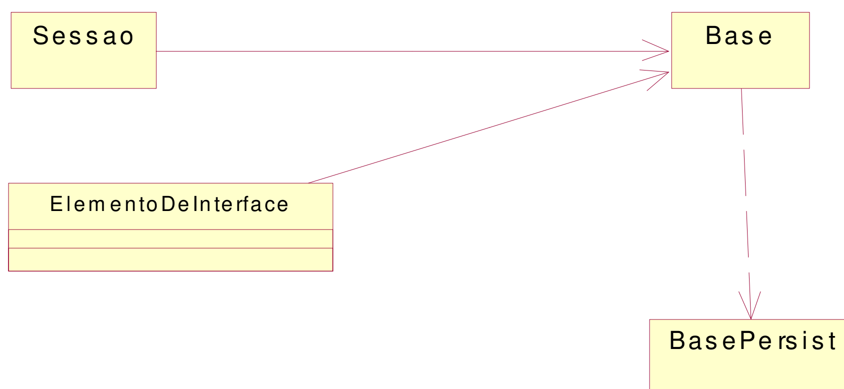


Figura 8.4 -Diagrama de classes para o reprocessamento de bases.

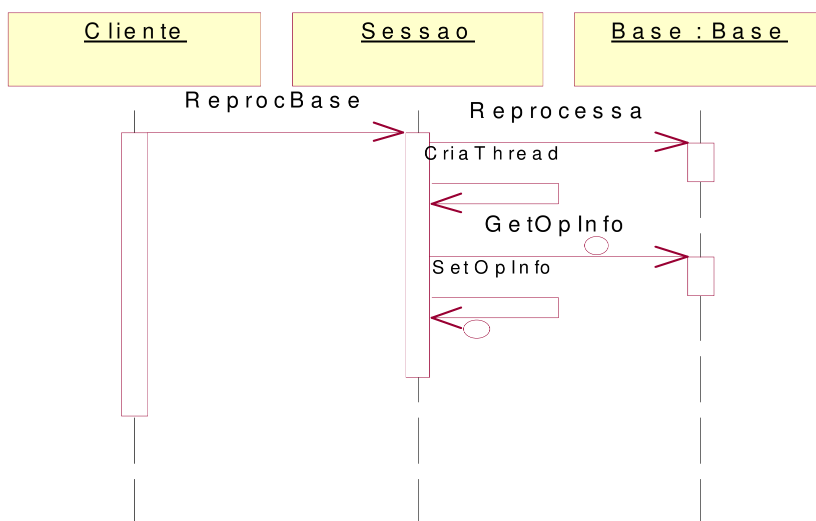


Figura 8.5 - Diagrama de seqüência simplificado referente à operação de reprocessamento.

O método `ReprocBase` de `Sessao` é o seguinte.

```
int Sessao::ReprocBase(Base *pBase, BOOL bCopyRec)
{
	int iRetVal = false;
	ThreadPar * p = new ThreadPar();
	p->setOrigem( pBase );
	p->setDestino( this );
	hThreadHandle = CreateThread(NULL, 0, ThreadFunc, pExec, 0, &dwThreadId);
	iRetVal = pBase->Reprocessa();
	if( hThreadHandle ){
		this->CancelOperation();
	WaitForSingleObject( hThreadHandle, INFINITE );
	}
	return iRetVal;
}
```

Neste trecho de código identifica-se a criação de uma linha de execução (`hTreadHandle`). A seguir, o código da função `ThreadFunc` executada por `hTreadHandle`.

```
DWORD ThreadFunc( LPVOID pParam )
{
	ThreadPar	*pThreadPar = (ThreadPar *) pParam;
	char		szMsg[ 512 ];
	float		fPerc;

	if(pThreadPar){
		do{
			Sleep( 1000 );
			if(!pThreadPar->getOrigem()->GetOpInfo()){
				pThreadPar->getDestino()->SetOpInfo(szMsg,fPerc);
			} else {
				break;
			}
		} while( fPerc < 100 );
		Sleep( 1000 );
	}
	return( 0 );
}
```

Ao examinar o código desta função, percebe-se claramente sua funcionalidade: acompanhar o progresso do processamento e atualizar esta informação no objeto `Sessao`. Esta modelagem é fraca por várias razões. Por exemplo, é obrigatória a criação de uma linha de execução para cada objeto que desejar informações acerca do processamento, a função está acoplada ao objeto do qual deseja obter informações, isso implica a criação de uma nova função para cada objeto que deve fornecer retorno de seu processamento. Estes exemplos explicitam algumas fraquezas estruturais desta modelagem que não é completamente orientada a objetos devido ao uso desta função.

Outro problema apresentado pela implementação é a definição do tamanho do intervalo de tempo estipulado para a verificação do estado do processamento. Caso seja escolhido um intervalo muito pequeno, é possível que tempo de processador seja gasto desnecessariamente averiguando o mesmo estado do processamento. Isto poderia afetar o desempenho do sistema.

Caso o intervalo escolhido seja muito grande, pode ocorrer a perda de informações já que mais de um estado pode ser alcançado e ultrapassado no intervalo de tempo escolhido.

Para tornar esta modelagem mais extensível, este refatoramento propõe a introdução do padrão *Observer*. Assim, o novo diagrama de classes para este trecho do sistema está apresentado na Figura 8.6.

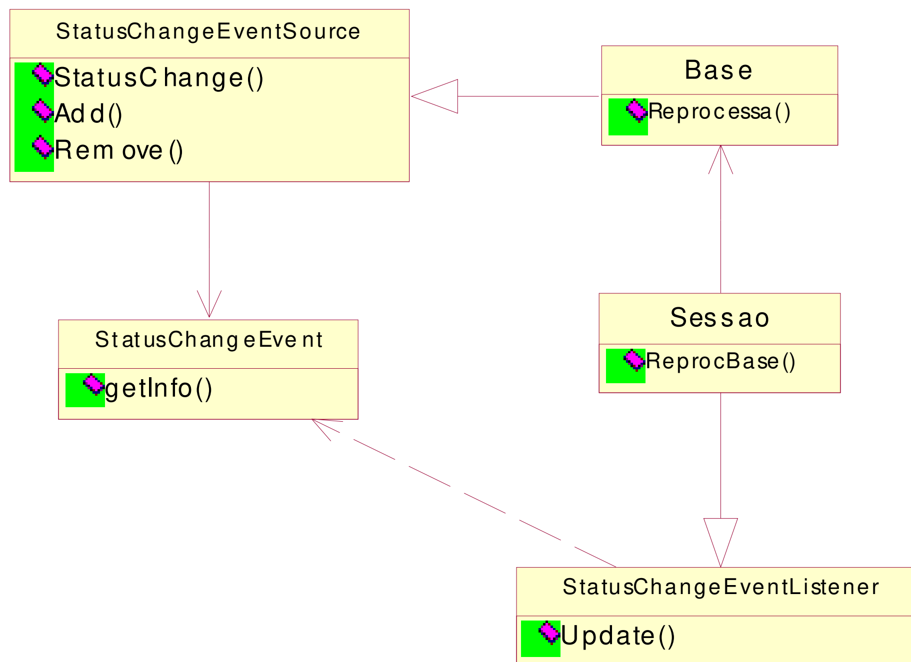


Figura 8.6 - Diagrama de classes após a aplicação do refatoramento.

Passos 1, 2 e 3

Os primeiros passos consistem na criação da classe que representa o evento ocorrido e da classe que representa o observador, ou seja, o interessado no evento. A classe `StatusChangeEvent` representa o evento de mudança de estado que é o evento gerado pela classe `Base`. A classe `StatusChangeEventListener` representa a classe interessada no evento de mudança de estado.

```
class StatusChangeEventListener
{
public:
	virtual ~StatusChangeEventListener( void ){};
	virtual void Update(StatusChangeEvent * theChanged ) = 0;
protected:
	StatusChangeEventListener( void ){};
};

class StatusChangeEvent
{
public:
	virtual ~ StatusChangeEvent ( char* msg, long p ){};
	const char * getInfo(void);
```

```
StatusChangeEvent( void ){};
};
```

Com as classes criadas, é necessário construir testes de unidades e executá-los para retirar possíveis problemas existentes.

Passos 4 e 5

O próximo passo consiste na criação da classe geradora do evento, no caso, `StatusChangeEventSource`. A seguir, a classe geradora de informações deve ser modificada para que herde da classe criada (`StatusChangeEventSource`).

```
class StatusChangeEventSource
{
public:
	virtual ~StatusChangeEventSource ( void ){};

	void add(StatusChangeEventListener * iObs );
	void remove(StatusChangeEventListener * iObs );
	void StatusChange( StatusChangeEvent* );
protected:
	StatusChangeEventSource( void ){};
private:
	List<StatusChangeEventListener> listeners;
};

void StatusChangeEventSource::add(StatusChangeEventListener* iObs)
{
	listeners.Add( (StatusChangeEventListener*)iObs );
}

void StatusChangeEventSource::remove(StatusChangeEventListener* iObs )
{
	for ( int i = 0; i < listeners.Size(); i++ ){
		if (((StatusChangeEventListener*)listeners[i])==iObs){
			listeners.Go ( i );
			listeners.Del( );
		}
	}
}

void StatusChangeEventSource::StatusChange(StatusChangeEvent* pEv)
{
    for ( int i = 0; i < listeners.Size(); i++ ){
    ((StatusChangeEventListener*)listeners[i])->Update(pEv);
	}
}
```

Com a classe geradora de eventos criada, deve-se fazer com que a classe `Base` herde de `StatusChangeEventSource` e gere os eventos associados com a mudança de estado durante o seu processamento.

```
class Base : public StatusChangeEventSource
{

public:
	~Base( void );

	…

protected:
```

```
        …
        void setMsg ( long );
        const char *getMsg( void );
        …
        int getDir( char * destiny, char * szBase, char * szPar );
        void restoreAccesIndex( void );
        void restoreUser(char * szOrigUser, LBSC_User * plbscuUser );
        void copyHeader( LBSC_Base * destino, LBSC_Base * origem );
        int copySlot( LBSC_Base * destino, LBSC_Base * origem );
        int copyRegs( LBSC_Base * destino, LBSC_Base * origem, FILE *pfile );

private:

        …

        LBSC_Base * plBase;
        char szMsg[ MAX_MSG_TAM ];
};
```

Passos 6, 7, 8 e 9

Ao mudar a etapa do reprocessamento, o método `StatusChange` deve ser invocado para que os interessados sejam avisados da mudança.

O trecho de código a seguir foi extraído do método `Start` da classe `Base`. Nele, o estado é modificado através da atualização da variável `szMessage` que indica, a mensagem associada à tarefa sendo executada.

```
…
index = count;
if( _pcInter && _pcInter->BuildStatus == CHECKISOK ){
        szMessage=_pcInter->GetGenMsgsAppVar( (char *)msgs[index]);
        if( szMessage ){
                strcpy( szMsg, szMessage );
                delete szMessage;
        }
}

// Verifica se a base estah aberta em modo exclusivo
if( plBase->IsExclusive() == FALSE ){
        return( LBSE_BASENOTEXCLUSIVE );
}
…
```

Com a nova modelagem, este trecho de código é modificado para o seguinte.

```
…
StatusChangeEvent * pEvent = new StatusChangeEvent(msgs[index]);
StatusChange( pEvent );
delete pEvent;

// Verifica se a base estah aberta em modo exclusivo
if( plBase->IsExclusive() == FALSE ){
        return( LBSE_BASENOTEXCLUSIVE );
}
…
```

Com estas alterações efetuadas, é chegado o momento de atualizar as classes interessadas no evento de mudança de estado da classe `Base`.

A primeira alteração é fazer com que a classe `Sessao` seja subclasse de `StatusChangeEventListener`. Para isto, o método `Update` deve ser implementado. As alterações nas demais classes de interface não serão apresentadas aqui por consistirem de alterações semelhantes à da classe `Sessao`.

```
void Sessao::Update(StatusChangeEvent * theChanged )
{
char                *szMsg = NULL;


szMsg = theChanged->GetInfo();
fPercent = CALCULA_PERCENT(fPercent)
SetOpInfo( szMsg, fPercent );
}
```

A seguir, o método `ReprocBase` deve ser alterado para que a classe `Sessao` inscreva-se como interessada nos eventos de mudança de estado da classe `Base` como segue.

```
int Sessao::ReprocBase(Base *pBase, BOOL bCopyRec)
{
       int iRetVal = false;
       ThreadPar * p = new ThreadPar();
       p->setOrigem( pBase );
       p->setDestino( this );
       hThreadHandle = CreateThread(NULL, 0, ThreadFunc, pExec, 0, &dwThreadId);
       iRetVal = pBase->Reprocessa();
       if( hThreadHandle ){
              this->CancelOperation();
       WaitForSingleObject( hThreadHandle, INFINITE );
      }
       return iRetVal;
}
```

O código a seguir apresenta o método anterior após as alterações propostas pelo refatoramento.

```
int Sessao::ReprocBase(Base *pBase, BOOL bCopyRec)
{
       int iRetVal = false;
((StatusChangeEventSource*)pBase)->add(this);
iRetVal = pBase->Reprocessa();
       ((StatusChangeEventSource*)pBase)->remove(this);
return iRetVal;
}
```

E com isto, elimina-se a linha de execução anteriormente utilizada simplesmente para verificar o estado do processamento. Além disso, outros benefícios foram obtidos como maior clareza e adaptabilidade do código.

O próximo passo é a geração e teste do sistema como um todo. Os elementos de interface interessados no processamento do objeto também são adicionados à sua lista de observadores.

8.2 Introdução de *Singleton*

8.2.1 Fundamentos

O padrão de criação *Singleton* [Gamma, 1994] permite certificar-se que uma classe será instanciada apenas uma vez e as demais classes que necessitarem de suas funcionalidades acessarão a mesma instância. E, ainda, provê um ponto único de acesso a esta instância.

A Figura 8.7 apresenta a estrutura proposta pelo padrão *Singleton*.

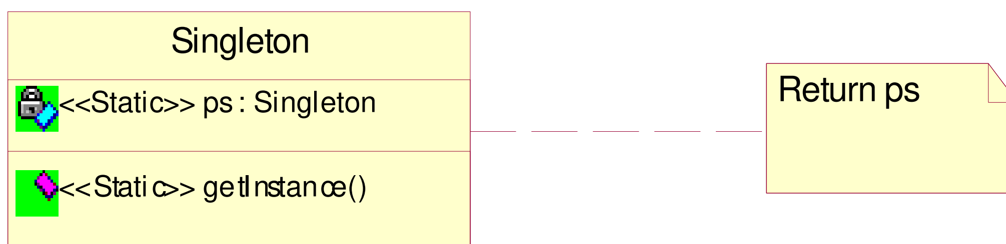


Figura 8.7 - Padrão de projeto *Singleton*.

Esta é a estrutura adequada quando deve existir apenas uma instância de uma dada classe e, quando é necessário que a classe seja extensível através de herança.

8.2.2 Sumário

É preciso certificar-se de que uma classe é instanciada apenas uma vez durante a execução do sistema. Esta situação está implementada através de métodos e variáveis de classe (estáticas) ou através de objetos globais, por exemplo. O refatoramento propõe a substituição da implementação atual pelo uso do padrão de criação *Singleton*.

8.2.3 Condições de aplicação

É comum em sistemas de software, a presença de classes que devem ser instanciadas uma única vez e esta instância precisar ser acessível em várias classes. Está é uma situação que pode ser modelada de várias formas. É possível encontrar implementações que utilizem uma classe com métodos e atributos de classe (estáticos), ou ainda soluções que utilizem objetos globais.

No caso da implementação através de métodos e atributos estáticos, há uma perda quanto à adaptabilidade do sistema. Métodos e atributos estáticos não podem ser sobrescritos, assim, uma importante característica do projeto orientado a objetos é perdida: o polimorfismo. Outra desvantagem desta abordagem é que futuras mudanças para permitir a existência de mais de uma instância da classe podem ser mais difíceis do que deveriam.

A segunda abordagem, o uso de objetos globais, não é garantia de que a classe será instanciada apenas uma vez. Ela pode ser instanciada mais de uma vez apesar de não estar o sendo, embora a implementação, provavelmente, permita. Nesta solução, a responsabilidade de certificar-se de que a instância é única recai sobre o programador e não sobre a classe. O programador deve estar atento ao uso da variável global, com o uso do padrão *Singleton*, a classe é a responsável pela sua única instância, pois ela também define a forma de acesso à classe através de um método já que seu construtor não está acessível.

8.2.4 Mecanismo

1 Fazer um construtor com o acesso protegido.

2 Criar um atributo de classe que é um ponteiro para ela mesma.

3 Criar o método de instanciação, também estático, que retorna a instância de classe anteriormente criada.

4 Transformar os métodos e atributos de classe em métodos e atributos de instância (caso seja esta a implementação).

5 Planejar, projetar, implementar e executar os testes que validem a nova classe.

6 Determinar a relação existente entre as classes clientes e a classe *Singleton*.

 6.1 Caso seja necessária uma referência permanente, adicionar um atributo de classe que deve ser iniciado no construtor da classe que contém a referência através da invocação do método de acesso à classe.

7 Modificar todas as referências à classe para usar a nova implementação. Ou usar o novo atributo da classe cliente, ou invocar, em cada trecho, o método de instanciação e, posteriormente, o método desejado.

8 Eliminar o objeto global (caso seja esta a implementação).

9 Compilar e testar o sistema.

8.2.5 Conseqüências

Com a aplicação deste refatoramento, a classe se responsabiliza pela gerência do número de objetos que podem ser instanciados. Além disso, uma possível mudança para permitir inúmeros objetos implica em modificação apenas no método da classe responsável por esta gerência.

Uma outra conseqüência da aplicação deste refatoramento é que o fato de a classe ser instanciada dinamicamente implica em certificações da efetiva instanciação do objeto. O fato de o objeto não ser carregado juntamente com o sistema implica na possibilidade de falha na criação do objeto, assim, esta situação deve ser verificada antes do uso do objeto retornado para evitar problemas com ponteiros nulos. Isto pode ser um problema especialmente em casos nos quais o método de instanciação é invocado várias vezes por não ser necessário armazenar uma referência ao objeto. Isto pode provocar a inclusão de várias condições para testar a integridade do valor retornado, aumentando a complexidade do código.

8.2.6 Exemplo

No módulo ***lbs*** do servidor LightBase existe um objeto para geração de informações sobre as atividades realizadas pelo servidor. Este objeto é responsável por gerar um *log* das operações executadas. Existe apenas uma instância do objeto que é compartilhada por todas as linhas de execução do servidor, e assim, toda a informação é gravada em um único arquivo de texto.

As linhas de código a seguir definem alguns macros de uso deste objeto.

```
#define USELOG(x)      C_Log       x();
#define EXTLOG(x)      extern      C_Log  x;
#define INITLOG(x)     x.InitLog();
#define ENDLOG(x, y)          x.EndLog( y );
#define PRINTLOG(x, y) x.SetParms(__FILE__,__LINE__); x.PrintLog y
#define FLUSHLOG(x)     x.FlushLog();
```

A macro `USELOG(x, y)` é utilizada apenas em um ponto do módulo LBS.

```
// objeto para gerar log no LBS
USELOG( _clLBSLog );
```

Esta linha de código define um objeto do tipo `C_Log` chamado `_clLBSLog` e este objeto será utilizado em todo o módulo para gerar o *log* do sistema.

A linha de código a seguir está presente 39 vezes no módulo ***lbs***, ou seja, 39 dos 50 arquivos fazem referência ao objeto `_clLBSLog` instanciado na inicialização do sistema.

```
EXTLOG( _clLBSLog );       // objeto para geracao de log
```

A classe `C_Log` está apresentada logo a seguir.

```
class C_Log
{
private:

       shdStruct* pGlobals;
       char   szFileName[ MAX_TAM ];
       int    iLineNum;
       char   szBuffer[ LOGLINESIZE ];
       char   szLine[ LOGLINESIZE ];
       char   szFmtAux[ LOGLINESIZE * 2 ];
       void   GetDateTime( char * );
       void   WriteLine( char * );
       void   TerminateLog();
       void   OpenFile();
       void   init();

public:
                    C_Log();
                    ~C_Log();
       void   SetParms( char *, int );
       void   PrintLog( char *, ... );
       void   FlushLog();
       void   EndLog( HANDLE );
       void   InitLog();
};
```

Esta classe representa a situação típica de modelagem através de um *Singleton*. Uma classe que deve ter uma única instância compartilhada entre todos os clientes.

Os primeiros passos do mecanismo propostos pelo refatoramento são a criação de um construtor e de um apontador para a classe ambos com restrições de acesso na própria classe que será modelada utilizando a idéia do padrão *Singleton*.

```
class C_Log
{
private:

       shdStruct* pGlobals;
       char   szModuleName[ MAX_TAM  ];
       char   szFileName[ MAX_TAM ];
       int    iLineNum;
       char   szBuffer[ LOGLINESIZE ];
       char   szLine[ LOGLINESIZE ];
       char   szFmtAux[ LOGLINESIZE * 2 ];
       void   GetDateTime( char * );
       void   WriteLine( char * );
       void   TerminateLog();
       void   OpenFile();
       void   init();

       static C_Log* pcLog;

protected:
                    C_Log();

public:
                    ~C_Log();
       void   SetParms( char *, int );
       void   PrintLog( char *, ... );
       void   FlushLog();
       void   EndLog( HANDLE );
```

```
	void   InitLog();
};
```

A próxima etapa é a criação de um método de acesso à instância única da classe como segue.

```
class C_Log
{
private:

	shdStruct* pGlobals;
	char   szModuleName[ MAX_TAM  ];
	char   szFileName[ MAX_TAM ];
	int    iLineNum;
	char   szBuffer[ LOGLINESIZE ];
	char   szLine[ LOGLINESIZE ];
	char   szFmtAux[ LOGLINESIZE * 2 ];
	void   GetDateTime( char * );
	void   WriteLine( char * );
	void   TerminateLog();
	void   OpenFile();
	void   init();

	static C_Log* pcLog;

protected:
			C_Log();

public:
			~C_Log();
	static C_Log* getInstance(void);
void   SetParms( char *, int );
	void   PrintLog( char *, ... );
	void   FlushLog();
	void   EndLog( HANDLE );
	void   InitLog();
};

C_Log* C_Log::getInstance(void)
{
	if( !pcLog ) {
		pcLog = new C_Log();
}
return pcLog;
}
```

O passo a seguir é compilar e testar a classe `C_Log`.

A seguir deve-se atualizar as referências à classe para que acessem `C_Log` através do método `getInstance()`.

As 39 referências à macro `EXTLOG` devem ser examinadas para determinar o tipo de relação com `C_Log` e a seguir eliminadas do código.

O primeiro exemplo examina a relação entre a classe `Base` e `C_Log`. Ao analisar os números tem-se que todos os 12 arquivos que compõem a classe `Base` fazem referência à macro `EXTLOG( _clLBSLog )`. Um total de 348 referências são feitas à macro

PRINTLOG e estas referências pertencem aos métodos da classe Base e muitos destes métodos são executados várias vezes. Assim, este é um caso onde é válida a adição de um apontador para a classe C_Log à classe Base, já que diminui a execução de um método para adquirir uma referência à classe sempre que algo for escrito no log. É interessante ressaltar que isto é válido devido à alta freqüência com que a classe Base referencia a classe C_Log.

A linha a seguir declara um apontador para a classe C_Log e deve ser adicionada a classe Base.

```
C_Log * pLog;
```

Este apontador deve ser preenchido no construtor da classe ao qual pertence.

```
Base::Base(){
...
pLog = C_Log::getInstance();
...
}
```

Na classe Base, as referências à _clLBSLog devem ser substituídas por pLog. Assim, onde existir invocação à macro PRINTLOG, deve-se colocar pLog.

Antes:

```
PRINTLOG( _clLBSLog, ("LBSC_Base::FirstFRecord") );
```

Depois:

```
pLog->SetParms(__FILE__,__LINE__);
pLog->PrintLog ("LBSC_Base::FirstFRecord");
```

Outra possibilidade é mudar a definição das macros, assim:

```
#define INITLOG(x)     x->InitLog();
#define ENDLOG(x, y)          x->EndLog( y );
#define PRINTLOG(x,y) x->SetParms(__FILE__,__LINE__);x->PrintLog y
#define FLUSHLOG(x)     x->FlushLog();
```

E modificar o código como segue.

```
       PRINTLOG( pLog, ("LBSC_Base::FirstFRecord") );
```

A próxima etapa é a eliminação do objeto global _clLBSLog e teste das classes alteradas bem como do sistema como um todo.

8.3 Introdução de *Singleton* de vários contextos

8.3.1 Fundamentos

O padrão *Singleton* já foi apresentado anteriormente, sendo assim, aqui será apresentado o formato do padrão para vários contextos. Um *Singleton* que trata vários contextos é apenas uma classe `Singleton` que pode ter valores de inicialização diferentes. E para cada valor de inicialização, apenas uma única instância é criada.

Existem várias formas de modelar esta situação, no entanto, o maior interesse aqui é pelo caso no qual apenas o valor de inicialização seja modificado, o comportamento da classe continua o mesmo. Esta situação pode ser modelada através de subclasses do *Singleton*, porém, os valores de inicialização podem não ser conhecidos previamente e, além disso, podem existir vários possíveis valores diferentes o que poderia comprometer a legibilidade do código dado o número excessivo de classes que seriam criadas.

Deste modo, a modelagem proposta é uma classe *Singleton* que recebe o valor de inicialização no método `getInstance` e devolve a instância adequada àquele valor. Para isso, ela deve manter uma lista de pares <instância, valor> para que possa manter uma única instância por valor de inicialização.

8.3.2 Sumário

Uma classe tem um objeto global para cada contexto que ela se aplica. Elimina-se estes objetos e transforma-se a classe em um *Singleton* que trata de vários contextos. A Figura 8.8 apresenta um resumo deste refatoramento.

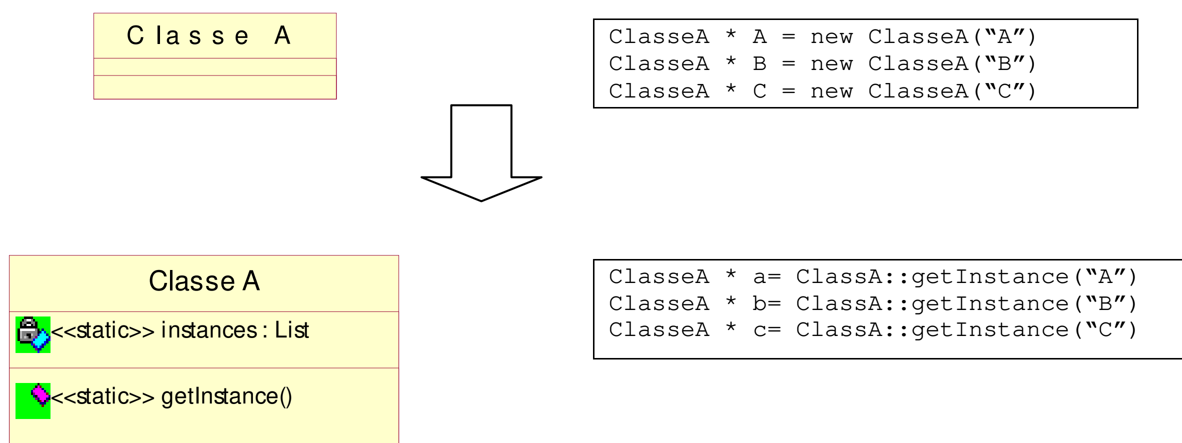


Figura 8.8 - Refatoramento *Introdução de Singleton de vários contextos*.

8.3.3 Condições de aplicação

O padrão *Singleton* [Gamma, 1994] deve ser aplicado quando se deseja assegurar que uma classe será instanciada apenas uma vez, e fornecer um ponto único de acesso à classe. No entanto, há situações em que se desejam várias instâncias de uma dada classe, porém uma única para cada contexto no qual ela está inserida. Ou seja, existem vários contextos de aplicação do *Singleton* e, em cada um deles, um único objeto deve ser instanciado. O que diferencia estes objetos (instâncias) é o valor de inicialização, o qual indica o contexto que se deseja.

Por exemplo, um sistema utiliza vários arquivos de controle modelados pela mesma classe. É importante que o acesso a estes arquivos seja controlado e sincronizado. Para assegurar a sincronização, instalam-se regiões críticas na classe. No entanto, ainda é necessário certificar-se que para cada arquivo, um único objeto será instanciado. Isto pode ser feito através do uso do padrão *Singleton* [Gamma, 1994].

Este tipo de situação pode ser modelado com vários objetos globais, um para cada contexto. Esta solução apresenta alguns problemas como responsabilizar o programador pela instância única da classe ou a poluição do programa com definições de objetos globais.

8.3.4 Mecanismo

1. Identificar a interface da classe que será implementada como um *Singleton* [Gamma, 1994].
2. Criar classe *Singleton* [Gamma, 1994] que implementa a interface definida.
3. O método `getInstance` deste *Singleton* deve ser implementado para retornar o objeto correspondente ao parâmetro passado ao método.
4. Adicionar toda a inteligência dos métodos contidos na interface definida no passo anterior.
5. Planejar, projetar, implementar e executar os testes que validem a nova classe.
6. Fazer com que o código antigo referencie a nova estrutura.
7. Eliminar os objetos globais espalhados pelo sistema.
8. Gerar e compilar o sistema.

8.3.5 Conseqüências

O refatoramento do código proposto aqui possibilita a redução do número de objetos globais declarados no sistema. As conseqüências observadas aqui são muito parecidas com as

conseqüências observadas com a aplicação do refatoramento *Introdução de Singleton* uma vez que eles apresentam soluções a problemas parecidos.

8.3.6 Exemplo

Sistemas de informação grandes e comerciais são, muitas vezes, concebidos com o requisito de que as informações exibidas possam ser traduzidas para vários idiomas. Este requisito é atendido pelo estudo de caso considerado neste trabalho de acordo com a solução descrita a seguir. Toda informação exibida pelo sistema está associada a um identificador. Antes de exibir uma informação, o sistema instancia um objeto que recupera a cadeia (de caracteres) correspondente ao identificador informado. Esta cadeia de caracteres é recuperada de acordo com o idioma especificado na instalação do sistema.

No estudo de caso, as informações e seus respectivos identificadores estão armazenados em arquivos de inicialização que detém o mesmo nome do módulo ao qual pertencem. Isto provoca a existência de um arquivo de inicialização por módulo presente no sistema. Cada módulo do sistema que precisa exibir informações detém uma instância global de uma classe que fornece acesso às informações presentes em um dado arquivo de inicialização. Ou seja, para cada módulo do sistema que apresenta informações ao usuário, existe uma instância de uma classe que resolve as informações a serem exibidas.

O problema da solução reside na modelagem adotada. Através desta descrição, identifica-se claramente a possibilidade de modelagem através do padrão *Singleton* [Gamma, 1994]. No entanto, esta situação está modelada através de objetos globais espalhados por todo o sistema o que, como analisado anteriormente, provoca conseqüências indesejadas.

No entanto, este caso exige uma modelagem um pouco mais elaborada. Cada arquivo de inicialização deve ser acessado através de uma única instância do objeto responsável por resolver as informações.

Passo 1

O primeiro passo do refatoramento é identificar a interface da classe que será responsável por mapear um identificador em uma cadeia de caracteres que será apresentada pelo sistema. A interface está apresentada no quadro a seguir.

```
class IMsgMapper {
protected:
	IMsgMapper();
public:
	virtual ~IMsgMapper();
	virtual void getMsg( const char * id, char * szMsg );
};
```

Passos 2 e 3

O próximo passo sugere a criação de uma classe que implemente a interface `IMsgMapper`. Esta classe deve ser criada como um *Singleton*.

```
class MsgMapper : public IMsgMapper {
protected:
       char * szIniFile;
       static List<IMsgMapper> objs;
       INIFile * pIni;

       ImsgMapper* getMapper( const char * szFile );
       const char * getId();
public:
       ~MsgMapper();
       void getMsg( const char * id, char * szMsg );
       static IMsgMapper *getInstance( const char * szModule );
};

MsgMapper:: MsgMapper( const char * szFile )
{
       szIniFile = strdup( szFile );
       pIni = new IniFile(szIniFile);
}

MsgMapper::~MsgMapper( void )
{
       if ( szIniFile ) {
              delete szIniFile;
}
}

const char * MsgMapper::getId()
{
       return szIniFile;
}
void MsgMapper::getMsg ( const char *id, char * szMsg )
{
pIni->getVar(id, szMsg );
}

IMsgMapper *MsgMapper::getMapper(const char * szModule )
{
    ImsgMapper* pObj = null;
    for ( int i = 0; i < objs.Size(); i++ ){
    if(strcmp(((IMsgMapper*)objs[i])->getId(),szModule)==0){
              pObj = (IMsgMapper*)objs[i];
              }
       }
       if ( !pObj ){
              pObj = new MsgMapper ( szModule );
              objs.Add( pObj );
}
return pObj;
}

IMsgMapper *MsgMapper::getInstance( const char * szModule )
{
       if ( !szModule ) {
              return NULL;
}
       return getMapper(szModule);
}
```

A classe anterior deve, então, ser compilada e testada.

Passos 4, 5 e 6

A próxima etapa é fazer com que o código antigo referencie a nova estrutura de classes. Alguns exemplos de modificações deste tipo são apresentados em seguida.

A linha a seguir está presente em 10 arquivos do módulo ***lbs***. Esta linha foi eliminada de todos os arquivos após o refatoramento.

```
extern AppParmsMngr *_pcInter;
```

A seguir a definição e instanciação do objeto `_pcInter` presentes no arquivo de inicialização do módulo.

```
AppParmsMngr* _pcInter = NULL;
pcInter = new AppParmsMngr( "LBS" );
```

Este trecho de código também deve ser eliminado do módulo uma vez que o construtor da classe é protegido e a instância da classe referente ao parâmetro de inicialização especificado (***lbs***) só pode ser acessada através do método `getIstance()`.

O trecho de código a seguir foi extraído de um método da classe `Base`.

```
szMsg = _pcInter->GetGenMsgsAppVar( "LBSMSG_INITIS" );
IncStep( szMsg );
delete szMsg;
```

Este código, após ser adaptado a nova estrutura de classes, passa ao que segue.

```
MsgMapper::getInstance("LBS")->getMsg("LBSMSG_INITIS", szMsg );
IncStep( szMsg );
delete szMsg;
```

A aplicação do refatoramento é concluída com o teste do completo sistema.

9 VALIDAÇÃO DOS REFATORAMENTOS

Métricas são utilizadas para tentar quantificar a qualidade de processos e produtos de software. Informações numéricas se tornam importantes por facilitar a comunicação e também por evitar discussões acerca de preferências e estilos, filosofia e cultura [Bansiya, 1998]. Medições também são importantes na avaliação e melhora dos processos e produtos de software.

A avaliação das transformações propostas pelos refatoramentos é feita através de medições. Métricas que avaliem projetos OO são coletadas e seus valores analisados. Tem-se, assim, que a avaliação dos efeitos dos refatoramentos é feita em termos de métricas de produto que avaliem as características de projeto de um sistema OO.

A primeira parte deste capítulo identifica as métricas de projeto OO consideradas na avaliação, bem como os seus significados. A segunda parte consiste da análise dos valores coletados antes e depois da aplicação dos refatoramentos.

9.1 Modelo de avaliação

Modelos de avaliação são propostos na literatura com diversos objetivos. Lindvall usou uma análise direcionada ao impacto nos requisitos para identificar o conjunto de entidades que sofreriam mudanças [Lindvall, 1997]. Kiran compara o impacto de mudanças em sistemas procedurais e sistemas OO [Kiran, 1997]. Outros trabalhos de avaliação do impacto de mudanças em sistemas através do uso de métricas [Han, 1997], [Munson, 1998] endereçam outros objetivos, mas não são conhecidos modelos de análise que avaliem os resultados da aplicação de técnicas de refatoramento. Ou seja, modelos que analisem, através de um conjunto de métricas, o impacto de mudanças que não são motivadas pela adição de funcionalidade ou correção de erros, mas sim pela melhoria do projeto do sistema.

9.2 Modelo conceitual

As transformações propostas pelos refatoramentos modificam a estrutura do programa e, possivelmente, alteram o valor das métricas do sistema [Miceli, 1999]. Deste modo, a avaliação do comportamento das métricas após a sua aplicação pode servir para mensurar os benefícios ou malefícios resultantes do projeto de refatoramento.

9.2.1 Métricas

O conjunto de métricas utilizado aqui foi extraído da literatura e não tem a pretensão de constituir o melhor conjunto de métricas para a avaliação de um produto de software. Este é apenas um conjunto julgado adequado para avaliar os principais efeitos dos refatoramentos no estudo de caso.

As métricas escolhidas permitem medir as principais características de um projeto OO tais como coesão, acoplamento e polimorfismo. Na Tabela 9.1 listam-se as métricas utilizadas e seus significados.

Métrica	Significado
CBO	Número de classes acopladas.
CIS	Número de métodos públicos.
DCC	Número de classes com as quais uma classe está diretamente relacionada.
DIT	Profundidade na árvore de herança.
LCOM	Percentual de perda de coesão nos métodos.
MCCABE_Avg	Complexidade ciclomática média
MCCABE_Max	Complexidade ciclomática máxima.
NOP	Número de métodos polimórficos (novos ou herdados).
RFC	Número de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto da classe.
WMC	Complexidade de uma classe.

Tabela 9.1 - As métricas, e seus significados, utilizadas na análise dos resultados.

As métricas constituintes do modelo são provenientes dos trabalhos propostos por Chidamber e Kemerer [Chidamber, 1998], [Chidamber, 1991], [Chidamber, 1994] são as *C&K Metrics*, métricas conhecidas como QMOOD [Bansiya, 1998], e adaptações da complexidade ciclomática proposta por McCabe [Pfleeger, 1998] para avaliar a complexidade de métodos e classes.

Estas métricas foram coletadas automaticamente através de duas ferramentas de coleta de métricas chamadas Cantata++ [Cantata] e Understand C++ [Understand C++].

9.2.2 Categorias

Para facilitar a análise, as métricas são agrupadas de acordo com o aspecto de projeto OO observado por cada uma delas. A Tabela 9.2 apresenta a divisão de grupos e as métricas que compõem cada um deles.

Categoria	Métricas
Acoplamento	CBO, DIT, DCC.
Coesão	LCOM, CIS.
Complexidade	MCCABE_Avg, MCCABE_Max, WMC, RFC.
Polimorfismo	NOP.

Tabela 9.2 - Grupos de métricas e seus componentes.

9.3 Avaliação do projeto de refatoramento

A avaliação dos resultados obtidos com o projeto de refatoramento foi feita através da análise do impacto sofrido pelas métricas OO do sistema. A organização desta análise segue o modelo descrito anteriormente.

9.3.1 Categoria: acoplamento

Acoplamento é o nível de interdependência entre módulos [Bansiya, 1998]. Uma classe estar acoplada a outra significa que atua sobre ela. Ou, ainda, significa que a classe precisa de outra para desempenhar sua atividade. Classes com muitas dependências são difíceis de reutilizar em outras situações, por exigirem a reutilização das demais classes das quais depende.

Em sistemas onde as classes estão fortemente acopladas, mudanças em uma funcionalidade podem surtir efeito em todo o restante do sistema, o que pode provocar alterações em diversos pontos do código. Esta situação torna a atividade de manutenção mais difícil. Sendo assim, o acoplamento interfere na possibilidade de reuso e extensão de um componente de software.

CBO

Definição. O valor da métrica CBO para uma dada classe representa o número de classes com as quais ela está acoplada. Uma classe está acoplada a outra se uma delas atua sobre a outra, ou seja, métodos de uma usam métodos ou atributos da outra [Chidamber, 1994].

Análise. Altos índices de acoplamento de uma classe dificultam o reuso da classe em outras aplicações.

Dados.

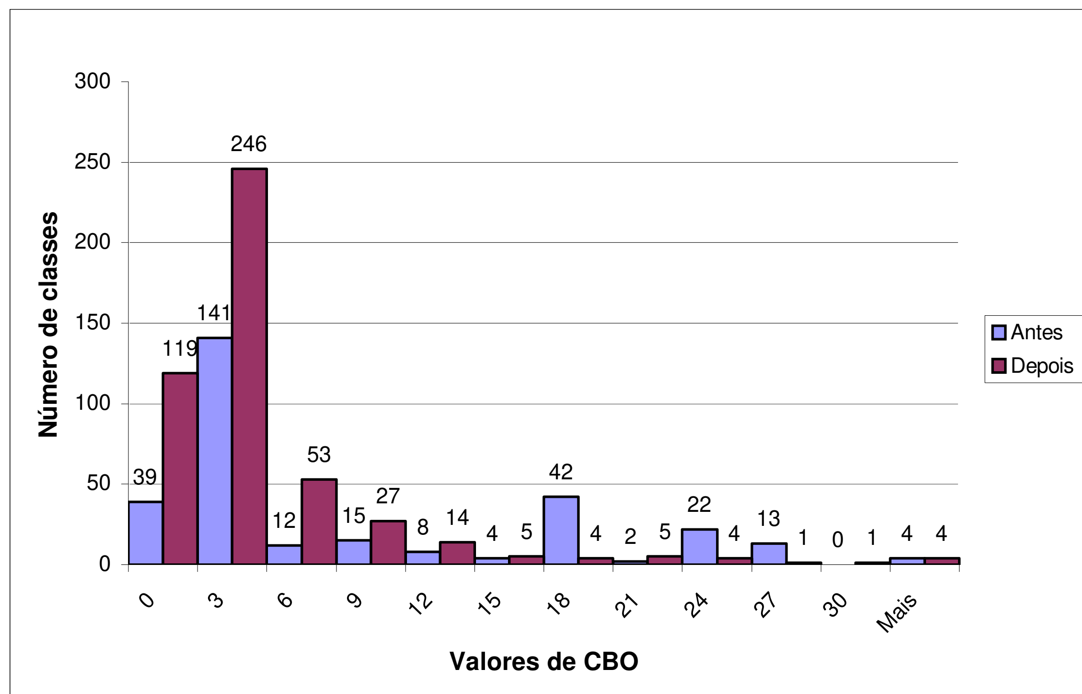


Figura 9.1 - Histograma mostrando a distribuição do número de classes do sistema em função dos valores da métrica CBO antes e depois do projeto de refatoramento.

Interpretação dos dados. Várias observações podem ser feitas através da análise do gráfico presente na Figura 9.1. Antes do refatoramento, o número total de classes no sistema era 302, no término do processo, este total subiu para 483. Esta observação pode ser feita em todos os demais gráficos apresentados nesta análise.

Analisando o segundo gráfico, após a execução do projeto, percebe-se que houve uma concentração mais acentuada de classes à esquerda do gráfico. Ou seja, o número de classes com valores baixos da métrica CBO aumentou após o projeto de refatoramento. Esta observação é reforçada pela análise da média que sofreu uma queda de 12.5, no primeiro, para 5, no segundo gráfico.

O aumento no número de classes do sistema pode significar que novas classes surgiram da subdivisão de outras. O fato de o gráfico insinuar uma diminuição no nível de acoplamento das classes do sistema reforça esta hipótese uma vez que altos índices de acoplamento em componentes de software podem significar inconsistência de funcionalidade e a divisão destes componentes representa uma possível solução ao problema.

DIT

Definição. Calcula a profundidade de uma classe na árvore hierárquica. Caso haja herança múltipla, o valor considerado será o maior valor existente entre a classe e a raiz da árvore.

Análise. Esta também pode ser considerada uma medida de acoplamento, pois herança é uma forma de reuso, mas que provoca forte acoplamento entre as classes.

É preciso observar ainda que quanto mais profunda uma classe estiver na hierarquia, maior será o número de métodos que esta classe pode herdar dificultando, desta forma, a descrição de seu comportamento. Isto também pode revelar um aumento na dificuldade de compreensão da classe. Além da complexidade que uma classe pode apresentar, o acoplamento existente entre as classes componentes de uma hierarquia pode dificultar que elas sejam reutilizadas separadamente.

Dados.

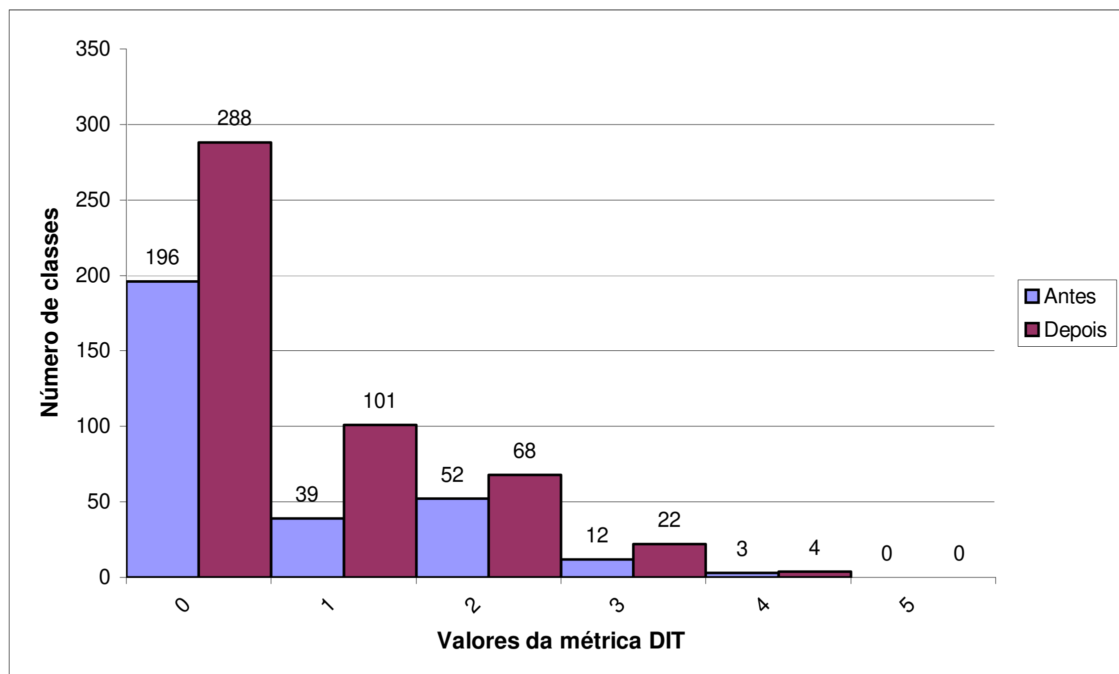


Figura 9.2 - Histograma da distribuição do número de classes do sistema em função dos valores da métrica DIT antes e depois do projeto de refatoramento.

Interpretação dos dados. Mais uma vez o gráfico depois do projeto de refatoramento mostra a ocorrência de uma concentração maior de classes à esquerda do gráfico. A herança é um tipo particular de acoplamento, pois as classes filhas dependem inteiramente de suas classes bases, porém fornece um meio efetivo de reuso. Deste modo, um *tradeoff* é encontrado durante o desenvolvimento e manutenção de um sistema OO. O uso excessivo da herança pode causar forte acoplamento e dificultar entendimento e reuso, porém é um mecanismo eficiente para a reutilização de projeto e implementação. No gráfico presente na Figura 9.2, percebe-se que o valor máximo da métrica DIT não ultrapassa 4. Uma possível explicação é que os projetistas optaram por reduzir o reuso através de herança como forma de favorecer a facilidade de compreensão do sistema, já que uma árvore hierárquica muito profunda pode implicar em dificuldade de entendimento e aumentar o esforço necessário para o teste das classes componentes desta árvore principalmente as classes folha.

DCC

Definição. Número de classes com as quais a classe está diretamente relacionada. Esta métrica inclui classes que estão diretamente relacionadas através de atributos, ou através de parâmetros nos métodos.

Análise. Assim como CBO revela o número de classes com as quais uma classe está relacionada, ou ainda, das quais depende.

Dados.

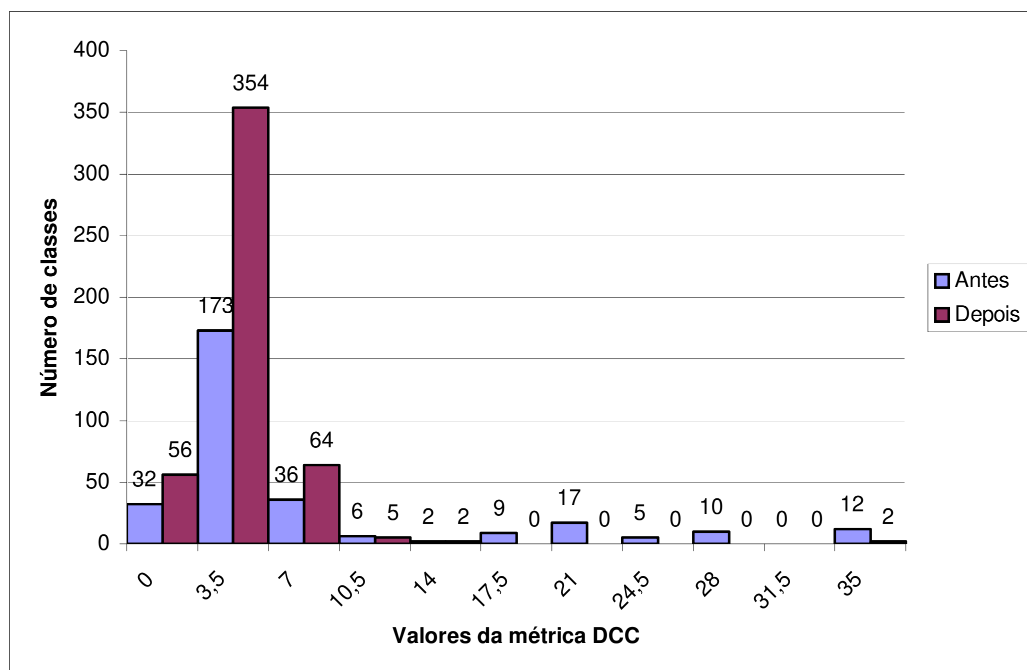


Figura 9.3 - Histograma da distribuição do número de classe em função dos valores da métrica DCC.

Interpretação dos dados. Através da investigação do gráfico apresentado na Figura 9.3, percebe-se, novamente, uma maior concentração de classes à esquerda depois do refatoramento, o que sugere uma diminuição do valor da métrica DCC nas classes do sistema. Ao analisar o valor da média, um resultado semelhante será observado, a média cai de 7.57 no gráfico antes do refatoramento para 3.8 no gráfico depois do refatoramento.

Os gráficos apresentados para análise do acoplamento no sistema de software utilizado como estudo de caso mostram uma visível melhora nos valores das métricas. Como fora citado anteriormente, o próprio aumento no número de classes pode ter contribuído para esta melhoria se classes complexas e possivelmente, fortemente acopladas, foram divididas e deram origem a novas classes mais independentes.

A Figura 9.4 ilustra os resultados obtidos para as medições de acoplamento no sistema antes e depois do projeto de refatoramento. Os valores utilizados para confecção do gráfico foram os valores médios de cada uma das métricas que compõem a categoria. Neste gráfico é possível observar a distância que o sistema se encontra do que seria ideal. No caso de acoplamento, quanto menos acoplamento o sistema apresentar, mais fácil será mantê-lo, adapta-lo, estendê-lo. Deste modo, o gráfico mostra que o sistema aproximou-se mais do ideal, ou seja, a origem do gráfico, após o projeto de refatoramento. Apesar do pequeno aumento no valor médio da métrica DIT, o acoplamento existente no sistema diminuiu consideravelmente.

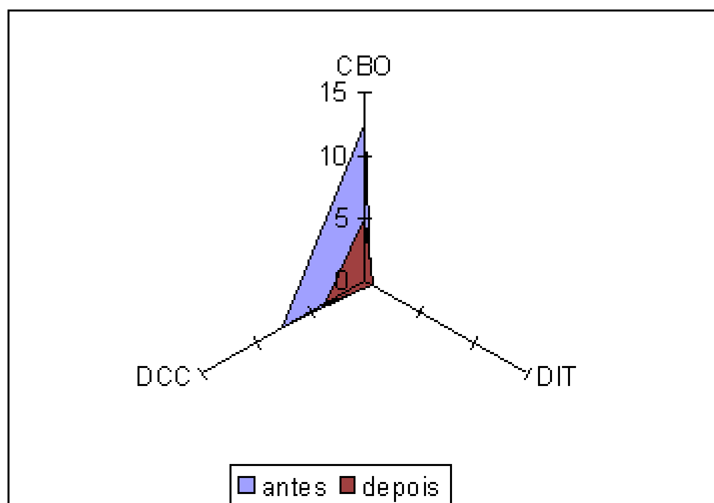


Figura 9.4 - Gráfico com as médias das métricas de acoplamento.

	Antes	*Depois*
CBO	***12,5***	***5***
DIT	***0,63***	***0,66***
DCC	***7,57***	***3,8***

Tabela 9.3 - Valores das médias de acoplamento.

9.3.2 Categoria: coesão

Coesão é uma medida de consistência na funcionalidade de um componente de software [Bansiya, 1998]. É uma característica de projeto desejável já que promove o encapsulamento de informações, a divisão de responsabilidades e ajuda no controle da complexidade. Um baixo índice de coesão pode significar que a classe responde por mais de uma tarefa e assim, poderia ser dividida em outras classes. Quando uma classe não pode ser dividida, esta classe está coesa.

Baixos índices de coesão podem aumentar a complexidade já que torna mais difícil a identificação de responsabilidades das classes. Além disso, uma classe que responde por várias funcionalidades também é difícil manter, pois alterações em uma funcionalidade podem alterar outras. A coesão de componentes de software pode interferir diretamente na facilidade de compreensão e no reuso deste componente.

LCOM

Definição. Esta métrica calcula o percentual de perda de coesão de uma classe. Para isto, analisa a freqüência com que os métodos referenciam os atributos da classe.

Definição Formal [Chidamber, 1994]. Considere a classe C1 que contém n métodos M1, M2, M3,..., Mn. Seja {Ii}=o conjunto de variáveis de instância usadas pelo método Mi. Existem n conjuntos deste I1, I2, I3, ..., In. Seja $P=\{ (I_i, I_j) \mid I_i \cap I_j = \emptyset \}$ e $Q=\{ (I_i, I_j) \mid I_i \cap I_j \neq \emptyset \}$. Se todos os n conjuntos {I1}, ..., {In} são ∅, então P = ∅.

LCOM = |P| - |Q|, se |P| >|Q| ou 0, caso contrário.

Análise. Esta métrica revela o nível de coesão de uma classe. É importante ressaltar que esta métrica mede a porcentagem de falta de coesão assim, um alto valor indica alta taxa de perda de coesão, ou seja, que a coesão é baixa.

Dados.

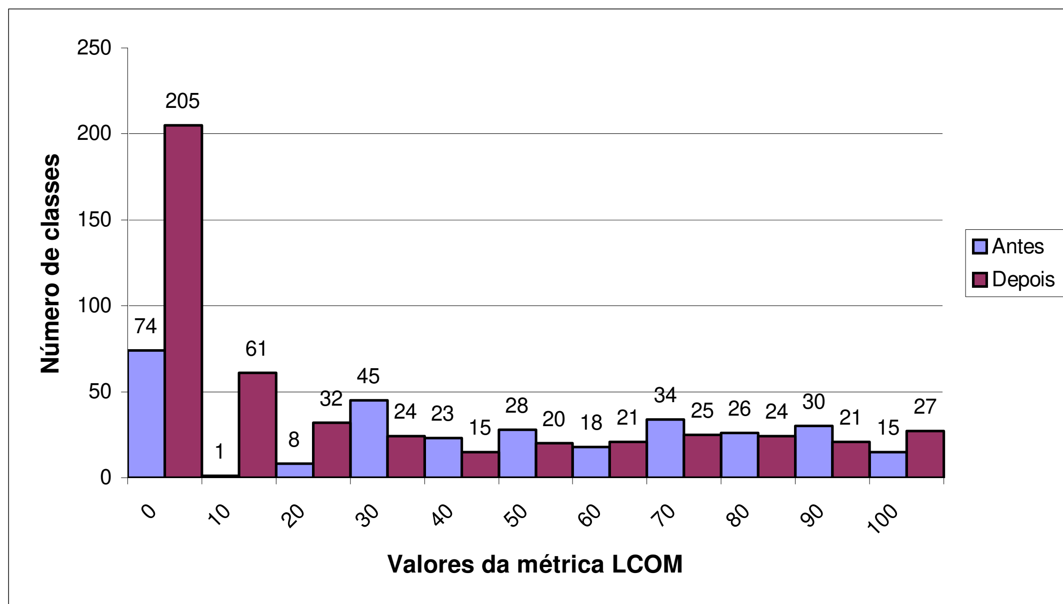


Figura 9.5 - Histograma da distribuição do número de classes em função dos valores da métrica LCOM.

Interpretação dos dados. O gráfico da Figura 9.5 apresenta uma sutil melhora nos valores da métrica LCOM após a execução do projeto de refatoramento. A média de valores passou de 26.87 para 24.52. Percebe-se também uma concentração do número de classes com valores de LCOM mais à esquerda.

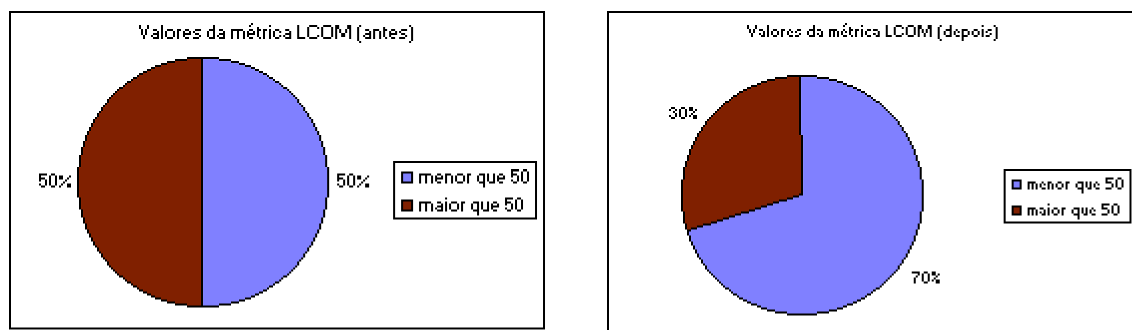


Figura 9.6 - Porcentagem dos valores da métrica LCOM.

No primeiro gráfico tem-se que 50% do número total de classes têm o valor de LCOM menor que 50. O que significa que, em apenas 50% (151) das classes, os atributos das classes são utilizados em mais de um de seus métodos. No segundo gráfico, na Figura 9.6, aproximadamente 70% das classes possuem valores de LCOM menores que 50. Isto sugere que 70% (337) das classes são compostas por métodos coesos, o que está de acordo com o princípio de coesão, onde os métodos devem refletir processamentos sobre os dados essenciais da classe.

CIS

Definição. Calcula o número de métodos públicos da classe.

Análise. Esta métrica é uma tentativa de quantificar a interface de uma classe através da contagem de seus métodos expostos. É considerada uma medida de coesão por quantificar o número de serviços oferecidos por uma classe. Quanto maior o número de métodos públicos, menos coesa uma classe deve ser por prover muitos serviços.

Dados.

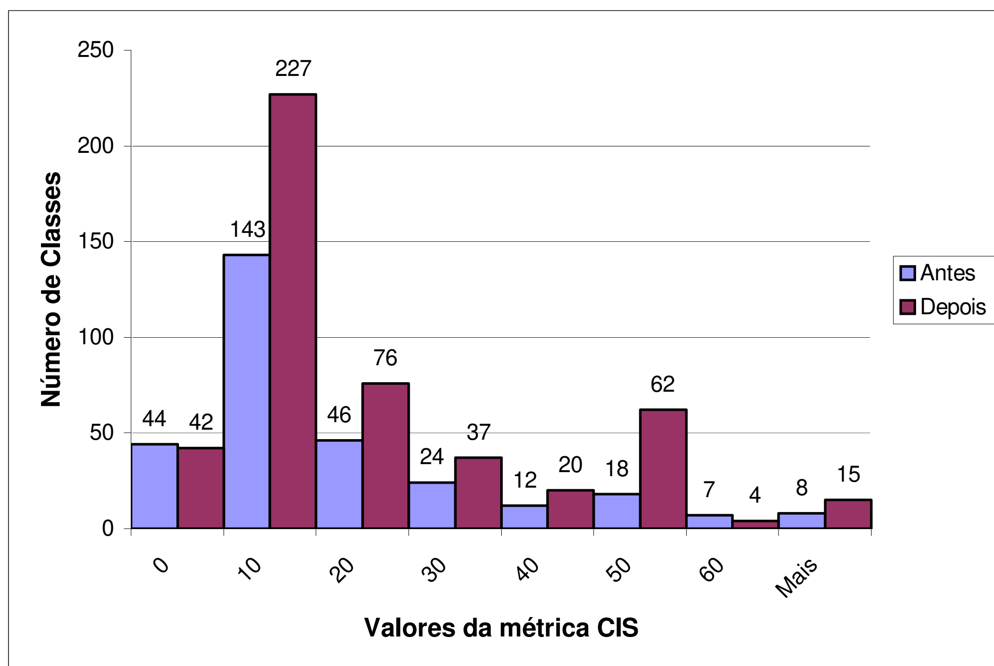


Figura 9.7 - Valores da métrica CIS antes e depois do projeto de refatoramento.

Interpretação dos dados. O gráfico da Figura 9.7 não apresenta grandes diferenças no número de métodos públicos das classes. O conjunto de valores coletados depois do projeto apresenta um formato que sugere a inclusão de dados oriundos de distribuições diferentes. Isto pode ser explicado pelo uso do refatoramento *Divisão de classe de fronteira* no qual uma classe de fronteira é transformada em um *Façade,* o que provoca a aparição de classes com vasta interface e isto causa o aparecimento desta distribuição diferente.

Esta métrica conclui a análise de coesão do sistema. De forma geral, pode-se afirmar que os índices de coesão do sistema melhoraram, porém esta melhora não foi tão expressiva como as melhoras observadas na categoria de acoplamento do sistema.

9.3.3 Categoria: complexidade

A complexidade de classes é avaliada através da complexidade de seus métodos. Esta característica é importante, pois classes complexas dificultam atividades de manutenção por implicar em mais tempo para entendimento, por serem mais susceptível a erros e conseqüentemente, implicar também em mais tempo para as atividades de teste. A depuração do código também é afetada pela complexidade da classe. Assim, complexidade interfere diretamente na facilidade de compreensão do componente de software o que, conseqüentemente interfere também na facilidade de seu reuso e na sua flexibilidade.

MCCABE_Avg

Definição. Calcula a complexidade ciclomática média de uma classe. Complexidade ciclomática consiste no número de caminhos linearmente independentes existentes em um programa. A métrica MCCabe_Avg calcula a média das complexidades ciclomáticas dos métodos da classes.

Análise. Esta métrica objetiva medir a facilidade de testar e manter uma classe. Quanto mais complexa for a classe, mais esforço será necessário para testá-la e mantê-la.

Dados.

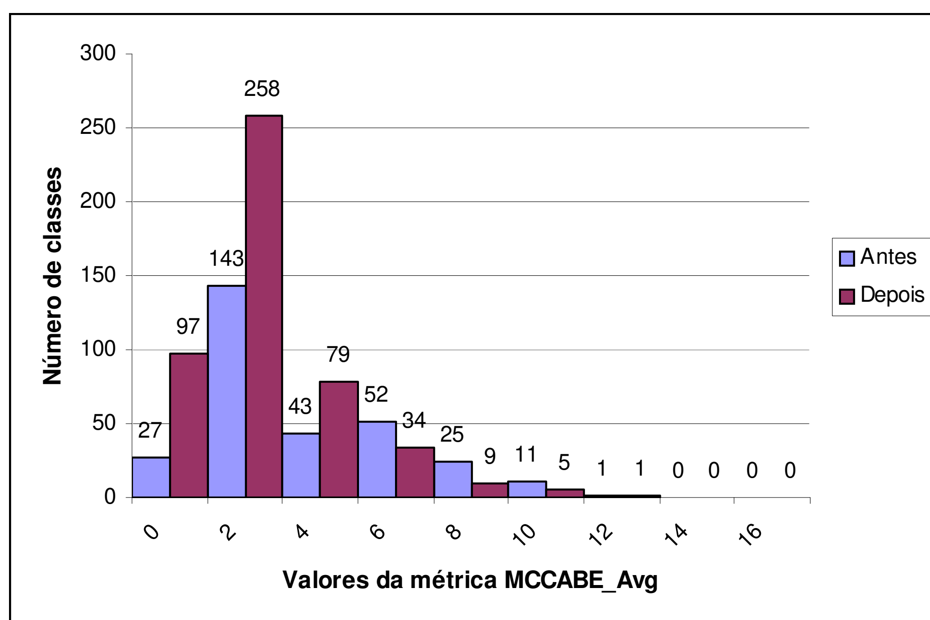


Figura 9.8 - Valores da métrica MCCABE_Avg antes e depois do refatoramento.

Interpretação dos dados. O gráfico da Figura 9.8 revela novamente que o projeto de refatoramento promoveu um acúmulo de classes à esquerda do gráfico, significando que

classes com menores valores de MCCabe_Avg foram criadas. Antes do projeto de refatoramento, aproximadamente 53.6% das classes tinham valores da métrica abaixo de 4, já após o projeto, este percentual sobe para 73.5%.

MCCABE_Max

Definição. Calcula a complexidade ciclomática máxima observada em uma classe. Este cálculo é feito considerando apenas a complexidade do método que apresentou o maior valor de complexidade ciclomática.

Análise. Esta métrica revela a complexidade máxima apresentada por uma classe. O objetivo é analisar os índices mais altos de complexidade como forma de obter um retrato mais fiel do software sendo analisado.

Dados.

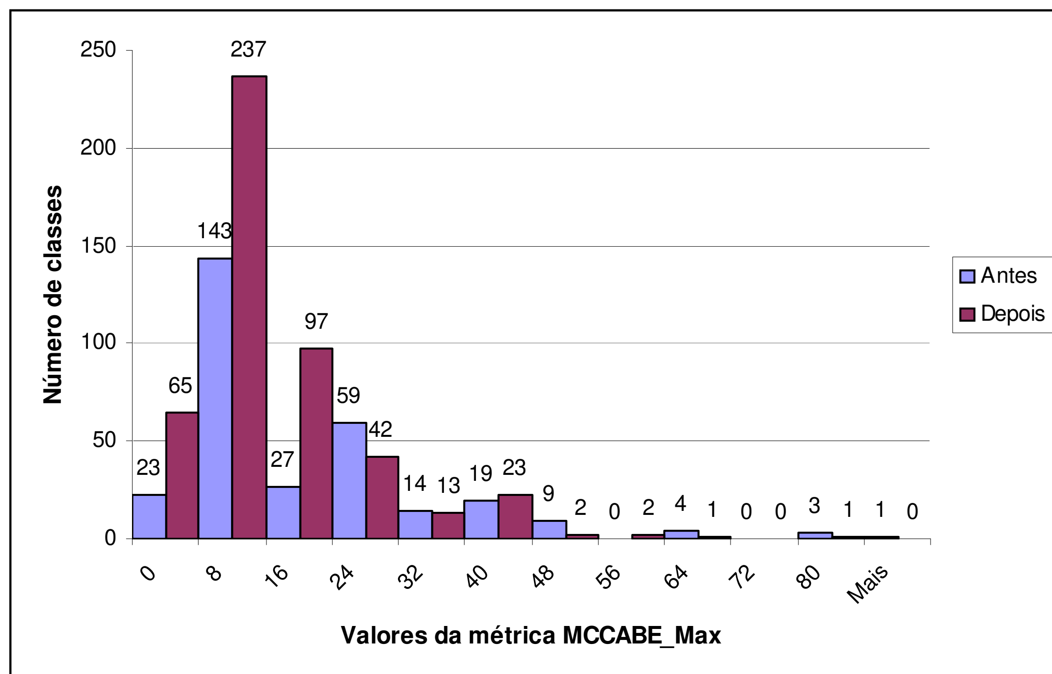


Figura 9.9 - Valores da métrica MCCABE_Max antes e depois do refatoramento.

Interpretação dos dados. Através do gráfico na Figura 9.9 percebe-se que também a complexidade máxima de cada classe foi reduzida. Antes do projeto, 88% das classes tinham valor de McCabe_Max menor que 40, enquanto que depois este percentual sobe para 94%.

WMC

Definição. O valor desta métrica é calculado através do somatório das complexidades dos métodos de uma classe[3]. Pode ser entendida como a complexidade total de uma classe.

Análise. A medida de complexidade de uma classe pode ser interpretada como uma medida de tempo e esforço necessário para manter a classe. Alto índice da métrica WMC pode revelar tanto um elevado número de métodos como métodos com altos índices de complexidade. Nenhuma das situações são desejáveis em sistemas OO, pois podem provocar impactos negativos nos índices de acoplamento, coesão e encapsulamento.

Dados.

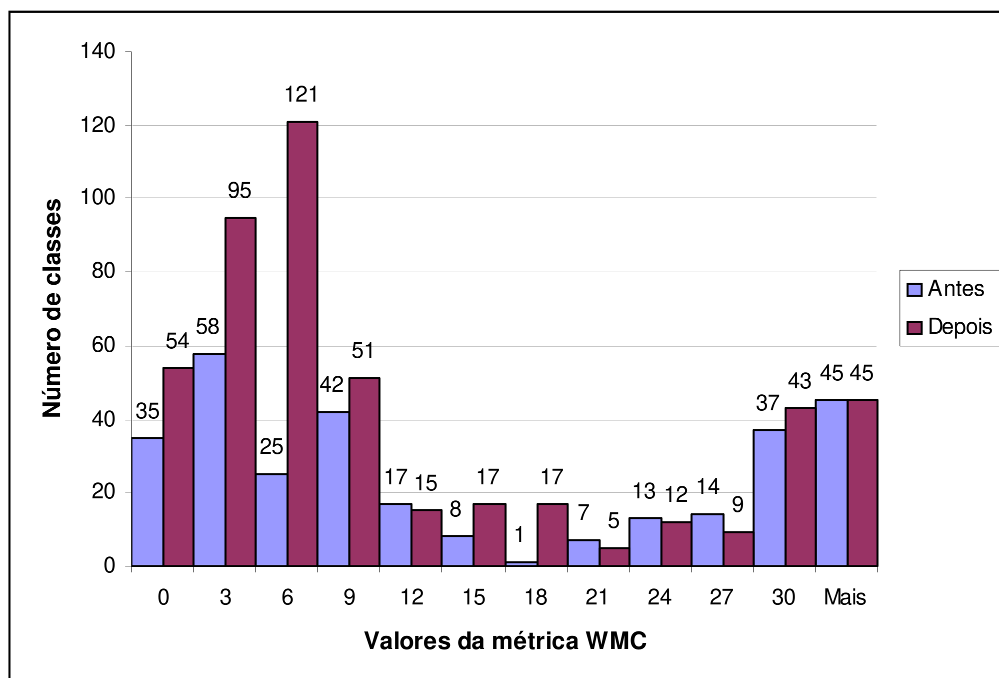


Figura 9.10 - Valores da métrica WMC antes e depois do refatoramento.

Interpretação dos dados. De acordo com o gráfico da Figura 9.10, antes do projeto de refatoramento, o sistema apresentava 58.6% de suas classes com complexidade menor do que 15, já após o término do projeto, este número subiu para 70%. Lembrando que, de acordo com a ferramenta utilizada, o valor desta métrica reflete o número de métodos presentes na classe. Assim, entende-se que 70% das classes contem menos de 15 métodos, o que sugere que o

3 Na definição formal de Chidamber e Kemerer a forma de calcular esta complexidade é uma decisão de implementação. A complexidade não é definida como forma de aumentar as possibilidades de aplicação da métrica. Na ferramenta utilizada para realizar o cálculo desta métrica, é atribuído valor 1 à complexidade de cada método.

sistema é composto, em sua maioria, por classes que provêem abstrações e funcionalidades específicas. Isto pode ser observado nos gráficos apresentados na Figura 9.11.

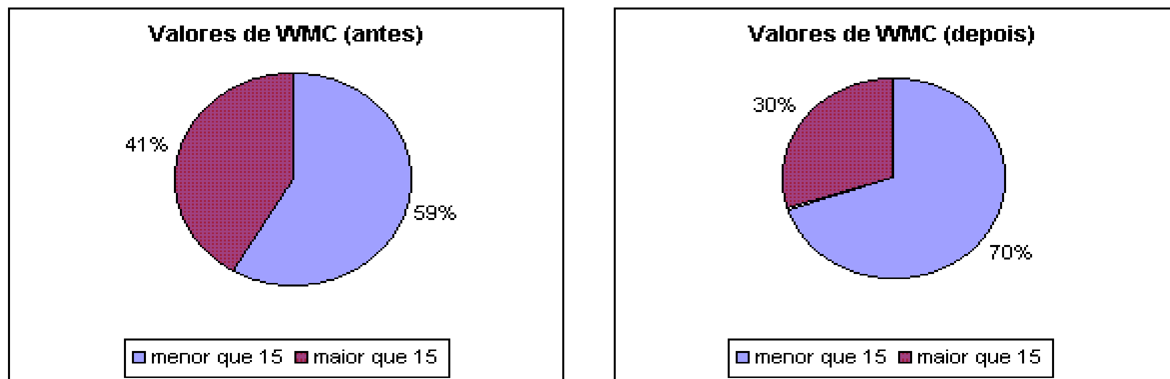


Figura 9.11 - Valores de WMC antes e depois do refatoramento.

RFC

Definição. O valor desta métrica é igual ao conjunto de respostas de uma classe. Ou seja, o somatório dos métodos da classe com o número de métodos invocados por cada um de seus métodos no desempenhar de sua atividade.

Definição Formal [Chidamber, 1994]. RFC = |RS|

RS = {M} ∪ {Ri}, para todo i.

Onde Ri = conjunto de métodos invocados[4] pelo método i e {M} = conjunto de todos os métodos da classe.

Análise. Se um número elevado de métodos estiver envolvido na resposta a uma mensagem, as atividades de teste e depuração se tornarão mais complicadas. Quanto maior o número de métodos que podem ser invocados por uma classe, maior a sua complexidade.

Dados.

4 Este conjunto considera apenas o primeiro nível de métodos invocados devido a considerações práticas envolvidas na coleta da métrica.

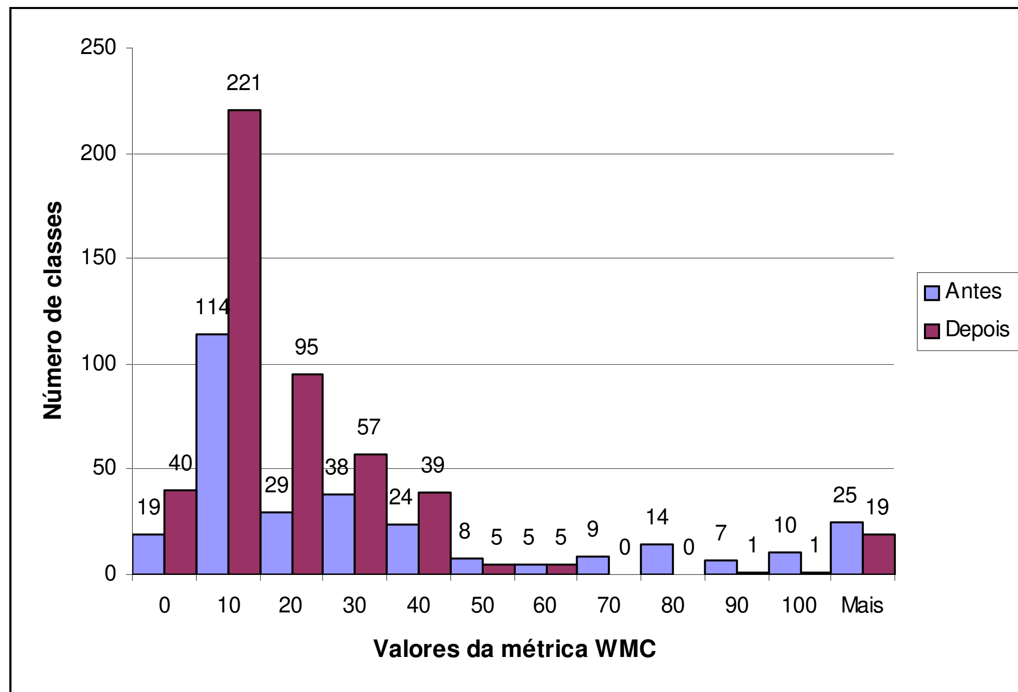


Figura 9.12 - Valores da métrica RFC antes e depois do refatoramento.

Interpretação dos dados. O gráfico que apresenta a distribuição do número de classes em função do valor da métrica RFC (vide Figura 9.12) sugere uma diminuição expressiva na complexidade das classes do sistema depois da aplicação do refatoramento. Enquanto antes do projeto de refatoramento 74% das classes têm valores de RFC menores que 50, depois este valor sobe para 94%. É importante lembrar que valores altos de RFC podem surgir tanto porque a classe tem muitos métodos como por invocar muitos métodos.

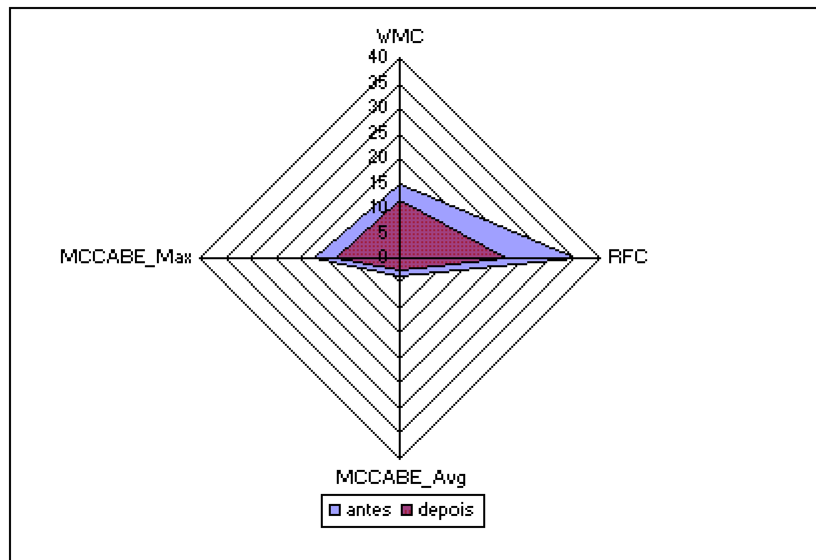


Figura 9.13 - Gráfico com a média das métricas de complexidade antes e depois do refatoramento.

	Antes	Depois
WMC	14.82	11.64
RFC	35.27	21.14
MCCABE_Avg	3.62	2.42
MCCABE_Max	17.01	12.72

Tabela 9.4- Valores das médias das métricas de Complexidade antes e depois do refatoramento.

A Figura 9.13 apresenta uma análise geral das métricas de complexidade do sistema. A diminuição da complexidade pode facilitar o entendimento do código, sua manutenção, sua adaptabilidade e a atividade de teste do sistema. Deste modo, quanto menos complexidade apresentar o sistema, mais fácil será a sua manutenção.

De acordo com o gráfico, as medidas de complexidade do sistema diminuíram. A métrica que mais alteração sofreu foi RFC, o que sugere um indicativo de onde os refatoramentos atuam mais intensamente. A introdução de interfaces, classes, hierarquias, propostas pelos refatoramentos promovem uma diminuição nos índices de resposta das classes. Assim, muito embora a complexidade de uma classe não seja endereçada diretamente pelos refatoramentos, ela sofre alterações, o que propõe que o projeto do sistema influencia a complexidade da implementação de suas classes.

9.3.4 Categoria: polimorfismo

Esta categoria revela uma tentativa de quantificar a possibilidade de que existam chamadas polimórficas no sistema.

Polimorfismo não é uma medida de qualidade de sistemas OO, no entanto bons projetos OO provavelmente fazem uso deste recurso numa tentativa de diminuir o acoplamento entre as classes do sistema, bem como para aumentar a capacidade de adaptação do sistema a novas situações.

NOP

Definição. Número de métodos polimórficos (virtuais ou puramente virtuais) novos ou herdados.

Análise. A presença de métodos virtuais ou puramente virtuais em uma classe indica a possibilidade de que haja chamadas polimórficas no sistema. O número destes métodos em uma classe pode ser interpretado como a quantificação desta possibilidade para cada classe. Quanto mais métodos virtuais ou puramente virtuais existirem em uma classe, mais possibilidade existe de que ela seja utilizada em chamadas polimórficas.

Dados.

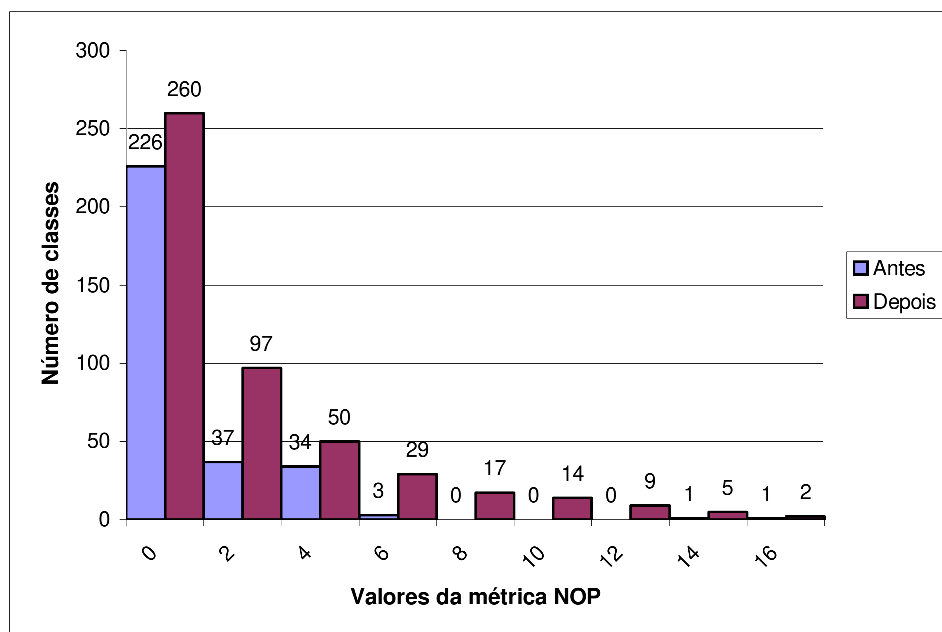


Figura 9.14 - Valores da métrica NOP antes e depois do refatoramento.

Interpretação dos dados. O gráfico mostra um aumento discreto no número de classes com métodos polimórficos. É importante lembrar que o refatoramento *Diminuição de*

acoplamento através de TADs propõe a introdução de TADs para promover o desacoplamento dos módulos do sistema. Em C++, estes TADs são definidos como classes com métodos virtuais. Esta é uma das razões para o aumento no número de classes com métodos polimórficos e, conseqüentemente, um aumento na possibilidade de chamadas polimórficas existirem no sistema.

9.4 Conclusões

Os gráficos apresentados demonstram as melhorias provocadas pelos refatoramentos no estudo de caso deste trabalho através da análise das métricas coletadas antes e depois de sua execução. Em grande parte dos gráficos apresentados observaram-se melhorias nos valores expressos pelas métricas do sistema.

As melhoras mais significativas foram observadas no acoplamento, exceto na métrica DIT, e nas medidas de complexidade. O aumento do número de classes sugere que classes com responsabilidades demais foram divididas em classes menores e com funcionalidades mais bem definidas. As classes subdivididas podem ter dividido também as suas dependências. Isto pode explicar o impacto maior nas medidas de acoplamento.

Os refatoramentos não foram propostos e realizados com o objetivo de melhorar as métricas de acoplamento, coesão ou complexidade do sistema. Eles se propõem a resolver, em sua maioria, problemas de integração de módulos, iteração entre o sistema e o ambiente hospedeiro, acessibilidade e visibilidade de módulos. Sendo assim, os bons resultados obtidos não são fruto de modificações pensadas com este objetivo.

Conseqüentemente, as características de qualidade contempladas pelos refatoramentos não são completamente endereçadas pelas métricas utilizadas na avaliação. As métricas OO tentam mensurar acoplamento, coesão, mas se detém ao nível de projeto não se adequando aos níveis mais altos como o acoplamento entre módulos do sistema.

O sistema é entendido como um conjunto de classes pelas métricas OO e não como um conjunto de módulos que são compostos por diversas classes. As métricas analisadas são eficientes para analisar os benefícios obtidos em termos de classes, mas não revelam as variações na integração dos módulos.

As métricas revelam um aspecto dos benefícios obtidos com a aplicação dos refatoramentos, mas não revelam as relações entre os módulos do sistema. A integração entre os diversos módulos do sistema é contemplada pelos refatoramentos, mas os benefícios obtidos neste campo não são indicados por este conjunto de métricas, pois elas se detêm à

interação entre classes. Métricas que avaliem a integração entre módulos do sistema não foram calculadas pela indisponibilidade de ferramentas que as coletem.

10 CONCLUSÕES

Neste trabalho investigamos o projeto de reestruturação de um sistema de software de grande porte com o objetivo de identificar meios para lidar com os problemas gerados pela Entropia de Software em tais sistemas. Os refatoramentos identificados foram aplicados em um sistema de software disponível comercialmente com o objetivo de melhorar sua estrutura para facilitar as atividades de manutenção e, principalmente, sua migração para outra plataforma.

A idéia de refatoramentos no combate à Entropia de Software já havia sido utilizada em outras situações. Os refatoramentos propostos por Fowler [Fowler, 1999], por exemplo, resolvem problemas mais simples e não abrangem o software por inteiro ao passo que os antipadrões [Brown, 1998] focam na identificação do problema e propõem uma solução, mas não enfatizam o meio de atingi-la. Em nenhum dos exemplos citados encontramos ajuda concreta na reestruturação de um sistema de grande porte.

Assim, ao desenvolver o projeto de reestruturação fomos capazes de identificar problemas que podem ser encontrados em outros sistemas de software e propor soluções e, também, formas para implementá-las. Associado aos problemas, soluções e meios de atingi-las, fazemos uma análise deste dois primeiros como forma de identificar fraquezas e ganhos estruturais do sistema.

Os refatoramentos apresentados neste trabalho são independentes de linguagem de programação. Os problemas encontrados e soluções propostas não restringem-se à linguagem de programação do estudo de caso, são problemas de **projeto** de sistemas orientados a objetos, não contemplam detalhes de implementação. Deste modo, podem ser utilizados em projetos de refatoramento de sistemas OO escritos em outras linguagens de programação.

10.1 Problemas e observações

Durante a definição dos refatoramentos apresentados neste trabalho sentimos a necessidade de tornar os passos do mecanismo de aplicação do refatoramento mais gerais que os passos dos refatoramentos definidos por Fowler [Fowler, 1999]. Os refatoramentos de larga escala lidam com problemas mais abrangentes no que se refere ao número de classes envolvidas, mais genéricos no que se refere à caracterização do problema e por estas razões não é adequado propor mecanismos tão específicos como os apresentados em [Fowler, 1999]. A aplicabilidade dos refatoramentos seria limitada a problemas iguais aos encontrados no

sistema e não apenas semelhantes. Deste modo, os mecanismos dos refatoramentos de larga escala são menos detalhados por vislumbrarem uma maior área de aplicação.

Uma das grandes dificuldades enfrentadas no desenvolvimento do trabalho foi a carência de documentação atualizada do sistema. Isto provocou a inclusão de uma etapa na metodologia de desenvolvimento do projeto específica para a aquisição de conhecimento do código do sistema. Mesmo apoiada em técnicas de engenharia reversa a etapa de conhecimento do sistema revelou-se uma das grandes dificuldades encontradas no desenvolvimento do trabalho.

Outro problema enfrentado foi a carência de um conjunto de testes que proporcionasse um mínimo de segurança na aplicação de técnicas de refatoramento. O conjunto de testes para exercitar o sistema fez parte do desenvolvimento do trabalho que, mesmo tendo sido executado com o apoio de terceiros, teve forte impacto no projeto.

Estes problemas enfrentados foram úteis por enriquecer a metodologia de trabalho utilizada. Nela, estes possíveis problemas são ressaltados e as soluções adotadas são apresentadas. Assim, a metodologia pode servir como referência para outros projetos de refatoramento.

Uma fraqueza apresentada pela metodologia é a carência de sugestões de meios para investigar o código do sistema atrás de possíveis problemas. Na metodologia apresentada e aplicada neste trabalho, em nenhum momento indicamos meios de automatizar a identificação de problemas ou mesmo, sugerimos regras ou perguntas a serem feitas que pudessem facilitar a identificação dos problemas do sistema. Nós identificamos trechos do sistema que apresentam problemas, mas não indicamos meios de chegar a esta parte do projeto do sistema. Na experiência relatada em [Poole, 2001] encontramos uma sutil sugestão de que indicativos de trechos do sistema que requerem a aplicação de técnicas de refatoramento podem ser encontrados através da análise das estatísticas de erros. No entanto, no projeto de refatoramento conduzido neste trabalho este tipo de análise não foi possível devido à inexistência de tais estatísticas. Assim, a técnica utilizada para identificar trechos que poderiam ser candidatos a refatoramentos foi a investigação do código fonte.

Consideramos os resultados alcançados com o trabalho como satisfatórios, pois julgamos que conseguimos identificar alguns problemas que podem ser encontrados em outros sistemas de software e, conseqüentemente, as análises apresentadas e soluções propostas podem ser úteis na correção de tais problemas. Além disso, a leitura dos refatoramentos pode contribuir com a identificação de problemas existentes no sistema através das análises contidas neste trabalho. Sobre a metodologia adotada, reconhecemos suas

falhas, no entanto, identificamos contribuições valiosas no que se refere a lições aprendidas e que devem ser observadas em outros projetos de refatoramento.

10.2 A validação dos refatoramentos

A análise das métricas do sistema antes e depois do projeto de refatoramento apresenta ganhos em alguns aspectos do projeto do sistema, mas também mostra pouca variação em outros. Os ganhos obtidos revelam-se importantes porque em nenhum momento os refatoramentos foram direcionados à melhora de métricas específicas e sim, direcionados à resolução de problemas identificados.

A carência de métricas que avaliem especialmente os benefícios obtidos com a aplicação de técnicas de refatoramento foi uma dificuldade encontrada durante a validação do trabalho. A validação foi feita com base nas melhorias indicadas pelas métricas no projeto do sistema como um todo. Porém, outros aspectos poderiam ter sido avaliados caso dispuséssemos de ferramentas mais apropriadas como, por exemplo, a análise da cooperação entre os módulos do sistema. Outra medição que poderia enriquecer a validação do trabalho seria a comparação do esforço dispensado na correção de erros ou introdução de novas funcionalidades antes e depois do projeto ou ainda a comparação dos índices de surgimento de erros no sistema. No entanto, este tipo de comparação exigiria um gasto de tempo considerável do qual não podíamos dispor.

10.3 Sumário das contribuições

Este trabalho define um conjunto de refatoramentos de larga escala que contribuem com a reestruturação de sistemas de software orientados a objetos. As seguintes contribuições podem ser listadas:

- Sumário de problemas e soluções que podem ser enfrentados em projetos de refatoramentos de sistemas de grande porte.
- Identificação de problemas de projeto de sistemas OO que podem ser úteis na identificação de problemas de outros sistemas de software.
- A metodologia de desenvolvimento do projeto de refatoramento.
- O padrão para apresentação do refatoramento já que foi uma evolução do padrão proposto por Fowler [Fowler, 1999] de modo a acomodar as necessidades do refatoramento de software de larga escala.

10.4 Trabalho futuro

A conclusão deste trabalho permite identificar alguns encaminhamentos futuros no sentido de estender a definição e validação de refatoramentos em larga escala. Outros trabalhos ainda são necessários de modo a definir novos problemas e soluções e também para comprovar a aplicabilidade dos refatoramentos aqui propostos.

Em segundo lugar, a definição de novos refatoramentos pode também contemplar áreas específicas como melhoras de desempenho, otimização no uso de recursos, entre outros. É possível também identificar refatoramentos para problemas encontrados em áreas mais específicas como na comunicação com bancos de dados, em sistemas de tempo real, tolerantes a falhas.

Em terceiro lugar é preciso investigar metodologias para o desenvolvimento de projetos de refatoramentos como o que foi desenvolvido neste trabalho. Em quarto lugar, a definição de métricas de refatoramento que melhor retratem as melhoras obtidas com tais técnicas é necessária para o fortalecimento dos estudos destas técnicas.

Por fim, deve-se estudar a construção de ferramentas de apoio a tais projetos que possibilitem a identificação de fraquezas estruturais do sistema e até a aplicação dos refatoramentos de forma automática.

11 APÊNDICE A - IDENTIFICAÇÃO DOS GRANDES PACOTES DO LIGHTBASE SERVER

Antes que o código de um software possa ser mexido, refatorado ou portado é extremamente importante que a pessoa responsável por este trabalho tenha um bom conhecimento dele. Desta forma, o objetivo deste documento é prover um avanço no sentido de um maior entendimento do produto.

A primeira etapa aqui é a identificação das dlls (dinamic link libraries), libs (static link libraries) e exes (executáveis) que fazem parte do LightBase Server bem como a sua funcionalidade. Esta etapa será importante para a compreensão da responsabilidade principal de cada módulo em especial.

A próxima etapa é a busca de um conhecimento mais vertical, no sentido de conhecer o interior dos módulos. Para isso, faz-se um levantamento das principais atividades desempenhadas pelo servidor e lista-se os módulos e as classes destes módulos envolvidas no processo. Esta atividade proporcionará uma noção maior do acoplamento entre os módulos e entre as classes. Todos estes estudos são extremamente importantes para possibilitar o entendimento do código. Apenas após a obtenção de um entendimento razoável deve-se passar para atividades mais avançadas, como a mudança do código.

11.1 A identificação dos módulos

O módulo central do servidor LightBase é a dll LBS. Apesar de usar outros módulos, este é o ponto central do produto. De modo que, este será o primeiro a ser apresentado.

11.2 Lbs

Este módulo é responsável pela execução de todas as atividades do servidor: manipulação de bases, controle de usuários, controle de concorrência, indexação/desindexação, entre outras. Para desempenhar o seu papel, o LBS relaciona-se com outros módulos que têm responsabilidades mais específicas e que serão apresentados mais adiante.

11.3 LbwServ

Este é o programa servidor. Sua responsabilidade é receber as requisições dos clientes, repassá-las aos módulos que desempenham a atividade, e, finalmente, responder ao cliente.

11.4 AppManager

Esta é a lib responsável pela internacionalização do produto. Aqui, é importante uma explicação um pouco mais detalhada. Esta lib é invocada por todos os módulos que necessitam apresentar alguma mensagem para o usuário. Estas mensagens estão todas disponíveis em arquivos de inicialização que são copiados no momento da instalação. Ao receber uma requisição de uma mensagem específica, AppManager lê o arquivo de inicialização previamente configurado e devolve a informação requisitada. Estas mensagens são unicamente identificadas por suas chaves nos arquivos .ini.

No momento da instalação apenas o arquivo referente ao idioma escolhido pelo usuário é copiado.

11.5 IPWorks

Esta dll foi comprada e é a responsável pela comunicação através do protocolo HTTP.

11.6 LiParser

A responsabilidade deste módulo é realizar a análise sintática da informação. Tanto na indexação como no brilha palavra, esta dll é responsável pelo reconhecimento de tokens.

11.7 Slot

Esta lib é responsável por informações acerca da base. Estas informações são definidas pelas aplicações que a utilizam. O Lbs só usa Slot para saber os grupos aos quais um campo pertence. Esta informação é útil no momento da ordenação de grupos.

11.8 Sort

Esta lib é responsável pela ordenação. O servidor é capaz de ordenar os registros de uma base, as repetições de um campo ou grupo. As chaves usadas para estas ordenações podem ser

compostas ou não. Uma chave ser composta significa que ela é construída a partir de mais de um campo da base de dados. É importante lembrar que estes campos também são priorizados.

Sort utiliza o conceito MMF (*Merge Maped File*), ou seja de arquivo mapeado em memória. Primeiro um arquivo é ordenado segundo o algoritmo *Quick Sort*, depois os arquivos são ordenados segundo o algoritmo *Merge Sort*.

11.9 Comparator

Comparator é usado nas atividades relativas a ordenação (Sort). Esta lib compara duas chaves que podem ser ou não compostas. Para tal, a lib conhece a lei de formação da chave e num sistema de *callback* consegue identificar a relação entre elas.

11.10 LT

Aqui se tem toda a manipulação do sistema de índices. As árvores binárias (B-Trees) são implementadas por CtawLib que nada conhece do significado da informação armazenada. A interpretação, o conceito de ocorrências está todo em LT. Esta lib é responsável, assim, pelo sistema de índices.

No momento de uma indexação, é Lt que sabe os índices que devem ser gerados e as informações que devem ser indexadas.

11.11 HtmlTools

As ferramentas para manipulação de html estão concentradas aqui. Nesta lib é realizada a conversão de HTML para ASP (*Active Server Pages*).

11.12 CtawLib

Esta é a lib que implementa as árvores binárias (B-Trees) dos índices do Lightbase. É uma lib comprada, porém seus arquivos fonte estão disponíveis.

11.13 Compress

Esta é uma lib que faz a compressão dos dados. Serve como uma casca de Alfc.

11.14 Alfc

É uma lib comprada, inclusive seus direitos, é a responsável pela compressão de dados.

11.15 LbStart

LbStart é a dll responsável pela personalização das cópias servidor. Realiza a verificação de chaves de ativação, números de série.

11.16 LiSvc

Esta lib faz com que o servidor execute como um serviço do Windows NT.

11.17 RpcStuff

Tratamento dos dados enviados e recebidos pela rede. Aqui os dados são serializados/deserializados depois de serem recebidos, ou antes, de serem enviados.

11.18 Li

Esta é uma lib genérica. Contém várias funcionalidades utilitárias. Faz tratamento e operações sobre campos data, hora, binário.

11.19 LiFile

É uma abstração de stdio. A leitura e escrita dos arquivos da base estão concentradas aqui. Inclui conceitos como cabeçalho, registro, etc.

11.20 Crypt

Responsável pela criptografia dos dados.

A Figura 11.1 apresenta o grafo de dependências do servidor LightBase.

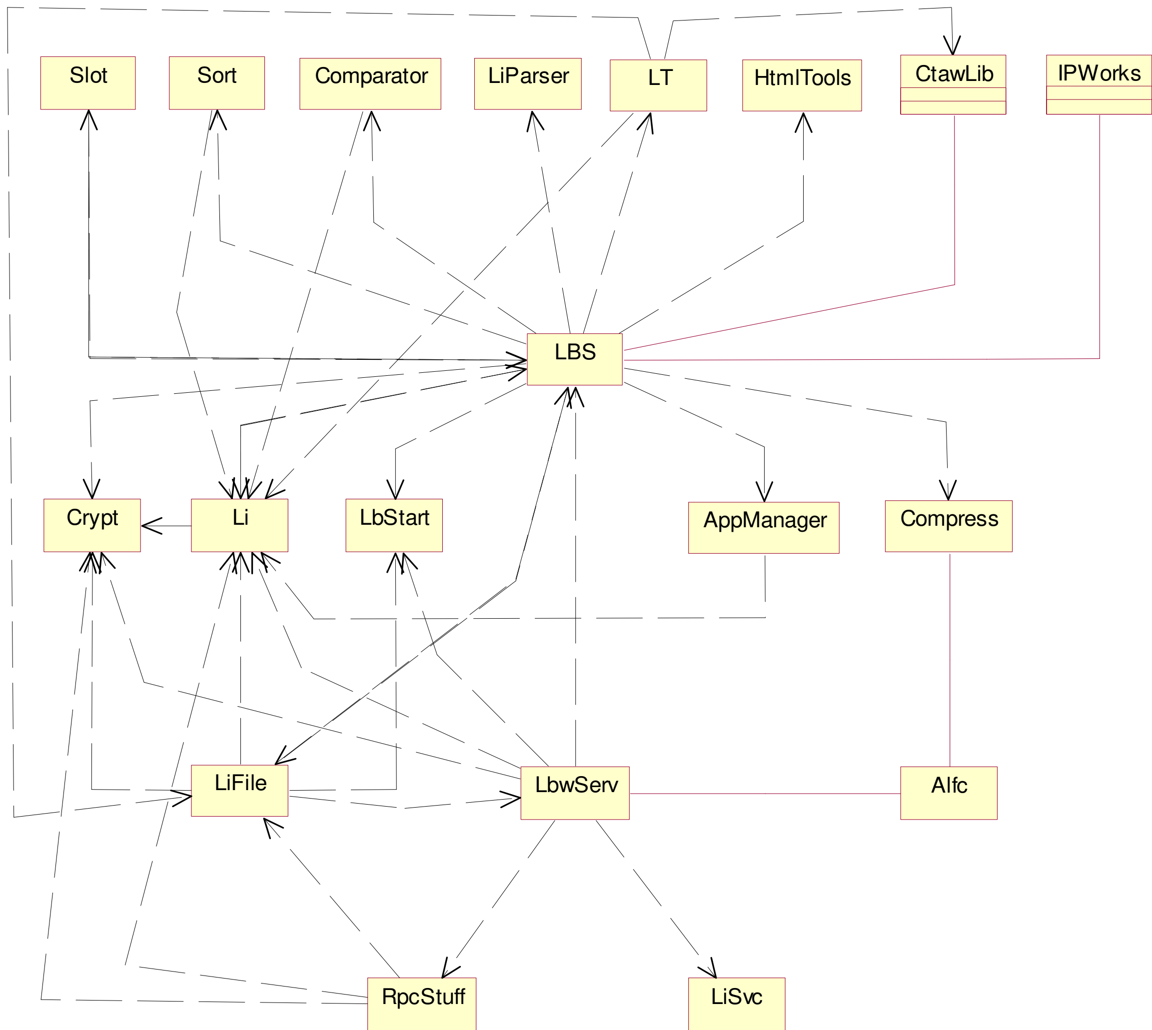


Figura 11.1 - Apresentação de todos os módulos do LightBase Server e suas visibilidades.

12 REFERÊNCIAS BIBLIOGRÁFICAS

[Aguiar, 2001] Aguiar, G. S.; Sauvé, J. P. Definição e Validação de Software em Larga Escala. *In* VI WTES – Workshop de Teses em Engenharia de Software. Rio de Janeiro, 2001.

[Bansiya, 1998] Bansiya, J. A Hierarchical Model for Quality of Assessment of Object-Oriented Designs. Tese Ph.D. Computer Science Dept. Univ. Alabama em Huntsville, Mai. 1998.

[Barkley, 1993] Barkley, J. Comparing Remote Procedure Calls. Technical Report NISTIR 5277. National Institute of Standards and Technology (Systems and Software Technology Division), 1993.

[Bass, 1998] Bass, L.; Clements, P.; Kazman, R. Software Architecture in Practice. Addison-Wesley, 1998.

[Beck, 1998] Beck, K.; Gamma, E. Test Infected: Programmers Love Writing Tests. Java Report. Vol 3, No. 7, p. 37-50. Jul. 1998

[Beck, 1999] Beck, K. Extreme Programming Explained Embrace Change. Addison-Wesley, 3. imp., 1999.

[Bisbal, 1999] Bisbal J. *et al.* Legacy Information Systems: Issues and Directions. *In* IEEE Software Magazine Set./Out. 1999, p. 103-111.

[Boehm, 1981] Boehm, B. Software Engineering Economics. Prentice Hall, 1981.

[Booch, 1994] Booch, G. Object-Oriented Analysis and Design. Benjamin/Cummings, Santa Clara, Califórnia, 2. ed., 1994.

[Brown, 1998] Brown, W. J. *et al.* Anti Patterns: Refactoring Software, Architectures and Projects in Crisis. John Wiley & Sons, 1998.

[Cantata] Cantata++, http://www.iplbath.com/products/tools/pt400.shtml

[Chidamber, 1991] Chidamber, S. R.; Darcy, D. P.; Kemerer, C. F. Towards a Metrics Suite for Object Oriented Design. *In* Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'91), Phoenix, AZ, p. 197-211, Nov. 1991.

[Chidamber, 1994] Chidamber, S. R.; Darcy, D. P.; Kemerer, C. F. A Metrics Suite for Object Oriented Design. *In* IEEE Transactions on Software Engineering, Vol. 20, No. 6, p. 476-493, Jun. 1994.

[Chidamber, 1998] Chidamber, S. R.; Darcy, D. P.; Kemerer, C. F. Managerial Use of Metrics for Object-Oriented Software. *In* IEEE Transactions on Software Engineering, Vol. 24, No. 8, p. 629-639, Ago. 1998.

[CppUnit] CppUnit,

[DCE, 1991] OSF DCE 1.0 Application Development Guide. Relatório Técnico, Open Software Foundation, Dezembro 1991.

[Ducasse, 1999] Ducasse, S.; Richner, T.; Nebbe, R. Type-check elimination: Two object-

oriented reengineering patterns. *In* WCRE'99 (6th Working Conference on Reverse Engineering), p. 157–168. IEEE Computer Society Press, Out. 1999.

[Eick, 2001] Eick, S. G. *et al.* Does Code Decay? Assessing the Evidence from Change Management Data. *In* IEEE Transactions on Software Engineering, Vol. 27, no. 1, 2001.

[Fowler, 1999] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[Gamma, 1994] Gamma, E. *et al.* Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[Gamma, 1999] Gamma, E.; Beck, K. Junit: A Cook's Tour. Java Report. P. 27-38. Mai. 1999.

[Han, 1997] Han, J. Supporting Impact Analysis and Change Propagation in Software Engineering Environments. *In* 8th Intl. Wokshop on Software Technology and Engineering Practice (STEP'97), Londres, Inglaterra, p. 172-182. Jul. 1997.

[ISO, 1991] [ISO, 1991] ISO Remote Procedure Call Specification. ISO/IEC CD 11578 N6561, ISO/IEC. Nov. 1991.

[Kiran, 1997] Kiran, G. A.; Haripriya, S.; Jalote, P. Effect of Object Orientation on Maintainability of Software. *In* International Conference on Software Maintenance (ICSM'97), Italy, p. 114-121. Out. de 1997.

[Larman, 1997] Larman, C. Applying UML and Patterns An Introduction to Object-Oriented Analysis and Desing. Prentice Hall, 1997.

[LightBase] LightBase, http://www.lightinfocon.com.br

[Lindvall, 1997] Lindvall, M. An empirical study of requirements-driven analysis in objected software evolution. Tese PhD. Linkoping University, Suécia, 1997.

[Miceli, 1999] Miceli, T.; Sahraoui, H.; Godin, R. A Metric Based Technique For Design Flaws Detection And Correction *In* IEEE Automated Software Engineering Conference (ASE'99), 1999.

[Mooney, 1997] Mooney, J. D. Bringing Portability to the Software Process. Relatório Técnico TR 97-1 University West Virginia Dept. of Statistics and Computer, Morgantown MW, 1997.

[Munson, 1998] Munson, J.C.; Elbaum, S. G. Code churn: A measure for estimating the impact of code change. *In* International Conference on Software Maintenance (ICSM'98), Bethsda, MD. p. 24-31. Nov. 1998.

[Opdyke, 1992] Opdyke, W. F. Refactoring Object-Oriented Frameworks. Tese Phd. Universidade de Illinois, Urbana-Champaign. 1992.

[Opdyke, 1999] Opdyke, W. F. Refactoring reuse and reality. Lucent Technologies http://st-www.cs.uiuc.edu/users/opdyke/wfo.990201.refac.html, 1999.

[Pfleeger, 1998] Pfleeger, S. L. Software Engineering – Theory and Practice. Prentice Hall, 1998.

[Poole, 2001] Poole, C.; Huisman, J. W. Using Extreme Programming in a Maintenance Environment. *In* IEEE Software. Vol. 18, No. 6, p. 42-50. Nov./Dez., 2001.

[Rational] Rational Rose, http://www.rational.com

[Roberts, 1999] Roberts, D. B. Practical Analysis for Refactoring. Tese PhD, University of Illinois, Urbana Champaign, 1999.

[Sneed, 1995] Sneed, H. Planning the reengineering of legacy systems. *In* IEEE Software. IEEE Computer Society Press. Jan. 1995.

[Stevens, 1998] Stevens, P.; Pooley, R. Systems reengineering patterns. *In* Software Engineering Notes, ACM Press. Vol. 23 n. 6. Nov. 1998.

[SUN, 1990] Sun Microsystems Inc. Network Programming Guide, Rev. A. 27 de março 1990.

[SUN, 1991] Solaris ONC: Design and Implementation of Transport-Independent RPC. Solaris 2.0 White Papers, SunSoft, 1991.

[Understand C++] Understand C++, http://www.scitools.com/ucpp.html

[VBUnit] VBUnit, http://www.vbunit.org/doc/Description.htm

[VC++6.0] Microsoft Visual C++ 6.0, http://www.microsoft.com