Usando padrões de projeto como base para reestruturação de software: um estudo de caso

Daniele Cristina Uchoa Maia Rodrigues

Trabalho Final do Mestrado Profissional

Instituto de Computação
Universidade Estadual de Campinas

Usando padrões de projeto como base para reestruturação de software: um estudo de caso

Daniele Cristina Uchoa Maia Rodrigues

Agosto de 2004

Banca Examinadora:

- Professor Dr. Rogério Drummond Burnier Pessôa de Mello Filho (orientador)
  Instituto de Computação - UNICAMP
- Professor Dr. Mario Jino
  Faculdade de Engenharia Elétrica e Computação - UNICAMP
- Professor Dr. Hans K. E. Liesenberg
  Instituto de Computação - UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Rodrigues, Daniele Cristina Uchoa Maia

R618u Usando padrões de projeto como base para reestruturação de software: um estudo de caso / Daniele Cristina Uchoa Maia Rodrigues -- Campinas, [S.P. :s.n.], 2004.

Orientador : Rogério Drummond Burnier Pessoa de Mello Filho

Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Padrões de projeto. 2. Software - Arquitetura. 3. Software – Desenvolvimento - Metodologia. 4. Java (Linguagem de programação de computador). I. Drummond, Rogério. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Usando padrões de projeto como base para reestruturação de software: um estudo de caso

Este exemplar corresponde à redação final do Trabalho final devidamente corrigida e defendida por Daniele Cristina Uchoa Maia Rodrigues e aprovada pela Banca examinadora.

Campinas, 11 de agosto de 2004.

Prof. Dr. Rogério Drummond Burnier Pessôa de Mello Filho

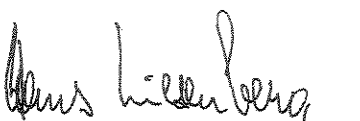
(Orientador)

Trabalho final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para obtenção do título de Mestre em Computação na área de Engenharia de Software.

TERMO DE APROVAÇÃO

Trabalho Final escrito defendido e aprovado em 11 de agosto de 2004, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Mario Jino
FEEC - UNICAMP



Prof. Dr. Hans Kurt Edmund Liesenberg
IC - UNICAMP

Prof. Dr. Rogério Drummond Burnier Pessoa de Mello Filho
IC - UNICAMP

Resumo

O crescimento da produção de software ao longo das últimas décadas tem exigido que a indústria de desenvolvimento seja capaz de produzir sistemas de forma rápida, com alta qualidade e facilidade para alteração de requisitos, fase em que se pode gastar muito tempo e esforço se o sistema não estiver preparado para estas alterações.

Neste contexto, os padrões de arquitetura e projeto surgem como soluções para problemas recorrentes, com o objetivo de diminuir o esforço gasto por desenvolvedores para solucionar problemas durante a fase de projeto. Estes padrões, quando bem utilizados, podem também facilitar a comunicação e entendimento da equipe de desenvolvimento do sistema, facilitar a alteração de requisitos projetando sistemas preparados para alterações futuras, e diminuir o tempo de desenvolvimento.

Este trabalho tem como objetivo mostrar situações práticas de utilização de padrões de arquitetura e projeto na reestruturação de um sistema real, e os benefícios desta utilização.

Abstract

The growth of software production in the last decades has demanded from development industry the capacity of producing systems in a small period, but with high quality and easy ways for changing requests, period in development that can take a long time and effort, if the system is not prepared to be changed.

In this context, architecture and design patterns come with solutions to recurrent problems, with the goal of decrease the effort spent by developers to solve project problems. These patterns, if well used, can make the communication and understanding of the development team easier, make the request changes easier by designing systems prepared for future changes, and decrease the development time.

This work has the goal to present the practical use of architecture and design patterns in the process of refactoring a real system, and the benefits of using them.

Agradecimentos

Agradeço a Deus pela oportunidade e pela força.

Ao meu marido, Clayton, por toda a paciência, companheirismo e carinho durante todo este tempo, e nos momentos mais difíceis.

Aos meus pais, Elaerto e Arlinda e ao meu irmão, Carlos, por incentivar, acreditar e me apoiar.

Ao Dr. José Antônio Martins, pelo incentivo e ajuda.

Ao Professor Dr. Rogério Drummond pela orientação, dedicação e disponibilidade.

Sumário

Índice de Figuras

Capítulo 1

Introdução

Atualmente, devido às características intrínsecas do mercado de software, faz-se necessária uma metodologia de engenharia de software que diminua o tempo necessário para desenvolvimento, produza sistemas com baixa quantidade de erros e com alta flexibilidade à alterações de requisitos. Para atender a esta demanda, as empresas de desenvolvimento de software têm enfocado a utilização de componentes, *frameworks* e padrões que atendam aos objetivos de qualidade, rapidez e flexibilidade no desenvolvimento.

Sabe-se pela experiência em desenvolvimento de sistemas de software, que apesar de as metodologias de desenvolvimento utilizarem uma grande parte do tempo total na fase de levantamento de requisitos e análise do sistema, para que os requisitos do sistema sejam bem definidos, acontece no entanto que o usuário, ao iniciar o seu contato com o sistema percebe a necessidade de alteração das funcionalidades existentes, e inserção de novos requisitos.

Estas alterações e inserções de novos requisitos no sistema, tanto durante quanto após o fechamento do ciclo análise-projeto-implementação podem se tornar bastante custosas para o desenvolvimento do projeto, caso o modelo do sistema não esteja flexível o suficiente para ser expandido. Desta forma, algumas alterações exigem que módulos inteiros do sistema tenham que ser refeitos, já que não podem ser adaptados por falta de flexibilidade e planejamento.

É neste cenário que os padrões têm se tornado cada vez mais populares, já que oferecem soluções para problemas recorrentes durante todas as fases do desenvolvimento (análise, projeto, testes e manutenção de requisitos), visando padronizar a metodologia de desenvolvimento, aplicar melhores práticas, deixar o sistema flexível às alterações de requisitos, e servindo como base para o desenvolvimento de componentes e *frameworks*.

Este trabalho tem como objetivo demonstrar a aplicação de alguns padrões bastante comuns e conhecidos na reestruturação de um projeto real, já em funcionamento, sendo testado por usuários e cuja estrutura é bastante rígida, não permitindo alterações de requisitos, mostrando

como a utilização de padrões flexibiliza a estrutura do projeto de forma rápida, simples e eficaz. Além disso, a aplicação destes padrões fornece um aumento de qualidade e facilidade de manutenção para o sistema, e o aprendizado mais aprofundado de orientação a objetos para a equipe de desenvolvimento.

Serão focados apenas padrões de arquitetura e projeto, tendo maior foco nos padrões de projeto propostos por [GAMMA - 95]. A aplicação prática realiza uma análise entre o modelo de projeto do sistema SACADA, desenvolvido pelo CPqD, sem a utilização de padrões de arquitetura e projeto, e um novo modelo proposto com a aplicação dos mesmos, destacando-se as vantagens proporcionadas pelos padrões, como o fortalecimento das melhores práticas de utilização da metodologia orientada a objetos, melhor documentação do projeto, padronização de técnicas para a equipe envolvida, e flexibilidade para alterações de requisitos.

Este trabalho está organizado de acordo com a seguinte estrutura de capítulos:

- Capítulo 2: Apresenta uma base teórica onde estão detalhados conceitos de reestruturação de software, padrões de projeto e arquitetura, que serão utilizados na aplicação prática do sistema SACADA

- Capítulo 3: Apresenta a descrição do sistema SACADA, mostrando suas características e funcionalidades em alto nível para que seja possível demonstrar em quais partes do projeto será necessária a aplicação dos padrões.

- Capítulo 4: Apresenta a aplicação prática dos padrões na fase de projeto do sistema SACADA, realizando-se uma análise comparativa entre o projeto original e o novo projeto proposto utilizando conhecimentos de padrões. Esta análise levará em consideração a qualidade do projeto e a facilidade de alteração de requisitos.

- Capítulo 5: Conclusão do trabalho.

Capítulo 2

Embasamento Teórico

Este trabalho realizou a reestruturação de um sistema utilizando os processos de engenharia reversa para recuperar a documentação e projeto original, para em seguida utilizar o processo de *forward engineering*, aplicando padrões de arquitetura e projeto, e propondo um novo modelo para o sistema, de forma a flexibilizar sua estrutura para facilitar a inserção de futuras alterações. Desta forma, será apresentada a seguir a definição de reestruturação e os padrões que serão utilizados na seção final de aplicação prática.

2.1 Definições

2.1.1 Reestruturação de Software

Em seu livro, [PRESSMAN - 01] define que existem quatro tipos diferentes de manutenção de software: corretiva, adaptativa, alteração de requisitos, e reestruturação. E do esforço total gasto na fase de manutenção, apenas 20% do trabalho é destinado à correções de erros, enquanto que 80% é destinado à adaptações do sistema para o ambiente externo, alterações de requisitos propostas pelos usuários e reestruturação do sistema para facilitar futuras alterações. Desta forma, a modelagem de sistemas com estrutura flexível diminui consideravelmente o esforço na fase de manutenção.

Ainda, segundo [PRESSMAN - 01], o processo de reestruturação de um sistema, contempla as fases de análise de inventário, reestruturação de documentação, engenharia reversa, reestruturação de dados e código e *forward engineering*.

O processo de engenharia reversa tem como objetivo recuperar a documentação de análise e projeto do sistema original, de forma que seja possível realizar um estudo do que foi projetado anteriormente, verificando os pontos com problemas e como eles podem ser melhorados.

O processo de *forward engineering* por sua vez tem como objetivo remodelar o sistema a partir de informações sobre a sua estrutura original (obtida através do processo de engenharia reversa) de forma a aumentar a qualidade e/ou performance.

O sistema SACADA, desenvolvido pelo CPqD, serviu como estudo de caso para este trabalho.

Este trabalho utilizou o sistema SACADA como estudo de caso. Este sistema está em fase de manutenção, onde foram detectados defeitos e necessidades adaptativas além de novos requisitos. Justamente neste ponto, foi encontrada uma grande dificuldade para realização da manutenção (no caso corretiva e de alteração de requisitos), devido às características internas do projeto original, que não permite a expansão do sistema.

Para tornar este sistema mais flexível à adaptações, o mesmo foi reestruturado, aplicando-se primeiramente o processo de engenharia reversa, seguido do processo de *forward engineering*.

Os conceitos de padrões de projeto e arquitetura, ilustrados no capítulo 4, foram aplicados em alguns pontos do sistema que apresentavam problemas, criando um novo modelo, que alterou e criou novas classes e interações entre objetos, com o objetivo de flexibilizar a estrutura e facilitar a integração de novos requisitos propostos pelos usuários, aumentando a qualidade do projeto e simplificando a documentação e entendimento de suas estruturas de forma a simplificar a manutenção.

2.1.2 Padrões

Padrão é uma estrutura que apresenta solução para um problema específico e recorrente em desenvolvimento de software. A solução proposta é genérica e pode ser utilizada em domínios de aplicação diferentes, onde o problema ocorre. Os padrões documentam experiências já comprovadas em projetos e têm como objetivo aplicar as soluções diretamente em problemas, sem a necessidade de que estas tenham que ser redescobertas a cada vez que o mesmo problema ocorre.

Os padrões são descritos através de um nome pelo qual são conhecidos, o problema que visam solucionar, fornecendo a solução para este problema e as conseqüências de sua aplicação. A solução para o problema consiste na descrição de um modelo de classes que se comunicam,

levando em consideração um problema genérico. Estas classes devem ser customizadas pelo desenvolvedor para solucionar problemas específicos.

Os padrões constituem também uma forma de documentar a arquitetura de um sistema que facilita a alteração e inserção de requisitos uma vez que o software já tenha sido implementado. Esta forma de documentação representa uma ferramenta bastante útil, porque uma vez que a equipe de desenvolvimento conheça e entenda as técnicas utilizadas na construção do sistema, fica mais fácil modificá-lo sem violar o padrão adotado anteriormente.

Existem vários tipos de padrões, que atuam em diferentes etapas no desenvolvimento de um sistema de software. Existem padrões de análise, padrões de arquitetura, padrões de projeto, e padrões de implementação. Neste trabalho serão focados apenas os padrões de arquitetura e projeto detalhados a seguir.

2.2 Padrões de Arquitetura

Os padrões de arquitetura têm como objetivo expressar o esquema de organização da estrutura de um sistema a partir da definição de um conjunto de subsistemas, especificando suas responsabilidades e incluindo regras e guias para organizar os relacionamentos entre estes subsistemas. Desta forma, especifica as propriedades estruturais do sistema, sendo necessário ainda à utilização dos padrões de projeto para fornecer as diretrizes de estruturação e comunicação entre elementos dos subsistemas. Os padrões de arquitetura podem ser divididos nas seguintes categorias, segundo [BUSCHMANN - 96]:

- Soluções *From Mud to Structure*[1]: Devem ser utilizadas em sistemas que possuam uma grande quantidade de objetos e componentes que necessitam de algum tipo de organização. Decompõem o sistema em subtarefas que cooperam. Fazem parte desta categoria os padrões *Layer*, *pipeFilter* e *Blackboard*.

- Soluções para sistemas distribuídos: Padrões que provêem soluções para sistemas distribuídos. Apenas o padrão *Broker* faz parte desta categoria.

[1] A tradução literal do inglês seria "da lama para a estrutura", significando da ausência de organização a uma solução estruturada.

- Soluções para sistemas interativos: Devem ser utilizadas para sistemas que possuam interação homem-computador; impedem que as classes de interface gráfica tenham alto acoplamento com as classes que realizam o processamento do sistema, permitindo desta forma que tanto a interface quanto o processamento possam ser alterados, sem impactos. Fazem parte desta categoria os padrões *Model-View-Controller* e *Presentation-Control*.

- Soluções para sistemas adaptáveis: Estas soluções se aplicam em sistemas que consistem em extensões de aplicações ou adaptações de tecnologias em evolução, ou sistemas que possuam muitas mudanças de requisitos. Fazem parte desta categoria os padrões *Reflection* e *microkernel*.

Em cada categoria, existem diferentes padrões que solucionam o problema tratado pela categoria. Desta forma, a escolha de um dos padrões dependerá dos requisito funcionais e não funcionais do sistema, como por exemplo, confiabilidade e flexibilidade para mudanças.

Será detalhado a seguir apenas o padrão de arquitetura *Model-View-Controller*, que será utilizado na remodelagem do sistema, devido às suas características e requisitos.

2.2.1 Padrão Model View Controller (MVC)

Em uma aplicação com interface gráfica é interessante que o pacote de classes da interface seja totalmente independente do pacote de classes de processamento (lógica) do sistema. Isto porque a interface gráfica é específica para um sistema, enquanto que os módulos de processamento podem ser reutilizados. Além disso, a interface gráfica por interagir com o usuário, se torna muito suscetível a alterações para se adaptar melhor às suas necessidades; e ainda, é válido ainda lembrar que um mesmo sistema poderá possuir diferentes interfaces gráficas, customizadas para cada cliente.

Sistemas que possuem estas características deverão permitir mudanças que as alterações na interface gráfica possam ser incorporadas de forma rápida e simples, sem a necessidade de alterações na camada de processamento do sistema.

Para que isto seja possível, é interessante manter todas as classes de interface gráfica em um pacote distinto, que contenha apenas objetos que interagem com o usuário, enquanto que os

pacotes de processamento não deverão conter nenhum objeto de interface gráfica, apenas objetos que realizam as funcionalidades lógicas do sistema; o último não deverá ter nenhum conhecimento do primeiro e a interação entre estes pacotes deve ser feita por meio de uma interface bem definida.

Este processo torna o sistema mais coeso, já que o pacote de interface gráfica contém apenas definições sobre a apresentação do sistema para o usuário, permitindo que o pacote de processamento não precise realizar acessos ao pacote de interface gráfica. Como o pacote de processamento se torna independente, pode ser facilmente reutilizado por outros sistemas e, além disso, alterações de requisitos na interface gráfica não causarão impactos nas classes de processamento do sistema.

Neste padrão, *Model* representa a camada lógica (processamento), enquanto que *View* é a camada responsável por toda apresentação de dados para o usuário, e *Controller* é responsável por realizar a ligação entre o *Model* (que contém as informações a serem apresentadas ao usuário) e *View*.

Originalmente, este padrão contém os três elementos descritos anteriormente: *Model*, *View* e *Controller* (MVC), segundo descrito por [BUSCHMANN - 96]. Mas, segundo [LARMAN - 98], as responsabilidades do *Controller* sempre acabam sendo implementadas junto com o elemento *View* (desenvolvedores Java costumam implementá-los em uma mesma classe), não existindo a necessidade de serem considerados separadamente. Portanto, neste trabalho, *View* e *Controller* serão sempre mencionados juntos.

A regra é que classes do pacote *Model* não devem ter conhecimento ou ser diretamente acopladas a classes do pacote *View-Controller*, e vice versa.

Como a interface gráfica precisa mostrar informações para o usuário, é necessário que esta consiga obter estas informações contidas em *Model*. Para que esta interação seja possível, e ainda, mantendo o baixo acoplamento entre as camadas, as classes de *Model* devem notificar as classes de *View* sempre que uma alteração em seu conteúdo ocorrer, para que a última possa apresentar estas alterações ao usuário do sistema. Esta notificação pode ser efetuada através do padrão de projeto *Observer* que será descrito posteriormente.

2.3 Padrões de Projeto

Os padrões de projeto têm como objetivo fornecer uma solução aplicável a projetos orientados a objetos (especificando estruturas e relacionamentos de classes e interações entre objetos) para problemas que são freqüentemente encontrados durante a fase de projeto de um sistema de software. Estes padrões foram elaborados com o objetivo de padronizar a solução de problemas, visando: compartilhamento de arquiteturas de classes entre projetos; facilitar o entendimento do projeto pelos desenvolvedores, já que a documentação se torna padronizada; facilitar a manutenção do sistema, já que ficam com estruturas mais flexíveis; diminuir a quantidade de erros, já que especificam a melhor interação entre os objetos participantes; favorecer a reutilização de código; e por fim, impor as melhores práticas de projetos orientados a objetos.

Para alcançar estes objetivos, os padrões de projeto devem seguir algumas recomendações gerais:

- A programação deve ser feita com base em interfaces e não em classes concretas;

- Favorecimento de composição funcional de objetos e não de herança;

- Encapsular variações;

Aplicando-se estes três conceitos básicos em projetos, consegue-se um resultado com alta coesão, ou seja, cada classe desempenha apenas um papel bem definido no sistema, e um baixo acoplamento entre as classes (a dependência entre elas é menor).

E ainda, a obtenção de alta coesão e baixo acoplamento permite que o sistema tenha sua manutenção facilitada, já que as classes possuem funções específicas e menores, tornando o entendimento mais simples e as alterações localizadas. Como o acoplamento é baixo, a dependência entre as classes é menor, portanto, alterações localizadas em algumas classes não gerarão alterações em cascata em todas as classes que interagem com estas. E ainda, estes dois conceitos permitem uma maior reutilização das estruturas de classes, já que suas funcionalidades são bem específicas e independentes.

É importante ressaltar que ao se utilizar padrões, os desenvolvedores implementam no sistema soluções que já foram amplamente testadas, já que são provenientes do acúmulo de experiência de vários profissionais de desenvolvimento.

A seguir serão detalhados os conceitos básicos utilizados pelos padrões de projeto, e posteriormente, os padrões de projeto utilizados para o desenvolvimento deste trabalho.

2.3.1 Conceitos utilizados em soluções de padrões de projetos

- Interfaces e Polimorfismo

Uma das bases dos padrões de objetos descritos por [GAMMA - 95] é o favorecimento de utilização de interfaces ao invés de classes concretas durante a elaboração do projeto. Isto porque quando se define um relacionamento de herança de classes concretas, está sendo definido que a subclasse herda não só a interface que define o comportamento, mas também o código de implementação dos métodos da superclasse concreta, havendo então um compartilhamento de código. Desta forma, alterações no código da superclasse poderão afetar o comportamento de toda a hierarquia de classes, gerando portanto as modificações em cascata que devem ser evitadas para que o sistema possa incorporar novos requisitos de forma rápida e simples.

Já a utilização de interfaces, define apenas que objetos que implementam a mesma interface possuem o mesmo comportamento, mas não existe compartilhamento de código. Define-se, portanto, o comportamento esperado para um determinado objeto, deixando que ele determine como este comportamento será implementado. Como não existe compartilhamento de código, não existem também alterações em cascata.

Tanto a herança quanto a implementação de interfaces permitem que objetos na mesma hierarquia possam ser utilizados sem distinção já que possuem o mesmo comportamento. Esta técnica é chamada de polimorfismo e é muito utilizada na orientação a objetos. É um conceito muito utilizado na construção de *frameworks*, já que permite a definição de interfaces de comportamento, mas deixa a implementação deste comportamento ser customizada de acordo com o projeto que o utiliza.

- Herança x Composição

A herança e a composição de objetos são dois mecanismos para reutilizar funcionalidades de classes.

A herança realiza a reutilização de código, já que permite que classes derivadas estendam atributos e métodos de uma superclasse. Mas esta técnica torna o acoplamento entre a superclasse e as subclasses muito forte, já que há uma dependência grande da implementação de métodos e declaração de atributos da subclasse em relação à superclasse. E com forte acoplamento, qualquer alteração na implementação da superclasse irá gerar também alterações em cascata nas implementações das subclasses.

Já a composição funcional de objetos estende as responsabilidades através da delegação de tarefas. Cada objeto que participa da composição possui uma responsabilidade bem definida, e a utilização destes objetos em conjunto permite a realização de funcionalidades mais complexas. Desta forma, garante-se que as classes sejam bastante coesas, porque se cada uma delas tem sua responsabilidade bem definida, nenhuma classe estará realizando funções de outras. Além disso, garante-se também um baixo acoplamento, pois se um objeto executa suas funcionalidades corretamente e é independente de outros objetos, ainda que sua implementação seja alterada, os objetos que o utilizam não deverão ser alterados, desde que sua interface seja mantida.

- Encapsulando variações

Em seu livro, [GAMMA - 95] propõe que os conceitos tradicionais de orientação a objetos utilizados pela maior parte dos desenvolvedores de sistemas são bastante limitados. Encapsular não é apenas esconder dados dentro de um objeto de forma que este objeto seja responsável por gerenciar estas informações sem que outros possam acessá-las diretamente. Deve-se encapsular também as variações entre objetos, criando níveis e impedindo que alterações ou criação de novos tipos em um nível afetem os níveis seguintes.

Para isso, as informações comuns entre objetos devem ser dispostas em uma classe, e as informações que variam, e são passíveis de alterações futuras devem ser colocadas em uma outra classe, de forma que a criação de novas variações não alterará o funcionamento da primeira.

Desta forma, o encapsulamento de informações pode ser muito útil na construção de projetos preparados para eventuais expansões de funcionalidades, tornando o sistema mais flexível.

Os padrões de projetos definidos por [GAMMA - 95], que utilizam os três conceitos descritos anteriormente, são divididos em três conjuntos:

- Padrões de criação
- Padrões estruturais
- Padrões de comportamento

Estes três conjuntos serão explicados a seguir, em [GAMMA - 95] de uma forma mais completa, em [SHALLOWAY - 02] e [ECKEL - 03] com diferentes exemplos e de uma forma mais simplificada, e em [COOPER - 98], onde são dados exemplos que mostram que a linguagem Java foi implementada utilizando padrões, e com exemplos práticos de utilização de padrões em projetos de interface gráfica.

A descrição a seguir utilizará a notação UML (*Unified Modeling Language*) para demonstrar a interação entre objetos (mais informações sobre a notação, vide [FOWLER - 00]) e conceitos da linguagem de programação Java (mais informações sobre Java, vide [ECKEL - 00], [BARKER - 02] e [DEITEL - 02]).

2.3.2 Soluções de Criação de Objetos

Os padrões de criação têm como objetivo tornar o sistema independente do processo de criação de objetos. Eles se preocupam em desvincular a instanciação de objetos produtos em classes clientes que os utilizam, para que os objetos criados, possam ser de tipos diferentes, dependendo do tipo de execução desejada para o sistema.

As classes clientes não precisam conhecer quais são as classes produtos concretas que realizam as funções requisitadas. Basta à classe cliente conhecer interfaces que especifiquem os métodos implementados, indicando seus parâmetros de entrada, processamento e resultados.

Os padrões de criação de objetos podem ser divididos em dois tipos, de acordo com o mecanismo adotado para variação da classe produto sendo instanciada: padrões de classes, que

utilizam a herança para variar a classe que é instanciada, e padrões de objetos, que utilizam a composição de objetos para variar a classe instanciada (delegando a instanciação para outros objetos).

A seguir, são descritos os padrões de criação que foram utilizados na reestruturação do estudo de caso.

2.3.2.1 *Abstract factory*

O padrão *abstract factory* é baseado no conceito de famílias de classes produtos. Ou seja, em um determinado sistema, cada produto pode ser implementado de formas diferentes, desta forma, cada tipo de implementação dá origem a uma hierarquia de classes.

Cada hierarquia é formada por uma interface (ou superclasse) que especifica o comportamento que o produto deve ter, e várias classes concretas que implementam (chamadas implementações) esta interface de formas diferentes, implementando também o seu comportamento.

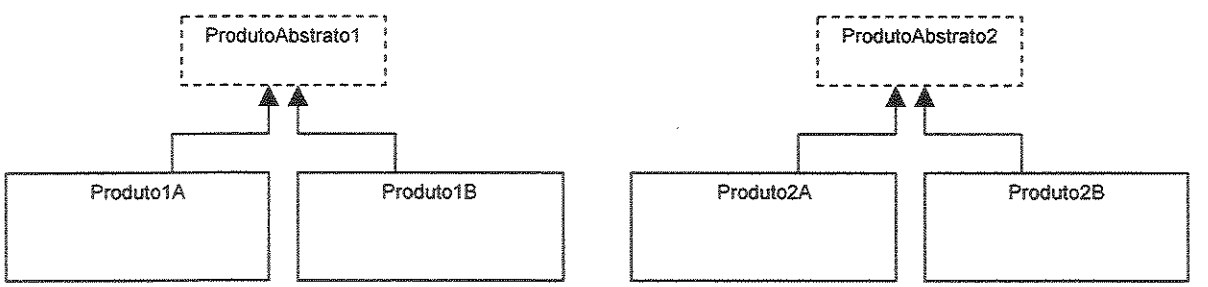


Figura 1. Modelo de duas hierarquias de classes

A Figura 1 apresenta duas hierarquias de classes, onde cada produto desta hierarquia possui dois tipos de implementações diferentes: A e B. Observando este modelo, pode-se notar que existem também duas famílias de produtos, de acordo com as implementações: uma família é formada pelos produtos com implementação A e outra família é formada pelos produtos com implementação B. Cada família possui uma implementação própria para cada produto abstrato, respeitando sempre a interface definida pelo produto abstrato, garantindo que o cliente possa acessar os produtos de uma forma padrão, independente de sua implementação.

Haverá também classes clientes que utilizarão esses produtos, de acordo com suas interfaces. Para o cliente, é indiferente qual classe concreta implementa a interface, sendo importante apenas a garantia de que o serviço requisitado será realizado.

Em sistemas deste tipo, caso este padrão não seja utilizado, cada classe cliente ao instanciar um produto, deveria primeiramente escolher a família de produtos (implementação) a ser utilizada, para depois instanciar a classe concreta que implementa a interface do produto. Desta forma, as instanciações de objetos e verificações sobre a família a ser utilizada estariam espalhadas pelas classes clientes. Caso uma nova família de objetos seja criada, a alteração no código cliente seria muito grande, já que seria necessário alterar cada ponto do código onde uma instanciação de produto é realizada, inserindo mais um tipo de escolha.

É possível perceber claramente que o problema se concentra no fato de que classes clientes necessitam instanciar explicitamente objetos produto, que podem variar. Para evitar este problema de instanciação, o padrão propões que seja criado um objeto intermediário que é responsável por instanciar a classe produto correta segundo a família de implementação a ser utilizada pelo sistema. Estas classes intermediárias são chamadas de *factories*. É definido para o sistema uma *factory* abstrata, que possui a interface para criação de produtos, e várias *factories* concretas (uma por família de implementação), que implementam métodos de criação de produtos definidos pela interface, e instanciam as classes produtos corretamente.

Para alcançar a independência de instanciação, os clientes utilizam as *factories* para instanciar novos produtos. Assim, na inicialização do sistema, deve-se definir qual a família de implementação a ser utilizada pelo sistema, e esta informação poderá ficar armazenada em um arquivo de configuração, por exemplo; Esta informação será única para todo o sistema, impedindo que se mude a família a ser instanciada durante a implementação. Com esta informação é possível definir qual *factory* concreta que deverá ser instanciada. A factory concreta é responsável por saber quais classes produtos devem ser instanciadas. Esta situação é apresentada na Figura 2.

Como todas as *factories* concretas obedecem à interface padrão, pode-se trocar a família de objetos apenas trocando informação do arquivo de configuração (antes da inicialização do sistema) que indica qual família de implementação será utilizada, e isso fará com que a *factory* concreta a ser utilizada seja alterada. Desta forma, os objetos clientes permanecem inalterados.

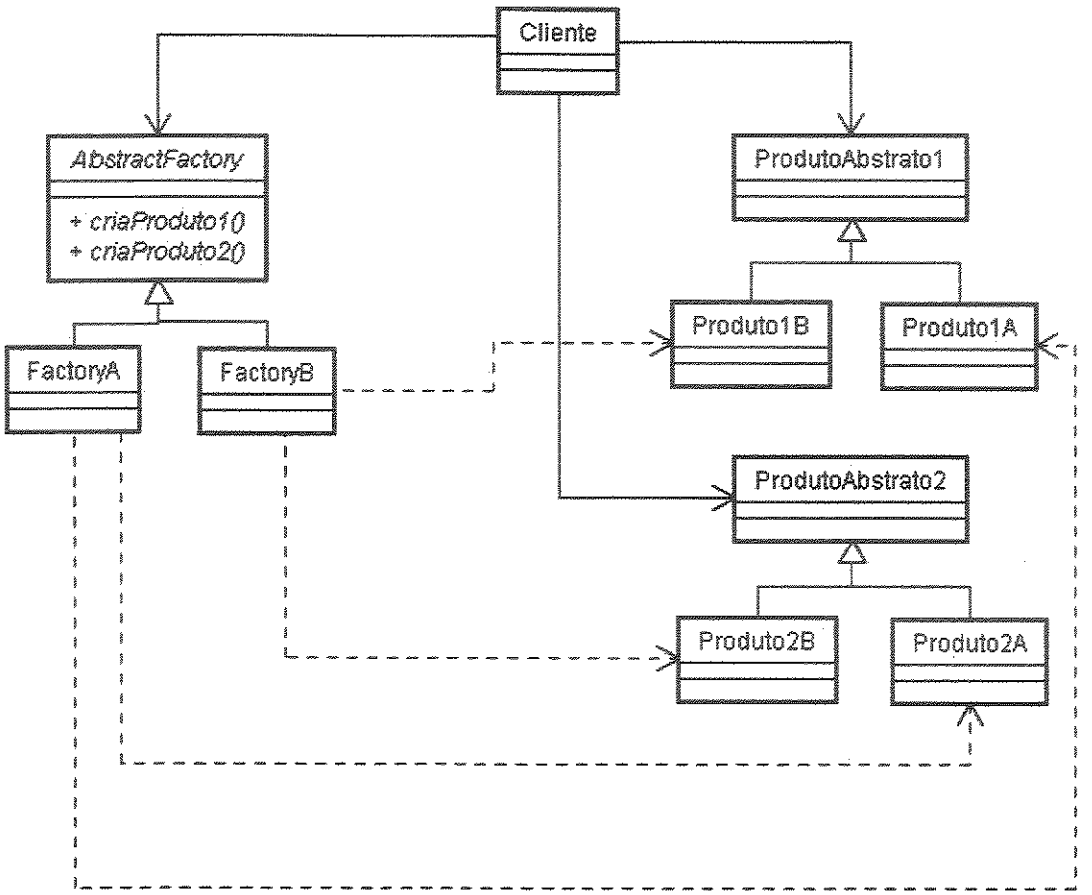


Figura 2. Modelo de *Abstract Factory*

Desta forma, caso seja necessário variar a implementação de produtos, ou seja, inserir uma nova família de implementação, por exemplo a implementação C, bastará criar os produtos concretos 1 e 2 para esta família, criar uma nova *factory* concreta (FactoryC) que conheça as novas classes concretas de produtos que poderão ser instanciadas, e inicializar o sistema com esta nova família. Observa-se que a alteração no código é localizada, e não ocasiona alterações nos clientes que utilizam os produtos.

2.3.2.2 Factory Method

Este padrão tem como objetivo designar a responsabilidade de criação de objetos produto para subclasses. Para isso, define uma interface para criar um objeto, mas deixa que as subclasses decidam qual a classe produto que deverá ser instanciada. Permite, portanto, que a responsabilidade da instanciação seja das subclasses.

Ou seja, uma classe define métodos abstratos de criação de objetos, mas não realiza a instanciação. As subclasses deverão implementar estes métodos, inserindo código de instanciação de objetos reais, possibilitando que se criem ganchos para que as subclasses realizem a instanciação.

Observe-se que o padrão *abstract factory* descrito anteriormente em 2.3.2.1 utiliza os *factory methods*, já que a *abstract factory* define vários métodos de criação de objetos (um para cada produto) e as subclasses concretas (*factories* concretas) que implementam os métodos inserem código para a instanciação real, como é apresentado na Figura 3.

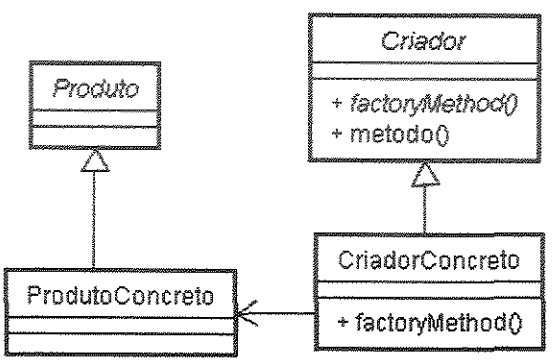


Figura 3. Modelo de *Factory Method*

2.3.2.3 *Singleton*

Este padrão tem como objetivo assegurar que uma classe possuirá apenas uma instância no sistema e que os outros objetos poderão acessar esta instância de forma simples.

Esta solução deverá ser utilizada quando, por questões de desempenho ou restrições do sistema, não possa existir mais de uma instância de uma mesma classe.

Para isso, ao invés de se criar um objeto que seja responsável por manter o controle da instanciação de outras classes, coloca-se a responsabilidade do controle de instanciação na própria classe, que se torna responsável por gerenciar a sua criação.

A classe deverá possuir um método público e estático responsável por realizar a instanciação do objeto único que qualquer outra classe consiga acessar. Este método verifica se já

existe uma instância da classe; se não existir, cria a instância. Se já existir, devolve a instância existente. O modelo desta classe é apresentado na Figura 4.

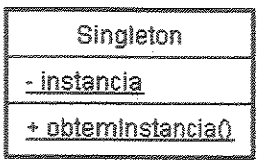


Figura 4. Classe *Singleton*

Para atingir este objetivo, o construtor da classe deverá ser privado ou protegido, garantindo que apenas a própria classe ou suas subclasses tenham acesso ao mesmo. Além disso, deverá ter um atributo privado e estático que representa a instância da classe. E por último, um método de obtenção da instância estático, que poderá ser acessado por qualquer outro objeto do sistema. É interessante observar que em sistemas *multithread* todos os métodos da classe *singleton* deverão ser sincronizados, para evitar que a concorrência de acesso de várias threads aos métodos do objeto, deixe seus atributos privados em um estado inconsistente [2].

2.3.2.4 Object Pool [3]

Este padrão tem como objetivo resolver o problema de gerenciamento de recursos do sistema que devem ser compartilhados por muitos objetos clientes.

Alguns recursos do sistema têm alto custo para a manutenção quando utilizados em grandes quantidades. Outros possuem limitação na quantidade que pode ser instanciada. Devido a estas restrições, estes recursos deverão ser compartilhados pelos objetos clientes que os requisitam no sistema, e este compartilhamento deve ser feito com algum gerenciamento para evitar inconsistências ou falhas nos recursos.

Com o objetivo de resolver este problema, utiliza-se o padrão *object pool*, que define um objeto *pool* responsável por manter e gerenciar uma quantidade de recursos compartilhados e alocá-los conforme as requisições dos objetos clientes. Este pool é um *singleton*, pois deverá ter

---

[2] Para saber mais sobre concorrência de acesso em objetos *singleton*, vide [HAGGAR - 2004].

[3] O padrão *Object Pool* foi classificado no grupo de soluções de criação porque o seu propósito é criar e gerenciar objetos compartilhados por clientes do sistema. Em [ECKEL-B - 2003] este padrão é classificado como *Object Quantity*.

uma instância única no sistema e deverá ser instanciado apenas na sua inicialização. Todas as utilizações seguintes do mesmo deverão ocorrer por meio da obtenção de sua instância única pelos objetos clientes. Além disso, possuirá métodos para obtenção e liberação de recursos após a sua utilização. Ambos os métodos são sincronizados pois acessam um recurso compartilhado do sistema, e deve-se evitar inconsistência do recurso devido a acessos simultâneos e não sincronizados.

O padrão define também um objeto recurso que conterá todas as informações referentes ao recurso compartilhado. Este objeto será utilizado para comunicação entre as classes clientes e o *pool* para que a requisição, utilização e liberação dos recursos possam ser realizadas. A Figura 5 representa o modelo de classes para este padrão

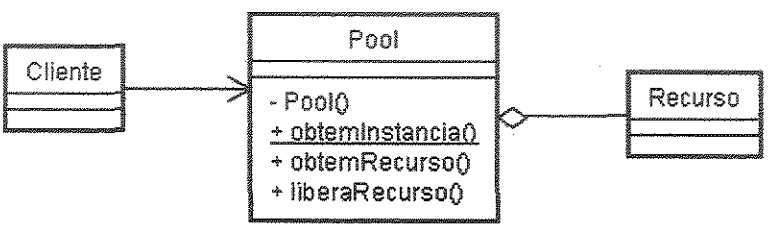


Figura 5. Modelo de classes de *Object Pool*

2.3.3 Soluções Estruturais

As soluções estruturais se preocupam em como classes herdam comportamento de outras classes, e como objetos com funções bem definidas são compostos para formar estruturas maiores. As soluções estruturais, muito mais do que as soluções de criação procuram sempre favorecer a composição de objetos sobre a herança, para fornecer maior flexibilidade para alterações de requisitos e reutilização de objetos.

Os padrões estruturais podem ser divididos em dois grupos, de acordo com o tipo de composição resultante da aplicação do padrão: padrões de classes são aqueles que utilizam a herança para compor interfaces ou implementações, enquanto que os padrões estruturais de objetos descrevem formas de compor objetos para realizar novas funcionalidades.

A seguir serão mostrados os padrões que representam as soluções estruturais que foram utilizados no processo de reestruturação do sistema SACADA.

2.3.3.1 *Façade*

O objetivo desta solução é mascarar em uma interface única várias interfaces de um subsistema, com o objetivo de facilitar a sua utilização pelos clientes.

A divisão de um sistema em subsistemas faz com que o acoplamento entre os objetos diminua, já que os clientes possuem apenas um ponto de acesso ao subsistema, e alterações neste subsistema não afetam o cliente desde que a interface de acesso seja mantida.

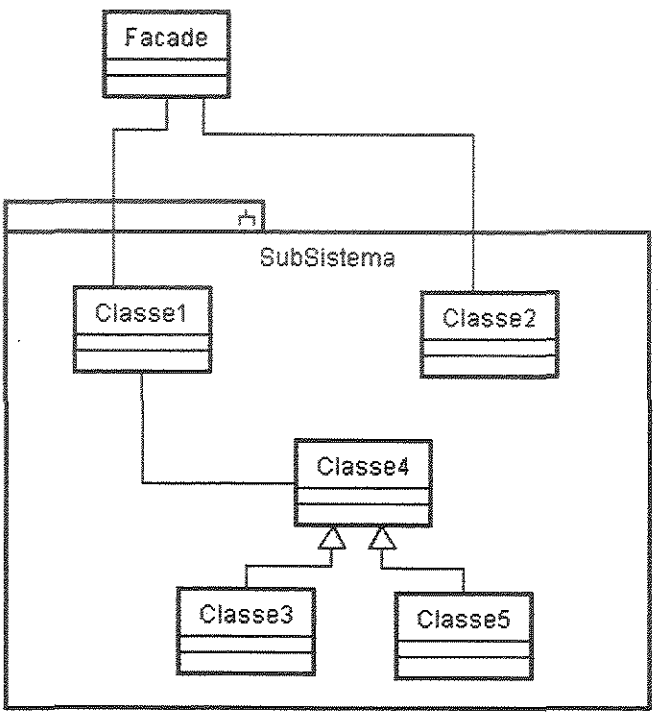


Figura 6. Modelo de *Façade*

A Figura 6 mostra o modelo de interação entre os objetos clientes, o objeto *Façade* e um subsistema.

A classe *Façade* opera fornecendo aos clientes uma interface simplificada para realização das tarefas necessárias, e delegando tarefas que deverão ser executadas pelas classes do subsistema. Estas últimas não têm conhecimento sobre a interface que delega as tarefas.

Além de delegar funções, a interface pode também necessitar criar novas funcionalidades, transformando as funcionalidades já existentes no subsistema, para que possam ser utilizadas pelas classes clientes.

As classes clientes utilizam o objeto *Façade* para acessar funcionalidades oferecidas pelo subsistema de uma forma bastante simplificada.

2.3.3.2 Bridge

Esta solução busca desacoplar as abstrações das implementações de forma que ambas possam ser alteradas independentemente.

Quando se tem uma estrutura em que abstrações podem ser implementadas de várias formas diferentes, a primeira idéia de implementação é criar uma herança de classes, onde em um primeiro nível estão as abstrações, e em um segundo nível estão as diferentes classes que implementam estas abstrações.

Esta estrutura é ruim já que torna complexa a implementação de qualquer variação. Isto porque a criação de novas abstrações ou de novos tipos de implementação começam a gerar um número muito grande de novas classes. A geração de uma nova abstração requer a geração de classes para cada implementação, e a geração de uma nova implementação requer a geração de uma classe para cada tipo de abstração.

A solução para este problema está em separar as abstrações e as implementações em hierarquias de classes diferentes, permitindo que ambas possam variar sem interferência de uma na outra.

Para que isso seja possível, é importante primeiramente conseguir definir claramente a diferença entre abstrações e implementações. Em um projeto orientado a objetos é importante analisar quais são as variações existentes entre os objetos e encapsular estas variações. Além disso, deve-se sempre favorecer a composição de objetos sobre a herança, e estas composições de objetos contêm as variações, e não mais as heranças. Para definir o que é comum e o que é variável entre diferentes objetos, o ideal é definir objetos de acordo com suas responsabilidades. Os conceitos comuns entre os objetos devem ser retratados em classes abstratas, e no caso desta solução, estes conceitos comuns serão as classes abstratas da abstração e as classes abstratas da

implementação. As classes derivadas das abstrações e das implementações conterão os conceitos variáveis de ambas, como está sendo mostrado na Figura 7.

Ao possuir hierarquias separadas para abstrações e implementações, os clientes conhecerão apenas a interface das abstrações, e as classes concretas das abstrações utilizarão as classes concretas das implementações para que as abstrações sejam realizadas.

Desta forma, a criação de uma nova implementação ou de uma nova abstração não provoca nenhum impacto em nenhum cliente e em nenhuma classe da hierarquia, facilitando a alteração de requisitos do sistema.

Pode-se associar a utilização desta solução à solução de criação *Abstract factory*. Desta forma, a classe concreta da Abstração (AbstraçãoRefinada) requisita um objeto derivado de implementador à *abstract factory*, e esta retorna a instância correta de acordo com parâmetros do sistema a serem analisados. Novamente, garante-se maior desacoplamento, já que a instanciação dos objetos fica limitada à *abstract factory*, e os clientes utilizam apenas as interfaces.

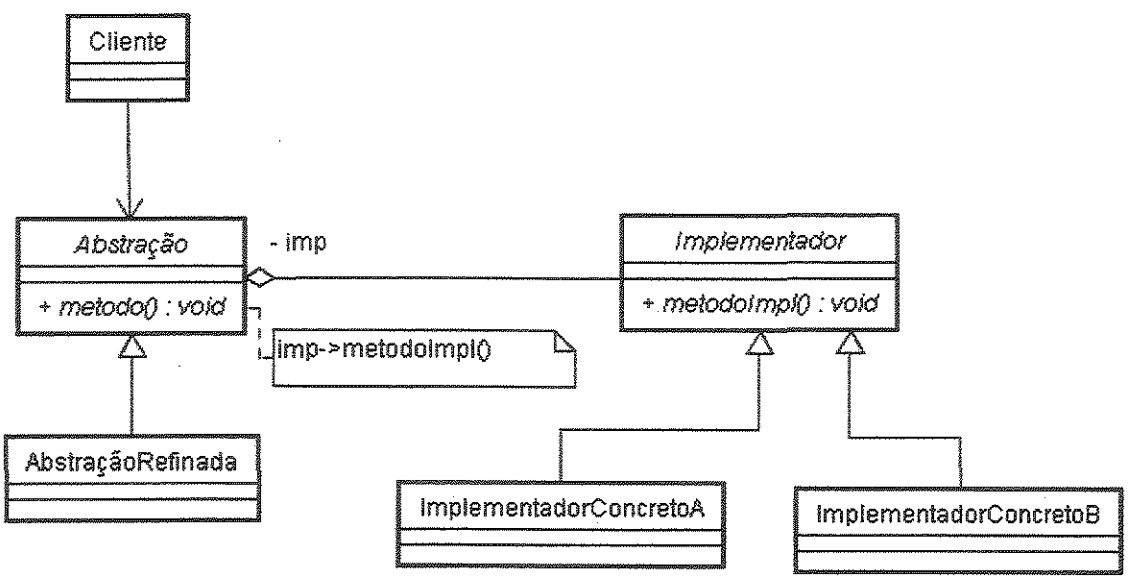


Figura 7. Modelo de *Bridge*

2.3.4 Soluções de Comportamento

As soluções de comportamento tratam objetos que manipulam ações particulares no sistema. Os objetos que compõem os padrões encapsulam principalmente procedimentos e

algoritmos, visando sempre à facilidade de alteração e extensão dos mesmos sem impactos no código cliente.

Além disso, auxilia na definição de como é feita a comunicação entre objetos em um sistema e como o fluxo da comunicação é controlado em um sistema complexo.

2.3.4.1 Commands

Este padrão tem como objetivo criar comandos que sejam independentes dos clientes que os utilizam e, portanto, podem ser utilizados em várias situações dentro do sistema.

Para atingir este objetivo, este padrão declara uma classe abstrata *Command*, que é responsável por definir a interface comum de operações que o objeto *Command* poderá executar. Em sua forma mais simples, esta interface possuirá apenas um método, que dispara a execução do comando. As subclasses serão responsáveis por especificar um par receptor-ação armazenando o receptor em uma variável e implementando o método de execução para realizar a requisição. Este modelo está sendo apresentado na Figura 8.

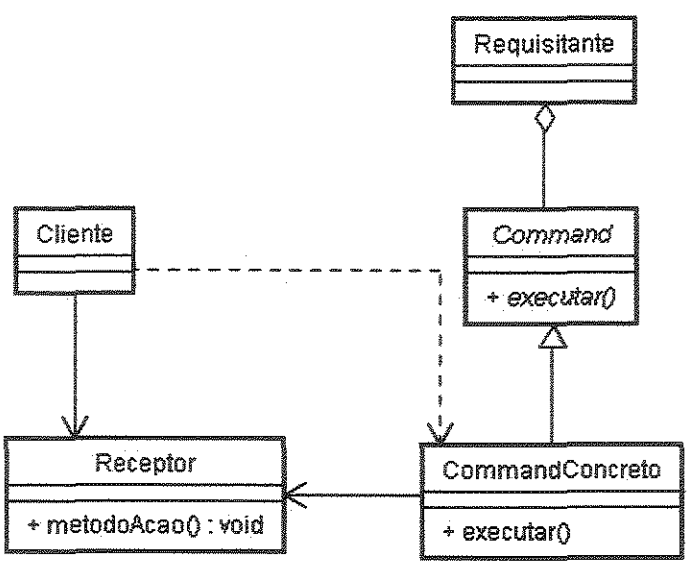


Figura 8. Modelo de *Command*

Com estas medidas, consegue-se desacoplar o objeto que invoca uma operação dos objetos que possuem conhecimento necessário para realizá-la. Desta forma, um mesmo comando pode ser executado partindo de diferentes cenários de execução do sistema.

Além disso, a adição de novos comandos, ou alterações na execução de comandos já existentes, se torna simples, ao passo que a modificação fica bem localizada, não precisando alterar as classes clientes que utilizam os objetos de classes derivadas de *Command*.

É importante ressaltar que o objeto concreto *Command* pode ter níveis diferentes de conhecimento a respeito do como executar uma requisição. Ele pode apenas invocar um objeto receptor que será responsável por realizar toda a execução do comando, ou pode conhecer todo o procedimento, e não chamar o objeto receptor em nenhum momento. Além disso, no caso de objetos concretos *Command* utilizarem um receptor para execução de requisições, é necessário levar em consideração quem será o responsável por instanciar os objetos receptores, o cliente que instancia o comando, ou o próprio objeto *Command* conhecê-lo implicitamente.

2.3.4.2 Observer

Este padrão tem como objetivo permitir que um determinado evento em um objeto seja observado por outros objetos do sistema. Para isso, vários objetos são notificados quando ocorrem eventos em um determinado objeto de interesse. Isto define uma dependência de um para muitos entre objetos.

Esta dependência deve ser mantida garantindo-se ainda um acoplamento baixo entre os objetos que observam e que são observados, de forma que a inclusão de novos objetos que observam não cause alterações no objeto observado.

Segundo [GAMMA - 95] o objeto sendo observado é chamado de *subject* ou *Publisher*, já que emite notificações de eventos, e os objetos que observam são chamados de *observers* ou *subscribers*, já que observam e recebem notificações de eventos (este padrão é também conhecido pelo nome *Publisher-Subscriber*).

Desta forma, [GAMMA - 95] sugere que se defina uma interface *Publisher*, e todas as classes concretas que devem ser observadas deverão implementá-la. Esta interface possuirá métodos para adicionar, remover e notificar seus observadores. Deve-se definir também uma interface *observer*, que possuirá um método de notificação, e todas as classes concretas que quiserem observar e ser notificadas quando da ocorrência de um evento, deverão implementá-la, como é apresentado na Figura 9.

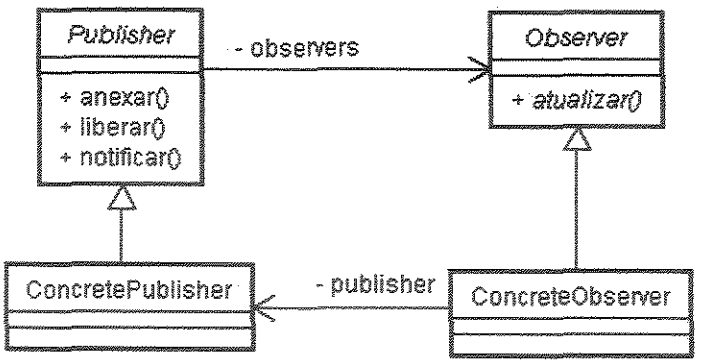


Figura 9. Modelo de *Observer*, segundo [GAMMA - 95]

Em Java existe a classe abstrata *Observable* e a interface *Observer*, que implementam o padrão descrito por [GAMMA - 95]. A utilização destas interfaces tem os seguintes problemas: como o *Observable* é uma classe concreta, esta deve ser estendida pelas classes que devem ser observadas por meio do mecanismo de herança, e como em Java não existe o conceito de herança múltipla, as classes a serem observadas não poderiam herdar comportamentos de nenhuma outra classe. Além disso, a interface *Observer* possui apenas um método de notificação, tornando difícil a observação de diferentes eventos.

A linguagem Java oferece ainda um outro mecanismo, os eventos (*events*). As classes concretas que desejam observar devem implementar a interface *listener (Observer)*. O objeto a ser observado permite que *listeners* se registrem e, quando ocorre algum evento, o objeto instancia um objeto *event* e notifica todos os *listeners* registrados, como é mostrado na Figura 10.

Este mecanismo garante baixo acoplamento entre os objetos, já que se comunicam apenas através de interfaces, e ainda, sem as desvantagens descritas anteriormente para a classe *Observer,* pois as classes que desejarem observar implementam uma interface, e não estendem uma classe abstrata, permitindo que observem diferentes eventos, e ainda herdem características de outras classes; e além disso, a interface *listener* pode possuir diferentes métodos, permitindo que eventos diferentes possam ser observados.

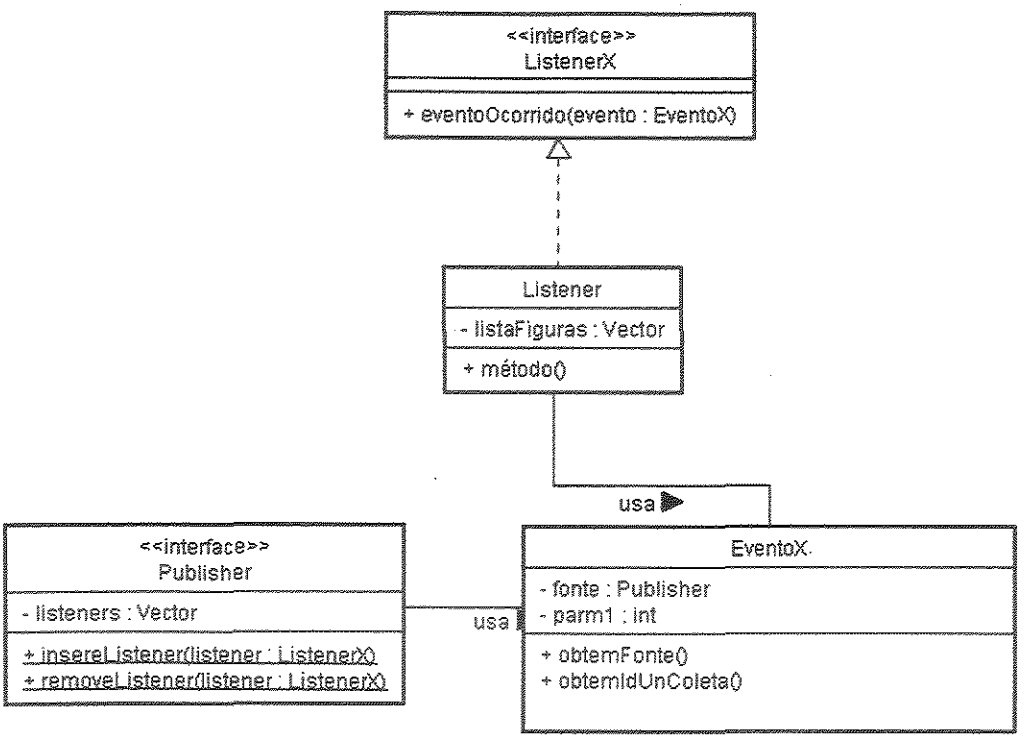


Figura 10. Modelo *Observer* utilizando eventos em Java

2.3.4.3 *Template Method*

Este padrão tem como objetivo permitir que uma classe faça a definição geral de um algoritmo, deixando que algumas partes do mesmo variem de acordo com a implementação específica feita por subclasses. Ou seja, o algoritmo é definido em um método concreto em uma classe pai na hierarquia de classes, e este utiliza durante a sua execução métodos abstratos; desta forma, quando uma classe deriva desta classe pai, deverá implementar estes métodos de acordo com suas necessidades. A Figura 11 apresenta o diagrama de classes para este padrão, com uma classe abstrata que possui um *template method*, que define um algoritmo utilizando os métodos primitivos abstratos definidos, e classes concretas derivadas que deverão implementá-los.

É interessante ressaltar que neste padrão, utiliza-se uma estrutura de controle invertida, na qual a classe pai utiliza métodos das classes filhas, e não o contrário.

Como este padrão utiliza um modelo onde se define um algoritmo básico em uma classe abstrata, e as classes derivadas devem implementar os métodos utilizados por este algoritmo da forma mais conveniente, este padrão caracteriza as bases para a construção de *frameworks*, onde

as classes do *framework* são abstratas e definem algoritmos básicos que não podem ser alterados, deixando a cargo do desenvolvedor criar classes derivadas que por meio da implementação dos métodos primitivos abstratos permitam que os algoritmos sejam customizados.

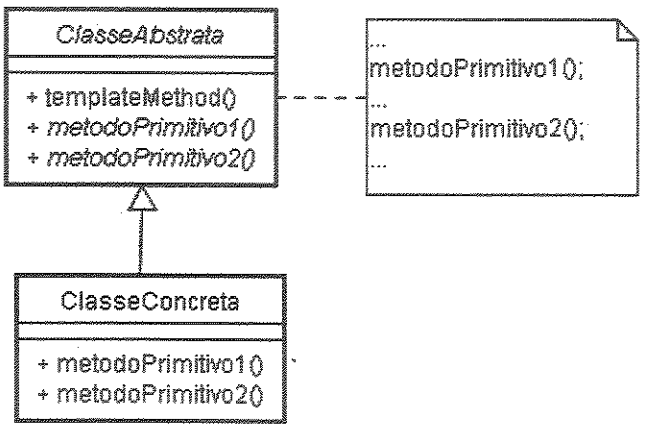


Figura 11. Modelo de *Template Method*

2.3.4.4 DAO (Data Acesses Objects)[4]

O problema retratado por este padrão consiste nas diferenças de acesso a sistemas gerenciadores de base de dados, dependendo da tecnologia utilizada para armazenamento (relacional, orientado a objetos, etc.) e do fabricante destes sistemas.

Este padrão tem como objetivo separar a camada de acesso a dados persistentes da camada do sistema que manipula estes dados. Desta forma, o tipo de base de dados utilizado para armazenamento de dados (pode-se utilizar bases de dados relacionais, bases de dados orientada a objetos, etc.) ou o fabricante do sistema gerenciador de base de dados podem ser alterados sem que as classes clientes que manipulam os dados precisem ser modificadas, e nem mesmo conhecer a alteração.

Para atingir este objetivo, este padrão encapsula o acesso à base de dados em uma camada separada do sistema, escondendo todos os detalhes de implementação que variam de acordo com

[4] O padrão DAO é descrito em [BLUEPRINTS - 04] e foi classificado no grupo de soluções de comportamento porque tem como objetivo criar classes de acesso a base de dados de forma que os objetos clientes não conheçam os algoritmos proprietários de cada tipo de base de dados, e permitir que independentemente da base utilizada, o sistema manipule apenas os objetos que deverão ser persistidos.

o tipo do sistema gerenciador de base de dados e fabricante, e disponibilizando aos objetos cliente uma interface para manipulação da base de dados. Essencialmente, a camada DAO age como um adaptador entre a camada de base de dados (o funcionamento da base de dados é encapsulado pela camada DAO), e os outros objetos do sistema. Para encapsular este funcionamento, o DAO irá disponibilizar aos clientes todos os métodos para acesso à base de dados (consultas, inserções, alterações e remoções).

Além disso, serão definidos também objetos *Data Transfer Object* (DTO), que encapsulam os dados que são armazenados na base de dados. Desta forma, os objetos DAO irão receber como parâmetro e devolver como retorno de seus métodos apenas objetos DTO.

Independentemente do tipo de base de dados utilizada, o cliente manipula apenas objetos DTO. Ou seja, do lado da base de dados, os dados podem ser tratados de qualquer forma, mas quando estes dados são passados ao cliente, sempre são mapeados em objetos. Da mesma forma, quando uma classe cliente desejar manipular dados na base, envia objetos DTO, e o DAO é responsável por transformar estes objetos em dados que podem ser manipulados na base.

O modelo geral deste padrão é apresentado na Figura 12:

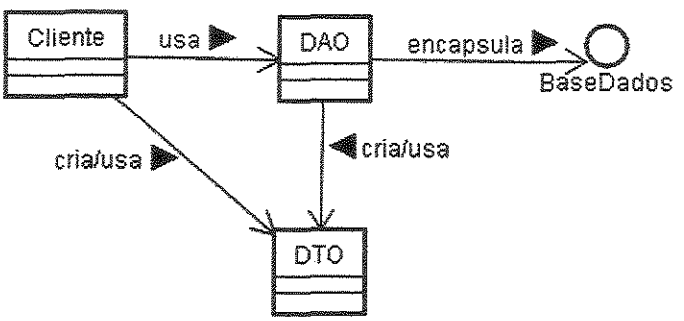


Figura 12. Modelo de DAO

É importante ressaltar que este padrão além de centralizar o acesso à base de dados do sistema, reduz a complexidade das classes clientes, que têm suas funcionalidades preservadas, utilizando funcionalidades de persistência dos objetos DTO, sem conhecer as especificações de cada fornecedor de gerenciadores de bases de dados.

Capítulo 3

Especificação do Sistema SACADA

Este capítulo faz uma introdução geral dos requisitos do sistema SACADA, que será utilizado como base para a análise comparativa deste trabalho. A descrição dos requisitos apresentada a seguir não é detalhada, pois o objetivo é apenas mostrar um panorama geral do sistema, para que se possa entender o projeto de arquitetura que será mostrado posteriormente.

3.1 Histórico do Sistema

Este trabalho utiliza o sistema SACADA (Sistema Automático de Coleta e Armazenamento de Dados) como modelo para o estudo de caso que será realizado. Este sistema foi desenvolvido pelo CPqD-Telebrás (Centro de Pesquisa e Desenvolvimento), para um projeto de pesquisa aplicada do Funtel (Fundo de Telecomunicações).

Este sistema foi utilizado como estudo de caso neste trabalho devido a sua complexidade, além da diversidade de funcionalidades, permitindo apresentar com clareza a utilização dos padrões de projeto em um sistema real, e em situações diversas.

Como o sistema já foi implementado, foi possível também realizar uma análise comparativa da arquitetura geral do sistema e do projeto propostos originalmente, sem a utilização de padrões de projeto e padrões de arquitetura, e um novo modelo de arquitetura e projeto focando a utilização de padrões de projeto e padrões de arquitetura, apresentando conseqüentemente os ganhos para o projeto do sistema.

Com relação a alguns detalhes do processo de desenvolvimento utilizado originalmente, este projeto foi desenvolvido utilizando metodologia em cascata passando pelas fases de análise, projeto, implementação e testes. Mas, antes do término da fase de implementação foi possível perceber a falta de alguns requisitos que não haviam sido considerados inicialmente. Desta forma, foi necessário a realização de alterações de requisitos durante a implementação do projeto.

Alterações estas que fizeram com que o sistema se tornasse muito mais complexo do que a modelagem de projeto original poderia suportar, sendo necessário, portanto, abrir concessões e muitas vezes violar a arquitetura original proposta. Desta forma, em vários pontos o projeto se tornou bastante inflexível, não permitindo inserção de algumas funcionalidades requisitadas pelo usuário quando da utilização do sistema.

Neste contexto, o objetivo deste trabalho é demonstrar a aplicação de padrões de arquitetura e projeto de forma a tornar o sistema flexível a alterações de requisitos, mostrando ainda outras vantagens para o projeto e para a equipe de desenvolvimento.

Este sistema foi desenvolvido utilizando a notação UML na fase de análise e projeto e com tecnologia orientada a objetos e linguagem de programação Java na fase de implementação.

3.2 Objetivos do sistema

Este sistema foi desenvolvido para ser utilizado por empresas que prestam consultoria para operadoras telefônicas, realizando coletas das redes celulares para análises por parte das empresas contratantes. Atualmente estas coletas são executadas manualmente por técnicos, *in loco* utilizando equipamentos especializados.

O objetivo do sistema é automatizar a coleta dos dados, de forma que o técnico possa se dedicar mais ao planejamento, pré-teste e a análise dos resultados. Desta forma, já que a fase de coletas, que consome mais tempo neste tipo de serviço, será feita de forma automática, a empresa poderá realizar um número maior de serviços com a mesma quantidade de técnicos.

O sistema SACADA tem como função automatizar a coletas de informações da rede celular. Para isso, coleta medidas de avaliação objetiva de voz, estatísticas de chamadas e parâmetros da rede de forma remota e automática, agendadas pelo técnico.

3.3 Descrição do Sistema

O sistema SACADA é dividido em duas partes que interagem: unidade central e unidades coletoras.

A estrutura do sistema consiste em uma unidade central, que é responsável pelo gerenciamento das informações referentes às unidades coletoras e testes, e unidades coletoras,

responsáveis pela realização dos testes agendados pelo operador, que conhecem e se comunicam apenas com a unidade central. Esta estrutura está ilustrada na Figura 13.

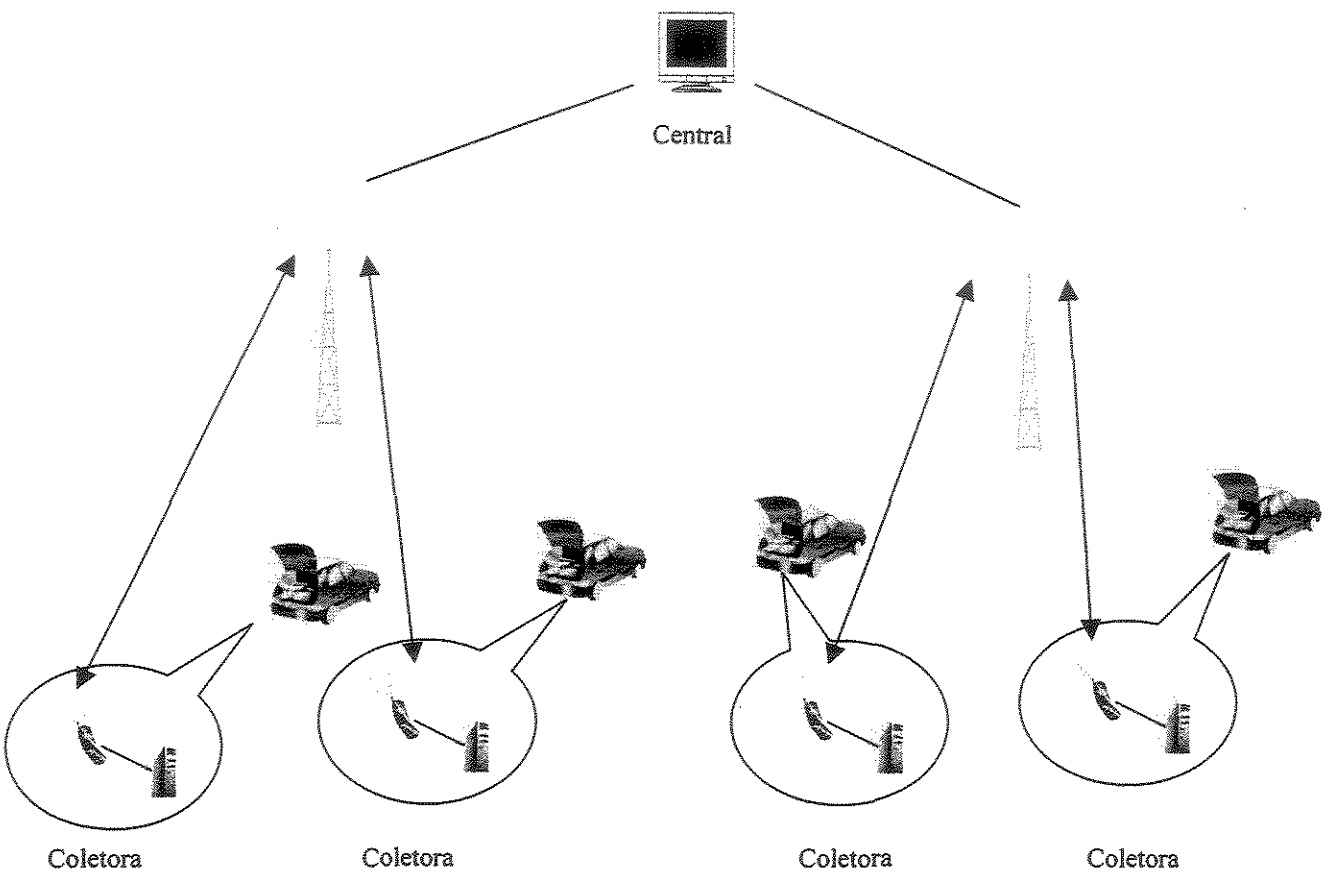


Figura 13. Funcionamento Geral do Sistema SACADA

A unidade central, por ser responsável pelo gerenciamento das coletoras e dos testes, possui todas as informações pertinentes ao sistema como um todo. Desta forma, contém as configurações das unidades central e coletoras cadastradas no sistema, informações sobre as tarefas agendadas e realizadas, e dados coletados resultantes da execução de coletas.

A unidade central possui linhas telefônicas fixas que permitem que ela se comunique com as coletoras cadastradas. Cada coletora possui uma linha celular para que possa se comunicar com a central.

Desta forma, cada unidade coletora recebe requisições de testes da unidade central. A requisição é transmitida da unidade central para a unidade coletora através de chamada telefônica de dados feita da linha telefônica fixa para a linha telefônica celular da coletora, onde são passadas todas as informações necessárias para que o teste agendado possa ser executado. Com estas informações, a coletora pode realizar as medidas para, posteriormente, transmitir os resultados para a unidade central (através de uma chamada de dados da linha celular para a linha fixa da central).

Após a execução da tarefa e recebimento dos dados, a central pode disponibilizar os resultados das coletas que o usuário possa visualizar.

3.4 Funcionamento geral do sistema

A unidade central deverá possuir todas as informações necessárias para que o sistema possa ser iniciado. Estas informações consistem em: telefones para chamadas de dados, e telefones para chamadas de voz. Com base nestes dados, é possível iniciar a comunicação com as unidades coletoras cadastradas, planejar e agendar coletas para as mesmas. Sem estas informações, o usuário não poderá utilizar nenhuma funcionalidade do sistema. Portanto, a configuração da unidade central é um dos pontos chave para que todas as outras funcionalidades do sistema possam ser iniciadas. Este cadastro de informações deverá ser realizado pelo usuário, dando início a testes dos telefones cadastrados, e atualização do estado de funcionamento da central, com base no estado de funcionamento das suas linhas telefônicas.

Após a configuração da unidade central, as funcionalidades oferecidas pelo sistema SACADA já podem ser utilizadas pelo usuário. As primeiras funcionalidades que podem ser realizadas é a visualização da configuração da unidade central, alteração dos dados, teste dos telefones e cadastro de unidades coletoras no sistema.

A unidade coletora, após inicializada pela primeira vez também requer uma configuração inicial que deverá ser feita pelo operador. Durante a inicialização, a coletora realiza a identificação dos equipamentos conectados à ela. Após esta a identificação, o usuário deverá completar informações de configuração dos equipamentos que não podem ser obtidas automaticamente. São executados então testes dos equipamentos de forma a se determinar o

estado de funcionamento de cada um deles, sendo possível identificar quais tipos de coletas a unidade é capaz de realizar.

Após este processo de identificação dos equipamentos, a coletora está pronta para receber informações da central. Os seguintes equipamentos podem ser conectados à unidade coletora: telefones celulares para transmissão de dados, realização de coleta de parâmetros da rede sem fio, avaliação objetiva de qualidade de voz, e estatísticas de chamadas (dados e voz); GPS para obtenção de coordenadas de latitude e longitude; Coletor de CW para realizar coletas de medidas de CW.

Para que o usuário possa planejar e agendar coletas, é necessário que as unidades coletoras responsáveis estejam cadastradas no sistema. Para que o cadastro possa ser feito, o usuário deverá fornecer algumas informações, entre elas o número da linha celular para transmissão de dados conectado à coletora. Com base nestas informações, a unidade central deverá iniciar o processo de habilitação da mesma, realizando um primeiro contato com a unidade coletora que está sendo cadastrada, que informa os equipamentos que possui para coletar dados, os tipos de coletas que estes equipamentos podem executar, o estado de funcionamento de cada um deles, e o estado geral de funcionamento da unidade coletora. Estas informações são recebidas e armazenadas na central para que o usuário possa agendar os testes.

Uma vez que uma unidade coletora esteja habilitada, a unidade central passará a receber mensagens, e poderá enviar mensagens para a mesma, não importando qual lado inicia a comunicação.

O sistema permite que o usuário visualize todas as informações de configuração das unidades coletoras cadastradas, desabilite unidades coletoras para que a unidade central não analise mais mensagens recebidas, mas mantenha seu cadastro, e exclua o cadastro de unidades coletoras desabilitadas.

Além disso, estando as unidades coletoras cadastradas, torna-se possível agendar tarefas de coletas de medidas. As tarefas que podem ser agendadas variam de acordo com os equipamentos instalados nas unidades coletoras e as atividades que os mesmos podem executar. O usuário poderá agendar o seguinte conjunto de tarefas:

- Coleta de Medidas de *sinal continuous wave* (CW)

- Coleta de parâmetros da rede sem fio para as seguintes tecnologias: IS-95 A/B, IS-136 A, CDMA 2000 1x, e GSM.
- Medidas de Avaliação objetiva de voz.
- Estatísticas de chamadas para as tecnologias já citadas.

Além da tarefa a ser executada, o usuário deve escolher a data e horário de início e fim da coleta, bem como a data e horário para transmissão dos dados coletados. Após o agendamento da coleta, a central envia para a coletora as informações referentes à coleta que deverá ser realizada.

Quando a coletora recebe uma requisição de teste, coleta as medidas conforme as informações contidas na requisição recebida (tipo da coleta, e horário de execução do teste e transmissão de dados coletados).

Os dados coletados resultantes desta execução são temporariamente armazenados, até o horário agendado pelo usuário para que estes sejam transmitidos. Neste momento, a unidade coletora inicia a comunicação com a unidade central com o objetivo de enviar os dados coletados.

Quando a central recebe os dados coletados resultantes da execução de uma tarefa em uma coletora, estes dados poderão ser visualizados pelo usuário em editores de texto padrão, ou ainda, os dados poderão ser pintados em um mapa selecionado pelo usuário.

Este trabalho se concentra no projeto de funcionalidades da unidade central, portanto a seção seguinte faz uma descrição mais detalhada das funcionalidades descritas nesta seção.

3.5 Funcionalidades da Unidade Central

Para atender aos requisitos descritos anteriormente, a unidade central deverá possuir as seguintes funcionalidades:

- Cadastrar unidade central: O cadastro da unidade central deve ser realizado para que o sistema conheça e teste os telefones com os quais poderá se comunicar com as unidades coletoras cadastradas. O usuário deve cadastrar telefones para transmissão de dados, que são utilizados para envio de comandos entre a unidade central e unidades coletoras, e vice-versa; e deve também cadastrar os telefones que são utilizados para execução da tarefa de Avaliação Objetiva

de Qualidade de voz e Estatísticas de chamadas – voz. Após a inserção, todos os números de telefone são testados, a fim de verificar o estado de funcionamento (OK ou falha) das linhas.

- Alterar cadastro da unidade central: O cadastro da unidade central pode ser alterado, sendo que é permitido alterar todas as informações inseridas pelo usuário, descritas no item anterior. É importante notar, no entanto, que na existência de unidades coletoras habilitadas, é necessário contatá-las para realizar uma nova habilitação, indicando os novos telefones para transmissão de dados da unidade central.

- Visualizar o cadastro da unidade central: O usuário pode visualizar o cadastro da unidade central, verificando o estado de funcionamento de cada telefone.

- Testar linhas telefônicas: a qualquer momento o usuário pode requisitar o teste de uma ou mais linha(s) telefônica(s). O sistema realiza o teste, e ao fim, apresenta ao usuário o(s) estado(s) da(s) linhas(s).

- Cadastrar unidades coletoras: O cadastro de unidades coletoras deve ser realizado para que o sistema possa conhecer a unidade de coleta e, a partir daí, o usuário possa utilizar todas as funcionalidades oferecidas pelo sistema. Para que o cadastro seja feito, o usuário deve inserir as seguintes informações sobre a unidade sendo cadastrada: número do telefone para transmissão de dados, para que o envio de comandos e transmissão de dados coletados possa ser feito; um nome para a unidade coletora, um identificador numérico que a identificará unicamente entre outras unidades cadastradas, e uma descrição geral sobre seu percurso. O processo de cadastro de uma unidade coletora envolve a habilitação da mesma, como será explicado na funcionalidade "Habilitar unidade coletora".

- Alterar cadastro da unidade coletora: O cadastro das unidades coletoras pode ser alterado, sendo que será permitido apenas que as informações de descrição e número de telefone para transmissão de dados sejam alteradas, já que o nome e o número identificam unicamente uma unidade. No caso de alteração do número de telefone da mesma, esta unidade coletora precisará novamente passar pelo processo de habilitação, que será explicado na funcionalidade "Habilitar unidade coletora".

- Visualizar o cadastro de uma unidade coletora: Após o cadastro de unidades coletoras, o usuário pode visualizar as informações inseridas, bem como os equipamentos e seus estados de

funcionamento (OK ou Falha), e as atividades de coleta que a unidade pode realizar. As últimas são fornecidas pela unidade coletora no momento de sua habilitação, e automaticamente atualizadas periodicamente pela unidade coletora após o seu cadastro.

- Habilitar uma unidade coletora: O usuário pode a qualquer momento requisitar a habilitação de uma determinada unidade coletora. O processo de habilitação envolve um contato da unidade central informando seus números de telefone para transmissão de dados, e uma resposta da unidade coletora informando os equipamentos conectados a ela, os estados de funcionamento destes equipamentos (OK ou falha), as atividades que a mesma pode executar.

- Desabilitar uma unidade coletora: O usuário pode requisitar a desabilitação de uma unidade coletora. Este processo envolve um contato da unidade central com a unidade coletora escolhida, informando que a mesma está sendo desabilitada, e portanto não deve mais enviar informações sobre o estado de funcionamento de seus equipamentos, ou dados coletados. Uma vez desabilitada, o usuário não pode agendar coletas para esta coletora. Uma unidade coletora desabilitada pode ser habilitada a qualquer momento. É importante ressaltar que uma unidade coletora desabilitada não tem seus dados removidos da base de dados da unidade central, para que a configuração seja removida, é necessário realizar a remoção, conforme descrito na funcionalidade "Remover uma unidade coletora".

- Remover uma unidade coletora: O usuário pode remover uma unidade coletora do sistema. Para que uma unidade coletora possa ser removida, é necessário que antes ela tenha sido desabilitada. A remoção envolve um contato com a unidade coletora selecionada, e a remoção do seu cadastro da base de dados da unidade central.

- Atualização automática do estado de funcionamento da unidade coletora: após a habilitação de uma unidade coletora, a mesma executa periodicamente um teste em seus equipamentos, a fim de constatar seu estado de funcionamento. Após o teste, a unidade coletora deve contatar a unidade central, informando o estado de funcionamento de seus equipamentos. A unidade central, ao receber estas informações, atualiza os registros da unidade coletora com os dados recebidos.

- Agendar coletas: O usuário pode agendar coletas de dados para unidades coletoras habilitadas no sistema. No agendamento, o usuário escolhe a unidade coletora que executará a

coleta, o tipo de coleta a ser realizada, os parâmetros específicos da tarefa, como quantidade de chamadas, duração de cada chamada, duração de intervalo para estatísticas de chamadas por exemplo, e o horário de início, horário de fim e horário de transmissão de dados. O sistema valida as informações fornecidas, verificando se existe conflito entre a tarefa que está sendo agendada, e os agendamentos existentes no sistema feitos pelo usuário anteriormente. Esta verificação é feita levando-se em consideração que uma unidade coletora não pode realizar duas coletas simultâneas em um mesmo equipamento. Desta forma, deve ser verificado se para uma tarefa já existe alguma tarefa de mesma atividade agendada, e caso exista, se os horários de execução se sobrepõem em algum ponto. Além disso, será verificado se a unidade central e a unidade coletora podem transmitir os dados coletados no horário escolhido pelo usuário. Para a unidade coletora verifica-se apenas se já existe outra transmissão de dados para o mesmo horário. Para a unidade central, verifica-se se existem linhas para transmissão de dados livres no horário escolhido, de acordo com as transmissões de dados já agendadas anteriormente. Apenas para a tarefa de estatísticas de chamadas e avaliação objetiva de qualidade de voz é feita a verificação de ocupação das linhas da unidade central reservadas para tarefas de estatísticas de chamadas, ou linhas para transmissão de voz (dependendo da modalidade do testes de estatística de chamadas). A falha de qualquer destas condições implica em alteração de horários de início, fim ou transmissão de dados pelo usuário para a tarefa sendo agendada.

Uma vez agendada, as informações ficam armazenadas no sistema, até o momento de serem realizadas, quando o sistema envia para a unidade coletora as informações necessárias para que esta possa executar a coleta.

- Visualizar tarefas: O usuário pode visualizar tarefas agendadas (que ainda não foram iniciadas), em andamento (tarefas que estão sendo executadas pela unidade coletora), e realizadas (tarefas cuja execução já foi terminada pela unidade coletora, e cujos dados coletados resultantes podem já ter sido recebidos pela unidade central).

- Visualizar dados coletados: O usuário pode visualizar os dados coletados recebidos das unidades coletoras. A visualização pode ser feita em arquivo texto, onde os valores medidos serão mostrados em um visualizador padrão, ou sobre mapas, quando o usuário escolhe um mapa, e um arquivo de dados coletados, e são indicados sobre o mapa os valores medidos em suas posições geográficas de latitude e longitude.

- Importar mapas: O usuário pode inserir mapas no sistema para visualizar medidas provenientes de arquivos de dados coletados sobre os mesmos. O usuário deve fornecer ao sistema o caminho e o nome do arquivo de mapa no formato *shapefile*. Apenas após a inserção de um mapa, é possível utilizá-lo na visualização de dados coletados.

- Visualizar mapas: O usuário pode visualizar mapas existentes na base de dados do sistema. Para isso, o sistema disponibiliza os mapas existentes (que foram importados anteriormente) e que podem ser visualizados.

Capítulo 4

Aplicação de padrões de projeto no projeto do sistema

O capítulo 3 descreveu as funcionalidades do sistema SACADA. O foco de interesse deste trabalho está na fase de projeto, onde será realizada a reestruturação do sistema, e aplicação de padrões de arquitetura e projeto.

É interessante ressaltar que a reestruturação proposta para este sistema leva em consideração apenas a aplicação de padrões de alta granularidade, ou seja, os padrões serão aplicados nos módulos do sistema em alto nível, principalmente nas interfaces dos subsistemas, com o objetivo de diminuir a dependência entre eles, de forma que alterações internas aos subsistemas não causem alterações em cascata para seus dependentes, atingindo portanto o objetivo deste trabalho que é flexibilizar a estrutura do sistema de forma a facilitar a alteração de requisitos e manutenção.

Neste capítulo será mostrado o modelo de arquitetura original projetado para o sistema SACADA, e depois, serão detalhados os pontos deste modelo que apresentam problemas, e como estes pontos podem ser alterados e melhorados, utilizando os conceitos de padrões de arquitetura e projeto explicados no capítulo 2. Com base nestas duas propostas, serão realizadas análises comparativas levando em consideração os quesitos: qualidade do modelo, facilidade de entendimento, facilidade de alteração de requisitos e manutenção.

4.1 Processo de utilização de padrões

A aplicação de padrões na reestruturação deste projeto foi feita da seguinte forma: em um primeiro momento, foi realizado um estudo detalhado sobre os padrões de arquitetura e projeto, através dos livros [GAMMA - 95], [SHALLOWAY - 02], [BUSCHMANN - 96], para entendimento dos conceitos de padrões e descrição e exemplos de utilização de cada um deles.

Após este primeiro conhecimento de padrões, foi possível notar que o estudo de padrões é mais bem aproveitado quando o desenvolvedor possui experiências anteriores em desenvolvimento de sistemas, já que desta forma, é possível reconhecer mais facilmente os problemas recorrentes, através de experiências passadas, podendo-se analisar então quais padrões podem ser aplicados.

Foi feito então uma listagem dos problemas apresentados pelo sistema SACADA na fase de manutenção. Esta listagem levou a conclusão que todos os problemas apresentados pelo sistema consistiam de uma mesma fonte: a inflexibilidade do projeto original para incorporar as alterações de requisitos. Localizados os pontos inflexíveis do projeto, foi feita uma análise, ponto a ponto, buscando extrair a raiz do problema da inflexibilidade, para se poder então aplicar os padrões, segundo o tipo de problema encontrado, e resultando em um novo modelo de projeto, flexível, resolvendo os problemas do projeto original.

4.2 Arquitetura geral original do sistema

De acordo as funcionalidades do sistema SACADA, é possível modelar o sistema, levando-se em consideração quatro camadas principais: uma camada de persistência de dados (de acesso à base de dados), uma camada de processamento de dados, uma camada de interface gráfica, e uma camada de transmissão de dados, como pode ser observado na Figura 14.

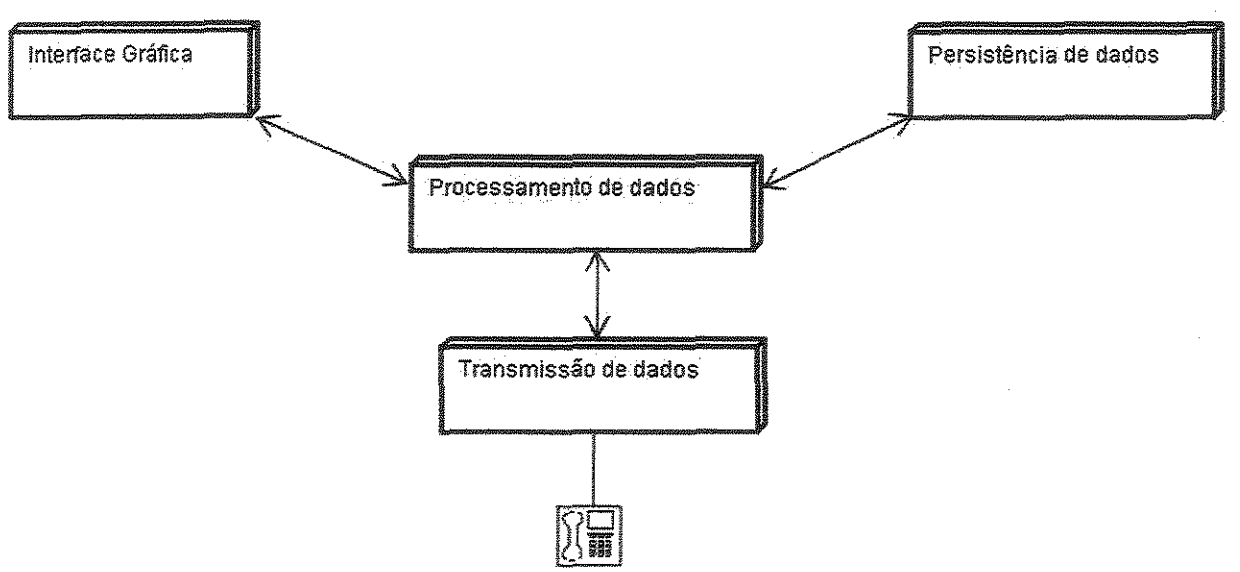


Figura 14. Arquitetura do Sistema Original

A camada de persistência de dados faz o tratamento e acesso à base de dados. O sistema original não utiliza um sistema gerenciador de base de dados, realizando a persistência dos dados através da serialização de objetos do sistema em arquivos. Desta forma, todo o código de acesso à base de dados ficará concentrado em uma camada, facilitando a alteração e inserção de novos algoritmos de manipulação. Para manipular a base de dados, todas as classes do sistema utilizam as classes especificadas na camada de persistência.

As classes da camada de persistência de dados foram projetadas de forma a dar suporte a todas as necessidades de acesso à base de dados, como inserção, alteração, remoção e obtenção de dados utilizando diferentes tipos de filtros.

A camada de interface gráfica contém apenas classes que fazem interação com o usuário. Estas classes não possuem processamento de informações, nem acesso à base de dados. Desta forma, quando uma funcionalidade é requisitada pelo usuário através da interface gráfica, esta repassa a requisição para alguma classe da camada de processamento que seja responsável por realizar a funcionalidade.

A camada de transmissão de dados faz a interface com os telefones, contendo todos os algoritmos necessários para que seja possível originar e receber chamadas telefônicas. Fornece à camada de processamento uma API (*Application Programming Interface)* que permite que o envio e recebimento de comandos entre a unidade central e unidades coletoras seja possível.

A camada de processamento foi projetada contendo todos os algoritmos para processamento das informações armazenadas pelo sistema, e algoritmos para processamento das funcionalidades do sistema. Faz interface com a camada da interface gráfica, para executar as requisições do usuário, com a camada de persistência de dados para o acesso à base de dados, e com a camada de transmissão de dados, para comunicação com as unidades coletoras.

O sistema "original" como um todo foi projetado sem a utilização formal de padrões. Por este motivo, alguns pontos do sistema apresentaram falhas e dificuldades para alteração de funcionalidades e reutilização de componentes. Estas dificuldades serão apresentadas neste capítulo, e será explicado também o ponto do modelo que impede que a alteração de funcionalidades seja realizada, e como a reestruturação do projeto, e aplicação de padrões flexibiliza o sistema de forma a permitira a alteração de forma simples.

4.3 Propostas de soluções de projeto baseadas em Padrões de Projeto

Serão apresentadas a seguir propostas de alteração da arquitetura do sistema original, com o objetivo de torná-lo mais flexível à mudanças de requisitos funcionais, ter maior clareza de entendimento, com módulos separados e estruturas de classes para resolver problemas mais complexos, maior reutilização de componentes do sistema, e até mesmo substituição destes componentes por outros similares que realizem as mesmas funcionalidades.

Inicialmente será mostrada uma solução para a arquitetura geral utilizando padrões de arquitetura, e a seguir, serão apresentados alguns requisitos do sistema que apresentaram problemas de projeto, e como os padrões de projeto ajudariam a resolvê-los. Neste trabalho, serão tratados os aspectos de base de dados, funcionamento da interface gráfica, e recepção de arquivos de dados coletados.

4.2.1 Arquitetura geral do sistema

Para arquitetura geral do sistema, será utilizado o padrão de arquitetura *Model-View-Controller*, já que, de acordo com a especificação descrita no Capítulo 3 trata-se de um sistema interativo, onde a interface gráfica com o usuário representa um papel fundamental no funcionamento do sistema.

Além disso, será mantida a divisão do sistema em camadas, de acordo com responsabilidades: uma camada para interface gráfica, que contém apenas classes que interagem com o usuário (nesta camada estarão contidos os objetos *View-Controller*); uma camada de persistência dos dados, que interage com o sistema gerenciador de base de dados; uma camada de transmissão de dados, que interage com os telefones responsáveis por realizar a troca de dados entre a central e as unidades coletoras; e uma camada de processamento de dados e funcionalidades (esta camada contém as classes *Model* do padrão MVC).

É importante ressaltar que a divisão do sistema em camadas permite que as camadas de acesso, como base de dados, transmissão de dados e interface gráfica possam ter sua implementação alterada a qualquer momento desde que continuem realizando suas responsabilidades, e mantenham a interface de operação esperada, sem a necessidade de alteração das classes da camada *Model*. Além disso, as camadas utilizadas por *Model* podem ser projetadas

de forma genérica o suficiente para que possam ser reutilizadas também por outras aplicações que necessitem de funcionalidades similares. Desta forma, este tipo de arquitetura contribui com a criação de módulos reutilizáveis, e facilita a manutenção e evolução do sistema.

4.2.2 Base de dados

Devido às características do sistema, se faz necessário projetar uma base de dados onde serão armazenadas informações de cadastro da central e coletoras, estado de operação das mesmas, agendamentos de coletas para as coletoras, coletas realizadas, e mapas onde as coletas devem ser localizadas.

O sistema original utiliza um modelo simplificado de base de dados (sem a utilização de um sistema gerenciador de base de dados), que se baseia na serialização de objetos em arquivos, armazenados em diretórios organizados e conhecidos pelo sistema, onde o nome do arquivo o identifica unicamente. Esta base de dados não utiliza nenhum protocolo de comunicação a não ser a busca em diretórios e escrita e leitura de arquivo. Em [LOOMIS - 95] se encontra uma descrição mais completa sobre modelos de bases de dados orientadas a objetos.

Esta base de dados foi proposta devido ao baixo volume e complexidade das informações que seriam armazenadas no sistema, e devido à baixa quantidade de recursos computacionais disponíveis nas unidades coletoras. Mas com a evolução de funcionalidades, ocorreu também um aumento do volume de dados gerados, e esta base se tornou ineficiente, dando margem a erros por acessos concorrentes de *threads*, erros em identificadores dos arquivos, e por fim, inviabilizando o controle de transações.

Em face do problema de ineficiência do esquema de armazenamento de dados original, se fez necessário uma nova proposta de armazenamento. Esta proposta envolve a utilização de sistemas gerenciadores de base de dados (comercial), possibilitando que o sistema utilizado possa ser alterado dependendo do cliente onde o sistema SACADA for instalado. A utilização de sistemas gerenciadores de bases de dados resolve o problema de persistência de dados de forma simples, já que elimina do projeto SACADA a necessidade de se preocupar em evoluir o esquema de persistência de dados original, já que o desenvolvimento de novos modelos de persistência não faz parte dos objetivos do sistema.

O projeto original possui, para cada tipo de dado a ser persistido, um gerenciador de acesso que contém todos os métodos necessários para que a persistência possa ser requisitada pelos objetos clientes. Os objetos clientes[5] instanciam os gerenciadores a fim de persistir as informações. O meio de comunicação entre os gerenciadores e os objetos clientes são objetos que contêm as informações que devem ser persistidas, sendo, portanto, o protocolo de comunicação entre a camada de persistência e a camada de processamento. A Figura 15 mostra este modelo.

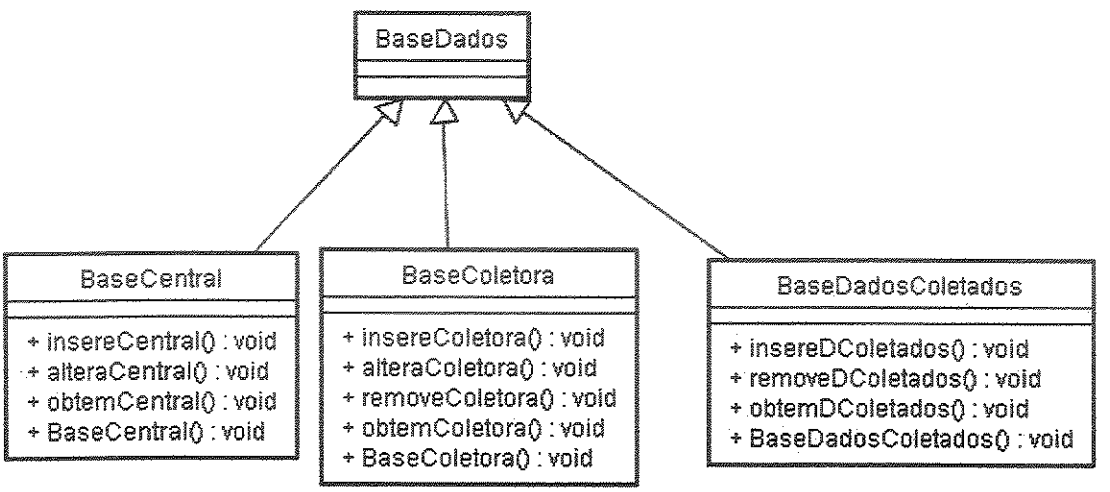


Figura 15. Modelo de Base de Dados original

Pode-se perceber analisando a Figura 15 que o projeto original respeita a arquitetura em camadas, onde as classes de acesso à base de dados estão concentradas em uma camada bem definida, com os gerenciadores de base de dados como interface para os clientes. Por separar a camada de persistência da camada de processamento e realizar um protocolo de comunicação entre elas através de objetos que contêm os dados, o projeto já segue o padrão DAO, mesmo que não formalmente.

Por outro lado, pode-se notar também que o modelo prepara o sistema para interagir com apenas um tipo de esquema de persistência de dados, no caso, a base de simplificada original.

[5] Durante a fase de projeto, foi considerado como classes clientes todas aquelas que acessem uma camada do sistema através de uma API. Desta forma, as classes clientes da camada de acesso à base de dados são as classes da camada de processamento, que utilizam a camada de base de dados para realizar operações de inserção, alteração, remoção e consultas às informações armazenadas.

Isso, porque as classes clientes instanciam classes produtos (no caso os gerenciadores de base de dados) diretamente, tornando-se dependentes destas classes; logo, a criação de novos gerenciadores para um novo tipo de SGBD se torna difícil, já que seria necessário alterar todas as classes clientes que a acessam.

Para inserir um novo tipo de base de dados, por exemplo, uma base de dados relacional, haveria duas possibilidades: uma delas seria criar classes similares, mas com nomes diferentes, e em cada ponto do sistema (classes clientes) onde fosse necessário instanciar um gerenciador, seria necessário escolher este produto, conforme o tipo de SGBD utilizado durante a execução. Uma outra possibilidade seria gerar duas versões do sistema, e para isso, seria necessário copiar todas as classes de gerenciadores de base de dados orientada a objetos, e implementar suas funcionalidades para a base de dados relacional. E, quando do empacotamento do sistema para execução, teria que se escolher qual dos dois conjuntos de classes seria utilizado (de acordo com o SGBD escolhido).

Como é possível notar nenhuma das duas soluções é razoável, já que a primeira implicaria em uma alteração no código de todas as classes clientes toda vez que uma nova família fosse inserida, e a segunda geraria repetição de código e dificuldades no controle de versões do sistema, inserindo erros e diferenças de versões indesejáveis para um mesmo produto.

Com este problema, foi necessário reestruturar a camada de persistência de dados do sistema. O novo modelo proposto deve considerar que o sistema pode trabalhar com diferentes tipos de sistemas gerenciadores de bases de dados (SGBD), com princípios e regras de funcionamento diferentes, como por exemplo, o esquema de armazenamento de dados simplificado original, e um SGBD relacional, baseado em tabelas de dados com linguagem de acesso SQL (*Statement Query Language*), implementação comercial, e maior capacidade de armazenamento e controle de segurança. A implementação do acesso a estes dois esquemas de armazenamento de dados é bem diferente, porque a base de dados simplificada não utiliza a linguagem SQL. E ainda, embora os SGBDs comerciais utilizem a linguagem SQL padrão, cada um deles pode implementar além dos comandos padrão, comandos que auxiliem ou simplifiquem a utilização do sistema. Desta forma, deve-se considerar que o acesso aos SGBDs será diferente, dependendo da base de dados utilizada. Cada um deles possui particularidades de comandos e regras de acesso. Por isso, o projeto deverá contemplar a possibilidade de expansão para novos

SGBDs de forma simples, e sem alterações nas classes clientes que utilizam a camada de persistência.

Com esta preparação, o sistema pode ser instalado nos clientes utilizando o SGBD mais conveniente, de acordo com as necessidades de armazenamento. Mas, a customização por cliente deverá ser feita de forma simples, sem a necessidade de manter versões diferentes do código fonte do sistema que utilizam classes diferentes para fazer o acesso, pois a manutenção de código em diferentes versões é complexa e dá margem à introdução de erros.

Resumindo, o novo modelo considerou os seguintes aspectos: a arquitetura geral em camadas do sistema, a possibilidade de existência de diferentes bases de dados, sem a necessidade de alteração dos outros componentes que a utilizam cada vez que o suporte a uma nova base for necessário, nem mesmo a manutenção de diferentes versões de código, de acordo com a base de dados que está sendo utilizada.

O novo projeto foi modelado de forma que as classes clientes da base de dados não necessitem conhecer o tipo de SGBD que está sendo utilizado, sua linguagem de acesso, etc., possibilitando que os objetos clientes vejam a persistência de dados de forma única, através de interfaces que disponibilizam todas as operações que podem ser realizadas, como por exemplo, inserção, remoção, alteração ou consulta de informações armazenadas. Além disso, o meio de comunicação de informações entre os clientes e a camada de persistência serão objetos de armazenamento temporário de dados. Estes objetos serão preenchidos pelo cliente e enviados para a camada de persistência quando da inserção de um registro na base, da mesma forma que estes objetos serão preenchidos pela camada de persistência, e devolvidos ao cliente quando este requisitar a consulta de informações na base de dados.

Para que este objetivo fosse alcançado, foi necessário fazer uma modelagem onde se mantêm duas camadas bem distintas: os clientes que manipulam objetos, e os gerenciadores de acesso, que se preocupam em transformar os objetos em informações que podem ser armazenadas permanentemente ou obtidas da base de dados.

Para manter a arquitetura geral proposta para o sistema, as classes de persistência deverão ficar separadas das classes clientes do sistema. Este é o princípio seguido pelo padrão DAO, tendo como objetivo principal concentrar as classes de acesso à base de dados em uma camada

bem definida, para que o código de acesso em baixo nível (conhecimento de detalhes de acesso do produto) não fique espalhado pelo sistema.

Para cada tipo de dado a ser persistido, foi criado um gerenciador de acesso, e este possui todos os métodos necessários para que a persistência possa ser requisitada pelos objetos clientes.

É possível notar também que a natureza deste problema consiste em famílias de objetos distintas, e neste caso, foram definidas, em princípio, duas famílias: uma contendo os gerenciadores de base de dados simplificada (que será chamada genericamente de base de dados orientada a objetos), e outra contendo gerenciadores de base de dados relacional. Estas famílias não deverão coexistir em uma mesma aplicação. Cada instância do sistema será iniciada com um SGBD específico (dependendo do cliente onde o sistema será instalado).

Além disso, novamente, as classes clientes não devem conhecer o tipo de SGBD que está sendo utilizado no sistema. Devem conhecer apenas a interface de acesso (gerenciadores).

A existência de famílias de objetos que não coexistem, associada ao fato de que os objetos cliente que as utilizam não possuem conhecimento sobre as mesmas, caracterizam a utilização do padrão *Abstract Factory*. A utilização deste padrão permitirá que os clientes se tornem independentes da família utilizada, já que não realizam a instanciação direta de objetos desta família, deixando a instanciação para a *factory* responsável por isso, e utilizando apenas a interface disponível. Neste caso, os objetos produto são os gerenciadores de base de dados, que serão utilizados pelos objetos cliente (classes da camada de processamento de dados do sistema, que necessitam manipular a base de dados).

A camada de acesso à base de dados possui interfaces que serão utilizadas pelos clientes, e classes concretas que implementam estas interfaces para cada tipo de base de dados existente no sistema, utilizando os modelos de comunicação específicos definidos para cada esquema de persistência de dados.

Seguindo o padrão *abstract factory*, é necessário ainda a criação do intermediador responsável por instanciar os produtos. Este intermediador é chamado *Factory*. Os clientes não instanciarão as classes produtos (gerenciadores) diretamente, já que não possuem conhecimentos sobre classes concretas, apenas sobre as interfaces. Portanto, eles deverão requisitar esta instanciação para a *Factory*, que sabe qual família de base de dados está sendo utilizada pelo

sistema e, portanto, qual objeto produto deverá ser instanciado. Será declarado um *factory* abstrato, BaseDados_Factory, e duas classes concretas, que implementam esta classes abstrata, de acordo com as particularidades de cada fabricante de SGBD. Desta forma, teremos duas classes concretas, BaseOO_Factory, que criar gerenciadores para base de dados orientada a objetos, e BaseRelacional_Factory, que cria gerenciadores que realizam o acesso ao SGBD relacional.

Neste modelo, as classes clientes conhecem as seguintes interfaces: BaseCentral, BaseColetora, e BaseDadosColetados que representam os gerenciadores da base de dados, (produtos) e a interface BaseDados_FactoryInterface, que contém os métodos responsáveis pela criação de cada gerenciador.

Observe-se que o cliente só tem conhecimento das interfaces, tanto dos produtos, quanto da *factory*. Uma tarefa que exige experiência e tempo é projetar interfaces que consigam atender a todas as funcionalidades que as classes herdeiras realizarão.

Sendo implementado a partir deste modelo, garante-se que o código de manipulação das bases está concentrado em uma camada, sem o conhecimento do sistema como um todo. Além disso, garante-se também que a inserção de um tipo de base de dados novo, produzida por outro fabricante, pode ser feita de forma simples, sem a necessidade de alteração dos clientes.

A interface BaseDados_FactoryInterface contém o método getInstance(). A classe abstrata BaseDados_Factory que implementa esta interface, deve implementar o método getInstance(), que é estático e responsável por instanciar uma *factory* concreta de acordo com os parâmetros do sistema a serem analisados, como por exemplo, o tipo de base de dados a ser utilizada, informação que poderá estar presente em um arquivo de configuração do sistema. As classes *factory* concretas são *singletons*, pois estas *factories* serão responsáveis por gerenciar a instanciação dos produtos. Além disso, as *factories* deverão ter um ponto de acesso bem definido e conhecido para todo o resto do sistema.

Para obter um gerenciador de base, o cliente deverá primeiramente obter a instância de uma *factory* concreta. Para isso, deverá utilizar o método estático getInstance() da BaseDados_Factory. Este método retornará uma *factory* concreta. Após isto, o cliente conseguirá criar uma base de dados, por meio dos *factory methods* (CriaBaseCentral, CriaBaseColetora, CriaBaseDColetados) declarados em BaseDados_Factory.

A Figura 16 mostra o novo modelo de base de dados descrito.

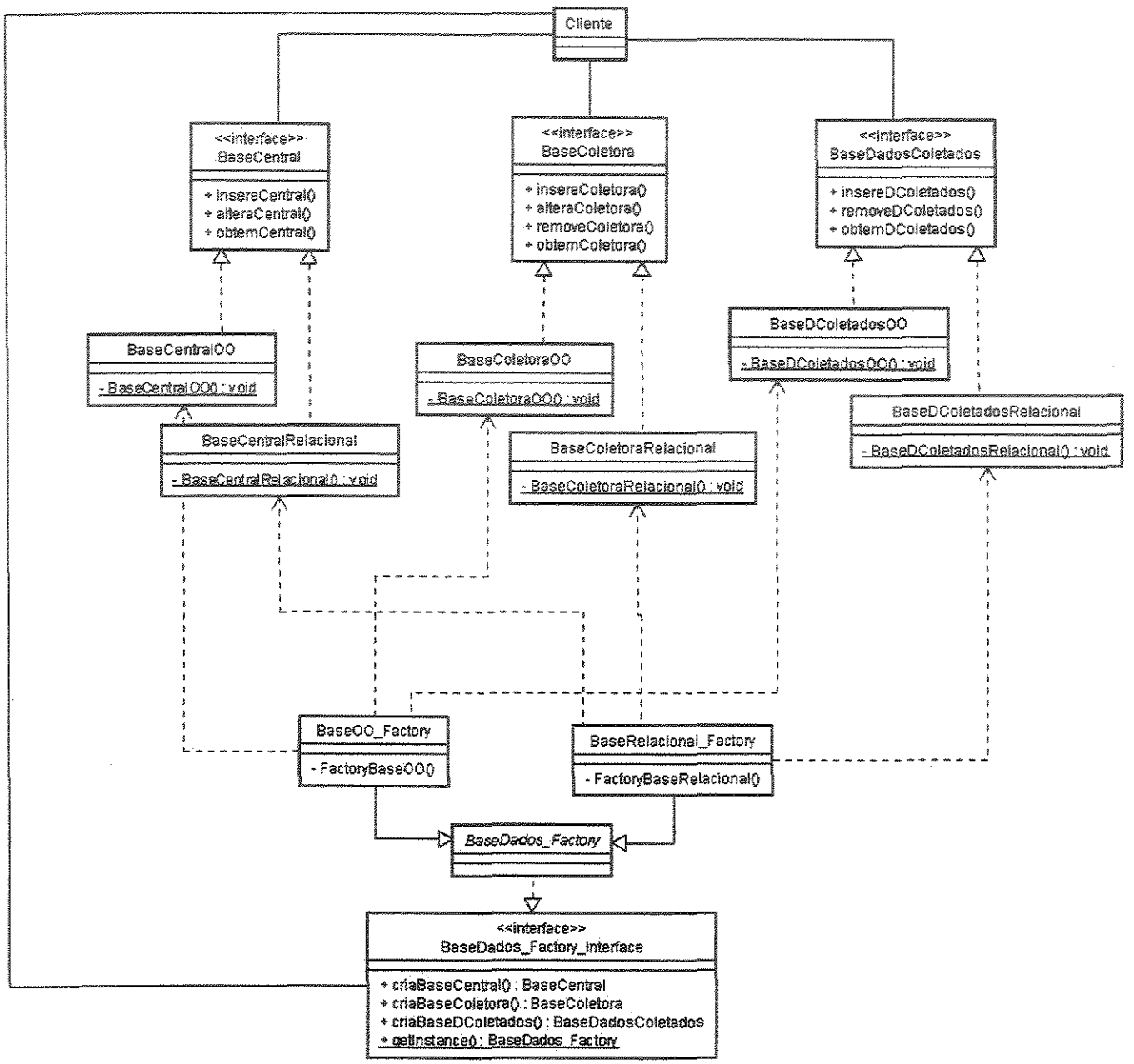


Figura 16. Novo Modelo de Base de Dados

Caso haja a necessidade de se inserir um novo tipo de base, será necessário derivar dos gerenciadores abstratos (BaseCentral, BaseColetora, BaseDadosColetados) gerenciadores concretos que implementem as funcionalidades de acordo com as particularidades de acesso da

base de dados sendo inserida, e derivar uma nova classe concreta de BaseDados_Factory que criará os gerenciadores corretamente quando estes forem requisitados pelo cliente. E o cliente não precisará ser alterado, já que conhece apenas as interfaces (que não foram alteradas com a inserção da nova base de dados), não se importando com as classes concretas que as implementam.

Desta forma, pode-se concluir que, a utilização dos padrões DAO e Abstract Factory foi bastante eficiente, já que sua utilização é simples, respeita a arquitetura geral do sistema em camadas bem separadas, auxilia a criação de um módulo de persistência de dados, facilita a manutenção do sistema, e permite a alteração de requisitos da aplicação de forma rápida, e com menor possibilidade de introdução de erros.

Uma das desvantagens da utilização deste padrão está no fato de que o número de classes geradas é bem maior, já que existe a necessidade de suporte a classes que poderão vir a existir, mas que não necessariamente serão concretizadas. Portanto, se por um lado, no caso de alteração de requisitos a estrutura não é abalada, por outro, caso exista apenas uma família de objetos, a estrutura é muito grande.

4.2.3 Pool de Conexões

O esquema de persistência de dados original do sistema não utiliza o protocolo JDBC (*Java Database Connectivity*) para comunicação, e não precisa, portanto do estabelecimento de uma conexão para que o sistema possa acessá-la. Porém, para a utilização de SGBDs relacionais comerciais, é convencionado em Java a utilização do protocolo JDBC, sendo necessário o estabelecimento de uma conexão para que o acesso a base de dados possa ser realizado[6].

O estabelecimento de conexões com o SGBD é um processo que demora algum tempo, já que as informações de *login* e senha do usuário, devem ser verificadas pelo sistema, antes que os acessos possam ser efetuados. Dependendo da quantidade de conexões estabelecidas durante a execução de um aplicativo, este processo pode se tornar bastante lento, e sobrecarregar as funcionalidades do sistema.

---

[6] Em [JONG - 04] é proposto uma implementação para obtenção de conexões via protocolo JDBC.

Para melhorar o desempenho da aplicação, será interessante manter um conjunto de conexões pré-estabelecidas, que poderão ser utilizadas sempre que um acesso à base de dados for requisitado. Como a conexão não será estabelecida sob demanda (para cada objeto que precisar acessar a base de dados, é necessária a obtenção de uma nova conexão), o tempo necessário para a realização de uma operação na base de dados irá diminuir.

Por este motivo, foi proposto um modelo de classes que resolvem o problema de desempenho. O conjunto de conexões com as características descritas acima é chamado *pool* de conexões. O projeto deste *pool* envolve a utilização dos padrões *object pool* e *singleton*. Foi projetada uma classe responsável pelo gerenciamento das conexões, chamada ConnectionPool. Esta classe tem uma instância única no sistema, e um ponto de acesso visível a qualquer objeto que queira utilizá-la. Assim, esta classe é um *singleton*. Todas as classes de base de dados que precisarem de uma conexão com a base de dados, utilizam o ConnectionPool para obter esta conexão, e são chamadas classes clientes.

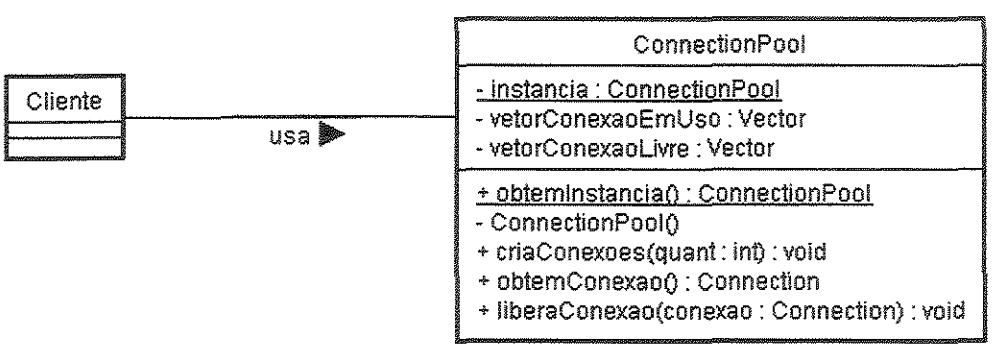


Figura 17. Modelo de *Pool* de conexões

A Figura 17 representa o diagrama de classes proposto para o pool de conexões. A classe ConnectionPool mantém dois vetores para controle das conexões, um que indica quais são as conexões livres, e outro que indica as conexões em uso. Como existe apenas um objeto ConnectionPool no sistema, estes vetores são estáticos, e o acesso a eles é sincronizado, evitando que dois métodos os acessem simultaneamente, impedindo que estes dados cheguem a um estado de inconsistência (por este motivo os métodos obtemConexao() e liberaConexao() também são sincronizados). O método obtemInstancia() é utilizado para controlar o acesso a uma instância única da classe no sistema, e o construtor da classe é privado, evitando que objetos a instanciem.

As informações de conexão (localização da base de dados, usuário e senha) ficam armazenadas em um arquivo de propriedades. Na primeira instanciação de um objeto desta classe, são criadas as conexões, que são armazenadas no vetor de conexões livres. O método obtemConexao() é bloqueante, pois caso não existam conexões disponíveis, o objeto deverá ficar bloqueado, aguardando que alguma conexão seja liberada.

O cliente deverá obter o *pool* de conexões por meio do método obtemInstancia() e depois deverá utilizar o método obtemConexao() para obter uma conexão através da qual deverá realizar os acessos à base de dados. Após a utilização da base de dados, deverá liberar a conexão por meio do método liberaConexao().

A classe ConnectionPool está localizada na camada de persistência de dados, e seus clientes serão os objetos derivados das classes BaseCentral, BaseColetora, e BaseDadosColetados (pertencentes à mesma camada).

Para o sistema original não foi projetado nenhum modelo de *pool* de conexões, porque a base de dados utilizada originalmente não utiliza o protocolo JDBC. Por este motivo, não será possível realizar a análise comparativa entre o projeto original e o novo projeto proposto, mas ainda assim, pode-se destacar que a utilização dos padrões *ObjectPool* e *Singleton* permitem que as classes de bases de dados compartilhem conexões, de forma a otimizar o tempo de acesso, e propõe uma estrutura simples e organizada para o gerenciamento das conexões do sistema, aumentando, portanto a qualidade do projeto, e a facilidade de manutenção.

4.2.4 Interface gráfica

A interface gráfica do sistema é composta por menus que oferecem ao usuário as funcionalidades descritas no capítulo 3.5, e um painel, onde são mostradas figuras representando a unidade central configurada, e as unidades coletoras configuradas. A unidade central é apresentada em cores diferentes, de acordo com o seu estado de funcionamento: vermelha em caso de falha em algum de seus telefones e azul se todos os telefones estiverem funcionando corretamente. As unidades coletoras são desenhadas nas seguintes cores, também de acordo com seu estado de funcionamento: vermelha em caso de falha em algum de seus equipamentos, ou se houver falha na comunicação com a mesma, cinza caso esteja desabilitada, e azul caso todos os seus equipamentos estiverem funcionando corretamente. Além disso, tanto a unidade central

quanto as unidades coletoras só são inseridas na tela conforme suas configurações são realizadas pelo usuário, e as unidades coletoras são removidas da tela conforme o usuário requisita a remoção da configuração da base de dados.

4.2.4.1 Funcionalidades Associadas

O projeto original do sistema levou em consideração apenas a arquitetura MVC, criando duas camadas bem separadas, uma para interface gráfica e outra para processamento dos dados, comunicando-se por objetos *Façade*.

É importante ressaltar que a utilização destes padrões não foi feita formalmente, apenas pela experiência da equipe no desenvolvimento de outros sistemas, que mostraram a importância de se separar a interface gráfica da camada de processamento por questões de simplicidade, e a importância de simplificar a camada de processamento para a interface gráfica. Será explicado a seguir como os padrões MVC e *façade* foram aplicados no sistema.

A falta de conhecimento dos padrões no projeto original, aliada às alterações de requisitos durante a fase de implementação causaram em muitos pontos do sistema a violação destes dois padrões.

Este trabalho propôs a utilização dos padrões MVC, *façade* e *command* para o novo modelo.

Para resolver o problema de organização das classes da camada de processamento que executam as funcionalidades da interface gráfica, foi mantida a utilização do padrão *Model-View-Controller*, que separa o pacote de interface gráfica do pacote de processamento, permitindo que o primeiro tenha conhecimentos sobre o segundo, mas que o inverso não seja válido. Este padrão deve ser mantido em todas as classes do sistema.

Foi mantida também a utilização do conceito de classes *Façade*, que são responsáveis por simplificar o pacote de processamento para a interface gráfica, disponibilizando apenas as funcionalidades necessárias para que as operações da interface gráfica possam ser processadas. Desta forma, existirão três tipos de classes: As classes da camada de interface gráfica, as classes *façade* pertencentes à camada de intermediária (que realizam a comunicação entre a interface gráfica e processamento), e as classes da camada de processamento. É importante observar

apenas que a camada de processamento de dados original, representada na Figura 14, foi dividida apenas para ilustrar melhor a situação em : camada intermediária e camada de processamento de dados, como é mostrado na Figura 18:

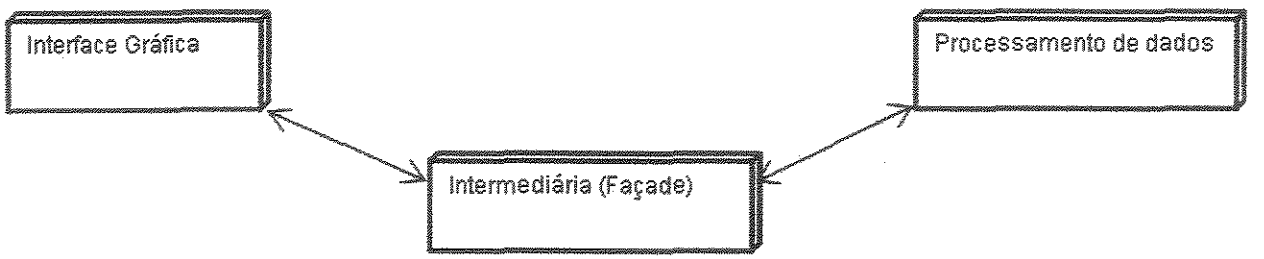


Figura 18. Interação entre Camadas

A camada de interface gráfica utiliza os objetos *Façade* para obter informações que são apresentadas ao usuário, e para redirecionar o tratamento de eventos gerados pelo usuário. Como os eventos gerados na interface gráfica são redirecionados para classes de processamento através dos objetos *Façade* da camada intermediária, alterações na camada de processamento não causam alterações na interface gráfica, desde que as interfaces das classes *Façade* se mantenham iguais e estas continuem executando suas funcionalidades.

Na camada de processamento, foi proposto a utilização do padrão *Command* para efetivamente realizar as funcionalidades requisitadas pelo usuário através da interface gráfica. A aplicação deste padrão deve-se ao fato que ele tem como função separar a execução de funcionalidades de forma que fiquem independentes dos clientes que as utilizam. Ou seja, atualmente as funcionalidades são comandadas por um usuário através da interface gráfica disponibilizada pela central mas, futuramente, as funcionalidades poderiam ser disparadas por uma interface gráfica em um cliente web, ou por um *script* que indicaria o funcionamento esperado. Utilizando este padrão, não importa quem seja o cliente que inicia a utilização das funcionalidades, há a garantia de que as informações necessárias para que estas possam ser executadas sejam fornecidas, as funcionalidades serão realizadas.

Para utilizar este padrão necessitou-se criar uma interface *Command* que representa um comando. Depois disso, derivou-se de *Command* classes específicas para a realização de cada funcionalidade. De acordo com a especificação funcional descrita anteriormente, foram criadas

classes derivadas para as seguintes funcionalidades: Configurar Unidade Central, Testar telefones, Configurar Unidade coletora, Habilitar Unidade coletora, Desabilitar Unidade coletora, Remover Unidade coletora, Agendar Tarefas, Visualizar dados Coletados. Para exemplificar o que foi feito, criaremos classes apenas para as funcionalidades de configurar Unidade Central, Configurar Unidade coletora e Agendar Tarefas, e todas as outras funcionalidades devem seguir o mesmo procedimento. Estas classes implementam a interface *Command*, que possui apenas o método executar(). Este método vai disparar a execução do comando em questão. Cada classe derivada possui um construtor que recebe como parâmetros todas as informações necessárias para que o comando possa ser executado (todas as informações de entrada do usuário na interface gráfica). Estas informações podem também ser passadas através do preenchimento de objetos com função apenas de armazenar as informações.

A Figura 19 mostra o diagrama de classes que representa a situação descrita anteriormente. Neste diagrama, são mostradas apenas as classes responsáveis por configurar a unidade Central, configurar unidades coletoras, e agendar tarefas.

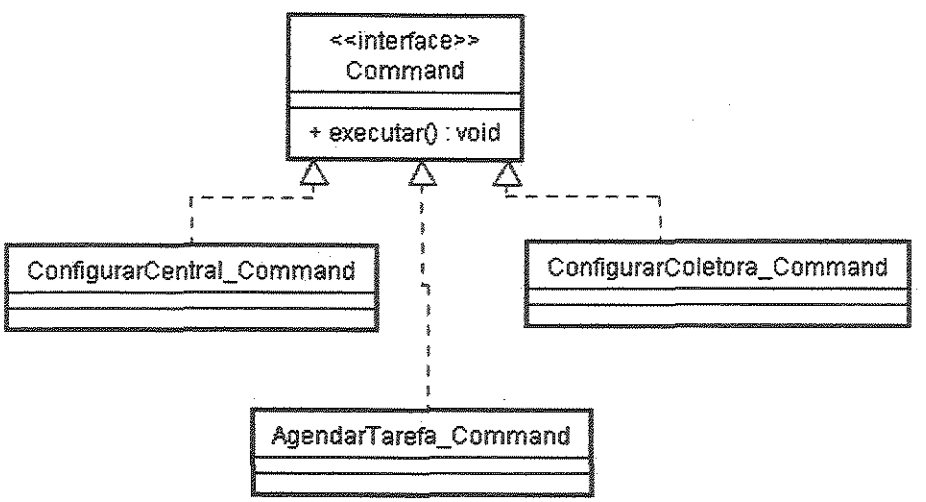


Figura 19. Diagrama de classes *Command*

A Figura 20 demonstra o relacionamento completo entre as 3 camadas descritas anteriormente. Para separar as camadas, as classes referentes a cada uma delas foram inseridas em um pacote, gerando, portanto os pacotes *Commands*, InterfaceGrafica e *Façade*.

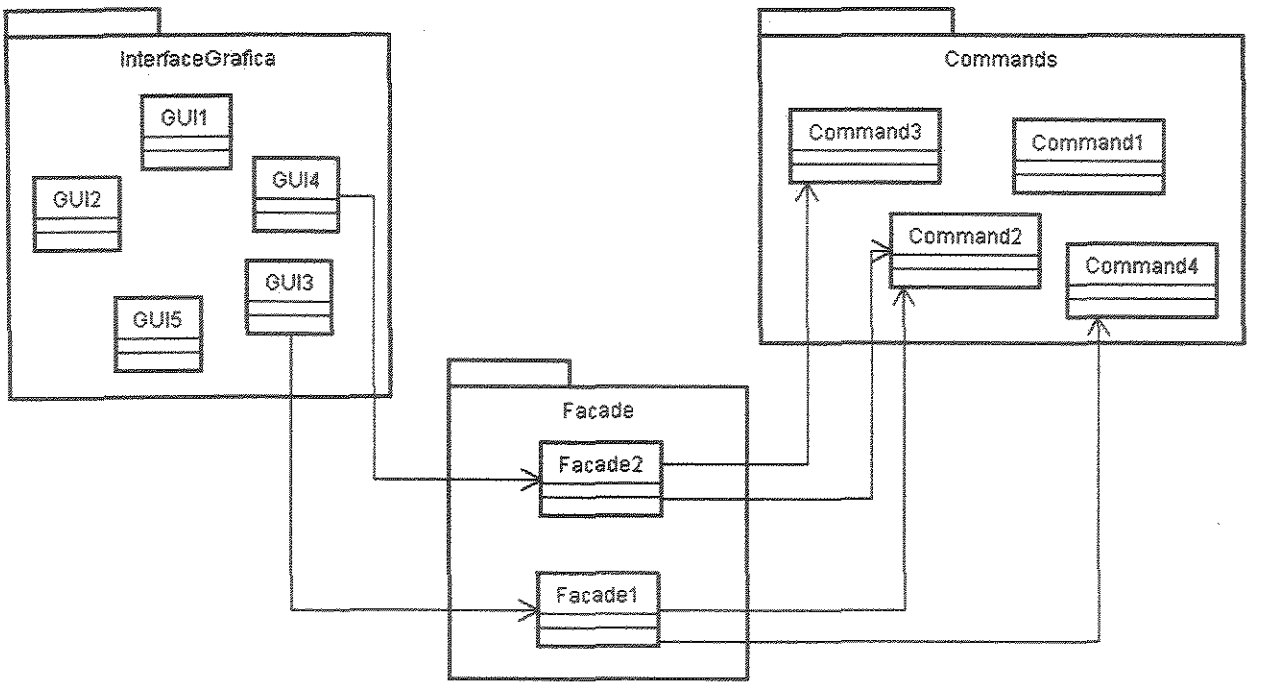


Figura 20. Interação das camadas *Commands*, *Façade* e InterfaceGrafica

Desta forma, realizando uma análise comparativa entre os dois modelos mostrados (original e proposto), é possível notar que a utilização dos padrões *Façade* e *Command* associados ao padrão de arquitetura MVC trazem modularidade e simplicidade à interação entre interface gráfica e processamento de requisições, de forma que ambas podem ser alteradas sem interferir uma na outra, além de permitir que diferentes clientes possam iniciar a execução de funcionalidades do sistema (facilidade de alteração de requisitos na interface gráfica).

4.2.4.2 Gerenciamento de figuras

Para que o usuário possa acompanhar o funcionamento da unidade central e das unidades coletoras, são disponibilizadas figuras na janela principal do sistema, sendo que a central é representada por um computador, e as coletoras são representadas por aparelhos celulares. Estas figuras terão suas cores alteradas conforme as atualizações no estado de funcionamento ocorridas durante a execução do aplicativo.

O projeto original do sistema violou o padrão MVC já que possui apenas uma classe para desenhar as figuras (DesenhistaFigura) e uma classe representando a figura a ser desenhada (Figura), pertencentes à camada de interface gráfica, e todas as classes de processamento que

realizam alterações no cadastro da central e coletoras acessam diretamente a classe responsável pelo desenho, indicando que uma figura deve ser inserida, alterada ou removida da base de dados. A Figura 21 mostra o modelo original do sistema.

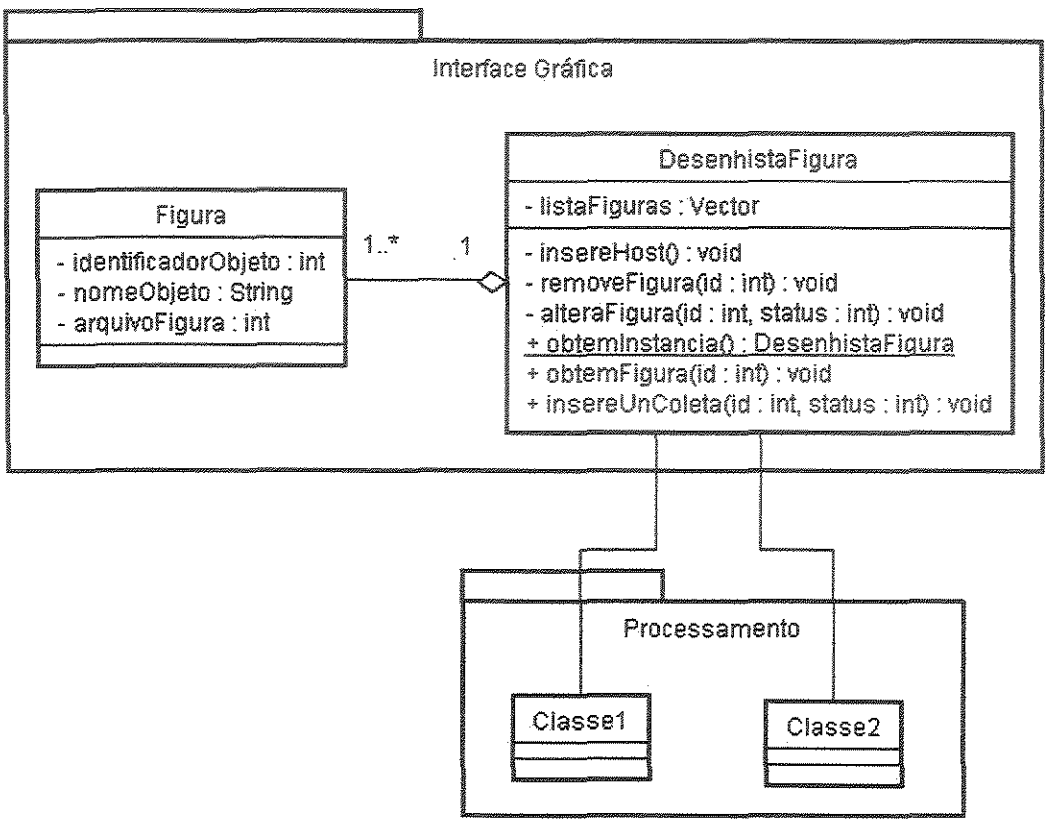


Figura 21. Modelo original de gerenciamento de figuras

Este modelo é ruim porque exige que o acesso à interface gráfica e conhecimento da alteração de figuras esteja espalhado em vários pontos nas classes de processamento, não respeitando a arquitetura MVC definida para este projeto. Além disso, alterações no modo de funcionamento das classes Figura e DesenhistaFigura podem causar alterações em vários pontos do código.

É necessário, portanto um mecanismo que permita inserir, remover ou alterar desenhos na tela de acordo com eventos ocorridos no sistema, sem perder de vista o tipo de arquitetura utilizado, MVC, onde os elementos da interface gráfica devem estar bem separados dos elementos de processamento do sistema. Desta forma, foi proposto um novo modelo para o

mecanismo de gerenciamento de figuras, de forma a eliminar os problemas descritos anteriormente.

Para este novo modelo, foram criadas as seguintes classes responsáveis por desenhar na tela: Figura, responsável por manter o arquivo de desenho, o identificador do objeto (unidade central ou unidade coletora), e o tipo do objeto (unidade central ou unidade coletora); e Desenhista, que é responsável por desenhar um objeto Figura no painel da tela.

Analisando o problema, é possível perceber que a inserção e remoção destas figuras, bem como alteração de suas cores, estão totalmente ligadas a eventos processados que alteram o estado dos elementos unidade central e unidades coletoras na base de dados. Este comportamento é característico do padrão *Observer*, já que os gerenciadores da base de dados caracterizam os objetos a serem observados, e a classe Figura deve ser notificada de acordo com alguns eventos ocorridos nos primeiros (objeto observador). Ou seja, os gerenciadores de base BaseCentral e BaseColetora são os Publishers, porque são origens dos eventos, e a classe DesenhistaFigura é subscriber, já que deve ser notificada quando um evento de alteração da base de dados ocorrer.

Foi utilizado o conceito de eventos e observadores de eventos (events e eventListeners) definidos em Java. Definiu-se duas classes de eventos chamadas EventoColetoraBaseDados e EventoCentralBaseDados, que representam as alterações nas bases de dados das unidades coletoras e da central respectivamente. Com estes objetos, os observadores de eventos conseguem saber o que ocorreu.

Foram modeladas também interfaces para *listeners* chamadas CentralBase_Listener e ColetoraBase_Listener, que devem ser implementadas pelas classes que desejarem observar os eventos ocorridos. Estas interfaces definem alguns métodos que representam os tipos de eventos que podem ocorrer na base de dados. Como a classe DesenhistaFigura é a observadora, deve implementar estas duas interfaces.

As classes derivadas de BaseCentral e BaseColetora são responsáveis por gerar eventos. Portanto, as classes que desejarem observar estes eventos, devem se registrar nelas como *listeners*. O registro é feito através do método insereListener e removeListener, existente nestas classes. Quando alguma alteração for realizada nas bases de dados, as classes geradoras de eventos deverão criar um objeto EventoCentralBaseDados ou EventoColetoraBaseDados,

preenchendo todos os atributos que descrevem o evento ocorrido, e de acordo com o evento, notifica através de um método da interface *listener* todos os objetos que implementam CentralBase_Listener ou ColetoraBase_Listener. É possível observar neste ponto, que os objetos derivados de BaseCentral e BaseColetora não conhecem as classes concretas registradas como listeners, conhecem apenas as interfaces, com quem se comunicam.

Como existe apenas uma central, o objeto EventoCentralBaseDados contém apenas o atributo fonte, que indica que objeto gerenciador da base de dados gerou o evento. Já o objeto EventoColetoraBaseDados contém os seguintes atributos: idColetora e fonte, para que os observadores consigam saber qual unidade coletora foi alterada, e qual base de dados gerou o evento.

Os métodos definidos para a interface CentralBase_Listener são os seguintes: centralComFalha, centralOK, centralInserido. Os métodos definidos para a interface ColetoraBase_Listener são os seguintes: coletoraComFalhaParcial, coletoraComFalhaTotal, coletoraOK, coletoraDesabilitada, coletoraHabilitada, coletoraRemovida, coletoraInserida. Todos estes métodos recebem como parâmetro um objeto EventoBaseDados, possibilitando ao objeto observador realizar as atividades de alteração de figuras quando um evento ocorrer.

A Figura 22 representa o diagrama de classes para o gerenciamento das figuras descrito acima.

Neste ponto, observe-se que a aplicação do padrão além de respeitar o padrão de arquitetura MVC, deixou o projeto mais simples em relação ao original, já que as classes da interface gráfica e de processamento interagem por meio de eventos; existe também a obrigação de comunicação com interfaces, e não com as classes concretas que as implementam; e por fim, o novo projeto permite uma maior flexibilidade em relação ao anterior, pois caso novas classes precisem realizar funções junto à interface gráfica de acordo com eventos ocorridos na base de dados, necessitam apenas implementar a interface de *listener* e se registrar na base de dados, não sendo necessário que a base de dados ou qualquer outra classe do sistema tenha conhecimento destes objetos.

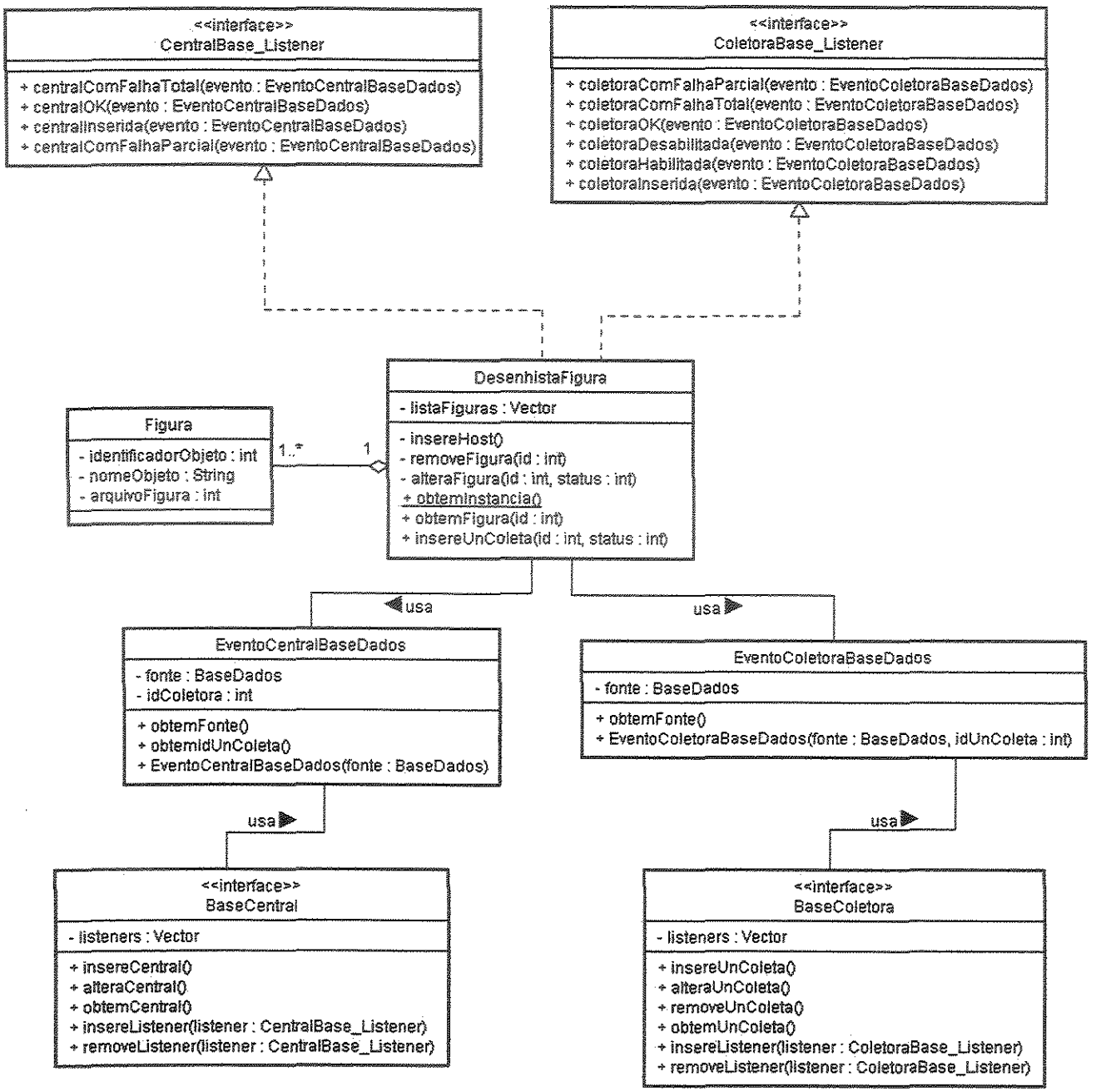


Figura 22. Novo modelo de gerenciamento de figuras

4.2.5 Recepção de Dados Coletados

A central recebe dados coletados de unidades coletoras, de acordo com os agendamentos realizados pelo usuário. Os dados coletados podem ser visualizados pelo usuário de duas formas diferentes: em arquivo texto, onde são apresentadas todas as informações recebidas da unidade coletora para uma determinada coleta escolhida pelo usuário; ou em mapas, onde são mostradas

apenas as medidas sobre os pontos geográficos onde elas ocorreram (coordenadas de latitude e longitude).

O arquivo texto contendo os dados coletados é exibido em um visualizador de textos padrão, que é definido de acordo com o sistema operacional onde o software foi instalado e executado.

Para visualização do mapa, é utilizado o visualizador *open source* OpenMap, que permite a visualização de um mapa escolhido pelo usuário em formato *shapefile*, e através de um arquivo em formato CSV (*Comma Separated Values*) a exibição das medidas associadas a pontos geográficos de diferentes cores pintadas em uma camada em cima do mapa.

Assim, na recepção dos arquivos de dados coletados, o sistema deve interpretá-los de forma a gerar dois arquivos diferentes: um arquivo texto, e um arquivo CSV. E ainda, estas interpretações irão depender do tipo de coleta (CW, Estatísticas de Chamadas, Avaliação objetiva de qualidade de voz).

Nos requisitos iniciais do sistema (quando o projeto foi modelado) existia apenas a possibilidade de visualização de arquivos em formato texto. Desta forma, o projeto original considerou apenas a criação de interpretadores de arquivos para o formato texto. Posteriormente à implementação dos interpretadores texto, foi necessária a inserção de interpretadores para arquivos CSV.

O projeto original define uma classe abstrata Interpretador que é responsável por realizar a análise e interpretação dos arquivos de dados coletados. Esta classe apresenta como resultado um arquivo que pode ser visualizado em formato de texto ou sobre um mapa. Para isto, a classe Interpretador possui várias classes concretas que a implementam de forma que se respeite todas as variações: variações de tipos de arquivos de entrada (dependente do tipo de coleta executada: CW, estatísticas de chamadas dados e voz, e avaliação objetiva de voz), e variações dos tipos de arquivos de saída: texto ou CSV. A Figura 23 mostra a hierarquia de classes que representa o problema descrito.

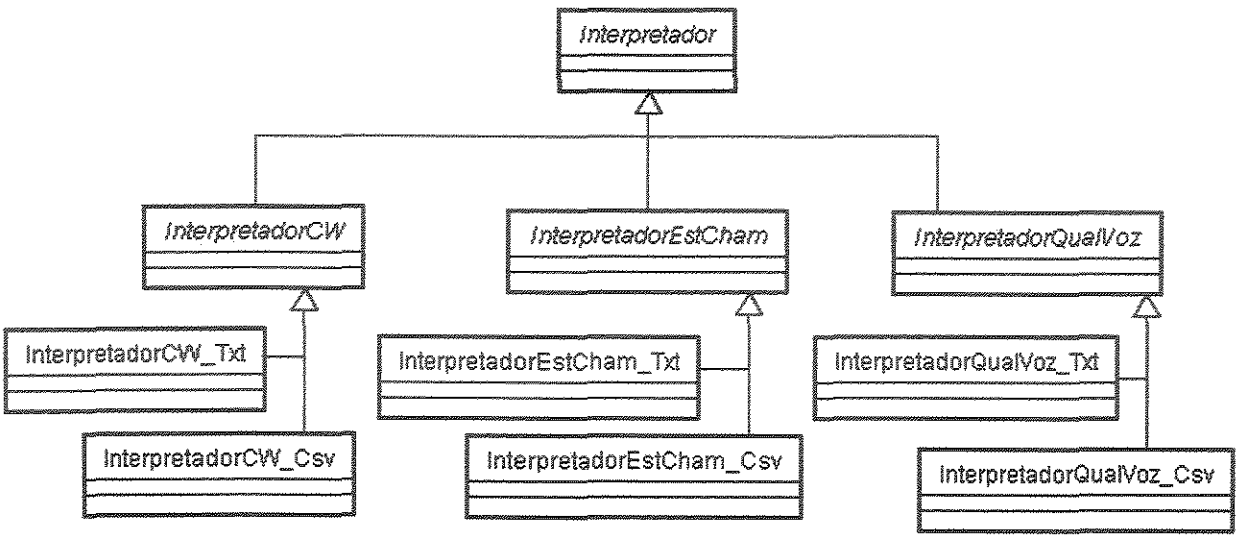


Figura 23. Projeto original de Interpretadores de dados coletados

Esta hierarquia poderá aumentar ainda mais quando se considera que tanto o tipo de arquivo de entrada quanto o tipo do arquivo de saída poderão variar, gerando novas classes no diagrama. Pode-se perceber que o erro cometido neste projeto está no fato de que não se considerou a variação de tipos de arquivos de saída.

Fazendo uma nova proposta de projeto, este problema pode ser resolvido através da utilização do padrão *Bridge*, já que se deseja separar abstração e implementação, de forma que a abstração utilize a implementação para realizar suas funções, e ambas possam variar livremente (poderão surgir novos tipos de arquivos de entrada e novos tipos de arquivos de saída).

Utilizando o *Bridge* pode-se pensar nas seguintes classes: Uma classe abstrata Interpretador, representando a abstração, e uma classe abstrata InterpretadorImpl, representando a implementação desta abstração. A classe Interpretador possui as seguintes subclasses: InterpretadorCW, InterpretadorEstCham, InterpretadorQualVoz, que têm como responsabilidade interpretar os arquivos de entrada, segundo seus tipos. A classe InterpretadorImpl possui as seguintes subclasses: InterpretadorTxt e InterpretadorCSV, que têm como responsabilidade formatar os dados de entrada, formatando-os para arquivos de saída TXT ou CSV. Foi projetada ainda a classe Conteudo que é um objeto de interface para comunicação de dados entre Interpretador e Interpretador Impl. A classe Conteudo possui os dados a serem formatados em cada um dos arquivos, os nomes das colunas, o texto de cabeçalho e o texto de rodapé que deverá

ser inserido nos arquivos. A classe Interpretador interpreta o arquivo de entrada segundo seu tipo, e preenche o objeto Conteudo. Este por sua vez é passado para a classe InterpretadorImpl de forma que se obtenha um arquivo final TXT ou CSV. A Figura 24 mostra o modelo de classes descrito acima.

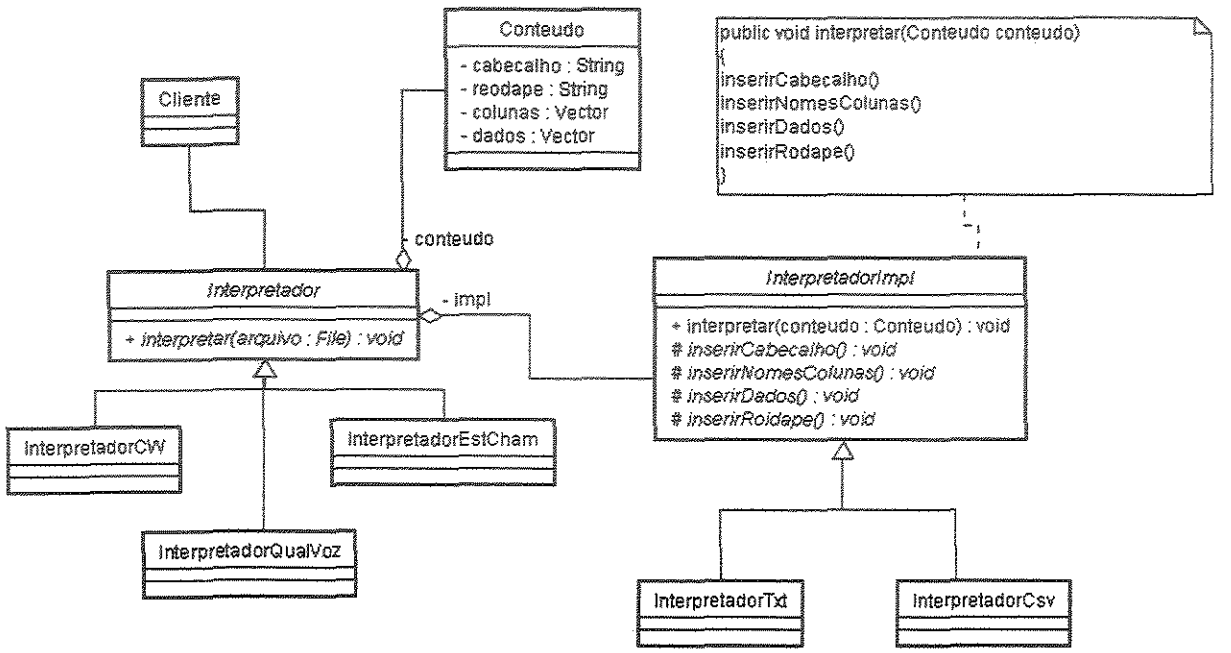


Figura 24. Novo modelo de classes para interpretação de dados coletados

É importante ressaltar que além do padrão *Bridge* este modelo utilizou também o *template method* na classe InterpretadorImpl, já que o método interpretar() define um algoritmo básico que suas subclasses deverão seguir, mas permite variação neste algoritmo já que deixa por conta das subclasses a implementação dos métodos que este algoritmo utiliza.

Com esta estrutura, as classes Interpretador utilizam as classes concretas de InterpretadorImpl para realizar suas tarefas. Desta forma, observa-se: a utilização de interfaces, já que as classes Interpretador conhecem apenas a interface de InterpretadorImpl, e as classes clientes conhecem apenas a interface das classes derivadas de Interpretador; e o favorecimento da composição de objetos ao invés da herança.

A criação de novos tipos de arquivos de entrada irá inserir apenas mais classes derivadas de Interpretador, que utilizarão os métodos de InterpretadorImpl para realizar suas funcionalidades,

e a criação de novos formatos de arquivos de saída irá inserir apenas mais classes derivadas de InterpretadorImpl, e estas classes serão utilizadas por Interpretador.

Pode-se ainda inserir entre Interpretador e InterpretadorImpl uma classe *abstract factory* para que as classe derivadas de Interpretador não precisem conhecer quais são as classes concretas de InterpretadorImpl que realizam as funcionalidades solicitadas, permitindo que futuramente, as classes de InterpretadorImpl possam variar, ou serem trocadas, sem a necessidade de as classes de Interpretador serem alteradas.

Além disso, é possível perceber que a inserção de um novo tipo de abstração ou implementação é simples, não havendo a necessidade de alteração em código já existente, e nem de criação de várias novas classes. Com isso, pode-se concluir que, a utilização do padrão diminuiu a quantidade de classes, que antes formaria uma árvore com fator de multiplicação não constante, podendo atingir valores elevados, e facilitou o entendimento do projeto, a manutenção corretiva e a inserção de novos requisitos.

Desta forma, analisando o modelo original, e o novo modelo proposto, é possível perceber que a utilização de padrões simplificou o projeto, deixando-o mais claro; facilitou a alteração de requisitos; e criou componentes de interpretação que podem ser reutilizados isoladamente.

Capítulo 5

Conclusão

Este trabalho discutiu a utilização de padrões de arquitetura e projeto no desenvolvimento de um sistema real. O fato de o sistema utilizado já ter sido implementado e estar em funcionamento foi interessante já que permitiu perceber as falhas de funcionamento por erros na fase de projeto, bem como as dificuldades enfrentadas quando houve a necessidade de se realizar alterações nos requisitos inicialmente projetados, e manutenção dos requisitos implementados.

Além disso, permitiu mostrar a simplicidade das soluções para problemas recorrentes durante o desenvolvimento de um sistema propostas pelos padrões, que utilizam apenas conceitos básicos de orientação a objeto, e englobam um senso comum de soluções e tentativas realizadas por desenvolvedores ao longo do tempo até que se chegasse à solução ideal [7].

Devido ao fator simplicidade, alguns dos padrões, como foi visto no capítulo 4, são utilizados por desenvolvedores que não têm conhecimento das técnicas, apenas porque após várias tentativas de soluções para problemas recorrentes em projetos, uma técnica específica foi adotada como ideal. Desta forma, percebe-se que os desenvolvedores são capazes de propor novos padrões conforme vão adquirindo experiência no desenvolvimento.

Também foi possível notar claramente a diferença entre os modelos reais e os novos modelos propostos para o sistema em termos de flexibilidade para alterações de requisitos, simplicidade de soluções, modularidade de camadas e utilização das melhores práticas de orientação a objetos.

É importante ressaltar que este trabalho discutiu a facilidade de flexibilização da estrutura de um projeto, mostrando novas propostas de modelos. Dando continuidade a este trabalho, seria interessante implementar estes novos modelos, verificando as dificuldades de entendimento enfrentadas pelos desenvolvedores, de forma a se realizar análises de desempenho e eficiência

[7] Para conhecer mais sobre a identificação de novos padrões, vide [VLISSIDES - 1998].

dos modelos. Desta forma, seria possível mostrar uma série de conseqüências (análise de custo e benefícios) da utilização dos padrões, que poderiam ser analisadas antes de aplicá-los em projetos.

Embora não tenha sido discutido neste trabalho, os padrões são altamente importantes no desenvolvimento de componentes e *frameworks* responsáveis respectivamente por uma maior reutilização de elementos independentes do sistema, e pelo fornecimento de arquiteturas pré-determinadas que devem ser estendidas pela equipe de desenvolvimento de forma a customizar uma aplicação. Ambas as técnicas têm como objetivo diminuir o tempo de desenvolvimento de projetos, fornecendo às empresas uma maior competitividade e um aumento na qualidade dos produtos finais, considerando-se principalmente que alguns elementos utilizados já foram amplamente testados.

O que torna as empresas diferenciadas e competitivas no mercado é sua capacidade de demonstrar conhecimento profundo em áreas específicas e saber aplicá-las de forma produtiva num mundo globalizado. O conhecimento específico no caso de grandes sistemas de software vai se adquirindo com tempo, e deve ser um conhecimento comum e disseminado por toda a equipe de software da empresa para que possa realmente ser efetivo e produtivo. Para cada nicho tecnológico existe uma forma sistemática de utilização de padrões, ou seja, embora os padrões sejam os mesmos, a forma como estes se relacionam será diferente, buscando resolver problemas recorrentes em cada tipo de domínio. O estabelecimento deste conjunto de padrões relacionados exige da equipe de desenvolvimento de sistemas experiência e amadurecimento na área.

Quanto mais uniformizados os mecanismos de desenvolvimento, melhores os resultados e mais fácil a manutenção do software. Os padrões de projeto são um grande avanço neste sentido.

O presente trabalho investigou a utilização de conjuntos de padrões associados no desenvolvimento de software para gerência de telecomunicações. Ainda há um longo caminho de amadurecimento pela frente até que se tenha uma comunidade letrada no uso de padrões de projeto e um conjunto bem definido de relacionamentos de padrões específicos para a área.

Bibliografia

[BARKER - 02] Barker, Jacquie, *Beginning Java Objects – From Concepts to Code.* Wrox Press Ltd., 2002, 665p.

[BUSCHMANN - 96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael, *Pattern-Oriented Software Architecture – A System of Patterns – Volume 1.* Wiley, 1996, 467p.

[COOPER - 98] Cooper, James W., *Design patterns*. Java Companion, 1998

[DEITEL - 02] Deitel, Harvey M.; Deitel, Paul J., *Java – How to program.* Prentice Hall, 2002, 1536p.

[ECKEL - 00] Eckel, Bruce, *Thinking in java*. Prentice Hall, 3ª Edição, 2000

[ECKEL - 03] Eckel, Bruce, *Thinking in patterns – Problem Solving Techniques using java*. 2003

[FOWLER - 00] Fowler, Martin; Scott, Kendall, *UML Distilled – A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, 2000, 185p.

[GAMMA - 95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John, *Design patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995, 395p.

[LARMAN - 98] Larman, Craig, *Applying UML and Patterns – An introduction to Object-Oriented Analysis and Design.* Prentice Hall PTR, 1998, 507p.

[LOOMIS - 95] Loomis, Mary E.S., *Object Databases – The essentials*. Addison-Wesley, 1995, 230p.

[PRESSMAN - 01] Pressman, Roger S., *Software Engineering – A practitioner's Approach.* McGraw-Hill, 2001, 860p.

[SHALLOWAY - 02] Shalloway, Alan; Trott, James R., *Design patterns Explained – A new Perspective on Object Oriented Design.* Addison-Wesley, 2002, 334p.

[VLISSIDES - 98] Vlissides, John, *Pattern Hatching – Design patterns Applied.* Addison-Wesley, 1998, 172p.

Artigos[8]

[BLUEPRINTS - 04] *Core J2EE Patterns Catalog*

http://java.sun.com/blueprints/corej2ee*patterns/Patterns*/DataAccess*Object*.html, Blueprints

[HAGGAR - 04] Haggar, Peter, *Double-checked locking and the Singleton pattern – A comprehensive look at this broken programming idiom*

http://www-106.ibm.com/developerworks/java/library/j-dcl.html

[JONG - 04] Jong, Wiebe de, *Implement a JDBC Connection Pool via the Object Pool Pattern.* Developer.com, 20/06/2004

http://www.developer.com/java/other/article.php/626291

[8] Todos os artigos com endereço *web* estavam disponíveis na data de publicação deste trabalho.