Engenharia de Software e Sistemas de Missão Crítica

Aluno: Thiago Pinheiro de Araújo
Orientador: Arndt von Staa

Introdução

A fim de avaliar a integridade de suas linhas, a Petrobrás lança mão de robôs instrumentados com sensores das mais variadas naturezas, tais como geométricos, magnéticos, de pressão, de temperatura e de ultra-som. Esses robôs, comumente chamados de PIGs, percorrem as tubulações coletando sinais que, posteriormente, são analisados a fim de localizar anomalias, bem como estimar a sua severidade.

O PIG possui um software embarcado, o qual controla a aquisição dos dados. Há os chamados PIGs umbilicais, os quais são conectados por um cabo de fibra ótica a uma estação e permitem aquisição em tempo real, e há os PIGs autônomos, os quais são liberados livremente para percorrer a linha pela própria diferença de pressão, sendo recolhidos posteriormente.

No caso do PIG umbilical, há um software que fica na estação de controle, responsável por gerar os mapas de visualização para os analistas. Este software também é o responsável pela análise *offline* dos dados, tanto no PIG umbilical quanto no PIG autônomo.

O software de controle e análise *offline* dos dados é extremamente complexo em termos de funcionalidades e de código. Seu desenvolvimento foi iniciado com uma grande preocupação com boas práticas de engenharia de software, porém a imensa avidez por resultados rápidos (decorrentes do desenvolvimento em paralelo do hardware do PIG) levou a um progressivo abandono dessa postura. Quando finalizada, a versão 1.0 do software de controle e análise *offline* possuía alguns problemas de concepção e engenharia mal resolvidos, tinha diversos pontos de difícil manutenção e deixava muito a desejar em termos de confiabilidade e robustez. Essas razões motivaram a sua reconstrução. O objetivo deste relatório é o de descrever as técnicas adotadas para tanto.

A Arquitetura do Software

O software divide-se em três partes: uma responsável unicamente pela comunicação com o PIG (no caso de PIGs umbilicais) uma responsável por gerenciar toda a parte de disponibilização, tratamento e armazenamento dos dados colhidos pela camada de comunicação e uma responsável pela visualização dos dados em gráficos que sejam passíveis de análise por parte dos operadores e analistas.

As partes responsáveis pela comunicação com o PIG e pela disponibilização e armazenamento de dados colhidos foram desenvolvidas com grande preocupação com a qualidade de engenharia de software. Porém, a visualização foi criada de forma bastante *ad hoc*, sendo a principal responsável pelos problemas existentes. Dessa forma, o foco o trabalho foi o de *reconstruir toda a camada de visualização e exibição dos dados*.

Conceitos Envolvidos

O PIG *colhe amostras* durante uma inspeção. Uma *amostra* é definida como o conjunto de valores lido em um grupo de *canais* em um dado ponto no espaço-tempo. Um *canal* é definido como um sensor. A *taxa de aquisição* de um PIG é definida como a quantidade de *amostras* que são colhidas a cada segundo. Isso é uma função do tipo de sensor envolvido – há sensores que requerem uma taxa de aquisição muito grande (da ordem de 512 Hz), ao passo que outros admitem taxas menores.

Cada *amostra* possui um *odômetro*, que é um referencial de posição para uma corrida e também um *timestamp*, que indica o momento em que a amostra foi colhida.

O Software

O software foi desenvolvido usando a biblioteca de programação QT, a qual permite a criação de sistemas multi-plataforma devido ao sistema GUI e bibliotecas que são convertidas para as do sistema operacional em tempo de compilação. No caso específico desse projeto, não havia o requisito de multi-plataforma – a plataforma alvo era o Linux. A escolha dessa biblioteca teve razões históricas (devido a projetos anteriores que tiveram amplo sucesso) e razões de competência da equipe. Porém, com pouco esforço, o uso de outras plataformas é possível.

O objetivo do software é apresentar as amostras colhidas para análise do duto. A visualização dos dados pode ser feita de seis formas distintas:

- Chapa: Visualização de todos os canais de um determinado segmento de amostras, onde a diferença entre os valores é transformada para cores de uma dada paleta de cores. Esta paleta de cores é calibrada de forma a permitir a identificação visual dos defeitos.
- Corte lateral: Visualização de todos os canais de uma determinada posição selecionada na chapa na forma de um gráfico de linhas.
- Corte longitudinal: Visualização de todas as posições de um determinado canal selecionado na chapa na forma de um gráfico de linhas.
- Régua horizontal: Marcações das posições visualizadas
- Régua vertical: Marcação dos canais na forma de horas.
- Informação de amostra: É um quadro com todas as informações sobre a amostra selecionada.

A visualização pode ser exibida em modo temporal ou espacial. O primeiro modo consiste em mostrar as amostras na ordem temporal em que foram coletadas. A segunda mostra as amostras em uma forma crescente dos valores de odômetro. O software é também capaz de trabalhar com discretização na visualização dos dados, ou seja, ter a funcionalidade de zoom.

O software é responsável tanto para aquisição dos dados em tempo real quanto para análise posterior dos mesmos, além de persisti-los em um arquivo.

Visão geral

Um dos grandes problemas que existia na versão 1.0 era a extrema complexidade da classe responsável pela exibição dos dados. Por ser a responsável pela geração de todas as visões possíveis, possuía uma quantidade muito grande de linhas de código (mais de 15 mil), além de uma pequena preocupação com engenharia de software. Essa classe foi extinta no novo software, e suas funcionalidades foram espalhadas por outras classes. Para tanto, foram utilizados os padrões de projeto *Decorator*, onde se tem uma classe abstrata Visão, e dela

estendem componentes e decoradores, e *Mediator*, que no software é representado pela classe Gerenciador, a qual todas as Visões conhecem.

Desenvolvimento da nova versão

Um ponto importante no desenvolvimento do software foi a utilização de técnicas de *Design by Contract*, ou seja, a utilização de pré e pós condições explicitamente especificadas e codificadas sobre a forma de assertivas executáveis.

A utilização de assertivas foi um fator importante para garantir esta robustez dos módulos. Após o planejamento da estrutura modular desta nova arquitetura foi simples definir as pré-condições e pós-condições de cada funcionalidade dos módulos. Então durante o desenvolvimento à medida que o código ia sendo escrito, foi gasto um esforço a mais criando assertivas que estabeleciam um contrato inicial e final para cada ação importante realizada pelo software.

O objetivo central da utilização do *Design by Contract* foi o de detectar estados inválidos, de forma a coletar o máximo possível de informações de depuração no momento em que as falhas ocorressem. Uma vez detectada uma falha, o sistema era abortado imediatamente – não era objetivo tentar qualquer tipo de recuperação no sistema.

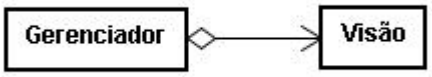
Das cerca de 30 mil linhas de código, 8% foram destinadas a assertivas. Como esperado, a importância deste esforço foi confirmada durante o desenvolvimento. À medida que se descobria uma falha, verificava-se a assertiva que notificou a mesma e descobria-se a correspondente falta rapidamente analisando o caminho que o software percorreu, juntamente com os valores de cada variável em cada etapa. A tabela abaixo é um exemplo da importância desta técnica:

Quantidade de faltas descobertas a partir de falhas detectadas por alguma assertiva, em ambiente de produção simulado durante os testes.	44
Quantidade de falhas que ocorreram, mas não foram detectadas por nenhuma assertiva (programa abortou, ou apresentou um comportamento errado sem qualquer tipo de aviso ou log), em ambiente de produção simulado durante os testes.	9
Tempo médio de correção das falhas descobertas a partir das assertivas (incluindo o tempo para descobrir a falta responsável pela falha observada)	30min
Tempo médio de correção das falhas que não foram descobertas a partir das assertivas (incluindo o tempo para descobrir a falta responsável pela falha observada)	3h

Como se pode ver, também foram descobertas falhas que ocorreram sem terem sido detectadas por alguma assertiva, e estas levaram em média seis vezes mais tempo para correção que as falhas descobertas a partir das assertivas.

O Gerenciador

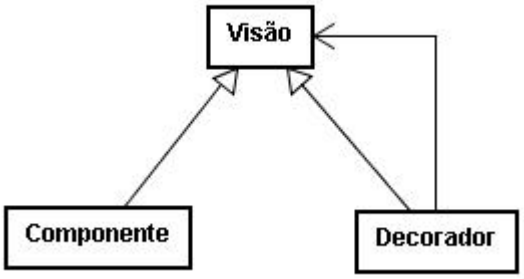
A classe Gerenciador é a responsável por gerenciar as posições e os dados exibidos em um dado momento no software. Esse componente abstrai o fato de como os dados são exibidos, somente se preocupando em manter o buffer de dados sincronizado de acordo com as entradas do usuário. Dessa forma, esse componente serve como fonte única dos dados a serem exibidos, o que garante a consistência entre os vários componentes visuais de exibição dos dados.



Os Decoradores e os Componentes

Toda a interface visual do software está escrita conforme o padrão de projeto *Decorator*. Conforme esse padrão prevê, existe a figura dos Componentes e dos Decoradores.

Os componentes são classes que produzem um trabalho independente, que poderia ser utilizado sem nenhuma alteração, caso necessário. Já os decoradores são classes que complementam o trabalho de um Componente. Um bom exemplo para entender como se dá o relacionamento entre decoradores e componentes é a árvore de natal. A árvore em si é o componente. As bolas, o pisca-pisca, a estrela na ponta, seriam decoradores. A figura a seguir ilustra a composição:



Os componentes possuem um *bitmap* interno, que é usado para pintura. Os decoradores acessam esse bitmap diretamente (através de método *get* específico), e realizam a sua pintura diretamente sobre ele.

Relacionamento entre as classes

A construção do relacionamento entre as classes é fortemente baseada no modelo de *signals* e *slots* existente no QT. Todos os eventos pertinentes a alterações de interface são atendidos pelos decoradores (ou seja, os *connects* dos eventos do QT são em *slots* nos decoradores). Os decoradores têm todas as configurações necessárias para realizar as suas tarefas.

Em caso de mudança de configuração, oriunda da interface, o decorador afetado muda a sua configuração e emite um sinal, que é recebido pelo gerenciador em um *slot*, o qual invocará o método de renderização de todas as visões (o que acarretará necessariamente a renderização do próprio decorador alterado).

Componentes cuja complexidade computacional do método de renderização seja alta fazem um *cache* de tela, para otimização. Esse *cache* é uma cópia pura da tela, em um *bitmap* (como na Chapa) ou mesmo uma cópia de dados tratados. Nesses casos, o elemento mantém em si um *fingerprint* dos dados a fim de identificar se os dados passados na renderização são os mesmos. Porém, isso é transparente a quem chama: é um processamento interno ao elemento.

Eventos pertinentes à alteração de dados exibidos são recebidos pelo gerenciador. Este repassa naturalmente aos decoradores que estiverem no topo da cadeia. Os eventos pertinentes à alteração dos dados são atendidos pelos seguintes métodos:

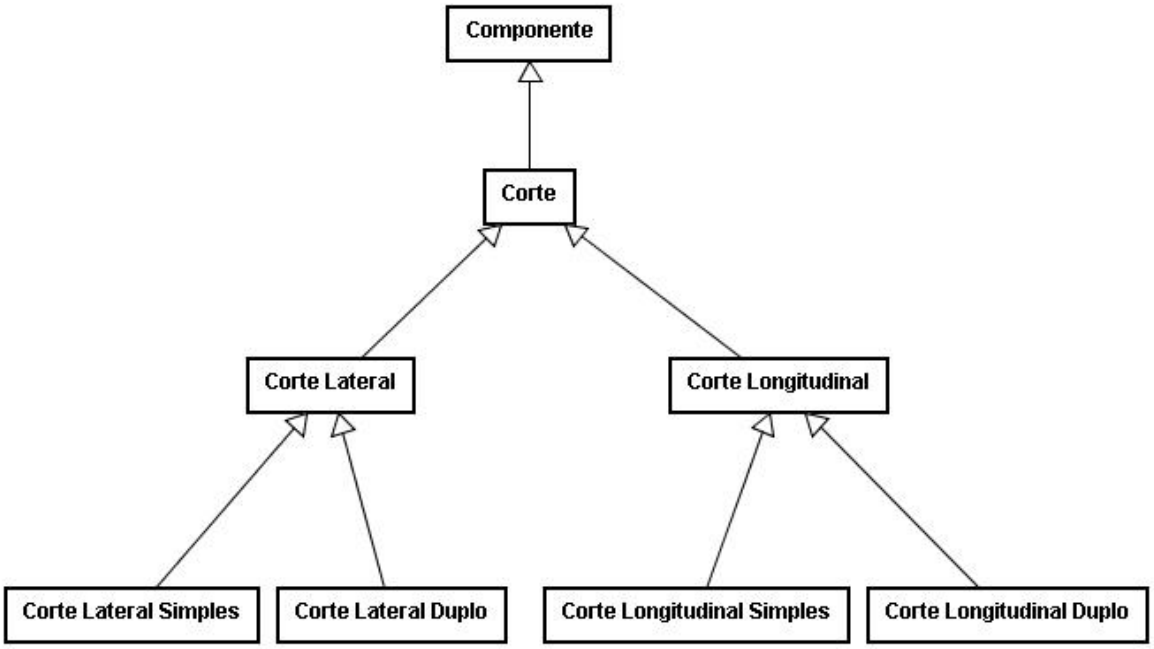
- **resize** : recalcula o tamanho do buffer interno do Gerenciador e repassa para todas as visões se ajustarem a tela.
- **redraw**: é invocado por todas as variações de eventos que incorram em necessidade de atualizar os dados exibidos em decorrência de alteração do intervalo visualizado em tela. Exemplos desses eventos são interações com a *scrollbar* de posicionamento, seleção manual de novo ponto central da janela, ou movimentação espontânea do PIG (no caso de PIGs umbilicais). Esse método recebe como parâmetro uma dica que descreve apenas os valores que mudaram desde o último estado.
- **refresh**: usado quando é preciso que o elemento se atualize em tela. Este método apenas renderiza em tela os dados armazenados em seu buffer interno.

A visão de Chapa é, de longe, a mais complexa e cara em termos computacionais. Por isso, a preocupação com sua otimização foi grande, de forma a viabilizar uma performance aceitável. Os resultados obtidos foram bastante satisfatórios, se comparados à performance do sistema anterior.

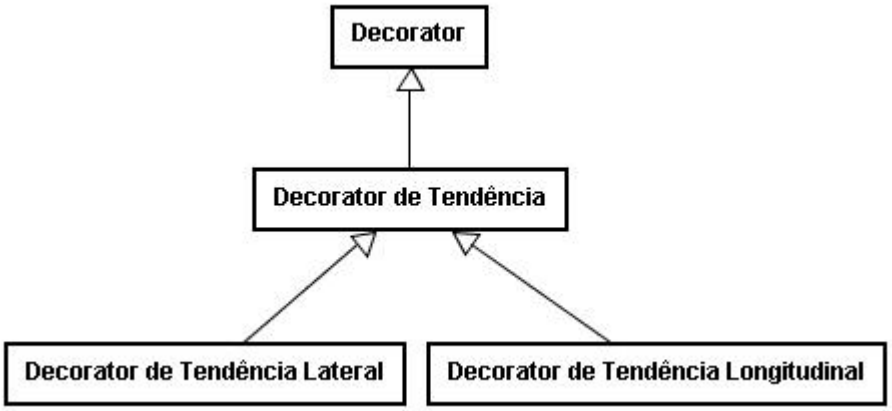
O gerenciador usa como buffer um tipo extensível. Esse tipo é estendido com novos parâmetros de acordo com a necessidade dos decoradores: dessa forma temos "parâmetros de conveniência", que poderão ser usados por um ou por outro decorador de acordo com a necessidade.

Exemplo de Decorador

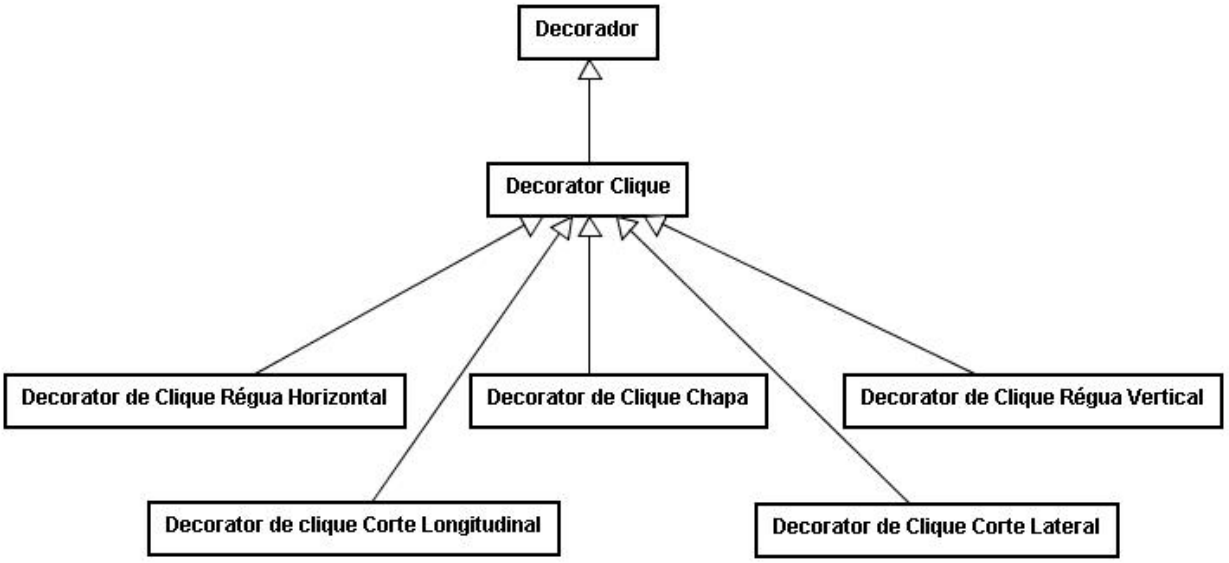
Um exemplo da utilização do padrão de projeto pode ser visualizado nas figuras abaixo. O primeiro diagrama mostra a relação de herança entre as classes que compõem os componentes de corte, e os diagramas seguintes mostram a hierarquia de classes de cada decorador utilizado na cadeia de decoradores dos cortes.



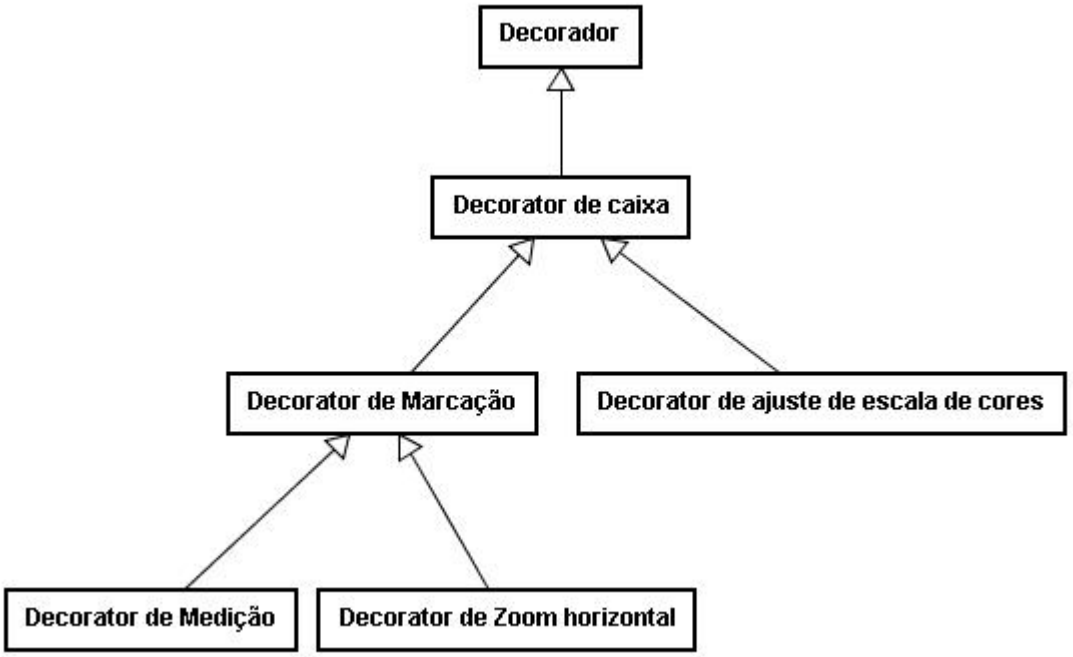
Decoradores de linha de tendência, responsáveis por traçar uma média móvel a partir dos dados exibidos em tela.



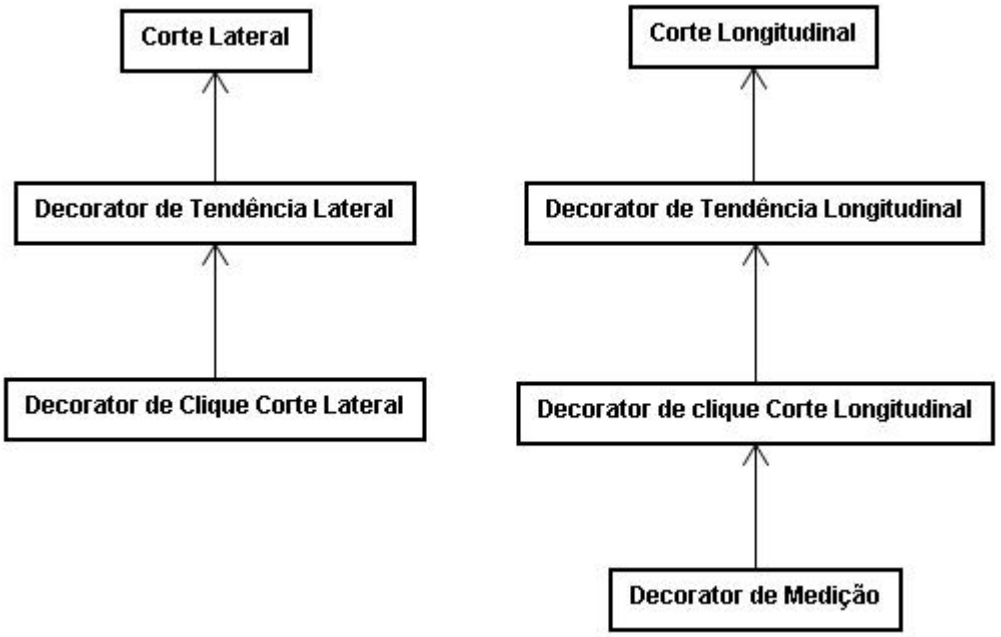
Decoradores de clique, responsáveis por traçar a posição atualmente selecionada. Um ponto interessante desta composição é a classe *Decorator Clique* armazenar a posição selecionada na forma estática, ou seja, é única para todas as instâncias dela ou estendidas dela. Com isso, conseguimos manter a consistência desta funcionalidade, não importando em qual decorador for realizada a ação de clique, todos serão capazes de traçar a mesma posição no componente que estiverem acoplados.



O decorador de caixa mostrado abaixo permitiu um grande reuso de código. Este apenas realiza um traçado no componente, disparado a partir de uma interação do usuário, porém possui um método abstrato chamado ao final da seleção. Este método é redefinido pelas suas subclasses, que realizam a ação correspondente ao decorador.



O diagrama abaixo representa a composição de duas cadeias de decoradores, que mostram como os cortes são montados. O último elemento das cadeias são os componentes, que podem existir por si só. Porém algumas das decorações mostradas acima são aplicadas:



O importante é perceber como é o funcionamento desta cadeia: quando é gerado um evento de renderização, o primeiro elemento recebe o sinal, e passa para o próximo, até chegar ao componente. O componente realiza sua renderização e retorna, sem ter conhecimento dos seus decoradores. Após o retorno, cada decorador gera sua renderização diretamente no bitmap gerado pela visão imediatamente anterior, na ordem inversa que foi chamado. É importante o fato do decorador só realizar sua decoração caso esteja ligado, caso contrário, retorna sem fazer alteração nenhuma.

Toda a visualização de dados foi reescrita desta forma, permitindo cada classe controlar uma única funcionalidade, sem interferir em outras.

Futuro do projeto

Existem evoluções já planejadas para o software. No entanto, apesar do grande esforço despendido em prol da reorganização estrutural dos componentes de visualização, ainda há muitos pontos que merecem revisão, principalmente devido ao fato que, paralelamente à reorganização, diversas novas funcionalidades foram inseridas.

Um exemplo de revisão a ser feita é no módulo Gerenciador. Atualmente, ele exerce além do seu papel principal (Mediador) outras funcionalidades, tais como tratamento de dados. O primeiro passo será criar um gerenciador de dados e novos decoradores de tratamento para este. Esta nova cadeia de decoradores será uma nova camada na arquitetura, sendo responsável pela obtenção e tratamento dos dados a serem exibidos em tela.

Conclusões

A nova arquitetura mostrou-se eficiente, e o correspondente código entregue não apresenta defeitos. Todos os problemas encontrados durante o processo de desenvolvimento foram rápida e completamente corrigidos devido à nova arquitetura do software.

Podemos citar como benefícios desta nova arquitetura:

- Melhor distribuição das responsabilidades;
- Flexibilidade na adição de novos tratadores (decoradores);
- Flexibilidade na adição de novos componentes;
- Viabilização de suporte à paralelização (ainda a ser discutido).

Os dois grandes módulos do software antigo possuíam 8 mil e 15 mil linhas respectivamente. Na nova arquitetura, estes dois módulos foram substituídos por aproximadamente 30 pequenos módulos. Apesar do número de módulos ter aumentado a depuração tornou-se mais fácil e, ao detectar algum defeito, sua remoção é rápida, tendo em vista que o contexto em que se encontra é, em geral, pequeno, facilitando sobremaneira localizar o código defeituoso.

Contribuíram também para a maior confiabilidade o uso de *Design by Contract*, o uso de padrões de projeto e a contínua preocupação com maximizar encapsulamento, e coesão e minimizar o acoplamento entre classes.

Em termos práticos, o produto final é um software fiel e robusto, capaz de fazer inspeções confiáveis. A evolução desta tecnologia de inspeção permite agora um aumento seguro das funcionalidades deste software.

Referências

1 – GAMMA, Erich. **Design Patterns: Elements of Reusable Object-Oriented Software.** 1.ed. Addison-Wesley Professional Computing Series, 1995

2 – VLISSIDES, John M. **Pattern Hatching : Design Patterns Applied** 1.ed. Addison-Wesley Professional Computing Series, 1998