USO DE INSTRUMENTAÇÃO E DISCIPLINA DE TRABALHO COMO MECANISMO DE AUMENTAR A CAPACIDADE DE DETECÇÃO DE FALHAS

Aluno: Thiago Pinheiro de Araújo
Orientador: Arndt von Staa

Introdução

Esta é uma continuação de estudos de caso que consistem em calcular quanto esforço é gasto com manutenção e depuração de código quando este é construído com uso de instrumentação e disciplina de trabalho. E a partir destes cálculos desejamos avaliar o quanto é vantajoso utilizar estas técnicas na construção de sistemas de missão crítica. Segundo Basili e Boehm [1] é vantajoso desenvolver-se software de elevada fidedignidade quando se leva em conta o custo total de desenvolvimento, uso e riscos do software.

O uso de instrumentação tem como finalidade principal minimizar o número de falhas existente em um software e diminuir o tempo gasto para corrigi-las [2]. Idealmente falhas não deveriam ocorrer, mas como quem escreve código são seres humanos e estes podem errar, é esperado encontrá-las no código. As causas são as mais variadas, dentre elas: desatenção, cansaço, complexidade do código, etc...

Um dos problemas mais relevantes na remoção de uma falha é a soma do tempo gasto procurando sua causa e o tempo necessário para corrigi-la. Em muitos casos é necessário reescrever grande parte do código. Para minimizar este tempo utilizamos as técnicas descritas neste documento.

O objeto estudado é um software de missão crítica que tem por objetivo a aquisição e visualização de sinais oriundos de sensores e a correspondente eletrônica. A aplicação prática deste software é utilizada para adquirir, tratar e exibir sinais de sensores de ultra-som montados sobre um robô de inspeção de dutos chamado PIG.

O objetivo deste estudo de caso é continuar o trabalho apresentado no ano anterior, que consiste no desenvolvimento do software mencionado e, ao concluir etapas, medir o quanto robusto e confiável é o sistema e avaliar quanto esforço foi gasto em manutenção e depuração de código.

Técnicas utilizadas

Sistemas de missão crítica são sistemas de software que requerem um elevado nível de qualidade e têm como principais requisitos não-funcionais a necessidade de uma elevada confiabilidade e robustez. Entende-se por confiável o software que, sempre que solicitado, produz resultados corretos, precisos e exatos. O requisito robustez diz respeito ao tratamento eficiente de exceções, podendo recuperar o sistema ou cancelar a operação, informando o local e o estado ao observar a falha. Outro requisito importante é a fidelidade das informações exibidas, que diz respeito à veracidade das informações visualizadas. Para alcançar estes objetivos propomos a utilização das técnicas descritas a seguir.

A primeira delas é a utilização de padrões de projeto [3], que são soluções de eficiência já comprovadas e amplamente utilizadas no desenvolvimento de software. Estas soluções são desenvolvidas por especialistas e tornam-se padrões por poderem ser reutilizadas em diversos projetos e por terem sua eficácia comprovada.

Sua utilização ajuda no processo de definição da interface de cada componente do software e conseqüentemente na divisão de responsabilidades entre eles. Um dos benefícios

obtido a partir da utilização de padrões de projeto é a facilidade na manutenção e evolução tendo em vista que para adicionar um novo componente, é necessário apenas conectá-lo no modelo existente, com um custo muito baixo em termos de codificação.

Outro benefício trazido é a facilidade da correção das falhas a partir do momento de sua detecção. Devido à arquitetura bem cuidada, o esforço gasto para a correção da maioria das falhas resume-se à alteração de poucos módulos.

Outra técnica é a utilização de *Design by Contract*, em que a idéia central é a criação de contratos entre os módulos do software [3, 4], ou seja, definir com precisão as condições que devem ser atendidas pelos clientes (código que chama um método) e servidores (os métodos chamados). Para verificar a corretude da execução, estas condições devem ser testadas antes (pré-condições) e depois (pós-condições) da execução de uma rotina. Um exemplo disso é o código abaixo que faz a média de n números entre 0 e 100. Para o método implementado neste código foram estabelecidas as seguintes pré-condições:

- O método deve receber um vetor não nulo.
- O número de valores contido no vetor deve ser maior do que zero.
- Nenhum dos valores deve ser maior que 100.

E também a pós condição:

- O valor de retorno deve estar entre 0 e 100.

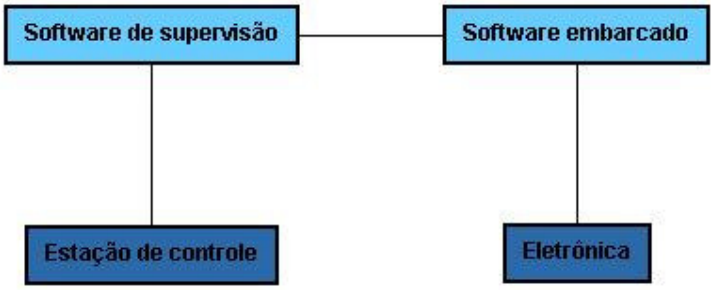
```
unsigned int media(
     unsigned int * vtValores,
     unsigned int numValores)
{
     ERR_ASSERT(vtValores != NULL);
     ERR_ASSERT(numValores > 0);

     unsigned int ulMedia = 0;
     for( unsigned int i=0 ; i<numValores ; i++ )
     {
          ERR_ASSERT(vtValores[ i ] <= 100);
          ulMedia += vtValores[ i ];
     }

     ulMedia /= numValores;

     ERR_ASSERT(ulMedia <= 100);

     return ulMedia;
}
```

Como pode ser visto no exemplo acima, o mecanismo para utilizar esta técnica é inserir assertivas executáveis (a macro ERR_ASSERT) e métodos de validação da integridade das estruturas [2]. Para isto são utilizados verificadores estruturais que consistem em métodos especificamente projetados e implementados para verificar se as estruturas satisfazem as suas assertivas estruturais. Uma condição não satisfeita cancela a execução do software, indicando a condição não satisfeita, os parâmetros utilizados para esta verificação quando possível e o local (linha de código e nome do módulo) em que foi realizada a verificação. Chamamos isso de programação defensiva [5].

E por fim a técnica de teste automatizado de módulos [2]. O objetivo de utilizar esta técnica é garantir a corretude dos componentes criados. Esta técnica se faz necessária tendo em vista o volume muito grande de funções que cada componente proporciona. Como os componentes são criados de forma incremental, um teste manual é muito suscetível a falhas já que deve ser realizado a cada implementação de uma nova funcionalidade.

Arquitetura do software

A arquitetura de utilização do sistema prevê a existência de uma estação de controle remoto do equipamento de inspeção, a qual deve conter um software supervisor (que é o foco deste estudo de caso). Essa estação se comunica com a eletrônica que possui um software embarcado (que não é foco desse estudo de caso). Essa eletrônica deve ser montada em uma mecânica adequada ao objeto da inspeção. Um diagrama da relação entre estes elementos pode ser visto abaixo:



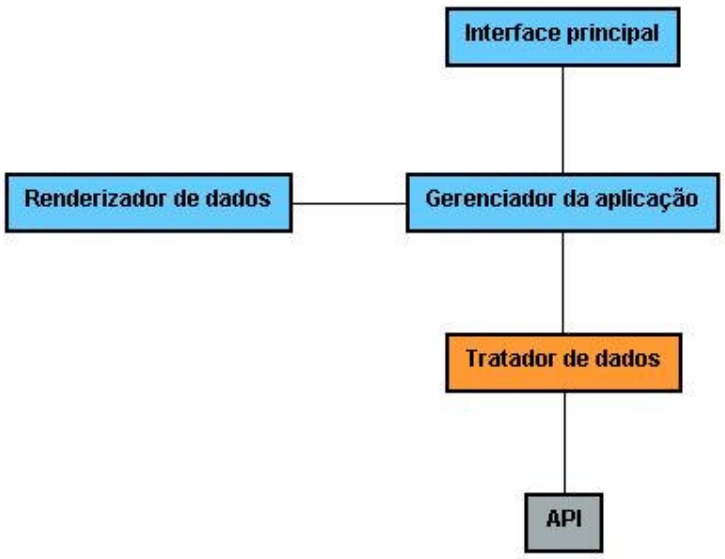
Relação entre o sistema de supervisão e o instrumento de inspeção

Uma eletrônica de inspeção é composta por diferentes sensores. Existem sensores que são utilizados para aquisição de dados da área inspecionada e sensores utilizados para aquisição de sinais adicionais. Ambos os tipos de sinais são armazenados em um conjunto de informações chamado de *amostra*, que consiste no armazenamento do valor de todos os sensores em um determinado instante.

Um exemplo disso é uma eletrônica que dispõe de alguns sensores de ultra-som cujas medições serão utilizadas para avaliar a integridade do objeto inspecionado. Porém existem outros sensores como odômetro, cuja medição é anexada à amostra como informação adicional e serve para determinar a posição espacial no instante em que a amostra foi colhida. Estes sinais adicionais são utilizados posteriormente no tratamento dos dados para viabilizar a visualização em tela. Internamente no software cada sensor é chamado de *canal*.

O software supervisor é utilizado para a aquisição e a análise dos dados colhidos. Este software é o foco do nosso estudo. Desta forma é estabelecido um contrato com o software embarcado onde este deve seguir um protocolo na transmissão de dados para estes serem corretamente interpretados pelo software supervisor.

A arquitetura deste software é dividida em três camadas. Cada uma dessas camadas pode ser visualizada na figura abaixo que mostra os componentes do software:



Arquitetura dos componentes do software

	Leitura dos dados brutos
	Tratamento dos dados
	Visualização dos dados

O componente responsável por uma aquisição é a API. Durante uma aquisição, todos os dados recebidos além de serem exibidos em tela são armazenados em um arquivo. Ao serem lidos, estes dados são colocados em *buffers* e ficam disponíveis para requisições.

O componente de tratamento de dados serve para organizar os sinais lidos ou melhorar a sua qualidade. Um exemplo de organização seria o tratamento realizado para visualizar os sinais na forma espacial: como são armazenados para indexação temporal, é necessário reorganizar as amostras, e em alguns casos fundi-las para permitir uma correta visualização espacial. E um exemplo de melhoria no sinal seria fundi-las de acordo com um determinado critério, tal como a média das amostras situadas na mesma posição espacial.

O componente de visualização é quem faz todas as requisições de dados para uma posterior renderização em tela. Este permite que o usuário analise o mesmo sinal de diferentes formas.

Metodologia de trabalho

O código dos componentes de tratamento e visualização de dados foi escrito utilizando as técnicas de *Design by Contract* e Padrões de Projeto. Para cada classe ou conjunto de classes foram estudados seus contratos, sua implementação e como se relacionariam com as demais classes no software.

As classes de tratamento de dados foram submetidas a testes automatizados. Como este componente possuía algumas dependências com a *API* e o *Gerenciador da aplicação*, foram criadas classes de enchimento, porém com interfaces idênticas as reais.

O diagrama de classes abaixo exemplifica um teste de um tratador de dados. *Tratador* é a classe submetida ao teste, e *Teste do tratador* é a uma subclasse de *Teste genérico* criada para definir como o teste deve ser feito.

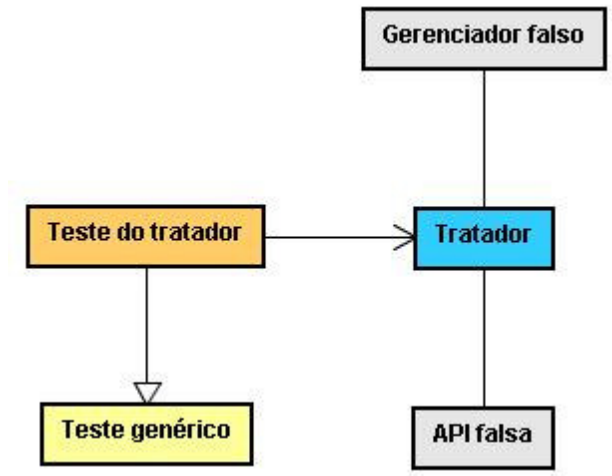


Diagrama de demonstração do uso da técnica de teste automatizado

Medições relativos à técnica de *Design by Contract*

Para avaliar a utilização da técnica de *Design by Contract* foram realizadas medições sobre a porcentagem do código destinada a assertivas e o número de falhas detectadas por elas. A partir dos arquivos fonte do software foram levantados os seguintes dados:

Número total de linhas nos arquivos fonte	36.686
Número de linhas em branco ou de comentários	1.861
Número de linhas de código puro	34.825
Número de linhas de código puro destinadas a assertivas	2.438
Porcentagem de linhas destinadas a assertivas	7,0 %

Para calcular o número de linhas de código foi contabilizado somente o número de linhas dos arquivos de implementação (os arquivos de definição fonte foram desconsiderados). O número de linhas em branco ou destinadas a comentários também foi retirado, obtendo-se apenas a porção escrita suscetível a falhas de execução. A partir dos valores obtidos foi possível calcular a porcentagem do código destinado a assertivas.

O estudo feito sobre o número de falhas detectadas a partir de assertivas consistiu em registrar cada falha informando se ela foi detectada a partir de uma assertiva ou não, classificando-a, ainda, como interna ou externa. Consideramos falhas internas quando descobertas durante o processo de teste ou codificação, e externa quando descoberta a partir da utilização do software por usuários finais.

De todas as falhas registradas, as que não poderiam ser detectadas a partir de assertivas foram desconsideradas. Consideramos para estudo apenas as que poderiam ter sido protegidas por assertivas, e as demais falhas que somente poderiam ser detectadas a partir da observação visual não foram consideradas nesta análise. A tabela abaixo mostra o número de falhas em cada categoria:

	Falhas internas	**Falhas externas**	**Total de falhas**
Descoberta a partir de assertivas	20	21	41
Descoberta a partir da utilização	14	3	17
Total de falhas	34	24	**58**

A partir dos dados acima se pode ver que aproximadamente 71% das falhas encontradas foram detectadas a partir de assertivas. As informações contidas nas assertivas foram úteis para depuração e correção da falha. As falhas que não foram detectadas a partir de assertivas, após serem corrigidas passaram a ser protegidas por elas.

Para avaliar o benefício obtido a partir desta técnica foram feitas medições sobre as falhas encontradas, avaliando o tempo gasto para descobrir a sua causa e o tempo necessário para corrigi-la. As tabelas abaixo exibem a média do tempo gasto e o desvio padrão respectivamente de cada categoria descrita:

Tempo médio:

	Tempo médio para descobrir a origem da falha	**Tempo médio para corrigir a falha**	**Tempo médio total**
Descoberta a partir de assertivas	9 min	24 min	34 min
Descoberta a partir da utilização	17 min	37 min	54 min

Desvio padrão:

	Desvio padrão do tempo médio para descobrir a origem da falha	**Desvio padrão do tempo médio para corrigir a falha**
Descoberta a partir de assertivas	3.16	8.97
Descoberta a partir da utilização	3.91	10.93

Como se pode ver, o tempo necessário para detectar a origem da falha chegou a ser em média quase a metade do tempo para detecção de falhas não descobertas a partir de assertivas. Já o tempo médio para corrigir uma falha foi em média dois terços do tempo utilizado para corrigir uma falha que não foi descoberta a partir de assertiva.

Os desvios padrão calculados para o tempo médio de detecção de cada falha foram pequenos como esperado. Porém os desvios padrão calculados para o tempo médio de correção de cada falha foram relativamente grandes.

Estes resultados demonstram que a utilização de assertivas reduz o tempo gasto na depuração para descoberta da falha aproximando-se de uma constante, mas a única coisa que podemos afirmar em relação ao tempo para correção da falha é que em média é menor que o tempo gasto na correção de outra não descoberta a partir de assertiva.

Futuro do projeto

Nos próximos meses do projeto espera-se aumentar a porcentagem de código escrita para proteção da sua execução e também criar teste automatizado para as principais *classes* do software. E a partir da técnica de teste automatizado fazer um estudo sobre o número de falhas descobertas.

Conclusões

Como esperado, o esforço gasto na inserção de assertivas executáveis foi compensado pelo tempo não desperdiçado na depuração de falhas. Outro benefício trazido pela utilização desta técnica foi o estudo realizado sobre o módulo em questão para a criação de suas

assertivas, que acabou mostrando possíveis falhas na implementação antes mesmo do código ser escrito.

O software desenvolvido é eficaz, pois atende aos requisitos: confiabilidade e robustez. O sucesso obtido deve-se a utilização instrumentação e disciplina de trabalho como mecanismos de detecção de falhas. O requisito de facilidade de evolução também foi atendido devido à distribuição de responsabilidades e à utilização de padrões de projetos na sua arquitetura. A adição de um novo componente limita-se à escrita dos módulos que o compõe, seguindo de um esforço desprezível para conectá-los ao software.

Concluindo, o esforço adicional realizado em um desenvolvimento utilizando mecanismos para detecção de falhas é muito pequeno quando comparado com os seus benefícios. Isto pode ser comprovado por este estudo de caso: obtivemos um sistema de grande porte, cuja maior parte das falhas foi descoberta a partir de assertivas. Como esperado a maioria das falhas puderam ser resolvidas alterando uma fração muito pequena de código, reduzindo assim o trabalho excedente, ou seja, o esforço gasto além do realizado na produção do software.

Referências

1. BASILI, V.R.; BOEHM, B.W.; “Software Defect Reduction Top 10 List”; IEEE Computer 34(1); Los Alamitos, CA: IEEE Computer Society; 2001; pags 135-137
2. STAA, A.v.; Programação Modular; Rio de Janeiro: Campus; 2000
3. MEYER, B.; Eiffel: The Language. Prentice Hall; 1992
4. MEYER, B.; “Applying Design by Contract”; IEEE Computer 25(10); Los Alamitos, CA: IEEE Computer Society; 1992; pags 40-51
5. Defensive programming, http://en.wikipedia.org/wiki/Defensive_programming