6
Conclusões e Trabalhos Futuros

6.1
Conclusões

Neste trabalho foi desenvolvido um sistema multi-agentes para monitoramento e aquisição em tempo real, composto por um software embarcado, modelado como um sistema multi-agentes e desenvolvido em C, e um software supervisor, desenvolvido em C++ utilizando as bibliotecas Qt, log4cxx e SQLite. Em ambos os software os requisitos de tolerância a falhas eram muito altos e por este motivo, foram utilizadas algumas tecnologias e conceitos os quais acreditava-se que, uma vez combinados, seriam capazes de promover a qualidade final necessária.

Como a principal conclusão deste trabalho, pode-se citar o sucesso no uso de tantas tecnologias (em conjunto), o que não foi encontrado em outros trabalhos relacionados. O fato de utilizar várias tecnologias poderia ter dificultado o desenvolvimento, mas ao invés disso, foi a chave para o sucesso do projeto. Sistemas dessa natureza (sistemas de monitoramento e aquisição em tempo real com severos requisitos de tolerância a falhas) possuem um alto grau de complexidade, o que foi minimizado pela abordagem utilizada neste trabalho. Outros sistemas da mesma natureza, que estão sendo desenvolvidos pela mesma equipe e que utilizam esta mesma abordagem, mostraram indícios de que irão obter o mesmo sucesso, o que nos traz fortes indícios de que esta abordagem de fato auxilia no desenvolvimento de sistemas dessa natureza.

Os resultados obtidos no desenvolvimento foram extremamente positivos, e os requisitos de qualidade previamente estabelecidos foram completamente atingidos. Embora seja impossível afirmar, são grandes as evidências de que o esforço extra durante o desenvolvimento gerado pela redação das pré e pós-condições e pela implementação das assertivas executáveis (uso de *Design by Contract*) foi o grande responsável pelo reduzido tempo gasto com testes do sistema (duas semanas em ambiente de produção simulado) e com a homologação (dois dias em ambiente real de produção), e também pelo reduzido

tempo médio de depuração das falhas encontradas.

Este fato é corroborado pelas estatísticas coletadas durante o desenvolvimento do sistema (Tabela 6.1). Essas estatísticas foram extraídas de dois sistemas de acompanhamento de projetos utilizados pela equipe durante o desenvolvimento. O primeiro é o *myHours*[55], que é um sistema usado para preencher *timesheets*. A partir dos *timesheets* preenchidos foi possível extrair relatórios gerais, como por exemplo: quanto tempo foi gasto com codificação de funcionalidades novas; quanto tempo foi gasto com codificação para reparação de faltas; quanto tempo foi gasto com testes; etc. O outro sistema de acompanhamento utilizado é o *Jira*[56] que é um *issue tracker* (sistema de gerenciamento de tarefas - em inglês, *issues*), onde cada tarefa realizada durante o desenvolvimento é cadastrada e gerenciada. Cada tarefa pode ser classificada em quatro diferentes categorias:

Bug: trata-se de um comportamento em desacordo com a especificação prévia do software.

New Feature: trata-se de uma nova funcionalidade que deve ser implementada no sistema.

Improvement: trata-se de uma melhoria que deve ser feita em uma funcionalidade pré-existente no sistema.

Task: trata-se de uma tarefa que não se encaixa em nenhuma das classificações anteriormente definidas. São típicos exemplos de *tasks*: liberar uma versão de software para uso; entrar em contato com o cliente para solucionar alguma dúvida; etc.

O uso combinado das duas ferramentas tornou possível extrair relatórios mais específicos, como por exemplo: quando tempo uma falha específica demorou para ser corrigida; quanto tempo determinada funcionalidade demorou a mais do que o planejado para ser implementada; etc.

Como pode ser visto na Tabela 6.1, o número de falhas identificadas (31 no total) pode ser considerado pequeno levando-se em conta o tamanho do sistema, que possui cerca de 50.000 linhas de código (em [57, 58] os autores estimaram uma média de 0.3 a 0.5 falhas para cada 100 linhas de código no, caso do sistema desenvolvido neste trabalho a razão ficou em torno de 0.062 falhas para cada 100 linhas de código). Uma possível razão para tanto é o fato de que a obrigação na redação de pré e pós-condições força o programador a ser mais criterioso no seu trabalho, fazendo com que o código gerado seja mais propenso a estar correto por construção. Além disso, as ferramentas utilizadas durante o desenvolvimento,

Quantidade de faltas descobertas a partir de falhas em alguma assertiva, em ambiente de produção simulado durante os testes.	22
Quantidade de falhas que ocorreram, mas não foram detectados por nenhuma assertiva (programa abortou, ou apresentou um comportamento errado sem qualquer tipo de aviso/*log*), em ambiente de produção simulado durante os testes.	5
Tempo médio de correção das falhas descobertas a partir das assertivas (incluindo o tempo para descobrir a falta responsável pela falha observada)	1h
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)	6h
Quantidade de falhas observadas através de falhas em assertivas em ambiente de homologação (ambiente de produção controlado)	2
Quantidade de falhas observadas em ambiente de homologação (ambiente de produção controlado), sem a ocorrência de falhas em assertivas	0
Quantidade de falhas observadas em ambiente de produção (primeiros 2 meses e meio de utilização da primeira versão liberada oficialmente), consideradas leves (i.e., sem perda de nenhum trabalho executado e cuja recuperação se limitou a executar o sistema novamente)	2
Quantidade de falhas observadas em ambiente de produção (primeiros 2 meses e meio de utilização da primeira versão liberada oficialmente), consideradas graves (i.e., com perda de trabalho executado ou cuja recuperação não se limitou a simplesmente executar o sistema novamente)	0
Quantidade de falhas observadas em ambiente de produção (3 meses posteriores aos meses iniciais de utilização), consideradas leves (i.e., sem perda de nenhum trabalho executado e cuja recuperação se limitou a executar o sistema novamente)	0
Quantidade de falhas observadas em ambiente de produção (3 meses posteriores aos meses iniciais de utilização) consideradas graves (i.e., com perda de trabalho executado ou cuja recuperação não se limitou a simplesmente executar o sistema novamente)	0

Tabela 6.1: Estatísticas do Sistema

tais como o valgrind[1] e o log4cxx também foram essenciais para os resultados. Um fato importante é que no caso das falhas que não foram detectados por nenhuma assertiva, descobriu-se posteriormente que a falha poderia ter sido evitada, ou pelo menos a sua causa real teria sido encontrada mais facilmente, se existissem algumas assertivas em ponto específicos do código. Essas assertivas, que depois foram incorporadas ao código, em geral não foram criadas porque: o programador considerava que por construção não havia necessidade de fazer o teste (por exemplo: considerar que por construção um determinado parâmetro ou variável nunca iria ser NULL); porque houve alteração na assinatura ou no contrato de um método, e alguma assertiva deixou de ser criada ou alterada em função desta alteração (por exemplo: um método que nunca retornava NULL foi alterado para retornar NULL em algumas situações); e em alguns casos por descuido ou esquecimento do programador.

Foram três meses de desenvolvimento intensivo, dos quais duas semanas dedicadas inteiramente a testes e homologações. Participaram do projeto dois programadores experientes e um engenheiro de software sênior. O código fonte do sistema possui cerca de 120 classes, totalizando mais de 240 arquivos entre headers (.h) e módulos de implementação (.cpp e .c). Das cerca de 50 mil linhas de código geradas, aproximadamente 16% são dedicadas exclusivamente à auto-verificação e recuperação do sistema, estando divididas nas seguintes categorias:

- 3.75% destinadas à identificação de falhas.
- 5,25% destinadas à "arrumação de casa", ou seja, a minimizar os danos ocasionados por uma falha.
- 7% destinadas à recuperação de falhas, ou seja, a levar o sistema a um estado consistente a partir de um estado inconsistente identificado.

É importante ressaltar que, durante os cerca de cinco meses iniciais após a liberação da primeira versão, o software foi utilizado diariamente (mesmo aos fins de semana) por cerca de 8h/dia, o que caracteriza mais de 1200 horas de uso intensivo.

Um ponto interessante a ser discutido foi que, durante boa parte do desenvolvimento, não se tinha acesso ao protótipo do hardware do sistema, pois este ainda estava em desenvolvimento. Para suprir essa necessidade, foi desenvolvido um simulador cujo objetivo era o de fornecer dados controlados para os testes do sistema supervisor. Essa iniciativa teve como base teórica a idéia dos *Mock Objects*, com a diferença de que, neste caso, o elemento falso gerado

[1] Valgrind é um software, disponível apenas para sistemas *unix*, que permite monitorar o uso de memória de um outro programa. Com ele é possível encontrar rapidamente problemas de uso de memória (como vazamento e acesso inválido) em um programa.

tem maior semelhança com um *Mock Component* ou um *Mock Agent*, dada a sua natureza independente (do sistema), pró-ativa e altamente configurável. Essa estratégia se mostrou tão bem-sucedida que, quando o sistema foi testado com o protótipo real, ocorreram somente alguns erros - todos imediatamente capturados pelas assertivas executáveis - corrigidos rapidamente, de forma que em menos de quatro horas dispunha-se de um sistema operacionalmente completo, desde o hardware do equipamento até o software supervisor.

Desde que foi liberada a versão inicial para produção, já houve diversas intervenções para a criação de novas funcionalidades. As assertivas executáveis continuam a ajudar, haja vista que minimizam o impacto de faltas introduzidas devido à alteração de interfaces e/ou de incompatibilidade de comportamento entre os componentes do sistema. Ao longo dos cinco meses transcorridos desde o primeiro *release*, o software já ganhou cerca de 10 mil linhas de código. Dessas, cerca de 8% são destinadas à identificação, tratamento e recuperação de falhas, número bastante inferior aos 16% medidos anteriormente, o que pode ser justificado devido ao fato de que muitas partes do código gerado fazem uso de código existente e compartilham a parte de tratamento e recuperação de falhas. Durante essa nova parte do desenvolvimento, foram muitas as vezes em que assertivas antigas falharam devido à inserção de código novo, o que auxiliou na identificação e solução de problemas.

Como não existe uma "contra-prova" disponível, ou seja, um sistema com características semelhantes o suficiente que tenha sido desenvolvido sem o uso das tecnologias e conceitos aplicados neste trabalho, é impossível afirmar que o sucesso atingido se deve à combinação dessas tecnologias e conceitos. Porém, as evidências encontradas apontam fortemente para isso.

6.2 Trabalho Futuros

Devido ao fato desse trabalho estar em andamento, alguns trabalhos futuros são esperados, como:

– Dar continuidade ao desenvolvimento do sistema, pois existe uma grande demanda por novas funcionalidades como: novos tipos de gráficos (e novas funcionalidades específicas para cada um deles); incluir suporte à análise utilizando algumas normas para análise de integridade de dutos; novas maneiras de importar e exportar dados do sistema (permitindo, por exemplo: analisar dados de uma corrida feita em um outro sistema - e provavelmente por outro tipo de ferramenta de inspeção);

etc. O desenvolvimento continuará fazendo uso das tecnologias aqui apresentadas, pois estas mostraram-se eficazes no desenvolvimento de sistemas desta natureza (sistemas de monitoramento e aquisição em tempo real com severos requisitos de tolerância a falhas), visto o sucesso que foi obtido no desenvolvimento da primeira versão.

– O software embarcado foi desenvolvido pensando-se na possibilidade de facilitar o seu reuso (com poucas alterações) em sistemas similares. Será desenvolvido um novo sistema de inspeção de dutos. Onde o software embarcado (que dentro do possível será o mesmo software usado no sistema desenvolvido neste trabalho) executará em um *scanner*[2] que será usado manualmente para inspecionar pontualmente o duto e o software supervisor executará em um PDA (ou celular) que apresentará em tempo real as leituras do *scanner*. O desenvolvimento deste sistema será feito (dentro do possível, uma vez que sua plataforma é mais limitada) utilizando as tecnologias apresentadas neste trabalho.

– Fazer um estudo comparativo entre o desenvolvimento do sistema descrito neste trabalho (cujo software supervisor é um software desktop, desenvolvido em C++) e o novo sistema (cujo software supervisor será um software para portáteis desenvolvido em J2ME[3]).

– Evoluir o conceito do *AssertionHandler* criando um módulo funcional e bem estruturado que possa facilmente ser utilizado em projetos existentes. Como estudo de caso, todo o tratamento de assertivas existente no sistema poderá ser substituído pelo novo módulo do *AssertionHandler*.

[2] O *scanner* consiste em uma versão menor (pouco maior que um ferro de passar roupa) e mais precisa da ferramenta de inspeção.

[3] J2ME - Java Microedition[59] é uma plataforma de desenvolvimento baseada na tecnologia Java direcionada para equipamentos com recursos limitados (como memória e processamento). Um exemplo típico de utilização do J2ME são os programas criados para executar nos equipamentos portáteis, tais como celulares e PDAs (palms e seus concorrentes).