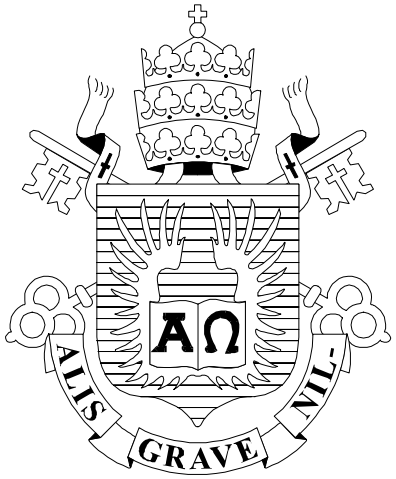

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 13/06

Engenharia de Software Fidedigno

Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, Nº 13/06 ISSN 0103-9741
Editor: Carlos J. P. Lucena Abril, 2006

Engenharia de Software Fidedigno

Arndt von Staa
arndt@inf.puc-rio.br

Abstract: In this article we present an abridged description of what is called Engineering of Dependable Software, aiming at establishing the objective of a research direction. The major characteristics of dependable software are described. The concepts involved when dealing with dependable systems are presented. Means to achieve dependable systems are briefly described. Finally, ongoing work is briefly described.

Keywords: Software Engineering, Dependability, Fault tolerance, Recoverability, Diagnosticability, Testing.

Resumo: Neste artigo é descrito sumariamente o que se entende por Engenharia de Software Fidedigno caracterizando uma diretriz de pesquisa que tem por objetivo contribuir para facilitar o desenvolvimento de software efetivamente fidedigno. São apresentadas as características mais relevantes de um tal software. São descritos os conceitos envolvidos ao desenvolver software que seja fidedigno. São descritas algumas maneiras de proceder para alcançar o objetivo. Finalmente são apresentados os trabalhos em andamento.

Palavras chave: Engenharia de Software, Fidedignidade, Tolerância a Falhas, Recuperabilidade, Diagnosticabilidade, Depurabilidade, Teste.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br

1 Motivação

Em virtude da crescente participação de software na sociedade, torna-se cada vez mais necessário assegurar que seja ***fidedigno***. Um software é dito ***fidedigno*** (inglês: *dependable*) quando se pode justificavelmente depender dele assumindo riscos de danos compatíveis com o serviço prestado pelo software [Avizienis et alii, 2004a]. Em [Avizienis et alii, 2004b] são estabelecidas as propriedades que sistemas fidedignos devem possuir, baseando-se em [Scott et alii, 1987]. Entretanto, esta lista de propriedades é fortemente focada em sistemas embarcados e não evidencia adequadamente algumas das propriedades relevantes no que toca a evolução do software. Temos então uma lista ligeiramente alterada:

Disponibilidade: estar pronto para prestar serviço correto sempre que se necessite do software.

Confiabilidade: habilidade de sempre prestar serviço correto. Cabe observar que mais de 50% das falhas observadas em sistemas em uso devem-se a erros de operador [Patterson et alii, 2004; Fox, 2002]. Ou seja, confiabilidade não é somente assegurar que o sistema esteja em correspondência exata com a sua especificação.

Segurança: (*safety*) habilidade de evitar conseqüências catastróficas, ou de grande envergadura, afetando tanto os usuários como o ambiente.

Proteção: habilidade de evitar tentativas de agressão bem sucedidas (em [Avizienis et alii, 2004b] esta é uma propriedade agregadora no mesmo nível que fidedignidade).

Privacidade: habilidade de proteger dados e código contra acesso indevido. Cabe observar que isto se refere não só aos dados contidos na máquina, mas também aos dados em trânsito entre máquinas, mesmo quando estiverem em uso mecanismos diferentes de redes de computadores.

Integridade: ausência de alterações não permitidas (corrupção de elementos).

Robustez: habilidade de detectar o mais cedo possível eventuais falhas de modo que os danos (as correspondentes conseqüências) possam ser mantidas em um patamar aceitável (em [Avizienis et alii, 2004b] esta é considerada uma propriedade secundária).

Recuperabilidade: habilidade em ser rapidamente reposto em operação fidedigna após a ocorrência de uma falha. Em [Avizienis et alii, 2004b] é considerada uma forma de implementação da tolerância a falhas. Já Patterson em [Patterson et alii, 2004] e Fox em [Fox, 2002] consideram recuperabilidade uma característica suficientemente marcante para merecer atenção individualizada no nível das demais características da fidedignidade.

Manutenibilidade: habilidade de ser modificado (evoluído) ou corrigido sem que novos problemas sejam inseridos e a um custo compatível com o tamanho da alteração.

Depurabilidade facilidade de diagnosticar e eliminar possíveis causas de problemas a partir de relatos gerados. Embora a depuração faça parte da manuten-

> ção, o diagnóstico da causa de uma falha a partir de um relato de usuário ou de sintomas observados, tem-se mostrado extremamente frustrante e desgastante, principalmente quando se lida com software produto. A manutenibilidade também não explicita a necessidade de se projetar e incluir instrumentação de modo que seja reduzido o tempo necessário para corretamente diagnosticar e eliminar as causas das falhas identificadas [Staa, 2000].

Basili e Boehm mencionam que cerca de 50% dos sistemas de software tornados disponíveis contêm faltas não triviais [Basili et al, 2001]. Mencionam ainda que cerca de 90% do tempo inoperante de um sistema se deve a cerca de 10% das faltas nele contidas. Por outro lado, é sabido que falhas podem acontecer, independentemente de quão bem se trabalhe. As causas vão desde a reconhecida falibilidade humana até condições que fogem do controle dos desenvolvedores, tais como erros de máquina (possivelmente por causas externas, por exemplo, interferência eletromagnética), erros na plataforma usada (sistema operacional, hardware, rede, banco de dados), falta de energia, falhas em software que coexiste na máquina, agressões (ex. vírus, furto de dados) [Avizienis et alii, 2004a].

Pode-se afirmar que é utópico almejar sistemas absolutamente fidedignos por construção [Berry, 1992]. Também é pouco provável conseguir-se construir sistemas que, embora imperfeitos, jamais acusem uma falha no decorrer de sua vida útil [Whittaker et al, 2000]. Consequentemente, como não se pode assegurar a ausência de falhas, deve-se tentar conviver com a presença delas [Patterson et alii, 2004; Fox 2002] através da construção de sistemas robustos, ou tolerantes a falhas, ou capazes de se auto-corrigirem (*self-healing*) ou capazes de continuar gerando serviços fidedignos apesar de em volume ou funcionalidade menor (*graceful degradation*).

Uma outra fonte de problemas é a deterioração do software [Eick et alii, 2001]. Diferente da manutenção de artefatos físicos, a deterioração do software tende a ser uma conseqüência da sua manutenção. Entretanto, é impossível evitar que ocorra a necessidade de evolução[1], conforme discutido por Lehman e outros em [Belady et al, 1980; Lehman et al, 1985; Lehman et al, 2002]. Considerando que 75% ou mais do esforço de manutenção é inevitável (decorre da evolução ou adaptação do software, imprevisível quando da especificação inicial), surge a necessidade de desenvolver software de forma que seja manutenível. Ou seja, deseja-se que o custo da manutenção seja compatível com a dimensão da funcionalidade alterada e que se possa assegurar a melhoria, ou, no mínimo, a manutenção, de níveis de fidedignidade à medida que se realizam manutenções. É reconhecido também que à medida que o software vai sendo modificado é necessário realizar manutenção preventiva [Avizienis et alii, 2004a] ou a sua reengenharia (refatoração, *refactoring*) conforme discutido em [Beck, 2000; Fowler, 2000].

1 Existe uma controvérsia com relação a manutenção. Alguns autores preferem tratar ***evolução*** e ***adaptação*** como atividades similares a desenvolvimento. Neste caso ***correção***, ***perfecção*** e ***prevenção*** (procurar e eliminar pró-ativamente as faltas possivelmente contidas no software) seriam as únicas atividades de manutenção propriamente dita. Entretanto, a maioria dos autores considera as cinco atividades como manifestações de manutenção, baseando-se na definição de manutenção (aplicável a artefatos físicos) como sendo "qualquer atividade de modificação realizada após a entrega inicial do sistema".

2 Conceitos

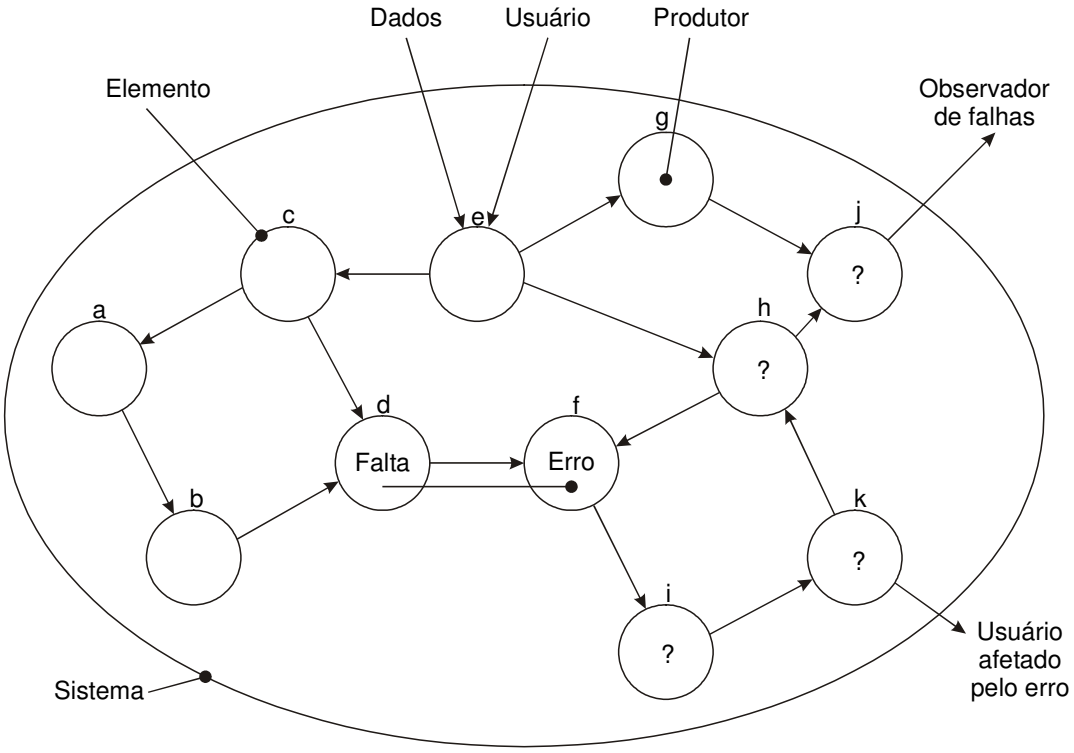

Figura 1. Itens envolvidos em sistemas tolerantes a falhas (adaptado de [Romanowsky, 2005])

Uma das propriedades chave de sistemas fidedignos é eles serem tolerantes a falhas. Na Figura 1 mostramos uma composição de um hipotético sistema de software. Um sistema é composto por diversos ***elementos*** interdependentes e que visam prestar algum ***serviço*** em um determinado ***grau de qualidade***. Sistemas que podem produzir elevados danos caso falhem, evidentemente devem ter um grau de qualidade assegurada bem maior do que sistemas que têm pequeno potencial de produzirem danos. São exemplos de *elementos* de sistemas de software: dados (espaços de dados), estruturas de dados residentes em memória, funções, classes, componentes, arquivos, bases de dados, e mesmo pessoas (usuários, operadores). Elementos são criados, alterados ou destruídos por algum ***produtor***. São exemplos de produtores: programadores, gerentes, usuários, outros sistemas, outros elementos, compiladores, transformadores, geradores, ferramentas, agressores.

Ao produzir ou alterar elementos podem ser cometidos ***erros de produção***. São exemplos de causas de erros de produção: falibilidade humana, falha de hardware, falha de ferramentas utilizadas ao desenvolver. Erros de produção introduzem ***faltas*** nos elementos. Faltas são defeitos latentes que, quando exercitadas, têm o potencial de produzir ***erros de processamento***. Um erro de processamento é um estado ou resultado diferente do desejado contido em algum elemento. São exemplos de erros de processamento: violações de assertivas (contratos), não observância de requisitos funcionais (ex. algoritmo errado) ou não funcionais (ex. desempenho, capacidade), fornecimento de dados ou comandos incorretos. Por exemplo, na Figura 1 o elemento ***d*** contém uma falta que pode produzir um erro de processamento no elemento ***f***. Note que, se não for exercitada, a falta não produzirá erros. Por outro lado, se for exercitada, a falta pode ou não produzir erros dependendo das circunstâncias em que é exercitada. Por exemplo, os erros de uso por parte de algum usuário ocorrem esporadicamente com alguma pequena freqüência. Faltas não necessariamente estão sempre contidas no códi-

go. Por exemplo, um erro de uso pode ser provocado por um auxílio *on-line* que não foi devidamente ajustado ao programa tal como implementado.

Erros de uso podem ser ***esperados*** ou ***excepcionais***. Dado que seres humanos são falíveis, é de se esperar que cometam erros de vez em quando. Da mesma forma ao utilizar redes, ou acesso a arquivos, podem ocorrer erros de transmissão, de leitura ou de gravação. Também ao alocar dados dinamicamente é possível que a memória disponível se esgote, possivelmente provocando erros de acesso. Todos estes erros caem na categoria de erros de uso esperados. Por outro lado, parte-se da hipótese que os programas foram desenvolvidos com perfeição. Neste caso observar uma assertiva não válida corresponde a um erro excepcional.

Em função de algum contexto, erros podem ou não ser **propagados** de elemento para elemento. Por exemplo, na Figura 1 o arco formado pelos elementos ***i***, ***k***, ***h***, ***j*** pode conter, mas ***não necessariamente*** contém, erros propagados a partir do elemento ***f***. Este arco é uma parte do ***domínio de propagação de erros*** a partir de ***f***. Quando um erro se torna observável por algum ***observador de erros***, ele passa a ser uma ***falha***. São exemplos de observadores de erros: validadores de dados de entrada, instrumentos de verificação (ex. assertivas executáveis, verificadores estruturais [Staa, 2000]), usuários, agentes inspetores, *watch dogs*. Infelizmente, também é comum erros passarem despercebidos para fora do sistema. Ou seja, um ***erro afeta um usuário*** externo ao sistema quando este não se dá conta da ocorrência deste erro. Evidentemente, falhas também afetam o usuário, neste caso, no entanto, o usuário estará alertado de que ocorreu o erro observado pela falha.

Falhas e erros não observados provocam ***danos***. Estes podem variar desde danos de pequena monta, por exemplo, perda de algum tempo, necessidade de repetir uma operação, passando por danos consideráveis, por exemplo, perda de oportunidade, quebra de equipamento, prejuízos financeiros, até danos catastróficos, tais como perda de vidas, danos à ecologia, ruína de empresas. Um dos objetivos a alcançar é virtualmente eliminar o ***risco*** de danos de modo que o seu custo médio potencial (risco × dano) seja compatível com o custo de sua prevenção e observação. Em uma análise superficial, e visto que ainda não é possível quantificar adequadamente riscos e danos, isto quer dizer que se deve procurar eliminar o risco de danos grandes ou catastróficos.

Além de provocar danos imediatamente ao ocorrer, um erro pode estar presente por um considerável período de tempo desde o instante em que é gerado até ao momento em que é finalmente observado e tratado, ou mesmo eliminado sem que jamais tenha sido observado. Esta demora entre a ocorrência e a observação pode levar a uma intensiva e catastrófica propagação. Por exemplo, um dado errado alojado em um banco de dados pode passar despercebido por muito tempo. Caso este dado participe em extensivos processamentos, podemos ter grandes volumes de resultados errados e, por conseguinte, danosos.

Para reduzir os riscos inerentes à ocorrência de falhas e de erros não observados, deseja-se ampliar a capacidade de observação de erros, idealmente fazendo com que ocorram durante os testes e transformando todos os erros rapidamente em falhas, ou seja, fazendo com que sempre sejam observados, jamais sendo intermitentes, e também reduzir no que for possível a dimensão do domínio de propagação de erros [Voas, 1997]. Infelizmente não se sabe quais são os elementos que contêm ou são levados a conter faltas. Conseqüentemente, temos que incluir em elevada quantidade eficazes observadores de erros (instrumentos) no código desenvolvido.

3 Meios para alcançar fidedignidade

O problema de fidedignidade não é relevante somente em sistemas de controle de processos ou similares. É importante também ao desenvolver software convencional. Quantas vezes perdemos horas de trabalho em virtude de algum defeito de software? Quantas vezes gastamos um tempo enorme para tentar fazer com que o resultado de um software se apresente tal como desejamos? Quantas vezes já fomos forçados a desistir de utilizar um software simplesmente por não conseguirmos que produza resultados na qualidade esperada?

Adaptando [Avizienis et alii, 2004a] são seguintes as ações que visam assegurar fidedignidade:

Prevenção: desenvolver e manter o software de modo a minimizar ou mesmo eliminar por construção faltas de especificação, arquitetura, projeto, implementação e desenvolvimento. A prevenção visa produzir e evoluir sistemas de modo que sejam fidedignos por construção, o que já mencionamos ser utópico. Visa também a construção de sistemas manuteníveis e evolutíveis. Um dos ideais[2] a alcançar é a ***ausência de faltas***, ou seja, a limitação por construção da presença de defeitos nos diversos elementos que compõem o sistema. Na prática o que desejamos é que seja muito pequeno o número de faltas contidas no software e em equipamentos. Por outro lado sabemos que não podemos impedir que humanos falhem, mas, através de uma preocupação com a interação deles com os sistemas, podemos reduzir o risco disto ocorrer [Neal et alii, 2003; Fox, 2002].

Controle da qualidade: utilizar técnicas de controle da qualidade capazes de observar a existência de faltas ainda durante o desenvolvimento ou da manutenção. O controle da qualidade realizado durante o desenvolvimento visa detectar as faltas injetadas no sistema apesar do esforço de prevenção. Na prática o desenvolvimento será sempre acompanhado de controle da qualidade. Por outro lado, sabe-se que o controle da qualidade não conduz à qualidade, produzindo somente um laudo de problemas observados. Estes precisam ser diagnosticados e a seguir eliminados, sem que novos problemas sejam inseridos. Conseqüentemente, é necessário atuar sobre o processo de desenvolvimento de modo que, através de disciplina e organização do trabalho, se reduza significativamente o número de faltas injetadas.

Observação de erros: utilizar técnicas capazes de observar a ocorrência de falhas durante a execução [Staa, 2000]. O objetivo é reduzir a dimensão do domínio de propagação de erros, a duração da permanência como erro não observado e, evidentemente, reduzir o risco de erros deixarem de ser observados.

Tolerância de falhas: utilizar técnicas capazes de tornar o sistema capaz de operar de forma fidedigna, mesmo na presença de erros ou falhas. É aceitável o sistema reduzir alguma funcionalidade e/ou capacidade de processamento desde que assegure que continue operando de forma fidedigna e

2 Um ***ideal*** é um objetivo ou meta que se sabe que não será ou muito dificilmente será alcançado com perfeição.

que esta redução não traga danos consideráveis (*graceful degradation* [Shelton et al, 2001; Bentley, 2005]).

Auto-recuperação: (*self-healing*) utilizar técnicas capazes de restabelecer o estado do sistema de modo que esteja correto e em perfeita consonância com o mundo externo [Ravi et al, 2004, Sidiroglou et alii, 2005].

Remoção: diagnosticar e eliminar eventuais faltas a partir das falhas observadas. Para tal torna-se necessário facilitar a diagnose precisa (***diagnosticabilidade***) e a depuração eficaz e eficiente (***depurabilidade***).

Predição: utilizar modelos capazes de estimar os potenciais problemas contidos no sistema.

4 Objetivo da diretriz de pesquisa

É reconhecido ser um grande desafio desenvolver a baixo custo software fidedigno satisfazendo as características discutidas. Em particular deve-se levar em conta a grande tendência atual para o desenvolvimento de sistemas distribuídos, ou residentes na *Web*, e sistemas autônomos ou multi-agentes. É sabido que tais sistemas sempre representaram um desafio às tentativas de garantia da fidedignidade tal como descrita acima.

É objetivo da presente diretriz de pesquisa: desenvolver, avaliar e aprimorar fundamentos, processos, *frameworks*, métodos, técnicas padrões de arquitetura e de projeto capazes de efetiva e eficazmente auxiliar desenvolvedores de software a desenvolverem e manterem sistemas fidedignos. Como muitos dos problemas são decorrentes da formação inadequada dos profissionais que atuam em Engenharia de Software, é necessário também reestruturar currículos e desenvolver material de ensino de modo que os profissionais formados estejam preparados a utilizar este novo instrumental.

5 Atividades em progresso

São inúmeras as possíveis ações a tomar de modo que se possa desenvolver sistemas mais fidedignos. A seguir enunciamos algumas delas, explicitando o atual (2006) foco de atenção de nossos trabalhos de pesquisa:

- Desenvolver software o mais correto por construção que se consiga [Larus et alii, 2004; Weinstock et alii, 2004; Staa, 2000]. Para isto são utilizados processos, padrões técnicos, ferramentas, técnicas de modelagem e técnicas formais. Embora teoricamente possa-se desenvolver software correto, a prática mostra que esta é uma meta esquiva em virtude da inerente falibilidade humana, das limitações das plataformas de software (p.ex. sistemas operacionais, redes), das limitações ou dificuldades inerentes às técnicas formais (p.ex. números vírgula flutuante, dependências do tempo), erros cometidos pelo usuário, e da possibilidade da ocorrência de falhas espúrias (p.ex. erro de hardware, interferência eletromagnética). Além disso, cabe lembrar que poucos desenvolvedores têm a formação necessária para habilitá-los a utilizar com desenvoltura técnicas formais sofisticadas. Este trabalho envolve o desenvolvimento de um arcabouço de apoio ao teste de programas redigidos em C, a melhoria dos padrões de programação contidos em [Staa, 2000] e a participação no projeto Anúbis coordenado pelo prof Hermann Haeusler.

- Desenvolver fundamentos, modelos, ferramentas e tecnologias visando a criação de software estável e recuperável [Patterson et alii, 2004]. Um foco de interesse específico são sistemas autônomos, sistemas multi-agentes, sistemas distribuídos e sistemas embarcados. Um software é ***estável*** na medida em que pode ser modificado e possivelmente reutilizado sem comprometer a sua fidedignidade. Um software é ***recuperável*** na medida em que pode ser rapidamente reposto em funcionamento fidedigno após a ocorrência de uma falha. Este aspecto é parte do trabalho dos alunos de doutorado João Magalhães, Matheus Leite e Roberta Coelho.

- Desenvolver e aprimorar meta-ferramentas de apoio ao desenvolvimento e à manutenção [Staa, 1993] apoiados em modelos, e capazes de verificar modelos, transformar modelos e gerar código a partir de modelos [Sendall et al, 2003; Meservy et al, 2005; Franca, 2000]. Este é o velho sonho de desenvolver uma nova versão de Talisman [Staa, 1993]

- Testar programas utilizando processos de teste e métodos de seleção de casos de teste que tenham elevada chance de acusar falhas. Estes processos de teste prevêem, entre outros, o estabelecimento de uma seqüência de teste e integração que procura aumentar a chance de se detectar cedo problemas ainda remanescentes nos programas. Massas de teste contêm uma seleção de exemplos de uso (cenários e casos de teste) de um conjunto potencialmente infinito de exemplos. Conseqüentemente, testes não podem ***assegurar*** a fidedignidade do software. Entretanto, mesmo que inerentemente limitados, a prática tem mostrado que testes bem projetados são eficazes técnicas de garantia da qualidade. Um foco de atenção explícito são sistemas autônomos multi-programados ou multi-processados. Este aspecto é parte do trabalho da aluna de doutorado Roberta Coelho.

- Desenvolver arcabouços (*frameworks*) para a automação dos testes e dos processos de desenvolvimento na parte que toca a testes [Beck, 2003; Mackinnon et alii, 2001; Succi et al, 2001; Fewster et al, 1999]. Esses arcabouços devem poder ser incluídos e excluídos de forma transparente durante o desenvolvimento e a manutenção. Ou seja, exceto pelo tempo de processamento, a sua presença ou ausência não deve ser perceptível à porção do programa que se encontra em teste. O teste automatizado permite submeter massas de teste tão extensas e tão complexas quanto se queira, realizando os testes sem intervenção humana. Embora adicione um custo não desprezível para a produção do código e das massas de teste, a automação contribui para uma sensível redução de custos e tempos de desenvolvimento, em virtude da possibilidade do reteste sistemático sempre que seja realizada alguma modificação nos programas. Viabiliza também uma modalidade de desenvolvimento denominada "*test driven development*" preconizada por métodos ágeis de desenvolvimento de software [Beck, 2003]. Tenho um arcabouço de apoio ao teste automatizado para C em uso pelos alunos de graduação e estou, no contexto do projeto Talisman, aprimorando o arcabouço para C++.

- Desenvolver e utilizar *design patterns* voltados para estabelecer a testabilidade de programas. Um *design pattern* é uma organização padronizada de elementos, em geral classes, visando resolver determinado problema de programação [Gamma et alii, 1995]. *Design patterns* voltados para o teste [Coelho et alii, 2005] procuram dar ao desenvolvedor mecanismos pré-fabricados e que comprovadamente ajudam a tornar programas amenos ao teste. Este aspecto é parte central do trabalho da aluna de doutorado Roberta Coelho.

- Incorporar instrumentação no código [Staa, 2000; Satpathy et al, 2004]. Da mesma forma como o arcabouço de teste, a instrumentação também deve ser transparente. Entre

as várias formas de instrumentação destaca-se o uso de assertivas e verificadores estruturais automatizados. Estes instrumentos são capazes de detectar inconsistências nos estados de execução dos programas. Uma vez detectados problemas, estes são repassados para tratadores capazes de impedir a propagação de danos ou até mesmo recuperar para poder retomar o processamento fidedigno. O uso de assertivas viabiliza uma modalidade de desenvolvimento denominada "*contract driven development*" em que os contratos são controlados em tempo de execução e não somente como instrumentos de especificação e verificação formal [Meyer, 1992; Mitchell, 2004]. Este aspecto é parte do trabalho do aluno de doutorado João Magalhães.

- Incorporar simuladores e módulos de verificação no código (*mock objects*) [Mackinnon et alii, 2001]. Com o uso de *mock objects* pode-se não só implementar *stubs*, como tornar estes sensíveis aos testes específicos sendo realizados, capacitando-os a gerarem condições anômalas sem que se tenha que adulterar o código, gerando mutantes, do programa em teste. Este aspecto é parte dos trabalhos dos alunos João Magalhães e Matheus Leite.
- Utilizar técnicas orientadas à recuperabilidade [Patterson et alii, 2004, Patterson, 2004, Fox, 2002]. Tais técnicas visam à recuperação seletiva disparada pela instrumentação incorporada ao código. Estas técnicas visam capacitar o programa falhado a restabelecer rapidamente um estado confiável com um mínimo de perdas de informação. Asseguram assim a minimização dos períodos de indisponibilidade de serviços prestados pelo programa. Aumentar a disponibilidade é uma necessidade para um grande número de sistemas, em especial os que efetuam alguma forma de *e-business*. Este aspecto é parte dos trabalhos dos alunos João Magalhães e Matheus Leite.
- Instanciar modelos de processos padronizados de modo que possam ser utilizados por pequenas e médias empresas [Dangle et alii, 2005; Beck, 2000; Cockburn, 2002; Highsmith 2002] no desenvolvimento de sistemas fidedignos [mpsBR 2005; Chrissis et alii, 2003]. Este é objetivo dos trabalhos dos alunos de mestrado: Gloria Oliveira, Henrique Prange e Rafael Espinha
- Desenvolver modelos e programas de medição com o objetivo de identificar problemas nos processos de desenvolvimento de software e características de artefatos que trazem risco ao alcance de elevada fidedignidade [Gallagher et alii, 2005]. Este é objetivo dos trabalhos dos alunos de mestrado: Bernardo Arraes Vinhosa e Carlos Freud.

Muitas destas técnicas vêm sendo utilizadas com sucesso e têm contribuído para uma significativa melhoria da qualidade concomitante com uma também significativa melhoria da produtividade. No entanto, estas técnicas têm sido estudadas de forma não integrada. Também faltam soluções eficazes visando sistemas multi-programados e/ou multi-processados, tais como os que se encontra em aplicações para a Web, em sistemas embarcados ou em sistemas multi-agentes.

6 Comentários finais

Este texto teve por objetivo delinear e contextualizar os atuais interesses de pesquisa no contexto do desenvolvimento de software fidedigno. Projetos de pesquisa específicos serão documentados em outras obras.

A necessidade de melhores práticas e de uma melhor formação dos profissionais é discutido em vários artigos e livros. Selecionando alguns poucos temos:

- Boehm e Basili em [Basili et al, 2001] que enumeram os 10 principais desafios na área de Engenharia de Software visando o desenvolvimento de software de qualidade adequada com produtividade satisfatória. Essencialmente são problemas encontrados no estado da prática e que há bastante tempo vêm desafiando tanto os pesquisadores como os profissionais que atuam no desenvolvimento de software.
- Charette discute em [Charette, 2005] várias razões para que software falhe. Apresenta também algumas conseqüências destas falhas. Ao final tece considerações sobre a forma pouco disciplinada com que software tem sido desenvolvido, mostrando que o emprego de técnicas mais formais reduz em muito o risco e o custo total do software.
- Finalmente, Glass em [Glass, 2003] discute diversos aspectos que caracterizam o desenvolvimento de software e que, por serem freqüentemente desconsiderados, conduzem a software de qualidade inadequada causando severos prejuízos.

Um fato que deixa muitas pessoas perplexas é o porquê da insistência na falta de rigor no desenvolvimento de software. Em qualquer escritório de engenharia produzem-se desenhos técnicos, realizam-se e verificam-se cálculos antes de iniciar qualquer construção ou fabricação. Todas as plantas (*blue-prints*) são devidamente catalogadas e arquivadas de modo que outros engenheiros possam, mais tarde, recorrer a elas sempre que necessário. Por que então na área de software diversos profissionais de renome sugerem que o uso de modelos, desenhos, formalismos são desnecessários? Por que os gerentes de desenvolvimento se interessam mais pela **quantidade** de código e pelo aspecto da interface do software, e quase nada pela sua qualidade? Por que programadores pouco controlam a qualidade de seus programas? Sabemos que software tende a ser cada vez mais complexo, mais e mais pessoas e serviços dependem dele. Mesmo assim a sua fidedignidade tem sido tratada com pouco caso. Ao mesmo tempo os prejuízos provocados pelo mau funcionamento ou pela inadequação às necessidades do usuário têm sido enormes. Por que este descaso? É interesse desta diretriz de pesquisa contribuir para melhorar este quadro.

Referências bibliográficas

[Avizienis et alii, 2004a] Avizienis, A.; Laprie, J-C.; Randell, B.; "Dependability and Its Threats: A Taxonomy"; in: Jacquart, R.; eds.; **Proceedings of the IFIP 18th World Computer Congress: Building the Information Society**, Toulouse, France; IFIP Congress Topical Sessions; Dodrecht: Kluwer; 2004; pags 91-120 Buscado em: 7/abril/2006; URL: http://rodin.cs.ncl.ac.uk/Publications/avizienis.pdf

[Avizienis et alii, 2004b] Avizienis, A.; Laprie, J-C.; Randell, B.; Landwehr, C.; "Basic Concepts and Taxonomy of Dependable and Secure Computing"; **IEEE Transactions on Dependable and Secure Computing** 1(1); Los Alamitos, CA: IEEE Computer Society; 2004; pags 11-33

[Basili et al, 2001] 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

[Beck, 2000] Beck, K.; **Extreme Programming Explained: Embrace Change**; New York, NY: Addison-Wesley; 2000

[Beck, 2003] Beck, K.; **Test-Driven Development by Example**; New York, NY: Addison-Wesley; 2003

[Belady et al, 1980] Belady, L.A.; Lehman, M.M.; "Programs, Life Cycles and Laws of Software Evolution"; ***Proceedings IEEE*** 68(9); 1980; pags pp 1060-1076

[Bentley, 2005] Bentley, P.; "Investigations into Graceful Degradation of Evolutionary Developmental Software"; **Natural Computing** 4(4); Berlin: Springer; 2005; pags 417-437

[Berry, 1992] Berry, D.M.; **Academic Legitimacy of the Software Engineering Discipline**; Technical Report 92-TR-34, Software Engineering Institute, Carnegie Mellon University; 1992

[Charette, 2005] Charette, R.N.; "Why software fails"; ***IEEE Spectrum*** 42(9); Los Alamitos, CA: IEEE Computer Society; 2005; pags 42-49

[Chrissis et alii, 2003] Chrissis, M.B.; Konrad, M.; Shrum, S.; **CMMI: Guidelines for Process Integration and Product Improvement**; New York, NY: Addison-Wesley; 2003

[Cockburn, 2002] Cockburn, A.; **Agile Software Development**; New York, NY: Addison-Wesley; 2002

[Coelho et alii, 2005] R. Coelho, U. Kulesza, A. Staa, C. Lucena, "The Layered Information System Test Pattern"; in: Kon, F.; Rising, L.; eds.; **Latin American Conference on Pattern Languages of Programming SugarLoafPLoP 2005**; São Carlos, SP: ICMC/USP; 2005; pags 114-129

[Dangle et alii, 2005] Dangle, K.C.; Larsen, P.; Shaw, M.; Zelkowitz, M.V.; "Software Process Improvement in Small Organizations: A Case Study**"; IEEE Software** 22(6); Los Alamitos, CA: IEEE Computer Society; 2005; pags 68-75

[Eick et alii, 2001] Eick, S.G.; Karr, A.K.; Graves, T.L.; Marron, J.S.; Mockus, A.; "Does Code Decay? Assessing the Evidence from Change Management Data"; ***IEEE Transactions on Software Engineering*** 27(1); Los Alamitos, CA: IEEE Computer Society; 2001; pags 1-12

[Fewster et al, 1999] Fewster, M.; Graham, D.; ***Software Test Automation***; New York, NY: Addison-Wesley; 1999

[Fowler, 2000] Fowler, M.; **Refactoring: Improving the Design of Existing Code**; New York, NY: Addison-Wesley; 2000

[Fox, 2002] Fox, A.; "Toward Recovery-Oriented Computing"; Invited talk; **VLDB 2002**; Buscado em: Março 2006; URL: www.cs.ust.hk/vldb2002/VLDB2002-proceedings/papers/S25P01.pdf

[Franca, 2000] Franca, L.P.A.; Um **Processo para a Construção de Geradores de Artefatos**; Tese de Doutorado; Departamento de Informática, PUC-Rio; 2000

[Gallagher et alii, 2005] Gallagher, B.P.; Case, P.J.; Creel, R.C.; Kushner, S.; Williams, R.C.; **A Taxonomy of Operational Risks**; Technical Report CMU/SEI -2005-TN-036, Software Engineering Institute, Carnegie Mellon University; 2005

[Gamma et alii, 1995] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; **Design Patterns: Elements of Reusable Object-Oriented Software**; New York, NY: Addison-Wesley; 1995

[Glass, 2003] Glass, R.L.; **Facts and Fallacies of Software Engineering**; New York, NY: Addison-Wesley; 2003

[Highsmith, 2002] Highsmith, J.; **Agile Software Development Ecosystems**; New York, NY: Addison-Wesley; 2002

[Larus et alii, 2004] Larus, J.R.; Ball, T.; Das, M.; DeLine, R.; Fähndrich, M.; Pincus, J.; Rajamani, S.K.; Venkatapathy, R.; "Righting Software"; **IEEE Software** 21(3); Los Alamitos, CA: IEEE Computer Society; 2004; pags 92-100

[Lehman et al, 1985] Lehman, M.M.; Belady, L.A.; **Program Evolution: Processes of Software Change**; London: Academic Press; 1985

[Lehman et al, 2002] Lehman, M.M.; Ramil, J.F.; "Sofware Evolution and Software Evolution Process"; **Annals of Software Engineering** 14; Dodrecht: Kluwer; 2002; pags 275-309

[Mackinnon et alii, 2001] Mackinnon, T.; Freeman, S.; Craig, P.; "Endo-Testing: Unit Testing with Mock Objects"; in: Succi, G.; Marchesi, M.; eds.; **Extreme Programming Examined**; New York, NY: Addison-Wesley; 2001; pags 287-302

[Meservy et al, 2005] Meservy, T.O.; Fenstermacher, K.D.; "Transforming Software Development: An MDA Road Map"; **IEEE Computer** (9); Los Alamitos, CA: IEEE Computer Society; 2005; pags 52-58

[Meyer, 1992] Meyer, B.; "Applying Design by Contract"; **IEEE Computer** 25(10); Los Alamitos, CA: IEEE Computer Society; 1992; pags 40-51

[Mitchell, 2004] Mitchell, R.; **Design by Contract: Bringing together Formal Methods and Software Design**; Tutorial; University of Brighton; 2004

[mpsBR 2005] **MPS.BR - Melhoria de Processo do Software Brasileiro: Guia Geral**, Versão 1.0; Rio de Janeiro: SOFTEX; 2005

[Neal et alii, 2003] Neal, A.; Humphreys, M.; Leadbetter, D.; Lindsay, P.; "Development of hazard analysis techniques for human-computer systems"; in Edkins, G.; Pfister, P.; eds.; **Innovation and Consolidation in Aviation**; Aldershot, UK: Ashgate; 2003; pags 255-262

[Patterson et alii, 2004] Patterson, D.; Brown, A.; Broadwell, P.; Candea, G.; Chen, M.; Cutler, J.; Enriquez, P.; Fox, A.; Kycyman, E.; Merzbacher, M.; Oppenheimer, D.; Sastry, N.; Tetzlaff, W.; Traupman, J.; Treuhaft, N.; **Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies**; Technical Report UCB/CSD-02-1175, Computer Science, University of California Berkeley; 2002; Buscado em: agosto/2004; URL: http://roc.cs.berkeley.edu/papers/ROC_TR02-1175.pdf

[Patterson, 2004] Patterson, D.; **Embracing Failure: Recovery Oriented Computing**; Seminário; University of California at Berkeley; 2002; Notas de apresentação. Buscado em: 7/8/2004; URL: roc.cs.berkeley.edu/talks

[Ravi et al, 2004] Ravi, R.K.; Sathyanarayana, V.; "Container Based Framework for Self-Healing Software System"; **10th IEEE International Workshop on Future Trends of Distributed Computing Systems** (FTDCS'04); 2004; pp. 306-310

[Romanowsky, 2005] Romanowsky, A.; **Software Dependability**; *Key Note Address*; 19o. Simpósio Brasileiro de Engenharia de Software (SBES'2005); Uberlândia, MG, Brasil; 2005

[Satpathy et al, 2004] Satpathy, M.; Siebel, N.T.; Rodriguez, D.; "Assertions in Object Oriented Software Maintenance: Analysis and Case Study"; Proceedings of the **20th IEEE International Conference on Software Maintenance** (ICSM'04); Los Alamitos, CA: IEEE Computer Society; 2004

[Scott et alii, 1987] Scott, R.K.; Gault, J.W.; McAllister, D.F.; "Fault-Tolerant Reliability Modelling"; **IEEE Transactions on Software Engineering** 13(5); Los Alamitos, CA: IEEE Computer Society; 1987; pags 582-592

[Sendall et al, 2003] Sendall, S.; Kozaczynski, W.; "Model Transformation - the Heart and Soul of Model-Driven Software Development"; **IEEE Software** (5); Los Alamitos, CA: IEEE Computer Society; 2003; pags 42-45

[Shelton et al, 2001] Shelton, C.P; Koopman, P.; "Developing a Software Architecture for Graceful Degradation in an Elevator Control System"; **Workshop on Reliability in Embedded Systems**; 2001; Buscado em: março 2006; URL:
URL: www.ece.cmu.edu/~koopman/roses/wres01/shelton_wres01.pdf

[Sidiroglou et alii, 2005] Sidiroglou, S.; Locasto, M.E.; Keromytis, A.D.; "Hardware Support For Self-Healing Software Services"; **ACM SIGARCH Computer Architecture News** 33(1); 2005; pags 42-47; Buscado em 9/4/2005; URL http://www.cs.columbia.edu/~angelos/ Papers/2005/comparch-shs.pdf

[Sneed, 2004] Sneed, H.M.; "A Cost Model for Software Maintenance & Evolution"; **Proceedings of the International Conference on Software Maintenance (ICSM'04)**; Los Alamitos, CA: IEEE Computer Society; 2004; pags 1063-1072

[Staa, 1993] Staa, A.v.; **Ambiente de Engenharia de Software Assistido por Computador - TALISMAN**; versão 4.3; Rio de Janeiro, RJ: Staa Informática Ltda.; 1993

[Staa, 2000] Staa, A.v.; **Programação Modular**; Rio de Janeiro, RJ: Campus; 2000

[Succi et al, 2001] Succi, G.; Marchesi, M.; eds.; **Extreme Programming Examined**; New York, NY: Addison-Wesley; 2001

[Voas, 1997] Voas, J.; "How Assertions Can Increase Test Effectiveness"; **IEEE Software** 14(2); Los Alamitos, CA: IEEE Computer Society; 1997; pags 118-122

[Weinstock et alii, 2004] Weinstock, C.B.; Goodenough, J.B.; Hudak, J.J.; **Dependability Cases**; CMU/SEI -2004-TN-016, Software Engineering Institute, Carnegie Mellon University; 2004

[Whittaker et al, 2000] Whittaker, J.A.; Voas, J.; "Toward a More Reliable Theory of Reliability"; **IEEE Computer** 33(12); Los Alamitos, CA: IEEE Computer Society; 2000; pags 36-42