Um Sistema de Padrões para Injeção de Falhas por Software

Nelson Guilherme Mendes Leme

Dissertação de Mestrado

Instituto de Computação (IC)

Universidade Estadual de Campinas (UNICAMP)

UM SISTEMA DE PADRÕES PARA INJEÇÃO DE FALHAS POR SOFTWARE

Nelson Guilherme Mendes Leme (1)

22 de Agosto de 2001

Banca Examinadora:

- Prof.ª Dr.ª Eliane Martins (Orientadora)
  IC – UNICAMP
- Prof. Dr. Luiz Eduardo Buzato
  IC – UNICAMP
- Prof.ª Dr.ª Taisy Silva Weber
  Instituto de Informática – UFRGS
- Prof. Dr. Edmundo Roberto Mauro Madeira (suplente)
  IC – UNICAMP

1 - Auxílio concedido pela FAPESP, processo # 99/09297-7.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Leme, Nelson Guilherme Mendes

L542s Um sistema de padrões para injeção de falhas por software / Nelson Guilherme Mendes Leme -- Campinas, [S.P. :s.n.], 2001.

Orientador : Eliane Martins

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Padrões de projetos. 2. Software – Desenvolvimeto. 3. Engenharia de software - Desenvolvimento. I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

UM SISTEMA DE PADRÕES PARA INJEÇÃO DE FALHAS POR SOFTWARE

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Nelson Guilherme Mendes Leme e aprovada pela Banca Examinadora.

Campinas, 22 de agosto de 2001.

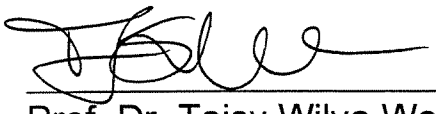
Prof.ª Dr.ª Eliane Martins
Universidade Estadual de Campinas
(Orientadora)

Prof.ª Dr.ª Cecília Mary Fischer Rubira
Universidade Estadual de Campinas
(Co-orientadora)

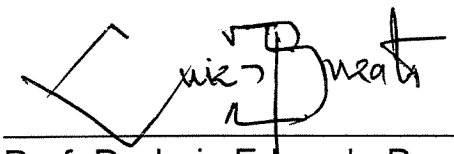
Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

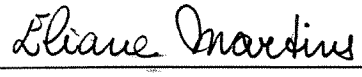
Tese defendida e aprovada em 22 de agosto de 2001, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Taisy Wilva Weber
UFRGS



Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP



Profa. Dra. Eliane Martins
IC - UNICAMP

AGRADECIMENTOS

À minha orientadora, Profa. Dra. Eliane Martins, pelo incentivo e apoio durante todo o trabalho em meu projeto de mestrado e em minha dissertação.

À minha co-orientadora, Profa. Dra. Cecília Mary Fischer Rubira, pelos valiosos conselhos e orientação dados durante o trabalho em meu projeto de mestrado e em minha dissertação.

Aos meus familiares, que me deram estímulo e apoio ao longo do meu trabalho, e por sempre acreditarem em mim.

Aos meus amigos, que muito contribuíram com seus incentivos, e por terem feito o trabalho no mestrado ter tido momentos muito divertidos.

À Universidade Estadual de Campinas (UNICAMP) que me proporcionou lugares incríveis para eu conhecer e explorar.

A Montgomery Scott, que me ensinou que o mínimo que se exige de um engenheiro é o impossível, e para ontem.

À FAPESP pelo apoio financeiro através da bolsa de mestrado sob o processo # 99/09297-7.

RESUMO

O uso de sistemas computacionais tem se expandido cada vez mais. Esses sistemas vêm sendo usados em aplicações críticas, que devem dar uma resposta esperada mesmo na presença de falhas. Uma das formas encontradas de se garantir isso é testar o sistema usando Injeção de Falhas. Nesse processo são simuladas falhas e observada a resposta do sistema nessas circunstâncias. Uma das formas mais populares de se fazer isso é através de Injeção de Falhas por Software, onde um trecho especial de código, associado ao sistema sob teste, procura simular a presença de falhas. Várias ferramentas e programas que realizam Injeção de Falhas por Software já foram desenvolvidas. Entretanto, novos tipos de sistema são criados, e é difícil encontrar ferramentas para esses sistemas. Portanto, há a necessidade de se desenvolver novos programas para realizar Injeção de Falhas por Software. Uma maneira de se facilitar isso é através da criação de Padrões para desenvolver tais sistemas. Através de Padrões, pode-se descrever a arquitetura de programas de Injeção de Falhas por Software, bem como estruturas que esses programas usariam. Isso de uma maneira já determinada e independente de linguagem de programação, e dessa maneira apta a ser utilizada pelo maior número possível de desenvolvedores. Esses Padrões para Injeção de Falhas por Software, organizados na forma de um Sistema, estão expostos neste trabalho. Também é mostrada aqui a ferramenta de Injeção de Falhas JACA, criada com base nesses Padrões, não só como exemplo da aplicação dos mesmos, mas também com utilidade própria, de testar sistemas na presença de falhas.

ABSTRACT

The utilization of computing systems has increased continuosly. That includes the increase in use of systems running in critical applications, when those systems must give an expected answer even in the presence of faults. One way of guaranteeing this is testing the system through Fault Injection. In that process, faults are simulated and the answer of the system to those conditions is observed. One of the most popular ways of doing this is using Software Fault Injection, in which a special piece of code, associated with the system under test, tries to simulate the presence of faults. Many tools and programs that perform Software Fault Injection have already been developed. However, new kinds of systems are being created, and there are no tools for such systems. Therefore, there is a need of development of new programs that make Software Fault Injection. This process could be eased through the creation of Patterns to develop such programs. Through the use of Patterns, the architecture and structures used by Software Fault Injection programs could be described. And that would be done in a standard and programming language independent way, and therefore it could be used by the majority of developers. Those Software Fault Injection Patterns, organized in a System, are exposed in the present paper. Also, a Fault Injection Tool, JACA, is described. That tool was designed based on the Pattern System. It is not only an example of the implementation of those Patterns in a real program, but also as Fault Injection tool with its own usefulness, which is able to test systems in the presence of faults.

ÍNDICE

ÍNDICE DE FIGURAS

Índice de Tabelas

CAPÍTULO I: INTRODUÇÃO

I.1. Contexto

Uso intenso de sistemas computacionais. Atualmente, sistemas computacionais tem tido uso cada vez mais amplo. Para uma enorme gama de atividades são usados computadores. Algumas vezes estes tem uma função de auxiliar uma dada atividade humana. Mas em muitas outras eles tem um papel essencial: sem eles não é possível realizar uma dada atividade. Tem-se como exemplo disso os modernos aviões de transporte de passageiros: eles são atualmente comandados por computadores, e em caso de falha destes o piloto não tem como controlar a aeronave. Outro exemplo são os sistemas bancários: sem os grandes sistemas computacionais usados pelas instituições financeiras, seria impossível controlar a movimentação financeira de um grande banco.

Problemas. De acordo com isso, um sistema computacional que subitamente cesse de realizar suas funções pode causar sérios problemas. Basta ver os exemplos acima. No caso de uma aeronave de passageiros, se seus computadores pararem de funcionar ela pode simplesmente cair. Já no caso de uma instituição financeira, ela pode ter de interromper suas operações, o que trará grandes prejuízos financeiros. Muito provavelmente, tal interrupção na prestação dos serviços feitos por um sistema computacional terá como causa uma *falha*. Uma falha consiste em um componente, tanto de hardware como de software, que apresenta um comportamento anômalo dentro do sistema. Portanto, sistemas computacionais que realizam aplicações críticas deverão continuar rodando, mesmo na presença de falhas. Para isso, esses sistemas deverão possuir um *Mecanismo de Tolerância a Falhas* (MTF). O MTF irá permitir que o sistema dê uma resposta esperada mesmo quando da ocorrência de falhas. Os sistemas que apresentam essa característica são denominados *sistemas tolerantes a falha*.

Necessidade de testes. Porém, não basta apenas se implementar um MTF para se ter um sistema tolerante a falha. É necessário também testá-lo adequadamente, para verificar se o MTF irá tratar como esperado a ocorrência de falhas. Se fizer isso, o sistema não irá apresentar algum comportamento inesperado, evitando-se possíveis danos. Portanto, é fundamental testar sistemas tolerantes a falha.

Teste por Injeção de Falhas. Uma abordagem que tem se mostrado popular para testar esses sistemas é o uso de *Injeção de Falhas*. Nas páginas seguintes esse conceito será melhor explicado, porém, pode-se dizer que é uma técnica de teste onde se procura produzir ou simular a presença de falhas e se observa o sistema sob teste para se verificar qual será sua resposta nessas condições. Com isso, se consegue uma avaliação mais precisa

do comportamento do sistema na presença de falhas. Se eventualmente o sistema não tratar de maneira esperada alguma falha, isso será detectado com o teste por Injeção de Falhas, e se poderá corrigir o sistema.

Ferramentas de Injeção de Falhas. Confirmando isso, um número de ferramentas que realizam teste por Injeção de Falhas vêm sendo desenvolvidas. Essas ferramentas realizam o processo descrito acima: executam o sistema sob teste e produzem ou simulam a presença de falhas, e monitoram o sistema para verificar qual é seu comportamento. Essas ferramentas realizam esse processo através de diversas abordagens, como será visto mais adiante. Atualmente sua utilidade não se resume a testar sistemas tolerantes a falha: também são usadas para testar sistemas que não possuem essa característica, para se avaliar que danos poderiam causar se forem executados na presença de uma falha.

I.2. Motivação

Ferramentas para novos sistemas. Já há um número de ferramentas de teste por Injeção de Falhas. Elas vêm se tornando parte importante do teste de sistemas tolerantes a falha, e também se dissemina seu uso em sistemas convencionais. Contudo, a computação é um campo muito dinâmico: novos tipos de sistemas computacionais são criados com freqüência. E muitos desses novos sistemas computacionais precisam ser tolerantes a falha. Por exemplo, os bancos começam agora a implementar sistemas para realizar transações bancárias em pequenos computadores de mão (*hand devices* ou *palm top computers*). Esses sistemas têm que ser tolerantes a falha, porque senão podem causar prejuízos financeiros aos bancos ou aos clientes. No entanto, pelo fato dos computadores de mão serem uma tecnologia ainda recente, é de se esperar que não haja uma ferramenta de Injeção de Falhas para testa-los.

Ferramentas para sistemas muito específicos. Outro exemplo, também importante, é o caso de sistemas embutidos (*embedded systems*). Esses sistemas são muito freqüentemente feitos sob encomenda, e portanto não haveria uma ferramenta de Injeção de Falhas que os pudesse testar. Todavia, pode ser muito importante testar um sistema embutido na presença de falhas, como por exemplo os sistemas embutidos aeronáuticos.

I.3. Objetivos

Padrões para Injeção de Falhas. Assim, há a necessidade de se desenvolver novas ferramentas de Injeção de Falhas. Tanto para suprir a necessidade de sistemas específicos como para novos sistemas. Para facilitar esse trabalho, este projeto se propõe a utilizar outro conceito que vem ganhando popularidade, que são os *padrões de software*. Eles serão melhor explicados no capítulo II, porém pode-se dizer que são, basicamente, a documentação de soluções para problemas recorrentes no desenvolvimento. Através deles, pode-se economizar tempo e esforço no desenvolvimento. Formulando-se, então, padrões para criar ferramentas de Injeção de Falhas, o desenvolvimento destas últimas seria facilitado. Como conseqüência, desenvolvedores poderiam criar ferramentas para fazer Injeção de Falhas em novos tipos de sistemas, ou em sistemas muito específicos, difundindo o uso de Injeção de Falhas para diversas aplicações.

Ferramenta JACA. Além disso, este trabalho se propõe a mais um objetivo. Tendo definido um sistema de padrões para Injeção de Falhas, resta verificar se o mesmo

realmente facilita o desenvolvimento de ferramentas de Injeção. Para tanto, irá se criar uma nova ferramenta de Injeção de Falhas, a ferramenta JACA. Essa ferramenta irá tomar por base o sistema de padrões. Então, a partir da experiência prática de se construir uma ferramenta com os padrões, poderá se avaliar a real utilidade dos mesmos. Mas, a ferramenta JACA não se restringe a ser um mero demonstrativo dos padrões. Também se pretende que ela seja uma ferramenta de Injeção de Falhas perfeitamente funcional, que poderá ser usada em testes por Injeção de Falhas. Desde já pretende-se que ela seja uma ferramenta altamente adaptável. A idéia é que se a ferramenta JACA não satisfizer exatamente os requisitos para o teste de um dado sistema com Injeção de Falhas, com um mínimo de reescrita ela poderia ser adaptada para trabalhar dentro desses requisitos.

Objetivo geral. Os objetivos acima, a construção de um sistema de padrões para Injeção de Falhas e o desenvolvimento da ferramenta JACA, na verdade podem ser resumidos em apenas um. Esse objetivo seria dar o máximo de recursos para o desenvolvedor poder utilizar Injeção de Falhas. Se, na fase de testes de um sistema, o desenvolvedor verificar que pode fazer testes por Injeção de Falhas no sistema usando a ferramenta JACA tal como ela se encontra, basta então tomar a ferramenta e utilizá-la. Mas, se seu projeto tiver características especiais, e a ferramenta JACA não puder trabalhar com essas características, nem as outras ferramentas de Injeção de Falhas já disponíveis, o desenvolvedor pode partir para a seguinte abordagem. Primeiro, ele poderá verificar se, através de alterações da ferramenta JACA, será possível lidar com essas características especiais. A ferramenta foi projetada para ser facilmente alterável, e portanto pode-se adaptá-la para diversas situações. Se isso não for possível, por causa de necessidades muito específicas, o desenvolvedor poderá criar seu próprio programa que realize Injeção de Falhas usando os padrões para Injeção de Falha. Ele poderá criar um programa altamente especializado apenas para a tarefa em questão. Dessa maneira poderá ser um programa pequeno, tanto mais porque não precisará ser uma ferramenta genérica de Injeção de Falhas.

I.4. Contribuições

Este trabalho pretende dar as seguintes contribuições:

- Disponibilizar aos desenvolvedores que queiram criar suas ferramentas de Injeção de Falhas um sistema de padrões. Este poderá economizar muito tempo e esforço de desenvolvimento na criação dessas ferramentas.
- Criar uma nova ferramenta de Injeção de Falhas, a ferramenta JACA, que pode ser utilizada para testes de Injeção de Falhas em alto nível em sistemas Java. Também, pelo fato de essa ferramenta ter uma arquitetura altamente adaptável, a mesma pode ser modificada para trabalhar em diferentes situações onde seja necessário utilizar Injeção de Falhas.

I.5. Terminologia

Ainda não há um consenso sobre a terminologia a ser utilizada no campo de Tolerância a Falhas. Já ocorreram várias discussões a esse respeito nos Workshops e Simpósios feitos na área, porém um padrão ainda não emergiu. Para evitar enganos, então, nesta dissertação, estamos utilizando os termos abaixo com os seguintes sentidos, baseado em [Lei87] e [Lap95]:

- *Falha (*fault*):* é o componente, tanto de software como de hardware, que apresenta comportamento anômalo dentro do sistema.
- *Erro (*error*):* sendo encontrada uma falha dentro de um dado processamento do sistema, essa falha levará a uma modificação no estado do sistema. Esse estado causado pela falha é denominado *erro*.
- *Defeito (*failure*):* tendo a falha levado a um erro, esse erro por sua vez fará com que o sistema apresente um *defeito*. Um sistema apresenta um defeito quando o serviço que deveria oferecer não está de acordo com o que tinha sido especificado.
- *Validação:* a validação da segurança do funcionamento é um processo no qual verifica-se se foram retiradas as falhas do sistema e se avalia medidas de confiabilidade, disponibilidade, eficiência dos MTFs e etc.

I.6. Organização do texto

A presente dissertação está organizada da seguinte forma:

- O capítulo II ("Padrões de Software") mostra uma das bases teóricas deste trabalho, que são os padrões de software (ou *software patterns*, como são mais conhecidos).
- O capítulo III ("Ferramentas de Injeção de Falhas") mostra um dos trabalhos feitos neste projeto, que foi o estudo de ferramentas de Injeção de Falhas. No início do capítulo, explica-se o próprio conceito de Injeção de Falhas, já que esta é também uma das bases teóricas deste trabalho. A partir daí, se mostra as ferramentas de Injeção de Falhas por software que foram estudadas, e se faz uma comparação entre elas.
- O capítulo IV ("Sistema de Padrões para Injeção de Falhas por Software") expõe um dos resultados deste projeto, que é um sistema de padrões que visa facilitar o desenvolvimento de ferramentas de Injeção de Falhas por software.
- O capítulo V ("Ferramenta de Injeção de Falhas JACA") expõe outro resultado deste projeto, que é a ferramenta JACA. São mostradas a arquitetura da ferramenta e a estrutura dos subsistemas da mesma.
- O capítulo VI ("Testes da Ferramenta JACA") exibe os resultados experimentais obtidos nos testes da ferramenta JACA.
- O capítulo VII ("Conclusões e Trabalhos Futuros") discute os resultados obtidos neste projeto e sugere possíveis trabalhos futuros que poderiam ser feitos com base no Sistema de Padrões para Injeção de Falhas por Software ou com base na ferramenta JACA.
- O capítulo VIII ("Referências Bibliográficas") mostra as referências bibliográficas pesquisadas na execução deste trabalho.

CAPÍTULO II: PADRÕES DE SOFTWARE

II.1. Conceito

Padrões de software constituem um conceito que surgiu recentemente e tem sido considerado muito útil para o desenvolvimento de software. Trata-se do seguinte: ao se examinar programas dentro de certo domínio de aplicação, pode-se observar que eles tiveram problemas comuns no seu desenvolvimento, e que para esses problemas os desenvolvedores deram soluções semelhantes, que se mostraram mais eficientes. Essas soluções seriam, então, documentadas de uma forma definida e independente de linguagem de programação. Dessa forma, quando um desenvolvedor fosse criar um programa nesse mesmo domínio, em vez de ter que novamente formular a mesma solução, ele apenas teria que consultar um catálogo de padrões para encontrar um padrão que já lhe resolvesse o problema. Vale dizer que um padrão traz uma *sugestão* de solução para um problema: não é obrigatório adotar essa solução, e o desenvolvedor pode também modificá-la para se adaptar a circunstâncias específicas de seu projeto [GHJ+94, BMR+96].

Padrões são documentados de uma forma definida, e podem ser estruturados da seguinte forma. Primeiramente mostra-se o contexto onde o padrão pode ser empregado, e depois fala-se mais especificamente do problema que o padrão pretende resolver, discutindo-se também as *forças*, ou elementos do problema que devem ser balanceados numa possível solução. Em seguida mostra-se a solução proposta pelo padrão, e nas seções seguintes expõe-se a estrutura dessa solução e sua dinâmica de execução. Após isso fala-se como implementar o padrão num sistema. Por fim, mostram-se alguns usos conhecidos do padrão, algumas variantes do mesmo e discute-se as vantagens e problemas advindos do uso deste [BMR+96].

II.2. Tipos de padrões

Com relação à diferentes tipos de padrões, no presente trabalho consideraremos dois tipos, os *padrões de arquitetura* e os *padrões de projeto* [BMR+96]. Eles são explicados em detalhe nas sub-seções seguintes, e a tabela I sumariza as características deles.

II.2.1. Padrões de arquitetura

Os padrões de arquitetura são utilizados numa fase inicial do projeto, quando está se discutindo qual será a arquitetura do sistema sendo desenvolvido, e portanto lidam com um aspecto mais global do sistema. Eles pretendem resolver um problema inicial do desenvolvimento, que é como irá se arquitetar o sistema sendo criado. Um exemplo seria um padrão de arquitetura para Inteligência Artificial. Esse padrão sugeriria uma arquitetura para sistemas de Inteligência Artificial, que já teria sido utilizada em outros sistemas e que teria se mostrado mais eficiente ou mais vantajosa de acordo com algum critério.

II.2.2. Padrões de projeto

Os padrões de projeto são usados numa fase posterior à arquitetura global do projeto, para resolver problemas localizados dentro do mesmo. Por exemplo, numa determinada parte do projeto, há um problema de como dividir a realização de uma tarefa. Há um padrão de projeto denominado *master-slave* (mestre-escravo) que propõe uma solução: um objeto mestre divide a tarefa em sub-tarefas a serem realizadas por objetos escravos, sendo que depois consolida os resultados destes últimos. Padrões de projeto podem ser usados conjuntamente com padrões de arquitetura: um padrão de projeto pode sugerir uma estrutura que pode ser utilizada dentro de uma das partes de uma arquitetura formulada por um padrão de arquitetura.

Tabela I: tipos de padrões considerados

Tipo de Padrão	*Momento de Utilização*	*Escopo*	*Inter-relação*
Padrão de Arquitetura	No início do projeto, quando se precisa arquitetar o sistema	Todo o sistema sendo projetado	Define partes do sistema, onde podem ser usados padrões de projeto
Padrão de Projeto	Após o início do projeto, quando se precisa detalhar o mesmo	Partes localizadas do sistema	Pode ser utilizada em alguma parte do sistema definida por um Padrão de Arquitetura

II.3. Sistemas de padrões

Há um conceito relacionado com o de padrões que é o de *Sistema de Padrões*. Sistemas de padrões consistem em conjuntos de padrões relacionados, que resolvem problemas que ocorrem no desenvolvimento de sistemas dentro de um mesmo domínio de aplicação. Dentro do sistema de padrões, os padrões são organizados seguindo um dado esquema de classificação, de forma que um desenvolvedor possa rapidamente localizar o padrão que resolva o problema que ele tem em mãos. Dessa maneira, o desenvolvedor pode economizar muito tempo no desenvolvimento do sistema, já que conforme vão surgindo os problemas, basta ir consultando o sistema de padrões, verificando rapidamente se há um padrão para resolver um dado problema [BMR+96].

II.4. Resumo

Neste capítulo, mostrou-se um conceito que vem ganhando popularidade, os padrões de software, ou simplesmente padrões. Eles documentam soluções recorrentes para problemas que ocorrem com freqüência no desenvolvimento de software. E fazem isso de uma forma estruturada e independente de linguagem. Por isso estão ficando populares: quando o desenvolvedor encontra um problema, olha em um catálogo de padrões e pode achar lá uma solução pronta para ser implementada, em vez de ter que ele próprio formular essa solução.

Neste trabalho, iremos considerar dois tipos de padrões. O primeiro é o padrão de arquitetura, que descreve como arquitetar todo o sistema, e é empregado numa fase inicial do desenvolvimento. O segundo tipo é o padrão de projeto, que dá uma solução para um problema localizado, sendo usado numa fase posterior do desenvolvimento.

Outro conceito relacionado é o de sistema de padrões. Um sistema de padrões é um conjunto de padrões que dão soluções para problemas que ocorrem no desenvolvimento dentro de um domínio de aplicação. Assim, se têm sistemas de padrões para Inteligência Artificial, para Sistemas Distribuídos, etc. Com isso, um desenvolvedor tem acesso a várias soluções para problemas comuns no desenvolvimento, e portanto o uso de um sistema de padrões pode acelerar esse processo.

CAPÍTULO III: FERRAMENTAS DE INJEÇÃO DE FALHAS

Dentro do âmbito do Projeto, foi executado um amplo estudo das principais ferramentas de Injeção de Falhas existentes. Esse estudo é descrito a seguir:

III.1. Introdução

Atualmente sistemas computacionais têm um importante papel em nossa sociedade. Nós estamos cada vez mais dependentes deles, para realizar uma diversidade de tarefas. Dependemos deles também em algumas áreas críticas, onde um sistema computacional não pode ter um comportamento inesperado, senão ele poderia causar danos financeiros ou mesmo físicos.

Os sistemas que operam nessas áreas críticas são chamados *sistemas com segurança no funcionamento* (*dependable systems*). Uma das características que esses sistemas tem que ter é *Tolerância a Falhas*, que quer dizer que, mesmo na presença de falhas, esses sistemas têm que apresentar uma resposta esperada. Aqui, é necessário dar uma melhor definição de "falha". Um *defeito* do sistema ocorre quando o serviço requerido de um sistema desvia de executar a função do sistema, sendo essa última a funcionalidade original para a qual o sistema foi projetado. Um *erro* é a parte do estado do sistema que levou a um defeito; um erro afetando um serviço é uma indicação de que um defeito ocorreu ou vai ocorrer. A causa real ou hipotética de um erro, finalmente, é uma *falha*. Uma falha ativa pode ser tanto uma falha interna que não tinha sido atingida como uma falha externa. Um erro pode permanecer latente, isto é, sem ser reconhecido como tal, ou pode ser detectado por um Mecanismo de Tolerância a Falhas (MTF). Um erro também pode se propagar, criando novos erros. Quando um erro afeta o serviço realizado por um componente, um defeito de componente ocorre [Lap95].

Pode-se dizer que a importância de sistemas confiáveis tem crescido e, consequentemente, também cresceu a importância de testar esses sistemas. Assim, algumas técnicas de teste de sistemas confiáveis foram desenvolvidas, uma que tem se tornado mais popular é chamada *Injeção de Falhas*. Essa técnica opera através da injeção de falhas em um sistema e, em seguida, observa-se se o mesmo continua a operar como esperado. Dessa forma, essa abordagem pode dar maior confiabilidade sobre o funcionamento do sistema.

Numa época que uma falha de computador é cada vez menos tolerada, a confiabilidade que o teste por Injeção de Falhas pode dar é vital. O desenvolvedor pode ficar certo da correta operação de seu software, e portanto ele pode ser mais rapidamente testado

e entregue. Por causa disso muitos desenvolvedores dão grande atenção à Injeção de Falhas e uma de suas principais implementações, que é Injeção de Falhas por Software (Injeção de Falhas realizada por trechos especiais de código, o que é mais flexível e barato). Nas seções seguintes, serão mostrados Injeção de Falhas, Injeção de Falhas por Software e algumas das principais ferramentas que implementam Injeção de Falhas por Software. Isso será feito de maneira que o leitor poderá verificar o trabalho já realizado nessa área e a importância que muitos desenvolvedores estão dando a ela.

III.2. Abordagens de Injeção de Falhas

Situação atual. Nas últimas décadas, tem ocorrido um grande aumento no uso de sistemas computacionais. Esses sistemas têm sido empregados numa variedade de usos e ambientes, e alguns desses usos precisam que o sistema dê uma resposta esperada mesmo na presença de falhas. Como exemplo, um caixa eletrônico de banco não pode permitir que alguém realize saques ilimitadamente, mesmo se o sistema estiver com uma falha. Ele tem que dar uma resposta adequada à falha. Neste caso, poderia ser exibir uma mensagem dizendo que o caixa está inoperante. Sistemas como esse são cada vez mais necessários hoje em dia, e são chamados *sistemas tolerantes a falhas*.

Problemas. Portanto, desenvolvedores de software tem que providenciar tolerância a falhas nos sistemas que precisam dessa característica. Entretanto, não é suficiente apenas se projetar um sistema com tolerância a falhas. Para ter certeza que o sistema é realmente tolerante a falhas, é necessário testá-lo. Essa é uma atividade de grande importância, porque é nesse processo que irá se verificar se o sistema se comporta como esperado na presença de falhas. Como conseqüência, muitos pesquisadores têm procurado maneiras mais garantidas de validar esses sistemas.

Solução possível. *Injeção de Falhas* já tem sido usada para assegurar a confiabilidade de computadores tolerantes a falhas. De acordo com [ClP95] Injeção de Falhas vem sendo empregada desde 1970 pela indústria para medir cobertura (probabilidade de efetuar com sucesso detecção e recuperação de erro) e latência de Mecanismos de Tolerância a Falhas. Desde meados da década de 80, quando começou a ser examinada pelos meios acadêmicos, seu escopo tem se expandido. Injeção de Falhas vem sendo aplicada a produtos que não têm que ser tolerantes a falhas com a intenção de aumentar sua confiabilidade [Chi96] e quantificar riscos de software [VGK+97]. Injeção de Falhas pode ser aplicada a diferentes fases do ciclo de vida de um sistema. Diferentes técnicas podem ser utilizadas, as mais comuns sendo Injeção de falhas por simulação, por hardware e por software.

Injeção de Falhas por simulação. Esta técnica é usada na fase de projeto e as falhas são introduzidas num modelo do sistema alvo. Diferentes níveis de representação do sistema podem ser considerados: arquitetural, funcional, lógico e elétrico [ClP95]. Simulação de modo misto, onde o sistema é hierarquicamente decomposto para simulação em vários níveis, também é útil para Injeção de Falhas. Um grande número de ferramentas já foram desenvolvidas [GoI90, ClP92, JeA92, Chi92...], já que Injeção de Falhas por simulação é útil para avaliar dependabilidade do sistema nas primeiras fases do desenvolvimento. Simulação também tem a vantagem de proporcionar grande controle e observação das faltas injetadas quando comparado com injeção num protótipo ou sistema final, no qual o grande nível de integração entre dispositivos e tecnologias de empacotamento muito denso pode limitar a acessibilidade do modelo do sistema de maneira a permitir que os resultados obtidos possam ser representativos, e é difícil

construir tais modelos. Injetar falhas em um protótipo ou versão operacional de um sistema é útil em complemento de Injeção de Falhas por simulação. Nessa linha, falhas podem ser injetadas no hardware (falhas elétricas ou lógicas) ou no software (código ou dados).

Injeção de Falhas por hardware. Nesse método, falhas são originadas de algum hardware especial, projetado especialmente para esse propósito. Por exemplo, se um processador espera receber um sinal "um" em um pino específico, esse hardware especial iria injetar um sinal "zero" nesse pino, e o processador seria observado. Existem vários métodos para injetar falhas de hardware: pela alteração de níveis lógicos em pinos de circuitos integrados [AvR72, AAA+90, MFS91...], por irradiação de íons pesados [GKT89], pela alteração de níveis de suprimento de força [Cor87], entre outros. Esses métodos são adequados para verificar a eficácia de Mecanismos de Tolerância a Falhas implementados em hardware. Contudo, eles não são tão úteis para testar esses mecanismos quando eles são implementados em software.

Injeção de Falhas por software. Esta técnica procura injetar falhas no nível de software: sistemas operacionais e aplicações [HTI97]. Não é preciso hardware especial, e os testes podem ser mais facilmente controlados. Por causa dessas vantagens, Injeção de Falhas por software tem ficado mais popular entre os desenvolvedores de sistemas tolerantes a falhas. Diversas ferramentas foram desenvolvidas para esse propósito, e elas são expostas na sub-seção a seguir. Injeção de Falhas por software é o foco desta seção, então ela é exposta em mais detalhes nas sub-seções a seguir.

Comparação entre as técnicas. A tabela II resume as diferenças de cada técnica em termos de vantagens e desvantagens.

Tabela II: comparação entre diferentes abordagens para Injeção de Falhas

Abordagem	*Vantagens*	*Desvantagens*
Injeção de Falhas por simulação	• Permite um melhor controle do tempo, tipo de falha e do componente afetado no modelo. • Pode ser aplicada de início, na fase de projeto.	• Precisa de um longo tempo de processamento. • Requer parâmetros de entrada exatos. • O desenvolvimento de simuladores é caro e difícil.
Injeção de Falhas por hardware	• Usa hardware e software reais. • Produz falhas reais.	• Requer hardware especial, especificamente para injetar falhas, que é caro para desenvolver. • Pode danificar o sistema alvo. • É difícil injetar falhas e observar seus efeitos no nível do software. • Apresenta problemas de acessibilidade para componentes internos de hardware.
Injeção de Falhas por software	• Usa software real. • Não é preciso usar o hardware onde o sistema alvo irá realmente rodar. • O trabalho e o custo de desenvolvimento são baixos, já que não necessita de hardware especial. • É mais fácil de monitorar, como na simulação, mas não necessita do grande tempo de processamento desta última. • É portável. • É facilmente expansível para incluir novos tipos de falhas. • Interferências físicas não afetam o processo. • Não há risco de se danificar o sistema. • É adequada para validar sistemas que operam com altos níveis de abstração (falhas nesses níveis podem ser injetadas).	• Se o modelo de falhas não é bem definido e organizado, pode apresentar resultados que não são representativos. • Não tem acesso a recursos que estão escondidos do software. • Pode alterar o comportamento do sistema alvo, e, através disso, invalidar os resultados da Injeção de Falhas.

III.3. Técnicas de Injeção de Falhas por software

Esta sub-seção apresenta aspectos básicos de Injeção de Falhas por software, tal como modelos de falha considerados e as diferentes formas de injetar essas falhas.

III.3.1. Modelos de Falha

Injeção de Falhas por software pode ser usada para simular tanto falhas de software como falhas de sistemas externos ao software, mas conectados a este através de interfaces [VoM98, 1.4.4]. *Falhas internas* representam falhas de projeto e implementação, tais como variáveis ou parâmetros que estão errados ou não inicializados, atribuições incorretas ou checagens incorretas de condições. *Falhas externas* representam todos os fatores externos que não estão relacionados com falhas no código alvo mas alteram o estado do software. As falhas mais comuns nessa categoria são aquelas que representam conseqüências de falhas de hardware, como valores de registradores da CPU ou modificações em posições específicas da memória. Como estudos têm mostrado que falhas de software são uma causa

importante de defeitos de sistemas computacionais em operação, manifestações de falhas de software no nível da máquina ou mais alto têm sido levados em consideração.

Outra dimensão para a caracterização das falhas, especialmente aquelas representando conseqüências de falhas de hardware, é a base de repetição do padrão: falhas podem ser transientes (nunca se repetem), intermitentes (repetem-se periodicamente) ou permanentes (sempre se repetem).

III.3.2. Métodos de Injeção de Falhas

Diferentes métodos são usados para injetar falhas por software. Uma maneira de categorizar essas formas é definir quando as falhas são injetadas, se em tempo de compilação ou em tempo de execução [HTI97].

Na injeção *em tempo de compilação*, falhas são introduzidas no código fonte ou assembly do programa alvo, e consiste na alteração de instruções do programa. Na execução, quando uma instrução modificada é executada, a falha é ativada. Este método não precisa de software extra durante a execução para injetar falhas. Como resultado disso, não causa perturbação no sistema alvo durante a execução. No entanto, o modelo de falhas que pode ser injetado é limitado a falhas de software e conseqüências de falhas permanentes de hardware.

Em injetores *em tempo de execução*, código extra é necessário para injetar falhas e monitorar seus efeitos, respectivamente, um injetor de falhas e um monitor. Além disso, também é requerido um mecanismo para disparar a injeção de falhas.

Métodos comuns de injeção de falhas são os seguintes:

- Usando *modo traço* (*trace mode*), disponível em microprocessadores ou depuradores. Neste caso, o programa é executado passo a passo, e uma rotina de traço é executada antes de cada instrução do programa alvo. A rotina de traço roda em modo supervisor, e assim tem acesso a toda a memória do programa e todos os registradores visíveis ao programa, sendo adequado então à Injeção de Falhas.
- *Time out*, na qual um *timer* de hardware ou software é usado para gerar eventos de time-out. A Injeção de Falhas é disparada a cada time-out. Este método não requer modificação no programa alvo: Injeção de Falhas e monitorização podem ser implementadas como parte de uma rotina de interrupção.
- *Exception/Traps*, que é também baseada em interrupções causadas por exceções de hardware ou software traps. Neste caso, falhas são disparadas baseadas em eventos que não sejam time-outs. Por exemplo, sempre que uma certa instrução é executada ou uma determinada posição de memória é acessada.
- Uso de um *processo especial* que tem o privilégio de acessar o espaço de memória do programa alvo, sendo assim capaz de injetar falhas e observar seus efeitos.
- *Inserção de código*, a qual consiste em adicionar código ao programa alvo que permita que a Injeção de Falhas ocorra antes da execução de uma dada instrução. Diferente dos mecanismos anteriores, o código de Injeção de Falhas é parte do programa alvo e roda em modo usuário, e não em modo supervisor ou algum modo privilegiado.

A escolha do método mais adequado depende de vários fatores, entre eles: acessibilidade dos componentes nos quais as falhas serão injetadas, o nível aceitável de perturbação do programa alvo, recursos oferecidos pelo ambiente de execução e objetivo dos experimentos. Isso pode explicar porque a maioria das ferramentas combina vários mecanismos, como é mostrado na próxima seção.

III.4. Ferramentas de Injeção de Falhas

III.4.1. Generalidades

Muitas ferramentas de Injeção de Falhas por software já foram desenvolvidas, usando uma variedade de métodos para realizar Injeção de Falhas. O texto seguinte apresenta algumas dessas ferramentas com o objetivo de ilustrar cada um dos métodos de Injeção de Falhas descritos na sub-seção anterior. Para cada ferramenta os aspectos seguintes serão abordados:

- Método de Injeção de Falhas;
- Modelo de falhas;
- Discussão sobre a ferramenta;
- Sistemas alvo.

III.4.2. FIAT

Método de Injeção de Falhas: *FIAT (Fault Injection based Automated Testing environment)* [ClP95, Ros98] é uma ferramenta de Injeção de Falhas desenvolvida pela Universidade Carnegie-Mellon. Utiliza inserção de código para corromper bytes em imagens de memória de programas. O código extra pode ser inserido no nível de aplicação ou de sistema operacional. Falhas são disparadas quando determinadas posições de memória são atingidas na execução.

Modelo de falhas: FIAT procura avaliar a eficiência de detecção de erro e mecanismos de recuperação de aplicações distribuídas. Seu método permite injetar falhas em código e dados da aplicação do usuário, bem como em mensagens e timers.

Discussão: por usar código extra para injetar falhas, FIAT pode fazer Injeção de Falhas com baixa perturbação quando comparado com modo traço. Também, o sistema sob teste pode rodar a plena velocidade. Porém, a ferramenta aumenta o tamanho do programa na memória; se o sistema sob teste já for grande, é possível que ele não caiba mais na memória, ou então precisará de mais swapping, o que terá um impacto na velocidade de execução do sistema. Além disso, para mudar os valores dos registradores da CPU, FIAT tem que injetar falhas no próprio sistema operacional, o que pode levar a um "crash" do sistema operacional. Deve-se lembrar que o que está se testando é o sistema alvo, e não o sistema operacional sobre o qual ele está rodando.

Sistemas alvo: FIAT foi programado originalmente para testar sistemas distribuídos de tempo real.

III.4.3. FERRARI

Método de Injeção de Falhas: *FERRARI (Fault and ERRor Automatic Real-time Injection)* [ClP95, HTI97], desenvolvido na Universidade do Texas, usa traps de software para injetar falhas. Os traps são acionados tanto pelo "Program Counter" como por um timer. Quando o trap é ativado, uma rotina de tratamento de trap é chamada, e ela injeta falhas numa posição específica, como, por exemplo, alterando o valor de um registrador ou de uma posição de memória.

Modelo de falhas: a ferramenta pode simular erros de processador, memória e de barramento. Também, as falhas injetadas podem ser permanentes ou transientes. FERRARI também usa um método chamado corrupção de memória, no qual falhas são injetadas numa imagem da memória da tarefa antes da execução começar.

Discussão: o mecanismo de trap é muito comum em vários sistemas, e dessa forma a ferramenta pode ser portada para muitos ambientes operacionais. Também, usando traps, não é preciso inserir código no código original do sistema sob teste. Entretanto, o sistema alvo não roda a plena velocidade: de fato o mecanismo de trap, quando ativado, muda o fluxo de execução do programa; fazendo assim, ele causa uma perturbação no sistema. Mais ainda, a ferramenta tem que rodar em modo de sistema, não em modo de usuário.

Sistemas alvo: FERRARI foi projetado sem um tipo específico de sistema alvo em mente.

III.4.4. Ftape

Método de Injeção de Falhas: *Ftape (Fault tolerance and performance evaluator)* [HTI97], projetado pela Universidade de Illinois, realiza Injeção de Falhas através da adição de drivers ao sistema operacional sobre o qual o sistema sob teste está rodando. A ferramenta coloca alguns drivers especialmente projetados no sistema operacional, e esses drivers irão injetar as falhas, mudando os valores de registradores da CPU, posições de memória e valores lidos do subsistema de disco.

Modelo de falhas: o tipo de falhas injetadas depende da acessibilidade aos recursos do sistema que um driver tem dentro de um dado sistema operacional. No sistema que Ftape foi originalmente desenvolvido, um computador tolerante a falhas Tandem, a ferramenta era capaz de mudar os valores de registradores da CPU acessíveis pelo usuário, de posições de memória e de valores lidos pelo subsistema de disco, e portanto simular falhas nesses componentes. Para injetar as falhas, Ftape usa uma estratégia baseada na carga de trabalho de cada componente: o componente que tem a maior carga de trabalho, por exemplo, será o próximo onde uma falha será injetada.

Discussão: a ferramenta não precisa adicionar código ao sistema sob teste. Igualmente, o sistema pode rodar a sua plena velocidade, resultando que a perturbação é mínima. Mas, como a ferramenta depende dos drivers do sistema operacional, ela é amarrada a um sistema operacional específico, e tem que passar por uma reescrita quase completa para rodar em outro sistema operacional. Também, diferentes sistemas operacionais dão diferentes graus de liberdade para seus drivers: isso significa que Ftape pode não conseguir injetar alguns tipos de falha em todos os sistemas.

Sistemas alvo: a ferramenta não foi desenvolvida para ser usada com um tipo específico de software.

III.4.5. Doctor

Método de Injeção de Falhas: *Doctor* (*Integrated Software Fault Injection Environment*) [HTI97], criado pela Universidade de Michigan, combina diversos métodos para injetar falhas. Ele usa time-outs, traps e alteração de código para fazer injeção de falhas. Para simular falhas de memória, Doctor usa time-out: ele programa um timer, e quando este expira dispara uma interrupção, previamente programa no vetor de tratadores de interrupção do sistema. Então, a ferramenta é capaz de mudar o valor da posição de memória. Traps são usados para simular falhas de processador, numa base não-permanente. Por exemplo, fazendo uma simples alteração da instrução a ser executada. Para simular falhas permanentes da CPU, o código do sistema sob teste é modificado, de maneira a mudar permanentemente a execução de algumas instruções ou dados do sistema.

Modelo de falhas: Doctor, usando time-outs, traps e modificação de código, pode simular a presença de falhas relacionas à CPU (numa base permanente ou não-permanente), à memória e à comunicação de rede (mudando os dados recebidos pelo sistema). Originalmente, Doctor foi usado na pesquisa do efeito da perda de mensagens no Harts, um sistema distribuído de tempo real.

Discussão: time-outs e traps são recursos disponíveis em quase todos os sistemas, e dessa maneira a ferramenta pode ser reescrita para rodar em muitos ambientes diferentes. Também, por usar uma variedade de métodos para injetar falhas, Doctor pode emular mais proximamente tipos diversos de falhas. Doctor também é capaz de simular falhas permanentes ou não-permanentes. Contudo, pelo próprio fato de usar time-outs (que causam interrupções do programa), traps e mudança de código do sistema, todos os quais causam perturbação no fluxo de execução do sistema, a execução do sistema sob teste pode ficar bem diferente da que seria rodando em condições reais. Igualmente, o uso de todos esses recursos pode não ser fácil do ponto de vista do usuário da ferramenta.

Sistemas alvo: Doctor foi projetado originalmente para trabalhar com sistemas distribuídos.

III.4.6. ProFI

Método de Injeção de Falhas: *ProFI* (*Processor Fault Injection*) [LoE93] foi criado pela Universidade de Essen e usa o mecanismo de traço disponível na família de processadores Motorola 68000 para injetar falhas. Ele usa traço para executar uma instrução do sistema de cada vez, podendo então injetar as falhas. Quando a ferramenta atinge uma instrução onde uma falha será injetada, ela pode modificar os dados da instrução, os registradores e os flags do processador, o resultado de alguma computação ou o "Program Counter". Também pode modificar a instrução que está sendo executada. Assim, ProFI pode simular falhas de processador. As falhas a serem injetadas pode ser especificadas utilizando uma linguagem de script especialmente projetada chamada ProFIL.

Modelo de falhas: ProFI é capaz de injetar, permanentemente ou transientemente, praticamente todo tipo de falha relacionado com o processador. Ela pode

mudar o valor dos registradores, dos flags, dos resultados da computação, do "Program Counter" e até mudar a própria instrução a ser executada.

Discussão: pelo fato de usar uma linguagem de script para descrever as falhas a serem injetadas, a ferramenta pode automatizar o processo de teste usando Injeção de Falhas. Mais ainda, a ferramenta pode simular falhas muito específicas relacionadas com o processador, como por exemplo falhas do registrador de status do processador Motorola 68000. Isso acontece porque desde o seu desenvolvimento, ProFI foi orientada para trabalhar com falhas de processador, como seu próprio nome diz. Em compensação, a ferramenta só pode trabalhar com esse tipo de falha. Falhas de memória, de subsistema de disco, de troca de mensagens tem que ser injetadas por outra ferramenta de Injeção de Falhas. Outra desvantagem é causada pela necessidade de se usar a linguagem de script ProFIL para selecionar as falhas. Por exemplo, depois de observar o resultado de uma injeção de falha, o desenvolvedor pode querer injetar mais uma falha, relacionada com essa primeira; usando ProFI, ele teria que escrever outro script apenas para injetar essa falha extra.

Sistemas alvo: ProFI foi desenvolvida procurando testar sistemas tolerantes a falha em geral, não se considerou trabalhar com algum tipo específico de sistema.

III.4.7. FINE e DEFINE

Método de Injeção de Falhas: *FINE* (*Fault Injection and Monitoring Environment*) e seu sucessor, *DEFINE* (*Distributed Fault Injection and Monitoring Environment*) [KIT93] foram desenvolvidos pela Universidade de Illinois em Urbana-Champaign e objetivam realizar Injeção de Falhas em sistemas baseados no sistema operacional Unix. As falhas são injetadas diretamente no núcleo do sistema operacional, e elas são monitoradas para observar as conseqüências das falhas no sistema. As ferramentas precisam modificar o núcleo do sistema Unix para fazer isso. Mas, usando essa abordagem, elas são capazaes de simular falhas de memória, CPU, barramento, Entrada/Saída e até falhas de software, já que todos esses recursos são acessáveis pelo núcleo do sistema operacional. DEFINE é uma evolução de FINE, na qual a ferramenta também é capaz de injetar falhas de comunicação em sistemas distribuídos.

Modelo de falhas: as ferramentas podem simular falhas de memória, de CPU, de barramento e de Entrada/Saída. Também, é possível simular as conseqüências de falhas de software. DEFINE, além disso, pode injetar falhas relacionadas com a comunicação em um sistema distribuído. As falhas injetadas podem ser tanto permanentes como transientes.

Discussão: já que as ferramentas dependem do núcleo do Unix para injetar as falhas, e como o núcleo controla todos os recursos do sistema, elas conseguem acessar todos esses recursos, e portanto podem injetar falhas relacionadas a todos esses recursos. E, já que o injetor de falhas reside no núcleo, o sistema sob teste roda a plena velocidade e sua perturbação é muito baixa. Como uma desvantagem, FINE e DEFINE necessitam do código fonte do núcleo do Unix: eles precisam modificá-lo. Também é difícil injetar falhas no nível de aplicação, já que o injetor de falhas fica no nível do sistema operacional.

Sistemas alvo: FINE e DEFINE foram desenvolvidos para lidar com sistemas alvo rodando sobre o sistema operacional Unix. No caso de DEFINE, também, a ferramenta foi projetada para trabalhar com sistemas distribuídos.

III.4.8. SOFIT

Método de Injeção de Falhas: *SOFIT (Software Object-oriented Fault Injection Tool)* [AvT95] é uma ferramenta desenvolvida na Universidade A&M do Texas, para validar um sistema de software na presença de falhas de hardware. A ferramenta roda sobre o sistema operacional Solaris e utiliza seu recurso de *proc*, o qual dá um grande controle sobre um processo. Para injetar a falha, SOFIT, usando o recurso de *proc*, pára o processo no qual a falha será injetada, e muda alguns dos valores que o processo está lidando (para simular falhas de memória, de registradores da CPU e de barramento). Se a falha é brevemente transiente, a ferramenta deixa o processo avançar uma instrução e então recupera os valores originais que tinham sido mudados; se é longamente transiente, isso é feito posteriormente.

Modelo de falhas: a ferramenta é capaz de injetar falhas de memória, de registradores da CPU, de barramento de dados e endereço, através da modificação de valores relacionados a esses recursos no processo. As falhas podem ser brevemente transientes (ou melhor, instantaneamente transientes) ou longamente transientes. SOFIT, dessa forma, lida somente com falhas de hardware.

Discussão: pelo fato de usar o recurso de *proc* do sistema operacional Solaris, SOFIT pode ter um grande controle do processo no sistema alvo. Assim, ele pode monitorar muito proximamente esses processos, sem adicionar código extra a eles. Igualmente, ele não precisa do código fonte do sistema sob teste. Mas, porque precisa parar o processo para fazer a Injeção de Falhas, pode-se verificar que o processo será interrompido várias vezes, e portanto não rodará a plena velocidade. Isso causa perturbação na execução do sistema alvo.

Sistemas alvo: SOFIT precisa do recurso de *proc* disponível no sistema operacional Solaris, logo só pode testar sistemas rodando sobre aquele sistema operacional.

III.4.9. ORCHESTRA

Método de Injeção de Falhas: *ORCHESTRA* [DJM97] é uma ferramenta que pretende usar Injeção de Falhas para testar implementações de protocolos distribuídos. Foi desenvolvido pela Universidade de Michigan. A ferramenta considera um protocolo como sendo uma pilha de diferentes camadas, cada camada independente das outras. As mensagens trocadas entre nós num sistema distribuído são geradas e passadas para a camada mais acima, que por sua vez as passa para a camada imediatamente abaixo, e assim vai até que as mensagens chegam a camada mais baixa e são transportadas para o nó de destino. Nesse nó, as mensagens chegam na camada mais baixa e o processo é revertido, até que as mensagens chegam na camada mais acima e são passadas à aplicação. ORCHESTRA, então, insere uma nova camada numa pilha já existente de camadas, chamada camada PFI (Protocol Fault Injection – Injeção de Falhas de Protocolo). O propósito dessa camada é modificar, apagar ou adicionar mensagens, e, fazendo assim, injetar falhas e testar se o protocolo distribuído pode lidar com essas falhas.

Modelo de falhas: como já foi dito, ORCHESTRA é capaz de injetar apenas falhas relacionadas com a transmissão de mensagens. A ferramenta pode modificar o conteúdo da mensagem, atrasar uma mensagem, duplicar uma mensagem, remover uma mensagem e até mesmo criar uma nova mensagem. Dessa maneira, a ferramenta de Injeção de Falhas

pode simular diversas falhas possíveis de ocorrerem num sistema distribuído. Ela pode fazer isso numa base permanente ou transiente.

Discussão: ORCHESTRA causa quase nenhuma perturbação: já que as camadas são independentes, o que uma delas faz não afeta o que as outras fazem nem a aplicação no topo do protocolo. Igualmente, essa abordagem significa que o sistema pode rodar a plena velocidade. Mais ainda, a ferramenta pode facilmente ser adaptada para testar outros protocolos distribuídos. A principal desvantagem é que a ferramenta trabalha somente com sistemas baseados em protocolos: não pode trabalhar com outros tipos de sistema. Relacionado a isso, também, é o fato que ORCHESTRA pode apenas injetar falhas relacionadas com a transmissão de mensagens por sistemas distribuídos. Falhas na aplicação que usa o protocolo não são cobertas pela ferramenta.

Sistemas alvo: a ferramenta pretende testar sistemas distribuídos com protocolos baseados numa estrutura de pilha.

III.4.10. Xception

Método de Injeção de Falhas: *Xception* (*Software Fault Injection and Monitoring in Processor Functional Units*) [CMS95], desenvolvido pela Universidade de Coimbra, Portugal, usa para fazer Injeção de Falhas algumas características especiais dos processadores mais recentes. Esses processadores têm algumas instruções especiais destinadas a realizar depuração e testes de performance dos chips. Xception usa essas instruções para simular falhas de hardware no sistema sob teste. Falhas são assim disparadas pela ocorrência de algum evento específico, causando uma exceção de hardware. Por exemplo, quando o processador atinge uma dada instrução no código do sistema, a exceção pode desviar a execução para um ponto aleatoriamente escolhido do código, simulando assim uma falha do ponteiro de instrução ou do barramento de código.

Modelo de falhas: a ferramenta objetiva emular falhas de hardware, como posições de memórias presas em zero, falhas de barramento de endereço, falhas de ponteiro de instrução e outras falhas relacionadas de hardware. As falhas podem ser ativadas tanto pela ocorrência de eventos pré-selecionados como por time-outs.

Discussão: o programa sob teste pode rodar a plena velocidade (não é utilizado modo traço) e não é necessário nenhum código extra para fazer a Injeção de Falhas e a monitorização, assim a perturbação causada é desprezível. Um problema, todavia, dessa ferramenta é sua dependência das instruções especiais de depuração e medição de performance dos modernos processadores. Por causa disso, a ferramenta tem que ser reescrita para cada novo processador que for trabalhar. Além disso, apesar dos autores corretamente apontarem que todos os modernos processadores têm tais instruções, sua disponibilidade de uso depende de um acordo com os fabricantes do chip. Isso porque essas instruções não são divulgadas, então é necessário se fazer um acordo com o fabricante para se conseguir documentação sobre elas. Outra desvantagem é que a ferramenta é adequada para injetar falhas de hardware, mas é muito difícil simular falhas de software com ela.

Sistemas alvo: Xception procura testar sistemas baseados nos processadores mais recentes, tais como IBM/Motorola PowerPC, Intel Pentium, HP PA-RISC, Sun SPARC e outros. Na sua versão original, a ferramenta trabalhava com o processador PowerPC.

III.4.11. FIESTA

Método de Injeção de Falhas: *FIESTA* (*Fault Injection for Embedded System Target Application*) [KJA98], desenvolvido na Universidade do Texas em Austin, é uma ferramenta de Injeção de Falhas projetada para o sistema operacional comercial de tempo real VxWorks. A ferramenta depende de um depurador, uma versão modificada do já conhecido depurador gdb, para injetar as falhas. Por exemplo, ele pode localizar uma instrução específica, e colocar um "breakpoint" naquela instrução. Depois que ela é executada *n* vezes (onde *n* pode ser especificado pelo usuário), o depurador modifica a instrução, executa-a, recoloca a instrução original e remove o "breakpoint". Dessa forma, FIESTA pode simular muitas falhas relacionadas com o hardware, utilizando um software já desenvolvido anteriormente e testado, no caso, o depurador gdb.

Modelo de falhas: quando a ferramenta atinge a instrução específica a ser modificada, ela pode mudar os dados da instrução ou a própria instrução. Fazendo isso, ela pode simular falhas dos registradores da CPU, falhas de memória, e falhas de barramento de dados ou endereço. Na sua versão atual, FIESTA trabalha com as seguintes falhas: falhas de linha de endereço quando se está buscando uma instrução, falhas transientes de registradores e mutação aleatória de código executável.

Discussão: a ferramenta se baseia em software já desenvolvido e testado, que é a versão modificada do depurador gdb, que é amplamente utilizado por desenvolvedores Unix. Isso dá robustez e confiabilidade a ferramenta. Também é possível, utilizando os recursos do depurador, um controle maior do sistema sob teste. A desvantagem primária da ferramenta é que ela não roda a plena velocidade. Como foi dito, se o usuário escolher injetar uma falha na décima quinta vez que se atinge uma instrução, o sistema sob teste irá parar quinze vezes no breakpoint naquela instrução. Assim, o tempo de execução do sistema sob teste irá diferir consideravelmente do tempo de execução sob reais condições. Ainda, para se usar o depurador, algum código do depurador tem que ser associado ao sistema alvo, o que causa alguma perturbação.

Sistemas alvo: a ferramenta não foi projetada objetivando-se algum tipo específico de sistema alvo.

III.4.12. IPA

Método de Injeção de Falhas: *IPA* (*Interface Propagation Analisys*) [Voa98] é uma ferramenta criada pela Reliable Software Technologies Corporation e pretende fazer Injeção de Falhas em sistemas baseados em componentes de software de "prateleira" (Components Off-The-Shelf – COTS) ou, mais simplesmente, componentes. Componentes se comunicam através de interfaces; um componente propaga seu estado para outra componente através de uma interface. De acordo com isso, IPA se localiza entre dois componentes que estão se comunicando, e irá corromper o estado que está sendo propagado de um para outro componente. Nesse momento, ela irá observar se este último componente pode dar uma resposta esperada apesar do estado corrompido. Fazendo assim, a ferramenta pode simular a presença de falhas em algum dos componentes do sistema.

Modelo de falhas: as falhas que IPA pode injetar são todas relacionadas com a propagação de estado entre componentes. Como um exemplo, um componente passa uma mensagem para um segundo componente dizendo que está *pronto*; o injetor de falhas pode mudar a informação de estado para *não-pronto* e aí se observará como o segundo componente irá reagir a esse estado corrompido.

Discussão: o injetor de falhas, localizado entre dois componentes, é considerado ele próprio um componente, e assim ele não afetará o resto do sistema sob teste (componentes devem trabalhar independentemente uns dos outros). Além disso, IPA é altamente controlável: o injetor de falhas pode alterar com precisão a informação de estado recebida de um componente para corromper essa informação de alguma forma. Mas, colocar um componente entre dois outros diminui a velocidade de comunicação entre os dois: assim, a injeção de falhas que dependam do tempo da comunicação não é possível. Mais ainda, para colocar o injetor de falhas entre os componentes, o código fonte dos componentes é necessário: IPA não consegue injetar falhas dispondo apenas do executável do sistema.

Sistemas alvo: IPA foi projetado para lidar com sistemas baseados em componentes (COTS).

III.4.13. FIRE

Método de Injeção de Falhas: *FIRE* (*Fault Injection using a REflective architecture*) [Ros98], criado na Universidade Estadual de Campinas, Brasil, se baseia em *reflexão* (também conhecido como *programação reflexiva*) para injetar falhas no sistema sob teste. Reflexão estabelece que, além dos objetos normais de um sistema orientado a objeto, chamados *objetos do nível base*, existem também os *meta-objetos*, que podem modificar o comportamento dos objetos do nível base. Essa modificação do comportamento dos objetos do nível base, feita pelos meta-objetos, é definida e controlada por um *protocolo de meta-objeto* (*Meta-Object Protocol – MOP*) que define que meta-objetos podem ser associados a que objetos do nível base, e o que o meta-objeto pode observar e mudar do objeto do nível base. FIRE usa esse mecanismo para mudar o valor de alguns atributos, e também para mudar o valor de retorno de alguns dos métodos dos objetos do nível base. Dessa maneira, a ferramenta simula falhas nos objetos. Por exemplo, considere um método no objeto do nível base que retorna o quadrado de algum atributo do objeto. O meta-objeto associado pode mudar o valor de retorno para um valor negativo, e então ele simula uma falha no cálculo do quadrado ou na armazenagem do atributo.

Modelo de falhas: FIRE poderia injetar as conseqüências de falhas de hardware. É possível que, por exemplo, um bit preso em zero numa posição de memória utilizada para armazenar o valor de um atributo faça o valor desse atributo ser errôneo. Nesse sentido, pode ser que por trocar os valores dos atributos e dos retornos de métodos algumas falhas de hardware possam ser simuladas. No entanto, FIRE objetiva muito mais simular as conseqüências de falhas de software. Como um exemplo, mudando-se o valor de retorno de um método, a ferramenta simula que esse método tem uma falha que o faz produzir resultados errôneos. FIRE pode injetar falhas transientes ou intermitentes.

Discussão: FIRE pode rodar o sistema sob teste a sua plena velocidade, não havendo necessidade de se usar modo traço. Embora ele utilize o método de inserção de código para implementar a Injeção de Falhas e a monitorização, esse código extra é parte do meta-nível, sendo assim completamente independente das funcionalidades da aplicação alvo. Esse método também apresenta grande flexibilidade com respeito ao modelo de falhas, permitindo a injeção tanto de falhas de hardware como de software. E, sendo totalmente baseada em objetos, a ferramenta é bem adequada para testar aplicações orientadas a objeto. E, FIRE é a única ferramenta entre as listadas aqui que é capaz de injetar falhas apenas na aplicação e não no sistema inteiro: assim, ela pode trabalhar no

processo de teste especificamente da aplicação e não só do sistema onde a aplicação reside. Entretanto, FIRE causa alguma perturbação no sistema sob teste: a adição de meta-objetos no sistema não modifica o código fonte, mas o código compilado será diferente do original, já que ele terá que ter código para lidar com os meta-objetos. E, para adicionar os meta-objetos ao sistema alvo, este último tem que ser recompilado, o que significa que o código fonte tem que estar disponível. Igualmente, a ferramenta foi escrita em OpenC++, que é a versão de C++ que trabalha com meta-objetos. Como resultado, FIRE trabalha somente com sistemas escritos em C++. Mais ainda, a ferramenta aumenta o tamanho do programa, já que ele adiciona código extra necessário para os meta-objetos.

Sistemas alvo: FIRE foi desenvolvido para testar sistemas orientados a objeto.

III.4.14. ComFIRM

Método de Injeção de Falhas: *ComFIRM* (*Communication Fault Injection through OS Resources Modification*) [BLW99], desenvolvido na Universidade Federal do Rio Grande do Sul, Brasil, modifica o sistema operacional sobre o qual o sistema sob teste está rodando para injetar falhas. A ferramenta foi desenvolvida para ser usada com o sistema operacional Linux, que é um software "open-source", e assim os desenvolvedores podem reescrever algumas partes do Linux. Dessa forma, eles conseguiram colocar algumas partes da ComFIRM dentro do sistema operacional, e a ferramenta é então capaz de injetar falhas de dentro do sistema operacional. A ferramenta pretende testar mecanismos de tolerância a falhas relacionados com a comunicação dentro de um sistema distribuído, e ela o faz através da troca ou da remoção de algumas mensagens trocadas entre nós do sistema distribuído. Quando um módulo, em algum nó do sistema distribuído, chama uma função do sistema operacional para transmitir a mensagem, essa função dentro do sistema operacional terá sido trocada por uma função do ComFIRM que irá alterar ou remover a mensagem a ser transmitida e em seguida irá transmitir a mensagem. Logo, ComFIRM pode injetar falhas de comunicação.

Modelo de falhas: ComFIRM pretende testar mecanismos de comunicação tolerante a falhas. Esse fato implica que somente podem ser injetadas falhas relacionadas com a transmissão de mensagens dentro de um sistema distribuído. A ferramenta é capaz de modificar o conteúdo das mensagens, ou de simplesmente removê-las, e pode injetar falhas permanentes, transientes ou bizantinas.

Discussão: como ComFIRM injeta falhas a partir do sistema operacional, ele é capaz de fazê-lo com baixa perturbação do sistema sob teste, que roda a plena velocidade de execução. Igualmente, ComFIRM tem capacidades notáveis de lidar com o conteúdo de mensagens trocadas dentro do sistema distribuído, e é capaz de fazer mais do que simplesmente remover ou mudar a ordem das mensagens. A ferramenta, no entanto, depende do sistema operacional Linux, e então só pode ser usada com sistemas alvo que rodem sobre esse sistema operacional. Igualmente, a portabilidade da ferramenta depende da portabilidade do Linux: ela só pode ser reescrita para rodar num ambiente computacional que disponha de uma versão do Linux. Mais ainda, a ferramenta só lida com falhas relacionadas à transmissão de mensagens; outras falhas não são cobertas como, por exemplo, se um nó do sistema distribuído tem um "crash" total.

Sistemas alvo: ComFIRM visa testar sistemas distribuídos rodando sobre o sistema operacional Linux.

III.5. Comparação entre ferramentas

Cada uma das ferramentas acima tem suas vantagens e desvantagens, como já foi exposto na sub-seção anterior. Entretanto, é importante compará-las baseado em algumas características comuns. A tabela III resume uma comparação entre as ferramentas considerando-se os seguintes aspectos:

- *Perturbação:* indica o nível de perturbação que cada ferramenta causa no nível de aplicação. Todas elas tentam manter a perturbação mais baixa quanto possível, mas algumas conseguem realizar isso melhor que outras.
- *Portabilidade:* expressa quão facilmente uma ferramenta, escrita para um dado sistema, pode ser reescrita para outro sistema. Para algumas ferramentas isso pode ser muito difícil; para outras isso é muito fácil, requerendo apenas umas poucas mudanças na ferramenta.
- *Tipos de falhas:* essa coluna na tabela diz que tipos de falhas podem ser injetadas por uma dada ferramenta, isto é, o modelo de falhas da ferramenta.
- *Sistemas alvo:* diz para que tipo de sistema especificamente a ferramenta foi projetada para testar. Como um exemplo, ORCHESTRA foi projetada para fazer Injeção de Falhas em sistemas distribuídos. Quando esta coluna diz "qualquer sistema", isso quer dizer que a ferramenta foi projetada sem ter um tipo específico de sistema a ser testado em mente.

Tabela III: comparação entre ferramentas de Injeção de Falhas

Ferramenta	*Pertur-bação*	*Portabilidade*	*Tipos de Falhas*	*Sistemas Alvo*
FIAT	baixa	fácil	falhas de memória, comunicação e registradores da CPU	qualquer sistema (originalmente sistemas distribuídos de tempo real)
FERRARI	baixa	fácil, mas o sistema hospedeiro tem que dispor de traps	falhas de CPU, memória e barramento	qualquer sistema
Ftape	quase nenhuma	média; tem que reescrever drivers do sistema operacional	tipicamente falhas de registradores do usuário da CPU, memória e Entrada/Saída (**)	qualquer sistema
Doctor	baixa à média	fácil, mas o sistema hospedeiro tem que ter traps e time-outs	falhas de CPU, memória e rede	qualquer sistema (originalmente sistemas distribuídos)
ProFI	baixa à média	fácil, mas o sistema hospedeiro tem que ter modo traço	falhas de CPU	qualquer sistema
FINE e DEFINE	muito baixa	fácil à média; tem que reescrever parte do núcleo do sistema Unix	falhas de CPU, memória, barramento, Entrada/Saída e de software (DEFINE: falhas de rede também)	sistemas baseados em Unix (DEFINE: sistemas distribuídos baseados em Unix
SOFIT	baixa à média	fácil, mas tem que rodar sobre o sistema operacional Solaris	falhas de registradores da CPU, memória, barramento de dados e endereço	sistemas baseados no sistema operacional Solaris
ORCHESTRA	quase nenhuma	fácil	falhas de transmissão de mensagem em um sistema distribuído	sistemas distribuídos
Xception	quase nenhuma	difícil (*)	falhas de CPU, memória e barramento	sistemas baseados em processadores modernos
FIESTA	baixa	depende da presença de um depurador no sistema hospedeiro	falhas de registradores da CPU, memória, barramento de dados e endereço	qualquer sistema
IPA	baixa à média	fácil	falhas de estados corrompidos	sistemas baseados em componentes (COTS)
FIRE	média à alta	fácil	falhas de software	sistemas orientados a objeto
ComFIRM	baixa	média; depende da portabilidade do sistema operacional Linux	falhas de transmissão de mensagem em um sistema distribuído	sistemas distribuídos baseados no sistema operacional Linux

(*) - A portabilidade do Xception é complexa porque a ferramenta depende de instruções especiais do processador (como descrito na sub-seção sobre o Xception). Portanto, os desenvolvedores terão que fazer um acordo com o fabricante para conseguir a documentação sobre essas instruções.

(**) - Ftape pode "tipicamente" injetar falhas de registradores do usuário da CPU, de memória e de Entrada/Saída porque isso depende de que recursos do sistema um driver pode acessar dentro de um dado sistema operacional. Que tipo de falhas Ftape poderá injetar irá depender do grau de liberdade que um sistema operacional dá aos seus drivers.

III.6. Arquitetura genérica para Injeção de Falhas

Já se percebeu que, a partir do estudo das ferramentas de Injeção de Falhas já desenvolvidas, se poderia formular uma arquitetura genérica para Injeção de Falhas. Em [HTI97], tal arquitetura é descrita. Ela está no diagrama da *fig. 1.*

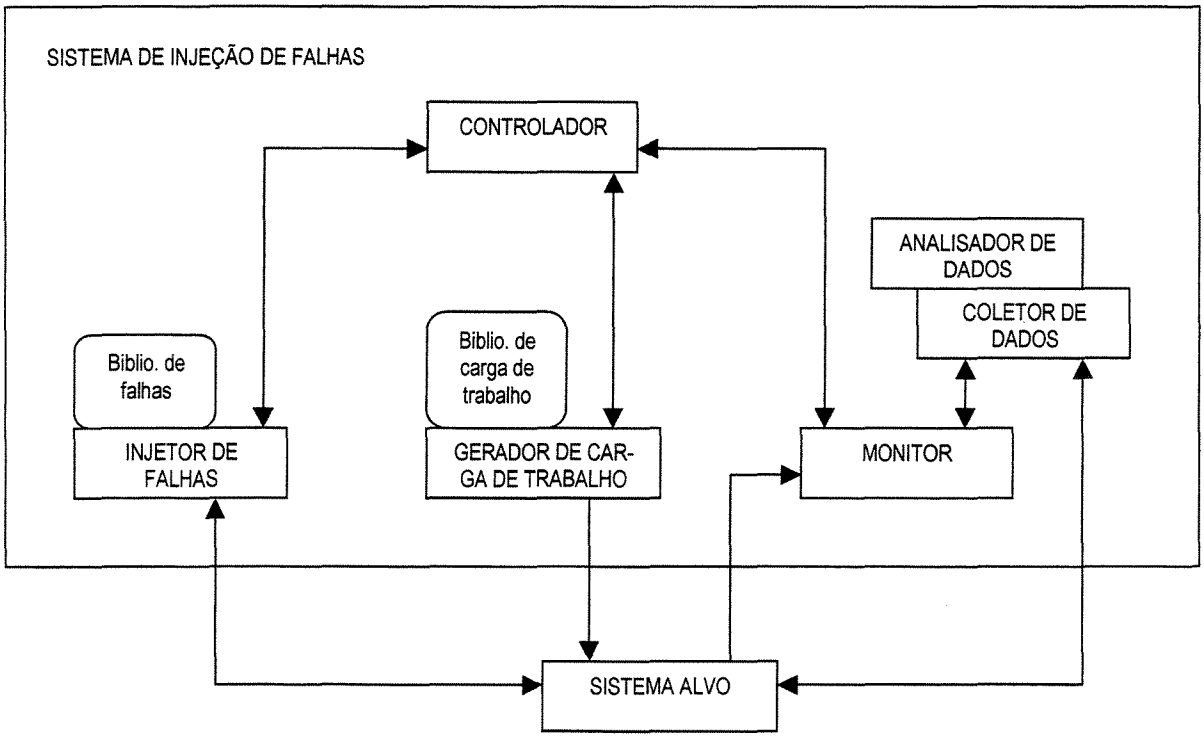


Fig. 1: diagrama de uma arquitetura genérica para Injeção de Falhas, adaptado a partir de [HTI97]

No diagrama acima, tem-se um injetor de falhas, que tenta simular a presença de falhas no sistema sob teste. As falhas estão armazenadas na biblioteca de falhas. Há também um monitor, que observa o comportamento do sistema alvo, bem como um gerador de carga de trabalho, que irá passar comandos para o sistema sob teste executar, de acordo com o que está na biblioteca de carga de trabalho. Dados sobre o sistema são recolhidos pelo coletor de dados, e posteriormente podem ser analisados pelo analisador de dados. Por fim, o controlador coordena a execução de todo o sistema. Em [HTI97], se descreve a arquitetura, mas não se discute como seria uma dinâmica de execução dessa arquitetura, nem como esses diferentes componentes se relacionariam. Essas informações adicionais, entretanto, constariam de um padrão de arquitetura, como pode se ver no padrão de arquitetura "Injetor de Falhas" (vide seção IV.2).

III.7. Conclusões

Pode-se perceber que, após ler o material precedente, que muito trabalho já foi feito em Injeção de Falhas por software, e que essa técnica já atingiu sua maturidade. Atualmente, ela pode dar ajuda valiosa na avaliação da dependabilidade de um sistema, com relação ao aspecto de como um dado sistema sob teste irá reagir na presença de falhas.

Como já foi dito antes, Injeção de Falhas por software não é a única técnica para injetar falhas durante a fase de teste: há também Injeção de Falhas por hardware. Esta última abordagem tem a vantagem de introduzir falhas reais no sistema alvo. Contudo, ela apresenta algumas limitações, tal como que o experimento é escassamente controlável; pode danificar o sistema alvo; é difícil de observar o comportamento dos componentes sendo testados; e também que alguns circuitos internos são difíceis de acessar. E, finalmente, Injeção de Falhas por hardware precisa de circuitos especialmente projetados, que são caros. Comparado a isso, Injeção de Falhas por software demonstra ser uma abordagem muito mais versátil, controlável e barata. Contudo, uma desvantagem de Injeção de Falhas por software, quando comparada com Injeção de Falhas por hardware, é que ela apenas simula falhas, não injeta falhas reais. Mas, dadas as vantagens da abordagem por software, esta última ainda é uma melhor opção.

Deve-se notar, também, que Injeção de Falhas por software não é uma técnica para ser utilizada isoladamente em um processo de teste. Injeção de Falhas dará seus melhores resultados quando trabalhando junto com outras técnicas de teste. Ela irá complementar essas técnicas dando ao desenvolvedor a maior garantia possível sobre a dependabilidade do sistema sendo desenvolvido. Então, Injeção de Falhas por software atualmente pode ter um papel muito importante no processo de teste de um sistema.

III.8. Resumo

Examinou-se neste capítulo o tópico de Injeção de Falhas. Foi visto como essa técnica pode ser um auxílio valioso na avaliação do comportamento de um sistema na presença de falhas. De acordo com isso, teste por Injeção de Falhas tem sido usado para testar sistemas tolerantes a falhas, para verificar se estes apresentam um comportamento esperado mesmo na presença de falhas. Além disso, Injeção de Falhas também pode dar uma estimativa de que danos um sistema, não necessariamente tolerante a falhas, pode causar se for executado na presença de falhas.

Foram vistas aqui também as diferentes abordagens para Injeção de Falhas. Comparou-se as vantagens e desvantagens de Injeção de Falhas por hardware, por software e através de simulação. Apesar de cada método ter suas vantagens específicas, chegou-se à conclusão que Injeção de Falhas por software se destaca, apresentando as vantagens de ser mais controlável e mais barata, quando comparada com injeção por hardware, e de ser mais próxima das reais condições de execução, quando comparada com simulação. Depois disso, viu-se os diferentes métodos de Injeção de Falhas por software, tais como o uso de modo traço, de *traps*, de *time-outs*, de um processo com privilégios especiais e através da modificação do código fonte. Em seguida a isso, mostrou-se em detalhe cada uma das ferramentas de Injeção de Falhas por software pesquisadas neste trabalho. Para cada ferramenta se discutia seu método de injetar falhas, seu modelo de falhas, as vantagens e desvantagens que apresentava e se indicava o tipo de sistema alvo para o qual ela tinha sido desenvolvida. Por fim, fazia-se uma comparação geral entre as ferramentas, na forma da tabela III.

Por último, conclui-se que Injeção de Falhas é uma técnica que já atingiu sua maturidade. Já há diversas ferramentas e foram criadas várias técnicas para implementar Injeção de Falhas, portanto não é mais um campo experimental. Entre essas técnicas, como já se disse, Injeção de Falhas por software tem se destacado como sendo uma das mais populares. Concluiu-se também que Injeção de Falhas não é um método de teste para ser usado isoladamente: ela pode obter seus melhores resultados quando utilizada conjuntamente com outras técnicas de teste.

CAPÍTULO IV: SISTEMA DE PADRÕES PARA INJEÇÃO DE FALHAS POR SOFTWARE

IV.1. Generalidades

IV.1.1. Introdução

Esta seção da dissertação está organizada da seguinte forma. Primeiramente, nesta introdução, discute-se o conteúdo do Sistema de Padrões para Injeção de Falhas por software [LMR00, LMR01]: fala-se para qual domínio de aplicação ele foi criado e diz-se quais são os padrões que contém. Em seguida, são mostrados os próprios padrões, na ordem indicada na sub-seção abaixo. Segue-se dessa forma uma abordagem semelhante à usada em [BMR+96].

IV.1.2. Conteúdo

O presente sistema de padrões visa facilitar o desenvolvimento de novos programas de Injeção de Falhas. Eles foram desenvolvidos com base no estudo de diversas ferramentas de Injeção de Falhas por software (ver capítulo III). São apresentados aqui três padrões. O primeiro é o seguinte padrão de arquitetura:

- *"Injetor de Falhas":* este padrão de arquitetura visa solucionar o problema de como arquitetar um programa para fazer Injeção de Falhas. Mais especificamente, quais seriam seus componentes, como estes se relacionariam e como seriam suas interfaces. Ou seja, o padrão propõe uma arquitetura para programas que realizam Injeção de Falhas por software.

Além desse padrão, há mais dois outros. Estes são padrões de projeto, e objetivam resolver problemas localizados dentro da arquitetura global do programa. São os seguintes:

- *"Injetor":* o padrão "Injetor" procura solucionar o problema de como seria uma estrutura para realizar a Injeção de Falhas propriamente dita. Isto é, ele sugere um esquema de objetos que iriam se comunicar com o sistema sob teste e simular uma determinada falha (ou falhas) dentro dele, de acordo com uma especificação de falha recebida externamente.
- *"Monitor":* já este padrão procura fornecer uma solução para o problema de criar uma estrutura que faça a monitorização do sistema sob teste. Uma vez feita a injeção propriamente dita, deve-se monitorar o sistema sob teste para se poder

observar se ele se comporta como esperado na presença de falhas. O esquema de objetos proposto neste padrão se presta a isso.

Os padrões descritos acima podem ser examinados em maiores detalhes nas páginas a seguir.

IV.1.3. Origem dos padrões

Os padrões deste sistema se originaram do estudo de ferramentas de Injeção de Falhas, que está descrito no capítulo III. Foi possível tirar de todas as ferramentas subsídios para a formulação dos padrões aqui descritos, porém é importante destacar algumas que deram uma maior contribuição na criação dos padrões. Elas são as seguintes ferramentas: FIRE, ComFIRM, ORCHESTRA, FIESTA, Xception, SOFIT e IPA. Vale dizer também que pudemos examinar o próprio código das três primeiras ferramentas citadas.

Além disso, após ter sido concluído o estudo sobre ferramentas de Injeção de Falhas, e já com o sistema de padrões formulado, descobriu-se mais uma ferramenta. Trata-se da ferramenta de Injeção de Falhas FlexFI (Flexible Fault Injection environment) [BRR99], desenvolvida pelo Instituto Politécnico de Turim, Itália. É mais uma ferramenta altamente estruturada para Injeção de Falhas. O ponto importante, a respeito dela, é que seus criadores realizaram um estudo sobre ferramentas de Injeção de Falha quando do início de seu trabalho, e a partir desse estudo definiram a arquitetura para a ferramenta. E essa arquitetura é muito semelhante à arquitetura descrita no sistema de padrões, dessa forma confirmando o trabalho realizado aqui.

IV.2. "Injetor de Falhas"

Exemplo

Várias ferramentas que usam Injeção de Falhas por software já foram desenvolvidas, e estas apresentam uma arquitetura na qual é possível injetar falhas; monitorar o sistema sob teste; ativar a execução deste último; controlar todo o processo; e informar o usuário da ferramenta sobre os resultados do teste, bem como receber suas solicitações.

Contexto

A criação de um programa ou ferramenta que realize injeção de falhas por software.

Problema

Como já foi dito na sub-seção "Exemplo", um programa ou ferramenta que faça Injeção de Falhas por software (doravante identificada apenas por "ferramenta") deve fazer o seguinte conjunto de atividades:

(i) De início, a ferramenta deve ativar o sistema sob teste (ou, como também pode ser chamado, sistema alvo); se o sistema estiver inativo, a ocorrência de uma falha não terá conseqüência alguma.

(ii) Feito isso, a ferramenta pode então injetar as falhas dentro do sistema sob teste, conforme a especificação do usuário.

(iii) Após a injeção, deve ser feita a monitorização do sistema alvo, para se verificar se ele se comporta como esperado.
(iv) As atividades de Injeção de Falhas, monitorização e ativação devem ser adequadamente controladas e coordenadas.
(v) Deve haver uma interface com o usuário na qual o mesmo pode especificar as falhas que deseja injetar, bem como receber de volta os resultados da injeção.

Assim, é necessária uma arquitetura para a ferramenta de Injeção de Falhas que possibilite a realização dessas atividades. Além disso, é preciso se balancear as seguintes *forças*:

- Deve ser possível alterar a forma com que algumas das atividades acima são realizadas, sem que isso interfira nas demais.
- Só devem se comunicar com o sistema sob teste os módulos da ferramenta que realmente necessitam disso.
- Deve haver uma forma coordenada de os módulos que realizam as atividades de injeção, monitorização e ativação se comunicarem, para evitar que troquem mensagens desnecessariamente.
- A perturbação do sistema alvo deve ser a menor possível. Isto é, o sistema sob teste deve ser executado nas condições mais semelhantes possíveis das reais.
- A arquitetura prevista nesse padrão deve ser tal que a ferramenta de Injeção de Falhas que se baseie nela não fique restrita a testar apenas um tipo específico de aplicação. Embora seja possível usar este padrão para se desenvolver uma ferramenta para um tipo específico de sistema alvo, deve ser possível também criar ferramentas de uso genérico.

Solução

A ferramenta de Injeção de Falhas deve ser estruturada da seguinte forma. Serão definidos cinco subsistemas, correspondendo às cinco atividades definidas na sub-seção "Problema":

- *Ativador:* ativa a execução do sistema alvo para que ele possa ser testado em suas condições de funcionamento normais.
- *Injetor:* realiza a injeção propriamente dita das falhas dentro do sistema sob teste.
- *Monitor:* faz a monitorização do sistema alvo, para verificar se ele opera como esperado.
- *Controlador:* controla os subsistemas acima, para que realizem suas atividades coordenadamente.
- *Interface com o usuário:* recebe as especificações do usuário para a realização do experimento e devolve os resultados do mesmo.

Dos subsistemas acima, apenas *injetor*, *monitor* e *ativador* se comunicam com o sistema sob teste. Além disso, esses subsistemas não se comunicam entre si, devendo trocar informações e receber comandos do subsistema *controlador*, que então fará a coordenação deles. Além disso, o *controlador* irá receber especificações do experimento a ser conduzido da *interface com o usuário*, que utilizará para determinar que parâmetros deverá passar para os demais subsistemas. Esses, posteriormente, repassarão ao *controlador* dados obtidos na realização do experimento, que serão devidamente recolhidos e passados à *interface com o usuário*.

Além dos subsistemas já citados, existem mais dois. Estes outros subsistemas auxiliam a execução dos demais funcionando como repositórios de dados. São os seguintes:

- *Gerenciador de falhas:* este repositório de dados armazena as falhas a serem injetadas no experimento de Injeção de Falhas.
- *Gerenciador de dados monitorados:* já a função deste repositório é armazenar os dados advindos da monitorização do sistema sob teste. Os dados podem ser armazenados assim como são recebidos do componente de monitorização (tal como se fosse um *log file*) ou podem ser armazenados como dados consolidados. Este repositório também pode gerar eventuais estatísticas sobre os dados monitorados, se necessário.

Cada um dos sete subsistemas acima se comunica de uma forma padronizada. Dessa maneira, pode-se trocar subsistemas, sem que isso interfira nos demais, e assim mudar a maneira com que um subsistema realiza sua atividade.

Estrutura

O seguinte diagrama mostra a estrutura da solução proposta na seção anterior.

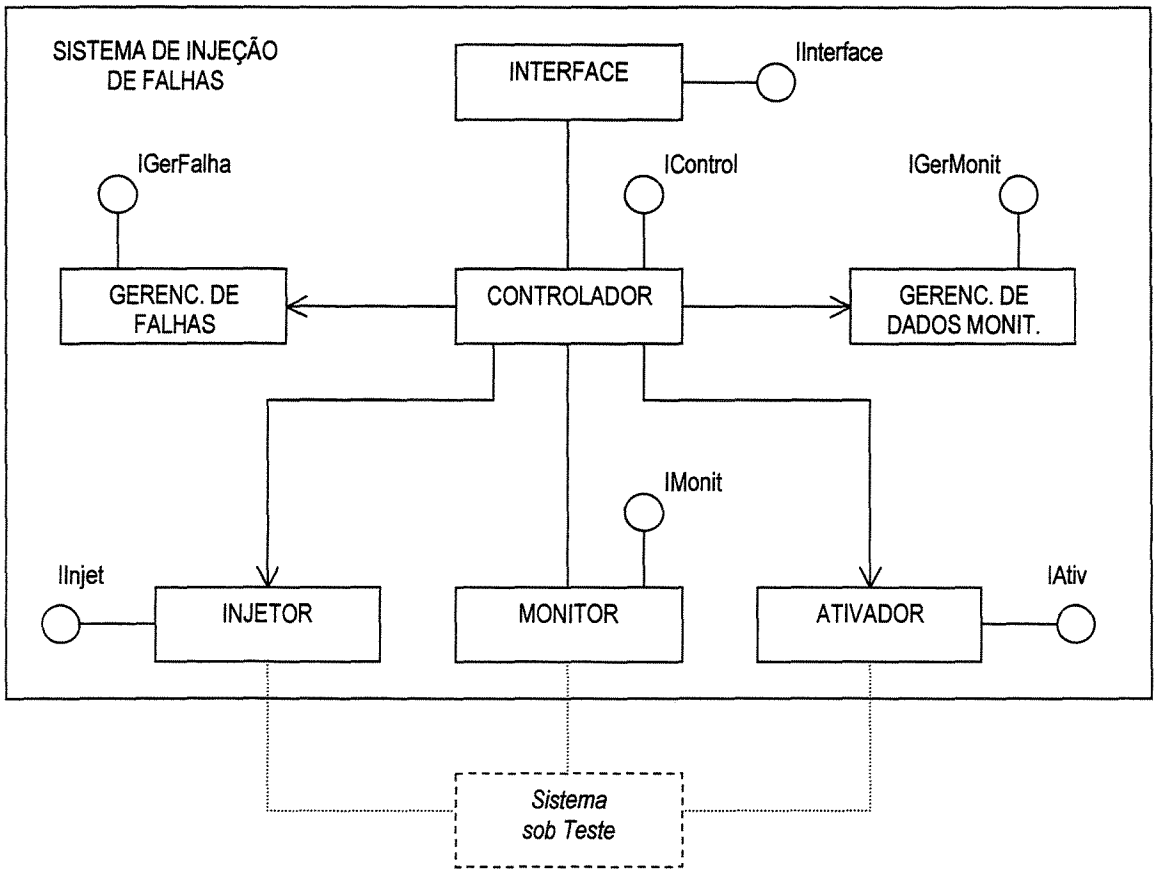


Fig. 2: estrutura do padrão de arquitetura "Injetor de Falhas"

Os retângulos correspondem aos subsistemas descritos na sub-seção "Solução", e o retângulo tracejado é o sistema sendo testado. Este último não faz parte do padrão: está no diagrama para indicar quais subsistemas que se comunicam com o sistema sob teste. A seguir, no diagrama da *Fig. 3*, está descrita cada uma das interfaces.

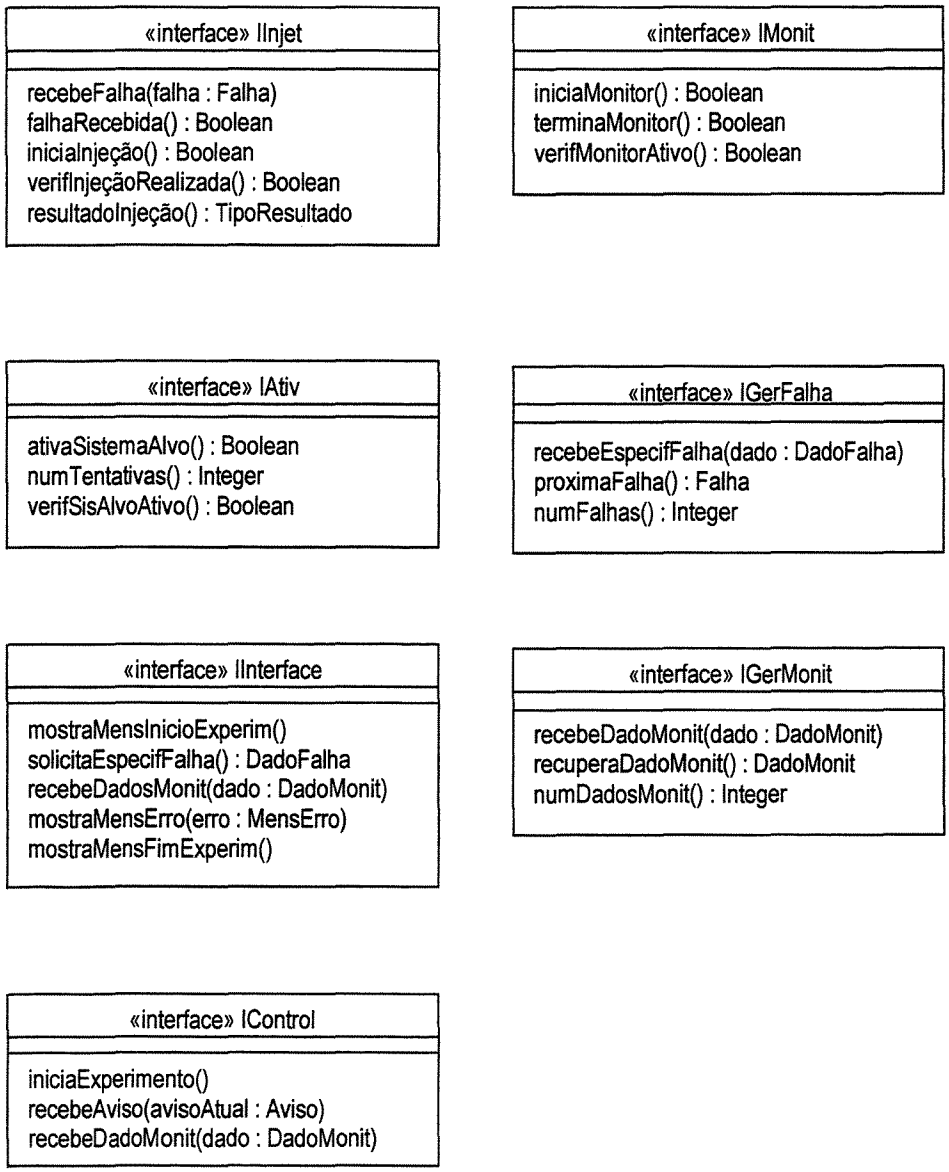


Fig. 3: interfaces dos subsistemas de "Injetor de Falhas"

A seguir, são dados maiores detalhes das operações de cada uma das interfaces acima.

IInjet:
`recebeFalha(falha : Falha)`
Recebe a especificação de uma falha (ou falhas) a ser injetada no sistema sob teste.

`falhaRecebida() : Boolean`
Indica se o subsistema de injeção já recebeu a especificação de uma falha a ser injetada, quando então retorna TRUE. Caso contrário, retorna FALSE.

`iniciaInjeção() : Boolean`
Indica que o subsistema *injetor* deve realizar a injeção de falhas, injetando a falha de acordo com a última especificação recebida. Caso não tenha recebido nenhuma especificação de falha ainda, o método retornará FALSE. De outra forma, retornará TRUE.

`verifInjeçãoRealizada() : Boolean`
Este método informa se já foi feita a injeção da última falha recebida pelo subsistema. Caso isso já tenha ocorrido, se retornará TRUE. Senão, retornará FALSE.

`resultadoInjeção() : TipoResultado`
Este método irá retornar o resultado da injeção da última falha especificada para o subsistema *injetor*. Esse resultado estará expresso como um valor dentro do tipo *TipoResultado*, que especifica os possíveis resultados do processo de injeção.

IMonit:
`iniciaMonitor() : Boolean`
Este método sinaliza ao subsistema *monitor* que ele deve iniciar a monitorização do sistema alvo. Caso ele consiga iniciar com sucesso, irá retornar TRUE. Se não for possível, irá retornar FALSE.

`terminaMonitor() : Boolean`
Determina que o subsistema *monitor* deve parar de monitorar o sistema sob teste. Retorna TRUE para indicar que a monitorização foi encerrada, ou FALSE caso isso não tenha sido possível.

`verifMonitorAtivo() : Boolean`
Indica se o subsistema está monitorando o sistema alvo ou não. Retorna TRUE se o *monitor* está ativo, ou FALSE, se não.

IAtiv:
`ativaSistemaAlvo() : Boolean`
Tenta ativar o sistema alvo. Retorna TRUE em caso de sucesso, FALSE caso contrário.

`numTentativas() : Integer`
Retorna o número de tentativas de ativação do sistema alvo desde que os componentes do subsistema *ativador* foram criados.

`verifSisAlvoAtivo() : Boolean`
Este método verifica se o sistema alvo foi ativado, quando retornará TRUE; senão retornará FALSE.

IGerFalha:
`recebeEspecifFalha(dado : DadoFalha)`
Recebe, do *controlador*, a especificação de uma falha a ser armazenada, para posteriormente ser injetada.

`proximaFalha() : Falha`
Retorna uma falha (ou falhas) a ser injetada.

`numFalhas() : Integer`
Retorna o número de falhas a serem injetadas que estão armazenadas.

IInterface:
`mostraMensInicioExperim()`
Solicita à interface que mostre ao usuário uma mensagem dizendo que está se iniciando o experimento de Injeção de Falhas no sistema sob teste.

`solicitaEspecifFalha() : DadoFalha`
Pede à interface que solicite ao usuário dados sobre as falhas a serem injetadas.

`recebeDadosMonit(dado : DadoMonit)`
Recebe dados monitorados do sistema sob teste, que imediatamente mostra ao usuário.

`mostraMensErro(erro : MensErro)`
Mostra uma mensagem indicando que houve um erro no experimento de injeção de falhas.

`mostraMensFimExperim()`
Mostra uma mensagem indicando o fim do experimento de injeção de falhas.

IGerMonit:
`recebeDadoMonit(dado : DadoMonit)`
Recebe um dado monitorado a partir do sistema alvo. Esse dado é armazenado.

`recuperaDadoMonit() : DadoMonit`
Retorna um dado monitorado dentre os que estão armazenados.

`numDadosMonit() : Integer`
Retorna a quantidade de dados monitorados que estão armazenados.

IControl:
`iniciaExperimento()`
Inicia o experimento de injeção de falhas sobre um dado sistema sob teste.

`recebeAviso(avisoAtual : Aviso)`
É invocado pela *interface*, para avisar o *controlador* de que o usuário fez alguma solicitação durante o processo de injeção de falhas.

`recebeDadoMonit(dado : DadoMonit)`
É invocado pelo *monitor*, para passar ao *controlador* um dado que obteve a partir da monitorização do sistema alvo.

Dinâmica

Um cenário possível da execução de um sistema que segue o padrão de arquitetura é o seguinte:

(i) A execução inicia-se no componente *controlador*.

(ii) *Controlador* solicita que a *interface com o usuário* peça ao usuário os dados sobre o experimento. *Interface com o usuário* recolhe esses dados e os repassa a *controlador*, que por sua vez passa esses dados para o *gerenciador de falhas*.

(iii) *Controlador* solicita a *ativador* que ative o sistema sob teste. Em caso de falha, o *controlador* desiste de realizar o experimento, e solicita à *interface* que mostre uma mensagem de insucesso.

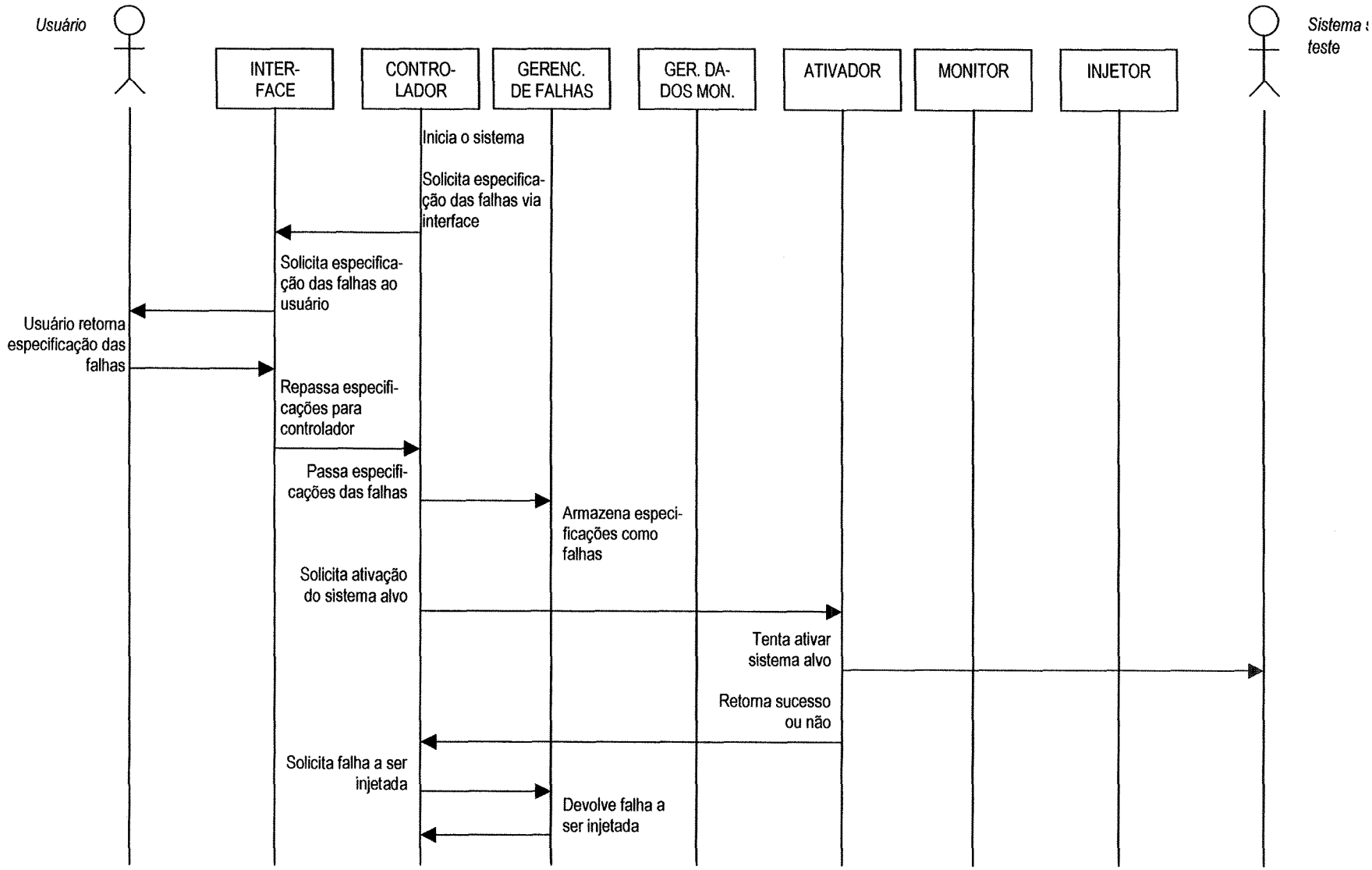
(iv) Em caso de sucesso do *ativador*, o *controlador* solicita ao *gerenciador de falhas* os dados a respeito de uma falha (ou falhas) a ser injetada. Esses dados são repassados pelo *controlador* para o *injetor*. O *injetor* inicia a injeção das falhas, e informa o *controlador* sobre o progresso da injeção das falhas especificadas.

(v) O *controlador* ativa o *monitor*, que passa a monitorar o sistema sob teste. Este último passa os dados recolhidos para o *controlador*, que por sua vez, recebe os dados e os repassa para o *gerenciador de dados monitorados*.

(vi) Injetadas todas as falhas e recolhidos todos os resultados, o *controlador* termina o experimento, desativando o *monitor*.

(vii) Os dados obtidos no experimento ficam guardados no *gerenciador de dados monitorados*. A partir de uma solicitação da *interface*, que é repassada pelo *controlador*, este *gerenciador* pode repassar à *interface* os dados armazenados.

Os passos desse cenário podem ser vistos esquematicamente no diagrama da *Fig. 4*, a seguir.


Usuário
INTER-FACE
CONTRO-LADOR
GERENC. DE FALHAS
GER. DA-DOS MON.
ATIVADOR
MONITOR
INJETOR
Sistema teste
Inicia o sistema
Solicita especifica-ção das falhas via interface
Solicita especifica-ção das falhas ao usuário
Usuário retorna especificação das falhas
Repassa especifi-cações para controlador
Passa especifi-cações das falhas
Armazena especi-ficações como falhas
Solicita ativação do sistema alvo
Tenta ativar sistema alvo
Retorna sucesso ou não
Solicita falha a ser injetada
Devolve falha a ser injetada

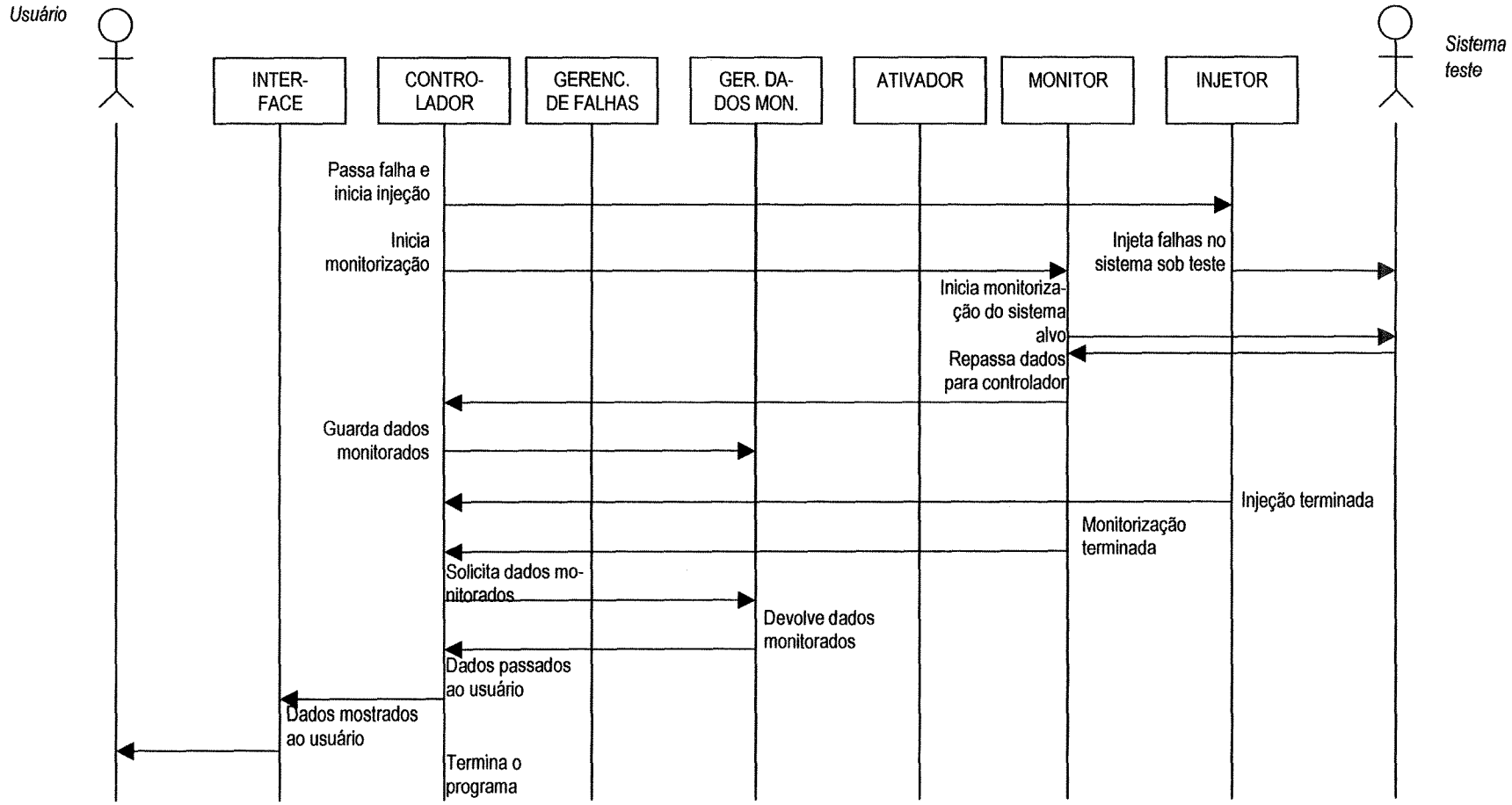


Fig. 4: diagrama de seqüência para um possível cenário de execução de um programa que segue o padrão de arquitetura "Injetor de Falhas"

Implementação

Os passos seguintes indicam um possível método para se implementar o padrão "Injetor de Falhas". Esses passos não são obrigatórios, podendo ser modificados conforme a necessidade.

(1) *Especificar o tipo de sistema que será testado.* Deve-se ter uma idéia clara do tipo de sistema a ser testado pela ferramenta sendo desenvolvida. É possível que se esteja criando uma ferramenta genérica. Mas, no caso de estar se escrevendo uma ferramenta destinada a um tipo determinado de sistema, deve-se saber as características desse tipo de sistema.

(2) *Determinar a forma de injeção de falhas.* O passo seguinte é se determinar como as falhas serão injetadas dentro do sistema sob teste. Deve se escolher uma técnica de Injeção de Falhas que melhor se adapte ao tipo de sistema que será testado.

(3) *Determinar a forma de monitorização do sistema alvo.* Deve ser encontrada uma forma para se realizar a monitorização do sistema alvo, de maneira que se possa verificar se o mesmo está se comportando como esperado na presença de falhas.

(4) *Construir os componentes* injetor *e* monitor. Deve-se construir os componentes que se comunicarão com o sistema sob teste, cada um de acordo com o que foi determinado nos passos (2) e (3). Pode-se utilizar, para se estruturar esses componentes, os padrões de projeto "Injetor" e "Monitor", respectivamente, também contidos neste sistema de padrões.

(5) *Determinar a forma de ativação do sistema alvo.* É necessário formular-se uma maneira de o sistema sob teste iniciar suas atividades. Tendo formulado como se fará a ativação do sistema sob teste, pode-se em seguida construir o subsistema *ativador*.

(6) *Construir o componente* controlador. Definidos os subsistemas dos passos (4) e (5), deve-se especificar um subsistema que coordenará suas atividades. É preferível que o *controlador* se comunique por mensagens de alto nível, sem entrar em detalhes de como as atividades são executadas, para que posteriormente seja mais fácil substituir os subsistemas que as realizam.

(7) *Construir os repositórios de dados (*gerenciador de falhas *e* gerenciador de dados monitorados*).* Neste passo irá se detalhar uma estrutura para guardar os dados sobre as falhas a serem injetadas (estrutura essa que ficará no *gerenciador de falhas*) e também uma estrutura para se armazenar os dados obtidos na monitorização do sistema sob teste (que corresponde ao *gerenciador de dados monitorados*). No caso deste último, também podem ser agregadas algumas funções, tal como elaborar estatísticas.

(8) *Construir o componente* interface com o usuário. De acordo com o que foi definido no passo (6), irá se elaborar uma interface, que deverá receber os dados necessários para os subsistemas gerenciados pelo *controlador*, e os repassar para este último, bem como mostrar os resultados dos experimentos de injeção de falhas.

Conseqüências

O uso do padrão *injetor de falhas* traz os seguintes *benefícios*:

- Permite que as formas de realizar as tarefas de injeção, monitorização, ativação, controle, exposição e armazenamento dos dados sejam alteradas indivi-

dualmente. Tome-se como exemplo uma ferramenta que faz injeção de falhas usando *traps*, e deseja-se mudar a mesma para usar modo traço. Basta apenas refazer o componente *injetor*; os demais componentes continuam os mesmos.

- Centraliza as comunicações entre subsistemas no *controlador*. Assim, evita-se a comunicação de mensagens desnecessárias ou redundantes. Por exemplo, se não houvesse essa centralização, os subsistemas de injeção e de monitorização, quando do início do experimento, iriam se comunicar com o *ativador* para ativar o sistema sob teste. No caso deste padrão, é o *controlador* quem faz isso.
- Apenas quem precisa se comunicar com o sistema alvo o faz. No caso, isso se restringe aos subsistemas *injetor*, *monitor* e *ativador*. Isso é feito para se minimizar a comunicação com o sistema sob teste, a fim de que a perturbação em sua execução seja a menor possível.

Mas, o padrão de arquitetura apresenta também os seguintes *problemas*:

- Não é possível fazer uma rápida comunicação entre os subsistemas *injetor*, *monitor* e *ativador*. Se for necessária uma rápida comunicação entre o *injetor* e o *monitor*, isso deve passar pelo *controlador*, o que implica em um atraso. Isso pode ser importante, visto que deve-se ter o menor atraso possível na injeção das falhas.
- Se durante o processo de injeção, o componente *injetor* ou *monitor* necessitar de um dado extra do usuário, para completar o experimento, essa requisição terá que passar pelo *controlador*, que irá se comunicar com a *interface*, e depois ele irá devolver o dado pedido para o *injetor* ou *monitor*. Isso irá implicar num atraso, o que irá diminuir a velocidade de execução do sistema alvo.

Padrões Relacionados

Os seguintes padrões podem auxiliar na implementação de um sistema que se baseie no padrão de arquitetura "Injetor de Falhas":

- Os padrões de projeto "Injetor" e "Monitor" descrevem, cada um deles, como estruturar os subsistemas de mesmo nome dentro da arquitetura descrita neste padrão.

Caso de uso típico de programa usando o Padrão de Arquitetura "Injetor de Falhas"

Primeiramente o usuário tomará o programa que ele está desenvolvendo e que gostaria de submeter a teste por Injeção de Falhas (doravante chamado "sistema sob teste" ou "sistema alvo"). Então ele irá determinar uma falha que gostaria de testar a resposta do sistema sob teste à mesma. Em seguida, irá tomar a ferramenta de Injeção de Falhas e irá fazer a instrumentação do sistema sob teste para que este possa ser observado pela ferramenta. Logo a seguir, inicializará a ferramenta. Esta solicitará a especificação da falha (ou falhas) a serem injetadas. O usuário passará a especificação da falha que ele determinou anteriormente, no formato requerido pela ferramenta. Então, a ferramenta irá iniciar a execução do sistema alvo. Após este ter sido iniciado, de acordo com o momento que foi especificado para ativação da falha, esta última será injetada no sistema sob teste. Após a injeção, a ferramenta irá monitorar o comportamento do sistema alvo, obtendo dados sobre seu funcionamento. Posteriormente, o experimento é terminado, e a ferramenta devolve ao usuário os resultados do mesmo, isto é, dados que ela obteve sobre o sistema alvo,

monitorando-o após a injeção da falha. Esses dados são mostrados na tela ou gravados em arquivo.

IV.3. "Injetor"

Exemplo

Para se criar um programa que realiza Injeção de Falhas por software, uma das primeiras dificuldades é determinar-se uma estrutura para realizar a injeção de falhas propriamente dita, isto é, uma estrutura que irá simular a ocorrência de falhas dentro do sistema alvo.

Contexto

Criar uma estrutura que permita injetar falhas via software.

Problema

Dentro da ferramenta ou programa que faz a Injeção de Falhas, deve haver um módulo que é o que simula a ocorrência de falhas dentro do sistema sob teste. Esse componente está em contato muito próximo com o sistema alvo, devendo operar nas mesmas condições deste último. Ele deverá alterar o comportamento de algum elemento do sistema sob teste, de tal maneira que este pareça apresentar uma falha. Entretanto, ele deve fazer isso causando o mínimo de perturbação na execução do sistema alvo. Além disso, ele deverá gerenciar a injeção das falhas, relativamente ao fato de elas poderem ser permanentes, intermitentes ou transientes.

Além dessas tarefas já especificadas, a estrutura que fará a Injeção de Falhas deve permitir o equilíbrio das seguintes *forças*:

- A estrutura que realiza Injeção de Falhas deve ser facilmente adaptável para trabalhar em outro ambiente computacional.
- A perturbação do sistema sob teste deve ser a menor possível.
- Deve-se poder utilizar a mesma estrutura para trabalhar com injetores *multi-thread* ou *single-thread*. Se o injetor for *multi-thread*, poderá injetar diversas falhas concorrentemente. Sendo *single-thread*, irá trabalhar com uma falha de cada vez, mas, usando a mesma estrutura.
- Deve ser facilmente extensível, no sentido de poder trabalhar com novos tipos de falha. Caso se decida acrescentar mais tipos de falhas a serem injetadas, deve-se poder adaptar a estrutura para trabalhar com elas.

Solução

Se propõe aqui uma estrutura para simular a presença de falhas dentro do sistema sob teste. Essa estrutura teria três componentes, listados a seguir:

- *Gerenciador de Injeção:* esse componente iria controlar o processo de injeção em si. Ele instanciaria os *injetores* (descritos no item seguinte) de acordo com o tipo de falha a ser injetada. Além disso, iria ativá-los de acordo com a temporização da falha: se esta é permanente, transiente ou intermitente. No fim,

verificaria se os injetores conseguiram realizar sua tarefa, quando iria retornar uma mensagem de sucesso.

- *Injetor:* é instanciado pelo *gerenciador de injeção* para cuidar da injeção de uma falha em específico. Ele deve ser uma subclasse de uma classe base para injetor (a qual é abstrata), mas cada subclasse é implementada para trabalhar com um dado tipo de falha. Contudo, não se comunica diretamente com o sistema sob teste: ele faz isso através de um outro componente denominado *injetor físico* (descrito no item abaixo). Quando se instancia um *injetor*, este por sua vez instancia automaticamente um *injetor físico* associado.
- *Injetor Físico:* é o componente que se comunica diretamente com o sistema sob teste. Apresenta uma interface padrão, que contém primitivas para comunicação com o sistema sob teste. O *injetor* utiliza essas primitivas para se comunicar com o sistema alvo através do *injetor físico*, e com isso simular as falhas. Caso seja necessário fazer a injeção em um novo ambiente, espera-se que seja necessário alterar apenas o *injetor físico*.

Além do que já foi dito acima, vale dizer ainda que espera-se tornar cada um dos componentes o mais coeso e fracamente ligado possível.

Estrutura

O seguinte diagrama mostra a estrutura da solução proposta na seção anterior.

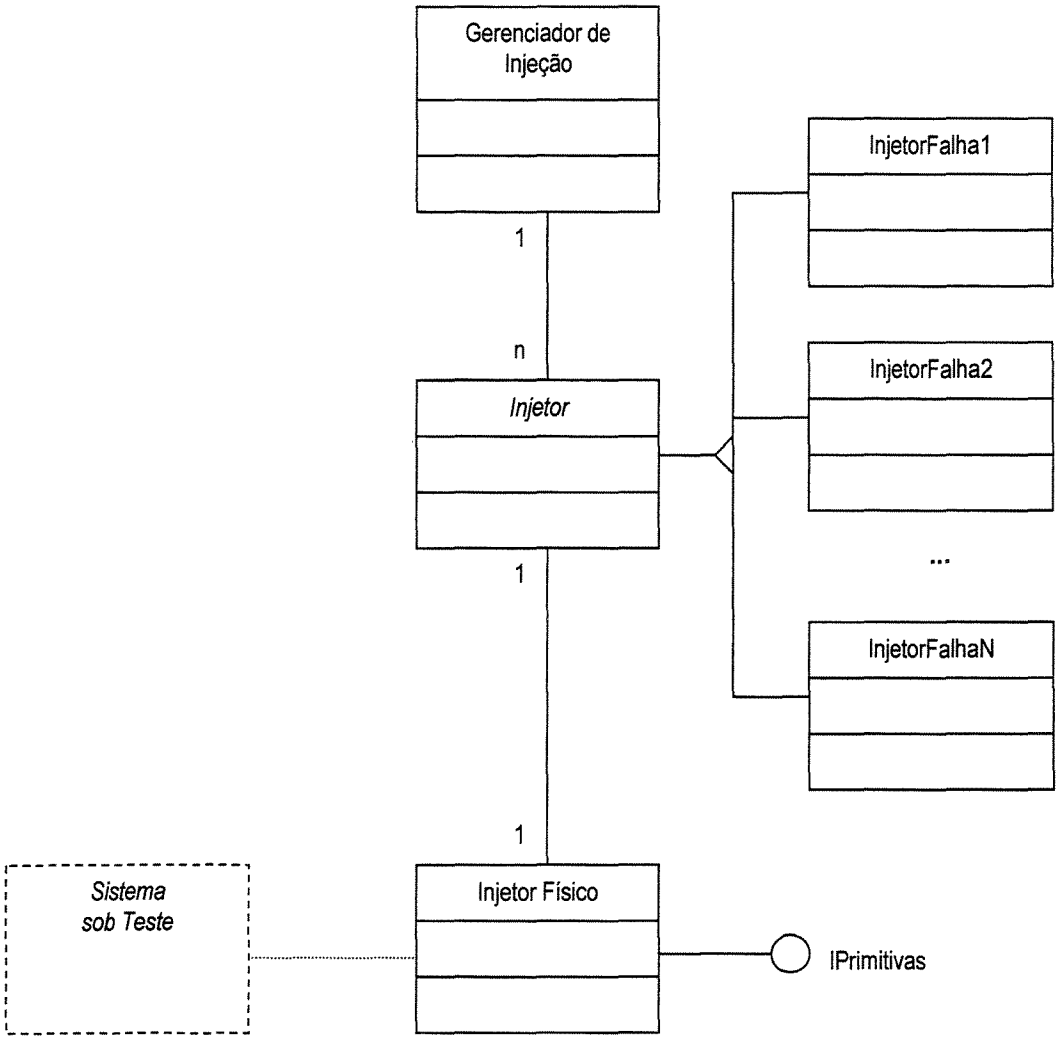


Fig. 5: estrutura do padrão de projeto "Injetor"

No diagrama da *Fig. 5*, o retângulo tracejado é o sistema sendo testado. Este último não faz parte do padrão de projeto aqui descrito: está no diagrama para indicar que apenas os *injetores físicos* se comunicam com o sistema sob teste. A interface *IPrimitivas* designa a interface constituída pelo conjunto de primitivas para comunicação com o sistema sob teste.

Dinâmica

Um cenário possível para a execução do componente cuja estrutura foi descrita acima poderia ser a seguinte:

(i) O componente *Gerenciador de Injeção* recebe a especificação de uma falha (ou falhas) a serem injetadas no sistema sob teste.
(ii) Ele instancia os *injetores* que cuidam de cada tipo das falhas recebidas.
(iii) Por sua vez, cada *injetor* instancia um *injetor físico* para se comunicar com o sistema sob teste.
(iv) O *Gerenciador de Injeção* ativa cada um dos *injetores*, para que façam a injeção de falhas. Se for um sistema *multi-thread*, isso pode ser feito simultaneamente; caso contrário, trata-se de um injetor de cada vez.
(v) Cada *injetor* realiza a injeção de falhas através dos *injetores físicos*.
(vi) Se alguma das falhas a serem aplicadas apresentar um padrão de repetição, o *Gerenciador de Falhas* novamente ativa os *injetores*, seguindo os passos (iv) e (v).
(vii) Cada *injetor* retorna o status da operação, isto é, se conseguiu ou não realizar a injeção.
(viii) O *Gerenciador de Injeção* recebe o status da operação de todos os *injetores* e retorna se teve sucesso ou não em injetar as falhas solicitadas.

Os passos desse cenário podem ser vistos esquematicamente no diagrama da *Fig. 6*, a seguir.

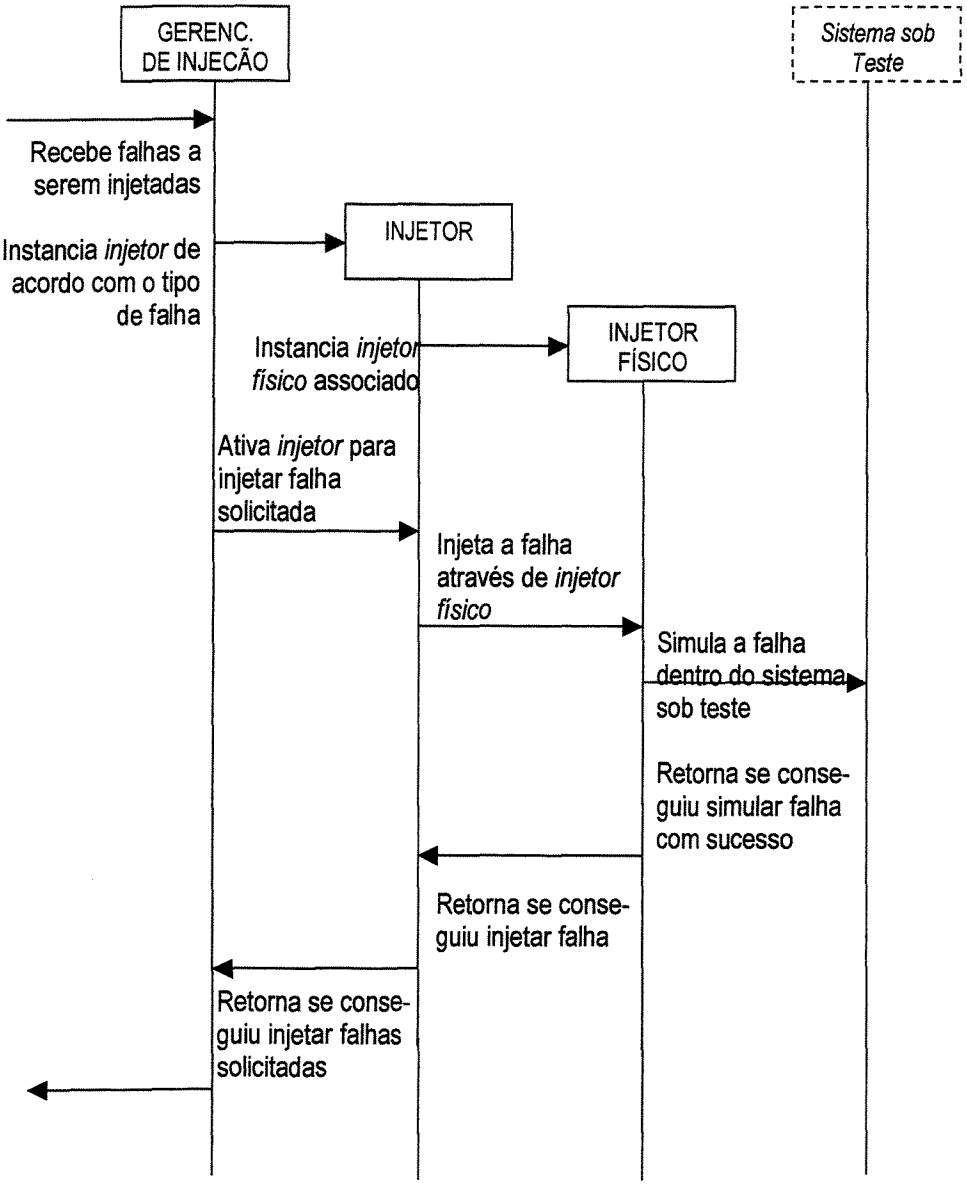


Fig. 6: diagrama de seqüência para um possível cenário de execução do padrão de projeto "Injetor"

Implementação

Uma maneira possível de se implementar o padrão de projeto "Injetor" está indicada nos passos abaixo.

(1) *Especificar se o programa que fará a injeção será multi-thread ou não.* Isto determinará se ele poderá injetar diversas falhas simultaneamente (se ele for multi-thread) ou se poderá apenas injetar falhas seqüencialmente.

(2) *Determinar um modelo de falhas.* Nesse passo, o desenvolvedor deve formular um *modelo de falhas* com o qual se irá trabalhar.

(3) *Elaborar um conjunto de primitivas para comunicar-se com o sistema sob teste.* Para se realizar a injeção das diversas falhas, através dos *injetores* físicos, deverá ser elaborado um conjunto de primitivas que possibilite a comunicação com o sistema sob teste.

(4) *Implementar os* injetores físicos. Deve-se implementar os *injetores físicos*, que irão seguir uma interface que é o conjunto de primitivas determinado no passo (3). Este é o momento da implementação onde o desenvolvedor terá que lidar com as características específicas do ambiente do sistema sob teste.

(5) *Implementar uma classe abstrata para* injetor. Deve-se criar uma classe abstrata que servirá de base para as demais classes de *injetores*. Essa classe estabelecerá uma interface comum para os *injetores* e poderá também conter alguns métodos *default* que poderão ser reaproveitados.

(6) *Implementar os* injetores *para cada tipo de falha.* Usando como base a classe base para os *injetores* desenvolvida no passo (5), implementa-se o *injetor* para cada tipo de falha que se pretende injetar, de acordo com o modelo de falhas determinado no passo (2). Para a implementação dos *injetores*, deve-se levar em conta o conjunto de primitivas estabelecido no passo (3), para comunicação com o sistema sob teste via *injetores físicos.*

(7) *Implementar o* gerenciador de injeção. Deve-se determinar como, a partir de uma especificação de falhas recebida, o *gerenciador de injeção* irá instanciar os *injetores* corretos e os ativará no momento adequado. Aqui deve-se levar em consideração o que foi estipulado no passo (1).

Conseqüências

O uso do padrão de projeto "Injetor" traz os seguintes *benefícios*:

- Torna mais fácil reescrever o componente de Injeção de Falhas para operar em outro ambiente. Todo o código que deve trabalhar com características específicas do ambiente está concentrado no *injetor físico*. Se for necessário mudar-se de ambiente, as mudanças necessárias serão restritas ao *injetor físico*, portanto.
- É facilmente extensível. Caso seja necessário incluir novos tipos de falhas, é fácil fazer essa inclusão. Basta se criar uma nova subclasse da classe base para os *injetores*, onde será implementada a forma de injeção para esse novo tipo de falha, e modificar-se o *gerenciador de injeção* para reconhecer uma especificação para essa nova falha.
- Pode-se usar a mesma estrutura de objetos tanto para ambientes multi-thread como single-thread. Caso o *gerenciador de injeção* receba a especificação de várias falhas a serem injetadas, ele irá instanciar correspondentemente vários *injetores*. Se estiver operando num ambiente *multi-thread* irá ativá-los simultaneamente, caso contrário eles serão ativados seqüencialmente. Uma vantagem adicional é que se estiver-se operando num ambiente multi-thread e tiver que se passar para um ambiente single-thread, as mudanças serão restritas ao *gerenciador de injeção.*
- A estrutura necessita de um mínimo de comunicação com outros módulos dentro do programa de injeção de falhas. A comunicação necessária com outros módulos do sistema é apenas uma especificação das falhas a serem injetadas, e posteriormente o resultado da injeção. Um nível maior de comunicação poderia possivelmente atrasar a execução do sistema sob teste.

Porém, o padrão de projeto aqui exposto também acarreta os seguintes *problemas*:

- O padrão de projeto coloca uma camada de indireção entre o *injetor* — que é quem realmente irá fazer a injeção de falhas — e o sistema sob teste, na forma do *injetor físico*. Essa indireção pode causar um atraso extra na execução do sistema sob teste.

Padrões Relacionados

O seguinte padrão se relaciona com o padrão de projeto "Injetor":

- O padrão de arquitetura "Injetor de falhas" descreve uma arquitetura para um programa que realize Injeção de Falhas. Isto é, nesse padrão estão descritos quais seriam os componentes de um tal programa e como eles se comunicariam. Dentro desse esquema proposto, existe um componente cuja responsabilidade é a de fazer a injeção de falhas propriamente dita. Esse componente poderia se estruturar como é sugerido no padrão "Injetor". Assim, o padrão de arquitetura "Injetor de falhas" indica uma arquitetura de um sistema onde o padrão de projeto "Injetor" poderia se encaixar.

IV.4. "Monitor"

Exemplo

Já se desenvolveram diversas técnicas para realizar a injeção de falhas propriamente dita. Contudo, esse é apenas um aspecto de um programa para realizar injeção de falhas. Precisa-se também *monitorar* o sistema alvo para observar-se seu comportamento. O desenvolvedor de um programa ou ferramenta de Injeção de Falhas precisa definir uma estrutura que lhe permita vistoriar diversos aspectos do comportamento do sistema a ser testado.

Contexto

Criar uma estrutura que possibilite monitorar o comportamento de um sistema submetido à teste por injeção de falhas.

Problema

Uma ferramenta ou programa de injeção de falhas procura gerar ou simular falhas dentro do sistema sob teste. Porém, apenas fazer isso não é o suficiente para se testar um sistema usando Injeção de Falhas. É necessário também monitorar o sistema para verificar como o mesmo irá reagir à presença de falhas. Deve haver portanto uma estrutura que contenha e controle um número de *sensores* ligados ao sistema alvo, para monitorá-lo. Assim, o desenvolvedor deve no projeto de sua ferramenta determinar como deve ser essa estrutura.

É importante lembrar aqui que se está procurando uma estrutura para monitorar um sistema *sendo submetido a teste por Injeção de Falhas*. Não é objetivo deste padrão definir uma arquitetura genérica para fazer monitorização de qualquer tipo.

Tal estrutura para monitorização deverá balancear as seguintes *forças* que estão envolvidas no problema:

- A estrutura para fazer a monitorização trabalha dentro de um determinado ambiente computacional. Caso esse ambiente mude, o trabalho necessário para adaptar a estrutura de monitorização deve ser o menor possível.
- A monitorização do sistema sob teste deve causar o mínimo de intrusividade na execução do sistema alvo.
- É desejável que a mesma estrutura para monitorização possa trabalhar tanto em ambientes multi-thread quanto single-thread. Sabe-se que poder monitorar um sistema dentro de um ambiente multi-thread traz a vantagem de se poder observar vários aspectos do sistema alvo simultaneamente, mas caso o ambiente não permita utilizar esse recurso, espera-se que a estrutura seja a mesma, apenas que ela opere de outra forma.
- Novos aspectos do sistema alvo a serem monitorados podem ser acrescentados depois que a estrutura para monitorização já estiver definida. Assim, essa estrutura deve ser facilmente extensível.

Solução

É proposta uma estrutura que permite observar diversos aspectos do sistema sob teste. Essa estrutura seria totalmente programada em software, e portanto dependeria de recursos compartilhados com o sistema alvo. Toma-se essa abordagem por se considerar que ela é mais versátil e barata. Essa estrutura teria três componentes:

- *Gerenciador de monitorização:* este componente coordenaria o processo de monitorização. Para cada aspecto do sistema alvo, ele instanciaria um *sensor* (vide item a seguir) para o tipo de dado a ser obtido. O *gerenciador de monitorização* iria receber os dados de cada um dos *sensores* e os colocaria em objetos de dados que mandaria para quem havia solicitado os dados monitorados.
- *Sensor:* este objeto é encarregado de monitorar um determinado aspecto do sistema sob teste. É definida uma classe abstrata *sensor* que define uma interface comum para cada tipo de *sensor*. A partir dessa classe, são criadas sub-classes concretas que especificam *sensores* para cada tipo de aspecto a ser monitorado. Os *sensores* se comunicam com o sistema alvo através de *sensores físicos*, que são acessados por um conjunto de primitivas (vide item a seguir). No *sensor* reside a lógica necessária para se obter os dados do sistema sob teste.
- *Sensor físico:* os objetos deste tipo fazem a comunicação dos *sensores* com o sistema sob teste. Eles têm como interface um conjunto de primitivas que permite realizar comunicação com o sistema alvo sem que se tenha que especificar detalhes próprios do ambiente computacional. Portanto, apenas esse componente tem que trabalhar com condições específicas da execução do sistema sob teste.

Espera-se que os componentes listados acima sejam o mais coesos e fracamente ligados dentro do possível.

Estrutura

Os componentes listados na seção anterior podem ser estruturados da maneira descrita no diagrama abaixo:

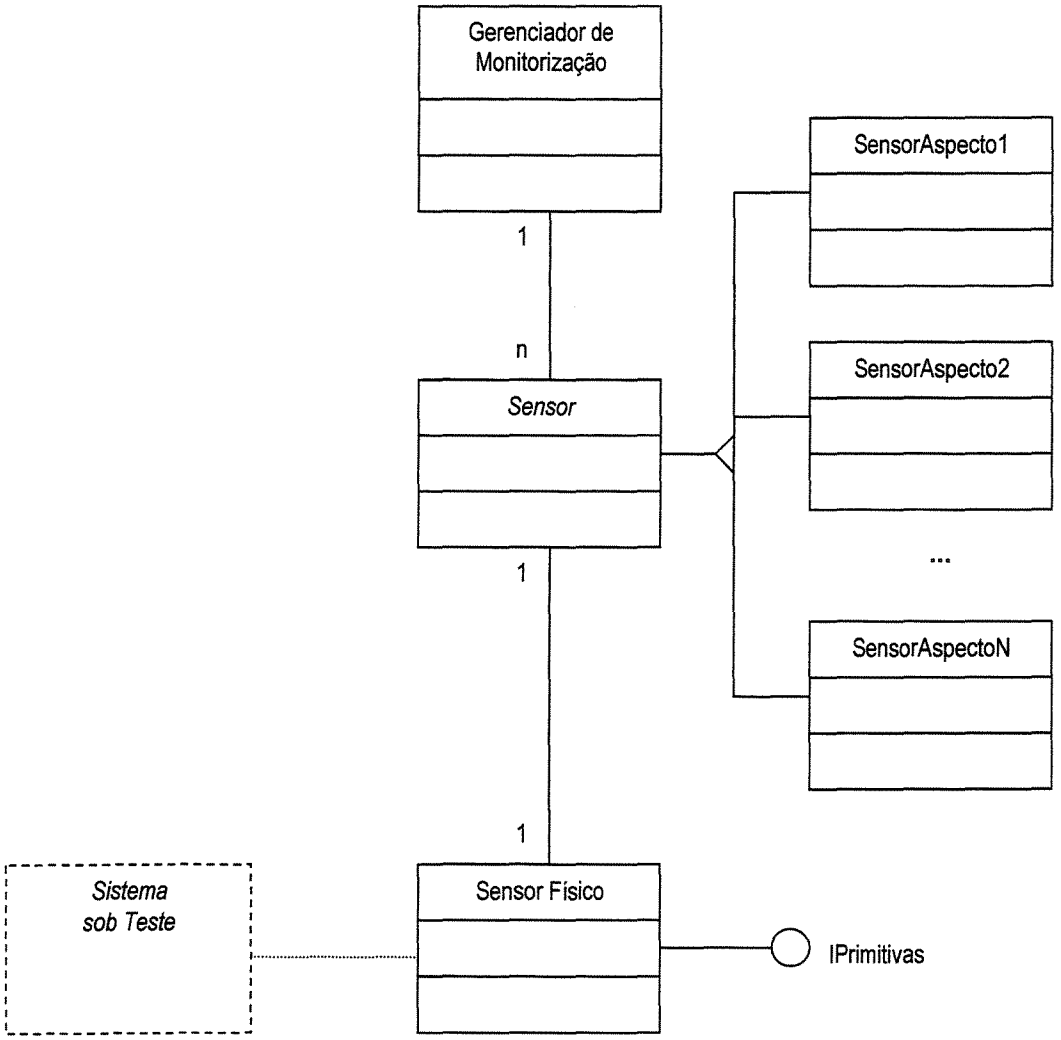


Fig. 7: estrutura do padrão de projeto "Monitor"

No diagrama da *Fig. 7*, o retângulo tracejado é o sistema sob teste. Este último não faz parte do padrão de projeto aqui descrito: está no diagrama para indicar que apenas os *sensores físicos* se comunicam com o sistema alvo. A interface *IPrimitivas* designa a interface constituída pelo conjunto de primitivas para comunicação com o sistema sob teste.

Dinâmica

Pode-se formular o seguinte cenário para uma execução da estrutura proposta para monitorização:

(i) O *gerenciador de monitorização* recebe uma solicitação para monitorar o sistema sob teste.
(ii) O *gerenciador de monitorização* instancia *sensores* para monitorar diversos aspectos do sistema alvo.
(iii) O *gerenciador de monitorização* ativa os *sensores*. Se ele estiver num ambiente multi-thread, eles são iniciados simultaneamente. Caso contrário, eles podem ser iniciados seqüencialmente.
(iv) O *sensor* se comunica com o *sensor físico*, através das primitivas de comunicação, para iniciar a obtenção dos dados do sistema sob teste.
(v) O *sensor físico* obtém os dados necessários. Ele os repassa para o *sensor*.
(vi) O *sensor* repassa os dados para o *gerenciador de monitorização*.
(vii) O *gerenciador de monitorização* empacota em um objeto os dados recebidos dos *sensores* e os envia para quem solicitou esses dados.
(viii) Repete-se o processo a partir do passo (iv), enquanto for necessário se obter dados a respeito do sistema alvo.

Os passos desse cenário podem ser vistos esquematicamente no diagrama da *Fig. 8*, a seguir.

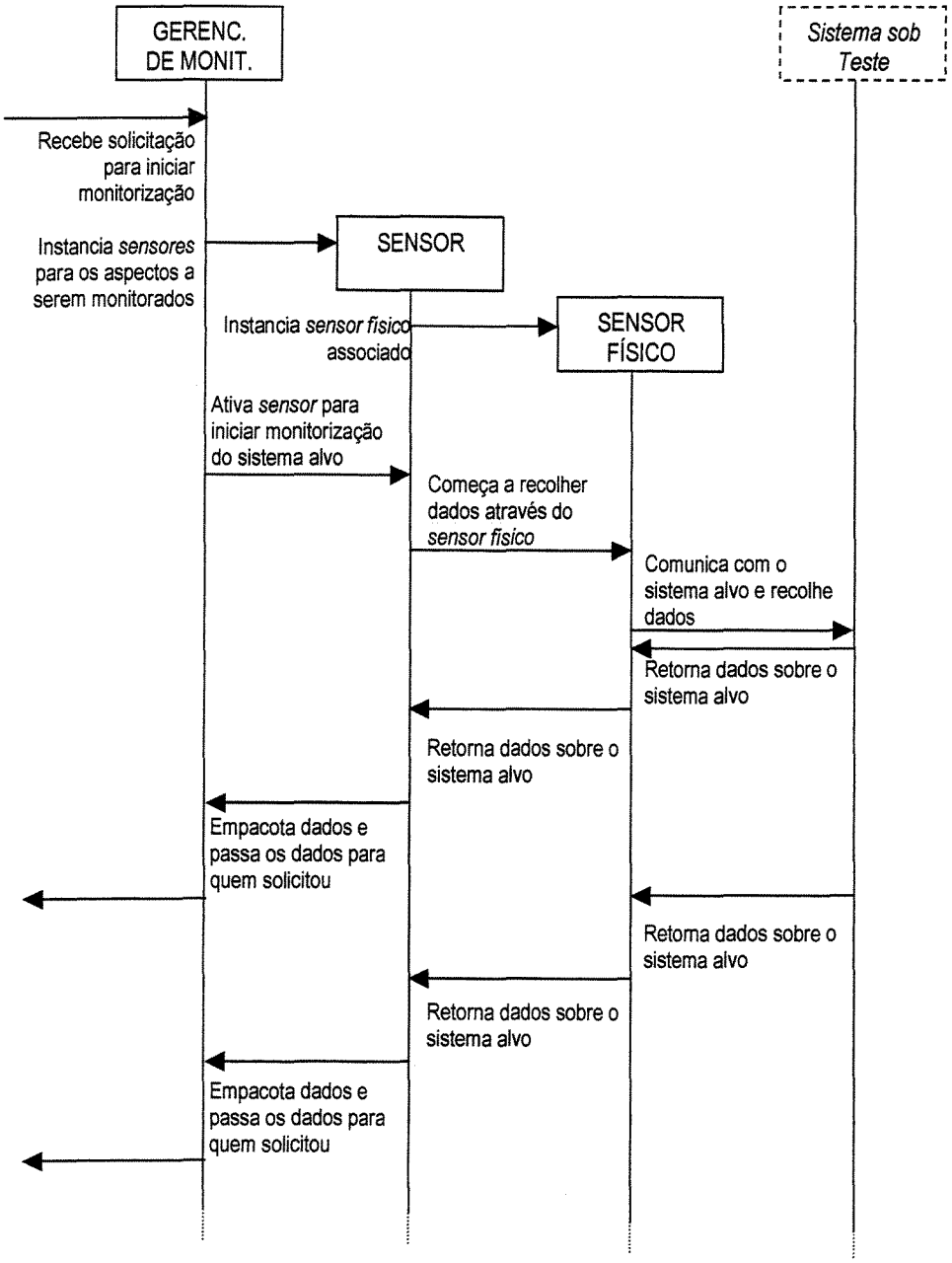


Fig. 8: diagrama de seqüência para um cenário de execução do padrão de projeto "Monitor"

Implementação

Abaixo estão indicados os passos para implementar a estrutura de monitorização.

(1) *Determinar se irá se trabalhar num ambiente multi-thread.* Um dos pontos mais importantes para se realizar a monitorização é saber se será possível trabalhar num ambiente multi-thread. Caso não seja possível, deve-se utilizar uma estratégia para se fazer isso sequencialmente.

(2) *Descrever que dados sobre o sistema alvo serão necessários.* Deve-se determinar que aspectos do sistema sob teste serão monitorados, e que dados serão obtidos.

(3) *Formular um conjunto de primitivas para se comunicar com o sistema alvo.* Deve-se pensar em um conjunto de operações primitivas que permitam comunicar com o sistema sob teste, de maneira a permitir retirar os dados necessários do mesmo.

(4) *Implementar os* sensores físicos. Com base nas primitivas determinadas no passo (3), deve-se implementar os *sensores físicos*. De fato, esses sensores terão o conjunto de primitivas como sua interface. Apenas nessa etapa é que o desenvolvedor vai ter que se ocupar com detalhes do ambiente computacional.

(5) *Implementar os* sensores. Inicialmente, cria-se uma classe base, abstrata, *sensor*. Em seguida, cria-se um *sensor* específico para cada tipo de aspecto a ser monitorado, de acordo com o passo (2). Na construção de cada *sensor*, deve-se levar em conta que eles se comunicarão com o sistema alvo via *sensor físico*.

(6) *Implementar o* gerenciador de monitorização. Deve-se formular uma forma para este ativar os *sensores*, levando-se em conta o que foi determinado no passo (1). Além disso, neste passo determina-se como o *gerenciador de monitorização* irá receber os dados dos *sensores*, e como ele irá empacotá-los, para passar esses dados para o componente externo que os solicitou.

Conseqüências

O uso do padrão de projeto "Monitor" traz os seguintes *benefícios*:

- Torna mais fácil adaptar a estrutura para monitorização para operar em outro ambiente. Todo o código que deve trabalhar com características específicas do ambiente está concentrado no *sensor físico*. Se for necessário mudar-se de ambiente, as mudanças necessárias serão restritas ao *sensor físico*.
- É facilmente extensível. Caso seja necessário monitorar outros aspectos do sistema alvo, basta se criar uma nova subclasse da classe base para os *sensores*, onde será implementada a forma de monitorização para esse novo tipo de dado, e modificar-se o *gerenciador de monitorização* para instanciar o novo *sensor*.
- Pode-se usar a mesma estrutura de objetos tanto para ambientes multi-thread como single-thread. O ambiente multi-thread permite a monitorização simultânea de diversos dados do sistema alvo. Mas, caso não seja possível trabalhar num ambiente multi-thread, ainda assim pode-se utilizar essa mesma estrutura, combinada com alguma estratégia como *polling* para se fazer a monitorização dos diferentes dados.
- A estrutura necessita de um mínimo de comunicação com outros módulos dentro do programa de injeção de falhas. A comunicação necessária com outros módulos é apenas receber uma solicitação para início da monitorização, e em seguida o *gerenciador de monitorização* envia os dados solicitados. Com esse nível mínimo de comunicação garante-se que o processo de monitorização possa causar o mínimo de atraso na execução do sistema sob teste.

Porém, o padrão de projeto "Monitor" também acarreta os seguintes *problemas*:

- O padrão de projeto coloca uma camada de indireção entre o *sensor* — que é quem realmente monitora o sistema — e o sistema sob teste, na forma do *sensor físico*. Essa indireção pode causar um atraso extra na execução do sistema sob teste.

- Pode acarretar uma maior intrusividade no sistema sendo monitorado. Pelo fato de trabalhar com os mesmos recursos do sistema alvo, uma ferramenta que utilize o padrão "Monitor" para monitorar um sistema pode ter uma maior intrusividade na execução do sistema alvo do que, por exemplo, uma abordagem que utilizasse hardware próprio.

Padrões Relacionados

O seguinte padrão pode se relacionar com o padrão de projeto "Monitor":

- O padrão de arquitetura "Injetor de falhas" procura resolver o problema de como arquitetar um programa que faça injeção de falhas. Isto é, ele mostra quais componentes esse programa teria, e como se relacionariam esses componentes. Um desses componentes procura monitorar o sistema sob teste, e então poderia ser estruturado de acordo com o que é proposto neste padrão de projeto. Dessa maneira, o padrão de projeto "Monitor" se encaixa dentro da arquitetura proposta pelo padrão de arquitetura "Injetor de falhas".

IV.5. Resumo

Neste capítulo, foram expostos os padrões do Sistema de Padrões para Injeção de Falhas por software. O primeiro deles é o padrão de arquitetura "Injetor de Falhas". Nesse padrão, se mostra uma arquitetura para ferramentas de Injeção de Falhas. Essa arquitetura estabelece que uma ferramenta de Injeção de Falhas deve ter cinco subsistemas: Injetor, Monitor, Ativador, Controlador e Interface com o usuário. Além desses subsistemas, deve haver ainda dois repositórios de dados, que guardam dados necessários à operação dos subsistemas. São eles: Gerenciador de Falhas e Gerenciador de Dados Monitorados. A execução básica de um programa baseado nessa arquitetura é a seguinte. A execução começa no Controlador, que solicita ao usuário, via Interface, dados sobre as falhas a serem injetadas. Esses dados são guardados no Gerenciador de Falhas. Depois, o Controlador solicita ao Ativador que inicie a execução do sistema sob teste. Feito isso, o Controlador toma as falhas a serem injetadas do Gerenciador de Falhas e as repassa ao Injetor, que é quem realmente simula a presença dessas falhas no sistema sob teste. Ao mesmo tempo, o Monitor fica observando o comportamento do sistema sob teste, e recolhendo dados sobre o mesmo. Esses dados são repassados ao Controlador, que os armazena no Gerenciador de Dados Monitorados.

O padrão seguinte no sistema é o padrão de projeto "Injetor". Ele trata de como seria uma estrutura para simular a presença de falhas no sistema sob teste. Tal estrutura teria três componentes: um Gerenciador de Injeção, um Injetor e um Injetor Físico. O Gerenciador de Injeção recebe as falhas a serem injetadas, e instancia os Injetores para essas falhas. Cada Injetor está associado a um Injetor Físico, da seguinte forma: no Injetor fica a lógica para se injetar uma dada falha, enquanto que no Injetor Físico ficam as características específicas da forma de se simular as falhas no sistema sob teste.

O último padrão do sistema é o padrão de projeto "Monitor". Esse padrão é parecido com o padrão anterior, isso porque Injeção de Falhas, se examinada à luz da teoria para Monitorização de sistemas, é na verdade um caso especializado de Monitorização. Neste padrão, é proposta uma estrutura para fazer a monitorização do sistema sob teste. Essa estrutura teria também três componentes: o Gerenciador de Monitorização, os Sensores e

os Sensores Físicos. O Gerenciador de Monitorização instancia Sensores para os diferentes aspectos do sistema sob teste que terá de monitorar. Cada Sensor está associado a um Sensor Físico, que é quem trata das características específicas de como recolher os dados necessários do sistema alvo. Esses dados são recolhidos pelo Sensor Físico, que os repassa ao Sensor, onde são empacotados. Depois são repassados ao Gerenciador de Monitorização, que passa esses dados para quem quer que os tenha solicitado.

CAPÍTULO V:
FERRAMENTA DE INJEÇÃO DE FALHAS JACA

V.1. Generalidades

Neste capítulo, a ferramenta de Injeção de Falhas JACA é descrita. A ferramenta JACA foi criada com um duplo objetivo. O primeiro, evidente, era verificar, num caso real, se o Sistema de Padrões para Injeção de Falhas por Software era útil na construção de ferramentas de Injeção de Falhas por software. O segundo objetivo era criar uma ferramenta de uso prático para realizar teste de Injeção de Falhas por software em sistemas, e não meramente um demonstrativo do uso do sistema de padrões.

V.2. Requisitos da ferramenta JACA

No início do trabalho na ferramenta JACA, se procedeu à formulação de requisitos que a ferramenta deveria satisfazer. Eles estão listados abaixo:

Requisitos:

(i) *Utilização do sistema de padrões.* O projeto da ferramenta JACA deve utilizar o que está descrito no Sistema de Padrões para Injeção de Falhas por Software, desenvolvido anteriormente dentro do mesmo projeto.

(ii) *Linguagem Java.* Deve ser escrita usando a linguagem Java, conforme definida pela Sun Microsystems. Isso é feito por dois motivos. O primeiro é que no momento do início dos trabalhos na ferramenta JACA não havia ainda uma ferramenta de Injeção de Falhas para sistemas Java. O segundo é que isso permite que a ferramenta rode em muitos diferentes tipos de sistema, podendo ter um número maior de aplicações.

(iii) *Reflexão Computacional.* A ferramenta utilizará, para realizar a injeção de falhas propriamente dita, reflexão computacional. A ferramenta JACA herdou isso da ferramenta FIRE (vide sub-seção III.4.13), a qual utilizava reflexão computacional para injetar as falhas, e que comprovou a validade de tal abordagem para Injeção de Falhas [Ros98]. Um processo semelhante ao descrito naquela sub-seção é empregado aqui. Como vantagem, tem-se que a ferramenta pode rodar em diferentes sistemas sem alterações.

(iv) *Injeção de falhas em alto nível.* De princípio, a ferramenta JACA deve trabalhar com falhas em alto nível. Isto é, falhas que para serem injetadas não requeiram que a ferramenta JACA tenha que operar com rotinas em Assembly ou com aspectos específicos do ambiente computacional onde a

ferramenta está rodando. Falhas em alto nível são relacionadas a aspectos da linguagem de programação do software do sistema sob teste. Mais especificamente, com problemas que podem ocorrer na atribuição de valores a atributos e parâmetros de chamadas de métodos, bem como com valores de retorno de métodos.

(v) *Uso da máquina virtual Java padrão.* Deve rodar em qualquer máquina virtual Java que siga as especificações da Sun Microsystems, exceto quando o protocolo de metaobjetos adotado necessitar de uma máquina virtual específica.

(vi) *Adoção do padrão 100% puro Java.* Afora a parte da ferramenta JACA que tem que trabalhar com o protocolo de metaobjetos, o resto da ferramenta deve ser 100% puro Java, tal como definido pela Sun Microsystems.

(vii) *Definição de falhas a serem injetadas.* Para especificar as falhas a serem injetadas, pode-se utilizar um esquema semelhante ao que foi usado na ferramenta FIRE, que é a utilização de um arquivo de especificação de falhas.

(viii) *Adaptação para injeção de falhas em baixo nível.* A ferramenta JACA deve poder ser adaptada para operar com falhas em baixo nível. Ou seja, falhas relacionadas a elementos de hardware do sistema sob teste, tasi como registradores, memória, portas, etc. Deve-se notar também que esta adaptação da ferramenta deve ser feita com um mínimo de reescrita.

(ix) *Reaproveitamento do projeto da ferramenta FIRE.* O projeto da ferramenta JACA poderá se basear em determinados aspectos da ferramenta de injeção de falhas por software FIRE (*Fault Injection using a REflective architecture*) [Ros98], desenvolvida dentro do próprio IC da UNICAMP. Isso é feito para se dar continuidade ao trabalho já realizado no IC da UNICAMP relativo à Injeção de Falhas.

(x) *Independência de protocolo de metaobjetos.* A ferramenta JACA irá trabalhar com um protocolo de metaobjetos para a linguagem Java. Entretanto, a ferramenta JACA não deve depender de um protocolo de metaobjetos em específico. Deve ser possível fazer a ferramenta JACA passar a operar usando outro protocolo de metaobjetos.

(xi) *Extensibilidade de falhas.* Deve ser possível adicionar tipos diferentes de falhas a serem injetadas com um mínimo de reescrita da ferramenta JACA.

V.3. Projeto da ferramenta JACA

Estabelecidos os requisitos para a ferramenta JACA, passou-se ao projeto da mesma. Nas sub-seções seguintes está descrito o projeto da ferramenta, como ela está em sua versão atual.

V.3.1. Arquitetura da ferramenta JACA

Para a arquitetura da ferramenta JACA, sua estrutura global como um sistema, tomou-se o que havia sido definido no Padrão de Arquitetura "Injetor de Falhas" (vide sub-seção IV.2), e que pode ser observado no diagrama abaixo.

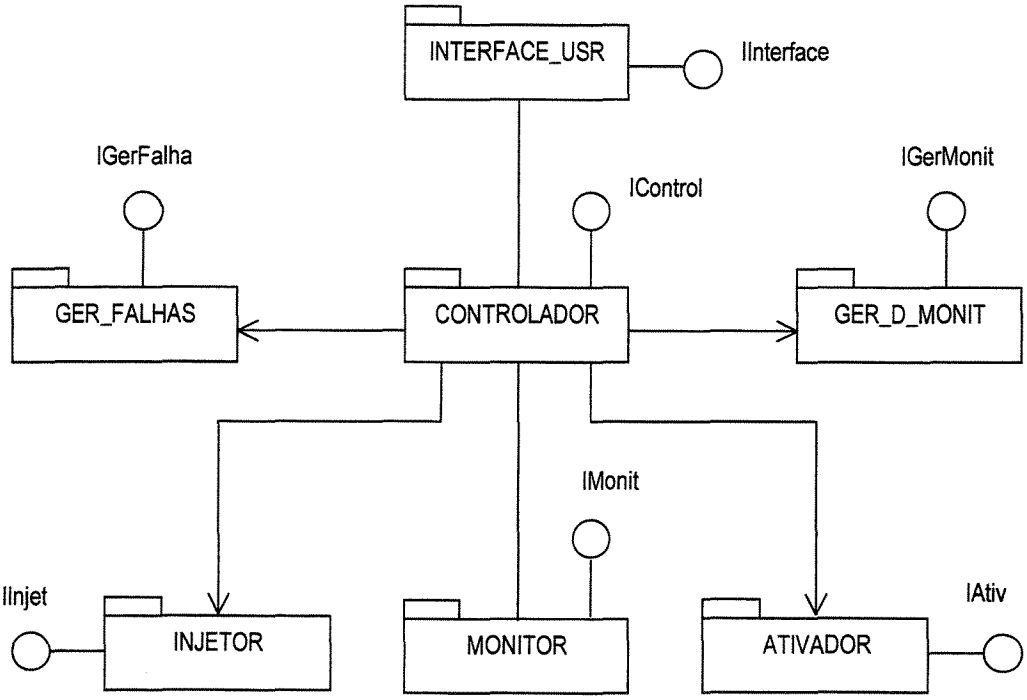


Fig. 9: arquitetura da ferramenta JACA

No diagrama da *fig. 9*, os pacotes "Injetor", "Monitor", "Ativador" e "Controlador" correspondem aos subsistemas de mesmo nome no Padrão de Arquitetura "Injetor de Falhas". No caso dos três pacotes restantes, "Ger_Falhas" corresponde ao subsistema "Gerenciador de Falhas", "Ger_D_Monit" corresponde ao subsistema "Gerenciador de Dados Monitorados" e "Interface_Usr" corresponde ao subsistema "Interface". Cada um desses pacotes é descrito nas sub-seções seguintes, mas pode-se dizer que a funcionalidade de cada um segue o que foi definido no padrão "Injetor de Falhas". A tabela IV, na página seguinte, sumariza a finalidade de cada pacote e lista as classes de cada um.

Deve-se dizer aqui também que todos os pacotes formam um único programa executável, *menos o pacote GerMonit*. Esse último pacote guarda em memória os dados monitorados sobre o sistema sob teste. Se rodasse junto com os outros pacotes (e, portanto, com o sistema sob teste) ele poderia perder os dados monitorados, na ocorrência de um "crash" total do sistema. Maiores informações sobre isso podem ser obtidas na sub-seção sobre o pacote *GerMonit*.

Tabela IV: pacotes da ferramenta JACA

Pacote	*Classes*	*Finalidade*
Controlador	Controlador	Controla e coordena toda a execução da ferramenta JACA.
Injetor	GerInjet, Injetor (e sub-classes InjetorFalhaAtributo, InjetorFalhaRetMetodo, InjetorFalhaParametro), InjetorFis, classes que implementam Alterador (AlteraInt, AlteraBool, AlteraChar, AlteraFloat e AlteraString).	Instancia e gerencia instâncias de Injetor, os quais, através de instâncias de InjetorFis, simulam a presença de falhas no sistema alvo.
Monitor	Monitor, Sensor (e sub-classe SensorObjeto), SensorFis	Instancia e gerencia instâncias de Sensor, os quais, através de instâncias de SensorFis, recolhem dados sobre o comportamento do sistema sob teste.
Ativador	Ativador, AtivadorFis	Uma instância de Ativador, através de AtivadorFis, inicia a execução do sistema alvo.
Interface_Usr	InterfaceUsr	Pede dados sobre o experimento ao usuário, e avisa sobre o andamento do teste por Injeção de Falhas.
Ger_Falhas	GerFalha	Armazena dados sobre as falhas a serem injetadas.
Ger_D_Monit	GerMonit	Guarda dados obtidos sobre o comportamento do sistema sob teste.
Classes usadas entre pacotes	Falha (e sub-classes FalhaRetMetodo, FalhaAtributo e FalhaParametro), DadoMonit (e sub-classes DadoMonitRetMetodo, DadoMonitAtributo, DadoMonitFalha)	Empacotam dados a serem repassados entre pacotes: Falha guarda dados sobre uma falha a ser injetada e DadoMonit guarda um dado obtido na monitorização do sistema alvo.

V.3.2. Pacote "Injetor"

O diagrama seguinte mostra a estrutura deste pacote:

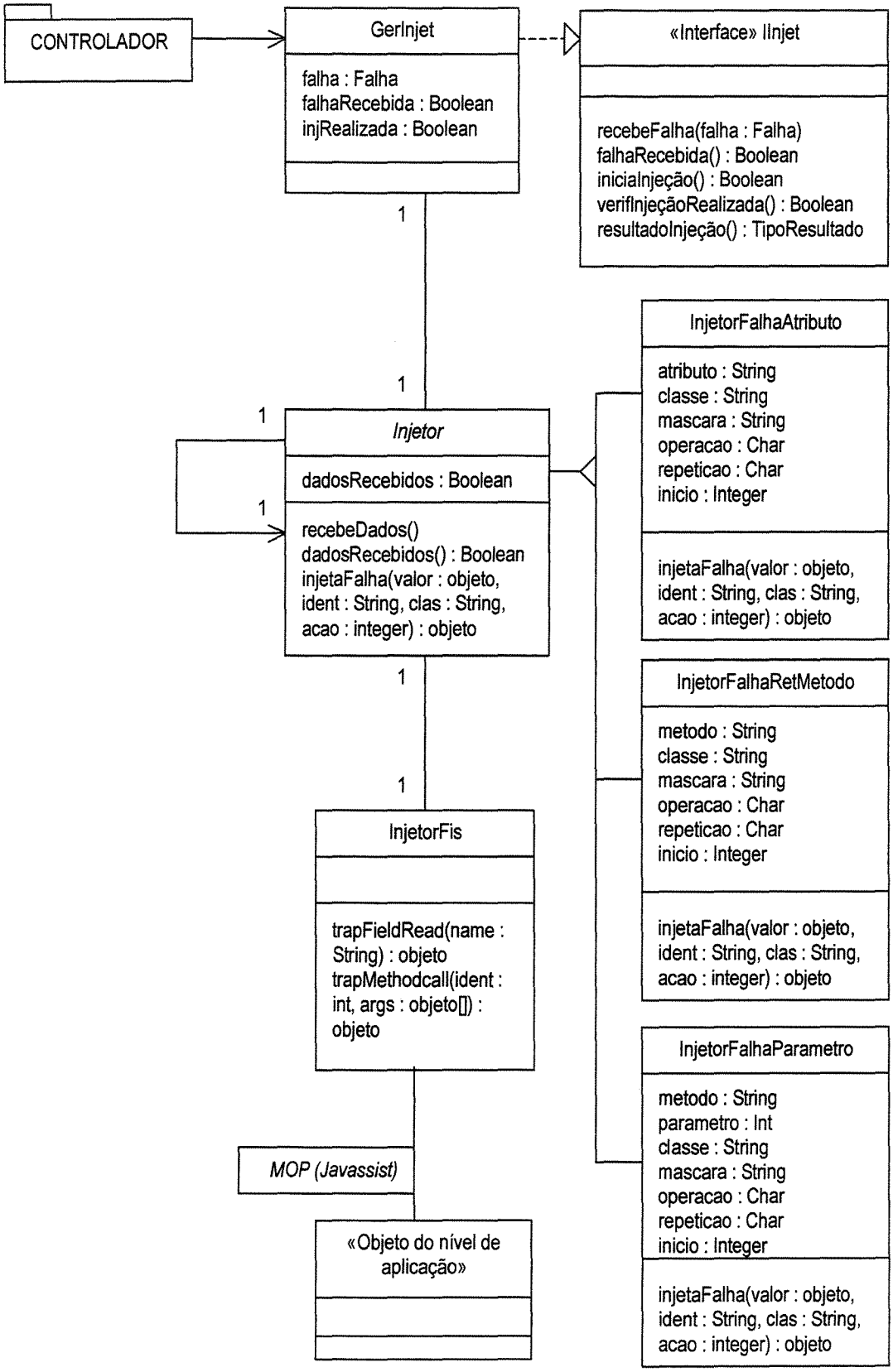


Fig. 10: estrutura do pacote "Injetor"

O pacote *Injetor* funciona assim. Quando *GerInjet* recebe uma falha, através de *recebeFalha(falha : Falha)*, ele instancia o *Injetor* correspondente, no caso um *InjetorFalhaAtributo* para injetar falhas de atributo, um *InjetorFalhaRetMetodo*, para falhas de retorno de método, ou ainda um *InjetorFalhaParametro*, para falhas de parâmetros de chamadas de métodos. Caso ele receba uma lista de falhas a serem injetadas simultaneamente, irá instanciar uma lista de injetores, e *GerInjet* irá apontar para o primeiro injetor dessa lista.

Após isso, entra em ação o protocolo de meta-objetos encarregado de implementar reflexão dentro da ferramenta JACA, para permitir a essa injetar falhas, tal como foi feito na ferramenta FIRE (vide sub-seção III.4.13), na qual JACA se inspirou. Pesquisou-se, no início do projeto, protocolos de meta-objetos, tais como *Guaraná* [Oli98], *OpenJava* [TCK+00] e *Dalang* [Wes99]. Entretanto, para a primeira versão da ferramenta JACA, acabou se escolhendo o protocolo de meta-objetos *Javassist* [Chi98]. Este é um protocolo relativamente simples de se trabalhar, o que facilitou a programação com ele e economizou tempo no projeto. Entretanto, apesar de sua simplicidade, Javassist tem como vantagem o fato de não necessitar recompilar o sistema e portanto dá a ferramenta JACA a vantagem de não necessitar do código fonte do sistema sob teste.

A reflexão irá operar assim. Quando um objeto, de uma classe em que se quer injetar falhas for instanciado, automaticamente Javassist irá instanciar como meta-objeto associado um objeto *InjetorFis*. O meta-objeto *InjetorFis* irá se comunicar com *GerInjet*, e se associará a lista de *Injetores*. Dentro de *InjetorFis* há métodos para interceptar cada leitura ou escrita nos atributos do objeto do nível de aplicação, e também para cada invocação de um método do objeto do nível de aplicação. Quando ocorre um desses eventos, *InjetorFis* passa os dados para os *Injetores*, que irão proceder a Injeção de Falhas, então. Por exemplo, se ocorre uma leitura do valor de um atributo do objeto do nível de aplicação, *InjetorFis* intercepta essa leitura, e repassa o valor do atributo e uma indicação de que se estava fazendo uma leitura dele, para os *Injetores*. Se houver um *Injetor* programado para injetar uma falha precisamente na leitura desse atributo, por exemplo, multiplicar por três o valor desse atributo, ele irá tomar o valor que deveria ser lido e multiplicar por três, retornando esse novo valor para *InjetorFis*. Por sua vez, *InjetorFis*, através da reflexão, fará com que o valor efetivamente lido para esse atributo seja o novo valor calculado pelo *Injetor*, e não o valor que realmente está no atributo, dessa maneira simulando que um valor errôneo foi armazenado no atributo.

Para falha de retorno de método, o processo é o mesmo: a invocação de um método do objeto do nível base é interceptada pelo *InjetorFis*, pega-se o valor que deveria ser retornado e ele é alterado por um *Injetor*, e o *InjetorFis* faz com que esse seja o valor de retorno, dessa forma simulando uma falha de software no método. Um processo similar ocorre com falha de parâmetro de chamada de método. A chamada de um método é interceptada pelo *InjetorFis*. Tomam-se os parâmetros dessa chamada de método e eles são passados para um *Injetor*, que irá alterar o valor de um desses parâmetros. *InjetorFis* irá, então, invocar o método com esses parâmetros, entre os quais está o parâmetro com valor alterado, simulando uma falha onde um valor errôneo é passado como parâmetro.

Note que é necessário realizar operações com os valores envolvidos na Injeção. Isso é válido para todos os tipos de falhas injetadas: tem que se modificar o valor do atributo, do retorno de método ou do parâmetro, mas todas essas falhas vão ter que alterar um valor de um dado tipo em Java. Para fazer isso, especificou-se as classes que implementam a interface *Alterador*. A interface *Alterador* define apenas um método, denominado

novoValor, que tem como argumentos a valor a ser alterado, a operação a ser realizada e uma máscara. Então, construiu-se uma classe para cada tipo que se ia trabalhar: *AlteraInt* (para inteiros), *AlteraBool* (para valores booleanos), *AlteraChar* (para caracteres), *AlteraFloat* (para valores de ponto flutuante) e *AlteraString* (para strings). Cada uma dessas classes implementa o método *novoValor* para o tipo em questão, e dentro desse método está então como alterar o valor do tipo especificado. Quando *Injetor* é solicitado para injetar uma falha, ele instancia uma classe que implementa *Alterador* para o tipo de valor que será modificado. Para cada tipo de dado, então, podem ser realizadas todas as operações permitidas para aquele tipo, de acordo com a especificação da Linguagem Java [GJS+00]. Por exemplo, operações de manipulação de bits só são permitidas em valores inteiros, em Java, e assim só *AlteraInt* implementa essas operações.

V.3.3. Pacote "Monitor"

O pacote "Monitor", que pode ser observado no diagrama da página seguinte, também utiliza reflexão. No caso, a reflexão permite fazer a monitorização do sistema sob teste. O processo é semelhante ao que foi descrito para o pacote *Injetor*. Tem-se um *SensorFis*, que é um meta-objeto associado a um objeto do nível de aplicação. O *SensorFis* irá interceptar cada escrita de um atributo ou chamada de método do objeto do nível de aplicação. Só que neste caso, *não* se irá alterar os valores envolvidos. O que se quer apenas é registrar esses valores. Os valores de escrita em atributo e retorno de chamada de método são passados para os respectivos *Sensores*, que empacotam os dados dentro de objetos *DadoMonitAtrib* e *DadoMonitRetMetodo* (vide sub-seção V.3.9). Esses objetos de dados, por sua vez, são repassados a *GerMonit*, que os repassa ao *Controlador*.

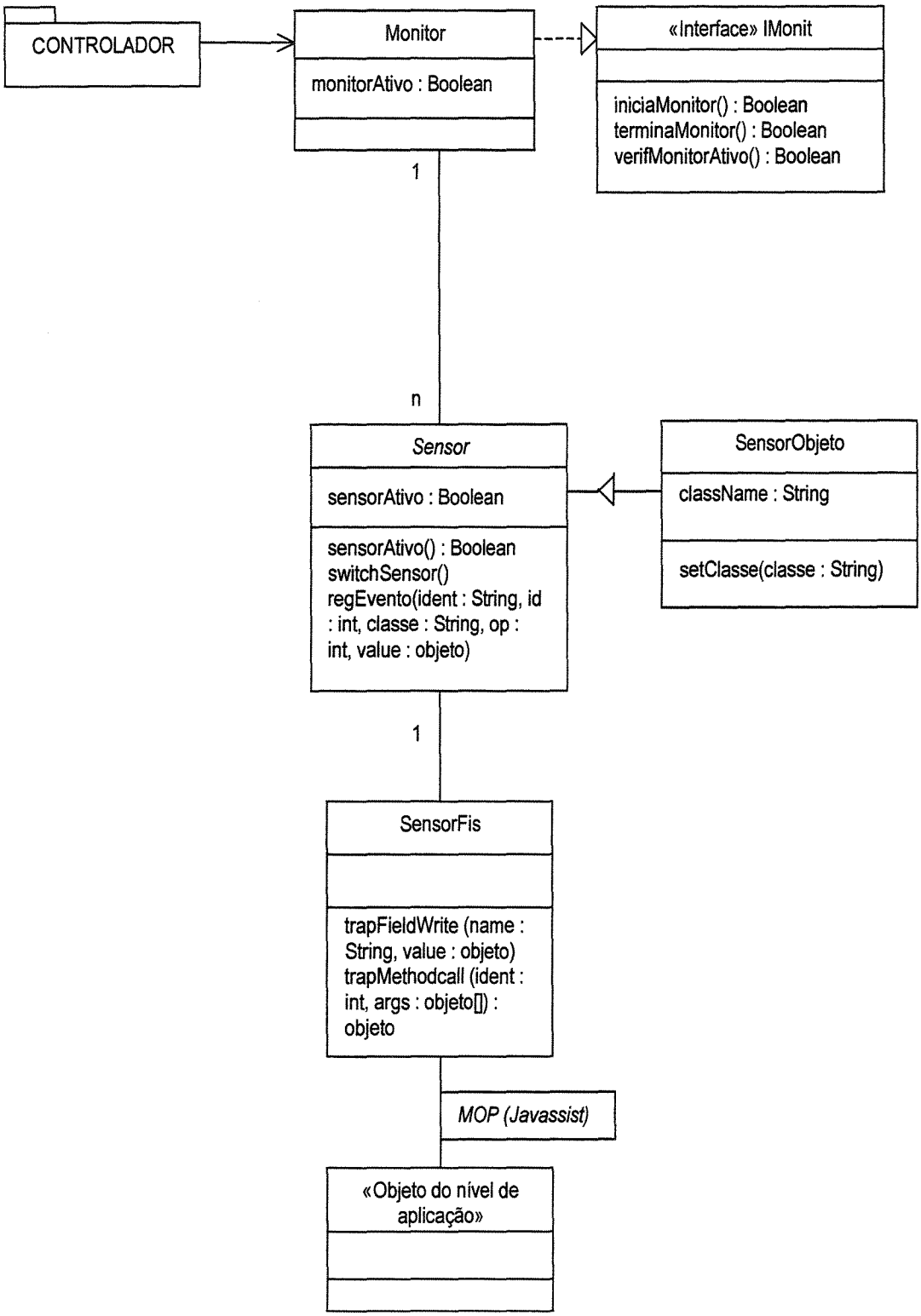


Fig. 11: estrutura do pacote "Monitor"

V.3.4. Pacote "Ativador"

No diagrama abaixo é descrito o pacote "Ativador".

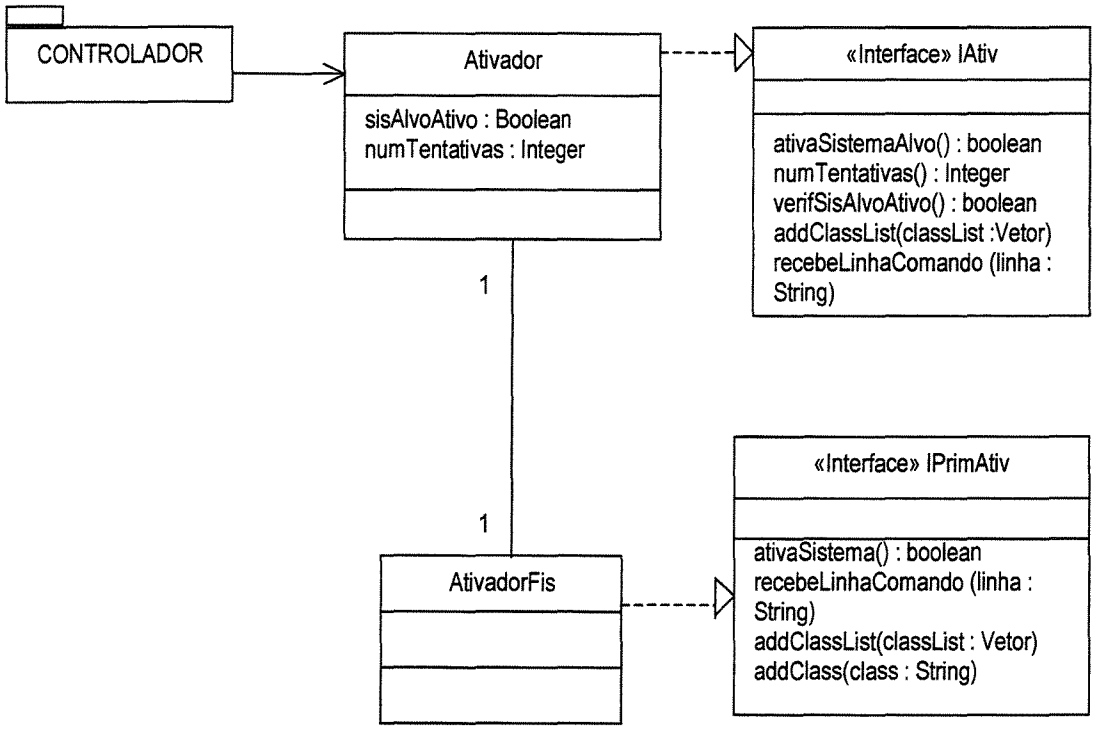


Fig. 12: estrutura do pacote "Ativador"

O pacote *Ativador* é composto por duas classes, e segue aproximadamente a mesma idéia empregada no pacote *Injetor* e no pacote *Monitor*. A primeira classe, *Ativador*, se concentra na lógica necessária para se ativar o sistema sob teste. Os detalhes específicos da ativação, no presente caso relativos à forma como Javassist inicia a execução de um sistema, ficam concentrados em *AtivadorFis*. Assim, se for necessário mudar a maneira específica de se ativar o sistema sob teste, tem-se que reescrever apenas *AtivadorFis*. O pacote *Ativador* ainda tem mais um papel. Dentro de *AtivadorFis*, serão chamados determinados métodos do Javassist para se indicar que classes do sistema sob teste serão reflexivas. Em outras palavras, para cada objeto instanciado de uma classe marcada como reflexiva, será instanciado um meta-objeto associado. Serão tornadas reflexivas as classes que sofrerão Injeção de Falhas ou Monitorização pela ferramenta.

V.3.5. Pacote "Controlador"

O diagrama abaixo representa o pacote "Controlador".

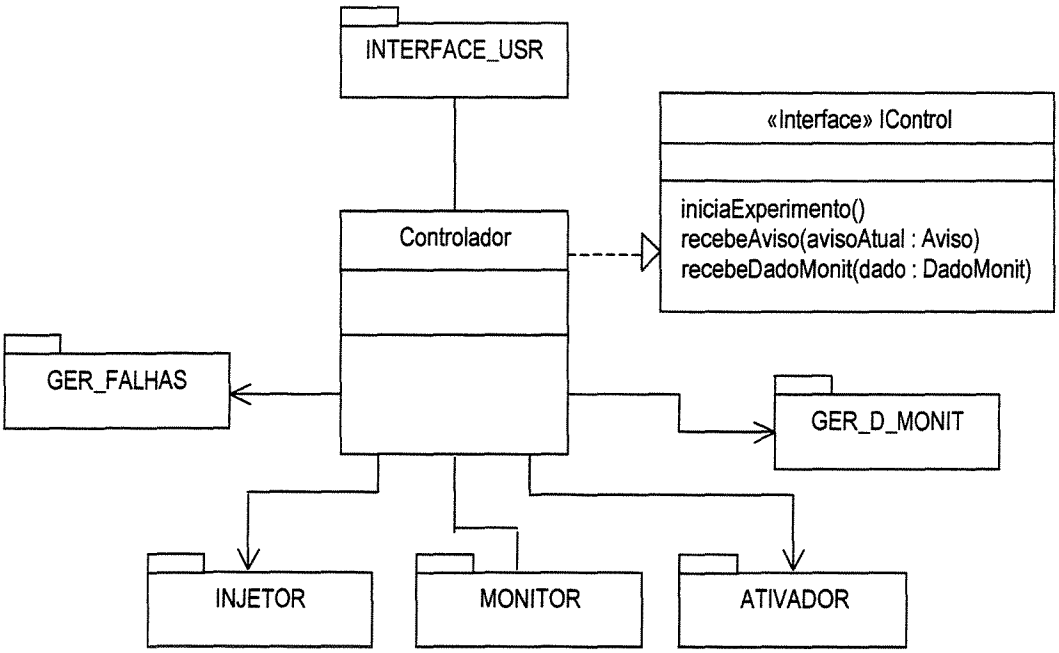


Fig. 13: estrutura do pacote "Controlador"

O pacote *Controlador*, apesar da importância que tem, é relativamente simples. Consiste apenas de uma classe, *Controlador*, que implementa os métodos definidos pela interface *IControl*. Além disso, *Controlador* contém apontadores para objetos dentro de todos os demais pacotes (como pode ser visto nos diagramas para as outras classes). Os métodos que *Controlador* implementa são os seguintes:

- *iniciaExperimento():* este é o método mais importante de *Controlador*. É ele que dá início a toda a seqüência de eventos (conforme mostrado no diagrama da *fig. 4*) que perfazem um teste por Injeção de Falhas.
- *recebeAviso(avisoAtual : Aviso):* esse método foi colocado como uma provisão para futuras alterações. Caso se decida por um novo tipo de interface com o usuário, e essa interface seja capaz de captar comandos do usuário no meio de algum processo que esteja sendo feito na ferramenta, o pacote *Interface_Usr* será capaz de informar o *Controlador* que o usuário fez uma solicitação que deve ser processada imediatamente. Para tal, basta invocar-se esse método, que deverá estar programado para tratar tais eventos.
- *recebeDadoMonit(dado : DadoMonit):* este método é invocado pelo pacote *Monitor*, para passar ao *Controlador* um dado que acabou de obter a partir da monitorização do sistema sob teste, empacotado na forma de um objeto *DadoMonit* (vide sub-seção V.3.9 – "Outras Classes"). O *Controlador* toma esse dado e o repassa ao pacote *GerMonit*.

V.3.6. Pacote "Interface_Usr"

O pacote "Interface_Usr" é descrito no diagrama abaixo.

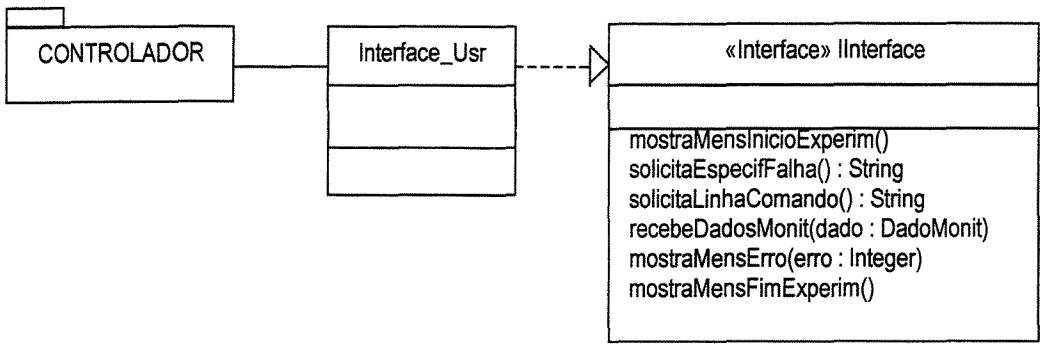


Fig. 14: estrutura do pacote "Interface_Usr"

O pacote *Interface_Usr* apenas contém métodos para realizar a interface com o usuário. Deve-se dizer que, atualmente, a ferramenta JACA utiliza uma interface feita em modo texto. Entretanto, futuras versões da ferramenta podem reescrever o pacote *Interface_Usr* para trabalhar com uma interface gráfica. Os métodos da classe *Interface_Usr*, que é a única do pacote, realizam o seguinte:

- *mostraMensInicioExperim():* mostra uma mensagem ao usuário informando que irá se iniciar o experimento de Injeção de Falhas.
- *solicitaEspecifFalha() : String:* solicita ao usuário que entre com o nome do arquivo de especificação de falhas.
- *solicitaLinhaComando() : String:* pede ao usuário para digitar a linha de comando que iria iniciar a execução do sistema sob teste. A ferramenta JACA usa essa linha de comando para iniciar a execução do sistema sob teste.
- *recebeDadoMonit(dado : DadoMonit):* esse método recebe um dado monitorado a partir da execução do sistema sob teste, que será mostrado para o usuário.
- *mostraMensErro(erro : Integer):* este método solicita à interface que mostre uma mensagem de erro ao usuário. Essa mensagem é definida por um código numérico, que seleciona qual mensagem de erro será mostrada dentre um conjunto de mensagens pré-existentes.
- *mostraMensFimExperim():* este método mostra ao usuário uma mensagem informando que o experimento de Injeção de Falhas terminou.

V.3.7. Pacote "Ger_Falha"

Abaixo está um diagrama representando o pacote "Ger_Falha".

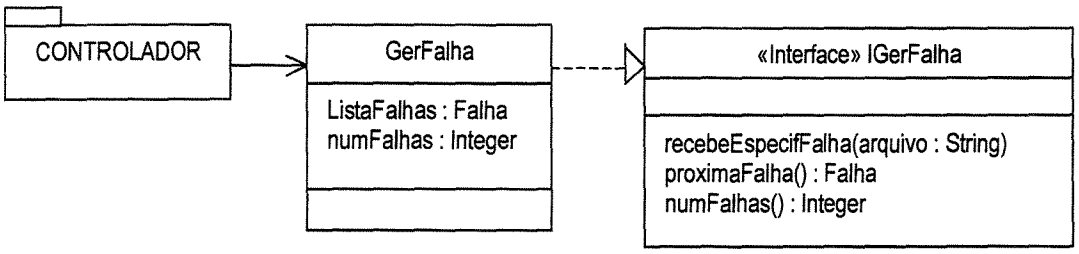


Fig. 15: estrutura do pacote "Ger_Falha"

O pacote *GerFalha* é muito simples, consistindo apenas de uma classe, *GerFalha*, que implementa a interface *IGerFalha*. O pacote tem a responsabilidade de armazenar informações sobre as falhas a serem injetadas. Ele faz isso da seguinte forma: primeiramente, *Controlador* invoca o método *recebeEspecifFalha()*, no qual ele passa para o objeto *GerFalha* um nome de arquivo onde está a especificação das falhas a serem injetadas. *GerFalha*, por sua vez, abre e lê esse arquivo, criando uma série de objetos *Falha* (vide sub-seção V.3.9) que ficam armazenados numa lista. Posteriormente, pode-se solicitar uma dessas falhas através da invocação do método *proximaFalha()*, sendo que a falha é retirada da lista. Pode-se saber o número de falhas que ainda restam invocando-se *numFalhas()*.

O formato do arquivo de especificação das falhas é o seguinte. Cada linha desse arquivo corresponde a uma falha, e dentro dessa linha os vários campos de informação estão separados por dois pontos (":"). Atualmente há três tipos de falha, e o primeiro é a falha de atributo, que é definido assim dentro do arquivo:

```
FalhaAtributo:<nome_do_atributo>:<nome_da_classe>:<máscara>:
<operação>:<padrão_de_repetição>:<início>:
```

A linha acima deve ser contida toda em uma linha só do arquivo: um caracter de fim de linha é considerado como o início da descrição de outra falha. O primeiro campo indica o tipo de falha, no caso, falha de atributo. O segundo, *nome_do_atributo*, indica qual o atributo que será alterado, dentro da classe definida por *nome_da_classe*. Sobre esse atributo dessa classe será realizada uma operação, definida por *operação*, sendo que o outro operador dessa operação é definido por *máscara*. *Padrão_de_repetição* indica qual é o padrão de repetição da falha, podendo ser <p>ermanente, <i>ntermitente ou <t>ransiente. Por fim, *início* diz a partir de que momento que se deve injetar essa falha: no caso, *início* deve ser um número que indica depois de quantas vezes tendo o atributo sido acessado que se deve fazer a injeção.

O segundo tipo de falha é a falha de retorno de método, na qual o valor de retorno de um método é alterado. Para se especificar essa falha, deve-se colocar no arquivo de especificação de falha a seguinte linha:

```
FalhaRetMetodo:<nome_do_metodo>:<nome_da_classe>:<máscara>:
<operação>:<padrão_de_repetição>:<início>:
```

Como no caso da falha de atributo, a linha acima também deve estar contida toda em uma linha. A linha acima é muito semelhante a linha da falha de atributo, exceto por dois aspectos apenas. O primeiro é *"FalhaRetMetodo"* colocado no início da linha para indicar que se trata da especificação de uma falha de retorno de método. O segundo é que o segundo campo da linha é o nome do método, *nome_do_metodo*, e não o nome do atributo. Os demais campos contêm as mesmas informações já descritas para falha de atributo.

Por fim, o terceiro tipo de falha é a falha de parâmetro. Essa falha altera o valor de um parâmetro dentro de uma chamada de método. Essa falha pode ser especificada pela seguinte linha:

```
FalhaParametro:<nome_do_metodo>:<num_parametro>:<nome_da_classe>:
<máscara>:<operação>:<padrão_de_repetição>:<início>:
```

Como nas outras falhas, a linha acima deve estar contida em uma linha apenas no arquivo de especificação. Essa linha é exatamente igual a linha para falha de retorno de método, exceto por um campo extra que ela possui. Entre os campos *nome_do_metodo* e *nome_da_classe* se localiza o campo *num_parametro*, que identifica em qual parâmetro da chamada de método será injetada a falha. Esse é um campo númerico, onde um número inteiro identifica o parâmetro. O primeiro parâmetro corresponde ao parâmetro número o (zero).

Para concluir, na tabela V tem-se a descrição das falhas com que a ferramenta JACA atualmente trabalha. Na tabela, "Identificador da Falha" é o nome que deve ser colocado no início de cada linha do arquivo de especificação de falhas para identificar qual é o tipo de falha que está sendo especificado.

Tabela V: falhas injetadas pela Ferramenta JACA

Tipo de Falha	*Identificador da Falha*	*Descrição*
Falha de Atributo	`FalhaAtributo`	Altera o valor de um atributo de uma dada classe, no momento em que este é lido.
Falha de Retorno de Método	`FalhaRetMetodo`	Altera o valor de retorno de uma chamada de um método de uma dada classe.
Falha de Parâmetro	`FalhaParametro`	Altera o valor de um parâmetro dentro de uma chamada de um método de uma dada classe.

V.3.8. Pacote "Ger_D_Monit"

Por fim, abaixo se descreve o último pacote da ferramenta JACA, o pacote "Ger_D_Monit".

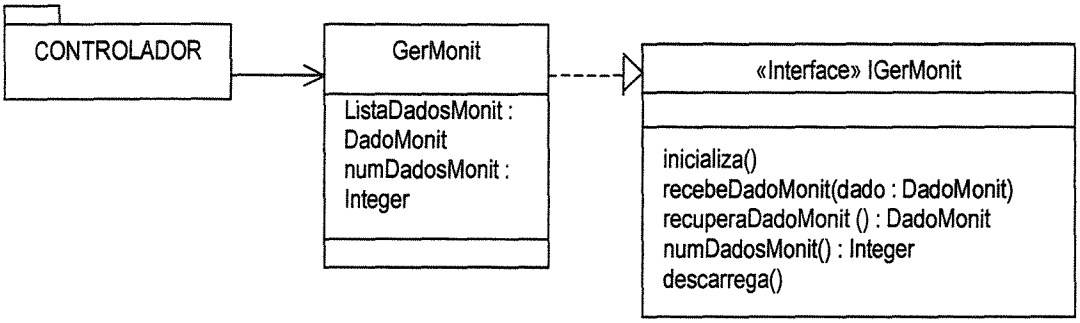


Fig. 16: estrutura do pacote "Ger_D_Monit"

O pacote é muito simples, consistindo simplesmente de um objeto da classe *GerMonit*. Entretanto, este pacote tem uma peculiaridade, com relação aos outros: ele não está no mesmo programa executável que todos os outros, ele é um programa executável por si só, e deve ser rodado antes da execução do programa que contém os demais pacotes. Isso acontece pelo seguinte fato. O pacote *GerMonit* guarda os dados monitorados a respeito do sistema sob teste numa lista em memória. Contudo, embora seja uma possibilidade remota, é sempre possível que uma falha injetada dentro do sistema sob teste cause um "crash" total do sistema, e se o pacote *GerMonit* estivesse rodando junto com os demais, e portanto *na mesma máquina*, os dados da monitorização se perderiam.

Por causa disso, *GerMonit* foi feito como um programa a parte para ser rodado em uma máquina em separado dos demais pacotes. Assim, se houver um *crash*, o pacote não irá perder seus dados, os quais poderão ser gravados num arquivo *logfile*. Se houver um término normal do experimento de Injeção de Falhas, no fim do experimento o *Controlador* irá mandar uma mensagem para *GerMonit* indicando que o experimento terminou e que ele pode gravar os dados monitorados no *logfile*. Caso a máquina tenha tido um "crash", e se deseje gravar os dados guardados no Gerenciador de Dados Monitorados, deve-se rodar o pequeno programa *Descarrega*, também escrito em Java e que acompanha a ferramenta, o qual irá comunicar ao Gerenciador que ele pode gravar os dados no *logfile*.

O pacote *GerMonit* contém apenas uma classe, a classe *GerMonit*, que implementa os métodos definidos na interface *IGerMonit*. Esses métodos são os seguintes:

- *recebeDadoMonit(dado : DadoMonit):* recebe um dado monitorado e armazena na lista em memória. Deve-se dizer que esse método é acessado remotamente, via Java RMI (Remote Method Invocation) [Sun99], o que permite que o *Controlador*, que está em outra máquina, transmita via rede o dado monitorado para ser armazenado.
- *recuperaDadoMonit() : DadoMonit:* retorna um dado monitorado da lista em memória. A lista funciona como uma fila: o primeiro dado a chegar é o primeiro dado a sair.
- *numDadoMonit() : Integer:* retorna o número de dados monitorados que estão armazenados na lista.

- *descarrega():* pega todos os dados monitorados da lista e escreve seus conteúdos no arquivo *logfile*, esvaziando a lista. O *Controlador* invoca este método no fim do experimento de Injeção de Falhas.

V.3.9. Outras classes

Além das classes citadas acima, existem outras classes que são necessárias para o funcionamento da ferramenta JACA, mas que não estão em nenhum dos pacotes acima. No caso, essas classes servem como "containers" de dados que são trocados entre os pacotes da ferramenta. Um grupo dessas classes "containers" de dados são a classe *Falha* e suas sub-classes, que armazenam informações sobre falhas a serem injetadas no processo de Injeção de Falhas (veja a *fig. 17*).

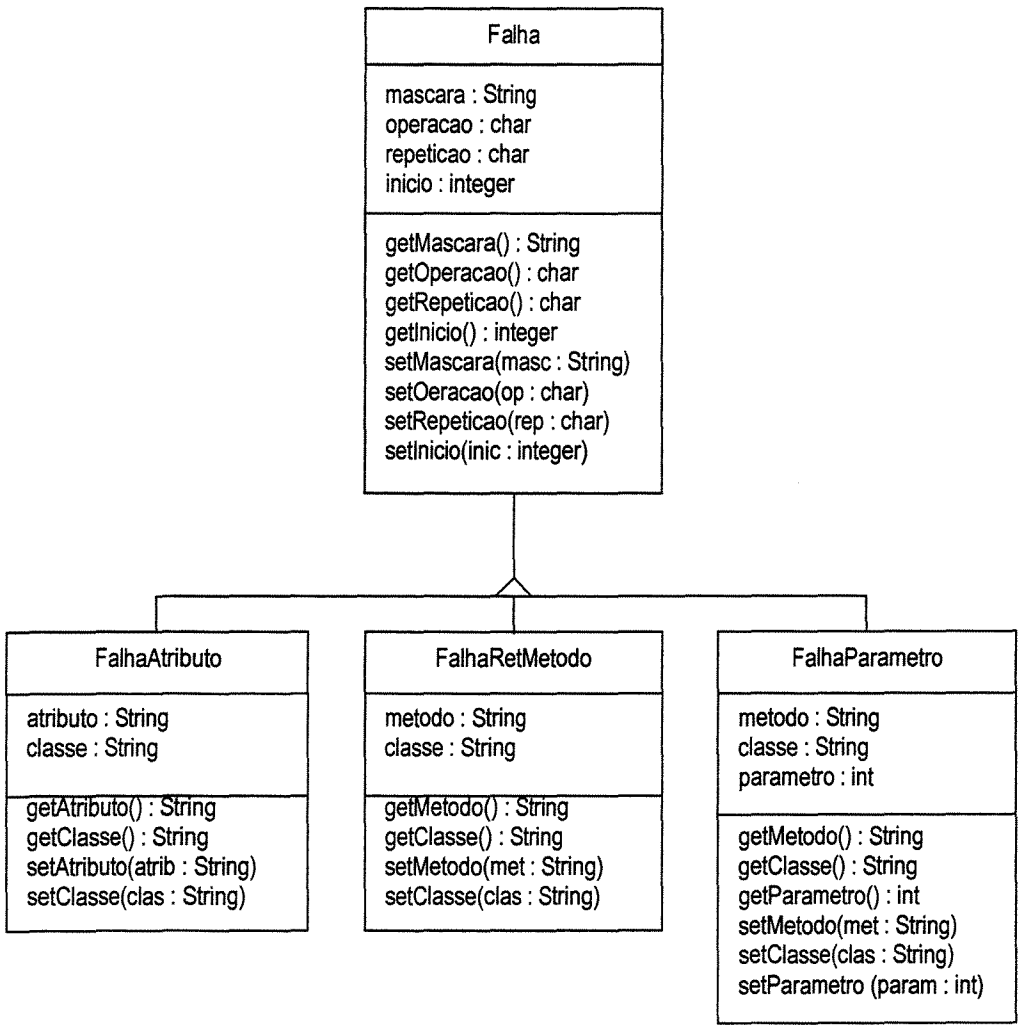


Fig. 17: classe *Falha* e suas subclasses

A classe *Falha* e suas sub-classes guardam dados sobre uma falha a ser injetada, sendo que as sub-classes guardam dados específicos para cada tipo de falha. Já a própria classe *Falha* guarda dados comuns a todas as falhas, no caso:

- *mascara:* um valor sobre o qual se realizará uma operação sobre um dado a ser alterado no sistema sob teste;
- *operacao:* definição de uma operação a ser realizada sobre um dado a ser alterado no sistema sob teste;
- *repeticao:* define um padrão de repetição para a falha;
- *inicio:* define um tempo para se iniciar a injeção da falha em questão.

Nas atuais sub-classes de *Falha*, se define que dado do sistema sob teste será alterado para se simular uma falha de software. A primeira sub-classe é *FalhaAtributo*, que define um atributo de uma classe que deverá ser alterado dentro da execução do sistema sob teste (de acordo com *mascara*, *operacao*, *repeticao* e *inicio*). Outra sub-classe é *FalhaRetMetodo*, que define que método de que classe terá seu valor de retorno alterado. A última sub-classe é *FalhaParametro*, que especifica qual parâmetro de que método de uma dada classe que terá seu valor modificado.

Outro grupo de classes necessário para a operação da ferramenta JACA é a classe *DadoMonit* e suas sub-classes. Essas classes empacotam dados obtidos na monitorização do sistema sob teste, dados esses que são posteriormente armazenados dentro do pacote "Ger_D_Monit". Essas classes podem ser examinadas no diagrama da *fig. 18*.

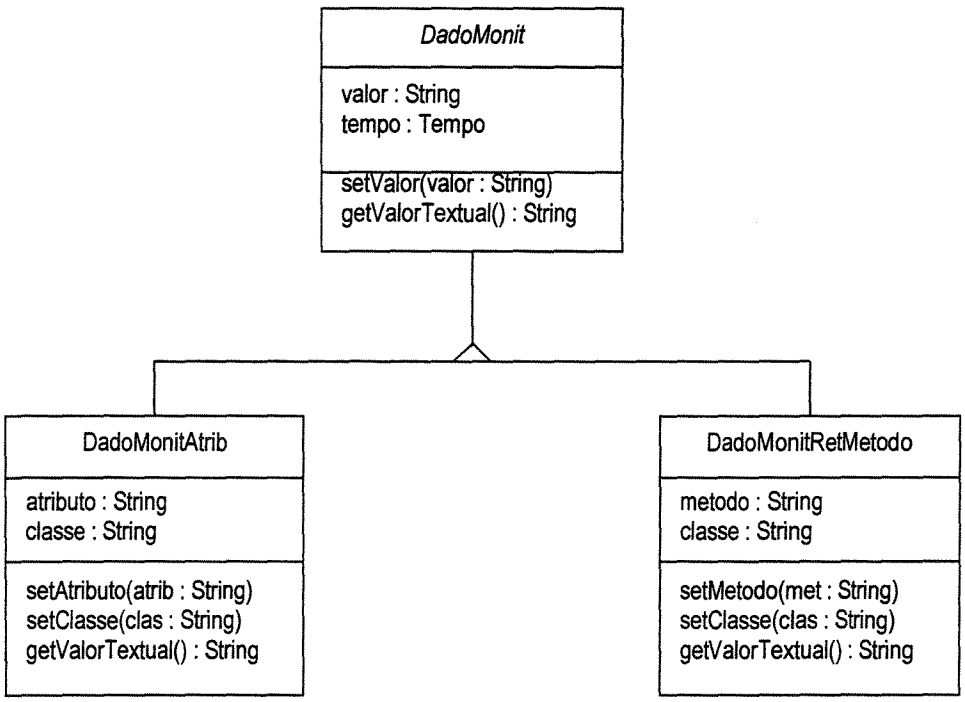


Fig. 18: descrição da classe DadoMonit e de suas subclasses DadoMonitAtrib e DadoMonitRetMetodo

A classe *DadoMonit* e suas sub-classes se destinam a armazenar dados obtidos na monitorização do sistema sob teste. A própria classe *DadoMonit* é abstrata, servindo apenas para definir alguns métodos e atributos para as sub-classes. Esses atributos são *valor*, onde é armazenado o valor obtido na monitorização, e *tempo*, que armazena o tempo de criação do objeto, e portanto o tempo no qual o *valor* foi obtido. Mais especificamente, esse tempo corresponde ao tempo *no qual o dado monitorado foi observado*. Por exemplo, se o dado corresponde a um valor de retorno de método, o tempo se refere a hora precisa na qual o sistema sob teste gerou esse valor de retorno. Além disso, a classe *DadoMonit* define um método, *getValorTextual()*, o qual retorna uma linha de texto contendo as informações do objeto. As atuais sub-classes de *DadoMonit* são *DadoMonitAtrib*, que guarda dados monitorados referentes a valores de atributos, e *DadoMonitRetMetodo*, que guarda dados monitorados a respeito de valores de retorno de métodos.

V.4. Implementação e codificação da ferramenta JACA

Tendo se definido o projeto da ferramenta JACA, passou-se a sua implementação na linguagem Java. Todas as classes acima foram codificadas, e dessa forma criou-se um protótipo da ferramenta JACA sobre o qual se podiam realizar testes, que estão descritos no próximo capítulo. Além disso, já se criou uma adaptação da ferramenta, o Demonstrativo de Injeção de Falhas em baixo nível, que é descrito na seção seguinte, e está se trabalhando em uma nova versão da JACA, a JAGUAR (vide seção V.7).

Não se colocou nesta dissertação o código da ferramenta JACA porque isso tomaria um espaço muito grande (atualmente a ferramenta JACA tem mais de 5800 linhas de código), além do que as informações mais importantes sobre a ferramenta podem ser obtidas nos diagramas de objeto expostos na seção anterior. Entretanto, caso o leitor queira inspecionar o código, deve consultar o Apêndice A desta dissertação, onde é explicado como se pode obter a ferramenta JACA.

V.5. Resultados do uso do Sistema de Padrões

Um dos objetivos da construção da ferramenta JACA era verificar se o Sistema de Padrões para Injeção de Falhas por software realmente auxilia no desenvolvimento de ferramentas. De fato, terminado o projeto e codificação da JACA, puderam ser obtidas algumas conclusões sobre o uso do Sistema de Padrões:

- Seu uso permitiu economizar tempo no projeto da ferramenta JACA. Parte do trabalho já estava feito, pois a arquitetura da JACA era a determinada pelo padrão de arquitetura "Injetor de Falhas", e as estruturas dos pacotes Injetor e Monitor eram dadas pelos padrões de projeto "Injetor" e "Monitor", respectivamente. Os demais pacotes eram todos menos complexos que esses, e portanto, puderam ser rapidamente projetados.
- O fato da estrutura proposta pelo padrão de projeto "Injetor" ser extensível foi de ajuda ainda durante o projeto. Com a ferramenta JACA já passando pelos seus testes iniciais, decidiu-se adicionar mais um tipo de falha a ser injetada, as falhas de parâmetro. Adicionar essa nova falha foi um processo relativamente simples, que implicou na criação das classes FalhaParametro e InjetorFalhaParametro, bem como na modificação da classe GerInjet (para gerenciar essa nova falha) e da classe GerFalha (para reconhecer essa nova falha

no arquivo de especificação de falhas). Dessa maneira, não foi necessário reescrever todo o pacote Injetor.

- Outro benefício do uso do sistema de padrões foi o fato de a arquitetura sugerida pelo padrão de arquitetura "Injetor de Falhas" ser altamente modular. Isso foi útil quando, durante o projeto, percebeu-se que seria melhor ter o Gerenciador de Dados Monitorados rodando numa máquina diferente da própria ferramenta JACA. Como os subsistemas, dentro do padrão "Injetor de Falhas", rodam independentemente um dos outros, o fato de se reescrever o pacote GerMonit para rodar remotamente implicou em alterações apenas no pacote Controlador.

Em resumo, o uso do Sistema de Padrões para Injeção de Falhas por software foi proveitoso. Portanto, o sistema de padrões realmente é útil no desenvolvimento de novas ferramentas de Injeção de Falhas, o que era um dos objetivos iniciais deste projeto.

V.6. Demonstrativo de Injeção de Falhas em baixo nível

Como se viu no capítulo III, já há um número de ferramentas de injeção de falhas que trabalham com falhas em baixo nível. Isto é, falhas mais relacionadas com aspectos do hardware do sistema, tal como registradores, memória, barramentos, etc. Entretanto, ainda não há muitas ferramentas que trabalhem com falhas em alto nível, ou seja, com aspectos mais relacionados ao software do sistema, tais como valores de retorno de métodos, valores de atributos, de parâmetros, etc. Por isso, decidiu-se escrever a ferramenta JACA para trabalhar com esse último tipo de falhas.

Entretanto, ainda é uma grande preocupação dos que trabalham com Injeção de Falhas a possibilidade de se injetar falhas em baixo nível. Para isso, então, decidiu-se construir um demonstrativo baseado na ferramenta JACA, que mostraria que a ferramenta pode trabalhar com falhas em baixo nível se assim se desejar.

O trabalho na construção desse demonstrativo provou que as alterações necessárias seriam pequenas. Primeiramente, deve-se explicar que a linguagem Java não impede o uso de recursos específicos da máquina. Para isso, pode-se definir os chamados *métodos nativos*, que podem ser implementados em um programa a parte, escritos em C ou Assembly [Sun97]. No caso, decidiu-se criar uma classe que serviria apenas para guardar os métodos nativos necessários para se injetar as falhas em baixo nível. Essa classe não teria atributos, e serviria apenas como um repositório de rotinas em Assembly. Cada Injetor Físico, então, teria associado um objeto dessa classe. A listagem dessa classe pode ser vista abaixo:

```
public final class LowLevelFunctions {

    public native void falhaBaixoNivel();

    static {
        System.loadLibrary("LowLevelFunctions");
    }

}
```

Posteriormente, deve-se implementar o método num módulo em linguagem C ou Assembly. Esse módulo será carregado quando da carga da classe *LowLevelFunctions*, como está indicado dentro da cláusula *static* da classe. Para o demonstrativo, implementou-se o método utilizando-se uma "casca" de linguagem C, para facilitar a interface com a linguagem Java, e essa "casca" continha uma pequena rotina em Assembly. Essa rotina foi escrita para rodar num processador Intel que segue a arquitetura IA-32 (Intel 80386, 80486, Celeron, Pentium, Pentium II e Pentium III), no qual esteja rodando o sistema operacional Linux. Como pode se ver, uma desvantagem da Injeção de Falhas em baixo nível é que amarra a ferramenta JACA a uma configuração específica de sistema. Quando o método nativo é invocado, a máquina virtual Java é deixada de lado e passa-se a se executar o código em Assembly. Ao final deste, retorna-se à execução da máquina virtual Java, porém esta não fará nenhuma troca de contexto ou qualquer outra verificação. Dessa forma, qualquer modificação realizada pela rotina em Assembly continuará valendo quando se retornar à execução do código em Java. Pode-se mesmo fazer operações sobre objetos Java de dentro da rotina em Assembly, que essas continuarão valendo quando se sair desse código [Sun97].

Essa rotina em Assembly realiza o seguinte. Primeiramente, ela move um código específico para o registrador EAX (de acordo com o manual da instrução CPUID [Int01]) e em seguida executa a instrução CPUID. Conforme definido pela Intel, para a arquitetura IA-32, essa instrução escreve simultaneamente nos registradores EAX, EBX, ECX e EDX. Corresponde, portanto, a quatro instruções MOVs simultâneas, comprovando a capacidade da ferramenta JACA de lidar com os registradores do sistema. Nesses registradores ficam escritos códigos correspondendo a informações sobre o processador e o sistema no qual o programa está rodando. A rotina em Assembly, então, acessa o conteúdo do registrador EDX, a fim de descobrir qual é o processador em que está rodando, se é um Intel Celeron, um Intel Pentium, um Intel Pentium II, etc.

Portanto, pode-se reescrever a ferramenta, sem ter que alterar suas características fundamentais, fazendo um injetor de falhas em alto nível passar a ser um injetor de falhas em baixo nível. Inclusive, a base para isso já existe. Basta tomar o demonstrativo para Injeção de Falhas em baixo nível e adicionar mais falhas a ele, bem como o código em Assembly para essas falhas. Deve-se acrescentar que o demonstrativo de Injeção de Falhas em baixo nível não perdeu sua capacidade de fazer Injeção de Falhas em alto nível. Ele pode injetar as mesmas falhas que a versão original da JACA *e também* as falhas em baixo nível.

V.7. JAGUAR

Com o Demonstrativo de Injeção de Falhas em baixo nível, se mostrou a capacidade de se reescrever a ferramenta JACA para trabalhar com outros tipos de falha. Entretanto, desejava-se também demonstrar que a ferramenta JACA poderia ser reescrita para trabalhar com outro protocolo de meta-objetos. Dessa maneira, se mostraria que a ferramenta aqui construída não estaria presa ao protocolo Javassist. Para essa nova versão da JACA, se escolheu o protocolo de meta-objetos *Guaraná* [Oli98]. Decidiu-se dar um novo nome a essa versão, que é JAGUAR (JAca baseada no GUARaná) [KiM01]. A construção da JAGUAR foi objeto de um projeto de Iniciação Científica dentro do Instituto de Computação da Universidade Estadual de Campinas, que presentemente ainda está em curso. Entretanto, os primeiros resultados desse projeto confirmam que se pode reescrever a ferramenta para se trabalhar com outro protocolo de meta-objetos. Dessa maneira, qualquer usuário futuro poderá ter a opção de com qual ferramenta irá trabalhar, se com a JACA original, baseada no Javassist, ou com a JAGUAR, baseada no Guaraná.

V.8. Resumo

Neste capítulo, mostrou-se o projeto da ferramenta JACA. De início, formulou-se uma série de requisitos para a mesma. Pode-se citar alguns mais importantes, tais como: a ferramenta deveria tomar por base, para seu projeto, o Sistema de Padrões para Injeção de Falhas por software. Além disso, deveria ser altamente adaptável, de maneira que se a ferramenta não satisfizesse exatamente os requisitos do usuário, poderia ser reescrita para fazê-lo. Também vale a pena dizer que a ferramenta JACA deveria ser escrita em Java e utilizar reflexão computacional para fazer a injeção de falhas, tal como a ferramenta FIRE [Ros98].

Levando-se esses requisitos em consideração, passou-se ao projeto propriamente dito da ferramenta. Adotou-se a arquitetura sugerida pelo padrão de arquitetura "Injetor de Falhas". A ferramenta JACA foi arquitetada tendo sete pacotes, correspondendo aos subsistemas do padrão de arquitetura. Esses pacotes são: Injetor, Monitor, Ativador, Controlador, Interface_Usr, Ger_Falha e Ger_D_Monit. Dentro desses pacotes ficam classes que realizam as funções especificadas para esses subsistemas de acordo com o definido no padrão "Injetor de Falhas". Os pacotes Injetor e Monitor, além disso, possuem as estruturas descritas pelos padrões de projeto "Injetor" e "Monitor", respectivamente. Além das classes contidas nos pacotes, há mais classes, usadas para empacotar dados que devem ser passados entre pacotes. São as classes Falha (e suas sub-classes) e DadoMonit (e suas sub-classes). Posteriormente, traduziu-se esse projeto para código em linguagem Java.

Como conseqüência desse trabalho, pode-se avaliar a real utilidade do Sistema de Padrões para construir ferramentas de Injeção de Falhas. Os resultados foram bons, e demonstraram que o uso dos padrões realmente economizou tempo no desenvolvimento. Um outro trabalho também conduzido após a codificação da versão original da JACA foi construir uma versão da ferramenta que demonstrasse que a mesma podia trabalhar com falhas em baixo nível. Mais explicitamente, com falhas mais ligadas ao hardware do sistema sob teste. Foi construído tal demonstrativo, sendo que ele altera valores de registradores do processador e invoca uma instrução CPUID, dos processadores da arquitetura IA-32 da Intel, para dizer que é o processador da arquitetura IA-32 no qual o sistema está rodando. Assim, pode-se demonstrar que a ferramenta JACA também pode ser adaptada para trabalhar também com falhas em baixo nível. Além disso, a construção da JAGUAR comprovou que a ferramenta JACA pode ser reescrita para trabalhar utilizando um outro protocolo de meta-objetos, no caso o Guaraná.

CAPÍTULO VI: TESTES DA FERRAMENTA JACA

VI.1. Organização dos testes

Os testes da ferramenta JACA se organizaram da seguinte forma. Primeiramente, procedeu-se a uma fase de testes iniciais, onde se procurou verificar se a ferramenta operava corretamente. Isto é, se ela não tinha falhas de software. Após essa fase, passou-se a uma fase de teste de aplicação. Nessa etapa se buscou um programa escrito em Java já existente, e ele foi submetido à Injeção de Falhas pela JACA. Dessa forma, pode-se testar a ferramenta mais proximamente de suas reais condições de uso e se medir sua performance e sua praticidade para testar sistemas. Os resultados desses testes estão detalhados nas seções seguintes.

VI.2. Testes iniciais

VI.2.1. Testes na versão original da ferramenta JACA

Os primeiros testes se destinaram a verificar se a ferramenta estava funcionando corretamente. Ou seja, inicialmente estava se procurando por *bugs* ou falhas de software na própria ferramenta. Os resultados desses testes foram úteis para tirar as falhas de software da ferramenta, sendo que agora a ferramenta JACA funciona corretamente. Deve-se observar que nessa primeira fase dos testes se trabalhou com a versão original da ferramenta JACA, isto é, com a versão desenvolvida para trabalhar exclusivamente com falhas em alto nível.

Para fazer esses testes, era necessário se dispor de um sistema muito simples para submeter a teste por Injeção de Falhas pela ferramenta JACA. Construiu-se esse sistema, e ele consiste de apenas três classes, que podem ser vistas na *fig. 19*.

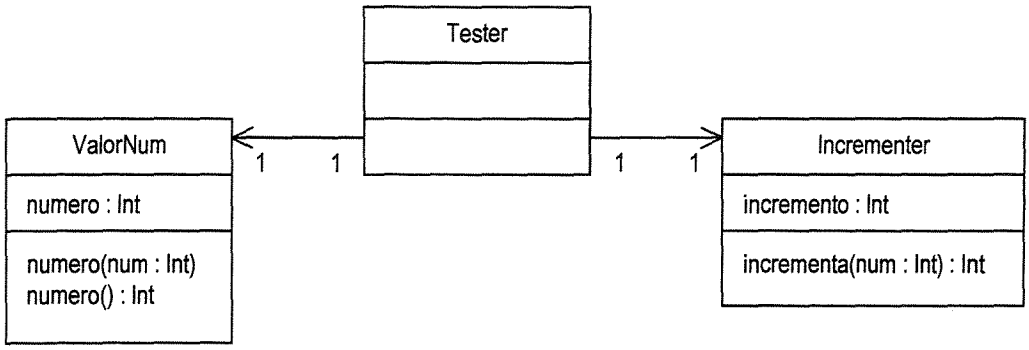


Fig. 19: diagrama de objetos do sistema de teste

A classe que controla a execução do sistema se chama *Tester*, e apenas contém o método *main()* para iniciar a execução da aplicação.

A segunda classe do sistema se chama *Incrementer*, e faz o seguinte: quando de sua construção, deve-se submeter um parâmetro numérico inteiro ao construtor, parâmetro esse que fica armazenado no atributo *incremento*. A classe *Incrementer* dispõe de um único método, chamado *incrementer(num : Integer)*. Esse método recebe um parâmetro númerico inteiro, e retorna um inteiro que é esse parâmetro mais o valor de *incremento*. O método *main()* de *Tester* cria um objeto *incrementer*, com o parâmetro de construção 3, e invoca o método *incrementer()* cinco vezes. Além disso, *Tester* também cria um objeto *ValorNum*. A classe *ValorNum* apenas armazena um valor numérico inteiro, e foi incluída aqui para demonstrar como a ferramenta JACA pode monitorar classes além das que estão sofrendo Injeção de Falhas. Neste caso, *Tester* armazena vários valores diferentes em *ValorNum*. Mostra-se abaixo a saída de uma execução de *Tester*, quando é executado isoladamente.

```
[/home/lsd/nelsongm/jaca]=> java Tester


Incrementos de 3:

Este numero: 95
Foi incrementado para: 98

Este numero: 98
Foi incrementado para: 101

Este numero: 101
Foi incrementado para: 104

Este numero: 104
Foi incrementado para: 107

Este numero: 107
```

```
Foi incrementado para: 110


-----> Executado... <-----
```

Tomou-se *Tester* como o sistema a ser testado pela ferramenta JACA. Abaixo se mostra a saída de duas execuções da ferramenta JACA, quando esta estava testando por Injeção de Falhas a aplicação *Tester*. Mostra-se também o arquivo de especificação de falhas, mostrando qual falha será injetada em *Tester* de cada vez. Mostra-se também o *logfile* gerado pela ferramenta JACA.

Arquivo de especificação de falhas:

```
FalhaAtributo:incremento:incrementer:2:*:p:0:
```

Saída da ferramenta JACA:

```
[/home/lsd/nelsongm/jaca]=> java javassist.Loader Controlador

===> Ferramenta JACA (prototipo) <===


Iniciando experimento de injecao de falhas...

Nome do arquivo de especificacao de falhas:
Entre linha de comando do sistema sob teste:


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


Incrementos de 3:

Este numero: 95
Foi incrementado para: 101

Este numero: 101
Foi incrementado para: 107

Este numero: 107
Foi incrementado para: 113

Este numero: 113
Foi incrementado para: 119

Este numero: 119
Foi incrementado para: 125


-----> Executado... <-----


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
Experimento de injecao de falhas encerrado.


Execucao da ferramenta JACA terminada
```

Arquivo logfile gerado:

```
[/home/lsd/nelsongm/jaca]=> more logfile
Falha Atributo -- atributo: incremento, classe: Incrementer (2*,
p).
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #1 (classe: ValorNum) -
- valor do atributo "numero": 3
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #1 (classe: ValorNum) -
- valor do atributo "numero": 6
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor do atributo "incremento": 3
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #1 (classe: ValorNum) -
- valor do atributo "numero": 3
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 101
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 107
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 113
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 119
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 125
(Thu May 17 01:47:31 GMT-03:00 2001) Objeto #1 (classe: ValorNum) -
- valor do atributo "numero": 9
```

Como pode se ver no exemplo acima, especificou-se uma falha de atributo a ser inserida. No caso, era uma falha permanente, referente ao atributo *incremento*: cada vez que esse atributo fosse lido, em vez de se ler seu valor correto, 3, seria lido o dobro disso, 6. Abaixo temos outro exemplo.

Arquivo de especificação de falhas:

```
FalhaRetMetodo:incrementa:incrementer:50:+:t:2:
```

Saída da ferramenta JACA:

```
[/home/lsd/nelsongm/jaca]=> java javassist.Loader Controlador

===> Ferramenta JACA (prototipo) <===


Iniciando experimento de injecao de falhas...

Nome do arquivo de especificacao de falhas:
Entre linha de comando do sistema sob teste:
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Incrementos de 3:

Este numero: 95
Foi incrementado para: 98

Este numero: 98
Foi incrementado para: 101

Este numero: 101
Foi incrementado para: 154

Este numero: 154
Foi incrementado para: 157

Este numero: 157
Foi incrementado para: 160


-----> Executado... <-----


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Experimento de injecao de falhas encerrado.


Execucao da ferramenta JACA terminada
```

Arquivo logfile gerado:

```
[/home/lsd/nelsongm/jaca]=> more logfile
Falha RetMetodo -- metodo: incrementa(), classe: Incrementer (50+,
t).
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #1 (classe: ValorNum) -
- valor do atributo "numero": 3
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #1 (classe: ValorNum) -
- valor do atributo "numero": 6
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor do atributo "incremento": 3
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #1 (classe: ValorNum) -
- valor do atributo "numero": 3
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 98
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 101
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 154
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 157
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #2 (classe:
Incrementer) -- valor de retorno de "incrementa": 160
```

```
(Thu May 17 02:02:01 GMT-03:00 2001) Objeto #1 (classe: ValorNum) -
- valor do atributo "numero": 9
```

Como pode se ver, esperou-se que o método *incrementa(num : Integer)* já tivesse sido chamado duas vezes, em seguida injetou-se a falha. Por essa falha ter sido definida como transiente, ela só alterou uma vez o valor de retorno de *incrementa(num : Integer)*, em seguida voltando o método a apresentar seu comportamento normal.

Além dos testes relatados acima, foram realizados outros. Exploraram-se combinações de diferentes tipos de falhas com diferentes padrões de repetição, valores de início de injeção, operações a serem realizadas, valores de máscara, etc. Citou-se apenas os casos de teste acima para mostrar como a ferramenta JACA está funcionando corretamente agora, e para dar ao leitor uma idéia de como é de fato a operação da ferramenta JACA.

Os testes realizados com *Tester* realizaram seu objetivo primário, que era remover as falhas de software da ferramenta JACA. Além disso, como um resultado adicional, os testes apontaram para a necessidade de alguns pequenos aperfeiçoamentos na ferramenta JACA:

- Modificou-se o processo de gravação do *logfile* para este conter mais informação, tal como a identificação dos objetos onde ocorriam os eventos monitorados e também sobre qual falha estava sendo injetada em qual momento. Essas informações já eram armazenadas no Gerenciador de Dados Monitorados, apenas não eram gravadas no *logfile*.
- Melhoria no processo de leitura do arquivo de especificação de falhas. Esse arquivo tinha que ser escrito segundo uma formatação muito rígida, senão a ferramenta simplesmente não lia as falhas a serem injetadas. Agora essa formatação já é mais livre.
- A criação de um Gerenciador de Dados Monitorados local. Como foi dito, o Gerenciador de Dados Monitorados fica rodando numa máquina em separado da ferramenta. No início da execução da JACA, o Controlador tenta se comunicar com o Gerenciador de Dados Monitorados. Se não conseguisse, anteriormente o Controlador simplesmente terminava a execução. Agora, ele informa o usuário que não consegue se comunicar, e instancia um Gerenciador de Dados Monitorados que roda na mesma máquina da ferramenta, e continua rodando. O usuário é avisado que se o sistema travar completamente, os dados poderão ser perdidos. Contudo, dessa forma, pode-se realizar os testes e se obter os dados sobre o sistema.

VI.2.2. Testes no demonstrativo de Injeção de Falhas em baixo nível

Os testes iniciais executados acima foram efetuados também com o demonstrativo de Injeção de Falhas em baixo nível da JACA. Tomou-se o mesmo programa para teste, e se colocou a definição de uma falha em baixo nível no arquivo de especificação de falhas. Em seguida, rodou-se a ferramenta JACA, num computador com o processador Intel Pentium, rodando o sistema operacional Linux. Pode-se ver a saída da ferramenta JACA, para esse caso de teste, abaixo:

```
[/home/lsd/nelsongm/jaca]=> java javassist.Loader Controlador
```

```
===> Ferramenta JACA (prototipo) <===


Iniciando experimento de injecao de falhas...

Nome do arquivo de especificacao de falhas:
Entre linha de comando do sistema sob teste:


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


Incrementos de 3:

Este numero: 95
Foi incrementado para: 98

Este numero: 98

###-> Teste de CPUID

###-> Valores dos registradores - EAX: 1670, EBX: 2, ECX: 0, EDX:
58980863

###-> Tipo do processador: Intel Pentium

Foi incrementado para: 101

Este numero: 101
Foi incrementado para: 104

Este numero: 104
Foi incrementado para: 107

Este numero: 107
Foi incrementado para: 110


-----> Executado... <-----


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Experimento de injecao de falhas encerrado.


Execucao da ferramenta JACA terminada
```

Como pode se ver, após a segunda chamada do método *incrementa*, de *Incrementer*, foi injetada a falha em baixo nível. Ela escreveu no registrador EAX o código necessário para a instrução CPUID retornar a identificação do processador, executou a

instrução, e em seguida leu o conteúdo dos registradores EAX, EBX, ECX e EDX. A partir do valor gravado no registrador EDX, pode-se descobrir que o processador em questão era um Intel Pentium. Assim como no caso dos testes iniciais descritos na sub-seção anterior, este não foi o único caso de teste aplicado no demonstrativo de Injeção de Falhas em baixo nível: aplicou-se uma combinação de casos de teste, para se exercitar as diferentes opções de Injeção de Falhas da ferramenta.

VI.3. Testes de aplicação

Efetuados os testes iniciais descritos na seção acima, procurou-se verificar como a ferramenta JACA operaria testando um sistema de maior complexidade. Passou-se, então, a procurar-se uma aplicação em Java que pudesse ser testada pela ferramenta. Encontrou-se essa aplicação no *site* da Sun Microsystems. A empresa mantém em seu *site* uma série de aplicações para fazer demonstrações da capacidade da linguagem Java. Escolheu-se então a aplicação *SpreadSheet*, que é uma pequena planilha eletrônica rodando como um *applet*. Essa aplicação apresentava a vantagem de ter seu código aberto e ser relativamente simples, possibilitando que se entendesse seu funcionamento.

Tendo se escolhido a aplicação, passou-se aos testes. Foi criado um pequeno script, que era passado à aplicação *SpreadSheet*, de tal maneira a preencher quatro células com valores, criar duas fórmulas baseadas nesses valores e mais uma fórmula baseada nas fórmulas anteriores, e em seguida se fazer o recálculo de tudo isso. Dessa forma, se tinha uma carga de trabalho imposta a planilha. Definido isso, estabeleceu-se como a aplicação rodaria, para se verificar o desempenho da ferramenta JACA em diversas situações. Decidiu-se que ela rodaria de cinco diferentes formas:

(1) A aplicação *SpreadSheet* rodaria sozinha.
(2) A aplicação *SpreadSheet* rodaria dentro da ferramenta JACA, mas não seria feita nem injeção de falhas nem a aplicação seria monitorada.
(3) A aplicação *SpreadSheet* rodaria dentro da ferramenta JACA, não seria feita injeção de falhas mas a aplicação seria monitorada.
(4) A aplicação *SpreadSheet* rodaria dentro da ferramenta JACA, seria feita injeção de falhas mas a aplicação não seria monitorada.
(5) A aplicação *SpreadSheet* rodaria dentro da ferramenta JACA, e seria feita injeção de falhas e monitorização.

Para cada um dos casos acima, rodou-se a aplicação 60 vezes. Não se utilizou um script do sistema operacional para se fazer isso: cada uma das vezes iniciou-se manualmente a execução, a fim de se ficar o mais próximo possível das reais condições de uso do sistema. Para cada vez se registrava o tempo de execução para *SpreadSheet* realizar o script indicado anteriormente. Dessa maneira se pode ter uma medida qualitativa do tempo de execução da aplicação quando rodando dentro da ferramenta. Os dados obtidos estão na tabela VI, a seguir.

Tabela VI: tempos de execução para casos de teste (em segundos)

Casos de teste	*Mínimo*	*Máximo*
(1) aplicação rodando isoladamente	2,6	12,2
(2) rodando dentro da JACA, sem injeção nem monitorização	2,4	8,8
(3) rodando dentro da JACA, sem injeção mas com monitorização	4,6	12,6
(4) rodando dentro da JACA, com injeção mas sem monitorização	2,9	8,2
(5) rodando dentro da JACA, com injeção e monitorização	3,2	11,0

Como pode se ver nos dados acima, houve um pequeno aumento nos tempos de execução, quando se compara a aplicação rodando isoladamente e rodando dentro da JACA. Entretanto, o tempo de execução dentro da ferramenta JACA continua razoável, já que os tempos máximos foram próximos do tempo máximo da aplicação rodando isoladamente, portanto a ferramenta não chega a comprometer a execução normal de uma aplicação sob teste. Pode-se observar também que a ferramenta, quando não está nem injetando falhas nem monitorando aumenta pouco o tempo de execução do sistema sob teste. Ocorre um atraso quando se injeta falhas na aplicação, mas o atraso maior ocorre quando se faz a monitorização do sistema. Quando é feita apenas a monitorização, a sistema é executado em média mais lentamente que nos outros casos. Quando a monitorização é feita conjuntamente com a injeção, o atraso diminui porque a injeção de falhas otimiza um pouco a monitorização. Por exemplo, quando um método irá retornar um valor alterado por uma falha, recolhe-se diretamente esse valor alterado como dado monitorado.

VI.4. Resumo

Os testes da ferramenta JACA comprovaram sua correção e praticidade. Numa fase inicial de testes, procurou-se ver se a ferramenta não tinha falhas de software. Esses testes foram conduzidos usando-se um pequeno programa em Java, escrito especialmente para tal. Testou-se esse programa com a JACA, de forma a se exercitar todos os possíveis casos de teste da ferramenta. Esses testes iniciais revelaram algumas falhas de software, que foram removidas. Mostraram também que alguns aperfeiçoamentos podiam ser feitos, os quais foram implementados. Além disso, testou-se também o demonstrativo de Injeção de Falhas em baixo nível da JACA, que revelou que a ferramenta pode injetar falhas em baixo nível, e também falhas em alto nível.

Posteriormente, passou-se a uma fase de testes de aplicação. Nessa etapa, procurou-se uma aplicação real em Java para se testar com a JACA. Com isso, era possível demonstrar melhor a eficiência da ferramenta JACA. Encontrou-se a aplicação *SpreadSheet* para isso. Foram efetuados os testes com essa aplicação, e resultaram que a ferramenta se comportou como esperado, isto é, sem causar um atraso irremediável no tempo de execução de um sistema. Assim, pode-se verificar a utilidade real da ferramenta JACA para conduzir testes por Injeção de Falhas.

CAPÍTULO VII: CONCLUSÕES E TRABALHOS FUTUROS

VII.1. Conclusões

Conforme foi dito na Introdução desta dissertação (seção I.3.), o objetivo mais geral deste trabalho era dar o máximo de recursos ao desenvolvedor para usar Injeção de Falhas. Pode-se dizer que esse objetivo foi alcançado. Criaram-se dois recursos importantes para o uso de Injeção de Falhas, que são o Sistema de Padrões para Injeção de Falhas por software e a ferramenta JACA. Munidos desses dois recursos, os desenvolvedores terão muito mais condições de poderem testar por Injeção de Falhas seus sistemas, podendo então ter uma avaliação mais precisa do funcionamento de seus sistemas na presença de falhas.

Deve-se acrescentar que esses recursos foram maturados pelo uso. De fato, o Sistema de Padrões para Injeção de Falhas por software foi realmente utilizado para desenvolver uma ferramenta de Injeção de Falhas. Com isso, pode-se ter uma garantia maior sobre sua real utilidade. Um processo similar ocorreu com a ferramenta JACA. Os testes pelos quais a ferramenta foi submetida verificaram que ela pode implementar testes por Injeção de Falhas no teste de um dado sistema. Também foi observado que ela pode ser alterada, sem necessitar de uma reescrita geral de seu código, para trabalhar com diferentes requisitos para os testes. Mesmo que esses requisitos incluam injetar falhas relacionadas com o hardware do sistema sob teste. Logo, a ferramenta JACA pode ser um recurso muito útil para os desenvolvedores.

Por fim, com a criação do Sistema de Padrões para Injeção de Falhas por software, consolida-se a experiência acumulada no trabalho já feito em Injeção de Falhas. Como o capítulo III mostrou, diversos pesquisadores já trabalharam com Injeção de Falhas. O conhecimento gerado nesses trabalhos pode ser consolidado no sistema de padrões. Assim, esse conhecimento fica numa forma independente de linguagem e plataforma, podendo ser o mais facilmente aplicado na construção de novas ferramentas de Injeção de Falhas, que poderão implementar essa técnica de teste onde ela ainda não está presente. Portanto, pode-se esperar agora a disseminação e a popularização definitiva da Injeção de Falhas como um método de teste.

VII.2. Trabalhos futuros

VII.2.1. Trabalhos futuros relacionados ao Sistema de Padrões

Poderiam ser feitos os seguintes trabalhos com base no Sistema de Padrões para Injeção de Falhas por software:

- Como se disse no capítulo III, se estudou aqui Injeção de Falhas *por software*, por causa das vantagens já citadas nesse capítulo. Entretanto, o sistema de padrões poderia muito bem ser estendido para tratar também da Injeção de Falhas *através de simulação*. Para isso bastaria se estudar as ferramentas de Injeção de Falhas por simulação, e tentar se obter padrões a partir das mesmas.
- Igualmente, o Sistema de Padrões aqui mostrado trata do desenvolvimento de ferramentas para teste por Injeção de Falhas. Mas, como já foi dito, Injeção de Falhas não é a única técnica para teste de sistemas tolerantes a falha. Poderia-se estudar outras ferramentas de teste, que utilizam outras técnicas, e se obter padrões a partir delas. Esses padrões seriam reunidos com os padrões aqui mostrados para formar um *Sistema de Padrões para Teste de Sistemas Críticos.*
- A ferramenta JACA foi criada para demonstrar a praticidade do Sistema de Padrões. Poderia se criar outra ferramenta baseada no Sistema de Padrões, mas fazendo Injeção de Falhas através de uma técnica diferente. Por exemplo, em vez de se usar Reflexão Computacional, poderia se utilizar *traps*. Também poderia se codificar usando outra linguagem de programação, como C++. Assim, se teria um outro exemplo de uma implementação dos padrões, o que confirmaria novamente sua utilidade no desenvolvimento de ferramentas de Injeção de Falhas.

VII.2.2. Trabalhos futuros relacionados à ferramenta JACA

Já com relação à ferramenta JACA, poderiam se fazer futuramente os seguintes trabalhos:

- Como foi dito no capítulo V, a ferramenta JACA atualmente trabalha com falhas em alto nível. Entretanto, demonstrou-se que ela tem plena capacidade de trabalhar com falhas em baixo nível (seção V.5.). Poderia-se modificar a JACA para trabalhar com um conjunto de falhas em baixo nível de um dado sistema computacional. Por exemplo, seria possível ter uma versão da JACA para injetar falhas de registradores, memória, barramento em sistemas baseados na arquitetura IA-32 dos processadores Intel.
- Outro trabalho possível é a construção de uma ferramenta JACA distribuída. Em sua atual versão, a JACA fica concentrada em uma máquina, com apenas o Gerenciador de Dados Monitorados rodando remotamente. A ferramenta poderia ser extendida para injetar falhas em diferentes máquinas num sistema distribuído, para realizar Injeção de Falhas distribuída. Poderia se colocar os Injetores Físicos e os Sensores Físicos nas máquinas que se quer injetar falhas, e o resto da ferramenta ainda rodaria junto em uma máquina. Seria possível ainda outra abordagem, para quando se necessitar injetar um grande número de falhas num sistema distribuído de grande porte, com muitos dados. Cada um dos módulos da JACA ficaria rodando em uma máquina diferente, e se comunicariam via rede.

CAPÍTULO VIII: REFERÊNCIAS BIBLIOGRÁFICAS

[AvR72] Avizienis, A.; Rennels, D. "Fault-Tolerance Experiments with the JPL STAR Computer". *Proc. COMPCON '72*, páginas 321-324.

[AvT95] Avresky, D. R.; Tapadiya, P. K. "A Method for Developing a Software Based Fault Injection Tool". *Texas A&M University, Department of Computer Science, Technical Report 95-021*. Texas, EUA, 1995.

[AAA+90] Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J. C.; Laprie, J. C.; Martins, E.; Powell, D. "Fault Injection for Dependability Validation – A Methodology and some Applications". *IEEE Transactions on Software Engineering*, 16 (2), Fevereiro/1990, páginas 166-182.

[BLW99] Barcelos, Patrícia P. A.; Leite, Fábio O.; Weber, Taisy Silva. "Implementação de um Injetor de Falhas de Comunicação". *Anais do SCTF '99 – VIII Simpósio de Computação Tolerante a Falhas*. Campinas, Brasil, Julho/1999, páginas 225-239.

[BMR96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, Chichester, EUA, 1996.

[BRR99] Benso, Alfredo; Rebaudengo, Maurizio; Reorda, Matteo Sonza. "FlexFi: A Flexible Fault Injection Environment for Microprocessor-Based Systems". *SAFECOMP 1999*. Springer-Verlag, Heidelberg, Alemanha, 1999, páginas 323-335.

[ChI92] Choi, G. S.; Iyer, R. K. "Focus: An Experimental Environment for Fault Sensitivity Analysis". *IEEE Transactions on Computers*, 41 (12), Dezembro/1992, páginas 1515-1526.

[Chi96] Chillarege, Ram. "Guiding fault-injection and broadering its scope using ODC triggers". *Invited paper: Computer-Aided Design, Test and Evaluation for Dependability*, Julho/1996. Disponível na World Wide Web em: www.chilarege.com/odc/articles.

[Chi98] Chiba, Shigeru. "Javassist – A Reflection-based Programming Wizard for Java". *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Outubro/1998.

[ClP93] Clark, J.A.; Pradhan, D. K. "REACT: A System and Analysis Tool for Fault-Tolerant Multiprocessor Architectures". *Proc. Annual Reliability and Maintenability Simposium*, 1993, páginas 428-435.

[ClP95] Clark, Jeffrey; Pradhan, Dhiraj. "Fault Injection: A Method for Validating Computer-System Dependability". *IEEE Computer*, Junho/1995, páginas 47-56.

[CMG+87] Cortes, M. L.; Millman, S. D.; Goosen, H. A.; McCluskey, E. J. "Techniques for Injecting Non Stuck-at Faults". *Stanford University, Tech. Report, CRC87-21*, Março/1987.

[CMS95] Carreira, J.; Madeira, H.; Silva, J. G. "Xception: Software Fault Injection and Monitoring in Processor Functional Units". *5th IFIP International Working Conference on Dependable Computing for Critical Applications*. Urbana-Champaign, EUA, 1995, páginas 135-149.

[DJM97] Dawson, S.; Jahanian, F.; Mitton, T. "ORCHESTRA: A Fault Injection Environment for Distributed Systems". Disponível na World Wide Web em: www.eecs.umich.edu.

[GHJ94] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, EUA, 1994.

[GJS+00] Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad. *The Java Language Specification – Second Edition*. Sun Microsystems, Inc., Palo Alto, CA, EUA, 2000.

[GKT89] Gunnelo, U.; Karlsson, J.; Torire, J. "Evaluation of Error Detection Schemes using Fault Injection by Heavy-Ion Radiation". *Proc. FTCS-19*, Junho/1989, páginas 340-347.

[GoI91] Goswami, K. K.; Iyer, R. K. "Depend: a Simulation Based Environment for System Level Dependability". *IEEE Transactions on Computers*, vol. 46, 1997, páginas 60-74.

[HTI97] Hsueh, Mei-Chen; Tsai, Timothy; Iyer, Ravishankar. "Fault Injection Techniques and Tools". *IEEE Computer*, Abril/1997, páginas 75-82.

[Int01] Intel Corporation. *AP-485 Application Note: Intel Processor Identification and the CPUID Instruction*. Intel Corporation, Mt. Prospect, IL, EUA, Fevereiro/2001.

[JAR+94] Jenn, Eric; Arlat, Jean; Rimen, M; Ohlsson, J.; Karlsson, J. "Fault Injection into VHDL models: The MEFISTO Tool". *Proc. 24th Symposium on Fault Tolerant Computing (FTCS-24)*, 1994, páginas 66-75.

[KiM01] Kikuti, Márcio; Martins, Eliane. "JAGUAR: Uma Ferramenta para Injeção de Falhas no Guaraná". *Relatório Técnico IC* (ainda em elaboração), 2001.

[KIT93] Kao, Wei-lun; Iyer, Ravishankar; Tang, Dong. "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults". *IEEE Transactions on Software Engineering*. Volume 19, número 11, Novembro 1993, páginas 1105-1118.

[KJA98] Krishnamurthy, N.; Jhaveri, V.; Abraham, J. "A Design Methodology for Software Fault Injection in Embedded Systems". *Proc of the 1998 IFIP International Workshop on Dependable Computing and its Applications*. Johannesburg, Africa do Sul, Jan. 12-14, 1998, páginas 237-248.

[Lap95] Laprie, Jean-Claude. "Dependability – Its Attributes, Impairments and Means". *Predictability Dependable Computing Systems (B. Randell, J.-C. Laprie, H. Kopetz, B. Littlewood, eds)*. Springer, Berlin, Alemanha, 1995, páginas 3-18.

[Lei87] Leite, J. C. B., Orlando G. Loques Filho, Software. *II SCTF*, cap. 4 do mini-curso intitulado: Introdução à Tolerância a Falhas, Campinas, SP, Brasil, 1987.

[LoE93] Lovric, T.; Echtle, K. "ProFI: a Processor Fault Injection for Dependability Validation". *IEEE International Workshop on Fault and Error Injection for Dependability Validation*, Gotemburgo, Suécia, Junho de 1993.

[LMR00] Leme, Nelson G. M.; Martins, Eliane; Rubira, Cecília M. F. "Um Sistema de Padrões para Injeção de Falhas por Software". *Anais do II Workshop de Testes & Tolerância a Falhas*. Curitiba, PR, Brasil, 15-16 de Julho, 2000, páginas 100-105.

[LMR01] Leme, Nelson G. M.; Martins, Eliane; Rubira, Cecília M. F. "A Software Fault Injection Pattern System". *Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing*. Florianópolis, SC, Brasil, 5-7 de Março, 2001, páginas 99-113.

[MFS91] Madeira, H.; Furtado, P.; Silva, J. G. "RIFLE: A General Purpose Fault Injector System". *Research Report, DEE-UC-007-91*, Novembro/1991.

[Oli98] Oliva, Alexandre. *Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java*. Dissertação de Mestrado, UNICAMP, Campinas, Brasil, 1998.

[Ros98] Rosa, Amanda. *Uma Arquitetura Reflexiva para Injetar Falhas em Aplicações Orientadas a Objetos*. Dissertação de Mestrado, UNICAMP, Campinas, Brasil, 1998.

[Sun97] Sun Microsystems, Inc. *Java Native Interface Specification*. Sun Microsystems, Inc., Palo Alto, CA, EUA, Maio/1997.

[Sun99] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*. Sun Microsystems, Inc., Palo Alto, CA, EUA, Dezembro/1999.

[TCK+00] Tatsubori, Michiaki; Chiba, Shigeru; Killijian, Marc-Olivier; Itano, Kozo. "OpenJava: A Class-Based Macro System for Java". *Lecture Notes in Computer Science 1826, Reflection and Software Engineering,* Walter Cazzola, Robert J. Stroud, Francesco Tisato (editores). Springer-Verlag, páginas 117-133, 2000.

[VMK+97] Voas, Jeffrey; McGraw, Gary; Kassab, Lara; Voas, Larry. "A 'Crystal Ball' for Software Liability". *IEEE Computer*, Junho/1997, páginas 29-36.

[Voa98] Voas, Jeffrey. "Certifying Off-the-Shelf Software Components". *IEEE Computer*. Junho/1998, páginas 53-59.

[VoM98] Voas, Jeffrey; McGraw, Gary. *Software Fault Injection: Inoculating Programs against Errors*. John Wiley & Sons, New York, EUA, 1998.

[WeS99] Welch, Ian; Stroud, Robert. "Dalang – A Reflective Extension for Java". *Techinical Report CS-TR-672*. Departamento de Ciência da Computação, Universidade de Newcastle upon Tyne, Reino Unido, Setembro/1999.

APÊNDICE A

COMO OBTER A FERRAMENTA JACA

Maiores informações sobre a ferramenta JACA podem ser obtidas em sua homepage na Internet, que pode ser acessada no seguinte endereço:

`http://www.ic.unicamp.br/~nelsongm/JACA.html`

Nesta página, pode-se encontrar documentos sobre a ferramenta JACA, bem como a própria ferramenta em sua última versão. Entretanto, só estará disponível nessa página o bytecode da ferramenta, portanto o usuário terá que utilizá-la exatamente como ela está no fim do seu desenvolvimento. Entretanto, como já foi dito, um dos objetivos da ferramenta JACA era que ela fosse altamente adaptável. Se ela não satisfizesse exatamente os requisitos do usuário, poderia ser reescrita para fazê-lo. Para tal, é necessário o código fonte em Java da ferramenta. Para conseguir o código fonte da JACA, deve-se entrar em contato com a Profa. Dra. Eliane Martins, do Instituto de Computação (IC), da Universidade Estadual de Campinas (UNICAMP). Pode-se fazer isso através do seguinte e-mail:

`eliane@ic.unicamp.br`

Além disso, está disponível também o "Manual de Operação da Ferramenta JACA". Este documento pode ser obtido livremente na homepage da ferramenta JACA, e a partir dele o usuário pode avaliar se será necessário algum trabalho de adaptação da ferramenta, ou se a mesma pode ser usada como está em sua última versão.