III CONGRESSO DE INICIAÇÃO CIENTÍFICA
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

PIBIC/CNPq/UFCG-2006

EXTRAÇÃO E VERIFICAÇÃO AUTOMÁTICA DE MODELOS DE SOFTWARE

João Arthur Brunet Monteiro[1], Dalton Dario Serey Guerrero[2]

RESUMO

A adoção de processos ágeis no desenvolvimento de software tem mudado a forma como a fase de projeto de software é conduzida. A maioria das equipes inicia o desenvolvimento de um novo software com um projeto inicial simples. À medida que o projeto evolui, componentes são incluídos ou removidos do projeto. Cada componente, por sua vez, pode ser quebrado em vários outros ou pode ser combinado com componentes existentes, formando um novo. O projeto aqui proposto tem por objetivo desenvolver uma ferramenta que permitirá acompanhar e conduzir a evolução de projetos de software, através da extração automática de modelos de projeto a partir de código fonte e de sua verificação em relação a regras de projeto (*design*) adotas.

Palavras-chave: Projeto (*design*) de software, refatoramento, processos ágeis, extração automática de modelos.

AUTOMATIC EXTRACTION AND VERIFICATION OF SOFTWARE MODELS

ABSTRACT

The use of agile processes has been changing the way the project phase is conduced during software development. Most of teams start the development of new software's as an initial simple project. During software evolution, components are added or removed from the project. Each component, in turn, may be broken into many other ones or they may also be combined with already existing components, thus creating a new component. The goal of this project is to develop a tool for software evolution tracking, through the automatic extraction of software models, from source code, and its verification concerning on the project specifications

Keywords: software project (design), refactoring, agile process, automatic extraction of models.

INTRODUÇÃO

A adoção de processos ágeis no desenvolvimento de software tem mudado a forma como a fase de projeto de software é conduzida. A maioria das equipes inicia o desenvolvimento de um novo software com um projeto inicial simples. À medida que o software evolui, componentes[3] são incluídos ou removidos do projeto. Cada componente pode ser quebrado em vários outros ou pode ser combinado com outros componentes existentes, formando um novo. As interfaces e funcionalidades dos componentes também evoluem, para melhorar sua usabilidade e eficiência. Os ganhos em flexibilidade e em agilidade, que permitem adaptar o projeto à medida que evolui o entendimento do problema e a solução proposta, são em parte, obtidos em detrimento de uma menor qualidade da documentação. Esta é uma das principais razões pelas quais a introdução e a perda de membros é especialmente pouco tolerada em processos ágeis. Introduzir um novo membro em um projeto requer que este tome ciência do estado atual do projeto, bem como de todas as decisões tomadas e das convenções adotadas para segui-las corretamente. Contudo, é

[1] Aluno do Curso de Ciência da Computação, Depto. de Sistemas e Computação, UFCG, Campina Gande , PB, e-mail: jarthur@dsc.ufcg.edu.br
[2] Ciência da Computação, Prof. Doutor, Depto. de Sistemas e Computação, UFCG, Campina Grande, PB, Email: dalton@dsc.ufcg.edu.br
[3] Componentes de Software é o termo utilizado para descrever o elemento de software que encapsula uma série de funcionalidades. Um componente é uma unidade independente, que pode ser utilizado com outros componentes para formar um sistema mais complexo (COMPONENTES, 2005).

comum em projetos abordados dessa forma que a única documentação confiável seja o próprio código fonte produzido. Como conseqüência, projetos de software que são desenvolvidos utilizando processos ágeis produzem sistemas com projetos desnecessariamente complexos, difíceis de verificar em relação a critérios objetivos de qualidade e, finalmente, mais difíceis de manter e evoluir.

Métodos ágeis de desenvolvimento são um conjunto de metodologias de desenvolvimento de software. Um método ágil, tal como qualquer metodologia de desenvolvimento de software, proporciona uma estrutura conceitual para a condução de projetos de software. Em metodologias ágeis de desenvolvimento são valorizados aspectos diferentes daqueles que as metodologias ditas convencionais costumam valorizar. No Manifesto Ágil (AGILEMANIFESTO, 2001), um documento considerado como a definição canônica de método ágil, publicado por um grupo de 17 desenvolvedores formalizou a disciplina, encontramos a seguinte lista de valores que caracteriza os processos ágeis em oposição a outras metodologias:

- **indivíduos e interações** ao invés de processos e ferramentas;
- **software em operação** ao invés de documentação completa;
- **colaboração do cliente** ao invés de negociações de contratos;
- **resposta a mudanças** ao invés de planos seguidos rigorosamente.

Métodos ágeis são freqüentemente caracterizados como o extremo oposto a metodologias "orientadas a planos". Contudo, essa distinção não é adequada porque mesmo em metodologias ágeis existem planos e contém uma disciplina rigorosa a ser seguida. Uma forma mais adequada de distinguir os processos é caracterizando-os sobre uma escala de quão adaptativos os processos podem ser. Métodos ágeis são desta forma, mais adaptativos que outros métodos tradicionais porque valorizam a possibilidade de mudanças e definem as técnicas de desenvolvimento considerando que a realidade muda freqüentemente. Isto inclui principalmente mudanças de requisitos. No extremo oposto, ficam as metodologias preditivas. Nestas o foco está no planejamento detalhado, baseado na premissa de que é possível obter toda a informação sobre requisitos de um software a priori e que tais requisitos não devem mudar. Um plano, assim, é definido detalhadamente para alcançar os objetivos estabelecidos. Devido a isso, mudanças de requisitos são mais difíceis de tratar em tais metodologias.

Uma das atividades características de métodos ágeis é o uso de refatoração. A atividade de refatoração visa promover mudanças estruturais do código e do projeto de um sistema de software sem mudanças funcionais aparentes. A idéia é apenas reestruturar código/*design* mantendo a funcionalidade intacta, mas melhorando a estrutura interna para uma futura expansão. Tipicamente, a refatoração é utilizada em sistemas de software quando é necessário adicionar uma funcionalidade que seria difícil no sistema em sua estrutura original. Existe todo um conjunto de técnicas experimentais de refatoração.

Diversas ferramentas de desenvolvimento permitem refatorar código de forma semi-automática, tais como Eclipse (ECLIPSE, 2006) e NetBeans (NETBEANS, 2006), cabendo ao desenvolvedor especificar qual tipo de refatoração se deseja fazer e o trecho do sistema em que será feito. Se por um lado, essa técnica permite lidar com mudanças, por outro ela impõe constantes mudanças no código e no projeto de um sistema. Em certa escala, tal processo tende a dificultar o acompanhamento da evolução do projeto. Por essa razão, processos ágeis valorizam o recurso humano como parte do projeto. A perda de membros de uma equipe tende a retirar do projeto o entendimento de partes do projeto. Além disso, a adição de novos membros também é difícil devido à necessidade de compreensão do projeto sem a existência de documentação completa.

O quadro descrito torna-se ainda mais complexo quando os sistemas em desenvolvimento requerem o uso de características como paralelismo, mecanismos de concorrência e processamento distribuído. *Deadlocks*, *livelocks*, *starvations* são alguns dos problemas típicos que surgem em sistemas com tais características, sendo de difícil identificação. É fato bem conhecido entre desenvolvedores, que esses erros são essencialmente erros de projeto e que não são detectados sem um procedimento extremamente metódico. De fato, em alguns casos somente uma abordagem formal permite realmente identificar estes erros. Devido à proliferação da tecnologia de redes e à disseminação da Internet, a grande maioria dos sistemas desenvolvidos hoje se inclui na categoria de sistemas descritos anteriormente.

No projeto aqui apresentado, é proposta uma solução para o problema da falta de mecanismos para controlar e acompanhar automaticamente a evolução de projetos de software conduzidos usando processos ágeis. A solução consiste em duas partes: i) desenvolver técnicas e ferramentas que permitam aos desenvolvedores *rastrear* e *controlar* a evolução do projeto de software durante seu desenvolvimento, sem tornar obrigatória sua documentação e sincronização ao longo do desenvolvimento e, portanto, mantendo as características originais de processos ágeis; e ii) desenvolver um conceito de projeto (*design*) que promova os aspectos comportamentais, igualando-os em relevância aos aspectos estruturais, tipicamente favorecidos nas definições adotadas em geral.

Por *rastrear*, referi-se à habilidade de observar e comparar objetivamente o projeto de um software em diferentes momentos de seu desenvolvimento. A habilidade de rastrear a evolução de um projeto proporcionará aos membros de uma equipe informação fundamental para o entendimento da arquitetura e de como e por que ela evolui dessa forma. Por *controlar*, nos referimos ao poder que alguns membros da equipe devem ter de decidir e impor regras e restrições à evolução do projeto. Assim, toda e qualquer

proposta de modificações no software será automaticamente verificada contra as regras pré-estabelecidas, antes de ser aceita no repositório central[4] do projeto. O controle da evolução de um projeto dá aos projetistas e/ou gerentes o poder necessário para garantir que o projeto atende a critérios objetivos de qualidade.

METODOLOGIA

Nesta seção iremos apresentar todo o processo de pesquisa efetuado, bem como os métodos utilizados, para que o projeto fosse concebido.

O projeto foi desenvolvido no Laboratório do Grupo de Métodos Formais (GMF) do Departamento de Sistemas e Computação (DSC) da Universidade Federal de Campina Grande (UFCG). Neste contexto, para a execução do projeto estavam envolvidos, pesquisadores doutores, doutorandos, mestrandos e alunos de graduação. A infra-estrutura atual do laboratório é utilizada para comportar os equipamentos adquiridos com os recursos do projeto, aumentando assim a quantidade de postos de trabalho.

A interação entre os membros envolvidos neste projeto foi realizada através da geração de documentos intermediários e de reuniões periódicas entre os participantes. Além dessas reuniões, seminários foram promovidos com a participação de alunos e demais pesquisadores envolvidos no projeto. Na divisão de responsabilidades, foram estabelecidos indicadores precisos de avaliação. Fizeram parte ainda desta metodologia as definições de ferramentas e meios de comunicação que permitiram uma interação padronizada entre os componentes da equipe do projeto. Nesta sistemática, os trabalhos relacionados foram divididos em atividades. Cada atividade possuía um coordenador, que atuou como elemento que agregava e sinalizava as ações a serem executadas em cada atividade. No cronograma para execução do projeto estão definidas as atividades e suas restrições de tempo.

O período inicial do projeto foi dedicado exclusivamente aos estudos de métodos ágeis e estudo do conceito de modelo de projeto. Após o primeiro mês, o foco foi pesquisar sobre os vários tipos de métodos de extração e estudar linguagens referentes a expressão de regras de projeto. A partir do segundo mês, tendo em vista que já se possuía um conhecimento para isto, a ferramenta *Design Wizard* passou a ser desenvolvida seguindo o processo de desenvolvimento ágil XP (*Extreme Programming*). Esse processo foi adotado por ser voltado para equipes pequenas, por sua simplicidade na conduta do desenvolvimento do software, por permitir flexibilidade no desenvolvimento das funcionalidades e por obtermos um amplo conhecimento e experiência nessa metodologia.

Tecnologias e Ferramentas estudadas

Java Fact Extractor

Java Fact Extractor (JFX) (KAASTRA&KAPSER, 2003) é uma ferramenta que extrai fatos[5] de código escrito na linguagem Java. Java é uma linguagem de programação orientada a objetos e que é usada por grande parte programadores.

O estudo e uso dessa ferramenta no projeto *Design Wizard* foi de extrema importância para a construção de um extrator que fosse satisfatório, ou seja, disponibilizasse informações relevantes e confiáveis sobre o código analisado. Esta ferramenta trabalha de forma a extrair fatos do *byte code*[6] gerado em tempo de compilação de um código Java e apresentar esse conjunto de fatos de forma textual.

Reflexão

Reflexão permite um programa Java examinar ou fazer a introspecção nele mesmo, ou seja, olhar e examinar suas propriedades e estrutura. Com isso, você pode, por exemplo, obter o nome de todos os membros de uma classe, como atributos e métodos, bem como executar um método usando a *Introspection*. O estudo dessa tecnologia foi útil para construirmos um extrator poderoso, unindo informações extraídas pelo JFX[7] e informações extraídas através de reflexão.

---

[4] Repositório Central é uma árvore de diretórios contendo todas as versões do software que está sendo desenvolvido (REPOSITORIO, 2006).

[5] Fatos são informações sobre a estrutura de um código de software.

[6] bytecode é uma espécie de codificação que traduz tudo o que foi escrito no programa para um formato que a máquina virtual entenda e seja capaz de executar.

[7] http://csg.uwaterloo.ca/~mdmkaastra/jfx/

Junit

A ferramenta JUnit (GAMMA&BACK, 2001) é um framework (FRAMEWORK, 2006), de código-fonte disponível, que tem como finalidade a construção de testes de unidade. Através de dados de entrada necessários à instanciação de objetos, o JUnit informa o resultado dos testes, quando o resultado é positivo é mostrada uma barra verde, quando não, a barra é vermelha. O Junit foi útil na definição de qual interface usar na construção do *Design Wizard*. Foi escolhida uma interface baseada em testes de unidade, já que o trabalho ao usuário seria mínimo, bastaria ele conhecer como compor um teste de unidade que seria fácil, daí então, compor testes estruturais.

ASM

È um framework[8] para manipulação de *bytecode*. Inicialmente estávamos usando a ferramenta Java Fact Extractor (KAASTRA&KAPSER, 2003) para fazer a extração de fatos do *bytecode*, porém nos deparamos com um problema, esta ferramenta era somente compatível com *bytecode* gerado por código java escrito na versão java 1.4. Como sabemos que java evoluiu para 1.5 e que muitos sistemas estão sendo escritos nessa versão, precisamos procurar um extrator que seja compatível com java 1.5, esse extrator é o ASM. (BRUNETON,KULESHOV&LOSKUT, 2006).

ASM é extremamente eficiente, o número de informações que ele extrai é bem maior que o Java Fact Extractor e a velocidade com que essas informações são extraídas é extremamente satisfatória, o que justifica nossa mudança de extrator.

SWING

Swing é uma API(API,2006) que fornece serviços para o desenvolvimento de interfaces gráficas com o usuário.

A API(API, 2006) Swing procura renderizar/desenhar por contra própria todos os componentes, ao invés de delegar essa tarefa ao sistema operacional, como a maioria das outras API's de interface gráfica trabalham.

Swing é bem mais completa que a maioria das API's para interface gráfica existentes, e os programas que usam Swing tem uma aparência muito parecida, independente do Sistema Operacional utilizado. Esse é um dos fatores que contribuíram para a escolha do uso de Swing.

Usamos Swing por ser muito simples de se construir uma interface gráfica, ser muito bem documentada e possuir vários tutoriais que nos permitiu conhecer a fundo a tecnologia.

RESULTADOS E DISCUSSÃO

Nesta seção é apresentada a ferramenta que foi desenvolvida, cujo nome é *Design Wizard*. Essa apresentação tem como objetivo mostrar as funcionalidades e os recursos estudados e utilizados para que as mesmas fossem concebidas. *Design Wizard* está disponível para *download* em http://www.gmf.ufcg.edu.br/index.php/DesignWizard

A nível de arquitetura

Design Wizard é composto por quatro módulos: Extrator, Tradutor, Verificador e Diferenciador.

- **Módulo Extrator.** Esse módulo é responsável por extrair informações estruturais do *byte code* e do código de sistemas escritos em Java. Recebemos como entrada um arquivo .jar, e dele são extraídas informações sobre o código relativo ao *bytecode* contido no arquivo .jar. Nesse módulo, faz-se uso também reflexão para que possamos obter o maior número de informações possíveis sobre o código a ser analisado. Essas informações são armazenadas para posterior verificação de propriedades estruturais do código.
- **Módulo Tradutor.** Como são usados dois extratores distintos, foi necessária a construção do módulo tradutor para que as informações fossem armazenadas seguindo um padrão, já que

[8] Framework é uma estrutura de suporte definida em que um outro projeto de software pode ser organizado e desenvolvido. Tipicamente, um *framework* pode incluir programas de apoio, bibliotecas de código, linguagens de script e outros softwares para ajudar a desenvolver e juntar diferentes componentes do seu projeto (FRAMEWORK, 2006).

as informações extraídas vinham com formato diferente uma da outra. Com a construção desse módulo, fica simples, caso precise, colocar um novo extrator no *Design Wizard*. A idéia é que possamos ter vários extratores e que eles se comuniquem com os outros módulos da ferramenta através do módulo tradutor. Feito isso, podemos ter vários extratores, cada um com sua particularidade, porém as informações extraídas por eles serão padronizadas pelo tradutor.

- **Módulo Verificador.** Esse módulo é responsável pela verificação dos testes estruturais escritos pelo programador. Toda a lógica de testes está presente neste módulo. No módulo verificador, estão armazenadas as informações sobre o código a ser analisado, com essas informações é possível fazer a verificação de conformidade de código com os testes estruturais escritos pelo programador.
- **Módulo Diferenciador.** Esse módulo é responsável pela lógica de verificar o que mudou em um software de uma versão para a outra. Este trabalho é feito por algoritmos que fazem uma comparação entre os fatos extraídos da primeira versão do software com os fatos extraídos da segunda versão do software.

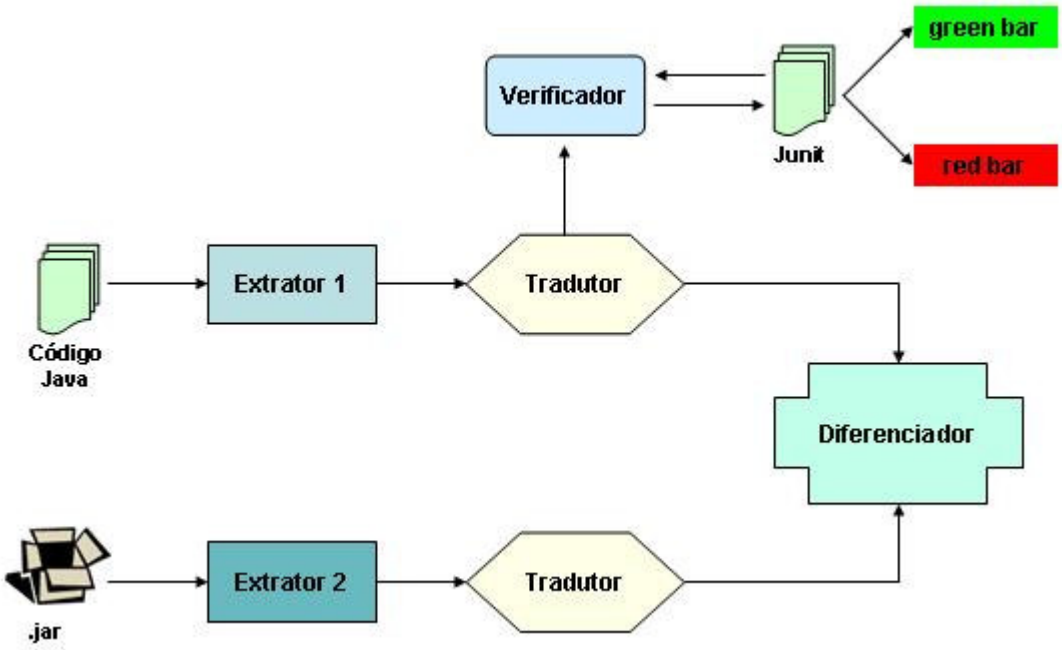

Figura 1 - Arquitetura do *Design Wizard*

A Figura 1 mostra a arquitetura do *Design Wizard*. A ferramenta recebe como entrada o código fonte de um software e/ou o arquivo .jar correspondente a esse código fonte.

O módulo extrator encarrega-se então de extrair os fatos referentes as entradas. Perceba que podem existir vários tipos de extratores no *Design Wizard*, por enquanto a ferramenta possui dois extratores, um é responsável pela extração de fatos através do *bytecode*, o seu nome é ASM (BRUNETON,KULESHOV&LOSKUT, 2006), o outro extrator foi implementado pela nossa equipe através da API[9] de Reflexão de Java.

Possuir vários extratores diferentes permitiu ao *Design Wizard* obter um grande número de informações relevantes, bem como uma flexibilidade na dependência desses módulos.

A partir daí as informações extraídas são repassadas ao módulo tradutor que é encarregado de receber os fatos em diversas formas e transformá-los em um padrão único, para que o os próximos módulos possam interpretar e manipular essas informações. É esse módulo que faz com que o *Design Wizard* possua uma arquitetura que é capaz de possuir vários extratores diferentes ao mesmo tempo. Pois as informações que cada extrator fornece, tem diferentes formatos, cabendo ao módulo Tradutor a função de padronizar todas essas informações.

Os próximos módulos são responsáveis por implementar as principais funcionalidades da ferramenta, são eles o módulo Verificador e o módulo Diferenciador.

[9] API, de Application Programming Interface (ou Interface de Programação de Aplicativos) é um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades por programas aplicativos -- isto é: programas que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços (API, 2006).

O módulo Verificador é responsável por checar a conformidade do código com a arquitetura pré-definida. Para isso, ele faz uso dos testes estruturais escritos pelo gerente do projeto e das informações repassadas pelo módulo tradutor. A verificação é feita de forma totalmente automática, e seu resultado é apresentado usando o framework (FRAMEWORK, 2006) Junit (GAMMA&BECK, 2001), que mostra uma barra verde na tela caso o teste seja bem-sucedido e uma barra vermelha caso contrário

O módulo Diferenciador tem a responsabilidade de fazer a comparação entre duas versões distintas de software e apresentar os resultados ao usuário. Veremos mais adiante o modo como é feita essa comparação e como é mostrado o resultado.

A nível de código

A Figura 2 denota bem os três módulos que compõe a ferramenta. As classes Jfx,ReflectExtractor e ASMExtractor fazem parte do módulo extrator cuja finalidade foi descrita acima.

- A classe *Translator* faz parte do módulo tradutor.
- As classes *Design* e *DesignWizard* fazem parte do módulo verificador e Diferenciador. A classe Design é a classe que contém as informações estruturais sobre o código, já a classe *DesignWizard* é responsável pela lógica de responder sobre determinadas características estruturais do código, além de dispor informações sobre a evolução do software. Nessa classe, estão dispostos as principais funcionalidades da ferramenta, por isso, vamos detalhar mais precisamente o que cada método dessa classe faz.

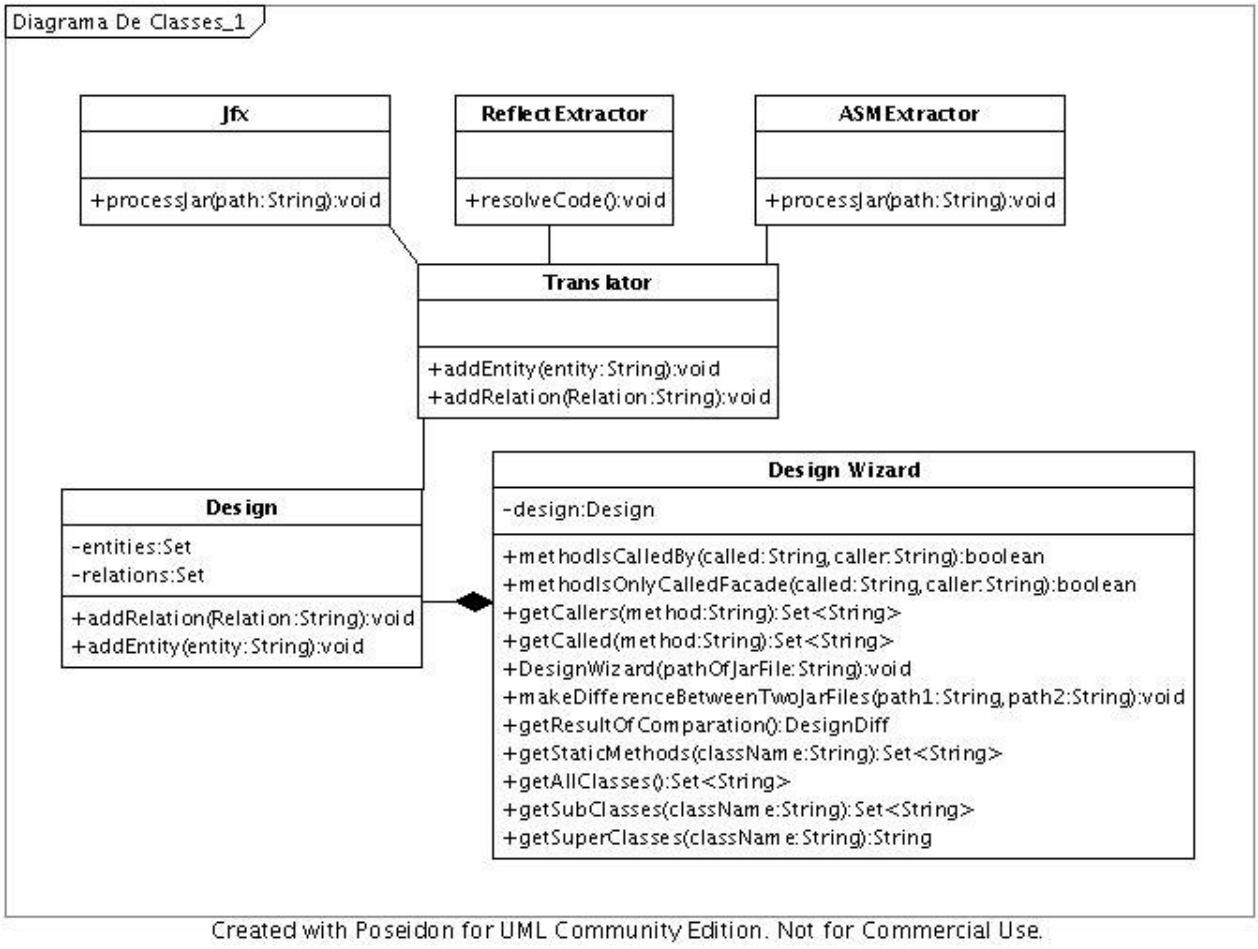

Figura 2 – Diagrama de classes do *Design Wizard*.

Principais métodos da classe DesignWizard

+DesignWizard(pathOfJarFile:String)

Cria uma nova instância de *DesignWizard*. Esse objeto será responsável por prover as principais funcionalidades da ferramenta.

+methodIsCalledBy(called:String, caller:String) : Boolean

Verifica se um método (*called*) é invocado por outro método (*caller*.).

+methodIsOnlyCalledFacade(called:String, caller:String) : Boolean

Verifica se um método (*called*) é invocado apenas por métodos de uma única classe (*caller*).

+getCallers(method:String) : Set<String>

Retorna todas as entidades que invocam um determinado método.

+getCalled(method:String) : Set<String>

Retorna todas as entidades que são invocadas por um método.

+makeDifferenceBetweenTwoJarFiles(path1:String, path2:String)

Faz a diferença entre duas versões distintas de um software, passado o caminho dos arquivos .jar dessas versões.

+getResultOfComparation() : DesignDiff

Retorna um objeto que representa o resultado da diferença entre duas versões distintas de um software.

+getStaticMethods(className:String) : Set<String>

Retorna o conjunto de métodos estáticos de uma determinada classe (*className*).

+getAllClasses() : Set<String>

Retorna o conjunto das classes do sistema.

+getSubClasses(className:String) : Set<String>

Retorna o conjunto de subclasses de uma determinada classe (*className*).

+getSuperClass(className:String) : String

Retorna a superclasse de uma determinada classe (*className*) do sistema.

Observação: Muitas classes foram omitidas do diagrama para que pudéssemos dar uma melhor explicação sobre o funcionamento da ferramenta. Em sua arquitetura completa, O módulo extrator possui 43 classes Java, enquanto que o módulo tradutor possui 2 classes Java e o módulo verificador, por sua vez, possui 11 classes Java. Além das classes de teste que foram construídas ao longo do desenvolvimento da ferramenta.

A ferramenta Design Wizard

1 - Extração e Verificação Automática

Design Wizard é uma ferramenta de extração direta de modelos de software e que faz uso dessas informações extraídas do código fonte e do arquivo .jar desse código fonte para verificar propriedades estruturais do código. *Design Wizard* provê uma série de métodos que verificam a existência de determinadas propriedades estruturais sobre o código, tais como: *quais métodos são chamados por determinado método*. Com esses métodos providos pela ferramenta, é extremamente simples checar regras de *design* de um código. Isso é feito com apoio do *framework* Junit[10]. Testes de unidade servem para

[10] http://junit.sourceforge.net/

verificar a funcionalidade do software, porém no *Design Wizard* eles são utilizados para verificar a estrutura do software. Com isso, testes estruturais são escritos de maneira muito semelhante a testes de unidade. Na Figura 1 é apresentado um exemplo de código que testa propriedades estruturais do software "Gerenciador de Hotéis", um software desenvolvido por alunos da disciplina de Estrutura de Dados na Universidade Federal de Campina Grande. Gerenciador de Hotéis é um software que gerencia as atividades de um Hotel, tais como reserva, aluguel, calculo de diárias etc.

```
1. package myproject.test;
2. import junit.framework.TestCase;
3. import designWizard.main.DesignWizard;

4. public class AssertionsTest extends TestCase {
5.   DesignWizard dw = null;
6.   public void testMethodCalledby() {
7.     dw = new DesignWizard("/home/jarthur/amsn_received/GerenciadorDeHoteis.jar");
8.     assertTrue(dw.methodIsCalledBy("Hotel.getReserva(java/lang/String)",
"Hotel.cancelarReserva()");
                                        9.   }
10.  public void testMethodNotCalledby() {
11.    dw = new DesignWizard("/home/jarthur/amsn_received/GerenciadorDeHoteis.jar");
12.    assertFalse (dw.methodIsCalledBy("Hotel.cancelarReserva()",
"ReservaJaCanceladaException.delete()"));
13.  }
14.  public void testMethodOnlyCalledByClass() {
15.    assertTrue(dw.methodIsOnlyCalledByFacade("ClassAMod2.calculateint()",
"FacadeClass"));
16.    dw = new DesignWizard("/files/testNOTOnly.jar");
17.    assertFalse(dw.methodIsOnlyCalledByClass("ClassAMod2.calculateint()",
"FacadeClass"));
18.  }
19.}
```

O trabalho de extração é realizado pela ferramenta *Design Wizard* a partir do momento em que o programador cria uma nova instância da classe *Design Wizard* (linha 7). Com a extração, a ferramenta contém informações estruturais sobre o código a ser testado. A forma de extração escolhida foi a técnica que extrai fatos do *bytecode* gerado ao compilar o código fonte. Essa escolha foi feita pelo grande número de informações relevantes que encontramos neste tipo de extração e por haver ferramentas que fazem esta extração de maneira muito eficiente. Sendo assim, foi incorporado ao *Design Wizard* a ferramenta *Java Fact Extractor*(KAASTRA&KAPSER, 2003) e o extrator ASM(BRUNETON,KULESHOV&LOSKUT, 2006). Além desses dois extratores, *Design Wizard* faz uso também de um extrator próprio implementado usando a API(API, 2006) de Releflexão de Java.

A semântica e a sintaxe dos testes estruturais são simples de serem entendidas, perceba que na linha 8, o projetista deseja verificar se o método getReserva (java/lang/String) da classe Hotel é chamado pelo método cancelarReserva() da mesma classe.

Da mesma maneira, na linha 12, o projetista deseja certificar que o método cancelarReserva() da classe Hotel não é chamado pelo método delete() da classe ReservaJaCanceladaException.

Outro método interessante para testes estruturais é o método methodIsOnlyCalledByClass(String method, String class). Esse método retorna se o método passado como parâmetro somente é chamado por métodos da classe que foi passada como parâmetro. Na linha 15, é possível ver um exemplo desse tipo de teste.

O ambiente para o uso *Design Wizard* é o mesmo utilizado pelos desenvolvedores. O ambiente mostrado na Figura 2 é a ferramenta de suporte a desenvolvimento de software Eclipse.

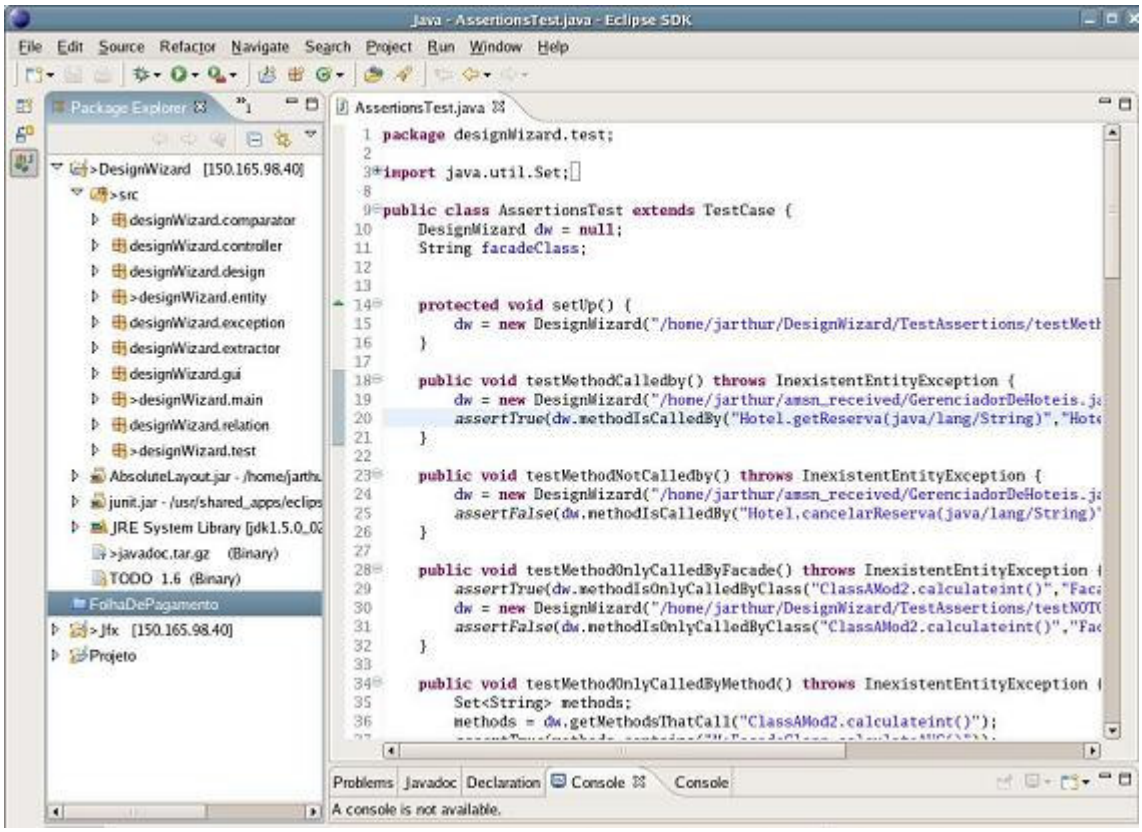

Figura 3 – Exemplo de Ambiente para se usar *Design Wizard*.

A interface do *Design Wizard* foi baseada em interface de teste. Isso foi feito por várias razões, a mais importante delas é que compor um *TestCase*[11], por ser uma prática muito difundida e usada na comunidade científica, o que causaria uma fácil adaptação ao usuário de *Design Wizard*, uma vez que o usuário não iria ter que aprender novas técnicas que possivelmente fossem impostas pela ferramenta *Design Wizard* para fazer testes estruturais. Sabe-se que aprender novas técnicas para fazer uso de uma nova ferramenta não é uma boa característica de um bom projeto, pois requer treinamento dos usuários.

Há também o fato da interface do JUnit ser bem conhecida. Sendo então adotada para aproveitar a simplicidade dessa interface.

Perceba na Figura 4 que os teste estruturais são escritos com apoio do *framework* JUnit e fazendo uso dos métodos oferecidos pela API(API,2006) do *Design Wizard*. Esse trabalho em conjunto, faz a verificação das propriedades estruturais. Os resultados são expressos pela forma tradicional que o JUnit costuma apresentar seus resultados, ou seja, uma barra verde na tela indica que os testes foram bem sucedidos, ao passo que uma barra vermelha indica que algum teste falhou. Mais informações, como por exemplo, quais testes falharam também são mostradas na tela.

Veja na Figura 3 um exemplo em que propriedades estruturais do código estão de acordo com os testes estruturais.

[11] TestCase é a classe que chama os métodos das classes para realizar os testes.

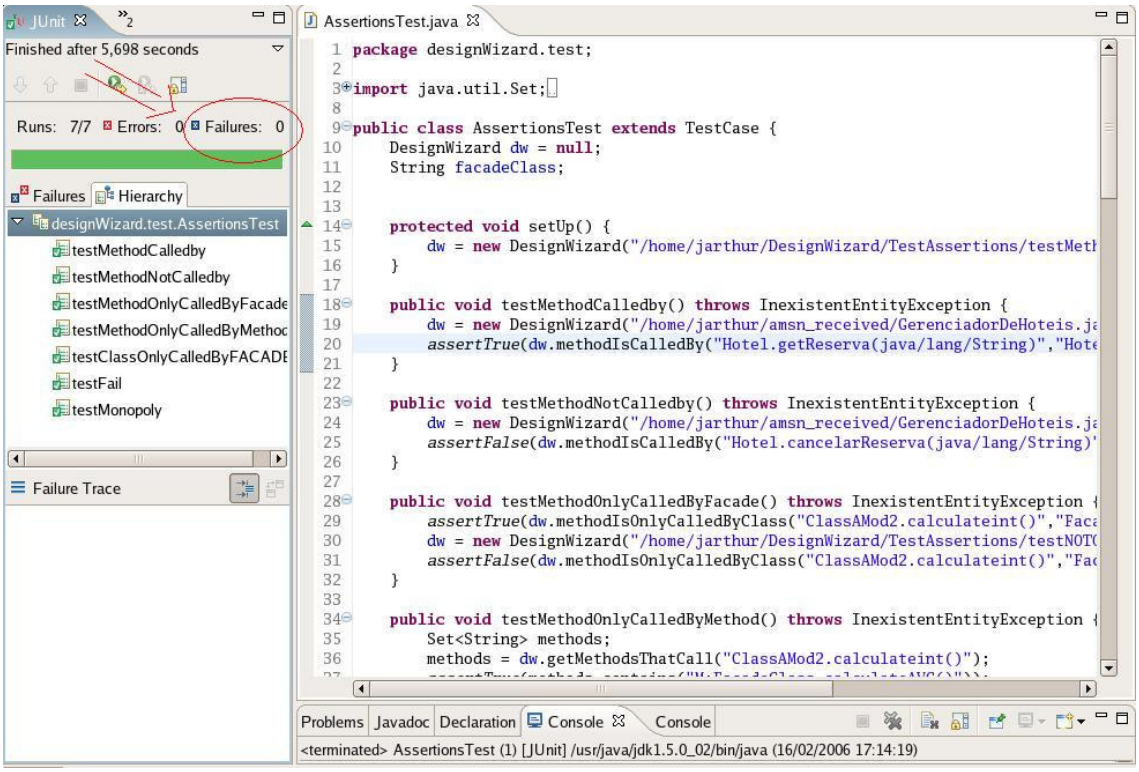

Figura 4 - Exemplo testes que certificam que as propriedades estruturais estão sendo satisfeitas.

Perceba que é mostrado no campo Failures em destaque, após a execução do testes, o número de teste que falharam. No caso da Figura 4 o número de testes falhos é zero, ou seja, todos os testes estão corretos, certificando então que a verificação estrutural do código foi feita e foi constatado que o código está de acordo com os testes.

Veja na Figura 5 um exemplo em que o código não está em conformidade com os testes escritos:

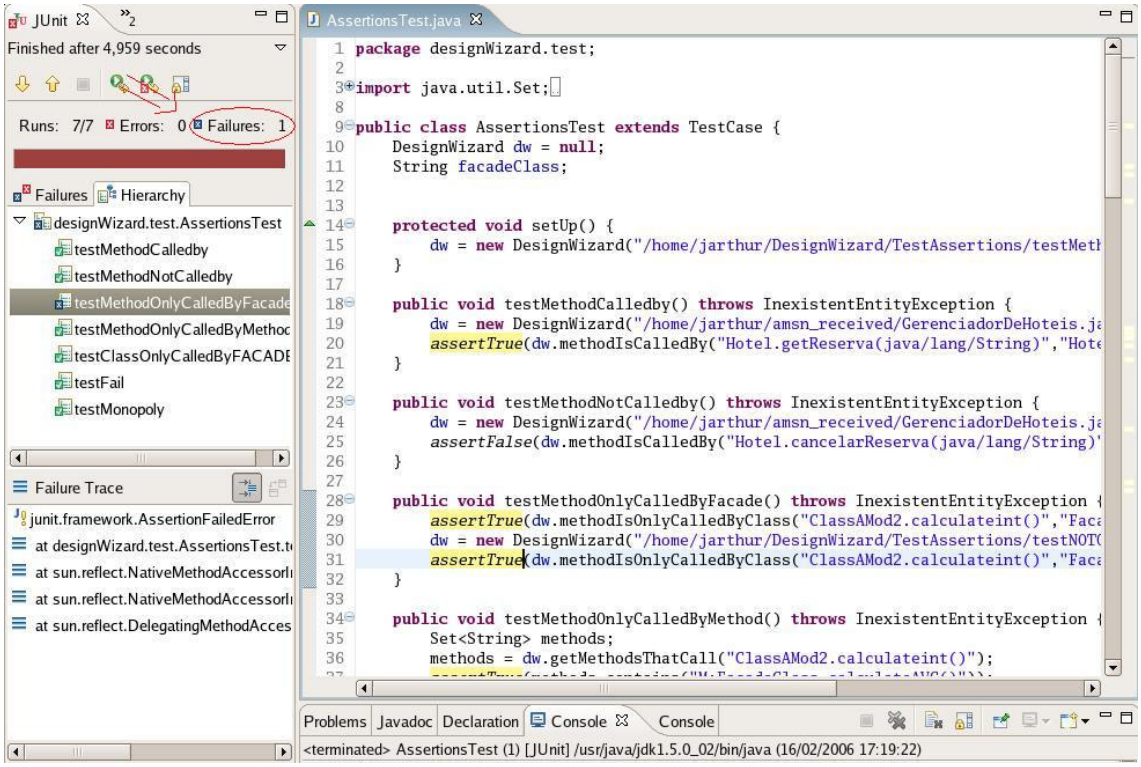

Figura 5 - Exemplo testes que certificam que algumas propriedades estruturais não estão sendo satisfeitas.

Veja na linha 31, que aparece na Figura 5, que foi mudado de assertFalse para assertTrue em relação a Figura 4. Isso causa um erro, que é denotado pelo campo em destaque que acusa um teste falho.

É possível perceber que os testes estruturais são simples de implementar, uma vez que o *framework* Junit é bem difundido na comunidade científica.

2 Rastreamento da Evolução de Projetos de Software

Com o trabalho de extração e verificação concluído, partimos então para o segundo objetivo da ferramenta: Rastrear a evolução de projetos de Software.

A tarefa constitui em extrair fatos de duas versões distintas de software e depois fazer a comparação entre as informações contidas. Seguindo esse raciocínio foi possível então, implementarmos essa funcionalidade. Adicionamos a ferramenta, um módulo que é responsável por verificar quais fatos continuam no software, quais foram retirados e quais foram adicionados, isso é feito fazendo a comparação entre as versões distintas do software. Todo esse trabalho é feito de maneira automática, cabendo ao usuário simplesmente fornecer os arquivos .jar[12] das versões distintas do software.

Veja, passo a passo, como o usuário pode acompanhar a evolução de um software, verificando quais mudanças foram efetuadas de uma versão para a outra. Para efetuar essa ação, o usuário precisa clicar em *compare* e depois em *extract two versions and compare them*. Veja,

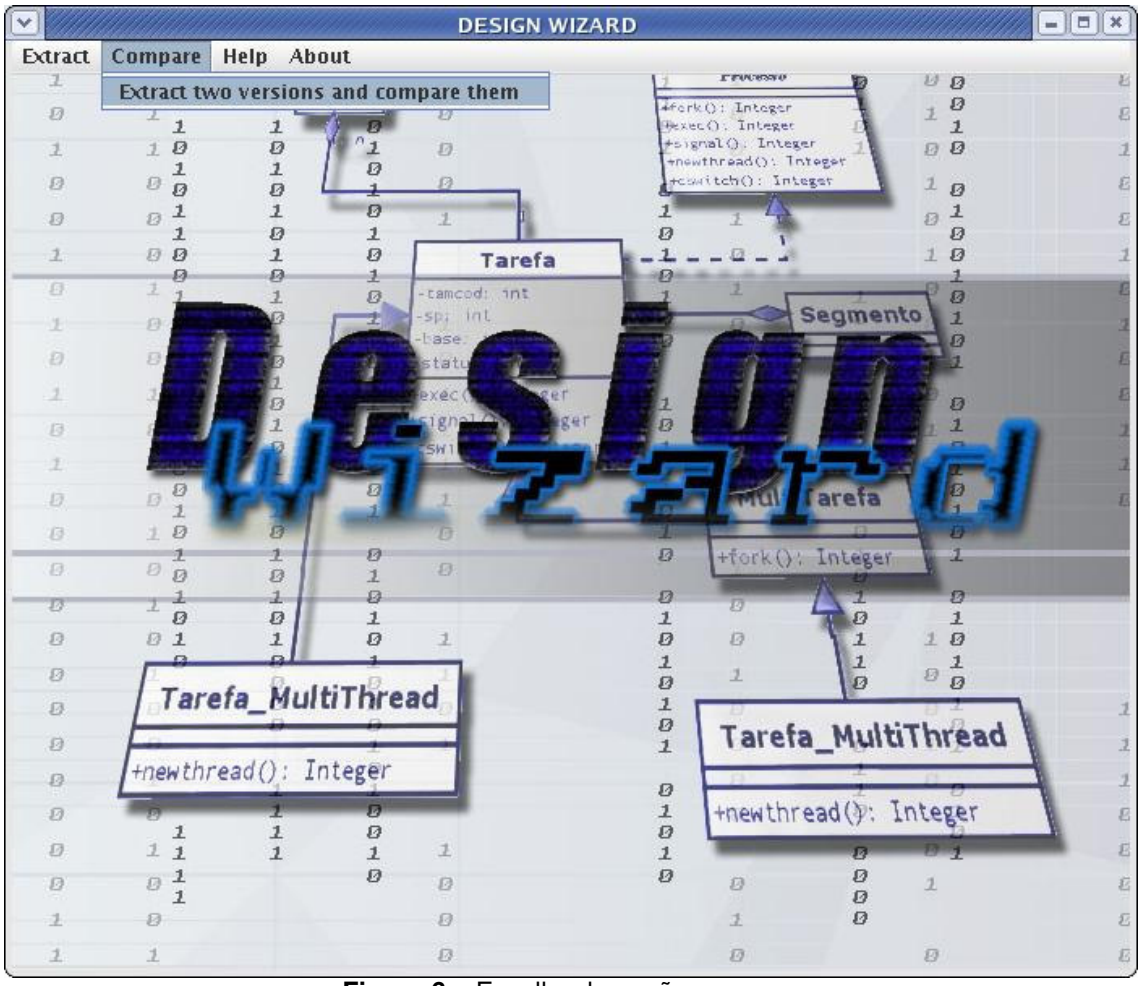

Figura 6 – Escolha da opção *compare*

Após a escolha da opção *compare*, será aberta uma janela em que o usuário e irá escolher o arquivo .jar a ser extraído, escolhemos o arquivo GerenciadorDeHoteisV1.jar que corresponde a primeira versão do software GerenciadorDeHoteis[13], que escolhemos para usar como exemplo. Veja na Figura 6,

[12] Um arquivo .jar (Executable Jar File), tem o mesmo funcionamento de um arquivo executável .exe, ele esconde a implementação (código) do usuário. Definição disponível em: http://www.portaljava.com/home/modules.php?name=Content&pa=showpage&pid=7

[13] Gerenciador de Hotéis é um software desenvolvido por alunos da disciplina de Estrutura de Dados na Universidade Federal de Campina Grande. Gerenciador de Hotéis é um software que gerencia as atividades de um Hotel, tais como reserva, aluguel, calculo de diárias etc.

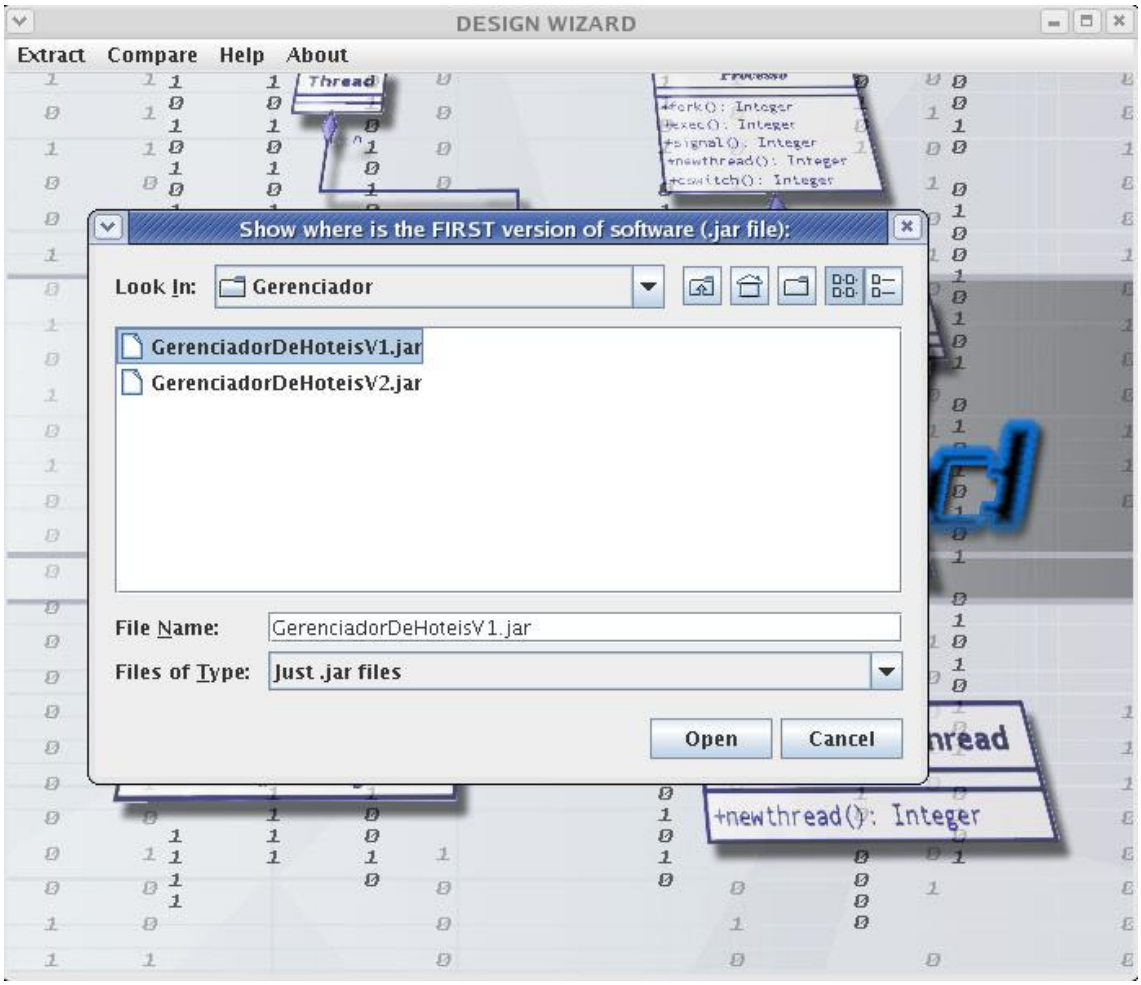

Figura 7 – Escolha da primeira .jar correspondente a primeira versão do software

A escolha do arquivo .jar correspondente a segunda versão do software (GerenciadorDeHoteisV2.jar) é feita de maneira análoga.

Após a escolha das duas versões**, automaticamente a ferramenta irá extrair os fatos e fazer a comparação entre duas versões do software**, ou seja, quais fatos foram adicionados, retirados ou mantidos. Feito isso, a ferramenta exibe um aviso de sucesso, veja na Figura 8.

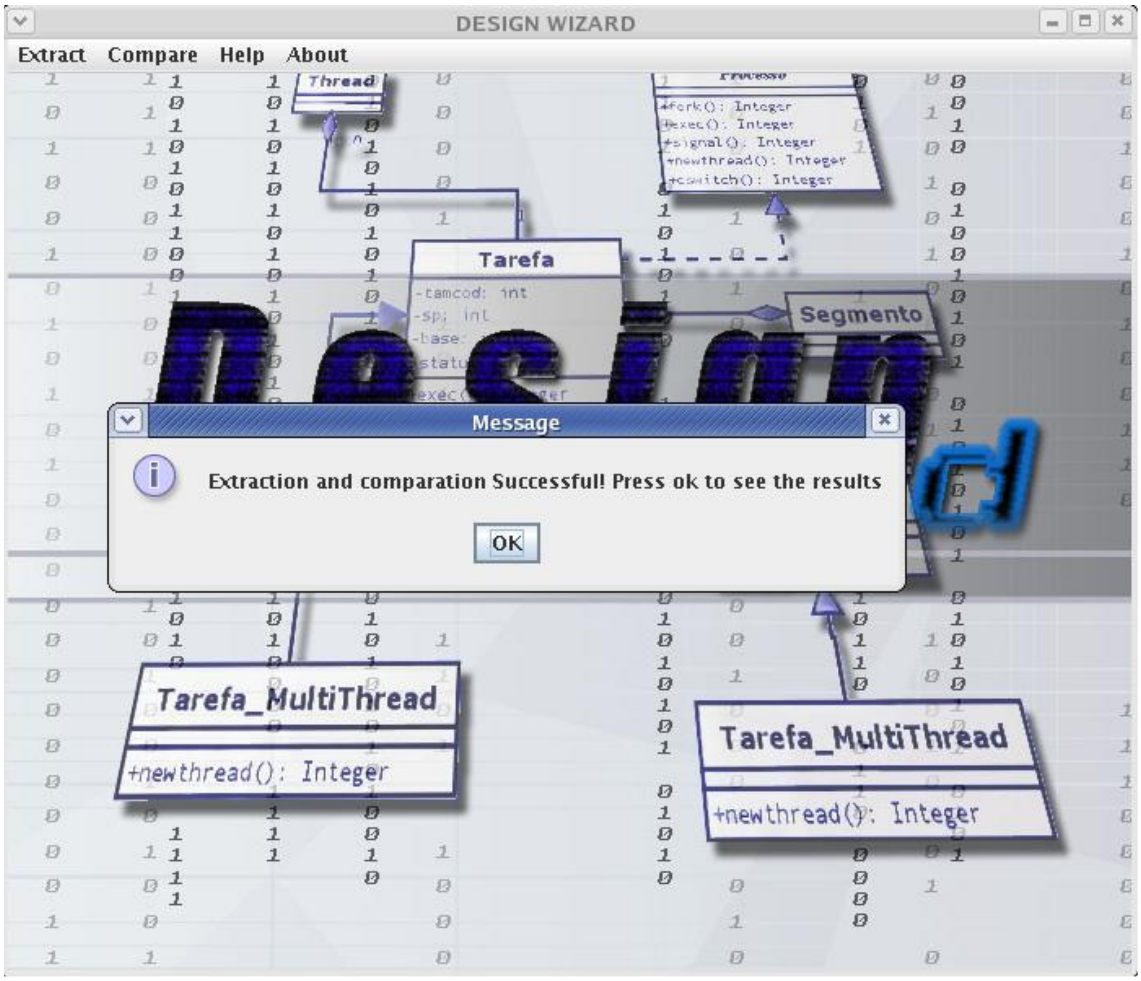

Figura 8 – Extração e verificação concluída com sucesso.

A partir desse momento o usuário tem acesso a todas as classes[14] Java do sistema que foi extraído e nelas, pode verificar quais foram as modificações efetuadas de uma versão para outra.

Veja na Figura 9 que o usuário pode verificar quais mudanças houveram nas classes do sistema. Essas mudanças são denotadas pelas palavras CONTINUED,ADDED ou REMOVED. No caso em destaque, está sendo verificado que a classe Date e seus membros (métodos e atributos), continuam não foram removidos em relação a primeira versão do software. Por isso o identificador CONTINUED.

[14] Classe Java é uma especificação, em Java, que define um tipo de objeto. Definição disponível em: http://www.unicamp.br/fea/ortega/info/cursojava/classejv.htm

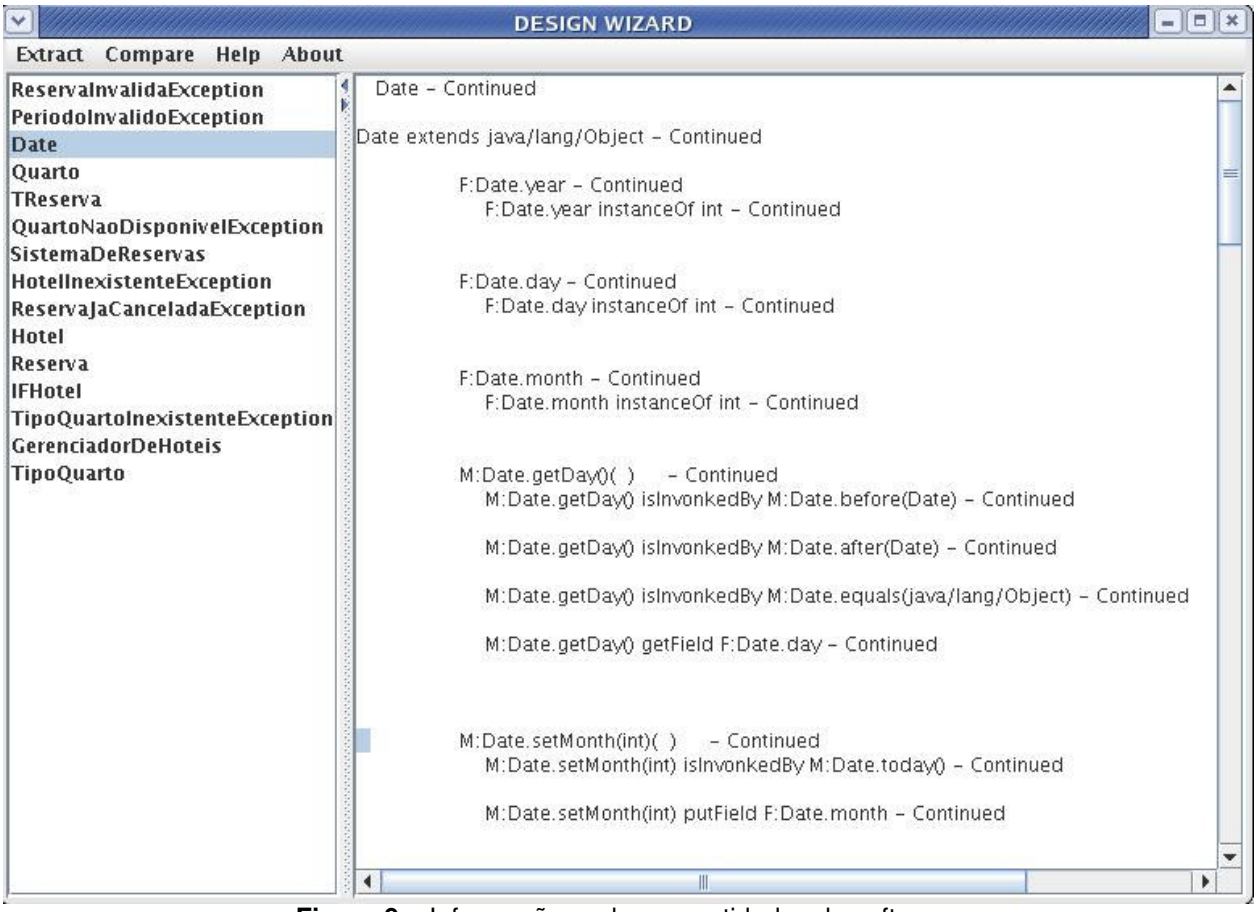

Figura 9 – Informações sobre as entidades do software.

CONCLUSÕES

Foi possível concluir que o trabalho efetuado até o momento foi produtivo, em relação a artefatos produzidos e também a conhecimentos adquiridos na área. O estudo de ferramentas como JFX serviu como embasamento teórico necessário para que o *Design Wizard* fosse desenvolvido. O contato com essas ferramentas foi importante para o inicio do nosso trabalho de iniciação científica, nos dando uma visão de quais eram os objetivos ainda não alcançados e do estado em que se encontram as soluções para o acompanhamento e verificação de modelos de software. Com essas informações ao alcance, foi possível ter certeza de que era preciso a construção de uma ferramenta que viesse a suprir as necessidades existentes no acompanhamento da evolução de softwares.

O estado atual do projeto consiste em uma ferramenta, o *Design Wizard*, que dá suporte ao projetista para que esse possa verificar propriedades do código do software que está sendo desenvolvido. Esse fato torna mais confiável a construção de um software, já que o acompanhamento da evolução do software é feito de forma automática e segura, uma vez que os testes estruturais comprovam se alguma regra está sendo violada ou não.

Além disso, a ferramenta tem como um dos seus principais adjetivos, a simplicidade de uso. Como foi descrito nesse documento, o usuário expressa os testes estruturais de maneira muito semelhante a testes de unidade. Isso aumenta a usabilidade do *Design Wizard*, já que JUnit é bem difundido na comunidade científica.

A relevância do trabalho feito com *Design Wizard* também é de suma importância. A ferramenta está sendo usada para acompanhar a evolução do projeto OurGrid, desenvolvido pelo Laboratório de Sistemas Distribuídos da Universidade Federal de Campina Grande, laboratório esse, mantido com recursos da Hewlett-Packard(HP). Essa experiência é extremamente proveitosa para o crescimento e amadurecimento de nossa ferramenta, uma vez que o projeto *OurGrid* faz o papel de "cliente", nos dando retorno sobre como podemos melhorar a ferramenta e quais funcionalidades deveríamos adicionar para que cada vez mais o *Design Wizard* se torne uma ferramenta de uso simples e eficiente.

Espera-se que o *Design Wizard* contribua de maneira direta á construção de softwares mais confiáveis. Ajudando no acompanhamento do desenvolvimento do software, checando propriedades que devem ser mantidas para que não haja diferença entre a arquitetura inicial proposta e a implementação. *Design Wizard* espera ajudar na manutenção do que chamamos conformidade da arquitetura com a implementação.

AGRADECIMENTOS

Ao CNPq pela bolsa de Iniciação Científica, a Dalton Serey e Ana Emília, pelo apoio, pela dedicação e orientação empregadas para que o projeto fosse realizado.

REFERÊNCIAS BIBLIOGRÁFICAS

AGILEMANIFESTO. **Manifesto for agile software development**. Disponível em: <http://agilemanifesto.org> . Acesso em 20 de Fevereiro de 2006.

API**. Interface de Programação de Aplicativos**. Disponível em: < http://pt.wikipedia.org/wiki/API >. Acesso em 22 de Agosto de 2006.

BOEHM AND TURNER. Boehm, Barry and Turner, Richard. **Balancing Agility and Discipline: A Guide for the Perplexed.** Addison-Wesley, 2003.

BOERMAN, 2004. Boerman, R. On **software architecture conformance in the context of evolving systems.** Master's thesis, Delft University of Technology, 2004.

BOWMAN. Bowman, I.; Godfrey, M. and Holt, R. **Extracting Source Models from Java Programs: Parse, Disassemble or Profile.** On line: http://citeseer.ist.psu.edu/687272.html.

BURDY, 2003. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., and Poll, E**. An overview of JML tools and applications**, 2003.

CHEN, 2006. Chen, Feng; D'Amorim, Marcelo and Roşu, Grigore. **Monitoring-Oriented Programming: A Tool-Supported Methodology for Higher Quality Object-Oriented Software, 2004.** On line: http://fsl.cs.uiuc.edu/mop/, Acesso em 21 de fevereiro de 2006.

CLARKE, 1999. Clarke, E., Grumberg, O., and Peled, D. **Model Checking**. MIT Press, Cambridge, MA, 1999.

COMPONENTES. **Componente de Software**. Disponível em: < http://pt.wikipedia.org/wiki/Componentes_de_Software >. Acesso em 18 de Agosto de 2006.

ECLIPSE. **Eclipse IDE**. Disponível em: < http://www.eclipse.org/ > . Acesso em 28 de Agosto de 2006.

REPOSITORIO. **Repositório**. Disponível em: < http://www.ead.unicamp.br/minicurso/cvs/texto/glossario.html >. Acesso em 22 de Março de 2006.

FRAMEWORK. **FrameWork**. Disponível em: < http://pt.wikipedia.org/wiki/Framework >. Acesso em 13 de Fevereiro de 2006.

FELDMAN, 2003. Feldman, Y. A. **Extreme Design by contract**. In XP, pages 261–270, 2003.

GAMMA&BECK, 2001. Gamma, E.; Beck, K. *JUnit*. On-line: http://junit.org, 2001.

IWPSE'03:Proceedings of the 6th International Workshop on **Principles of Software Evolution**, page 59, Washington, DC, USA. IEEE Computer Society, 2003.

JOHANSSON&WENNOLF, 2002. Johansson, C. and Wennolf, L.. **Software architecture verification tool for software platforms**. Tese de Mestrado, Lund University, 2002.

KAASTRA&KAPSER, 2006. Kaastra. M.; Kapser, C. **Java Fact Extractor**, 2003. Disponível em: < http://csg.uwaterloo.ca/~mdmkaastra/jfx/, >. Acesso em 15 de Fevereiro de 2006.

MENS, 2003. Mens, K. **Automating Architectural Conformance Checking by means of Logic Meta Programming.** Tese de Doutorado, Vrije Universiteit Brussel, 2000.

NETBEANS. **NetBeans IDE**. Disponível em: < http://www.netbeans.org/ > . Acesso em 28 de Agosto de 2006.

O'REILLY, 2004. O'Reilly, C., Morrow, P., and Bustard, D. *Lightweight prevention of architectural erosion*. In OSTROFF, 2004. Ostroff, J. S., Makalsky, D., e Paige, R. F. **Agile specification-driven development**. In XP, pages 104–112, 2004.

REYNALDS, 2005. Reynolds, P., Killian, C., Wiener, J., Vahdat, A and Mogul, J. *Pip*: **Detecting the unexpected in distributed systems**.On line: http://issg.cs.duke.edu/pip/ , 2005.

THE STANDISH, 1999. The Standish Group (1999). **Chaos, recipe for success**. Disponível em: < http://www.pm2go.com/sampleresearch/chaos1998.pdf. >. Acesso em 25 de Junho de 2005.

TURK, 2002. Turk, D., France, R., and Rumpe, B. **Limitations of agile software processes**. In Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2002), Alghero, Sardinia, Italy, 2002.

WHITTAKER, 2002. Whittaker, James A., J. M. V. **50 Years of Software: Key Principles for Quality.** IT Professional, 4(6):28–35, 2002.