Faculdade de Engenharia da Universidade do Porto

Licenciatura em Engenharia Informática e de Computação

Engenharia de Software
Refactoring

020509066 - Márcio André Brandão Ribeiro.
020509017 - André Filipe Lourenço Lessa
020509022 - Bruno Alexandre Silva de Sousa
020509043 - João Carlos Martins Eiras
020509078 - Nuno Miguel Das Neves Machado
040509093 - Rodrigo Manuel Lopes de Matos Moreira

Docentes:

Raul Moreira Vidal

Ademar Aguiar

João Carlos Pascoal de Faria

3 de Dezembro de 2004

1. Introdução

Refactoring é um processo de reorganização da estrutura interna do software, sem alterar o seu comportamento externo. É um modo disciplinado de reorganizar código, facilitando posterior depuração, eliminando código presumível de introduzir erros. De modo geral quando se refabrica código, melhora-se o desenho, a estruturação deste, após ter sido escrito.

No entanto, típico de uma boa pratica aquando o planeamento do software, primeiro define-se a estrutura, o design, depois programa-se. Mas, resultante de sucessivas modificações do código, ao longo do tempo, e do sistema em que está inserido, o código gradualmente perde a estruturação pretendida. O processo de refactoring tende a contrariar esta pratica, capaz de produzir código caótico, que no entanto refabricado, adquire uma boa estruturação. A realização de simples e subtis alterações de código, cumulativamente, pode radicalmente melhorá-lo, e simultaneamente introduzir erros. Dai um comportamento disciplinado aquando o refactoring, seguindo uma lógica de passos tomados a cada alteração, nos quais estão indubitavelmente incluídos testes a secções de código.

Assim o design do código é algo contínuo durante o desenvolvimento do software, assegurando uma boa estruturação deste.

2. Princípios

2.1. Definição de refactoring

Não existe, à partida, uma definição concreta, do que *refactoring* realmente significa. Dependendo do contexto, e de pessoa para pessoa, pode haver algumas diferenças.

No entanto, vamos considerar a definição de *refacotring*, segundo o que pesquisamos nas referências bibliográficas, que se insere mais correctamente no contexto.

Encontramos então, dois significados do uso da palavra *refactoring*, como nome e como verbo:

Refactoring (nome): Modo de alteração da estrutura interna de um software, tornando-o de mais fácil compreensão e mais acessível a alterações, sem alterar o seu comportamento.

To Refactor ou **refabricar** (verbo): reestruturar o software existente, por várias aplicações de *refactoring,* sem alterar o seu comportamento.

Para uma boa parte de programadores na área de desenvolvimento de software, o *refactoring* consiste numa limpeza de código para melhor compreensão. Mas na verdade, esta prática passa por uma reformulação do código, aplicando uma série de métodos descritos posteriormente, de maneira a que o programa execute sobre um menor conjunto de custos, não afectando as funcionalidades do programa em geral.

Em muitos casos, a prática do *refactoring*, não fica só por uma melhor "maneira de dizer" todo o programa em si para baixar os custos. Quando um software se encontra em desenvolvimento, várias funcionalidades vão sendo acrescentadas ao longo do tempo, o *refactoring* resolve também, o encontro de um melhor caminho de acrescentar uma nova função, alterando o código existente, para facilitar a sua implementação. Este caso surge muitas vezes, quando alternamos entre reestruturar o código existente e adicionar novas capacidades durante o processo de desenvolvimento de um programa.

2.2. Porquê o refactoring?

A importância do *refactoring* surge, como já foi dito anteriormente, para garantir maior eficiência e facilidade de compreensão de como está organizado, Porém, não significa que a aplicação desta prática seja a solução para todos os problemas de código, mas no entanto, é considerado essencial.

Para além das vantagens já referidas, o *refactoring* permite aos programadores, evitar escrever código desnecessário. Por exemplo, um programador inicia o desenvolvimento, seguido de outro, e assim por diante até ao seu fim, cada um à sua maneira sem se reocupar com o "aspecto" do código. Uma reconstrução do código segundo um conjunto de regras universais, pode evitar este problema, assim como outros associados, como por exemplo, a repetição da mesma parte do código para criar uma funcionalidade ou limitar a possibilidade de alterar partes do código sem afectar o programa em geral.

Uma das grandes dificuldades do *refactoring* é conciliar a comunicação entre os programadores e a maquina. Enquanto o código é construído pelo programador, este preocupa-se com o entendimento entre si e a máquina, onde o meio de comunicação é o código. Ao criar uma função, o programador procura que o seu objectivo seja cumprido, esquecendo muitas vezes, que o próximo elemento da equipa a trabalhar no projecto terá de entender o seu código. Programando recorrendo ao *refactoring*, é uma forma de tornar universal o aspecto do código. Isto torna-se útil, mesmo para o próprio programador, quando for necessária uma alteração depois de um enorme espaço de tempo.

Outra vantagem na programação do mesmo código é a detecção de erros. O programador pode perder muito tempo a encontrar um *bug* se a estrutura do código estiver demasiado confusa. Com o programa bem organizado, o problema fica

bastante facilitado, principalmente para o conjunto de pessoas que sente muita dificuldade nessa área.

Por fim, o hábito de uma boa construção de programação, pode diminuir o tempo desperdiçado, sendo possível manter a qualidade num tempo não muito extenso.

2.3. Quando deve ser aplicado o refactoring?

A questão mais frequente dos programadores relativamente a este assunto é quando deve ser feita a recriação do programa. A resposta vem dependendo de vários factores, tanto dos elementos da equipa, como o *software* em si. Mas não há concretamente um grande espaço de tempo onde se aplique o *refactoring*, é algo cuja pratica surge com a necessidade em pequenos espaços de tempo.

Um método conhecido para determinar quando se aplica o *refactoring* é, por exemplo, o programador constrói uma função, mais tarde volta a criar algo semelhante, à terceira vez que voltar a acontecer, aplica-se a reconstrução do código para o generalizar.

Outra ocasião para voltar a olhar para trás no código é quando vamos tentar implementar uma nova característica ao nosso programa. Obter uma boa base no código vai torná-lo mais conveniente à nova implementação. Nesta situação torna-se mais difícil manter o software com uma eficiência óptima, uma vez que a maior parte dos programadores coloca em prioridade a facilidade de implementação.

Talvez a oportunidade mais comum, e por vezes involuntária, de reconsiderarmos uma alteração do software, é quando nos deparamos com um *bug* de programação. Neste caso, é obrigatório rescrever as partes necessárias, sendo por vezes necessário mudanças radicais.

A situação de *refactoring* é invocada quando estamos a fazer uma revisão de código, principalmente, se o código tratado nos é alheio. Considera-se uma boa parte do tempo no decorrer do desenvolvimento do *software*, quando o código de outros não é suficientemente compreensível. Quando este inconveniente ocorre, o *refactoring*, além de simplificar, pode evitar que este tipo de incidentes se verifique entre os membros da equipa.

2.4. Problemas com o *refactoring*

Nem em todos os casos de programação e desenvolvimento de software o *refactoring* é tão satisfatório de modo a compensar as penalizações no tempo de garantir bons progressos. Existem também desvantagens na consolidação de código por meio de o reformular para o tornar mais simples.

Em todas as empresas em que um sistema de bases de dados é vital, existem problemas em consolida-lo quando grandes alterações têm de ser feitas, devido às associações entre os objectos.

No campo do desenvolvimento de interfaces, verificam-se problemas com a alteração do *software*, onde cada código, quer do programa, quer da visualização, são à partida distintos, e até uma mudança de um nome pode destabilizar, provavelmente causando um erro de compilação. Na maior parte das vezes, a parte correspondente à interface depende da porção de código que gera as funcionalidades do programa, mas o contrário não acontece. Facto este que o programador tem de ter em mente quando se trata de uma mudança brusca.

Podemos considerar ainda os casos mais comuns, os já referidos códigos alheios, situações complicadas de mau *design*, ou código de difícil compreensão. Em todos estes casos, quanto maior for a mudança, mais obstáculos serão encontrados, e maior será o tempo desperdiçado.

2.5. De onde surgiu o *refactoring*?

Não foi encontrada nenhuma informação concreta de onde o *refactoring* realmente veio, e quem teve a primeira ideia de que seria uma boa prática que poderia aumentar a qualidade do *software*. Porém desde sempre que um bom programador passou alguma parte do seu tempo, sob algum método de manutenção, para obter um código limpo.

Segundo a bibliografia consultada, dois dos pioneiros desta técnica foram *Ward Cunningham* e *Kent Beck*, que trabalhavam para o *Smalltalk*, no início da década de 80. O *Smalltalk* consistia num ambiente de desenvolvimento de software, cuja linguagem era orientada a objectos e tinha suporte para interfaces. O método de compilação era bastante rudimentar. Consoante o seu desenvolvimento, os autores do projecto, aperceberam que refinando o código obteriam melhores resultados no âmbito de criar software mais rápido.

Desde então a prática do *refactoring* passou a ser uma política muito apoiada na comunidade do projecto *Smalltalk*, onde mais tarde viria a ser cada vez mais utilizada na pesquisa e desenvolvimento de *software*, em base na linguagem C++.

A verdade, é que hoje-em-dia, o *refactoring* faz parte da aprendizagem de uma linguagem de programação, sendo cada vez mais exigida eficiência e compreensão do código, através da utilização de normas para o conceber.

3. Métodos de refactoring

3.1. Composição de Métodos

Uma grande parte de *refactoring* em programação orientada a objectos, consiste em compor métodos de modo a empacotar o código. A maioria dos problemas surgem dos métodos demasiado longos, devido ao facto de estes conterem muita informação, a qual, pode trazer mais complexidade neste processo. Quando isto acontece, converte-se esse fragmento de código proveniente desse método, no seu próprio método. A designação do método que descreve este comportamento, é o "Extract Method", que constitui ele próprio, a chave para o refactoring. O maior problema deste método é o lidar com variáveis locais, em que também as variáveis temporárias constituem uma das maiores fontes deste problema. Quando se pretende eliminar todas as variáveis temporárias, utiliza-se o "Replace Temp with Query". Se as variáveis temporárias forem utilizadas para vários contextos utiliza-se o "Split Temporary Variable".

Contudo, por vezes, existem variáveis temporárias que se tornam complicadas substituí-las. É necessário usar o "Replace Method with Method Object". Isto permite partir o método que até então se mostrava complicado e inseparável, com o custo de se ter de introduzir uma nova classe para o efeito. Relativamente aos parâmetros, estes constituem um problema menos complicado do que o das variáveis temporárias.

O "Inline Method" é exactamente o contrário do "Extract Method", quando se encontra uma chamada de um método, substitui-se pelo corpo do código respectivo. Este método é bastante útil quando se fazem múltiplas extracções de código e se apercebe que o resultado produzido por alguns métodos não está a ter o "peso suficiente", ou quando se precisa de reorganizar a maneira através da qual os métodos foram "partidos".

Quando um método é "partido" torna-se mais fácil perceber como é que ele funciona, podendo-se assim, encontrar uma maneira de melhorar o algoritmo actual e substituí-lo por outro mais claro e eficaz.

3.2. Mover características entre objectos

A decisão mais fundamental e importante no design de um objecto é decidir onde colocar as suas "responsabilidades". Este mover de características entre objectos é uma solução muito utilizada em *refactoring*.

Normalmente para resolver os problemas que surgem desta situação, utilizam-se o "Move Method" e o "Move Field", para mover o comportamento. Se for necessário utilizarem-se ambos os métodos, existe a preferência de invocar primeiro o "Move Field" seguido do "Move Method".

Na maioria das vezes, as classes estão carregadas com muitas "responsabilidades". Quando isto se verifica usa-se "Extract Class" para separar algumas dessas

"responsabilidades". Se uma classe se torna demasiado "irresponsável" utiliza-se "Inline Class" , para a juntar com uma outra classe. Se a outra classe estiver a ser utilizada, é frequentemente útil esconder este facto com "Hide Delegate". Às vezes esconder uma classe resulta constantemente em mudar a relação do proprietário. Sempre que tal se verificar necessita-se de usar "Remove Middle Man".

Os *refactorings* "Introduce Foreign Method" e "Introduce Local Extension", são casos especiais. Apenas são utilizados quando não é possível aceder ao código fonte de uma classe, contudo pretendem-se mover "responsabilidades" para esta imutável classe. Se for apenas um ou dois métodos utiliza-se "Introduce Foreign Method"; para mais de um ou dois métodos usa-se "Introduce Local Extension".

3.3. Simplificar as chamadas aos métodos

A chave para desenvolver software adequado e fiável orientado a objectos, consiste na utilização de interfaces que são fáceis de perceber e utilizar. Iremos abordar neste texto os *refactorings* que tornam os interfaces mais directos.

O nomear é a ferramenta chave para comunicar. Este processo é simples e importante consistindo apenas em mudar o nome de um método. Quando nos apercebemos do comportamento de um programa, podemos usar "Rename Method" de forma a criar esse conhecimento. Deve-se também, apesar de não ser obrigatório, renomear variáveis e classes. No conjunto, todos estes processos de renomear consistem em substituir texto, razão pela qual não é necessário adicionar *refactorings* extra.

"Add Parameter" e "Remove Parameter" são *refactorings* comuns, que desempenham um papel importante nas interfaces. Normalmente programadores iniciantes a objectos usam, por vezes, listas de parâmetros longas. Listas essas que são típicas de outros ambientes de desenvolvimento. Os objectos permitem-nos manter listas de paramêtros reduzidas, e outros *refactorings* fornecem-nos maneiras de encurtar essas listas. Quando estamos a passar vários valores de um objecto usa-se "Preserve Whole Object" por forma a reduzir todos os valores num único objecto. Se esse objecto não existir podemos criá-lo com o "Introduce Parameter Object". Se se conseguir obter dados para os quais o método já teve acesso, podemos eliminar parâmetros com "Replace Parameter With Explicit Methods". Podemos combinar, também, vários métodos similares adicionando parâmet

ros através do "Parameterize Method".

Uma das convenções mais valiosas utilizadas por Martin Fowler, foi separar métodos que mudam o estado – os chamados "modifiers" – daqueles que interrogam o estado. Sempre que estes aparecerem combinados utiliza-se "Separate Query From Modifier", de forma a nos livrarmos deles.

Podemos melhorar um interface ocultando coisas. Todos os dados devem estar escondidos, assim como, alguns métodos sempre que possível se possam também ocultar. Aquando do refactoring necessitamos por vezes de tornar algumas coisas

visíveis durante algum tempo, e depois voltarmos a ocultá-los. Para tal usa-se o "Hide Method" e o "Remove Setting Method".

Os construtores são uma característica particularmente não conveniente de Java, porque nos obrigam a saber a classe do objecto que pretendemos criar. Dado que não necessitamos de saber isso usamos "Replace Constructor With Factory Method".

O Java e C++ têm um mecanismo de controlo de excepções. Programadores que não estão habituados a isto, normalmente utilizam códigos de erro para sinalizarem os problemas. Usa-se "Replace Error Code With Exception" para se utilizar as novas excepções. Contudo existem excepções que não constituem a resposta adequada, daí termos que testar primeiro com "Replace Exception With Test".

3.4. Organização de dados

Muitas pessoas acham o "Self Encapsulate Field" desnecessário. É sempre discutível se um objecto deverá aceder aos dados de forma directa ou através de acessores. Muitas vezes são necessários os acessores, então poderemos obtê-los com o "Self Encapsulate Field".

Uma das coisas mais úteis das linguagens orientadas aos objectos é que nos permitem definir novos tipos, que vão além do que se poderá fazer com os tipos de dados simples das linguagens tradicionais. Poderá demorar algum tempo para nos habituarmos mas, sempre que começamos com um valor de dados simples, percebemos que o uso de um objecto seria bastante melhor. "Replace Data Value with Object" permite transformar esses dados em objectos articulados. Quando o realizamos, esses objectos são instâncias que serão precisas por todo o programa. Poderemos usar "Change Value to Reference" para os tornar em referências.

Se um "array" actua como estrutura de dados, poderemos torná-la mais clara com "Replace Array with Object". A grande vantagem surge quando se usa o "Move Method" para dar comportamento aos novos objectos.

Números mágicos, ou números com um significado especial, sempre foram um problema. Apesar de se evitar usá-los, estes continuam a aparecer. Para isso usa-se o método "Replace Magic Number with Symbolic Constant" para nos livrarmos destes números, sempre que entendemos o que eles representam.

As ligações entre objectos poderão ser unidireccionais ou bidireccionais. As associações unidireccionais são mais simples, mas muitas vezes precisamos de as alterar para bidireccionais, de modo a suportar uma nova função. Para isso usamos "Change Unidirectional Association to Bidirectional". Por outro lado, o "Change Bidirectional Association to Unidirectional" remove a complexidade desnecessária, quando entendemos que afinal não iremos precisar da associação bidireccional.

Uma forma de dados que necessita de tratamento especial é o código "tipo": um valor especial que indica algo particular sobre o tipo de instância. Estes costumam aparecer como enumerações, normalmente implementados como inteiros finais estáticos. Se os códigos são apenas para informação e não alteram o comportamento da classe, podemos usar o "Replace Type Code with Class", o qual nos dá um melhor tipo de verificação e uma plataforma para ver se o comportamento se altera

posteriormente. Se o código "tipo" altera o comportamento da classe, usa-se "Replace Type Code with Subclasses", se possível. Se não for possível, poderemos usar o método "Replace Type Code with State/Strategy". Este método é mais complicado, mas também mais flexível.

3.5. Simplificação de Expressões Condicionais

A lógica condicional tem uma forma de começar complicada, pelo que apresentamos aqui um conjunto de "refactorings" para a tornar mais simples. O "refactoring" central é o "Decompose Conditional", o qual envolve quebrar uma condicional em partes.

Os outros "refactorings" neste capítulo envolvem outros casos importantes. Usamos o "Consolidate Conditional Expression" quando temos vários testes com o mesmo efeito. Usamos o "Consolidate Duplicate Conditional Fragments" para remover qualquer duplicado existente no código da condicional.

Os programas orientados a objectos costumam ter menos comportamento condicional relativamente aos programas processuais, porque muito do comportamento condicional é gerado por polimorfismo. Polimorfismo é melhor porque o que chama não necessita saber do comportamento condicional, e torna-se mais fácil aumentar as condições. Todas que aparecem são candidatas ao método "Replace Conditional with Polymorphism".

Um dos mais usuais, mas menos óbvio, uso do polimorfismo é usando o "use Introduce Null Object" para remover as entradas para um valor nulo.

4. Testes

O primeiro passo, no processo de refactoring, é construir uma base de testes suficientemente sólida, permitindo-nos assegurar que as alterações feitas ao código não alteram o funcionamento geral do programa. Mesmo utilizando ferramentas que executam automaticamente o refactoring, estes continuam a ter um papel importante. Tê-los permite poupar tempo a nível da programação, pois reduzem o tempo gasto no debugging. Uma forma de implementar os testes, consiste em que cada classe de um dado programa contenha os seus próprios testes, assim executando estes testes e acrescentando código ao já existente, no caso de erros, torna-se muito mais fácil determinar onde se encontram. Portanto, à medida que se vai implementando código deve-se ao mesmo tempo criar testes, para esse mesmo código, sendo uma atitude correspondente a *extreme programming*.

4.1. JUnit

Uma ferramenta muito utilizada para testes em Java é o JUnit, simples e intuitiva, foi escrita por Erich Gamma e Kent Beck [JUnit]. Esta permite ao utilizador agrupar os seus testes em grupos (Fig.1), facilitando-lhe a possibilidade de correr vários testes no mesmo instante.

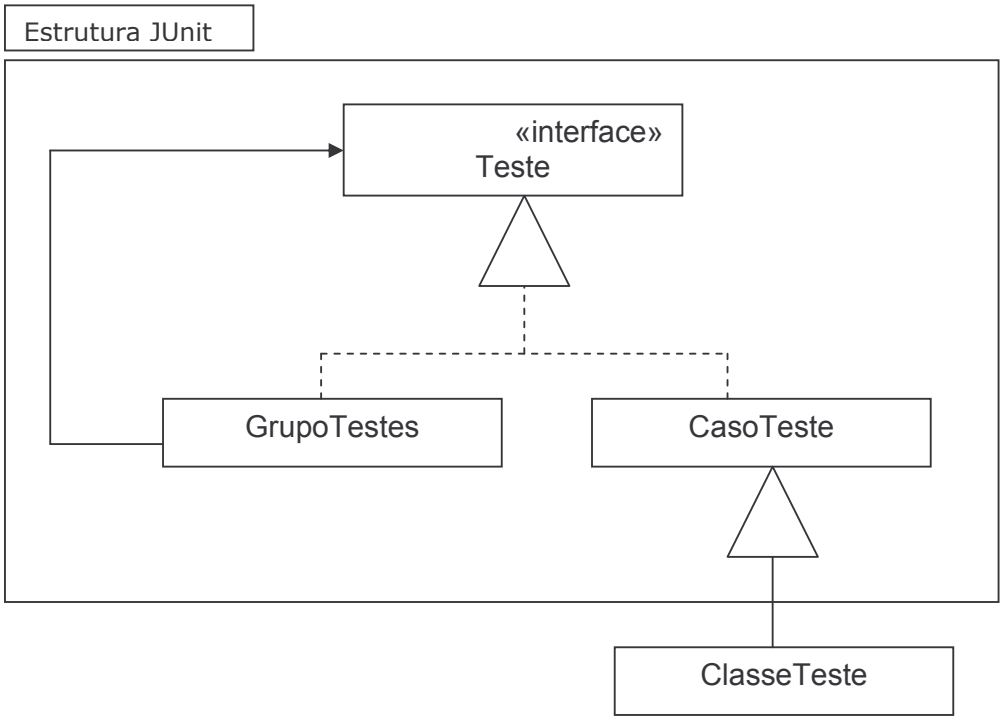

Fig.1 – Estrutura de Testes

O JUnit trabalha com «*unit tests*», ou seja, cada classe de teste trabalha dentro de um simples pacote (package), testando a interface para outros pacotes assumindo que tudo o resto funciona. Existe no entanto um outro tipo de testes – *«functional tests»*, estes são escritos com o objectivo de testar o software como um todo, tratando o sistema como se este fosse uma caixa preta. Quem desenvolve este tipo de teste dá uma maior importância ao utilizador final do software, do que à actividade dos programadores que o desenvolvem.

O JUnit inclui uma interface gráfica (Fig.2), que permite ao utilizador uma mais fácil utilização do programa. A barra «progress», apresenta uma cor verde, caso todos

os testes passem e vermelho no caso contrário. Existe ainda a possibilidade de correr o JUnit sem a interface.

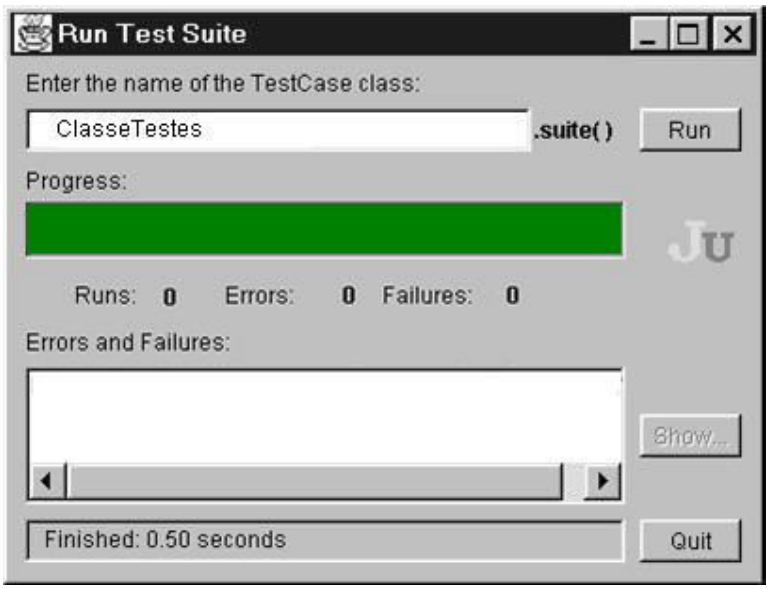

Fig.2 – interface gráfica do JUnit

4.2. Porquê, quando e o que testar?

Porquê testar o software? Porque o ser humano tem limitações cerebrais, o que o leva a cometer muitos erros, particularmente quando o número o linhas e complexidade do código, começa a ser elevada. Quanto menor for o número de erros, maior é a qualidade do código implementado, logo menor é a probabilidade de um programador ser acordado as 3 da manha por causa de um servidor que «crashou».

Quando testar? Testar ao longo da implementação, para que se possa encontrar rapidamente os erros que possa ter cometido e assim corrigi-los rapidamente. Quando encontra-mos um erro, devemos fazer um teste, por forma a corrigi-lo e acrescenta-lo a nossa lista de testes, pois um erro tem tendência a repetir-se.

O que testar? Todos os valores susceptíveis de criar erros, condições fronteiras e limite.

Testar pode ser trabalhoso mas traz os seus benefícios, pois torna o código mais seguro e optimizado.

5. Bibliografia

FOWLER, Martin; Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999

JUnit, [Em linha] Disponível em www.junit.org/ [Consultado em 11/2004]

Refactoring, [Em linha] Disponível em http://en.wikipedia.org/wiki/Refactoring [Consultado em 11/2004]

Refactoring, [Em linha] Disponível em www.refactoring.com/ [Consultado em 11/2004]

Refactoring, [Em linha] Disponível em http://c2.com/cgi/wiki?WhatIsRefactoring [Consultado em 11/2004]