PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Curso de Sistemas de Informação - Barreiro

DESENVOLVIMENTO GUIADO POR TESTES:

Comparando a qualidade do design gerado através do processo de desenvolvimento guiado por plano versus processo de desenvolvimento guiado por testes.

Daniel Cassemiro Freire

Belo Horizonte
2011

Daniel Cassemiro Freire

DESENVOLVIMENTO GUIADO POR TESTES:

Comparando a qualidade do design gerado através do processo de desenvolvimento guiado por plano versus processo de desenvolvimento guiado por testes.

Monografia apresentada ao Curso de Sistemas de Informação da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Orientadora: Maria Augusta Vieira Nelson

Belo Horizonte
2011

Dedicado a meus pais,

Que sempre me apoiaram e me incentivaram a nunca desistir

AGRADECIMENTOS

Gostaria de agradecer a Deus, aos meus pais, por insistirem e sempre acreditarem no meu trabalho e esforço.

Agradeço também a professora Guta, grande incentivadora e que não me fez desistir. Também agradeço ao colega Guilherme Froes pelo apoio e crítica dada ao trabalho e aos meus colegas de trabalho da Techbiz que me apoiaram e me apoiam diversas vezes.

RESUMO

Modelos tradicionais de desenvolvimento de software são caracterizados por seguirem um plano rígido e bem definido. Existem fases bem definidas para a definição do problema, o desenho da solução para resolver esse problema, a construção da solução, os testes e a entrega do produto para o cliente. Paralelo a isso, existem metodologias de desenvolvimento de software que pregam a agilidade e a absorção das mudanças. Nessas metodologias o foco maior é o código e o produto e as mudanças são mais bem aceitas. Uma das técnicas usadas geralmente em projetos desenvolvidos através de metodologias ágeis é o Desenvolvimento Guiado Por Testes (TDD). Essa técnica consiste em codificar antes um teste que verifica se um requisito é assertivo. Após isso, codifica-se a parte funcional, que vai fazer com que esse requisito seja atendido. O presente trabalho pretende comparar se o Design de um sistema é afetado, de maneira positiva ou negativa, pela construção através de técnica ágil e guiada por testes ao invés de uma construção guiada por um plano rígido. Para isso, foi usado um sistema de avaliação de desempenho de funcionários, construído por uma empresa de desenvolvimento de software e concentrado em um trecho desse sistema. Os mesmos requisitos desse trecho foram feitos através da técnica de desenvolvimento guiada por testes (TDD) e métricas de análise foram coletadas e comparadas. Após a análise dessas métricas, foi possível concluir que a rigidez exigida por uma metodologia guiada por plano faz com que uma complexidade desnecessária seja inserida no sistema, refletindo no design do sistema, por consequência. O TDD auxilia não apenas na qualidade do Sistema de informação desenvolvido, indicando facilmente que os requisitos, implementados por testes, estejam funcionando, como ajuda a manter uma solução de design mais simples e limpa.

Palavras-chave: Desenvolvimento guiado por testes, Extreme Programming, GQM, Big DesignUp Front, princípios SOLID

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Germoglio (2009) define alguns princípios e técnicas que resultam em um bom produto de design de software. Dentre eles, citamos o Acoplamento e a Coesão que servem como medida para medir a divisão de módulos do Design de um software.

O desenvolvimento de software usando uma metodologia guiada por planos (baseadas em RUP, por exemplo) defende a criação do Design do Software como uma atividade do processo de desenvolvimento de software. Segundo Jether (2010), a disciplina de Análise e Design tem como um dos objetivos desenvolver uma arquitetura estável para possibilitar o desenvolvimento de todo o sistema. Dessa maneira, nota-se que é uma atividade a ser feita antes da codificação e implementação, atuando como a construção de uma espécie de alicerce para o desenvolvimento do software.

Já o desenvolvimento guiado por testes tem como premissa refazer o Design do Software a todo instante, a partir da prática de refatoração. Beck (2003) defende a prática de um ciclo curto de desenvolvimento no qual se escreva um teste automatizado que falhe; codificação de um código que apenas faça o teste passar; e finalmente uma refatoração do código.

1.1 Definição do problema

O problema abordado neste trabalho é justamente comparar a qualidade de um Design de Software gerado a partir do desenvolvimento de sistemas guiado por planos, no qual o Design do Software é criado antes da codificação com o Design de Software que é criado e modificado a todo instante, através da prática do desenvolvimento guiado por testes.

1.2 Objetivos e Motivação

O objetivo principal deste trabalho é comparar qualitativamente o Design de Software produzido através de atividades antes da codificação, num contexto de desenvolvimento guiado por planos e o Design de Software através do desenvolvimento guiado por planos e analisar o impacto do ato de refatoração contínua no Design de Software no processo de desenvolvimento guiado por testes.

1.2.1 Objetivos específicos

- Definir como é feita a construção do Design de Software em desenvolvimento guiado por planos e como é feita a construção do Design de Software em desenvolvimento guiado por testes.
- Estabelecer métricas qualitativas confiáveis para Design de Software.
- Comparar o Design de Software de uma aplicação desenvolvida por uma metodologia guiada por planos com a mesma aplicação reescrita através do desenvolvimento guiado por testes.
- Analisar o impacto da refatoração contínua no Design de Software gerado através do desenvolvimento guiado por testes.

1.3 Justificativa

No contexto da Engenharia de Software, o Design de Software é um dos pilares para se construir um bom Software. Através do Design de Software é possível garantir expansibilidade às funcionalidades existentes. Segundo Wells (2010), um bom Design de Software é aquele considerado simples e justamente por ser simples, se gasta menos tempo para fazer o que precisa ser feito.

Ambler(2009) ainda defende que documentação extensa pode fazer com que ao se criar o Design de Software, ele se torna complicado demais. E que o bom Design de Software é feito de maneira incremental e que ele é de extrema importância que deve ser revisto dia a dia.

Com a metodologia de desenvolvimento guiado por testes, o Design é revisto a todo instante. Segundo Beck(2003),dentre as implicações que levam o desenvolvimento guiado por testes, estão o refatoramento.

Acredita-se que tais abordagens podem contribuir para um Design de Software que sejam mais extensíveis e com propriedades que podem classificá-los como de maior qualidade que o Design de Software produzido por metodologias de desenvolvimento guiadas por planos.

Em contrapartida, não existem trabalhos que façam a comparação entre dois designs produzidos através de abordagens diferentes. Este trabalho visa estabelecer estes pontos de comparação para comprovar ou contradizer esta hipótese sobre design de softwares.

1.4 Resultados esperados/contribuições

No presente trabalho espera-se verificar se de fato a metodologia de desenvolvimento guiada por testes produz um Design de Software com uma qualidade maior do que metodologias de desenvolvimento guiadas por planos, que defendem a criação do Design de Software como um artefato para a produção de códigos.

Pretende-se ainda analisar as métricas que são utilizadas para mensurar qualidade de Design de Software.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Desenvolvimento guiado por testes

O desenvolvimento dirigido por testes (Test driven-development ou TDD), segundo o Beck (2003), é uma abordagem de desenvolvimento que combina o desenvolvimento de testes unitários antes que se escreva o código apenas suficiente para passar no teste unitário e então refatora-se o código. Segundo Beck (2003), algumas "forças" nos desviam de um código limpo e que funciona. A proposta desse estilo de desenvolvimento de software é fazer com que o desenvolvedor tenha um feedback constante, pois o ciclo de repetições é bem curto. Obviamente isso implica em um código simples e que faz apenas o que os testes especificam. Ainda segundo Beck (2003) a efetiva implantação do Desenvolvimento dirigido por testes leva a algumas implicações positivas, como:

Design da aplicação crescendo organicamente, no qual o código rodando provê feedback instantâneo durante as decisões.

O desenvolvedor escreve seus próprios testes, evitando espera para que uma outra pessoa escreva estes testes.

O ambiente de desenvolvimento deve prover rápidas respostas a mudanças rápidas.

O design da aplicação deve apresentar alta coesão e baixo acoplamento. Dessa maneira, fica mais fácil poder testar o design gerado.

Beck (2003) também define as três regras que devem obrigatoriamente serem seguidas no Desenvolvimento dirigido por testes:

- Red: Escrever um pequeno teste que não funciona. Talvez nem mesmo seja compilado.
- Green: Faça o teste funcionar rapidamente, cometendo alguns pecados necessários no processo.
- Refactor: Eliminar todas as duplicações criadas apenas para fazer o teste passar.

O Desenvolvimento dirigido por testes visa encorajar desenvolvedores a codificarem o que realmente é pedido. Beck (2003) menciona algumas boas ações que o time de desenvolvimento obtém ao adotar o Desenvolvimento dirigido por testes:

- Ao invés de iniciar tentando codificar, inicia-se aprendendo concretamente o mais rápido possível.
- Ao invés de ficar calado, o time comunica mais claramente.
- Ao invés de evitar feedbacks, o time busca auxílios e feedbacks concretos.

Beck (2003), entretanto deixa claro que apesar do Desenvolvimento dirigido por teste ser uma técnica oriunda do Extreme Programming, é uma técnica que não é exclusiva do Extreme Programming. É possível praticar Desenvolvimento dirigido por teste sem aplicar demais técnicas do Extreme Programming.

2.2 Design de aplicações

Design de Software, segundo Katki (1991), é tanto o processo de definição da arquitetura, módulos, interfaces e outras características de um sistema quanto ao resultado desse processo. Ou seja, o Design de Software é o artefato produzido através de diversas técnicas para se produzir um desenho do sistema de acordo com os requisitos que foram levantados. Podemos entender Design de Software como um desenho do sistema, que reflete o modelo do problema de acordo com os requisitos obtidos até o momento.

Segundo Germoglio (2009) existem alguns princípios e técnicas que são essenciais estarem presentes no bom Design de Software. São eles:

- Divisão e conquista.
- Abstração.
- Encapsulamento.
- Modularização.
- Separação de preocupações.
- Acoplamento e coesão.
- Separação de política de execução de algoritmos.
- Separação de interfaces de suas implementações.

Vale destacar o princípio de Acoplamento e Coesão, que segundo Germoglio, “são princípios usados para medir se módulos de um Design foram bem divididos”. Entende-se como Acoplamento do Design de Software a medida de interdependência entre módulos de software. E como Coesão a medida da relação entre tarefas dentro de um mesmo módulo.

Existem algumas abordagens que tratam a maneira com que um software é desenhado. Metodologias baseadas no Processo Unificado (UP) defendem que o Design de Software deve ser uma solução tecnológica para o problema exposto. Segundo Jether (2010), a finalidade da disciplina de Análise e Design é:

- Transformar os requisitos no modelo de design sob qual o sistema será construído.
- Desenvolver/evoluir uma arquitetura estável e robusta para o sistema.
- Adaptar o Design com foco na implementação.

Logo, é possível verificar que o modelo deve ser criado antes da implementação. Isso leva à construção de um design feito antes da codificação, definido por alguns autores como Big Design Up Front.

Evoluir sobre a abordagem BDUF – vantagens de se planejar TODO o design antes da aplicação. Onde obter? Verificar no material de metodologias dirigidas por planos.

Outra abordagem é a de que o Design deve evoluir conforme a construção do sistema. Em metodologias ágeis como o Extreme Programming, defende-se que a atividade de criação do Design deve ser praticada a todo instante e não apenas no início do projeto. Abordagens como o Design Evolutivo e a Refatoração de código permitem que as preocupações com o Design da aplicação fiquem apenas nas funcionalidades a serem implementadas naquele momento, porém deixando pontos para a extensão.

Segundo Marchenko (2008), o Design Incremental é o reflexo do fato de que o melhor Design emerge da tentativa e erro. Porém ele menciona que o Design Incremental não proíbe de se pensar em questões de alto nível do sistema, mas o time é encorajado a planejar em detalhes apenas o que está sendo feito naquele instante e fazer a evolução do Design iteração por iteração de acordo com as prioridades do cliente.

2.3 Requisitos

Um requisito é definido, de acordo com o Manual do RUP, como "uma condição ou capacidade coma qual o sistema deve estar de acordo".

Segundo Grady (1992) uma maneira de categorizar os Requisitos de acordo com o modelo FURPS+, um acrônimo para descrever as principais categorias de requisitos.

- Funcionalidade: Os Requisitos podem incluir conjunto de recursos, habilidades e segurança.
- Usabilidade: Os requisitos de usabilidade podem incluir subcategorias como fatores humanos, estética, consistência na interface do usuário, ajuda on-line e contextual, assistentes e agentes, documentação do usuário e materiais de treinamento.
- Confiabilidade: Os requisitos de confiabilidade a serem considerados são freqüência e gravidade de falha, possibilidade de recuperação, possibilidade de previsão, exatidão e tempo médio entre falhas (MTBF).
- Desempenho: Um requisito de desempenho impõe condições aos requisitos funcionais. Por exemplo, para uma determinada ação, ele pode especificar parâmetros de desempenho para velocidade, eficiência, disponibilidade, exatidão, taxa de transferência, tempo de resposta, tempo de recuperação e uso de recurso.
- Suportabilidade: Os requisitos de suporte podem incluir possibilidade de teste, extensibilidade, adaptabilidade, manutenibilidade, compatibilidade, possibilidade de configuração, possibilidade de serviço, possibilidade de instalação e possibilidade de localização (internacionalização).

O "+" em FURPS+ menciona que não deve ser esquecido Requisitos como:

- Requisitos de Design (ou restrições de Design): Um requisito de design, frequentemente chamado de uma restrição de design, especifica ou restringe o design de um sistema.
- Requisitos de Implementação: Um requisito de implementação especifica ou restringe o código ou a construção de um sistema. Como exemplos, podemos citar padrões obrigatórios, linguagens de implementação, políticas de integridade de banco de dados, limites de recursos e ambientes operacionais.

- Requisito de Interface: Um requisito de interface especifica um item externo com o qual o sistema deve interagir ou restrições de formatos, tempos ou outros fatores usados por tal interação.
- Requisito Físico: Um requisito físico especifica uma característica física que um sistema deve possuir, por exemplo, material, forma, tamanho e peso. Esse tipo de requisito pode ser usado para representar requisitos de hardware, como as configurações físicas de rede obrigatórias.

As metodologias dirigidas por planos defendem a produção de artefatos da coleta e gerenciamento de Requisitos. Um dos principais artefatos produzidos pela disciplina de Requisitos, segundo o Manual do RUP são os documentos de Caso de Uso. Segundo o Manual do RUP, um caso de uso define um conjunto de instâncias de casos de uso, no qual cada instância é uma sequência de ações realizada por um sistema que produz um resultado de valor observável para determinado ator.

Abordar metodologias dirigidas por planos e coleta de requisitos

Já a abordagem dada pelas metodologias ágeis defende uma flexibilidade e aceitação maior a mudanças de requisitos. O Extreme Programming, segundo o Ambler (2008), é definido como uma abordagem de desenvolvimento de software que valoriza mais a mudança de requisitos e tem como principal artefato o código fonte produzido.

Metodologias ágeis, apesar de defenderem a redução do custo de mudanças, por dispensar artefatos documentais excessivos para gerenciar os requisitos, possuem diversas maneiras de fazer esta gestão. Testes unitários, segundo Beck (2003), asseguram que uma determinada funcionalidade é atendida pelo seu código funcional, através de um indicador explícito. Um conjunto de funcionalidades é escrito em forma de testes unitários codificados. Se o código que implementa "quebra" estes testes sabemos que a funcionalidade não está corretamente implementada. Ainda temos as User Stories (Estórias de Usuário) que são descrições textuais parecidas com os Casos de Uso e servem para auxiliar a criação dos Testes de Aceitação. Segundo Wells (2009), as Estórias de Usuário devem ser criadas também para auxiliar a criação dos Testes de Aceitação.

2.4 Métricas de Software

As Métricas de Software, segundo Boehm (2000), são dados numéricos relacionados ao desenvolvimento de software. As Métricas de Software suportam fortemente as atividades de Gerência de Projetos.

Ainda segundo Boehm (2000), as métricas de software se relacionam com quatro atividades do Gerenciamento de Projeto, que são:

- Planejamento: Métricas servem como base para estimativa de custo, planejamento de treinamento, planejamento de recursos.
- Organização: Métricas de tamanho influenciam a organização do projeto.
- Controle: Métricas são usadas para relatar e rastrear atividades do desenvolvimento de software para o cumprimento dos planos.
- Melhorias: Métricas são usadas como uma ferramenta para o processo de melhorias e identificar onde os esforços de melhorias podem ser concentrados e medir esforços na melhoria dos processos.

As Métricas de Software vem cada vez mais ganhando espaço nas empresas de desenvolvimento de software. Pressman (2006) cita diversos motivos para isso, como indicar a qualidade do produto; avaliar a produtividade das pessoas que produzem o produto; formar uma base histórica para estimativas; ajudar a justificar os pedidos de novas ferramentas ou treinamento adicional.

Uma maneira de agrupar e classificar as Métricas de Software, segundo o Wikipedia, é:

- Métricas da produtividade, baseadas na saída do processo de desenvolvimento do software com o objetivo de avaliar o próprio processo;
- Métricas da qualidade, que permitem indicar o nível de resposta do software às exigências explícitas e implícitas do cliente;
- Métricas técnicas, nas quais se encaixam aspectos como funcionalidade, modularidade, manutenibilidade, etc.

2.4.1 GQM – Goal, Question, Metrics

De acordo com Basili, Caldiera e Rombach (1994), assim como em qualquer Engenharia, o Desenvolvimento de Sistemas também precisa obter um mecanismo para efetuar medições e obter feedbacks. Ainda de acordo com Basili, Caldiera e Rombach (1994), para que as medições sejam eficazes devem ser:

- Focadas em metas específicas.
- Aplicadas a todo o ciclo de vida do produto, processos e recursos.
- Interpretada com base na caracterização e compreensão da organização ambiente, contexto e objetivos.

Então, Basili, Caldiera e Rombach (1994) define que a abordagem GQM pressupõe que para uma organização medir de forma proposital, deve primeiro especificar os objetivos para si e seus projetos, então ele deve traçar as metas para os dados que se destinam definir os objetivos operacionalmente e então fornecer um framework para interpretação dos dados.

3 TRABALHOS RELACIONADOS

Aniche, Gerosa (2010) apresentam um estudo sobre o impacto do desenvolvimento dirigido por testes na qualidade do Design de software. Os autores apresentam uma série de métricas destinadas a medir a qualidade do software, mas não fazem um estudo de caso comparando este tipo de design com o design gerado através do desenvolvimento dirigido por planos. O trabalho apresentado é bastante relevante pela definição das métricas usadas para medir qualidade do design de software.

Baseado na abordagem GQM, os autores estipularam algumas métricas que poderão ser usadas neste trabalho. São elas:

M1: *LCOM*. O princípio da responsabilidade única diz que uma classe deve ter apenas um único motivo para mudar. Quando isso ocorre, é dito que essa classe possui alta coesão. A métrica LCOM (*Lack of Cohesion of Methods*) mede a falta de coesão dos métodos.

M2: Acoplamento eferente e aferente. Acoplamento é o grau de dependência de um módulo em relação a todos os outros módulos do sistema. Um baixo acoplamento indica um sistema bem estruturado e de mais fácil manutenção.

M3: Instabilidade. Segundo Martin (2006), um design é dito rígido quando uma alteração desencadeia alterações em módulos independentes. Isso faz com que seja difícil prever o custo de uma alteração. A métrica de instabilidade visa, portanto, medir o grau de resistência a mudanças de uma classe.

M4: Abstração. Pacotes que são totalmente estáveis devem ter abstração máxima e pacotes instáveis devem ser totalmente concretos. Ou seja, a abstração de um determinado módulo é totalmente proporcional à sua estabilidade. A métrica sugerida calcula o grau de abstração de determinado módulo ou pacote.

M5: Distância da sequência principal. É a relação entre estabilidade e abstração. Ao traçar um gráfico, tendo estabilidade no eixo X e abstração no eixo Y, a sequência principal seria a

reta que liga os pontos (0,1) e (1,0). Essa métrica calcula a distância do módulo em relação à sequência principal.

M6: Complexidade ciclomática. Mede o número de diferentes caminhos que possam ser seguidos em um método. Um alto valor indica código complexo, difícil de ser mantido.

M7: Número de parâmetros por método. Calcula o número de parâmetros por método. Como levantado por Janzen (2008), códigos produzidos por TDD costumam receber mais parâmetros, já que eles tendem a expor as suas dependências mais explicitamente.

M8: Total de complexidade ciclomática recursiva que não pode ser dublada. Segue o mesmo raciocínio da complexidade ciclomática detalhada acima, com exceção dos elementos que possam ser substituídos por objetos dublês, que não são contabilizados.

M9: Número de linhas por método. Calcula o número de linhas por método. Métodos com muitas linhas podem indicar código complexo e até problemas de design.

M10: Revisão de código por especialista. Um especialista fará uma avaliação do design final, ressaltando as qualidades e os defeitos encontrados.

4 METODOLOGIA

É proposto como forma de validar o presente trabalho a realização de um estudo em uma empresa especializada em desenvolvimento de software de Belo Horizonte, Minas Gerais. Este estudo será focado em um sistema apenas, desenvolvido através de uma metodologia guiada por planos e que o design de software foi definido antes da construção do sistema.

Este trabalho foi dividido em cinco etapas: a primeira consiste em pesquisar sobre os modelos de desenvolvimento de software guiados por planos e os modelos de desenvolvimento de sistemas guiados por teste; a segunda, estabelecer métricas confiáveis que permitam a classificação qualitativa de um design de software; a terceira etapa consiste em analisar o problema proposto a ser tratado no sistema da empresa alvo deste estudo e coletar seus requisitos; a quarta etapa envolve a codificação dos requisitos que são analisados no modelo de desenvolvimento guiado por testes; a quinta e última etapa finalmente verifica os resultados obtidos através das métricas estabelecidas e chega-se às conclusões finais.

A primeira etapa é composta da pesquisa do referencial teórico, sugerido pela orientadora, além de material coletado através de pesquisas. Dessa maneira, as métricas confiáveis que serão usadas ao longo do trabalho serão obtidas e explicadas, cumprindo assim a meta da segunda etapa. A terceira etapa é composta da obtenção de requisitos de uma empresa de desenvolvimento de sistemas, localizada em Belo Horizonte. Apenas a Especificação de Requisitos será utilizada neste momento e mais nenhum outro artefato. Um sistema já existente será usado como base para as medições. A funcionalidade do sistema a ser estudada ainda não foi definida. A quarta etapa envolve o desenvolvimento de um novo modelo, porém usando o processo de desenvolvimento guiado por testes. As atividades de desenvolvimento serão feitas por uma única pessoa e ela se guiará baseada unicamente na Especificação de Requisitos. Finalmente, a quinta e última etapa consiste em coletar os dados de ambos as modelos, desenvolvidos através de uma metodologia guiada por planos e através da metodologia guiada por testes, estabelecidos através de métricas de

design definidas anteriormente. Será feita a contagem e posteriormente chega-se às conclusões finais.

5 DESENVOLVIMENTO

O trabalho consiste em aplicar técnicas de desenvolvimento guiado por testes (TDD) em requisitos já definidos e codificados anteriormente por uma empresa especializada em desenvolvimento de sistemas.

No estudo, o desenvolvedor que está codificando usando a técnica TDD, é também o autor do trabalho, não tem larga experiência usando a técnica e não teve contato com o modelo feito inicialmente.

A lista de requisitos foi definida num documento formal e inicialmente foram levantados os requisitos funcionais e regras de negócio. A lista de requisitos se encontra no Apêndice xxx.

A linguagem de programação que é usada para o experimento é C#, que é a mesma usada no sistema já existente. No trabalho também será usada a IDE de desenvolvimento Visual Studio 2010.

A codificação da aplicação usando TDD se deu através da leitura de um documento de especificação de requisitos, com um formato que é bastante comum e usual para seguir um desenvolvimento guiado por planos, de acordo com o MANUAL DO RUP (2010).

Apesar de o documento conter desde o detalhamento macro dos requisitos do sistema até um detalhamento bem específico, com detalhamento de Casos de Uso, por exemplo, foram utilizadas como base para o estudo deste trabalho apenas as Regras de Negócio e os Requisitos Funcionais que constam nesse documento.

Durante a codificação algumas métricas de qualidade de design de código são executadas e gravadas para posterior análise. As métricas são analisadas através de uma ferramenta integrada ao Visual Studio 2010. As métricas analisadas são definidas abaixo, segundo o site oficial da ferramenta, o MSDN (http://msdn.microsoft.com/en-us/library/bb385914.aspx)

•Cyclomatic Complexity (complexidade ciclomática): Mede a complexidade estrutural do código. Ela é criada calculando o número de diferentes caminhos do código no fluxo do programa. Um programa que tem um complexo fluxo de controle irá requerir mais testes para atingir a meta de cobertura de código e será menos manutenível.

•Class Coupling (Acoplamento de classes): Mede o acoplamento para classes através de parâmetros, variáveis locais, tipos de retorno, chamada a métodos, instanciação de generics ou templates, classes base, implementação de interfaces, campos definidos em tipos externos e decoração de atributos. Bom design de software diz que tipos e métodos devem ter alta coesão e baixo acoplamento. Alto acoplamento indica um design que é difícil de reusar e manter porque ele é muito interdependente de outros tipos.

•Lines of Code (linhas de código): Indica o número aproximado de linhas de código. A conta é baseada no código IL (intermediare language) e não é o número exato de linhas que está no arquivo do código fonte. Um alto número de linhas deve indicar que um tipo ou método está tentando fazer muito trabalho e deve ser quebrado. Indica também que o método ou tipo é difícil de prestar manutenção.

•Maintainability Index (Índice de manutenibilidade): Calcula um valor de índice entre 0 e 100 que representa uma facilidade relativa de manutenção de código. Um alto valor significa melhor manutenibilidade. Cores também são usadas para identificar rapidamente pontos de conflito no seu código. A cor verde representa a faixa de valor entre 20 e 100 e indica que o código tem boa manutenibilidade. O valor amarelo indica uma faixa entre 10 e 19 e indica que o código é razoavelmente manutenível. A cor vermelha indica valor entre 0 e 9 e indica baixa manutenibilidade.

De posse desses índices e seguindo o TDD Mantra, [Beck 2000], o trabalho se dá nos seguintes passos:

1) Um dado requisito é selecionado para codificação.

2) Escreve-se um teste para esse requisito que falha.

3) Ajusta-se o código funcional para que o teste funcione.

4) É analisada alguma possível possibilidade de refatoração e implementada.

Em alguns pontos são extraídas as métricas de acesso e registradas em uma planilha com os valores extraídos. A refatoração é baseada nos princípios SOLID, introduzidos por Martin(2002). Os princípios são:

•Single reponsibility principle
•Open/closed principle
•Liskov substitution principle
•Interface segregation principle
•Dependency inversion principle

5.1. Exemplo da implementação de um requisito usando TDD

Esta seção apresenta a exemplificação da implementação de um requisito usando TDD. Como foi dito anteriormente, o que foi utilizado para Elicitação de Requisitos foi um documento de Especificação de Requisitos, usado para documentar um sistema já existente que foi construído através do modelo de desenvolvimento guiado por planos, o que invariavelmente leva a planejar todo o design antes mesmo da codificação (BDUF).

Podemos usar como exemplo, a implementação do Requisito Funcional a seguir: *"Quando um Processo de Avaliação de Desempenho for criado, ele deve possuir três avaliações: A AUTOAVALIAÇÃO, a AVALIAÇÃO DO GESTOR e a AVALIAÇÃO DE CONSENSO".*

Desse modo, foi visto que temos em posse um requisito, que é a necessidade de um Processo de Avaliação de Desempenho deve comportar de tal forma quando for criado, isto é, deve possuir três avaliações. E elas devem ser do tipo "AUTOAVALIAÇÃO", "AVALIAÇÃO DO GESTOR" e "AVALIAÇÃO DE CONSENSO".

Sendo assim, o desenvolvedor adicionou no projeto uma nova classe de testes. A intenção é implementar o comportamento de uma entidade que foi identificada numa rápida lida dos Requisitos Funcionais e as Regras de Negócio. É importante frisar que um modelo que foi previamente desenhado anteriormente não foi desenhado com a preocupação de se obter o modelo final do sistema, mas apenas para nortear o desenvolvimento e dar uma base mínima para prosseguir através dos testes.

A classe *"ProcessoDeAvaliacaoDeDesempenhoTests"* foi criada. Este nome foi escolhido para que os testes ali implementados sejam relacionados com os comportamentos esperados para a entidade chamada *"ProcessoDeAvaliacaoDeDesempenho"*, que é o Processo de Avaliação de Desempenho, citado na Especificação de Requisitos.

O código inicial que a IDE Visual Studio gera para uma classe de testes usa um template, que é mostrado abaixo:

```
[TestClass]
    public class ProcessoDeAvaliacaoDeDesempenhoTests
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
```

O primeiro teste a ser feito será um teste para verificar a quantidade de Avaliações que devem estar associadas a um Processo De Avaliação de Desempenho. Para isso, definimos utilizar o padrão de escrita de testes chamado AAA (Arrange, Act, Assert). Nosso método de teste então ficou dessa forma:

```
[TestMethod]
public void
Dado_Um_Processo_De_Avaliacao_De_Desempenho_Ao_Ser_Criado_Deve_Assoc
iar_Tres_Avaliacoes()
{
            //Arrange
            var administrador = new Administrador();
            var gestorJose = Gestor.CriarGestor("José");

            //Act
            var processoDeAvaliacaoDeDesempenho =
administrador.CriarProcessoDeAvaliacaoDeDesempenho("Processo de
Avaliacao 1", gestorJose);

            //Assert
            Assert.AreEqual(3,
processoDeAvaliacaoDeDesempenho.QuantidadeDeAvaliacoes);
 }
```

Ao compilar, o projeto retorna erros, já que classes e métodos não foram encontrados. A classe “Administrador” deve ser criada, já que existe a necessidade do Processo de Avaliação de Desempenho ser criado pelo usuário administrador.

A classe “Administrador” foi criada e sua referência para o projeto de testes foi adicionada. Ao compilar novamente, um outro erro foi obtido. O método *“CriarProcessoDeAvaliacaoDeDesempenho”* não existe na classe *“Administrador”*. Após solucionar todos os problemas apontados na compilação, apenas para que funcione, os testes foram executados.

Conforme figura abaixo, o teste falha, de acordo com o mantra do TDD. A primeira execução irá falhar.

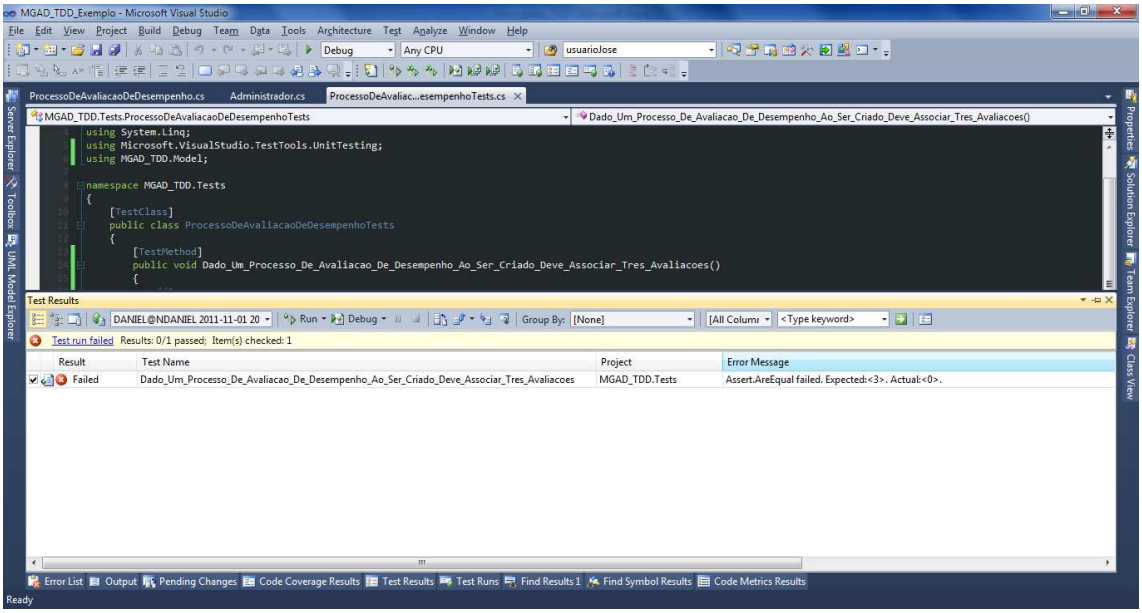

Figura 1: Primeira execução dos testes

Podemos simplesmente adicionar ao código funcional o valor “3”. Desse modo o teste passa. Note que ainda não efetuamos a refatoração no código, e essa etapa é previsível no TDD. O código funcional da classe “ProcessoDeAvaliacaoDeDesempenho” fica dessa forma:

```
public class ProcessoDeAvaliacaoDeDesempenho
    {
        public int QuantidadeDeAvaliacoes
        {
            get {   return 3;  }
            private set { }
        }
    }
```

Vejam agora que ao rodar o teste, ele passa normalmente.

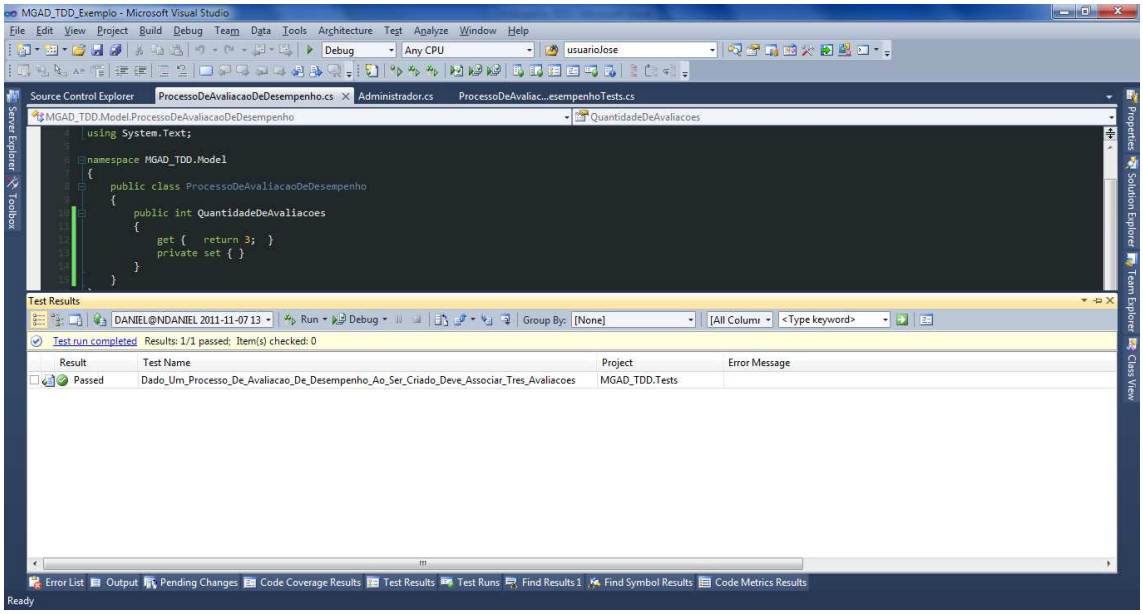

Figura 2: Segunda execução dos testes

Agora temos um teste que funciona. Ele verifica a quantidade de avaliações associadas a um processo de avaliação de desempenho e funciona. Vamos agora à implementação da criação de uma das avaliações exigidas pelo requisito no Processo de Avaliação de Desempenho.

O teste para implementação desta funcionalidade é este:

```
        [TestMethod]
        public void
Dado_Um_Processo_De_Avaliacao_De_Desempenho_Ao_Ser_Criado_Deve_Possu
ir_Uma_Auto_Avaliacao()
        {
            //Arrange
            var administrador = new Administrador();

            //Act
            var processoDeAvaliacaoDeDesempenho =
administrador.CriarProcessoDeAvaliacaoDeDesempenho();

            //Assert
            Assert.AreEqual("Auto avaliação",
processoDeAvaliacaoDeDesempenho.SelecionarAvaliacaoPorTipo("Auto
avaliação").Tipo);
        }
```

Note na criação do método “SelecionarAvaliacaoPorTipo”, que recupera uma avaliação associada ao Processo de Avaliação de Desempenho, ao passar seu tipo como parâmetro do método. Esse método não existe, e ao tentar compilar, recebemos um erro de compilação.

Criamos então este método e executamos o código. Ainda não foi suficiente, pois não existe a classe “Avaliacao”. Vamos criar a classe, já criando o atributo “Tipo”, necessário neste teste a ser feito. Agora sim, o projeto está compilando corretamente.

Ao executar os testes, no entanto, recebemos um erro no novo teste feito. Note que o TDD pede que executemos todos os testes existentes. A figura 3 apresenta:

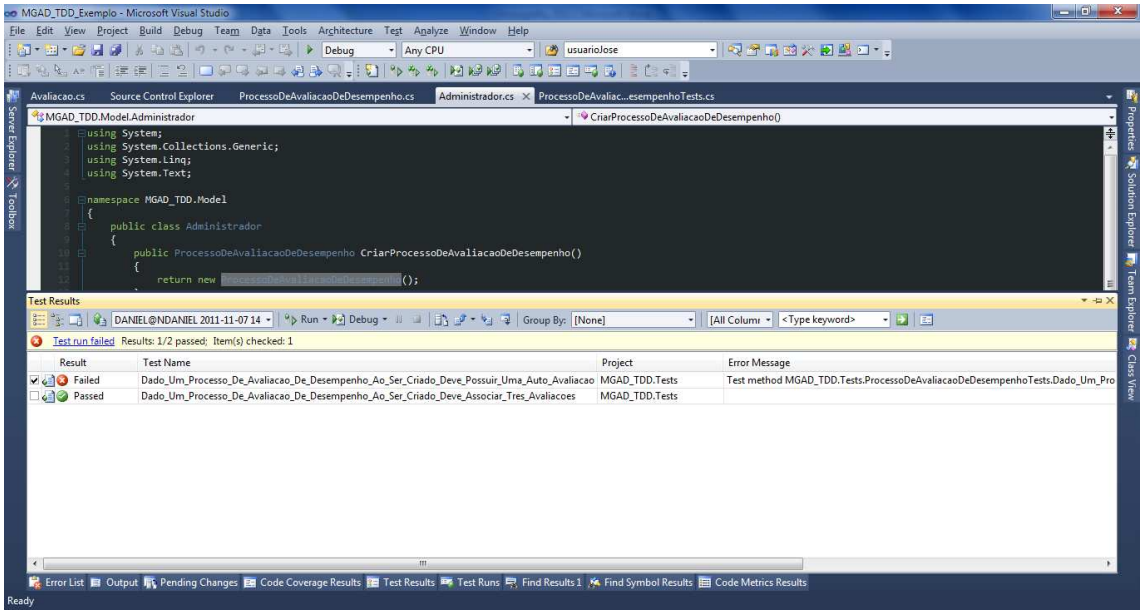

FIgura 3: Terceira execução dos testes

Podemos implementar, dentro do método “SelecionarAvaliacaoPorTipo”, uma simples instanciação do objeto “Avaliacao”, informando o tipo da avaliação. Para isso, foi criado um método que instancia a classe “Avaliacao” e configura o valor do Tipo para o valor desejado. O código da classe segue abaixo:

```
public class ProcessoDeAvaliacaoDeDesempenho
{
    public int QuantidadeDeAvaliacoes
    {
        get {   return 3;  }
        private set { }
    }

    public Avaliacao SelecionarAvaliacaoPorTipo(string tipo)
    {
        return Avaliacao.CriarAvaliacao("Auto avaliação");
    }
}
```

E dessa forma, os testes foram executados com sucesso. Veja que ainda não foram executadas refatorações. A próxima implementação exigirá refatoração.

O próximo teste será recuperar então uma “AVALIAÇÃO DO GESTOR” que está contida em um Processo de Avaliação de Desempenho. Seu código é o seguinte:

```
[TestMethod]
        public void
Dado_Um_Processo_De_Avaliacao_De_Desempenho_Ao_Ser_Criado_Deve_Possu
ir_Uma_Avaliacao_Do_Gestor()
        {
            //Arrange
            var administrador = new Administrador();
            //Act
            var processoDeAvaliacaoDeDesempenho =
administrador.CriarProcessoDeAvaliacaoDeDesempenho();
            //Assert
            Assert.AreEqual("Avaliação do gestor",
processoDeAvaliacaoDeDesempenho.SelecionarAvaliacaoPorTipo("Avaliaçã
o do gestor").Tipo);
        }
```

Essa implementação irá compilar, mas a execução do teste irá falhar, já que o método “SelecionarAvaliacaoPorTipo” cria explicitamente uma avaliação do tipo “AUTO AVALIAÇÃO”. Sendo assim, é o momento de refatorar a classe “ProcessoDeAvaliacaoDeDesempenho” e fazer com que ela realmente crie as avaliações necessárias. Veja o código abaixo. Repare que o construtor de “ProcessoDeAvaliacaoDeDesempenho” cria as avaliações até então exigidas e que o método “SelecionarAvaliacaoPorTipo“ faz uma busca entre as avaliações disponíveis.

```
    public class ProcessoDeAvaliacaoDeDesempenho
    {
        private List<Avaliacao> avaliacoes;

        public ProcessoDeAvaliacaoDeDesempenho()
        {
            avaliacoes = new List<Avaliacao>();

            avaliacoes.Add(Avaliacao.CriarAvaliacao("Auto
avaliação"));
            avaliacoes.Add(Avaliacao.CriarAvaliacao("Avaliação do
gestor"));
        }

        public int QuantidadeDeAvaliacoes
        {
            get {   return 3;  }
```

```
            private set { }
        }

        public Avaliacao SelecionarAvaliacaoPorTipo(string tipo)
        {
            return avaliacoes.Find(a => a.Tipo == tipo);
        }
    }
```

Dessa forma, os testes agora passam. Abaixo uma figura da IDE Visual Studio, com todos os testes com sinal positivo:

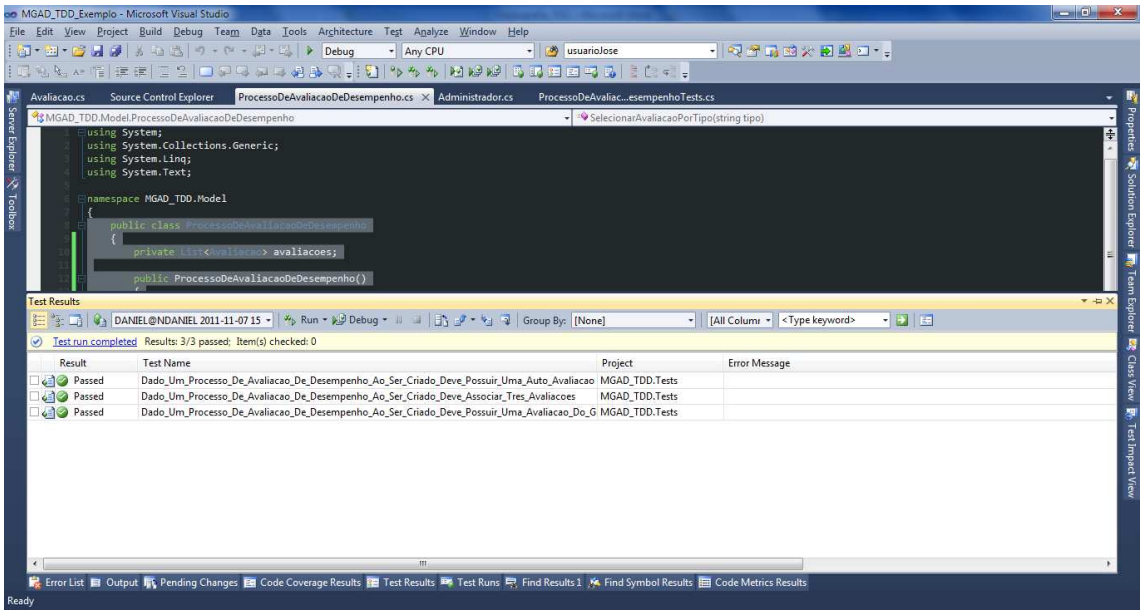

FIgura 4: Quarta execução dos testes

Agora iremos implementar o último teste, que é a necessidade de uma avaliação de consenso. Este teste será bem parecido com os últimos dois testes criados, mas irá falhar, conforme prevê o mantra do TDD. Sua implementação segue abaixo:

```
[TestMethod]
        public void
Dado_Um_Processo_De_Avaliacao_De_Desempenho_Ao_Ser_Criado_Deve_Possu
ir_Uma_Avaliacao_De_Consenso()
```

```
        {
            //Arrange
            var administrador = new Administrador();

            //Act
            var processoDeAvaliacaoDeDesempenho =
administrador.CriarProcessoDeAvaliacaoDeDesempenho();

            //Assert
            Assert.AreEqual("Avaliação de consenso",
processoDeAvaliacaoDeDesempenho.SelecionarAvaliacaoPorTipo("Avaliaçã
o de consenso").Tipo);
        }
```

E o resultado de sua execução falha:

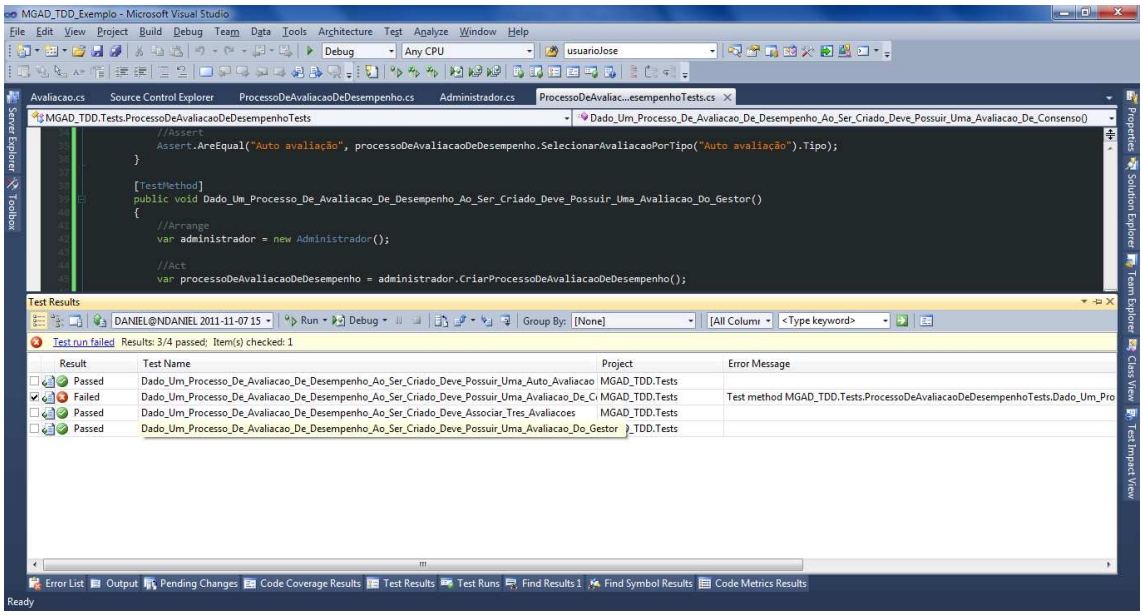

Figura 5: Quinta execução dos testes

Dessa forma, uma refatoração é necessária, que é a inclusão da criação de uma avaliação do tipo “AVALIAÇÃO DE CONSENSO” no processo de avaliação. Fizemos esta modificação, além de modificar a propriedade “QuantidadeDeAvaliacoes”, que a classe “ProcessoDeAvaliacaoDeDesempenho” possui para recuperar realmente a quantidade de avaliações associadas ao processo. Veja o resultado da classe abaixo:

```
    public class ProcessoDeAvaliacaoDeDesempenho
    {
        private List<Avaliacao> avaliacoes;

        public ProcessoDeAvaliacaoDeDesempenho()
        {
            avaliacoes = new List<Avaliacao>();

            avaliacoes.Add(Avaliacao.CriarAvaliacao("Auto
avaliação"));
            avaliacoes.Add(Avaliacao.CriarAvaliacao("Avaliação do
gestor"));
            avaliacoes.Add(Avaliacao.CriarAvaliacao("Avaliação de
consenso"));
        }

        public int QuantidadeDeAvaliacoes
        {
            get {   return avaliacoes.Count;  }
            private set { }
        }

        public Avaliacao SelecionarAvaliacaoPorTipo(string tipo)
        {
            return avaliacoes.Find(a => a.Tipo == tipo);
        }
    }
```

E a execução dos testes, todos, segue abaixo. Dessa forma, o requisito inicial foi implementado, com sucesso. Além disso, qualquer alteração no código que por ventura altere a funcionalidade dessas regras terá um feedback quase instantâneo, já que os testes que verificam essa funcionalidade irão falhar.

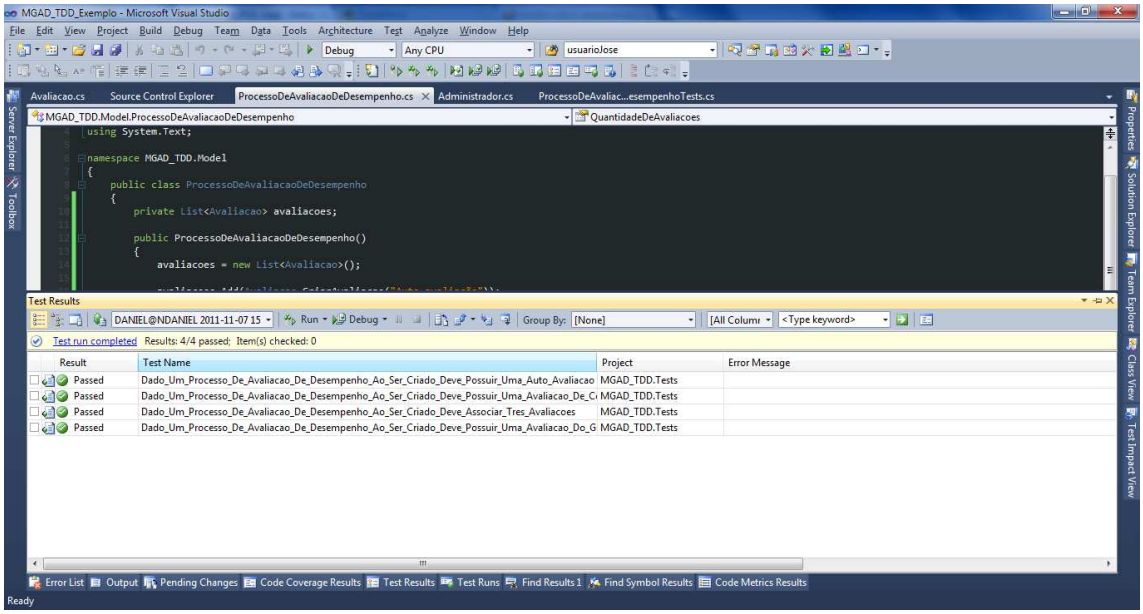

Figura 6: Sexta execução dos testes

6 RESULTADOS

De posse do projeto com a codificação feita através da técnica de TDD, o próximo passo é comparar o resultado das métricas que foram coletadas ao longo do desenvolvimento com o resultado das métricas coletadas na solução existente, codificada através de uma metodologia guiada por planos.

É importante frisar que as métricas foram coletadas no projeto existente apenas uma vez, visto que o acesso à solução só existiu após a codificação dos requisitos usando a técnica de TDD, para não influenciar na solução.

6.1. Complexidade Ciclomática

A extração desse indicador foi feita através de uma ferramenta de análise de códigos. Como é possível observar abaixo, vemos uma evolução crescente desse índice, de acordo com a codificação dos requisitos através de testes. Nota-se também um incremento considerável desse índice após a codificação da persistência de dados.

Com relação à medição deste índice no projeto já existente, é importante observar que esta é uma implementação completa de todos os requisitos existentes, levantados pela empresa de TI que usamos no trabalho. Dessa forma, o índice de complexidade ciclomática coletado aqui é bem superior que este mesmo índice coletado na solução codificada usando TDD. Sendo assim, mais abaixo usaremos como comparação um índice que é derivado da razão entre a complexidade ciclomática e o número de linhas de código.

A tabela 1 apresenta as comparações entre os índices :

Eventos de execução da medição	Complexidade ciclomática	
	Projeto usando TDD (medido apenas do código funcional)	**Projeto construído através de BDUF**
Início da implementação	19	-
Após refatoração de alguns requisitos	27	-
Usando código para recuperação de alguns objetos	40	-
Criação da camada de persistência	53	-
Refatoração de alguns requisitos	45	-
Implementação de requisitos da "Avaliação"	69	-
Implementação da funcionalidade "Processo de Avaliação de Desempenho"	75	-
Fim da codificação usando TDD	101	2262

Tabela 1: Dados de complexidade ciclomática

6.2. Acoplamento de classes

Este indicador demonstra o acoplamento entre as classes existentes no sistema. A solução que já existente mostra um forte acoplamento entre as classes. E isso é explicado, em partes, pela separação feita entre comportamento e dados, causando o "Modelo Anêmico", citado por Fowler(2003).

A solução já existente adotou a estratégia de separar as classes que transportam dados e classes que efetivamente implementam comportamentos. Isso trouxe um alto índice de acoplamento das classes na solução existente. Mesmo considerando um número maior de

classes na solução já existente, o índice é bem superior ao encontrado na solução codificada através de TDD.

A tabela 2 apresenta as medições deste índice segue abaixo:

Eventos de execução da medição	**Acoplamento de classes**	
	Projeto usando TDD (medido nas classes de modelo e persistência)	**Projeto construído através de BDUF (medido nas classes de modelo, negócio e persistência)**
Início da implementação	5/0	-
Após refatoração de alguns requisitos	20/17	-
Usando código para recuperação de alguns objetos	24/18	-
Criação da camada de persistência	22/18	-
Refatoração de alguns requisitos	22/18	-
Implementação de requisitos da "Avaliação"	30/20	-
Implementação da funcionalidade "Processo de Avaliação de Desempenho"	31/20	-
Fim da codificação usando TDD	36/20	59/58/31

Tabela 2: Dados do Acoplamento de Classes

6.3. Quantidade de Linhas de código

Neste indicador, é claro que temos uma disparidade entre os resultados obtidos. Conforme explicado anteriormente, a codificação do projeto usando a técnica TDD foi usada num universo restrito de requisitos, e dessa forma a comparação do resultado

deste indicador entre o projeto usando TDD e o projeto que já existia sofre interferências.

Vale compreender também que a estratégia adotada, mencionada por Fowler(2003) de Modelos Anêmicos (Anemic Models) pode gerar classes um pouco mais extensas.

De toda forma, este indicador será usado para calcular a razão entre a complexidade ciclomática e a quantidade de linhas de código.

A tabela 3 coleta deste índice:

Eventos de execução da medição	**Quantidade de linhas de código**	
	Projeto usando TDD (medido nas classes de modelo e persistência)	**Projeto construído através de BDUF (medido nas classes de modelo, negócio e persistência)**
Início da implementação	31/0	-
Após refatoração de alguns requisitos	59/46	-
Usando código para recuperação de alguns objetos	126/92	-
Criação da camada de persistência	106/92	-
Refatoração de alguns requisitos	105/90	-
Implementação de requisitos da "Avaliação"	202/178	-
Implementação da funcionalidade "Processo de Avaliação de Desempenho"	209/178	-
Fim da codificação usando TDD	289/270	275/310/160

Tabela 3: Dados de quantidade de linhas de código

6.4. Razão entre Complexidade Ciclomática e Quantidade de Linhas de Código

A intenção de exibir este índice é ilustrar uma idéia mais aproximada da complexidade ciclomática real. Como o projeto já existente contempla a implementação real e usada numa solução corporativa, obviamente este projeto possui mais classes e linhas de código que o projeto codificado para este experimento.

A tabela 4 apresenta as métricas “Complexidade Ciclomática” e “Quantidade de Linhas de Código” dos dois projetos:

Razão entre Complexidade Ciclomática e Quantidade de Linhas de Código	
Projeto feito usando TDD	**Projeto já existente, codificado através de BDUF**
101/(289+270) = 0,18	2262/(275+310+160) = 3,04

Tabela 4: Dados da razão entre Complexidade Ciclomática e Quantidade de Linhas de Código

6.5. Inspeção de código por um especialista

Para extrair esta métrica, foi necessário contar com um especialista que não esteve envolvido com o trabalho e nem com o domínio. De acordo com o especialista, houve uma dificuldade em avaliar um código sem estar totalmente envolvido no domínio e resolução do problema.

A avaliação foi feita apenas com relação ao design do código, tanto da solução codificada através de TDD quanto solução existente. O foco das observações era principalmente a qualidade do design e a aplicação dos princípios SOLID, citados por Martin(2000).

A tabela 5 apresenta algumas observações feitas, de forma tabular:

Comentários feitos por especialistas, avaliando design das soluções	
Projeto feito usando TDD	**Projeto já existente, codificado através de BDUF**
Algumas classes violam o prinípio SRP, pois realizam tarefas distintas e não apenas o que foi proposto pelo nome.	Projeto confuso e com muitas heranças em vários níveis.
Classes de negócio que estão lidando com persistência e recuperação de dados.	Regras de negócio espalhadas por toda a solução. Algumas regras de negócio inclusive estão contidas em SQLs, o que as torna impossíveis de serem testadas isoladamente.
Classes de persistência deveriam estar em um pacote próprio, no qual elas dependeriam das classes de negócio (e não o contrário)	Adoção de modelo anêmico, já que existem entidades que apenas contém dados e entidades que apenas executa comportamentos.
Violação do DIP pelos Models, já que elas criam objetos de acesso a dados e ficam dependentes delas.	Solução totalmente acoplada, onde dependências não estão isoladas. Isso torna impossível o reuso.
Alto acoplamento entre as classes de domínio e persistência prejudica a testabilidade, já que os testes exercitam códigos de ambas camadas	Quantidade de código e complexidade demasiadamente altas, visto que existem muitas camadas com regras de negócio espalhadas por todo o projeto.

Tabela 5: Dados das comparações feitas por um especialista

7. CONCLUSÕES

Este trabalho se propôs a comparar o impacto no design do projeto que foi codificado usando uma técnica de desenvolvimento ágil, guiada por testes (TDD) com um projeto já existente, codificado através de um modelo de desenvolvimento guiado por planos, mais tradicional no mercado de software atual.

A princípio, a intenção era verificar se o desenvolvimento guiado por testes (TDD) realmente provia melhorias na qualidade do design do software. Para definir se um design é melhor ou pior que outro design de software, calculou-se métricas que usualmente são usadas para definir qualidade de design de software, mencionadas e definidas anteriormente.

O experimento consistiu em obter alguns requisitos de um sistema de avaliação de desempenho, feito por uma empresa de desenvolvimento de sistemas real, codificar através da técnica de TDD e coletar algumas métricas, comparando com as mesmas métricas calculadas no projeto existente do sistema tomado para o experimento.

Ao comparar os valores extraídos das medições, notou-se que os valores não ilustravam uma mudança tão drástica de melhoria. Nota-se que a simples adoção de codificar teste unitário do requisito antes do código funcional, sendo esta uma descrição simples da técnica definida por Beck(2003) não basta para se obter melhorias significativas e claras com relação ao design de software.

De fato, usar a técnica TDD traz além do benefício de uma alta cobertura de código por testes, faz com que o desenvolvedor tenha fortes indícios de que o design que seu software é de fácil ou difícil manutenção. A dificuldade em escrever os outros testes indica que esse design desse software está complicado demais para uma expansão. Contudo, observa-se que mesmo no código feito usando TDD houveram críticas feitas na inspeção do código mencionando violações a princípios de design de software SOLID mencionados por Martin(2002).

O que se pode concluir é que a técnica de TDD pode ser uma ferramenta no auxílio do desenvolvimento de um software com um bom design, mas é necessário estar atento aos

paradigmas da orientação a objeto, aos princípios SOLID e boas práticas, definidas anteriormente neste trabalho.

Vale destacar também a rigidez exigida para a codificação dos ciclos, aplicando-se a técnica de TDD. O mantra do TDD definido por Beck(2003) é bem definido e leva o desenvolvedor a ser mais disciplinado com os procedimentos de codificação de um requisito. Tudo isso mencionado até aqui são benefícios da adoção da técnica de TDD, mas não garantem de fato uma melhoria real e significativa da qualidade do design de software, tema deste trabalho.

Para meus trabalhos futuros, pretendo abordar melhor a maneira como podemos usar os feedbacks obtidos através do TDD para fazer códigos melhores. Acredito que entender melhor o que é expresso pelos princípios SOLID seja essencial para uma codificação mais coesa e concisa e por este caminho que pretendo guiar meus trabalhos futuros.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott W. Agile Design. 2001-2008. Disponível em <http://www.agilemodeling.com/essays/agileDesign.htm>. Acessado em 08-nov-2010

ANICHE, Maurício, GEROSA, Marco. Um Estudo sobre o Impacto de Test-Driven Development na Qualidade do Design de Software. In: XV Workshop de Teses e Dissertações em Engenharia de Software. Salvador, Brasil. 2010.

BASILI, Victor R. CALDIERA, Gianluigi. ROMBACH, H. Dieter. The Goal Question Metric Approach. Disponível em <ftp://ftp.cs.umd.edu/pub/sel/papers/gqm.pdf>. Acesso em 04 out 2010.

BECK, Kent. Test Driven Development: By Example. Addison-Wesley Professional, 2003.

BOEHM, Barry. Software Metrics Guide. 2000. Disponível em <http://sunset.usc.edu/classes/cs577b_2001/metricsguide/metrics.html>. Acessado em 04 out 2010.

GERMOGLIO, Guilherme. Disponível em <http://cnx.org/content/m17494/latest>. Acesso em 02 out 2010.

GRADY, Robert. Practical Software Metrics for Project Management and Process Improvement. Prentice-Hall, 1992.

JETHER. Disponível em <http://www.tinews.net.br/2010/08/12/rup-disciplines-analise-design >. Acesso em 02 out 2010.

KATKI, Freny at al, editors. IEEE Standard Computer Dictionary, 1991.

MANUAL DO RUP. Disponível em <http://www.wthreex.com/rup/portugues/index.htm>. Acesso em 02 out 2010.

MARTIN, Robert. Agile Software Development, Principles, Patterns , and Practives. Prentice Hall, 2002.

MARCHENKO, Artem. Agile Software Development. 2008. Disponível em <http://agilesoftwaredevelopment.com/xp>. Acesso em 26-out-2010.

PRESSMAN, Roger S. Engenharia de Software. Editora McGrawn-Hill, 2006.

WELLS, Don. Extreme Programming. Disponível em <http://www.extremeprogramming.org>. Acesso em 04 out 2010.

WIKIPEDIA. Disponível em <<http://pt.wikipedia.org/wiki/Test_Driven_Development >>. Acesso em 02 out 2010.

WIKIPEDIA. Disponível em <http://en.wikipedia.org/wiki/Extreme_Programming>. Acesso em 04 out 2010.