MANUTENIBILIDADE EM PROJETOS DE SOFTWARE: UMA ABORDAGEM ÁGIL UTILIZANDO BEHAVIOUR-DRIVEN DEVELOPMENT

Renato Yoshio Murata[1], Marcelo Nogueira[2]

Abstract *— Maintenance can consume more than 60% of the effort spent by a company in a software project, thus being recognized as an important factor by several quality models. However, since it's not directly perceived by users, it often does not receive the same attention devoted to other quality requirements. The consequence becomes clear only after months of work, when the software goes into production and is not able to accommodate necessary modifications to adapt to changes in the environment where it operates. The software begins to lose its value. This paper describes how the use of Behaviour-Driven Development, an agile method, can assist the development of maintainable software, relating its practices, processes and work products with each of the maintainability subcharacteristics defined by the ISO 9126 quality model.*

Index Terms — Agile method, Maintainability, Software Engineering, Test-Driven Development.

INTRODUÇÃO

Os computadores têm sido usados numa variedade de áreas de aplicação cada vez maior e sua correta operação é frequentemente crítica para o sucesso de negócios e para a segurança humana. Deste modo, desenvolver ou selecionar produtos de software de alta qualidade é de primordial importância [1].

Entretanto, a qualidade de software não é um conceito simples. Diferentemente da manufatura, ela apresenta dificuldades inerentes à própria natureza do software. Por ser algo intangível, frequentemente os usuários têm dificuldade em expressar suas expectativas em relação ao que necessita ser desenvolvido. Algumas características, muitas vezes, são difíceis de serem descritas de maneira clara, mas acabam exercendo grande influência na qualidade percebida do sistema.

Este é o caso da manutenibilidade. Ela integra os modelos de qualidade de McCall, Boehm, e da ISO 9126, sendo assim reconhecida como um importante fator de qualidade. Entretanto, por não ser uma característica diretamente percebida pelo usuário, frequentemente não recebe a mesma atenção dedicada aos demais requisitos. O resultado são sistemas que, mesmo apresentando uma qualidade percebida aceitável, internamente são extremamente frágeis a mudanças, ou seja, são de difícil manutenção.

Sistemas assim tendem a atingir com maior rapidez a fase de declínio, na qual se torna praticamente impossível adicionar novas funcionalidades a um custo aceitável, pois os efeitos colaterais não são mais gerenciáveis.

Eventualmente, o projeto para por completo. Se não é possível realizar alterações no software para adequá-lo às necessidades do negócio, ele perde sua posição estratégica.

MANUTENÇÃO DE SOFTWARE

Há mais de três décadas, a manutenção de software foi caracterizada como um iceberg. Espera-se que o imediatamente visível seja tudo o que existe, mas sabe-se que uma enorme massa de possíveis problemas e custo fica sob a superfície [2]

A manutenção de software existente pode ser responsável por mais de 60% de todo o esforço despendido por uma organização de desenvolvimento [3]. Ela envolve não só "consertar erros", mas também adaptar o sistema a mudanças no ambiente externo, fazer melhorias solicitadas pelos usuários e realizar processos de reengenharia.

Portanto, a manutenibilidade de um software está atrelada a sua capacidade de aceitar mudanças, e mudanças são caras.

Na economia moderna, é frequentemente difícil ou impossível prever como um sistema baseado em computador irá evoluir ao passar do tempo. As condições do mercado mudam rapidamente, e novas ameaças competitivas surgem sem aviso. Em muitas situações, não é possível definir os requisitos completamente antes de o projeto iniciar. É preciso ser ágil o suficiente para responder a um ambiente de negócios fluido. Fluidez implica aceitar mudanças, e esta uma das características mais importantes da abordagem ágil: a sua habilidade de reduzir custos de mudanças através do processo de software [4].

MÉTODOS ÁGEIS

Os métodos ágeis tratam as mudanças como algo intrínseco ao processo de desenvolvimento. Elas não são exceções, mas sim fazem parte da própria natureza do software.

1 Renato Yoshio Murata, Pós-Graduando em Engenharia de Software, UNIP, Rua Antônio de Macedo, 505, 03.087-040, Parque São Jorge - São Paulo - SP, Brazil, renatoymurata@gmail.com
2 Marcelo Nogueira, Doutor em Engenharia de Produção, UNIP, Rua Antônio de Macedo, 505, 03.087-040, Parque São Jorge - São Paulo - SP, Brazil, marcelo@noginfo.com.br

Trabalhando com um ciclo de entregas frequentes, os processos ágeis tornam uma obrigação a preocupação com a manutenibilidade. Como uma versão inicial é colocada em produção, geralmente, após poucas semanas do inicio do projeto, o sistema entra na fase de manutenção rapidamente. Problemas com requisitos de manutenibilidade são identificados cedo, evitando assim que suas consequências só venham a ser descobertas meses após entrar em produção, quando então já é tarde para intervir na qualidade do software.

Histórico e o Manifesto Ágil

Apesar de alguns métodos ágeis terem existido de uma forma ou outra durante uma ou duas décadas, o termo "Método ágil" só veio a ser cunhado mais recentemente, em Fevereiro de 2001, por dezessete dos desenvolvedores lideres e proponentes das então denominadas metodologias "leves". Estas pessoas reuniram-se com o objetivo de encontrar pontos em comum entre as várias metodologias leves [5]. O resultado foi a elaboração de um documento, o Manifesto Ágil, composto por valores e princípios sobre os quais todos os métodos ágeis são construídos

Desde então, estes métodos têm se difundido cada vez mais nos círculos de desenvolvimento de software. O Extreme Programming (XP) e o Scrum são exemplos de métodos ágeis amplamente adotados atualmente.

Apesar de cada um propor diferentes técnicas e processos para atingir seus objetivos, todos compartilham os mesmos valores e princípios, tendo assim, muito em comum. Frequentemente, práticas de uma metodologia são adotadas por outras, ou ainda novas metodologias são criadas com base em anteriores que demonstraram boa aceitação pela comunidade de desenvolvimento.

Este é o caso do Behaviour-Driven Development, um dos métodos ágeis mais recentes, que evoluiu a partir de uma prática de sucesso do XP, na qual testes são escritos antes da codificação, o Test-Driven Development.

A seção seguinte apresenta os conceitos e técnicas aplicadas no Test-Driven Development, assim como as dificuldades comumente enfrentadas por desenvolvedores que levaram ao surgimento do Behaviour-Driven Development.

TEST-DRIVEN DEVELOPMENT

Conceituando, o Test-Driven Development (TDD) é uma forma de programar que encoraja bom design, e um processo disciplinado que auxilia a evitar erros de programação. Isso é obtido escrevendo pequenos testes automatizados, que eventualmente acabam formando um sistema de alarme eficaz para proteger o código contra regressão [6].

O ciclo do TDD pode ser definido em três passos [7]: escrever um teste; fazer o teste passar; e melhorar o código.

Escrever um teste é imaginar como a operação a ser implementada será representada no código. É o ato de escrever uma história. Inventar uma interface ideal para ser utilizada. São incluídos todos os elementos da história necessários para chegar aos resultados corretos [7].

O passo seguinte, fazer o teste passar, deve ser rápido. O que se busca neste momento é a solução mais simples possível capaz de passar nos testes, não só naquele que acabou de ser criado como também em todos os outros que já foram escritos. Otimizações, ou a melhor solução, mas que leve mais de alguns minutos devem ser postergadas.

Uma vez que o teste tenha passado, o código é então analisado para uma possível necessidade de *refactoring*. Busca-se alterar o código, melhorando o design da aplicação, de forma que todos os testes continuem passando.

Refactoring

Refactoring é o processo de alterar um sistema de software sem que o comportamento externo do código sofra alterações, melhorando sua estrutura interna. Em essência, melhorar o design do código depois que ele foi escrito [8].

Além de não introduzir defeitos com as mudanças na estrutura interna do sistema, o *refactoring* também não deve adicionar novas funcionalidades. Ou seja, o *refactoring* deve preservar o comportamento do sistema [6].

Testes de regressão

O teste de regressão é a atividade que ajuda a garantir que modificações não introduzam comportamento indesejável ou erros adicionais [4].

Aplicando o TDD, cada ciclo de teste-codificação-*refactoring* adiciona um novo item na suíte de testes automatizados que são executados sempre que uma alteração é realizada no sistema. Estes testes formam uma rede de segurança, que possibilita realizar o *refactoring* com mais confiança, identificando rapidamente erros introduzidos, e facilitando a integração de novas funcionalidades no código base.

Design emergente

O design do software, ao invés de ser feito todo no início, ocorre de forma contínua durante o desenvolvimento. Através da construção do sistema aprende-se como melhorar o design. A interação resultante leva a um programa cuja qualidade do design é mantida conforme o desenvolvimento progride [8].

Este design emergente é um dos subprodutos mais significativos do Test-Driven Development [9].

Dificuldades em aplicar TDD

Apesar do ciclo de teste-codificação-*refactoring* proposto pelo TDD ser aparentemente simples, na prática muitos encontram dificuldades em aplicá-lo.

Astels [10] defende que o principal motivo das dúvidas e dificuldades enfrentadas é a visão errônea de que o foco do TDD está em testar. Apesar de testes serem executados com frequência e uma suíte de testes de regressão resultar do

processo de desenvolvimento, estes são apenas efeitos colaterais positivos.

Esta visão focada em testes pode ser explicada pela origem do próprio TDD.

Originalmente, o XP defendia a codificação de testes para tudo que pudesse falhar, escrevendo testes unitários para os desenvolvedores e testes funcionais do ponto de vista do cliente [11]. Posteriormente, a prática de teste do XP evoluiu para o TDD, e nesta mudança, apesar das ferramentas e muitos termos utilizados ainda serem os mesmos, o foco deixou de estar nos testes e passou a estar no comportamento esperado do software.

Ao invés de escrever testes para cada método de uma classe, refletindo sua divisão estrutural, passou-se a escrever especificações sobre o que se espera do código, ou seja, a especificar o comportamento do código antes de escrevê-lo.

Percebe-se então que o foco real do TDD está em especificar comportamento e não em escrever testes, como o seu próprio nome erroneamente leva a crer.

Esta confusão de termos, e as dúvidas resultantes dela levaram Dan North a cunhar o termo Behaviour-Driven Development para descrever uma abordagem do TDD voltada ao comportamento do software, distinguindo-a do TDD tradicional, muitas vezes mal aplicado.

BEHAVIOUR-DRIVEN DEVELOPMENT

O foco do Behaviour-Driven Development (BDD) está em implementar uma aplicação descrevendo seu comportamento a partir da perspectiva dos seus stakeholder [9].

Isso implica a adoção do ponto de vista dos stakeholders do projeto para que se possa entregar um software que realmente atenda às suas necessidades. É preciso compreender o domínio, as oportunidades e desafios dos stakeholders, assim como a linguagem que eles utilizam para descrever o que se espera da aplicação a ser desenvolvida.

Além disso, fica evidente a existência de mais de um stakeholder, que no BDD é qualquer pessoa com um interesse no projeto. Podem ser as pessoas cujo problema está se tentando resolver – denominados *core stakeholders* – ou aquelas que irão auxiliar a resolve-lo – os *incidental stakeholders*.

Concepção do projeto

Antes de iniciar o ciclo de desenvolvimento e entrega de um projeto, é necessário compreender qual o seu propósito, o que se está tentando alcançar. Para isso realiza-se uma reunião envolvendo todos os stakeholders com o objetivo de definir a visão do projeto, ou seja, qual o seu propósito.

O passo seguinte é identificar, juntamente com os *core stakeholders*, os resultados esperados do projeto. Estes devem estar alinhados com a visão previamente estabelecida, e possibilitarão saber quando o projeto atingiu seu propósito.

Estes resultados, por sua vez, nortearão o desenvolvimento da aplicação que possibilitará atingi-los. Para isso, passa-se a descrever em nível de software o que será necessário desenvolver para cada resultado esperado. Isso é feito utilizando temas, ou *features sets*, que representarão cada um deles.

Com os temas identificados, não necessariamente todos, pode-se finalmente dividi-los em partes sobre as quais o trabalho de desenvolvimento do dia-a-dia será realizado, o das *features*. De modo simples, uma feature é algo que entrega valor coeso para um stakeholder. Cada tema é composto por uma ou mais *features* e estas descrevem o comportamento que será implementado no software.

Ciclo de entregas

O ciclo de entregas do BDD tem inicio com um stakeholder discutindo um requisito com o analista de negócio. O requisito pode ser um problema a ser resolvido ou uma ideia a ser explorada. O analista auxilia o stakeholder a articular o requisito em termos de *features* e talvez indo além, descrevendo o requisito em pequenas partes verificáveis denominadas histórias, que representam não mais que alguns dias de trabalho [9].

A seguir o stakeholder e o analista de negócio trabalham com um *tester* para determinar o escopo das histórias. Enquanto o analista de negócio pensa em termos abstratos, o *tester* geralmente pensa em cenários concretos [9]. Para cada cenário é estabelecido um critério de aceitação, na forma de um teste, que define explicitamente o que "completo" significa para cada um deles. Enquanto estiver falhando, um teste de aceitação demonstra que o sistema ainda não implementa a funcionalidade esperada.

Quando todos os critérios de aceitação de uma história forem satisfeitos, significa que a história está completa. Do mesmo modo, quando todas as histórias de uma *feature* forem implementadas a *feature* estará completa, e assim sucessivamente, até que a visão seja atingida.

Busca-se desta maneira fazer apenas o necessário para atender a necessidade original do stakeholder e, consequentemente, evitar desperdício de esforço.

BDD E MANUTENIBILIDADE SEGUNDO A ISO 9126

A norma ISO 9126 - "Engenharia de software - Qualidade do produto" foi desenvolvida em uma tentativa de identificar os fatores chave de qualidade para produto de software.

Entre as características tratadas pela ISO 9126 está a manutenibilidade, definida como: "A capacidade do produto de software de ser modificado através de correções, melhorias ou adaptações devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais".

A manutenibilidade, por sua vez, é dividida em quatro subcaracteristicas principais: analisabilidade, modificabilidade, testabilidade e estabilidade.

Os tópicos seguintes descrevem como as práticas, processos e produtos de trabalho do BDD abordam cada uma dessas subcaracteristicas.

Analisabilidade

A analisabilidade de um software pode ser examinada em três níveis: legibilidade, compreensibilidade e rastreabilidade [12].

A legibilidade está relacionada com a consistência de estilo. Isto inclui formatação de expressões, indentação, nomenclatura, comentários, etc.

O BDD utiliza um formato padrão para descrição de histórias composto por um título, uma narrativa e os critérios de aceitação. Apesar de existirem variações, o formato mais comum para a narrativa é o Connextra: *como um [stakeholder], eu quero [feature], para que [beneficio]*.

Assim como as histórias, cada cenário, que representa os critérios de aceitação, também possui uma estrutura própria, geralmente expressa na forma: *dado que [contexto], quando [evento], então [resultado]*.

Esta padronização não apenas facilita a leitura do código como também torna simples o processo de automatização dos testes.

A compreensibilidade, um nível acima da legibilidade, trata de blocos de código maiores e como estes são organizados. Possibilita que não só os desenvolvedores, mas também os demais stakeholders sejam capazes de entender a estrutura geral do sistema.

No BDD, todo o código é gerenciado com base nas *features*, que funcionam como contêineres de histórias e cenários. Isso deixa claro o relacionamento de cada código implementado com sua respectiva *feature*.

Além disso, o uso de uma linguagem única entre todos os membros do projeto, denominada linguagem ubíqua (técnica originalmente proposta pelo Domain-Driven Design), possibilita que até mesmo os *core stakeholders* compreendam as *features*, histórias e cenários do sistema, descritos utilizando esta linguagem.

Por ultimo, a rastreabilidade é obtida pela própria estrutura dos ciclos de desenvolvimento do BDD: todo código só existe como parte de um cenário; cada cenário pertence a uma história, e esta a uma *feature*; cada *feature* só existe porque adiciona valor a um tema; cada tema contribui para um ou mais resultados, e cada resultado é parte do propósito geral do projeto.

Modificabilidade e testabilidade

As subcaracteristicas de modificabilidade e testabilidade estão fortemente relacionadas aos conceitos de coesão e acoplamento.

Coesão é uma extensão natural do conceito de encapsulamento. Um módulo coeso realiza uma única tarefa, necessitando de pouca interação com outros componentes de outras partes de um programa. Classes e componentes que exibem coesão são relativamente fáceis de implementar, testar, e manter [4] .

Acoplamento é um indicador de interconexão entre os módulos em uma estrutura de software. O acoplamento depende da complexidade das interfaces entre os módulos, do local onde a referencia é feita a um módulo, e de que dados passam através das interfaces. Em design de software, deve-se buscar o menor acoplamento [4].

Portanto, alta coesão e baixo acoplamento são características importantes e que devem ser buscadas no design de um software. Sistemas com baixa coesão e alto acoplamento tendem a ser mais difíceis de compreender e, consequentemente, modificar, pois seus módulos realizam mais tarefas do que deveriam, tornando-se complexos, e funções que deveriam estar centralizadas em um único local encontram-se espalhadas em diversas partes do código. Como resultado, um erro introduzido em um ponto, muitas vezes acaba se propagando por todo o sistema.

Do mesmo modo, a testabilidade de um software também é afetada. Um alto acoplamento significa que não é possível isolar partes especificas do software para serem testadas. Ao tentar definir uma unidade de teste, as dependências entre os módulos muitas vezes revelam que a única opção é testar o sistema como um todo.

O BDD força o desenvolvimento de módulos fracamente acoplados devido a sua abordagem de especificação de comportamentos, representados por testes, em múltiplos níveis. Em cada nível uma unidade de teste de diferente granularidade é abordada.

Como os testes são escritos antes do código, o foco na testabilidade da aplicação faz parte do próprio processo de desenvolvimento. Dificuldades em escrever testes com alta granularidade, como exige o BDD, evidenciam problemas com o design do software. Por possibilitar a rápida identificação desses problemas, o BDD evita que estes se mantenham ocultos no código e venham a prejudicar a manutenibilidade do sistema mais tarde.

A coesão é obtida através do *refactoring*, que pode ser visto como uma revisão. Esta tarefa engloba não só o código que acabou de ser escrito, mas também o design geral do software, que evolui de forma incremental. Duplicações de código e responsabilidade são identificadas e eliminadas, buscando melhorar a estrutura interna do código. Assim, o *refactoring* pode resultar em alterações que envolvem diversas partes do sistema.

Estabilidade

Apesar do software não sofrer desgaste com o passar do tempo, como ocorre com hardware, ele se deteriora. Durante sua vida, o software sofrerá mudanças, e conforme estas são feitas, é provável que erros sejam introduzidos, resultando em “picos” na sua curva de falhas. Antes de a curva poder retornar ao seu estado original, outra mudança é solicitada, fazendo com que ela suba novamente. Aos poucos, o nível da taxa mínima de falhas começa a aumentar – o software está se deteriorando devido às mudanças [4].

Estudos empíricos [4] demonstram que quanto mais tarde um erro no software é identificado, mais cara é a sua

correção. Através de atividades de verificação e validação frequentes, o BDD busca aproximar a curva real da curva ideal de falhas identificando e corrigindo estes erros o mais cedo possível para que eles não venham a se tornar falhas posteriormente.

Inicialmente o BDD procura minimizar o número de erros introduzidos durante a implementação do software forçando a codificar cada funcionalidade duas vezes. A primeira codificação é a tradução para código, na forma de um teste, do comportamento esperado do software. A segunda é a própria implementação do código que fará este comportamento funcionar.

Como descrito anteriormente, a revisão do código é realizada através do *refactoring*, identificando erros e melhorando a qualidade interna do software.

Um efeito colateral positivo de cada ciclo do BDD é a produção de um teste automatizado que passa a ser executado a cada alteração posterior no software, formando um conjunto de testes de regressão. Quando são executados manualmente, muitas vezes estes testes tornam-se inviáveis, porém o BDD dilui o esforço necessário para sua execução automatizando-os durante todo o processo de desenvolvimento.

A execução frequente dos testes de regressão, geralmente a cada *commit* no repositório, impede que um erro se mantenha no sistema por muito tempo, e, consequentemente, possa resultar em uma falha posterior.

CONSIDERAÇÕES FINAIS

Apesar da manutenção de software ser parte integral do ciclo de vida de um software, historicamente, ela não tem recebido o mesmo nível de atenção que as demais fases de desenvolvimento. Entretanto, conforme as empresas lutam para obter o maior retorno possível sobre os investimentos realizados em software, este quadro está mudando.

Os métodos ágeis oferecem uma alternativa às metodologias tradicionais, incorporando o conceito de manutenibilidade no próprio processo de desenvolvimento.

O Behaviour-Driven Development, além de possuir características comuns a outros métodos ágeis como foco no cliente, entregas frequentes, simplicidade e rápida adaptação a mudanças, também inclui processos, práticas e produtos de trabalho particulares que auxiliam o desenvolvimento de softwares manuteníveis e que atendam às reais necessidades dos usuários.

A analisabilidade do software é abordada padronizando o formato de escrita de histórias e cenários; utilizando a linguagem ubíqua; e possibilitando a rastreabilidade de todo código que compõe o sistema.

As subcaracteristicas de testabilidade e modificabilidade, fortemente relacionadas à coesão e acoplamento de módulos, são tratadas aplicando ciclos TDD em diferentes níveis, forçando o desacoplamento das partes do sistema, e realizando *refactoring* em cada ciclo para manter uma alta coesão.

Por ultimo, a estabilidade é mantida realizando revisões e validações frequentes. O conjunto de testes de regressão automatizados protege o sistema contra introdução de erros acidentais.

Finalmente, como qualquer outra metodologia de desenvolvimento, o BDD não se propõe a ser uma bala de prata. Características da empresa, da equipe de desenvolvimento, do cliente, e do projeto em questão devem ser consideradas antes de se decidir por adotá-lo. Porém, como descrito neste artigo, muitas de suas práticas, ainda que aplicadas isoladamente ou em conjunto com outros métodos, ágeis ou não, tendem a incentivar o desenvolvimento de sistemas com design flexível, capazes de acomodar mudanças, podendo o seu uso, mesmo em projetos experimentais, agregar importante conhecimento tanto à equipe de desenvolvimento quanto à própria empresa.

REFERÊNCIAS

[1] Associação Brasileira de Normas Técnicas, "NBR ISO/IEC 9126-1 Engenharia de software - Qualidade de Produto", 2003.

[2] Canning, Richard, "The Maintenance 'Iceberg'", EDP Analyzer, vol. 10, no. 10, Out. 1972.

[3] Hanna, Mary, "Maintenance Burden Begging for a Remedy", Datamation, Abr. 1993.

[4] Pressman, Roger S., Engenharia de Software, 6th ed. São Paulo: McGraw-Hill, 2010.

[5] Koch, Alan S., Agile Software Development: Evaluating the Methods for Your Organization. Norwood: Artech House, 2005.

[6] Koskela, Lasse, Test Driven: TDD and Acceptance TDD for Java Developers, Manning Publications, 2007.

[7] Beck, Kent, Test-Driven Development by Example, Addison Wesley, 2002.

[8] Fowler, Martin , Refactoring: Improving the Design of Existing Code.: Addison-Wesley Professional, 1999

[9] Chelimsky, David; Dennis, Zach; Hellesøy, Aslak; Helmkamp, Bryan; and North, Dan, The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends.: Pragmatic Bookshelf, 2010.

[10] Astels, Dave, "A new look at Test-Driven Development", 2006. Disponível em: <http://blog.daveastels.com/files/BDD_Intro.pdf>. Acesso em: 03 Maio 2011.

[11] Beck, Kent, Extreme Programming Explained: Embrace Change.: Addison-Wesley Professional, 1999.

[12] Spinellis, Diomidis, Code Quality: The Open Source Perspective.: Addison Wesley Professional, 2006.