Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Compondo Métodos Ágeis de Desenvolvimento de Software

Tiago de Farias Silva

TRABALHO DE GRADUAÇÃO

Recife
01 de Julho de 2009

Universidade Federal de Pernambuco
Centro de Informática

Tiago de Farias Silva

Compondo Métodos Ágeis de Desenvolvimento de Software

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Hermano Perrelli de Moura

Recife
01 de Julho de 2009

“Quem refreia suas próprias declarações é possuído de conhecimento e o homem de discernimento é de espírito frio.” - Provérbios 17:27

RESUMO

Com o surgimento do Manifesto Ágil em 2001, o processo de gerenciamento de projetos de software sofreu intensas mutações. Com princípios focados em resultados rápidos e entregas constantes, diversas metodologias se consolidaram. Essas metodologias, utilizadas amplamente no indústria mundial de softwares, mostram-se muito úteis em projetos que apresentam mudanças bruscas e ambientes pouco previsíveis. Dessas metodologias, as mais conhecidas atualmente são: Scrum, XP e Feature Driven Development. Por sua vez, essas metodologias podem trazer consigo catalisadores, ou seja, técnicas orientadas ao desenvolvimento de Software propriamente dito, que aumentam o desempenho das metodologias ágeis e provêem mais qualidade ao produto final. Esse trabalho tem por objetivo mostrar como essas técnicas de desenvolvimento de Software e metodologias ágeis são mais poderosas combinadas do que individualmente.

Palavras-chave: Agilidade, Scrum, XP, Feature Driven Development, Domain Driven Design, Test Driven Development, Técnicas Ágeis, Gerenciamento de Projetos, Desenvolvimento de Software, Composição de Métodos Ágeis.

ABSTRACT

With the emergence of the Agile Manifesto in 2001, the process of project-management software has undergone intense mutations. With new principles, this time focused on quick results and constant deliverables, several methodologies have arisen. These methods, widely used in the world market for software, are very useful in projects that have sudden changes and hardly predictable environments. These methodologies, the best known today are: Scrum, XP and Feature Driven Development. In turn, these methodologies can bring their own catalysts, that is, oriented techniques to software development itself, which increase the performance of agile methodologies and provide more quality to the final product. This work has as its prime objective to show how the techniques of software development and agile methodologies are more powerful together than individually.

Keywords: Agile, Scrum, XP, Feature Driven Development, Domain Driven Design, Test Driven Development, Agile Techniques, Project Management, Software Development, Composition of Agile Practices.

AGRADECIMENTOS

Relutei um pouco pra fazer essa parte porque eu sabia desde o começo que é muita gente que eu tenho de agradecer. Desde colegas do curso, do colégio, do trabalho, do ex-trabalho, são muitas pessoas e é possível que eu não consiga citar todos por nome. De qualquer forma, esses que não foram citados diretamente saibam que toda vez que eu vir vocês, sempre lembrarei o apoio (por menor que tenha sido) dado.

Primeiramente agradeço a meus pais pelo apoio incondicional em meus empreendimentos, tanto em sentido emocional, quanto financeiro, jamais será esquecido por mim (talvez quando eu estiver com Alzheimer daqui há 83 anos).

Agradeço profundamente ao meu orientador, Professor Hermano Perrelli, que foi muito paciente e extremamente prestativo comigo durante a elaboração deste trabalho. Apesar de muito atarefado, mostrou-se pronto pra ajudar a qualquer momento com seu conhecimento da área e raciocínio.

Agradeço aos meus colegas de curso (por ordem alfabética): Apebão, Arara, Alysson, Borba, Braga, Caio, Felipe Fifo, Jera, João Doido, João Paulo, Leila, Mário, Rafael Formiga, Reaf, Riffa e Sosô.

Ao pessoal da Educandus, primeiramente ao Professor Ricardo Lessa pela grande oportunidade concedida a mim desde o começo do curso, assim como também por ser uma fonte de inspiração pela grande pessoa que é. Mas também a outros que passaram por lá e me ensinaram muitas lições: Mestre Tairone, Tia Laís, Bruna, Léo, João Augusto, Ricardo Teixeira, Teoria, enfim, muito obrigado a todos pelo aprendizado do dia a dia!

A Provider Sistemas e os caras de lá, pois foi nesse ambiente de trabalho que pude ver na prática os benefícios de se usar métodos ágeis. Marcos Pereira e Igor Macaúbas eram os pioneiros nisso, mas agradeço a todos por esclarecer coisas sobre o assunto e a abrir minha mente para novas tecnologias.

Obviamente não podia esquecer da galerinha do CPI. Foi onde "a instiga" começou. A todo o grupo de estudiosos: Bruno BB, Jefferson Fefa, Humberto Panzá, Paulo Bolado, Renata, Ana Rita, Zoby e todos os demais que de alguma forma me inspiraram a conseguir essa conquista.

Enfim, muito obrigado a todos!

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

1.1 Contexto do Trabalho

Com o crescente aumento da competitividade dos mercados internacionais, a exigência sobre o produto tornou-se um diferencial determinante. E visto que o software tornou-se essencial para a manutenção do controle e da produtividade nos mais diversos setores organizacionais, já era previsível o surgimento de novas metodologias e técnicas que buscassem garantir agilidade e ao mesmo tempo robustez no processo de desenvolvimento de software.

Assim surgiu o Manifesto Ágil, com o objetivo de promover práticas ágeis de desenvolvimento de software e gerenciamento de projetos. Essas práticas têm como conceito base o incentivo ao trabalho em equipe, equipes auto-organizadas e responsáveis, um conjunto de técnicas de engenharia que permitem rápidas entregas de produtos com alto nível de qualidade, além de uma nova abordagem ao alinhar os interesses dos clientes com os objetivos dos desenvolvedores.

Como os conceitos advindos do Manifesto Ágil mostraram-se de intenso valor para as boas práticas organizacionais, uma grande quantidade de engenheiros de software e gerentes de projeto começaram a adaptar tais conceitos para o seu próprio contexto. Dessa forma surgiram novas metodologias e diferentes técnicas focadas em dinamizar o processo de desenvolvimento. Entretanto, essas técnicas e metodologias podem se mostrar também deficientes em alguns aspectos. Porém, ao analisar as fraquezas e os pontos fortes de cada uma delas, é possível encontrar os pontos onde estas se complementam.

1.2 Motivação

Com o crescimento do mercado mundial de Software, novas empresas com novas estratégias de marketing e idéias inovadoras têm surgido. Nos últimos 8 anos, por exemplo,

muito têm-se falado sobre Agilidade. Não é de admirar, pois a comunidade ágil tem-se mostrado bastante eficaz, trazendo maneiras novas de resolver problemas antigos na área de Gerenciamento de Projetos e Desenvolvimento de Software.

Dentro dessa comunidade, existe um verdadeiro ecossistema composto por metodologias, técnicas e abordagens que, se executadas em conjunto de forma equilibrada, são extremamente robustas do ponto de vista organizacional, garantindo qualidade, objetividade e agilidade ao projeto.

É importante definir o conceito de metodologia, técnica, abordagem e método no contexto desse trabalho:

- Uma metodologia é um conjunto estruturado de práticas (por exemplo: Material de Treinamento, Programas de educação formais, Planilhas, e Diagramas) que pode ser repetível durante o processo de produção de software. Na verdade, metodologias de Engenharia de Software abrangem muitas disciplinas, incluindo Gerenciamento de Projetos, e as suas fases como: análise, projeto, codificação, teste, e mesmo a Garantia da Qualidade.
- Uma técnica é uma prática de desenvolvimento de software aplicada a uma determinada fase. Por exemplo, uma técnica pode existir tanto na fase de codificação quanto na fase de testes.
- Uma abordagem é uma forma de visualizar o projeto de software em questão. Com uma série de boas práticas e princípios, propõe uma forma diferente de encarar o projeto numa determinada fase. Não é completamente especificada com regras. Mas, deixa o desenvolvedor do projeto “livre” para aplicar regras que se adéquem aos conceitos propostos pela abordagem.
- Método, no contexto desse trabalho, é um termo geral utilizado para se referir a metodologias, técnicas e abordagens.

1.3 Objetivos

O objetivo deste Trabalho de Graduação é mostrar como as metodologias, abordagens e técnicas de desenvolvimento de software existentes atualmente na comunidade ágil podem ser mais bem aproveitadas do ponto de vista organizacional se aplicadas em conjunto, ao invés de individualmente. Tornando-se assim mais robustas e eficientes sem perder o foco na agilidade.

1.4 Estrutura do Trabalho

No Capítulo 2 temos a definição do que são Metodologias Ágeis. Será abordado como tudo começou com o surgimento do Manifesto Ágil em 2001 e o que este propõe, ou seja, quais seus princípios básicos. Neste mesmo capítulo serão apresentadas as metodologias mais conhecidas de toda comunidade ágil: Scrum, Extreme Programming (também conhecida pela sua abreviação XP) e Feature Driven Development. Ao final deste capítulo será realizada uma análise comparativa sobre estas metodologias.

O Capítulo 3 descreve as técnicas de engenharia utilizadas como catalisadores para os processos ágeis. São apresentadas as suas respectivas definições, funcionamento e aplicações no contexto de análise e desenvolvimento de software. É apresentada a técnica Test Driven Development (TDD) e a abordagem Domain Driven Design (DDD), assim como outras técnicas de programação como Design Patterns e ferramentas de controle de versão.

O Capítulo 4 é o "coração" deste Trabalho de Graduação, analisando como as metodologias ágeis podem ser aliadas as técnicas ou práticas de engenharia de modo a conceber agilidade e qualidade em todas as camadas de um projeto de software. Primeiramente, analisamos a composição das metodologias Scrum, XP e TDD. Depois, a metodologia FDD, a abordagem DDD e a técnica de Design Patterns.

O Capítulo 5 apresenta as conclusões do trabalho, trabalhos relacionados e propostas futuras.

2. METODOLOGIAS ÁGEIS

A filosofia aceita e pregada para o desenvolvimento de sistemas é de que o processo de desenvolvimento é perfeitamente previsível, podendo ser planejado, estimado e completado com sucesso [3]. No entanto, isso tem se mostrado incorreto na prática. O ambiente de desenvolvimento é caótico e sofre constantes mudanças, muitas vezes bruscas, o que causa grande impacto no produto final.

O processo de desenvolvimento ágil propõe uma nova abordagem. Essa abordagem é um conjunto de novas idéias, velhas idéias e velhas idéias modificadas, e enfatiza [2]:

- Uma maior colaboração entre programadores e especialistas do domínio;
- Comunicação "cara-a-cara" (como sendo mais eficiente do que comunicação documentada);
- Entregas constantes de novas funcionalidades com valor de negócio;
- Equipes auto-organizadas;
- Maneiras de adaptar o código e a equipe a novos requisitos que poderiam causar grande confusão;

De forma geral, metodologias ágeis priorizam "software funcionando" como principal métrica de progresso. Tendo em vista a preferência por comunicação "cara-a-cara", os métodos ágeis normalmente produzem menos documentação do que os outros métodos. É notável também a grande adaptabilidade dos projetos em relação a mudanças, o que no consagrado modelo "Waterfall" ou "Cascata" permanece uma grande dificuldade [4].

Neste capítulo será apresentado o ideal por trás do Manifesto Ágil, seus princípios e objetivos. Serão apresentadas também algumas das metodologias mais praticadas atualmente no mercado de software, como: Scrum, Extreme Programming (XP) e Feature

Driven Development (FDD). Analisaremos o funcionamento de cada uma delas, assim como também suas principais características, pontos fortes e pontos fracos, com o objetivo de comparar cada uma dessas metodologias.

2.1 Uma breve história de agilidade – Manifesto Ágil

"O Manifesto Ágil é uma declaração sobre os princípios que servem como base para o desenvolvimento ágil de software." [5] Foi elaborado de 11 a 13 de Fevereiro de 2001, em Utah, onde representantes de várias novas metodologias, tais como Extreme Programming, Scrum, DSDM, Adaptive Software Development, Crystal, Feature Driven Development, Pragmatic Programming, se reuniram para discutir a necessidade de alternativas mais leves em contraposição as tradicionais metodologias existentes [1].

Simplificadamente, o Manifesto Ágil prega os seguintes valores [5]:

"Indivíduos e interações acima de processos e ferramentas;

Software funcional acima de documentação compreensiva;

Colaboração do cliente acima de negociação do contrato;

Responder a mudanças acima de seguir um plano fixo;"

Nomeando a si mesmos como "Agile Alliance", esse grupo de inovadores em desenvolvimento de software concordaram em chamar esses novos valores de "Manifesto para o Desenvolvimento Ágil de Software" (em inglês: "Manifesto for Agile Software Development"). A frase final do Manifesto expressa de forma sucinta esse ideal [5]: "Sendo assim, enquanto existe valor agregado aos itens da direita, nós valorizamos mais os itens da esquerda."

Através desses valores, os fundadores do Manifesto Ágil chegaram a conclusão de doze princípios que deveriam reger as metodologias ágeis [6]:

1. Nossa maior prioridade é satisfazer o cliente, mediante a rápida e contínua entrega de software funcional.

2. Permitir mudanças nos requisitos, mesmo que tardiamente no projeto. Processos ágeis aproveitam mudanças para o benefício do cliente.

3. Entregar software funcional frequentemente, podendo variar entre semanas ou meses, com uma preferência por escalas de tempo menores.

4. Especialistas do domínio e desenvolvedores devem trabalhar juntos diariamente durante o projeto.

5. Desenvolver projetos em torno de indivíduos motivados. Dar a eles o ambiente e apoio de que precisam, e confiar neles para a realização do trabalho.

6. O método mais prático e eficiente de se comunicar com a equipe de desenvolvimento é conversar cara-a-cara com os integrantes.

7. Software funcional é a medida primária de progresso.

8. Processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante.

9. Atenção contínua a excelência técnica e bom design aumenta a agilidade.

10. Simplicidade - a arte de maximizar a quantidade de trabalho não terminado - é essencial.

11. As melhores arquiteturas, requisitos e modelos surgem de equipes auto-organizadas.

12. Em intervalos regulares de tempo, a equipe reflete em como se tornar mais eficiente, então se prepara e se ajusta de acordo.

Esses princípios, ao contrário do que muitos dizem, não formam a base para uma anti-metodologia. De fato, as metodologias ágeis não são uma forma de contrariar completamente as metodologias tradicionais. Na verdade, o Manifesto Ágil propôs uma nova abordagem na forma de desenvolver projetos. Por exemplo, modelagem é uma prática utilizada, mas não a ponto de se tornar primordial. Documentação é outra prática adotada

nos projetos, mas jamais levantar uma série de relatórios que jamais serão completamente lidos ou que levarão muito tempo para serem revisados. Logo, os princípios e valores propostos pelo Manifesto Ágil são uma nova forma de pensar desenvolvimento e gerenciamento de software, tendo em mente um conjunto de valores compatíveis, baseados na confiança e respeito pelos companheiros de equipe e promovendo modelos organizacionais simplificados e centrados em pessoas e em colaboração. Dessa maneira, construindo comunidades organizacionais nas quais seria prazeroso trabalhar.

2.2 Scrum

Scrum é um processo leve que visa gerenciar e controlar o desenvolvimento de software: é uma metodologia de gerência de projetos. No entanto, ao invés de promover a tradicional abordagem do modelo cascata ou "waterfall", análise – modelagem – codificação – teste – instalação do sistema, Scrum adota as práticas iterativa e incremental. Assim, Scrum não é dirigido a artefato (ou "artifact-driven"), onde são criados grandes documentos de requisitos, especificações, documentos de modelagem e diagramas. Pelo contrário, Scrum exige pouquíssimos artefatos. Ele se concentra no que é de fato importante: "Gerenciar um projeto ou codificar um software que produza valor de negócio" [4].

2.2.1 Definições

Sprint: Uma Sprint é uma iteração que segue o ciclo PDCA (Plan – Do – Check - Act) e entrega incremento de software funcional. Essas iterações duram normalmente de 2 a 4 semanas e começam e terminam com uma reunião da equipe.

Scrum Master: É o responsável por proteger os membros da equipe de desenvolvimento de qualquer pressão ou ameaça externa, seja isto clientes esbravejantes, diretores da empresa insatisfeitos ou qualquer outra coisa que seja considerado "perigoso" para a produtividade da equipe. Tenta garantir que todas as práticas do Scrum sejam utilizadas com perfeição pela equipe. Assim como também tem um papel de facilitador nas reuniões da Sprint. Normalmente assumem esse papel os gerentes de projeto ou líder

técnico, mas na prática pode ser assumido por qualquer um com experiência o suficiente na metodologia.

Product Owner: É o responsável por priorizar o Product Backlog. Este é um papel importante pois a equipe de desenvolvimento observará o Product Backlog priorizado e construirá o Sprint Backlog, comprometendo-se a entregar os itens postados. O Product Owner garante que durante a Sprint não haverá mudanças nos requisitos solicitados, porém, nos intervalos entre Sprints ele possui total liberdade para modificá-los. Essa função é realizada por uma pessoa que simulará o papel de cliente em relação ao produto (daí o nome "Owner"). O papel é normalmente exercido por alguém da área de marketing ou por algum usuário muito importante no contexto do desenvolvimento do produto.

Scrum Team: É a própria equipe de desenvolvimento. No entanto, isso não significa que é uma equipe tradicional de desenvolvimento de software com programadores, especialistas em BD, testadores e etc. . Na verdade, as equipes de desenvolvimento em Scrum são incentivadas a serem multidisciplinares, ou seja, todos trabalham cooperativamente para entregar as funcionalidades que a equipe se comprometeu a entregar no começo do Sprint. O tamanho típico de uma equipe varia entre 6 e 10 pessoas, mas nada impede que as equipes possuam mais colaboradores.

Dashboard: Visto que o Scrum prega a transparência dentro de uma equipe de desenvolvimento, um quadro onde todas as histórias (requisitos) existentes para um produto estejam expostas é extremamente útil. Este quadro é conhecido como Dashboard. Nele estão registradas, preferencialmente através de "post-its", as issues a serem entregues para o desenvolvimento de um produto. O dashboard possui o status das issues a serem entregues pela equipe. As issues vão se movendo no Dashboard com o passar do tempo de acordo com seu status, por exemplo: issue no Product Backlog (não está contida na Sprint atual), issue no Sprint Backlog (está contida na Sprint atual), issue no Work in Progress (trabalho sendo feito no momento exato), issue na revisão e issue finalizada. As divisões existentes podem ser modificadas de acordo com a necessidade de cada equipe. A figura 1 mostra como deve ser a organização de um dashboard;

Product Backlog: É a lista principal de funcionalidades para o desenvolvimento de um dado produto. O Product Backlog pode e deve mudar no decorrer das sprints tornando-se maior, melhor especificado e mudando as prioridades das issues. É um elemento essencial na reunião de Planning, pois é com o Product Backlog (priorizado) que o Product Owner irá se comunicar com a equipe de desenvolvimento com o objetivo de expressar suas maiores necessidades para aquela sprint.

Sprint Backlog: Na reunião de planejamento da Sprint (Planning) o Product Owner deverá mostrar o Product Backlog priorizado e a equipe irá escolher quais issues daquela lista serão realizadas durante aquela sprint. As issues escolhidas serão colocadas na sessão de Sprint Backlog, ou seja, lista de funcionalidades a serem implementadas na sprint corrente.

Issue ou Estória: Cada elemento da lista é conhecido como Issue e deve ser identificado com um nome e/ou id com objetivos de identificação e rastreamento. Essas issues devem possuir também uma breve descrição das funcionalidades requeridas pelo Product Owner para um dado produto. O time pode dividir uma issue em tarefas, de preferência pequenas, para que, com a conclusão de todas as tarefas, a própria issue esteja concluída. É uma forma de usar a abordagem "dividir para conquistar" em gerenciamento de projetos.

Sprint Planning: É a reunião principal do Scrum. Nela são planejadas as atividades para a sprint corrente. Nesta reunião, o Product Owner apresenta o Product Backlog priorizado, para daí então, os membros da equipe decidirem quais issues do Product Backlog serão incluídas no Sprint Backlog. De forma coletiva, o Product Owner e a equipe de desenvolvimento definem um objetivo para o Sprint, que é na verdade uma breve descrição sobre o que se deseja entregar na Sprint.

Sprint Review: No final de cada sprint, a equipe de desenvolvimento, o Scrum Master e o Product Owner se reúnem para verificar o que foi feito durante a sprint. A equipe apresenta as funcionalidades prontas e o Product Owner aprova ou desaprova o produto. Essa reunião é aberta para qualquer outra equipe presente na empresa que queira observar a apresentação das issues completadas durante o sprint. A reunião possui a forma

de uma demo de produto, onde são apresentadas novas funcionalidades, bugs consertados entre outras coisas.

Sprint Retrospective: É realizada logo após o Sprint Review e tem como objetivo extrair para todos da equipe a essência do que houve de melhor e do que pode melhorar na sprint analisada. Essa reunião é fechada para apenas a equipe e o Scrum Master. Durante essa reunião serão colhidas, através de comunicação cara-a-cara entre os integrantes da equipe, informações sobre o andamento do projeto. Os membros da equipe são incentivados a falar sobre o que estão achando do projeto (em todos os sentidos) e é realizada pelos próprios membros uma retrospectiva em formato de "timeline" especificando as datas chave para o sprint. Essa reunião é considerada muito importante para o crescimento da equipe, pois é nela onde serão mostrados os erros e consequentemente as lições aprendidas durante a sprint.

Daily Scrum: É uma reunião realizada diariamente entre os membros da equipe, Scrum Master e opcionalmente com a presença do Product Owner com o objetivo de analisar o que cada membro da equipe fez no dia anterior, o que pretende fazer durante o dia presente e quais os impedimentos ou obstáculos que tem para cumprir com as tarefas. Normalmente é feita em formato de Stand-up-meeting e é de duração bem curta. É uma forma eficiente de manter a comunicação aberta entre os membros da equipe de desenvolvimento e o Scrum Master.

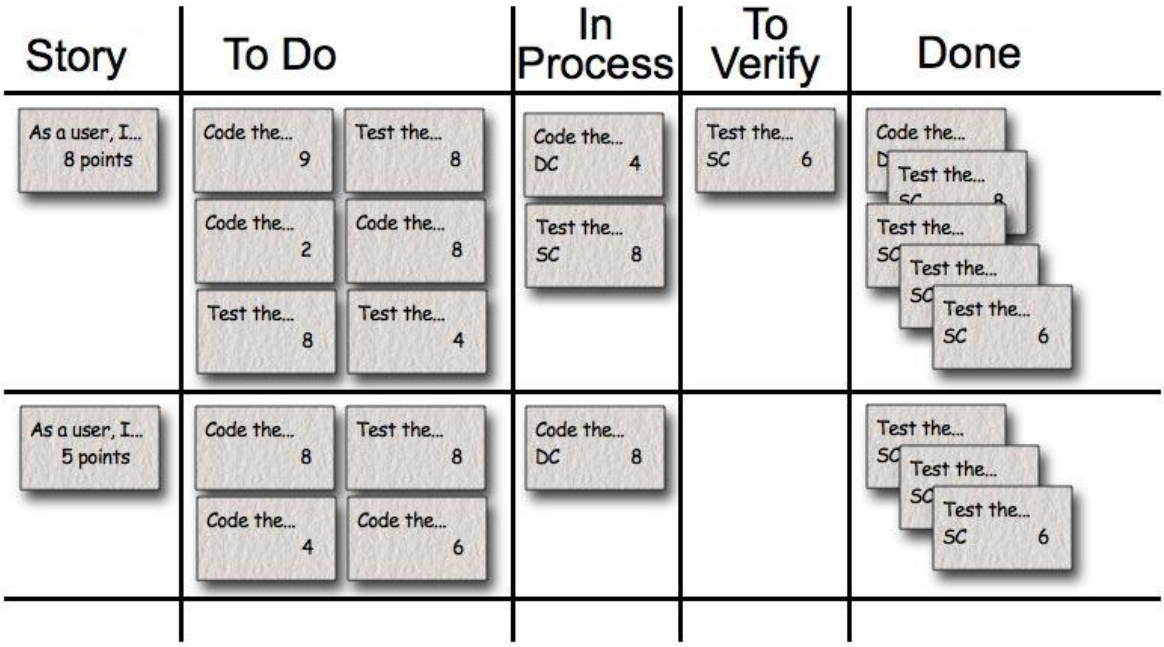

Figura 1. Esquema de um dashboard no meio de uma sprint, mostrando tarefas a fazer, em processo, a verificar e finalizada. [13]

2.2.2 Fases

Planejamento: Essa fase marca o início de uma nova Sprint. O Sprint Planning é a cerimônia na qual a equipe se reúne com o Product Owner para analisar as issues priorizadas por este e decidir quais destas a equipe se compromete em entregar. Essas reuniões costumam durar de 1 a 3 horas com a participação ativa de todos da equipe. Geralmente o Product Owner explica brevemente o objetivo de cada issue e, caso surja alguma dúvida, explica mais detalhadamente para todos.

Desenvolvimento: Essa fase é a maior do Sprint, pois representa todos os dias em que a equipe está trabalhando concentradamente na entrega do software. Em cada um desses dias é realizado o Daily Scrum com os membros da equipe. Neste são discutidas questões relativas à entrega do software, como “o que foi feito” e “o que pretendo fazer até o próximo Daily Scrum”.

Conclusão: A conclusão de uma sprint é marcada por duas cerimônias: Sprint Review e Sprint Retrospective. A Sprint Review é uma cerimônia aberta para toda a empresa com o objetivo de mostrar as novas funcionalidades implementadas pela equipe de desenvolvimento durante a Sprint. A equipe faz uma breve demonstração para o Product Owner, que aprova ou não a nova funcionalidade. Normalmente a Sprint Retrospective acontece logo após a Sprint Review. A Sprint Retrospective é de fato a última reunião da Sprint. Nesta reunião moderada pelo Scrum Master, a equipe discute seus melhores e piores momentos no decorrer da Sprint. É comum acontecer que, ao final desta reunião, surjam feedbacks de coisas a melhorar, seja na equipe ou na própria organização. É válido salientar que esta reunião não é aberta nem mesmo ao Product Owner, mas somente a equipe e ao Scrum Master, pois tem como objetivo realmente extrair o que cada membro está sentindo com relação ao andamento do projeto. Por este motivo, é desaconselhada a presença de pessoas externas ao grupo na sala de reunião da equipe.

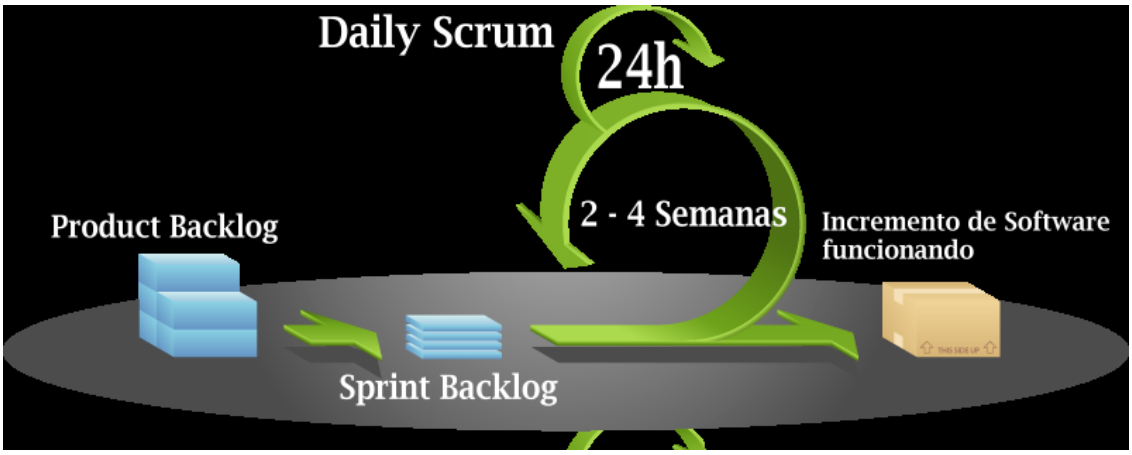

Figura 2. Funcionamento de uma Sprint recebendo Sprint Backlog como entrada e após vários Daily Scrums (duração média da Sprint de 2 a 4 semanas), a entrega do incremento de software funcionando. [13]

2.2.3 Vantagens e desvantagens

As metodologias tradicionais são projetadas para reagir a mudanças imprevisíveis apenas no começo de um ciclo de desenvolvimento. Mesmo as metodologias que não

seguem o modelo cascata (seguem o modelo espiral por exemplo), mostram-se limitadas para reagir a mudanças uma vez que o projeto tenha começado.

Scrum, por outro lado, é "lightweight", ou seja, leve. Foi modelado para ser adaptável a mudanças durante todas as fases do projeto. Ele provê mecanismos de controle para planejar uma entrega de produto e, enquanto o projeto avança, provê variáveis de gerenciamento. É essa abordagem que permite a mudança do projeto e de entregas a qualquer momento, entregando assim, a parte mais apropriada do software desenvolvido.

Scrum incentiva a criatividade dos membros da equipe de desenvolvimento na resolução de problemas. Isso acontece quando os problemas surgem no projeto e/ou o ambiente de desenvolvimento sofre mudanças. Neste caso, os membros da equipe podem vir com as soluções mais diferentes possíveis de modo a resolver o problema e trazer a equipe novamente ao rumo correto. A decisão nesses casos é sempre da equipe.

Equipes pequenas, colaborativas e multidisciplinares compartilham conhecimento sobre todas as fases do processo de desenvolvimento. A conseqüência imediata disto é a baixa concentração de informação em apenas um membro da equipe. Caso algum membro precise sair da equipe por qualquer que seja o motivo, o impacto sobre o restante do grupo não será muito grande. Além do mais, essa característica provê um treinamento excelente para todos os membros da equipe em todas as áreas do processo de desenvolvimento.

A tecnologia Orientada a Objetos (OO) provê a base para o Scrum. Por ser bastante prática e "limpa", "a visão OO garante que os processos de Scrum serão simples de entender e de usar" [7].

O curto intervalo de tempo entre os releases é uma vantagem. Dessa forma, o cliente pode acompanhar de perto a evolução do projeto e fornecer feedbacks a cada sprint, tornando-se assim também uma vantagem para a equipe desenvolvedora. Desenvolve-se desde cedo uma relação de confiança com o cliente, o que é um grande fator de sucesso organizacional atualmente. [7]

Scrum não se mostra vantajoso no caso de grandes projetos com duração muito longa, prazo muito curto, muitas pessoas trabalhando em muitas camadas e orçamento

muito alto. Ken Schwaber, um dos fundadores de Scrum diz que quando se investe muito alto num projeto com um prazo muito apertado, os líderes da organização não estão apoiando a independência das equipes de desenvolvimento. O que costuma acontecer é que, após apenas uma Sprint, os gerentes de projeto analisam o Release Burndown e constatam que é impossível entregar o produto no prazo determinado. Normalmente os investidores recuam e o projeto é cancelado. Nesses casos é mais vantajoso usar o modelo cascata, pois só se constata que o projeto vai atrasar de fato após alguns meses de trabalho e já é tarde demais para cancelar o projeto [7].

Scrum é um método que exige uma gestão constante e muito próxima. Isso significa que o gestor deve estar sempre removendo barreiras e monitorando as equipes de desenvolvimento, se certificando de que as práticas da metodologia estão sendo postas em prática como deveriam. Monitoramento constante é exigido do gestor.

Como Scrum provê um alto nível de independência às equipes, o gestor deve permitir que as equipes tomem suas próprias decisões, permitindo até que estas falhem ao fazê-lo. Assim, é possível haver prejuízos em curto prazo, tanto em sentido financeiro quanto em tempo.

É válido lembrar que Scrum é uma metodologia relativamente nova e que as pessoas são resistentes a mudanças. Na fase inicial de implantação da metodologia numa organização, desconforto pode existir. Há também a possibilidade de alguns elementos dentro da organização não conseguirem se adaptar ao novo ritmo e nova filosofia do Scrum. [6]

2.3 Extreme Programming

Extreme Programming (XP) é na verdade uma abordagem para o desenvolvimento de Software. Esta metodologia foi criada com o objetivo de entregar o software que o cliente precisa quando ele precisa. XP encoraja os desenvolvedores a se adaptarem a mudanças no escopo do projeto, mesmo que tardiamente no ciclo de produção do software.

Assim como a metodologia Scrum, XP enfatiza o trabalho em equipe como chave para um processo de produção de Software mais ágil e com melhor qualidade. [13]

2.3.1 Princípios e diferenças para outras metodologias

Os princípios vistos no Manifesto Ágil são a base dos princípios do XP [14]. São eles:

- Codificar é a atividade chave;
- Cultura de Software pronto no tempo do cliente;
- Ciclos freqüentes de lançamento (Release);
- Comunicação é feita a partir do código;
- Melhorar código com Refactoring.

Esses princípios não indicam que o desenvolvedor, quando estiver trabalhando, apenas comece a escrever código rápida e desesperadamente. Pelo contrário, é preciso ser disciplinado nesse sentido ou o efeito será contrário ao desejado.

A aplicação desses princípios num projeto de desenvolvimento de software é a base para o estabelecimento da metodologia XP. Logo, diferentes características surgirão desta nova metodologia em relação a outras existentes no mercado. Podemos citar algumas dessas diferenças [17]:

1. Feedback rápido e contínuo advindo de um ciclo de vida mais curto do projeto;
2. Planejamento incremental, não estático. Neste o plano de projeto pode evoluir;
3. Implementação de forma flexível, de modo a se adequar às necessidades do cliente;

4. Baseia-se em comunicação oral, testes, e escrever código como uma forma de descrever a estrutura e propósito do sistema;

5. Alto nível de colaboração entre programadores com menos habilidades;

2.3.2 Boas Práticas

Pair-Programming: Programação em par ou Pair-Programming é uma técnica para o desenvolvimento de software na qual um programador não trabalha sozinho, mas sim com um companheiro de equipe. Assim, compartilham experiências, pensam juntos antes de escrever código, tiram dúvidas um do outro, enfim, trabalham conjuntamente num mesmo computador. Nesta técnica, temos um piloto, que é o usuário que está escrevendo código, e temos o navegador, que é o usuário responsável por revisar cada linha de código escrita. Isso costuma aliviar a pressão em cima do piloto, pois o navegador possui também a responsabilidade de procurar eventuais "falhas" tanto na escrita quanto no design do código [15]. Para que nenhuma das duas partes se limite a apenas um tipo de atividade (só codificar ou só revisar), os usuários devem trocar de lugar frequentemente. A grande vantagem da programação em par é que, dessa forma, o código final apresenta maior legibilidade, melhor design e menor susceptibilidade a erros. Por outro lado, a programação em par pode causar intimidação em alguns programadores mais retraídos ou menos habilidosos, diminuindo a produtividade da dupla.

Planning Game: O processo de planejamento na metodologia Extreme Programming é chamado de Planning Game ou Jogo do Planejamento. Esse jogo acontece uma vez por iteração e é dividida em duas fases [15]:

- Planejamento do Release: O foco desta reunião é planejar quais requisitos serão escolhidos para a próxima iteração e quando será realizada a entrega dessas funcionalidades. Primeiramente, o cliente proverá uma lista de requisitos que deseja para a sua aplicação. Estes requisitos serão mapeados em estórias e escritos em cartões. Daí a equipe desenvolvedora irá escolher algumas dessas estórias para ser

implementadas e se comprometerão com a entrega dessas. Finalmente, com as estórias em mãos, a equipe e o cliente poderão se reunir e decidir se algo mais será acrescentado àquela estória, removido da estória ou replanejado.

- Planejamento de Iteração: Neste planejamento, o cliente não participa. Apenas a equipe de desenvolvedores se reúne para dividir as estórias escolhidas em tarefas menores. Primeiramente, as estórias são repartidas em tarefas e atividades e escritas em cartões. Daí, os desenvolvedores decidem quem ficará com qual tarefa e estimam quanto tempo gastarão para terminá-las. Após as tarefas serem concluídas, é feita uma comparação entre o que foi planejado e o que de fato foi feito.

Testes: Um fator que chama atenção na metodologia XP é que num projeto, a equipe inteira tem responsabilidade pela qualidade do produto. Dessa maneira, todos os participantes da equipe, tanto programadores quanto testadores, são responsáveis por criar testes (incluindo todos os tipos) para a aplicação. Testar é uma atividade integrada num projeto XP. Os testadores devem fazer o máximo para "transferir" suas habilidades relativas à atividade para todos os membros da equipe. Abaixo, citamos algumas atividades dos testadores numa equipe XP [16]:

- Negociar qualidade com o cliente (o cliente é que decide o quão padronizado ele quer o código);
- Esclarecer estórias e retirar dúvidas que surgem durante a implementação;
- Certificar-se de que os testes de aceitação verificam a qualidade especificada pelo cliente;
- Ajudar a equipe a automatizar testes;
- Testar a qualquer momento, sem qualquer restrição para isso. Qualquer integrante pode testar na hora em que quiser;

Definição e redefinição da arquitetura a todo o momento: Visto que em XP, a equipe de desenvolvimento está sempre analisando código, testando funcionalidades e alterando o design da aplicação, a arquitetura está sempre sendo atualizada. Isso não implica dizer que a aplicação não possui uma padronização na arquitetura. Pelo contrário, a equipe está frequentemente alterando a estrutura da aplicação para ter maior flexibilidade escalabilidade e legibilidade, atributos que envolvem diretamente padronização no código. Não há confusão quanto às redefinições de arquitetura porque a equipe está se comunicando a toda hora (um dos princípios do Manifesto Ágil).

Stand-up Meeting: Reunião de curta duração (entre 5 e 10 minutos) entre os membros de uma equipe XP para discutir assuntos relacionados ao projeto. Esse tipo de reunião é realizada normalmente todos os dias (podendo ser convocada a qualquer hora) com os membros todos de pé (daí o nome "stand-up"). O objetivo é conseguir maior concentração dos membros durante a reunião assim como evitar distrações, como conversas paralelas, assistir a vídeos no computador, chats e coisas do tipo. Por estarem todos de pé, os membros não irão tirar o foco do assunto em pauta e não demorarão muito para terminar a reunião, que deve ser curta.

2.3.3 Elementos

Gerente de Equipe: É o responsável por assuntos administrativos do projeto. Deve possuir grande habilidade ao lidar com pessoas, visto que irá tratar de assuntos ligados ao projeto com a equipe e com os clientes. É um tipo de mediador entre os interesses da equipe e do cliente. Outro papel do gerente de equipe é manter as ameaças externas longe da equipe sob sua supervisão. Também é responsável por tentar manter o cliente informado do projeto em questão e, ainda mais importante para a empresa, participando do desenvolvimento de forma direta. Obviamente, um bom gerente de equipe deve acreditar em todos os valores de XP para que possa cobrar isto da sua equipe.

Coach: Responsável pelas questões técnicas do projeto de software. O coach deve possuir grande conhecimento em todas as fases do XP e conhecer bem os membros da equipe. É o responsável por estimular a equipe a dar o melhor sempre que possível. Ao

conhecer e aplicar os conceitos de XP, deve estar apto para sinalizar a sua equipe os erros cometidos ao longo do projeto e dar os passos necessários para consertá-los.

Analista de teste: Responsável por garantir a qualidade do sistema desenvolvido através da atividade de testes do sistema. Normalmente o analista de testes ajuda o cliente a escrever os casos de testes necessários para garantir que os requisitos ordenados serão devidamente satisfeitos. No final duma iteração, deve realizar os testes no software a ser entregue com o objetivo de verificar se existem bugs e requisitos não-cumpridos. É perigoso usar um desenvolvedor do software para testá-lo e construir casos de teste junto ao cliente. O que pode acontecer é que, pelo fato de o desenvolvedor conhecer o código, ele terá uma visão tendenciosa, evitando "tocar" nos pontos que ele sabe serem sensíveis no sistema. Por outro lado, o analista de testes deve ter uma visão muito parecida com a do cliente.

Redator técnico: Pessoa responsável por cuidar da documentação do projeto. Dessa forma, a equipe de desenvolvedores pode se concentrar exclusivamente em codificar. Para que a documentação esteja de acordo com o código escrito e as necessidades do cliente, o redator deve estar sempre em comunicação com os desenvolvedores, sabendo o que estão desenvolvendo e como.

Desenvolvedor: É o papel principal no XP. O desenvolvedor é responsável por analisar, projetar e escrever código para o sistema em desenvolvimento. Numa equipe de desenvolvedores utilizando XP, os desenvolvedores estão todos no mesmo nível. Por exemplo, não existe diferença entre analista e programador uma vez que eventualmente todos executarão uma dessas atividades.

2.3.4 Fases

- **Exploração**: Esta é a primeira fase de um projeto utilizando XP e é realizada anteriormente à construção do sistema. São estudadas várias soluções para resolver o problema do cliente e verificadas as viabilidades de cada uma destas soluções. Nesta fase são pensadas as possíveis arquiteturas e ambientes, levando

em conta as condições necessárias para a implementação do sistema, como plataforma, sistema operacional, configurações da rede, hardware e linguagem utilizada. O objetivo desta fase é fazer os programadores e os próprios clientes ganharem confiança no planejamento do projeto para que, logo mais, comecem a escrever estórias e começar a construção do sistema.

- **Planejamento inicial**: É utilizada para que os clientes e a empresa desenvolvedora do sistema possam acordar em uma data para o primeiro release. O planejamento tem início com os programadores estimando as estórias escritas na fase de exploração e as escrevendo com seu valor estimado em cartões. Após a estimativa de cada uma das estórias, a equipe informa quantas estórias podem ser entregues levando em consideração a média de estórias entregues em iterações passadas. Por sua vez, o cliente prioriza as estórias em seu ponto de vista. O planejamento inicial obtém como saída uma série de cartões com as estórias priorizadas e estimadas pela equipe, prontas para guiar os desenvolvedores na próxima fase. Normalmente a fase de planejamento inicial dura de uma a três semanas e cada release dura de dois a quatro meses.
- **Iterações do release**: Nessa fase começa a construção do sistema propriamente dita. Os desenvolvedores começam a codificar, testar, refatorar, analisar código, depurar bugs e projetar, sempre seguindo os princípios e valores da metodologia XP. Frequentemente durante uma iteração, a equipe realiza reuniões de stand-up meeting, onde serão discutidas questões relativas a implementação do sistema. O resultado da primeira iteração de um produto é essencial para o cliente ter uma idéia de como o produto está sendo construído (se está cumprindo os requisitos acordados anteriormente) e se haverá atraso na entrega da release.
- **Produção**: Após um release, o sistema está pronto para ser simulado em um ambiente de produção. Nesta fase, o sistema será testado observando atributos como confiabilidade e performance. Testes de aceitação podem ser realizados de acordo com a necessidade do cliente.
- **Manutenção**: Após um release o sistema estará funcionando bem o suficiente para poderem ser feitas manutenções. Isso pode envolver várias atividades como por exemplo: refatoramento de código existente, introdução de uma nova

arquitetura, instalação de um novo servidor no ambiente de desenvolvimento ou introdução de uma nova tecnologia. É válido ressaltar que uma vez que o sistema estiver em produção, fazer manutenções se torna perigoso. Por isso, a equipe deve tomar muito cuidado ao modificar o sistema, visto que uma alteração errada ou mal planejada pode causar danos gigantescos ao sistema, causando prejuízo ao cliente.

- **Morte**: Essa fase constitui o fim de um projeto XP. Normalmente ocorre quando o cliente já está plenamente satisfeito com o sistema obtido e não consegue visualizar nenhuma funcionalidade essencial para ser implementada futuramente. Porém, podem ocorrer casos de um projeto XP morrer antes de completar o produto final. Por exemplo, se após a primeira iteração o cliente observar que o tempo e orçamento necessários para a finalização do projeto estão fora do alcance. Nestes casos, os clientes podem cancelar o projeto, causando seu fim.

2.4 Feature Driven Development

Feature Driven Development (FDD) é uma metodologia ágil para desenvolvimento de software, voltada para o cliente e orientada a modelagem. Consiste de cinco atividades ou fases básicas e possui métricas para marcar o progresso dessas atividades [22]. Nessa metodologia, as entregas, assim como nas outras apresentadas nesse trabalho, são incrementais. Atualmente, é crescente o número de empresas de software que a utilizam por esta oferecer um esquema orientado a modelagem e agilidade ao processo de desenvolvimento [23].

2.4.1 Elementos

- **Gerente de Projeto:** É o personagem central deste método. Está à frente do projeto. Possui como tarefas, delegar responsabilidades aos arquitetos-chefe e ao gerente de configuração (podendo também acumular esse papel), assim como também resolver problemas administrativos ligados ao projeto.

- **Arquitetos-chefe:** Responsáveis pela modelagem do sistema em todas as fases. Como o FDD dá bastante ênfase à modelagem, os arquitetos-chefe são bastante cobrados e são diretamente responsáveis pelo sucesso do projeto.
- **Programadores-chefe:** Os programadores-chefe são os desenvolvedores mais experientes. A eles são atribuídas funcionalidades a serem construídas. Entretanto, eles não as constroem sozinhos. Ao invés disso, o programador-chefe identifica quais classes estão envolvidas para se desenvolver a funcionalidade e reúne seus proprietários de classes, criando uma equipe para implementar aquela funcionalidade. O programador-chefe age como um coordenador, designer líder e mentor, enquanto os proprietários de classes fazem a maior parte da programação das funcionalidades.
- **Proprietários de código/classes:** São os desenvolvedores da equipe. Codificam, projetam e fazem análise de código a qualquer nível. Possuem propriedade sobre o código, de forma que ficam responsáveis pela parte de código que lhes foi designada. Assim, são diretamente responsáveis pela legibilidade, bom design e clareza do código.
- **Especialistas do domínio:** Podem ser membros externos ou internos do projeto. Interno seria algum membro designado para entender completamente as necessidades do cliente de modo que pudesse representá-lo nas reuniões da equipe. Externo no caso de o próprio cliente participar das reuniões no projeto. São as pessoas que mais conhecem sobre o assunto abordado. O objetivo deles é passar para a equipe o máximo de conhecimento sobre o domínio, de modo que os membros da equipe possam também se tornar experts no domínio.
- **Gerente de configuração:** Membro responsável por controlar o código fonte de todo o sistema, mantendo sempre atualizadas as alterações realizadas ao término ou início de cada funcionalidade. Normalmente o gerente de configuração assume outros papéis relativos à configuração dos sistemas da equipe de desenvolvimento, aliviando a quantidade de trabalho desperdiçada pelos desenvolvedores com problemas relativos a falhas técnicas das máquinas (formatação do sistema, backups, troca de mouses ou teclados e etc.) entre outras coisas.

- **Testador:** Membro responsável por testar as funcionalidades implementadas pelos desenvolvedores do projeto. Escrever testes, casos de teste e testar funcionalidades manualmente são as principais tarefas do testador.
- **Redator técnico:** Responsável pela documentação gerada nas reuniões de planejamento da equipe. Dessa maneira, alivia os membros da equipe de trabalhos relativos à escrita de documentos, focando exclusivamente na modelagem e construção das funcionalidades.

2.4.2 Boas práticas

Feature Driven Development possui um conjunto de boas práticas baseadas nas práticas de engenharia de software. Essas práticas são baseadas na visão de valor do cliente, e não da equipe de desenvolvimento [23]. É justamente essa fusão de práticas e técnicas que torna o FDD tão útil. Algumas práticas que têm sido amplamente aproveitadas no mercado de desenvolvimento de software são listadas.

Modelagem do domínio: Explora e explica o domínio do problema a ser resolvido tendo como perspectiva a orientação a objetos. Esta é considerada uma boa prática, pois ajuda a descrever o domínio da aplicação, provendo uma modelagem genérica de alto nível no qual podem ser adicionadas várias funcionalidades.

Desenvolvimento por funcionalidade: Qualquer atividade a ser desenvolvida deve ser analisada para verificar se esta pode ser quebrada em atividades menores, ou funcionalidades atômicas (indivisíveis). Essa prática torna o código inteiro mais seguro contra erros. Além do mais, garante flexibilidade e escalabilidade ao código [24].

Posse sobre o código: Em FDD, os desenvolvedores possuem individualmente posse de código. Por exemplo, a classe A, B e C pertencem ao programador 1, enquanto as classes X, Y e Z pertencem ao programador 2. O proprietário do código torna-se automaticamente responsável pela performance, consistência, corretude e integração da classe. O proprietário é aconselhado a utilizar os melhores padrões da engenharia de software. No entanto, está livre para criar seus próprios padrões.

Equipes de funcionalidades: São equipes pequenas responsáveis por desenvolver uma pequena atividade. Dessa forma, a equipe decide de forma conjunta como será feito o

design de determinada funcionalidade. Com mais de uma pessoa pensando no mesmo problema, soluções diferentes podem surgir. O que se torna muito eficiente, visto que no final essas soluções podem ser fundidas em uma solução final da equipe.

Inspeções: Inspeção de código é uma ótima prática de engenharia, pois além de fazer verificações contra erros, incentiva uma melhor modelagem do sistema, junto com atributos como legibilidade e alta coesão.

Gerência de configuração: Essa prática tem como objetivo manter um controle sobre o código fonte das funcionalidades implementadas, mapeando as funcionalidades aos seus respectivos códigos-fontes e aos seus proprietários. Além disso, mantém as datas de modificação do código, guardando um histórico de alterações.

Build constante: Deve existir sempre uma versão do sistema rodando numa máquina. Isso garante que a equipe possui pelo menos uma versão do sistema que funciona. Dessa forma, sempre que for necessário apresentar alguma funcionalidade para o cliente, existirá uma versão do sistema que pode ser utilizada para isso.

Visibilidade do progresso: Como sempre é mantido um relatório de progresso das atividades do projeto, todos na equipe e fora dela sabem exatamente como estão em termos de produtividade. No entanto, essa atividade deve ser feita com muita precisão e frequência. Caso contrário, os dados extraídos do relatório serão enganosos e poderão levar a decisões desastrosas para o cliente.

2.4.3 Fases

O Feature Driven Development é dividido em 5 atividades, cada uma destas possuindo sub-atividades que compõem o processo como um todo. Essas 5 atividades são (em ordem) [23]: Desenvolver um modelo geral, gerar uma lista de funcionalidades, planejar por funcionalidade, modelar por funcionalidade e construir por funcionalidade. As três primeiras são realizadas uma vez antes da realização do projeto e as 2 restantes são realizadas a cada iteração. Todas essas atividades serão explicadas e suas sub-atividades serão listadas nesse tópico.

Desenvolver um modelo geral: O projeto começa com uma análise superficial do escopo do sistema e seu contexto. Assim, são estudados os domínios de negócio do sistema e criado um modelo geral baseado nestes. Depois é criada uma modelagem superficial para cada área de domínio existente. Cada um desses modelos é revisado por um grupo aleatório de membros do projeto e melhorias são discutidas. É escolhido o modelo de domínio melhor projetado, ou senão é escolhido um modelo que é a junção de mais de um domínio projetado. Finalmente, os modelos escolhidos para cada área de domínio são fundidos para gerar um modelo do domínio como um todo (ou domínio principal) do sistema.

Sub-atividades:

- Formar equipe de modelagem;
- Estudar o domínio de negócio;
- Estudar os documentos;
- Formar várias equipes pequenas para sugerir uma solução de modelo;
- Desenvolver o modelo escolhido;
- Refinar o modelo geral gerado;
- Escrever notas explicativas sobre o modelo final;

Gerar uma lista de funcionalidades: O conhecimento obtido na fase de desenvolvimento do modelo geral é essencial para esta fase. Nesta, será elaborada uma lista de funcionalidades do sistema decompondo as áreas de domínio obtidas. Cada funcionalidade é uma pequena tarefa a ser implementada que gere valor para o cliente. Devem seguir o formato <ação> <resultado> <objeto>, por exemplo: “Gerar relatório de vendas” ou “Validar a senha do usuário”. Essas funcionalidades são muito importantes para o processo como um todo. A não identificação delas causa um impacto enorme sobre o projeto, pois significa que a primeira fase não foi feita com sucesso e consequentemente os efeitos aparecem em cascata nas próximas fases [22].

Sub-atividades:

- Escolher uma equipe para gerar uma lista de funcionalidades;
- Gerar a lista de funcionalidades.

Planejar por funcionalidade: É realizado o planejamento de desenvolvimento de cada funcionalidade da lista obtida da fase anterior. São designadas classes ou código específico para os programadores-chefe tomarem conta. Daí em diante, os programadores-chefe serão responsáveis pelo código produzido nessas classes.

Sub-atividades:

- Formar uma equipe de planejamento
- Determinar a sequência de desenvolvimento das funcionalidades;
- Designar atividades de negócio para os programadores-chefe;
- Designar classes para os desenvolvedores.

Modelar por funcionalidade: Os programadores-chefe escolhem algumas funcionalidades para que, junto com os proprietários de código, sejam feitos os diagramas de sequência e a modelagem completa das funcionalidades. A diferença desta modelagem para a modelagem feita na primeira fase é que esta é específica para a funcionalidade em questão. Ou seja, nesta etapa pensa-se nas classes, métodos e atributos que irão existir. Ao final da modelagem, é realizada uma inspeção do modelo pela equipe que a fez ou por outra equipe designada.

Sub-atividades:

- Formar uma equipe para a funcionalidade em questão;
- Estudar a funcionalidade como parte inserida no modelo de domínio;
- Estudar documentos relacionados á funcionalidade;
- Desenvolver diagrama de sequência;
- Refinar objeto modelo;
- Escrever as classes e as assinaturas dos métodos (tipo de retorno, parâmetros e exceções lançadas);
- Realizar inspeção da modelagem.

Construir por funcionalidade: Após a modelagem, o código é finalmente implementado e os testes finalmente escritos. Assim, o código fonte é produzido e as funcionalidades ganham vida. O programador-chefe designa um programador para implementar uma certa funcionalidade. Logo, este último será o proprietário do código escrito. Após uma inspeção no código escrito, a funcionalidade é terminada.

Sub-atividades:

- Implementar as regras de negócio das classes;
- Inspecionar código;
- Conduzir testes unitários;
- Release da funcionalidade.

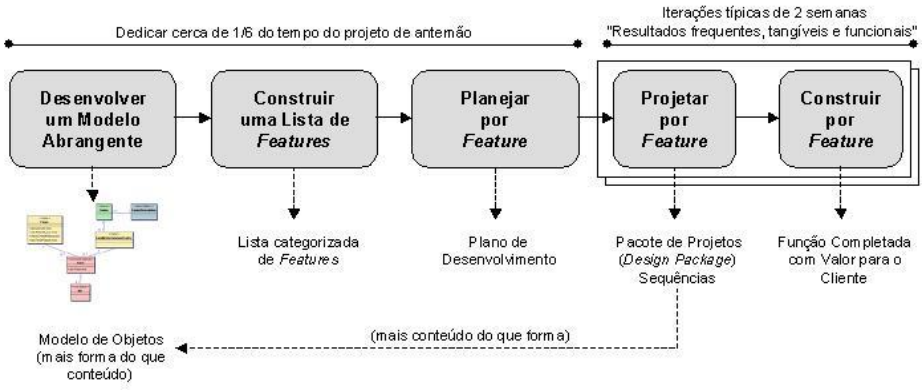

Figura 3. Os 5 processos do FDD.[24]

2.4.4 Vantagens e desvantagens

Por ser orientado a modelo e utilizar logo cedo no projeto (na fase de Desenvolvimento de um modelo geral) transferência de conhecimento ao utilizar várias equipes para projetar várias soluções para o domínio pedido, FDD agrega valor aos membros das equipes [24]. E ainda por cima, o conhecimento do domínio estará crescendo de forma similar através de todos os membros de todas as equipes. Isso é possível pelo fato

de a modelagem realizada inicialmente ser feita de maneira superficial e não específica, habilitando todos os membros das equipes a aprenderem sobre o domínio.

Para um gerente de projeto, ter um modelo preciso para poder se basear é essencial para o andamento do projeto. Para se ter esse modelo preciso, deve haver algum tipo de atividade de análise que gere informação. É exatamente o que faz a primeira fase do FDD [24].

FDD torna o trabalho do gerente de projeto mais fácil ao medir o progresso de cada funcionalidade em termos de valor para o cliente. Se todo o projeto é feito utilizando a noção de valor do cliente, será mais fácil discutir assuntos como mudanças e escopo do projeto. Ou seja, unifica a linguagem utilizada por todos no projeto, incentivando a comunicação de todos os pontos.

FDD enfatiza bastante a modelagem do sistema. Apesar de a última fase ser implementação do código, a fase mais crítica dessa metodologia é a fase de modelagem das funcionalidades. A arquitetura do sistema é analisada em quase todas as fases. Por estar sempre sendo testada, a modelagem do sistema tende a ser mais flexível e reutilizável.
Apesar de possuir um ciclo completo de documentação, a especificação original da aplicação não comenta sobre sua aplicabilidade a manutenção de sistemas. Este é um ponto "em aberto" para a metodologia FDD [22].

Por ser extremamente focado em documentação de modelagem do sistema, FDD foge um pouco de um dos princípios do Manifesto Ágil: "Software funcional sobre documentação abrangente.". Muita documentação é produzida durante as 4 primeiras fases e o código só é realmente experimentado na última fase, perdendo prioridade diante de outras atividades de modelagem.

Por possuir vários papéis, fica evidente que uma equipe FDD não pode ser muito menor que o número de papéis existentes na metodologia. Caso isto ocorra, uma mesma pessoa passará a assumir várias responsabilidades distintas dentro de uma mesma equipe. Assim, o foco deste membro não será tão consistente, poderá ocorrer sobrecarga de trabalho e conflitos de interesses, causando impacto sobre o projeto.

2.5 Análise Comparativa

As 3 metodologias analisadas possuem um objetivo comum: conferir agilidade ao processo de desenvolvimento de software ganhando eficiência. Porém, as três possuem características diferentes. São essas características que serão comparadas neste tópico. No quadro abaixo são mostrados alguns atributos e como as metodologias se adaptam a estes. Observe a legenda:

- * - 1 asterisco (Acontece com freqüência razoável)
- ** - 2 asteriscos (Acontece com freqüência alta)
- - 0 asterisco (Ausência desse atributo)

Características/Metodologia	FDD	XP	SCRUM
Equipes pequenas	-	*	**
Ênfase no modelo	**	-	-
Ênfase gerencial	*	-	**
Interação Cliente/Desenvolvedor	*	**	**
Gerenciamento de risco	-	-	*
Projetos grandes/complexos	*	-	-
Stand-up meetings	-	**	**
Especialista em	**	*	-

documentação			
Código coletivo	-	**	*

Tabela 1. Análise comparativa entre Scrum, XP e FDD.

Analisemos agora, essas características uma por uma.

- **Equipes pequenas**:

Scrum: Em Scrum existem poucos papéis (Scrum Master, Product Owner e membro da equipe) que não podem ser acumulados por um mesmo membro. Além disso, as técnicas gerenciais do Scrum são muito flexíveis, podendo ser usadas em projetos não ligados a área de software, inclusive com pouquíssimas pessoas numa equipe.

XP: Prega-se a comunicação pessoal como um dos princípios da metodologia. Comunicação que deve ser feita a todo momento entre os membros de uma equipe. Obviamente, esse princípio é melhor colocado em prática no caso de equipes pequenas.

FDD: Como envolve muitos papéis entre os membros e esses papéis não podem ser exercidos pela mesma pessoa por questões de conflitos de interesses, FDD não dá margem para equipes muito pequenas. Normalmente, quando bem distribuídos entre os membros, os papéis são de vital importância para o sucesso do projeto. XP e Scrum não possuem este problema.

- **Ênfase no modelo**:

Scrum: Em Scrum não há essa preocupação com a arquitetura do sistema, pois a metodologia não foi projetada com o intuito exclusivo de ser utilizada no desenvolvimento de software [7]. Assim, a ênfase é nas interações entre os indivíduos.

XP: Em XP, a ênfase é na codificação/manutenção do código.

FDD: É a única destas três metodologias que possui um foco na arquitetura ou modelagem do sistema. Nesta metodologia, a arquitetura é pensada desde a primeira fase do ciclo iterativo e passa por inspeções durante todas as outras fases [23].

- **Ênfase gerencial**:

Scrum: É uma metodologia focada em processos gerenciais [7]. Não existe uma lista de técnicas agregadas aos processos de Scrum assim como existe em XP, por exemplo. Por outro lado, existe uma carga maior de práticas gerenciais que incentivam agilidade em equipe quando comparado a outras metodologias ágeis.

XP: É uma metodologia orientada a processos de desenvolvimento. Assim, não possui ênfase gerencial.

FDD: Possui certa ênfase gerencial. Existem membros dentro de uma equipe FDD responsáveis por outros setores. Por exemplo, programador-chefe é um papel gerencial, no sentido de que coordena os desenvolvedores nas atividades destes. Por sua vez, respondem a superiores (gerentes de equipe), afirmando um caráter gerencial a metodologia.

- **Interação cliente/desenvolvedor:**

Scrum: Por ser uma das principais características das metodologias ágeis, todas as três metodologias mencionadas aplicam-na de alguma forma. Scrum aplica intensivamente, visto que o Product Owner faz o papel do cliente durante a Sprint e suas cerimônias. Assim, participa das reuniões de planejamento junto aos desenvolvedores e explica os requisitos pessoalmente, priorizando os de maior necessidade. O ideal é que o Product Owner seja o próprio cliente ou algum representante do cliente. No entanto, caso isso não seja possível, a equipe do projeto designará um membro experiente para estudar as necessidades do cliente e executar o papel deste. Dessa maneira, em Scrum, sempre há um alto nível de interação entre os desenvolvedores e o cliente.

XP: Este é na verdade um dos poucos requisitos do XP. O cliente deve estar presente todo momento. Não apenas para ajudar o time de desenvolvimento, mas também para ser parte dele. Todas as fases de um projeto XP dependem de comunicação com o cliente, de preferência cara a cara [14]. O cliente escreverá estórias junto com os desenvolvedores, avaliará funcionalidades implementadas e darão feedback o mais rápido possível para a equipe de desenvolvimento.

FDD: Existe interação com o próprio cliente. O cliente deve estar sempre presente nas reuniões de planejamento e principalmente durante a primeira fase de modelagem do ciclo. O cliente é normalmente também o especialista no domínio, explicando as regras de negócio para os arquitetos e os programadores-chefe. Entretanto, como os programadores-chefe não são propriamente desenvolvedores, o contato entre o cliente e os desenvolvedores será realizado indiretamente, diminuindo a intensidade da interação Cliente/Desenvolvedor. Além disso, os desenvolvedores começam a trabalhar baseados na modelagem existente do sistema, a qual é tarefa dos arquitetos e programadores-chefe junto ao cliente.

- **Projetos grandes e complexos**:

Scrum: Scrum foi primeiramente pensado para controlar grupos pequenos. No entanto, pode ser escalável para trabalhar com grupos de 50 ou 60 pessoas [7]. Mas, nesse caso é necessário dividir o grupo em grupos menores e executar um Scrum de Scrums, onde cada grupo utiliza Scrum internamente e um Scrum Master controla as interações entre os grupos existentes. Além disso, por haver uma quantidade grande de pessoas trabalhando num mesmo projeto, torna-se mais difícil a comunicação cara-a-cara entre elas, prejudicando a obediência ao princípio da comunicação no Scrum.

XP: Segue o mesmo princípio de pequenas equipes auto-organizadas que existe no Scrum. Semelhantemente, pode ser escalável para grandes equipes em grandes projetos, mas originalmente foi idealizado para equipes pequenas com poder de decisão próprio.

FDD: O primeiro projeto utilizando FDD foi realizado para um banco em Cingapura no ano de 1997. Neste projeto, participavam 50 pessoas e o prazo era de 15 meses [22]. Assim, a própria origem da metodologia indica que FDD pode ser utilizado em projetos grandes sem necessitar repartir os membros em pequenos grupos de FDD. Isto se dá pelo perfil hierárquico ao qual a metodologia se apega. Diferentemente de XP e Scrum, FDD possui uma hierarquia específica, podendo assim delegar grupos de pessoas sob a responsabilidade de um membro, por exemplo, um gerente. É o que acontece entre os papéis de desenvolvedor e programador-chefe - vários desenvolvedores estão subordinados a um ou mais programadores-chefe.

- **Gerenciamento de riscos**:

Scrum: A cada Daily Scrum, a equipe se reúne e conversa sobre o que foi feito desde a última reunião até então. São discutidos também possíveis impedimentos e riscos que podem atrapalhar o trabalho de desenvolvimento da equipe. Os membros da equipe devem se sentir absolutamente livres para expressar algo que está ameaçando a equipe: influências externas, falta de recursos, desmotivação, falta de clareza nos requisitos, entre outros. Esse impedimento é escrito em um cartão e colado no Dashboard da equipe, indicando para o Scrum Master que existe um obstáculo a ser retirado. É uma forma rápida e descomplicada de levantar riscos e eliminá-los (sempre que possível) [8]. Por outro lado, não há um plano explícito para (1) identificar riscos, (2) estabelecer estratégias de gerenciamento de riscos e (3) mitigar riscos.

XP: Como a ênfase é no processo de desenvolvimento e não no processo de gerência, não há uma base para tratar riscos em XP.

FDD: Cabe ao gerente de projeto identificar e eliminar riscos durante o projeto. Não existe um documento de Gerenciamento de Riscos assim como também não há impedimentos levantados pela equipe. Os riscos são vigiados pelo gerente de

projeto e tratados de acordo com sua estratégia. O objetivo dessa abordagem diante de riscos é justamente evitar se concentrar em problemas gerenciais, de modo a se concentrar principalmente no design e conseqüente implementação do sistema.

- **Stand-up meetings**:

Scrum: Esse estilo de reunião proposto inicialmente na metodologia XP é utilizado também por Scrum. Em Scrum, essas reuniões são chamadas de "Daily Scrum" e as equipes programam um horário específico para realizá-la todos os dias.

XP: Não há um horário específico para realizar essas reuniões. É desejável que seja realizada todos os dias, mas não é uma cerimônia formal e qualquer membro da equipe pode convocar um stand-up meeting a qualquer momento [13].

FDD: Apesar de a comunicação interna ser estimulada, não existem traços específicos deste estilo de reunião entre os membros de uma equipe.

- **Especialista em documentação**:

Scrum: Não há um redator técnico. O próprio Scrum Master ou Product Owner se responsabilizam pela documentação produzida.

XP: Em XP, existe um especialista em documentação assim como no FDD, se responsabilizando sobre documentos de requisitos, casos de uso, mudanças de requisitos, entre outros. No entanto, como esses documentos só são produzidos quando promovem algum valor para o cliente, a importância do redator técnico não é tão grande.

FDD: Como FDD dá muita atenção à documentação produzida durante a modelagem, uma pessoa é responsável por estes documentos: o redator técnico. Este é responsável por aliviar a carga sobre os desenvolvedores e gerentes, tomando conta da documentação produzida durante as fases de desenvolvimento.

- **Código coletivo**:

Scrum: Quando utilizado em projetos de software, não prega abertamente a coletividade do código. No entanto, seus princípios se adéquam a esta característica advinda do XP [18]. Quando uma estória está sendo implementada, a equipe inteira já se comprometeu com aquela estória. Logo, toda equipe deve ter acesso ao código fonte e liberdade para modificá-lo.

XP: Possui a coletividade do código como um princípio [13]. O código pertence à equipe e todos devem estar cientes disso. Esse princípio é estimulado por outra técnica de XP - Pair-Programming. Ferramentas de desenvolvimento como CVS e SVN são utilizadas como forma de repositório para o código implementado.

FDD: Designa alguns membros específicos como proprietários de classe. Isso bate de frente com a idéia de código coletivo pregada por XP. Na verdade, FDD foi pensado realmente desta forma. Seu criador, Jeff De Luca, escreveu que "código coletivo não é uma estrutura em que ele realmente acredita" [22].

3. TÉCNICAS E ABORDAGENS ÁGEIS

Neste capítulo serão apresentadas algumas técnicas ágeis de desenvolvimento bastante utilizadas na indústria de software. As explicações sobre essas técnicas de engenharia são breves e enfatizam três pontos: funcionamento; benefícios de aplicar a técnica; e limitações desta.

3.1 Test Driven Development

Test Driven Development é uma técnica de desenvolvimento de Software que utiliza pequenas iterações de desenvolvimento através de testes unitários escritos antes de ser escrito o código pertencente à funcionalidade em questão. Cada iteração produz código necessário exclusivamente para passar pelos testes daquela iteração. Quando essa fase de iterações termina, o programador faz um refactor no código gerado.

3.1.1 Como funciona

1. Criando um teste: Antes de uma funcionalidade ser criada, o programador escreve um teste para ela. Obviamente, nesta primeira tentativa o teste irá falhar. Mas ele deve falhar, porque não há como testar uma funcionalidade sem antes construí-la. A grande vantagem de escrever um teste antes de implementar uma funcionalidade é que o desenvolvedor irá pensar nos requisitos antes de codificar, ao contrário do que se faz normalmente. É essencial que o programador entenda bem a regra de negócio em questão antes de escrever o teste, pois dessa forma, evita escrever um teste errado do ponto de vista da lógica de negócio, anulando o teste [8].
2. Teste o teste: O próximo passo é rodar todos os testes e verificar se o teste em questão passa. Se ele passar, significa que é inútil, pois está “aprovando” código que na verdade não deveria passar (a funcionalidade correspondente daquele teste ainda não existe). Esse passo é realizado com o propósito de assegurar para

o programador que esse teste realmente está funcionando de forma correta, falhando quando deve falhar [8].

3. Escreva código: O programador deve escrever código correspondente a funcionalidade testada. Ou seja, implementar a funcionalidade exclusivamente para o respectivo teste passar. Caso o teste passe mas de alguma forma o programador não tenha ficado satisfeito com a forma como isso aconteceu, não se deve ajustar o código imediatamente. Esse passo está mais adiante.
4. Faça todos os testes passarem: O próximo passo é rodar todos os testes e verificar se passaram com a inclusão do novo teste em questão. Caso aconteça algum erro em qualquer dos testes, o programador repete o passo anterior até que o teste passe.
5. Faça o refactor: Finalmente, após o novo teste passar junto com todos os testes existentes anteriormente, o programador deve "limpar" o código. Isso pode envolver várias coisas, como por exemplo, remover código duplicado ou melhorar desempenho [11]. Nesse passo é que serão feitas as adaptações necessárias de acordo com a vontade do programador, por exemplo, remover algum artifício técnico (um "if" a mais ou algum "número mágico" colocado no código para fazer o teste passar). Caso aconteça algum erro em qualquer dos testes, o programador deve voltar aos passos 3 e 4, até que o teste passe e o código esteja refatorado de maneira satisfatória.
6. Loop: Com o código pronto e todos os testes passando, o programador deve passar para a próxima iteração, repetindo o ciclo desde o começo. É importante que o tamanho das alterações seja pequeno, forçando o programador a pensar em cada pequena etapa individualmente. É uma forma de usar a abordagem "Dividir para Conquistar" para assegurar que, se as pequenas partes do código estão funcionando e seus respectivos testes estão sendo aprovados, a aplicação como um todo irá funcionar e estará completamente testada [9].

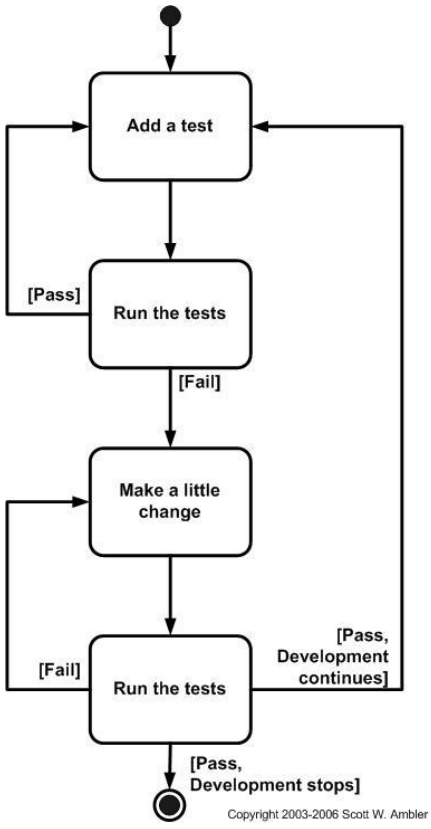

Figura 4. Grafo mostrando as etapas do Test Driven Development. [12]

3.1.2 Benefícios

Um estudo realizado em 2005 mostrou que os programadores que utilizavam TDD escreviam mais testes e, por sua vez, programadores que escreviam mais testes tinham a tendência de ser mais produtivos [10]. Logo, Test Driven Development é indicado para equipes de desenvolvimento que desejam construir software de forma mais rápida sem perder qualidade. Essa técnica oferece mais do que apenas corretude, mas pode também melhorar a arquitetura do software.

Um grande benefício obtido por utilizar TDD é a capacidade de uma aplicação inteira em pequenos passos. Isso significa que o programador irá se concentrar na tarefa atual - fazer o teste passar sem interferir nos outros testes já aprovados. Cada pequena parte do código será analisada cuidadosamente e terá a atenção do programador, cada uma por

vez. Casos excepcionais e erros complexos não são considerados inicialmente. Para esses casos especiais, devem ser criados separadamente testes para verificá-los [9].

Outra vantagem de se utilizar TDD num projeto de desenvolvimento de software é a grande cobertura de testes provida pela técnica. Se as pequenas partes do código estão funcionando e testadas, o código como um todo estará seguro. Além disso, a futura manutenção do código será mais fácil e uma grande margem de segurança quanto a erros será dada aos clientes.

Normalmente um programador que utiliza TDD escreve mais testes (e consequentemente mais código) do que um programador que não utiliza TDD [10]. No entanto, o tempo total de implementação é menor utilizando TDD [8]. Uma grande quantidade de testes limita a quantidade de erros num projeto de software. Se esses testes forem aprovados junto com o término das funcionalidades, ou seja, no início do projeto, haverá uma base concreta de prevenção contra erros endêmicos, poupando o desenvolvedor do trabalho de lenta e custosa depuração de código.

TDD tende a deixar o código mais flexível, modularizado e extensível. Esse efeito é obtido pela característica do próprio TDD de analisar cada parte do código individualmente quanto à corretude e design. Dessa forma, as classes tendem a ser menores, centradas num só objetivo e com baixo acoplamento.

Ao incentivar os desenvolvedores a programarem o suficiente apenas para o teste correspondente passar, TDD torna o código mais simples e fácil de entender.

3.1.3 Limitações

Assim como toda técnica de engenharia, TDD também tem seus pontos fracos, por assim dizer. Na verdade, são situações onde se torna difícil aplicar seus conceitos e obter bons resultados.

Ao desenvolver partes do código concernente a interfaces gráficas o TDD não é proveitoso porque nesse caso são necessários testes completamente funcionais. Qualquer outra situação que exigir testes completamente funcionais não se aplica a TDD [9].

Em casos onde a organização/empresa inteira não está totalmente convencida de que TDD aumenta a agilidade e a produtividade, TDD não costuma obter sucesso. Isso acontece pelo fato de, mais cedo ou mais tarde, a gerência começa a achar que o tempo gasto escrevendo testes é tempo desperdiçado [9].

Visto que nesta técnica o desenvolvedor escreve os testes para o código que ele mesmo implementou, pode haver algum "ponto cego", ou seja, um pedaço de código implementado com potencial de erro, mas nenhum teste escrito para este pedaço. Pair Programming pode ser a solução nesses casos.

Se o desenvolvedor não entendeu completamente o fundamento do TDD, ao criar os testes para seu próprio código, ele pode cair no erro de simplesmente querer fazer os testes passarem para livrar seus códigos de erros, sem realmente analisar se os testes são efetivos. Isso gera um falso senso de segurança na equipe desenvolvedora e, uma vez que encontrados os erros, o prejuízo será muito maior.

3.2 Outras técnicas

Existem inúmeras outras técnicas de engenharia que podem tornar o desenvolvimento de software mais rápida e de melhor qualidade. Essas técnicas apresentadas aqui não são consideradas técnicas ágeis propriamente ditas, mas, são técnicas que aceleram o desenvolvimento de software dentro de uma empresa. No entanto, abordamos aqui duas técnicas que se mostram muito populares atualmente na indústria de desenvolvimento de software. A primeira delas é Design Patterns, que é uma técnica de modelagem/desenvolvimento de software. E a segunda delas é Controle de Versão, que é uma técnica de suporte ao desenvolvimento de software.

3.2.1 Design Patterns

Design Patterns, termo conhecido em português como padrões de projeto ou padrões de desenho de software, descrevem soluções para problemas decorrentes no desenvolvimento de sistemas de software orientados a objetos. Por padrão, um design pattern descreve [26]:

- Um problema recorrente;
- Uma solução;
- Quando aplicar a solução;
- Conseqüências.

Os objetivos dos design patterns são: permitir o reuso do código, facilitar o design das classes orientadas a objetos e melhorar a documentação e o aprendizado de um projeto de software. Para isso, o padrão precisa definir uma solução para um problema específico, ser independente e mostrar claramente onde se aplica.

Os design patterns GoF (Gang of Four) totalizam 23 design patterns documentados e catalogados, que são, ao mesmo tempo, os mais populares e mais utilizados entre desenvolvedores de software na indústria [26]. Esses design patterns são divididos em 3 categorias ou tipos de padrão:

- **Criacional**: Esse padrão lida com a melhor forma de se criar instâncias de classes. A idéia é que um programa não dependa da forma como os objetos e classes são criados e inicializados dentro dele. Os padrões criacionais são:
  - Abstract Factory;
  - Builder;
  - Factory Method;
  - Prototype;
  - Singleton.

- **Estrutural**: Este padrão lida com a implementação das classes e dos objetos. Tem por objetivo suavizar o design do sistema por identificar formas de realizar relacionamentos entre entidades existentes no modelo de uma maneira simples. Os padrões estruturais são:
    - Adapter;
    - Bridge;
    - Composite;
    - Decorator;
    - Façade ou Fachada;
    - Flyweight;
    - Proxy.

- **Comportamental**: Tem por objetivo encontrar dinamicamente padrões de comunicação entre objetos e identificar esses padrões. Dessa forma, os padrões comportamentais aumentam a flexibilidade na forma como a comunicação é exercida. Os padrões comportamentais são:
    - Chain of Responsability;
    - Command;
    - Iterator;
    - Mediator;
    - Memento;
    - Observer;
    - State;
    - Strategy;
    - Template Method;
    - Visitor.

O uso de design patterns durante a etapa de desenvolvimento de software possui diversas vantagens. Para os desenvolvedores de uma equipe é extremamente produtivo, pois incentiva uma forma clara e otimizada de comunicação. Projetistas de software podem discutir a solução para um problema de forma transparente, visto que todos na equipe de

desenvolvimento entendem os padrões de projeto. O tempo de aprendizado da equipe de desenvolvimento é reduzido drasticamente [27]. Também, provê uma melhor forma de documentar soluções e resolver problemas de maneira prática.

Além disso, por ter sido utilizada em diversos projetos que apresentaram o mesmo problema, essas soluções mostram-se comprovadamente eficazes. Padrões de projeto tornam o código mais coeso e com acoplamento mínimo, atributos desejáveis de engenharia de software [27]. A complexidade do código é reduzida por causa da forma como são tratadas as abstrações de classes e instâncias.

Por apresentar atributos como rapidez no processo de aprendizagem dos desenvolvedores, transparência das soluções apresentadas, alto nível de reuso de código no sistema, baixo nível de complexidade, praticidade ao resolver problemas recorrentes e melhoria na capacidade de expressão da equipe, os padrões de projeto mostram ser uma técnica de engenharia bastante eficaz para ser utilizada durante a fase de codificação/modelagem de software.

3.2.2 Controle de versão

Um sistema de controle de versão é um software que tem por finalidade gerenciar as versões geradas de um software durante todo o processo de fabricação deste [28]. Na verdade, estes softwares são utilizados para salvar histórico de modificações de qualquer documento. Entretanto, na indústria de softwares eles são extremamente necessários por guardar diferentes versões de código-fonte e documentação do projeto.

Sistemas de controle de versão são muito populares atualmente e podem ser encontrados em forma de softwares livres, como CVS e SVN, ou softwares comerciais, como o ClearCase da IBM. Uma diferença entre as soluções comerciais e livres, é que as versões livres não oferecem garantia contra erros, por exemplo, perda de dados. Por outro lado, as versões livres possuem melhor performance e maior segurança contra ataques externos [28].

O funcionamento básico de um sistema de controle de versão é bastante simples e pode ser explicado brevemente por meio dos elementos que compõem este sistema [28]:

- **Servidor**: Uma máquina que possui espaço para armazenamento de arquivos. Deve possuir um endereço lógico, que pode ser uma URL ou um conjunto IP/porta. Esse endereço é necessário para o cliente poder se conectar ao servidor.

- **Repositório**: É o sistema de arquivos responsável por armazenar os dados gravados dos documentos. Esses dados devem ser guardados de forma persistente (como num banco de dados) e facilmente resgatáveis. Um servidor pode possuir mais de um repositório. Por exemplo, o repositório "A" conterá os arquivos produzidos pelo projeto "X". Enquanto o repositório "B" armazenará os arquivos produzidos pelo projeto "Y".

- **Cópia local**: Todo desenvolvedor possui em sua máquina uma cópia dos seus documentos. Esses documentos são sempre a versão mais atual existente no sistema.

Quando um desenvolvedor resolve salvar (fazer um commit) um documento no sistema de controle de versão pela primeira vez, este documento fica salvo no repositório especificado pelo desenvolvedor. Caso outro desenvolvedor do mesmo projeto que possui acesso ao documento salvo faça uma alteração na sua cópia local e deseje salvá-la no respositório, o sistema não permitirá. De fato, o sistema indicará para este usuário que ele possui uma versão desatualizada daquele documento. Uma vez que o desenvolvedor atualiza a versão mais nova do documento, ele é capaz de alterar este documento e salvá-lo no sistema novamente.

Num projeto de software, é extremamente útil manter um histórico das versões do produto em desenvolvimento. É exatamente esse o objetivo dos sistemas de controle de versão. Esses sistemas oferecem facilidade ao fazer/desfazer modificações nos documentos do projeto, possibilitando uma análise do histórico de desenvolvimento. Alguns sistemas, como CVS e SVN, por exemplo, exibem até mesmo comparações entre versões produzidas

[29]. Assim, todos os elementos da equipe podem compartilhar código-fonte do sistema em desenvolvimento sem medo de fazer alterações irreversíveis. Além disso, o código se torna acessível e compreensível por todos da equipe, promovendo a integração entre os membros. Ao mesmo tempo, evita conflitos de versões, que pode ocorrer sempre que dois usuários estão modificando um mesmo arquivo ao mesmo tempo.

Muitos sistemas permitem que os usuários salvem uma versão estável do projeto inteiro ou "tag". A idéia é que, caso o lançamento de uma nova versão aconteça com erros, fazendo surgir uma versão instável, é possível pedir ao sistema para voltar à última versão estável do projeto [28]. Assim, uma equipe de desenvolvimento pode sempre possuir uma versão relativamente estável do produto em desenvolvimento, facilitando inclusive a manutenção do software.

Sistemas de controle de versão mostram-se também ser uma ferramenta de suporte que confere agilidade ao projeto. Ao utilizar esse sistema, uma equipe de desenvolvimento pode desenvolver funcionalidades em paralelo, sem que uma interfira na outra.

3.3 Domain Driven Design – Uma abordagem ágil

Modelagem de software é como uma arte e não pode ser ensinada como uma ciência exata por meio de fórmulas e teoremas. Podem ser utilizados princípios e métodos para melhorar a construção de um software, no entanto, jamais existirá um caminho exato a ser seguido para prover as necessidades de um sistema no mundo real com um software. Assim como na construção de um prédio, a construção de um software irá refletir características das pessoas que o modelaram e o implementaram.

Existem várias formas de abordar o modelo de um sistema. Nos últimos 20 anos, a indústria de software tem utilizado vários métodos de modelagem para criar seus produtos, cada um deles com suas vantagens e desvantagens [26]. O objetivo deste tópico é abordar uma abordagem de modelagem que tem sido aprimorada por mais de duas décadas,

contudo, tem se afirmado mais claramente como um método há poucos anos: Domain Driven Design (DDD).

DDD é uma abordagem da modelagem que visa tornar a idealização e consequentemente a construção de software mais ágil, fazendo isso de uma forma transparente a todos envolvidos no processo. DDD combina práticas de modelagem e desenvolvimento, e mostra como bom design e desenvolvimento podem trabalhar juntos para criar uma solução ótima. Um bom design irá acelerar o desenvolvimento, enquanto feedback constante provido por parte do desenvolvimento aumentará a qualidade do design.[25]

3.3.1 Conhecimento do domínio

Como o próprio nome da técnica diz, o foco desta técnica é o domínio de negócio. Por onde começar a idealização de um software? Entendendo o domínio. A melhor forma de compreender bem um determinado domínio é conversar com as pessoas que melhor o entendem. Isso significa que no caso de um software para uma central de teletáxi, por exemplo, o mais recomendado seria conversar com as pessoas que trabalham na central atendendo os telefonemas. No caso de um software para controle de tráfego aéreo, seriam os controladores de vôo. E assim por diante. O objetivo é encontrar o expert do domínio, a pessoa que tem em sua mente o modelo de como o sistema no mundo real funciona para, desta forma, transferir o modelo para o papel.

O arquiteto, analista ou desenvolvedor do software deve conversar com o expert do domínio, trocar conhecimento, fazer perguntas e escutar bem as respostas. Dessa forma, o idealizador do software estará como que escavando o problema, captando conceitos essenciais do domínio em questão. Esses conceitos podem parecer confusos e desorganizados de início. Mas, de qualquer forma são essenciais para o entendimento do domínio. O objetivo ao fazer perguntas é extrair o modelo dentro da mente dos experts de domínio. O ideal é que o idealizador do software junto com o expert do domínio consigam chegar a um consenso sobre a visão do domínio e produzam um modelo. Essa visão não

será completa nem correta inicialmente. Contudo, é algo necessário para começar a pensar nos conceitos essenciais do domínio [25].

Em algum momento durante o processo de modelagem do sistema, os especialistas em software talvez queiram fazer um protótipo do modelo para testar se este funciona. Essa é uma atitude produtiva, pois ao encontrar erros, os idealizadores do software podem querer fazer modificações no modelo. É interessante que a comunicação não é unilateral, sempre vinda do especialista do domínio até os desenvolvedores, arquitetos e analistas. Pelo contrário, os especialistas do software devem, sempre que possível, prover feedbacks. Isso aumenta a clareza e a qualidade do modelo produzido [25]. De forma geral, especialistas do domínio conhecem bem sobre suas áreas de trabalho, entretanto, organizam as idéias de um modo particular. Cabe ao especialista do software sintetizar esse conhecimento de forma útil para montar o design do software. Essa atividade pode parecer custosa no começo, e realmente deve ser, pois no final dessa atividade o sistema em forma de software deve refletir perfeitamente o funcionamento do sistema no mundo real [25].

3.3.2 Uma linguagem ubíqua

Os especialistas de software ao falar com especialistas do domínio estão sempre com a mente cheia de assuntos ligados a área de desenvolvimento. Enquanto o especialista do domínio explica os conceitos, os especialistas de software estão pensando em termos de Orientação a Objetos, herança, poliformismo, atributos e coisas do tipo. Mas os especialistas do domínio não fazem idéia do que seja um framework, uma classe ou padrões de projetos. Logo, não há como conversar sobre o domínio usando uma linguagem que apenas um lado dos especialistas entenderá. Da mesma forma, especialistas do domínio possuem uma visão bem particular da sua área de trabalho. Usam jargões que normalmente não são compreendidos por pessoas que não trabalham na área. Essa linguagem também não seria compartilhada por ambos os lados. E quando ambos os lados não conseguem se comunicar efetivamente significa que o projeto tem problemas.

Deste problema surgiu a idéia para uma linguagem ubíqua. Esse é um dos princípios centrais do DDD e tem como objetivo unificar a linguagem de ambos os especialistas. A

idéia é criar uma linguagem baseada no modelo e não na área de domínio de nenhum dos lados. O modelo é o fator comum entre as duas equipes. É onde a área de domínio se funde com o software. O modelo serve de esqueleto para a comunicação entre as partes. DDD prega que a linguagem ubíqua deve ser utilizada consistentemente por todas as fases do projeto, inclusive na implementação [25].

Criar uma linguagem ubíqua não é um trabalho fácil. Pelo contrário, a linguagem deve ser criada junto com o modelo. Quando o modelo sofre modificações, a linguagem sofre modificações. É necessário encontrar os termos-chave que definirão o modelo e o domínio. Alguns termos são fáceis de encontrar, outros são difíceis. Mas uma vez decidido qual termo utilizar para um conceito do domínio, este termo deve ser utilizado em todo documento, diagrama, classe ou método que se refira ao mesmo conceito. Às vezes é necessário esclarecer um termo entre as equipes de especialistas para que não haja confusão sobre conceitos importantes do sistema futuramente. Caso os especialistas de domínio achem que determinado termo é muito obscuro para ser entendido facilmente, significa que este termo não deve ser utilizado. Deve ser discutido um termo alternativo para substituí-lo. Da mesma forma, especialistas de software devem estar atentos a ambigüidades ou inconsistências de termos que irão aparecer durante a modelagem.

3.3.3 Model Driven Design

Na visão do DDD, construir um modelo é a parte mais importante de todo o processo de desenvolvimento de software [25]. É a partir daí que o software vai tomar forma e se tornará extensível, flexível e de fácil manutenção.

Um problema comum que ocorre no desenvolvimento de um modelo é quando o modelo é feito separadamente pelos especialistas do domínio e arquitetos de software. Estes tendem a enxergar o modelo descartando características intrínsecas da tecnologia usada, pois não são desenvolvedores. Por exemplo, talvez o modelo produzido independentemente não seja compatível com o conceito de objetos persistentes. Talvez o framework utilizado trate os objetos do modelo de uma forma não transparente aos desenvolvedores. Na fase de

implementação, o modelo codificado pelos desenvolvedores não será equivalente ao modelo produzido inicialmente, produzindo software de qualidade questionável [26].

A solução que DDD propõe é incluir os desenvolvedores nas reuniões de definição do modelo. “O modelo do domínio deve ser construído com um olho aberto ao software e a características da arquitetura.”[25] Ou seja, desenvolvedores devem ser incluídos na produção do modelo. Isso provê feedback de ambas as partes, o que é produtivo para o produto final. É importante lembrar que a comunicação nessas reuniões deve utilizar a linguagem ubíqua criada para o modelo [25]. Dessa forma, todos envolvidos na elaboração do modelo (desenvolvedores, analistas, arquitetos de software e especialistas do domínio) serão capazes de compreender os problemas e soluções apresentados.

3.3.4 Elementos do Model Driven Design

Neste tópico serão apresentados de forma breve os elementos ou padrões mais importantes do Model Driven Design. O objetivo é mostrar alguns conceitos principais usados na modelagem de objetos ou design de softwares do ponto de vista do DDD. A Figura 5 representa alguns desses elementos e indica como eles se relacionam.

1. **Arquitetura em camadas**: Model Driven Design divide o software basicamente em 4 camadas: Interface Gráfica (apresentação das informações na tela), Aplicação (coordena atividades da própria aplicação aplicação), Domínio (regras de negócio) e Infra-estrutura (classes de suporte ao sistema). DDD incentiva que cada uma dessas camadas deve ser dependente apenas das camadas mais abaixo. Ou seja, os desenvolvedores devem isolar todo o código relativo ao domínio e manter esse código em apenas uma camada. Isso promove baixo acoplamento e quebra a complexidade do sistema em partes menores. Se o código não estiver separado corretamente em camadas, a manutenção se tornará muito difícil [26].

2. **Entidades**: Nem todo objeto é uma entidade. Mas todo objeto que possua uma identidade própria deve ser uma entidade. DDD incentiva a utilização de

entidades dentro do sistema. Ou o objeto possui um atributo que serve como identificação única dele ou o objeto será identificado por uma junção de vários atributos dele. Os desenvolvedores devem saber quando utilizar objetos como entidades e quando não. Entretanto, invariavelmente um sistema deve ter suas próprias entidades, isto é, objetos que possuem uma identificação própria para se diferenciar de outros objetos idênticos.

3. **Objetos-Valor:** São objetos utilizados para descrever certo aspecto do domínio e não possuem uma identificação única [25]. Obviamente, nem todo objeto dentro de um sistema será uma entidade. Isso degradaria a performance do sistema e criaria atributos inexpressivos para os objetos. Logo, não seria uma boa prática de modelagem do domínio. Objetos-valor são característicos de objetos que não possuem uma necessidade de continuidade durante a vida do sistema, ou seja, podem ser criados e descartados a qualquer momento sem conflitos de identidade. DDD dá muita ênfase no fato de que os idealizadores do modelo devem saber quando um objeto é uma entidade ou um objeto-valor.

4. **Serviços:** Caracteriza um objeto do domínio que provê algum tipo de serviço. Mas, não funciona como uma entidade nem como um objeto-valor. Na verdade, é uma operação realizada pelo sistema que não se encaixa como um comportamento de um objeto. São classes que provêem determinados tipos de serviços e servem como ponto de conexão entre vários objetos do domínio. As operações realizadas pelos serviços são conceitos do domínio que não pertencem a um objeto-valor ou a uma entidade. No entanto, as operações normalmente se comunicam com outros objetos e não guardam um estado próprio (stateless).

5. **Módulos:** Modelos muito grandes são difíceis de se analisar como um todo. Por isso, DDD incentiva a criação de módulos, que são utilizados como uma forma de organizar conceitos relacionados dentro do modelo com o objetivo de quebrar a complexidade. Essa prática incentiva a alta coesão, um atributo muito importante da engenharia de software. Ao organizar módulos, os especialistas de software devem agrupar classes com o mesmo propósito e isolar classes com

propósitos diferentes. Para haver boa integração entre os módulos, devem existir interfaces bem definidas e acessíveis de dentro de outros módulos.

6. **Aggregates:** Esse é um padrão de projeto bastante útil para eliminar relacionamentos complexos no domínio. Muitos relacionamentos entre objetos são complicados de administrar e muitas vezes podem tornar o software performático. Modelos com relacionamentos complexos sofrem com a manutenção destes, principalmente se os relacionamentos forem bidirecionais. DDD prega que este tipo de modelagem não é eficiente [25]. Portanto, um modelo do domínio deve possuir o mínimo de relacionamentos possível. Um dos principais problemas encontrados com os relacionamentos complexos e multidirecionais é a dependência criada entre os objetos. Caso um objeto precise ser deletado do sistema, todas as referências que apontam para ele devem ser notificadas. O mesmo ocorre no caso de uma atualização no estado do objeto. O padrão aggregates resolve isso de forma simples. Como o próprio nome sugere, são criados objetos agregados dentro de outro objeto. Porém, a única parte acessível externamente é a raiz do Aggregates, que normalmente é uma entidade. É uma forma simples de substituir a idéia de relacionamentos por objetos dependentes. Assim, evitam-se relacionamentos de outros objetos externos com os internos ao Aggregates. Assim, caso a raiz precise ser deletada do sistema, suas dependências serão deletadas conjuntamente sem processamentos demorados.

7. **Factories:** É comum aparecerem objetos complexos na modelagem do domínio. Objetos complexos seriam, por exemplo, objetos que encapsulam outros objetos ou objetos com construtores extensos. No momento de instanciar os objetos, os construtores desses objetos tornam-se muito grandes, complicando a legibilidade do código e separando o modelo do domínio do domínio no mundo real. Como DDD prega que os sistemas devem espelhar o domínio real, construtores extensos não devem ser utilizados [25]. Além disso, objetos não deveriam criar a si próprios. Mas, deveriam existir objetos que criassem outros objetos de forma descomplicada. Esses objetos criadores são chamados de Factories. As

Factories não fazem parte do domínio real, mas fazem parte da modelagem da aplicação. Elas provêem uma interface que abstrai a criação de objetos complexos, por exemplo, Aggregates. Além disso, Factories não exigem que o cliente referencie classes concretas ao instanciar objetos. Toda informação para a obtenção do novo objeto é escondida dentro da Factory deste objeto. Existem diversos padrões de projetos que implementam o modelo de Factories [26]. De qualquer forma, uma instância do tipo Carro seria obtida através de uma Factory de Carros, por exemplo.

8. **Repositórios:** O propósito de um repositório é encapsular toda lógica necessária para obter referências de objetos. Os objetos do domínio não terão de lidar com a camada de infra-estrutura do sistema para obter referências de outros objetos do domínio. Eles simplesmente pegam as referências para esses objetos no repositório. Dessa forma, o modelo fica mais claro e coeso. Objetos podem ser guardados no repositório para uso posterior. Além disso, caso o repositório receba uma requisição por uma referência que ele não possua, ele sempre pode obter esta referência da camada de infra-estrutura. De qualquer forma, o repositório age como um local de persistência dos objetos acessíveis globalmente. Dessa maneira, a camada de infra-estrutura será invocada bem menos do que anteriormente (num modelo sem repositório). Essa técnica confere agilidade e simplicidade ao sistema [26].

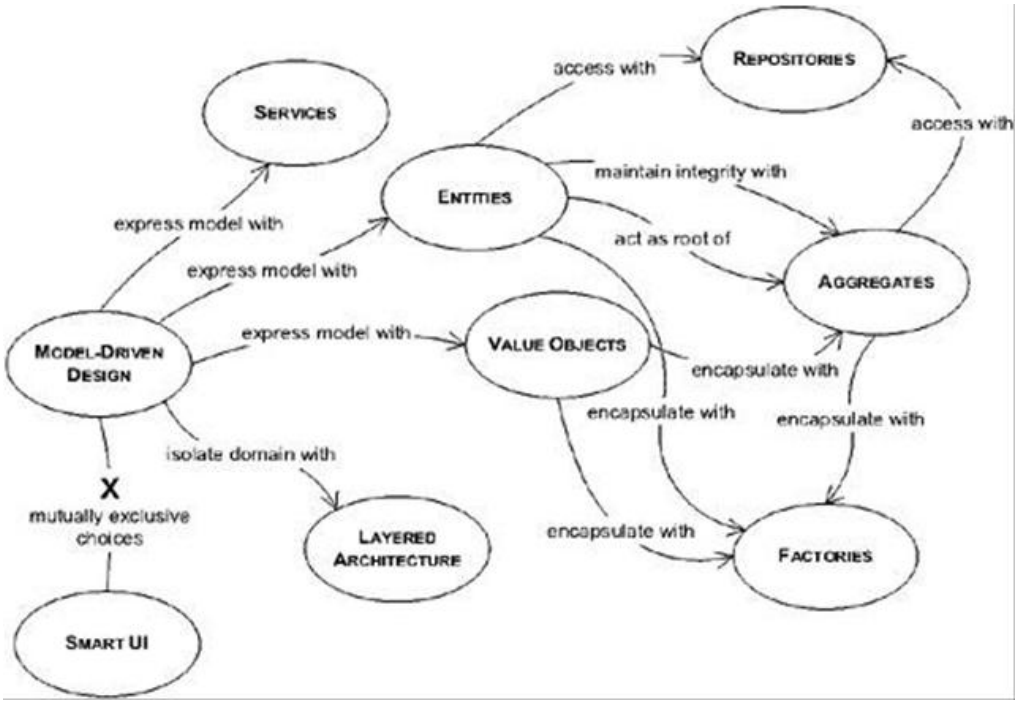

Figura 5. Relações entre os elementos que compõem o Domain Driven Design. [25]

3.3.5 Benefícios

Domain Driven Design não é uma metodologia de desenvolvimento de software. Na verdade, é uma nova abordagem que provê um conjunto de boas práticas para modelagem de um sistema. Essas boas práticas incentivam o trabalho em equipe, com todos os membros participando da troca de conhecimento e do desenvolvimento do modelo do domínio.

DDD tem como principal alvo a compreensão do domínio do sistema. A modelagem de um sistema de software deve se espelhar o máximo possível na área de domínio a qual o sistema pertence. Dessa forma, são evitados objetos complexos e esquemas complicados demais para a compreensão de pessoas que não são especialistas em software.

Ao utilizar técnicas como Divisão por Camadas, DDD maximiza a escalabilidade do sistema e confere maior facilidade na manutenção do software. Ao utilizar técnicas de modelagem como Aggregates e Objetos-valor, reduz a necessidade por relacionamentos

complexos no modelo. Logo, o sistema projetado tende a ter melhor desempenho e maior coesão.

A comunicação e o feedback existente entre todos envolvidos no projeto, desde especialistas do domínio até desenvolvedores, provê grande agilidade ao desenvolvimento do modelo e consequentemente do sistema. Possíveis erros de modelagem são evitados cedo durante o design do sistema. Problemas de integração são logo detectados e discutidos através de uma linguagem comum entre todos os envolvidos. Más decisões de implementação são refatoradas para uma melhor construção do código. Enfim, DDD evita erros na modelagem do software, confere agilidade ao processo de produção e ao mesmo tempo agrega valor ao produto final.

4. COMPONDO MÉTODOS ÁGEIS

Neste capítulo, serão mostradas composições dos métodos ágeis apresentados neste trabalho.

Será explicado como podemos combinar Scrum, XP e TDD, de forma a agilizar todas as atividades da equipe de desenvolvimento. Da mesma forma, supriremos FDD com técnicas de modelagem existentes no DDD e com Design Patterns na parte de desenvolvimento.

Muitas outras composições poderiam ser explicadas neste capítulo. No entanto, a escolha das composições mostradas neste capítulo é fruto de uma análise realizada pelo autor. Os critérios utilizados para escolher os métodos que fariam parte das composições foram:

1. Princípios convergentes e não conflitantes entre si;

2. Abordagens semelhantes entre si;

3. Ênfase do método (gerencial, testes, modelagem e etc.).

4.1 Scrum + XP + TDD

Scrum e XP são duas metodologias com abordagens diferentes, mas com um objetivo comum. Ambas propõem aumentar a produtividade do desenvolvimento de uma equipe. O motivo disto é que, na realidade, ambas foram idealizadas sobre o mesmo conjunto de princípios. Princípios estes que se encontram hoje fundamentados no Manifesto Ágil [5]. Enquanto isso, TDD será aplicado dentro das técnicas do XP como um método que agiliza ainda mais os processos deste.

Enquanto Scrum confere agilidade focando essencialmente no processo gerencial, XP provê uma série de técnicas de engenharia que comprovadamente aumentam a produtividade das equipes de desenvolvimento de software. Pode-se dizer então que XP preenche o espaço deixado em Scrum com relação a técnicas de desenvolvimento de software, assim como Scrum envolve XP com uma camada a mais de agilidade, estabelecendo passos bem definidos para gerenciar as fases de desenvolvimento.

Como essas duas metodologias se complementam, podemos utilizar Scrum e XP num mesmo projeto, de modo que Scrum seja executado na parte de gerência de pessoas e atividades, enquanto XP, sobre o desenvolvimento das tarefas que geram valor de negócio para o cliente. Dessa forma, teríamos agilidade em todas as partes do projeto, desde a gerência até o desenvolvimento. Scrum funcionaria então, como uma embalagem para as técnicas providas por XP. Dessa maneira, como Scrum não possui regras específicas sobre a forma como os desenvolvedores devem programar ou testar um software, XP pode ser usado durante toda uma sprint, controlando a forma como o desenvolvimento é realizado, por exemplo, usando Pair-Programming. Ao mesmo tempo, TDD pode ser utilizado como método de desenvolvimento e testes durante toda uma sprint.

Visto que ambas as metodologias, XP e Scrum possuem os mesmos princípios e objetivos, mas não possuem princípios contraditórios entre si, não há razão para concluir que a união entre elas seja improdutiva. Pelo contrário, muitas empresas de renome no mercado de Software têm optado continuamente pela união entre Scrum e XP na execução dos seus projetos [19].

Suponhamos um projeto de desenvolvimento de software X para uma dada empresa. O cliente que encomendou este projeto quer receber o produto o mais rápido possível. Mais tarde a empresa é notificada que os requisitos podem mudar bruscamente durante o projeto. A empresa decide utilizar métodos ágeis: Scrum e XP. Como seria possível executar as duas metodologias de forma a cumprir com o objetivo final – entregar software funcionando rápido e cumprir com os requisitos do cliente? Vamos analisar, passo a passo, uma solução para essa questão.

4.1.1 Praticando Scrum + XP + TDD

Pré-Sprint ou Sprint Zero: No começo do projeto, o Scrum Master decide se irá fazer uma Sprint Zero, a qual normalmente é opcional, na qual seria organizada a arquitetura inicial da aplicação, escolhidos os membros da equipe de desenvolvimento e explicado o propósito do projeto. É papel do Scrum Master, não apenas assegurar que a equipe cumpra com os princípios e valores do Scrum, mas agora, que também as técnicas do XP sejam seguidas. Deve incentivar a equipe a ser o mais comunicativa possível. E é claro, assegurá-los de que estará trabalhando para retirar impedimentos que possam atrapalhar o andamento do projeto.

Sprint Planning : Serão definidas as issues para o projeto. O Product Owner priorizará as issues definidas de acordo com a vontade do cliente. Antes de se comprometer sobre quais issues serão entregues, a equipe deve estar a par de quais técnicas de XP serão utilizadas no desenvolvimento das tarefas. Só então, levando esse fator em consideração, a equipe estima as issues e as divide em tarefas de acordo com a vontade dos desenvolvedores. O Planning deverá gerar como resultado um Sprint Backlog. Cada issue do Sprint Backlog deve conter uma lista de técnicas de XP que devem ser utilizadas para aquela issue. Dependendo do cumprimento das técnicas, a issue não poderá ser finalizada. Por exemplo, caso o Pair-Programming não tenha sido feito durante uma atividade, o Scrum Master deve repor essa técnica por outra que tenha a mesma finalidade, que é aumentar a margem contra erros e melhorar a qualidade do código.

Daily Scrum : Essas reuniões por si só já aplicam originalmente o conceito de Stand-up Meeting presente no XP. Durante essas curtas reuniões entre a equipe, é explicado e discutido o que foi feito desde a última reunião de Daily Scrum e o que será feito até a próxima. Normalmente, se alguma atividade em progresso estiver sendo realizada em par, apenas uma pessoa explica o que está sendo feito. No entanto, no próximo Daily Scrum, a outra pessoa, que não falou na reunião passada, deve explicar. A intenção dessa estratégia é sempre garantir que nenhum dos dois elementos da dupla esteja “por fora” da implementação em algum momento durante a Sprint.

Dias de trabalho da Sprint : Durante os dias de trabalho da Sprint, a equipe deve sempre procurar pôr em prática tanto os conceitos de Scrum quanto os de XP. É nessa fase que as técnicas de XP serão realmente úteis. Como XP possui uma série de técnicas de engenharia para agilizar o desenvolvimento de software, a equipe deve aplicá-las durante os dias de trabalho para otimizar a produtividade. Uma dessas técnicas é a programação em par ou Pair-Programming. Em Scrum, a responsabilidade por uma funcionalidade é da equipe. Assim, todos os membros devem estar a par do código escrito. Por esta razão, o código é considerado coletivo e normalmente as equipes utilizam ferramentas de suporte como CVS e SVN. Entretanto, XP não contraria este princípio. Nesta metodologia o código é também considerado coletivo. A diferença é que os programadores que utilizam Pair-Programming têm contato direto com o código um do outro. Assim, se alguém da dupla repentinamente sair do projeto, por exemplo, o programador que ficou já conhece o código bem o suficiente, de modo que o trabalho não pára. Consequentemente este problema será reduzido a um problema menor: substituir o membro que saiu. Técnicas de padronização de código podem ser utilizadas para facilitar ainda mais o entendimento do código por parte de outros programadores da equipe [21]. Durante os dias de trabalho da Sprint, os membros devem estar sempre atentos a qualidade do código escrito. Uma das atividades de XP que ajuda a melhorar a qualidade do código é o Refactoring ou Refatoração do Código. Normalmente esta técnica é feita conjuntamente com o Pair-Programming: enquanto o piloto escreve o código, o navegador pensa, entre outras coisas, em como refatorar o código para simplificá-lo. Os programadores também devem ter em mente um dos princípios do XP: Design Simples. Isso significa que, para resolver um problema, a solução mais simples sempre deve ser escolhida ao invés de soluções mais complexas, mesmo que as últimas sejam mais robustas. A idéia é visar agilidade em primeiro lugar. Obviamente, não se pode esquecer da confiabilidade do produto entregue. Por isso, os desenvolvedores devem sempre testar o que foi implementado. Normalmente, uma equipe só considera uma issue como finalizada após a mesma ter sido testada. O teste pode acontecer após o fim da codificação ou pode acontecer durante o desenvolvimento, utilizando TDD sempre que possível. Dessa forma, os testes serão escritos antes das respectivas funcionalidades, aumentando a capacidade do programador de prever possíveis falhas e conseqüentemente evitando mais bugs. Como TDD possui refatoração como uma de suas atividades, a equipe

estaria assim testando, refatorando e codificando ao mesmo tempo, priorizando mais uma vez a agilidade sem perder o foco em qualidade.

Conclusão da Sprint: Essa fase é composta pelas reuniões de Sprint Review e Sprint Retrospective. O Sprint Review é realizado da forma tradicional seguindo os padrões do Scrum. Já na Sprint Retrospective, a equipe se reunirá numa sala segura e livre de pessoas externas a equipe para discutir sobre o que aconteceu naquela Sprint. Serão levantadas questões sobre o que houve de bom e o que há ainda para melhorar. Assim como também lembradas as lições aprendidas durante aquela Sprint. Portanto, a equipe tem uma grande oportunidade para conversar sobre o que está funcionando e o que não está. Entre essas, devem ser levadas em consideração os métodos utilizados, como: Pair-Programming, Design Simples, Refactoring, TDD e padrões de codificação. Por exemplo, se no Projeto X os programadores não estão satisfeitos na forma como os pares são distribuídos, devem levantar isso na reunião e comunicar ao Scrum Master. Dessa forma, a comunicação efetiva entre os membros da equipe levará a equipe ao auge de sua capacidade. Ao fim da Sprint Retrospective, uma Sprint terá sido finalizada e o cliente já estará com um software funcionando e devidamente testado, cumprindo a exigência de entrega rápida.

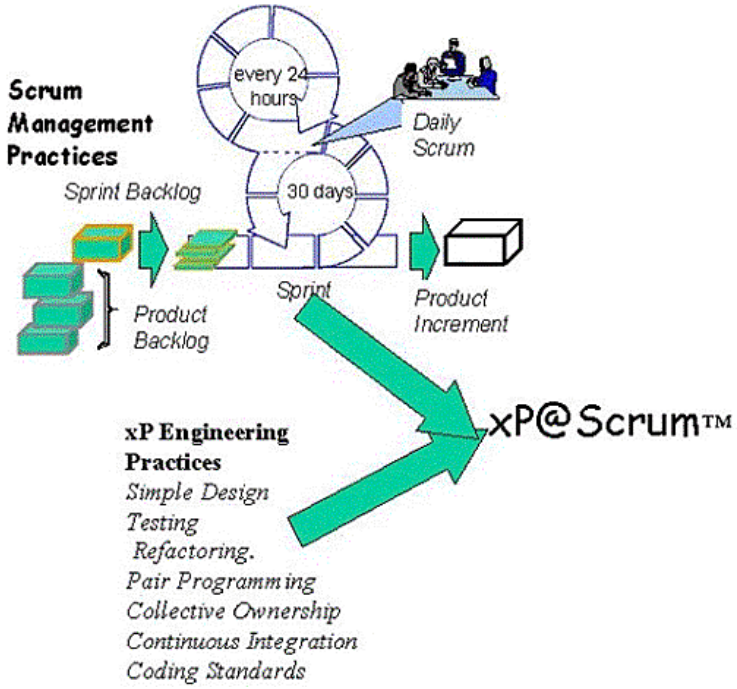

Figura 6. XP sendo executado dentro das iterações do Scrum. [19]

4.1.2 Vantagens da composição

O uso das técnicas de XP e TDD dentro de um projeto de software causam uma grande melhoria na qualidade do código. Apesar de cada uma das práticas influenciar diferentes aspectos do código, em conjunto elas têm um efeito profundo na melhoria da qualidade do produto final. Entretanto, esse efeito na qualidade do produto só é notado após um período de tempo, quando o código existente já passou por diversos refatoramentos e a aplicação já foi testada de diversas maneiras. Esse acontecimento é explicado pela forma incremental como o Scrum funciona, ou seja, a cada Sprint o produto vai ficando cada vez mais funcional e mais valorizado pelo cliente.

Scrum aplicado junto com XP num projeto de software melhora ainda a produtividade da equipe. O motivo principal disto é a união de idéias, princípios e valores semelhantes unidos com um único objetivo: Permitir aos desenvolvedores focar-se em funcionalidades úteis para o cliente ao invés de entregar artefatos de valor questionável. Isso não significa que essas duas metodologias desprezem documentação. Na realidade, atividades de análise e modelagem são realizados quando necessário para o projeto, ou seja, quando é sentido que elas agregam valor ao produto. Entretanto, quando é constatado que essas atividades não geram valor para o cliente, a equipe responsável pelo projeto deve ignorá-las [19].

Executar Scrum junto com XP torna o projeto transparente para o cliente. O motivo disto é que ambas pregam que o cliente (ou um representante do cliente) deve estar sempre presente durante o planejamento das atividades e sempre disponível para sanar dúvidas dos desenvolvedores. Também as funcionalidades criadas de forma incremental são entregues diretamente ao cliente, o que aumenta a capacidade deste de fornecer feedbacks para a equipe de desenvolvimento. Quando o Product Owner prioriza as issues durante a reunião de Planning, está na verdade comunicando aos desenvolvedores quais são suas principais necessidades naquele momento. Consequentemente, quando as respectivas funcionalidades

estiverem prontas, o cliente ficará mais do que satisfeito. Por sua vez, a excitação do cliente contaminará a equipe do projeto, aumentando a confiança deste. E assim sucessivamente.

A necessidade de Scrum por uma técnica ou uma série de técnicas ágeis de desenvolvimento é suprida de forma bastante oportuna com a metodologia XP. Da mesma forma, há a necessidade por parte do XP de uma metodologia com ênfase em processos gerenciais para controlar a execução iterativa e incremental de um projeto. Logo, há uma relação de dependência entre as duas metodologias. Quanto mais XP for incorporado dentro de um projeto Scrum, mais a equipe de desenvolvimento sairá ganhando, pois estará eliminando os pontos fracos das duas metodologias paralelamente [20].

Ao utilizar TDD em união com as técnicas de XP, a equipe será ainda mais ágil com relação ao desenvolvimento [21]. Escrever testes para qualquer método antes mesmo deste existir, pode parecer uma perda de tempo. Mas o que acontece na prática é o contrário. O tempo perdido com testes no começo do desenvolvimento agilizará muito mais o andamento do projeto no futuro, pois poupará o programador de depurar linha por linha de códigos imensos.

TDD tende a melhorar a qualidade do código e a robustez dos softwares produzidos[10]. Assim, escrever testes para toda e qualquer funcionalidade de uma aplicação antes mesmo de implementá-la significa garantir uma maior confiabilidade do código como um todo. A quantidade de bugs encontrados no software será menor e consequentemente o custo com manutenção de software será menor.

Utilizar Pair-Programming com TDD numa equipe que já utiliza Scrum é extremamente indicado para equipes que possuem programadores de diferentes níveis técnicos [19]. Como no caso das duas metodologias o código é coletivo, os desenvolvedores com menos experiência e qualidade técnica podem formar par com desenvolvedores mais experientes e de nível técnico mais alto. Isso fará com que a evolução do programador menos experiente se torne visível a todos os membros da equipe. Dessa forma, a confiança de uns para com os outros será fortificada. Com os membros compartilhando experiências entre si e se ensinando mutuamente, a equipe será cada vez

mais auto-organizada e multi-disciplinar, exaltando os valores de uma equipe que segue Scrum.

4.1.3 Conclusão

Scrum e XP integram facilmente uma a outra para construir projetos de software de forma ágil, e também podem ser usados independentemente. A principal força destas duas tecnologias é o compartilhamento de valores e princípios convergentes [20]. De fato, elas possuem mais semelhanças do que diferenças e, sendo assim, podem ser acopladas de modo a produzir software de qualidade incrementalmente. Enquanto Scrum se refere a práticas de gerenciamento, XP é focada em práticas de engenharia. Consequentemente, Scrum pode ser utilizado como uma "embalagem" para projetos XP, tornando tais projetos mais escaláveis [21]. Na prática, XP é executado principalmente dentro dos dias de trabalho da Sprint, onde os desenvolvedores aplicarão suas técnicas de desenvolvimento de software.

4.2 FDD + DDD + Design Patterns

Ao contrário da composição entre Scrum e XP, que são duas metodologias com focos diferentes, Feature Driven Development e Domain Driven Design possuem o mesmo foco: modelagem do sistema. Ambos visam projetar o sistema num documento antes que em uma máquina. Após toda a modelagem pronta, design patterns podem ser usadas como uma técnica de programação eficaz para expressar os conceitos orientado a objeto do modelo.

FDD provê uma sequência de fases para fabricar um modelo inicial, que será modificado de acordo com as mudanças encontradas nas iterações do projeto. DDD é uma forma de abordar a modelagem do sistema com uma série de técnicas aplicáveis no próprio design. Tanto FDD quanto DDD incentivam boas práticas para gerar um modelo para o sistema. Ambos pregam que o modelo é a base para um bom projeto de software. Assim, DDD supre os processos do FDD com alguns métodos que incentivam a excelência no desenvolvimento do modelo.

Como o próprio DDD possui alguns elementos de design como Factories e Aggregates por exemplo, design patterns, que possui esses padrões por definição, pode ser muito bem aproveitado na implementação das funcionalidades do modelo do sistema. Além de prover suporte as abstrações do modelo, design patterns pode oferecer outras técnicas para melhorar as boas práticas de reuso de software num dado projeto.

Por possuírem o mesmo foco e não possuírem princípios contraditórios, FDD e DDD podem trazer muitos benefícios quando aplicados conjuntamente num mesmo projeto de desenvolvimento de software. FDD funciona como uma base para gerenciar os passos do desenvolvimento, já que possui uma sequência de fases do desenvolvimento de software. Enquanto DDD pode ser usado como um provedor de boas práticas dentro de cada uma dessas fases do FDD. Podemos dizer que existiria um módulo DDD dentro de cada um dos processos de FDD.

Por outro lado, FDD possui uma técnica onde são escolhidos proprietários de código. Logo, o código não é compartilhado por toda equipe, mas é uma responsabilidade restrita ao proprietário. Dessa forma, ao chegar à fase de construção do software, os especialistas no domínio não tem acesso ao que está acontecendo no sistema que eles encomendaram. DDD corrige essa falha, como será mostrado mais a frente.

Suponhamos um projeto de software “X” numa dada empresa. Esse software a ser construído possui um domínio específico, digamos, um software para gerenciar ligações dentro de um call-center. O cliente precisa do produto com certa urgência. E informa que os requisitos podem mudar bruscamente e devem ser esquematizados perfeitamente, pois qualquer falha na modelagem do sistema causará prejuízo para a empresa. A empresa fabricante do software decide utilizar FDD com DDD. Mostraremos no próximo tópico como seria o passo a passo do projeto “X” ao aplicar esses métodos ágeis.

4.2.1 FDD + DDD + Design Patterns na prática

Desenvolver um modelo geral: Nessa primeira fase, os especialistas do sistema precisam aprender sobre o domínio do negócio. DDD pode ser aplicado cedo no projeto,

com os especialistas em software entrevistando os especialistas do domínio (gerentes, supervisores, atendentes e diretores de call-center) sobre as peculiaridades da área. Entender questões estruturais, por exemplo, saber como são feitas as chamadas telefônicas, qual o principal objetivo da empresa, qual a hierarquia de comando existente e como acontece o controle de acesso dos agentes, é vital para essa primeira fase do processo. DDD propõe que, para haver um melhor entendimento entre as partes, os membros da equipe de software junto com os especialistas na área de call-centers devem encontrar uma linguagem em comum. Porque linguagem orientada a objetos e jargões da área de programação não é familiar a especialistas na área de telefonia. Da mesma forma que as expressões utilizadas no dia a dia dos especialistas do domínio não são conhecidas pelos especialistas em software. Essa linguagem proposta pelo DDD é chamada "linguagem ubíqua" e é um meio termo entre o conhecimento entre as duas áreas. Na verdade, a linguagem deve ser encontrada durante o entendimento do domínio e os termos combinados devem ser respeitados por ambas as equipes até o fim do projeto. Nessa fase será criado um modelo geral, com o propósito de entender o domínio e retratar suas entidades mais importantes. FDD prega que devem ser divididas várias equipes de desenvolvimento do projeto e cada uma fazer uma modelagem baseada no seu entendimento do domínio. O melhor modelo, com design mais completo e correto é escolhido como modelo inicial ou base. DDD incentiva esta prática, pois prega que não apenas os arquitetos devem participar na modelagem, mas todos da equipe de desenvolvimento devem estar envolvidos nesta atividade. Inclusive, o próprio cliente, no caso o especialista do domínio, deve validar os modelos gerados, mostrando falhas no entendimento da regra de negócio e as expondo para todas as equipes, como num ambiente de sala de aula. Ao concretizar o modelo inicial, a equipe de arquitetos deve escrever ou documentar notas explicativas sobre as entidades e elementos do modelo escolhido.

Gerar uma lista de funcionalidades: Esta fase utilizará o conhecimento obtido na fase anterior para criar tarefas no projeto. Uma equipe é selecionada para criar uma lista de funcionalidades baseadas no modelo. Assim, é elaborada uma lista de funcionalidades do sistema decompondo as áreas de domínio no modelo. Essas funcionalidades são pequenas tarefas que geram valor para o cliente. Ao seguir o formato <ação><resultado><objeto>, conceitos de DDD se sobressaem, como Serviços (Services), Entidades (Entities) e

Objetos-valor. A ação pode ser representada por um elemento Serviço no modelo. E um objeto pode ser representado por uma entidade, um objeto-valor ou até mesmo um elemento Aggregate no caso de composição de objetos. No projeto em questão, teríamos por exemplo a funcionalidade "Validar senha do atendente" no contexto de login do sistema. Assim, o modelo teria um serviço "Validar" e uma entidade (pois um atendente deve possuir um identificador único) "Agente". É importante que o cliente ou especialista do domínio esteja sempre pronto para dar feedbacks sobre o projeto. Pois apesar de DDD ser um método ágil para produzir modelos abstratos, feedbacks conferem qualidade ao produto final ao validar os modelos gerados.

Planejar por funcionalidade: É planejado o desenvolvimento das funcionalidades extraídas do modelo inicial. São designados proprietários de código. Não há muitos princípios do DDD explicitamente aplicáveis nessa fase. Mas um deles seria a continuidade da linguagem ubíqua durante essa fase. É a linguagem ubíqua que possibilitará os especialistas no domínio a entenderem se as funcionalidades descobertas pelos especialistas em software fazem sentido no contexto ou não.

Modelar por funcionalidade: É nessa fase que os programadores-chefe irão escolher funcionalidades para fazer seus diagramas de sequência e sua modelagem completa. Essa modelagem é diferente da modelagem realizada na primeira fase, visto que não é tão abrangente. Na verdade, essa fase tem por objetivo criar modelos para as funcionalidades obtidas do passo anterior. São idealizadas classes, atributos e métodos. Model Driven Design é utilizado diretamente nessa fase. De fato, as classes são criadas como Entidades, Objetos-valor ou Aggregates. Ainda são criados os repositórios, que servem como estrutura de coleção para a persistência de objetos na memória. Ao final dessa fase, uma inspeção é realizada e um refatoramento no modelo do sistema já é possível de ser feito através da máquina dos desenvolvedores. Assim, desenvolvedores podem encontrar desde já, inconsistências do modelo com alguma parte técnica do projeto. Essas inconsistências devem voltar como feedback para os programadores-chefe e os especialistas no domínio. Caso haja alguma correção, a solução é repassada para os programadores para que possam terminar suas tarefas. Também é importante que os desenvolvedores vejam o quanto antes onde será necessário inserir padrões de projeto.

Construir por funcionalidade: Com o modelo preparado, não há mais o que fazer no modelo encontrado para o sistema. Essa fase diz respeito a implementação ou codificação das funcionalidades. Design Patterns provê boas soluções para problemas recorrentes. Dessa forma, problemas não precisam ser pensados por muito tempo. Pois existe um Design Pattern correspondente aquele problema. Entretanto, os desenvolvedores podem decidir utilizar padrões de projeto para uma certa peculiaridade da funcionalidade que seria transparente ao modelo. Ao final da implementação, os testes unitários são escritos e aplicados, gerando mais feedbacks.

4.2.2 Vantagens da composição

Essa composição traz benefícios durante todo o processo de produção do software, desde a modelagem até a implementação do código. Isso acontece pelo fato de DDD ser parte integrante de cada uma das fases do FDD. Como o objetivo desses dois métodos é fundamentalmente o mesmo, ou seja, prezar por uma estrutura sólida e flexível antes de começar a construir, a composição deles reafirma suas características. Por não possuírem princípios conflitantes entre si, DDD e FDD apóiam-se mutuamente durante todo o projeto.

A linguagem ubíqua é uma técnica bastante eficiente do DDD e pode ser útil antes mesmo de a modelagem do sistema começar. Na fase de coleta de requisitos e estudo do domínio, a construção de uma linguagem não-ambígua que esclareça termos obscuros entre as partes é essencial para a compreensão do desejo do cliente. A linguagem é construída durante a fase inicial do projeto, de modo que durante as iterações, a maioria dos termos já estão estabelecidos como padrão da linguagem. As pessoas envolvidas falam esse novo "idioma" sempre que estiverem se comunicando sobre o projeto. A linguagem é utilizada tanto na comunicação falada, quanto na comunicação escrita, por meio de diagramas e modelos preliminares. A equipe de desenvolvimento deve expressar os conceitos dessa linguagem durante todo o projeto, até a codificação do sistema. Dessa forma, os especialistas no domínio sempre irão entender o que está acontecendo no desenvolvimento do software, podendo observar falhas no desenho do modelo e esclarecendo dúvidas dos especialistas do sistema relativas ao domínio em qualquer fase do projeto.

Existe um alto nível de acoplamento entre os elementos do Model Driven Design e Design Patterns. Elementos do DDD como Aggregates, Serviços e Repositórios possuem um padrão equivalente em Design Patterns. Portanto, padrões de projeto podem ser utilizados para expressar ou traduzir os elementos do modelo do domínio em código do sistema. Por exemplo, um Aggregate pode ser representado por um padrão Composite. Um Serviço pode ser representado por um padrão Command ou Chain of Responsibility. Ainda um elemento Factory pode ser expresso através de um objeto Factory. Ou seja, Design Patterns são úteis para expressar elementos do modelo do domínio por meio de objetos do sistema. No entanto, padrões de projeto só aparecem nas duas últimas fases do FDD, onde a implementação das classes e métodos do sistema passam a existir.

A arquitetura em camadas, uma característica de boas modelagens orientadas a objetos, é incentivada tanto por FDD quanto por DDD. Quando FDD utiliza a divisão em camadas sugerida pelo DDD, a complexidade do sistema diminui e o aprendizado dos desenvolvedores aumenta. Além disso, a manutenção do software, uma vez que este estiver pronto, será pouco custosa. Design Patterns possui alguns padrões estruturais que funcionam melhor em arquiteturas em camadas. Daí a vantagem de unir os 3 métodos com o mesmo objetivo.

DDD é uma abordagem na forma de encarar a modelagem do sistema na qual a preocupação é entender o domínio acima de qualquer coisa. Com o aprendizado de toda equipe de desenvolvimento sobre o domínio de negócio completo, o resto do projeto será simplificado. É uma forma de tornar ágil a forma de produzir um modelo. De fato, FDD é uma metodologia ágil por natureza que prioriza a modelagem do sistema, enquanto DDD é uma abordagem que, quando levada em consideração pelos membros do projeto, tende a aumentar a qualidade do modelo final, fazendo isso de forma simples e prática. Juntas, a metodologia e a abordagem reforçam uma a outra.

5. CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, abordamos algumas das metodologias de gerenciamento de software mais usadas na indústria de software: Scrum, Extreme Programming e Feature Driven Design. Cada uma dessas metodologias possui uma maior ênfase em um determinado aspecto. Por exemplo, Scrum possui um foco totalmente centrado na área gerencial. Já Extreme Programming contém uma série de técnicas para agilizar o desenvolvimento propriamente dito. Enquanto Feature Driven Design enfatiza bastante a modelagem do sistema, ou seja, se um projeto possui uma modelagem de alta qualidade, todas as outras fases produzirão, por conseqüência, alta qualidade. Foram analisadas vantagens e desvantagens de se utilizar cada uma dessas metodologias individualmente e foi apresentada uma análise comparativa das metodologias em questão.

Apresentamos também algumas técnicas de engenharia e abordagens que são, com certa freqüência, utilizadas em muitos projetos de desenvolvimento de software. Foi apresentada a técnica conhecida como Test Driven Design, onde primeiramente é escrito um teste unitário para cada nova funcionalidade do sistema e só então é escrito o pedaço de código correspondente. Foi explicada a abordagem do Domain Driven Design sobre a modelagem de um sistema, apresentando seus conceitos sobre linguagem ubíqua e elementos padrões de modelo de sistemas. Finalmente, foram abordadas técnicas mais utilizadas na codificação do software propriamente dito, como o uso de Design Patterns e sistemas de controle de versão.

Como parte principal deste trabalho de graduação, foram combinadas algumas metodologias e técnicas ágeis. Esses métodos foram escolhidos de acordo com uma análise realizada pelo autor. Primeiramente, foi combinado Scrum com XP e TDD, ou seja, duas metodologias ágeis com uma técnica de engenharia ágil. A análise mostrou-se positiva, pois as metodologias não possuem princípios conflitantes. Pelo contrário, foi aproveitado o melhor de cada uma delas. Enquanto Scrum guia a parte gerencial do projeto, criando sprints com planejamento, desenvolvimento e retrospectiva, as técnicas de desenvolvimento de Extreme Programming são utilizadas dentro dessas sprints, como que preenchendo as

lacunas deixadas por Scrum. Ao mesmo tempo, a técnica voltada para a área de testes de software, TDD, é utilizada como se fosse uma das técnicas de XP. Depois, analisamos a combinação entre FDD, DDD e Design Patterns. Neste caso, temos uma metodologia, uma abordagem e uma técnica de programação. O interessante nessa composição é que tanto a metodologia (FDD), quanto a abordagem (DDD) e a técnica (Design Patterns) são focadas na modelagem do sistema de software. Por esta razão, essa combinação também mostrou-se positiva, com FDD agregando pontos fortes de DDD nas suas iterações e, sempre que possível, expressando isso no código por meio de Design Patterns.

4.1 Trabalhos futuros

Na finalização deste trabalho, podem-se identificar alguns encaminhamentos futuros no sentido de elaborar uma nova metodologia.

Essa nova metodologia criada seria o aperfeiçoamento de uma das duas composições analisadas neste trabalho. No final, seria produzida uma nova metodologia ágil "X" com duas possibilidades:

1. Herdar características gerenciais de Scrum, práticas de XP e uma tendência a testar todo e qualquer pedaço de código antes do desenvolvimento, de TDD;

2. Possuir ênfase em modelagem e design de sistemas. Característica herdada de FDD e de DDD. Design Patterns seria considerada uma boa prática da nova metodologia.

De qualquer forma, o intuito nas duas possibilidades é o mesmo: Combinar boas práticas das metodologias ágeis com as técnicas ágeis de desenvolvimento, baseando-se sempre nos princípios do Manifesto Ágil, para que não haja conflitos de interesses entre os métodos. Mas para que, por outro lado, a composição funcione como um complemento das boas características existentes entre os métodos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Agile Alliance. Acesso em: 01 de Março de 2009. Disponível em: http://www.agilealliance.org/.

[2] COHEN, D., LINDVALL, M., & COSTA, P. An introduction to agile methods**.** In Advances in Computers (pp. 1-66). New York: Elsevier Science, 2004.

[3] PRESSMAN, R. Software Engineering (A practitioner's approach). 5th edition, 2000. Mc Graw-Hill Education.

[4] AMBLER, S. (April 2008). Scaling Scrum - Meeting Real World Development Needs. Acessado em: 07 de Abril de 2009. Disponível em: http://www.ddj.com/architect/207100381.

[5] Agile Manifesto: Acesso em: 07 de Abril de 2009. Disponível em: http://agilemanifesto.org/ .

[6] Agile Manifesto History: Acesso em: 07 de Abril de 2009. Disponível em: http://agilemanifesto.org/history.

[7] SCHWABER, K. SCRUM Development Process. In Advanced Development

Methods.

[8] NEWKIRK, J.,VORONTSOV, A. Test-Driven Development in Microsoft .NET, Microsoft Press, 2004.

[9] BECK, K. Test-Driven Development by Example. Addison Wesley, 2003.

[10] ERDOGMUS, H., MORISIO, T. "On the Effectiveness of Test-first Approach to Programming". Proceedings of the IEEE Transactions on Software Engineering, 31(1). Janeiro de 2005. Acesso em: 08 de Maio de 2009. Disponível em: http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-47445_e.html.

[11] FOWLER, M. Refactoring - Improving the design of existing code. Boston: Addison Wesley Longman, 1999.

[12] UML.ORG: Acesso em: 06 de Maio de 2009. Disponível em: http://www.uml.org.cn/softwareprocess/200904012.asp.

[13] Mountain Goat Software: Acesso em: 08 de Maio de 2009. Disponível em: www.mountaingoatsoftware.com/.

[14] "Extreme Programming" (lecture paper), USFCA.edu. Acesso em: 26 de Abril de 2009. Disponível em: http://www.cs.usfca.edu/~parrt/course/601/lectures/xp.html.

[15] STEPHENS, M. & ROSENBERG, D. Extreme Programming Refactored. Berkeley: APress, 2003.

[16] Wikipedia: Acesso em 02 de Maio de 2009. Disponível em: http://en.wikipedia.org/wiki/Extreme_Programming.

[17] Agile Alliance: Acesso em: 03 de Maio de 2009. Disponível em: http://www.agilealliance.org/system/article/file/1376/file.pdf.

[18] SCHWABER, K. & BEEDLE, M. Agile Software Development with Scrum. Prentice Hall, 2001.

[19] LASCANO, J. Extreme Programming embedded inside Scrum. Acesso em: 25 de Maio de 2009. Disponível em: http://www.sstc-online.org/Speaker/Papers/2276.pdf.

[20] WELLS, D. “Extreme Programming: A gentle introduction”, CRC Cards. Acesso em: 25 de Maio de 2009. Disponível em: http://www.extremeprogramming.org/rules/crccards.html.

[21] Control Chaos. Scrum It’s About Common Sense. “XP @ Scrum”. Acesso em: 25 de Maio de 2009. Disponível em: http://www.controlchaos.com/about/xp.php.

[22] IT-Agile Magazine – Entrevista com Jeff DeLuca. Acesso em: 26 de Maio de 2009. Disponível em: http://www.it-agile.com/56.html.

[23] HARTMANN, D. “Feature Driven Development: Still relevant?” Acesso em: 26 de Maio de 2009. Disponível em: http://www.infoq.com/news/Another_look_at_FDD.

[24] AMBLER, S. “Feature Driven Development and Agile Modeling”. Acesso em: 27 de Maio de 2009. Disponível em: http://www.agilemodeling.com/essays/fdd.htm.

[25] AVRAM, A., MARINESCU, F. “Domain-Driven Design Quickly”. InfoQ Enterprise Software Development Series. 2006 C4Media Inc. Acesso em: 03 de Março de 2009. Disponível em: http://www.infoq.com/books/domain-driven-design-quickly.

[26] GAMMA, E. et al. Design Patterns**:** Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

[27] FREEMAN, E. et al. Head First Design Patterns. O'Reilly Media, 2004.

[28] MOLINARI, L. “Gerência de Configuração - Técnicas e Práticas no Desenvolvimento do Software”. Florianópolis: Visual Books, 2007.

[29] CRISTIANO, C. “CVS: Controle de Versões e Desenvolvimento Colaborativo de Software”. Novatec, 2004.

Orientador

_____

Prof. Dr. Hermano Perrelli de Moura

Aluno

_____

Tiago de Farias Silva